

Deep learning

Chapter 1: Introduction

Bernd Bischl

Department of Statistics – LMU Munich

Winter term 2018



TEAM

- Bernd Bischl
Department of Statistics - Working Group Computational Statistics
LMU Munich
E-mail: **bernd.bischl@stat.uni-muenchen.de**
- Janek Thomas
E-mail: **Janek.Thomas@stat.uni-muenchen.de**
- Xudong Sun
E-mail: **Xudong.Sun@stat.uni-muenchen.de**
- Niklas Klein
E-mail: **Niklas.Klein@campus.lmu.de**

COURSE OUTLINE

- Brief history of deeplearning
- Introduction to neural networks
- Deep feedforward networks
- Regularization and optimization of neural networks
- Advanced nets for supervised problems:
 - Convolutional neural networks: search for patterns regardless of their position in the input (e.g an image)
 - Recurrent Nets: for sequential data, units use its internal memory to apply information about previous inputs
- Advanced nets for unsupervised problems:
 - Autoencoders: for pretraining and dimensionality reduction (the net tries to reconstruct its input)
- Maybe: automatic hyperparameter tuning

SUPERVISED AND UNSUPERVISED LEARNING

- Dealing with supervised problems means we know the ground truth of the data (i.e. we have labeled training data).
 - Our goal is to learn an output y (real valued or categorical) based on its features: $\hat{y} = f(x)$.
 - Those features are the elements of the input vectors x_1, x_2, \dots, x_p .
 - Upon the prediction we can compute the loss to evaluate our model.
- Unsupervised on the other hand means that we do not know the ground truth of the data.
 - Thus, there is no possibility to evaluate the the model against the ground truth.

A BRIEF HISTORY OF NEURAL NETWORKS

- **1943:** The first artificial neuron, the "Threshold Logic Unit (TLU)", was proposed by Warren McCulloch & Walter Pitts.
 - In this model the neuron fires a 1 if the input exceeds a certain threshold ϕ .
 - However, this model didn't have adjustable weights, so learning could only be achieved by changing the threshold ϕ .

A BRIEF HISTORY OF NEURAL NETWORKS

- **1957:** The perceptron was invented by Frank Rosenblatt.

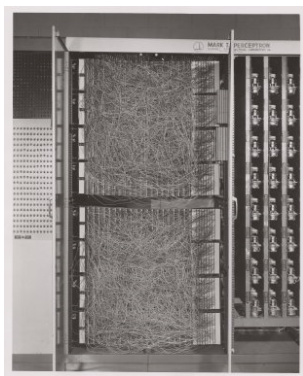


Figure: The Mark I Perceptron (<https://en.wikipedia.org/wiki/Perceptron>)

A BRIEF HISTORY OF NEURAL NETWORKS

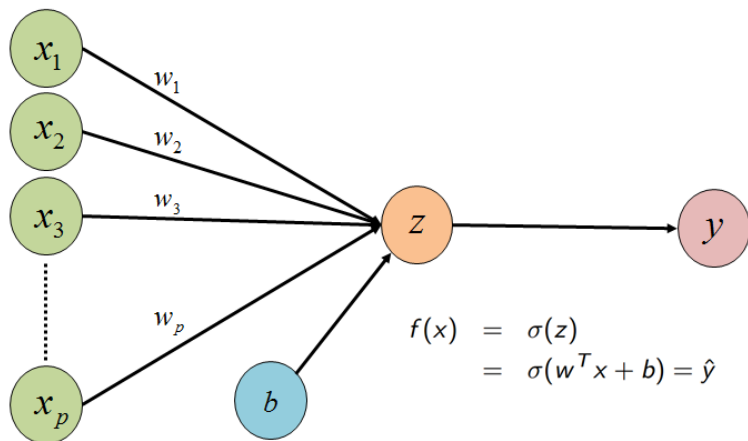


Figure: The perceptron with a single neuron z , input units x_1, x_2, \dots, x_p , weights w_1, w_2, \dots, w_p and a constant bias term b .

A BRIEF HISTORY OF NEURAL NETWORKS

- The features $x = (x_1, \dots, x_p)^T$ are connected with the neuron z via links.
- The bias term must not be confused with the statistical bias. It actually is an intercept parameter.
- We may add a column containing only ones to the feature matrix, so we can drop the bias term in notation ($x = (x_0, \dots, x_p)^T$).
- The weights $w = (w_0, \dots, w_p)$ control the impact of each input unit (i.e. feature) on the prediction.
- σ is called activation function. It can be used for a non-linear transformation of the input.
- So the neuron can be expressed as: $z = \sigma(w^T x) = \hat{y}$ and:

$$\sigma(w^T x) = \begin{cases} 1 & w^T x \geq \phi \\ 0 & \text{otherwise} \end{cases}$$

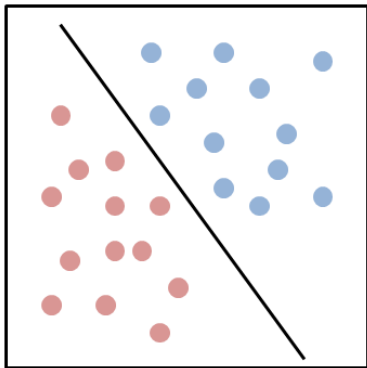
A BRIEF HISTORY OF NEURAL NETWORKS

- Linear classifier:

2 weights: linear boundary:

$$w_1x_1 + w_2x_2 = \phi$$

- Weights are learned by a delta-update rule.
- Only linearly separable problems can be learned with a perceptron.



A BRIEF HISTORY OF NEURAL NETWORKS

- The delta-update rule is an iterative learning rule for updating the weights of the inputs in a single-layer neural network, such that:

Algorithm 1 delta-update rule

- 1: **initialize** weights w_p (to 0 or to a small random value)
 - 2: **while** stopping criterion not met **do**
 - 3: compute the output of the net: $f(x) = \sigma(w^T x + b) = \hat{y}$
 - 4: update weights: $w_{p,(t+1)} = w_{p,t} + (y - \hat{y}_t)x_p$
 - 5: **end while**
-

- The stopping criterion can be either epochs or an error threshold.

A BRIEF HISTORY OF NEURAL NETWORKS

Choices for σ :

- If we choose the identity our model collapses to a simple linear regression:

$$y = \sigma(w^T x) = w^T x$$

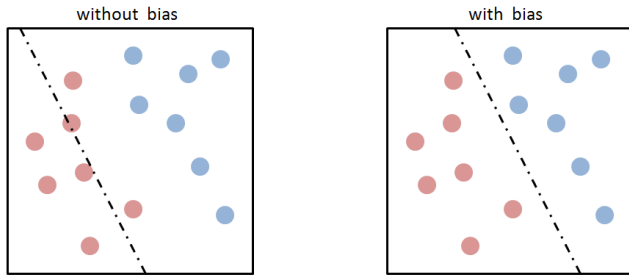
- Using the logistic function gives us:

$$y = \sigma(w^T x) = \frac{1}{1 + \exp(-w^T x)}$$

This is just logistic regression!

A BRIEF HISTORY OF NEURAL NETWORKS

- As already stated, the bias term must not be confused with the statistical bias.
- It is actually an intercept parameter, which puts the decision boundary at the correct position in the learned space.



A BRIEF HISTORY OF NEURAL NETWORKS

- **1960:** ADALINE by Bernard Widrow & Tef Hodd.
 - Weights are now adjusted according to the weighted sum of the inputs.
- **1965:** Group method of data handling (also known as polynomial neural networks) by Alexey Ivakhnenko. The first general working learning algorithm for supervised deep feedforward multilayer perceptrons.
- **1969:** The first "AI Winter" kicked in.
 - Marvin Minsky & Seymour Papert proved that a perceptron cannot solve the XOR-Problem (linear separability).
 - Less funding \Rightarrow Standstill in AI/DL research

A BRIEF HISTORY OF NEURAL NETWORKS

- **1985:** Multi-layered perceptron with backpropagation by David Rumelhart, Geoffrey Hinton and Ronald Williams.
 - Method to efficiently compute derivatives of differentiable composite functions.
 - Backpropagation was developed already in 1970 by Seppo Linnainmaa.
- **1985:** The second "AI Winter" kicked in.
 - Overly optimistic/exaggerated expectations concerning potential of AI/DL.
 - Angering investors, the phrase "AI" even reached a pseudoscience status.
 - Kernel machines and graphical models both achieved good results on many important tasks.
 - Some of the fundamental mathematical difficulties in modeling long sequences were identified.

A BRIEF HISTORY OF NEURAL NETWORKS

- **2006:** Age of deep neural networks began.
 - Geoffrey Hinton showed that a kind of neural network called deep belief network could be efficiently trained using a strategy called greedy layer-wise pretraining.
 - This wave of neural networks research popularized the use of the term deep learning to emphasize that researchers were now able to train deeper neural networks than had been possible before.
 - At this time, deep neural networks outperformed competing AI systems based on other machine learning technologies as well as hand-designed functionality.

A BRIEF HISTORY OF NEURAL NETWORKS

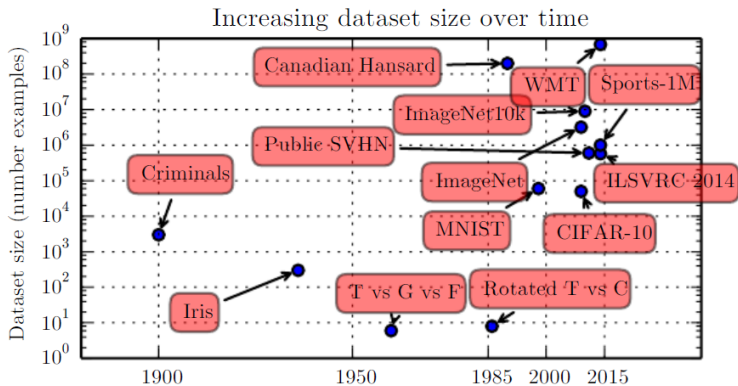


Figure: Dataset sizes over time (Goodfellow et al. (2016))

A BRIEF HISTORY OF NEURAL NETWORKS

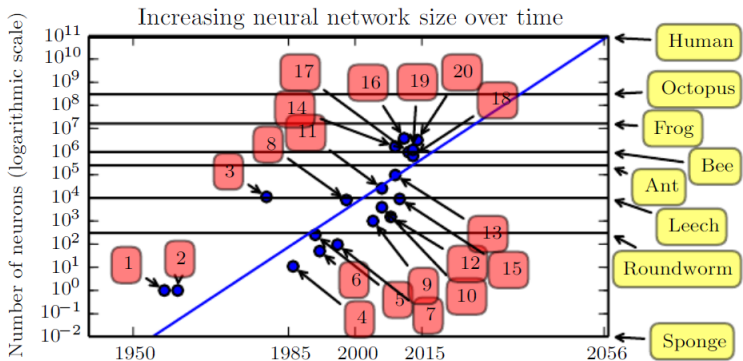


Figure: Network sizes over time. 1: Perceptron, 5: Recurrent neural network for speech recognition, 8: LeNet-5, 10: Deep belief network, 20: GoogLeNet. For more details, see: Goodfellow et al. (2016)

NEURAL NETWORKS

We will now extend the perceptron:

- Our perceptron from before only had one neuron z .
- Extension: m neurons z_1, \dots, z_m
- The neurons are arranged in layers:
 - The first layer is the input layer.
 - The intermediate layer is called “hidden layer”, because its output is not observed directly.
 - The last layer is the output layer.
- Each neuron is connected to all neurons in the previous and next layer over directed links (fully connected layers!). Directed means that information can only be passed in one direction. We'll call such graphs “feed-forward neural networks”.

NEURAL NETWORKS

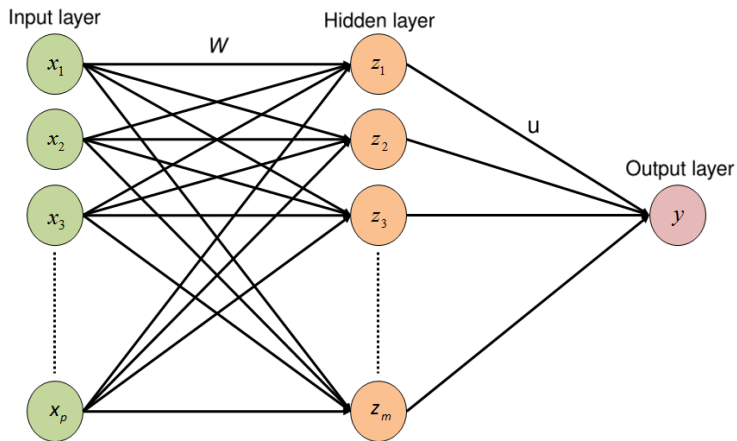


Figure: Structure of a single hidden layer, feed-forward neural network for regression or binary classification problems (bias term omitted).

