# Chapter 8

# Multi-layer Perceptron

*Perceptron. Multi-layer Perceptron. Backpropagation. Regularization in MLP.*

## 8.1 Perceptron

Figure 8.1 shows a neuron cell. 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. The neuron provides the inspiration for the *perceptron*. Originally, the *perceptron* model consisted of a linear combination of the inputs, plus a bias value, and a non-linear threshold response function:

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

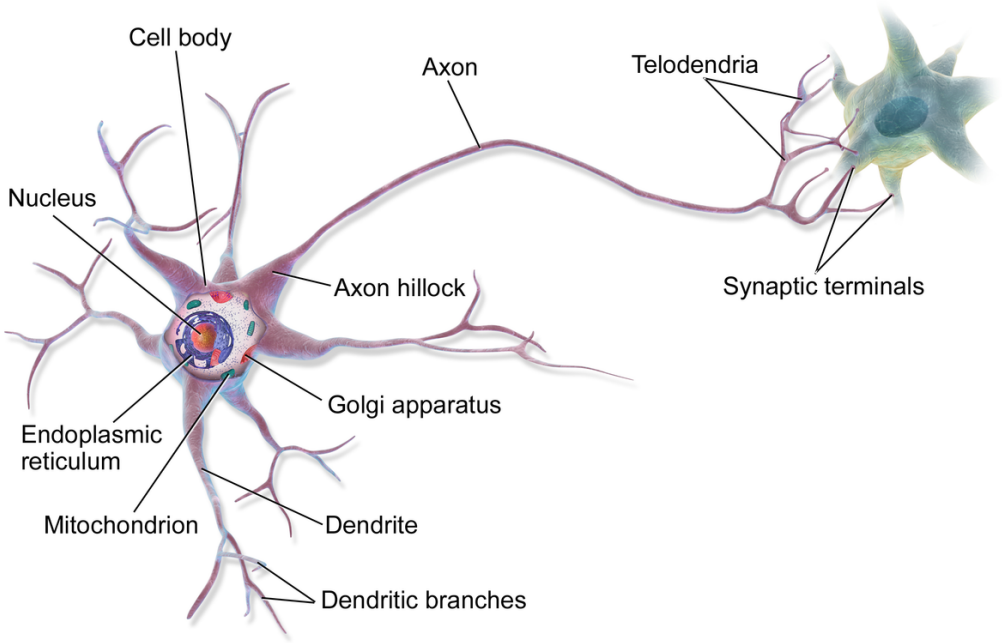


Figure 8.1: Neuron anatomy (BruceBlaus, CC-BY, source Wikipedia).

Note that, as we did in the case of logistic regression, we can include this bias value in the product of the inputs and the coefficients by adding a bias value of 1 to the input vector. The *perceptron* represents a hyperplane that separates the inputs into two classes, 0 and 1. To train a *perceptron*, we present labelled examples and adjust the weights 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. The problem with this 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 functions. An alternative is to use a differentiable threshold function. One often used function is the *logistic* function, also called the *sigmoid* function:

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

There are other functions that can be used in this role, such as the *hyperbolic tangent*, for example. However, here we will only focus on the familiar logistic function. Although this is strictly not the same as the perceptron, in the original formulation, it is also common to call this variant a perceptron too.

## 8.2 A Single Neuron

Training a logistic response neuron can be done by minimizing the squared error between the response of the perceptron and the target class. This is the idea behind the *Brier score* we saw in Chapter 5. So we minimize the error function:

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

But 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 8.2 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 $\Delta w_i^j$ updates for the whole training set (an epoch) and then updating the weights with the total. 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.

