Aprendizagem Profunda

Lecture Notes

Chapter 1

Introduction and course overview

Course objectives and structure. AI and the origin of Artificial Neural Networks. Machine Learning. The power of nonlinear transformations. What deep learning offers

Note: for details on assignments, class schedules and assessment, please refer to the course page

1.1 Course objectives

The goal of this course is to give you an introduction to deep learning. Here we will cover the foundations of deep neural networks, looking at different architectures and their applications, activation functions, optimizers and how to train deep neural networks in different contexts of supervised, unsupervised and reinforcement learning.

We will also see how to implement these networks in practice and optimize the results using regularization, data augmentation ans similar techniques, and selecting the best architecture.

Finally, this course will give you some practical experience in implementing these models in Tensorflow 2.0 and Keras, and also cover some open problems in deep learning such as interpretation and biases.

1.2 AI: historical overview

The Dartmouth Summer Research Project on Artificial Intelligence, in 1956, was a seminal event in AI. The goal of this project, according to the proposal presented by John McCarthy, Marvin Minsky, Nathaniel Rochester and Claude Shannon, was to "proceed on the basis of the conjecture that every aspect of learning or any other feature of intelligence can in principle be so precisely described that a machine can be made to simulate it."[68] Although this included learning, the most successful approach in the beginning was to use the power of computers for symbolic operations to try to extract knowledge from rules. This was the motivation behind expert systems, logic programming and what is called traditional symbolic AI. One example of this approach is MYCIN[66], a rules-based classifier that predicted which bacteria were causing the symptoms and recommended a therapy. MYCIN relied on a set of six hundred if...then rules provided by experts in the field. For example:

```
If:
```

```
(1) the stain of the organism is gram positive, and
(2) the morphology of the organism is coccus, and
(3) the growth conformation of the organism is chains
Then :
there is suggestive evidence (0.7) that the identity of the organism is streptococc
```

This is an example of a classifier that does not learn automatically. All knowledge is fixed from the start. In other words, humans must decide which features to use and humans must program the rules the system will follow.

A different branch of AI was more focused on learning, and artificial neural networks were part of this branch right from the beginning, with the work of McCulloch and Pitts [45] and Rosenblatt's perceptron [59].

1.3 Perceptron

Figure 1.1 shows a neuron cell and a schematic representation of the neuron response. Neurons have a set of dendritic branches which can be stimulated by other cells. If the stimulus passes a threshold, then the neuron fires an impulse over the axon, consisting of a wave of membrane depolarization. This in turn leads to the release of neurotransmitters in the synaptic terminals.

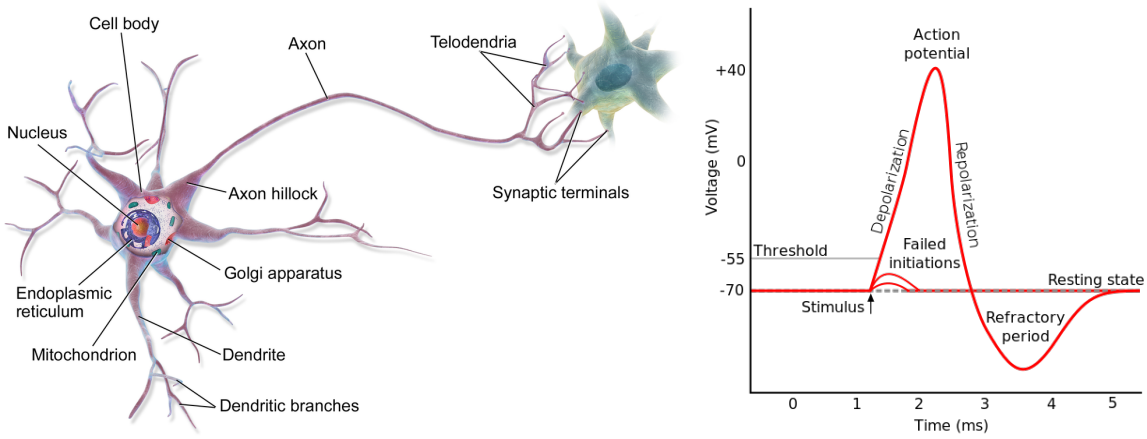


Figure 1.1: Neuron anatomy (BruceBlaus, CC-BY, source Wikipedia) and action potential response.

The neuron provides the inspiration for Rosenblatt's *perceptron*, a neuron model consisting of a linear combination of the inputs, plus a bias value, and a non-linear threshold response function:

$$y = \sum_{j=1}^{d} w_j x_j + w_0 \qquad s(y) = \begin{cases} 1, & y > 0 \\ 0, & y \leq 0 \end{cases}$$

Rosenblatt's algorithm for updating the weights of the perceptron made it possible to train the model by iteratively presenting it with examples and correcting mistakes until the best set of weights was found. The weights are updated according to the following rule:

$$w_i = w_i + \Delta w_i \qquad \Delta w_i = \eta(t - o)x_i$$

where t is the target label of the example, o the output of the *perceptron* for that example, x_i the input value for feature i and w_i the coefficient i of the perceptron. Since the output of the *perceptron* is either

0 or 1, as is the target class of each example, the training rule consists essentially of adjusting the weights of the *perceptron* for every example that is incorrectly classified. Figure 1.2 shows a schematic representation of the neuron model and the Rosenblatt perceptron.

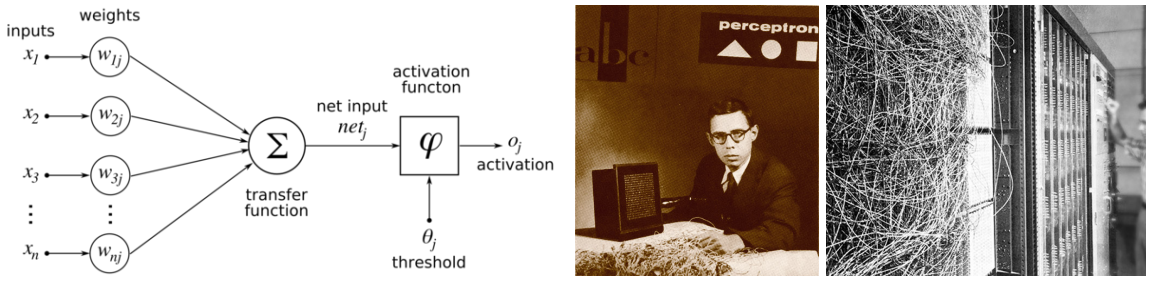


Figure 1.2: Neuron model, consisting of a weighted sum of the inputs, including a bias input of 1, and a non-linear activation function, such as the perceptron's threshold function. The middle and right panels show the original Rosenblatt perceptron (photos from the Arvin Calspan Advanced Technology Center and Hecht-Nielsen, R. Neurocomputing).

There were great hopes for this system capable of learning from examples. In 1958, the New York Times proclaimed it to be "the embryo of an electronic computer that [the Navy] expects will be able to walk, talk, see, write, reproduce itself and be conscious of its existence." Unfortunately, as it turned out, these initial neural models were equivalent to the generalized linear models already known to statisticians, such as logistic regression. Due to the limitations of the perceptron and the difficulties in training multi-layer networks, neural networks, and AI in general, went through a period of disappointment and skepticism between around 1960 and up to the mid 1980s, a period known as the "the AI winter". Although these systems could learn linear transformations of the features, the features needed to be chosen by the humans and only classes that could be linearly separated given those features could be adequately classified.

In 1986, Rumelhart, Hinton and Williams established that backpropagation as a method for training multi-layer neural networks [60]. Given that the activation functions at each layer provided a nonlinear transformation, training a stack of layers allowed the learning of sequences of nonlinear transformations. In the 1990s, AI research and applications shifted increasingly towards machine learning and data-driven applications. However, the practical limitations of fitting large neural networks meant that other machine learning approaches using nonlinear transformations were often more successful. Support vector machines using kernels, for example [7]. It also meant that, despite gaining the ability to solve nonlinear learning problems, it was still necessary for humans identify the right features.

In the late 1990s things started o improve for artificial neural networks. For example, by using convolution and multi-layered networks for processing checks [37] it was possible to automate reading handwritten digits. The modified National Institute of Standards and Technology (MNIST) database [40] pioneered the sharing of large datasets for machine learning research. In 2007, Hinton proposed a method for pre-training multi-layered networks one layer at a time [28], which helped solve the problems of backpropagation over multiple layers. In 2010, Hinton and Nair showed how Rectified Linear Units (ReLU) could improve training of neural networks, eventually overcoming the need for pre-training [28].

Furthermore, the improvement in hardware, with general-purpose graphics processing units (GPGPU)[52] and specialized processors such as the Tensor Processing Unit (TPU) from Google or the Nervana from Intel, keeps increasing the size of neural networks that can be trained. This created an important paradigm shift. Instead of using features selected by humans and learning a nonlinear

transformation, deep learning involves fitting deep stacks of nonlinear transformations on the raw data, dispensing with human intervention in the selection of features and making it easier to use unstructured data.

1.4 Machine Learning

Machine learning is the science of building systems that improve with data. This is a broad concept that includes instances ranging from self-driving cars to sorting images on a database and from recommendation systems for diagnosing diseases to fitting parameters in climate change models. The fundamental idea is that the system can use data to improve its performance at some task. Which immediately points us to the three basic elements of a well-posed machine learning problem:

1. The task that the system must perform.
2. The measure by which its performance can be evaluated.
3. The data that can be used to improve its performance.

Different tasks will determine different approaches. We may want to predict some continuous value, such as the price of apartments, which is a *Regression* problem. Or we may have a *Classification*, when we want to predict in which category, from a discrete set, each example belongs to. If we do this from a set of data containing the right answers, so we can then extrapolate to new examples, we are doing *Supervised Learning*.

There are other other types of problems that can be solved with machine learning, such as clustering, for example, which is an example of *Unsupervised Learning*. While *Supervised Learning* requires that all data be labelled, *Unsupervised Learning* uses unlabelled data. But it is possible to use data sets in which some data is labelled but the rest, usually most of the data, is not. In this case, we have *Semi-supervised Learning*. This approach has the advantage that, usually, unlabelled data is much easier to find than correctly labelled data. For example, it is possible to obtain from the World Wide Web many examples of English texts but to label correctly each grammatical element of each sentence would be very laborious. By combining clustering and classification it is possible to use unlabelled texts to improve the parsing and classification of elements from a set of labelled texts.

Regardless of the approach or the problem, the basic goal of machine learning is to create some representation of regularities in data.

Deep Learning

The term *deep learning* is used to refer to machine learning with models with multiple layers of nonlinear transformations. Though the idea may be more generic, this usually means neural networks with several layer that can learn different representations of the data. Figure 1.3 illustrates this for a generic classifier using deep learning, but the same idea could apply to regression or unsupervised learning such as with autoencoders.

A simple example is the solution of the exclusive or (XOR) function, which results in two classes that are not linearly separable, as Table 2.2 illustrates. As we will see in more detail later on, we can solve this using a neural network with two layers, with the neurons in the first layer transforming the data so that it can be linearly separated by the last layer. In other words, the hidden layer is learning a new representation of the data that allows it to be properly classified by the last neuron.

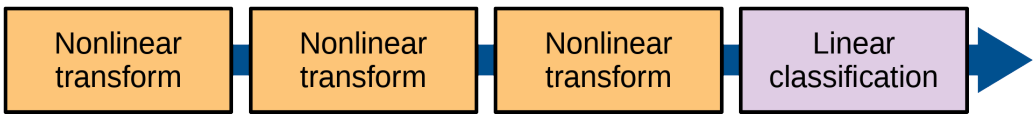


Figure 1.3: Schematic illustration of a deep learning model for classification. The last stage is a linear classifier operating on the result of multiple nonlinear transformations that change the representation of the original data.

Table 1.1: XOR

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

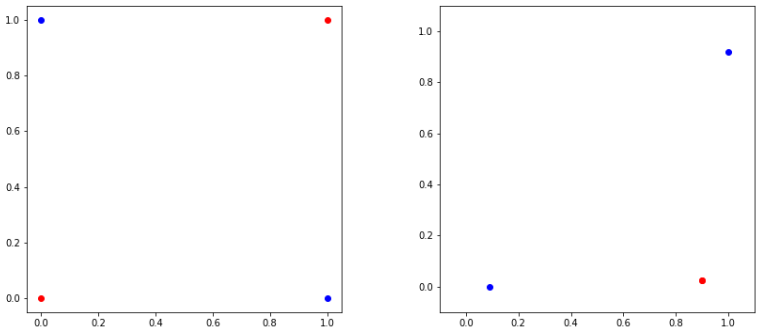


Figure 1.4: Original and transformed points from the OR function.

With more layers, more representations can be learned, each corresponding to a transformation of the representation in the previous layer. Figure 1.5 shows another data set and two transformations in the hidden layers, with 2 neurons each, resulting in a nearly linearly separable set of points that can be classified by the last neuron.

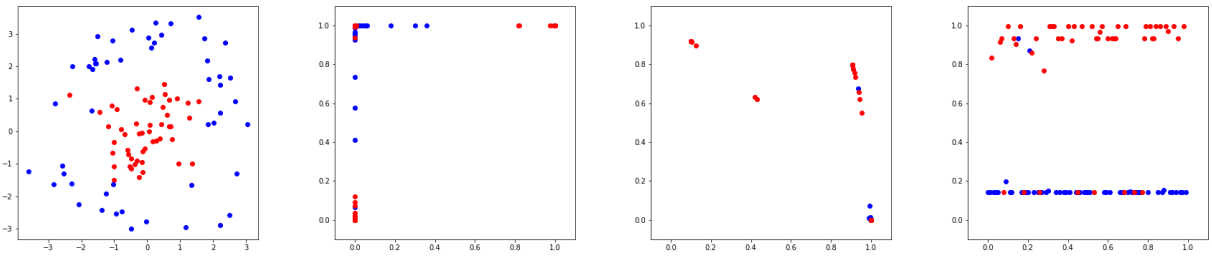


Figure 1.5: From left to right, the original data, the representation in the first hidden layer, in the second hidden layer and theclassification of each data point in the vertical axis, with the points sorted in the horizontal axis.

Deep learning can also be used in unsupervised learning in order to learn useful representations even without using class labels. The example shown in Figure 1.6 uses the banknote authentication data set from the UCI Machine Learning Repository [1]. This is a set of data from 1372 banknotes, divided in two classes (real and fake) and with each banknote described by four numerical features (variance, skewness and curtosis of wavelet transformed image and entropy). Figurerefbanknotes shows these four-dimensional data projected into the first two principal components using principal component analysis (PCA), the graph of an autoencoder with 6, 4, 2, 4, and 6 hidden layers trained to output values as close as possible to the input values and the representation of the data in the middle layer, with

[1] https://archive.ics.uci.edu/ml/datasets/banknote+authentication

2 neurons. We will revisit this later on, in more detail, both the computation graphs represented in Tensorboard and autoencoders, but at this point this serves to show how the representation learned by the deep network captures the structure of the data much better than classical methods such as PCA.

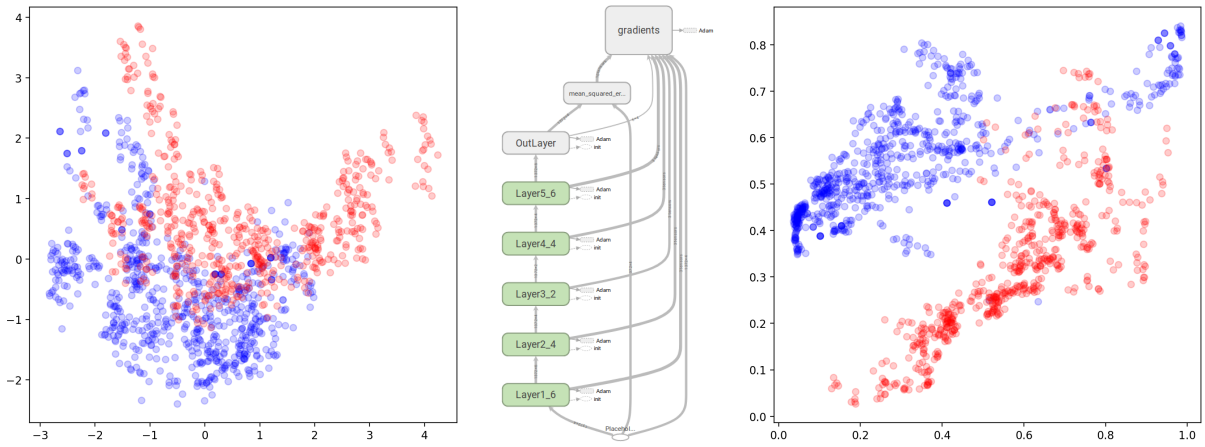


Figure 1.6: From left to right, a PCA projection of the original data using the first 2 components, the graph of the autoencoder and the representation learned in the middle hidden layer. The colors represent the different classes for the banknotes.

1.5 Linear Classifiers

Linear classification has a long history in statistics and machine learning. Examples include linear discriminant analysis, logistic regression and the original perceptron and support vector machine algorithms. These classifiers are adequate only if the classes to be distinguished can be separated by a linear combination of the features. Logistic regression, for example, finds a hyperplane where the probability of a point belonging to each of two classes, estimated with the logistic function, is equal:

$$g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x}) \qquad g(\vec{x}, \widetilde{w}) = \frac{1}{1 + e^{-(\vec{w}^T\vec{x} + w_0)}}$$

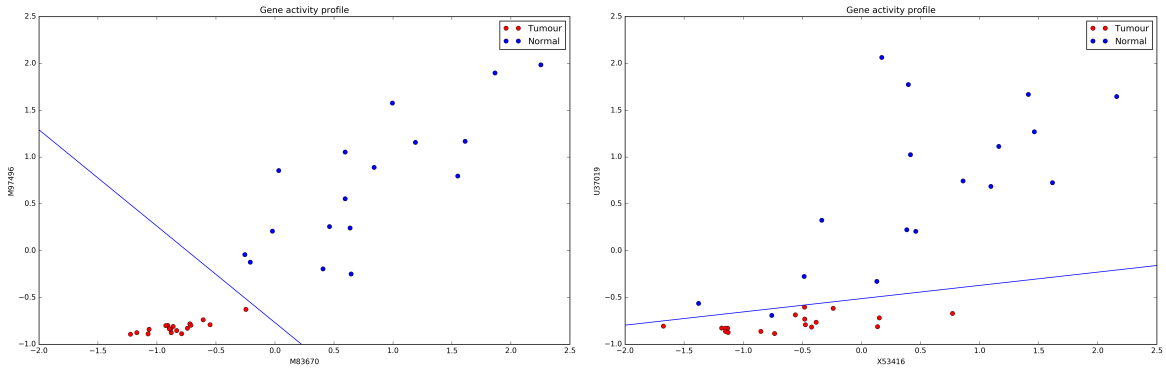


Figure 1.7: Linear classifiers are adequate for linearly separable classes but are unable to separate classes that are not linearly separable.

However, if the classes are not linearly separable, linear classifiers are inadequate. Figure 21.3 illustrates two data sets classified with logistic regression.

1.6 Shallow classifiers

One way of solving this problem is to expand the data with a non-linear transformation. Since the transformation is not linear, the data will be expanded into more dimensions and be spread out over a curved surface. With this transformed data, it may then be possible to separate the classes correcly with a linear classifier. Using the kernel trick, support vector machines can do this implicitly:

$$\arg\max_{\vec{\alpha}} \sum_{n=1}^{N} \alpha_n - \frac{1}{2} \sum_{n=1}^{N} \sum_{m=1}^{N} \alpha_n \alpha_m y_n y_m K(\vec{x}_n, \vec{x}_m)$$

Where $K(\vec{x}_n, \vec{x}_m)$, the kernel function, returns the dot product of some non-linear expansion ϕ of our original data. Figure 1.8 illustrates the difference between a linear classifier and a shallow, non-linear, classifier. This difference is due to a transformation of the selected features prior to classification. Generally, this involves expanding the features to a higher dimensional space (with more features) with a transformation that is not learned from the data by the classifier, but is fixed and chosen previously. In this example, the original features x_1 and x_2 were augmented by adding the computed features $x_1 x_2, x_1^2, x_2^2, x_1^3, x_2^3$.

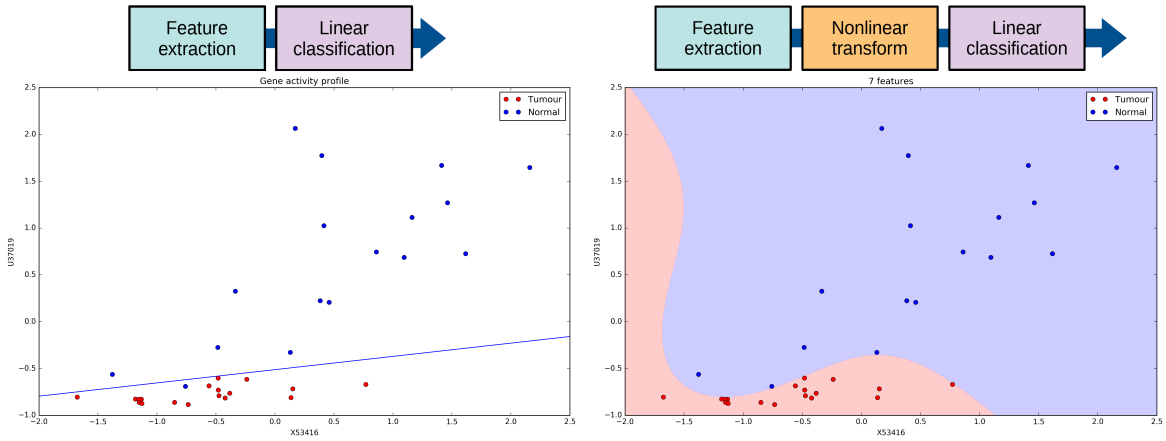


Figure 1.8: Linear classification without (left) or with (right) a non-linear transformation of the original features. In shallow classifiers, this transformation is generally controlled by hyperparameters that are fixed and not learned from the data. For example, the kernel function in a support vector machine.

1.7 Deep classifiers

Deep classifiers, such as deep neural networks, chain different non-linear transformations. This makes them highly non-linear, since each transformation operates on the result of the previous transformation, but also helps solve two problems that shallow classifiers have. Figure 1.9 illustrates this with the GoogLeNet network [71], which we will cover in a bit more detail in the next section.

This cascade of non-linear transformations in deep neural networks has advantages over the non-linear expansion typical of shallow classifiers. Since it is not a function of hyperparamaters that are fixed during training, but rather learned by the classifier, these transformations are adapted to the training data and result in more effective representations of different aspects of the data. The flexibility of these transformations and their stacking in different layers can also reduce the number of dimensions in which the data must be represented in order to be properly classified. For complex problems such as image or voice recognition, this can be a reason why shallow learning systems fail and deep learners

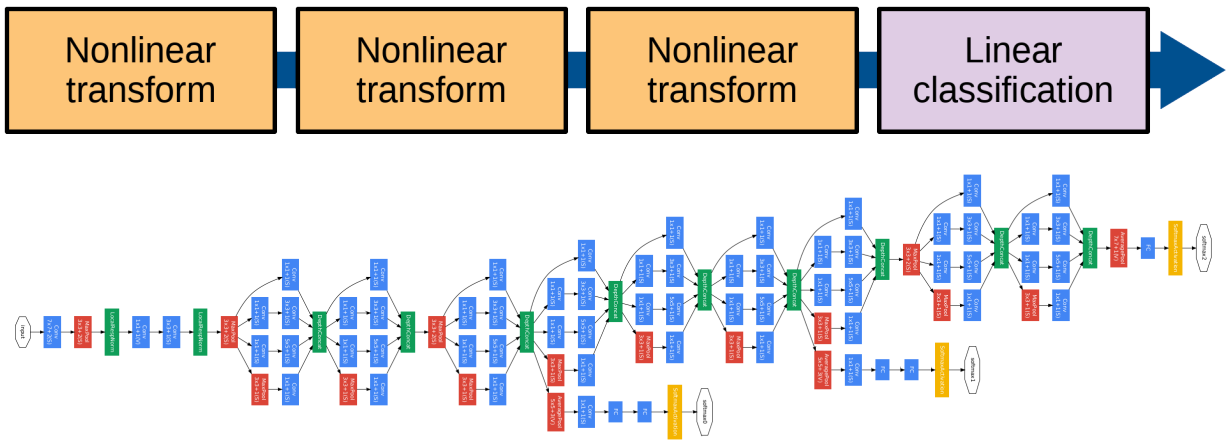


Figure 1.9: Example of a deep classifier, the GoogLeNet network, [71].

succeed. Figure 1.10 illustrates different representations at different layers of a deep neural network (a convolution network, in this case).

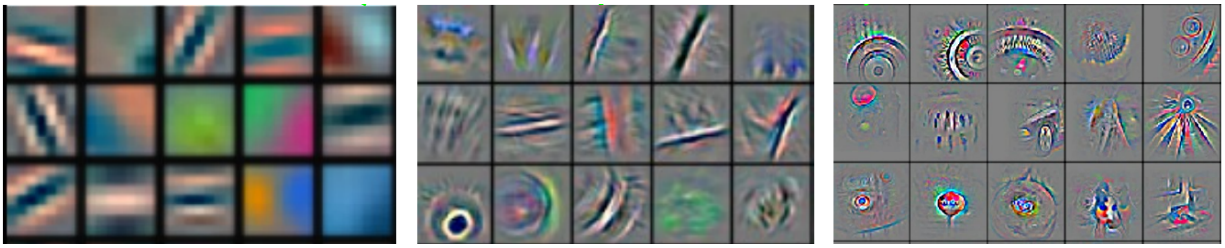


Figure 1.10: Example of representations in a deep convolution network, showing different aspects of the input images being detected at different layers, such as straight edges in the first layer, more complex edges in the second, complex shapes in the third and so on. Image from Zeitler 2014 [80].

No free lunch and overfitting

This is not to say that deep learning is, overall, better than shallow learning. In fact, there is no such thing as a better learning algorithm for the general cases. The "no-free-lunch" theorems prove that any learning algorithm that performs better than average for some type of problems will always pay for that performance by performing worse than average in other types of problems[78].

This also applies to the hypotheses obtained by instantiating any model. When we finish training a learning model, the set of parameters we obtained is especially suited to perform well over the training data but will perform poorly over data that is not similar to the training data. This is the problem of *overfitting*, which results from the model learning aspects of the training set that do not generalise to new data outside this set, thus resulting in a high true error despite a low training error. Figure 1.11 illustrates this, showing how the error measured with the training set or with data outside the training set (the test set) can diverge as we increase the power of the model to adjust to the training data.

Despite the advantages of deep learning models, these models always have a large number of parameters to be adjusted during training, and are extremely capable of adapting to the structure of the training data. This adaptability makes them better at solving complex problems but also more prone to overfitting if the training data is not representative of the universe of data where the learner is to be applied. In practice, this means that shallow learners can perform better than deep learners for simpler problems and when there are few examples available for training. It is only when the training

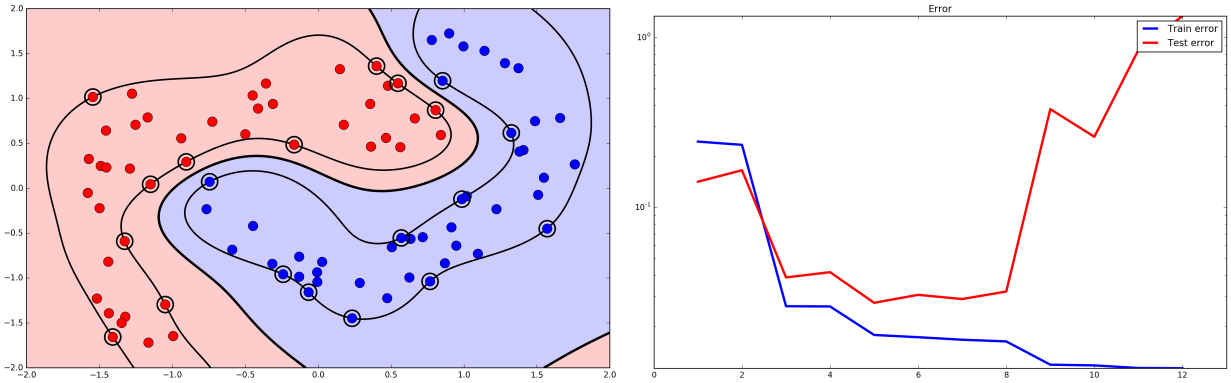


Figure 1.11: Example of overfitting. By adjusting too much to the training data, the resulting hypothesis may generalize poorly to data outside the training set (the test data in the right panel), leading to a greater true error, estimated by the test error, the greater the ability to adjust to the training data.

set is large enough to minimize overfitting that deep learning performs better, and in those cases the effectiveness of its representations enable this approach to solve problems that shallow learners cannot solve as effectively.

The availability of data, new techniques for training deep models and the development of efficient hardware for the computations is what made deep learning so successful in the last decade. For example, using the Caltech 101 images classification dataset, created in 2006 with 101 categories and an average of 50 images per category, classical computer vision algorithms and shallow classifiers performed best, with a 26% error rate. When the ImageNet database was created in 2012, with 1.2 million images in 1000 categories, deep convolution networks immediately dominated the competitions for image classification, beginning with the AlexNet, with a 15% error rate for top-5 predictions[35]. Over the years, results have been steadily improving but always with deep classifiers. For these problems and with such a large data set, shallow classifiers can no longer compete:

- AlexNet, Krizhevski et al. 2012, 15% top-5 error[35]
- OverFeat, Sermanet et al. 2013, 13.8% error[61]
- VGG Net, Simonyan, Zisserman 2014, 7.3% error[67]
- GoogLeNet, Szegedy et al. 2014 6.6% error[71]
- ResNet, He et al. 2015 5.7% error[54]

1.8 Examples

AlexNet was an early example of the power of deep networks for solving image classification problems. Used in 2012 for the ImageNet database, it was trained on two NVIDIA GeForce GTX 580 graphics cards and consisted in six convolution layers followed by three fully connected layers [35].

After this demonstration of de performance of deep models, the tendency was to increase the depth and complexity of the classifiers. In 2014, GoogLeNet reduced the top-5 classification error on ImageNet to 6.7% with 22 layers and 100 different processing blocks [71].

But deep models are not just for image classification. For example, in 2016 Hendricks et. al. [26] used a deep convolution networks to provide features for two long short term memory (LSTM) networks

so that, in addition to classifying images of birds, the system also learned to generate sentences to "explain" why the image was classified in that manner.

1.9 The promise of Deep Learning

There are several obstacles to solving a machine learning problem, even after we formulate the problem adequately by specifying what task we want the machine to solve, how to measure performance in that task and what data to use for training. In classical machine learning, the first step is to extract from the data the appropriate features that our model will use. This can be a demanding task, requiring human expertise in the domain of the problem. With deep learning this task can be greatly simplified because deep models have the ability to find good sequences of transformations that will extract the necessary features from the data.

We also need to choose the right model, and in classical machine learning this requires experimenting with fundamentally different approaches. A bayesian network, a support vector machine and a random forest classifier are very different algorithms, with different behaviors, requirements and computational methods. Deep neural networks provide a unified framework that can be used to create many different models that, despite being geared to different types of problems, all rest on the same basic principles.

1. Skansi, Introduction to Deep Learning, Chapter 1 [68]
2. Goodfellow *et. al.* Deep Learning [21], Chapters 1 and 5

Chapter 2

Deep learning, intro

Backpropagation. Stochastic Gradient Descent. Deep model architectures. Data and features. Why the success now? Some examples of deep learning with unstructured data

2.1 Backpropagation

As we saw, Rosenblatt's perceptron was only a linear classifier. However, if we can stack layers with non-linear responses we can go beyond linear classification. The problem with the original formulation of the perceptron is that the response function is discontinuous. This may be nearer to the biological features of the neuron but raises problems with the minimization of the error function if we stack several layers.

To solve this problem we can use a differentiable threshold function. One often used function is the *logistic* function, also called the *sigmoid* function:

$$s(y) = \frac{1}{1 + e^{-y}} = \frac{1}{1 + e^{-\vec{w}^T \vec{x}}}$$

But before we see how to solve the optimization problem for a neural network, we will start with a single neuron.

2.2 A Single Neuron

One possible way to train a logistic response neuron is by minimizing the squared error between the response of the neuron and the target class. So we minimize the error function:

$$E = \frac{1}{2} \sum_{j=1}^{N} (t^j - s^j)^2$$

We can do this in a way similar to the one used for the *perceptron*, by adjusting the weights of the neuron in small steps as a function of the error at each example j, $E^t = \frac{1}{2}(t^j - s^j)^2$, where t^j is the class of example j and s^j is the neuron's response for example j. To do this, we need to compute the derivative of the error as a function of the weights of the neuron in order to compute how to update the neuron weights. Since the error is a function of the activation of the neuron for example j (s^j), the

activation is a function of the weighted sum of the inputs (net^j) and this is, in turn, a function of the weights, we use the *chain rule* for the derivative of compositions of functions to obtain the gradient as a function of each weight:

$$-\frac{\delta E^j}{\delta w} = -\frac{\delta E^j}{\delta s^j}\frac{\delta s^j}{\delta net^j}\frac{\delta net^j}{\delta w}$$

where

$$s^t = \frac{1}{1+e^{-net^j}} \qquad net^j = w_0 + \sum_{i=1}^{M} w_i x_i$$

Since

$$\frac{\delta net^j}{\delta w} = x$$
$$\frac{\delta s^j}{\delta net^j} = s^j(1-s^j)$$
$$\frac{\delta E^j}{\delta s^j} = -(t^j - s^j)$$

We obtain the following update rule for the weight i of the neuron given example j:

$$\Delta w_i^j = -\eta\frac{\delta E^j}{\delta w_i} = \eta(t^j - s^j)s^j(1-s^j)x_i^j$$

Using this update function we descend the error surface in small steps in different directions according to each example presented to the net. With examples presented in random order, this is a *stochastic gradient descent*. Figure 2.1 illustrates this process of stochastically descending the error surface. The process of updating the weights at each example is called *online learning*. An alternative training schedule consists of summing the Δw_i^j updates for a set of examples and then updating the weights with the total change. This is called *batch learning*. These are examples of *stochastic gradient descent* because they are ways of descending along the gradient of the error function along random paths depending on the data. One pass through the whole training set is called an *epoch*.

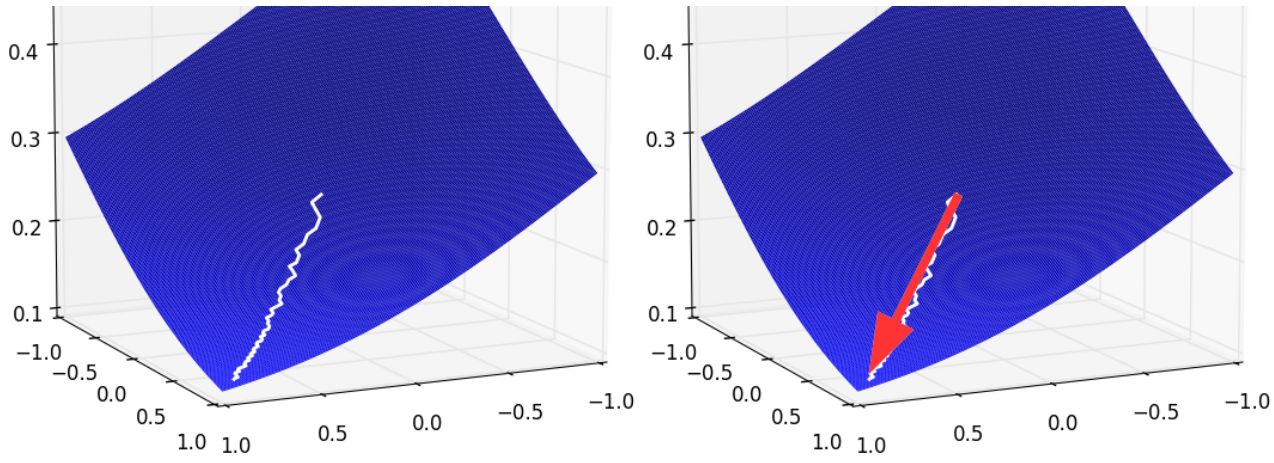


Figure 2.1: Stochastic gradient descent with online training (left panel) and batch training (right panel).

With a single neuron it is possible to learn to classify any linearly separable set of classes. One classical example is the `OR` function, as shown in Table 2.1.

Table 2.1: The OR function

x_1	x_2	OR
0	0	0
0	1	1
1	0	1
1	1	1

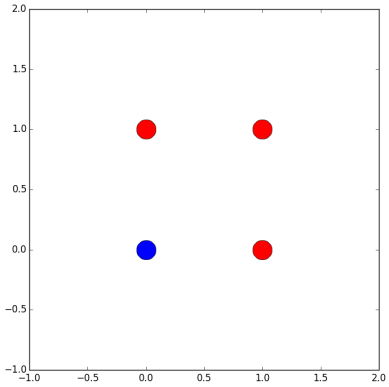


Figure 2.2: Set of points from the OR function.

Figureřef8-neuro-or shows the training error for one neuron being presented the four examples of the OR function and the final classifier, separating the two classes. The frontier corresponds to the line where the response of the neuron is 0.5.

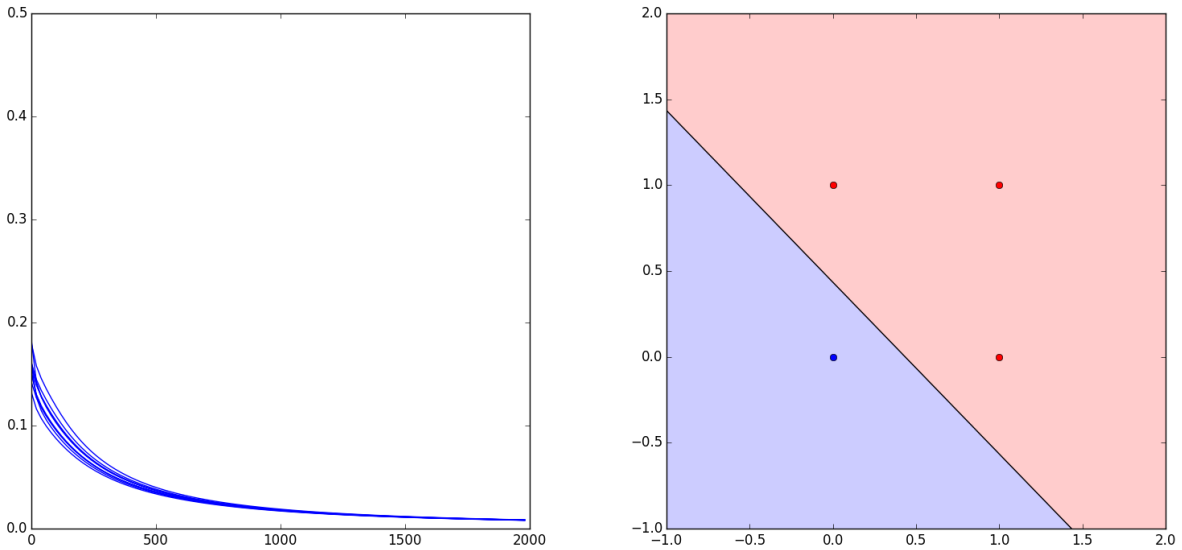


Figure 2.3: Training error and final classifier for one neuron trained to separate the classes in the OR function.

However, if the sets are not linearly separable, a single neuron cannot be trained to classify them correctly. This is because the neuron defines a hyperplane separating the two classes. For example, the exclusive or (XOR) function results in two classes that are not linearly separable, as Table 2.2 illustrates. So, if we try to train a neuron to separate these classes there is no reduction in the training error nor does the final classifier manage to separate the classes, as shown in Figure 2.5.

Table 2.2: The XOR function

x_1	x_2	XOR
0	0	0
0	1	1
1	0	1
1	1	0

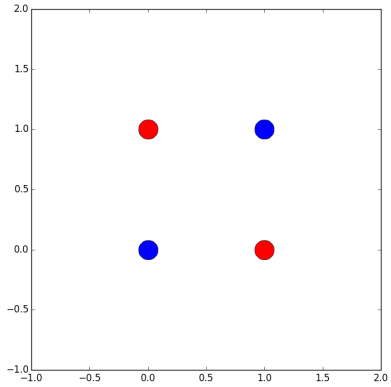


Figure 2.4: Set of points from the OR function.

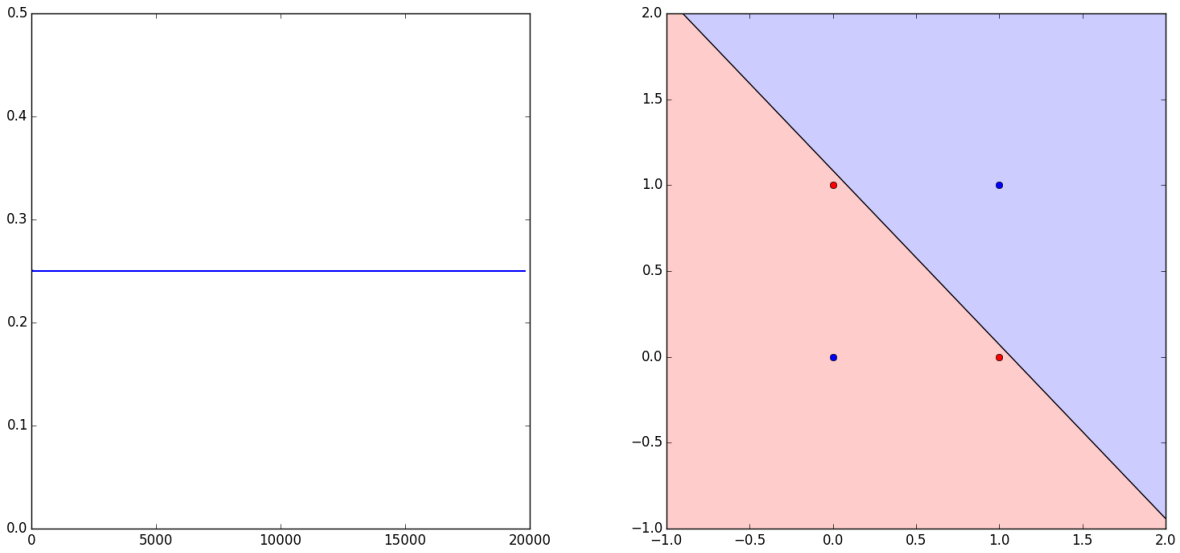


Figure 2.5: Training error and final classifier for one neuron trained to separate the classes in the OR function.

The solution for this problem is to add more neurons in sequence.

2.3 Multilayer Perceptron and Backpropagation

The *multilayer perceptron* is a fully connected, feedforward neural network. This means that each neuron of one layer receives as input the output of all neurons of the layer immediately before. Figure 2.6 shows two examples of multilayer perceptrons (MLP).

To update the coefficients of the output neurons, we derive the same update rule as for the single neuron with the only difference that the input value is not the value of an example feature but rather the value of the output of the neuron from the previous layer. Thus, the update rule for weight m of neuron n in layer k is:

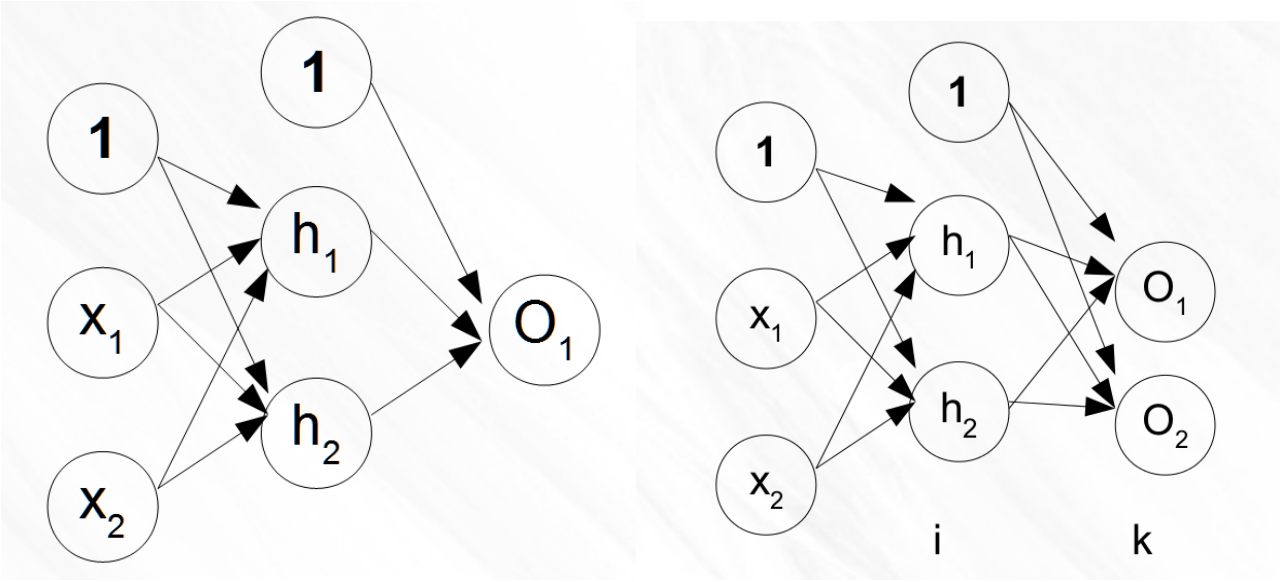


Figure 2.6: Two examples of multilayer perceptrons. Both have a hidden layer. The left panel shows a MLP with one output neuron, the right panel an MLP with two output neurons.

$$\begin{aligned}\Delta w^j_{m,k,n} &= -\eta \frac{\delta E^j_{k,n}}{\delta s^j_{k,n}} \frac{\delta s^j_{k,n}}{\delta net^j_{k,n}} \frac{\delta net^j_{k,n}}{\delta w_{m,k,n}} \\ &= \eta(t^j - s^j_{k,n}) s^j_{k,n}(1 - s^j_{k,n}) s^j_{i,n} = \eta \delta_{k,n} s^j_{k-1,n}\end{aligned}$$

Where $s^j_{k-1,n}$ is the output from neuron n of layer $k-1$.

For neurons in hidden layers, we need to backpropagate the error through the layers in front:

$$\begin{aligned}\Delta w^j_{m,i,n} &= -\eta \left(\sum_p \frac{\delta E^j_{k,p}}{\delta s^j_{k,p}} \frac{\delta s^j_{k,p}}{\delta net^j_{k,p}} \frac{\delta net^j_{k,p}}{\delta s^j_{i,n}} \right) \frac{\delta s^j_{i,n}}{\delta net^j_{i,n}} \frac{\delta net^j_{i,n}}{\delta w_{m,i,n}} \\ &= \eta(\sum_p \delta_{kp} w_{m,k,p}) s^j_{in}(1 - s^j_{i,n}) x^j_i = \eta \delta_{i,n} x^j_i\end{aligned}$$

The intuition for this is that the neuron in the hidden layer will contribute its output to several neurons in the layer ahead. Thus, we need to sum the errors from the neurons of the front layer, propagated through the respective coefficients of those front neurons.

This is the *backpropagation algorithm*:

- Present the example to the MLP and activate all neurons, propagating the activation forward through the network.
- Compute the $\delta_{n,k}$ for each neuron n of layer k, starting from the output layer and then backpropagating the error through to the first layer.
- With the $\delta_{n,k}$ values .

With this algorithm and the MLP architecture shown on the left panel of Figure 2.6, we can train the network to classify the XOR function output. During training the two neurons on the hidden layer learn to transform the training set so that their outputs result in a linearly separable set that the neuron on the output layer can then separate.

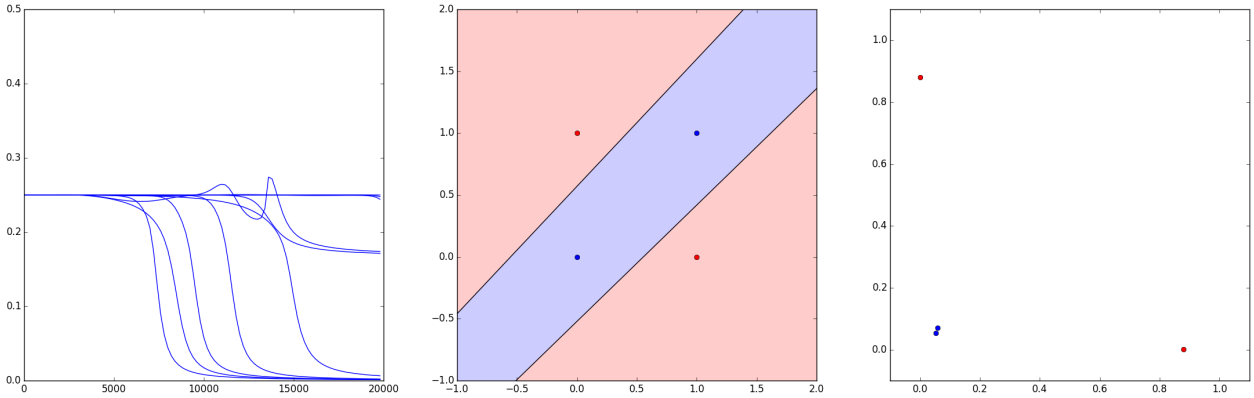


Figure 2.7: Training the MLP with one hidden layer for classifying the XOR function output. The first panel shows the training error over 10 training runs. Note that, due to the stochastic initialization and ordering of the examples presented, there are differences between different runs. The second panel shows the resulting classifier, successfully separating the classes. The third panel shows the output of the two neurons in the hidden layer of the network. This layer transforms the features of the training set making it linearly separable.

2.4 Data and features

Classical machine learning models work best with structured data, where all examples have the same set of features and each feature has a specific meaning. Each example differs in the values of its features, but the features are all the same. Thus, structured data can be represented as a table or matrix of features and examples. This is the best case for most machine learning algorithms because each feature of each example will provide a value in the input vector and these vectors will all be the same, both in size and in what each value represents. For example, we can characterize different patients by measuring temperature, blood pressure, glucose levels, height, weight and so forth, and each entry in the vector describing each patient would always correspond to one feature. This can be done either with numerical features or categorical features. The requirement for the data to be structured is simply that the features are fixed and the same for all examples.

Unstructured data is data that does not meet these requirements, and it is likely that the majority of useful data is stored without structure. Examples include text documents, phone conversations, video and photographs and social media posts and comments. In these cases, there is no predetermined relation between each value and features with some semantic significance. We can look at the brightness of the pixel at some position in all images but this pixel may be part of an eye, or hand, or wheel depending on the image.

Classical methods of dealing with unstructured data require a first step of feature extraction to create a structured representation of the data. Since this explicit step is guided by humans, using their knowledge of the domain, this is a labor-intensive. But, now, deep learning is changing the approach to unstructured data. Deep neural networks can be used to learn, automatically how to extract useful features from unstructured data, and so deep learning is making available a large volume of data that would otherwise be left unusable.

2.5 The Triumph of Deep Learning

(work in progress...)

2.6 Further Reading

1. LeCun, Bengio and Hinton, Deep learning, Nature 2015[38]
2. Goodfellow et. al., Chapter 5 and beginning of Chapter 6 (sections 6.1 and 6.2)[21]
3. Patterson and Gibson, Deep learning: A practitioner's approach, introduction to chapter 3 (pp 81-91)[51]

Chapter 3

Training Neural Networks

Algebra (revisions), The computational graph and AutoDiff, Training with Stochastic Gradient Descent.

3.1 Algebra

To understand how to implement artificial neural networks, and how to use `tensorflow`, we will start by reviewing some concepts of algebra. We will use the following terms:

- Scalar: a single number
- Vector: a one-dimensional array of numbers
- Matrix: a two-dimensional array of numbers
- Tensor: formally any relation between sets of algebraic objects, but we will use this term to refer to n-dimensional arrays of numbers

In particular, we will be using tensors. Hence the name `tensorflow`, which is fundamentally a library to apply operations to tensors. And that is why an important attribute of tensors is their shape, since we must take care how tensors fit with the operations.

One example of this is the algorithm for multiplying two matrices, or two 2D tensors. For the product $C = AB$, the matrix C is obtained by the sum of the element-wise products of each row of A with each column of B, as illustrated in Figure 3.1.

This is a very useful operation in artificial neural networks because, if matrix A contains a batch of examples with one example per row and one feature per column, and matrix B has the weights of the neurons of one layer in our network, with each neuron as a column, then the product C will have, for each row, the sum of the products of the features of each example by the weights of the neurons, with each neuron in each column and each example in each row. This means that we can simply add the bias vectors to matrix C and then apply the activation function for these neurons and we obtain a matrix that we can feed into the next layer, repeating these operations.

`tensorflow` even helps us do this for batches of matrices using higher dimensional tensors. If we use the `matmul` function we can perform matrix multiplication on the 2D matrices defined by the two

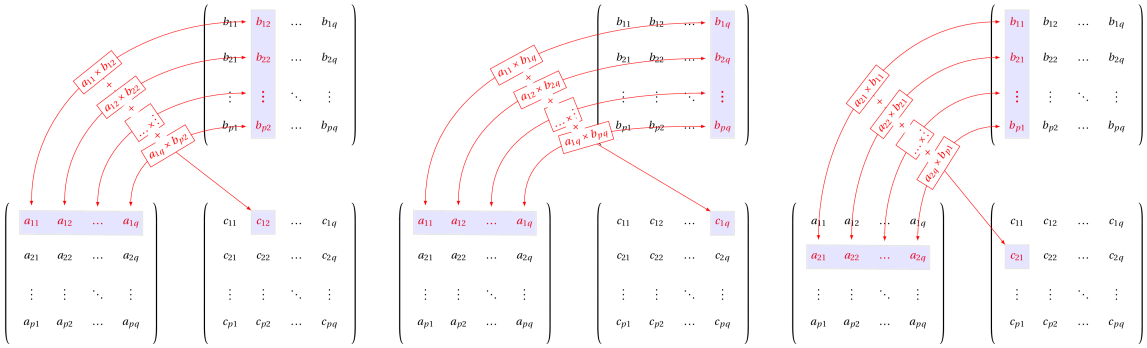


Figure 3.1: Matrix multiplication

last dimensions of the tensors, broadcast through the other dimensions using the normal broadcasting rules like in `numpy`. The code below illustrates this, by first creating two constant tensors of shapes (2,2,3) and (2,3,2) and then applying matrix multiplication between the two 2D matrices in each tensor. Note that the shapes of theses matrices, (2,3) and (3,2), match the requirements for matrix multiplication using algebra rules. The remaining dimensions of these tensors must allow broadcasting [1]

```
In : a = tf.constant(np.arange(1, 13, dtype=np.int32), shape=[2, 2, 3])
In : b = tf.constant(np.arange(13, 25, dtype=np.int32), shape=[2, 3, 2])
In : c = tf.matmul(a, b) # or a * b
Out: <tf.Tensor: id=676487, shape=(2, 2, 2), dtype=int32, numpy=
array([[[ 94, 100],
        [229, 244]],
        [[508, 532],
        [697, 730]]], dtype=int32)>
```

Further along this course we will be using the Keras API, so we will not generally have to worry with these low level operations. However, the shape of the tensors is always something to which we must pay attention.

3.2 Automatic Differentiation and the computational graph

As we saw previously, neural networks are trained through backpropagation, which requires computing the derivative of the loss function with respect to each adjustable parameter in the network. In previous examples, we solved this problem by manually computing the analytical expression of the derivatives. For example, for updating the weights of a single neuron with a quadratic loss function and sigmoid activation, we obtained this expression:

$$\Delta w_i^j = -\eta \frac{\delta E^j}{\delta w_i} = \eta (t^j - s^j) s^j (1 - s^j) x_i^j$$

However, it is easy to see how this would not be practical for large networks with different architectures. Symbolic derivation of all expressions would require a lot of work before we could train any network, since we would need the expressions for all adjustable parameters.

A generic approach to differentiation, when we cannot derive the analytical expressions, is to use numerical differentiation. This approximates derivatives by computing function values over small steps.

[1] For more details on broadcasting, see the documentation here: https://www.tensorflow.org/xla/broadcasting

The problem is that this would be too inefficient for training neural networks, since we would have to do numerical differentiation at each step during training, and has other problems such as accuracy and convergence conditions.