NEURAL NETWORKS

- The input x is a column vector with dimensions $p \times 1$
- W is a weight matrix with dimensions $p \times m$:

$$W = \begin{pmatrix} w_{1,1} & w_{1,2} & \cdots & w_{1,m} \\ w_{2,1} & w_{2,2} & \cdots & w_{2,m} \\ \vdots & \vdots & \ddots & \vdots \\ w_{p,1} & w_{p,2} & \cdots & w_{p,m} \end{pmatrix}$$

- For example, to obtain z_1 , we pick the first column of W :

$$W_1 = \begin{pmatrix} w_{1,1} \\ w_{2,1} \\ \vdots \\ w_{p,1} \end{pmatrix}$$

and compute $z_1 = \sigma(W_1^T x)$

NEURAL NETWORKS

General notation:

- For regression or binary classification: one output unit y .
- m hidden neurons z_1, \dots, z_m , with:

$$z_i = \sigma(W_i^T x), \quad i = 1, \dots, m$$

- Linear combinations of derived features z :

$$y = u^T z, \quad z = (z_1, \dots, z_m)^T$$

NEURAL NETWORKS

Output function $\tau(u^T z)$:

- For regression one may use any function mapping from $\mathbb{R} \rightarrow \mathbb{R}$, for example the identity function:

$$\tau(u^T z) = u^T z$$

- For binary classification one could apply the sigmoidal logistic function.

NEURAL NETWORKS

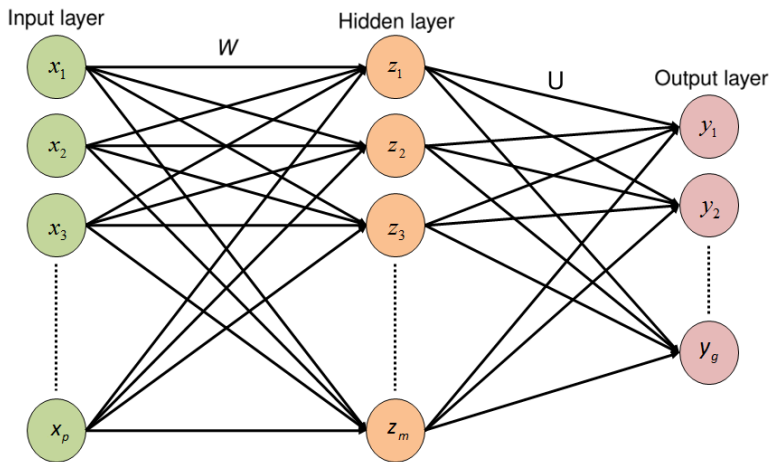


Figure: Structure of a single hidden layer, feed-forward neural network for g -class classification problems (bias term omitted).

NEURAL NETWORKS

Notation:

- For g -class classification: g output units $y = (y_1, \dots, y_g)$
- m hidden neurons z_1, \dots, z_m , with:

$$z_i = \sigma(W_i^T x), \quad i = 1, \dots, m$$

- Compute linear combinations of derived features z :

$$y_k = U_k^T z, \quad z = (z_1, \dots, z_m)^T, \quad k = 1, \dots, g$$

NEURAL NETWORKS

Output softmax function $\tau(y)$:

- For g-classification a usual choice is the softmax-function, which gives us a probability distribution over g different possible outcomes:

$$\tau_j(y) = \frac{\exp(y_j)}{\sum_{k=1}^g \exp(y_k)}$$

- This is the same transformation used in the multilogit-model!
- Derivative $\frac{\delta \tau(y)}{\delta y} = \text{diag}(\tau(y)) - \tau(y)\tau(y)^T$
- It is a “smooth” approximation of the argmax operation, so $\tau((1, 1000, 2)^T) \approx (0, 1, 0)^T$ (picks out 2nd element!).

NEURAL NETWORKS

Loss of a neural network:

- To optimize a neural network, we have to minimize a loss function $L(y, f(x))$, where y corresponds to the ground truth and $f(x)$ to the networks prediction.
- For regression, we typically use the L2 loss (rarely L1):

$$L(y, f(x)) = \frac{1}{2}(y - f(x))^2$$

- For classification we typically apply the cross entropy (binomial loss):

$$L(y, f(x)) = -\frac{1}{n} \sum_{i=1}^n \left[y_i \log f(x) + (1 - y_i) \log(1 - f(x)) \right]$$

NEURAL NETWORKS

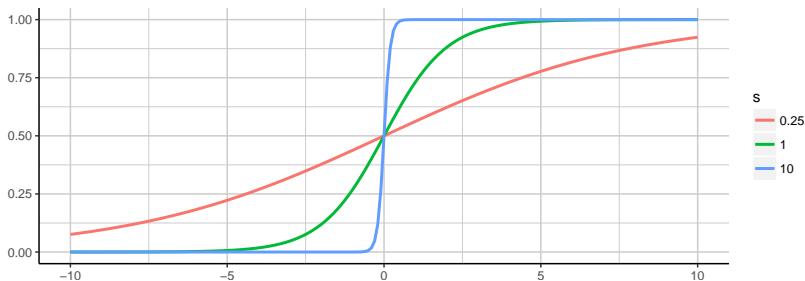
- The term cross-entropy is widely used for the negative log-likelihood of a bernoulli or softmax distribution, but that is a misnomer.
 - Any loss consisting of a negative log-likelihood is a cross-entropy between the empirical distribution of the training data and the probability distribution defined by model!
 - For example, the mean squared error is the cross-entropy between the empirical distribution and a Gaussian model.
- Thus, maximum likelihood estimation is as an attempt to make the model distribution match the empirical distribution.

NEURAL NETWORKS

Activation function σ :

- σ is in general non-linear, monotonically increasing and bounded.
- One may use the sigmoidal logistic function:

$$\sigma(v) = \frac{1}{(1 + \exp(-s \cdot v))}$$



NEURAL NETWORKS

- Some important properties of the sigmoidal logistic function include:

- limits:

$$\lim_{v \rightarrow -\infty} \sigma(v) = 0 \text{ and } \lim_{v \rightarrow \infty} \sigma(v) = 1$$

- the derivative for $s = 1$:

$$\frac{\delta \sigma(v)}{\delta v} = \frac{\exp(v)}{(1 + \exp(v))^2} = \sigma(v)(1 - \sigma(v))$$

- for any s :

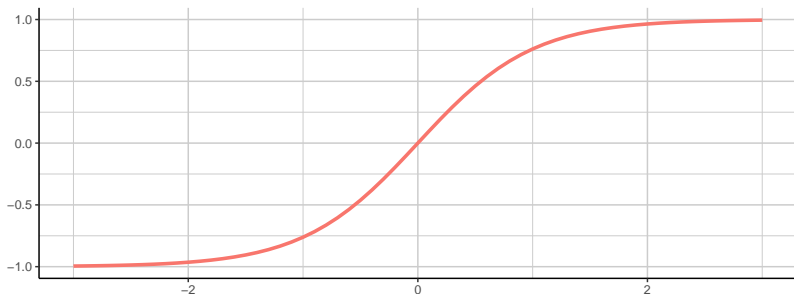
$\sigma(v)$ is symmetrical in $(0, 0.5)$

NEURAL NETWORKS

Activation function σ :

- Another choice might be the hyperbolic tangent:

$$\sigma(v) = \tanh(v) = \frac{\sinh(v)}{\cosh(v)} = 1 - \frac{2}{\exp(2v) + 1}$$



NEURAL NETWORKS

- Some important properties of the hyperbolic tangent function include:

- limits:

$$\lim_{v \rightarrow -\infty} \sigma(v) = -1 \text{ and } \lim_{v \rightarrow \infty} \sigma(v) = 1$$

- derivative:

$$\frac{\delta \sigma(v)}{\delta v} = 1 - \tanh^2(v)$$

- symmetry:

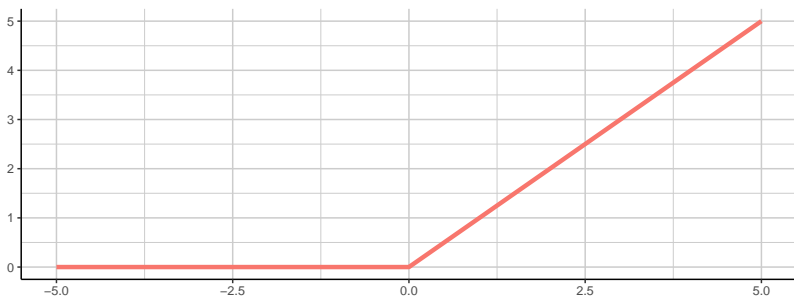
$\sigma(v)$ is symmetrical in $(0, 0)$

NEURAL NETWORKS

Activation function σ :

- Currently the most popular choice is the ReLU (rectified linear unit):

$$\sigma(v) = \max(0, v)$$



NEURAL NETWORKS

- Some important properties of the relu function include:

- limits:

$$\lim_{v \rightarrow -\infty} \sigma(v) = 0 \text{ and } \lim_{v \rightarrow \infty} \sigma(v) = \infty$$

- derivative:

$$\frac{\delta \sigma(v)}{\delta v} = \begin{cases} 1 & \text{if } v > 0 \\ 0 & \text{else} \end{cases}$$

EXAMPLE: XOR PROBLEM

- Suppose we have four data points

$$X = \{[0, 0]^T, [0, 1]^T, [1, 0]^T, [1, 1]^T\}$$

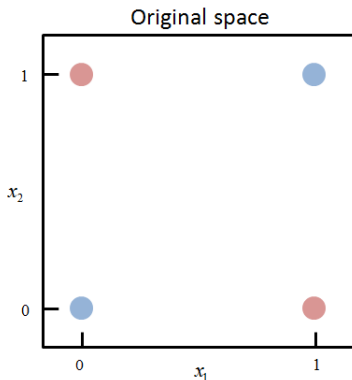
- The XOR gate (exclusive or) returns true, when an odd number of inputs are true:

x_1	x_2	XOR = y
0	0	0
0	1	1
1	0	1
1	1	0

- Can you learn the target function with a logistic regression model?

EXAMPLE: XOR PROBLEM

- Logistic regression cannot solve this problem and will always output 0.5.
In fact, any model using simple hyperplanes for separation can't (including a perceptron).
- A small neural net can easily solve the problem by transforming the space!



EXAMPLE: XOR PROBLEM

- Consider the following model:

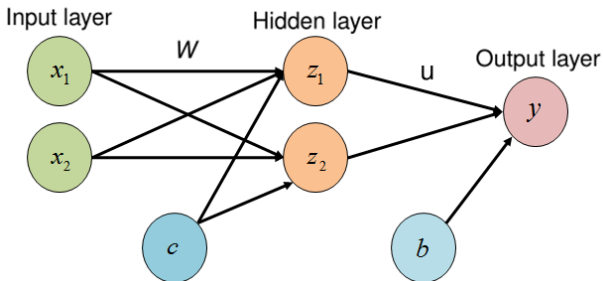


Figure: A neural network with two neurons in the hidden layer. The matrix W describes the mapping from x to z . The vector u from z to y . The c and b are biases.

