5 - Activations and Loss Functions

Ludwig Krippahl

Introduction

Summary

- Why go deep?
- The vanishing gradients problem
- ReLU to the rescue
- Different activations: when and how
- Loss functions

Why go deep?

Wide vs Deep

Universal approximation theorem

Wikipedia, https://en.wikipedia.org/wiki/Universal_approximation_theorem

- Given ϕ nonconstant, bounded and continuous
- Given $I_m = [0,1]^m$, $\epsilon > 0$ and any function f continuous in I_m
- There are N constants v_i, b_i and $\overrightarrow{w_i}$ such that:

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

- for all $\vec{x} \in I_m$
- Proven in 1989 for sigmoid activation by George Cybenko,
- In other words, all we need is one hidden layer

Wide vs Deep

Universal approximation theorem

- One layer can approximate any function within a bounded region
- However, more oscilations by stacking layers

- Activation (and loss) can oscilate more with fewer neurons in a deep network

- Oscillations are related to Vapnik-Chervonenkis dimension

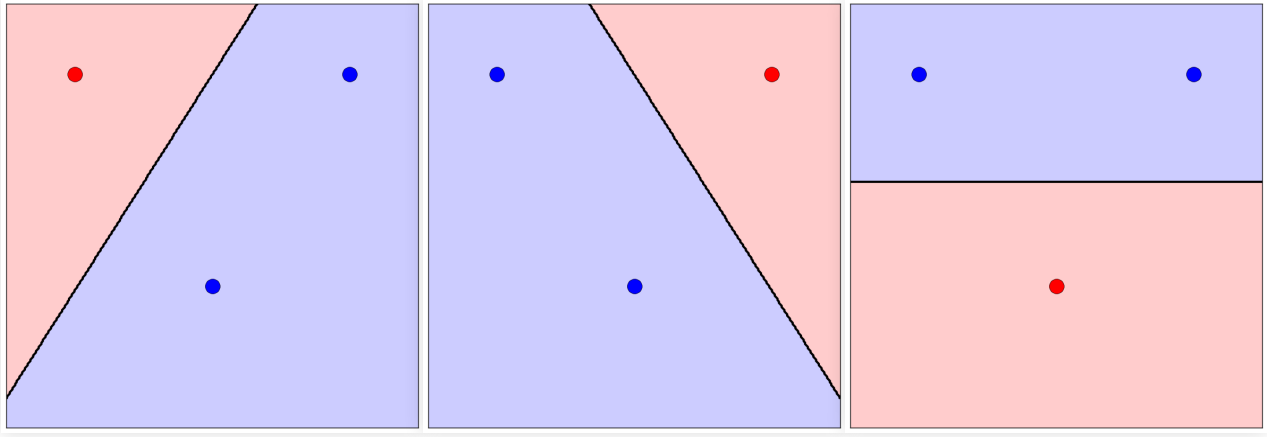
- (largest set that can be shattered)

Wide vs Deep

Vapnik-Chervonenkis dimension

- Hypothesis class $\mathcal{H}$ shatters set $\mathcal{S}$ if, for any labelling S, there is a $h \in \mathcal{H}$ consistent with S (classifies without errors)

- Example: linear classifier in 2D shatters 3 points



- VC dimension measures classification "power"
- Deep networks are more powerful for the same number of neurons

Wide vs Deep

"No free lunch" theorem:
«... 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, Neural Computation 8, 1341-1390 (1996, MIT)

- Demonstrated for test error (generalization), assuming any possible distribution of data
- In real life, data is not distributed in any possible way
- Best algorithm depends on the problem

Wide vs Deep

How to choose: some pros and cons of deep learning

- Linear models with nonlinear feature transformations (wide):

- Better at memorization of feature interactions and more interpretable
- Generalization requires more feature engineering effort.

- Example:

- "Customers who purchased that also purchased ..."
- Works if we have data on exactly the same purchases
- Hard to generalize for "similar" purchases without engineering features (e.g. type of movie, ...)

Wide vs Deep

How to choose: some pros and cons of deep learning

- Deep learning models:

- Better at generalizing by learning relevant features, even with little engineering
- But "black box", difficult to understand which features they use

Image credits: teenybiscuit, Twitter.

Wide vs Deep

Deep models

- Pros:

- More shattering power with fewer parameters
- Learn feature extraction
- Good for complex problems and for generalizing

- Cons:

- More powerful models require more data to avoid overfitting
- Learned features may be harder to interpret

Vanishing gradients

Vanishing gradients

Backpropagation in Activation and Loss

- Output neuron n of layer k receives input from m from layer i through weight j

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

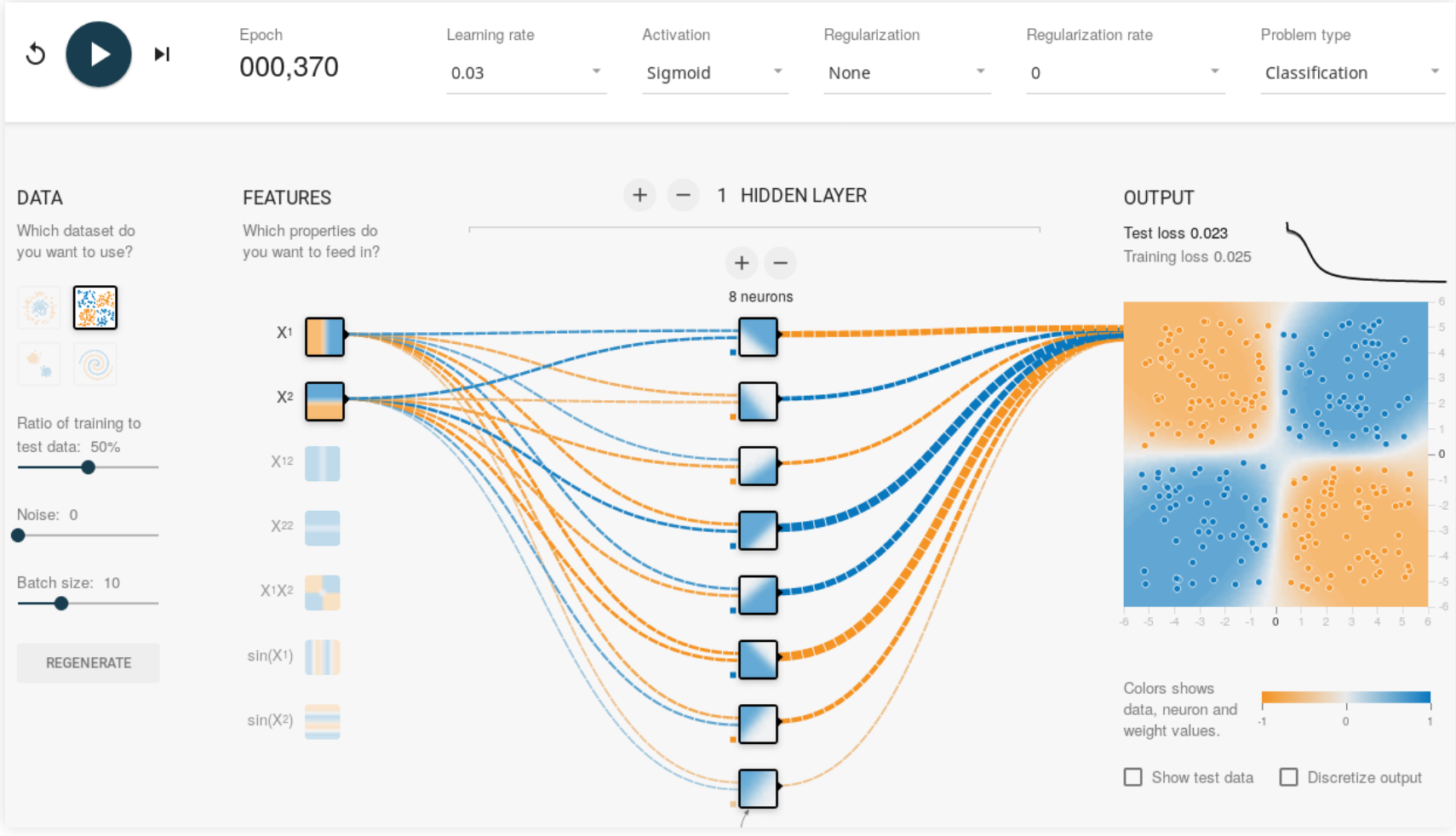
- For a weight m on hidden layer i, we must propagate the output error backwards from all neurons ahead

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

- If δs is small (vanishing gradient) backpropagation becomes ineffective as we increase depth
- This happens with logistic activation (or similar, such as TanH)