To solve these problems, `tensorflow` uses automatic differentiation, more specifically reverse-mode automatic differentiation, of which backpropagation is a particular case. Basically, `tensorflow` creates a computational graph where all the nodes are operations and the arcs are the tensors with the data being operated upon. This graph includes not only the operations we specify but also the corresponding derivatives, which are analytically defined for all operations `tensorflow` supports. During the forward pass through the computational graph, `tensorflow` records all operations and stores the intermediate outputs. Then in the backwards pass it computes the gradients by chaining the corresponding derivatives. This is done using a `GradientTape` object as we will see later. The details of reverse-mode automatic differentiation are complex and outside the scope of this course, but you can see the derivatives in the computation graph generated by `tensorflow`. For example, suppose we wanted to find the value of x that minimizes the cosine function. Associated with the cosine function, we also include the derivative in the computation graph as part of the gradients, as Figure 3.2 shows.

$$\underset{x}{\operatorname{argmin}}\,(\cos x) \qquad \frac{d\cos x}{dx} = -\sin x$$

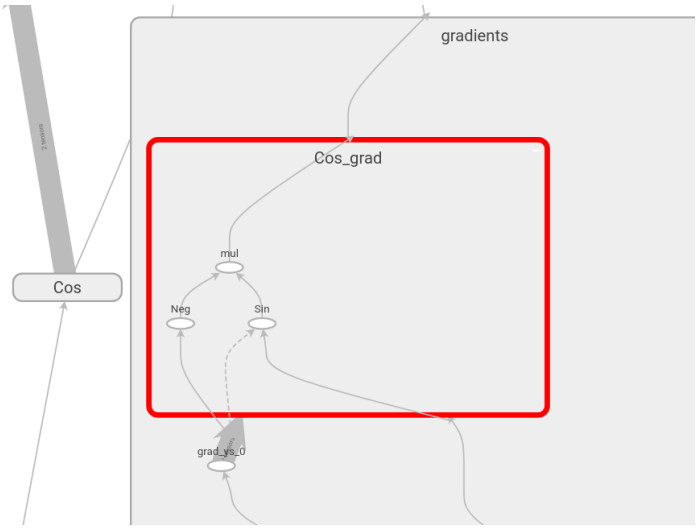


Figure 3.2: Matrix multiplication

3.3 Training with Stochastic Gradient Descent

To train the MLP it is important to start with small, random weights, close to 0. Initializing the network weights properly can be important but, unfortunately, there is no good understanding of precisely what the best way to do this may be. But it is clear that weights cannot be the same for all neurons, otherwise the gradient will be the same for all parameters and all neurons will be optimized in the same way. This symmetry must be broken from the start. It is not a problem to start with bias values at zero, but the weights must be randomized to guarantee that different neurons start at different combinations of parameters.

There are other considerations, depending on the networks. Recurrent networks are more susceptible to instability if the weights start too large and, in any case, large initial weights may saturate activations

or cause other numerical problems. On the other hand, larger weights are better at breaking neuron symmetry and "spread out" the network more widely from the start.

One standard way of choosing initial weights is to simply draw them at random from a Gaussian distribution with mean zero and variance 1. Other initialization schemes include taking into account the number of neurons in each layer, or the number of inputs, and other factors. See section 8.4 of [21] for more details, but bear in mind that weight initialization heuristics may not always give the best results when compared to a simple normal distribution.

This is because the sigmoidal activation functions saturate away from zero. It is also important to run the training process several times, since the training is not always exactly the same. Normalizing or standardizing the inputs is also important, since input features at different scales will force the network to adjust weights at different rates.

3.4 Further Reading

1. Goodfellow et. al., Chapter 2, 4 and 8 citeGoodfellow-et-al-2016

Chapter 4

Introduction to Tensorflow

Tensorflow basics: tensors ans computation graphs. Getting started with tensorflow: simple examples

4.1 Tensorflow

Tensorflow is a library for computing with tensors. A tensor is a multidimensional matrix, and Tensorflow allows us to specify a graph of operations that input and output tensors. This computation graph can then be compiled into CUDA for running on GPU and decomposed into several processors for more efficient execution. So to work with Tensorflow we must first define the computation graph and then execute it. This is done within a Tensorflow session. The thing to remember is that, unlike it is usual in imperative programming, the operations are not executed with the interpreter encounters them. Rather, they merely create the computation graph to be executed when the computations are run.

For example, to compute $a + b$ assigning 3 to a and 4 to b we can use the following code:

```
1 import tensorflow as tf
2 a = tf.constant(3.0, dtype=tf.float32, name='a')
3 b = tf.constant(4.0, name='b')
4 total = tf.add(a,b, name='total')
```

Fist we import the `tensorflow` library. In Tensorflow 2 the default execution mode is eager execution, meaning that each operation is executed by Tensorflow as it is interpreted by the Python interpreter. We will see another execution mode, graph execution, in which the whole computation graph is created first and then executed (this was the default in previous versions of Tensorflow).

We create the two constants `a` and `b`. The type of value can be specified in `dtype`; otherwise, it will be set according to the value of the constant. Then we use the addition operation that will output a tensor with the sum

4.2 Logistic Regression with Tensorflow

In a logistic regression, we assume that there is a function $g(\vec{x}, \widetilde{w})$ that, given an example $\vec{x}$ and some parameters $\widetilde{w}$ approximates the probability of the example belonging to some class. We also assume that the function has this shape:

$$g(\vec{x}, \widetilde{w}) = \frac{1}{1 + e^{-(\vec{w}^T \vec{x} + w_0)}}$$

The function

$$f(x) = \frac{1}{1 + e^{-k(x - x_0)}}$$

is called the *logistic function* and has the useful feature of varying from 0 to 1 around a threshold value but being nearly constant away from this threshold. This is what we need to find a hyperplane separating the two classes. The *logistic regression* is a regression model; when fitting the model we are trying to approximate this continuous probability function. However, because we also choose a cut-off value where we separate the two classes — where $P(C_1|\vec{x}) = P(C_0|\vec{x})$ — we turn this regression model into a classifier.

Now, given this function

$$g(\vec{x}, \widetilde{w}) = P(C_1|\vec{x})$$

the maximum likelihood solution for the problem of finding the parameters $\widetilde{w}$ is the minimum of this cost function:

$$E(\widetilde{w}) = -\sum_{n=1}^{N} [t_n \ln g_n + (1 - t_n) \ln(1 - g_n)]$$

with

$$g_n = \frac{1}{1 + e^{-(\vec{w}^T \vec{x_n} + w_0)}}$$

We will now see how to implement a logistic regression classifier with Tensorflow. We will be using the gene activity data shown previously. We will start by importing the data and standardizing it. The data is in a text file with one example per row and the two features values followed by the class label, separated by tabs. This is the beginning of our script:

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Logistic regression demo
"""
import tensorflow as tf
import numpy as np
import matplotlib.pyplot as plt

mat = np.loadtxt('gene_data.txt',delimiter='\t')

Ys = mat[:,-1]
Xs = mat[:,:-1]
means = np.mean(Xs,0)
stdevs = np.std(Xs,0)
Xs = (Xs-means)/stdevs
```

Standardization consists in subtracting the average and dividing by the standard deviation of all values in order to convert a distribution of values into a distribution with a mean of zero and a standard deviation of one. This is useful to prevent our parameters from varying over a too broad range of values. Now we create our model, which consists of the parameters and the prediction function, which simply applies the network operations.

```
18 weights = tf.Variable(tf.random.normal((2,1)), name="weights")
19 bias = tf.Variable(0.0, name="bias")
20
21 def prediction(X):
22     t_X = tf.constant(X.astype(np.float32))
23     net = tf.add(tf.matmul(t_X, weights), bias, name="net")
24     return tf.reshape(tf.nn.sigmoid(net, name="output"),[-1])
```

For the parameters of our model, we create two variables. Tensorflow variables are persistent objects, like constants, but they can be changed during the computation. Specifically, variables are what the optimizers will adjust in order to minimize a cost function. So our variables are the two weights that will be multiplied by the features and the bias value that will be added before the sigmoid activation. This is implemented in the `prediction` function, using the `sigmoid` activation from Tensorflow. The `prediction` function multiplies the inputs by the weights of the neuron, then adds the bias value and finally applies the sigmoid activation function. One important note: after the matrix operations, we have a two-dimensional matrix. This makes the output of the function be a two-dimensional matrix with a single column with the predictions for the batch of examples. However, since this is just a single column with the predictions, we want to reshape it into a one-dimensional vector so we can compare it with the vector of labels `Ys`. It is always important to match tensor shapes.

Now that we have our model we create the loss function, which will be minimized, and the function for computing the gradients of the loss function:

```
25 def logistic_loss(predicted,y):
26     t_y = tf.constant(y.astype(np.float32))
27     cost = -tf.reduce_mean(t_y * tf.math.log(predicted) \
28            + (1-t_y) * (tf.math.log(1-predicted)))
29     return cost
30
31 def grad(X, y):
32     with tf.GradientTape() as tape:
33         predicted = prediction(X)
34         loss_val = logistic_loss(predicted,y)
35     return tape.gradient(loss_val, [weights, bias]),[weights,bias]
```

The `logistic_loss` functions computes the logistic loss, which will be minimized by using a `GradientDescentOptimizer` object. To do this, the `grad` function uses a `tf.GradientTape` object to trace all computations and compute the derivatives. Within the context of this object, `tensorflow` keeps track of all operations and backtraces them to compute the gradients of the given tensor with respect to the specified variables. In this case, we want the gradients of the loss with respect to the weights and biases.

With everything setup, we create our training loop:

```
1 optimizer = tf.optimizers.SGD(learning_rate=0.1)
2
3 batch_size = 1
4 batches_per_epoch = Xs.shape[0]//batch_size
5 epochs=20
6
7 def run():
8     for epoch in range(epochs):
```

```
9         shuffled = np.arange(len(Ys))
10        np.random.shuffle(shuffled)
11        for batch_num in range(batches_per_epoch):
12            start = batch_num*batch_size
13            batch_xs = Xs[shuffled[start:start+batch_size],:]
14            batch_ys = Ys[shuffled[start:start+batch_size]]
15            gradients,variables = grad(batch_xs, batch_ys)
16            optimizer.apply_gradients(zip(gradients, variables))
17        y_pred =prediction(Xs)
18        loss = logistic_loss(y_pred,Ys)
19        print(f"Epoch {epoch}, loss {loss}")
```

In this loop, we take a batch of examples (by default it is a batch of 1 example only, but we can change that) after shuffling all examples, so that they are presented in a random order, and we pass that batch to the `grad` function, which computes the activations and the derivatives. Then we update the weights using the optimizer. The optimizer receives a list of tuples with each gradient and corresponding variable and updates the variables according to the update algorithm of the optimizer. In this case, if we use a simple SGD optimizer without momentum, this will just involve adding to the weights a fraction of the derivatives determined by the learning rate.

4.3 Solving the OR problem

We can use this same neuron to solve the OR problem. All we need to to is to change the data to examples of the OR function:

```
10 Xs = np.array([(0,0),(0,1),(1,0),(1,1)])
11 Ys = np.array([0,1,1,1])
```

We also need more epochs to train the neuron in this case, since each epoch consists of only four examples. You can also change the loss function to a quadratic loss:

```
10 def mse_loss(predicted,y):
11     t_y = tf.constant(y.astype(np.float32))
12     diffs= tf.math.square(t_y-predicted)
13     cost = tf.reduce_mean(diffs)
14     return cost
```

But the rest of the code should work unchanged.

4.4 Exercises

Start by implementing the examples shown above in this lecture. Make sure you understand the operations, the shapes of the resulting tensors and how everything fits together.

To solve the XOR problem, we need more than a neuron. Create a network with two neurons on the first layer (the hidden layer) and one output neuron. Note that you need to include this extra layer in the prediction function and also obtain the gradients to all the variable tensors you use for your network. Use these data for the XOR problem examples:

```
10 Xs = np.array([(0,0),(0,1),(1,0),(1,1)])
```

```
11 Ys = np.array([0,1,1,0])
```

Try varying the learning rate and use a larger number of epochs. You can also try using momentum. When you create the `SGD` optimizer you can add a `momentum` argument (*e.g.* 0.9). Then you can also apply this network to the data sets provided in the zip file and, if you want, make the network more complex by adding more neurons or even an additional layer. Note, however, that if you add more layers training may become significantly slower. We will see why in future lessons.

4.5 Questions

- If the batch of examples is presented as a matrix with examples in rows and features in columns, why do we encode the layers with the weights of each neuron in a column?
- What is the purpose of the `GradientTape` object used in the `grad` function?
- What does the optimizer do when you call the `apply_gradients` method?

Chapter 5

Activations and Loss Functions

Wide and deep networks. The vanishing gradient problem. Rectified Linear Units. Choosing activation functions

5.1 Wide vs Deep

The universal approximation theorem, proven by Cybenko in 1989 [14], implies that a single hidden layer is sufficient to approximate any function, to arbitrary precision, within a finite region. More specifically, given a function ϕ that is not constant, is bounded and continuous and the unit volume $I_m = [0,1]^m$ for any function f continuous in I_m there are N constants v_i, b_i and a $\vec{w}_i$ such that:

$$F(\vec{x}) = \sum_{i=1}^{N} v_i \phi(\vec{w_i}^T \vec{x} + b_i) \qquad |F(\vec{x}) - f(\vec{x})| < \epsilon$$

for all $\vec{x} \in I_m$, with $\epsilon > 0$.

However, this approximation may require a very large number of neurons.

Although it has still not been demonstrated whether deep or wide is best[43]. Telgarsky [73] demonstrated that deep networks are more capable of approximating oscillating functions with fewer elements. This is important not only for the ability to approximate functions in general but also because this is related to the *Vapnik-Chervonenkis dimension* (VC dimension) of a classifier. The VC dimension of a classifier is the dimension of the largest set of examples a model shatters. A model shatters a set of examples if it can provide hypotheses that classify all those examples correctly whatever the class each example belongs to. For example, a linear classifier in two dimensions can shatter a set of 3 points forming a triangle, as shown in Figure 5.1. However, with 4 examples this is not true, as we saw in the case of the XOR function previously.

In general, functions that are more capable of oscillations can be used as classifiers with higher VC dimension, suggesting that deep neural networks have a larger classification power than shallow and wide neural networks with the same number of neurons. However, this does not mean that deep networks will be best in all cases.

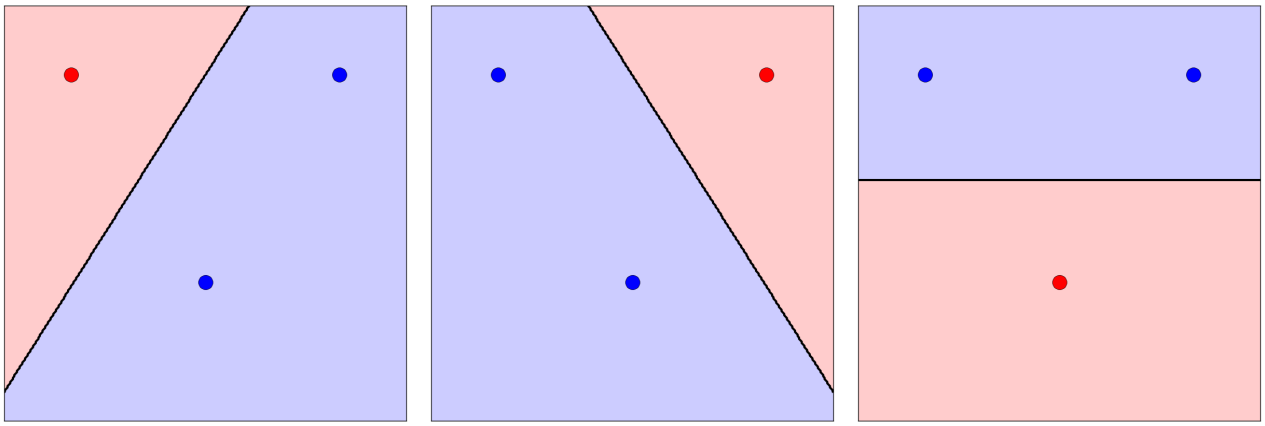


Figure 5.1: A linear classifier in two dimensions can shatter this set of 3 points by correctly classifying them whatever their labels. Note that two other cases, where all points belong to the same class, were omitted for being trivial.

5.2 No Free Lunch

"... for any two learning algorithms A and B, [...] there are just as many situations (appropriately weighted) in which algorithm A is superior to algorithm B as vice versa." (David Wolpert [78])

The "no free lunch" theorems are a set of theorems in statistics, machine learning and optimization that show that any approach that performs better at solving one problem will perform worse on other problems. Thus, considering the universe of all theoretically possible problems, no algorithm is better than any other. In the case of machine learning, Wolpert demonstrated that test error, which measures the capacity for generalization outside the training data, will be the same for all machine learning algorithms when averaged over all possible distributions of the data.

However, in real life data does not come from all mathematically possible distributions. Rather, for each problem, data follows some particular distribution and it is solely that distribution that is relevant. Thus, in real applications, there are algorithms and models that perform better than others. It is just a matter of what the data is like.

In general, shallow models composed of linear combinations combined with a non-linear transformation are easier to interpret and good at memorizing simple interactions between given features. For example, which other products were purchased by customers who purchased some product. However, these models are not suited to generalizing more complex properties unless given engineered features for that purpose.

Deep models are more powerful and better at generalizing and extracting relevant features, although this extra power comes at a greater risk of overfitting. For example, [22] tested a number of deep neural networks for face recognition and found that their performance degraded substantially if the image quality fell below what the networks were trained with.

In short, deep neural networks may not be the best choice in all cases, not only because of the need for a large training set to reduce overfitting but also because of how difficult it is to interpret the trained model. Figure5.2 shows some popular examples of images that can be easily confused. It is possible to train a deep neural network to distinguish even between categories as similar as these, but it is not possible to determine by examining a trained deep neural network if it learned the correct features because the features are extracted by the network itself and not easily intelligible, as is the case with

shallow models [1].

Figure 5.2: Three popular examples of confusing image datasets. Given a deep model, only by testing can we determine if it can distinguish between such examples since the resulting classifier is hard to interpret. Shallow models are easier to interpret but do not perform well in this type of problems. Image credits: teenybiscuit, Twitter.

5.3 Vanishing Gradients

One reason why deep neural networks were not popular for many years, even after backpropagation, was the difficulty in training such networks with classical activation functions, such as the sigmoid or the hyperbolic tangent functions. The reason for this is the chain rule in backpropagation combined with the shape of these functions. To propagate the error gradient backwards through the network we need to multiply the gradients at each step. For example, for the output neuron, we multiply the gradient of the error function, the activation function and the weighted sum of the inputs (from the previous layer) by the weights. For output neuron n of layer k receiving input from m from layer i through weight j, this is:

$$\Delta w^j_{mkn} = -\eta \frac{\delta E^j_{kn}}{\delta s^j_{kn}} \frac{\delta s^j_{kn}}{\delta net^j_{kn}} \frac{\delta net^j_{kn}}{\delta w_{mkn}} = \eta(t^j - s^j_{kn}) s^j_{kn}(1 - s^j_{kn}) s^j_{im} = \eta \delta_{kn} s^j_{im}$$

For a weight m on hidden layer i, we must continue the multiplications to propagate the output error backwards from all neurons ahead:

$$\Delta w^j_{min} = -\eta \left(\sum_p \frac{\delta E^j_{kp}}{\delta s^j_{kp}} \frac{\delta s^j_{kp}}{\delta net^j_{kp}} \frac{\delta net^j_{kp}}{\delta s^j_{in}} \right) \frac{\delta s^j_{in}}{\delta net^j_{in}} \frac{\delta net^j_{in}}{\delta w_{min}}$$

When using functions like sigmoid or hyperbolic tangent (see Figure 5.3), when the neuron is activated the output levels off at one. This makes the derivative of the activation function, δs, approach zero. Since we are multiplying all the derivatives, the result will tend to zero. The problem with this is that, although the neuron is active for a particular example, all derivatives computed through that neuron will be close to zero and thus will not inform the gradient descent to reduce the cost function. With networks formed of multiple layers of neurons this makes training so slow that backpropagation becomes impractical.

[1] See also performance tests of popular image recognition netw on the chihuahua muffin classification problem at freeCodeCamp:"Chihuahua or muffin? My search for the best computer vision API", by Maryia Yao.

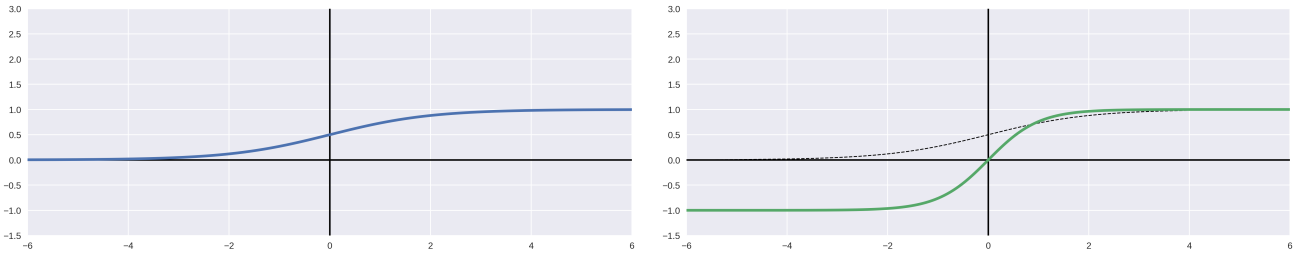


Figure 5.3: Two formerly widely used neuron activation functions: sigmoid and hyperbolic tangent. Note how the gradient drops to zero at high input values as the function saturates.

Until 2010, pre-training methods were used to help train deep networks. But with Rectified Linear Units (ReLU) the problem of vanishing gradients vanished.

5.4 Rectified Linear Units

Rectified Linear Units (ReLU) were first used in 2009 for object recognition [31], then for restricted Boltzmann machiones [50] and deep neural networks in 2011 [20]. The ReLU is a piecewise linear function that is zero for negative x_i inputs and equal to x_i for positive values of the input x_i. This is enough to provide the non-linearity necessary for the network while, at the same time, keeps the derivative constant and non zero when the neuron is active. Figure 5.4 shows several variants of ReLU. The leaky ReLU has a value of $a_i x_i$ for the output for negative values of x_i, with a fixed a_i value. The Randomized leaky ReLU is similar but a_i is random during training, drawn from a given probability distribution, and after training is fixed to the expected value of that distribution. In another variant, the Parametric ReLU, the a_i value is also learned during training. The Exponential Linear Unit is similar to a ReLU in the effect of avoiding the vanishing gradients for active neurons, but has a continuous derivative, with the activation value equal to $a(e^{x_i} - 1)$ for negative inputs [13]. These variants solve a problem with the original ReLU which is the "death" of neurons that become inactive over all the training set, since the gradients for these neurons in the original ReLU becomes zero and their weights will thus no longer change.

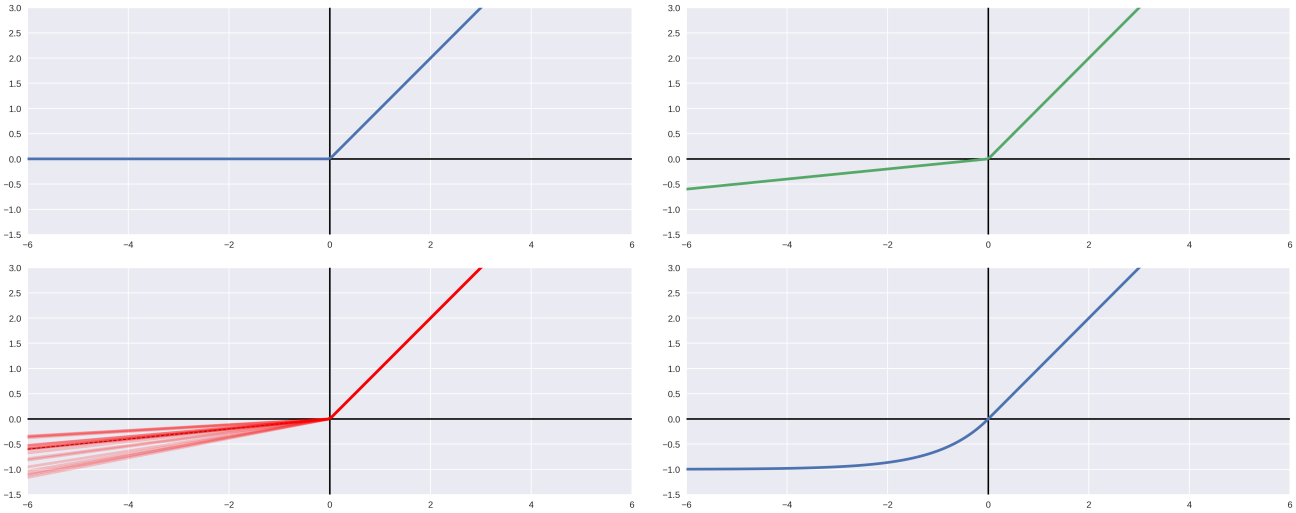


Figure 5.4: Rectified Linear Units and similar variants: ReLU; Leaky ReLU, with a non zero slope for negative values of x; Randomized Leaky ReLU, where the slope for negative inputs is random; and the Exponential Linear Unit, with a continuous derivative.

Concatenated ReLU is another ReLU variant that combines linear activations, with opposing slopes, for positive and negative inputs [62].

5.5 Choosing activation functions

In 2015, Xu *et al* compared the performance of different ReLU variants in deep convolution networks on the CIFAR image datasets, concluding that a non-zero slope in the negative inputs improves performance over the original ReLU. Thus, leaky variants of ReLU are generally used in the hidden layers of a deep network.

For the output layer, the activation function depends on the type of problem we are solving. For regression problems, the output neuron should not have an activation function, outputting only the weighted sum of the inputs multiplied by the neuron weights. This makes it possible for the neuron to output any desired value for the regression function.

For binary classification problems, it is useful to have an output value that can give us a probability of the example belonging to one of the two classes. One useful output function in this case is the signoid function.

For n-ary classification problems, the most used activation function is the *softmax* function:

$$\sigma : \mathbb{R}^K \rightarrow [0,1]^K \qquad \sigma(\vec{x})_j = \frac{e^{x_j}}{\sum_{k=1}^{K} e^{x_k}}$$

This function returns a vector of values where $\sigma_j \in [0,1]$ and $\sum_{k=1}^{K} \sigma_k = 1$, which means that σ_j can be used to represent the probability of the example belonging to class j of k.

5.6 Loss and Likelihood

We train neural networks by minimizing a loss (or cost) function with the objective of finding the best parameters. But how do we decide which function to miminize? One approach is to choose a function that corresponds to a *maximum likelihood* solution, meaning that we instantiate our model with parameters that maximize the probability of the data being distributed as we find it to be. This is a common approach to machine learning and neural networks [2]. Let us consider a simple problem of linear regression.

A *linear regression* is a regression in which the hypothesis class corresponds to the model $y = \theta_1 x_1 + \theta_2 x_2 + ... + \theta_{n+1}$, where each x_n is one dimension of the input space. Suppose, to simplify, that our input space has only one dimension and we have a set of (x, y) points and want to find the best way to predict the y value of each point given the x value assuming that the best fit is some straight line $y = \theta_1 x + \theta_2$. Figure 5.5 shows an example of a data set of points and possible lines from our *hypothesis class*, obtained by instantiating the model with different values of θ_1 and θ_2.

How can we determine the best line? Let us assume that the dependent variable y is some (unknown) function of the independent variable x plus some error:

$$y = F(x) + \epsilon$$

[2] Another good approach is Bayesian learning, but we will not cover that in this course

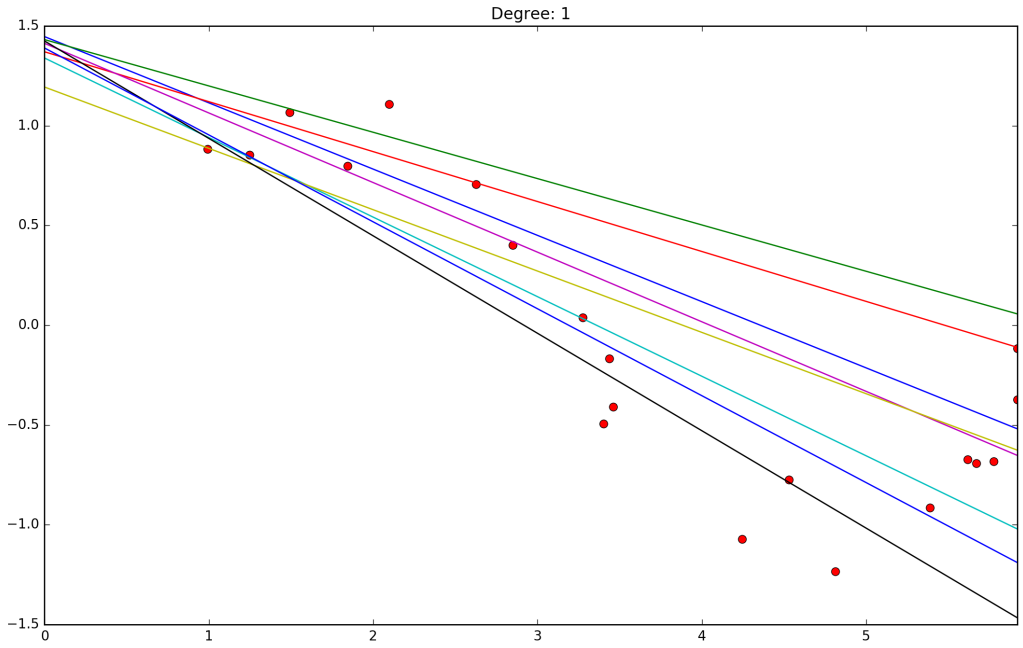


Figure 5.5: Example of lines for predicting the y values in these data.

We want to approximate $F(x)$ with a model $g(x, \theta_1, \theta_2)$. Assuming that the error is random and normally distributed:

$$\epsilon \sim N(0, \sigma^2)$$

then, if $g(x, \theta_1, \theta_2)$ is a good approximation of the true function $F(x)$, the probability of having a particular y value given some x value can be computed from our function $g(x, \theta_1, \theta_2)$ as:

$$p(y|x) \sim \mathcal{N}(g(x, \theta_1, \theta_2), \sigma^2)$$

This allows us to estimate the probability of the data coming out with the distribution we observe in our data set given any hypothesis instantiating θ, representing the vector of all $\theta_1, ..., \theta_n$ parameters (in this case, θ_1, θ_2). The probability of the data given the hypothesis is the *likelihood* of the hypothesis. Note that we cannot assume a probability for the hypothesis, at least in a frequentist sense, because the hypothesis is not a random variable. What we assume to be random here is the sampling of data that resulted in obtaining this dataset from the universe of all possible data.

Thus, given our dataset $\mathcal{X} = \{x^t, y^t\}_{t=1}^N$ and knowing that $p(x, y) = p(y|y)p(x)$, then the likelihood of the set of parameters θ is

$$l(\theta|\mathcal{X}) = \prod_{t=1}^{n} p(x^t, y^t) = \prod_{t=1}^{n} p(y^t|x^t) \times \prod_{t=1}^{n} p(x^t)$$

Now we know how to choose the best hypothesis: we pick the one with the *maximum likelihood*. In other words, we pick the hypothesis that estimates the largest probability of obtaining the data we have. To simplify, let us change the expression by noting that the hypothesis that maximizes the likelihood also maximizes the logarithm of the likelihood, so we can focus on the logarithm of the likelihood, $\mathcal{L}$, instead of the likelihood l:

$$\mathcal{L}(\theta|\mathcal{X}) = log \prod_{t=1}^{n} p(y^t|x^t) + log \prod_{t=1}^{n} p(x^t)$$

We can also ignore the $p(x)$ term since this corresponds to the probability of drawing those x values in our data from the universe of possible values and this is the same for all hypotheses (all values of θ) we are considering.

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} p(y^t|x^t)$$

Since we assume that the probability of obtaining some y value given some x is approximately normally distributed around our prediction, we can replace that term with the corresponding distribution:

$$p(y|x) \sim \mathcal{N}(g(x,\theta), \sigma^2)$$

and then replace it with the expression for the normal distribution:

$$\mathcal{N}(z,\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}} e^{-(z-\mu)^2/2\sigma^2}$$

leaving:

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} \frac{1}{\sigma\sqrt{2\pi}} e^{-[y^t - g(x^t|\theta)]^2/2\sigma^2}$$

which can be simplified to:

$$\mathcal{L}(\theta|\mathcal{X}) \propto log \prod_{t=1}^{n} e^{-[y^t - g(x^t|\theta)]^2}$$

$$\mathcal{L}(\theta|\mathcal{X}) \propto -\sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

But this is the expression of the square of the training error:

$$E(\theta|\mathcal{X}) = \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

So, basically, to find the hypothesis with the *maximum likelihood* we need (under our assumptions) to find the hypothesis with the *minimum squared error* on our training set. This problem is called a *Least Mean Squares minimization*.

Note that the squared error is often represented by this expression:

$$E(\theta|\mathcal{X}) = \frac{1}{2} \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

The reason for this is that, when computing the derivative of this error as a function of the parameters, the square power cancels the 2 in the denominator, simplifying the algebra. However, the values obtained for the parameters minimizing the squared error or one half the squared error are the same. This is merely an algebraic convenience.

Now we can do gradient descent on this quadratic error curve to find the best set of parameters, as Figure 5.6 illustrates

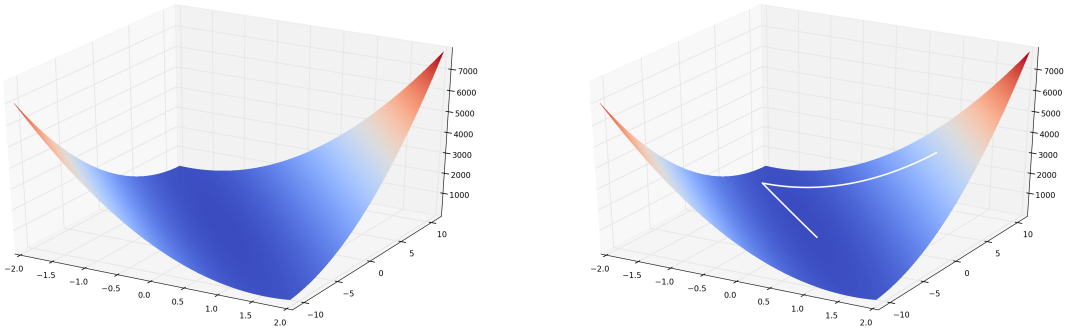


Figure 5.6: Gradient descent on the squared error surface.

5.7 Maximum Likelihood

For a general case, suppose we have a set of examples $\mathbb{X} = \{x^1, ..., x^m\}$ drawn randomly from the population with some probability distribution. We also have a family of probability distributions $p_{model}(x; \theta)$ which tell us the probability of x as a function of θ. The maximum likelihood estimator for θ is the vector that maximizes the joint probability of all the examples being what they are:

$$\theta_{ML} = \arg\max_{\theta} p_{model}(x; \theta) = \arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^i; \theta)$$

Since multiplying many small numbers will often lead to numerical underflow, it is best to use logarithms:

$$\arg\max_{\theta} \prod_{i=1}^{m} p_{model}(x^i; \theta) = \arg\max_{\theta} \sum_{i=1}^{m} \log\, p_{model}(x^i; \theta)$$

Since the parameters corresponding to the maximum value do not change by a simple rescaling of the function, we can divide by m consider the expectation function of the log-probabilities of an x according to our model considering the empirical distribution of examples in our data. In other words, we are maximizing the probability of any x given by our model weighted by the probability of that x being drawn from the distribution probability of examples.

$$\arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} \log\, p_{model}(x; \theta)$$

This value is maximized when $p_{model}(x; \theta)$ is as similar as possible to the distribution of examples $hatp_{data}(x)$. This can be measured by the Kullback–Leibler divergence, a measure of the dissimilarity between different distributions and which is the expectation of the log-probability differences between them. In our case:

$$D_{KL}(\hat{p}_{data}, p_{model}) = \mathbb{E}_{x \sim \hat{p}_{data}} [\log \hat{p}_{data} - \log p_{model}]$$

Since $\hat{p}_{data}$ does not depend on θ, minimizing the KL divergence is the same as minimizing the expectation of the $-\log p_{model}$, which is equivalent to the maximization we had before:

$$\arg\min_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} - \log\, p_{model}(x; \theta) = \arg\max_{\theta} \mathbb{E}_{x \sim \hat{p}_{data}} \log\, p_{model}(x; \theta)$$

This means that maximizing likelihood is equivalent to minimizing the divergence between the probability distribution of the data we draw and the probability distribution of the data given by your model. This corresponds to minimizing cross-entropy between the two distributions.

In supervised learning, we want to adjust our parameters θ to predict some set of target values or labels Y from the features X. So in this case the maximum likelihood solution can be written as

$$\theta_{ML} = \arg\max_{\theta} P(Y|X;\theta) = \arg\max_{\theta} \sum_{i=1}^{m} \log P(y^i|\vec{x}^i;\theta)$$

For linear regression, as we saw before, we assumed that any y is given by our function plus a normally distributed error around that value, $p(y|x) \sim \mathcal{N}(g(x,\theta),\sigma^2)$, and in this case the cost function (cross-entropy) is the squared error:

$$E(\theta|\mathcal{X}) = \sum_{t=1}^{n} [y^t - g(x^t|\theta)]^2$$

5.8 Binary Classification

If we have a binary classification problem using a sigmoid activation function to predict the probability of each example belonging to one class based on the θ parameters and the features x, then if $g(\vec{x},\theta) = P(t_n = 1|\vec{x})$ and $t_n \in \{0,1\}$, the maximum likelihood solution corresponds to:

$$\mathcal{L}(\theta|X) = \prod_{n=1}^{N} \left[g_n^{t_n}(1-g_n)^{1-t_n}\right] \quad l(\theta|X) = \sum_{n=1}^{N} [t_n \ln g_n + (1-t_n)\ln(1-g_n)]$$

where $g(\vec{x},\theta)$ is the sigmoid activation of our output neuron. In this case, the maximum likelihood solution corresponds to minimizing the logistic loss:

$$E(\widetilde{w}) = -\frac{1}{N}\sum_{n=1}^{N} [t_n \ln g_n + (1-t_n)\ln(1-g_n)] \quad g_n = \frac{1}{1+e^{-(\vec{w}^T\vec{x_n}+w_0)}}$$

5.9 Multi-class Classification

If we have a multi-class classification problem, then we want to use the softmax function to predict the probability of each example belonging to each class, as we saw in the previous chapter, and the corresponding maximum likelihood solution is the softmax cross entropy:

$$-\sum_{c=1}^{C} y_c \log \frac{e^{a_c}}{\sum_{k=1}^{C} e^{a_k}}$$

5.10 Further Reading

1. Goodfellow et. al., Chapters 5 and 6[21]
2. Tensorflow, activation functions and linked relevant papers: https://www.tensorflow.org/api_guides/python/nn#Activation_Functions

Chapter 6

Tensorflow and Tensorboard

AutoGraph and graph mode. Tensorboard. Examples

6.1 Tensorboard (Tutorial)

Tensorboard is a visualization tool that makes it easy to monitor the training and other statistics for our models. To use Tensorboard, we must create a log file in a log folder. Tensorboard expects the log for each run to be in a different subfolder of a given logging folder. To help with this, we can create such a subfolder using the current timestamp, so that each run gets logged in its own folder. Here is an example of the code for this:

```
# -*- coding: utf-8 -*-
"""
Logistic regression demo
"""
import tensorflow as tf
import numpy as np
from datetime import datetime
now = datetime.utcnow().strftime("%Y%m%d%H%M%S")
root_logdir = "logs"
log_dir = "{}/model-{}/".format(root_logdir, now)
```

This will set as the log folder the folder `logs/model-[timestamp]` each time the script is run. Now we need to write our data to the log file in this folder. To do that, after the model is setup and before we begin training, we create a file writer object with the computation graph. But before we do that, we need to take a look at how to create a computation graph.

Graph mode

So far we have used Tensorflow with eager execution, where operations are compiled for the appropriate hardware and executed when the Python interpreter interprets each instruction. But Tensorflow can also be used in graph mode, in which a computation graph is first laid out before being distributed across the different processors (CPU, GPU or TPU) and executed.

To do this we can use AutoGraph, which converts Python code into Tensorflow graph code. To do this we just need to use `tf.function`, which receives a Python function and constructs a new

function that executes a Tensorflow graph by tracing and compiling all operations starting from the functions supplied. This can be done easily in our code by using the `@tf.function` decorator.

We will not cover graph mode in detail, but this can be useful to improve performance. By building the complete graph before having Tensorflow compile it, Tensorflow can optimize the execution of the graph much better than with eager execution. However, using graph mode requires deeper understanding of the workings of Tensorflow than what we cover in this course, so we will be using it only for exporting the computational graph to Tensorboard. For more information on optimizing computations with AutoGraph, please see the Tensorflow documentation [1]

Building the graph

Let us start by creating our model and computation. We will use the logistic regression example we saw before, applied to the gene data. However, we will write all the functions assuming the inputs are tensors, since AutoGraph will build all the graph with tensors and not Numpy arrays. Here is our model and loss function, using the logistic loss:

```
13 weights = tf.Variable(tf.random.normal((2,1)))
14 bias = tf.Variable(0.0)
15
16 def prediction(X):
17     net = tf.add(tf.matmul(X, weights), bias, name="net")
18     return tf.reshape(tf.nn.sigmoid(net, name="output"),[-1])
19
20 def logistic_loss(predicted,Y):
21     cost = -tf.reduce_mean(Y * tf.math.log(predicted) + (1-Y) * (tf.math.log(1-predicted)))
22     return cost
23
24 def grad(X, y):
25     with tf.GradientTape() as tape:
26         predicted = prediction(X)
27         loss_val = logistic_loss(predicted,y)
28     return tape.gradient(loss_val, [weights, bias]),[weights,bias]
```

Note that we reshape the output of `prediction` so that it is a vector and not a matrix. We also assume that all arguments are tensors in these functions. This is useful to create the computation graph using only tensors.

So now we can create the code to write the graph to Tensorboard. Note that we will be running our model with eager execution, as before, but we can compile a graph just to save it and see the whole computation. We will include everything in the graph: the model, the loss function and the gradients.

To do this, we create a function just to trace all the computations with AutoGraph, and then save the corresponding graph. Here is the code:

```
30 @tf.function
31 def create_graph(X,Y):
32     grad(X,Y)
33
34 def write_graph(X,Y,writer):
35     tf.summary.trace_on(graph=True)
36     create_graph(tf.constant(X.astype(np.float32)),tf.constant(Y.astype(np.float32)))
```

[1] https://www.tensorflow.org/guide/function

```
37     with writer.as_default():
38         tf.summary.trace_export(name="trace",step=0)
39
40 mat = np.loadtxt('gene_data.txt',delimiter='\t')
41
42 Ys = mat[:,-1]
43 Xs = mat[:,:-1]
44 means = np.mean(Xs,0)
45 stdevs = np.std(Xs,0)
46 Xs = (Xs-means)/stdevs
47
48 writer = tf.summary.create_file_writer(log_dir)
49 write_graph(Xs,Ys,writer)
```

The `create_graph` function simply calls the `grad` function. However, since it is decorated with the `@tf.function` decorator, it will be wrapped by Tensorflow's `tf.function`, which implements AutoGraph, and will trace all the computations over the calls to the gradient, loss and predictions and build the complete graph.

The `write_graph` function shows how we can save this trace to Tensorboard. We start the trace with a call to the `f.summary.trace_on` function and then export the graph with the `tf.summary.trace_export` function, which terminates the trace. Since events written to the the Tensorboard logs need an integer value indicating at which step the event occurs, we add the `step=0` argument to the export function. To export the graph we need a Tensorboard file writer, which we create in line 48.

This code does not yet train our model but we can run it and see the graph in Tensorboard. After running this script, open a shell console in you working folder and type:

```
> tensorboard --logdir logs
```

Then point your browser to the URL `http://127.0.0.1:6006`. Tensorboard runs as a local http server. Figure 6.1 shows the Tensorboard interface and the graph.

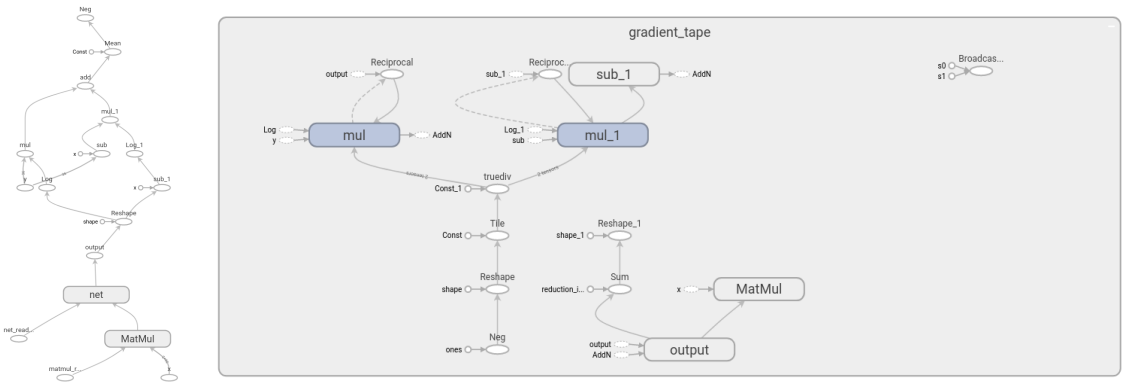


Figure 6.1: The computation graph for our logistic regression.

Monitoring training

Let us continue with the logistic regression example. We have nearly everything set up, we just need the code for training our model. But we will add to the Tensorboard summary the training error at each epoch, as shown below (lines 70 and 71).

```
optimizer = tf.optimizers.SGD(learning_rate=0.1)

learning_rate = 0.01
batch_size = 1
batches_per_epoch = Xs.shape[0]//batch_size
epochs = 50

for epoch in range(epochs):
    shuffled = np.arange(len(Ys))
    np.random.shuffle(shuffled)
    for batch_num in range(batches_per_epoch):
        start = batch_num*batch_size
        batch_xs = tf.constant(Xs[shuffled[start:start+batch_size],:].astype(np.float32))
        batch_ys = tf.constant(Ys[shuffled[start:start+batch_size]].astype(np.float32))
        gradients,variables = grad(batch_xs, batch_ys)
        optimizer.apply_gradients(zip(gradients, variables))
    y_pred =prediction(tf.constant(Xs.astype(np.float32)))
    loss = logistic_loss(y_pred,Ys)
    print(f"Epoch {epoch}, loss {loss}")
    with writer.as_default():
        tf.summary.scalar('Train loss', loss, step=epoch)

writer.close()
```

Now we can look at our training results, shown in Figure 6.2. Things seem to be going well for the first 30 epochs, but then we start getting `NaN` values, indicating something went wrong with our training.

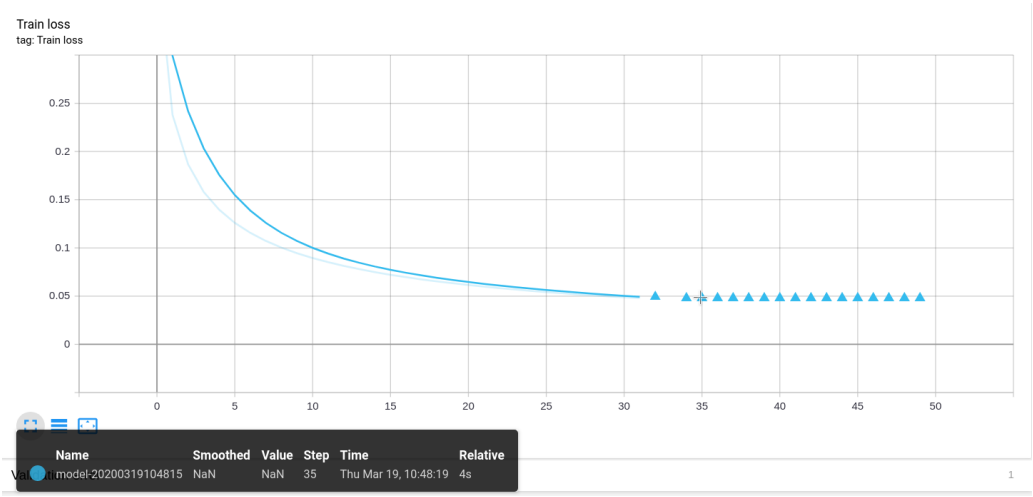


Figure 6.2: The logistic loss during training, with problems after 30 epochs.

This is a problem with the numerical stability of our implementation of the logistic loss. To solve this, we will use Tensorflow's optimized logistic loss function.

6.2 Tensorflow loss functions

The instability we saw with our loss function comes from combining the logarithm with the sigmoid function. This is because the sigmoid tends towards zero as the argument approaches minus infinity

while the logarithm tends towards minus infinity as the argument tends towards 0. It is easy to see how this is a disaster waiting for some rounding error to happen. But, as we can see in Figure 6.3, the logarithm of the sigmoid function tends to become linear in the problematic range of large negative x values.

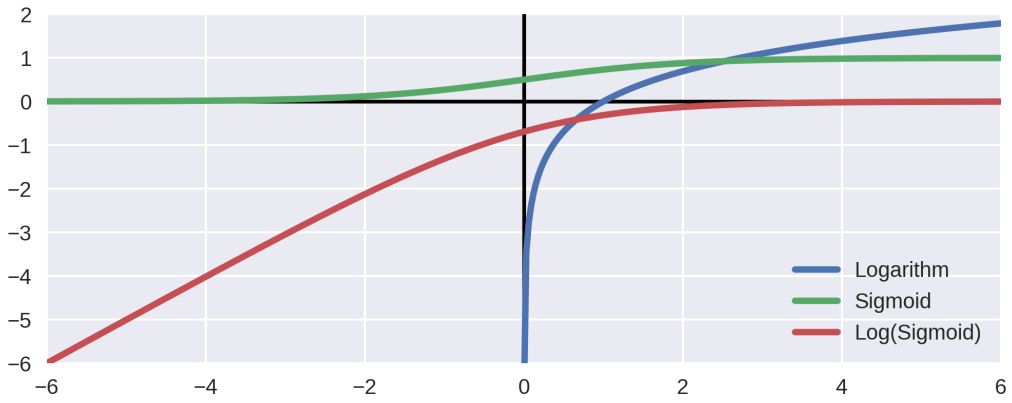


Figure 6.3: Functions log(x), sigmoid(x) and log(sigmoid(x)).

One safe and accurate implementation of the logistic loss function, could be, for example [49]:

$$\log(s(x)) = \begin{cases} x & x < -33.3 \\ x - \exp(x) & -33.3 \leq x \leq -18 \\ -\text{log1p}(\exp(-x)) & -18 \leq x \leq 37 \\ -\exp(-x) & 37 \leq x \end{cases}.$$

where $log1p$ is the logarithm of one plus the argument.

Tensorflow uses the expression `max(x, 0) - x * y + log(1 + exp(-abs(x)))` to compute the logistic loss function where x is the argument for the sigmoid activation (the *logit*) and y the target values. This can be done with the `tf.nn.sigmoid_cross_entropy_with_logits` function. We just need to remember to feed the *logits* as argument, which is the activation value before applying the sigmoid.

We just need to adapt our code to distinguish between the loss function calculation, which uses the output without the sigmoid, and the prediction, which uses the sigmoid activation:

```
16 def prediction(X):
17     net = tf.add(tf.matmul(X, weights), bias, name="net")
18     return tf.reshape(tf.nn.sigmoid(net, name="output"),[-1])
19
20 def logits(X):
21     net = tf.add(tf.matmul(X, weights), bias, name="net")
22     return tf.reshape(net,[-1])
23
24 def logistic_loss(X,Y):
25     net = logits(X)
26     cost = tf.reduce_mean(tf.nn.sigmoid_cross_entropy_with_logits(Y,net))
27     return cost
28
29 def grad(X, Y):
30     with tf.GradientTape() as tape:
31         loss_val = logistic_loss(X,Y)
```

```
32      return tape.gradient(loss_val, [weights, bias]),[weights,bias]
```

We must also adapt our training loop:

```
63 for epoch in range(epochs):
64     shuffled = np.arange(len(Ys))
65     np.random.shuffle(shuffled)
66     for batch_num in range(batches_per_epoch):
67         start = batch_num*batch_size
68         batch_xs = tf.constant(Xs[shuffled[start:start+batch_size],:].astype(np.float32))
69         batch_ys = tf.constant(Ys[shuffled[start:start+batch_size]].astype(np.float32))
70         gradients,variables = grad(batch_xs, batch_ys)
71         optimizer.apply_gradients(zip(gradients, variables))
72     loss = logistic_loss(tf.constant(Xs.astype(np.float32)),tf.constant(Ys.astype(np.float32)
73     print(f"Epoch {epoch}, loss {loss}")
74     with writer.as_default():
75         tf.summary.scalar('Train loss', loss, step=epoch)
```

And now things work fine, and we can run for more epochs, as shown in Figure 6.4:

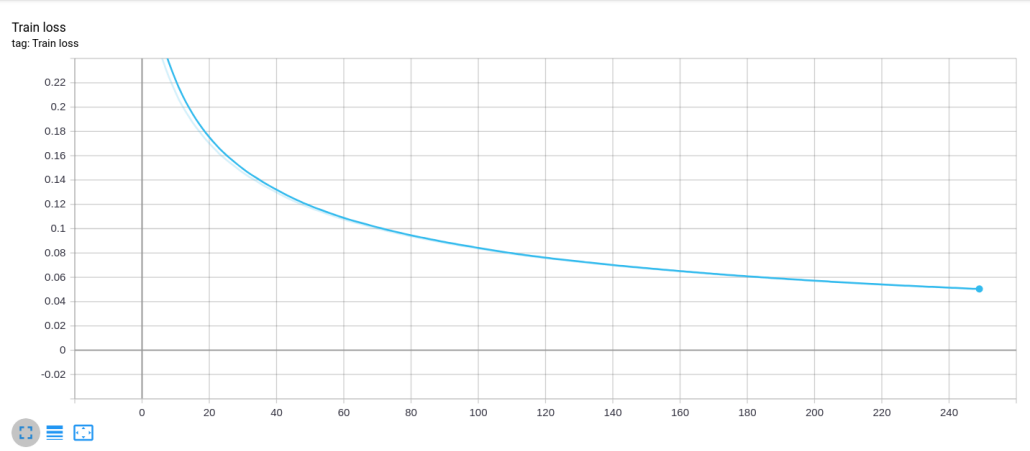


Figure 6.4: The logistic loss during training, now fixed.

The `tensorflow.losses` module has the most used loss functions implemented:

`tensorflow.losses.mean_squared_error` The mean squared error loss function, normally used in regressions with a linear output. Use a linear activation function for prediction.

`tensorflow.losses.sigmoid_cross_entropy` The cross entropy function for a sigmoid output. Note that the input to the loss function must be the *logit*, which is the linear weighted sum of the neuron's inputs. This is because the `sigmoid_cross_entropy` function uses the `tensorflow.nn.sigmoid_cross_entropy_with_logits` function which includes the sigmoid as part of the optimization for gradient calculations. So the input must be the same linear function that serves as input to the logistic activation. For the prediction, input the logit into the `sigmoid` activation function.

`tensorflow.losses.softmax_cross_entropy` The cross entropy function for a softmax output. Note that the input to this loss function must also be the *logits*, which is the linear outputs of the neurons in the last layer. This is because the `softmax_cross_entropy` function uses

the `tensorflow.nn.softmax_cross_entropy_with_logits_v2` function, with optimizations gradient calculations, so the input must be the same linear function that serves as input to the softmax activation. For the prediction, input the logits into the `softmax` activation function.

6.3 Regression: predicting miles per galon with the Auto MPG Data Set (Tutorial)

In this example, we will create a regression multilayer perceptron to predict the fuel consumption of cars using a modified version the Auto MPG Data Set, available at the UCI repository [15]. You'll find the data set in the class materials, and it was modified by removing a text column and some rows with missing values. You can read the file provided using `loadtxt` from the Numpy library but skipping the first row (the headers row).

First, we must preprocess the data adequately. We will shuffle the data before splitting into training and validation sets and use the validation set to monitor for overfitting.

The first column, MPG (miles per gallon), is what we want to predict, so separate this as an array of target values. But first standardize the data[2]. In classification problems we only standardize the features, since the class labels are discrete. But with regression it better to standardize everything to prevent very large values resulting in convergence problems, and also to centre the output of the model on 0. Remember to keep the rescaling parameters not only because they need to be applied to the features of future examples but also because we need to rescale the predictions again to get meaningful results if we apply our trained model.

This dataset has only 392 examples. We will use 300 for training and 92 for validation.