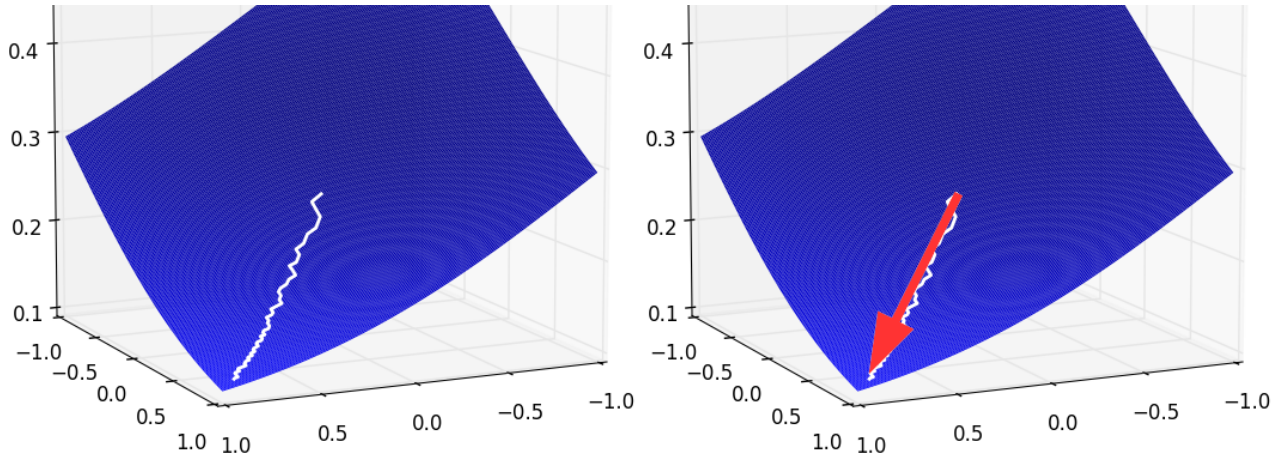


Figure 8.2: Stochastic gradient descent with online training (left panel) and batch training (right panel).

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 **??**.

Table 8.1: The OR function

| $x_1$ | $x_2$ | OR |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 1 |

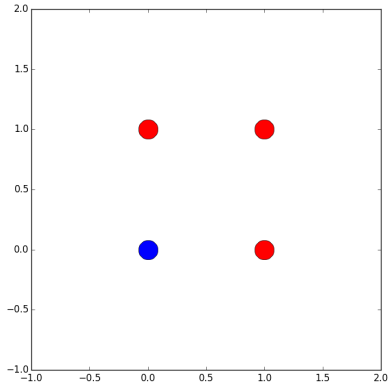

Figure 8.3: Set of points from the OR function.

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.

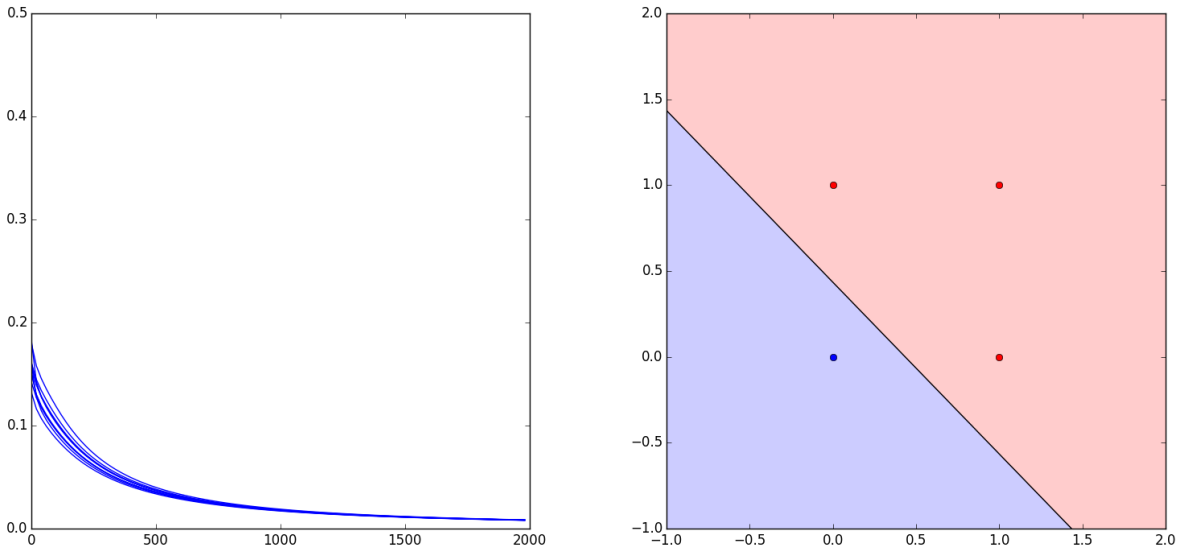

Figure 8.4: Training error and final classifier for one neuron trained to separate the classes in the OR function.

