MASTER OF SCIENCE
IN ENGINEERING

Week 3: Learning (cont'd) and Shallow Networks

TSM_DeLearn

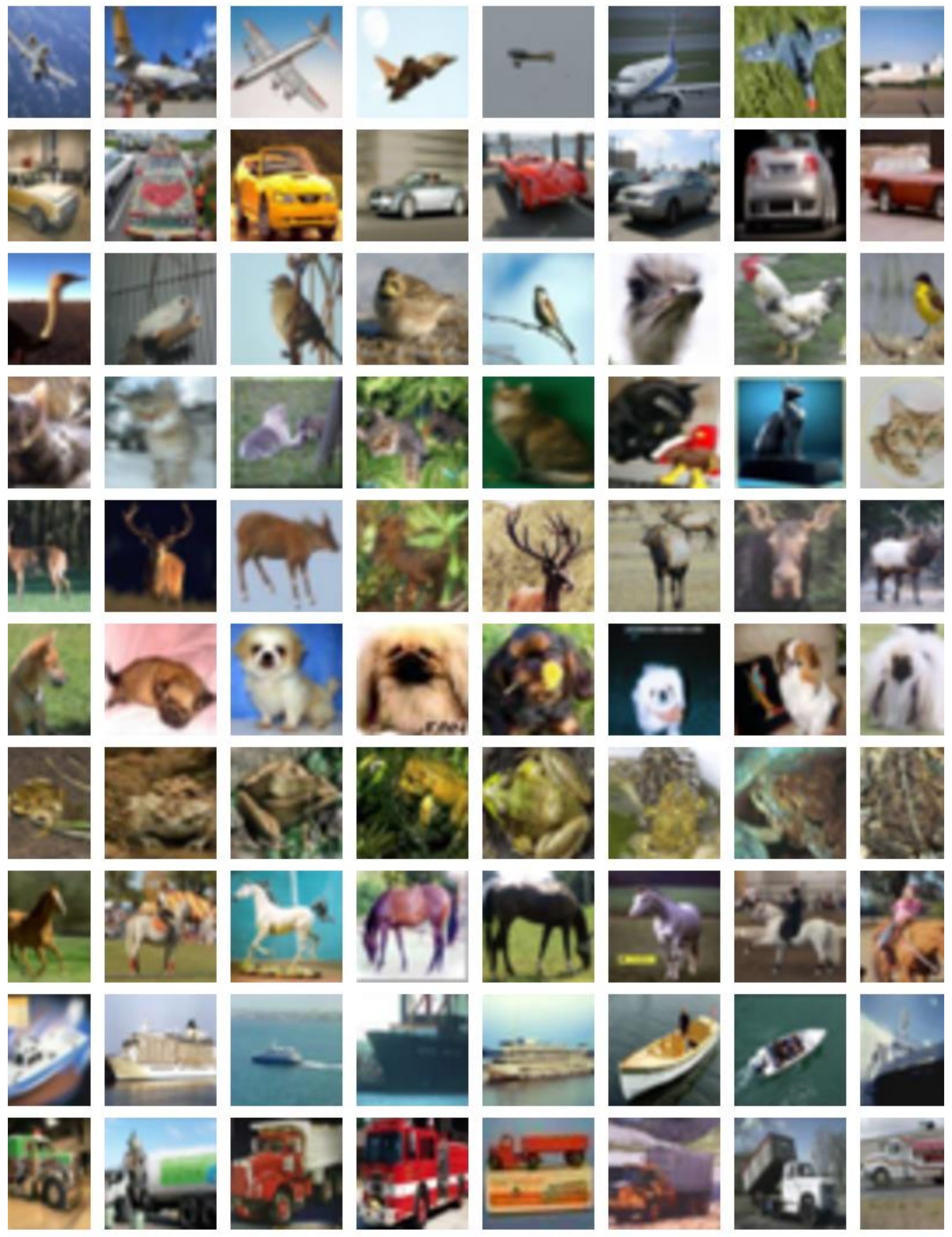
Andreas Fischer
Klaus Zahn

We are grateful to J. Hennebert and M. Melchior, that they provided their slides

Plan for this week

- Learn Softmax activation for general multi-class classification and regression tasks.
- Increase of model capacity to improve MNIST results
- Introduce Bias and Variance of Model
- This will leads us to generalisation error and overfitting.
- Learn the tools how to analyse and optimise them:
  - Hyper-Parameter tuning and model selection
  - Performance measures

Multi-Class Classification and Softmax



Use Multiple Binary Classifiers for Multi-Class?

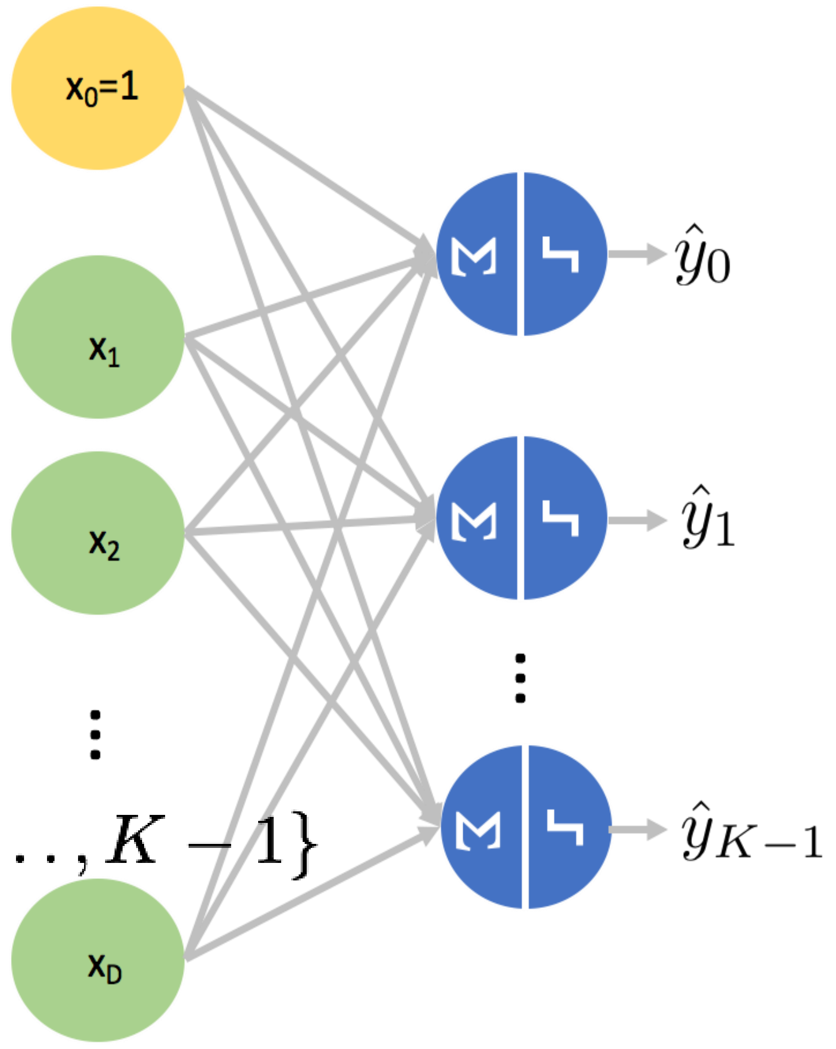
$$0 \leq l < K$$

$$h_{\theta_l}(\mathbf{x}) = \sigma(\mathbf{w}_l \cdot \mathbf{x} + b_l)$$

$$\sigma(z) = \frac{1}{1+e^{-z}}$$

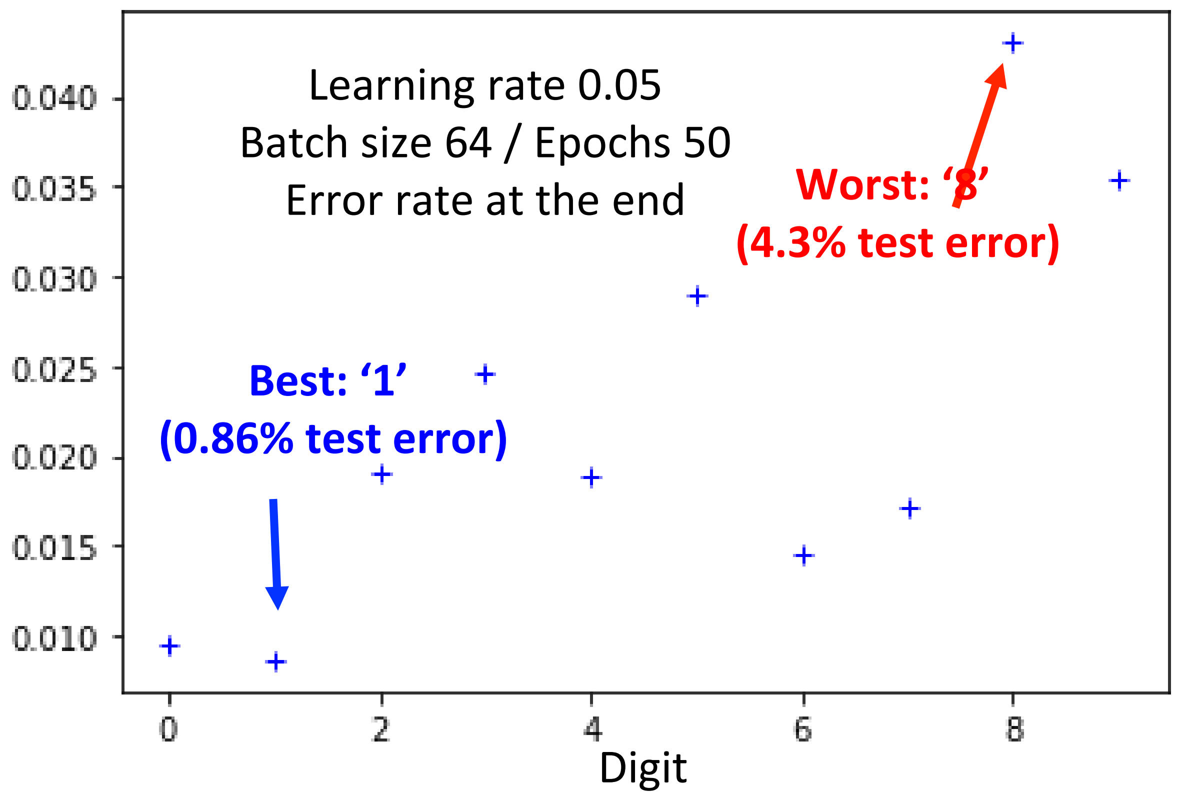
$$\theta_l = (\mathbf{w}_l, b_l)$$

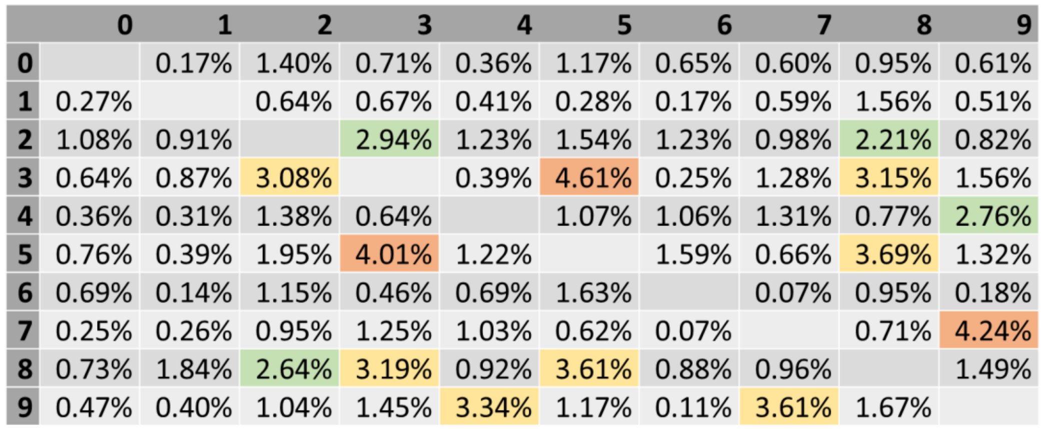
$$\hat{y}(\mathbf{x}) = \underset{l}{argmax}\{h_{\theta_l}(\mathbf{x})\} \in \{0, 1, \ldots, K-1\}$$