```
data = np.loadtxt('AutoMPG.tsv',skiprows=1)
np.random.shuffle(data)
means = np.mean(data,axis=0)
stds = np.std(data,axis=0)
data = (data-means)/stds

valid_Y = data[300:,0]
valid_X = data[300:,1:]

Y = data[:300,0]
X = data[:300,1:]
```

Since this is a regression problem, we will use a linear output for the last neuron (it should be only one neuron), since we do not want to constrain the output with an activation function, and we will use mean squared error loss function for training. Also, to speed up training, we will use leaky ReLU activations.

Here is a first version of the model. We start by creating auxiliary functions to generate all the weights and biases variables for a multilayer perceptron with the given architecture:

```
def layer(inputs,neurons):
    weights = tf.Variable(tf.random.normal((inputs.shape[1],neurons)) )
    bias = tf.Variable(tf.zeros([neurons]))
    return weights,bias
```

[2]Standardization is subtracting the mean and dividing by the standard deviation, so that all feature distributions have mean of 0 and standard deviation of 1

```
def create_network(X,layers):
    network = []
    variables = []
    previous = X
    for ix, neurons in enumerate(layers):
        weights,bias = layer(previous,neurons)
        network.append( (weights,bias) )
        variables.extend( (weights,bias) )
        previous = weights
    return network, variables
```

This way we can create a network just by specifying how many neurons we want in each layer. For example:

```
layers = [25,15,1]
network,variables = create_network(X,layers)
```

To complete our model, we need to define how to compute the activations. And we can also look at our model in Tensorboard. For this we need the following code:

```
def predict(X):
    net = X
    layer = 1
    for weights,bias  in network[:-1]:
        net = tf.add(tf.matmul(net, weights), bias)
        net = tf.nn.leaky_relu(net)
        layer += 1
    weights,bias = network[-1]
    net = tf.add(tf.matmul(net, weights), bias)
    return tf.reshape(net,[-1])

@tf.function
def create_graph(X):
    predict(X)

def write_graph(X):
    tf.summary.trace_on(graph=True)
    create_graph(tf.constant(X.astype(np.float32)))
    with writer.as_default():
        tf.summary.trace_export(name="trace",step=0)
```

Note that our `predict` function applies the leaky ReLU activations to the hidden layers but leaves the output just with the linear activation. This is because we are solving a regression problem and do not want to constrain the output of the network with an activation function.

Now we write the graph as before. Note that in this case we are only writing the graph for our model. We are not including the loss function and gradients. The left panel of Figure 6.5 shows how the graph looks like. This is not ideal, because all we have are the different operations. We can improve the graph by using the `tf.name_scope` function to name groups of operations and the `name` attribute of the operations to make the graph clearer. Note that these names are not used in eager execution mode but the can be useful in graph mode.

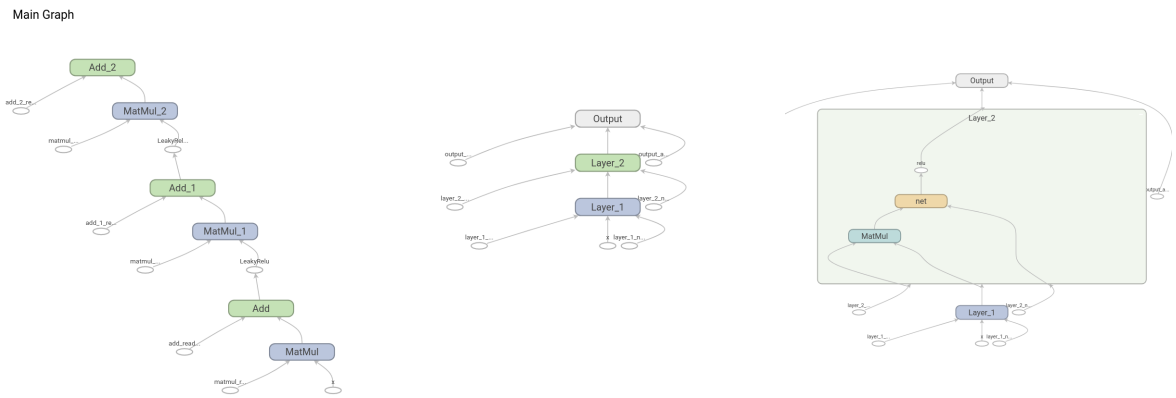


Figure 6.5: Two graphs for the MPG problem. The mid and right panels show the result using named scopes and operations.

Here is the change to the `predict` function we need:

```
def predict(X):
    net = X
    layer = 1
    for weights,bias  in network[:-1]:
        with tf.name_scope(f'Layer_{layer}'):
            net = tf.add(tf.matmul(net, weights), bias,name='net')
            net = tf.nn.leaky_relu(net, name="relu")
        layer += 1
    weights,bias = network[-1]
    with tf.name_scope('Output'):
        net = tf.add(tf.matmul(net, weights), bias)
    return tf.reshape(net,[-1])
```

Now the graph looks better and we can zoom inside each layer to check the internal operations.

Now we train our model. The training loop is left as an exercise. I recommend using the SGD optimizer with a learning rate of 0.005 and momentum of 0.9 and a batch size of 32. Also, note that you need a mean squared error loss function.

However, you will probably find that training does not work because of the NaN problem again.

This time the culprit is the weight initialization. We will cover this problem in more detail in the next chapter, but for now we can implement a quick fix. Unlike the sigmoid activation, ReLU can output very large values. With many neurons, using a normal distribution with a standard deviation of 1 (the default) results in large outputs since the inputs add up. One quick solution is to reduce the range of values with which we initialize the weights by dividing by the number of neurons in the layer. There are more sophisticated initialization methods but we will cover those later.

```
1 def layer(inputs,neurons):
2    weights = tf.Variable(tf.random.normal((inputs.shape[1],neurons), stddev = 1/neurons ))
3    bias = tf.Variable(tf.zeros([neurons]))
4    return weights,bias
```

Now things work and we can look at the training and validation errors on Tensorboard. To plot these, it may be useful to convert from the mean squared error to an error in miles per galon. You can do this by taking the square root of the MSE and then multiplying by the scaling factor used when standardizing the data (the original standard deviation):

```
...
    train_error = loss(tf.constant(X.astype(np.float32)),
                       tf.constant(Y.astype(np.float32)))**0.5*stds[0]
    valid_error = loss(tf.constant(valid_X.astype(np.float32)),
                       tf.constant(valid_Y.astype(np.float32)))**0.5*stds[0]
    with writer.as_default():
        tf.summary.scalar('Train error', train_error, step=epoch)
        tf.summary.scalar('Validation error', valid_error, step=epoch)
...
```

Looking at the training error profile in Figure 6.7 we can see many fluctuations. This suggests a problem with the training parameters or optimizer. We will see more of these issues in the next chapter.

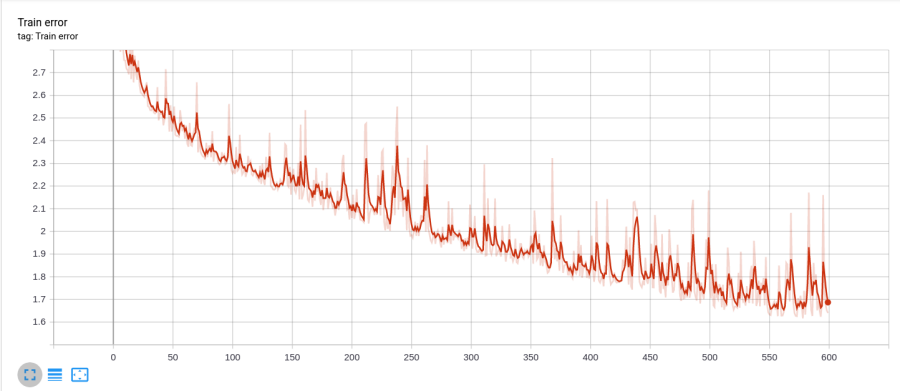


Figure 6.6: Training error for the MPG problem.

Another problem is with the validation. It's easy to see that the validation error went down at first but then started to climb. Our network seems to be overfitting the data as we train it.

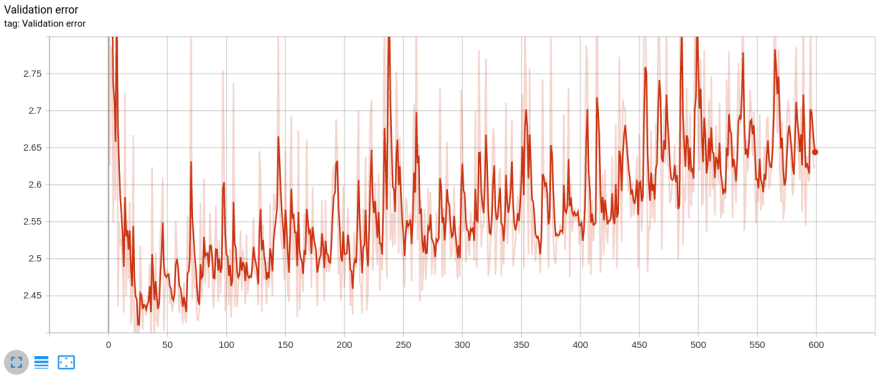


Figure 6.7: Training error for the MPG problem.

As an exercise, experiment with different architectures for this problem, as well as different learning rates, to see if you can improve these results.

6.4 Exercises

Follow the two previous sections and the lecture as tutorials for binary classification and regression. Try to do as much as you can on your own, but look up the code as needed.

Once you understand how to use Tensorflow and Tensorboard for this type of problems, create a classifier for the banknotes (the data file is `banknontes.csv`). Remember to standardize the features and to use one of the Tensorflow sigmoid logistic loss functions since implementing your own may result in numerical instability. Try different network architectures and look at training and validation errors to check for overfitting (reserve about 20% of the data for validation). Note that the class label is the last column on the csv file: four feature columns and then one label column.

Finally, implement a multi-class classification network for the Modified NIST (MNIST) dataset [40]. To obtain the data, use the following code:

```
import tensorflow as tf
import numpy as np
from tensorflow import keras

mnist  = keras.datasets.mnist
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()
```

This will download the data to the folder `/.keras/datasets/mnist.npz`, where is your home folder in Linux or user folder in Windows, if the dataset is not there yet. Once it is donwloaded, it will no longer download it in future executions.

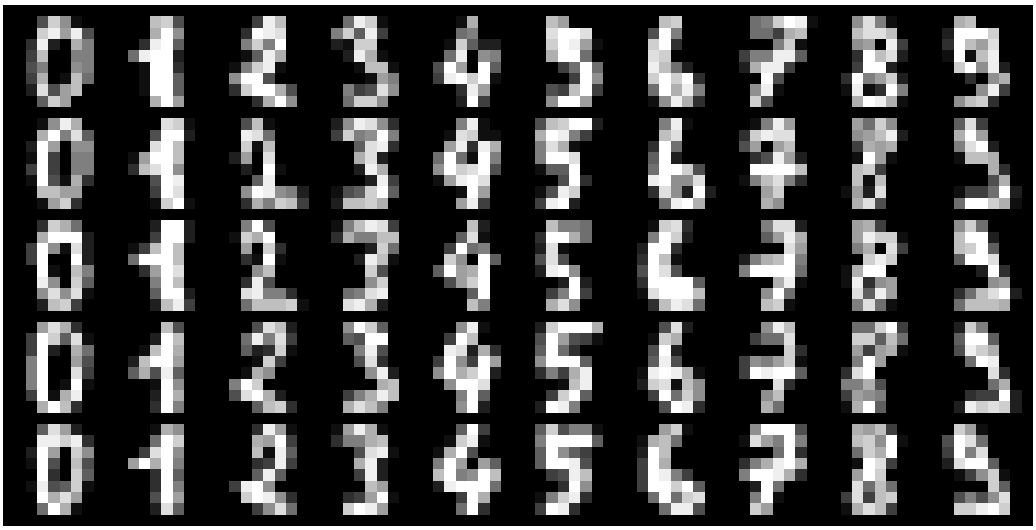


Figure 6.8: Example of images from the MNIST dataset.

The MNIST dataset is a set of 70000 images, divided into a test set of 10000 and a training set of 60000 images. Each image is a grayscale matrix of 28×28 pixels, as shown in Figure 6.8. The labels are arrays of integer values from 0 to 9 indicating the digit depicted in the corresponding image. To use the data in our network, we must reshape the image arrays to arrays of $N \times 784$, with N being the number of images, instead of $N \times 28 \times 28$, so that each example has the 784 (28×28) features in a single vector. You can do this easily in Python in with the `reshape` method of the arrays. We will keep the test images for our final estimate of the true error, so we will split the 60000 training images into a training set of 50000 and a validation set of 10000. You can assume the images are in a random order, so you just need to take the last 10000 of the images and labels.

You also need to use one-hot encoding for your class labels in order to compare with the softmax activation. One-hot encoding is a way to encode categorical data using a vector with as many elements as the number of categories filled with a value of 1 for the corresponding category and 0 for the

remaining categories. This one-hot encoding of the ten categories in the MNIST data set is what you want to fit to the ten output neurons of your network, using softmax activation.

Here is the code to reshape the images and split the training set, as well as for creating the one-hot encoding using the `to_categorical` utility function

```
valid_images = train_images[50000:].reshape(-1,28*28)
valid_labels = tf.keras.utils.to_categorical(train_labels[50000:])
train_images = train_images[:50000].reshape(-1,28*28)
train_labels = tf.keras.utils.to_categorical(train_labels[:50000])
```

Note that in this case your labels will be a two-dimensional matrix, with one row per example in the mini-batch and one column for each of the 10 categories. So do not reshape your predictions to arrays.

Now create a multilayer perceptron. I suggest starting from the `layer` function presented in the previous sections. Remember that both the `tensorflow.nn.softmax` and the `tensorflow.nn.softmax_cross_entropy_with_logits` functions expect the linear part of the neurons (the logit) without the non-linear activation function. Thus, your last layer should be composed of 10 neurons, one for each class, but without the activation function when computing the loss function.

Also, remember that your loss function should be computed with the `tensorflow.nn.softmax_cross_entropy_with_logits` function because this one is optimised for numerical stability during training. For predicting the probabilities of each class, use the `tensorflow.nn.softmax` operation. Finally, you can obtain the class corresponding to each activation using the `np.argmax` to find which neuron has the highest output value.

For this exercise, compare different activation functions in the hidden layers (sigmoid, ReLU, Leaky ReLU, for example), different network dimensions and different learning rates and momentum. Once you are satisfied with the results, train the final network using the complete training set and evaluate its performance on the test set. The MNIST page [3] has a table with some results using different networks.

6.5 Questions

- Why use ReLU activations in the hidden layers instead of sigmoid activations?
- When should we use sigmoid activations in the output layers, and why?
- What is the softmax activation and when should we use it?
- What activation function should we use in the output layer of a network used to solve a regression problem?
- Why is it best to use a function like `tensorflow.nn.sigmoid_cross_entropy_with_logits` instead of computing the cross entropy using the sigmoid activation ourselves?

6.6 Further Reading

1. Goodfellow et. al., Chapter 6 (sections 6.1 and 6.2)[21]
2. (Optional: also chapter 8 until end of section 8.3)

[3] http://yann.lecun.com/exdb/mnist/

Chapter 7

Optimizing Networks

Optimizers. Learning rates. Weight initialization. Batch normalization. Model selection, overfitting, bias and variance. Regularization. Choosing hyperparameters

7.1 Optimization

Numerical optimization algorithms fall outside the scope of this course. However, it is useful to have an idea of how effective the different options are and how they work. The basic gradient descent algorithm, implemented on Tensorflow in `tf.keras.optimizers.GradientDescentOptimizer`, simply computes the gradient of the loss function and then updates the parameters in that direction with a step proportional to the gradient and a single learning rate. Stochastic gradient descent uses the gradient computed at each example, selected at random, and mini-batch gradient descent applies the update after computing the total gradient from a batch of randomly selected examples. This randomness helps avoid local minima and improves the chances of converging to better solutions.

One problem with basic gradient descent is that the learning rate is constant, preventing the optimizer from adapting to different conditions along the minimization. This is especially problematic with the gradient varies differently along different dimensions, which may cause the gradient descent algorithm to oscillate and fail to find the best path towards minimization. The use of momentum helps solve this problem by also accumulating previous gradient directions. This causes oscillations to cancel out and the correct directions to become reinforced, speeding up optimization as we saw previously.

However, this still has the problem of using the same learning rate for all parameters, and different parameters may benefit from different update rates. The AdaGrad algorithm [16] (`tf.keras.optimizers.Adagrad`) divides the learning rate of each parameter by the sum of past (squared) gradient values for that parameter. This way the algorithm increases the learning rate for parameters with smaller gradients and reduces the learning rate for those with large gradients, speeding up learning with less risk of stepping too far.

RMSprop is similar, but keeps a moving average of the squared gradients and divides the gradient by the square root of the mean squares (hence the "RMS"). This evens out the size of the updates for all parameters. RMSprop is famous by being widely used and yet never officially published, having been proposed by Geoff Hinton in a machine learning course. RMSprop is implemented in Tensorflow in the class `tf.keras.optimizers.RMSprop`.

The Adaptive Moment Estimation (Adam) [34] algorithm combines momentum and different

Figure 7.1: Examples of a model trained with high and lower training rates.

learning rates for different parameters using an exponentially decaying average over the previous gradients. Adam seems to perform better than previous optimization algorithms on average, although it can have some convergence problems[53]. In Tensorflow, Adam is implemented in the (`tf.keras.optimizers.Adam`). Adam can be a good first choice to experiment with your models but pay attention to the error values because it may not converge well.

Choosing the best optimizer for a particular problem may require some experimentation.

7.2 Learning Rate

The choice of optimizer and learning rate can have a significant impact not only on training times but also on the performance of your trained network. A low training rate makes training take longer, because the optimization steps are smaller. But if the training rate is too high, the optimizer may fail to converge to the best solution, or even a good solution. The actual values to use depend on the problem and the optimizer, but you can check for this problem also by looking at the error profiles. Figure 7.1 shows the same model trained with a higher and lower training rate. The panel on the left shows that the optimizer is having difficulty converging to a solution, jumping too much over different values. When this happens in the training error it is often a bad sign. Although the training error drops faster in the beginning, with a higher leaning rate, the end result may not be good.

7.3 Weight Initialization

Initializing the network weights properly can be important but, unfortunately, there is no good understanding of precisely what the best way to do this may be. But it is clear that weights cannot be the same for all neurons, otherwise the gradient will be the same for all parameters and all neurons will be optimized in the same way. This symmetry must be broken from the start. It is not a problem to start with bias values at zero, but the weights must be randomized to guarantee that different neurons start at different combinations of parameters.

There are other considerations, depending on the networks. Recurrent networks are more susceptible to instability if the weights start too large and, in any case, large initial weights may saturate activations or cause other numerical problems. On the other hand, larger weights are better at breaking neuron symmetry and “spread out” the network more widely from the start.

One standard way of choosing initial weights is to simply draw them at random from a Gaussian distribution with mean zero and variance 1. Other initialization schemes include taking into account the number of neurons in each layer, or the number of inputs, and other factors. See section 8.4 of [21]

for more details, but bear in mind that weight initialization heuristics may not always give the best results when compared to a simple normal distribution.

Keras offers a choice of initializers, including random uniform and normal distributions, truncated normal distributions, as well as more sophisticated initializers. The default initializer for dense layers, such as those used in multilayer perceptrons, is normalized initialization, also known as the Glorot uniform initializer:

$$W_{i,j} \sim U\left(-\sqrt{\frac{6}{fan_{in}+fan_{out}}}, \sqrt{\frac{6}{fan_{in}+fan_{out}}}\right)$$

In this initialization method, the weight distribution is scales in proportion to the number of inputs into the neuron (fan_{in}) and the number of neurons in the next layer receiving the output of this neuron ((fan_{out}).

7.4 Batch Normalization

We always normalize (or standardize) the inputs to a neural network, as it is easier for the network to learn by avoiding extreme values and by having the data centred on the origin. With a deep network, we can imagine that each layer is receiving the activations of the previous layer as inputs. It will thus benefit from normalizing these activations. Furthermore, during training the activations keep changing as the weights are updated. This causes the inputs to each layer to shift in mean and variance constantly, forcing each layer readjust to these shifts caused by changes in the previous layers. These processes can hinder training.

Batch normalization[30] solves these problems by standardizing the activations at each layer using running statistics for mean and variance collected during training. This not only makes the input more "well behaved" for the next layer but also eliminates the constant changes in mean and variance of the activations during training. Since backpropagation can work through these rescaling operations, these can be inserted into the network as if it was an additional layer, and that is precisely how we can do it with Keras.

7.5 Overfitting

Now that we now which loss function to choose for our problem and how to minimize it, we need to consider the danger of fitting our data so much that our network performs poorly when applied to new examples. This is called overfitting, and basically consists in having an error rate outside the training set that is much larger than the error rate inside the training set. We should select a network architecture (and other aspects of training, such as regularization, which we will see later) that minimizes overfitting.

In order to monitor overfitting, we must measure our network's performance outside the training set. To do this there are two possibilities. One is to split our data set in training and validation sets and use only the former for training the network. This leaves us with one set of data to monitor how the network is classifying examples outside the training set.

A more computationally demanding approach, but giving better results, is to use cross-validation. To do cross-validation, we partition our training data into k disjoint *folds*. For example, if we have 50.000 examples and want to use 5-fold cross-validation, we place 10.000 examples into each fold. Then we train our model with all folds but one and validate on the fold that was left out. Then we repeat

training leaving another fold out, and do this k times, averaging the validation error. This gives us a better estimate of the true error that, on average, instances of our model will have when trained on data such as this of data. Figure 7.2 shows an example of 5-fold cross validation using the gene expression data with a linear classifier. Each panel shows an hypothesis obtained by fitting the model to four of the folds (indicated by the smaller points) and then validating using the fold left out.

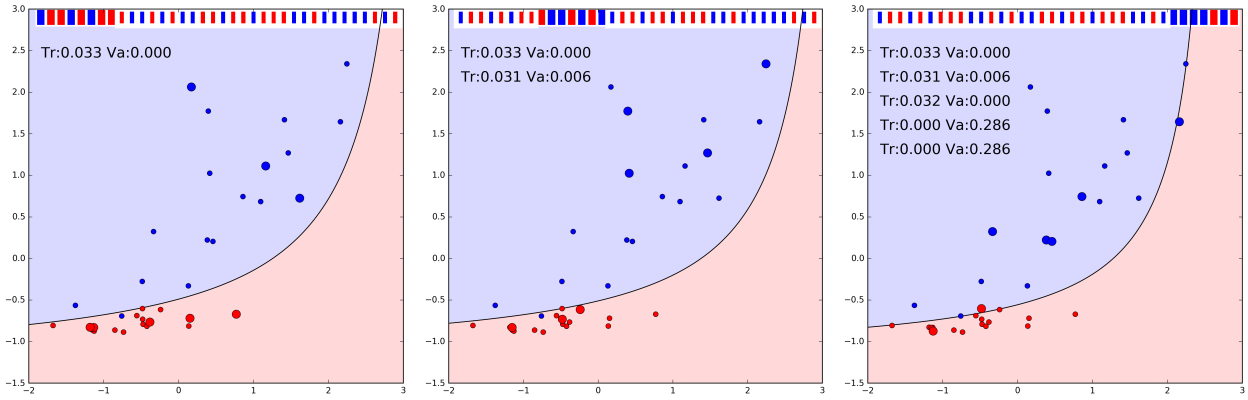


Figure 7.2: Example of 5-fold cross validation, showing the plots for folds 1, 2 and 5. In each panel, one of the folds is left out for validation, the other folds are used for training. The larger points are those used for validation in each fold. The training and validation errors are kept for each fold and then averaged in the end.

Cross validation gives us a better estimate of the true error (the average error over all possible data) than simple training and validation because we are averaging k estimates of the true error. However, this requires training the network k times. This is why cross validation, although widely used in classic machine learning, is often discarded in favour of simple training and validation in deep learning, where models take much longer to train. In any case, it is important to evaluate how our networks perform outside the training data, otherwise we are likely to overfit the data and get a much poorer performance than the training results suggest.

Finally, it may be useful to have an unbiased estimate of the true error of our final network, so we can judge how it will perform. This should not be done with validation or cross validation errors if we have been fine-tuning the network and other hyperparameters, as is generally the case. The reason for this is that, while selecting hyperparameters, we choose those that minimize the validation (or cross validation) error, causing the error value obtained in the end to be biased towards underestimating the true error. The solution for this problem is to leave out a subset of our initial data, the test set, that was never used to choose any model or parameter and can thus serve to obtain an unbiased estimate of the true error of the final network.

In machine learning, we want to use the data available to train a model so that the hypothesis (the trained instance) we obtain can be applied to new data. This is especially true for supervised learning, although sometimes in unsupervised learning we just want to make sense of the data that we have. In these cases, we must consider the performance of our networks outside the training data, as we saw previously.

7.6 Model Selection

If our model is able to learn irrelevant details about the training set, that do not generalize, then the resulting hypothesis may perform poorly on new data. This is called *overfitting* and we can check for

Figure 7.3: Examples of a model that is overfitting (left) and one that is not (right).

this effect using cross-validation or with a validation set, as we saw in the previous chapter. This way we can select the best model to use for our problem. Figure 7.3 illustrates how we can easily notice this problem when plotting the training and validation errors. Even though the model on the left seems better at adjusting to the training set, it fares poorly on the validation set, and thus it is best to use the one on the right.

7.7 Bias and Variance

When considering how to best solve a machine learning problem it is good to separate two sources of error, at least conceptually, to better understand how we can optimize the results. Statistically, the *bias* is the difference between the expected value of an estimator and the true value being estimated. Thus, the *bias* of a model at some point is the difference between the true value we are trying to predict and what, on average, the model predicts at that point after training with some set of data. Note that this average is not only for one specific training set but for all possible training sets. If we had many training sets drawn from the same universe of possible data, the *bias* for the model in one point could be estimated as the quadratic error between the true value and the average of what our model predicts for that value. Averaging this for all possible values would give us the expected bias of the model:

$$bias_n = (\bar{y}(x_n) - t_n)^2 \qquad bias = \frac{1}{N}\sum_{n=1}^{N}(\bar{y}(x_n) - t_n)^2$$

Figure 7.4 shows an example of a model that cannot adequately fit the data. The estimates for the point marked as a large blue circle are all tendentiously above the true value and thus there is a difference between the average and the true value.

Variance is a measure of the dispersion of values. Applying this concept to a regression model, the *variance* of the model at some point is the expected variance of the predicted values for that point when the model is trained over any data set. The *variance* for the model is the average of the variances for all points. To estimate the variance of a point and on N points of a model trained on M data sets, we compute:

$$\frac{1}{M}\sum(\bar{y}(x_n) - y_m(x_n)) \qquad var = \frac{1}{NM}\sum_{n=1}^{N}\sum_{m=1}^{M}(\bar{y}(x_n) - y_m(x_n))^2$$

where $\bar{y}(x_n)$ is the average of the predictions for point x_n. Figure 7.5 shows a model that overfits the data, which results in a large *variance*, showing that, for the point marked as a large circle, the predictions of individual hypotheses are spread in a broad range around their average.

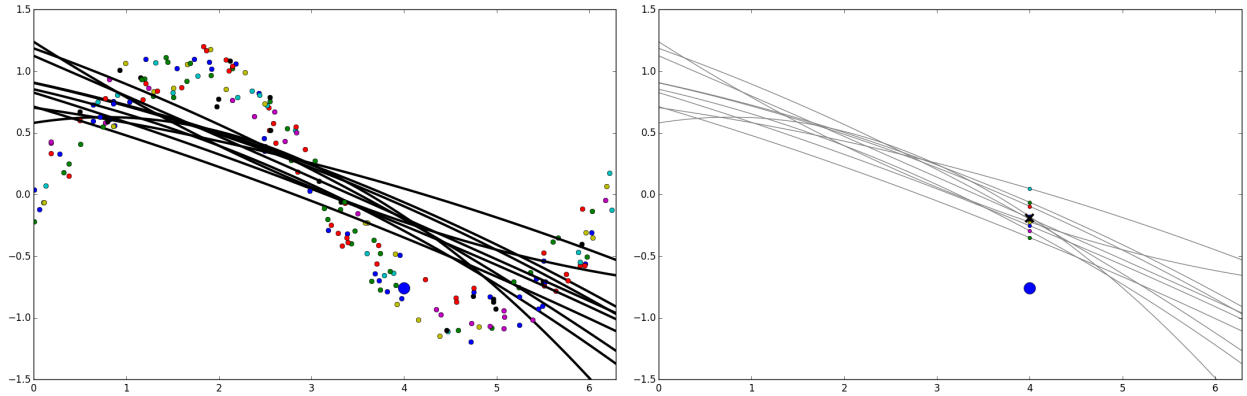


Figure 7.4: This model cannot adjust to the data and thus has a large *bias* in some points.

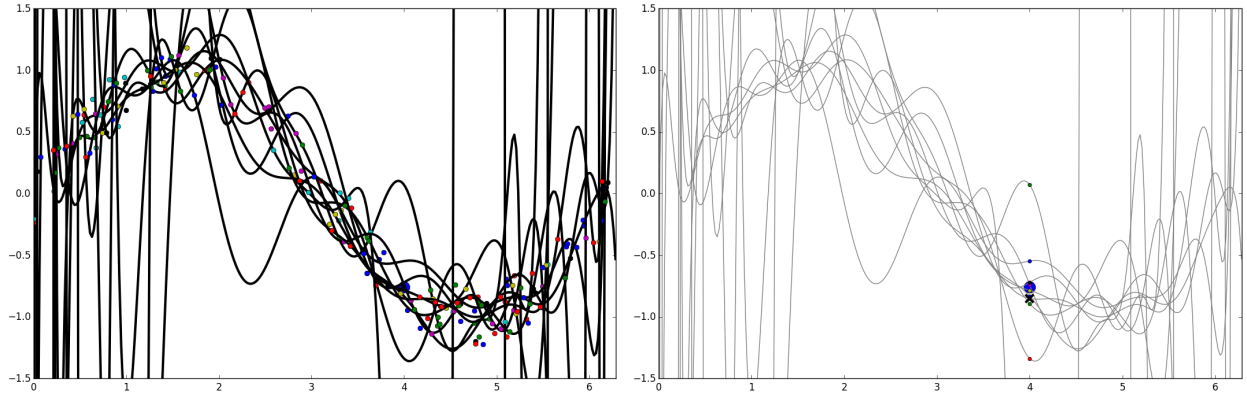


Figure 7.5: This model overfits the data and thus has a large *variance* in some points.

These are the two fundamental sources of error that we have control over. There is also an additional possible source of error, which is that associated with our data, but there is nothing we can do about that. With respect to our models, we can consider that the error we get outside the training set (the true error, over all possible data) will be due to a combination of the error due to the model not being able to perfectly adapt to the shape of the data, which is the *bias*, and the model varying its predictions too much as a function of details in the training set, which is the *variance*.

This distinction is important because bias can only be mitigated if we use a better model or if we combine simpler models so that we get a greater power to adjust to the data. Variance, however, can be mitigated either by opting for simpler models, which will generally increase bias, or by changing the way the model is trained in order to prevent it from varying so much in its predictions. Thus, it is often the case that we deliberately choose a model in overfitting and then mitigate that problem by reducing its variance with regularization.

7.8 Regularization

Regularization is any way of changing the training of the model in order to decrease its variance. This may increase bias, but if the model is overfitting and the bias increase is modest, regularization can greatly improve results. There are several ways of regularizing neural networks.

Parameter Norm Penalties

Penalizing the norm of the parameters vector is a traditional regularization method in statistics and machine learning. In brief, we can imagine that the set of parameters in our model represents a vector and we want our training algorithm to try not only to reduce the loss function but also the some measure of the "length" of this vector. Thus we change our loss function to also include a penalty term as a function of the parameters themselves:

$$\tilde{J}(\theta; X, y) = J(\theta; X, y) + \alpha\Omega(\theta)$$

where X, y is our data, θ the parameters and $J(\theta; X, y)$ our original loss function. α is a regularization parameter that we can adjust to give greater or lesser weight to the regularization.

With L^2 regularization, also known as *ridge regression* or *Tikhonov regularization*, the penalty function is the square of the modulus of the parameter vector, $||\theta||^2$. This forces the overall magnitude of the parameters to go closer to zero. In neural networks, this can also be implemented in the parameter update rule as *weight decay*: whenever a weight on a neuron is updated, a fraction of its previous weight is subtracted:

$$w_{i+1} = w_i - \eta\frac{dJ}{dw} - 2\alpha w_i$$

where w is some weight, η the learning rate, J the loss function and α the regularization weight. Note the factor of 2 multiplying α, which results from computing the derivative of the quadratic penalty function applied to the weight.

Another used norm penalty function is L^1 regularization, which penalizes the sum of the absolute values of the parameters, $\sum_i |\theta_i|$. While L^2 regularization drives the whole vector θ to be smaller, L^1 regularization makes some weight fall to zero, resulting in a sparser solution, where weights are completely "turned off"

Although in general, L^2 and L^1 regularization apply the penalty to all parameters in the model, in neural networks these penalties are usually applied only to the neuron weights and not the bias values. This is because the bias values, unlike the weights, do not contribute to the sensitivity of the model to slight variations in the data.

Dataset Augmentation, Noise and Semi-supervised Learning

We can always reduce overfitting by using more data for training. Although data is generally not easy to come by, sometimes we can augment our data set with some simple operations. For example, with images we can rotate our mirror images in our dataset to create variants that are different but retain the same labels, if appropriate. But some care must be taken, depending on the data itself, for while a mirror image of a cat is still a cat, the mirror image of `b` is `d`.

Another regularization method involves injecting random noise in the features whenever an example is presented to the neural network. This can be considered a form of data augmentation, providing the network with a greater diversity of examples which can be useful as long as the noise injected is not excessive.

Noise can also be applied to the weights of the network, which is equivalent to penalizing the gradient of the error and leading the optimizer to find sets of weights in more stable regions of the loss function.

In classification problems, *label smoothing* may be useful. This involves making the target labels, such as in a one-hot encoding or binary classification, not hard values of 0 and 1 probability but softer values close to zero and one. For example, $\frac{\epsilon}{k-1}$ instead of zero and $1-\epsilon$ instead of one, with a small ϵ. This prevents the optimizer from trying to drive the sigmoid or softmax functions to exactly zero and one, which is impossible, and is equivalent to adding some noise to the class labels, with a small probability of an example being presented as belonging to the wrong class.

Semi-supervised learning uses unlabelled data to enrich the data set used for supervised learning. One way of doing this is to use an unsupervised learning algorithm to cluster all data, both labelled and unlabelled, and assume that unlabelled examples close to labelled examples belong to the same class. Other approaches involve combining labelled and unlabelled examples to learn the probability distributions of features, which can then inform the training of the classifiers on the labelled data.

Early Stopping

Another way of regularizing neural networks is to stop training to prevent overfitting from getting worse. As the optimizer moves the parameter vector away from the initial values (close to 0) it can improve the fit in the training set while degrading performance outside this set. Early stopping keeps the optimization closer to the initial values, thus restricting this walk into more extreme values that worsen overfitting. Figure 7.6 illustrates this. Even though the training error continues to fall, the validation error starts to rise after about 15 thousand epochs, showing that overfitting is becoming worse and the training should be stopped at that point.

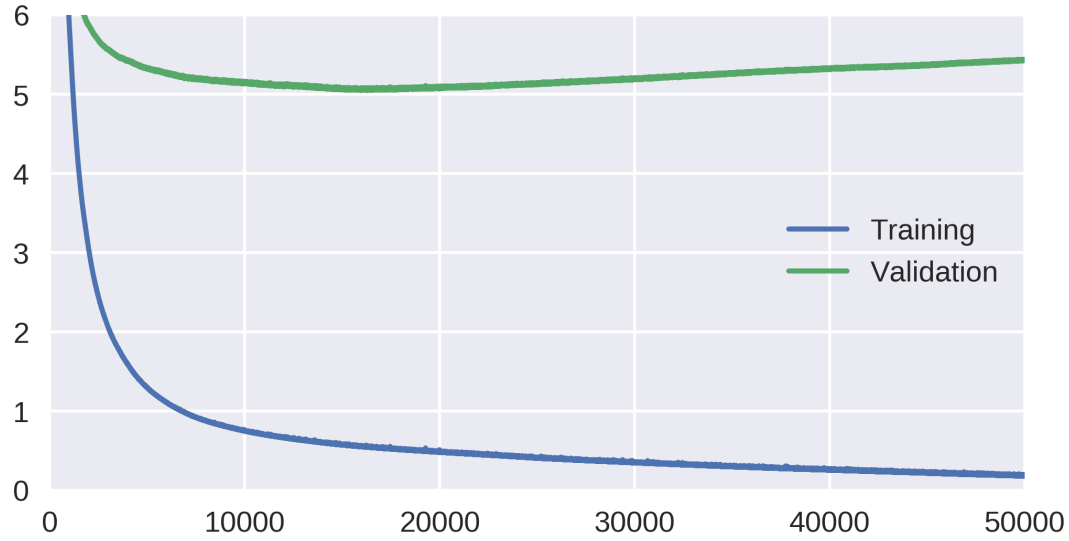


Figure 7.6: This model overfits the data and the problem gets worse after about 15 thousand epochs.

Ensembles and Dropout

One commonly used method for reducing variance in machine learning is to group together several instances of the model, or even several models, trained in different subsets of the training data. *Bagging* is an example of this, using randomly drawn subsets of the training data to train the model into different hypotheses and then classifying new data with the average of all hipotheses.

However, this is not practical for deep neural networks because of how long it takes to train them or to use an ensemble of networks to classify new examples. *Dropout*[70] is a better way of doing this

with neural networks. During training, at each example or mini-batch, neurons are "dropped" randomly by ignoring their weights and activation. Typically, the probability of doing this for any hidden neuron at any iteration is 0.5, or 0.2 for input neurons (output neurons are not dropped). Thus, we are training an ensemble of different networks given by the combination of neurons that are "alive" at each iteration. Figure 7.7 shows two iterations of the same network with different neurons dropped off, at random.

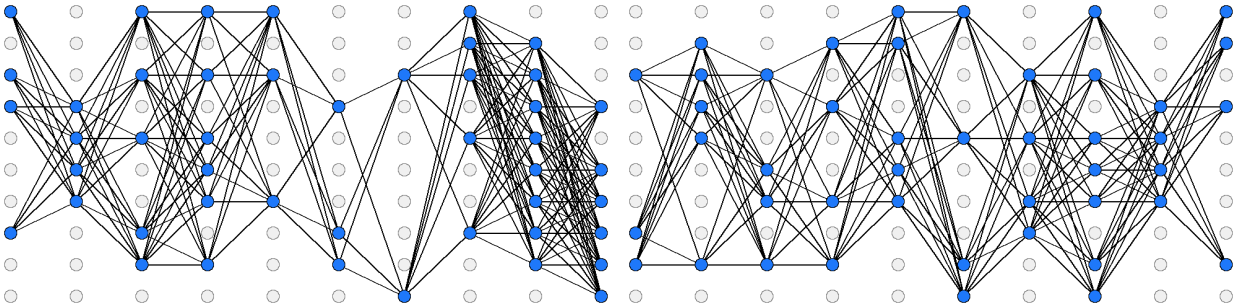


Figure 7.7: Dropout: neurons are ignored at random at different iterations (examples or mini-batches.

After training, the activations of neurons (or weights in the forward neurons) can be rescaled by the dropout probability to retain the same expected activation. Alternatively, as is implemented in Tensorflow, we can use *inverse dropout* during training. In this case, neurons that are not dropped will have their activations multiplied by $1/p$ so that expected activations will then be the same on test.

7.9 Selecting Hyperparameters

With so many options to choose from, in network architecture, optimizers, regularization and so forth, selecting the right combination of hyperparameters can be difficult. One option is to try different combinations manually, try to understand what effects they have and gradually narrow down a good combination.

Alternatively, it is possible to automate the search for the best hyperparameters. A *grid search* systematically runs through combinations of parameters within a given range and with a given step. Given the large number of possibilities, making it unfeasible to systematically run through all combinations, a *random search* may be best. Simply pick hyperparameter values at random and repeat, keeping the best combination. Finally, hyperparameter choice can be seen as an optimization problem. One approach is to use a Bayesian model to estimate the validation error for each combination based on combinations tested so far and use that to decide what new combinations to test to improve results.

7.10 Further Reading

1. Goodfellow et. al., Chapters 7 and 11, and sections 8.4, 8.5 and 8.7.1 [21]

Chapter 8

Convolutional Networks

Convolution. Convolution layers and networks. Pooling. Classification with convolutional networks. The Keras sequential model.

8.1 Convolution

(Note: this section is still work in progress...)

Mathematically, the convolution of two functions f and g at t, represented $(f*g)(t)$, is the integral of the product of the two functions with one being reversed and then shifted by a parameter t. Convolution is commutative:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau = \int_{-\infty}^{\infty} f(t-\tau)g(\tau)d\tau$$

This seems a strange operation but it crops up in many situations. For example, the probability density function of the sum of two continuous random variables is equal to the convolution of the probability density functions of the two variables. More easy to understand is the modelling of linear time invariant systems. These are systems that respond to inputs with some response function such that the response is linear with respect to the inputs and depends only on the timing of the inputs. Let's consider a spring with a dampener, responding to an impulse with a function $g(t)$ in the form of a decaying sine wave, for example. We hit the spring with a function $f(t)$ and want to know how the fork reacts.

8.2 Introduction to the Keras Sequential API in Tensorflow

To build a model using the Keras Sequential API, you just need to create a `Sequential` object an then add the different layers. We will use the Fashion MNIST exercise (below) as an example. First, we import the necessary classes:

```
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import BatchNormalization,Conv2D,MaxPooling2D,
from tensorflow.keras.layers import Activation, Flatten, Dropout, Dense
```

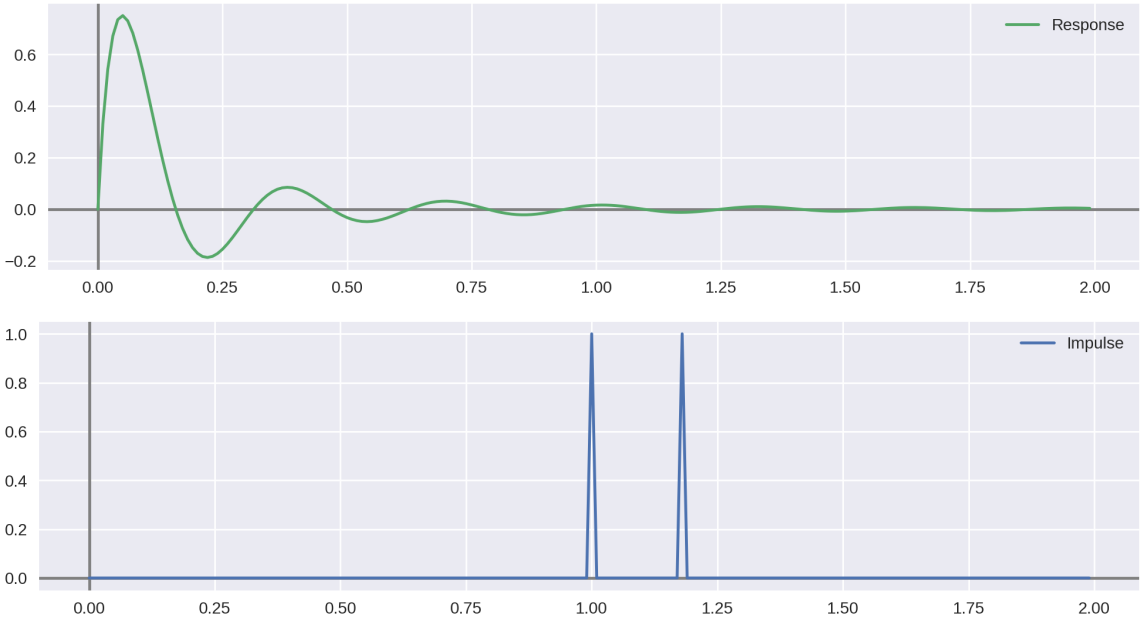


Figure 8.1: Response (top) and impulse (bottom) functions for an hypothetical dampened spring.

Now we create the model. To illustrate, we will add a 2D convolution layer with a 3×3 kernel and 32 filters as the first layer. The input shape corresponds to the image shapes in the Fashion MNIST dataset. Then will add a ReLU activation and a batch normalization layer. The first layer needs the input shape specified as it will receive the data. Note that the batch size is omitted and the shape corresponds to a single example. The shape of the subsequent layers is determined automatically.

```
model = Sequential()
model.add(Conv2D(32, (3, 3), padding="same", input_shape=(28,28,1)))
model.add(Activation("relu"))
```

To add pooling, dropout and dense layers, we just use the appropriate classes. The parameters are mostly self-evident, but for more details please look up the documentation online. Note the `Flatten` layer before the dense layers. This is needed to flatten the input into a single dimension for the features, which is what the dense layer expects.

```
...
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Conv2D(64, (3, 3), padding="same"))
model.add(Flatten())
model.add(Dense(512))
model.add(Activation("relu"))
model.add(BatchNormalization())
model.add(Dropout(0.5))
model.add(Dense(10))
model.add(Activation("softmax"))
```

To compile the model we need to select an optimizer and specify the loss function and metrics to register during the run.

```
opt = SGD(lr=INIT_LR, momentum=0.9, decay=INIT_LR / NUM_EPOCHS)
model = create_model()
```

```
model.compile(loss="categorical_crossentropy", optimizer=opt,
    metrics=["accuracy"])
```

Now we can train the model, calling the `fit` method to which we supply the training features and labels, as well as the validation data if we wish to record validation metrics during training. We can also set the batch size and epochs. The `fit` method returns a `history` object with a dictionary, also called `history`, with the record of all metrics collected during training.

```
history = model.fit(trainX, trainY, validation_data=(testX, testY),
                    batch_size=BS, epochs=NUM_EPOCHS)
```

After training we can save our model as a JSON file and the weights in a hierarchical data format (HDF5) file:

```
model.save_weights('fashion_model.h5')
model_json = model.to_json()
with open("fashion_model.json", "w") as json_file:
    json_file.write(model_json)
```

8.3 Exercise: Exploring Hyperparameters

Use the data for the miles per gallon example from section 6.3 but create your regression model using the Keras API. See https://keras.io/losses/ to find the appropriate loss function for this model. Explore different hyperparameters (network, optimizer, learning rate and so on) to and try improve results on the validation set.

8.4 Exercise: Fashion MNIST with a convolution network.

In this exercise you will create a CNN to classify the Fashion MNIST data set[1].

First, we need to load the data set and format it adequately. The loader will check if the dataset is already available locally and download it if not. Then we will reshape the features to $28 \times 28 \times 1$ to fit our convolution layers and convert the image data from integer to float.

For the class labels we will need one-hot encoding. We can convert them with the `to_categorical` function in `keras.utils`.

```
from tensorflow import keras

((trainX, trainY), (testX, testY)) = keras.datasets.fashion_mnist.load_data()
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))

trainX = trainX.astype("float32") / 255.0
testX = testX.astype("float32") / 255.0

# one-hot encode the training and testing labels
trainY = keras.utils.to_categorical(trainY, 10)
```

[1] Based on a PyImageSearch tutorial by Adrian Rosenbock, https://www.pyimagesearch.com/2019/02/11/fashion-mnist-with-keras-and-deep-learning/