EXAMPLE: XOR PROBLEM

- Let us treat the XOR task as a regression problem and apply ReLU as activation function. We can represent the models architecture by the following equation:

$$\begin{aligned}f(x|\theta) &= f(x|W, c, u, b) \\&= u^T \sigma(W^T x + c) + b \\&= u^T \max\{0, W^T x + c\} + b\end{aligned}$$

- So how many parameters does our model have?
 - In a fully connected neural net, the number of connections between the nodes equals our parameters:

$$\underbrace{(2 \times 2)}_W + \underbrace{(2 \times 1)}_c + \underbrace{(2 \times 1)}_u + \underbrace{(1)}_b = 9$$

EXAMPLE: XOR PROBLEM

$$\text{Let } W = \begin{pmatrix} 1 & 1 \\ 1 & 1 \end{pmatrix}, c = \begin{pmatrix} 0 \\ -1 \end{pmatrix}, u = \begin{pmatrix} 1 \\ -2 \end{pmatrix}, b = 0$$

$$X = \begin{pmatrix} 0 & 0 \\ 0 & 1 \\ 1 & 0 \\ 1 & 1 \end{pmatrix}, XW = \begin{pmatrix} 0 & 0 \\ 1 & 1 \\ 1 & 1 \\ 2 & 2 \end{pmatrix}, XW + c = \begin{pmatrix} 0 & -1 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix}$$

$$z = \max\{0, XW + c\} = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix}$$

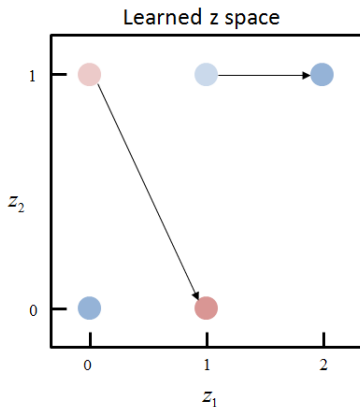
- Note that we computed all examples at once.

EXAMPLE: XOR PROBLEM

- Our transformed space has form

$$z = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix}$$

which is easily separable.

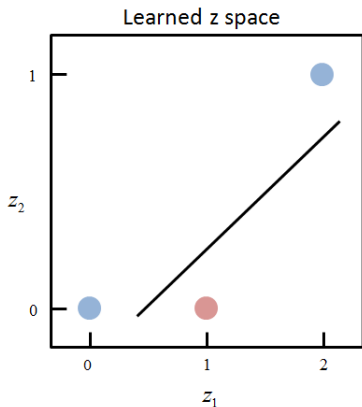


EXAMPLE: XOR PROBLEM

- Our transformed space has form

$$z = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix}$$

which is easily separable.



EXAMPLE: XOR PROBLEM

- In a final step we have to multiply the activated values of matrix z with the vector u :

$$f(x; W, c, u, b) = \begin{pmatrix} 0 & 0 \\ 1 & 0 \\ 1 & 0 \\ 2 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ -2 \end{pmatrix} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 0 \end{pmatrix}$$

- This solves the XOR problem perfectly!

x_1	x_2	XOR = y
0	0	0
0	1	1
1	0	1
1	1	0

TRAINING NEURAL NETWORKS

- In this simple example we actually “guessed” the values of the parameters for W , c , u and b .
- That won't work for more sophisticated problems!
- Training of neural nets is composed of two iterative steps:
 - ➊ Forward pass: the information of the inputs flow through the model to produce a prediction.
Based on that, we compute a loss which is sometimes called the cost.
 - ➋ Backward pass: the information of the error flows backwards through the model.
Thereby we use the error values to calculate the gradient of the loss with respect to each weight.
In a final step we update the weights (i.e. “move” them in the direction of the steepest descent of the loss).

REVISION OF GRADIENT DESCENT

- First we need to recall the method of gradient descent in numerical optimization.
- Let $f(x)$ be an arbitrary, differentiable, unrestricted target function, which we want to minimize.
 - We can calculate the gradient $\nabla f(x)$, which always points in the direction of the steepest ascent.
 - Thus $-\nabla f(x)$ points in the direction of the steepest descent!

REVISION OF GRADIENT DESCENT

- Standing at a point x_k during minimization, we can improve this point by doing the following step:

$$f(x_{k+1}) = f(x_k) - \nu \nabla f(x_k)$$

“Walking down the hill, towards the valley.”

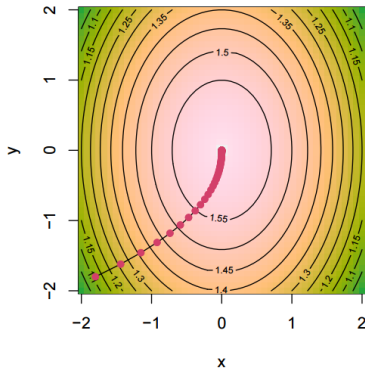
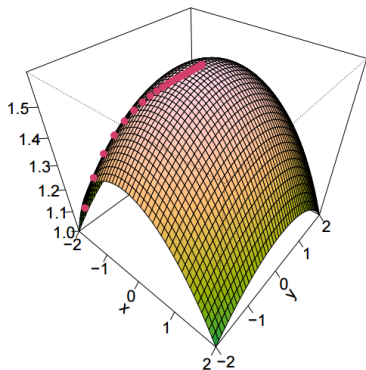
- ν determined the length of the step and is called step size. To find the optimal ν we need to look at:

$$g(\nu) = f(x_k) - \nu \nabla f(x_k) = \min!$$

- This minimization problem only has one real parameter, and is therefore “easy” to solve. These kind of methods are known as line search methods.

REVISION OF GRADIENT DESCENT

$$f(x, y) = \left(-1 \cdot \left(x^2 + \frac{y^2}{2} \right) + 16 \right) / 10$$



COMPUTATIONAL GRAPHS

- Computational graphs are a very helpful language to describe algorithms like the backpropagation.
- Each node describes a variable.
- Operations are functions applied to one or more variables.

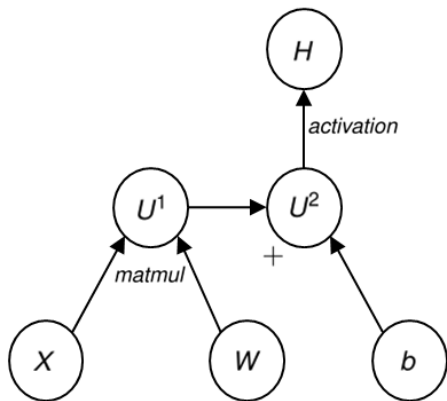


Figure: The computational graph for the expression $H = \sigma(XW + b)$.

CHAIN RULE OF CALCULUS

- The chain rule can be used to compute derivatives of the composition of two or more functions.
- Let $x \in \mathbb{R}^m$, $y \in \mathbb{R}^n$,
 $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ and $f : \mathbb{R}^n \rightarrow \mathbb{R}$.
- If $y = g(x)$ and $z = f(y)$, the chain rule yields

$$\frac{\delta z}{\delta x_i} = \sum_j \frac{\delta z}{\delta y_j} \frac{\delta y_j}{\delta x_i}$$

or in vector notation

$$\nabla_x z = \left(\frac{\delta y}{\delta x} \right)^T \nabla_y z,$$

where $\frac{\delta y}{\delta x}$ is the $n \times m$ jacobian matrix of g .

CHAIN RULE OF CALCULUS

- Suppose the following computational graph.
- To compute the derivative of $\frac{\delta z}{\delta w}$ we need to recursively apply the chain rule. That is:

$$\begin{aligned}\frac{\delta z}{\delta w} &= \frac{\delta z}{\delta y} \frac{\delta y}{\delta x} \frac{\delta x}{\delta w} \\ &= f'_3(y) f'_2(x) f'_1(w) \\ &= f'_3(f_2(f_1(w))) f'_2(f_1(w)) f'_1(w)\end{aligned}$$

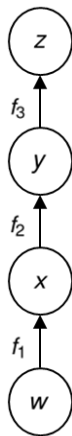


Figure: A computational graph, such that $x = f_1(w)$, $y = f_2(x)$ and $z = f_3(y)$.

CHAIN RULE OF CALCULUS

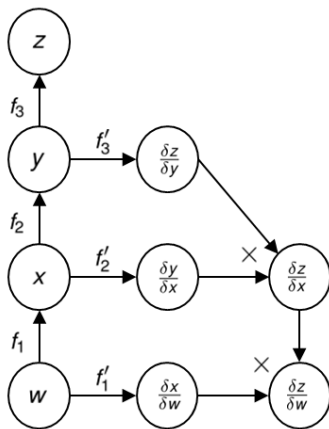


Figure: Applying the chain rule to the example yields us a computational graph with a symbolic description of the derivatives.

BACKPROPAGATION

- Backpropagation is a method used to calculate the error contribution of each weight after a batch of data is processed.
- This is used by an enveloping optimization algorithm to adjust the weight of each neuron (e.g. SGD, Adam, ..).
- To apply backpropagation, we have to choose a loss function $L(y, f(x))$, which we would like to minimize.

WEIGHT UPDATE RULE

- After obtaining the gradients we can finally update the weights according to

$$w_{i+1} = w_i - \alpha \cdot \frac{\delta L(y, f(x))}{\delta w},$$

where α corresponds to a learning rate with $\alpha \in (0, 1)$.

- We can think of how fast the network will abandon its old beliefs. Thus, we would like to apply a learning rate which is low enough to converge to something useful, but also high enough that we do not have to spend years of training.
- We will inspect the details of the weight update in the optimization chapter.

BACKPROPAGATION EXAMPLE

- Let us recall the XOR example, but this time with randomly initialized weights.
- For activations we apply the sigmoidal logistic function.
- To perform one forward and one backward pass we feed our neural network exemplarily with $X = [1, 0]^T$ (positive sample).
- We will divide the forward pass into four steps:
 - the inputs of z_m : $z_{i,in}$
 - the activations of z_m : $z_{i,out}$
 - the input of \hat{y} : \hat{y}_{in}
 - and finally the activation of \hat{y} : \hat{y}_{out}
- Then we compute the backward pass and apply backpropagation exemplarily to update two weights.
- Finally we evaluate the model with our updated weights.

BACKPROPAGATION EXAMPLE

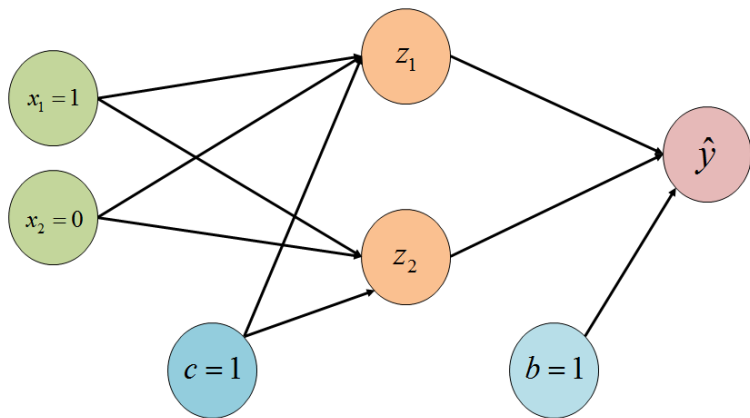
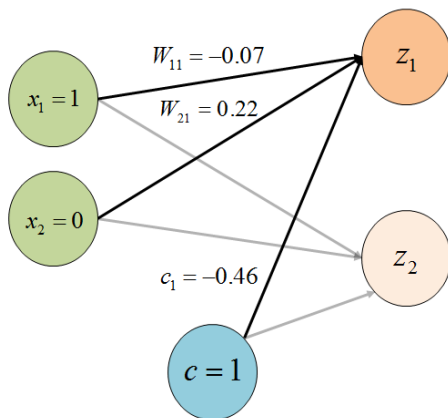


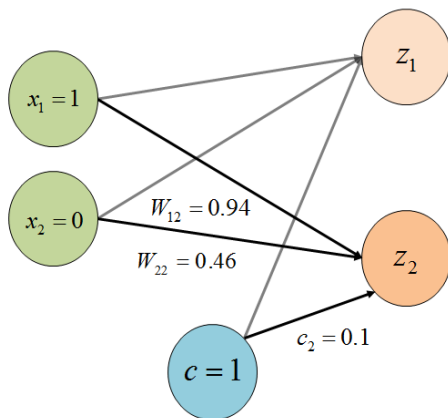
Figure: A neural network with two neurons in the hidden layer and two bias terms c and b . Based on the values for the weights, we will perform one forward and one backward pass.