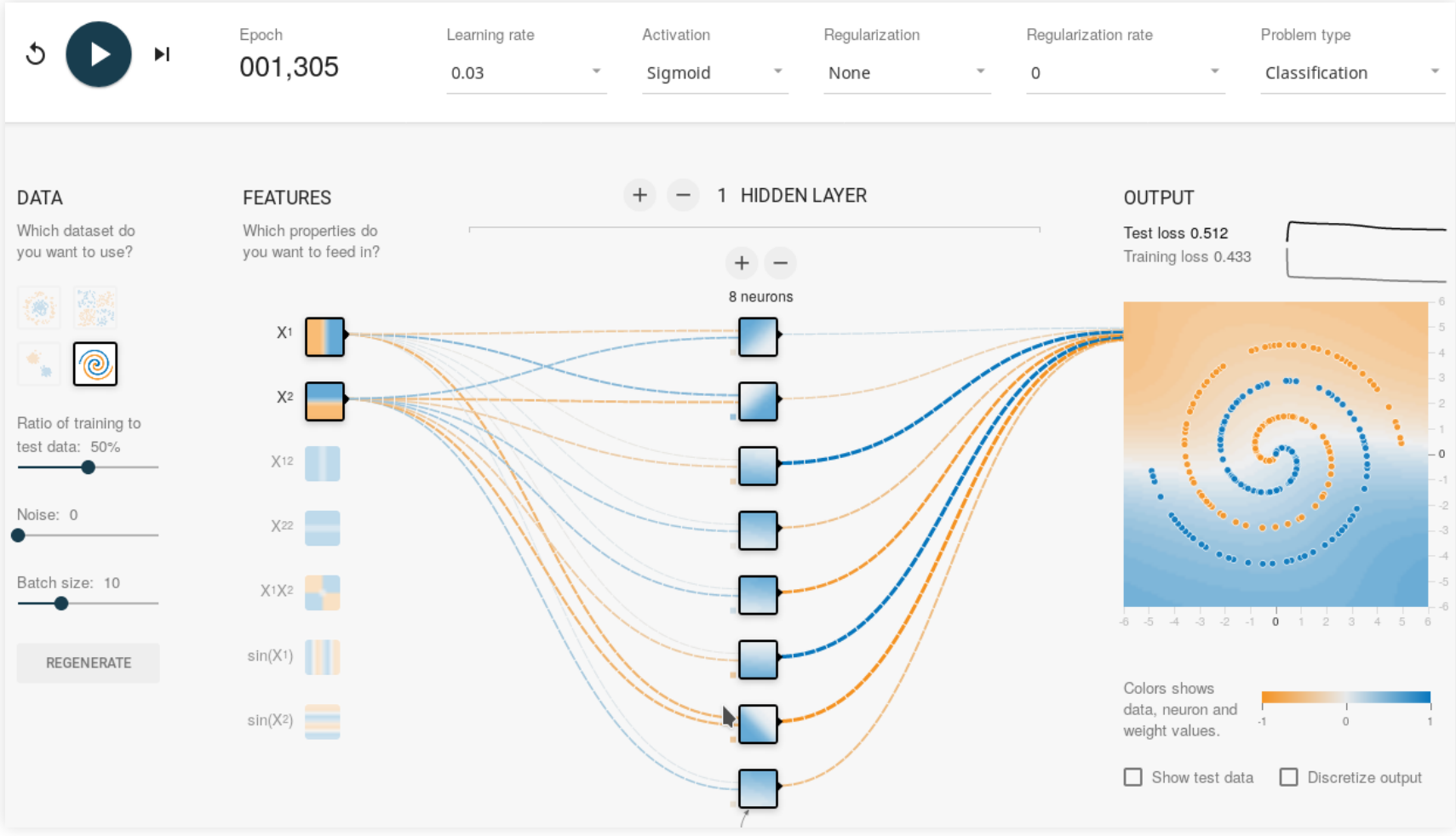
Vanishing gradients

- Single hidden layer, sigmoid, works fine here



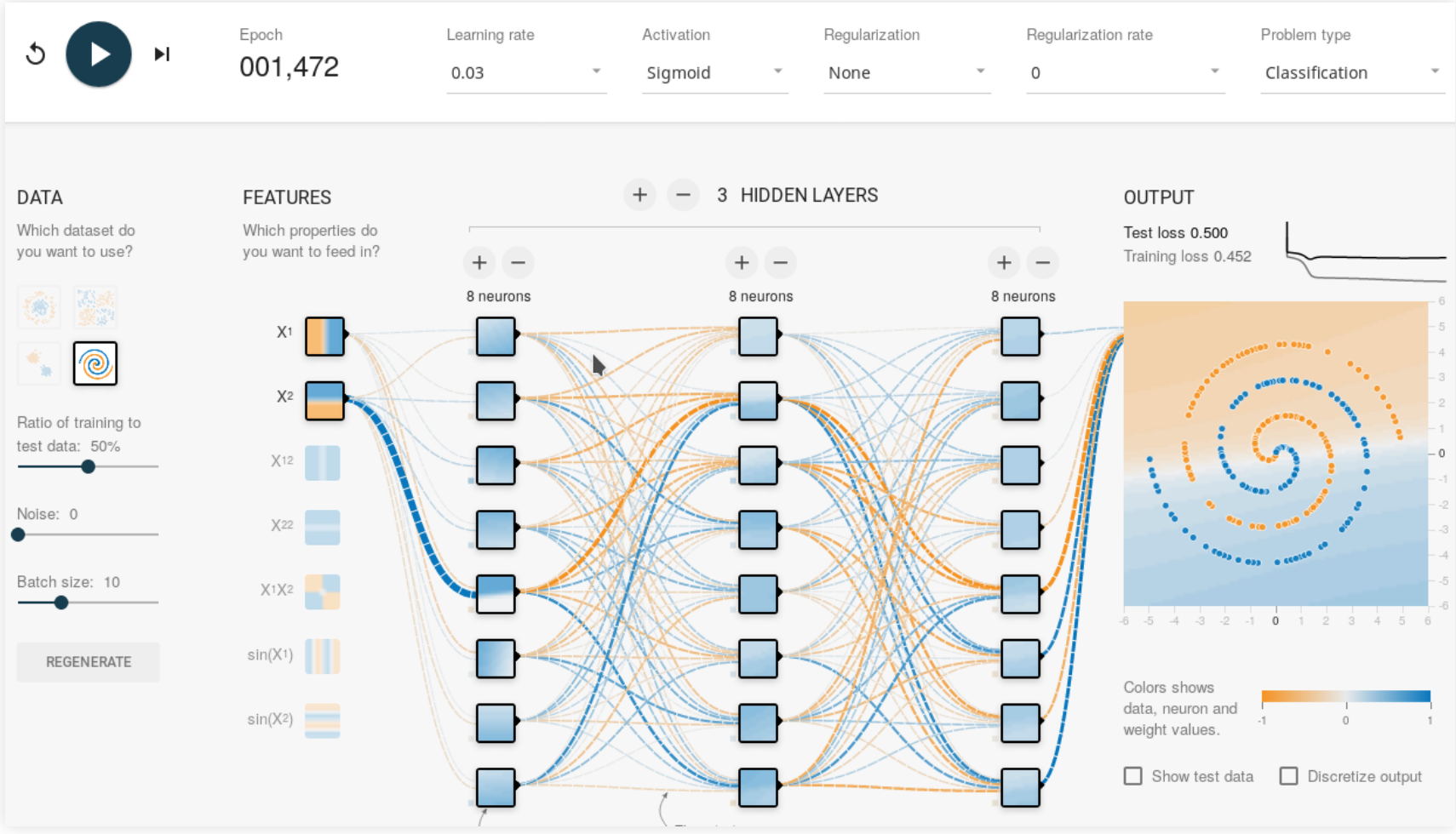
Vanishing gradients

- Single hidden layer, sigmoid, doesn't work here with 8 neurons



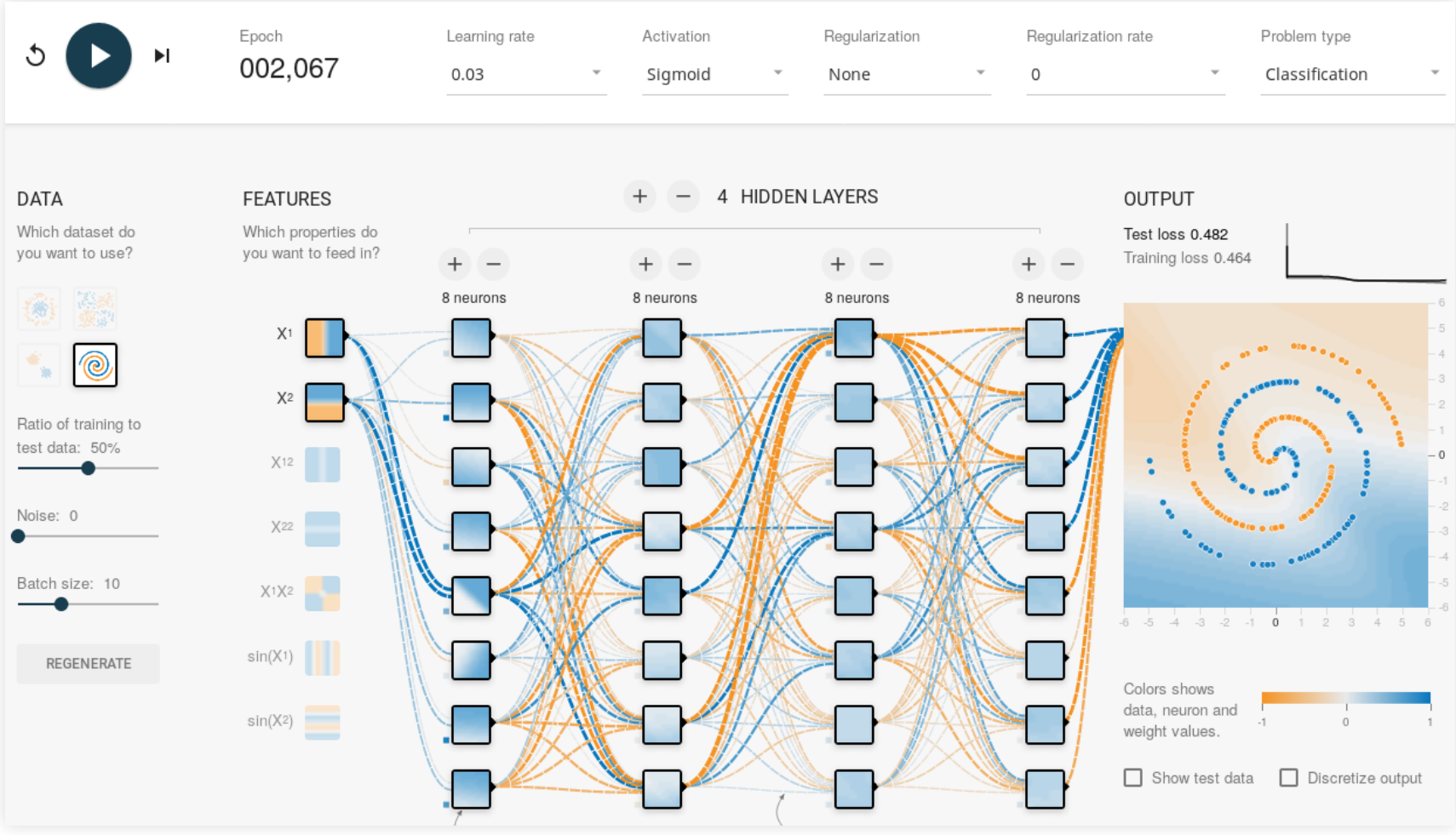
Vanishing gradients

- Increasing depth does not seem to help



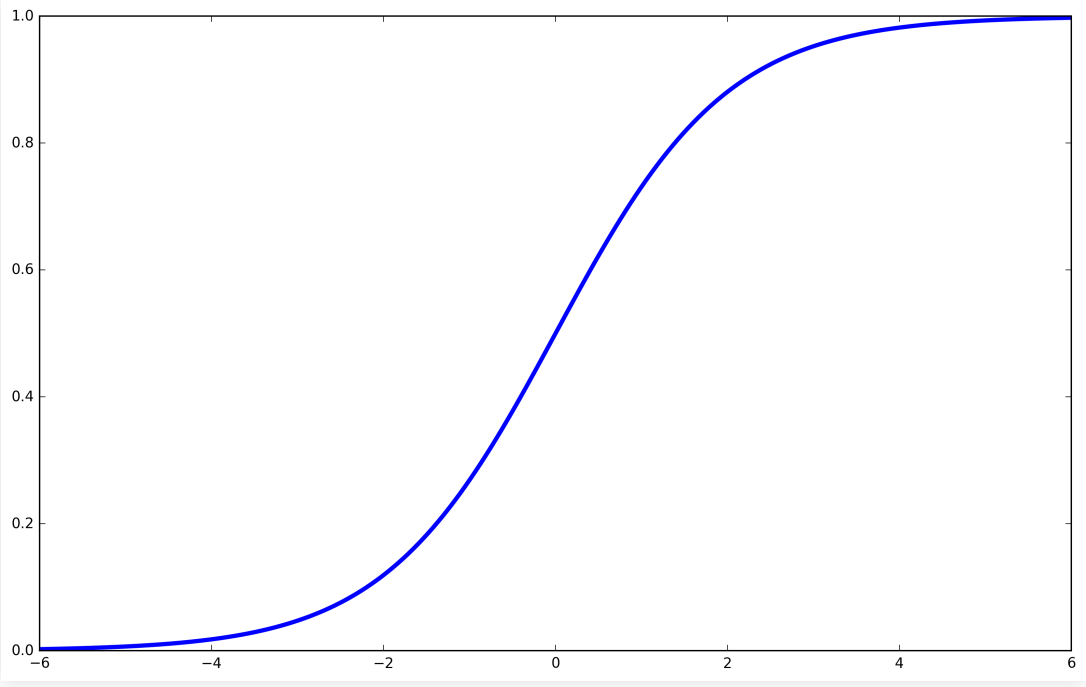
Vanishing gradients

- Increasing depth does not seem to help



Vanishing gradients

- Increasing depth does not seem to help
- Sigmoid activation saturates and gradients vanish with large coefs.