Multi-Classification Results for MNIST

Individual test errors rates of binary classification problems





	0	1	2	3	4	5	6	7	8	9
0		0.17%	1.40%	0.71%	0.36%	1.17%	0.65%	0.60%	0.95%	0.61%
1	0.27%		0.64%	0.67%	0.41%	0.28%	0.17%	0.59%	1.56%	0.51%
2	1.08%	0.91%		2.94%	1.23%	1.54%	1.23%	0.98%	2.21%	0.82%
3	0.64%	0.87%	3.08%		0.39%	4.61%	0.25%	1.28%	3.15%	1.56%
4	0.36%	0.31%	1.38%	0.64%		1.07%	1.06%	1.31%	0.77%	2.76%
5	0.76%	0.39%	1.95%	4.01%	1.22%		1.59%	0.66%	3.69%	1.32%
6	0.69%	0.14%	1.15%	0.46%	0.69%	1.63%		0.07%	0.95%	0.18%
7	0.25%	0.26%	0.95%	1.25%	1.03%	0.62%	0.07%		0.71%	4.24%
8	0.73%	1.84%	2.64%	3.19%	0.92%	3.61%	0.88%	0.96%		1.49%
9	0.47%	0.40%	1.04%	1.45%	3.34%	1.17%	0.11%	3.61%	1.67%	

Digits Classification

As result pick the digit with the highest score predicted by the 'binary' classifiers.

Overall Test Error Rate: 9.0%

Digits are confused !
(e.g. 8 with 3 and 5)

Multi-Classification with Softmax

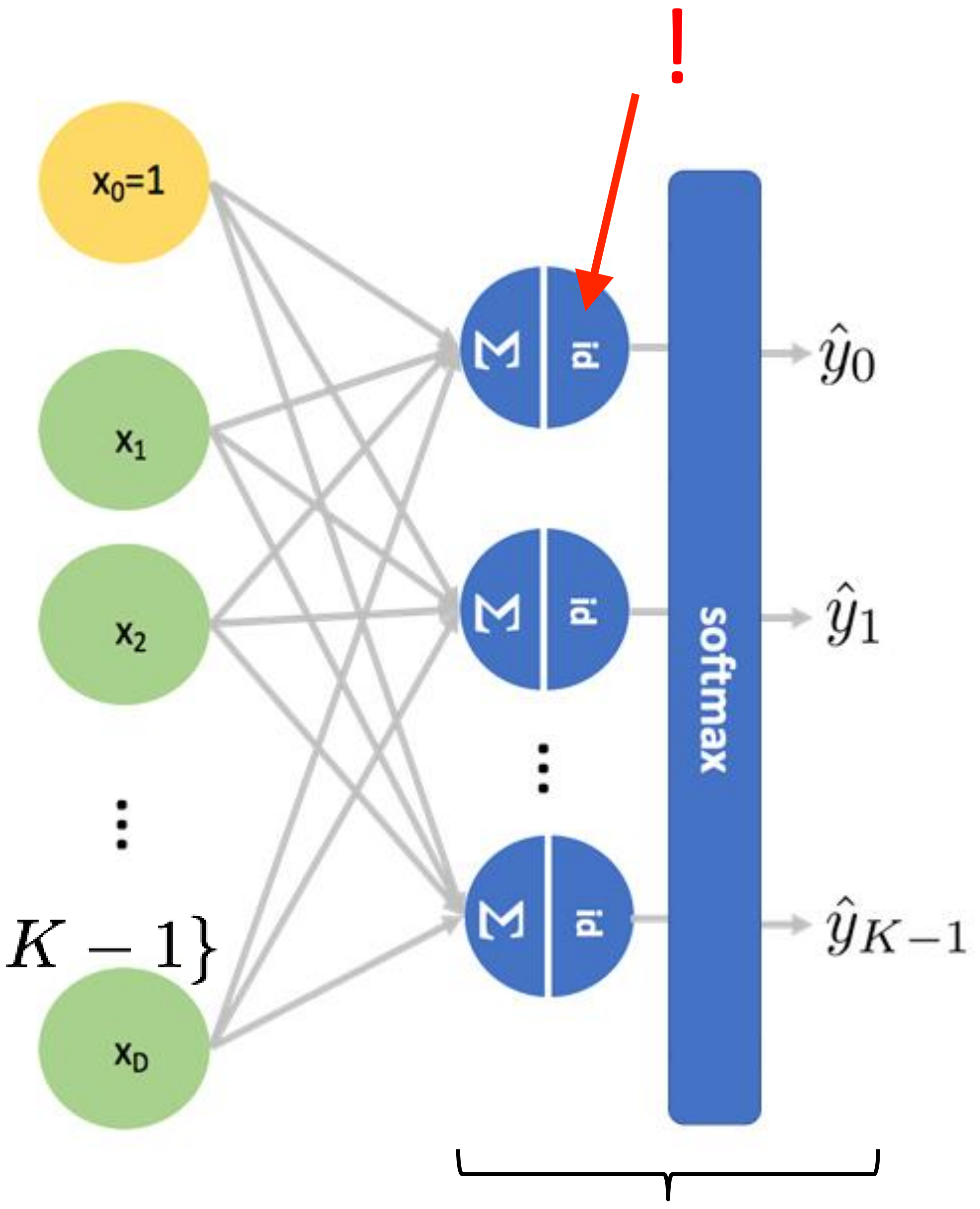
$$0 \leq l < K$$

$$z_l = \mathbf{w}_l \cdot \mathbf{x} + b_l$$

$$h_{\theta,l}(\mathbf{x}) = p_l = \frac{\exp z_l}{\sum_{j=0}^{K-1} \exp z_j}$$

$$\theta_l = (\mathbf{w}_l, b_l)$$

$$\hat{y}(\mathbf{x}) = \underset{l}{argmax}\{h_{\theta_l}(\mathbf{x})\} \in \{0, 1, \ldots, K-1\}$$



softmax layer

Multi-Classification with Softmax

softmax function

$$\boxed{h_{\theta,l}(\mathbf{x}) = \frac{\exp z_l}{\sum_{j=0}^{K-1} \exp z_j}}$$

where $z_l = \mathbf{w}_l \cdot \mathbf{x} + b_l, \;\; 0 \leq l < K$s vector $\mathbf{z} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$

Properties

- Normed: $\sum_{l=0}^{K-1} h_{\theta,l}(\mathbf{x}) = 1$ $\rightarrow$ interpret as conditional probability $p_\theta(y = l|\mathbf{x})$ of finding output $y = l$ given the input x
- Peaks at the largest z_lThe components of softmax smoothly approximate the index of the largest element, i.e.

$$p_l \approx \delta_{l,\mathrm{argmax}\{z_k\}} \text{ if } z_l \gg z_k (\forall k \neq l)$$

Cost Function with Softmax

- Predicted class:

$$\hat{y}(\mathbf{x}) = \underset{l}{\operatorname{argmax}} \{h_{\theta,l}(\mathbf{x})\} \in \{1,\ldots,K\}$$

- Cost function: Cross entropy in accordance with the maximum likelihood principle

$$\boxed{J_{CE}(\boldsymbol{\theta}) = -\frac{1}{m}\sum_{i=1}^{m} \log\, p_\theta\left(y^{(i)}\middle|\mathbf{x}^{(i)}\right) = -\frac{1}{m}\sum_{i=1}^{m} \log\, h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)}$$

- require $\frac{\partial}{\partial\theta_l} J_{CE}(\boldsymbol{\theta})$ for all parameters $\boldsymbol{\theta}_l = (\mathbf{w}_l, b_l)\ (0 \le l < K)$

Calculation of the Gradient

$$\frac{\partial}{\partial \mathbf{w}_l} \log\, h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right) = \frac{\partial}{\partial \mathbf{w}_l} \log \left[\frac{\exp z_{y^{(i)}}}{\sum_{j=0}^{K-1} \exp z_j} \right] =$$

Calculation of the Gradient

$$\frac{\partial}{\partial \mathbf{w}_l} \log\ h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right) = \frac{\partial}{\partial \mathbf{w}_l} \log\left[\frac{\exp z_{y^{(i)}}}{\sum_{j=0}^{K-1} \exp z_j}\right] =$$

$$\frac{\partial}{\partial \mathbf{w}_l}\left[z_{y^{(i)}} - \log \sum_{j=0}^{K-1} \exp z_j\right] =^{(1)} \left[\delta_{l,y^{(i)}} - \frac{\exp z_l}{\sum_{j=0}^{K-1} \exp z_j}\right] \cdot \mathbf{x}^{(i)} =$$

$$\left(\delta_{l,y^{(i)}} - h_{\theta,l}\left(\mathbf{x}^{(i)}\right)\right) \cdot \mathbf{x}^{(i)}$$

in step $=^{(1)}$ we used:

- derivative of logit $\quad \frac{\partial}{\partial \mathbf{w}_j} z_j = \frac{\partial}{\partial \mathbf{w}_j}\left[\mathbf{w}_j \cdot \mathbf{x}^{(i)} + b_j\right] = \mathbf{x}^{(i)}$

- Kronecker delta $\quad \frac{\partial}{\partial \mathbf{w}_l} z_{y^{(i)}} = \delta_{l,y^{(i)}}$

Training with Softmax

Gradient with respect to the weights and biases of the Softmax layer:

$$\nabla_{\mathbf{w}_l} J_{CE}(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{l,y^{(i)}} - h_{\theta,l}\left(\mathbf{x}^{(i)}\right)\right)\cdot \mathbf{x}^{(i)}$$

$$\nabla_{b_l} J_{CE}(\mathbf{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{l,y^{(i)}} - h_{\theta,l}\left(\mathbf{x}^{(i)}\right)\right)$$