BACKPROPAGATION EXAMPLE



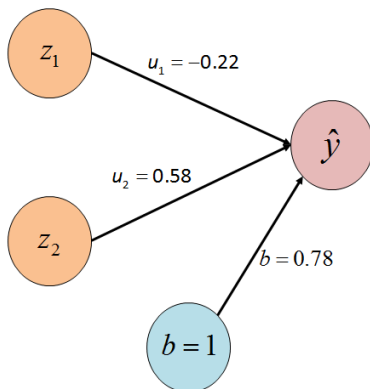
- $z_{1,in} = 1 \cdot (-0.07) + 0 \cdot 0.22 + 1 \cdot (-0.46) = -0.53$
- $z_{1,out} = \frac{1}{(1 + \exp(-(-0.53)))} = 0.3705$

BACKPROPAGATION EXAMPLE



- $z_{2,in} = 1 \cdot 0.94 + 0 \cdot 0.46 + 1 \cdot 0.1 = 1.04$
- $z_{2,out} = \frac{1}{(1 + \exp(-1.04))} = 0.7389$

BACKPROPAGATION EXAMPLE



- $\hat{y}_{in} = 0.3705 \cdot (-0.22) + 0.7389 \cdot 0.58 + 1 \cdot 0.78 = 1.1122$
- $\hat{y}_{out} = \frac{1}{(1 + \exp(-1.1122))} = 0.7525$

BACKPROPAGATION EXAMPLE

- The forward pass of our neural network predicted a value of

$$\hat{y}_{out} = 0.7525$$

- Now we would like to evaluate the result by computing a suitable error rate.
- For convenience, we use the squared error to evaluate our result:

$$\begin{aligned} L(y, f(x)) &= \frac{1}{2}(f(x) - f(x|\theta))^2 = \frac{1}{2}(y - \hat{y}_{out})^2 \\ &= \frac{1}{2}(1 - 0.7525)^2 = 0.0306 \end{aligned}$$

BACKPROPAGATION EXAMPLE

- Assume we would like to know how much and in which direction a change in u_1 affects the total error. To this we have to recursively apply the chain rule and compute:

$$\frac{\delta L(y, f(x))}{\delta u_1} = \frac{\delta L(y, f(x))}{\delta \hat{y}_{out}} \cdot \frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}} \cdot \frac{\delta \hat{y}_{in}}{\delta u_1}$$

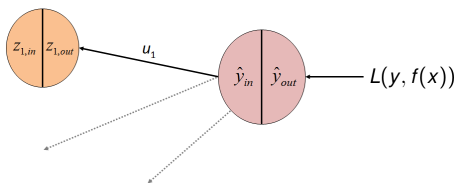


Figure: Snippet from our neural network showing the backward path to compute the gradient with respect to weight u_1 .

BACKPROPAGATION EXAMPLE

- 1st step (backwards)

$$\begin{aligned}\frac{\delta L(y, f(x))}{\delta \hat{y}_{out}} &= \frac{\delta}{\delta \hat{y}_{out}} \frac{1}{2} (y - \hat{y}_{out})^2 = -(y - \hat{y}_{out}) \\ &= -(1 - 0.7525) = -0.2475\end{aligned}$$

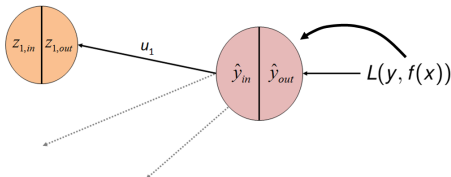


Figure: The first term of our chain rule $\frac{\delta L(y, f(x))}{\delta \hat{y}_{out}}$

BACKPROPAGATION EXAMPLE

- 2nd step (backwards). $\hat{y}_{out} = \sigma(\hat{y}_{in})$ and we apply the rule for σ'

$$\begin{aligned}\frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}} &= \sigma(\hat{y}_{in})(1 - \sigma(\hat{y}_{in})) \\ &= 0.7525(1 - 0.7525) = 0.1862\end{aligned}$$

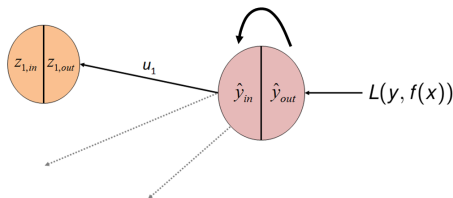


Figure: The second term of our chain rule $\frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}}$

BACKPROPAGATION EXAMPLE

- 3rd step (backwards). $\hat{y}_{in} = u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + b \cdot 1$

$$\frac{\delta \hat{y}_{in}}{\delta u_1} = z_{1,out} = 0.3705$$

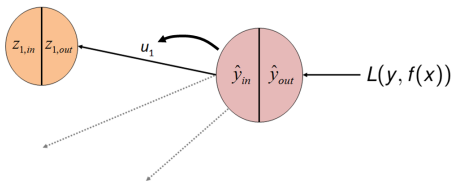


Figure: The third term of our chain rule $\frac{\delta \hat{y}_{in}}{\delta u_1}$

BACKPROPAGATION EXAMPLE

- Finally we are able to plug all three parts together and obtain:

$$\begin{aligned}\frac{\delta L(y, f(x))}{\delta u_1} &= \frac{\delta L(y, f(x))}{\delta \hat{y}_{out}} \cdot \frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}} \cdot \frac{\delta \hat{y}_{in}}{\delta u_1} \\ &= -0.2475 \cdot 0.1862 \cdot 0.3705 \\ &= -0.0171\end{aligned}$$

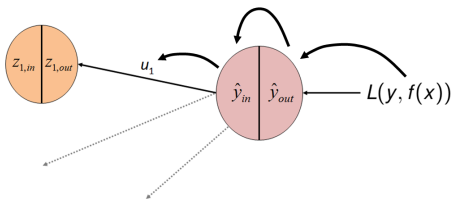


Figure: All three terms of our chain rule $\frac{\delta L(y, f(x))}{\delta u_1} = \frac{\delta L(y, f(x))}{\delta \hat{y}_{out}} \cdot \frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}} \cdot \frac{\delta \hat{y}_{in}}{\delta u_1}$

BACKPROPAGATION EXAMPLE

- Consider a learning rate of $\alpha = 0.5$. Then we obtain:

$$\begin{aligned}u_{1,new} &= u_{1,old} - \alpha \cdot \frac{\delta L(y, f(x))}{\delta u_1} \\&= -0.22 - 0.5 \cdot (-0.0171) \\&= -0.2115\end{aligned}$$

BACKPROPAGATION EXAMPLE

- Now assume we would also like to do the same for weight W_{11} .
This time we have to compute:

$$\frac{\delta L(y, f(x))}{\delta W_{11}} = \frac{\delta L(y, f(x))}{\delta \hat{y}_{out}} \cdot \frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}} \cdot \frac{\delta \hat{y}_{in}}{\delta z_{1,out}} \cdot \frac{\delta z_{1,out}}{\delta z_{1,in}} \cdot \frac{\delta z_{1,in}}{\delta W_{11}}$$

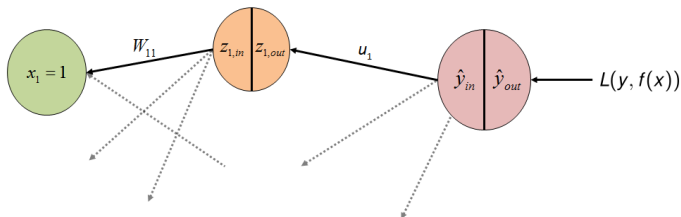


Figure: Snippet from our neural network showing the backward path to compute the gradient with respect to weight W_{11} .

BACKPROPAGATION EXAMPLE

- We already know $\frac{\delta L(y, f(x))}{\delta \hat{y}_{out}}$ and $\frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}}$ from the computations before.

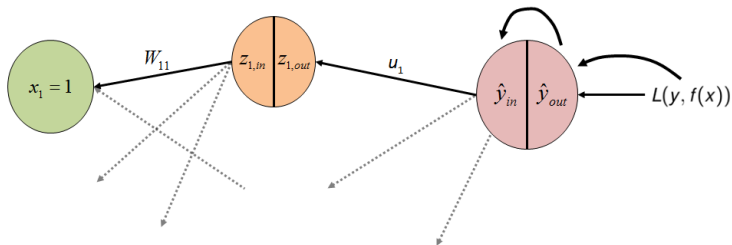


Figure: The first and second term of our chain rule $\frac{\delta L(y, f(x))}{\delta \hat{y}_{out}}$ and $\frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}}$

BACKPROPAGATION EXAMPLE

- With $\hat{y}_{in} = u_1 \cdot z_{1,out} + u_2 \cdot z_{2,out} + b \cdot 1$ we can compute:

$$\frac{\delta \hat{y}_{in}}{\delta z_{1,out}} = u_1 = -0.22$$

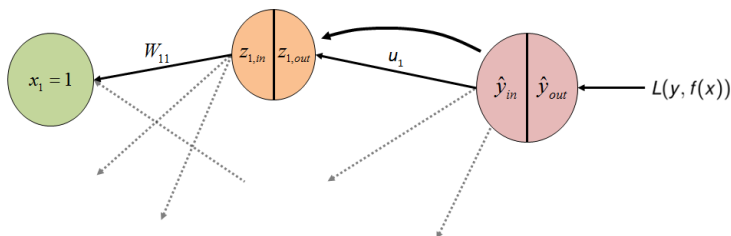


Figure: The third term of our chain rule $\frac{\delta \hat{y}_{in}}{\delta z_{1,out}}$

BACKPROPAGATION EXAMPLE

- Next, we need

$$\begin{aligned}\frac{\delta z_{1,out}}{\delta z_{1,in}} &= \sigma(z_{1,in})(1 - \sigma(z_{1,in})) \\ &= 0.3705(1 - 0.3705) = 0.2332\end{aligned}$$

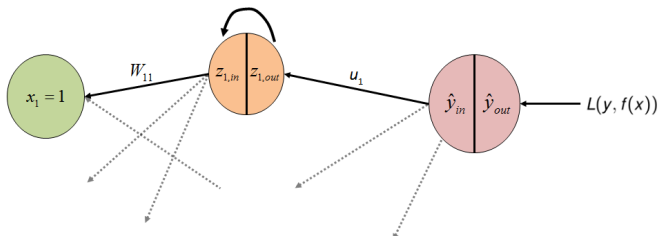


Figure: The fourth term of our chain rule $\frac{\delta z_{1,out}}{\delta z_{1,in}}$

BACKPROPAGATION EXAMPLE

- With $z_{1,in} = x_1 \cdot W_{11} + x_2 \cdot W_{21} + c \cdot 1$ we can compute the last component:

$$\frac{\delta z_{1,in}}{\delta W_{11}} = x_1 = 1$$

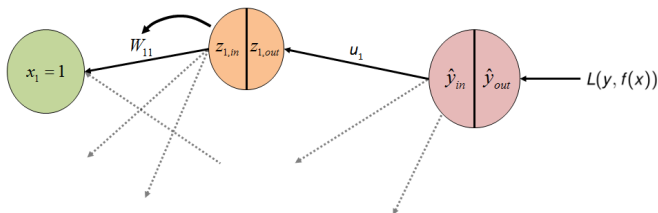


Figure: The fifth term of our chain rule $\frac{\delta z_{1,in}}{\delta W_{11}}$

BACKPROPAGATION EXAMPLE

- Plugging all five components together yields us:

$$\begin{aligned}\frac{\delta L(y, f(x))}{\delta W_{11}} &= \frac{\delta L(y, f(x))}{\delta \hat{y}_{out}} \cdot \frac{\delta \hat{y}_{out}}{\delta \hat{y}_{in}} \cdot \frac{\delta \hat{y}_{in}}{\delta z_{1,out}} \cdot \frac{\delta z_{1,out}}{\delta z_{1,in}} \cdot \frac{\delta z_{1,in}}{\delta W_{11}} \\ &= (-0.2475) \cdot 0.1862 \cdot (-0.22) \cdot 0.2332 \cdot 1 \\ &= 0.0024\end{aligned}$$

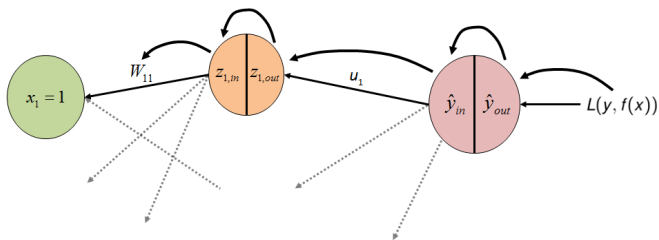


Figure: All five terms of our chain rule

BACKPROPAGATION EXAMPLE

- Consider the same learning rate of $\alpha = 0.5$. Then we obtain:

$$\begin{aligned}W_{11,new} &= W_{11,old} - \alpha \cdot \frac{\delta L(y, f(x))}{\delta W_{11}} \\&= -0.07 - 0.5 \cdot 0.0024 = -0.0712\end{aligned}$$

- We would now like to check how the performance has improved.
Our updated weights are:

$$W = \begin{pmatrix} -0.0712 & 0.9426 \\ 0.22 & 0.46 \end{pmatrix}, c = \begin{pmatrix} -0.4546 \\ 0.1003 \end{pmatrix},$$

$$u = \begin{pmatrix} -0.2115 \\ 0.5970 \end{pmatrix} \text{ and } b = 0.7980$$

BACKPROPAGATION EXAMPLE

- Plugging all values into our model yields

$$f(x|W, c, u, b) = 0.7615$$

and a squared error of

$$L(y, f(x)) = \frac{1}{2}(1 - 0.7615)^2 = 0.0284.$$

- The initial weights predicted $\hat{y} = 0.7525$ and a slightly higher error value of $L(y, f(x)) = 0.0306$.
- Keep in mind that this is the result of only one training iteration. When applying a neural network, one usually conducts thousands of those.