```
12 testY = keras.utils.to_categorical(testY, 10)
```

Now create this model with the sequential API provided by Keras:

- Two convolution layers with a 3×3 kernel, "same" padding, 32 filters, ReLU activation and batch normalization. Note that batch normalization uses the last axis as default direction to normalize. If the channels are at the end, this is correct.
- Max pooling of size 2×2 and the same stride (note that omitting the stride will use the pool size for stride too). Optionally, you can try a dropout layer with 25% dropout probability, but dropout in convolution layers is generally not very useful.
- Two more convolution layers like the ones before, followed by the same pooling, but use 64 filters in the convolution layers.
- Finally, a dense layer of 512 neurons also with ReLU activation, batch normalization and dropout of 50% followed by a softmax layer with 10 neurons.

Use the optimizer shown before. Train the model for about 25 epochs (or fewer, if this is too slow) and then save the weights. We will be using these later on for transfer learning.

8.5 Further Reading

1. Goodfellow et. al., Chapter 9)[21]

Chapter 9

CNN Architectures

Convolution network architectures for classification: AlexNet, VGG, ResNet and Inception. Image segmentation with U-Net

9.1 Image classification with CNN

This section presents some architectures for image classification with convolution networks. CNN are used for other applications but they became most famous for their success in dealing with images.

AlexNet

In 1989, LeCun and others proposed a neuron network with weight sharing for classifying handwritten digits [39]. This weight sharing property is an important feature of convolutional layers, which helps feature detection while reducing the number of free parameters. But a major milestone in the use of CNN for image recognition came with the work of Krizhevsky, Sutskever and Hinton in 2012 [35], with AlexNet, winning that year's ImageNet classification competition.

AlexNet had 60 million trainable parameters, was implemented to run in two graphics cards, due to memory limitations of the available hardware, and consisted of a first layer of 96 kernels with a dimension of 11 by 11 (split into two blocks of 48), then a layer of 256 kernels of 5 by 5 dimension, followed by layers of kernels with dimension 3 by 3. Striding and max pooling served to lower the size of the feature maps until entering the dense layers at the end for classification. All hidden layer activations used the Rectified Linear unit (ReLU) function. A scheme of the architecture can be seen in Figure 9.1

This is a typical solution for using convolution networks in image classification problems. The convolution layers are used to find the appropriate features from the unstructured image data, and then these features are fed into dense layers for classification.

VGG16

One problem with the AlexNet architecture is the large number of parameters in the kernels with greater width and height. In 2014, Simonyan and Zisserman [67] proposed an architecture, later known as VGG16, that used only convolution kernels with dimensions of 3 by 3. The rationale for this is that the

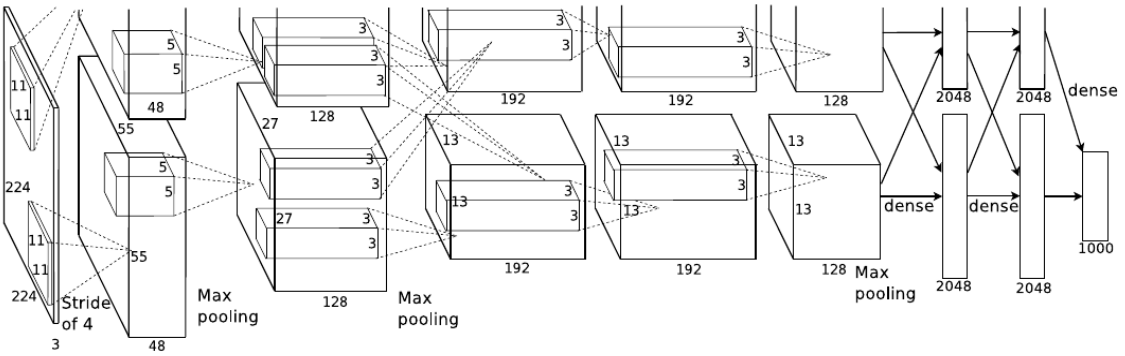


Figure 9.1: AlexNet, from [35].

same receptive field of a larger kernel can be obtained by stacking smaller kernels and thus reducing the number of parameters needed. Figure 9.2 illustrates this. The kernel of width 3 at the lower layer only receives inputs from a window of width 3. However, the kernel at the next layer, receiving the output of 3 consecutive kernels from the lower layer, will be receiving inputs from a window of width 5 in the input layer. The reason why this saves parameters in two dimensions is that a 5 by 5 kernel would require 25 parameters while two 3 by 3 kernels only need a total of 18 parameters (9 parameters each).

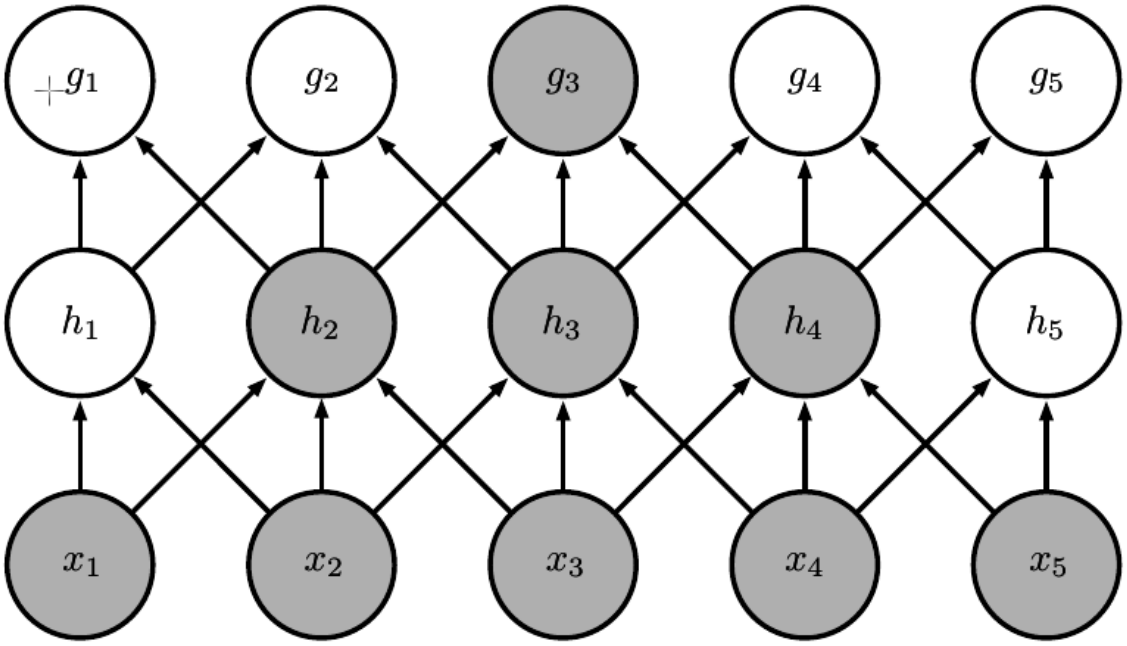


Figure 9.2: Stacking smaller kernels. Image from [21].

The effect is even more pronounced with larger receptive fields. For example, a 11 by 11 kernel, as used in the first layer of AlexNet, can be replaced by 5 layers of 3 by 3 kernels. These 3 by 3 kernels would need only 45 parameters in total (9 each for the 5 kernels) instead of the 121 parameters of a single 11 by 11 kernel. Figure 9.3 illustrates the VGG16 architecture. Note that there are two or three convolution layers between max-pool layers. All layers used the ReLU activation function.

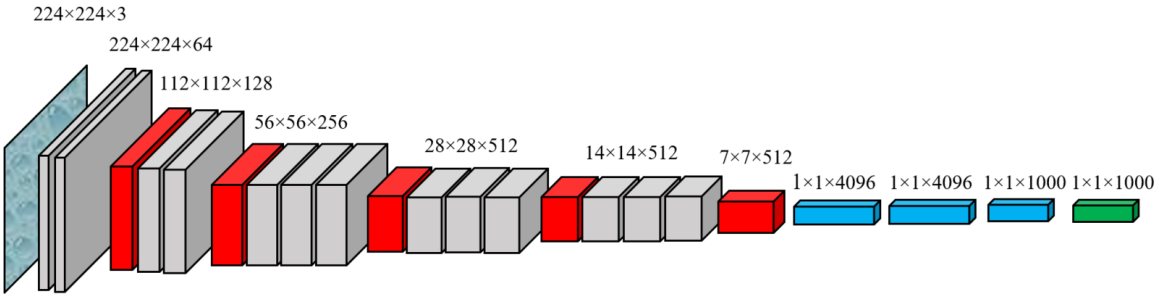


Figure 9.3: VGG network. Gray blocks represent convolution layers. Red blocks represent max-pool layers. Blue blocks are fully connected layers (and green for softmax at the end). Image by Nshafiei, CC-SA.

ResNet

AlexNet and VGG16 demonstrated that deep convolutional networks can be very powerful for image classification, and this power came from the depth of those networks. However, deep networks can be difficult to train. The problem of vanishing gradients, which we saw previously, can be mitigated by using activation functions that do not saturate. But simply stacking additional layers can lead to a degradation in performance, even in the training set, by making optimization harder. Each layer must "wait" for the previous layers to learn useful features until the network can start improving performance, which makes it harder for the optimizer to find appropriate parameters when the network is too deep.

In 2015, He and others beat the performance of the VGG-16 network with residual networks [25], with up to 152 layers, eight times deeper than VGG-16. The solution was to use identity connections that skipped several layers, copying the values and adding them to the output of a layer further along the network. This forced the intermediate blocks to learn the difference between the input and the desired output, or the *residual* to be added to the input in order to provide the best answer. Figure 9.4 illustrates this.

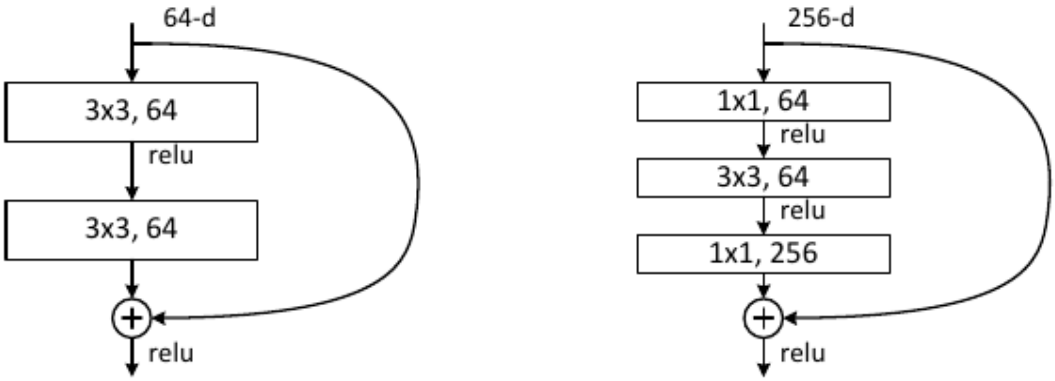


Figure 9.4: Examples of residual blocks with two and three convolution layers and a shortcut connection. Image from [25].

This can help with training of very deep networks because the identity shortcut connections skipping layers bring useful information from the input deeper into the network even in the beginning, when parameters are random. This helps the optimizer find the right path to minimize the loss function.

Another interesting feature of the deeper ResNets (*e.g.* ResNet 152) is the use of 1 by 1 convolution kernels for changing the number of filters. This is illustrated on the right panel of Figure 9.4. It may

seem strange at first to have a convolution filter using a kernel with a an input of 1 by 1, but note that the kernel will have weights for all channels in the input. So a 1 by 1 kernel will compute a linear combination of the incoming channels and by using a number of such kernels we can change the number of channels without requiring more parameters for more complex convolutions. In the case of ResNet152 the input containing 256 channels is compressed into 64 channels by a 1x1 kernel, then passed through 64 3x3 kernels, which in this case only need 64x3x3 parameters each, and then expanded again using a convolution with 256 1x1 kernels.

The Inception module

One clear trend with deep convolution networks is that increasing the size and complexity of the network also makes it more powerful. However, this also makes it harder to train and more expensive in computation and memory requirements. Furthermore, it is likely that only part of the connections are necessary. One evidence for this is the success of dropout as a regularization method. With this in mind, Szegedy and others proposed a CNN (GoogLeNet) composed of Inception modules[71], as illustrated in Figure 9.5

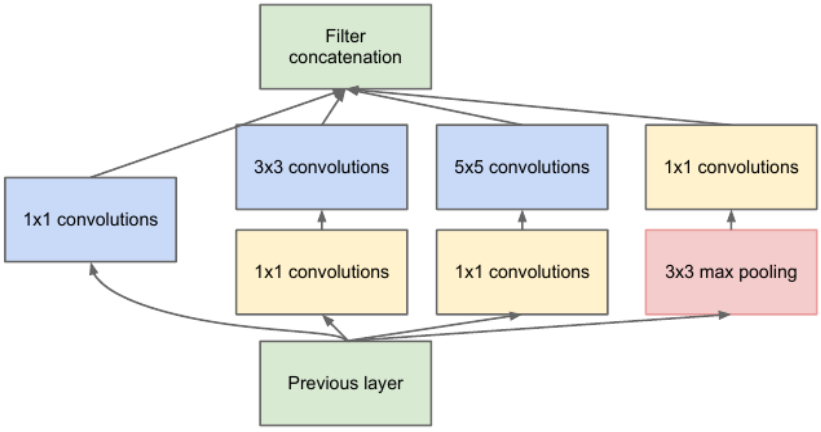


Figure 9.5: Inception module, version 1, from [71].

The 1x1 convolution layers in the inception module reduce the number of channels that enter the module, learning linear combinations of input maps that compress the relevant information. Even though the main purpose of these layers is just linear compression, the authors find that they work better with a non-linear activation (ReLU), thus serving a dual purpose of linear compression and non-linear transformation.

The outputs of these 1x1 convolution layers are fed into blocks of different sized kernels and then the resulting feature maps are concatenated in depth (*i.e.* stacked together) to form the output of the module. This includes a max pooling layer with a stride of 1, so that even though the input channels are pooled in a 3x3 window, their size remains the same.

The GoogLeNet stacks 9 inception modules, along with pooling and other layers, as illustrated in Figure 9.7. One other feature of GoogLeNet is the addition of two intermediate classification outputs. These were intended to help train the first layers by forcing them to create useful representations for the classification task but, empirically, these additional classifiers seem be of benefit simply by serving as additional regularization, merely improving the performance of the network at the final stages of training.

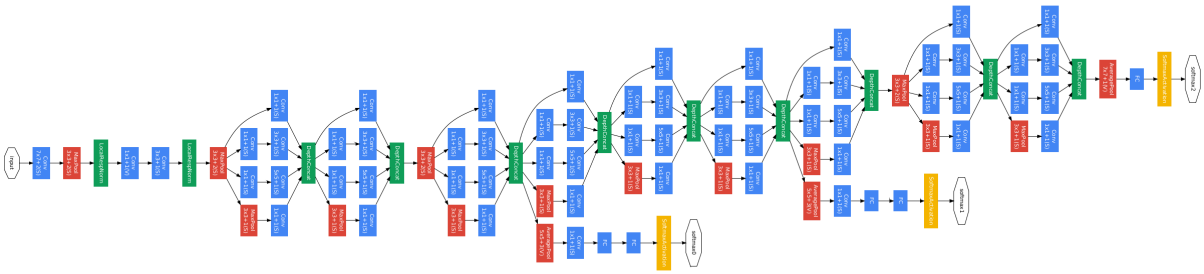


Figure 9.6: GoogLeNet, from [71].

In a subsequent paper, the authors proposed some optimizations to the Inception modules[72]. The optimizations involved replacing large kernels of size n by n with a stack of two kernels of dimensions 1 by n and n by 1. The authors used kernels of size 1x7 and 7x1. The rationale for this is that instead of using a 7x7 kernel, which requires 49 parameters per input channel, a 1x7 kernel can scan for horizontal patterns while the next kernel, of 7x1, will scan for vertical patterns in combination with the previous horizontal patterns. This results in scanning the same window size but simplifying the network by reducing the number of parameters. In this case, from 49 to 14.

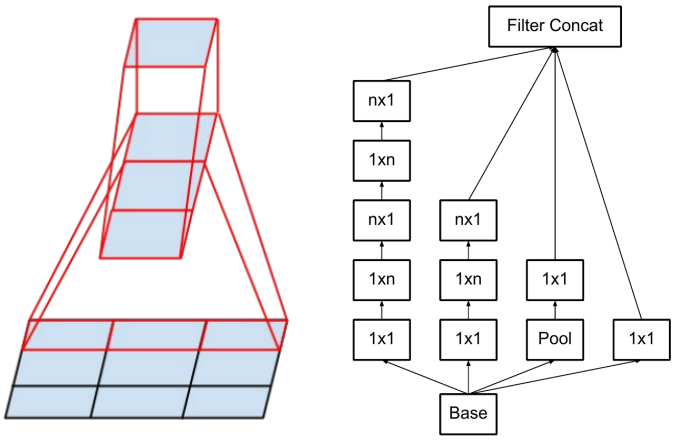


Figure 9.7: Optimization of the Inception modules, from [72]. The left panel shows the decomposition of a 3x3 kernel, requiring 9 parameters per input channel, into two kernels of 1x3 and 3x1, which in total only require 6 parameters per input channel.

9.2 Image segmentation with fully convolutional networks

Image segmentation is the assignment of a class to each pixel in the image, thus outlining each object. Segmentation can be divided into semantic segmentation, where pixels are classified according to the type of object, and instance segmentation, which distinguishes different instances of the same type (*e.g.* different people in the image).

Fully Convolutional Network (FCN)

In 2015, Long *et. al.* proposed a method to convert CNN classifiers into image segmentation networks by considering that dense layers are equivalent to convolution layers with kernels that overlap the whole input space [42]. By converting the fully connected layers at the end of CNN classifiers to convolution

layers, we can obtain a fully convolutional network that can output a coarse map classifying patches of the input.

To obtain a finer segmentation map, the authors proposed using fractional stride convolution to increase the resolution of the feature maps until the desired resolution is reached. Fractional stride convolution and transposed convolution are two convolution methods that can be used for this purpose while applying learned filters. However, as we will see later, this may not be the best approach. Figure 9.8 illustrates these convolutions.

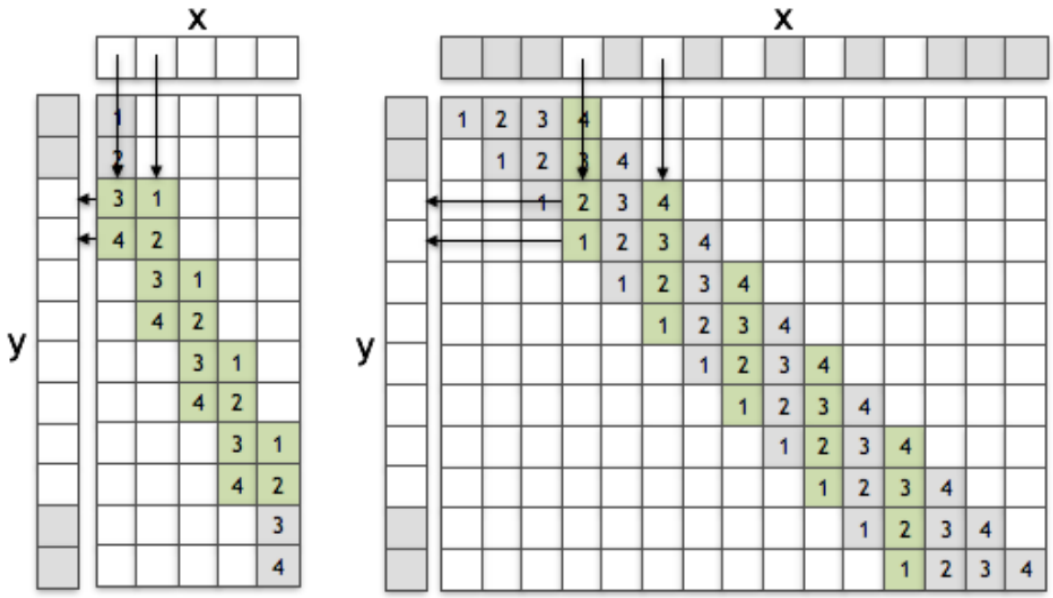


Figure 9.8: Transposed (left) and fractional stride (right) convolutions. See text for explanation. Image source [65]

In the left panel of Figure 9.8 we see a transposed convolution. To compute the output y of the convolution, instead of applying the 1D kernel to a window of the input x, we can imagine that the kernel has a window on y and we are adding the weights multiplied by each value of x to different positions of y, and keep adding as we shift the kernel over the output y. In a fractional-stride convolution, as shown on the right panel, we can think of the input vector x as having "holes" between values and as we shift the kernel only a few weights are used at a time. This is almost equivalent to upsampling x by padding with zeros. In both cases the output y has more elements than x. The authors experimented with a single upsampling step starting from the last convolution, corresponding to the prediction layer, combined with upsampling from previous layers in the original CNN, all using fractional stride convolutions.

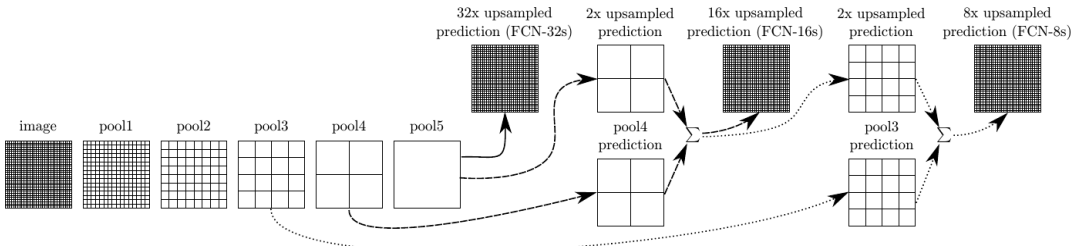


Figure 9.9: The final upsampling step combining the resulting convolutions from different layers. From [42]

Given that semantic segmentation is a pixel classification problem, the output of the network can be one feature map if we only have two classes or a stack of N feature maps for N classes when N is greater than 2, with sigmoid activation in the first case or softmax in the second.

U-Net

The U-Net architecture differs from the FCN in two main aspects. First, it creates a symmetric path with convolutions and pooling to reduce spatial resolution while increasing filters, and then reverting this path with upsampling and filter reductions until the original resolution is reached. And second, in the upwards path it includes feature maps from the equivalent level in the downwards path, thus skipping the intermediate layers. Figure 9.10 illustrates this architecture.

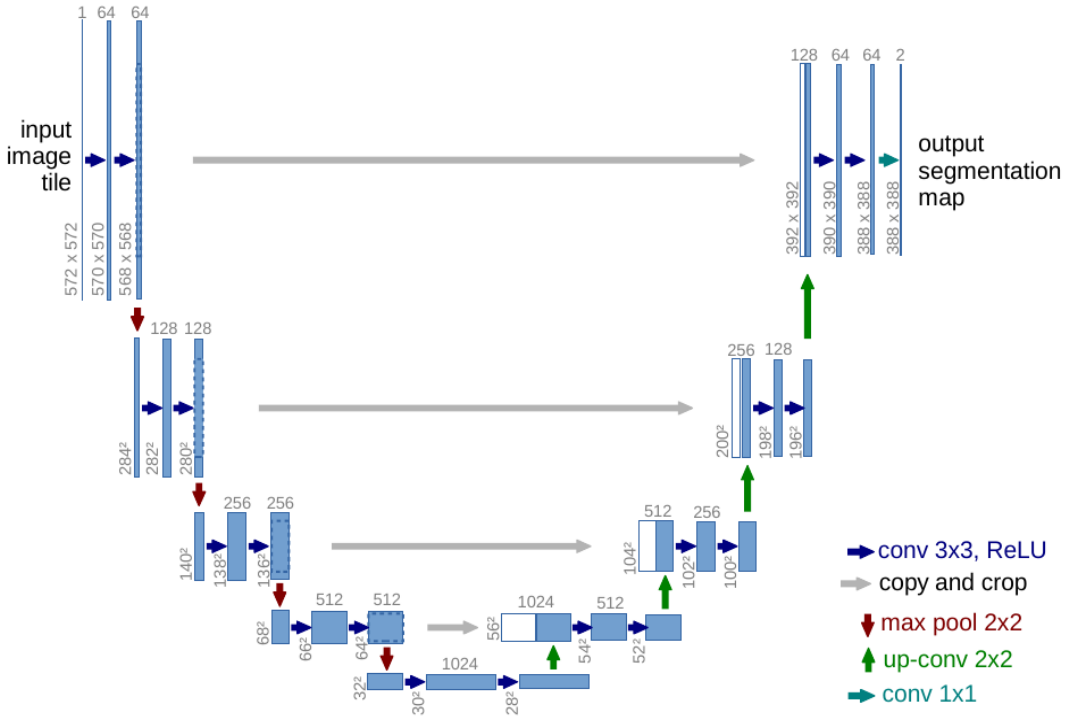


Figure 9.10: The U-Net architecture, from [58]

Another difference is in the upsampling steps, which use using a combination of a simple upsampling operation that duplicates the resolution followed by a normal 2x2 convolution[1]. Although this may seem equivalent to the fractional stride convolutions used in FCN, combining simple upsampling with a standard convolution avoids some artefacts that can be caused due to overlapping effects in the kernels. We will see this in more detail when examining generative model.

Finally, U-Net also uses a more sophisticated loss function. In addition to the pixel-wise softmax cross entropy for classifying each pixel, it includes additional weights for pixels in the border of each region in the training set to encourage a more precise segmentation.

[1] At least as described in the original paper; some implementations use transposed convolutions for this

9.3 Further Reading

Although more detailed than the level covered in this course, it may be useful to read the papers referred to in the previous sections.

Chapter 10

The Keras functional API

The Keras functional API. Transfer learning

10.1 Exercise: transference learning with the Keras functional API

The Keras functional API is more versatile than the sequential API. Layer instances in Keras are all callable, as Python functions. Each layer instance, when called, receives a tensor as argument and returns a tensor. This way we can chain layers in sequence. For example, here is a code fragment to create a convolution layer similar to the tutorial on Chapter 8. Note that we are now loading the MNIST data set instead of the Fashion MNIST which was used for training the previous model.

```
from tensorflow import keras
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Input, BatchNormalization, Conv2D, Dense
from tensorflow.keras.layers import MaxPooling2D, Activation, Flatten, Dropout

((trainX, trainY), (testX, testY)) = keras.datasets.mnist.load_data()
trainX = trainX.reshape((trainX.shape[0], 28, 28, 1))
testX = testX.reshape((testX.shape[0], 28, 28, 1))
trainX = trainX.astype("float32") / 255.0
testX = testX.astype("float32") / 255.0

inputs = Input(shape=(28,28,1),name='inputs')
layer = Conv2D(32, (3, 3), padding="same", input_shape=(28,28,1))(inputs)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
layer = Conv2D(32, (3, 3), padding="same")(layer)
layer = Activation("relu")(layer)
layer = BatchNormalization(axis=-1)(layer)
layer = MaxPooling2D(pool_size=(2, 2))(layer)
...
```

As we feed the output tensor of each layer into the next, this creates a chain of operations and tensors in the computation graph. In this way, the functional API is similar too the sequential API.

However, since we can decide which tensors go into which layers, we can create more complex graphs as necessary.

Note also that we can give names to specific layers. For example, the input layer is named `inputs` in this model. This makes it easier to find the right layers if we want to connect them in different ways. In this example, we want to recreate the CNN model from Chapter 8[1] to reuse the trained weights for the convolutional part of the network. This network was trained on the Fashion MNIST dataset and, instead of retraining the whole network, we will retain the feature extraction of the convolutional part and merely retrain the dense part of the network.

Once the layers are chained in the same way as the original model, we can create a model instance and compile it. To create the model we specify the inputs and outputs of the chain of tensors and operations.

```
#continuing the model
...
features = Flatten(name='features')(layer)
layer = Dense(512)(features)
layer = Activation("relu")(layer)
layer = BatchNormalization()(layer)
layer = Dropout(0.5)(layer)

layer = Dense(10)(layer)
layer = Activation("softmax")(layer)
old_model = Model(inputs = inputs, outputs = layer)
```

Now that we have the model instance, we compile the model and load the weights. We will also disable training for all layers of the old model. This in order to use this model for training new layers.

```
old_model.compile(optimizer=SGD(), loss='mse')
old_model.load_weights('fashion_model')
for layer in old_model.layers:
    layer.trainable = False
```

Now we create a new model. First, we chain dense layers starting from the output of the convolutional net in the old model. This is why we named the `Flatten` layer as `features`. This makes it easier to get the correct layer from the old model. The `output` attribute of a layer is the tensor that the layer outputs, and this is the tensor we use as input for a new dense layer.

```
layer = Dense(512)(old_model.get_layer('features').output)
layer = Activation("relu")(layer)
layer = BatchNormalization()(layer)
layer = Dropout(0.5)(layer)
layer = Dense(10)(layer)
layer = Activation("softmax")(layer)
model = Model(inputs = old_model.get_layer('inputs').output, outputs = layer)
```

After creating a new chain of layers, we create the new model whose input is the input tensor to the old model and the output is the last layer in the new chain. This way we are using the convolutional part of the old model and, since we disabled training for this part, we can now fit the new model by training only the new part.

[1] Based on a PyImageSearch tutorial by Adrian Rosenbock, https://www.pyimagesearch.com/2019/02/11/fashion-mnist-with-keras-and-deep-learning/

```
opt = SGD(lr=1e-2, momentum=0.9)
model.compile(loss="categorical_crossentropy", optimizer=opt, metrics=["accuracy"])
model.summary()
NUM_EPOCHS = 5
BS=512
H = model.fit(trainX, trainY,validation_data=(testX, testY),
              batch_size=BS, epochs=NUM_EPOCHS)
```

The `model.summary()` gives us a summary of the model. You should be able to see the different layers and, at the end, the indication of the trainable and non trainable parameters. These include some non-trainable layers, like batch normalization layers, but also all parameters in the convolution layers, which were frozen after loading. You should see something like this in the summary:

```
...
features (Flatten)           (None, 3136)              0
_________________________________________________________________
dense_2 (Dense)              (None, 512)               1606144
_________________________________________________________________
activation_6 (Activation)    (None, 512)               0
_________________________________________________________________
batch_normalization_5 (Batch (None, 512)               2048
_________________________________________________________________
dropout_3 (Dropout)          (None, 512)               0
_________________________________________________________________
dense_3 (Dense)              (None, 10)                5130
_________________________________________________________________
activation_7 (Activation)    (None, 10)                0
=================================================================
Total params: 1,679,082
Trainable params: 1,612,298
Non-trainable params: 66,784
_________________________________________________________________
```

Using the pretrained convolution network to extract useful features, just a few epochs are enough to reach around 97% validation accuracy.

Chapter 11

Autoencoders

Autoencoders. Undercomplete autoencoders. Regularization in autoencoders.

11.1 Autoencoders

An autoencoder is an artificial neural network that is trained to recreate the input in the output. In other words, it is trained to approximate a function F so that $F(x) = x$. The motivation for this is to obtain from the autoencoder some useful representation in the hidden layers. Figure 11.1 illustrates an undercomplete autoencoder trained to recreate cat images. The designation of undercomplete refers to the smaller size of the hidden layer at the middle of the autoencoder.

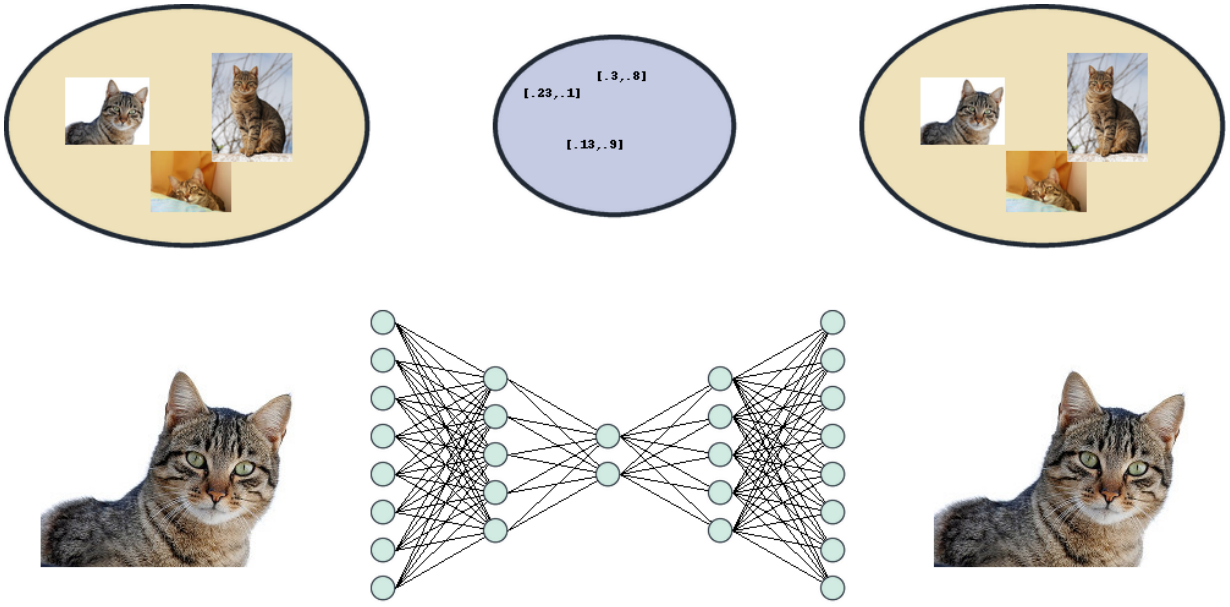


Figure 11.1: Representation of an undercomplete autoencoder. Cat images by Joaquim Alves Gaspar CC-SA.

We can consider that the first half of the autoencoder, the encoder part, will transform the input into a representation at the middle of the autoencoder. Then the second half, the decoder, will convert this representation into the input vector again. In other words, we are training the network to learn two functions, f and g, corresponding to the encoder (f) and decoder (g) such that $g(f(x)) = x$. If we

constrain the encoder adequately, the encoding function $h = f(x)$ may give us a useful representation of x. And an advantage of the autoencoder architecture is that we do not need any additional labels. This means that, even though the network is trained by minimizing a loss function with backpropagation, as usual, in practice this works as unsupervised learning in that we do not need labelled data to train the autoencoder.

The requirement is simply that the network be constrained, by its architecture or by regularization, so that the encoder needs to transform the input x into a useful latent representation h. One simple way of doing this is to force the dimension of h to be smaller than the dimension of x, which is the case of an undercomplete autoencoder. But there are other options, as we will see below. But first we can take a look at one intuitive application of autoencoders: manifold learning.

11.2 Regularized Autoencoders

We do not need to reduce the dimension of the latent representation in order to force the autoencoder to give us a useful representation. In theory, we could even expand the input data into a larger dimensional space. This would be an overcomplete autoencoder, as illustrated in Figure 11.2. As long as we restrict the ability of the autoencoder to simply copy the input to the output, we can force it to learn a useful representation. And we can do this with regularization, by changing the training of the autoencoder. In particular, by changing the loss function.

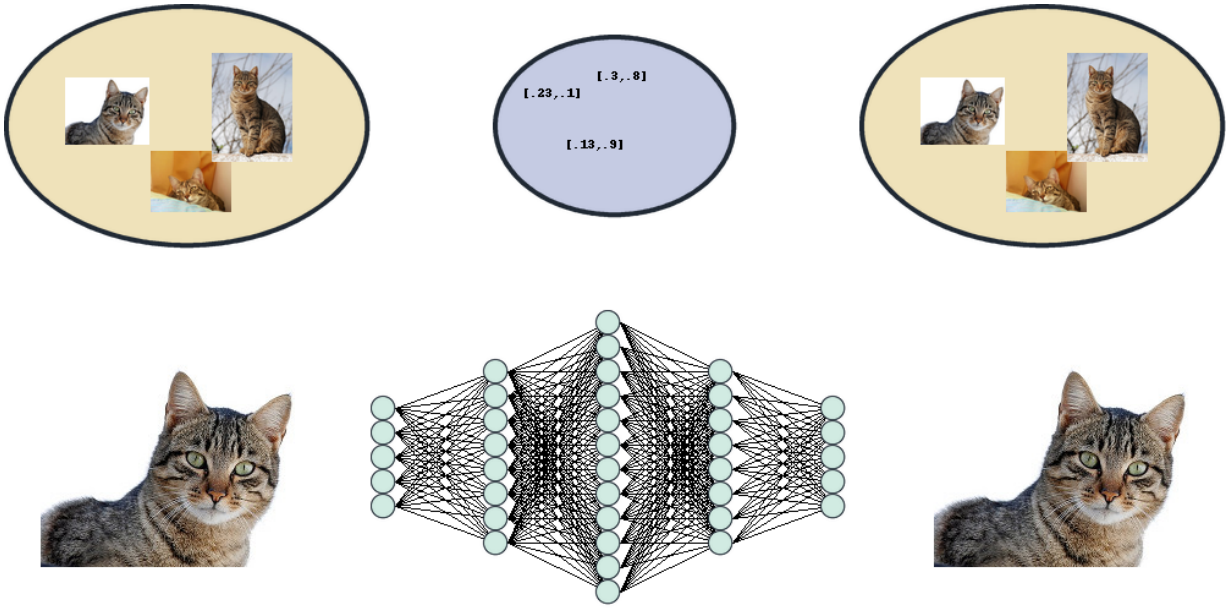


Figure 11.2: Representation of an overcomplete autoencoder. Cat images by Joaquim Alves Gaspar CC-SA.

11.3 Sparse Autoencoders

In a sparse autoencoder, we want the neurons in the latent representation layer to be mostly inactive for each example. We can specify this by choosing a target probability for the average activation of each neuron in the middle hidden layer of the autoencoder. This can be estimated by the average activation over a set of examples:

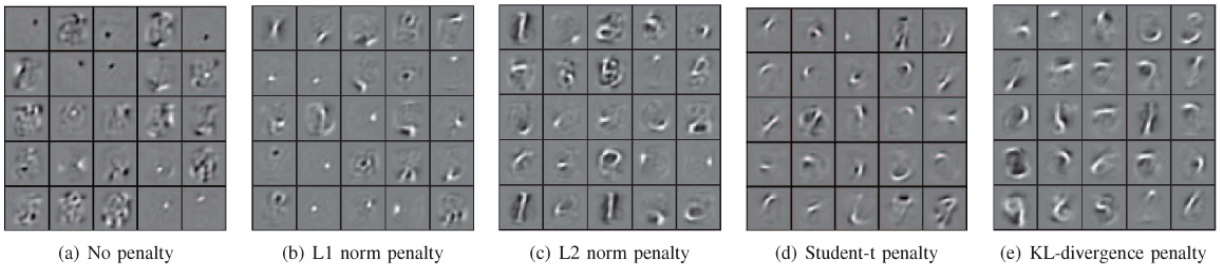
$$\hat{p}_i = \frac{1}{m} \sum_{j=1}^{m} h_i(x_j)$$

If we want the probability of h_i firing to be p_i, we can include in the loss function a penalization term corresponding to the Kullback-Leibler divergence between the activations of this neuron and a binomial distribution with probability p_i:

$$L(x, g(f(x))) + \lambda \sum_i \left(p \log \frac{p}{\hat{p}_i} + (1-p) \log \frac{1-p}{1-\hat{p}_i} \right)$$

By forcing only a few neurons to fire for each example, using a small activation probability (*e.g.* 0.05) these neurons will specialize in identifying particular features that distinguish the examples.

Other regularization options for sparse autoencoders include L1 regularization applied to the activations instead of the weights, L2 regularization, and others. Jiang and others show a comparison of difference sparseness penalties using the MNIST dataset[32]. Some results are illustrated in Figure 11.3.



(a) No penalty (b) L1 norm penalty (c) L2 norm penalty (d) Student-t penalty (e) KL-divergence penalty

Figure 11.3: Normalized input maps that maximize activation of latent neurons in sparse autoencoders, using different sparseness penalties. Image from [32].

11.4 Denoising Autoencoders

We saw previously that nose injection can be used for regularization in neural networks, since it forces the network to rely less on individual input values and more on larger patterns. This also works for autoencoders. In a denoising autoencoder, the network is trained to reconstruct the original x from a corrupted version $\tilde{x}$, obtained by adding some noise to x, typically by randomly setting values to zero. Figure 11.4 illustrates this.

11.5 Contractive Autoencoders

Rifai and others proposed adding a to the loss function a regularization term that reduces the autoencoder's sensitivity to small variations in the input[57], in the form of a penalization proportional to the square of the gradients of the activations in the hidden layer with respect to the inputs:

$$L(x, g(f(x))) + \lambda \sum_i ||\nabla_x h_i||^2$$

This makes the encoding more robust and groups similar examples closer together in the manifold of the latent representation h (hence the name of contractive autoencoder). Nevertheless, the autoencoder

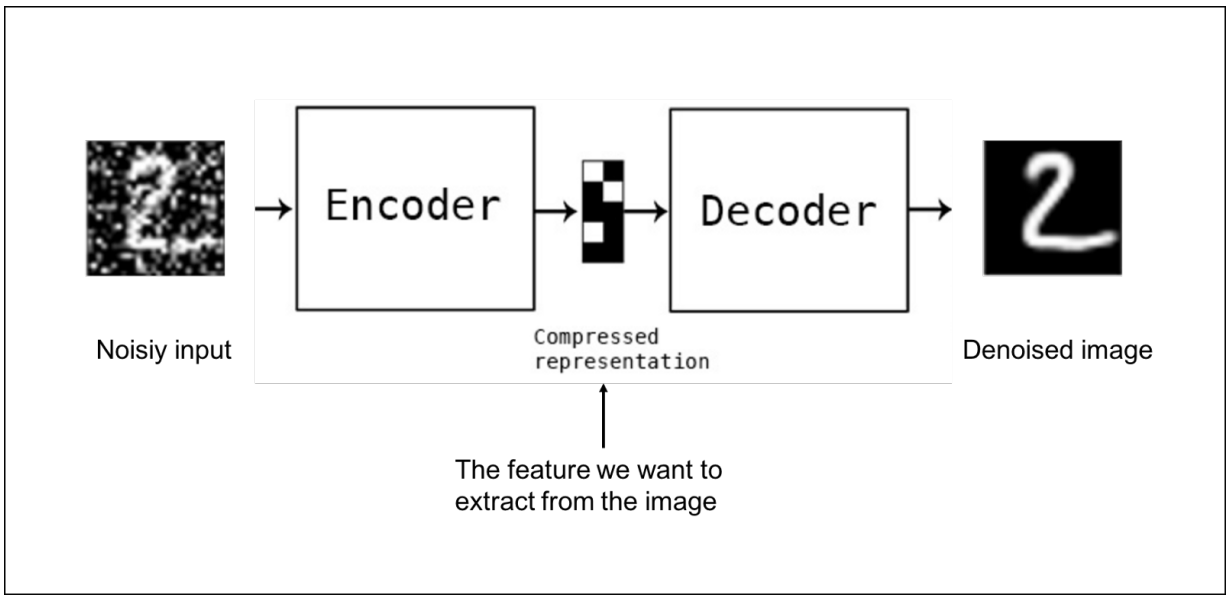


Figure 11.4: Normalized input maps that maximize activation of latent neurons in sparse autoencoders, using different sparseness penalties. Image by Adil Baaj.

also needs to reduce the reconstruction loss for the input, so it must distinguish between more or less penalized directions when moving from each point. We can visualize these directions and compare the result of a contractive autoencoder and other dimensionality reduction techniques, like PCA. This is what Rifa *et. al.* did for the manifold tangent classifier[57], based on stacking contractive autoencoders, as shown in Figure 11.5

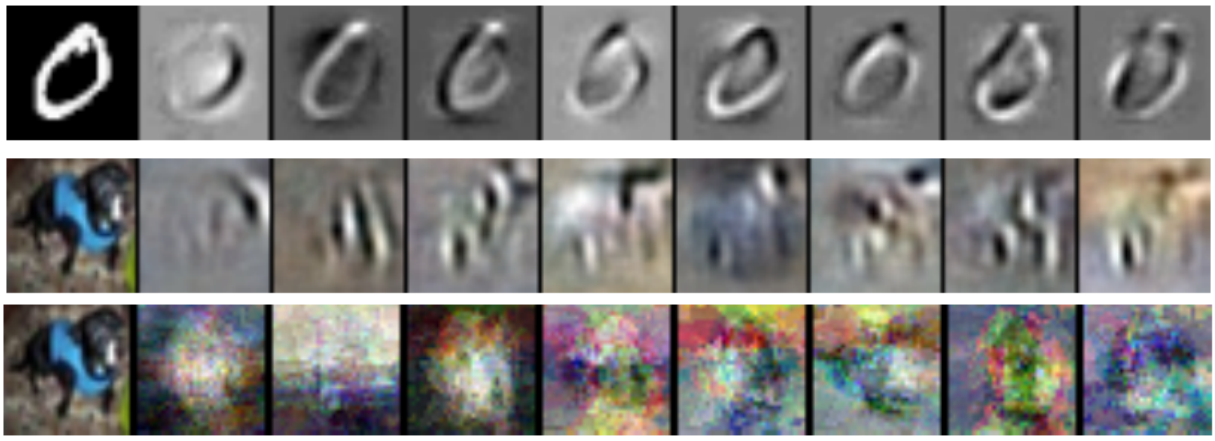


Figure 11.5: Principal singular values of the Jacobian matrix, showing tangents of each point in a contractive autoencoder, and compared to PCA. From [57].

The effect is similar to that of a denoising autoencoder, and a contractive autoencoder can be more expensive to train due to the need to compute the gradients with respect to the inputs.

11.6 Convolutional Autoencoders

Unsupervised Segmentation

In 2017, Xia and others proposed the W-Net, an autoencoder based on the U-Net, to perform unsupervised segmentation. Figure 11.6 compares these networks. In addition to the reconstruction loss, the

W-Net includes a soft normalized cut loss for the encoding relative to the original image.

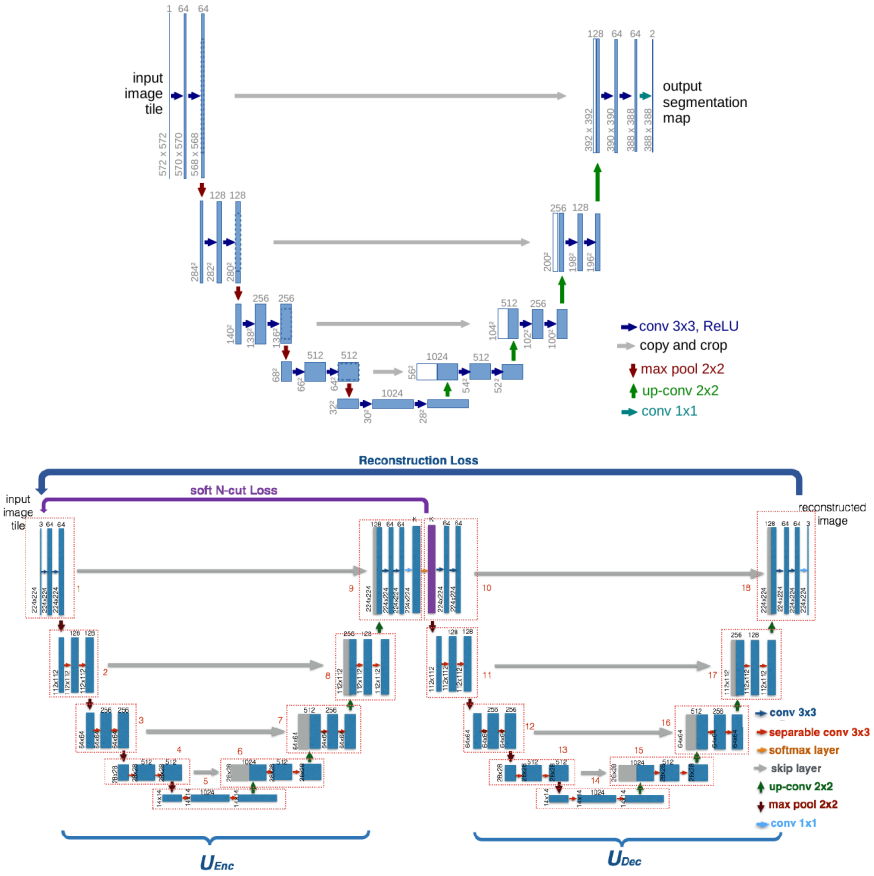


Figure 11.6: U-Net [57] and W-Net [79]. Images from the papers.

In graph theory, a cut is the sum of the weights between nodes in two partitions of the graph. If we split the graph into subgraphs A and B, the cut will be:

$$cut(A, B) = \sum_{u \in A, v \in B} w(A, B)$$

The normalized cut[64] is divided by the sum of the weights between each partition A and B and all nodes in the graph V:

$$Ncut(A, B) = \frac{cut(A, B)}{assoc(A, V)} + \frac{cut(A, B)}{assoc(B, V)}$$

$$assoc(X, V) = \sum_{u \in X, t \in V} w(u, t)$$

Minimizing the normalized cut we can find a good way to partition the graph. This can be used for image segmentation by considering the image to be a graph where each pixel is a node and nearby pixels are connected by edges. The weights are a measure of the similarity between these pixels, which can include color, distance, brightness and other features (see [64] for more details).

Since the normalized cut is not differentiable, W-Net uses an adaptation (the soft normalized cut), but the idea is the same: trying to find a partition that splits the pixel graph in a way that minimizes the weights between the parts relative to the weights involving each part.

W-Net is then trained in each minibatch first by updating the encoder to minimize the soft normalized cut loss function and then updating the complete network to minimize the reconstruction error. These alternating steps are then repeated.

The encoding is then subjected to post-processing to smoothen the regions and cluster similar pixels. Figure 11.7 shows the result for several images and different levels of post-processing.

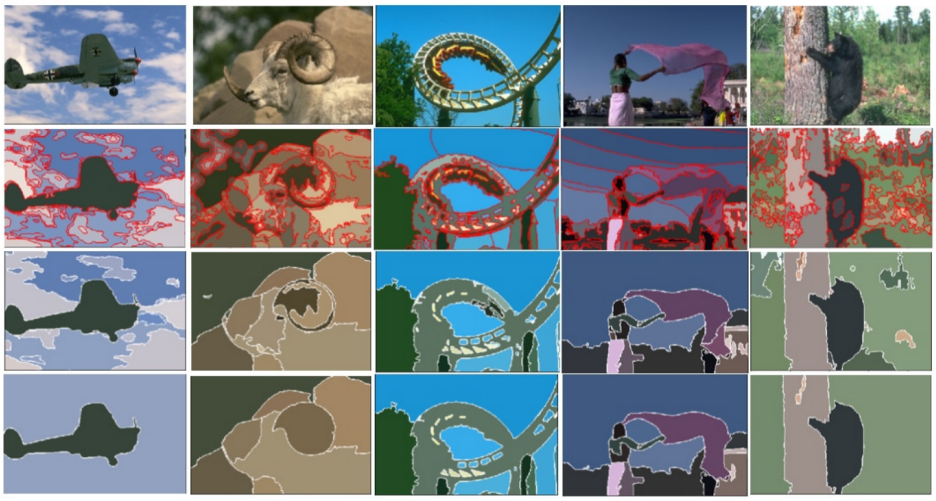


Figure 11.7: Results of unsupervised segmentation with W-Net [79]. Images from the papers.

11.7 Application

Manifold Learning

Mathematically, an n-dimensional manifold, or n-manifold, is a set of points such that each point and its neighbours form an approximately Euclidean space. For example, seismic events at the surface of the Earth form a two-dimensional surface in three-dimensional space. Figure 11.8 shows two examples of manifolds.

If our data follows a lower-dimensional manifold, then it is possible to compress it into a lower-dimensional representation without losing much information. In other words, we can train an undercomplete autoencoder to find a representation h with a dimension lower than x while retaining the ability to reconstruct the data outside the training set. This last point is important: overfitting the autoencoder to the training data can give good reconstructions but the latent representation may become useless.

11.8 Further Reading

1. Goodfellow et. al., Chapter 14[21]

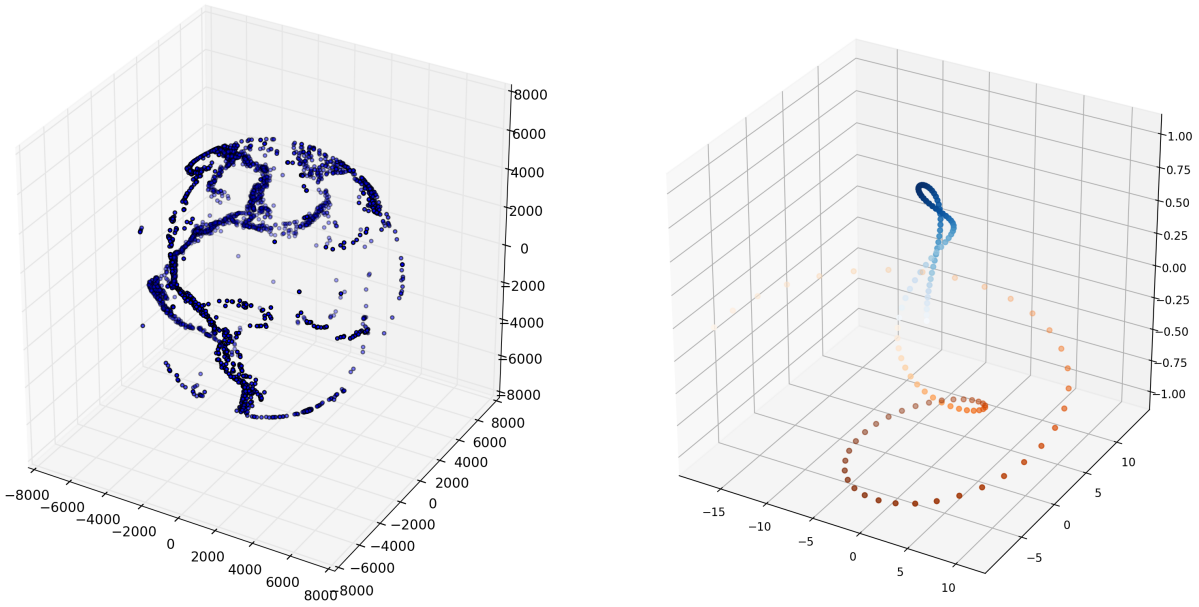


Figure 11.8: Examples of two manifolds in three-dimensional space: the 2-dimensional manifold of seismic events on the surface of the Earth and a 1-dimensional manifold of hypothetical data that follows a line in 3D.

Chapter 12

Representation Learning

Learning adequate features. Unsupervised pretraining.

12.1 Learning Features

Representation learning, or feature learning, is the automated extraction of useful features from data. The motivation for this is that having adequate features may make problems much easier to solve. For example, representing numbers with Arabic numerals makes algebraic operations much easier to perform than using Roman numerals. Think of the difference between adding 58 and 12 or adding LVIII and XII.

In classical machine learning, features are usually extracted by human intervention. This not only requires domain knowledge but the design of procedures for converting the original data in a more useful representation. But with deep neural networks we can rely on the model finding good representations of the data for whatever target we select. There are several reasons for assuming this to be the case, besides empirical confirmation that it is so in several fields. Here are some examples from Bengio and others[6]:

Manifolds Real data is not uniformly distributed over the whole space of the original features. Rather, it usually spans a lower dimensional manifold in which we can project it without losing much information while simplifying the representation. In addition, after projecting into the manifold it is likely that any function of the data we may want to learn becomes smoother.

Smoothness In simple terms, this is a property of a function that has a similar outputs for similar inputs. In general, when we use data with a large number of dimensions, this property does not hold for functions we may want to learn (for classification or regression, for example). However, if we obtain a better representation of the features, these functions tend to be smoother.

Disentanglement Useful features tend to be able to vary independently, because relations between features lead to redundancy and may make learning harder. A good representation will disentangle such features so that the features that are extracted are more independent.

Hierarchy Useful features are generally organizable in hierarchies, with different levels of representations. For example, in image processing lower level features may include edges of different

orientations, contrast and so on, while higher level features may be patterns such as eyes or wheels. Deep models are good for learning these hierarchical representations, and such hierarchies can help reuse parts of these models to extract features at relevant levels in different problems. We saw this previously in the transfer learning example by using the convolutional layers of a pretrained network, which is related to the shared factors aspect we cover next.

Shared factors A related aspect of adequate features is that they can be useful for different problems and across similar datasets. This means not only that adequate features can be reused in other problems but also that we can learn these features in different datasets and then use the ability to extract them in the problem we are interested in solving. This is why transfer learning is often useful.

Sparsity When considering useful features, for most examples only a few features will be useful. We can have a network that detects teeth, eyes, wings, feathers, fur and so on and for each example only some of these features will be relevant. This means that the representation will be less sensitive to small changes in the input vector.

Semi-supervised learning In general, the same features that can help us model the probability distribution of the set of input vectors, $P(X)$, can also be useful to model the conditional probability of some target given these vectors, $P(Y|X)$. This means that we can use unlabelled data to help us extract useful features for some supervised learning task, and since unlabelled data is generally more abundant this can be very helpful.

12.2 Unsupervised Pretraining

Autoencoders, covered in the Chapter 11, are one type of model that can help us extract features from unlabelled data. We train a neural network to output the same vector as the input but with some constraint to force a hidden layer to extract useful features. This can be used for a greedy pretraining of individual layers and was an important technique before the use of ReLU activations mitigated the vanishing gradient problem.

We can start with an autoencoder with a single hidden layer and train it to reconstruct the inputs after adding noise, for example. This is the denoising autoencoder we saw in 11.4. If after training we discard the decoder part of the autoencoder and retain only the encoder, we now have a layer capable of creating this representation of the data. We can use this tranformed data to train a new denoising autoencoder and repeat the process. This is called a stacked denoising autoencoder and can be used to pretrain the network with unsupervised learning before fine tuning the complete network with labeled data.

This pretraining technique is now rarely used, since modern activation functions and optimizers allow us to train deep networks all at once, which is more practical, but as Erhan and others showed in 2010, this pretraining acts as a form of regularization by compressing the training trajectories in a specific region of the parameter space [18], as shown in Figure 12.1. This figure represents training trajectories of two groups of 50 instances of the same model, one with pretraining and the other without, projected using t-SNE and ISOMAP. We can see that the different groups are spread in different regions of the parameter space and, using the ISOMAP projection that preserves distances at a larger scale, we can see that the pretrained networks span a much smaller region than with random initialization.

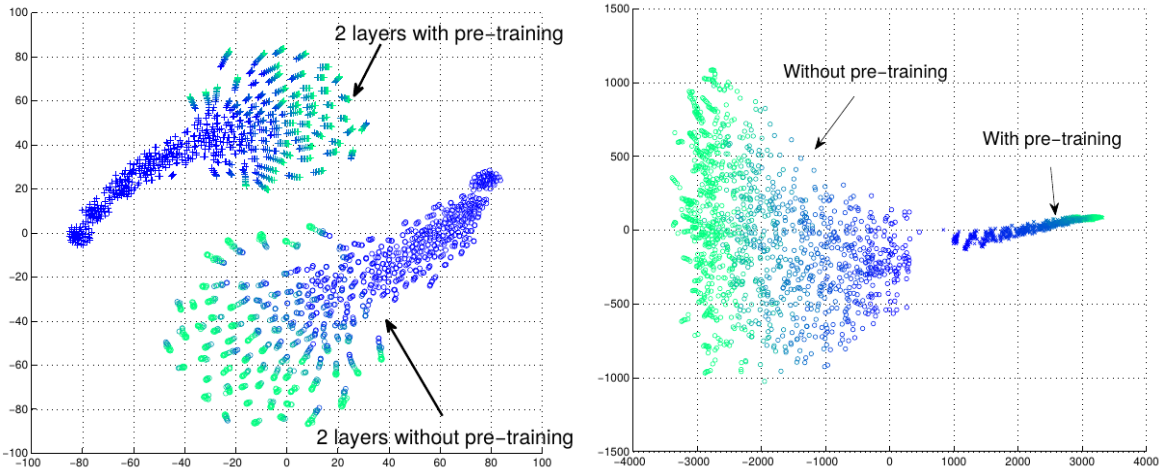


Figure 12.1: Projection with t-SNE[44] and ISOMAP[74], from Erhan et. al [18].

Although pretraining with stacked denoising autoencoders is now seldom used, unsupervised pretraining of layers to obtain desirable features is relevant in applications where the input data is highly dimensional, very sparse or has other undesireable attributes and when unlabelled data is abundant. For example word embedding in natural language processing.

One example of this is feature extraction from very sparse and high-dimensional data from motion sensors to classify human movements [47]. The original data has a dimension of 5000 but for each example only about 2% of the features are not zero. Using a denoising autoencoder followed by a PCA projection for visualization different groups of motions are apparent, as shown in Figure 12.2

12.3 Transfer learning

Good representations for a particular problem do not need to be learned specifically for that problem and dataset. We can use the same transformations to extract useful features in different problems or different data. This is called transfer learning. There are different situations where reusing the same transformations or starting from previously trained networks can be of benefit to solving some problem.

In the case of *domain adaptation* we model the same mapping from input to extracted features or output but with a different data set. For example, we trained a model to classify the sentiment of movie reviews but now want to classify the sentiment of reviews of items sold on an electronics store. Although the review texts are different, it is likely that the relation between the words and whether the person was happy or angry is similar.

In the case of *concept drift* the problem arises because either the mapping from input to output is gradually changing or the data distribution is changing. In either case, this change is gradual so we may not need to retrain our model from zero and it may suffice to just refresh the model regularly by training with small sets of recent data starting from the parameters of the previous instance of the model.

Zero-shot learning is an extreme case of transfer learning where we can use exactly the same instance to new data outside the distribution of the data used to train the model. Socher and others [69] give an example of this in cross-modal learning, where the models relate images and text. After learning the manifolds of images and words, obtaining representations of each modality of examples using unsupervised learning, and learning a supervised mapping between the image manifold and the word manifold. This means that even if new images do not correspond to any of the classes in the

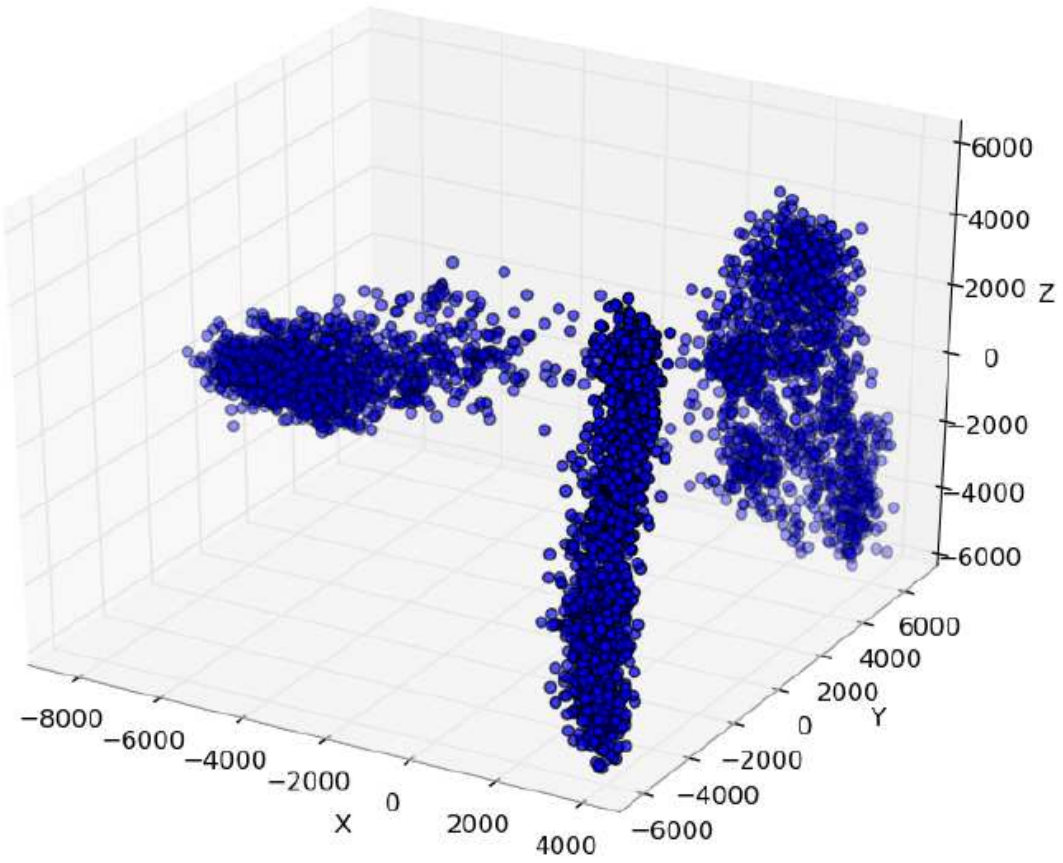


Figure 12.2: PCA from denoising autoencoder hidden layer[47]

training set, it may still be possible to map them into the word manifold and assign them novel classes. Figure 12.3 illustrates this for classes “truck” and “cat”.

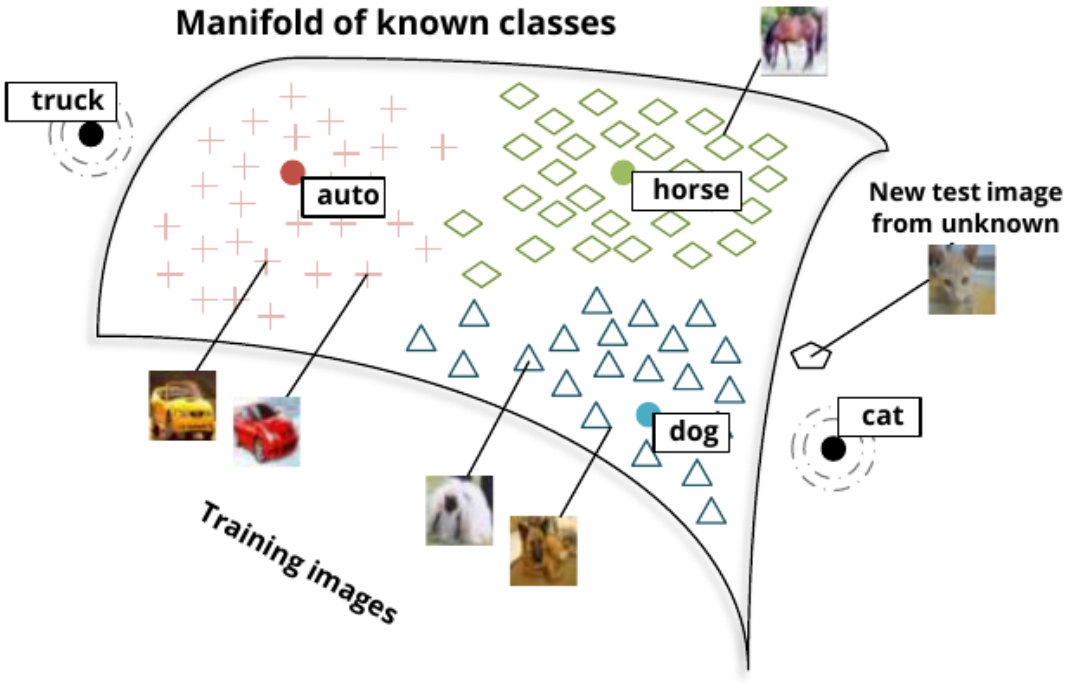


Figure 12.3: Zero-shot learning

12.4 Exercise: Autoencoder for projecting the UCI banknote dataset

Create an autoencoder to project into 2 dimensions the ICI banknote dataset we used in the first tutorial. Experiment with different architectures but one starting point could be to use 16, 8, 2, 8, 16 neurons followed by 4 linear neurons for the output. You can use leaky rectified linear units:

```
from tensorflow.keras.layers import LeakyReLU
...
layer = Dense(16)(layer)
layer = Activation(LeakyReLU())(layer)
```

For getting the encoded features after training you can create a new model with the same inputs as the autoencoder and outputs the latent features in the layer of 2 neurons (you can give it a suitable name when creating the autoencoder). This is just the encoder part of the autoencoder. You can also use the PCA class from Scikit-learn for comparison.