Assume that the labels l can assume values 0,1,…,K-1

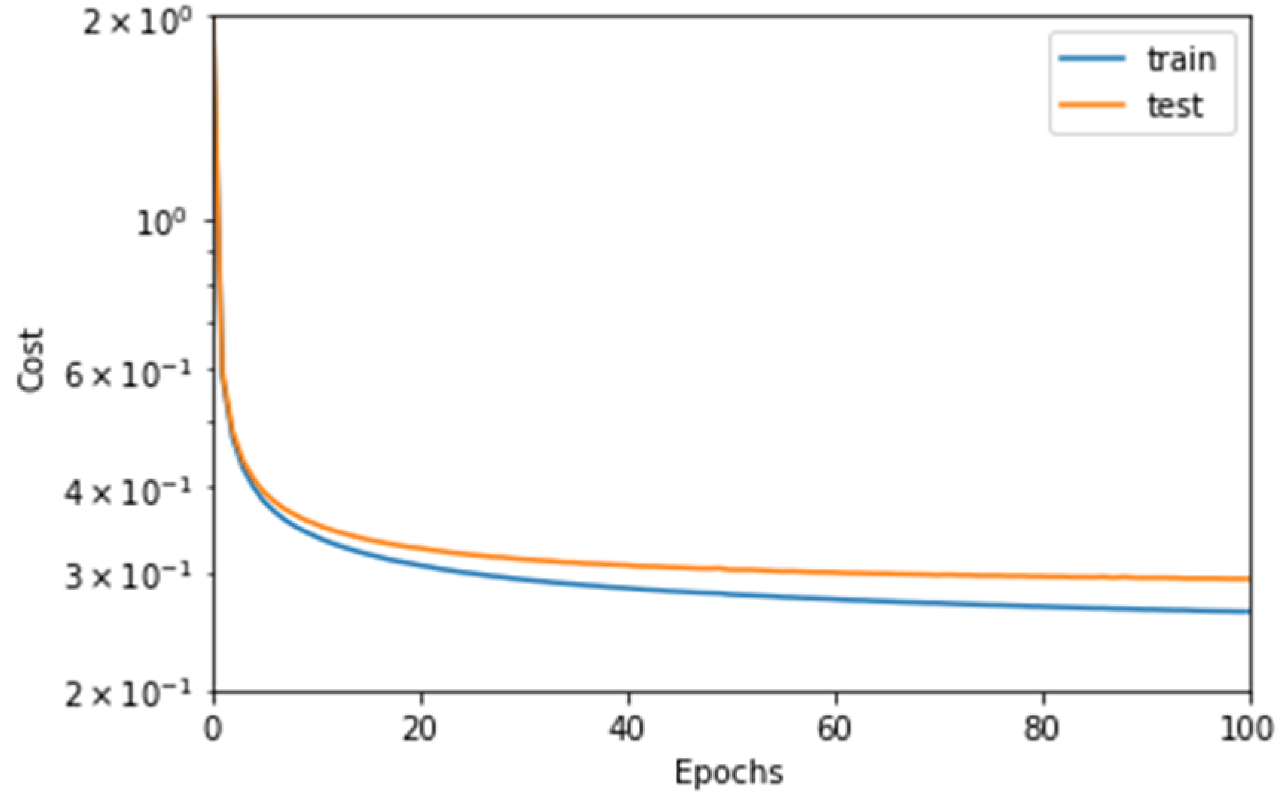
Update rules for the weights and biases of the softmax layer:

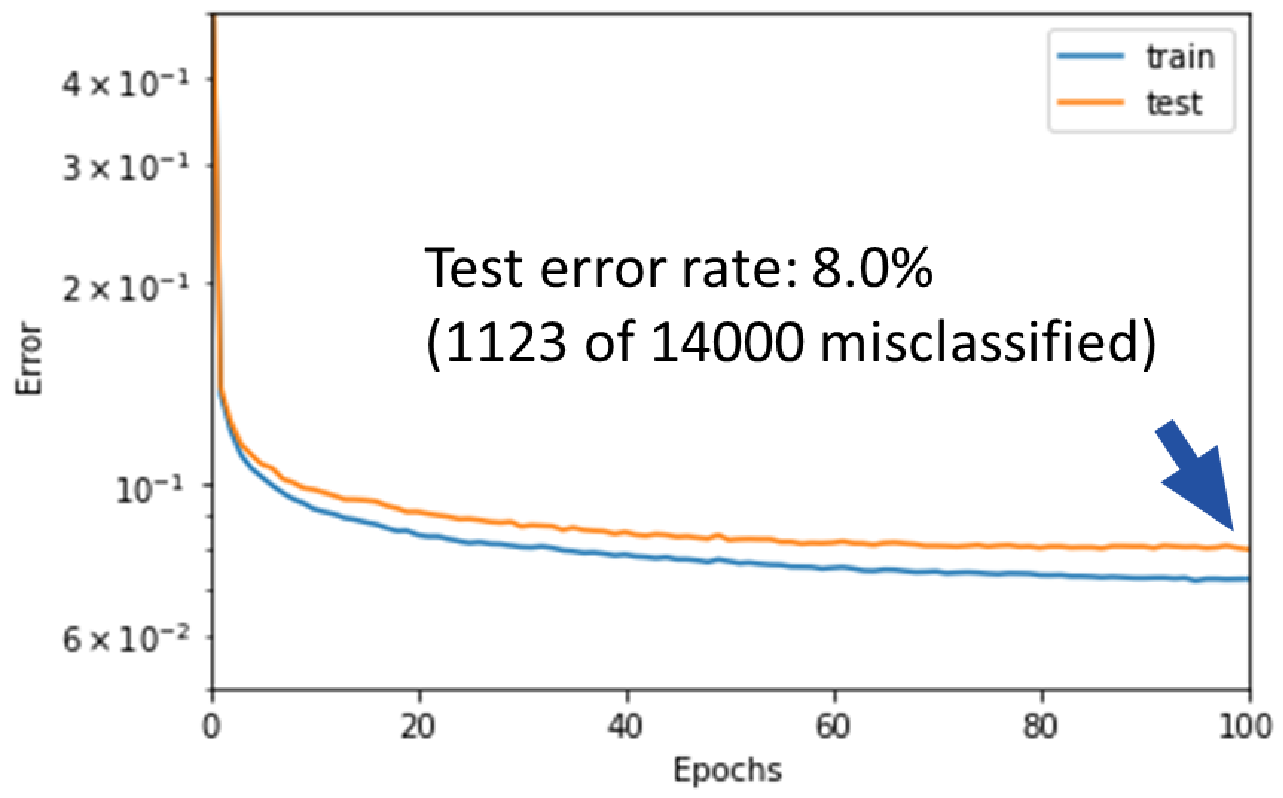
$$\mathbf{w}_l \leftarrow \mathbf{w}_l - \alpha \cdot \nabla_{\mathbf{w}_l} J_{CE}(\mathbf{w}, b)$$

$$b_l \leftarrow b_l - \alpha \cdot \nabla_{b_l} J_{CE}(\mathbf{w}, b)$$

Results for MNIST: Using MBGD

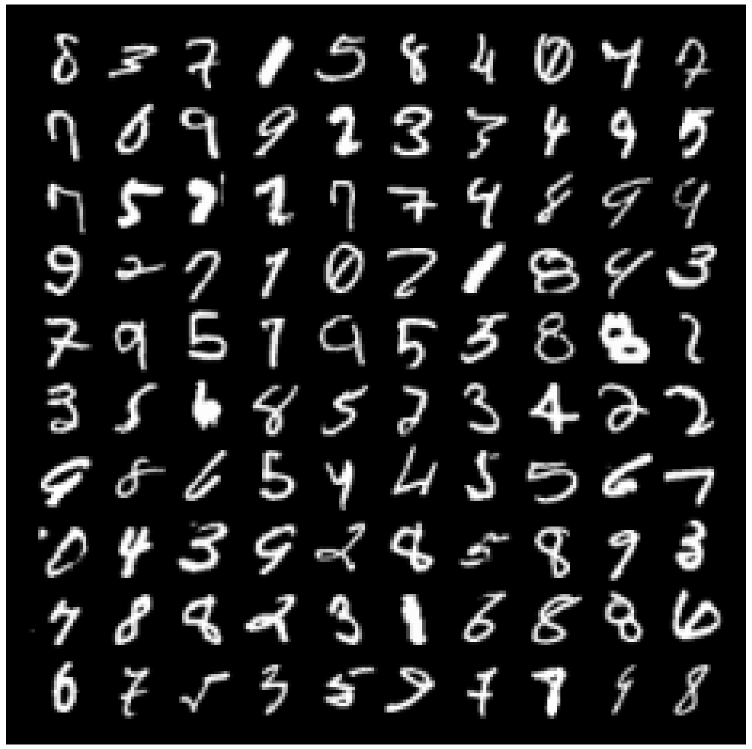
learning rate 0.05, batch size = 256, epochs = 100





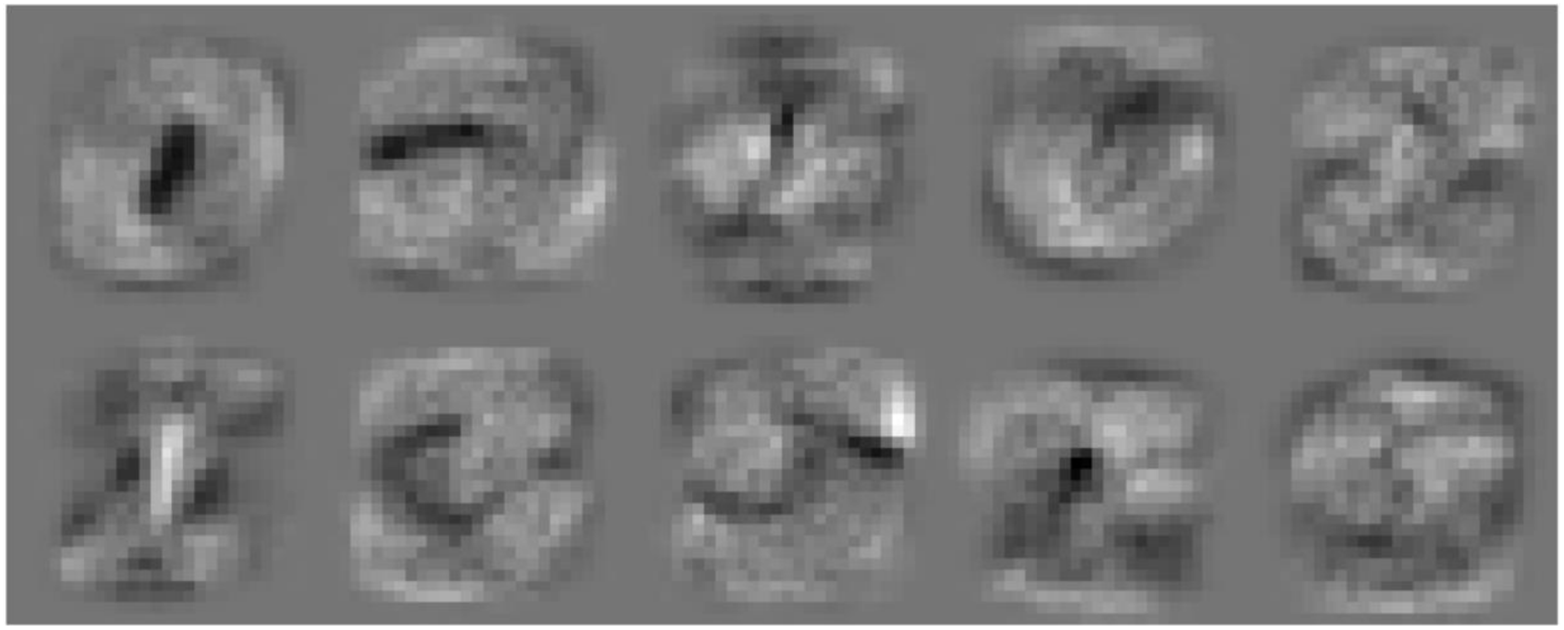
Test error rate: 8.0%
(1123 of 14000 misclassified)