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 8.2 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 8.6.

Table 8.2: The XOR function

| $x_1$ | $x_2$ | XOR |
|---|---|---|
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

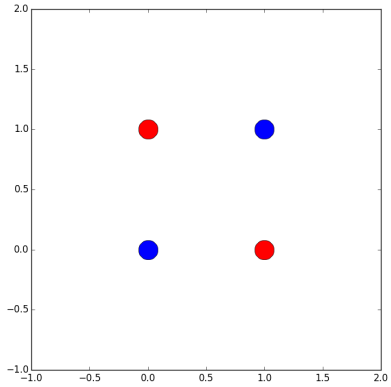

Figure 8.5: Set of points from the OR function.

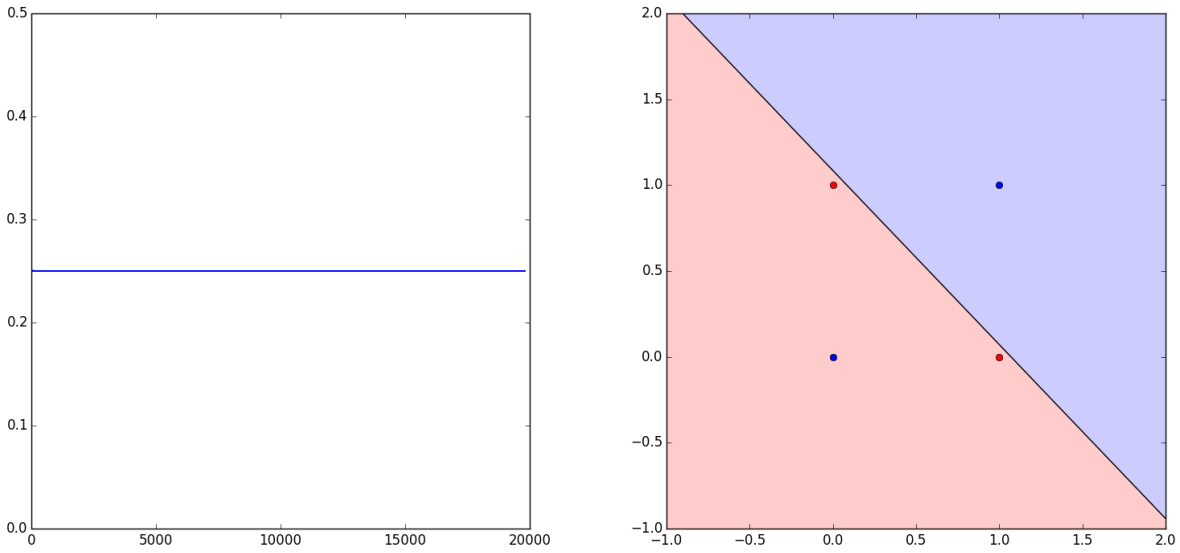

Figure 8.6: Training error and final classifier for one neuron trained to separate the classes in the OR function.

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

## 8.3 Multilayer Perceptron

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 8.7 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:

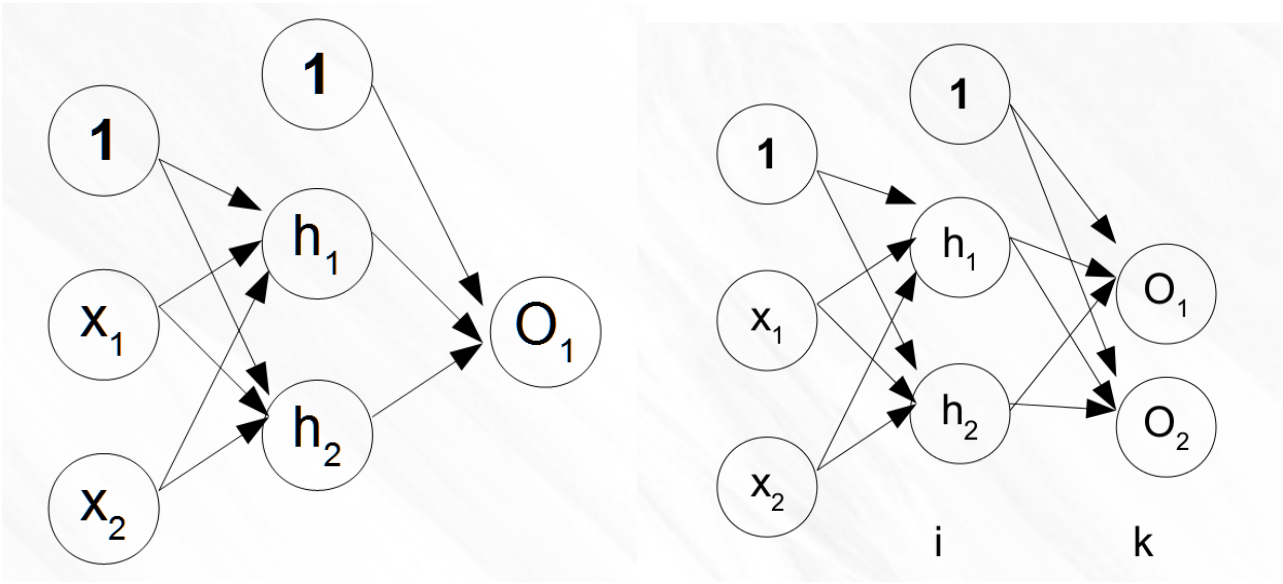


Figure 8.7: Two examples of multilayer perceptrons. Both have a hidden layer. The left panel shows a MLP with one output neuron, the right panel an MLP with two output neurons.

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

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 8.7, 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.

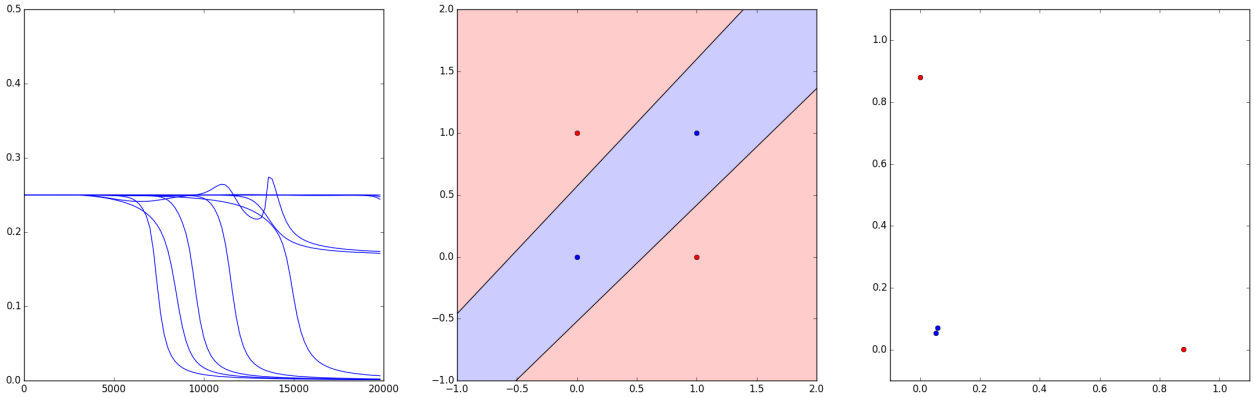

Figure 8.8: Training the MLP with one hidden layer for classifying the XOR function output. The first panel shows the training error over 10 training runs. Note that, due to the stochastic initialization and ordering of the examples presented, there are differences between different runs. The second 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.

This ability to recode the features can be used explicitly in autoassociator networks. These networks are trained so that the output equals the input, while a hidden layer with a smaller number of neurons re-encodes the data. Figure 8.9 shows an example, from Mitchell [18], showing a MLP with 8 inputs, 8 output neurons and 3 neurons on the hidden layer. By forcing the output neuron activated to correspond to the input neuron set to 1, the hidden layer learns to recode the 8 possible values in combinations of three 0,1 values.