UNIVERSAL APPROXIMATION PROPERTY

Theorem. Let $h : \mathbb{R} \rightarrow \mathbb{R}$ be a continuous, non-constant, bounded, and monotonically increasing function. Let $C \subset \mathbb{R}^p$ be compact, and let $\mathcal{C}(C)$ denote the space of continuous functions $C \rightarrow \mathbb{R}$. Then, given a function $g \in \mathcal{C}(C)$ and an accuracy $\varepsilon > 0$, there exists a hidden layer size $N \in \mathbb{N}$ and a set of coefficients $w_i^{(1)} \in \mathbb{R}^p$, $w_i^{(2)}$, $b_i \in \mathbb{R}$ (for $i \in \{1, \dots, N\}$), such that

$$f : K \rightarrow \mathbb{R}; \quad f(x) = \sum_{i=1}^N w_i^{(2)} \cdot h\left((w_i^{(1)})^T x + b_i\right)$$

is an ε -approximation of g , that is,

$$\|f - g\|_{\infty} := \max_{x \in K} |f(x) - g(x)| < \varepsilon .$$

The theorem extends trivially to multiple outputs.

UNIVERSAL APPROXIMATION PROPERTY

Corollary. Neural networks with a single sigmoidal hidden layer and linear output layer are universal approximators.

- This means that for a given target function g there exists a sequence of networks $(f_k)_{k \in \mathbb{N}}$ that converges (pointwise) to the target function.
- Usually, as the networks come closer and closer to g , they will need more and more hidden neurons.
- A network with fixed layer sizes can only model a subspace of all continuous functions. Its dimensionality is limited by the number of weights.
- The continuous functions form an infinite dimensional vector space. Therefore arbitrarily large hidden layer sizes are needed.

UNIVERSAL APPROXIMATION PROPERTY

- Why is universal approximation a desirable property?
- Recall the definition of a Bayes optimal hypothesis $h^* : X \rightarrow Y$. It is the best possible hypothesis (model) for the given problem: it has minimal loss averaged over the data generating distribution.
- So ideally we would like the neural network (or any other learner) to approximate the Bayes optimal hypothesis.
- Usually we do not manage to learn h^* .
- This is because we do not have enough (infinite) data. We have no control over this, so we have to live with this limitation.
- But we do have control over which model class we use.

UNIVERSAL APPROXIMATION PROPERTY

- Universal approximation \Rightarrow approximation error tends to zero as hidden layer size tends to infinity.
- Positive approximation error implies that no matter how good the data, we cannot find the optimal model.
- This bears the risk of systematic under-fitting, which can be avoided with a universal model class.

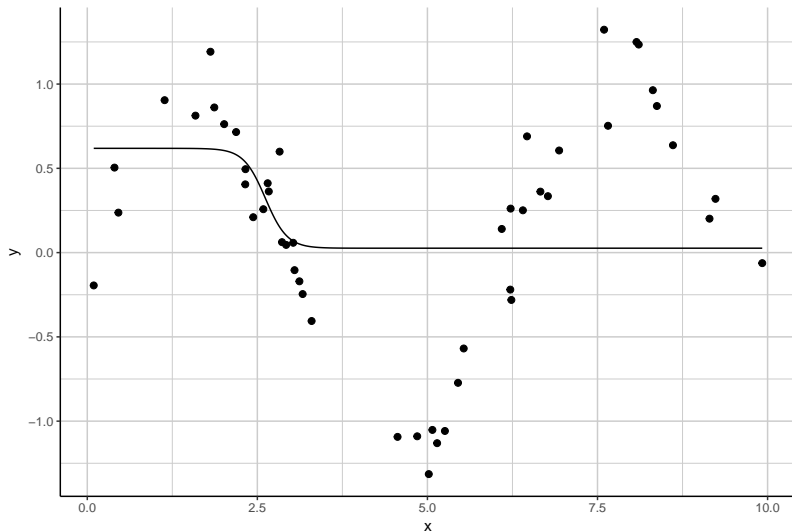
UNIVERSAL APPROXIMATION PROPERTY

- As we know, there are also good reasons for restricting the model class.
- This is because a flexible model class with universal approximation ability often results in over-fitting, which is no better than under-fitting.
- Thus, “universal approximation \Rightarrow low approximation error”, but at the risk of a substantial learning error.
- In general, models of intermediate flexibility give the best predictions. For neural networks this amounts to a reasonably sized hidden layer.

REGRESSION: 100 TRAINING ITERATIONS

nnet: size=1; maxit=100

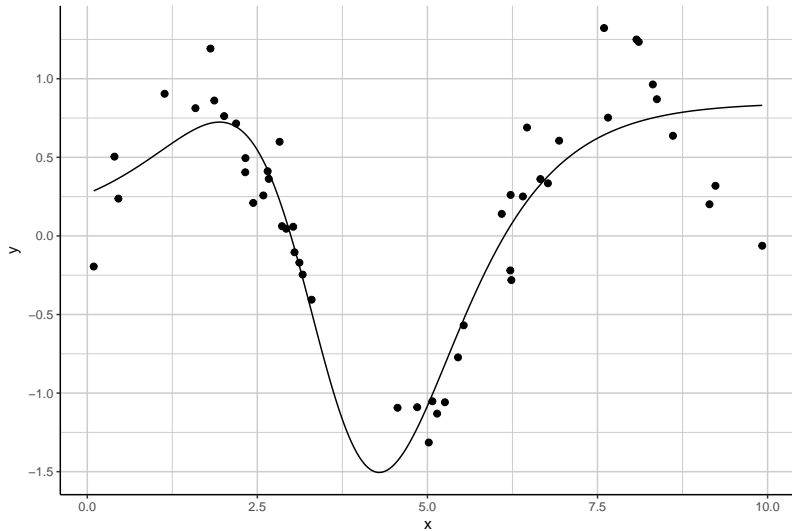
Train: mse=0.391; CV: mse.test.mean=0.418



REGRESSION: 100 TRAINING ITERATIONS

nnet: size=2; maxit=100

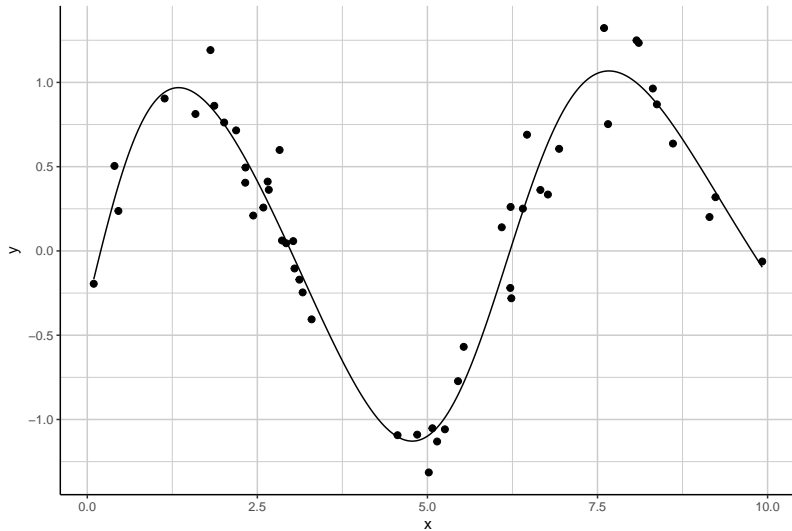
Train: mse=0.0888; CV: mse.test.mean=0.113



REGRESSION: 100 TRAINING ITERATIONS

nnet: size=3; maxit=100

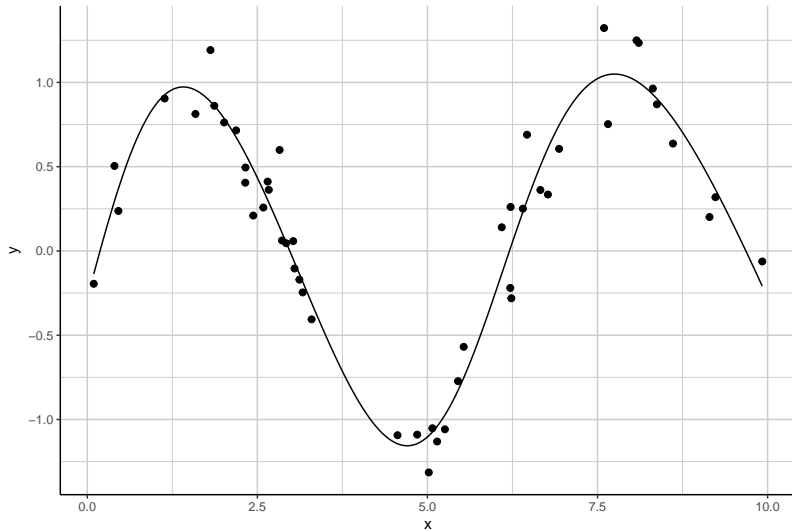
Train: mse=0.0324; CV: mse.test.mean=0.0897



REGRESSION: 100 TRAINING ITERATIONS

nnet: size=4; maxit=100

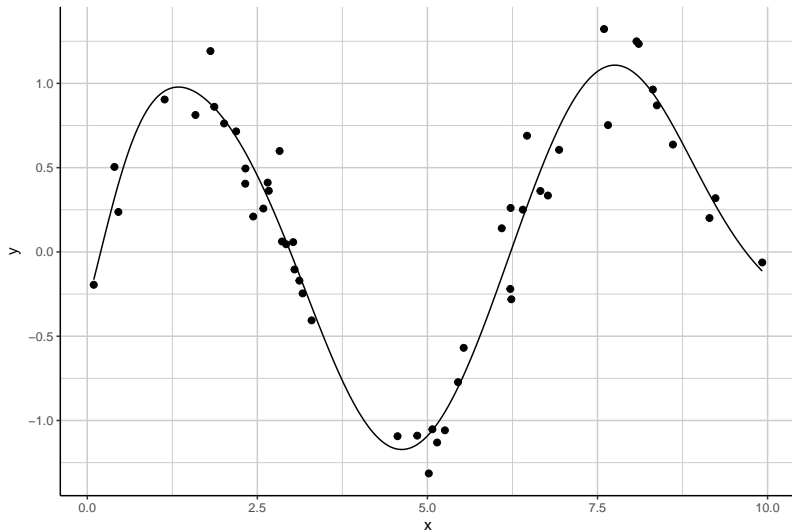
Train: mse=0.032; CV: mse.test.mean=0.0535