Results for MNIST: Using MBGD



real label	false label
[8 3 7 1 5 8 4 0 7 7]	[2 2 9 8 0 1 9 8 4 9]
[7 0 9 9 2 3 3 4 9 5]	[9 3 3 4 7 8 7 9 4 9]
[7 5 7 2 7 7 4 8 9 9]	[4 8 8 9 9 4 9 1 4 7]
[9 2 7 7 0 2 1 8 4 3]	[8 4 9 1 8 7 8 2 5 0]
[7 9 5 7 9 5 5 8 8 2]	[2 7 6 1 2 8 3 5 2 1]
[3 5 6 8 5 2 3 4 2 2]	[5 1 9 4 3 7 5 6 3 3]
[9 8 6 5 4 4 5 5 6 7]	[5 1 2 6 9 6 6 0 4 9]
[0 4 3 9 2 8 5 8 9 3]	[3 9 5 4 4 2 2 9 7 8]
[7 8 8 2 3 1 6 8 8 6]	[9 3 9 4 5 8 5 5 9 2]
[6 7 5 3 5 9 7 8 9 8]	[8 2 2 5 4 7 1 7 1 1]

The weight vectors (top 0, 2, 4, 6, 8; bottom 1, 3, 5, 7, 9) after training organized as 28x28 patches

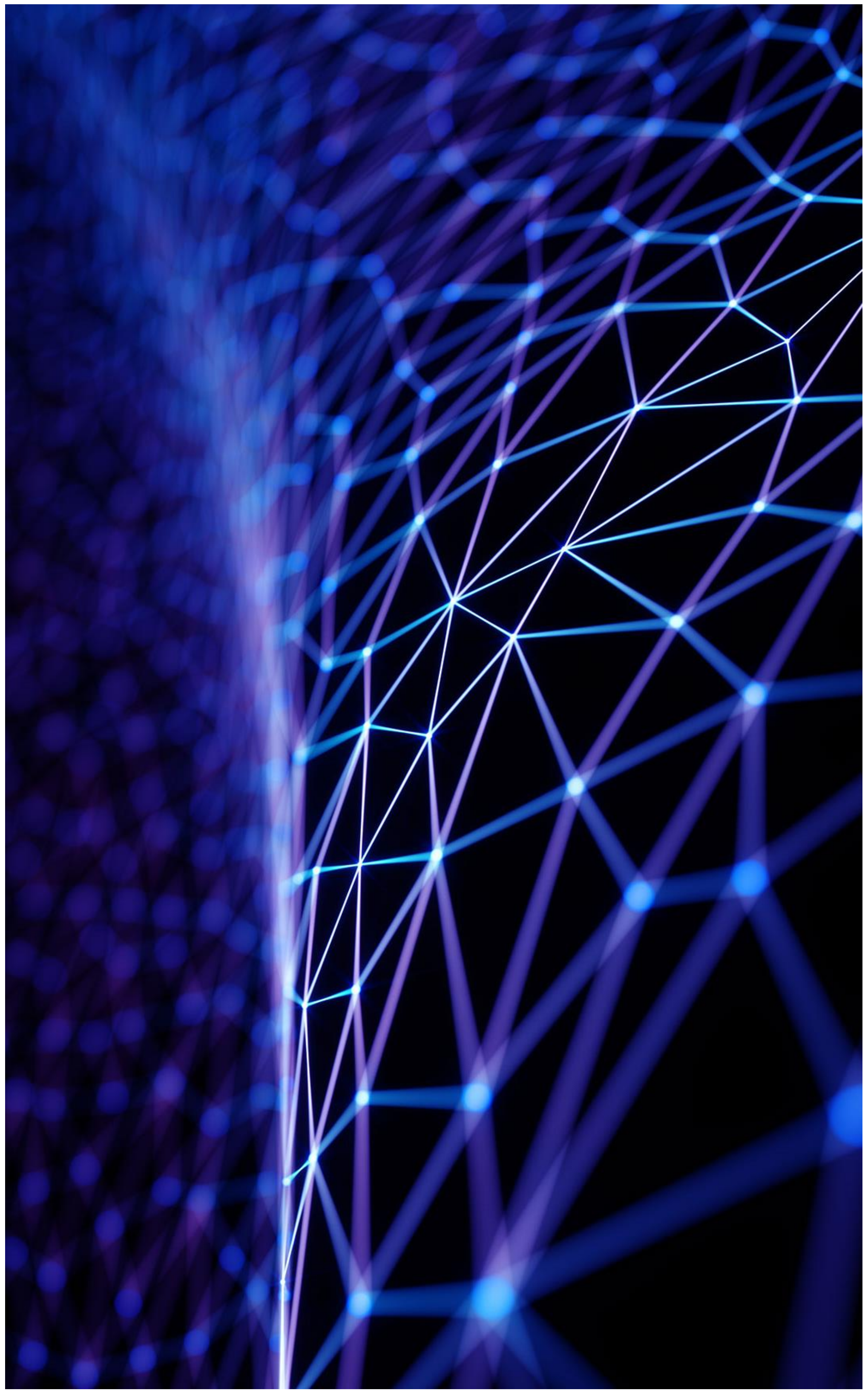


Open Issues

- MNIST (original dataset):
  - **Test Error: ~7.9%**
    (after 5'000 epochs with BGD, learning rate 0.8, test data size 10'000)
- How good can we expect a classifier to be?
  - Best performance (reported on this overview page):
    Test Error : **0.21%** (Li Wan, et al, ICML 2013)
- To achieve this performance, larger models with higher representational capacity and (Many) more model parameters