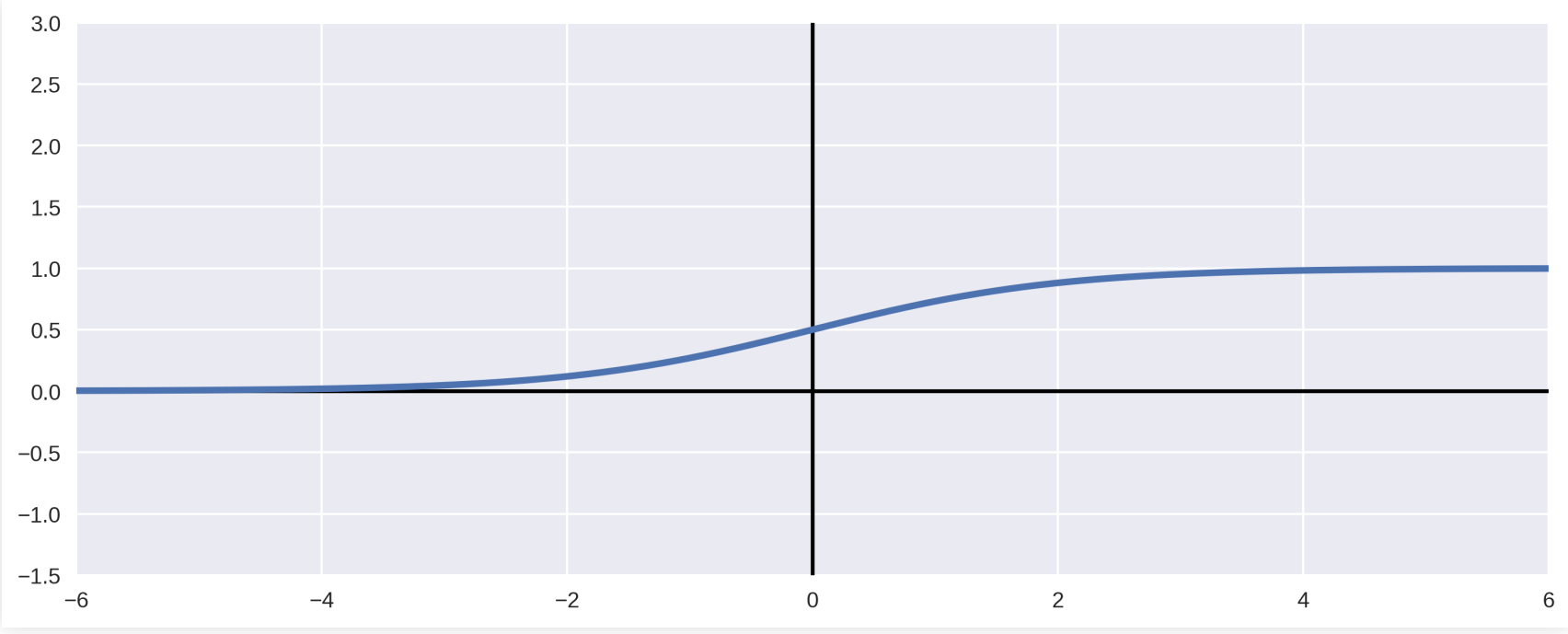


Rectified Linear Unit

Rectified Linear Unit (ReLU)

- Sigmoid activation units saturate

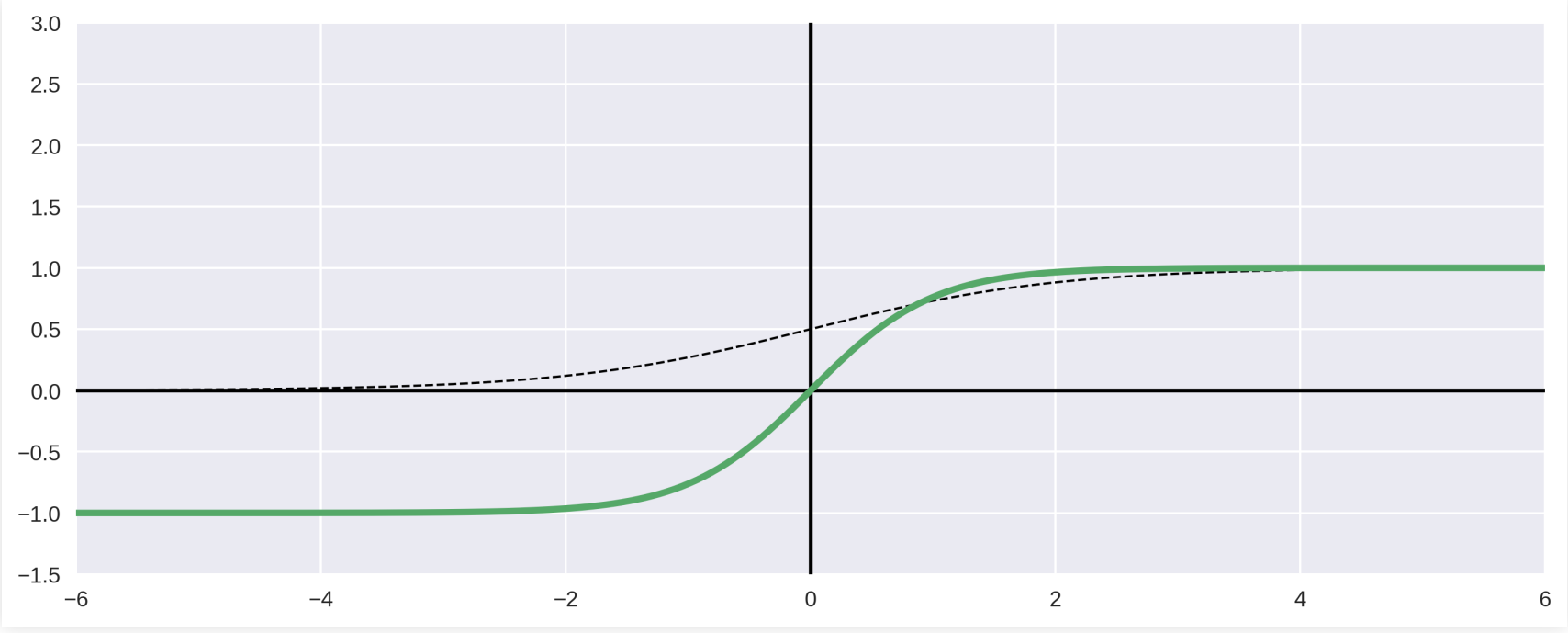
$$y_i = \frac{1}{1 + e^{-x_i}}$$



Rectified Linear Unit (ReLU)

- The same happens with hyperbolic tangent

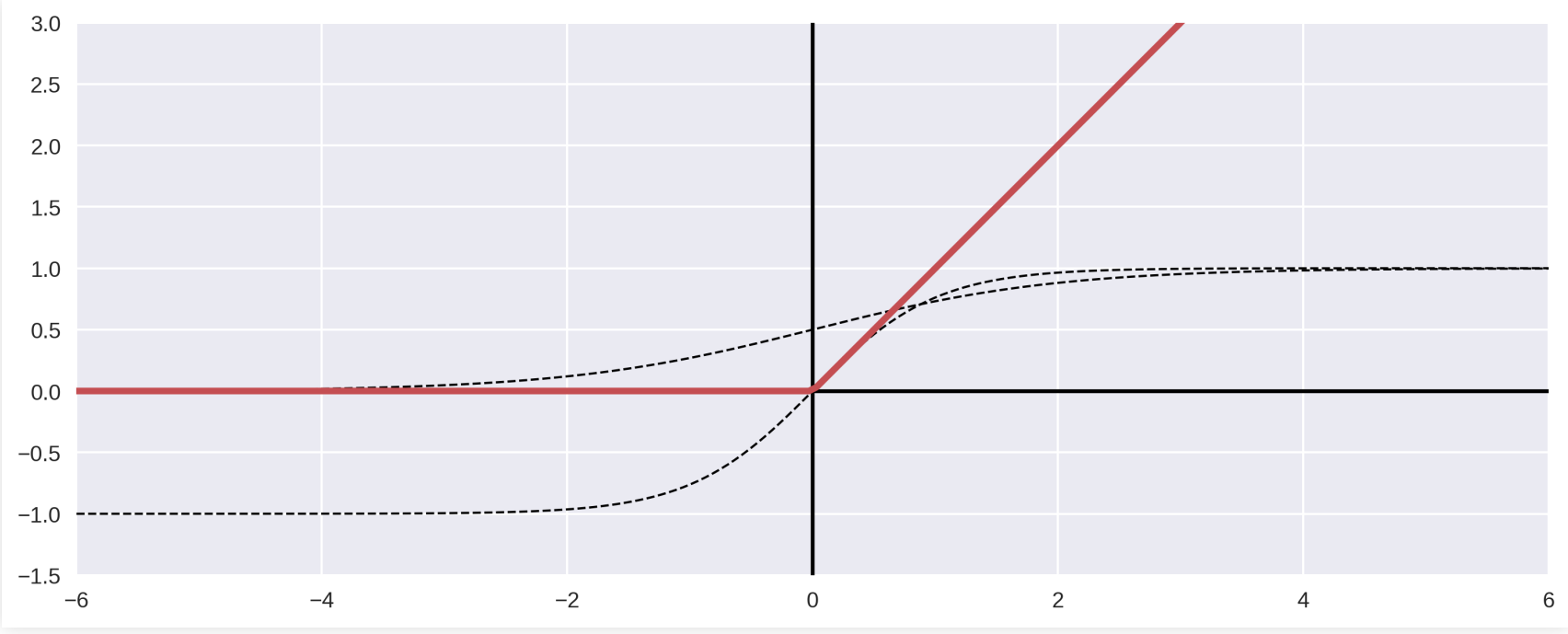
$$y_i = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$



Rectified Linear Unit (ReLU)