```
from sklearn.decomposition import PCA
...
encoder = Model(inputs = inputs, outputs = model.get_layer('encoded').output)
encoded = encoder.predict(Xs)

pca = PCA(n_components=2)
pca_result = pca.fit_transform(Xs)
```

Figure 12.4 illustrates possible results.

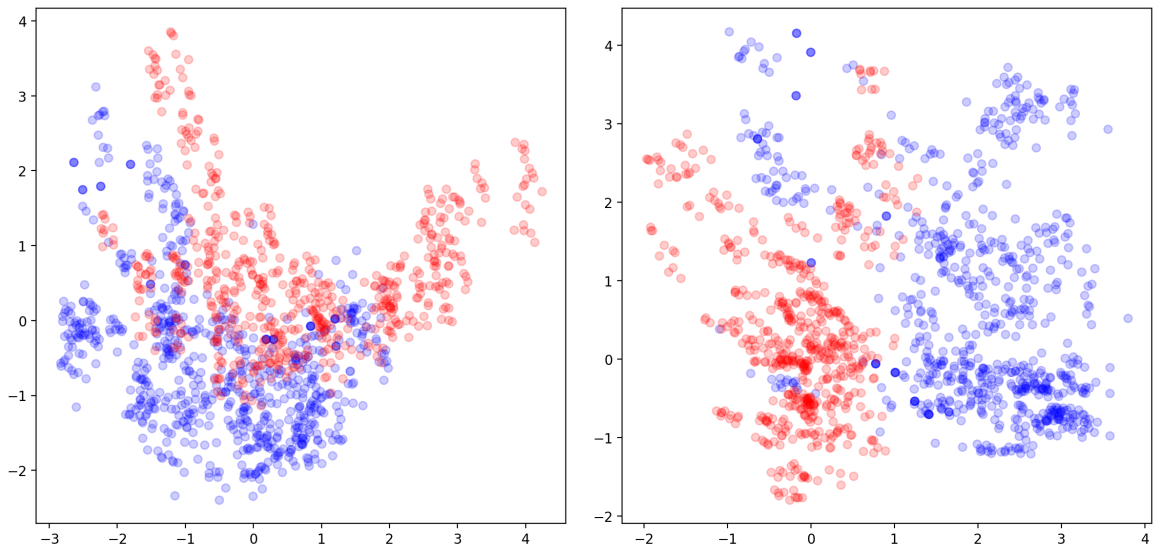


Figure 12.4: PCA (left) compared to the autoencoder projection (right).

12.5 Further Reading

1. Goodfellow et. al., Chapter 15[21]

Chapter 13

Image generation

Variational autoencoders and Generative Adversarial Networks.

13.1 Variational Autoencoders

As we saw previously, an autoencoder is an artificial neural network that can learn a representation of the input while being trained to recreate in the output the same vectors it receives in the input. By imposing constraints on this latent representation, with regularization or forcing a dimensionality reduction, we can obtain a useful representation of our data. In the case of images, the encoder will map from the original image to this hidden representation and the decoder part of the autoencoder can recreate the image from this representation. If we knew the hidden representations for other images the decoder can create, then we could use it to generate new images, even if they do not belong to the training set. Unfortunately, if we just sample at random from the space of possible values to feed into the decoder, it is likely that the output is not a useful image. Without knowing the distribution of values in the latent representation that correspond to encoded images we cannot use the autoencoder to generate useful images.

We can think of this problem considering the difference between a discriminative model and a generative model. A discriminative model approximates the conditional probability of the desired values y given the input x, or $p(y \mid x)$. A generative model tries to approximate the joint probability of target labels and inputs, $p(x, y)$. This allows us to generate data because we can compute the distribution of the inputs x for any desired output. Thus, we can consider the decoder in an autoencoder trained with images to be a discriminative model that outputs some image y given a latent representation vector z because the decoder approximates $p(y \mid z)$. If we know the probability distribution of z then we can use the decoder for generating images because $p(y, z) = p(y \mid z)p(z)$. This we can do with a variational autoencoder.

In the variational autoencoder, for each input the encoder will generate the parameters for a probability distribution. For example, a mixture of independent Gaussian distributions. The decoder will then receive as input a vector sampled from that distribution. Figure 13.2 illustrates this.

One issue that needs to be solved is how to train the autoencoder given that the input to the decoder is a vector randomly sampled from the probability distribution given by the output of the encoder. Since we cannot differentiate a random sampling, backpropagation cannot work through this gap. To solve this we can use a reparamitrization trick: instead of sampling from each Gaussian distribution,

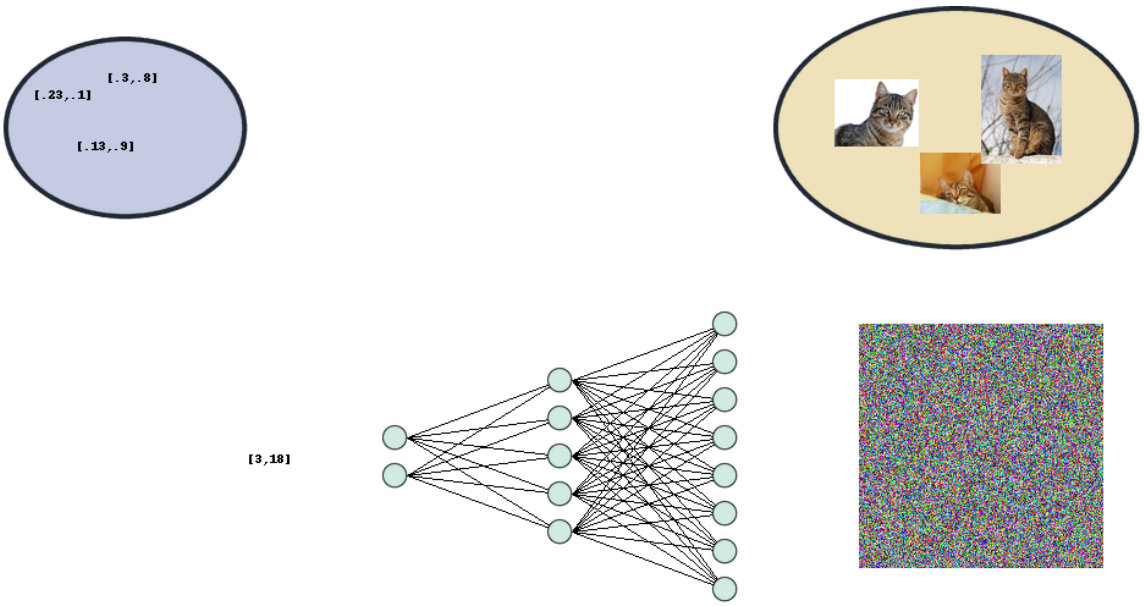


Figure 13.1: Feeding a random latent representation into the decoder is likely not to output meaningful results because we do not know which latent vectors output desired images. Cat images by Joaquim Alves Gaspar CC-SA.

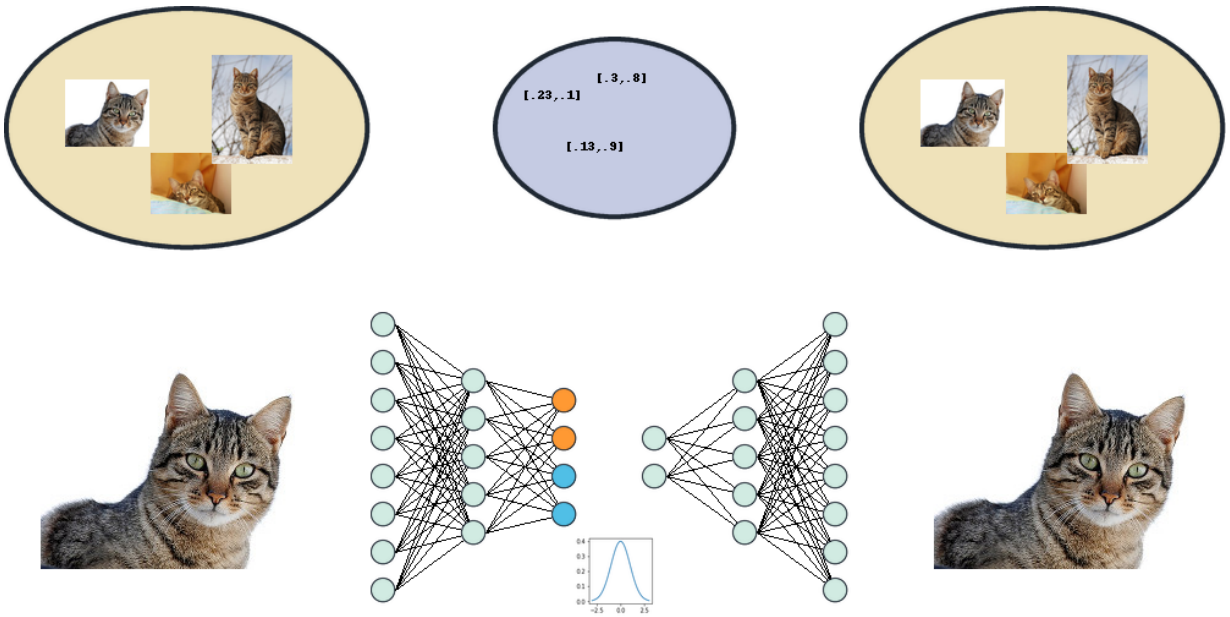


Figure 13.2: Variational autoencoder. The blue and orange output neurons of the encoder provide the means and variances of Gaussian distributions. The encoder then receives a vector sampled from those distributions. Cat images by Joaquim Alves Gaspar CC-SA.

we add the mean to the standard deviation multiplied by a random number drawn from a Gaussian distribution of mean 0 and variance of 1. This is illustrated in Figure 13.3. We cannot propagate the error by differentiating through this branch supplying the random number but that is not a problem, since there are no trainable parameters in that part of the graph. And the remainder of the graph is connected by sum and product operations, so there is no problem on that part.

So now we have an autoencoder that learns parameters of a probability distribution given each example. If we draw representations from such a distribution, we should obtain similar examples in the output since the decoder is trained to output such images from samples drawn from that distribution.

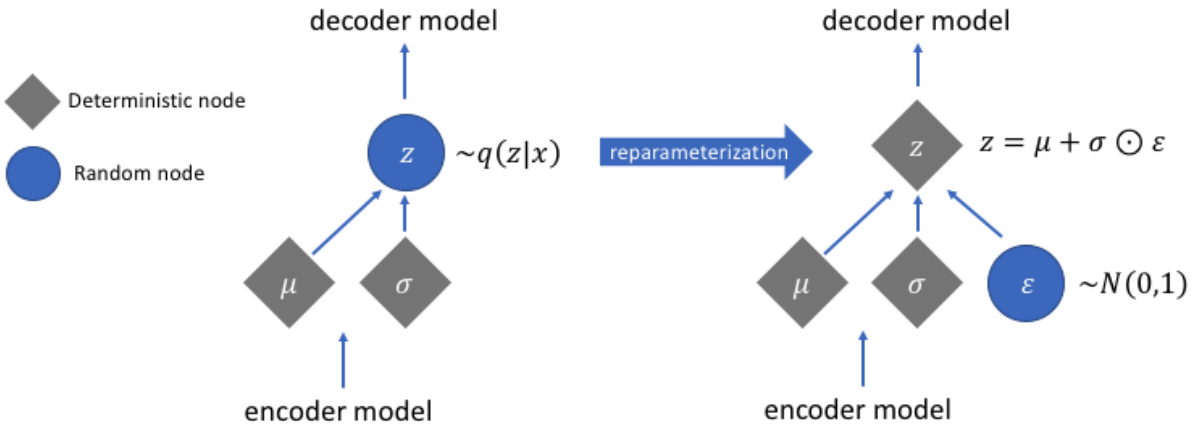


Figure 13.3: Reparametrization trick to allow backpropagation across the random sampling step connecting the encoder to the decoder in a variational autoencoder. Image by Jeremy Jordan.

What is left to do is to regularize the autoencoder so that these parameters do not vary widely and produce a smoother manifold. We can do this by adding a regularization term for the divergence between the parameters obtained and a multivariate Gaussian distribution for which the all the elements of the covariance matrix are 0 except in the diagonal, where they are 1. This is a multivariate Gaussian distribution with a variance of 1 in all dimensions and with no correlation between the different dimensions. So the parameters for the multivariate Gaussian for each example would be these:

$$p(z \mid x) = N(z|\mu(x), \Sigma(x)) \qquad \Sigma(x) = \text{diag}(\sigma_1^2, \sigma_2^2, ...)$$

and we add the Kullback-Leibler divergence between these distributions and the target multivariate Gaussian distribution of mean 0 and variance 1 in all dimensions. Thus the loss function will include the reconstruction error (cross-entropy of the encoder output and the input) plus the KL divergence:

$$l(\theta; \phi; x, z) = -E_{q_\phi(z|x)} \left[\log p_\theta(x \mid z)\right] + D_{KL} \left(q_\phi(z \mid x) \,||\, p(z)\right)$$

which for this particular target distribution of a multivariate Gaussian distribution of mean 0 and variance 1 in all dimensions, would be given by the following expression:

$$D_{KL} = -\frac{1}{2}\left(\sum_i (log\ \sigma_i^2 + 1) + \sum_i log\ \sigma_i^2 + \sum_i \mu_i^2\right)$$

Implementing a Variational Autoencoder[1]

To implement a variational autoencoder we will create 3 models: the encoder, the sampler and the decoder. Then we will chain them together to form the autoencoder, but this way we will also be able to use the decoder to generate new examples and the encoder if we want the parameters for the probability distribution of some example. The decoder will have a vector of size 2 for the latent space, so we can represent the manifold in two dimensions. Thus, the encoder will output 4 values, the variance and mean for each distribution. In fact, the encoder will output the logarithm of the variance. This makes it simpler to force the variance to be a positive number greater than zero.

[1] Adapted from: https://keras.io/examples/generative/vae/

We will use this regularization:

$$D_{KL} = -\frac{1}{2}\left(\sum_i (log\ \sigma_i^2 + 1) + \sum_i log\ \sigma_i^2 + \sum_i \mu_i^2\right)$$

First we import the library functions, the MNIST data set and create the constants with the input and encoder dimensions:

```
from tensorflow.keras.datasets import mnist
import numpy as np
import matplotlib.pyplot as plt
from tensorflow.keras.optimizers import Adam,SGD
from tensorflow.keras.models import Model
from tensorflow.keras.layers import Reshape,Input,BatchNormalization,Conv2D, \
                                    MaxPooling2D,Activation, Flatten, Dropout, \
                                    Dense, UpSampling2D
from tensorflow.keras.losses import binary_crossentropy
from tensorflow.keras import backend as K

ORIGINAL_DIM = 28*28
LATENT_DIM = 2
```

Now we create the encoder graph. The `last_conv` variable is just a convenient way of keeping track of the dimensions of the last convolution layer, so that we can then create a decoder that is symmetrical to the encoder.

```
def encoder_graph(filters, dense_neurons):
    inputs = Input(shape=(28,28,1),name='inputs')
    x = inputs
    last_conv_w = 28
    for filt in filters:
        x = Conv2D(filt, (3, 3), padding="same", )(x)
        x = Activation("relu")(x)
        x = BatchNormalization(axis=-1)(x)
        x = Conv2D(filt, (3, 3), padding="same", )(x)
        x = Activation("relu")(x)
        x = BatchNormalization(axis=-1)(x)
        x = MaxPooling2D(pool_size=(2, 2))(x)
        last_conv_w = last_conv_w // 2

    x = Flatten()(x)
    x = Dense(dense_neurons)(x)
    x = Activation("relu")(x)
    x = BatchNormalization()(x)
    z_mean = Dense(LATENT_DIM,name='means')(x)
    z_log_var = Dense(LATENT_DIM,name='log_sigma')(x)
    return inputs,z_mean,z_log_var,last_conv_w
```

This function returns the input tensor, the means and the logarithms of the variances for the two Gaussian distributions and the size of the last convolution layer.

Now we create the sampler graph, using the reparametrization trick:

```
def sampler_graph(epsilon_sigma):
    means = Input(shape=(LATENT_DIM,))
```

```
39        log_vars = Input(shape=(LATENT_DIM,))
40        epsilon = K.random_normal(shape=(K.shape(means)[0], LATENT_DIM),
41                                  mean=0., stddev=epsilon_sigma)
42        z_sample = means + K.exp(0.5*log_vars) * epsilon
43        return means, log_vars, z_sample
```

In theory, the sampler should multiply the standard deviation by a random number drawn from a $N(0, 1)$ distribution. However, this will cause too much randomness and makes it harder to train the decoder. So we use an `epsilon_sigma` global variable to define a smaller standard deviation for the noise function, such as 0.1 for example, which makes it easier to train the network.

Finally, we create the decoder graph, using the width of the last convolution layer of the encoder to create the first convolution layer on the decoder.

```
45    def decoder_graph(filters, dense_neurons, last_conv_w):
46        decoder_inputs = Input(shape=(LATENT_DIM,))
47        x = Dense(dense_neurons)(decoder_inputs)
48        x = Activation("relu")(x)
49        x = BatchNormalization(axis=-1)(x)
50        x = Dense(last_conv_w*last_conv_w*filters[-1])(x)
51        x = Reshape((last_conv_w,last_conv_w,filters[-1]))(x)
52        x = Activation("relu")(x)
53        for filt in filters[::-1]:
54            x = UpSampling2D()(x)
55            x = Conv2D(filt, (3, 3), padding="same")(x)
56            x = Activation("relu")(x)
57            x = BatchNormalization(axis=-1)(x)
58            x = Conv2D(filt, (3, 3), padding="same")(x)
59            x = Activation("relu")(x)
60            x = BatchNormalization(axis=-1)(x)
61        x = Conv2D(1, (3, 3), padding="same")(x)
62        decoder_outputs = Activation("sigmoid")(x)
63        return decoder_inputs,decoder_outputs
```

We also need the loss function, which includes the binary cross entropy for the reconstruction term and the Kullback-Leibler divergence to a sum of independent $N(0, 1)$ Gaussian distributions as the regularization term:

$$D_{KL} = -\frac{1}{2}\left(\sum_i (log\ \sigma_i^2 + 1) + \sum_i log\ \sigma_i^2 + \sum_i \mu_i^2\right)$$

```
65    def vae_loss(inputs, outputs, z_mean, z_log_var):
66        reconstruction = binary_crossentropy(inputs, outputs)
67        kl_loss = 1 + z_log_var - K.square(z_mean) - K.exp(z_log_var)
68        kl_loss = -0.5 * K.sum(kl_loss, axis=-1)
69        return K.mean(reconstruction,axis=[-1,-2])+ 0.1*kl_loss
```

Note that the regularization term must be calibrated. If it is too large we loose reconstruction accuracy but if it is too small the manifold will not be as well-behaved.

Now we bring everything together to create our models:

```
72    def create_models(filters,dense_neurons, epsilon_sigma):
73        inputs,z_mean,z_log_var,conv_width = encoder_graph(filters,dense_neurons)
```

```
74        input_means, input_vars, z_sample = sampler_graph(epsilon_sigma)
75        decoder_inputs,decoder_outputs = decoder_graph(filters,dense_neurons,conv_width)
76
77        encoder = Model(inputs= inputs,outputs = [z_mean,z_log_var])
78        sampler = Model(inputs = [input_means, input_vars], outputs = z_sample)
79        decoder = Model(inputs = decoder_inputs, outputs = decoder_outputs)
80
81        vae_outputs = decoder(sampler(encoder(inputs)))
82        vae = Model(inputs = inputs, outputs = vae_outputs)
83
84        loss = vae_loss(inputs,vae_outputs,z_mean, z_log_var)
85        return vae,encoder,decoder,loss
```

Now we create the models and train them. Note that the optimizer and loss function for the encoder and decoder are just placeholders. We will not train these models separately, so we can just put anything in these parameters to compile these models. The one that we are going to train is the `vae` model, which chains the three models we created. Since our loss function is not a standard loss function that simply compares inputs and outputs, we need to include it in our model using the `add_loss` method. Apart from this detail, training the `vae` is just like training any other network.

```
90    filters = [8,16]
91    dense_neurons = 16
92    epsilon_sigma = 1
93
94    vae,encoder,decoder,loss = create_models(filters,dense_neurons,epsilon_sigma)
95    encoder.compile(optimizer='adam', loss = 'mse')
96    decoder.compile(optimizer='adam', loss = 'mse')
97    vae.add_loss(loss)
98    vae.compile(optimizer='adam')
99
100   (x_train, y_train), (x_test, y_test) = mnist.load_data()
101   x_train = x_train.astype('float32').reshape((-1,28,28,1)) / 255
102   x_test = x_test.astype('float32').reshape((-1,28,28,1)) / 255
103   vae.fit(x_train, x_train,
104           epochs=200,
105           batch_size=128,
106           validation_data=(x_test, x_test))
```

We can now visualize how the encoder organizes the test set. We can plot for each example the mean of each of the two Gaussian distributions produced by the encoder:

```
x_test_mean,x_test_std = encoder.predict(x_test)
plt.figure(figsize=(6, 6))
plt.scatter(x_test_mean[:, 0], x_test_mean[:, 1], c=y_test)
plt.colorbar()
plt.show()
```

We can also inspect the manifold by plotting the output of the decoder when we consider different values ranged around the origin.

```
n = 15  # figure with 15x15 digits
digit_size = 28
figure = np.zeros((digit_size * n, digit_size * n))
```

```
# We will sample n points within [-15, 15] standard deviations
grid_x = np.linspace(-15, 15, n)
grid_y = np.linspace(-15, 15, n)

for i, yi in enumerate(grid_x):
    for j, xi in enumerate(grid_y):
        z_sample = np.array([[xi, yi]])*epsilon_sigma
        x_decoded = decoder.predict(z_sample)
        digit = x_decoded[0].reshape(digit_size, digit_size)
        figure[i * digit_size: (i + 1) * digit_size,
               j * digit_size: (j + 1) * digit_size] = digit

plt.figure(figsize=(10, 10))
plt.imshow(figure)
plt.show()
```

Figure 13.4 illustrates these results.

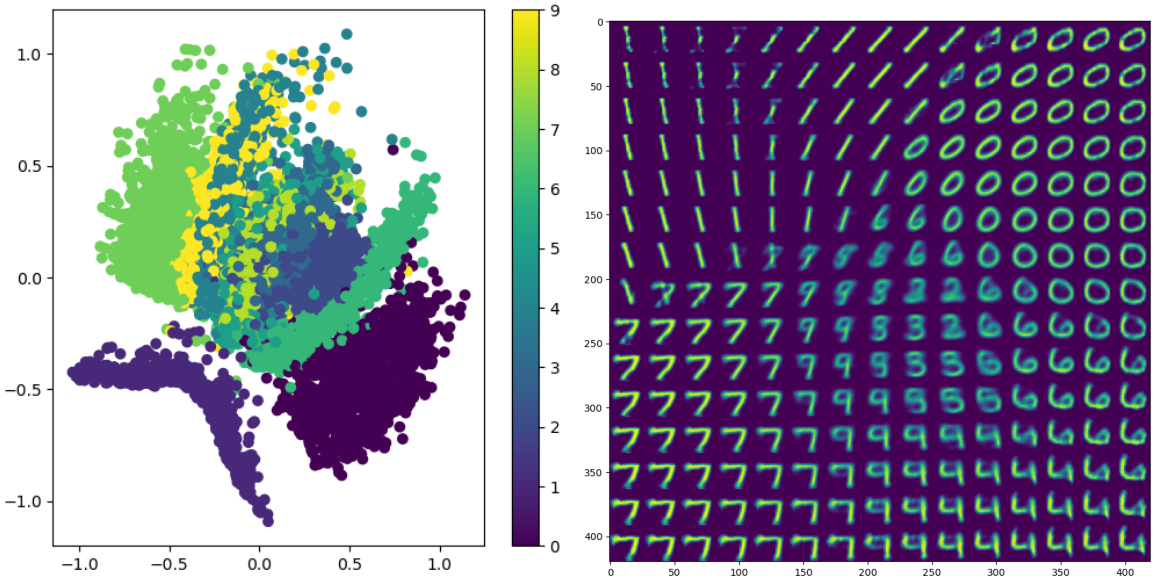


Figure 13.4: Results from the MNIST variational autoencoder, showing how the encoder organizes the test set (using the mean of the distributions) and the manifold around (0,0).

13.2 Generative Adversarial Networks

With the Variational Autoencoder we solved the problem of generating new examples by forcing the encoder to learn the parameters of probability distributions close to a mixture of independent Gaussian distributions $N(0, 1)$, which then allows us to generate useful examples because the manifold is bell behaved.

An alternative approach is to use a fixed distribution and then train a generator network to map from that distribution to meaningful outputs. The way we do this is using an auxiliary network that learns to distinguish between the real examples (in the training set) and the artificial examples generated by the generative network. This training of the generative model in parallel with the discriminator model is what gives the name of Generative Adversarial Network (GAN) to this approach.

For example, we can train a network to generate handwritten digits from a random vector of 100 numbers drawn from a $N(0, 1)$ distribution [2]. First, we define the generator model, which we do not need to compile since the generator will be compiled later integrated with the discriminator, for training. The generator receives a random vector as input, creates a 7x7 feature map with 128 filters using a dense layer followed by a reshape, and then upsampling and convolutions operations until it creates the final map of 28x28.

```
def define_generator(latent_dim):
    model = Sequential()
    n_nodes = 128 * 7 * 7
    model.add(Dense(n_nodes, input_dim=latent_dim))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Reshape((7, 7, 128)))
    model.add(UpSampling2D())
    model.add(Conv2D(128, (4,4), padding='same'))
    model.add(UpSampling2D())
    model.add(Conv2D(128, (4,4), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Conv2D(1, (7,7), activation='sigmoid', padding='same'))
    return model
```

Next we create the discriminator model. This is a convolution network for binary classification, with convolution layers and pooling followed by a final sigmoid neuron. This model we can compile with a binary cross-entropy loss function, since this model will be trained independently of the discriminator.

```
def define_discriminator(in_shape=(28,28,1)):
    model = Sequential()
    model.add(Conv2D(64, (3,3), strides=(2, 2),
                     padding='same', input_shape=in_shape))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(Conv2D(64, (3,3), strides=(2, 2), padding='same'))
    model.add(LeakyReLU(alpha=0.2))
    model.add(Dropout(0.4))
    model.add(Flatten())
    model.add(Dense(1, activation='sigmoid'))
    opt = Adam(lr=0.0002, beta_1=0.5)
    model.compile(loss='binary_crossentropy', optimizer=opt,
                  metrics=['accuracy'])
    return model
```

Now we can define the full GAN. Note that when we compiled the discriminator in the previous function, it is fully trainable. Now, we set the `trainable` flag to false when including the discriminator in the GAN. This serves to freeze all layers in this model when the GAN is compiled, leaving only the generator model trainable inside the GAN. Note that once again we use the binary cross-entropy loss function because the output of the GAN is the output of the discriminator.

```
def define_gan(g_model, d_model):
    d_model.trainable = False
    model = Sequential()
```

[2] Based on a tutorial by Jason Brownlee, https://machinelearningmastery.com/how-to-develop-a-generative-adversarial-network-for-an-mnist-handwritten-digits-from-scratch-in-ker

```
        model.add(g_model)
        model.add(d_model)
        opt = Adam(lr=0.0002, beta_1=0.5)
        model.compile(loss='binary_crossentropy', optimizer=opt)
        return model
```

To train the GAN, we will alternate training the discriminator using sets of real examples and fake examples generated by the generator, and then train the generator by feeding random vectors into the GAN and requiring the discriminator, which is frozen at this stage, to classify generated images as real. To do this we can use some auxiliary functions, below, to draw random samples from the training set, generate random points to feed into the generator and generate synthetic images using the generator.

```
def generate_real_samples(dataset, n_samples):
    ix = np.random.randint(0, dataset.shape[0], n_samples)
    X = dataset[ix]
    y = np.ones((n_samples, 1))
    return X, y

def generate_latent_points(latent_dim, n_samples):
    x_input = np.random.randn(latent_dim * n_samples)
    x_input = x_input.reshape(n_samples, latent_dim)
    return x_input

def generate_fake_samples(g_model, latent_dim, n_samples):
    x_input = generate_latent_points(latent_dim, n_samples)
    X = g_model.predict(x_input)
    y = np.zeros((n_samples, 1))
    return X, y
```

Now we can implement the `train` function. For batch, first we create a batch of real and fake examples by obtaining half the batch from the training set and the other half from the generator. This batch is used to train the discriminator model to output a 1 for a real image and a 0 for a fake image. Then we create a batch of random vectors and train the complete GAN. Note that in the GAN the discriminator is frozen, so only the generator will be trained. And now the target output for the discriminator is 1 for every example, so the generator is being trained to generate images that fool the discriminator.

```
def train(g_model, d_model, gan_model, dataset,
          latent_dim, n_epochs=100, n_batch=256):
    bat_per_epo = int(dataset.shape[0] / n_batch)
    half_batch = int(n_batch / 2)
    for i in range(n_epochs):
        for j in range(bat_per_epo):
            X_real, y_real = generate_real_samples(dataset, half_batch)
            X_fake, y_fake = generate_fake_samples(g_model,
                                                   latent_dim,
                                                   half_batch)
            X, y = np.vstack((X_real, X_fake)), np.vstack((y_real, y_fake))
            d_loss, _ = d_model.train_on_batch(X, y)
            X_gan = generate_latent_points(latent_dim, n_batch)
            y_gan = np.ones((n_batch, 1))
            g_loss = gan_model.train_on_batch(X_gan, y_gan)
```

Now we can bring everything together and use the `train` function to train the GAN.

```
latent_dim = 100
d_model = define_discriminator()
g_model = define_generator(latent_dim)
gan_model = define_gan(g_model, d_model)
(trainX, _), (_, _) = load_data()
dataset = trainX.reshape((-1,28,28,1)).astype('float32')/255
train(g_model, d_model, gan_model, dataset, latent_dim)
```

Figure 13.5 shows the digit images generated by the GAN in the beginning and after 50 epochs of training.

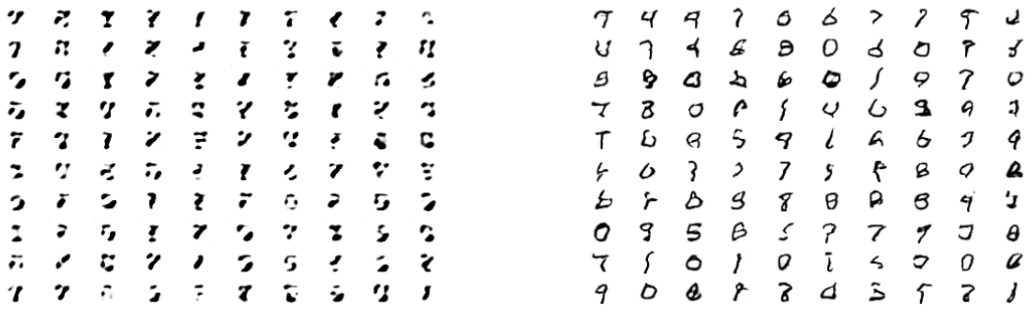


Figure 13.5: Output from the MNIST GAN after one epoch (left) and 50 epochs of training (right).

13.3 Further Reading

1. Goodfellow et.al, Deep learning, Sections 20.10.3 and 20.10.4[21]
2. (Optional) Brownlee, Generative Adversarial Networks with Python[9]

Chapter 14

Visualization of Image Classifiers

Occlusion maps. Saliency maps. Revision exercises.

14.1 Occlusion maps

One way of understanding which features in an image a deep classifier is using to classify a given example is to measure the effect that occluding parts of the image will have on the output of the classifier. The rationale behind this is that if we cover an important part of the image this should have a large impact on classification. Conversely, if the part of the image covered is not important, it should not affect classification. Figure 14.1 shows examples of this modification by replacing a region of the image with a gray square.

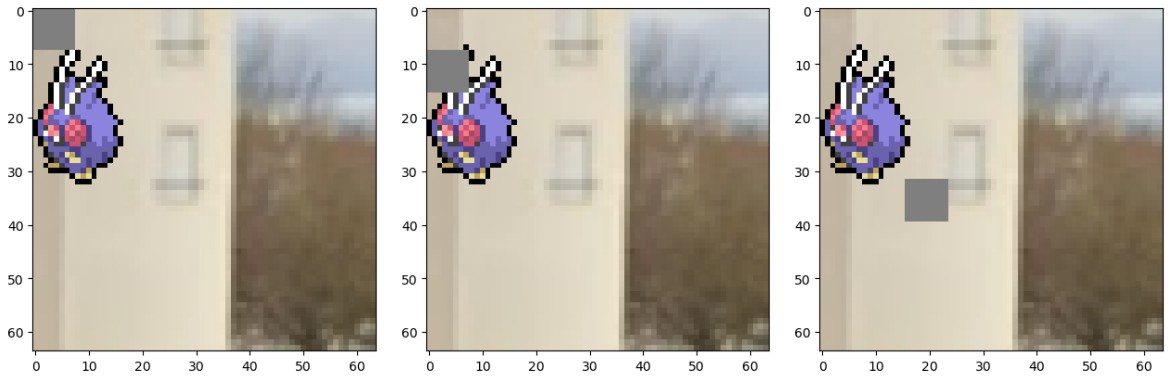


Figure 14.1: Examples of occlusions to generate an occlusion map. The image is modified by replacing a region with a gray square.

To implement this method, we can start by creating an iterator that will loop through all the regions that we want to occlude and return the modified images. This can be done in Python using the `yield` instruction instead of `return`

```
def iterate_occlusion(image, size=8):
    for row in range(0, image.shape[1]-size+1, size):
        for col in range(0, image.shape[2]-size+1, size):
            tmp = image.copy()
            tmp[0,row:row+size,col:col+size,:]=(0.5,0.5,0.5)
            yield row,col, tmp
```

Now we can compute the occlusion map given a model and an image by storing in each region of the occlusion map the impact of the occlusion on the output probability of the correct class for the image. We are assuming a softmax activation function for a multiclass classifier, but it would be easy to adapt this to a binary classifier.

```
8   def occlusion_map(image,model,true_class,size = 8):
9       occ_map = np.zeros(image.shape[1:-1])
10      for row,col, occluded in iterate_occlusion(image,size):
11          pred = model.predict(occluded)
12          print(row,col, pred[0,true_class])
13          occ_map[row:row+size,col:col+size] = pred[0,true_class]
14      return occ_map
```

Now we can load our model and plot the occlusion map for an image.

```
16  model = keras.models.load_model('multiclass.model')
17  ds = load_data()
18
19  img_ix = 2
20  img = ds['train_X'][img_ix:img_ix+1]
21  class_ix = np.argmax(ds['train_classes'][img_ix])
22
23  occ_map = occlusion_map(img,model,class_ix,size=8)
24  tmp = img[0].copy()
25  tmp[:,:,0] = 1-occ_map
26  plt.imshow(tmp)
```

Figure 14.2 show the occlusion maps for images in the test set of the Pokemon image data set.

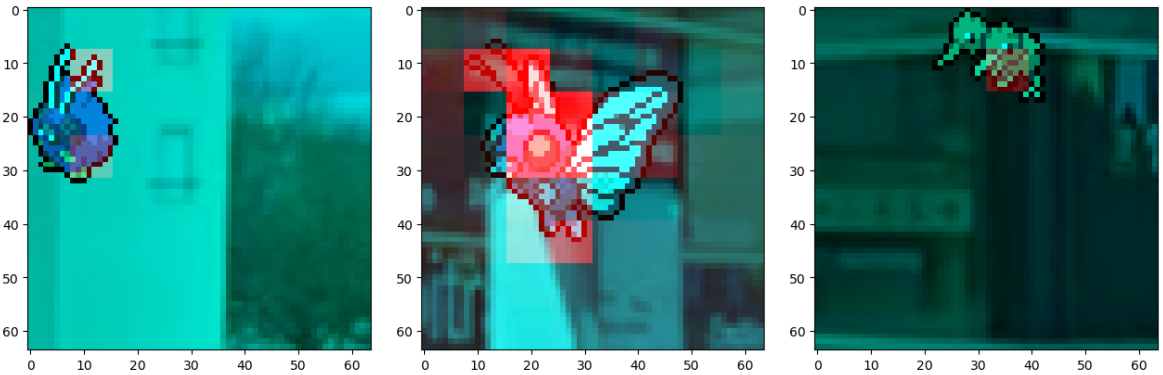


Figure 14.2: Examples of occlusion maps for different images of the test set for the Pokemon data set. The red regions show the more important regions of the image.

14.2 Saliency Maps

Another way of visualizing which parts of an image are more important for classification is to measure the derivative of the output probability for the correct class with respect to each pixel in the input image. To compute this we can use the `GradientTape` class, which we used before to train a network.

```
1   def saliency_map(model, image, class_idx):
2       image_tensor = tf.constant(image)
```

```
        with tf.GradientTape() as tape:
            tape.watch(image_tensor)
            predictions = model(image_tensor)
            loss = predictions[:, class_idx]
        gradient = tape.gradient(loss, image_tensor)
        gradient = tf.reduce_max(tf.math.abs(gradient), axis=-1)
        gradient = gradient.numpy()
        min_val, max_val = np.min(gradient), np.max(gradient)
        smap = (gradient - min_val) / (max_val - min_val + keras.backend.epsilon())
        return smap
```

Given a model, the image and the index of the correct class in the softmax output, we use the gradient tape to obtain the derivative of the predicted probability of the correct class, which is used here in place of a loss function. Note that since each pixel contains 3 channels, we take the maximum absolute derivative of the 3 channels. Then we rescale the saliency map.

Figure 14.3 show the saliency maps for images in the test set of the Pokemon image data set.

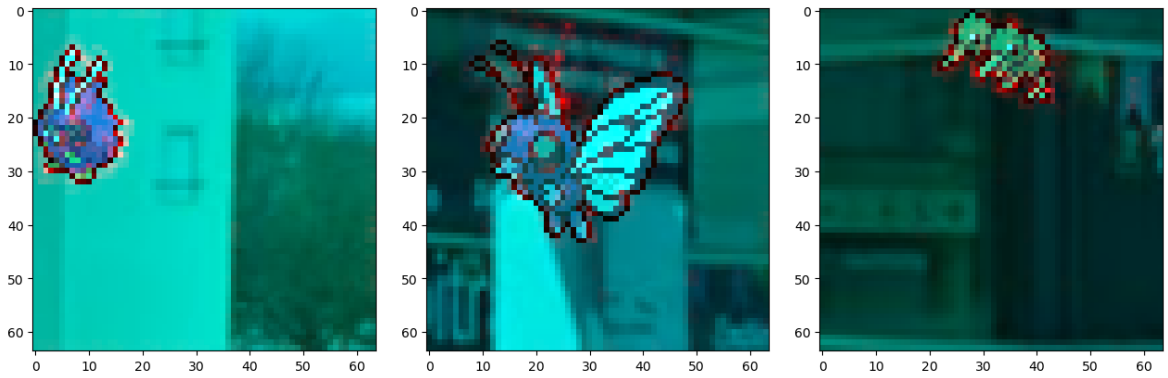


Figure 14.3: Examples of saliency maps for different images of the test set for the Pokemon data set. The red pixels have a greater impact in the output probability for the correct class.

14.3 Revision Exercises

Polynomial regression with Tensorflow

We can think of a polynomial regression of the form

$$y = w_0 + w_1 x + w_2 x^2 + w_3 x^3 + ...$$

as a linear regression using as input the expanded vector $[x, x^2, x^3, ...]$

Load the data in the `polydata.csv` file, standardize it and then expand it to some degree using this auxiliary function:

```
def expand_data(x,degree):
    expanded = np.zeros((len(x),degree))
    expanded[:,0] = x
    for d in range(2,degree+1):
        expanded[:,d-1] = x**d
    return expanded
```

Now use a single neuron to compute a linear regression using the expanded data and then plot the result. Use only basic Tensorflow and not Keras. For a degree of 3 the result should look something like shown in Figure 14.4:

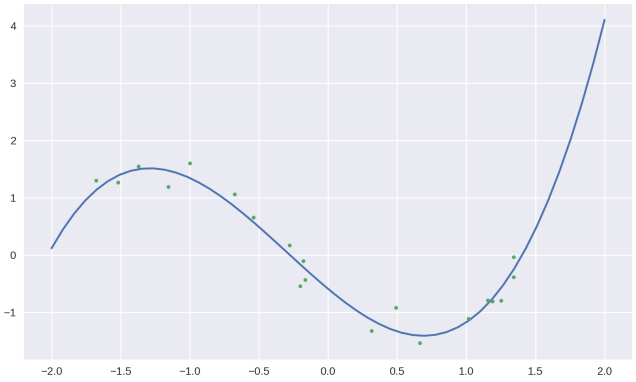


Figure 14.4: Example of a polynomial regression of degree 3 on the given data.

- What is the activation function of this neuron? Why?
- What is the shape of the weights matrix for the neurons? Does this matrix include the w_0 parameter in the equation above?
- What loss function did you use? What is the justification for using this loss function?

Distinguish odd and even digits in MNIST

Using Keras, load the MNIST dataset and prepare labels for distinguishing odd and even digits (note: In Python you can use the `%` operator to get the remainder of the division between two integers). Use a convolution net for classification.

- Which activation function did you use in the hidden layers? Why?
- How many neurons and which activation function did you use in the output of your network? Why?
- Which loss function did you use in this case?

GAN and Generating Pokemon

Answer the following questions:

- In an GAN, what is the input to the generative model?
- What is the role of the discriminative model?
- Why is the discriminative model frozen when training the generative model?

Optional: Adapt the Generative Adversarial Network code shown in Chapter 13 to train a GAN to generate images similar to those in the Pokemon dataset. Note that this is an optional exercise because it may take a large number of epochs to train an adequate network. Do not worry if your results are not good; just make sure you understand the architecture well enough to adapt it to the Pokemon dataset.

Chapter 15

Recurrent Networks

Introduction to recurrent networks. Long Short Term Memory networks.

(Note: this chapter is not yet complete...)

15.1 Recurrent Networks

To understand recurrent networks we can start with two concepts we have seen so far. The first one is the idea of stacking consecutive transformations, which we did in deep neural networks by chaining layers, with each layer feeding its output as input to the next layer. The other idea is parameter sharing, which we saw in convolution layers. Parameter sharing allows us to greatly simplify a model by reusing the same parameter in different places. In the case of convolution layers we reuse the kernel parameters over the whole input map.

In convolution layers, parameter sharing allows us to scan the input maps to find patterns that can occur anywhere without having to train the network to identify the correct patterns at each location, as would happen with a dense layer. However, in a convolution layer the kernel is only responding to each location individually and independently of other locations. This works well for data like images but it may not be ideal for sequential data or time series where it is useful to keep track of previous patterns.

In the case of recurrent networks, we can imagine that we are sharing parameters between different layers of a network. This conception of a recurrent network as a sequence of layers with shared parameters is equivalent to thinking of the network as feeding into itself over different time steps. Figure 18.1 illustrates this, showing the recurrent network and the same network "unfolded" into a feed-forward network with shared parameters.

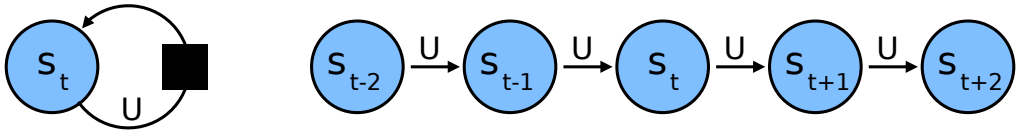


Figure 15.1: Schematic representation of a simple recurrent network. We can think of a RNN as a multilayer network with shared parameters across the different layers. In this case, the matrix V determines how the state evolves over time. The black square on the recurrent representation indicates a delay of one time step.

If we think of the state of the network at time t as represented by s_t in Figure 18.1, we can define it recursively by:

$$s_t = f(U, s_{t-1})$$

where U is the parameter matrix used to compute the next state. If we "unfold" this recursive definition over 2 steps we obtain:

$$s_t = f(U, f(U, f(U, V, x_{t-2})))$$

So, basically, we can think of a recurrent network both as a single layer that receives as input the state in the previous time step or as an stack of layers sharing parameters that transform the state of the previous layer into the current state.

Using this basic idea, we can think of more complex networks. For example, we can think of the input to the network consisting of a sequence of vectors and each layer of the unfolded network receiving part of the input. In this case, each layer will receive one of these vectors and also some information, from the previous layer, encoding aspects of the sequence received so far. For problems like time series analysis or natural language processing this can be very useful because, on the one hand, we need the network to recognize some patterns in different positions, which is easier to do by using the same weights on all these layers, for all regions of the input. But, on the other hand, we also want the network to keep track of the history of the sequence so far when looking for each pattern. For example, Figure 15.2 shows a recurrent network that receives a sequence of vectors x_t and the hidden state h_t of the network is a function of both the input at that time step and the state at the previous time step.

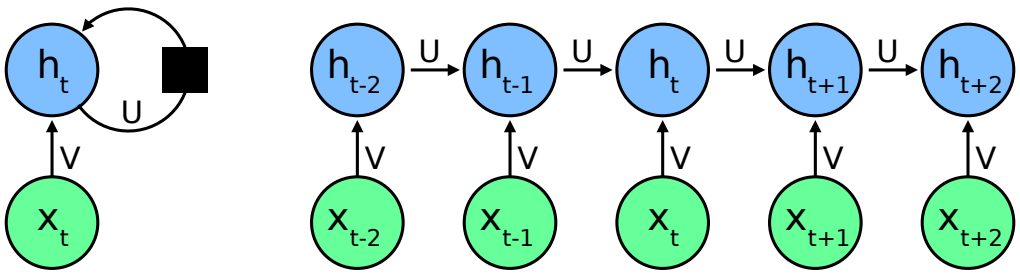


Figure 15.2: Schematic representation of a recurrent network receiving a sequence of inputs. Matrices U and V determine respectively how the hidden state depends on the input and the previous hidden state of the network.

Our network can also output values along the sequence and we can train it with supervised learning if we have a loss function to compare the output of the network with the target values. There are many other combinations we can think of, such as connecting the output to the hidden state of the next time step, or only having a single output at the end of the sequence. Figure 15.3 illustrates two examples. If there is a feedback loop from the output to the hidden state in the next time step, we can train the network with *teacher forcing* which consists in replacing the output at each time step with the target value (ground truth) of y_t. In this case, if the model also lacks hidden to hidden connections, as the one illustrated on the right panel of Figure 15.3, training may be done in parallel for examples in the same batch and we do not need to propagate gradients through time, since for each activation we will only use the input value for that time step and the target y_t. However, if there are hidden to hidden connections then we need to use backpropagation through time.

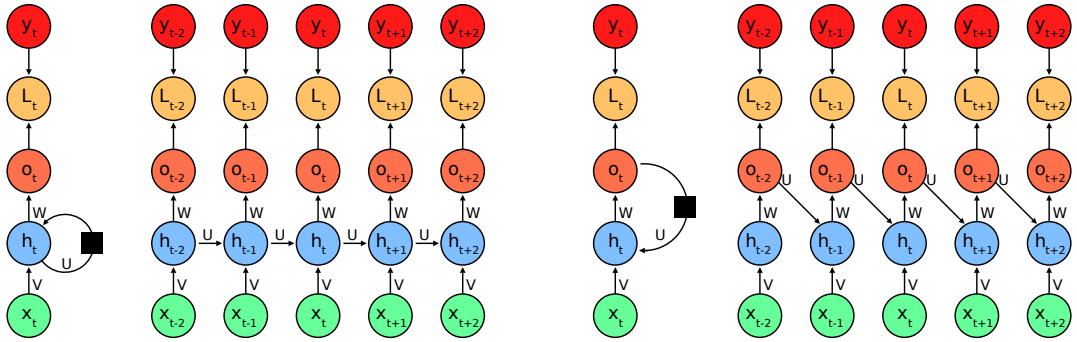


Figure 15.3: Schematic representation of a recurrent network receiving a sequence of inputs and providing one output per time step with the hidden state dependent on the previous hidden state, and a variant where the hidden state depends on the output of the previous time step. Matrices U, V and W are the trainable parameters optimized by minimizing the loss function L.

15.2 Backpropagation through time (BPTT)

To train a recurrent network we need to propagate the gradient of the loss function through all computations. Since the state of the network is changing over the different time steps and we need to pass through several time steps before backpropagating the error, this would create a problem with which values to use for each step. But, as we saw, we can unfold the recurrent network over a number of steps and train it as if it was a multilayer network with shared parameters. So to train over k time steps, we can use k copies of the recurrent network as the unfolded layers. We start with an initial state of zero, use the first input in the sequence, activate the network and then feed its output to the next copy, activating that layer and moving on until the end of the unfolded computation graph. Then we backpropagate the gradients through all these layers as usual, with the difference that the parameters in the diferent layers are all copies of the same set of parameters.

One problem with this approach is how large k should be. If we simply unfold the network over all the training sequence and then propagate the gradients through the whole unfolded network, this is not only computationally expensive but tends to aggravate problems like the vanishing or exploding gradients problems because these updates are applied to the same parameters. So, in practice, the algorithm used is truncated backpropagation through time, or TBPTT, in which the sequence of k_1 steps is fed into the unfolded network but then backpropagation is only performed for k_2 steps, with $k_2 < k_1$. As we will see in the next chapter, Keras has some basic mechanisms to adjust these parameters.

Chapter 16

Implementing RNN in Keras

Using LSTM networks for time series prediction

Note: This tutorial is based on a tutorial by Jason Brownlee, available on https://machinelearningmastery.com/time-series-prediction-lstm-recurrent-neural-networks-python-keras/.

16.1 Univariate time series prediction with LSTM

We will be using the Box and Jenkins international airline passenger dataset, with monthly passengers (in thousands) from 1949 to 1960. Since we are working with a time series and we are trying to predict future values form previous values, our test set should be the last part of the time series. This to simulate training the model with data up to some point and then using that trained network in future events. Figure airlineset shows the data we will use and the split into training and testing.

Here is the code for loading the data (using Pandas) and splitting the last third for testing:

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  import pandas as pd
4  from tensorflow.keras.models import Sequential
5  from tensorflow.keras.layers import Dense,LSTM
6  from sklearn.preprocessing import MinMaxScaler
7  from sklearn.metrics import mean_squared_error
8
9  df = pd.read_csv('passengers.csv')
10 dataset = df['Passengers'].values.astype('float32')
11
12 scaler = MinMaxScaler(feature_range=(0, 1))
13 dataset = scaler.fit_transform(dataset.reshape(-1,1)).reshape((-1,))
14
15 train_size = int(len(dataset) * 0.67)
16 test_size = len(dataset) - train_size
17 train, test = dataset[0:train_size], dataset[train_size:len(dataset)]
```

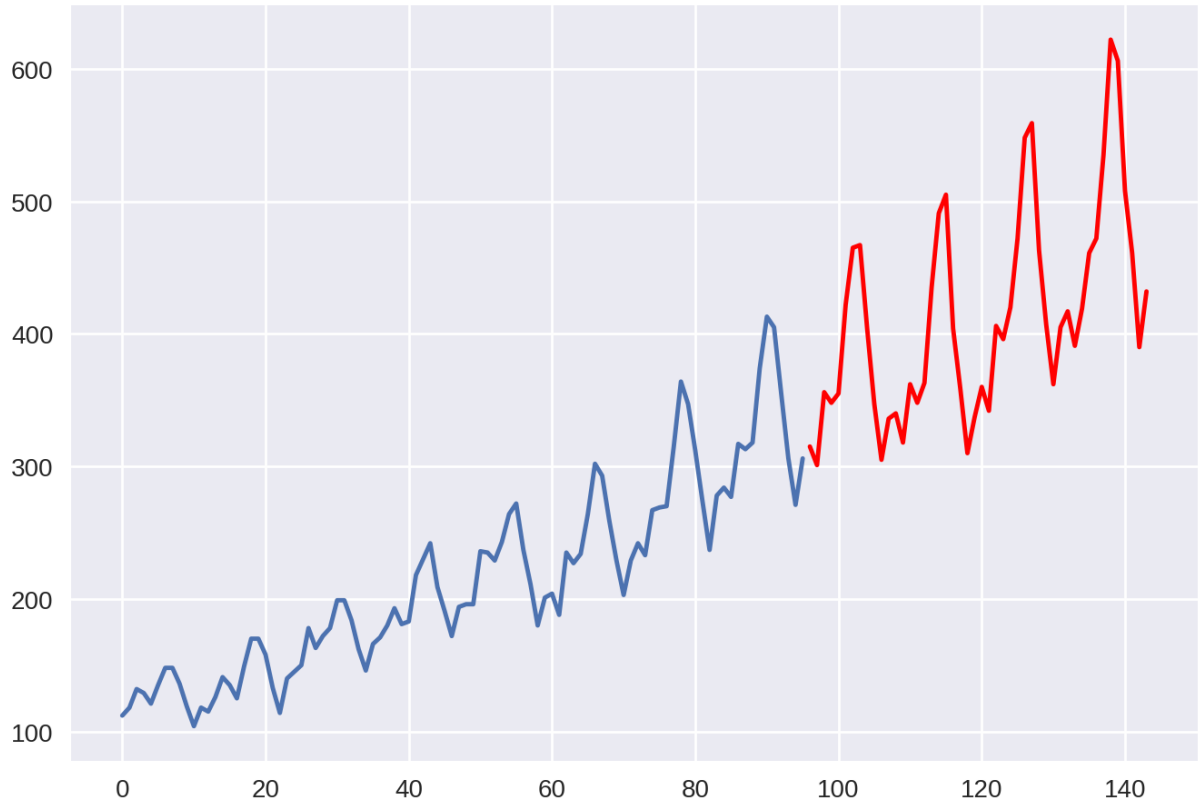


Figure 16.1: International airline passenger data, train (blue) and test (red) sets.