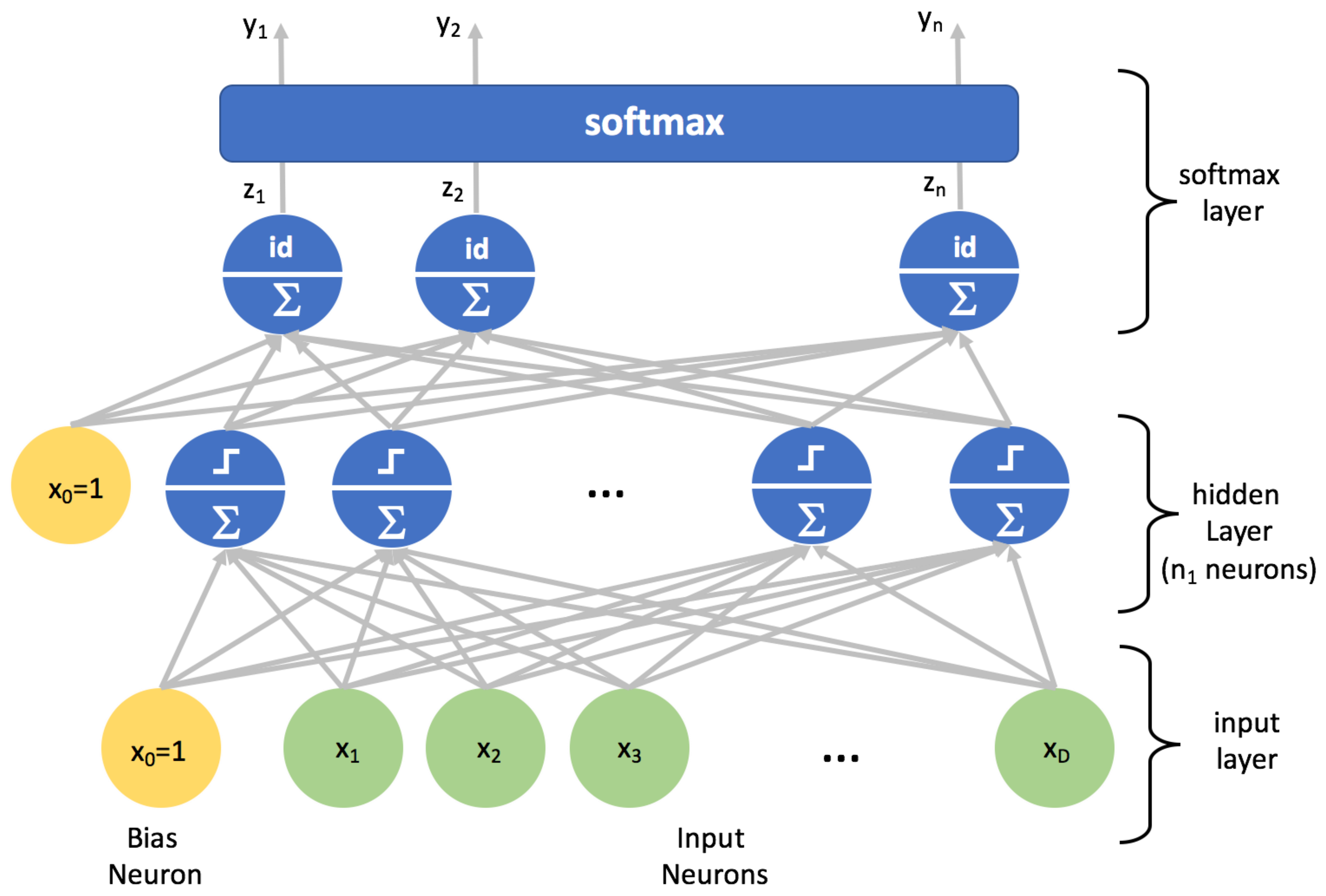
Increasing the Capacity of Models

Adding Hidden Layers
Role of the
Activation Function



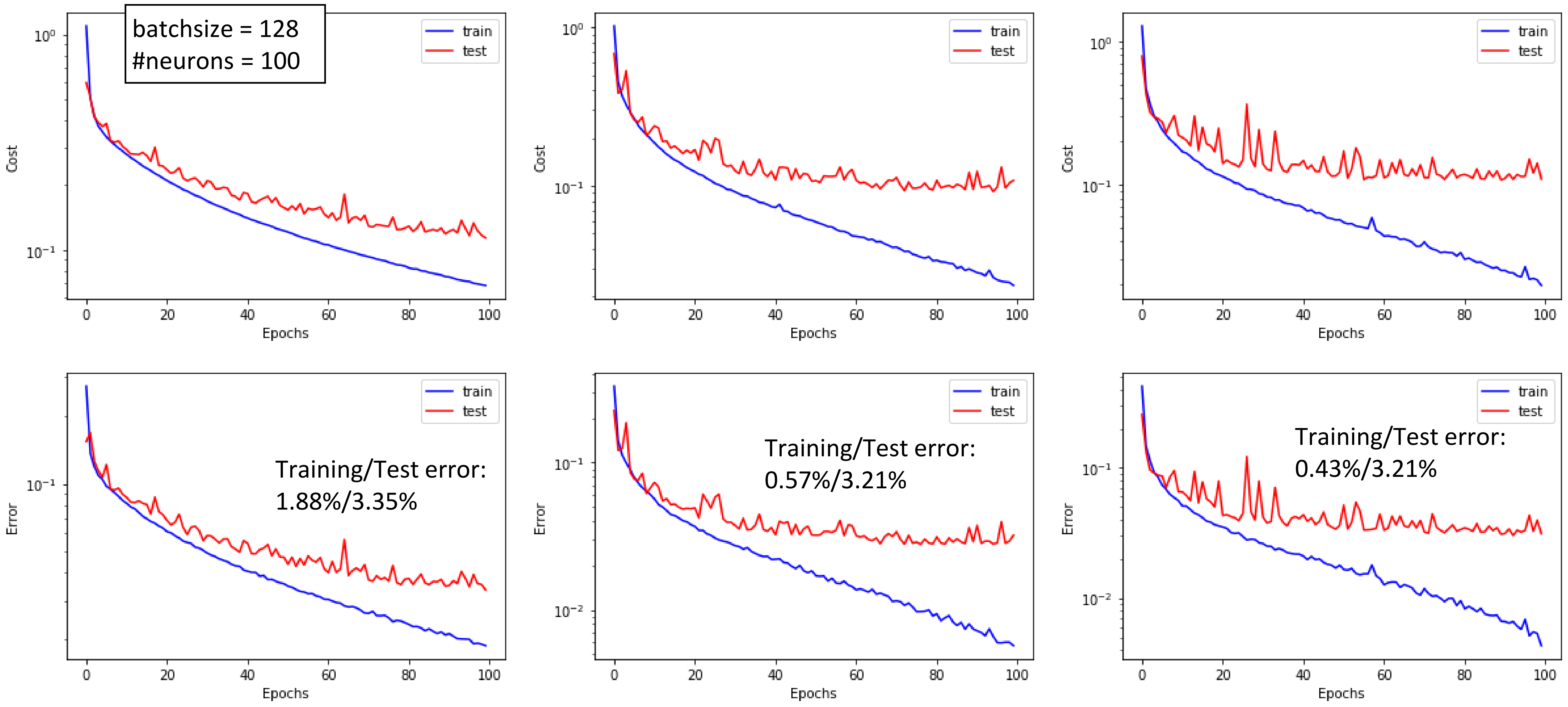
Adding One Hidden Layer

- One additional layer with *n₁* neurons, sigmoid activation function.
- "Fully connected": Weights for each connection between
  - input and hidden layer neurons ●→●
  - hidden layer and softmax neurons ●→●
- Bias for each connection to a bias neuron. ●→●



Results for MNIST - Learning Rate

learning rate 0.1 | learning rate 0.3 | learning rate 0.5

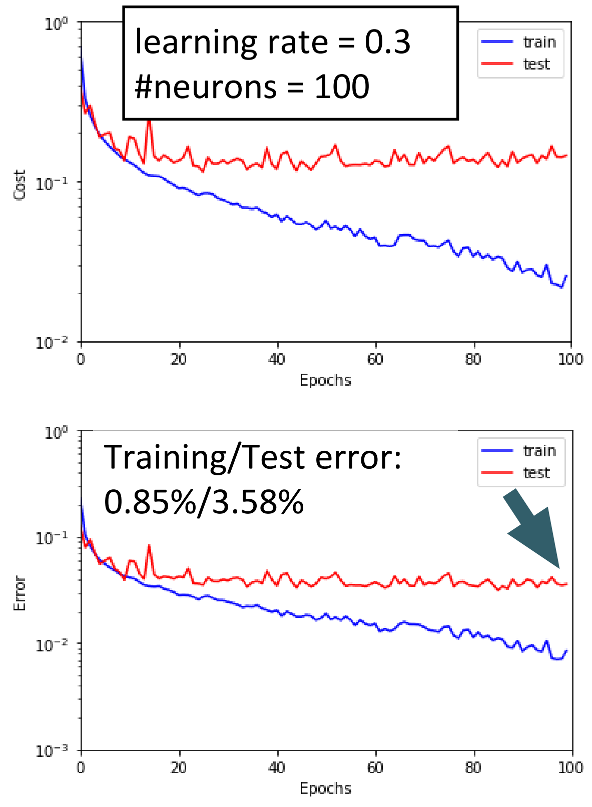


Increasing the learning rate increase the uncertainty in the GD step and thus the fluctuations of the test error rates. The error rates for the smallest learning rate (left) may not yet have reached their final values.

Results obtained with tensorflow/keras

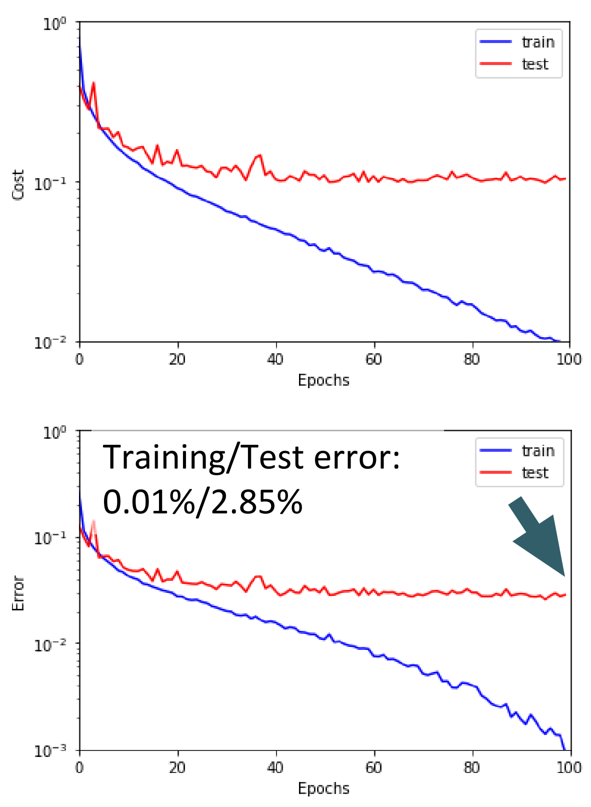
Results for MNIST – Batchsize

batchsize = 32

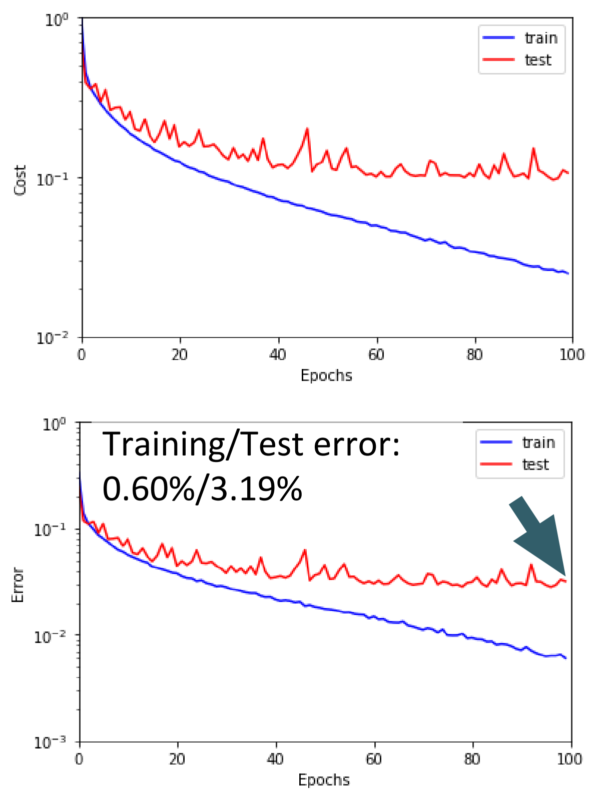

learning rate = 0.3
#neurons = 100

Training/Test error:
0.85%/3.58%

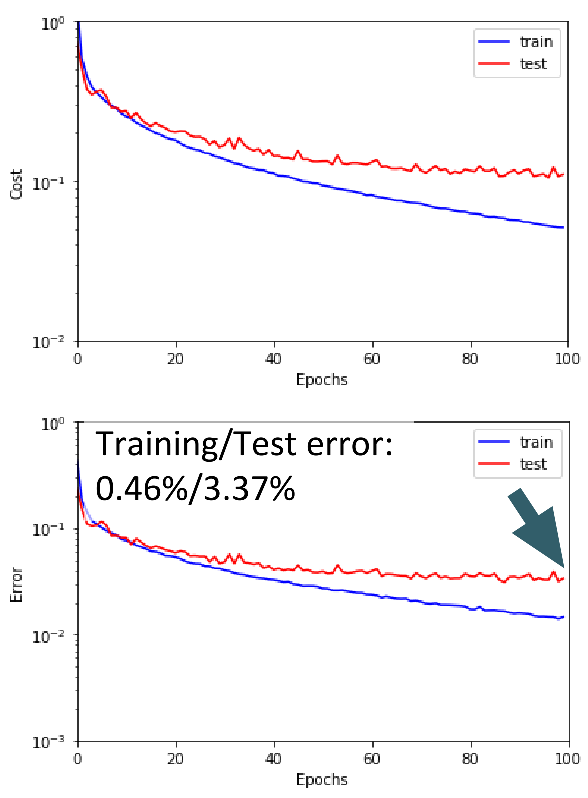
batchsize = 64


Training/Test error:
0.01%/2.85%

batchsize = 128


Training/Test error:
0.60%/3.19%

batchsize = 256

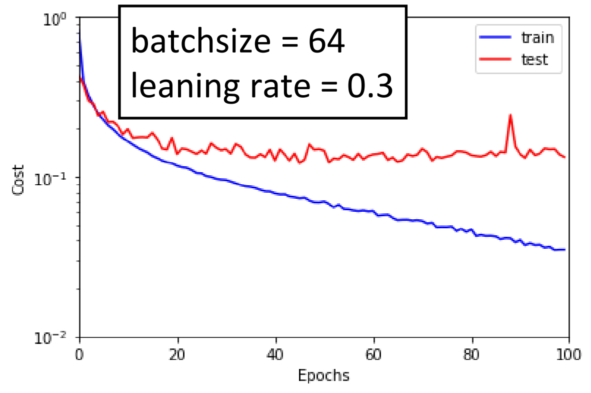

Training/Test error:
0.46%/3.37%

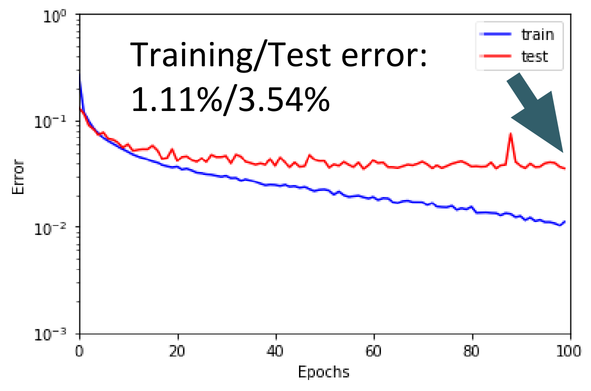
An intermediate batch size of 64 leads to the fasted convergence