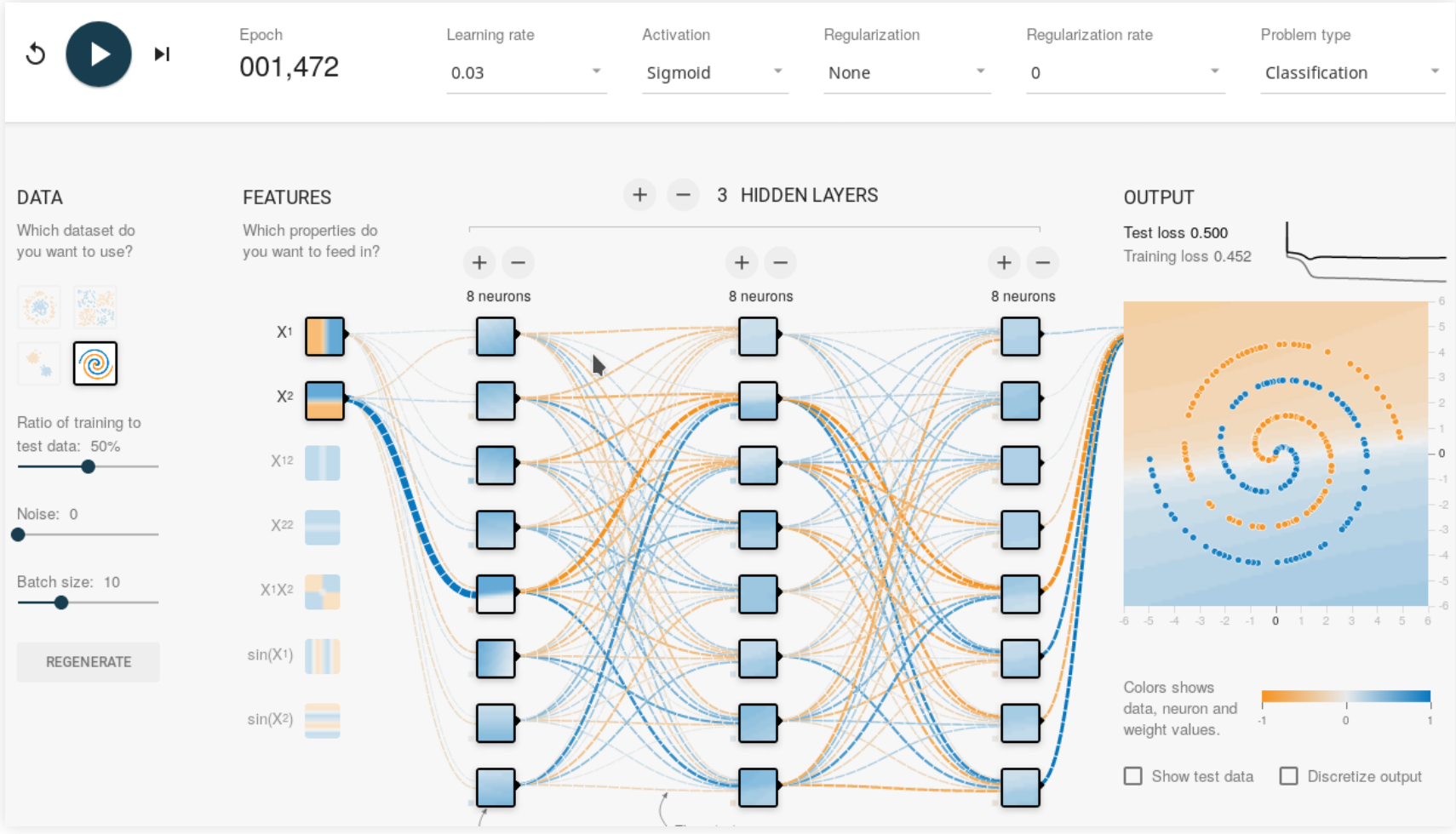
- Rectified linear units do not have this problem

$$y_i = \begin{cases} x_i & x_i > 0 \\ 0 & x_i \leq 0 \end{cases}$$



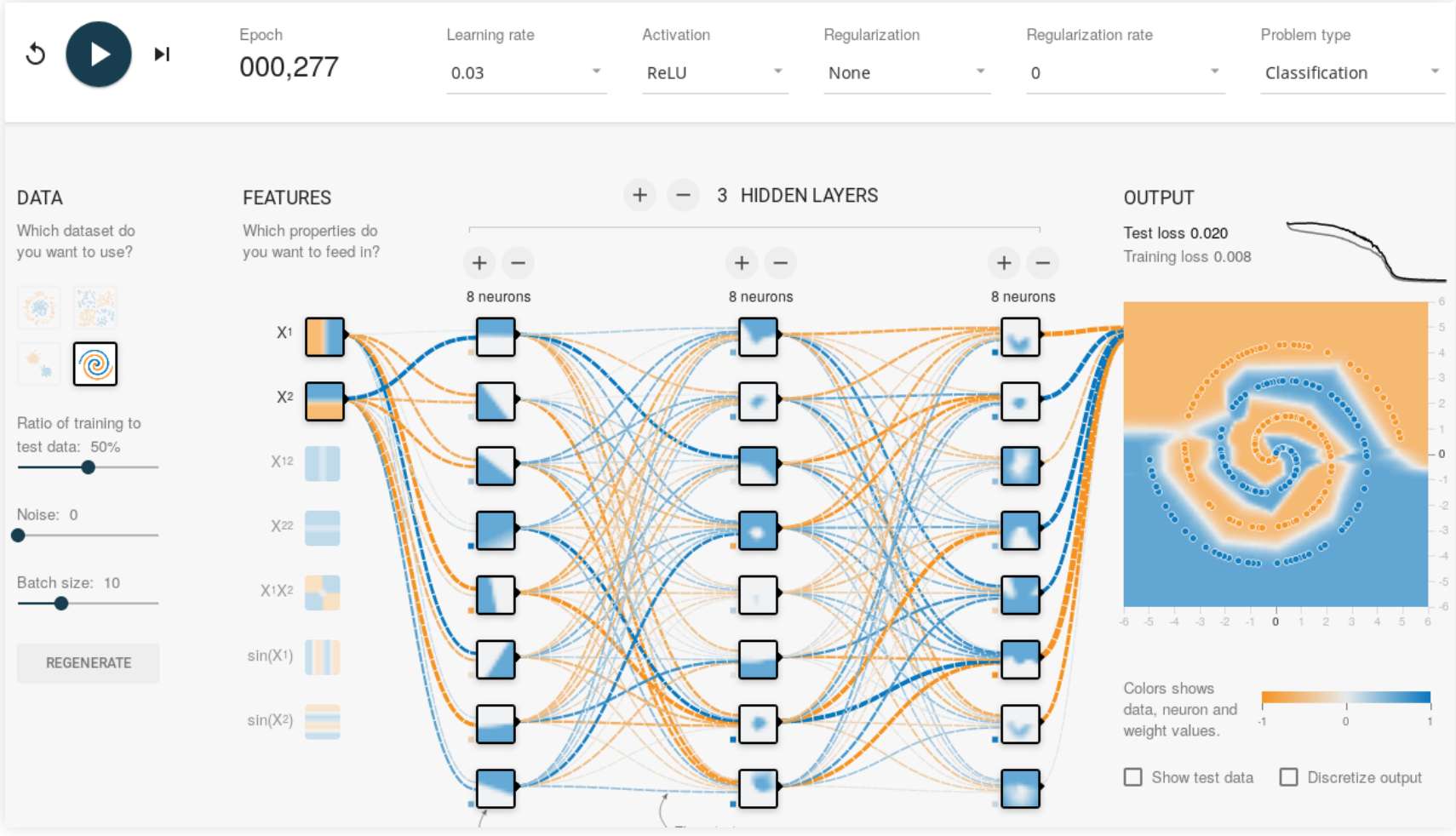
ReLU

- Sigmoid activation, 3 layers



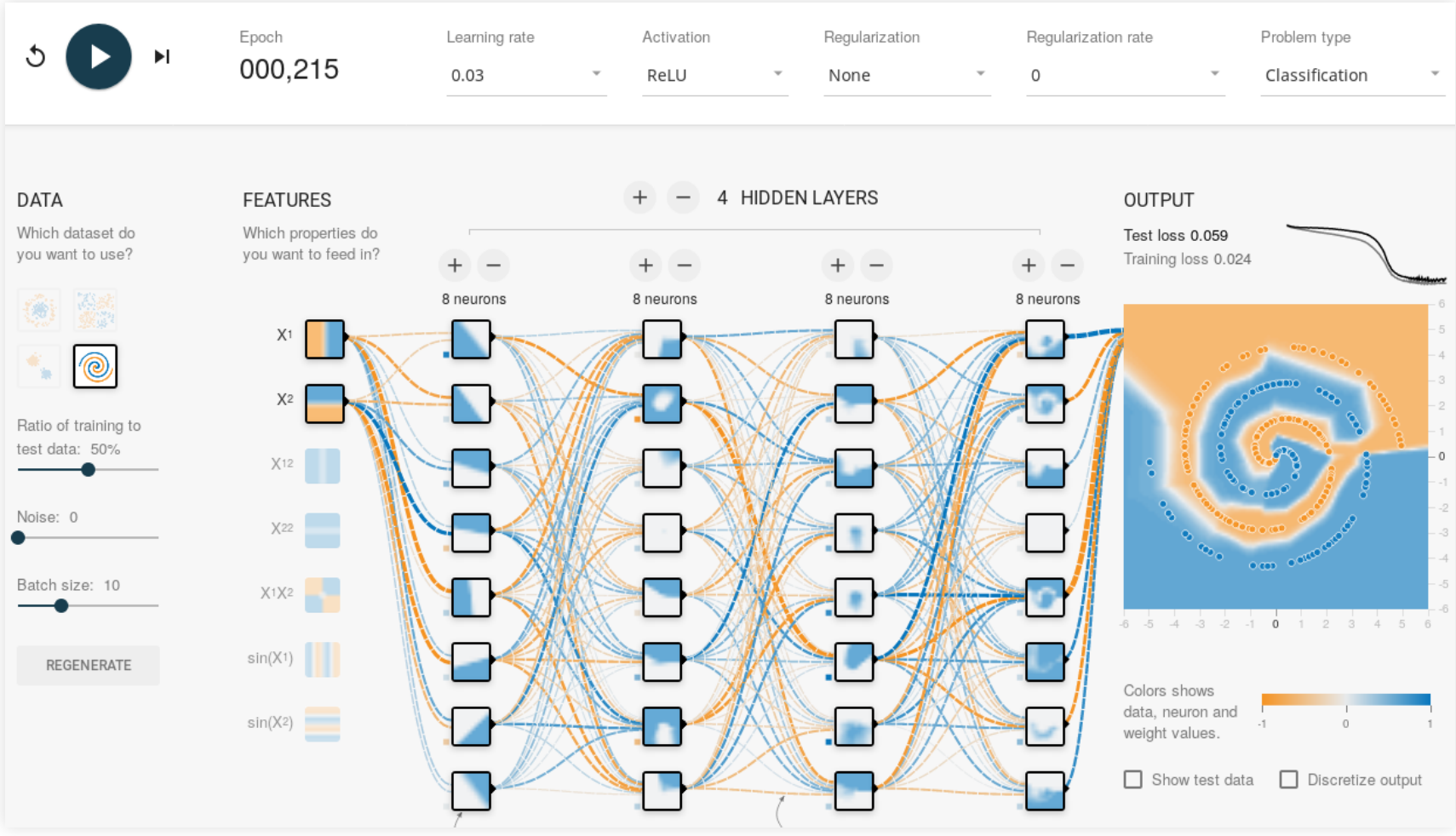
ReLU

- ReLU activation, 3 layers



ReLU

- ReLU activation, 4 layers



Rectified Linear Unit (ReLU)

- Advantages of ReLU activation:

- Fast to compute
- Does not saturate for positive values, and gradient is always 1

- Disadvantage:

- ReLU units can "die" if training makes their weights very negative
- The unit will output 0 and the gradient will become 0, so it will not "revive"