Note the use of the `MinMaxScaler` to rescale the data to values ranging from 0 through 1. The need for the `reshape` method comes from the convention in Scikit-learn that data sets are two-dimensional, with samples in rows and features in columns. In this case we have only one feature, the number of passengers, but we need to shape the matrix appropriately to use the rescaler. After rescaling, we reshape it again into a vector.

To train the network we will need to create a dataset that includes, for each window of past values, the next value we want to predict. For example, if we want to use the last 3 months to predict the number of passengers in the next month, we will need to have an array X (the features) with the values of the last 3 months and the Y, the ground truth of the prediction, with the value of the next month. The function below serves to create these data sets, both for training and testing:

```
19 def create_dataset(dataset, look_back=1):
20     dataX, dataY = [], []
21     for i in range(len(dataset)-look_back-1):
22         a = dataset[i:(i+look_back)]
23         dataX.append(a)
24         dataY.append(dataset[i + look_back])
25     return np.array(dataX), np.array(dataY)
```

Now we need to create the X, Y pairs and shape the X matrix appropriately, because the LSTM layers need to receive the time steps and the features in different dimensions: (samples, time, features). So we need to reshape these matrices into 3D arrays.

```
28 look_back = 3
29 trainX, trainY = create_dataset(train, look_back)
30 testX, testY = create_dataset(test, look_back)
```

```
31
32 trainX = np.reshape(trainX, (trainX.shape[0], trainX.shape[1], 1))
33 testX = np.reshape(testX, (testX.shape[0], testX.shape[1], 1))
```

Now we create the model. To use truncated backpropagation through time, we will use a stateful LSTM layer. This means that Keras will retain the cell states from one batch to the other. Since we are using 3 time steps, then backpropagation will only be applied over 3 time steps. But as the cells retain their states, the forward pass continues from one batch to the other and will effectively be as long as the whole time series.

When we create stateful recurrent cells we need to tell Keras the batch size, because one state will be retained for each element in the batch. Also, if the batch size is larger than 1 and we want to train on several time series in parallel, we must be careful not to change the order of the time series in the batches because there is one different cell state for each element in the batch. In this case we only have one time series, so we will use a batch size of one and that will not be a problem.

```
35 batch_size = 1
36 model = Sequential()
37 model.add(LSTM(4, batch_input_shape=(batch_size, look_back, 1), stateful=True))
38 model.add(Dense(1))
```

If we check the model architecture, we can see that the output of the LSTM layer with 4 LSTM cells is a vector of 4 values. This is because by default the LSTM layer outputs only the final output at the end of all time steps:

```
Layer (type)                 Output Shape              Param #
=================================================================
lstm_4 (LSTM)                (1, 4)                    96
_________________________________________________________________
dense_3 (Dense)              (1, 1)                    5
=================================================================
```

The output neuron has a linear activation because this is a regression problem, and now we compile the model with a MSE loss function and train it. For that we need to take care to manage the cell states appropriately. Since the cell states are retained from one batch to the next, we can train a full epoch by passing through our time series as long as we do so in order, from past values to future values. So we must disable shuffling in the `fit` method. Also, at the end of the epoch we need to reset the cell states before restarting from the beginning.

```
41 model.compile(loss='mean_squared_error', optimizer='adam')
42 for i in range(100):
43     model.fit(trainX, trainY, epochs=1, batch_size=batch_size, verbose=2, shuffle=False)
44     model.reset_states()
```

Note that in this example we are creating samples for training using a sliding window with a width of 3 and a step of 1 (in function `create_dataset`). This is a simple approach and lets us extract more data points from the time series but has a problem. From one batch to the next we are actually going back in time, since the first batch will receive months (1, 2, 3), and then the second batch months (2, 3, 4). A more rigorous approach would be to slide the window in steps of three, so that the second batch would continue with months (4, 5, 6). Or, alternatively, split the data into 3 streams starting at different weeks and use a batch size of 3. For our particular example this is not necessary but it is important to

bear these details in mind when dealing with time series and recurrent networks, and I encourage you to experiment with these variants in this problem.

After training you can check the errors and plot the predictions. Here are some examples of how to do this. To predict using the training and test sets we use the `predict` method. To simulate what would happen when predicting the test set assuming we did not have access to previous values, we must reset the network states before predicting for those data points. Then we need to rescale the predictions back to the original scale. We can use the `scaler` object created previously, since it retains the scaling factors necessary for the operation.

Also note that in order to see the prediction error in the correct units you must compute the square root of the mean squared error, otherwise you are seeing the error in units of passenger squared which is not intuitive.

```
46 trainPredict = model.predict(trainX, batch_size=batch_size)
47 model.reset_states()
48 testPredict = model.predict(testX, batch_size=batch_size)
49 trainPredict = scaler.inverse_transform(trainPredict).reshape((-1,))
50 trainY = scaler.inverse_transform(trainY.reshape(-1,1)).reshape((-1,))
51 testPredict = scaler.inverse_transform(testPredict).reshape((-1,))
52 testY = scaler.inverse_transform(testY.reshape(-1,1)).reshape((-1,))
53 trainScore = mean_squared_error(trainY, trainPredict)**0.5
54 testScore = mean_squared_error(testY, testPredict)**0.5
```

To plot the predictions you need to shift the predicted values to the correct positions in the plotting axis. Remember that for each set of 3 months we are predicting the month immediately after these 3. Here is an example of how you can do this:

```
56 trainPredictPlot = np.empty_like(dataset)
57 trainPredictPlot[:] = np.nan
58 trainPredictPlot[look_back:len(trainPredict)+look_back] = trainPredict
59 testPredictPlot = np.empty_like(dataset)
60 testPredictPlot[:] = np.nan
61 testPredictPlot[len(trainPredict)+(look_back*2)+1:len(dataset)-1] = testPredict
62 plt.plot(scaler.inverse_transform(dataset.reshape(-1,1)))
63 plt.plot(trainPredictPlot)
64 plt.plot(testPredictPlot)
```

Chapter 17

Probabilistic Models

Graphical models. Sampling distributions. Restricted Boltzmann Machine

17.1 Graphical Models

Discriminative models are trained to estimate the conditional probability of some target prediction given the input features, $p(y \mid x)$. For generative models, we need some way of estimating the joint probability distribution of features and target, $p(y, x)$, so that we can sample from this distribution to generate new examples. With Variational Autoencoders and Generative Adversarial Networks we saw some ways of solving this problem, but another approach is to explicitly model the joint probability distribution of all variables. This allows us not only to sample de distribution of examples, and thus generate new examples, but also to solve problems like estimating the distribution of the data, denoising or filling in missing values.

Probabilistic graphical models provide a useful notation to model relations and dependencies between random variables, observations and parameters. The graphical notation makes it easy to represent and understand relations, as well as deriving the equations necessary to compute probabilities. This is useful because, without some assumption about relations between variables, storing the joint probability distribution would be unfeasible in any but the simplest problems. Suppose we have n categorical variables with each having k possible values, to store individual probability values for all combinations we would need a table with k^n entries. This would not only require too much storage but also too much computation time in any realistic problem, and would be very hard to find the appropriate probabilities when training such a model.

Bayesian networks

A Bayesian Network is a directed acyclic graph where each node corresponds to a variable and each directed edge represents that a variable affects the probability distribution of another variable, indicating conditional dependencies. For example, Figure 17.1 shows a Bayesian Network with 9 nodes. The joint probability distribution can be computed by multiplying al the conditional probability distributions indicated in the graph, and using the marginal probability distributions for nodes without parents. In this case:

$$p(x_1, ..., x_9) = p(x_1)p(x_2)p(x_3 \mid x_1, x_2)p(x_4 \mid x_1)p(x_5 \mid x_2)p(x_6 \mid x_5)p(x_7 \mid x_6, x_4)p(x_8 \mid x_3)p(x_9 \mid x_7, x_8)$$

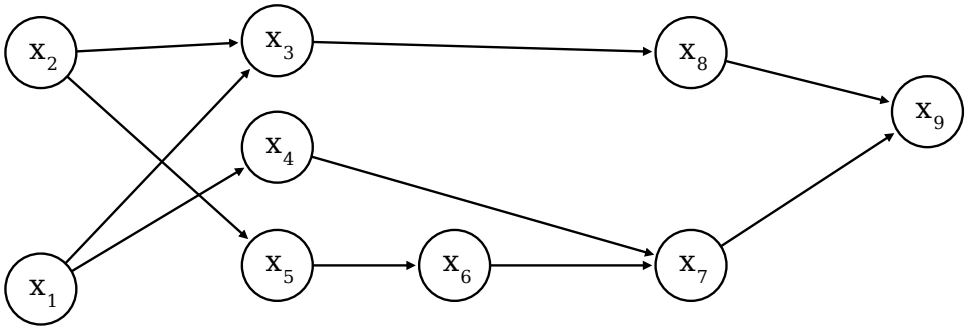


Figure 17.1: Bayesian Network, example

Directed models are especially useful when we can specify dependency relations between variables. For example, we know that diseases cause symptoms so we can specify that the probability of having some symptoms is conditioned on having certain diseases. But in some cases we cannot specify these dependencies, and then undirected models may be more appropriate.

Markov Random Fields

Markov random fields (MRF) are undirected graphs where the nodes represent random variables and the edges represent variable interactions. However, since the edges are undirected, they do not specify that one variable depends on the other, merely that they are not independent. The joint probability distribution of all variables in a MRF can be determined as a function of the *cliques* in the graph (a *clique* is a set of nodes that are all directly connected). In fact, we can consider simply the maximal cliques, which are the cliques to which no nodes can be added while still remaining a clique. Figure 17.2 shows an example of a MRF model indicating the three maximal cliques.

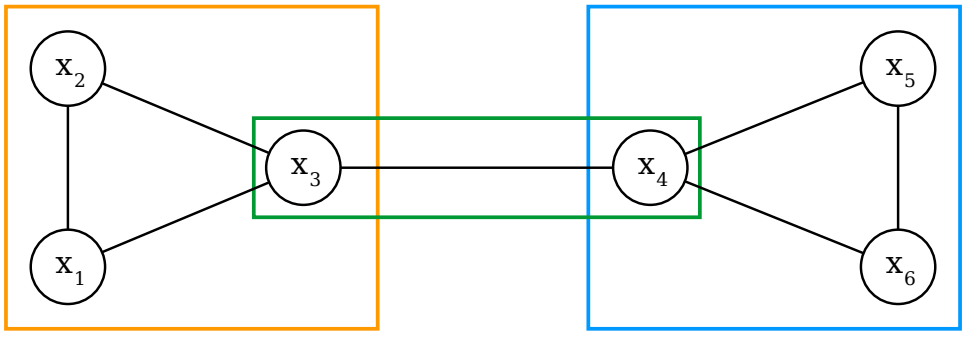


Figure 17.2: A Markov random field and its three maximal cliques

For each clique C in our MRF (or each maximal clique, if we do not want to consider all cliques) we can consider a nonnegative set of factors $\phi(C)$ that measure the tendency for the variables in C to have each combination of values. Multiplying for all cliques, we get an unnormalized probability distribution, since it is nonnegative but it may be larger than 1:

$$\tilde{p}(x) = \prod_C \phi(C)$$

To normalise $\tilde{p}(x)$ we need to divide by the *partition function* Z, which is the integral of the unnormalized probability over all possible values of x:

$$p(x) = \frac{1}{Z} \prod_C \phi(C) \qquad Z = \int \tilde{p}(x) dx$$

Unlike directed models, which specify the probability distributions, undirected models specify these propensity functions that can be converted into probabilities if the functions used allow us to compute Z, which may not always be possible because the integral may not converge. However, by adequately choosing the architecture and the ϕ functions we can use these models in practice.

One solution, taking inspiration from statistical thermodynamics (where the partition function also comes from) is to use an analogous of an energy function and the Boltzmann distribution, which in physics relates the energy of micro-states in a system to the probability of the system being in each state. In energy based models, the unnormalized probability of state x is e raised to minus the energy of state x, which is the Boltzmann distribution:

$$\tilde{p}(x) = e^{-E(x)}$$

Since this guarantees a value greater than zero for $\tilde{p}(x)$ whatever the energy function we use, it gives us a broad choice of functions for our models. Furthermore, since $e^a e^b = e^{a+b}$, this allows us to factor the unnormalized probability distribution over the set of cliques in a RMF by simply adding the corresponding energies, which can be even more simplified if we compute $log\tilde{p}(x)$ instead of $\tilde{p}(x)$. These MRF using energy functions and the Boltzmann distribution are called Boltzmann machines.

17.2 Sampling

One of the goals of using graphical models is to be able to sample examples from the distributions described by the model. With directed graphical models, like Bayes networks, we can use *ancestral sampling*. With this method we sort the variables from ancestors to descendants, so that variable x_i comes before any variable that has x_i as an ancestor, and then sample them in this order. This way, when we reach a variable x_j we already sampled all of its ancestors and so can sample its probability distribution conditioned on the values of its parents. This is a fast method when we can sample all the distributions but it only applies to directed models.

For undirected models, like Markov random fields, we can use methods like Gibbs sampling. Suppose we want a sample from a probability distribution $p(x_1, x_2, ..., x_n)$. In the first iteration we start with a random vector $(x_1^1, x_2^1, ..., x_n^1)$, which will not be drawn from the correct distribution. But in the second iteration we sample each variable conditioned on the values assigned to all other components in the previous iteration and repeat these steps. This will eventually converge to the correct distribution.

For a Markov random field, we need to sample the value of each variable conditioned only on the values of the variables to which it is directly connected in the graph.

17.3 Restricted Boltzmann Machine

A Restricted Boltzmann Machine (RBM) is a Markov random field that is also a bipartite graph, meaning that the nodes can be split into two sets such that all edges only connect one set to the other. In a RBM we can think of these two sets of variables as two layers. Furthermore, in an RBM one layer contains observable variables and the other hidden, or latent, variables. Although it is common for the layers to be fully connected in an RBM, this is not mandatory and there can be sparse connectivity. The important feature is that there are only connections between the two layers, of hidden and observable variables, and not within each layer, thus creating a bipartite graph. Although RBM are shallow models, they can be stacked to create Deep Boltzmann Machines or hybrid models like Deep Belief Networks, which include directed edges between the last hidden layer and the layer of observable variables. Figure **??** illustrates these architectures.

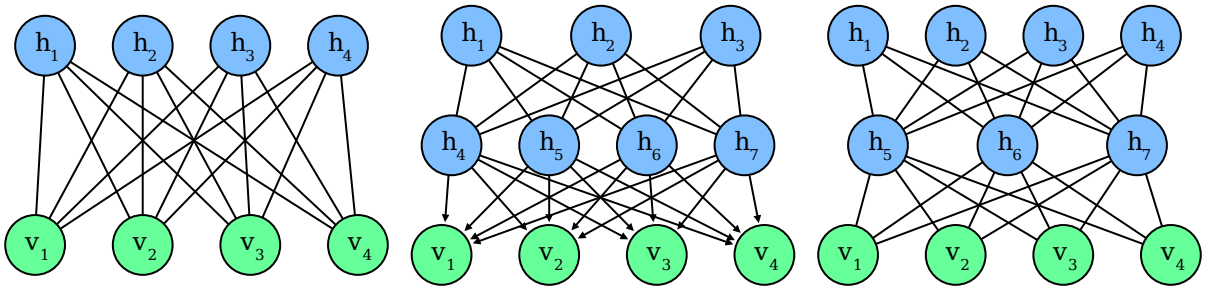


Figure 17.3: From left to right, examples of a Restricted Boltzmann Machine, Deep Belief Network and Deep Boltzmann Machine. Hidden (latent) variables are represented in blue, observable variables are represented in green.

Since the RBM is a Boltzmann machine, it uses an energy function and the Boltzmann distribution for the unnormalized probability distribution of each variable. Considering v to denote the set of visible units and h the set of hidden units, in a RBM the probability of some state $(\mathrm{v} = v, \mathrm{h} = h)$ is:

$$P(\mathrm{v} = v, \mathrm{h} = h) = \frac{1}{Z} e^{-E(v,h)} \qquad Z = \sum_v \sum_h e^{-E(v,h)}$$

with Z being the normalization constant given by the partition function. The energy is given by:

$$E(v,h) = -\sum_i b_i v_i - \sum_j c_j h_j - \sum_i \sum_j v_i w_{i,j} h_j \qquad E(v,h) = -b^\top v - c^\top h - v^\top W h$$

where b and c are the biases for the visible and hidden variables and W the weights matrix for the relations between the visible and hidden variables.

Since in an RBM there are only edges connecting hidden to visible units, we can factorize the computation of the conditional probabilities of $P(h \mid v)$ and $P(v \mid h)$. In each case, we can consider the other variables to be constant. So for $P(h \mid v)$, considering v constant:

$$
\begin{aligned}
P(h \mid v) = \frac{P(h, v)}{P(v)} &= \frac{1}{P(v)} \frac{1}{Z} e^{b^\top v + c^\top h + v^\top W h} \\
&= \frac{1}{Z'} e^{c^\top h + v^\top W h} \\
&= \frac{1}{Z'} \prod_{j=1}^{n_h} e^{c_j h_j + v^\top W_{:,j} h_j}
\end{aligned}
$$

Considering the simpler case of an RBM where each variable is binary, taking only values of 0 or 1, we can compute the probability $P(h_j = 1 \mid v)$:

$$
\begin{aligned}
P(h_j = 1 \mid v) &= \frac{\tilde{P}(h_j = 1 \mid v)}{\tilde{P}(h_j = 0 \mid v) + \tilde{P}(h_j = 1 \mid v)} \\
&= \frac{e^{c_j + v^\top W_{:,j}}}{e^0 + e^{c_j + v^\top W_{:,j}}} \\
&= \sigma(c_j + v^\top W_{:,j})
\end{aligned}
$$

Training the RBM

Since the probability distribution of variables x in a Boltzmann machine depends on the unnormalized distribution and the partition function:

$$
p(\mathrm{x}; \theta) = \frac{\tilde{p}(\mathrm{x}; \theta)}{Z(\theta)}
$$

if we want to maximize the likelihood maximizing the logarithm of $p(\mathrm{x}; \theta)$, we can decompose the gradient in two factors:

$$
\nabla_\theta \log p(\mathrm{x}; \theta) = \nabla_\theta \log \tilde{p}(\mathrm{x}; \theta) - \nabla_\theta \log Z(\theta)
$$

This is known as the decomposition of learning into the positive phase and the negative phase. When training on a batch of examples, we can think of the positive phase as adjusting the parameters to maximize the unnormalized probability for that batch of examples and the negative phase as trying to minimize the unnormalized probability of the model's states sampled from all possible states. This negative phase will tend to minimize "hallucinations" of the model, which would be states the model considers to have a high probability but do not correspond to the data.

This leads us to the contrastive divergence algorithm for training a Restricted Boltzmann Machine:

1. Start with a sample v, compute the probabilities of h and obtain a sample h.

2. Compute the positive gradient to update W as the outer product of v and the sampled h. Intuitively, this indicates how to change W to maximize the probability the model assigns to the data.

3. Now we need to sample the model states from the probability distribution $p(v, h)$. Since we cannot do this directly, we can use Gibbs sampling, iteratively generating samples v' and h' until convergence. Since we started this sequence with a data sample, we do not need many iterations.

4. Compute the negative gradient to update W as the outer product of sampled v' and sampled h'. Intuitively, this indicates how to change W to minimize the model's "halluciunations".

5. The update to the weight matrix W is the positive gradient minus the negative gradient (multiplied by some learning rate).

6. The updates for the biases for the visible and hidden variables are given by the differences, respectively, of $v - v'$ and $h - h'$. Intuitively, this helps adjust the biases so that the state probabilities are closer to what we observe in the data.

17.4 Implementing a RBM

This example is based on a demo by Gulli *et al.* [24]. We will do unsupervised training of a RBM using MNIST. We will start by importing the libraries, the MNIST data and setting the dimensions of our RBM:

```
8 import tensorflow as tf
9 import numpy as np
10 import matplotlib.pyplot as plt
11
12 (train_data, _), (test_data, _) = tf.keras.datasets.mnist.load_data()
13 train_data = train_data/np.float32(255)
14 train_data = np.reshape(train_data, (train_data.shape[0], 784))
15 test_data = test_data/np.float32(255)
16 test_data = np.reshape(test_data, (test_data.shape[0], 784))
17
18 learning_rate = 1.0
19 visible_size = train_data.shape[1]
20 hidden_size = 200
21 batchsize = 100
```

We will use 785 visible variables, corresponding to the pixels in the MNIST images (28 by 28) and 200 hidden variables. Now we define the functions that give us the unnormalized probabilities $\tilde{P}(h \mid v)$ and $\tilde{P}(v \mid h)$, based on the weights matrix `w` and the biases for the visible and hidden variables, `vb` and `hb`

```
23 def prob_h_given_v(visible, w, hb):
24     return tf.nn.sigmoid(tf.matmul(visible, w) + hb)
25
26 def prob_v_given_h(hidden, w, vb):
27     return tf.nn.sigmoid(tf.matmul(hidden, tf.transpose(w)) + vb)
```

We will also need to sample variable states, which can be 0 or 1, from the unnormalized probabilities:

```
29 def sample_prob(probs):
30     return tf.nn.relu(tf.sign(probs - tf.random.uniform(tf.shape(probs))))
```

Note that the `relu` call in this case is not an activation function. It is merely for convenience. The difference between the probability matrix and the uniform random numbers will give negative or positive values. The `sign` function converts these to values of -1 or 1 and then the `relu` simply turns the -1 values to 0 keeping the values of 1 intact.

Now we write a function for Gibbs sampling, which we will need for training and for obtaining samples from the model's distribution:

```
33 def gibbs_sampling(batch,k=1):
34     hidden_prob = prob_h_given_v(batch, weights,h_bias)
35     hidden_states_0 = sample_prob(hidden_prob)
36     original_hs = hidden_states_0
37     for _ in range(k):
38         visible_prob = prob_v_given_h(hidden_states_0, weights, v_bias)
39         visible_states = sample_prob(visible_prob)
40         hidden_prob = prob_h_given_v(visible_states, weights,h_bias)
41         hidden_states = sample_prob(hidden_prob)
42         hidden_states_0 = hidden_states
43     return original_hs,hidden_states,visible_states
```

We start from a batch of observations, obtain the hidden state probabilities given these observations and a sample from these hidden states. This original sample will be useful for the positive phase in training. Then we loop through k iterations reconstructing the visible and hidden states iteratively. We then return the original hidden states and the reconstructed hidden and visible states.

Now we create the training function. To keep the code simple we will assume that the weights and bias variables are global variables. This is not elegant but helps to avoid more sophisticated Python structures like classes.

```
45 def train(X, epochs=10):
46     global weights, h_bias, v_bias
47     loss = []
48     for epoch in range(epochs):
49         for start, end in zip(range(0, len(X), batchsize),range(batchsize,len(X), batchsize)):
50             batch = X[start:end]
51             ohs,hs,vs = gibbs_sampling(batch,k=2)
52             positive_grad = tf.matmul(tf.transpose(batch), ohs)
53             negative_grad = tf.matmul(tf.transpose(vs), hs)
54             b_size = tf.dtypes.cast(tf.shape(batch)[0],tf.float32)
55             dw = (positive_grad - negative_grad) / b_size
56             weights += learning_rate * dw
57             v_bias += learning_rate * tf.reduce_mean(batch - vs, 0)
58             h_bias += learning_rate * tf.reduce_mean(ohs - hs, 0)
59         err = tf.reduce_mean(tf.square(batch - vs))
60         print (f'Epoch: {epoch}, reconstruction error: {err}')
61         loss.append(err)
62     return loss
```

For the training loop, we use a batch of examples to obtain the initial hidden states, reconstructed hidden states and reconstructed visible states for each example (line 61). Since we are using the sampled data as a starting point and the probability distribution of our model should converge to that of the data, we do not need many iterations in the Gibbs sampling. The authors recommend 1 to 20 iterations. To save time in this example I used only 2.

Then from those states we compute the gradients according to the contrastive divergence algorithm, and update the weights.

We can also create an auxiliary function to deterministically reconstruct an example. Basically, we use the computed probabilities of the hidden state based on the visible data to compute the probabilities

of the visible variables and output these. This gives us a smoother reconstruction than what we would obtain by sampling the discrete states.

```
def rbm_reconstruct(X):
    h = tf.nn.sigmoid(tf.matmul(X, weights) + h_bias)
    reconstruct = tf.nn.sigmoid(tf.matmul(h, tf.transpose(weights)) + v_bias)
    return reconstruct
```

Now to train our RBM, we initialize the parameter tensors and call the training function:

```
weights = tf.zeros([visible_size, hidden_size], np.float32)
h_bias = tf.zeros([hidden_size], np.float32)
v_bias = tf.zeros([visible_size], np.float32)

err = train(train_data,50)
```

We can now test if our RBM can reconstruct test examples correctly and how it can denoise them:

```
sample = test_data[:10,:]
noisy = np.copy(sample)
noisy[np.random.rand(*sample.shape)>0.98] = 1
out = rbm_reconstruct(noisy)
orig_out = rbm_reconstruct(sample)
row, col = 4, 10
fig, axs = plt.subplots(row, col, sharex=True, sharey=True,figsize=(20,8))
for c in range(col):
    axs[0,c].imshow(tf.reshape(sample[c],[28, 28]), cmap='Greys_r')
    axs[1,c].imshow(tf.reshape(orig_out[c],[28, 28]), cmap='Greys_r')
    axs[2,c].imshow(tf.reshape(noisy[c],[28, 28]), cmap='Greys_r')
    axs[3,c].imshow(tf.reshape(out[c],[28, 28]), cmap='Greys_r')
for ax in axs.flatten():
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

The result looks something like shown in Figure 17.4.

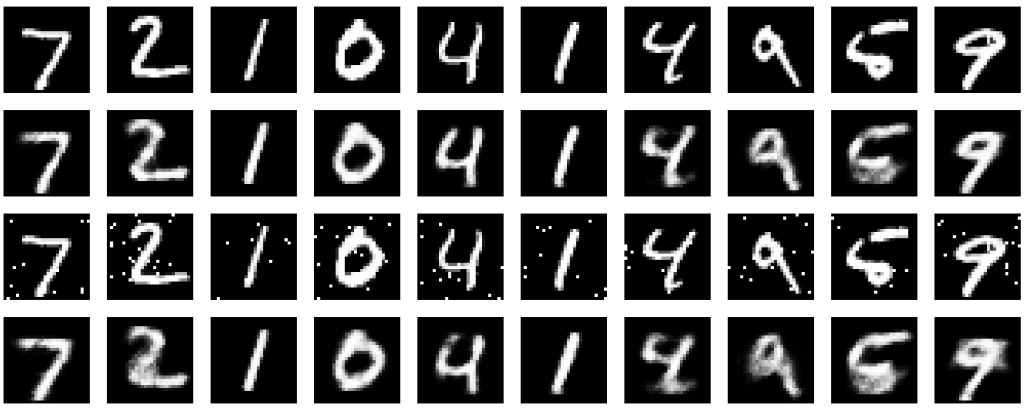


Figure 17.4: Testing the Restricted Boltzmann Machine. The top row shows some examples from the MNIST test set. The second row shows the same examples reconstructed by the RBM. The third row shows the examples with added noise and the final row the reconstruction of the noisy examples.

We can also use Gibbs sampling to generate random examples from the RBM. The results are shown in Figure 17.5. Note that this is a very simplified approach. More sophisticated sampling methods

should be used for better results because simple Gibbs sampling tends to converge to the same modes. We can see this easily when we start from actual samples in the test set instead of starting from random inputs (Figure 17.6)

```
sample = np.random.rand(10,28*28).astype(np.float32)
k = 1
row, col = 6, 10
outs = []
for _ in range(1,row):
    _,_,o = gibbs_sampling(sample,k)
    k *= 10
    outs.append(o)

fig, axs = plt.subplots(row, col, sharex=True, sharey=True,figsize=(20,2*row))
for c in range(col):
    axs[0,c].imshow(tf.reshape(sample[c],[28, 28]), cmap='Greys_r')
    for ix in range(1,row):
        axs[ix,c].imshow(tf.reshape(outs[ix-1][c],[28, 28]), cmap='Greys_r')
for ax in axs.flatten():
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
```

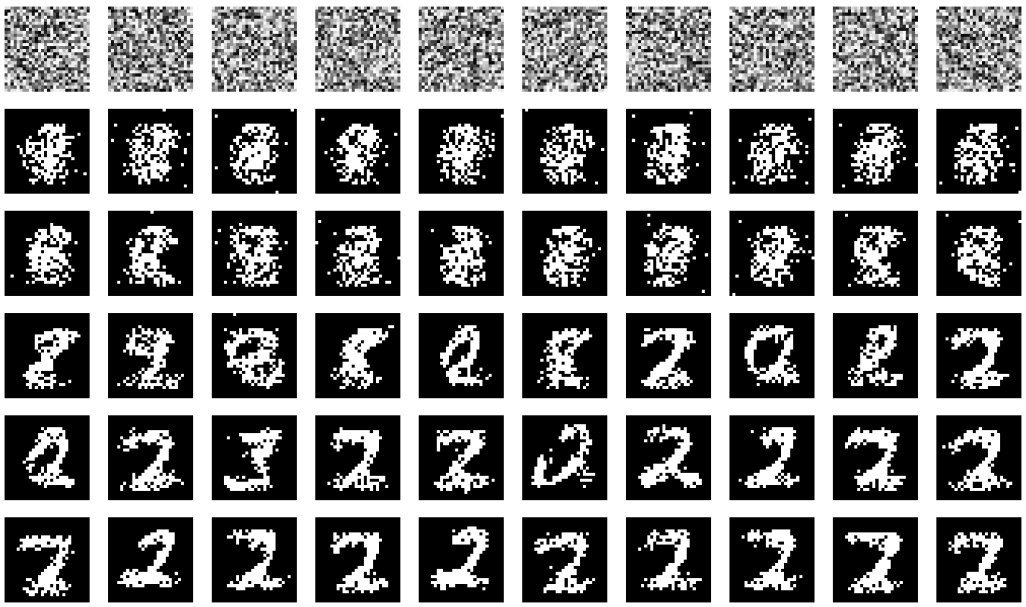


Figure 17.5: Generating examples from the RBM using Gibbs sampling. The first row shows the random input. The second row shows examples generated with 1 step of Gibbs sampling. For the subsequent rows the number of steps is ten times the number in the previous row.

17.5 Further Reading

1. Goodfellow et. al., Chapter 16 and Sections 18.1, 18.2, 20.1-4)[21]

2. (Optional: Breleux *et. al.*, Quickly Generating Representative Samples from an RBM-Derived Process,[8]

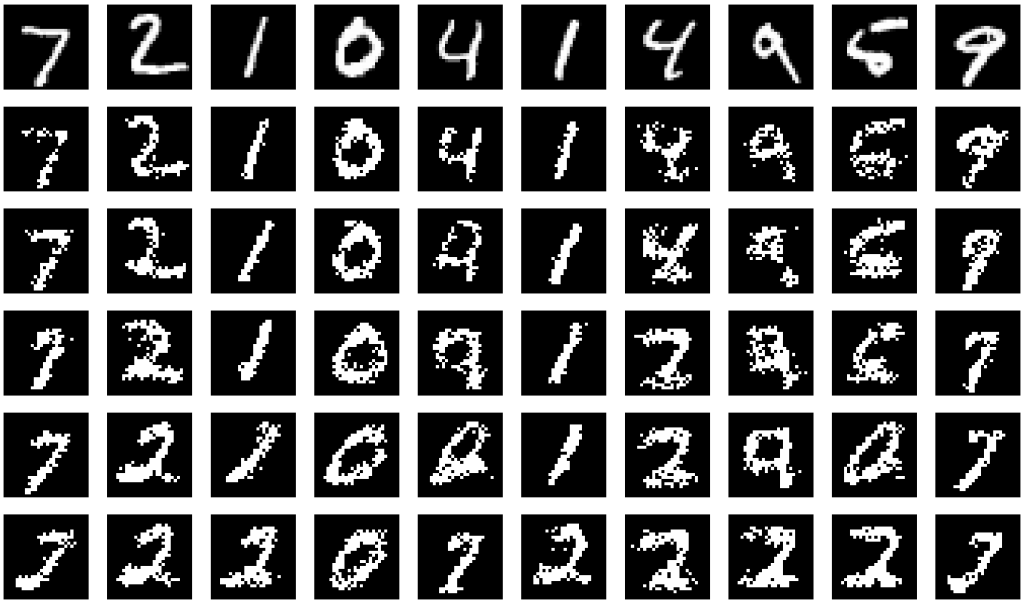


Figure 17.6: Convergence of Gibbs sampling. Starting from examples in the test set and using increasing numbers of steps in the Gibbs sampling (starting from 1 step in the second row and multiplying by 10 in each subsequent row).

Chapter 18

Transformers

Attention. Transformers. Implementing a transformer for translation

18.1 Attention

When we process sensory data, we do not consider all the inputs equally important. Instead, we give more weight to some inputs by focusing our attention on those details that are more relevant. Attention mechanisms in artificial neural networks try to mimic this capability of natural systems like our brains. The motivation for attention mechanisms arose with encoder-decoder recurrent networks and the need to help the networks learn long term dependencies. Figure 18.1 illustrates a recurrent network to translate from English to Chinese. The encoder creates a context vector from a sequence of English that the decoder then uses to generate the sequence of Chinese words.

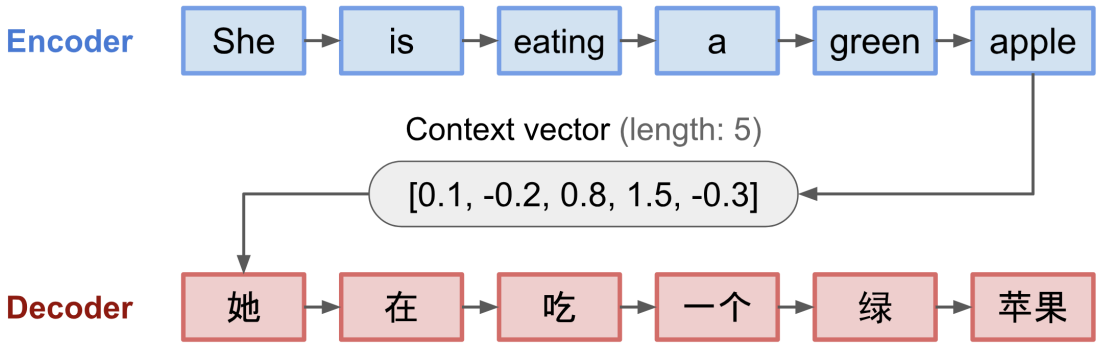


Figure 18.1: Example encoder-decoder recurrent network. Image by Lilian Weng.

Bahdanau *et al.* proposed an attention mechanism involving a feed-forward network that learned a set of weights from the states of the encoder. The weighted sum of the encoder states creates a dynamic context vector for the decoder. This improved significantly the ability of the model to learn dependencies between elements of the input sequence because the encoder no longer needed to encode all the information in a single vector and the decoder could pay different attention to different states of the encoder depending on the element being generated by the decoder. Figure 18.2 illustrates this.

In fact, attention mechanisms result in such an improvement that they seem to make recurrent networks unnecessary. Vaswani *et al.* proposed the Transformer architecture[75] as a more efficient

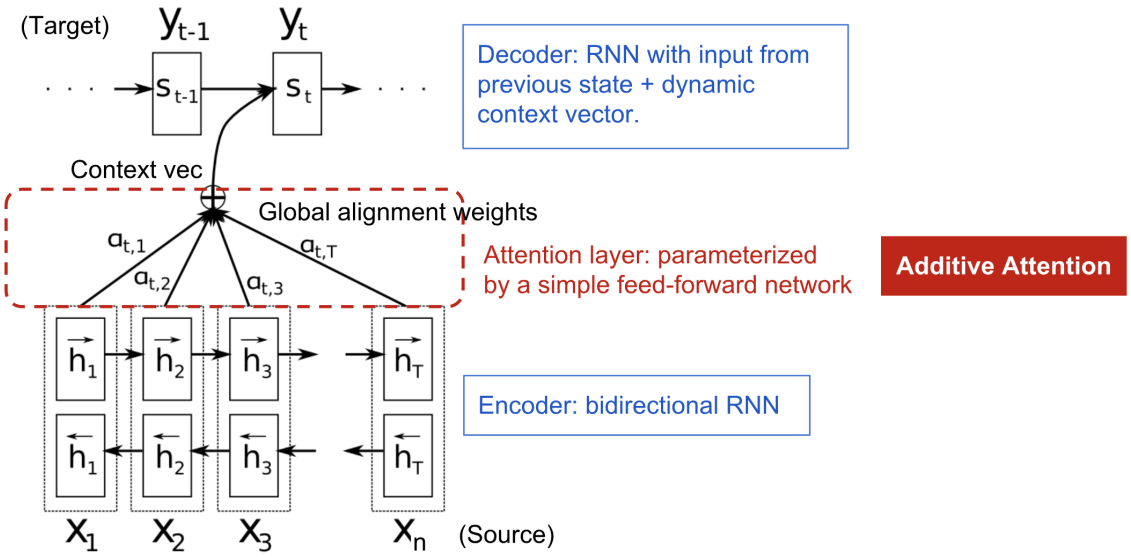


Figure 18.2: Example additive attention mechanism for an encoder-decoder recurrent network. Image by Lilian Weng, adapted from [3].

alternative, since feed-forward networks can be trained more efficiently with parallelization. There are several different attention mechanisms, but we will focus on the scaled dot product attention described in [75]. Conceptually, we can think of two sequences, which the authors called query and keys, and with dimension d_k, whose alignment will determine the attention paid to different elements of a values sequence with dimension d_v. The expression for the values modified by attention is:

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^\top}{\sqrt{d_k}}\right)V$$

Note that the distinction between keys, query and values is merely conceptual. This attention mechanism can (and will) be used for self-attention too in the Transformer model we will see below.

In addition, the authors found that splitting the attention layer into several independent "heads" facilitates learning relations different scales. The outputs of the different attention heads are then concatenated and projected to the initial dimension. Figure 18.3 illustrates the operations of the scaled dot-product attention and the multi-head attention architecture.

Transformer network

The Transformer model is a feed-forward network that can be used to generate sequences by iteratively receiving as input the input sequence and the output sequence generated so far. The output sequence begins with a special start token and the network is repeatedly run until it outputs an end token, indicating the end of the sequence. Figure 18.4 illustrates this model.

The model consists of an encoder that receives the input sequence and has a sequence of self-attention and dense networks. It also includes residual layers, to improve learning with deeper models, and layer normalization, which normalizes the activations for each example individually.

The output of the multi-head self-attention in the encoder is fed into the attention layer of the decoder as the query input. Combined with the keys coming from the output sequence generated so far, this will generate the attention weights for the vectors describing the output sequence, coming

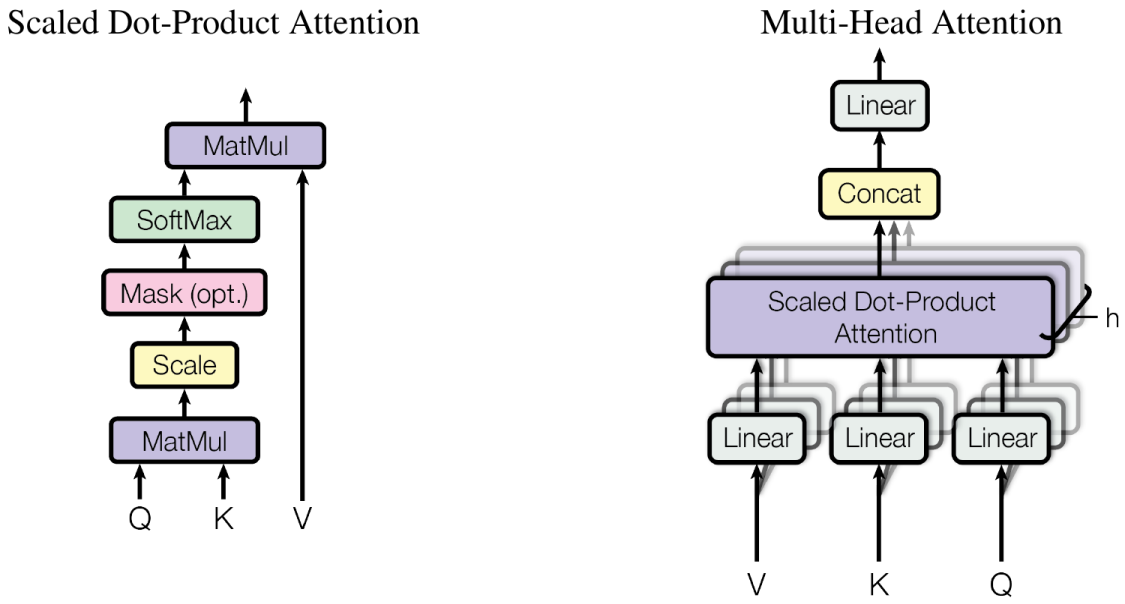


Figure 18.3: Multi-head dot-product attention architecture. Image from [75].

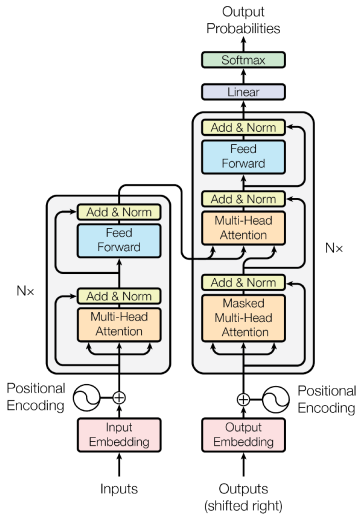


Figure 18.4: The Transformer model. Image from [75].

from a previous self-attention layer. In the next section we will see how to implement this model for translating Portuguese to English.

Example: using a transformer for translation

In this example[1] we will create a transformer network for translating Portuguese to English. Before we start, you may need to install the Tensorflow datasets and text libraries. You can use the pip installer. Just make sure you are installing the libraries on the environment you use for the exercises. Alternatively, you may be able to do this exercise using Colab.

```
pip install tensorflow_datasets
pip install tensorflow_text
```

[1] Based on the Tensorflow tutorial "Transformer model for language understanding", https://www.tensorflow.org/tutorials/text/transformer

We begin by importing the necessary libraries:

```
14 import time
15 import numpy as np
16 import matplotlib.pyplot as plt
17 import tensorflow_datasets as tfds
18 import tensorflow as tf
```

We will also need the data set and a tokenizer. A tokenizer model can convert sequences of words into numerical vectors with each value encoding a different word. This is necessary because we cannot use the words themselves as input into our model. This code will download the data set and the tokenizer models specially created for these data. After the first run of this code, the cached data and model will be used.

```
21 examples, metadata = tfds.load('ted_hrlr_translate/pt_to_en', with_info=True,
22                                 as_supervised=True)
23 train_examples, val_examples = examples['train'], examples['validation']
24
25 model_name = "ted_hrlr_translate_pt_en_converter"
26 tf.keras.utils.get_file(
27     f"{model_name}.zip",
28     f"https://storage.googleapis.com/download.tensorflow.org/models/{model_name}.zip",
29     cache_dir='.', cache_subdir='', extract=True)
30
31 tokenizers = tf.saved_model.load(model_name)
```

Now we create the data pipeline for training our model. To do this we will create a function that receives pairs of sentences in Portuguese and English and uses the tokenizers to generate the zero-padded tensors with the numerical encoding of the words. The `make_batches` function then prepares the datasets by specifying the data processing pipeline.

```
33 def tokenize_pairs(pt, en):
34     pt = tokenizers.pt.tokenize(pt)
35     pt = pt.to_tensor()
36     en = tokenizers.en.tokenize(en)
37     en = en.to_tensor()
38     return pt, en
39
40 BUFFER_SIZE = 20000
41 BATCH_SIZE = 64
42
43 def make_batches(ds):
44   return (
45       ds
46       .cache()
47       .shuffle(BUFFER_SIZE)
48       .batch(BATCH_SIZE)
49       .map(tokenize_pairs, num_parallel_calls=tf.data.AUTOTUNE)
50       .prefetch(tf.data.AUTOTUNE))
51
52 train_batches = make_batches(train_examples)
53 val_batches = make_batches(val_examples)
```

Since our model will be looking at whole sequences, we must add positional information to the vectors describing each word. The idea is that after embedding (we will see this later), similar words will be represented by similar vectors. If we add positional encoding vectors that are also similar for words in similar positions but different for words in different positions, the similarity between the words in the embedding will also depend on their positions along the sentence. One way to do this is to use the following formula to generate the vectors of length d_{model} for each position along the sequence:

$$PE_{(pos,2i)} = \sin(pos/100000^{2i/d_{model}}) PE_{(pos,2i+1)} = \cos(pos/100000^{2i/d_{model}})$$

This seems strange but it can generate vectors that are similar in close positions and different as the positions become more distant, as Figure 18.5 illustrates.

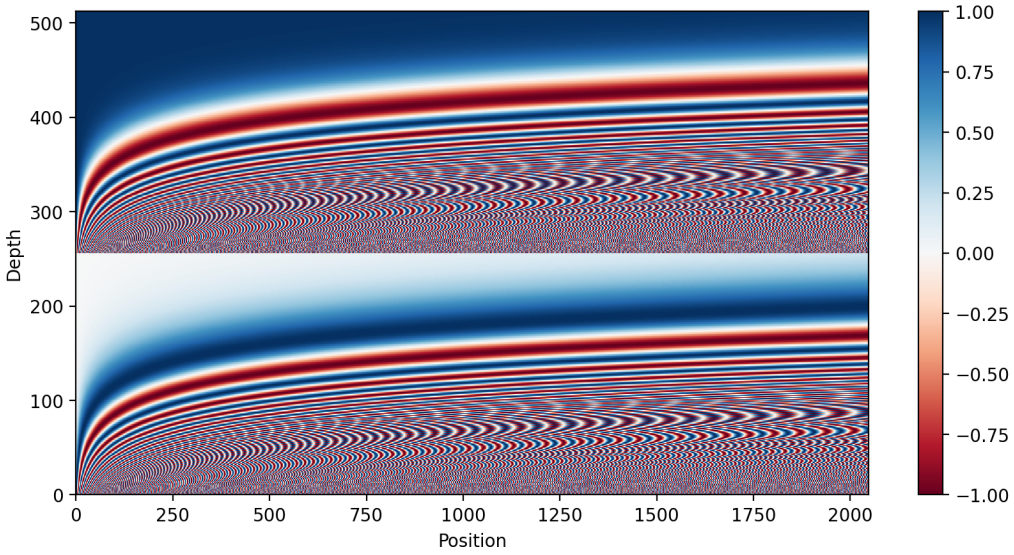


Figure 18.5: Positional encoding vectors.

```
56 def get_angles(pos, i, d_model):
57   angle_rates = 1 / np.power(10000, (2 * (i//2)) / np.float32(d_model))
58   return pos * angle_rates
59
60 def positional_encoding(position, d_model):
61   angle_rads = get_angles(np.arange(position)[:, np.newaxis],
62                           np.arange(d_model)[np.newaxis, :],
63                           d_model)
64   angle_rads[:, 0::2] = np.sin(angle_rads[:, 0::2])
65   angle_rads[:, 1::2] = np.cos(angle_rads[:, 1::2])
66   pos_encoding = angle_rads[np.newaxis, ...]
67   return tf.cast(pos_encoding, dtype=tf.float32)
```

Another detail we must solve is to generate the masks we need to ignore the parts of the input vectors that contain no words at the moment, either because the input sentence is shorter than the maximum or because we are feeding the output sentence gradually.

```
70 def create_padding_mask(seq):
71   seq = tf.cast(tf.math.equal(seq, 0), tf.float32)
72   return seq[:, tf.newaxis, tf.newaxis, :]  # (batch_size, 1, 1, seq_len)
73
```

```
74 def create_look_ahead_mask(size):
75   mask = 1 - tf.linalg.band_part(tf.ones((size, size)), -1, 0)
76   return mask
```

The padding mask function merely returns a tensor with a value of 1 where the original is 0. The look ahead mask returns a tensor that indicates, in each row, which elements to hide. For example:

```
In : create_look_ahead_mask(4)
Out:
<tf.Tensor: shape=(4, 4), dtype=float32, numpy=
array([[0., 1., 1., 1.],
       [0., 0., 1., 1.],
       [0., 0., 0., 1.],
       [0., 0., 0., 0.]], dtype=float32)>       [0., 0., 0.]], dtype=float32)>
```

Now we can start implementing the attention mechanism. First, the scaled dot product. Note how the mask is applied, by adding a large negative number to the masked regions of the vector. In the softmax, the large negative number will correspond to a zero, so there will be no attention paid to the masked regions.

```
78 def scaled_dot_product_attention(q, k, v, mask):
79   matmul_qk = tf.matmul(q, k, transpose_b=True)
80   dk = tf.cast(tf.shape(k)[-1], tf.float32)
81   scaled_attention_logits = matmul_qk / tf.math.sqrt(dk)
82   if mask is not None:
83     scaled_attention_logits += (mask * -1e9)
84   attention_weights = tf.nn.softmax(scaled_attention_logits, axis=-1)
85   output = tf.matmul(attention_weights, v)
86   return output, attention_weights
```

Now we create a Keras layer for the multi-head attention. To do this we can subclass the basic Keras `Layer` class and implement the `__init__` and `call` methods. First we create the dense layers for the query, keys and values, and the output dense layer that projects the multiple heads into a single vector with size d_{model}.

```
88 class MultiHeadAttention(tf.keras.layers.Layer):
89   def __init__(self, d_model, num_heads):
90     super(MultiHeadAttention, self).__init__()
91     self.num_heads = num_heads
92     self.d_model = d_model
93     assert d_model % self.num_heads == 0
94     self.depth = d_model // self.num_heads
95     self.wq = tf.keras.layers.Dense(d_model)
96     self.wk = tf.keras.layers.Dense(d_model)
97     self.wv = tf.keras.layers.Dense(d_model)
98     self.dense = tf.keras.layers.Dense(d_model)
```

Now the `call` method that specifies the computations:

```
100   def split_heads(self, x, batch_size):
101     x = tf.reshape(x, (batch_size, -1, self.num_heads, self.depth))
102     return tf.transpose(x, perm=[0, 2, 1, 3])
103
```

```
104   def call(self, v, k, q, mask):
105     batch_size = tf.shape(q)[0]
106     q = self.wq(q)
107     k = self.wk(k)
108     v = self.wv(v)
109     q = self.split_heads(q, batch_size)
110     k = self.split_heads(k, batch_size)
111     v = self.split_heads(v, batch_size)
112     scaled_attention, attention_weights = scaled_dot_product_attention(q, k, v, mask)
113     scaled_attention = tf.transpose(scaled_attention, perm=[0, 2, 1, 3])
114     concat_attention = tf.reshape(scaled_attention,(batch_size, -1, self.d_model))
115     output = self.dense(concat_attention)
116     return output, attention_weights
```

Now we create a function for the point wise network, which is a stack of two dense layers with a ReLU activation between them:

```
120 def point_wise_feed_forward_network(d_model, dff):
121   return tf.keras.Sequential([
122       tf.keras.layers.Dense(dff, activation='relu'),
123       tf.keras.layers.Dense(d_model)
124       ])
```

and use this in the encoder and decoder layers, which correspond to the blocks used in the Transformer model. First the encoder, starting with the self attention layer, then dropout and normalization, then a point-wise feed-forward layer and the final normalization. Note that the normalization layers are residual layers too, since they receive as input the sum of the input to the block and the output of the previous layers.

```
126 class EncoderLayer(tf.keras.layers.Layer):
127   def __init__(self, d_model, num_heads, dff, rate=0.1):
128     super(EncoderLayer, self).__init__()
129     self.mha = MultiHeadAttention(d_model, num_heads)
130     self.ffn = point_wise_feed_forward_network(d_model, dff)
131     self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
132     self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
133     self.dropout1 = tf.keras.layers.Dropout(rate)
134     self.dropout2 = tf.keras.layers.Dropout(rate)
135
136   def call(self, x, training, mask):
137     attn_output, _ = self.mha(x, x, x, mask)
138     attn_output = self.dropout1(attn_output, training=training)
139     out1 = self.layernorm1(x + attn_output)
140     ffn_output = self.ffn(out1)
141     ffn_output = self.dropout2(ffn_output, training=training)
142     out2 = self.layernorm2(out1 + ffn_output)
143     return out2
```

Now the decoder block. The decoder starts with a self-attention layer, but the input in this case is the previous sequence of generated tokens. Then in the second multihead attention layer it includes the output from the encoder to determine the weights given to the representation of the sequence generated so far.

```
126 class DecoderLayer(tf.keras.layers.Layer):
127   def __init__(self, d_model, num_heads, dff, rate=0.1):
128     super(DecoderLayer, self).__init__()
129     self.mha1 = MultiHeadAttention(d_model, num_heads)
130     self.mha2 = MultiHeadAttention(d_model, num_heads)
131     self.ffn = point_wise_feed_forward_network(d_model, dff)
132     self.layernorm1 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
133     self.layernorm2 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
134     self.layernorm3 = tf.keras.layers.LayerNormalization(epsilon=1e-6)
135     self.dropout1 = tf.keras.layers.Dropout(rate)
136     self.dropout2 = tf.keras.layers.Dropout(rate)
137     self.dropout3 = tf.keras.layers.Dropout(rate)
138
139   def call(self, x, enc_output, training,
140            look_ahead_mask, padding_mask):
141     attn1, attn_weights_block1 = self.mha1(x, x, x, look_ahead_mask)
142     attn1 = self.dropout1(attn1, training=training)
143     out1 = self.layernorm1(attn1 + x)
144     attn2, attn_weights_block2 = self.mha2(
145         enc_output, enc_output, out1, padding_mask)
146     attn2 = self.dropout2(attn2, training=training)
147     out2 = self.layernorm2(attn2 + out1)
148     ffn_output = self.ffn(out2)
149     ffn_output = self.dropout3(ffn_output, training=training)
150     out3 = self.layernorm3(ffn_output + out2)
151     return out3, attn_weights_block1, attn_weights_block2
```

Now we create the encoder. The encoder will begin with an `Embedding` layer. This Keras layer maps each integer value corresponding to a word into a vector of floats. This mapping is learned when training the model to find the best embedding for the particular problem that is being solved. The encoder also adds the positional encoding data before stacking a number of encoding layers.

```
172 class Encoder(tf.keras.layers.Layer):
173   def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
174                maximum_position_encoding, rate=0.1):
175     super(Encoder, self).__init__()
176     self.d_model = d_model
177     self.num_layers = num_layers
178     self.embedding = tf.keras.layers.Embedding(input_vocab_size, d_model)
179     self.pos_encoding = positional_encoding(maximum_position_encoding,
180                                             self.d_model)
181     self.enc_layers = [EncoderLayer(d_model, num_heads, dff, rate)
182                        for _ in range(num_layers)]
183     self.dropout = tf.keras.layers.Dropout(rate)
184
185   def call(self, x, training, mask):
186     seq_len = tf.shape(x)[1]
187     x = self.embedding(x)
188     x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
189     x += self.pos_encoding[:, :seq_len, :]
190     x = self.dropout(x, training=training)
191     for i in range(self.num_layers):
192       x = self.enc_layers[i](x, training, mask)
193     return x
```

The decoder also includes embedding and positional encoder. Additionally, the decoder returns a dictionary with the weights for the attention layers. We will not use these in this example, but they can be used to examine the attention layers (see the original tutorial for more details).