Results obtained with tensorflow/keras

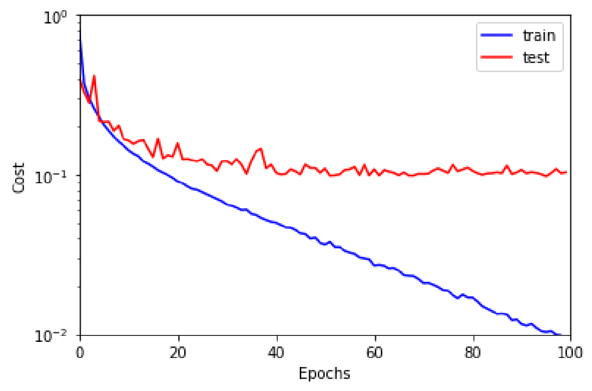
Results for MNIST - Number of Hidden Neurons

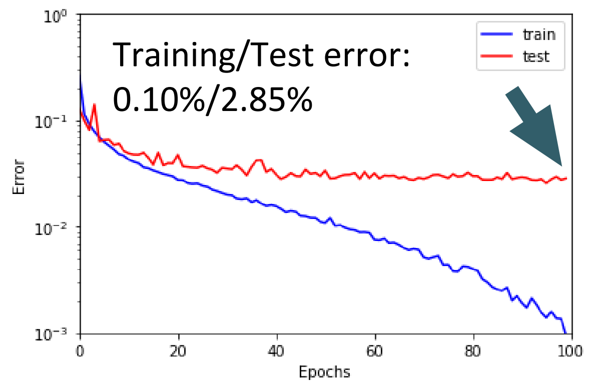
# neurons = 50



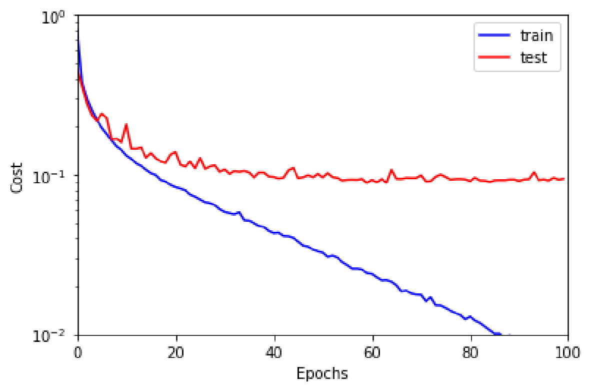


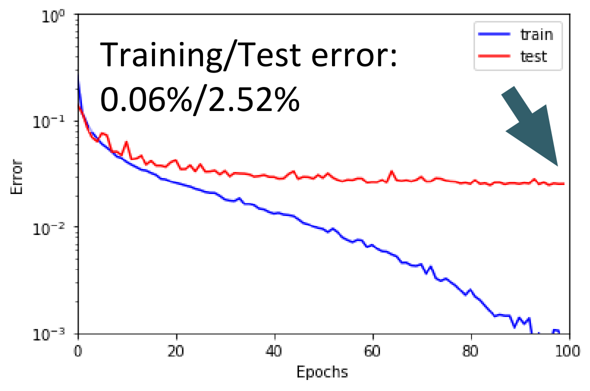
neurons = 100



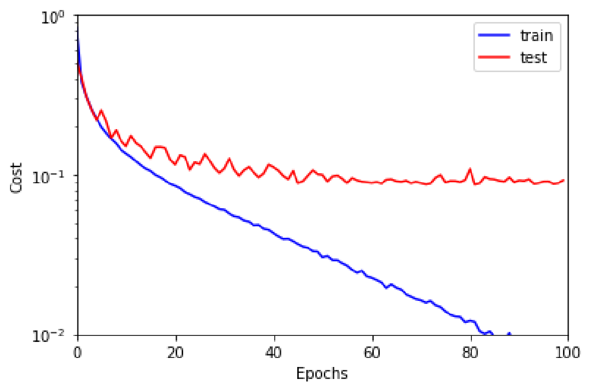


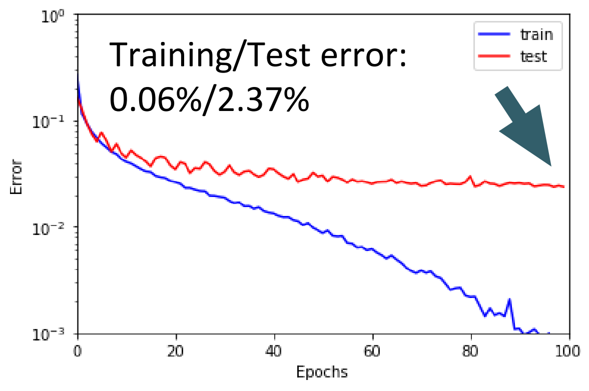
neurons = 150





neurons = 200

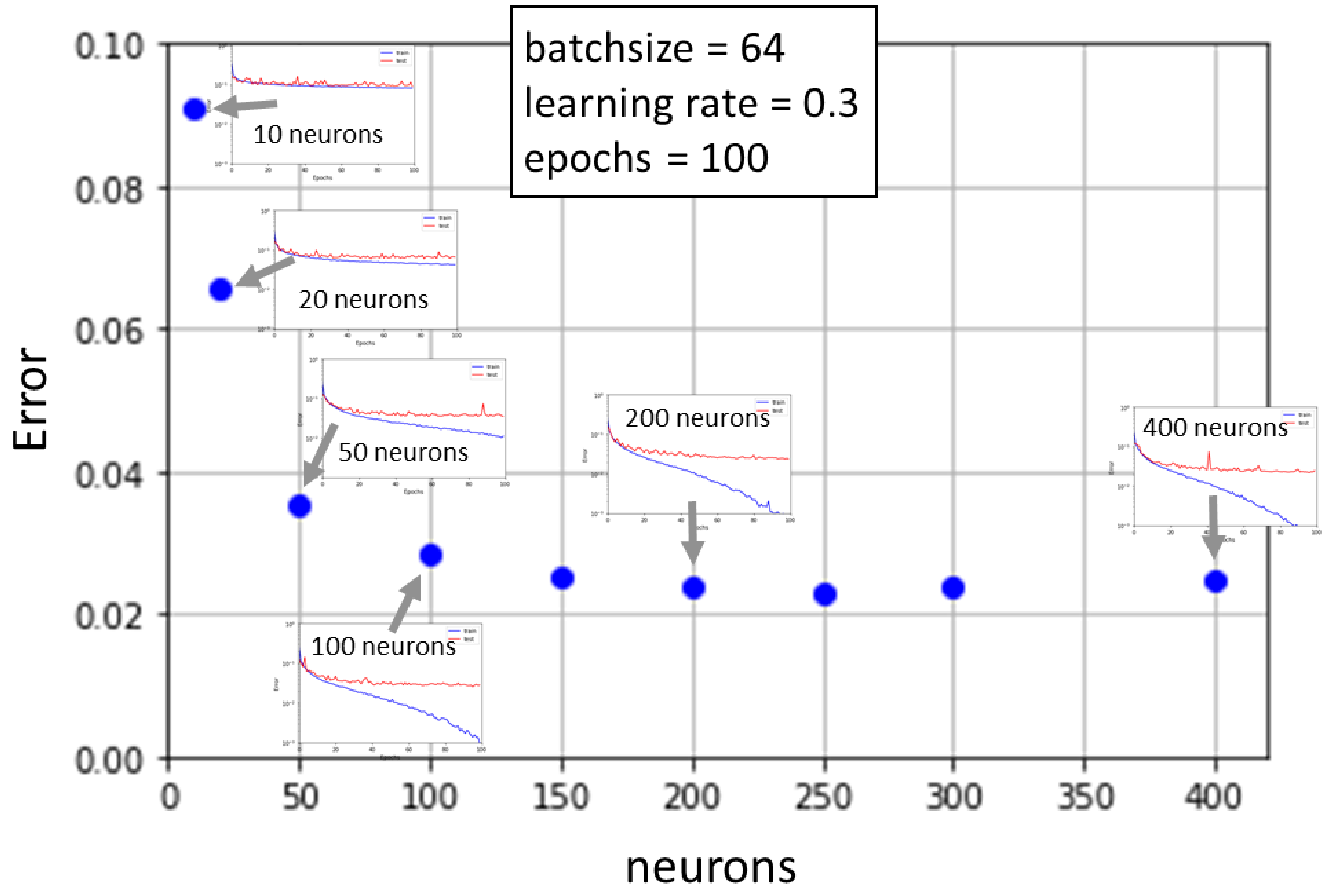




Increasing the number of hidden neurons increases the representational capacity of the network and allows for smaller error rates