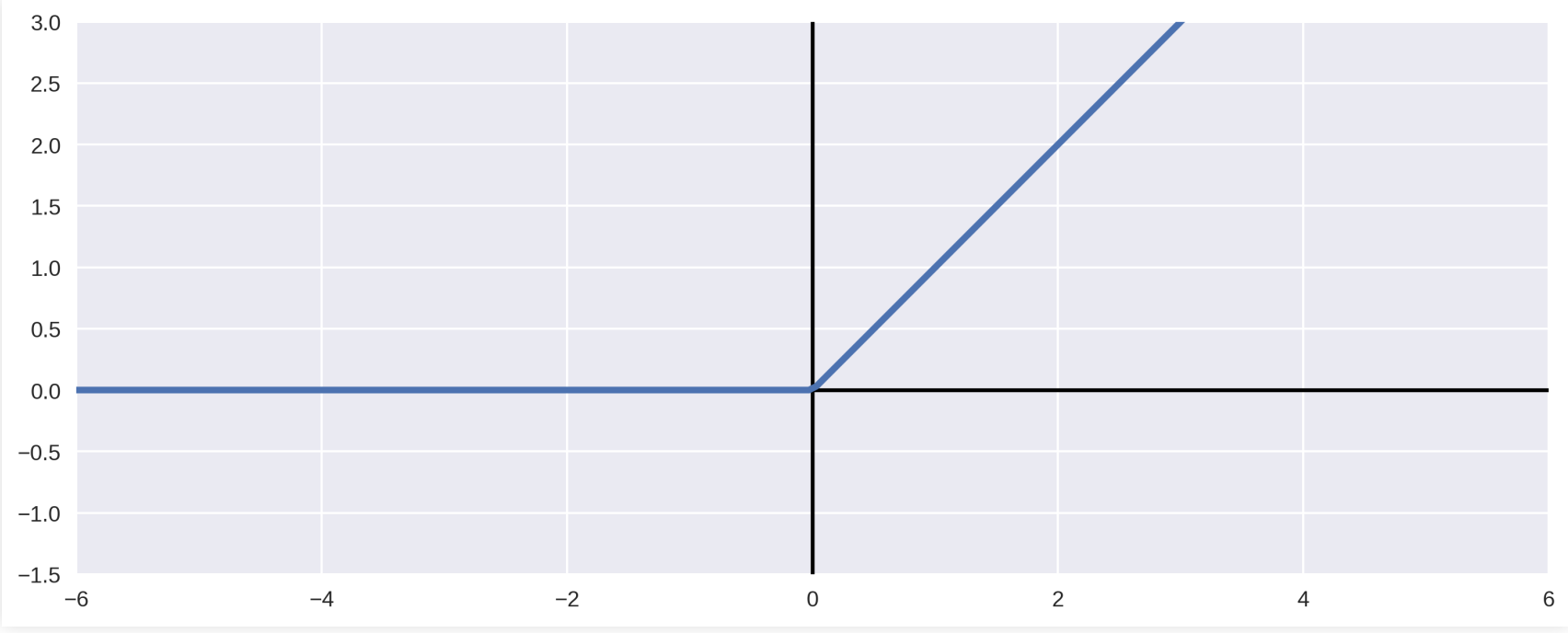
- There are variants that try to fix this problem

ReLU

(Some) ReLU variants

- Simple ReLU can die if coefficients are negative

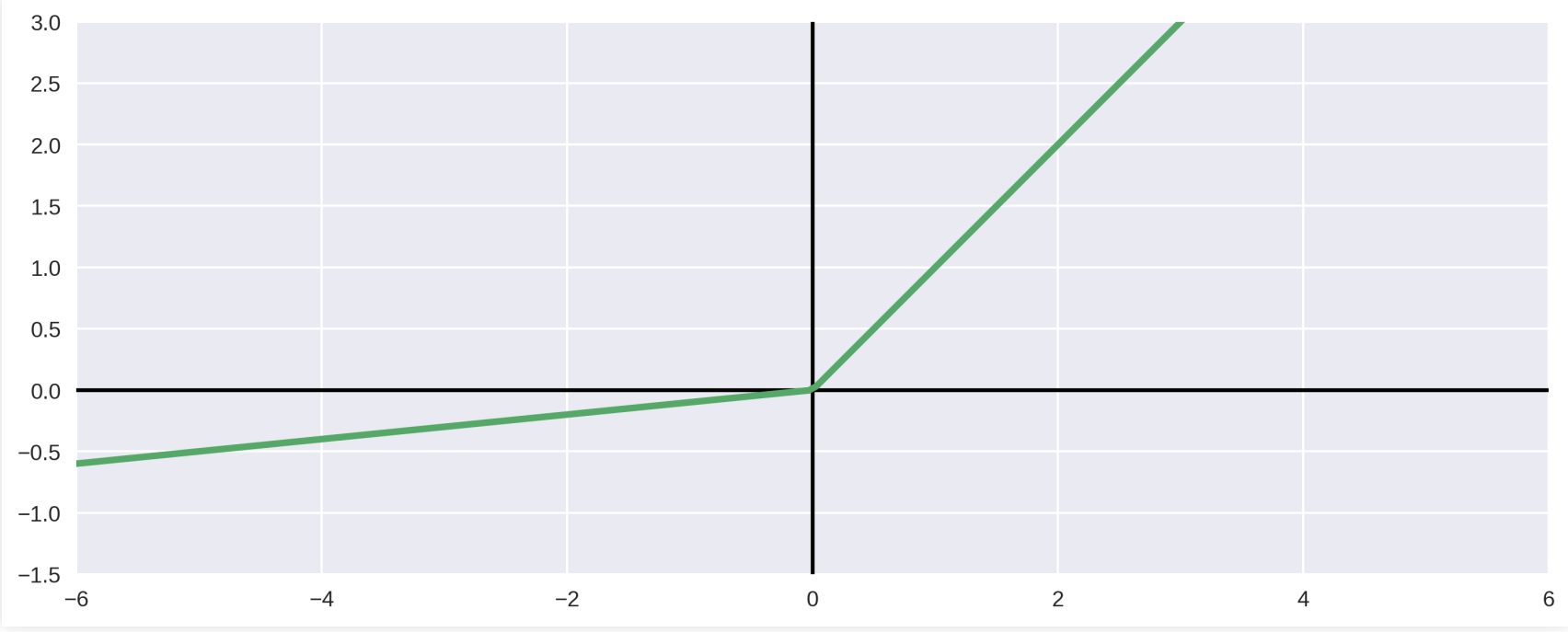
$$y_i = \begin{cases} x_i & x_i > 0 \\ 0 & x_i \leq 0 \end{cases}$$



ReLU variant: Leaky ReLU

- Leaky ReLU gradient is never 0

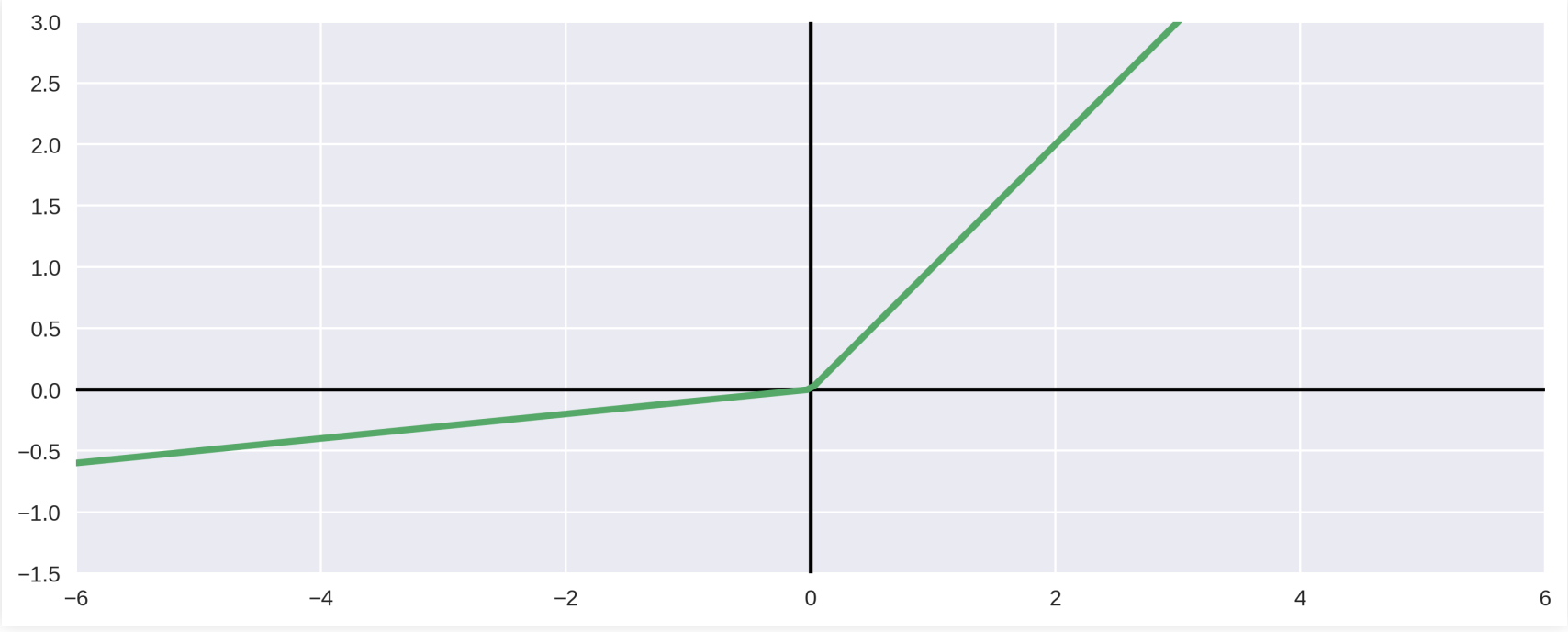
$$y_i = \begin{cases} x_i & x > 0 \\ \frac{x_i}{a_i} & x_i \leq 0 \end{cases}$$



ReLU variant: Leaky ReLU

- Note: in Tensorflow

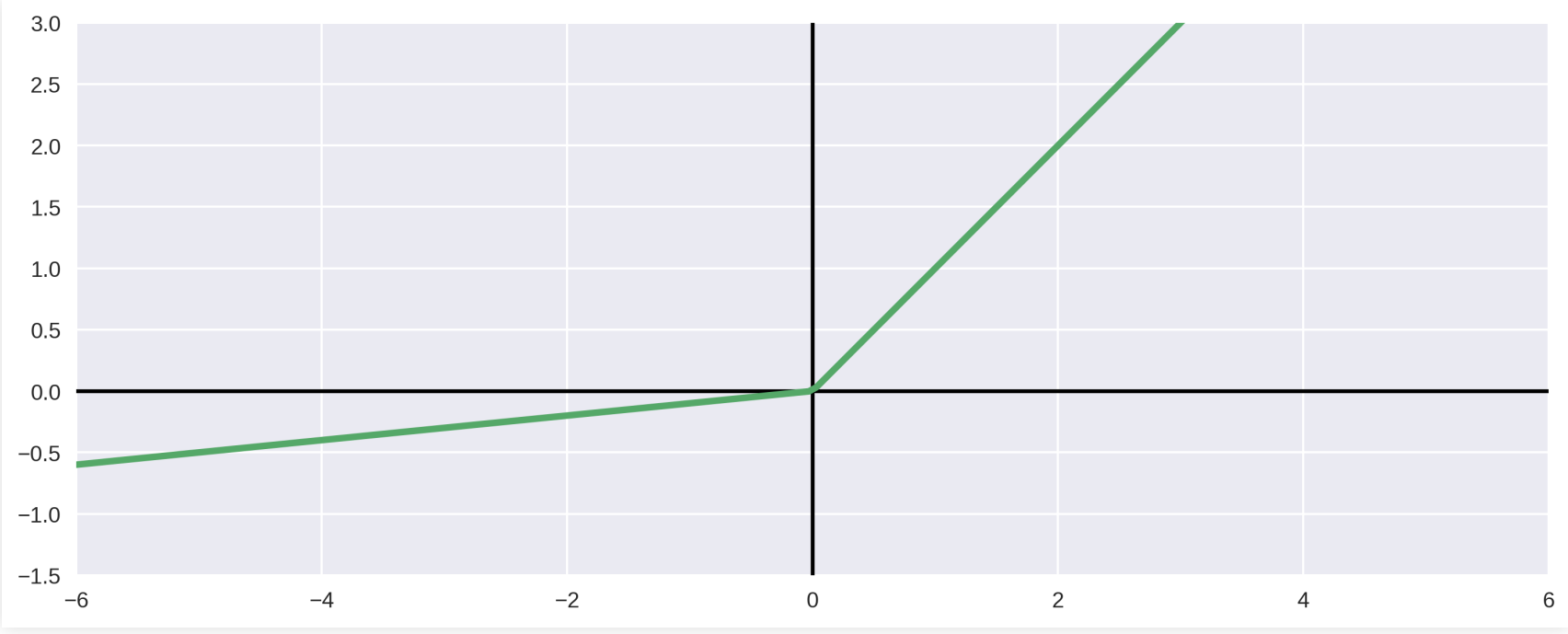
$$y_i = \begin{cases} x_i & x > 0 \\ a_i x_i & x_i \leq 0 \end{cases}$$