```
class Decoder(tf.keras.layers.Layer):
  def __init__(self, num_layers, d_model, num_heads, dff, target_vocab_size,
               maximum_position_encoding, rate=0.1):
    super(Decoder, self).__init__()
    self.d_model = d_model
    self.num_layers = num_layers
    self.embedding = tf.keras.layers.Embedding(target_vocab_size, d_model)
    self.pos_encoding = positional_encoding(maximum_position_encoding, d_model)
    self.dec_layers = [DecoderLayer(d_model, num_heads, dff, rate)
                       for _ in range(num_layers)]
    self.dropout = tf.keras.layers.Dropout(rate)

  def call(self, x, enc_output, training,
           look_ahead_mask, padding_mask):
    seq_len = tf.shape(x)[1]
    attention_weights = {}
    x = self.embedding(x)
    x *= tf.math.sqrt(tf.cast(self.d_model, tf.float32))
    x += self.pos_encoding[:, :seq_len, :]
    x = self.dropout(x, training=training)
    for i in range(self.num_layers):
      x, block1, block2 = self.dec_layers[i](x, enc_output, training,
                                             look_ahead_mask, padding_mask)
      attention_weights[f'decoder_layer{i+1}_block1'] = block1
      attention_weights[f'decoder_layer{i+1}_block2'] = block2
    return x, attention_weights
```

Now we create the transformer model, which simply wraps up the blocks we created so far. In this case, we subclass the Keras `Model` class, but the necessary methods are the same.

```
class Transformer(tf.keras.Model):
  def __init__(self, num_layers, d_model, num_heads, dff, input_vocab_size,
               target_vocab_size, pe_input, pe_target, rate=0.1):
    super(Transformer, self).__init__()
    self.tokenizer = Encoder(num_layers, d_model, num_heads, dff,
                             input_vocab_size, pe_input, rate)
    self.decoder = Decoder(num_layers, d_model, num_heads, dff,
                           target_vocab_size, pe_target, rate)
    self.final_layer = tf.keras.layers.Dense(target_vocab_size)

  def call(self, inp, tar, training, enc_padding_mask,
           look_ahead_mask, dec_padding_mask):
    enc_output = self.tokenizer(inp, training, enc_padding_mask)
    dec_output, attention_weights = self.decoder(
        tar, enc_output, training, look_ahead_mask, dec_padding_mask)
    final_output = self.final_layer(dec_output)
    return final_output, attention_weights
```

To optimize training, we will create a scheduler to adjust the learning rate during training. The `__call__` method is used to obtain the learning rate at each training step and is called by the optimizer.

```
class CustomSchedule(tf.keras.optimizers.schedules.LearningRateSchedule):
  def __init__(self, d_model, warmup_steps=4000):
    super(CustomSchedule, self).__init__()
    self.d_model = d_model
    self.d_model = tf.cast(self.d_model, tf.float32)
    self.warmup_steps = warmup_steps

  def __call__(self, step):
    arg1 = tf.math.rsqrt(step)
    arg2 = step * (self.warmup_steps ** -1.5)
    return tf.math.rsqrt(self.d_model) * tf.math.minimum(arg1, arg2)
```

The loss function and accuracy measures will need to ignore the masked elements of the vectors (set to zero). Also, we will use a global object for the sparse categorical cross-entropy so that it is not created for each execution of the loss function. The sparse categorical cross-entropy is a variant of the categorical cross-entropy with hard labels of 0 and 1.

We will also create the mean operations for computing loss and accuracy during training as global objects. This allows us to compile the computational graph for execution as we will see below.

```
loss_object = tf.keras.losses.SparseCategoricalCrossentropy(
    from_logits=True, reduction='none')

def loss_function(real, pred):
  mask = tf.math.logical_not(tf.math.equal(real, 0))
  loss_ = loss_object(real, pred)
  mask = tf.cast(mask, dtype=loss_.dtype)
  loss_ *= mask
  return tf.reduce_sum(loss_)/tf.reduce_sum(mask)

def accuracy_function(real, pred):
  accuracies = tf.equal(real, tf.argmax(pred, axis=2))
  mask = tf.math.logical_not(tf.math.equal(real, 0))
  accuracies = tf.math.logical_and(mask, accuracies)
  accuracies = tf.cast(accuracies, dtype=tf.float32)
  mask = tf.cast(mask, dtype=tf.float32)
  return tf.reduce_sum(accuracies)/tf.reduce_sum(mask)

train_loss = tf.keras.metrics.Mean(name='train_loss')
train_accuracy = tf.keras.metrics.Mean(name='train_accuracy')
```

Now we create the model for training. This is a simplified version so that training does not take too long. In the original paper the model reported is larger.

```
num_layers = 4
d_model = 128
dff = 512
num_heads = 8
dropout_rate = 0.1

learning_rate = CustomSchedule(d_model)
optimizer = tf.keras.optimizers.Adam(learning_rate, beta_1=0.9, beta_2=0.98,
                                     epsilon=1e-9)
transformer = Transformer(
```

```
    num_layers=num_layers,
    d_model=d_model,
    num_heads=num_heads,
    dff=dff,
    input_vocab_size=tokenizers.pt.get_vocab_size(),
    target_vocab_size=tokenizers.en.get_vocab_size(),
    pe_input=1000,
    pe_target=1000,
    rate=dropout_rate)
```

During training it is useful to create regular checkpoints and save the model. We can do this with a checkpoint manager:

```
checkpoint_path = "./checkpoints/train"
ckpt = tf.train.Checkpoint(transformer=transformer,
                           optimizer=optimizer)
ckpt_manager = tf.train.CheckpointManager(ckpt, checkpoint_path, max_to_keep=5)
if ckpt_manager.latest_checkpoint:
  ckpt.restore(ckpt_manager.latest_checkpoint)
  print('Latest checkpoint restored!!')
```

We also need an auxiliary function to create the different masks we will need. The encoder and decoder padding masks will mask the parts of the input text that have no words, since not all sentences have the maximum length. The look ahead mask will mask the next words to be generated so that the decoder cannot see which word comes next, and this will be combined with a padding mask for the target to mask the empty parts of the target sentence.

```
def create_masks(inp, tar):
  enc_padding_mask = create_padding_mask(inp)
  dec_padding_mask = create_padding_mask(inp)
  look_ahead_mask = create_look_ahead_mask(tf.shape(tar)[1])
  dec_target_padding_mask = create_padding_mask(tar)
  combined_mask = tf.maximum(dec_target_padding_mask, look_ahead_mask)
  return enc_padding_mask, combined_mask, dec_padding_mask
```

Finally, we can create the training step. We will use the `@tf.function` decorator so that Tensorflow compiles the whole graph. This makes execution faster. Since in a compiled graph the tensors cannot change, we need to specify the signature of the input tensors, because the batch size may vary (the last batch may be smaller) and we will use the global objects for the loss and models. Using a compiled graph requires some additional coding but can speed up training significantly.

Note that in the training step the target is shifted relative to the target input because we want the decoder to predict the next word. Also, we are feeding as the previously generated sequence not the output from the decoder but the target sentence. This is the teacher forcing method we saw before in the context of recurrent networks.

```
train_step_signature = [
    tf.TensorSpec(shape=(None, None), dtype=tf.int64),
    tf.TensorSpec(shape=(None, None), dtype=tf.int64),
]

@tf.function(input_signature=train_step_signature)
def train_step(inp, tar):
```

```
  tar_inp = tar[:, :-1]
  tar_real = tar[:, 1:]
  enc_padding_mask, combined_mask, dec_padding_mask = create_masks(inp, tar_inp)
  with tf.GradientTape() as tape:
    predictions, _ = transformer(inp, tar_inp,
                                 True,
                                 enc_padding_mask,
                                 combined_mask,
                                 dec_padding_mask)
    loss = loss_function(tar_real, predictions)
  gradients = tape.gradient(loss, transformer.trainable_variables)
  optimizer.apply_gradients(zip(gradients, transformer.trainable_variables))
  train_loss(loss)
  train_accuracy(accuracy_function(tar_real, predictions))
```

Finally, the training loop:

```
EPOCHS=20
for epoch in range(EPOCHS):
  start = time.time()
  train_loss.reset_states()
  train_accuracy.reset_states()
  for (batch, (inp, tar)) in enumerate(train_batches):
    train_step(inp, tar)
    if batch % 50 == 0:
      print(f'Epoch {epoch + 1} Batch {batch} Loss {train_loss.result():.4f} Accuracy {train_a
  if (epoch + 1) % 5 == 0:
    ckpt_save_path = ckpt_manager.save()
    print(f'Saving checkpoint for epoch {epoch+1} at {ckpt_save_path}')
  print(f'Epoch {epoch + 1} Loss {train_loss.result():.4f} Accuracy {train_accuracy.result():
  print(f'Time taken for 1 epoch: {time.time() - start:.2f} secs\n')
```

Note that the `Mean` objects we created for the loss and accuracy metrics keep running averages so they must be reset at the beginning of each epoch.

To translate sentences we can use this function. It converts the sentence to a tensor of tokens, starts the output tensor with the start token for English, and then keeps feeding the output sequence into the transformer, along with the input sequence, to obtain the next token which is added to the output sequence until an end token is generated.

```
def evaluate(sentence, max_length=40):
  sentence = tf.convert_to_tensor([sentence])
  sentence = tokenizers.pt.tokenize(sentence).to_tensor()
  encoder_input = sentence
  start, end = tokenizers.en.tokenize([''])[0]
  output = tf.convert_to_tensor([start])
  output = tf.expand_dims(output, 0)
  for i in range(max_length):
    enc_padding_mask, combined_mask, dec_padding_mask = create_masks(encoder_input, output)
    predictions, attention_weights = transformer(encoder_input,
                                                 output,
                                                 False,
                                                 enc_padding_mask,
                                                 combined_mask,
                                                 dec_padding_mask)
```

```
    predictions = predictions[:, -1:, :]  # (batch_size, 1, vocab_size)
    predicted_id = tf.argmax(predictions, axis=-1)
    output = tf.concat([output, predicted_id], axis=-1)
    if predicted_id == end:
      break
  text = tokenizers.en.detokenize(output)[0]
  tokens = tokenizers.en.lookup(output)[0]
  return text, tokens, attention_weights
```

Here are some examples of translations. For straightforward sentences the model works quite well. Idiomatic expressions, however, are much harder to translate.

```
def translate(sentence):
    translated_text, _, _ = evaluate(sentence)
    print(translated_text.numpy().decode("utf-8"))

In : translate("este é um problema que temos que resolver")
Out: this is a problem we have to solve .

In : translate("este modelo parece funcionar bem")
Out: this model seems to work well .

In : translate("quem tudo quer tudo perde")
Out: who always wants everything to lose .

In : translate("estás aqui estás a levar")
Out: these are actually taking over here .

In : translate("estou feito ao bife")
Out: i ' m done by the different tree .
```

Chapter 19

Introduction to Reinforcement Learning

Reinforcement learning problems. Markov Decision Process. Policies and how to evaluate states and actions. Improving policies.

19.1 Introduction

So far we have seen examples of supervised learning, with problems like classification or segmentation, and unsupervised learning, as in autoencoders or restricted Boltzmann machines, where the goal was to fit a model to some aspect of a data set. Reinforcement learning problems are fundamentally different. Instead of fitting a model to a data set we want to optimize the performance of an agent that interacts with an environment. Our data is not a static set of examples but will result from how the agent explores the environment. Furthermore, the performance of the agent will be determined by the outcome of sequences of decisions and not just a sum of independent examples. Figure 19.1 illustrates the context of a reinforcement learning problem, where an agent will act in an environment and receive information about the state of the environment and the reward associated to that state given by some interpreter. For our purposes, the agent is strictly speaking the party responsible for deciding how to act. For example, in the case of a real-life robot, the agent would be the software controlling the robot. The environment would include the hardware of the robot as well as the objects the robot interacts with, since all these elements would react to the controller's decisions. The robot would also need some sensors which would interpret the state of the environment and feed the agent (the controller software) with observations. During training, the agent would also need to have some feedback on the outcome of its decisions, in the form of a reward. The observations and rewards are not given directly by the environment but by some system, which we can call an interpreter, that mediates between the environment and the agent.

We can breakdown the reinforcement learning cycle in four steps, which are repeated during training:

1. The agent receives observations about the environment and, at least some times, some feedback (reward) evaluating the state of the environment. This reward can be negative (*e.g.* a large penalty if the robot collides with a wall) or positive (*e.g.* if a goal was achieved) but in many cases it may happen that there is no reward or penalty for the current state.

2. The agent adjusts its decision algorithm based on the observations and rewards. This is what reinforcement learning aims to improve.

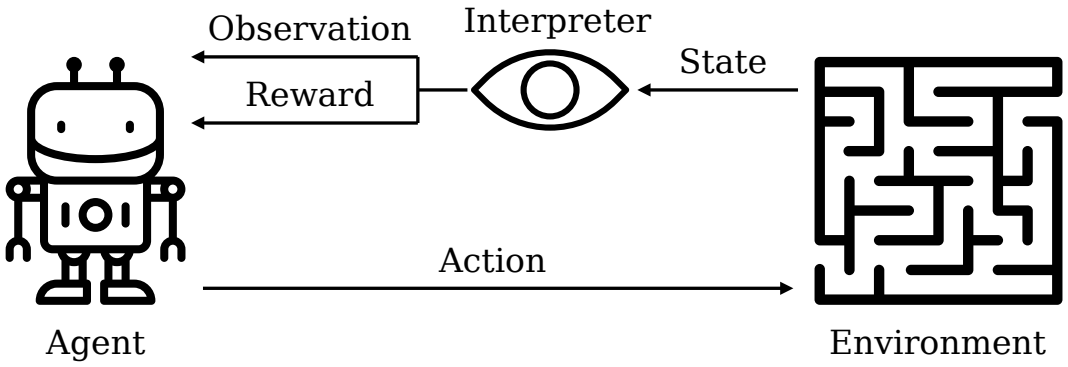


Figure 19.1: Schematic representation of a reinforcement learning problem. Images from Freepik (robot) and SVG Repo.

3. The agent chooses an action.

4. The action affects the environment and the state of the environment may change as a consequence of that action. Training continues with step 1.

19.2 Markov Decision Process

In probabilities, the Markov property is the property of a stochastic process not retaining memory of previous states. So in a Markov property, the probability of an event at time t depends solely on what was true at $t-1$:

$$P(X_t = x_t \mid X_{t-1} = x_{t-1}, X_{t-2} = x_{t-2}, ..., X_1 = x_1) = P(X_t = x_t \mid X_{t-1} = x_{t-1})$$

Note that whether or not a process has the Markov property may depend on what we consider to be specified in each state. For example, if we have a bag with blue a white marbles and are taking out one marble at a time, the probability of taking a white marble at time t will depend on how many times we drew white and blue marbles at previous drawings. This would not have the Markov property. However, if at each state we keep track of how many marbles of each colour there are left in the bag, then the probability of drawing a white or blue marble will depend only on the current state and the previous states are irrelevant. This property of being without memory is very useful for solving reinforcement learning problems, because with this property the agent needs only consider the current state. So in reinforcement learning we try to describe the state of the environment in such a way that all relevant information is included so that the agent can decide on the next action simply from the current state.

This leads us to the definition of a Markov decision process (MDP): a tuple $(S, A, P_a, R, S_\theta, \gamma, H)$ in which:

S is the set of states that are given to the agent to describe the environment.

A is the set of actions the agent can take.

$P_a(s, s')$ gives the probability of transitioning from state s to state s' if action a was chosen on state s: $P_a(s, s') = P(s_{t+1} = s' \mid s_t = s, a_t = a)$.

$R(s, a, s')$ is the reward given by transitioning from state s to state s' with action a.

S_θ is the distribution of initial states.

γ is the discount factor for future rewards, so that a reward that is worth r at time 1 is only worth $\gamma^t r$ at time t.

H is the planning horizon, how many time steps the agent will plan for. This can be infinite or can specify a maximum number of steps.

In a Markov decision process, the decision is taken solely as function of the current state.

Let us consider this simple problem as an example. Our agent can move on a frozen lake and must go from the starting position (S) to the goal position (G). The environment has 16 locations, corresponding to 16 possible states. Frozen locations (F) are safe but some locations have holes (H) in the ice and if the agent falls in a hole they cannot get out. So state 0, S, is an *initial state*, because it is a state in which the agent can begin an *episode*. States G and H (states 5, 7, 11, 12 and 15) are *terminal states*. When the agent reaches one of these states the episode ends. We can reward the agent with a positive reward if they reach state G. Figure 19.2 illustrates this environment.

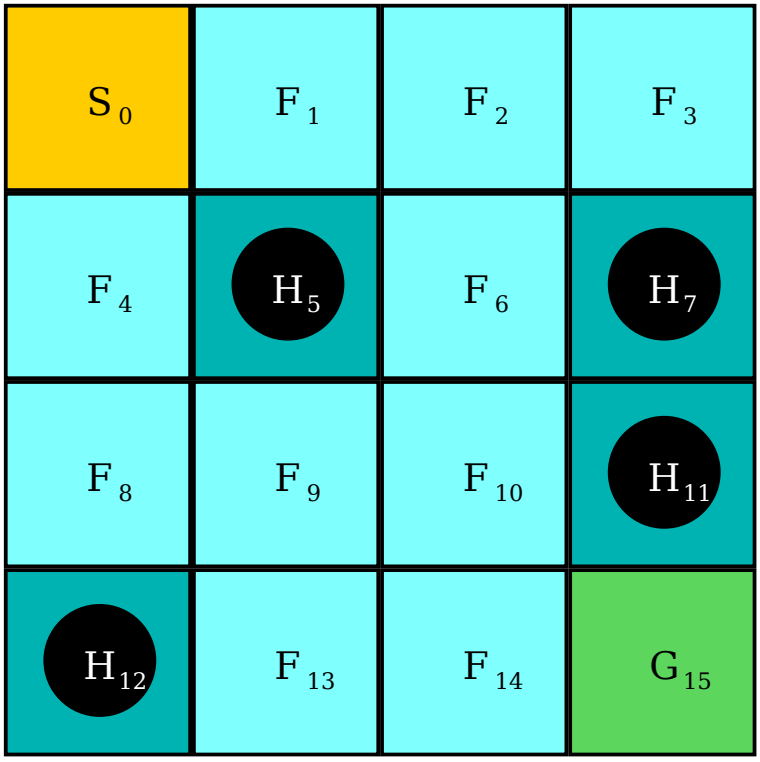


Figure 19.2: The frozen lake environment.

We can consider that the agent can take one of four actions, to try to move one step in each direction (N, S, E, W). However, the ice is slippery so that when trying to move there is a probability 1/3 probability that the agent will move in the desired direction and 1/3 of sliding 90° to either side of the desired direction. If the agent tries to move outside of the lake, the agent hits the snow and doesn't move. This way, we can create the P_a matrix. We can use a table:

State	Action	Actual Move	Probability	Next State
0	N	N	0.333	0
0	N	E	0.333	1
0	N	W	0.333	0
0	S	S	0.333	4
0	S	E	0.333	1
0	S	W	0.333	0
1	N	N	0.333	1
1	N	E	0.333	2
1	N	W	0.333	0
5	E	E	0.333	5
5	E	N	0.333	5
5	E	S	0.333	5
5	W	W	0.333	5
5	W	N	0.333	5
5	W	S	0.333	5

Now we need the reward function. The general form of this function is $R(s, a, s')$, since the reward may depend on the previous state, the action and the next state. But in this example we can ignore the previous state and the action and consider simply that $R(s')$ is 1 if $s' = 16$ or 0 otherwise. Thus the agent is only rewarded when reaching the goal state.

The distribution of initial states, S_θ, is simple in this case because the agent always starts in state $s_1 = 1$. For the planning horizon H we can assume a maximum of 15 steps, since this would be enough to cover all states. Now we need to choose the discount factor. The purpose of this parameter is to favour the agent obtaining the reward sooner rather than later by discounting the rewards if they are far in the future. Thus, the total discounted reward (the *return*) at time t, called G_t, with a discount of γ is the sum of the rewards for future time steps, each discounted by a increasing exponent of γ:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ...\gamma R_T = \sum_{k=0}^{T-1} \gamma^k R_{t+k+1}$$

Or, recursively, the return at time t is the reward at time $t+1$ plus the return at time $r+1$ discounted by γ:

$$G_t = R_{t+1} + \gamma G_{t+1}$$

19.3 Plans and Policies

To navigate the frozen lake problem we can think of a *plan*, which is a sequence of actions. For example, go East twice, then South three steps and finally go East. If the ice was not slippery this would solve our problem. However, since we may not move like we want, the plan may go wrong, as Figure **??** illustrates. For example, when trying to go South in state 3 we may end up going East to state 4 instead of state 7 as desired.

This is why we need a *policy* instead of simply a plan. While a plan is just a sequence of actions, a policy is a sort of universal plan for all contingencies. A policy may map each state to an action or to a probability distribution of actions, thus planning for all non-terminal states. The right panel of Figure **??** illustrates a deterministic policy for the frozen lake, showing the action to be chosen in each non-terminal state. Note that we do not need to specify actions for terminal states like the goal state or holes since these terminate the episode.

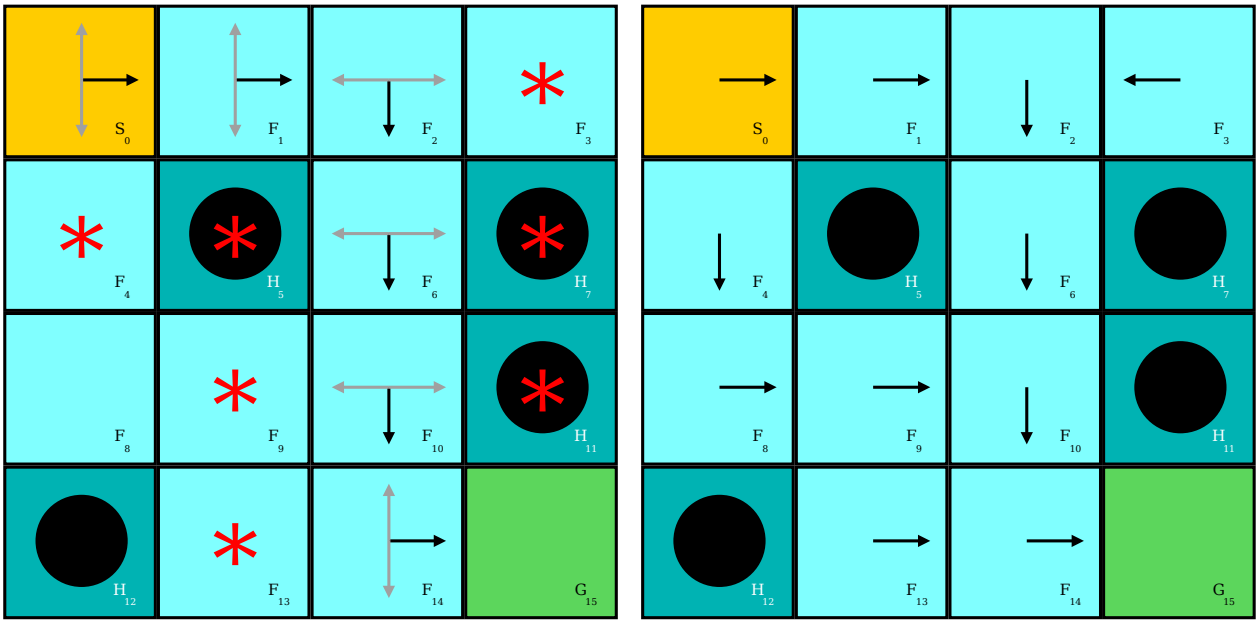


Figure 19.3: On the left, the plan and what can go wrong. Gray arrows indicate possible slipping directions and red stars show examples of where the agent may end up inadvertently. On the right panel, a deterministic policy for all non-terminal states (note that holes are terminal states, we do not need actions for those).

One problem with the policy shown in Figure 19.3 is that the agent may end up falling in a hole. It tries to guide the agent as quickly as possible towards the goal but at some peril. We may consider an alternative policy that avoids this danger. Figure 19.4 compares these two policies. How can we determine which policy is better?

19.4 The State-Value and Action-Value functions

To compare policies we must have some way to evaluate a policy. We can ask, given a policy, what is the expected return if we follow that policy from state s. Let π be our policy, then the value of π from state s:

$$v_\pi(s) = \mathbb{E}_\pi\left(G_t \mid S_t = s\right) = \mathbb{E}_\pi\left(R_t + \gamma R_{t+1} + \ldots \mid S_t = s\right) = \mathbb{E}_\pi\left(R_t + \gamma G_{t+1} \mid S_t = s\right)$$

The general case for the value of a policy at a given state is when we consider the probabilities of each action at that state, the probabilities of rewards and state transitions and the returns can be written as:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a)\left(r_{s,a,s'} + \gamma v_\pi(s')\right)$$

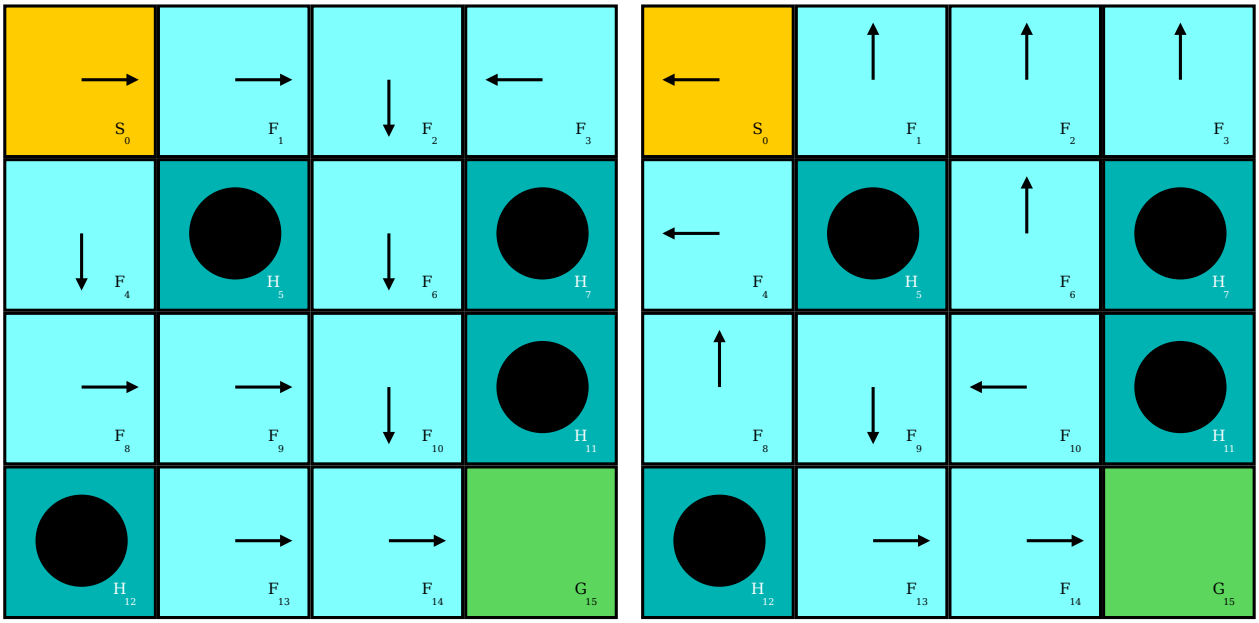


Figure 19.4: On the left, a risky and greedy policy that tries to move quickly towards the goal. On the right, a safe policy that avoids the danger of falling into holes but may take longer to reach the goal.

This is an example of Bellman's principle of optimality: "An optimal policy has the property that whatever the initial state and initial decision are, the remaining decisions must constitute an optimal policy with regard to the state resulting from the first decision."[5]. This type of equations are called Bellman equations.

In addition to knowing the return expected from our policy in any given state, we can also consider the return expected from an action in a given state assuming our policy. This is the action-value function Q:

$$q_\pi(s,a) = \mathbb{E}_\pi\left(G_t \mid S_t = s, A_t = a\right) = \mathbb{E}_\pi\left(R_t + \gamma G_{t+1} \mid S_t = s, A_t = a\right)$$

and the corresponding Bellman equation:

$$q_\pi(s,a) = \sum_{s'} P(s' \mid s,a)\left(r_{s,a,s'} + \gamma v_\pi(s')\right)$$

The action-advantage function tells us how much better a certain action a may be when compared to the expected return of following policy π:

$$a_\pi(s,a) = q_\pi(s,a) - v_\pi(s)$$

This gives us an intuition about how to optimize policies. Intuitively, if we find actions that have an advantage over what a policy recommends in some state, we can benefit by updating the policy to recommend that more profitable action. This leads us to consider the optimal state-value and action-value functions. We can say that policy π is better than policy π' if we obtain a better or equal return for all states. The optimal state-value function v_* is the one that maximizes the state-value for all states, and the optimal action-value function q_* is the one that gives us the best expected return for all states and action pairs:

$$v_*(s) = \max_\pi v_\pi(s), \forall s \in S \qquad q_*(s,a) = \max_\pi q_\pi(s,a), \forall s \in S, \forall a \in A$$

In other words, we obtain the optimal state-value and action-value functions by finding the best policy. Since the best policy gives us the best action at each state, intuitively we can also think of the optimal state-value function as the maximum over the actions:

$$v_*(s) = \max_a \sum_{s'} P(s' \mid s, a) \left(r_{s,a,s'} + \gamma v_*(s')\right)$$

We can do the same for the optimal action-value function q_*, but note that this function, also determined by the policy, gives us the maximum returns for all state, action pairs. Thus:

$$q_*(s, a) = \sum_{s'} P(s' \mid s, a) \left(r_{s,a,s'} + \gamma \max_{a'} q_*(s', a')\right)$$

or, intuitively, the optimal action-value function gives us, for the pair s, a, the largest expected return of all possible subsequent states given action a assuming that we choose the best actions afterwards.

Also note that $\max_{a'} q_*(s', a')$ will be the same as $v_*(s')$, because the optimal value at s' must be equal to the maximum return we can obtain by choosing the best action. This means that if we know v_* we can compute q_* by looking for the value of each state that follows each action.

19.5 Iterative Policy Evaluation

We know that we need to find the policy that maximizes v_π, but how can we obtain v_π? One solution is the iterative policy evaluation algorithm. We start by initializing $v_\pi(s)$ arbitrarily for all states, except for terminal states, for which $v_\pi(s)$ is zero. Then we use the policy-evaluation equation:

$$v_{\pi,k+1}(s) = \sum_a \pi(a \mid s) \sum_{s'} P(s' \mid s, a) \left(r_{s,a,s'} + \gamma v_{\pi,k+1}(s')\right), \qquad \forall s \in S$$

This converges to v_π as k increases. Let us compare our two policies for navigating the frozen lake. First, we need the MDP for the lake environment, so we can compute the transition probabilities and rewards for all states and actions. We can easily obtain this from the Open AI Gym library. First, install it if you do not have it yet:

```
pip install gym
```

Now we can obtain the complete MDP like this:

```
import gym
P = gym.make('FrozenLake-v0').env.P
```

This is a dictionary that has, for each state, for each action, the state transition probabilities and rewards, also indicating if the state is terminal:

```
In : P = gym.make('FrozenLake-v0').env.P
In : P
Out:
{0:
  {0:[(0.3333333333333333, 0, 0.0, False),
      (0.3333333333333333, 0, 0.0, False),
      (0.3333333333333333, 4, 0.0, False)],
```

```
  1: [(0.3333333333333333, 0, 0.0, False),
      (0.3333333333333333, 4, 0.0, False),
      (0.3333333333333333, 1, 0.0, False)],
...
 14:
  {0: [(0.3333333333333333, 10, 0.0, False),
       (0.3333333333333333, 13, 0.0, False),
       (0.3333333333333333, 14, 0.0, False)],
   1: [(0.3333333333333333, 13, 0.0, False),
       (0.3333333333333333, 14, 0.0, False),
       (0.3333333333333333, 15, 1.0, True)],
...
```

To evaluate a policy, we need a dictionary `pi` that returns the action for state s. This assumes we use a deterministic policy. If we use a probabilistic policy, then we would have to consider all possible actions weighted by their probabilities.

Under the assumption of a deterministic policy π, we can compute v_π like this[1]: loop through all the states, for each state get the appropriate action and all possible states that follow from that action and add the expected reward plus the discounted expected return of future actions for nonterminal states. Note that terminal states have no future actions, so the expected return of future actions in those cases is zero. We repeat this for all states until convergence, which is determined by the values varying very little from iteration to iteration. Here is the function to compute v_π where the policy π is represented by a dictionary mapping from states to actions (it is a deterministic policy):

```
def policy_evaluation(pi, P, gamma=1.0, theta=1e-10):
    prev_V = np.zeros(len(P))
    while True:
        V = np.zeros(len(P))
        for s in range(len(P)):
            for prob, next_state, reward, done in P[s][pi[s]]:
                V[s] += prob * (reward + gamma * prev_V[next_state] * (not done))
        if np.max(np.abs(prev_V - V)) < theta:
            break
        prev_V = V.copy()
    return V
```

Figure 19.5 is a comparison of our two policies, the greedy and safe policies, with the state values using a discount factor of $\gamma = 0.99$:

Generalized Policy iteration

By computing the state-value function (or V-function) v_{π_n} for each policy π_n we can compare policies and decide which is best. But generating policies at random is not a practical way of finding the best one. Fortunately, we have a better way. The action-value function (or Q-function) gives us the expected value for each action in each state:

$$q_\pi(s,a) = \sum_{s'} P(s' \mid s,a)\left(r_{s,a,s'} + \gamma v_\pi(s')\right)$$

[1] Code based on an example from Morales [48]

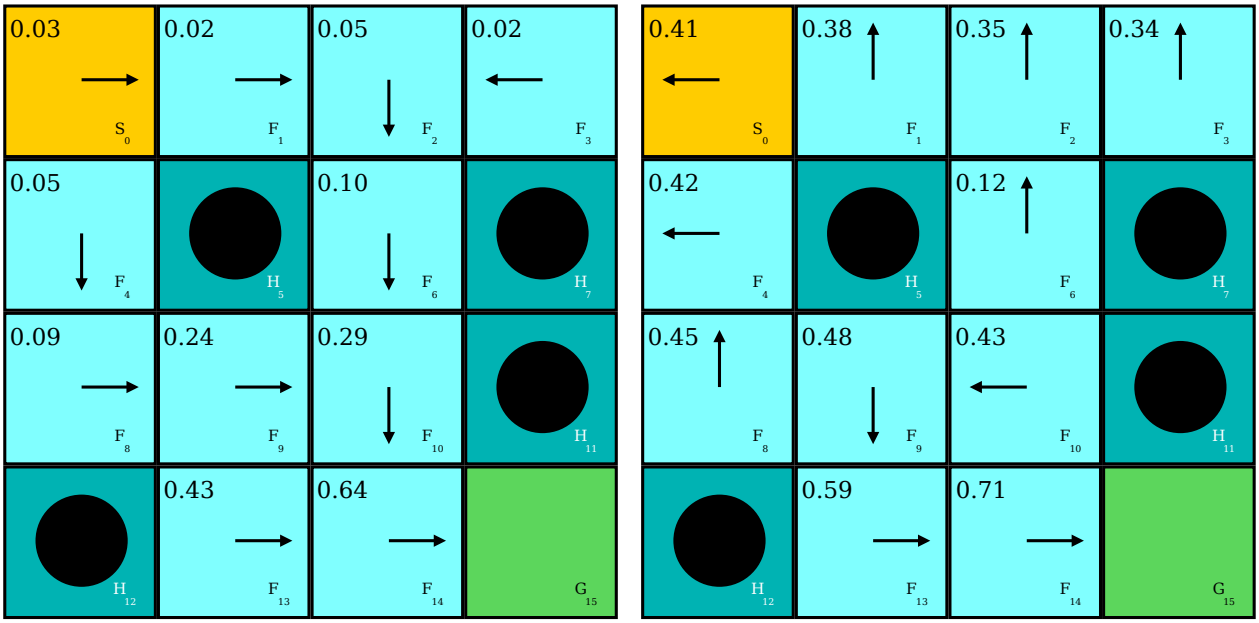


Figure 19.5: The greedy policy on the left is not as good as the safe policy on the right, as we can see from the state values for each state.

and we can compute $q_\pi(s, a)$ for each action a in state s from the V-function for that policy, v_π. Using this, we can adjust the policy so that the action taken at state s is now the one that maximizes the value. This is the policy-improvement equation, another example of a Bellman equation:

$$\pi'(s) = \underset{a}{\operatorname{argmax}} \sum_{s'} P(s' \mid s, a) \left(r_{s,a,s'} + \gamma v_\pi(s')\right)$$

So we can improve a policy given its V-function by computing the Q-function and them choosing actions that correspond to the maximum of the Q-function at each state:

```
def policy_improvement(V, P, gamma=1.0):
    Q = np.zeros((len(P), len(P[0])), dtype=np.float64)
    for s in range(len(P)):
        for a in range(len(P[s])):
            for prob, next_state, reward, done in P[s][a]:
                Q[s,a] += prob * (reward + gamma * V[next_state] * (not done))

    new_pi = {s:a for s, a in enumerate(np.argmax(Q, axis=1))}
    return new_pi
```

If we do this iteratively, we end up with the optimal policy:

```
def policy_iteration(P, gamma=1.0, theta=1e-10):
    random_actions = np.random.choice(tuple(P[0].keys()), len(P))
    pi ={s:a for s, a in enumerate(random_actions)}
    while True:
        old_pi = pi
        V = policy_evaluation(pi, P, gamma, theta)
        pi = policy_improvement(V, P, gamma)
        if old_pi == pi:
            break
    return V, pi
```

Figure 19.6 compares the two previous policies and the best one obtained with this process.

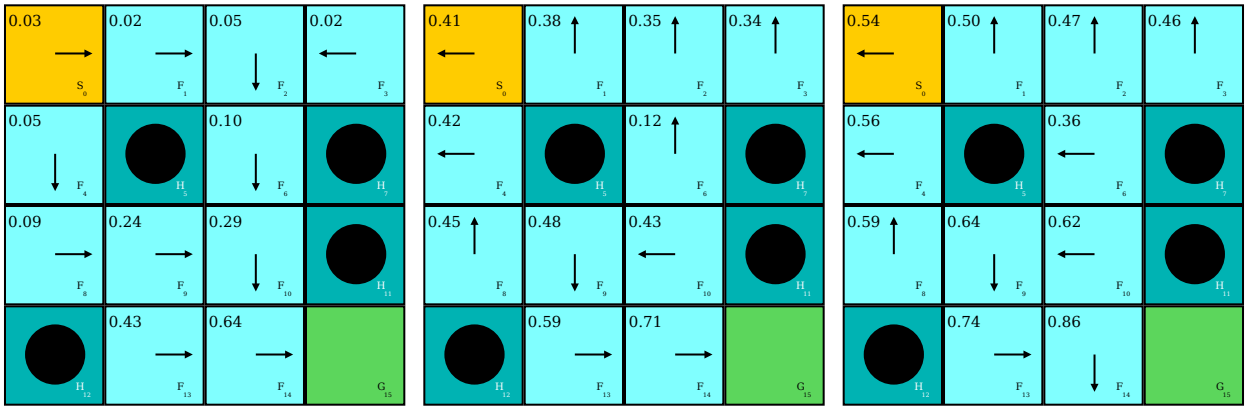


Figure 19.6: The greedy policy, the safe policy and the optimal policy for this problem.

19.6 Further Reading

1. (Optional) Miguel Morales, Grokking Deep Reinforcement Learning, 2020 [48], Chapters 1-3

Chapter 20

Deep Reinforcement Learning

Introduction to Deep Reinforcement Learning. Exploration and Exploitation. Learning policies with Deep Neural Networks

20.1 Introduction

In Chapter 19 we saw how, knowing the Markov Decision Process (MDP), we can compute the V-function giving the expected (discounted) reward for each state under some policy, and the Q-function, giving the expected rewords for each action in each state given a policy. Using these functions, we can improve the policy until we find the optimal policy.

Unfortunately, this requires full knowledge of the MDP, complete information about the states and the possibility of enumerating all combinations of states and actions. In many cases this is not possible. For example, in the inverted pendulum problem the state is composed of continuous variables with the position and orientation of the pendulum. Games like Chess, Go or Breakout have huge numbers of states. Figure 20.1 shows two examples of problems where we cannot use the methods covered in Chapter 19.

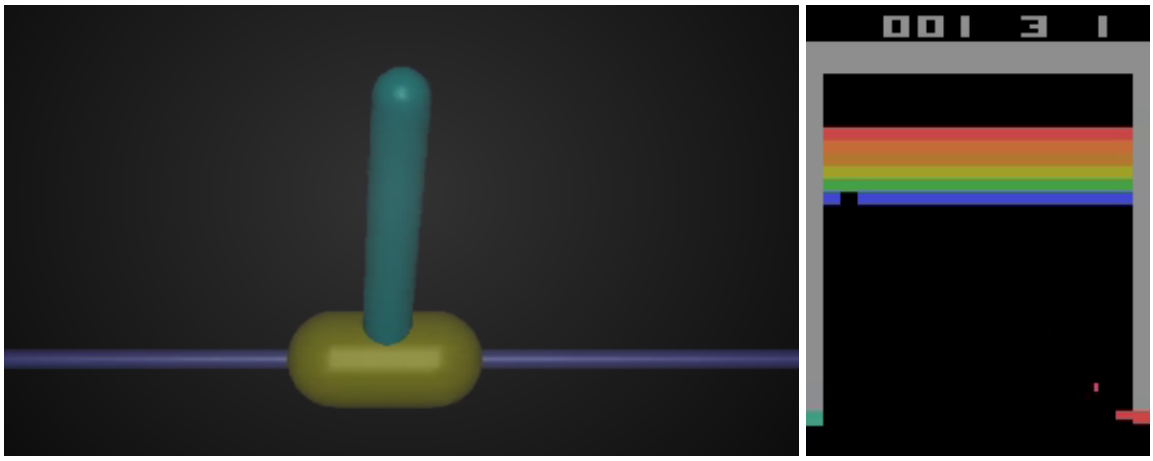


Figure 20.1: Illustration of the inverted pendulum problem and the Atari Breakout game. Images from Open-AI Gym.

However, we can adapt the same basic ideas and expand reinforcement learning with the power of deep learning. This is Deep Reinforcement Learning (DRL), and is characterized by these features:

- Delayed feedback Since we cannot consider the complete set of possible states and actions and must rely on sequences of events, it is often the case that feedback for the decisions our agents make is delayed.
- Evaluative feedback While in supervised learning we have a target value to predict, in reinforcement learning we only have relative evaluations of different outcomes. We can find that some option is better than another but we do not have access to the optimum value.
- Sampled feedback Since we do not know the complete decision problem, we can only use samples of actions, states and rewards. This means that the information we have will depend on how we explore the state and action space.
- Feature extraction Though not always, it is common in deep reinforcement learning that the observation of the system does not immediately provide the best features to describe the state. In a chess game we can consider the positions of the pieces but when training an agent to play Starcraft or drive a car, we must start from images or other unstructured data from sensors and then extract the appropriate features to inform the agent.

20.2 Exploration and Exploitation

In a DRL problem the agent needs to interact with the environment and observe the results in order to gather data and feedback about the consequences of its actions. The questions is how the agent does this. If the agent has some estimate about what the best action is at each state, it can choose that action and find better rewards. However, if the policy followed is pure exploitation, taking advantage of what the agent learned, the agent will never try new actions and thus may miss better alternatives. Thus, it is also necessary to balance exploitation with exploration where the agent acts differently from what seems to be the best option to try to find better policies. This balance can be achieved in several ways. The following are some examples:

ϵ-greedy exploration Most of the time follow what the agent estimates to be the best action but with a small probability ϵ take a random action instead.

Decaying ϵ-greedy exploration Same as ϵ-greedy exploration but ϵ is high at the beginning of training and declines during training to a small value (but greater than zero to always allow for some exploration).

Optimistic initialization In this strategy we initialize the Q-function with high values, so that during learning the values will tend to be adjusted down. This means the agent will prefer to explore combinations of actions and states that were less visited before.

Softmax exploration For each state, our Q-function will estimate the reward of each action. We can use those values as input to a softmax function that will convert them in a probability of selecting each action, giving greater weight to those with a higher Q-value.

Upper Confidence Bound (UCB) UCB favours less visited combinations of states and actions but with a more conservative approach than optimistic initialization. Using a confidence bound c, UCB computes a weighted sum of the Q-function estimate and the uncertainty due to the number of times $N_t(a)$ this action was tried before.

$$a_t = \underset{a}{\text{argmax}} \left(Q_t(s,a) + c\sqrt{\frac{2\ln t}{N_t(a)}} \right)$$

20.3 Learning Policies

The method we saw in Chapter 19, of iteratively updating our V-function estimates, cannot be used without knowing the full MDP, so in deep reinforcement learning we need a different approach. One possibility is to use a Monte Carlo exploration of the action and state space, sampling at random, and then averaging to find the expected values for each state and fill in our $v_\pi(s)$ function. However, this may not be very efficient.

A better approach is to use temporal-difference learning. Recall that the state-value function at state s under policy π is the expected reward obtained in the next state plus the discounted expected return at the next state. But the expected return can be estimated by v_π. Thus we can update our state-values like this:

$$v_\pi(s_t) = \mathbb{E}_\pi \left(R_{t+1} + \gamma v_\pi(s_{t+1}) \right)$$

Thus, if we want to improve our V-function gradually, we can use the observed reward at state s_{t+1} and the estimated discounted return with a learning rate to improve our estimates of the state-values. Assuming the policy π, the next value for $v_\pi(s)$ is:

$$v_{t+1}(s_t) = v_t(s_t) + \alpha_t \left(R_{t+1} + \gamma v_t(s_{t+1}) - v_t(s_t) \right)$$

This way we gradually adjust the estimates to the target value $R_{t+1} + \gamma v_t(s_{t+1})$.

We can do the same for the action values, or Q-function. The state-action-reward-state-action algorithm, or SARSA algorithm, uses the reward obtained by action a_t on state s_t plus the discounted Q value of the next step, $Q(s_{t+1}, a_{t+1})$.

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) + \alpha_t \left(R_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}) - q_t(s_t, a_t) \right)$$

SARSA considers the estimate for the future action values to be given by the action the policy will choose at the next step, a_{t+1}. But, as we saw before, we can improve the Q-function by considering the action that maximizes the return. This is the Q-learning algorithm, and the update is:

$$q_{t+1}(s_t, a_t) = q_t(s_t, a_t) + \alpha_t \left(R_{t+1} + \gamma \underset{a}{\text{argmax}} \left(q_t(s_{t+1}, a) \right) - q_t(s_t, a_t) \right)$$

This is why SARSA is considered an on-policy algorithm, because it follows the policy, while Q-learning is off-policy since it computes the expected future return using the maximum value and not the value for the action proposed by the policy.

These learning algorithms assume that we have a V table and a Q table that we can update. This is not the case in deep reinforcement learning, since we cannot store all combinations in a table.

20.4 Training a Deep Neural Network as a Q-function

In DRL we need to approximate the Q-function, because we cannot store all state, action pairs. To do this we use a deep neural network, that can be trained both to extract the appropriate features from the

observation of the system and approximate the Q-function for all state, action pairs. This DNN will receive as input the observation of the state s and the output will be the estimates of the action values for all actions in state s, or $q(s, a) \forall a \in A$. Thus we need one output neuron for each possible action, with linear activation in order to approximate the Q-function. Since this is a regression problem, we use a MSE loss function (or similar, as we will see below).

To train our network we will need to gather experiences, which we may think of as consisting of tuples with the state, action, next state, the reward obtained and if we reached a terminal state: $(s_t, a_t, s_{t+1}, r_{t_1}, terminal)$. With this we can adapt the SARSA or Q-learning algorithms to create training examples for our network. Each training example will consist of the observation of state s as input and the target values for the output. In the case of SARSA:

$$y_i^{SARSA}(a_t) = R_{t+1} + \gamma q_t(s_{t+1}, a_{t+1}; \theta_i)$$

where $q_t(s_{t+1}, a_{t+1}; \theta_i)$ is the predicted value given by our network for action a_{t+1} in the next state s_{t+1}. For Q-learning we use

$$y_i^{Q-learn}(a_t) = R_{t+1} + \gamma \underset{a}{\text{argmax}} \left(q_t(s_{t+1}, a; \theta_i)\right)$$

in other words, the maximum (discounted) value of any action in state s_{t+1} as estimated by our network. It is important to note that this will only affect the prediction of the action a_t, so the y_i vector should have all the other values as output by the network.

20.5 Demo: solving the cartpole problem

This demo[1] illustrates deep Q-learning plus some tricks that can be used to optimize training.

The problem is the Open-AI CartPole-V1, which consists in applying a force of +1 or -1 to a cart on a rail on which a pole is balanced. The pole must be kept within 15° of the vertical and the cart within 2.4 units from the centre (see Figure 20.2).

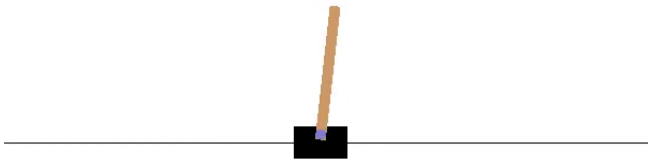


Figure 20.2: The cartpole problem. Image from Open-AI Gym.

We will be using a neural network to learn the state, action values (the Q-function), and a double queue (`deque`) to store a pool of examples for training.

[1] Based on a tutorial by Mike Wang, https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc

```
import gym
import tensorflow as tf
import numpy as np
from tensorflow import keras
from collections import deque
import random

RANDOM_SEED = 5
tf.random.set_seed(RANDOM_SEED)

env = gym.make('CartPole-v1')
env.seed(RANDOM_SEED)
np.random.seed(RANDOM_SEED)
```

Full training will be for 300 episodes, where one episode corresponds to a full "game" until the pole falls or the cart gets too far from the centre.

```
train_episodes = 300

def agent(state_shape, action_shape):
    """ The agent maps X-states to Y-actions
    e.g. The neural network output is [.1, .7, .1, .3]
    The highest value 0.7 is the Q-Value.
    The index of the highest action (0.7) is action #1.
    """
    learning_rate = 0.001
    init = tf.keras.initializers.HeUniform()
    model = keras.Sequential()
    model.add(keras.layers.Dense(24, input_shape=state_shape,
                                 activation='relu',
                                 kernel_initializer=init))
    model.add(keras.layers.Dense(12, activation='relu',
                                 kernel_initializer=init))
    model.add(keras.layers.Dense(action_shape,
                                 activation='linear',
                                 kernel_initializer=init))
    model.compile(loss=tf.keras.losses.Huber(),
                  optimizer=tf.keras.optimizers.Adam(lr=learning_rate),
                  metrics=['accuracy'])
    return model
```

Two optimizations in this model are the He uniform initializer, which uses a uniform distribution ranging from minus to plus 6 divided by the square root of the number of inputs to the layer, and the Huber loss function, which is similar to the quadratic error loss function for small values but increases linearly for larger absolute values, so it penalises outliers less than the MSE loss.

Now we need the training function:

```
def train(env, replay_memory, model, target_model, done):
    discount_factor = 0.618
    batch_size = 64 * 2
    mini_batch = random.sample(replay_memory, batch_size)
    current_states = np.array([transition[0] for transition in mini_batch])
    current_qs_list = model.predict(current_states)
```

```
54      new_current_states = np.array([transition[3] for transition in mini_batch])
55      future_qs_list = target_model.predict(new_current_states)
56      X = []
57      Y = []
58      for index, (observation, action, reward, new_observation, done) in enumerate(mini_batch):
59          if not done:
60              max_future_q = reward + discount_factor * np.max(future_qs_list[index])
61          else:
62              max_future_q = reward
63          current_qs = current_qs_list[index]
64          current_qs[action] = max_future_q
65          X.append(observation)
66          Y.append(current_qs)
67      model.fit(np.array(X), np.array(Y), batch_size=batch_size, verbose=0, shuffle=True)
```

This function randomly samples a batch of experiences from the memory queue, which acts as a pool of experiences for training. Each experience consists of a tuple

```
(observation, action, reward, new_observation, done)
```

containing the previous observation, the action taken by the agent, the reward for the new state, the observation of the new state and a flag indicating whether the new state is a terminal state. The Q-values for the previous states are those predicated by the main model. The Q-values for the new states are predicted by a target model that lags behind the main model. Basically, every 100 steps the weights from the main model are copied to the target model and the target model is not trained. This helps stabilize the target for learning since the estimates of future returns do not change constantly as the main model is changed.

The features X matrix is composed of the observation of the current states and the targets Y matrix is computed by combining the reward of the future state with the estimated return from the future state assuming the action with maximum estimated return will be taken at that future state. This is Q-learning. Note that for terminal states the future return is zero and only the reward is considered. Also note that the future return is discounted by the discount factor γ.

And now the main function:

```
69 def main():
70      epsilon = 1
71      max_epsilon = 1
72      min_epsilon = 0.01
73      decay = 0.01
74      MIN_REPLAY_SIZE = 1000
75
76      model = agent(env.observation_space.shape, env.action_space.n)
77      target_model = agent(env.observation_space.shape, env.action_space.n)
78      target_model.set_weights(model.get_weights())
79
80      replay_memory = deque(maxlen=50000)
81      steps_to_update_target_model = 0
```

We start by setting up the constants for the ϵ-greedy exploration with a decaying ϵ, create the model and copy the weights to the target model. The pool of examples will be kept in a double queue with a

maximum of 50 thousand examples. This object automatically discards older examples if new examples are inserted over the maximum capacity.

Now we start the training loop:

```
    for episode in range(train_episodes):
        total_training_rewards = 0
        observation = env.reset()
        done = False
        while not done:
            steps_to_update_target_model += 1
            if True:
                env.render()
            random_number = np.random.rand()
            if random_number <= epsilon:
                action = env.action_space.sample()
            else:
                reshaped = observation.reshape([1, observation.shape[0]])
                predicted = model.predict(reshaped).flatten()
                action = np.argmax(predicted)
            new_observation, reward, done, info = env.step(action)
            replay_memory.append([observation, action, reward, new_observation, done])
```

This part loops through the actions for each episode and decides whether to explore, with random actions, or exploit the knowledge of the agent. For each action, the experience is stored in the memory queue.