Results obtained with tensorflow/keras

Improvements by Adding One Hidden Layer

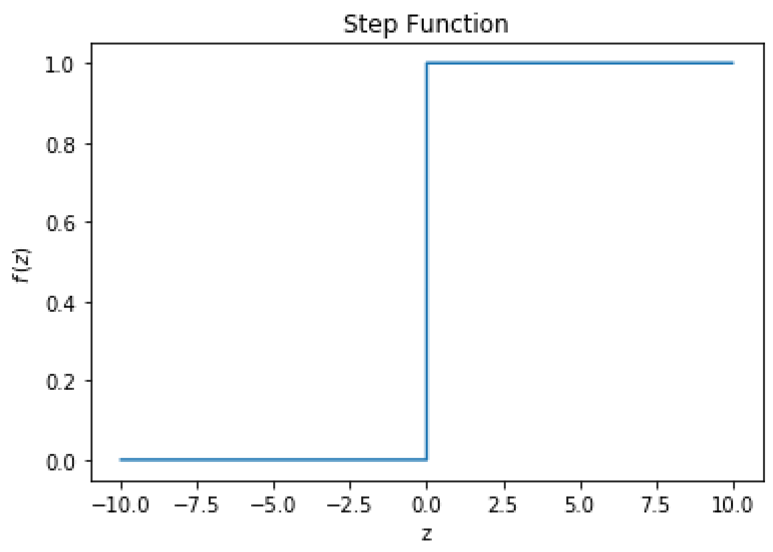


Open Points

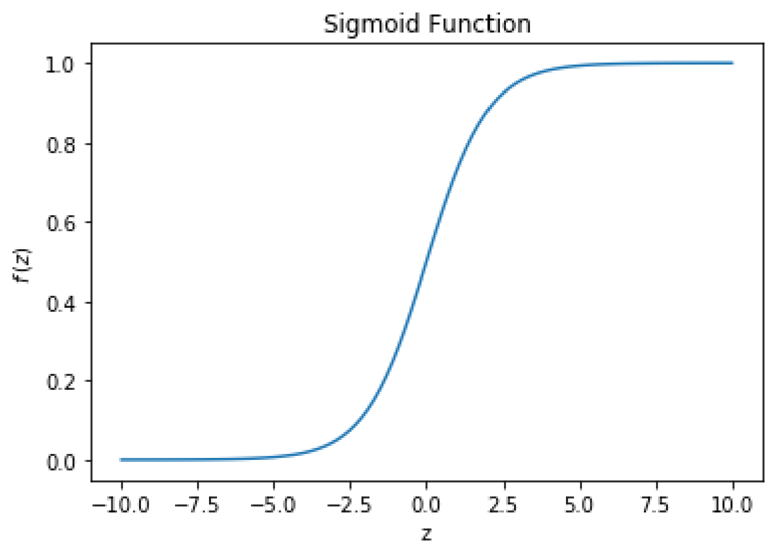
- What improvements are necessary to further boost the performance from ~2.4% down to ~0.2% test error?
  - Better strategy for choosing/evolving the learning rate and number of epochs —> hyper-parameter tuning
  - Further improvements of the GD optimisation scheme
  - A model architecture especially suited for images the so-called Convolutional Neural Networks (CNN)
- Just adding more hidden layers? —> Performance cannot be improved much for MNIST.
  - Possible explanation: Correlation between pixels is captured already with a single layer and there is not much more structure beyond to be captured for MNIST.
- How do the formulas look like when applying gradient descent to networks with hidden layer(s)?
  - $\rightarrow$ backpropagation algorithm

Activation Functions (Reading assignment)

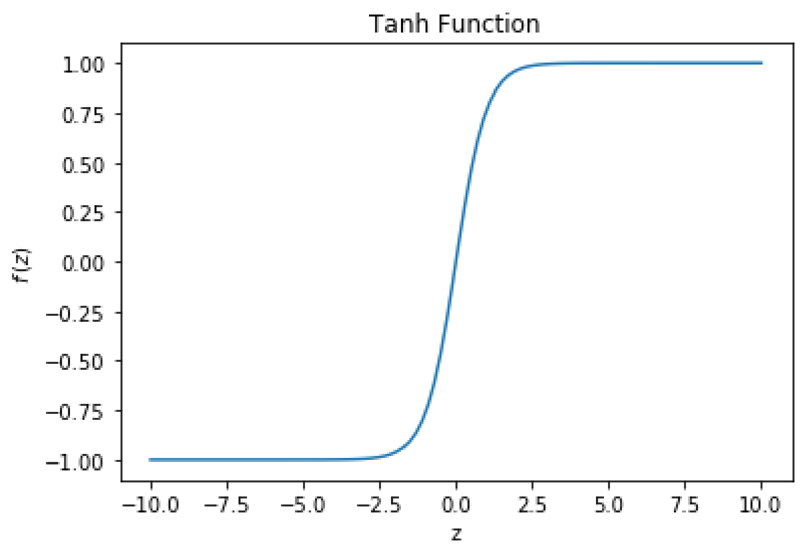
Heaviside



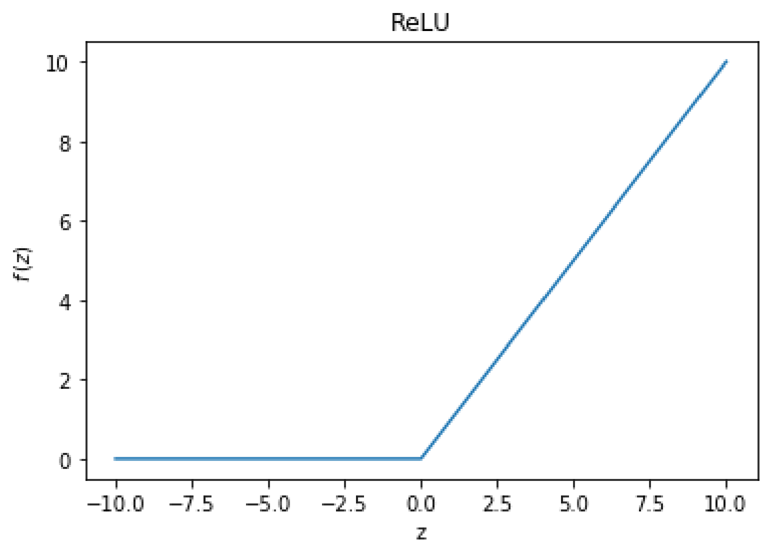
Sigmoid



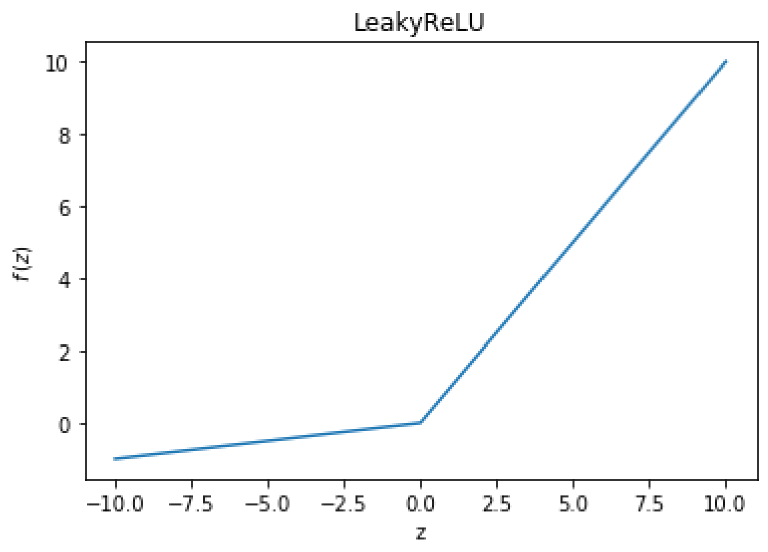
Tanh



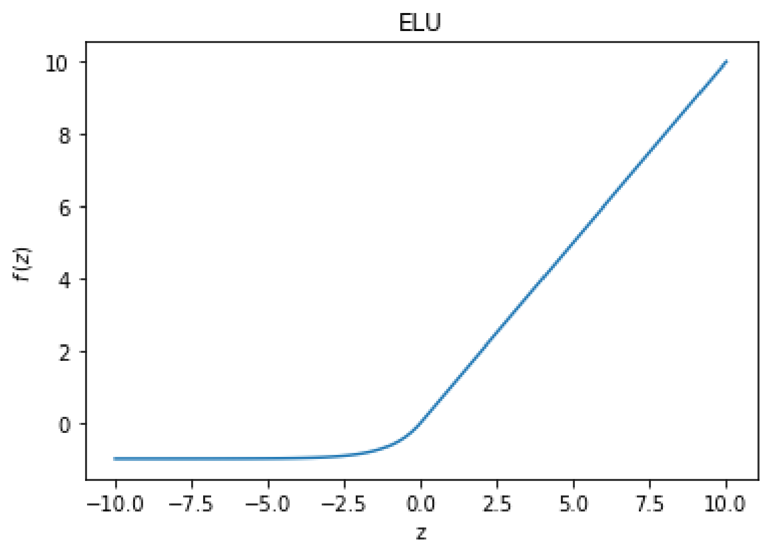
Recti-Linear Unit



Leaky Recti-Linear Unit



Exponential Linear Unit



What is the “(representational) capacity" of a ML model ?

Goodfellow (Deep Learning, Chapter 5.2, Link)

„capacity“:

„representational capacity“:

Why are non-linearities crucial for the representational capacity ?

Role of Activation Function

Non-Linearities Crucial for Sufficient Representational Capacity

What are the "vanishing gradient" and "dying unit" problems ?

„vanishing gradient problem“:

„dying unit problem“:

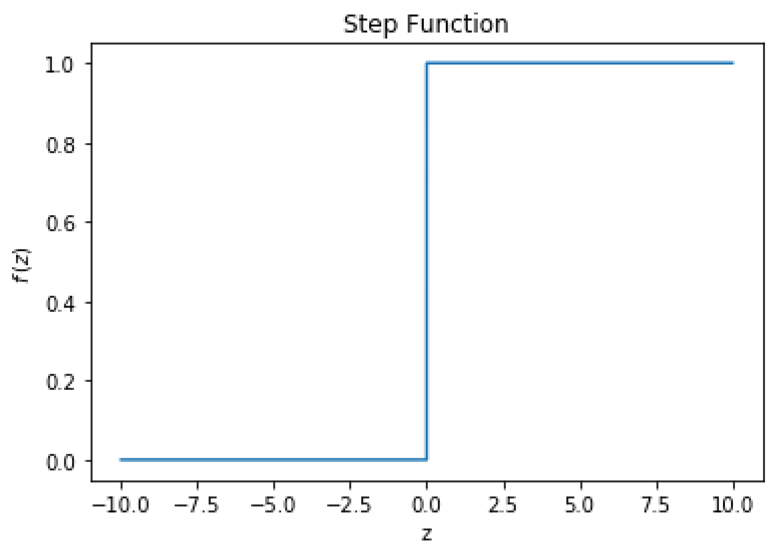
Which activations functions suffer from the "vanishing gradient" and which from the "dying unit" problem ?