REGRESSION: 100 TRAINING ITERATIONS

nnet: size=5; maxit=100

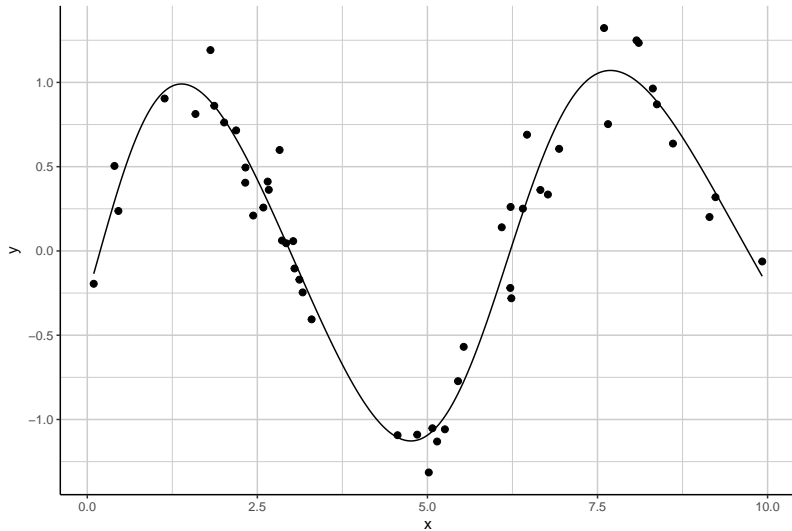
Train: mse=0.0304; CV: mse.test.mean=0.0479



REGRESSION: 100 TRAINING ITERATIONS

nnet: size=6; maxit=100

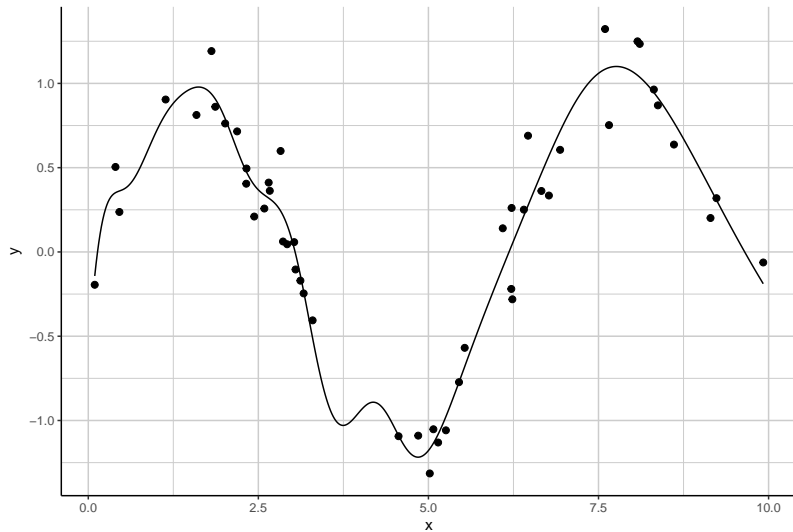
Train: mse=0.0322; CV: mse.test.mean=0.0836



REGRESSION: 100 TRAINING ITERATIONS

nnet: size=100; maxit=100

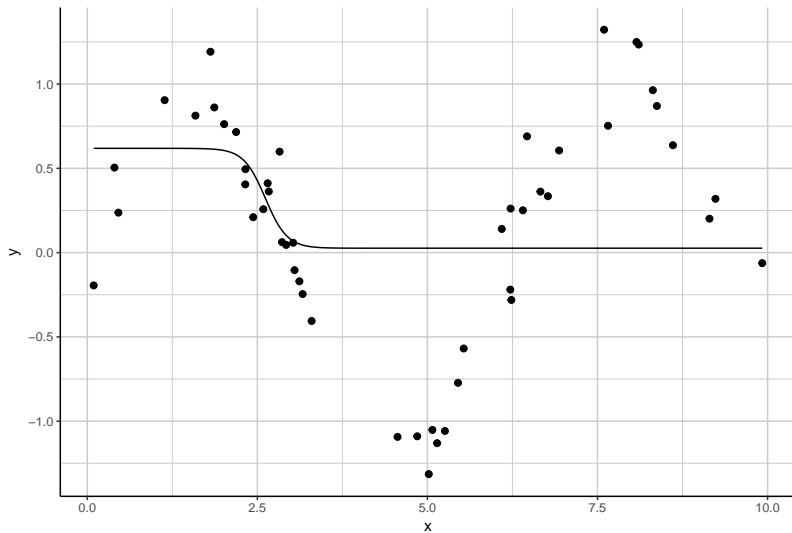
Train: mse=0.0262; CV: mse.test.mean=0.15



REGRESSION: 1000 TRAINING ITERATIONS

nnet: size=1; maxit=1e+03

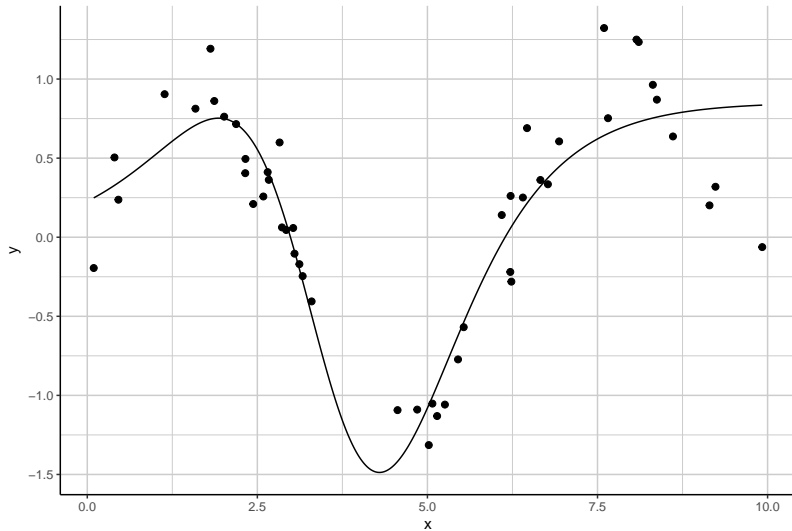
Train: mse=0.391; CV: mse.test.mean=0.419



REGRESSION: 1000 TRAINING ITERATIONS

nnet: size=2; maxit=1e+03

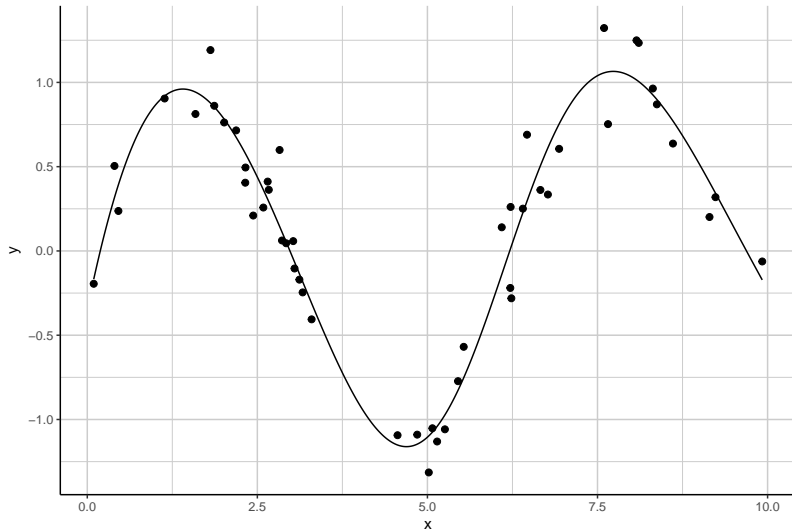
Train: mse=0.0878; CV: mse.test.mean=0.112



REGRESSION: 1000 TRAINING ITERATIONS

nnet: size=3; maxit=1e+03

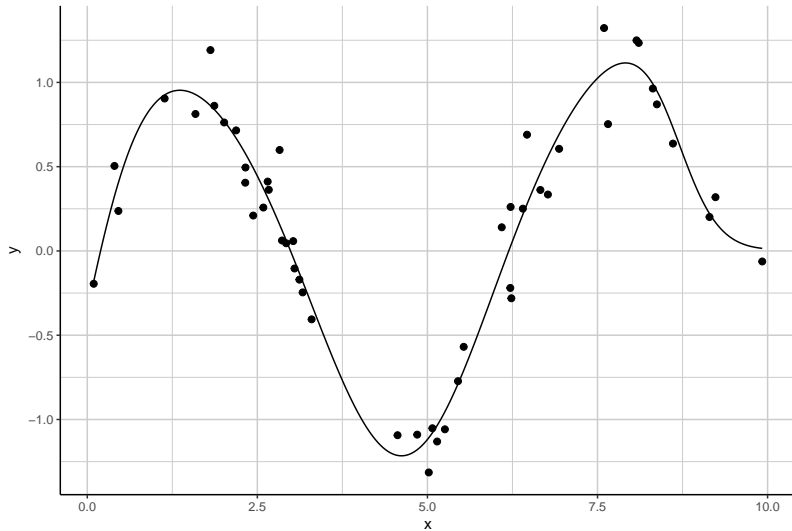
Train: mse=0.0315; CV: mse.test.mean=0.0628



REGRESSION: 1000 TRAINING ITERATIONS

nnet: size=4; maxit=1e+03

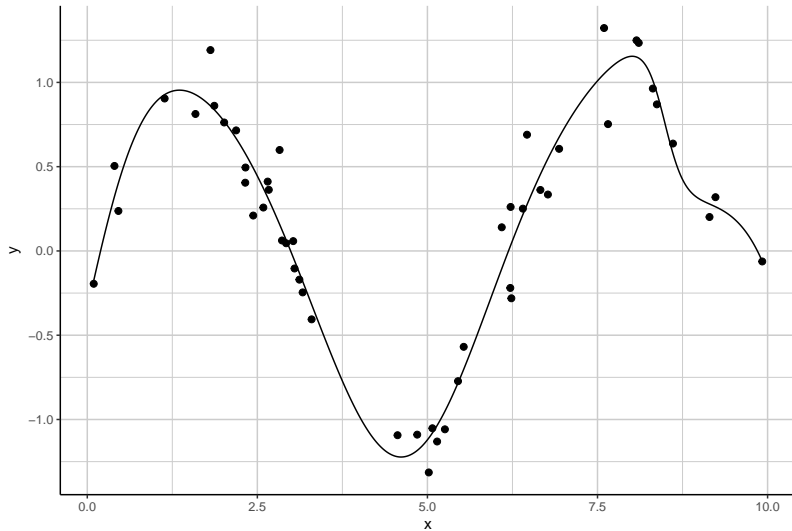
Train: mse=0.0288; CV: mse.test.mean=0.0527



REGRESSION: 1000 TRAINING ITERATIONS

nnet: size=5; maxit=1e+03

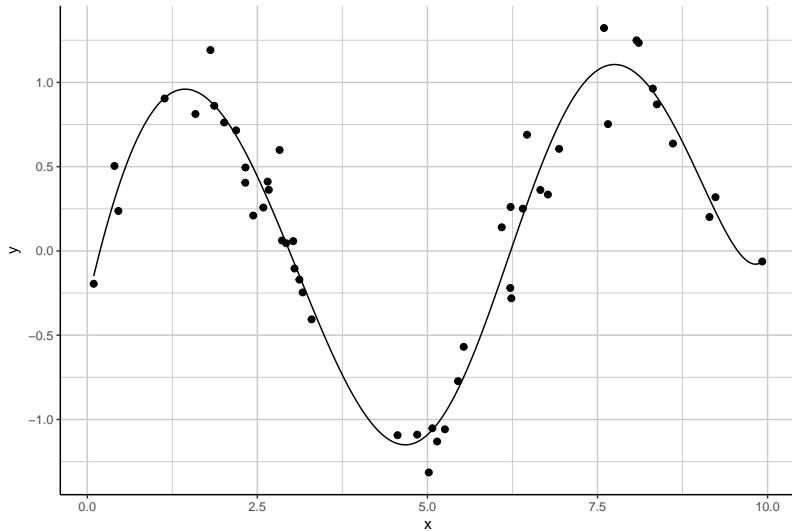
Train: mse=0.0277; CV: mse.test.mean=20.9