```
            if len(replay_memory) >= MIN_REPLAY_SIZE and \
                (steps_to_update_target_model % 4 == 0 or done):
                train(env, replay_memory, model, target_model, done)
            observation = new_observation
            total_training_rewards += reward
            if done:
                print('Rewards: {} after n steps = {} with final reward = {}'.format(
                          total_training_rewards, episode, reward))
                total_training_rewards += 1

                if steps_to_update_target_model >= 100:
                    print('Copying main network weights to the target network weights')
                    target_model.set_weights(model.get_weights())
                    steps_to_update_target_model = 0
                break
        epsilon = min_epsilon + (max_epsilon - min_epsilon) * np.exp(-decay * episode)
    env.close()
```

The model is trained only once there are enough examples in the experience pool. Once that condition is met, the model is trained every four steps and the target model is updated with a new copy of the model weights every 100 steps. This process is repeated until the end of the episode, at which time the ϵ value is updated and another episode is run.

20.6 Further Reading

1. (Optional) Miguel Morales, Grokking Deep Reinforcement Learning, 2020 [48], Chapters 5, 6,

and 8

Chapter 21

Interpreting deep models

Motivation for explanations. Problems with black box models. Explanations. LIME. LRP. TCAV.

21.1 Motivation for Explainable AI

Suppose you want to organize your vacation photos and use a deep neural network to classify all photos that contain faces, so you can separate photos of you and your friends from photos of buildings, landscapes and such. In this case, what matters is that the network is accurate and manages to correctly classify most photos.

But now imagine that a surgeon uses a deep neural network to process radiological images and recommend a therapy and the network recommends removing the patient's left kidney. Even if the network was very accurate on the test set, the surgeon cannot remove the patient's kidney just because the network outputs this result. Without an explanation for why the kidney should be removed the network and its recommendations will not be very useful.

There are many situations where a black box algorithm is undesirable, giving the motivation for transparency so that humans can understand how the decisions are made. Here are some examples (from [77]):

- Helping improve a system or correct errors, since debugging black box systems is much harder.
- Trust that the system is working correctly. Without understanding how it works this is difficult.
- For social acceptance of such systems, especially those that impact daily life (*e.g.* targeted advertising, filtering of messages in social media, etc).
- Ensure that a decision was reached correctly (*e.g.* credit rating, selection of job candidates).
- Auditing the system if something goes wrong (*e.g.* an accident involving an autonomous vehicle)
- For regulation, such as safety standards.
- For greater impact (*e.g.* an automated recommendation to buy a product may be more influential if the system can explain why the product is recommended).

Identifying problems

One important reason for peeking into the black box is to identify errors. Figure 21.2 illustrates an example where a classifier relied on the copyright tag to classify images of horses. This could pass unnoticed by the developers of they did not try to interpret what the classifier was doing.

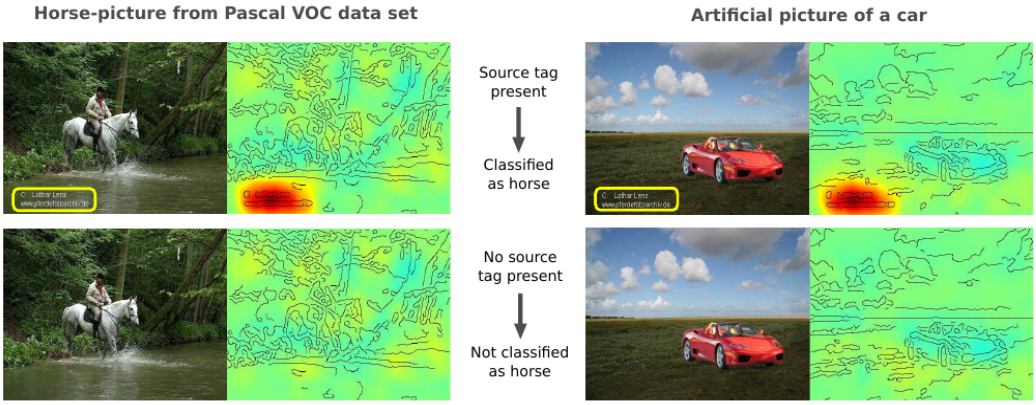


Figure 21.1: This Fisher vector classifier relies on the copyright tag to classify horse images in the PASCAL VOC 2007. Image from [36]

Trust and Accountability

Explanations are an important part of human interactions and of how we build trust. We do not need to understand how a machine works if we are using the machine. For example, a toaster or microwave oven. But if the machine is telling us what to do or what we can do, such as selecting job applications or recommending therapies, we need some justification in order to trust these recommendations.

Furthermore, transparency is required for accountability and oversight. Regulators, for example, need to understand how a system works to determine if it complies with requirements for safety, consumer protection, human rights, etc.

Scientific Applications

Machine learning, and deep learning in particular, is increasingly used to solve complex scientific problems. For example, the AlphaFold2 network, by DeepMind, beat all competition by a large margin in protein structure prediction, a very hard scientific problem that has been attracting researchers for half a century. But it would be very useful to understand what principles AlphaFold2 has discovered.

Legal requirements

The EU's General Data Protection Regulation (GDPR) includes the right to an explanation, although this right is stated in Recital 71 and not in the legally binding regulation. Furthermore, article 22 states that "The data subject shall have the right not to be subject to a decision based solely on automated processing", which means that any machine learning solution must be part of a process involving human decision. Since the person deciding will be responsible for the decision, the system should provide some explanation that justifies the recommended decision.

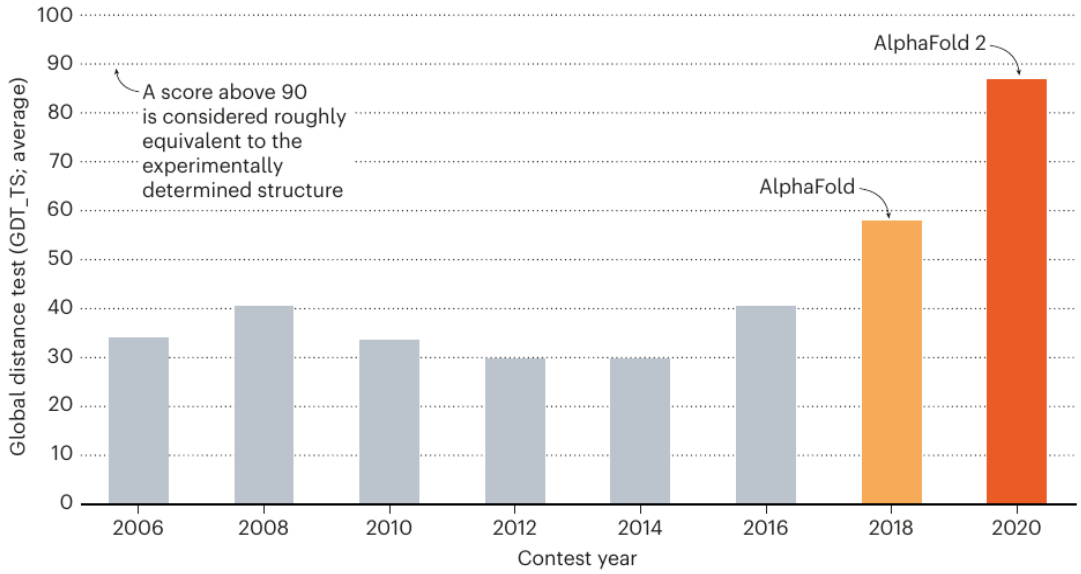


Figure 21.2: Performance of AlphaFold2 at the Critical Assessment of Protein Structure Prediction competition. Image from [11]

21.2 What is an explanation?

In a broad sense, an explanation is a narrative that links together different events or entities in a way that makes something intelligible, describing causal relations, consequences and the system explained. But for our purpose we are more interested in practical aspects of an explanation. In short, how useful it is.

This depends on the target audience. The best explanation for helping a surgeon understand the output of a network is different from the best explanation that would help the patient understand the recommendation. The developers of deep networks who want explanations to help debug and optimize their models need to understand different aspects that the end user who is deciding whether or not to trust the output of the network. So when considering the application of interpretability methods to make deep networks more transparent we must take into account who will benefit from those explanations.

The explanation may also focus on interpreting different aspects of the network or data. We may want to understand how the network learned to represent the data internally, obtain a justification for how the network labelled a specific example, identify which aspects of the data that the network considers more important. There are many different ways of interpreting deep neural networks and the techniques used should be chosen according to what we want to explain and who the target of the explanation is.

Local Interpretable Model-agnostic Explanations (LIME)

.

Some simple machine learning models are easy to interpret. A decision tree with few branches, for example, is understandable by a human (although if it has many rules it stops being intelligible). A paradigmatic example of interpretable models are linear models. Since in linear models the coefficients corresponding to each feature will be proportional to the importance of that feature for the prediction, it is easy to understand how the model works. However, linear models are of limited use in many real life problems, and when we introduce nonlinearity the importance of different features now depends on the example being classified and interpretation becomes much harded.

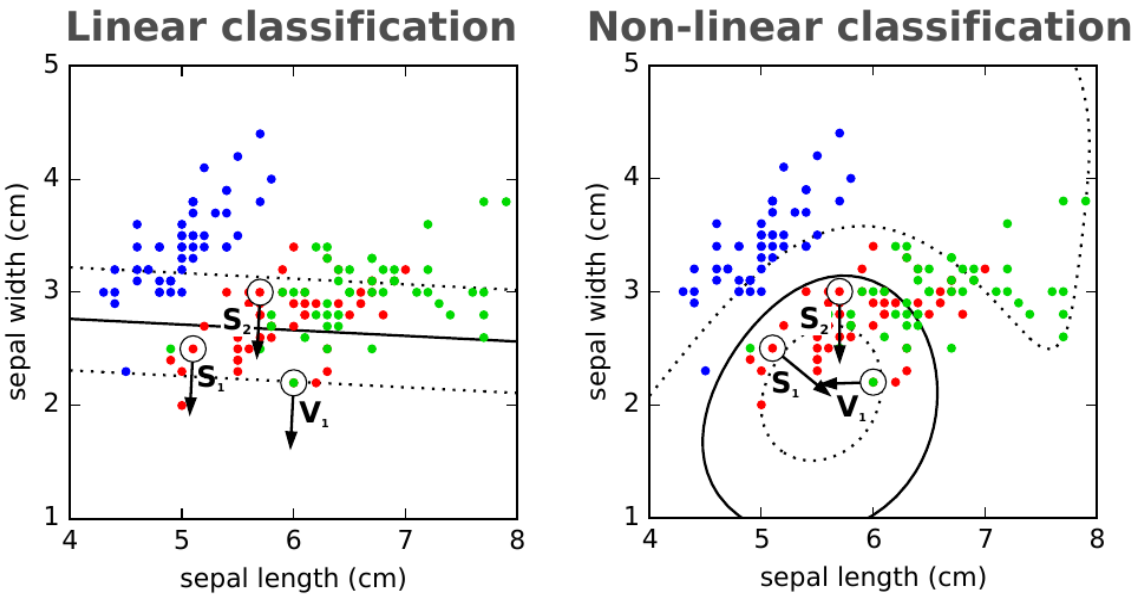


Figure 21.3: Linear models are easy to interpret because the weights indicate the relative importance of different features. With nonlinear models the importance of different features depends on the example. Image from [36]

But even if the data set, globally, requires nonlinearity for adequate predictions, it may be that locally this nonlinear decision surface can be approximated with a linear model. This is the idea of the Local Interpretable Model-agnostic Explanations (LIME) method [56]. To explain the prediction for a given example, LIME gathers nearby examples in the dataset, weighted according to the distance to the example to explain, and then fit a local linear model to that neighbourhood. This allows LIME to propose an explanation for the prediction for a single example. Figure 21.4 illustrates this. On the right panel, it shows the results of two kernel SVM trained to classify posts in a discussion group as christian or atheistic. Even though both SVM classify this example correctly as atheistic, one seems to rely on undesirable features such as the presence of terms like "by", "Re" or "Nntp".

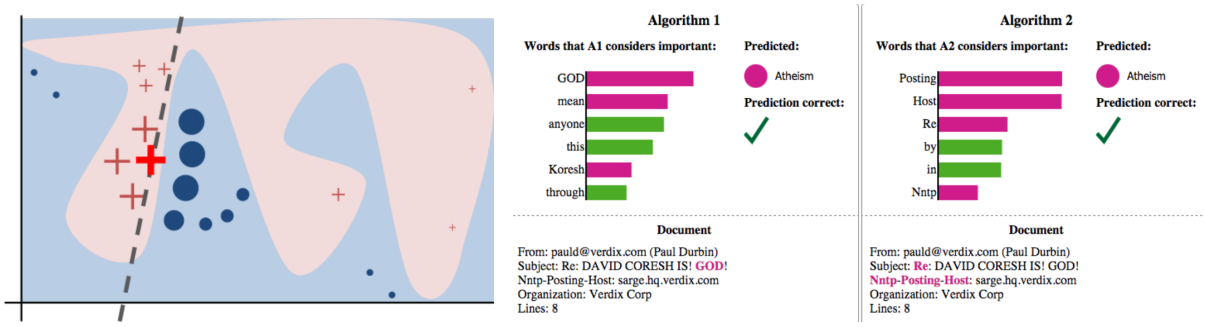


Figure 21.4: LIME gathers examples in the neighbourhood of the example to explain and builds a linear model for a local explanation (left panel). This can them be used even with nonlinear models, such as kernel SVM. In the right panel the linear local model is used to identify the features that two models appear to be using to classify an example. Images from [56]

LIME does not provide a global explanation of the model being explained. Rather, it provides a local explanation around a particular example. And it is model-agnostic because it does not need to consider how the model works; it simply approximates its output around the example. Formally, LIME provides an explanation with the following minimization:

$$\xi(x) = \underset{g \in G}{\text{argmin}} L(f, g, \pi_x)\Omega(g)$$

where g is a linear model from the set of all linear models using any combination of features, L measures the loss between the model to be explained f and the explainer model g in some neighbourhood π of point x and $\Omega(g)$ is a measure of the complexity of model g so that, by becoming too complex, g does not lose interpretability.

21.3 Layer-wise Relevance Propagation (LRP)

Layer-wise Relevance Propagation (LRP)[2] is a technique to assign to each input of a deep neural network classifier a relevance measure for a particular output. Unlike LIME, this interpretation method takes into account the architecture and parameters of the trained model. The relevance of the output neuron is its activation. For example, the activation of the neuron corresponding to the chosen class. Then the relevance of each neuron j will be the sum for all neurons k in the following layer of their relevance R_k multiplied by the product of the activation of j and the weight connecting j to each neuron k divided by the sum of these products for all neurons in the previous layer:

$$R_j = \sum_k \frac{a_j w_{jk}}{\sum_i a_i w_{i_k}} R_k$$

This can be propagated all the way to the inputs. Figure 21.6 illustrates this.

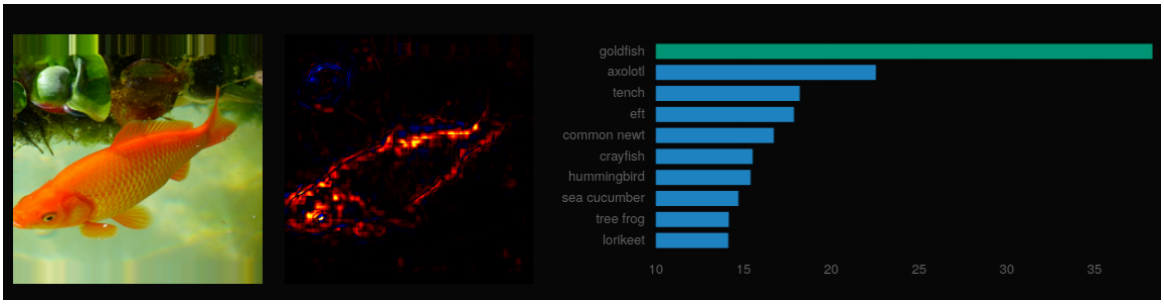


Figure 21.5: LRP propagates the relevance for a particular class (in this case fish) to each pixel in the input. Image from Explainable AI demos at https://lrpserver.hhi.fraunhofer.de/

21.4 Testing with Concept Activation Vectors (TCAV)

Testing with Concept Activation Vectors (TCAV) [33] tries to interpret the sensitivity of classifications to selected concepts using a set of examples that match some concept and a set of examples that do not match that concept. First, we gather the activations of all neurons at a layer l in the network for all these examples and fit a binary linear classifier using as input these activation vectors. For example, if we have a network trained to identify zebras among other classes, we may use a set of example images with striped patterns and another set without striped patterns and fit a binary linear classifier on the activations of some layer l. The vector normal to the hyperplane separating the two classes, which is computed by the classifier, is the Concept Activation Vector (CAV) for these examples. Each CAV will be a feature of the network, and independent of any particular example.

Now we can measure how sensitive the classification of example x in class k is at layer l for a concept C by taking the derivative of the activation of the output neuron for class k with respect to the activations at layer l measured in the direction of the CAV for that concept C. In other words, we are seeing how much moving in the direction of concept C, along the vector v_C^l at layer l, will affect the output of the network for class k:

$$S_{C,k,l}(x) = \lim_{\epsilon \to 0} \frac{h_{l,k}\left(f_l(x) + \epsilon v_C^l\right) - h_{l,k}\left(f_l(x)\right)}{\epsilon}$$

Figure **??** illustrates what we need for this process:

(a) A set of chosen examples to illustrate a concept, such as striped, and a set of random examples.

(b) A set of labelled examples of some class from the training data, such as zebras.

(c) The network trained to classify examples into several classes that include the zebra class.

(d) Compute the vector normal to the decision surface using the activations of layer l and a linear binary classifier that distinguishes the concept and random examples (the CAV).

(e) The sensitivity of layer l to this concept for this class for some example is the derivative of the activation of the zebra class output neuron with respect to the activations of layer l changed in the direction of the CAV for concept C.

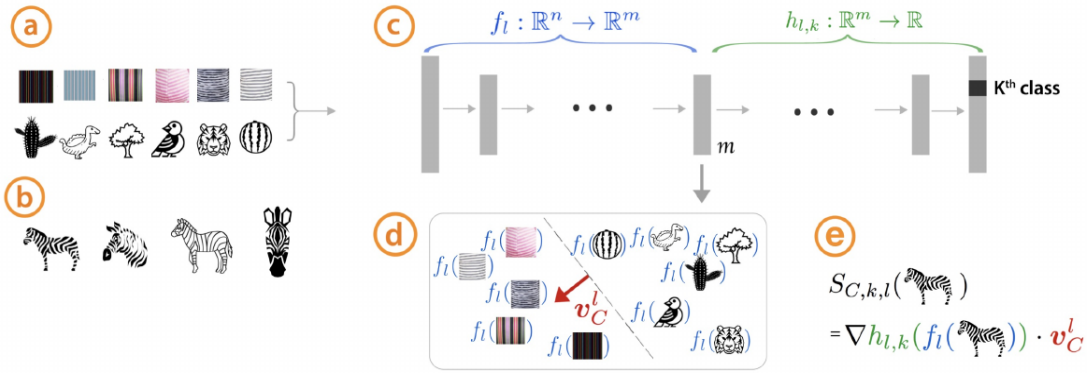


Figure 21.6: The TCAV process. Image from [33]

21.5 Mapping concepts to ontologies

Recent work on deep model interpretation takes this idea of concepts identification further by mapping from the activations of the network to concepts in an ontology [55]. An ontology is a formal specification of concepts and their logical relations. For example, the shapes and colours of traffic signs follow specific rules: a STOP sign is an octagon, has a red background and the word "STOP" written in white; a warning sign is a triangle with a red border, and so on. With an ontology we can formally specify the background knowledge experts can use to understand how an example is being classified.

Given a trained deep neural network and examples illustrating different concepts in the ontology, we can create auxiliary models, such as very simple neural networks, that map from neuron activations to these concepts in the ontology. This is similar to TCAV but without the constraint that the mapping

model needs to be a linear model. And an additional advantage is that, since the target concepts are part of an ontology, it is possible to use automated reasoning algorithms to extract justifications from these mappings. For example, if the network identified a STOP sign and we find that the activations match the mappings to concepts of red background and octagon shape, an automated inference process may identify that these concepts are part of the requirements for a STOP sign according to the ontology.

This approach also allows us to evaluate the network itself and its correspondence to our background knowledge as encoded in the ontology. If the internal representations are capturing these relations, it is easy to map from the network activations to relevant concepts in the ontology even if the network was trained to find other classes. Conversely, concepts that are not relevant will not be properly represented in the network and thus are harder to map. Figure 21.7 shows these mappings from a network trained to identify four types of traffic signs. Relevant concepts like having a black bar, a red background or a triangular shape are well encoded in the network activations and easy to map accurately. In contrast, concepts like being blue or having a pole do not seem to be adequately represented in the network. Unlike most interpretation methods, this gives us insight not only on how the network is classifying some specific examples but also on properties of the network itself.

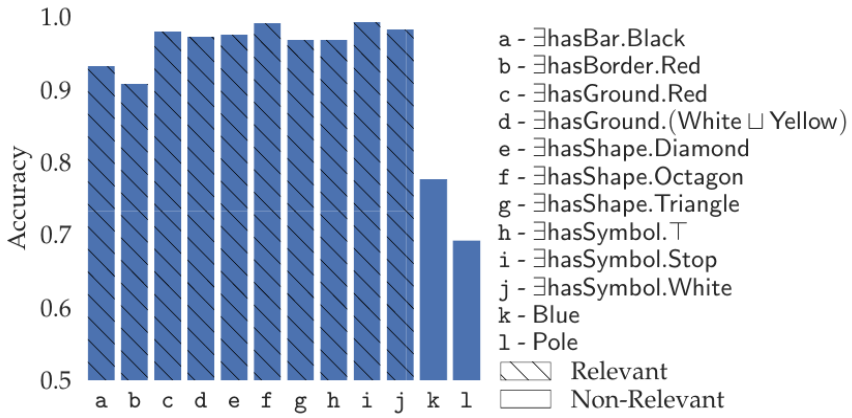


Figure 21.7: Accuracy when mapping from network activations to a selection of relevant and irrelevant examples. Image from [55]

21.6 Risks of (partial) transparency

Although helpful in many cases, interpretations and explanations that claim to make the model transparent create some risks when the interests of those who would benefit from the explanations are not aligned with the interests of those who provide them. Suppose my request for credit at the bank is denied and I ask for an explanation. The bank will apply interpretation methods to the neural network used to decider not to grant me credit and may choose those interpretations that best serve the bank's purpose. This ability to select interpretations may end up defeating the purpose of explainability and transparency. Maybe we need to implement transparency in deep learning models as part of fully transparent procedures to mitigate these problems.

Chapter 22

Bias and Fairness

Bias and its ethical problems. Sources of bias. Mitigating undesirable biases.

22.1 Bias

It is not possible to generalize from examples without assuming something about our hypotheses. In machine learning this is known as *inductive bias* and is an unavoidable feature of any learning system, including ourselves. When we categorize things we make some assumptions that simplify the categories and allow us to generalize. For example, that bananas are usually yellow. So if we find yellow bananas we just call them bananas. There is no need to point out that they are yellow, because that is assumed. Only if we find blue bananas, like the Blue Java banana variety, do we feel the need to mention the colour.

Figure 22.1: A bunch of bananas and a bunch of blue bananas. Note that the in the case of the former we do not need to specify the colour.

Although inductive bias is a fundamental part of learning, whether machine or biological learning, in some cases stereotypes and implicit assumptions may have undesired consequences. Ethical concerns force us to pay attention to these problems, especially since machine learning is becoming increasingly influential in our society. For example, an evaluation of commercial facial recognition applications by the National Institute of Standards and Technology in the US found that the false match rate is 68 times higher for American Indian women than for white men. It was also 47 times higher for American

Indian men and 10 times higher for black women[23]. This has worrying implications due to the use of such algorithms in law enforcement and job application background checks.

22.2 Fairness

It is important to distinguish between bias that we can legitimately use to improve our predictions, such as the correlation between high levels of blood sugar and diabetes, and bias that we should not use to make decisions that have impact on other humans. The portuguese Constitution, for example, explicitly prohibits discrimination in function of characteristics like sex, race, religious beliefs, political ideology or sexual orientation. Evidently, when developing models that have impact on people's lives we must not include such features.

This is not easy to do with unstructured data. Suppose we train a deep neural network to classify photos of people to distinguish between those that are CEO of important companies and those that are not, and we get 95% accuracy in our test set. This may seem a good result, but only about 5% of the CEO of large companies are women. If the classifier simply discards all women as negative examples it could still have a 95% accuracy on the whole data set but with 0

This leads us to an intuitive notion of fairness in machine learning: we want our systems to work equally well for all relevant categories and not just as best as possible on the aggregate. If these categories are important for ethical or legal reasons, this may even force us to accept an overall reduction in accuracy. Figure 22.3 illustrates this.

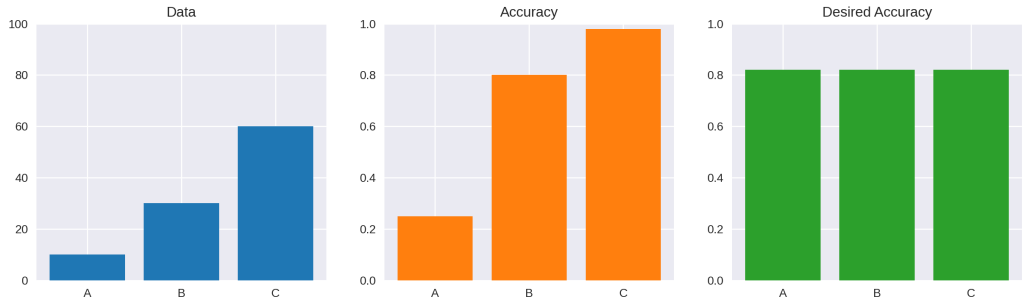


Figure 22.2: Imbalanced data may lead to different accuracy values in different categories. Ideally, we want to have the same accuracy over all categories.

In short, if we are developing machine learning models that will have impact on persons we need to be aware of aspects in which bias is not desirable. This is not to say that we need to eliminate all bias, as that would be impossible and make learning impossible, but that we need to be careful about what correlations in the data our system is using for its predictions when those predictions have a significant impact on other people.

22.3 Sources of Bias

Bias can have different sources. Bias in data can easily arise due to inadequate sampling. For example, the ImageNet and Open Images datasets are used to train state of the art object recognition networks but the majority of their images come from the USA and from European countries, with very little representation of many geographic regions. This makes some categories strongly biased towards western norms. As a result, networks trained with these datasets may become biased. For example,

when predicting the `bridegroom` category from images of grooms from western countries, the network performs well. Figure **??** shows the log-likelihood of correctly classifying images of grooms from the USA and Australia. But

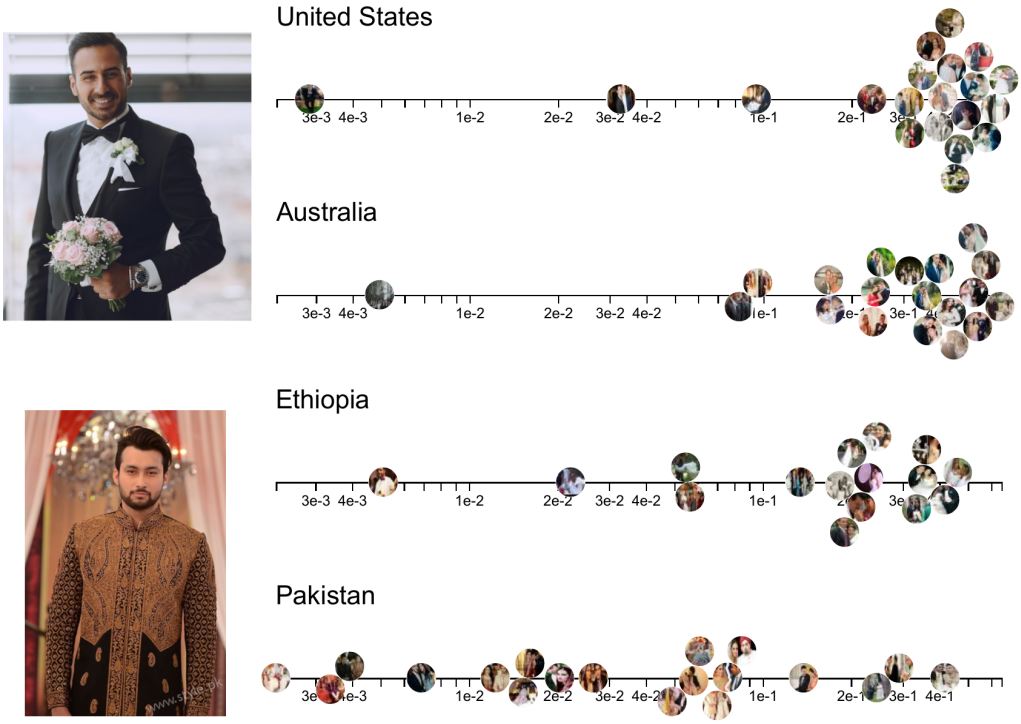


Figure 22.3: ImageNet is biased towards western examples and this makes networks trained in this data set perform poorly when classifying images from other countries. Images adapted from [63].

Bias can be also appear in data simply because reality is itself biased. For example, gender imbalances in some professions, like nursing, construction, engineering or sociology, cannot be solved by random sampling because the population from which any sample is drawn is itself imbalanced. These problems require other solutions.

Bias can also be caused by the way we select the features to use. For example, men make up the vast majority of violent criminals. Models for risk estimations used in law enforcement and courts should obviously not use the sex of the subjects because it is unethical to discriminate based on this characteristic, regardless of what correlations there are in the population. But we must also be aware of other characteristics that may provide the model with indirect means of undesirable discrimination, such as height and weight, which are strongly correlated with sex.

Finally, undesirable biases my occur because of the way we aggregate data. In medicine, genetic risk factors for some diseases may be more or less significant depending on the ethnicity of the patient. For example, haemoglobin A1c, a glycated form of haemoglobin, is used as an indicator for blood glucose levels and diabetes. However, evidence shows that different ethnic groups have significantly different base levels of haemoglobin A1c [27], which means that any procedure based on the average level for the population will perform differently in different groups.

22.4 Mitigating undesirable bias

There are different ways of mitigating undesirable bias, depending on the source of these biases we wish to reduce.

Sampling Bias

If the undesirable bias comes from problems in sampling the population, these can be best solved by improving the quality of the data. The Pilot Parliaments Benchmark [10] is an example. It is a face recognition data set with images drawn from parlamentarians from 6 countries in Africa and Europe. Figure 22.4 illustrates this data set.

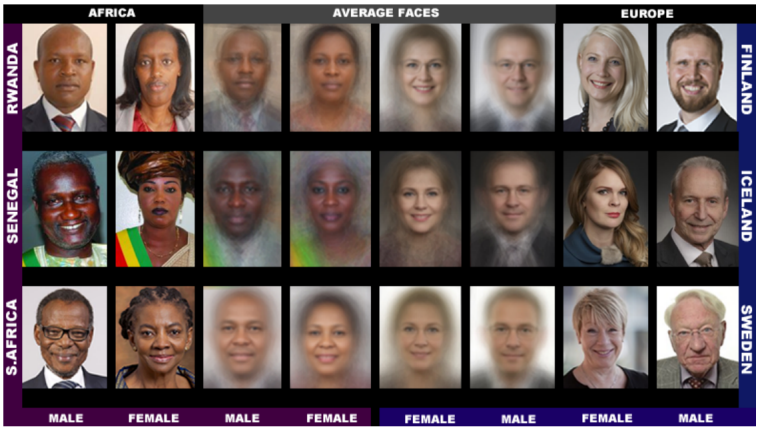


Figure 22.4: The Pilot Parliaments Benchmark. Image from [10].

Resampling

If the undesirable bias is in the population itself and not merely a sampling problem, then we cannot solve it with representative sampling. In this case, we must alter the sampling in order to compensate the data imbalances. For example, there are very few male nurses and this correlation will persist on any representative sample of this profession. If we want to remove this correlation, we will have to oversample male nurses or undersample female nurses to compensate for this imbalance. This is the idea behind the Rpresentation Bias Removal (REPAIR) algorithm [41].

Suppose we have a deep neural network classifier trained on a data set. We can think of the last layer of the network, with the softmax activation, as a linear classifier that uses the features extracted by all the previous layers. Using a cross-entropy loss function to encourage the output of the softmax to approximate the probability of the class given the features, the loss function for the dataset D and parameteres θ will be:

$$L(D,\theta) = \mathbb{E}\left(-\log P(Y \mid X)\right) = -\frac{1}{|D|}\sum_{(x,y)\in D} \log P(y \mid x)$$

We can consider that the bias is the reduction in uncertainty about the class normalized by the entropy of the class distributions of the data:

$$B(D,\theta) = 1 - \frac{L(D,\theta)}{H(Y)} \qquad H(Y) = -\frac{1}{|D|}\sum_{(x,y)\in D} \log p_y$$

REPAIR attempts to change the sampling of the dataset to minimize the bias for the best possible classification. To do this, it assigns weights to examples to influence the probability of being selected for training. With weights, the equations above become

$$L(D', \theta) = -\frac{1}{\sum_{i=1}^{|D|} w_i} \sum_{i=1}^{|D|} w_i \log P(y_i \mid x_i)$$

$$B(D', \theta) = 1 - \frac{L(D', \theta)}{H(Y')} \qquad H(Y') = -\frac{1}{\sum_{i=1}^{|D|} w_i} \sum_{i=1}^{|D|} w_i \log \frac{\sum_{i:y_i=y} w_i}{\sum_i w_i}$$

And now the weights are computed by minimizing $L(D', \theta)$ with respect to θ while also minimizing $B(D', \theta)$ with respect to the weights that change the distribution in D'. This second minimization will be the equivalent of maximizing the ratio of the loss to the entropy of the classes, and thus favour both the undersampling of examples that make training too easy and the balancing of the classes to increase their entropy. In this way the authors propose to find weights to rebalance the data set by changing the way it is sampled in order to minimize bias.

Unsupervised Unbiasing

Another example of a method to reduce bias by resampling is to use a variational autoencoder to learn a latent representation with probability distributions along different components of the manifold. The authors call it the Debiasing Variational Autoencoder [1]. Figure 22.5 illustrates the architecture. The output z_0 is used for the classification task, in a supervised manner exactly like in a deep neural classifier. The remaining output of the encoder is the set of parameters for the normal distributions describing the latent space. These are learned in an unsupervised manner, as usual with a VAE, by minimizing the reconstruction error of the input and the Kullback-Leibler divergence for the latent space. These losses are weighted with the cross-entropy loss for the classification.

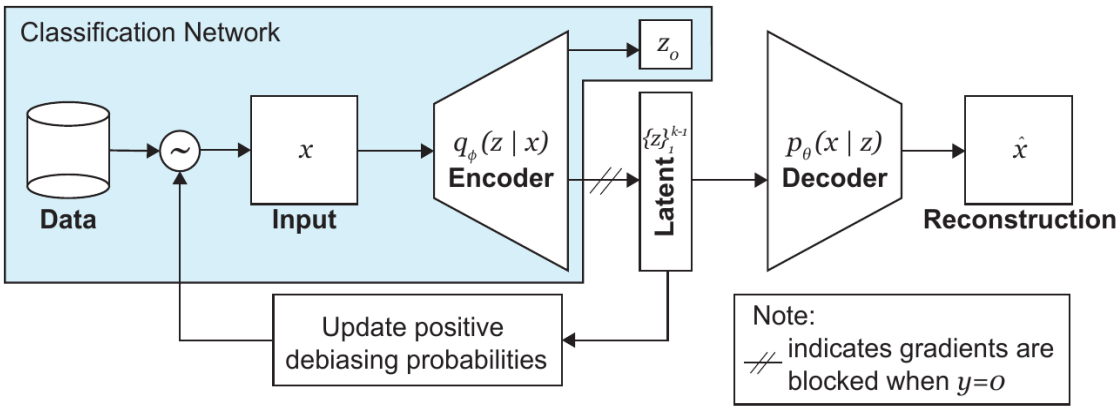


Figure 22.5: The Debiasing Variational Autoencoder. Image from [1].

The training set is then passed through the encoder and the latent representations are used to build histograms with the distribution of points in the latent space. These are then used to rescale the sampling probability by the inverse of the probability in the latent space, thus giving greater weight to points that represent less visited regions of the latent space. Figure 22.6 illustrates the result. Images identified by the VAE as being in a more crowded region of the latent space are given lower resampling probabilities, while outliers are given a higher resampling probability.

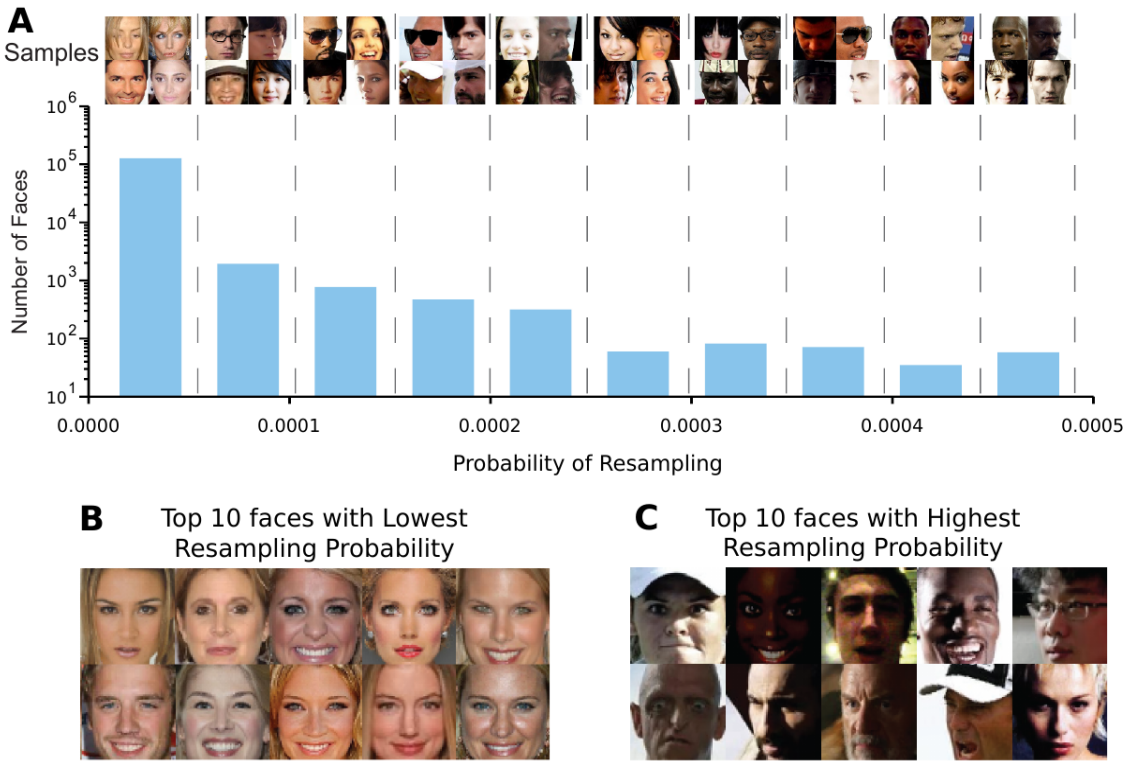


Figure 22.6: Resampling probabilities computed from the histograms describing the distribution of images in the latent space. Image from [1].

22.5 Conclusion

Biases are fundamental for learning, because they are those correlations that allow us to generalize from examples. But some biases are undesirable, whether because they are due to sampling problems or incorrect assumptions or because even if those correlations exist, we do not want our systems to depend on them for ethical or legal reasons. In such cases it is important to be able to identify and mitigate such biases. It is thus important to follow best practices of transparency, documentation and representation in data curation, critically examine data sources for undesired biases and be aware of method to mitigate such problems when they are identified.

Chapter 23

(Some) Open Problems in Deep Learning

AutoML and Neural Architecture Search. Validation and Verification of Artificial Neural Networks. Large data sets. The neuro-symbolic gap.

23.1 Neural Architecture Search

We solve machine learning problems by selecting the best algorithms and models. But what if machine learning did this for us? This is the idea of Automated Machine Learning, or AutoML. Ideally, we would present the computer with a problem and some algorithm would select the best solution for this problem. Currently there are several commercial and free frameworks that attempt this with a set of classical machine learning algorithms. For example, the `auto-sklearn`[1] library uses the machine learning algorithms available in Scikit-learn to automate solving classification problems by selecting algorithms and hyper-parameters [19].

In classical machine learning automation faces the additional problems of data preparation, feature engineering and feature selection, which are crucial for classical ML algorithms, since these are very dependent on the features chosen. Deep neural networks seem less dependent on these aspects, since the network can learn to extract the more useful features. So it would seem that neural networks would be a prime candidate for automated machine learning. In fact, the idea of automating neural network design is not new [76]. However, success has been limited and this is still an open problem.

One difficulty is in finding the right strategy for optimizing a network architecture. Unlike optimizing the weights, this is a discontinuous problem for which we cannot compute the derivatives of the target function. We can evaluate each network by measuring its performance but cannot compute the derivative of the performance with respect to changing parts of the network. This has led to several different strategies:

- Evolutionary methods, such as genetic algorithms, can be used to simulate populations of artificial networks and, through recombination and mutation, "evolve" better architectures. These have been used for decades now but the problem is that they tend to be inefficient, requiring the evaluation of a large number of networks.
- Reinforcement learning is a more recent approach that has shown promise. For example, we can imagine that the agent is trying to design a network by choosing a sequence of actions about layer

[1] https://automl.github.io/auto-sklearn/master/

and connections until the network is built, receiving rewards that are a function of the network's performance.

- Bayesian optimization tries to maximize a black box function f without assuming anything about the function by gathering from function evaluations to update some prior. For example, Sequential Model-based Algorithm Configuration (SMAC) [29] is a bayesian optimisation method that evaluates the performance of combinations of configuration parameters for some model and uses the results to train a random forest to predict what performance new combinations would have. This guides the selection of new combinations to test and de data gathered improves the predictions. Using this method it has been possible to automate the generation of multilayer perceptrons that outperform human-designed networks in some cases [46].

Another problem with automating network design is the evaluation of each candidate architecture. Deep neural networks can be expensive to train, so different strategies are used to try to speed up their evaluation[17]

- Train for a few epochs or on small subsets of data.
- Extrapolate performance from first training epochs.
- Inherit weights from previous model transforming the architecture in ways that can be used to preserve weights. For example, we can add a new layer with weights initialized so that it performs an identity operation, but now these weights are free to learn new transformations[12].
- Share weights between different models that are subsets of a large original model.

But perhaps the biggest problem with automated generation of network architectures is the search space. We can search through different architectures of multilayer perceptrons, or add convolution networks and such. But there does not seem to be a unified representation of network architectures that would allow such a search to discover new types of networks. For example, that could discover convolution networks, or residual blocks, or recurrent networks, attention and transformers, and so on, before we had an idea of such options to include them in the search.

23.2 Validation and Verification

Software validation and verification is important in general but particularly important in mission critical software, for which failures cannot be tolerated. Given the power of deep models to solve problems such as autonomous driving, medical diagnosis, credit risk prediction and such applications where failure may have significant consequences, validating and verifying deep neural models would be highly desirable. But both steps are difficult to perform with deep neural models.

Validation is the assessment of the conformity of the software to the requirements specification. In other words, it answers the question "is the software being built correctly?". This is usually done during development to check the implementation according to the requirements. A meaningful application of this principle to deep neural networks is challenging because, other than simply estimating the network's performance on some test data, it is not easy to assess conformity to some specifications or even decide on what specifications to use.

Verification is the assessment of the adequacy of the software to the use it will be put to. In other words, it answers the question "Is the right software being built?". This is also challenging because of the need to specify the conditions and actual performance requirements for the application.

Borg *et al.* identified some challenges and tentative solutions for the validation and verification of deep neural networks.

- The very large state-space for the data and network responses makes it difficult to estimate how the network can respond in anomalous situations. Such problems have resulted in fatal accidents in testing autonomous vehicles. The solutions for this tend to focus on probabilistic models or process control methods, based on established limits for proper operation.
- Test specification is another challenge. It is not easy to specify adequate tests for deep neural networks if we want to evaluate the response under anomalous data, for example. Genetic algorithms and other forms of test data generation are attempted solutions to this problem.
- Formal methods are used for validating and verifying software for critical applications. By formally describing the specifications and the algorithms it is possible to automate the proof of correctness and that the software will perform as specified. However, this is not a trivial task even in software engineering; its application to deep neural networks is especially difficult.

23.3 Dataset Size

One salient feature of deep neural networks is their requirement of large data sets. In part this is understandable because the models have many parameters. But we know that our own neural networks, even though highly complex, seem to learn with far fewer examples. And the usefulness of deep models could be greatly increased if they could be trained with smaller data sets. There are several techniques that can help with this:

- Few-shot and one-shot learning uses networks trained on large data sets that can then be used in smaller datasets with little or no additional training. An important aspect of this approach is *metric learning*, in which the network is trained to create a representation of the examples such that examples that are semantically similar are close together and those that are different are farther apart. This can then generalize to different data sets and even new categories as long as the representations are still adequate.
- Data augmentation can be used in some cases to mitigate the effects of small data sets. This is applicable if we know how to generate new examples adequately.
- Regularization can be used to mitigate overfitting. Deep neural networks are very powerful models that can overfit easily with small data sets but techniques like dropout or weight penalizations can be used to reduce these effects without losing the power of the network to adequately represent the structure of the data.
- The loss function used can have a large impact on the ability to learn from small data sets. For example, Barz and Denzler report a 30% improvement in accuracy by using a cosine loss function instead of the usual categorical cross-entropy loss for classification problems with small data sets [4]

23.4 Bridging the neuro-symbolic gap

Neural networks are simultaneously very basic algorithms but very complex. They are basic because they consist in the composition of very simple operations. Essentially, sums, products and a few nonlinear functions. But they are complex because they compose a very large number of these basic elements. In this they are analogous to our own brains, which inspired these networks. The response of individual neurons is comparatively simple when we consider the complexity of the brain.

In our brains the sub-symbolic representations learned by the neural networks are somehow connected to parts of our brain capable of symbolic reasoning. We can describe what we see and understand what others tell us. For example, if one person learns to pick out pictures of boats from a large stack of photographs, it is easy to ask them to pick out yellow boats only. It may take some additional effort at first to remember that the boat must be yellow and then gradually improve with practice, but this symbolic part can inform the sub-symbolic classifier.

Can we do this with neural networks? Mapping from network activations to concepts in order to interpret what the network is doing seems to be possible. But doing it in the opposite direction would be a major step towards true artificial intelligence.

Bibliography

[1] Alexander Amini, Ava P Soleimany, Wilko Schwarting, Sangeeta N Bhatia, and Daniela Rus. Uncovering and mitigating algorithmic bias through learned latent structure. In *Proceedings of the 2019 AAAI/ACM Conference on AI, Ethics, and Society*, page 289–295, 2019.

[2] Sebastian Bach, Alexander Binder, Grégoire Montavon, Frederick Klauschen, Klaus-Robert Müller, and Wojciech Samek. On pixel-wise explanations for non-linear classifier decisions by layer-wise relevance propagation. *PloS one*, 10(7):e0130140, 2015.

[3] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473*, 2014.

[4] Bjorn Barz and Joachim Denzler. Deep learning on small datasets without pre-training using cosine loss. In *Proceedings of the IEEE/CVF Winter Conference on Applications of Computer Vision*, page 1371–1380, 2020.

[5] Richard Bellman. *Dynamic Programming*. Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

[6] Yoshua Bengio, Aaron Courville, and Pascal Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[7] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. A training algorithm for optimal margin classifiers. In *Proceedings of the fifth annual workshop on Computational learning theory*, page 144–152. ACM, 1992.

[8] Olivier Breuleux, Yoshua Bengio, and Pascal Vincent. Quickly generating representative samples from an rbm-derived process. *Neural computation*, 23(8):2058–2073, 2011.

[9] Jason Brownlee. *Generative Adversarial Networks with Python: Deep Learning Generative Models for Image Synthesis and Image Translation*. Machine Learning Mastery, 2019.

[10] Joy Buolamwini and Timnit Gebru. Gender shades: Intersectional accuracy disparities in commercial gender classification. In *Conference on fairness, accountability and transparency*, page 77–91. PMLR, 2018.

[11] Ewen Callaway. 'it will change everything': Deepmind's ai makes gigantic leap in solving protein structures. *Nature*, 2020.

[12] Tianqi Chen, Ian Goodfellow, and Jonathon Shlens. Net2net: Accelerating learning via knowledge transfer. *arXiv preprint arXiv:1511.05641*, 2015.

[13] Djork-Arné Clevert, Thomas Unterthiner, and Sepp Hochreiter. Fast and accurate deep network learning by exponential linear units (elus). *arXiv preprint arXiv:1511.07289*, 2015.

[14] George Cybenko. Approximations by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2:183–192, 1989.

[15] Dheeru Dua and Casey Graff. UCI machine learning repository, 2017.

[16] John Duchi, Elad Hazan, and Yoram Singer. Adaptive subgradient methods for online learning and stochastic optimization. *Journal of Machine Learning Research*, 12(Jul):2121–2159, 2011.

[17] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. Neural architecture search: A survey. *Journal of Machine Learning Research*, 20(55):1–21, 2019.

[18] Dumitru Erhan, Yoshua Bengio, Aaron Courville, Pierre-Antoine Manzagol, Pascal Vincent, and Samy Bengio. Why does unsupervised pre-training help deep learning? *Journal of Machine Learning Research*, 11(Feb):625–660, 2010.

[19] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Springenberg, Manuel Blum, and Frank Hutter. Efficient and robust automated machine learning, 2015. *URL http://papers. nips. cc/paper/5872-efficient-and-robust-automated-machine-learning*.

[20] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, page 315–323, 2011.

[21] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[22] Klemen Grm, Vitomir Štruc, Anais Artiges, Matthieu Caron, and Hazım K Ekenel. Strengths and weaknesses of deep learning models for face recognition against image degradations. *IET Biometrics*, 7(1):81–89, 2017.

[23] Patrick J Grother, Mei L Ngan, and Kayee K Hanaoka. Face recognition vendor test part 3: demographic effects. 2019.

[24] Antonio Gulli, Amita Kapoor, and Sujit Pal. *Deep learning with TensorFlow 2 and Keras: regression, ConvNets, GANs, RNNs, NLP, and more with TensorFlow 2 and the Keras API*. Packt Publishing Ltd, 2019.

[25] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, page 770–778, 2016.

[26] Lisa Anne Hendricks, Zeynep Akata, Marcus Rohrbach, Jeff Donahue, Bernt Schiele, and Trevor Darrell. Generating visual explanations. In *European Conference on Computer Vision*, page 3–19. Springer, 2016.

[27] William H Herman and Robert M Cohen. Racial and ethnic differences in the relationship between hba1c and blood glucose: implications for the diagnosis of diabetes. *The Journal of Clinical Endocrinology & Metabolism*, 97(4):1067–1072, 2012.

[28] Geoffrey E Hinton. Learning multiple layers of representation. *Trends in cognitive sciences*, 11(10):428–434, 2007.

[29] Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration (extended version). *Technical Report TR-2010–10, University of British Columbia, Computer Science, Tech. Rep.*, 2010.

[30] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[31] Kevin Jarrett, Koray Kavukcuoglu, Yann LeCun, et al. What is the best multi-stage architecture for object recognition? In *2009 IEEE 12th International Conference on Computer Vision (ICCV)*, page 2146–2153. IEEE, 2009.

[32] Nan Jiang, Wenge Rong, Baolin Peng, Yifan Nie, and Zhang Xiong. An empirical analysis of different sparse penalties for autoencoder in unsupervised feature learning. In *2015 international joint conference on neural networks (IJCNN)*, page 1–8. IEEE, 2015.

[33] Been Kim, Martin Wattenberg, Justin Gilmer, Carrie Cai, James Wexler, Fernanda Viegas, and Rory Sayres. Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (tcav). *arXiv preprint arXiv:1711.11279*, 2017.

[34] Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[35] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, page 1097–1105, 2012.

[36] Sebastian Lapuschkin, Stephan Wäldchen, Alexander Binder, Grégoire Montavon, Wojciech Samek, and Klaus-Robert Müller. Unmasking clever hans predictors and assessing what machines really learn. *Nature communications*, 10(1):1–8, 2019.

[37] Yann Le Cun, Leon Bottou, and Yoshua Bengio. Reading checks with multilayer graph transformer networks. In *1997 IEEE International Conference on Acoustics, Speech, and Signal Processing*, volume 1, page 151–154. IEEE, 1997.

[38] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[39] Yann LeCun, Bernhard Boser, John S Denker, Donnie Henderson, Richard E Howard, Wayne Hubbard, and Lawrence D Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[40] Yann LeCun, Léon Bottou, Yoshua Bengio, Patrick Haffner, et al. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[41] Yi Li and Nuno Vasconcelos. Repair: Removing representation bias by dataset resampling. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, page 9572–9581, 2019.

[42] Jonathan Long, Evan Shelhamer, and Trevor Darrell. Fully convolutional networks for semantic segmentation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, page 3431–3440, 2015.

[43] Zhou Lu, Hongming Pu, Feicheng Wang, Zhiqiang Hu, and Liwei Wang. The expressive power of neural networks: A view from the width. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems 30*, page 6231–6239. Curran Associates, Inc., 2017.

[44] Laurens van der Maaten and Geoffrey Hinton. Visualizing data using t-sne. *Journal of machine learning research*, 9(Nov):2579–2605, 2008.

[45] Warren S McCulloch and Walter Pitts. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133, 1943.

[46] Hector Mendoza, Aaron Klein, Matthias Feurer, Jost Tobias Springenberg, and Frank Hutter. Towards automatically-tuned neural networks. In *Workshop on Automatic Machine Learning*, page 58–65. PMLR, 2016.

[47] Grégoire Mesnil, Yann Dauphin, Xavier Glorot, Salah Rifai, Yoshua Bengio, Ian Goodfellow, Erick Lavoie, Xavier Muller, Guillaume Desjardins, David Warde-Farley, et al. Unsupervised and transfer learning challenge: a deep learning approach. In *Proceedings of the 2011 International Conference on Unsupervised and Transfer Learning workshop-Volume 27*, page 97–111. JMLR. org, 2011.

[48] M. Morales. *Grokking Deep Reinforcement Learning*. Manning Publications, 2020.

[49] Martin Mächler. Accurately computing log (1- exp (-| a|)) assessed by the rmpfr package. Technical report, 2012.

[50] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, page 807–814, 2010.

[51] Josh Patterson and Adam Gibson. *Deep learning: A practitioner's approach*. " O'Reilly Media, Inc.", 2017.

[52] John D. Owens Purcell, David Luebke, Naga Govindaraju, Mark Harris, Jens Krüger, Aaron Lefohn, and Timothy J. A survey of general-purpose computation on graphics hardware. *Computer Graphics Forum*, 26(1):80–113, 2007.

[53] Sashank J Reddi, Satyen Kale, and Sanjiv Kumar. On the convergence of adam and beyond. 2018.

[54] Shaoqing Ren, Kaiming He, Ross Girshick, and Jian Sun. Faster r-cnn: Towards real-time object detection with region proposal networks. In *Advances in neural information processing systems*, page 91–99, 2015.

[55] Manuel António de Melo Chinopa de Sousa Ribeiro. Neural and symbolic ai-mind the gap! aligning artificial neural networks and ontologies. Master's thesis, 2021.

[56] Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, page 1135–1144, 2016.

[57] Salah Rifai, Yann N Dauphin, Pascal Vincent, Yoshua Bengio, and Xavier Muller. The manifold tangent classifier. In *Advances in Neural Information Processing Systems*, page 2294–2302, 2011.

[58] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. U-net: Convolutional networks for biomedical image segmentation. In *International Conference on Medical image computing and computer-assisted intervention*, page 234–241. Springer, 2015.

[59] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[60] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[61] Pierre Sermanet, David Eigen, Xiang Zhang, Michaël Mathieu, Rob Fergus, and Yann LeCun. Overfeat: Integrated recognition, localization and detection using convolutional networks. *arXiv preprint arXiv:1312.6229*, 2013.

[62] Wenling Shang, Kihyuk Sohn, Diogo Almeida, and Honglak Lee. Understanding and improving convolutional neural networks via concatenated rectified linear units. In *international conference on machine learning*, page 2217–2225, 2016.

[63] Shreya Shankar, Yoni Halpern, Eric Breck, James Atwood, Jimbo Wilson, and D Sculley. No classification without representation: Assessing geodiversity issues in open data sets for the developing world. *arXiv preprint arXiv:1711.08536*, 2017.

[64] Jianbo Shi and Jitendra Malik. Normalized cuts and image segmentation. *IEEE Transactions on pattern analysis and machine intelligence*, 22(8):888–905, 2000.

[65] Wenzhe Shi, Jose Caballero, Lucas Theis, Ferenc Huszar, Andrew Aitken, Christian Ledig, and Zehan Wang. Is the deconvolution layer the same as a convolutional layer? *arXiv preprint arXiv:1609.07009*, 2016.

[66] Edward H Shortliffe and Bruce G Buchanan. A model of inexact reasoning in medicine. *Mathematical biosciences*, 23(3-4):351–379, 1975.

[67] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*, 2014.

[68] Sandro Skansi. *Introduction to Deep Learning: From Logical Calculus to Artificial Intelligence*. Springer, 2018.

[69] Richard Socher, Milind Ganjoo, Christopher D Manning, and Andrew Ng. Zero-shot learning through cross-modal transfer. In *Advances in neural information processing systems*, page 935–943, 2013.

[70] Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *The Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[71] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, page 1–9, 2015.

[72] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, page 2818–2826, 2016.

[73] Matus Telgarsky. Benefits of depth in neural networks. *arXiv preprint arXiv:1602.04485*, 2016.

[74] Joshua B Tenenbaum, Vin De Silva, and John C Langford. A global geometric framework for nonlinear dimensionality reduction. *science*, 290(5500):2319–2323, 2000.

[75] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. *arXiv preprint arXiv:1706.03762*, 2017.

[76] E Vonk, Lakhmi C Jain, and Ray P Johnson. *Automatic generation of neural network architecture using evolutionary computation*, volume 14. World Scientific, 1997.

[77] Adrian Weller. Transparency: motivations and challenges. In *Explainable AI: Interpreting, Explaining and Visualizing Deep Learning*, page 23–40. Springer, 2019.

[78] David H Wolpert, William G Macready, et al. No free lunch theorems for optimization. *IEEE transactions on evolutionary computation*, 1(1):67–82, 1997.

[79] Xide Xia and Brian Kulis. W-net: A deep model for fully unsupervised image segmentation. *arXiv preprint arXiv:1711.08506*, 2017.

[80] Matthew D Zeiler and Rob Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, page 818–833. Springer, 2014.