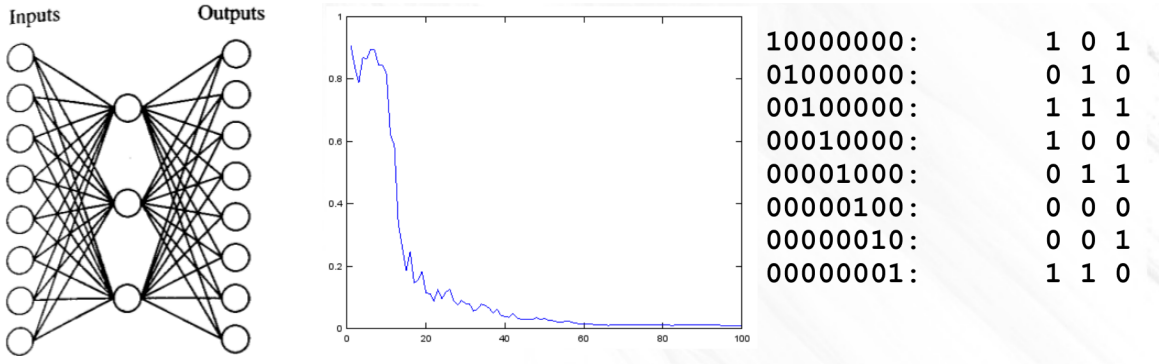


Figure 8.9: Autoassociator example. The network, shown on the left, was trained with the 8 different values consisting of one input set to 1 and the remainder set to 0, and forced to generate the same output. The hidden neurons recode the input into different combinations of neuron activations.

## 8.4 Training the Multilayer Perceptron

To train the MLP it is important to start with small, random weights, close to 0. 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.

To train the network, we present all training examples in a random order. One pass through all the training examples is one *epoch*. Then we repeat this process until the error converges or we detect overfitting. We can detect overfitting using cross-validation. This is also a form of regularization in training neural networks, as it allows us to stop training before the training error converges to a fixed value and thus avoid overfitting. Figure 8.10 illustrates this method.

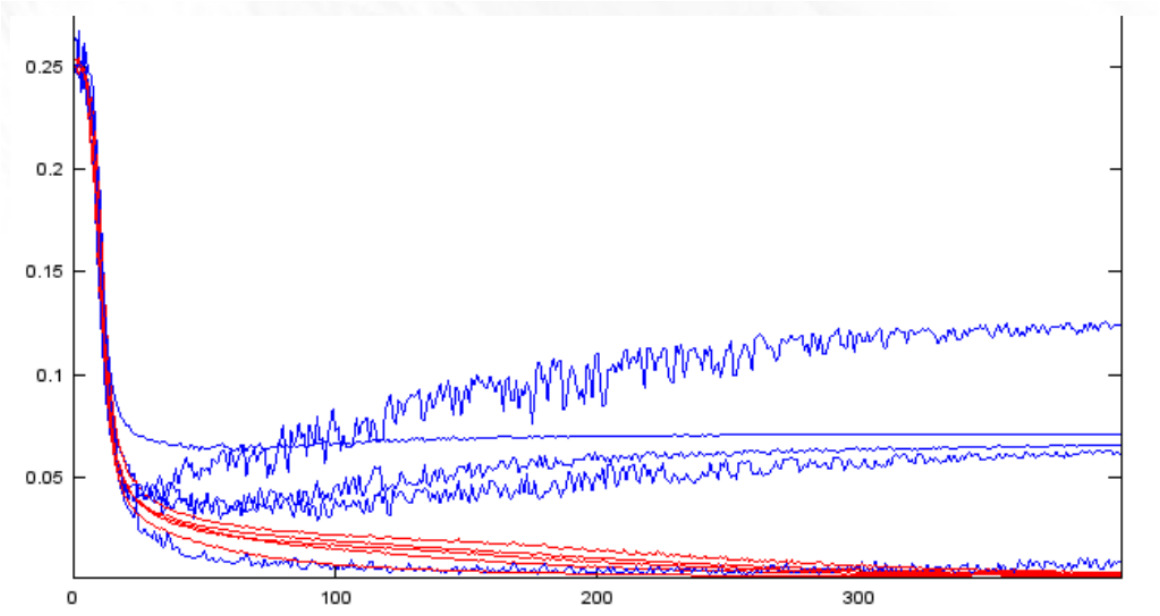


Figure 8.10: Validation (blue) and training error for five-fold cross-validation. Although training error keeps decreasing, it is best to stop training at epoch 40 to prevent overfitting.