ReLU variant: Parametric ReLU

- Same as leaky, but a_i is also learned

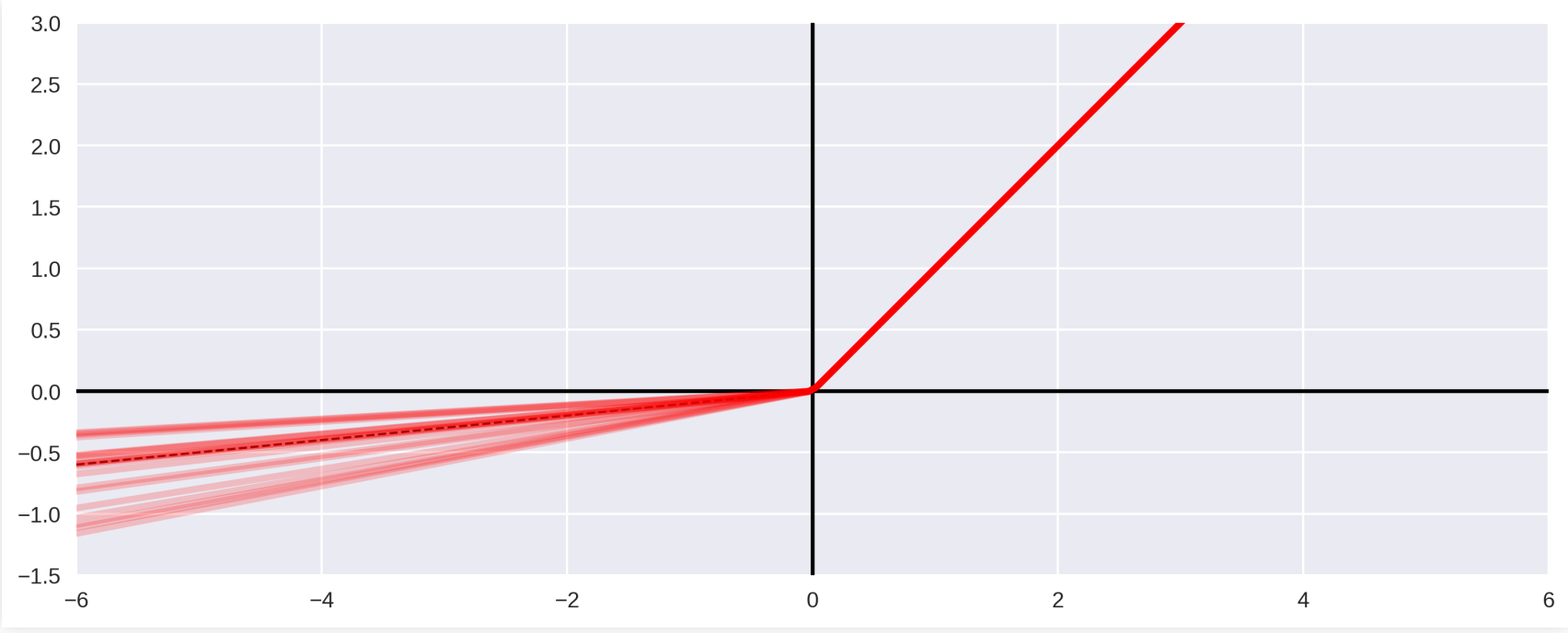
$$y_i = \begin{cases} x_i & x > 0 \\ \frac{x_i}{a_i} & x_i \leq 0 \end{cases}$$



ReLU variant: Randomized Leaky ReLU

- Similar, but $a_i \sim U(l, u)$
  (average of l, u in test)

$$y_i = \begin{cases} x_i & x > 0 \\ a_i x_i & x_i \leq 0 \end{cases}$$



Comparing ReLU variants

Empirical Evaluation of Rectified Activations in Convolution Network (Xu et. al. 2015)

- Compared on 2 data sets

- CIFAR-10: 60000 32x32 color images in 10 classes of 6000 each
- CIFAR-100: 60000 32x32 color images in 100 classes of 600 each

Activation	Training Error	Test Error
ReLU	0.00318	0.1245
Leaky ReLU, $a = 100$	0.0031	0.1266
Leaky ReLU, $a = 5.5$	0.00362	**0.1120**
PReLU	0.00178	0.1179
RReLU ($y_{ji} = x_{ji}/\frac{l+u}{2}$)	0.00550	**0.1119**

Table 3. Error rate of CIFAR-10 Network in Network with different activation function

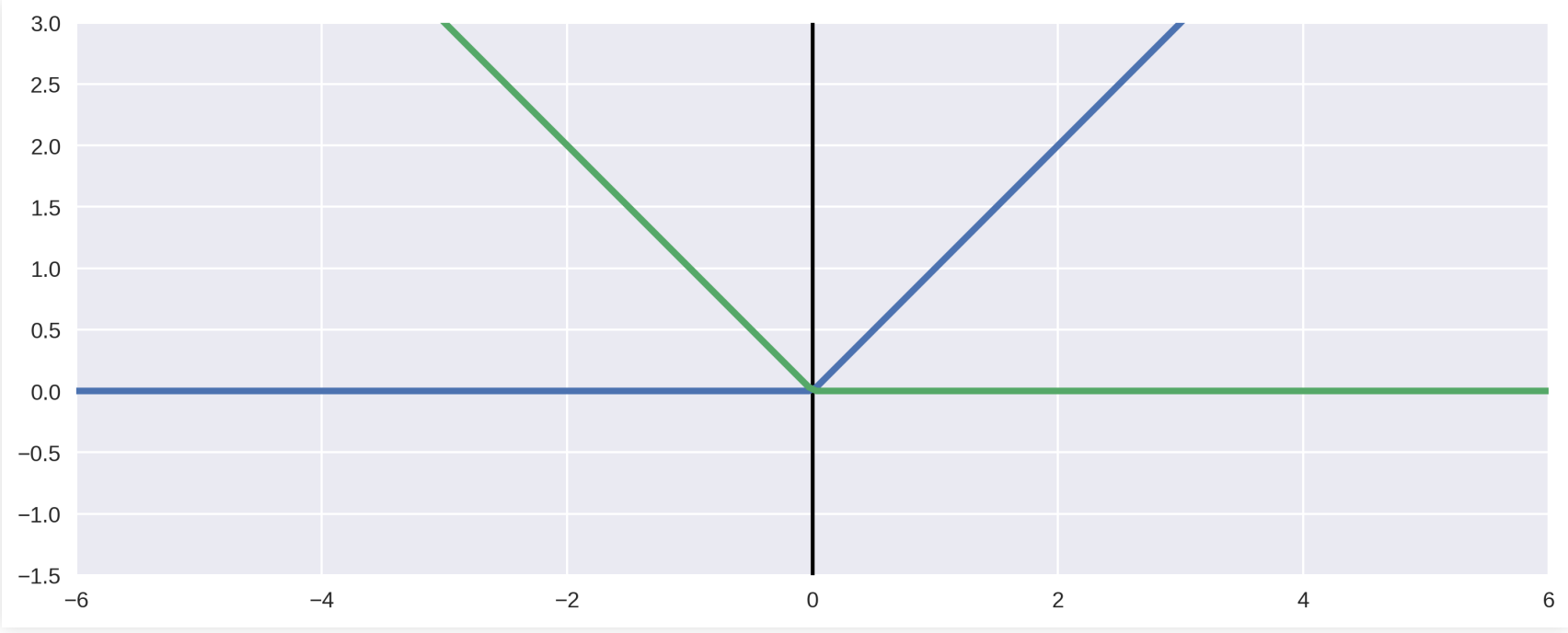
Activation	Training Error	Test Error
ReLU	0.1356	0.429
Leaky ReLU, $a = 100$	0.11552	0.4205
Leaky ReLU, $a = 5.5$	0.08536	**0.4042**
PReLU	0.0633	0.4163
RReLU ($y_{ji} = x_{ji}/\frac{l+u}{2}$)	0.1141	**0.4025**

Table 4. Error rate of CIFAR-100 Network in Network with different activation function

CReLU

- Concatenated ReLU combine two ReLU for x and $-x$

$$y_i = \begin{cases} x_i & x_i > 0 \\ 0 & x_i \leq 0 \end{cases} \qquad z_i = \begin{cases} 0 & x_i > 0 \\ -x_i & x_i \leq 0 \end{cases}$$