REGRESSION: 1000 TRAINING ITERATIONS

nnet: size=6; maxit=1e+03

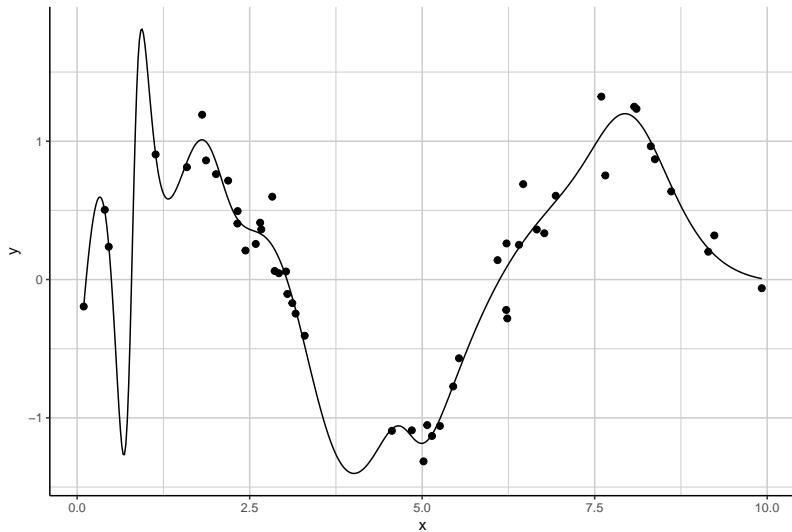
Train: mse=0.0305; CV: mse.test.mean=4.57



REGRESSION: 1000 TRAINING ITERATIONS

nnet: size=10; maxit=1e+03

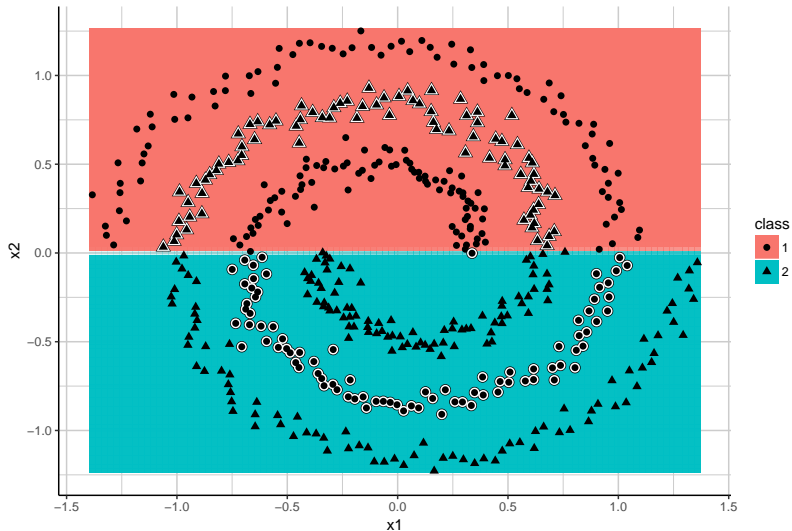
Train: mse=0.0214; CV: mse.test.mean=0.255



CLASSIFICATION: 500 TRAINING ITERATIONS

nnet: size=1; maxit=500

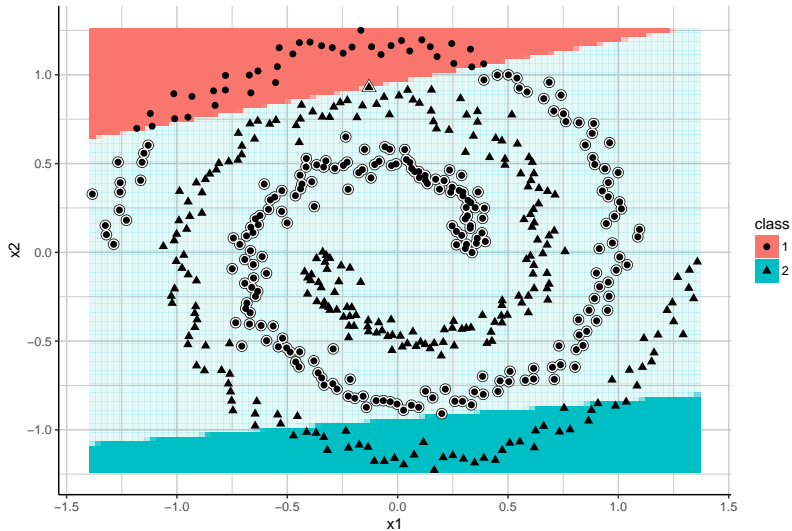
Train: mmce=0.332; CV: mmce.test.mean=0.346



CLASSIFICATION: 500 TRAINING ITERATIONS

nnet: size=2; maxit=500

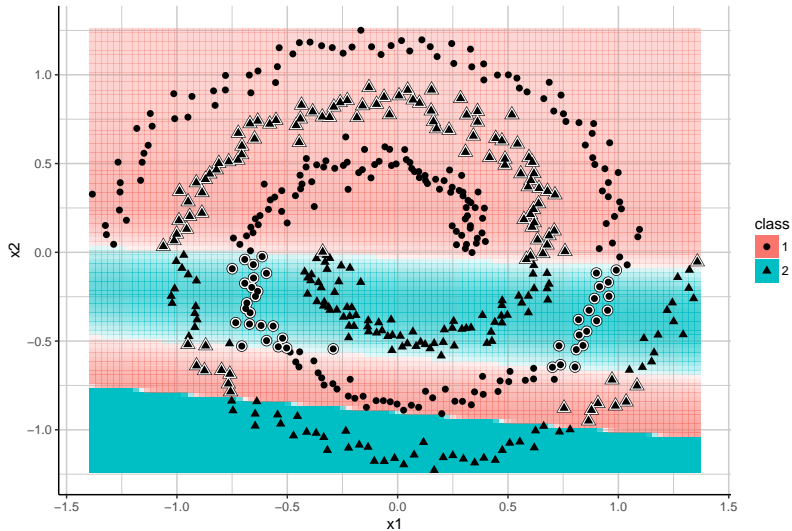
Train: mmce=0.426; CV: mmce.test.mean=0.412



CLASSIFICATION: 500 TRAINING ITERATIONS

nnet: size=3; maxit=500

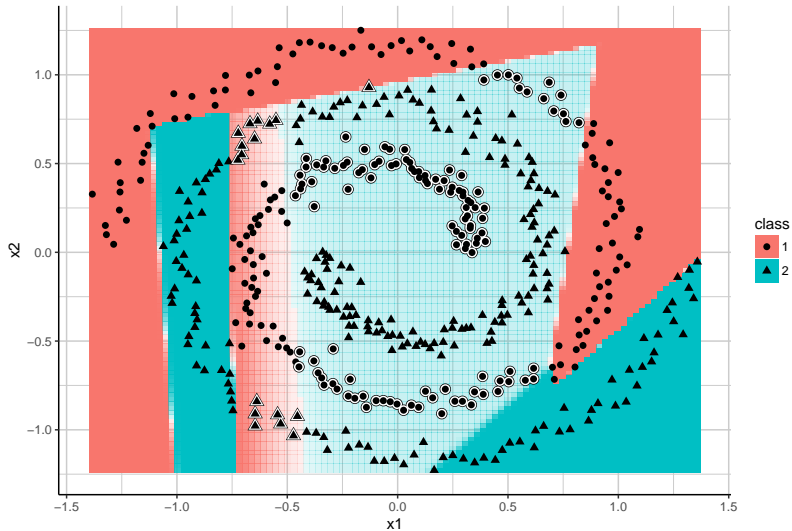
Train: mmce=0.29; CV: mmce.test.mean=0.37



CLASSIFICATION: 500 TRAINING ITERATIONS

nnet: size=5; maxit=500

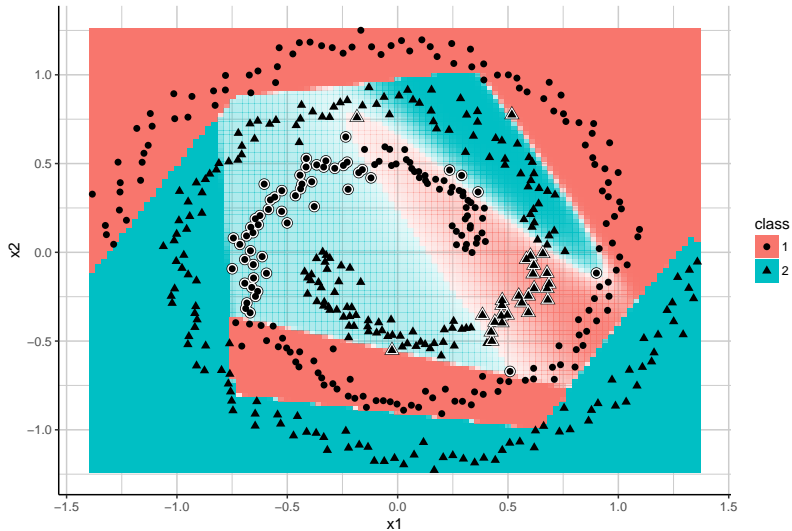
Train: mmce=0.272; CV: mmce.test.mean=0.322



CLASSIFICATION: 500 TRAINING ITERATIONS

nnet: size=10; maxit=500

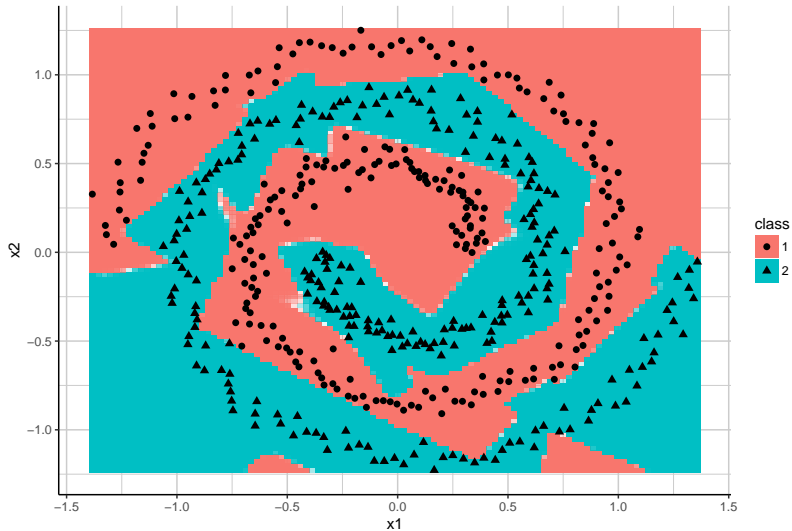
Train: mmce=0.158; CV: mmce.test.mean=0.112



CLASSIFICATION: 500 TRAINING ITERATIONS

nnet: size=30; maxit=500

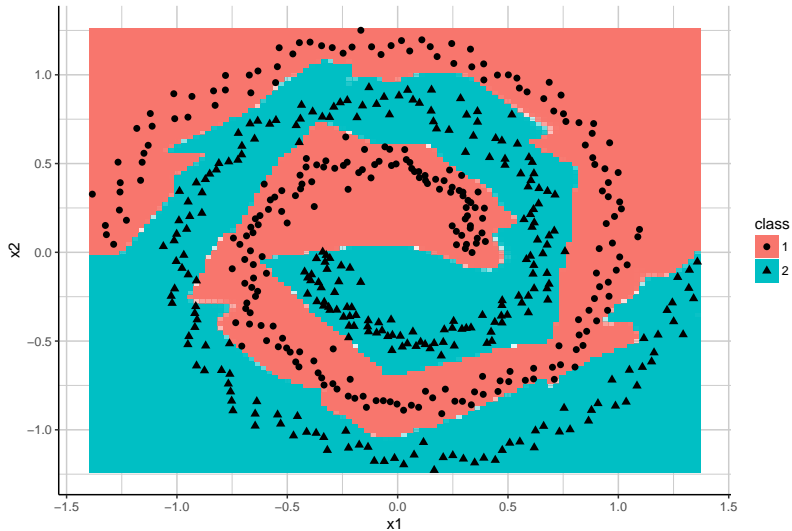
Train: mmce= 0; CV: mmce.test.mean=0.032



CLASSIFICATION: 500 TRAINING ITERATIONS

nnet: size=50; maxit=500

Train: mmce= 0; CV: mmce.test.mean=0.024



SUMMARY

- We have seen that neural networks are far more flexible than linear models. Furthermore, neural networks are able to approximate any continuous function.
 - Yet, in reality, there is no way to make full use of the universal approximation property. The learning algorithm will usually not find the best possible model. At best it finds a locally optimal model.
- The XOR example showed us how neural networks extract features to transform the space and actually learn a kernel (learn a representation).
- Neural networks can perfectly fit noisy data. Thus, neural networks are endangered to over-fit. This is particularly true for a model with a huge hidden layer.
- Fitting neural networks with sigmoidal activation function is nothing else but fitting many weighted logistic regressions!