Another form of regularization is to decay the coefficient weights by a small amount at each iteration, changing the update function to :

$$\Delta w_j = -\eta \frac{\delta E}{\delta w_j} - \lambda w_j$$

## 8.5 Further Reading

1. Alpaydin [2], Chapter 11
2. Mitchell [18], Chapter 4
3. Marsland [17], Chapter 3
4. (Bishop [4], Chapter 5)

# Bibliography


[1] Uri Alon, Naama Barkai, Daniel A Notterman, Kurt Gish, Suzanne Ybarra, Daniel Mack, and Arnold J Levine. Broad patterns of gene expression revealed by clustering analysis of tumor and normal colon tissues probed by oligonucleotide arrays. *Proceedings of the National Academy of Sciences*, 96(12):6745–6750, 1999.

[2] Ethem Alpaydin. *Introduction to Machine Learning*. The MIT Press, 2nd edition, 2010.

[3] David F Andrews. Plots of high-dimensional data. *Biometrics*, pages 125–136, 1972.

[4] Christopher M. Bishop. *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Springer, New York, 1st ed. edition, oct 2006.

[5] Deng Cai, Xiaofei He, Zhiwei Li, Wei-Ying Ma, and Ji-Rong Wen. Hierarchical clustering of www image search results using visual. Association for Computing Machinery, Inc., October 2004.

[6] Guanghua Chi, Yu Liu, and Haishandbscan Wu. Ghost cities analysis based on positioning data in china. *arXiv preprint arXiv:1510.08505*, 2015.

[7] Le Cun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Handwritten digit recognition with a back-propagation network. In *Advances in Neural Information Processing Systems*, pages 396–404. Morgan Kaufmann, 1990.

[8] Pedro Domingos. A unified bias-variance decomposition. In *Proceedings of 17th International Conference on Machine Learning. Stanford CA Morgan Kaufmann*, pages 231–238, 2000.

[9] Hakan Erdogan, Ruhi Sarikaya, Stanley F Chen, Yuqing Gao, and Michael Picheny. Using semantic analysis to improve speech recognition performance. *Computer Speech & Language*, 19(3):321–343, 2005.

[10] Martin Ester, Hans-Peter Kriegel, Jörg Sander, and Xiaowei Xu. A density-based algorithm for discovering clusters in large spatial databases with noise. In *Kdd*, volume 96, pages 226–231, 1996.

[11] Brendan J Frey and Delbert Dueck. Clustering by passing messages between data points. *science*, 315(5814):972–976, 2007.

[12] Arthur E Hoerl and Robert W Kennard. Ridge regression: Biased estimation for nonorthogonal problems. *Technometrics*, 12(1):55–67, 1970.

[13] Patrick Hoffman, Georges Grinstein, Kenneth Marx, Ivo Grosse, and Eugene Stanley. Dna visual and analytic data mining. In *Visualization'97., Proceedings*, pages 437–441. IEEE, 1997.

[14] Chang-Hwan Lee, Fernando Gutierrez, and Dejing Dou. Calculating feature weights in naive bayes with kullback-leibler measure. In *Data Mining (ICDM), 2011 IEEE 11th International Conference on*, pages 1146–1151. IEEE, 2011.

[15] Stuart Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[16] James MacQueen et al. Some methods for classification and analysis of multivariate observations. In *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, volume 1, pages 281–297. Oakland, CA, USA., 1967.

[17] Stephen Marsland. *Machine Learning: An Algorithmic Perspective*. Chapman & Hall/CRC, 1st edition, 2009.

[18] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

[19] Yvan Saeys, Iñaki Inza, and Pedro Larrañaga. A review of feature selection techniques in bioinformatics. *bioinformatics*, 23(19):2507–2517, 2007.

[20] Roberto Valenti, Nicu Sebe, Theo Gevers, and Ira Cohen. Machine learning techniques for face analysis. In Matthieu Cord and Pádraig Cunningham, editors, *Machine Learning Techniques for Multimedia*, Cognitive Technologies, pages 159–187. Springer Berlin Heidelberg, 2008.

[21] Giorgio Valentini and Thomas G Dietterich. Bias-variance analysis of support vector machines for the development of svm-based ensemble methods. *The Journal of Machine Learning Research*, 5:725–775, 2004.

[22] Jake VanderPlas. Frequentism and bayesianism: a python-driven primer. *arXiv preprint arXiv:1411.5018*, 2014.