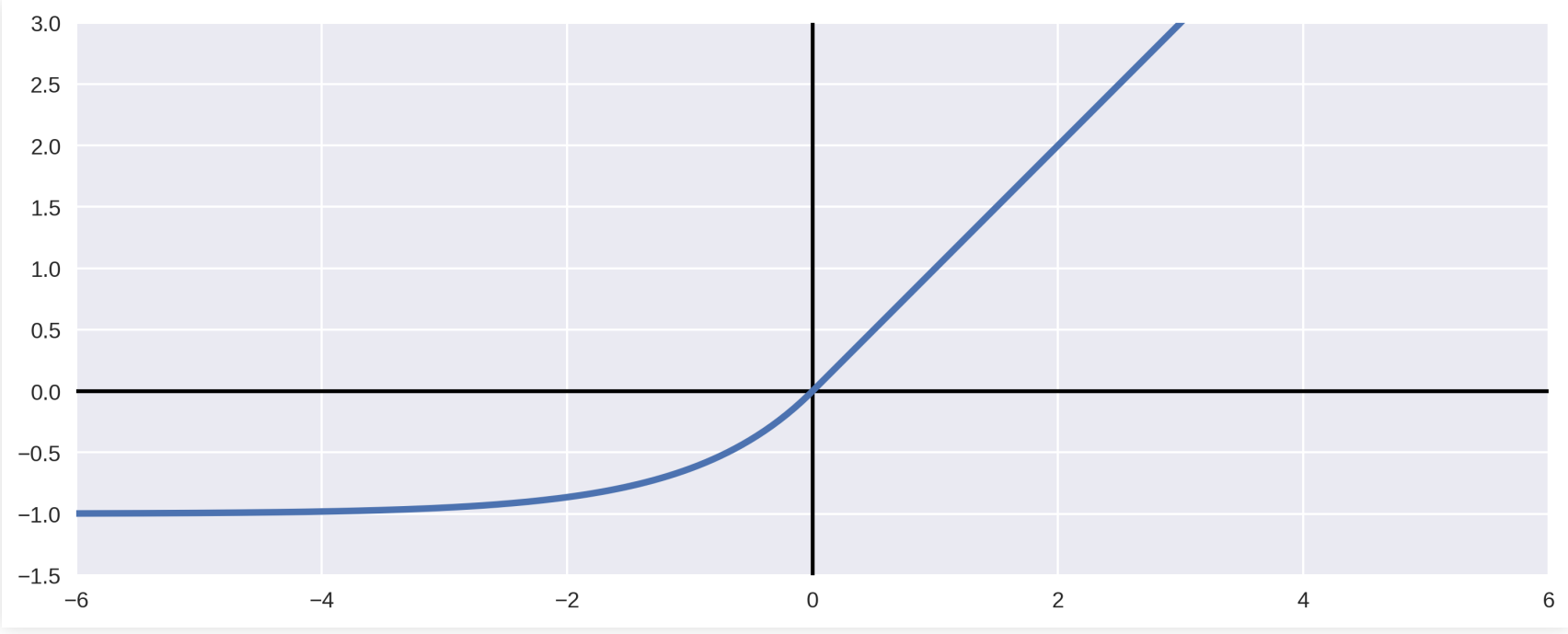


Shang et. al., Understanding and Improving CNN via CReLUs, 2016

Exponential Linear Unit

- Exponential in negative part

$$y_i = \begin{cases} x_i & x_i > 0 \\ a(e^{x_i} - 1) & x_i \leq 0 \end{cases}$$



Clevert et. al. Fast and Accurate Deep Network Learning by ELUs, 2015

Activations: which, when, why?

Choosing activations

Hidden layer activations

- Hidden layers perform nonlinear transformations

- Without nonlinear activation functions, all layers would just amount to a single linear transformation

- Activation functions should be fast to compute
- Activation functions should avoid vanishing gradients
- This is why ReLU (esp. leaky variants) are the recommended choice for hidden layers

- Except for specific applications.
- E.g. LSTM, Long short-term memory recurrent networks

Choosing activations

Output layer activations

- Output layers are a different case.

- Choice depends on what we want the model to do

- For regression, output should generally be linear

- We do not want bounded values and there is little need for nonlinearity in the last layer

- For binary classification, sigmoid is a good choice

- The output value $[0, 1]$ is useful as a representation of the probability of C_1, like in logistic regression

- Sigmoid is also good for multilabel classification

- One example may fit with several labels at the same time

- Use one sigmoid output per label

Choosing activations

Output layer activations

- For multiclass classification, use softmax:

- Note: multiclass means each example fits only one of several classes

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

- Softmax returns a vector where $\sigma_j \in [0,1]$ and $\sum_{k=1}^{K} \sigma_k = 1$
- This can fit a probability of example belonging to each class C_j
- Softmax is a generalization of the logistic function

- It combines the activations of several neurons

Loss and likelihood

Likelihood

Basic concepts

- We have a set of labelled data

$$\{(\vec{x}^1, y^1), \ldots, (\vec{x}^n, y^n)\}$$

- We want to approximate some function $F(X) : X \rightarrow Y$ by fitting our parameters
- Given some training set, what are the best parameter values?

Simple example, linear regression

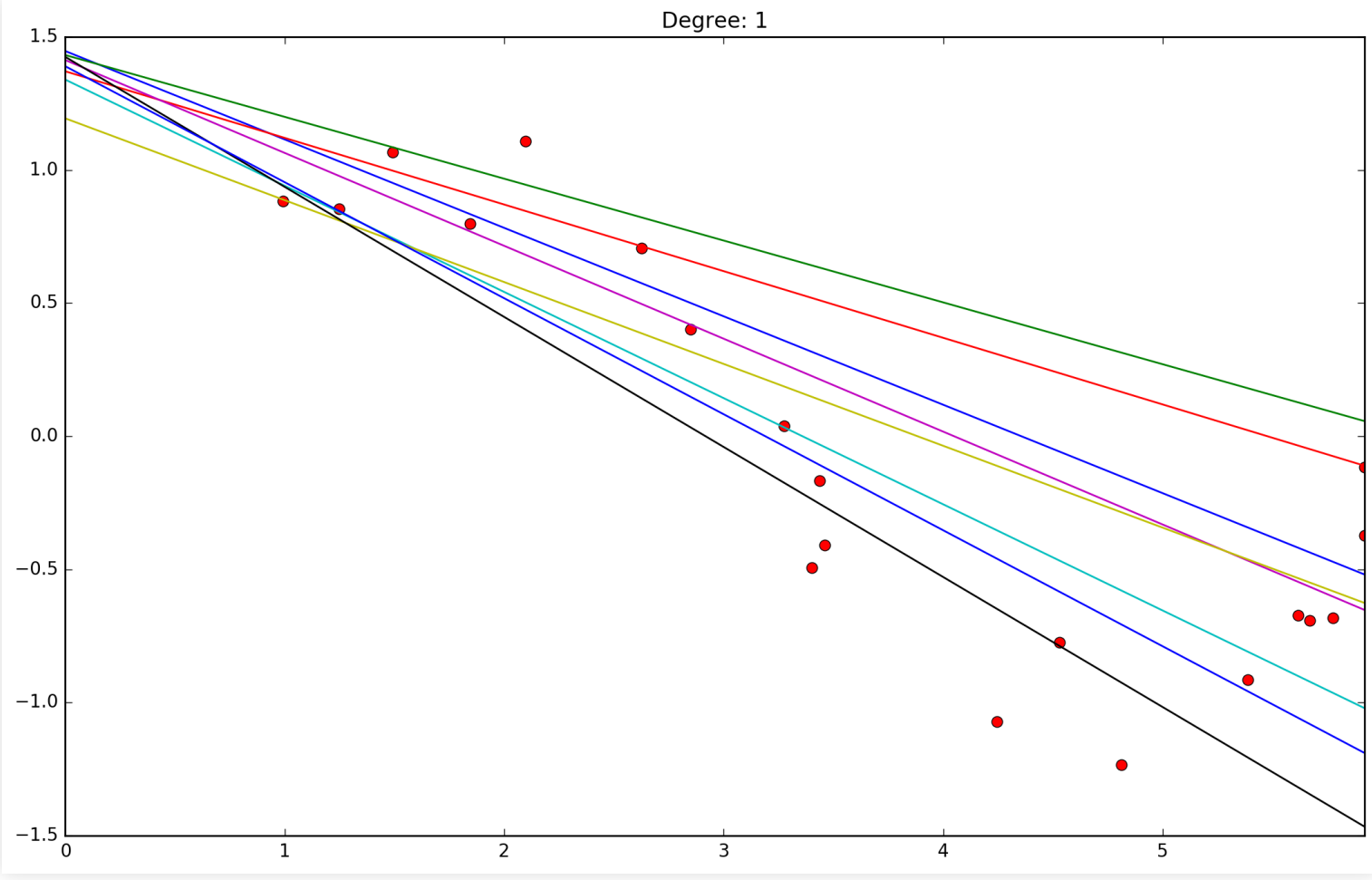
$$y = \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_{n+1}$$

- We have a set of (x, y) examples and want to fit the best line:

$$y = \theta_1 x + \theta_2$$

Likelihood

What to optimize?



Likelihood

What to optimize?

- Assume y is a function of x plus some error:

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

- We want to approximate $F(x)$ with some $g(x, \theta)$
- Assuming $\epsilon \sim N(0, \sigma^2)$ and $g(x, \theta) \sim F(x)$, then:

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

- Given $\mathcal{X} = \{x^t, y^t\}_{t=1}^N$ and knowing that $p(x, y) = p(y|x)p(x)$

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

Likelihood

What to optimize?

- The probability of (X, Y) given $g(x, \theta)$ is the likelihood of θ:

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

Likelihood

- The examples $(\vec{x}, y)$ are randomly sampled from all possible values
- But θ is not a random variable
- Find the θ for which the data is most probable

- In other words, find the θ of maximum likelihood

Maximum likelihood for linear regression

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

- First, take the logarithm (same maximum)

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

- We ignore $p(X)$, since it's independent of θ

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

- Replace the expression for the normal:

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

Maximum likelihood for linear regression

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

- Simplify:

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

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

Maximum likelihood for linear regression

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

- Max(likelihood) = Min(squared error)

- The θ that maximizes likelihood is the same that minimizes squared error:

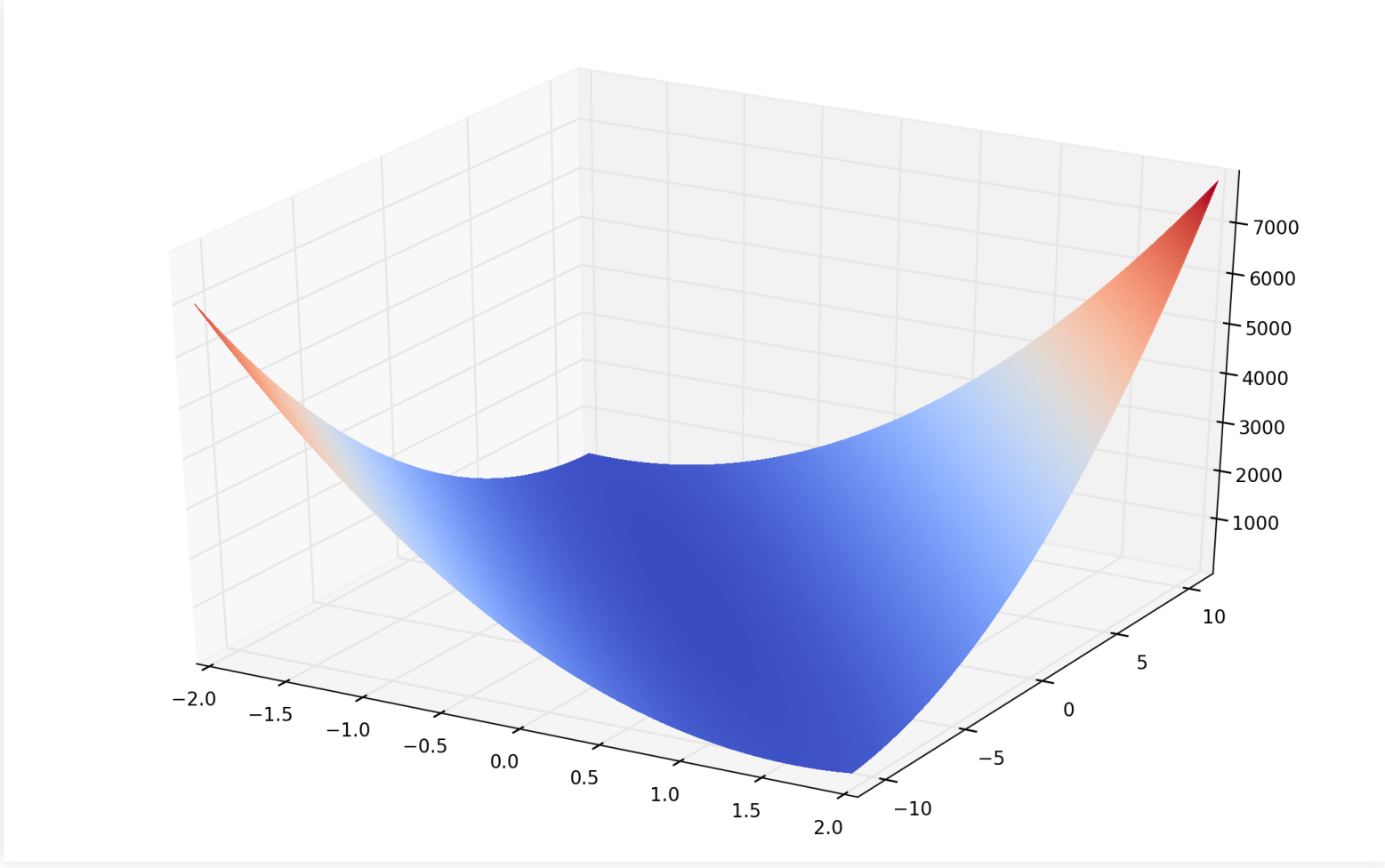
$$E(\theta|\mathcal{X}) = \sum_{t=1}^{n}[y^t - g(x^t|\theta)]^2$$