DEEP FEEDFORWARD NETWORKS

- We will now extend the model class once again, such that we allow an arbitrary amount of k layers.
 - For more than one hidden layer, we call such graphs deep feedforward networks.
- We can characterize those models by the following chain structure:

$$f(x) = g(f_{(k)}(f_{(k-1)}(f_{(k-2)}(\dots(f_{(1)}(x))\dots)))$$

where $f_{(1)}$ corresponds to the first and $f_{(k)}$ to the last layer of the network.

DEEP FEEDFORWARD NETWORKS

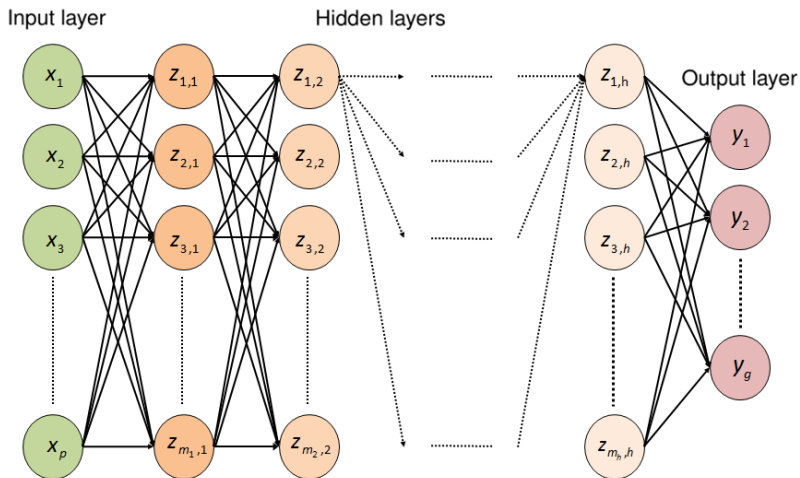


Figure: Structure of a deep neural network with k layers (bias term omitted).

DEEP FEEDFORWARD NETWORKS

- Mathematically, we observe the following mappings:

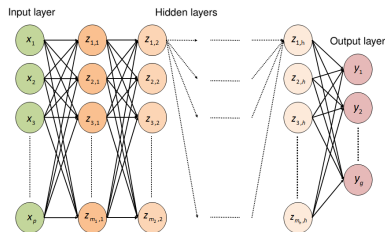
$$f_1(x) : \mathbb{R}^P \rightarrow \mathbb{R}^{M_1}$$

$$f_2(f_1) : \mathbb{R}^{M_1} \rightarrow \mathbb{R}^{M_2}$$

...

$$f_H(..) : \mathbb{R}^{M_{H-1}} \rightarrow \mathbb{R}^{M_H}, \forall h = 2, \dots, H$$

$$g(..) : \mathbb{R}^{M_H} \rightarrow \mathbb{R}^K$$



WHY ADD MORE LAYERS?

- Each layer in a feed-forward neural network adds its own degree of non-linearity to the model.

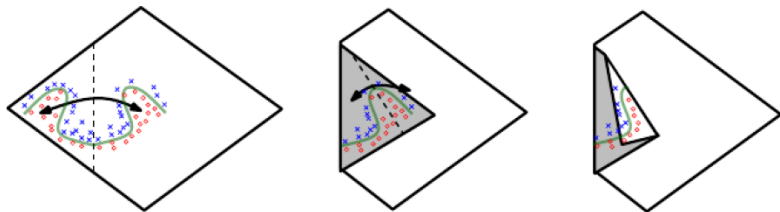


Figure: An intuitive, geometric explanation of the exponential advantage of deeper networks formally (Montúfar et al. (2014))

WHY NOW AND NOT EARLIER?

- Better algorithms (optimization chapter)
 - Vanishing gradient problem (relu)
- Better regularization (regularization chapter)
- Unsupervised pretraining (autoencoder chapter)
- More layers inevitably lead to a significant increase of parameters.
 - Back then, processing power was simply not capable to handle such huge amounts of parameters.
⇒ Nowadays, deep neural networks are trained on GPUs (graphic processing units), not on CPUs (central processing units).

WEIGHT DECAY

- In terms of neural networks, weight decay is the common terminology for L2 penalization.
- Applying weight decay means nothing but regularizing the network to reduce overfitting.

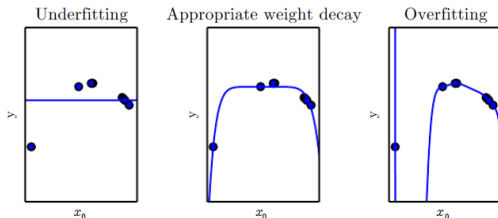


Figure: The effect of weight decay when applied to a simple regression model (Goodfellow et al. (2016)). We will examine weight decay in greater detail in the regularization chapter.

INTRODUCTION TO MXNET

- Open-source deep learning framework written in C++ and cuda (used by Amazon for their Amazon Web Services)
- Scalable, allowing fast model training
- Supports flexible model programming and multiple languages (C++, Python, Julia, Matlab, JavaScript, Go, **R**, Scala, Perl)
- Installation instructions for different operating systems:
http://mxnet.io/get_started/install.html

KAGGLE CHALLENGE DIGIT RECOGNIZER

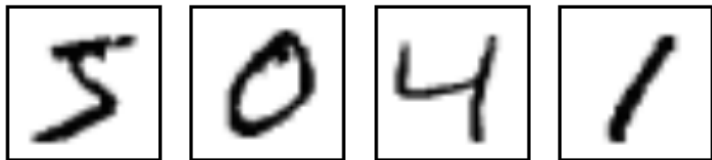


Figure: Snipped from the mnist data set (LeCun and Cortes (2010)).

- 70k image data of handwritten digits with 28×28 pixels.
- Classification task with 10 classes.
- One of the most popular benchmark data sets.

KAGGLE CHALLENGE DIGIT RECOGNIZER

- Since competing with others is more fun, we dare ourselves to face the mnist kaggle challenge.
- Therefore, we download the data sets (train.csv and test.csv) from <https://www.kaggle.com/c/digit-recognizer/data>.
- We obtain 42.000 images for training and 28.000 for testing.

```
# assign the location of the data as your wd()  
  
train = read.csv("train.csv", header = TRUE)  
test = read.csv("test.csv", header = TRUE)  
  
train = data.matrix(train)  
test = data.matrix(test)
```

KAGGLE CHALLENGE DIGIT RECOGNIZER

```
# Split data into matrix containing features and vector with labels
train.x = train[,-1]
train.y = train[,1]

# normalize to (0,1) and transpose data
train.x = t(train.x/255)
dim(train.x)

## [1]    784 42000

test = t(test/255)

table(train.y)

## train.y
##      0      1      2      3      4      5      6      7      8      9
## 4132 4684 4177 4351 4072 3795 4137 4401 4063 4188
```

KAGGLE CHALLENGE DIGIT RECOGNIZER

- Now we define the architecture of our model.

```
require("mxnet")

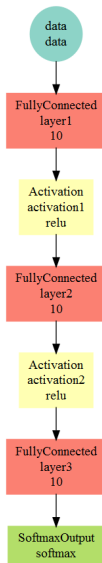
data = mx.symbol.Variable(name = "data")

layer1 = mx.symbol.FullyConnected(data = data, name = "layer1",
    num_hidden = 10L)
activation1 = mx.symbol.Activation(data = layer1, name = "activation1",
    act_type = "relu")
layer2 = mx.symbol.FullyConnected(data = activation1, name = "layer2",
    num_hidden = 10L)
activation2 = mx.symbol.Activation(data = layer2, name = "activation2",
    act_type = "relu")
layer3 = mx.symbol.FullyConnected(data = activation2, name = "layer3",
    num_hidden = 10L)
softmax = mx.symbol.SoftmaxOutput(data = layer3, name = "softmax")
```

KAGGLE CHALLENGE DIGIT RECOGNIZER

- Mxnet enables us to easily visualize the models architecture

```
graph.viz(model$symbol)
```



KAGGLE CHALLENGE DIGIT RECOGNIZER

- In a final step, we have to assign some parameters.

```
devices = mx.cpu()

mx.set.seed(1337)

model = mx.model.FeedForward.create(
    symbol = softmax,
    X = train.x, y = train.y,
    ctx = devices,
    num.round = 10L, array.batch.size = 100L,
    learning.rate = 0.05,
    eval.metric = mx.metric.accuracy,
    initializer = mx.init.uniform(0.07),
    epoch.end.callback = mx.callback.log.train.metric(100L))
```

KAGGLE CHALLENGE DIGIT RECOGNIZER

```
## Start training with 1 devices
## [1] Train-accuracy=0.65945107398568
## [2] Train-accuracy=0.897738095238094
## [3] Train-accuracy=0.913333333333333
## [4] Train-accuracy=0.920738095238095
## [5] Train-accuracy=0.92504761904762
## [6] Train-accuracy=0.928452380952381
## [7] Train-accuracy=0.930499999999999
## [8] Train-accuracy=0.931571428571428
## [9] Train-accuracy=0.932833333333333
## [10] Train-accuracy=0.935023809523809
```

- After 10 epochs, our neural network begins to stagnate at a training accuracy of roughly 93.5%
- Following up, we use the model to predict the test data.

KAGGLE CHALLENGE DIGIT RECOGNIZER

```
preds = predict(model, test)
# this yields us predicted probabilities for all 10 classes
dim(preds)

## [1]      10 28000

# we choose the maximum to obtain quantities for each class
pred.label = max.col(t(preds)) - 1
table(pred.label)

## pred.label
##      0      1      2      3      4      5      6      7      8      9
## 2666 3329 2591 2818 2842 2110 2845 2930 2947 2922
```

KAGGLE CHALLENGE DIGIT RECOGNIZER

- Finally we want to submit our predictions on kaggle to see how good we performed.
- Thus, we save our results in a csv file and upload it on <https://www.kaggle.com/c/digit-recognizer/submit>

```
submission = data.frame(ImageId = 1:ncol(test), Label = pred.label)

write.csv(submission, file = 'submission.csv', row.names = FALSE,
          quote = FALSE)
```

KAGGLE CHALLENGE DIGIT RECOGNIZER

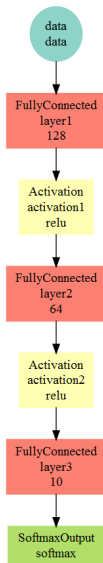
- After making your submission, you should see something like this:

Your most recent submission				
Name	Submitted	Wait time	Execution time	Score
submission.csv	just now	0 seconds	0 seconds	0.89843
Complete				
Jump to your position on the leaderboard ▾				

- For this competition Kaggle uses accuracy (score) to measure each participants performance.
- While the ratio of the train to test data makes the problem really difficult, 89.843% is still a very bad result and we would like to improve our performance.

KAGGLE CHALLENGE DIGIT RECOGNIZER

- Let us try the following, much larger, network (all other parameters remain the same):



KAGGLE CHALLENGE DIGIT RECOGNIZER

Your most recent submission






Name	Submitted	Wait time	Execution time	Score
submission.csv	just now	0 seconds	0 seconds	0.96514

Complete

[Jump to your position on the leaderboard ▾](#)

- Rerunning the training with the new architecture, this model yields us a training accuracy of 99.39% and a test accuracy of 96.514%.

REFERENCES

-  Guido Montúfar, Razvan Pascanu, Kyunghyun Cho and Yoshua Bengio (2014)
On the Number of Linear Regions of Deep Neural Networks
<https://arxiv.org/pdf/1402.1869.pdf>
-  Yann LeCun and Corinna Cortes (2010)
MNIST handwritten digit database
<http://yann.lecun.com/exdb/mnist/>
-  Yann Lecun, Leon Bottou, Genevieve B. Orr and Klaus-Robert Müller (1998)
Efficient BackProp
<http://yann.lecun.com/exdb/publis/pdf/lecun-98b.pdf>
-  Geoffrey Hinton and Ruslan Salakhutdinov (2006)
Reducing the Dimensionality of Data with Neural Networks
<https://www.cs.toronto.edu/%7Ehinton/science.pdf>
-  Ian Goodfellow, Yoshua Bengio and Aaron Courville (2016)
Deep Learning
<http://www.deeplearningbook.org/>