- Note: the squared error is often written like this for convenience:

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

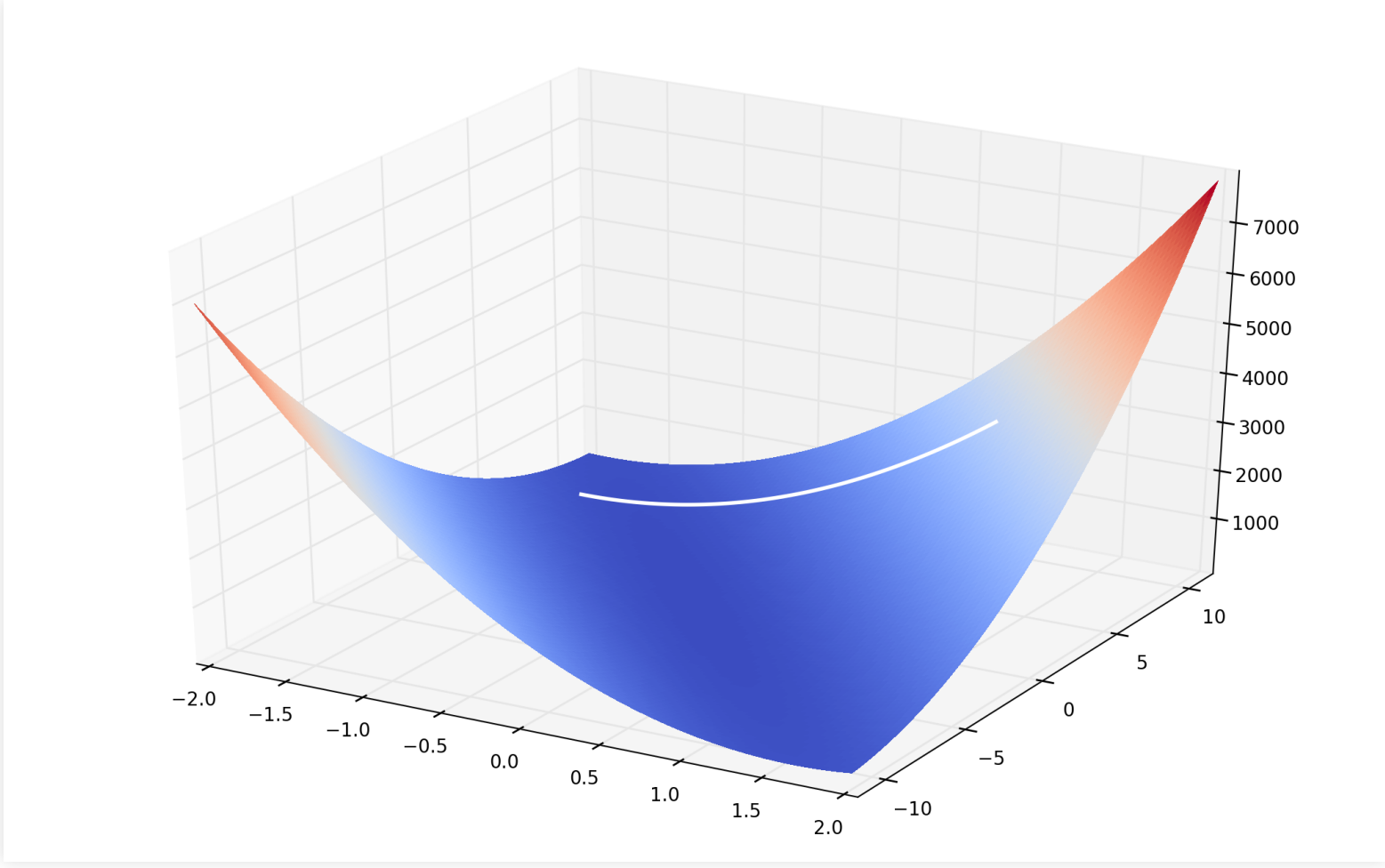
Likelihood

- Having the Loss function, we do gradient descent



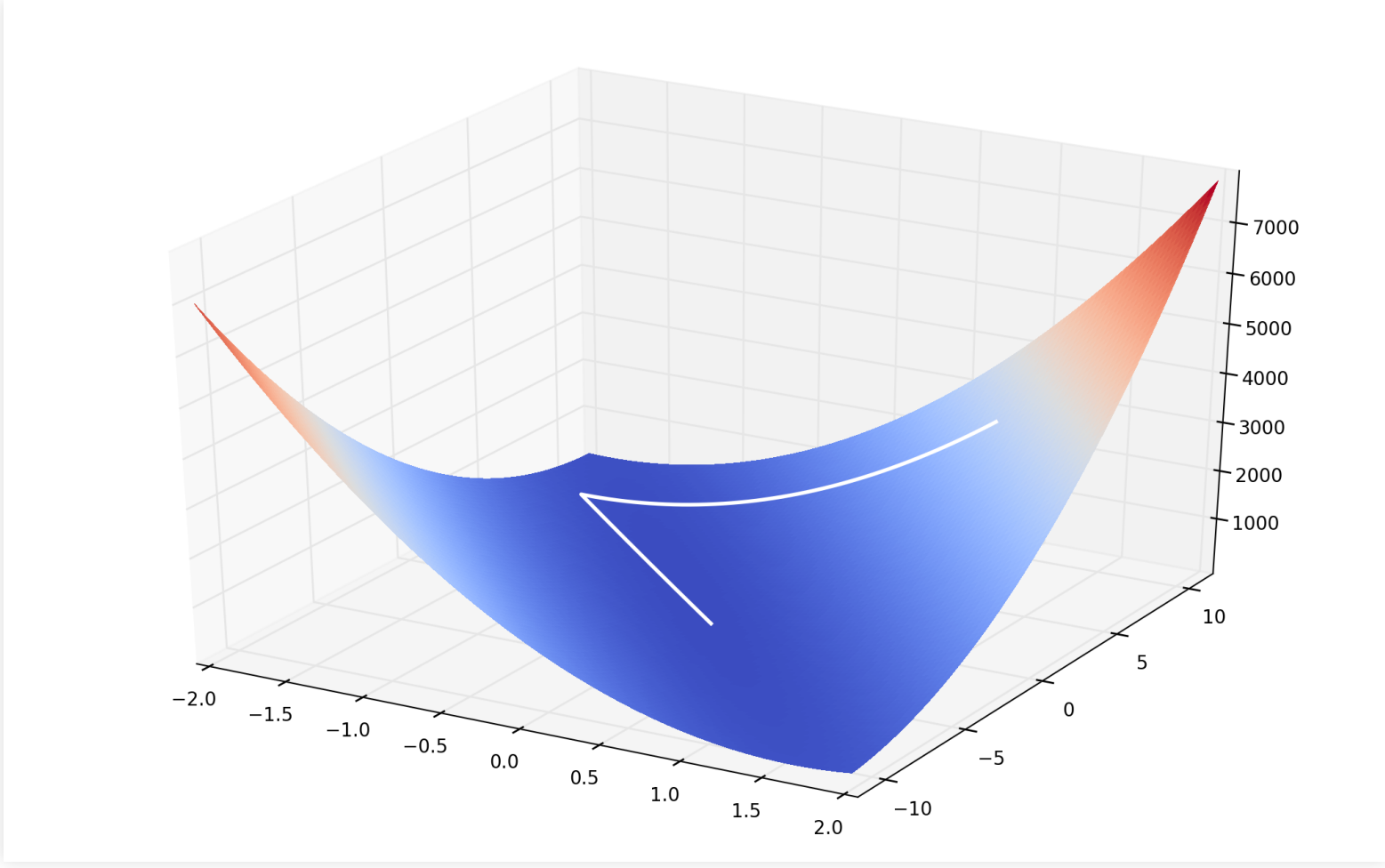
Likelihood

- Having the Loss function, we do gradient descent



Likelihood

- Having the Loss function, we do gradient descent



Maximum Likelihood

Maximum Likelihood

Finding a loss function by ML

- In general, suppose we have a set $\mathbb{X} = \{x^1, \ldots, x^m\}$ 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 (i.e. the "best" θ) is:

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

Finding a loss function by ML

- These products may lead to underflow, so best use logarithms:

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

- We can also rescale by m and obtain expectation of the log-probabilities given the empirical distribution of examples in our data:

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

- Given that our samples are drawn with $x \sim \hat{p}_{data}$, this is maximized when $p_{model}(x;\theta)$ is as close as possible to $x \sim \hat{p}_{data}$

Maximum Likelihood

Finding a loss function by ML

- The Kullback–Leibler divergence between two distributions is the expectation of the log-probability differences between them

- KL divergence between the data and the model is:

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

- Since $\hat{p}_{data}$ does not depend on θ, minimizing the KL divergence is:

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

- I.e. Maximizing likelihood is minimizing the divergence between the data distribution and what our model predicts

Maximum Likelihood

Finding a loss function by ML

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

- Minimizing KL divergence corresponds to minimizing cross-entropy between distributions
- In general, that is what we minimize: a cross-entropy loss function
- Also, in supervised learning the models usually give conditional probabilities of the target value given the features

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

Maximum Likelihood

Finding a loss function by ML

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

- For linear regression, we assumed $p(y|x) \sim \mathcal{N}(g(x,\theta),\sigma^2)$
- In this case, our loss function (cross-entropy) is the squared error

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

Maximum Likelihood

Finding a loss function by ML

- For a sigmoid predicting probability in binary classification

- Given $g(\vec{x}, \theta) = P(t_n = 1|\vec{x})$ and $t_n \in \{0, 1\}$

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

- Minimizng the cross-entropy between model and data distributions in this case corresponds to the logistic loss:

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

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

Maximum Likelihood

Finding a loss function by ML

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

- We want to maximize likelihood
- This means minimizing cross entropy between model and data
- Loss function depends on the model output:

- Regression: linear output, mean squared error
- Binary classification: class probability, sigmoid output, logistic loss
- (Also for multilabel classification, with probability for each label)
- N-ary classification, use softmax and the softmax cross entropy:

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

Summary

Activation and Loss

Summary

- Wide vs deep
- The vanishing gradients problem
- And how ReLUs (and similar) solve it
- Activations for hidden and output layers
- Loss functions

Further reading:

- Goodfellow et.al, Deep learning, Chapters 5 and 6
- Tensorflow, activation functions:

• https://www.tensorflow.org/api_guides/python/nn#Activation_Functions