„vanishing gradient problem“:

„dying unit problem“:

Activation Functions (1)

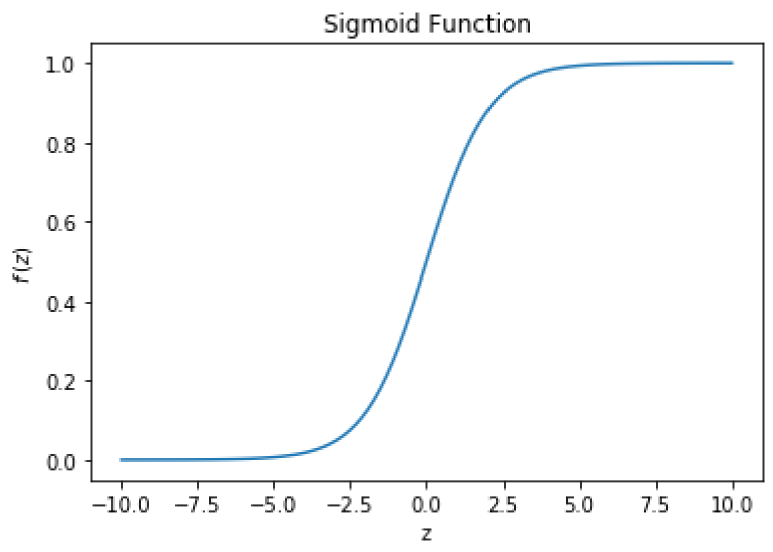
Heaviside



$$f(z) = \begin{cases} 1 & (z \geq 0) \\ 0 & (z < 0) \end{cases}$$

As in Rosenblatt's perceptron.
Not differentiable, i.e. gradient descent not possible.
No practical use.

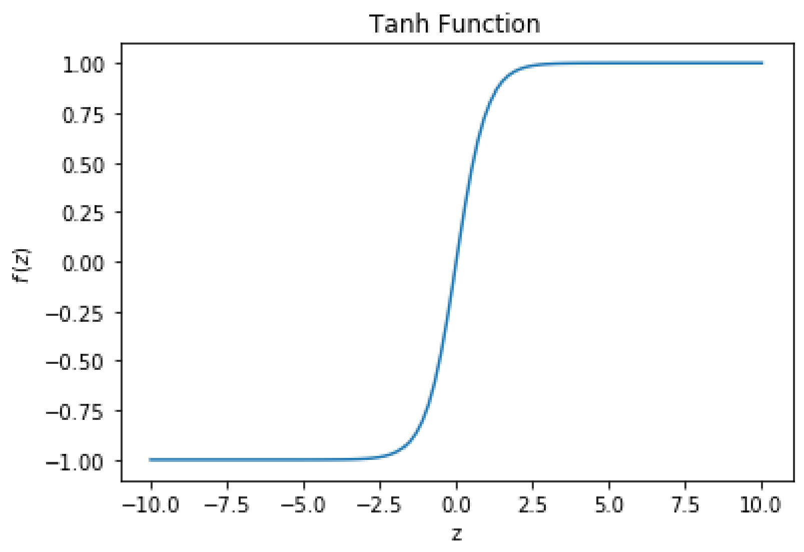
Sigmoid



$$f(z) = \frac{1}{1+\exp(-z)}$$

Most commonly used in textbooks and in illustrative examples.
In practice, typically used in output layers in binary classification.
Smooth and differentiable, i.e. gradient descent works.
But saturation regions leading to vanishing gradients. See Week 7.

Tanh

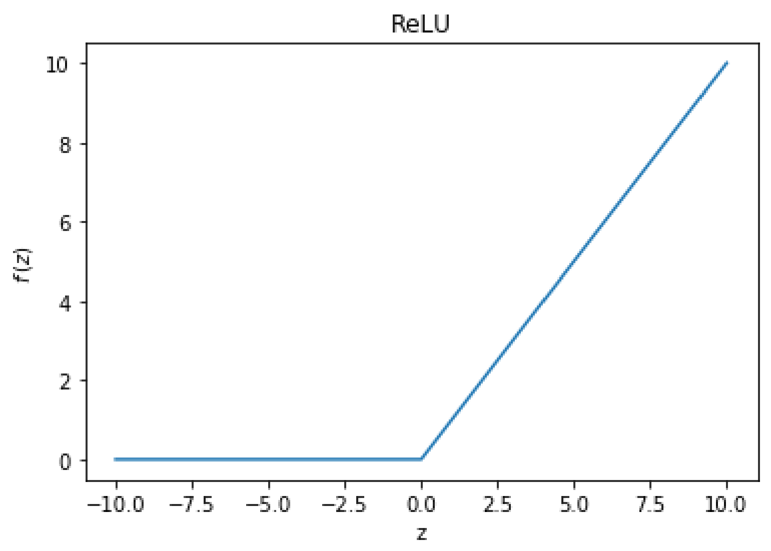


$$f(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

Often preferred over sigmoid since the output is centred around 0.
In practice, used e.g. in LSTM's (see week 12)
Smooth, i.e. gradient descent works.
But saturation regions leading to vanishing gradients. See Week 7.

Activation Functions (2)

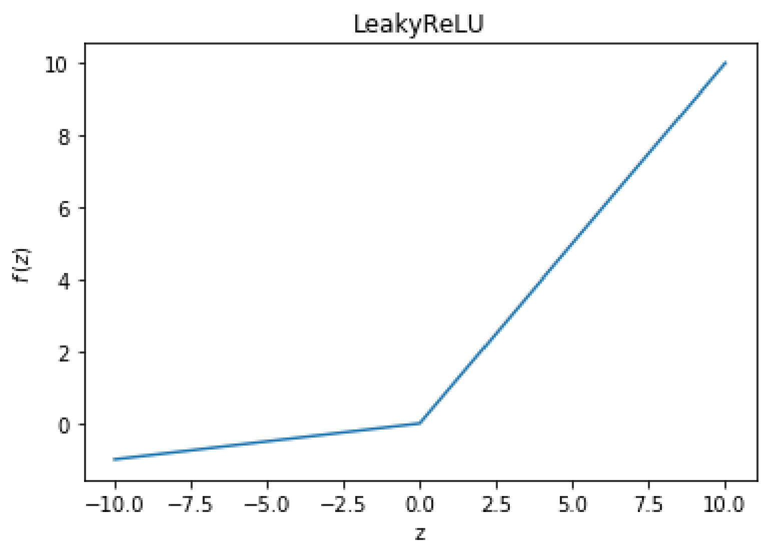
Recti-Linear Unit



$$f(z) = \max(0, z)$$

Used as de facto standard.
Introduced to alleviate the vanishing gradient problem (see Week 7).
However, suffer from dying units problem for z<0 where the activation and the gradient is 0.

Leaky Recti-Linear Unit



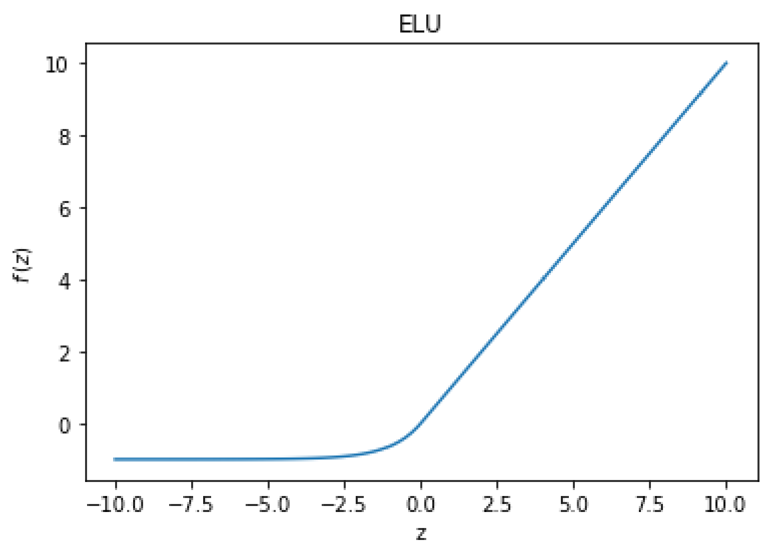
$$f(z) = \max(\alpha z, z)$$

Alleviates both, the vanishing gradient problem and the dying units problem (see Week 7).
Uses a small hyper-parameter

$$0 < \alpha < 1$$

which makes sure that the unit never dies (typical default: $\alpha = 0.01$

Exponential Linear Unit

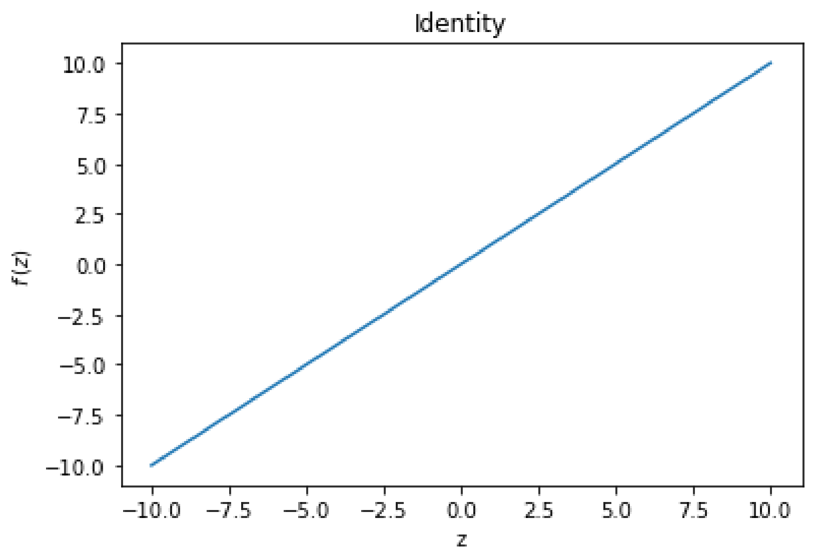


$$f(z) = \begin{cases} \alpha\left(\exp(z) - 1\right) & (z < 0) \\ z & (z \geq 0) \end{cases}$$

Similar to Leaky ReLU.
Negative activation stabilises at $-\alpha-$ but the gradient vanishes at small negative values.
More expensive to compute.

Activation Functions (3)

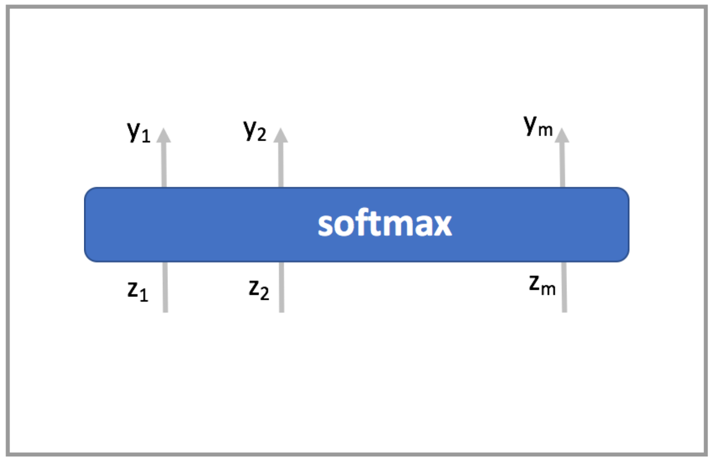
Identity



$$f(z) = z$$

Used only in specific cases such as in the last layer of a network for performing a regression task.

Softmax

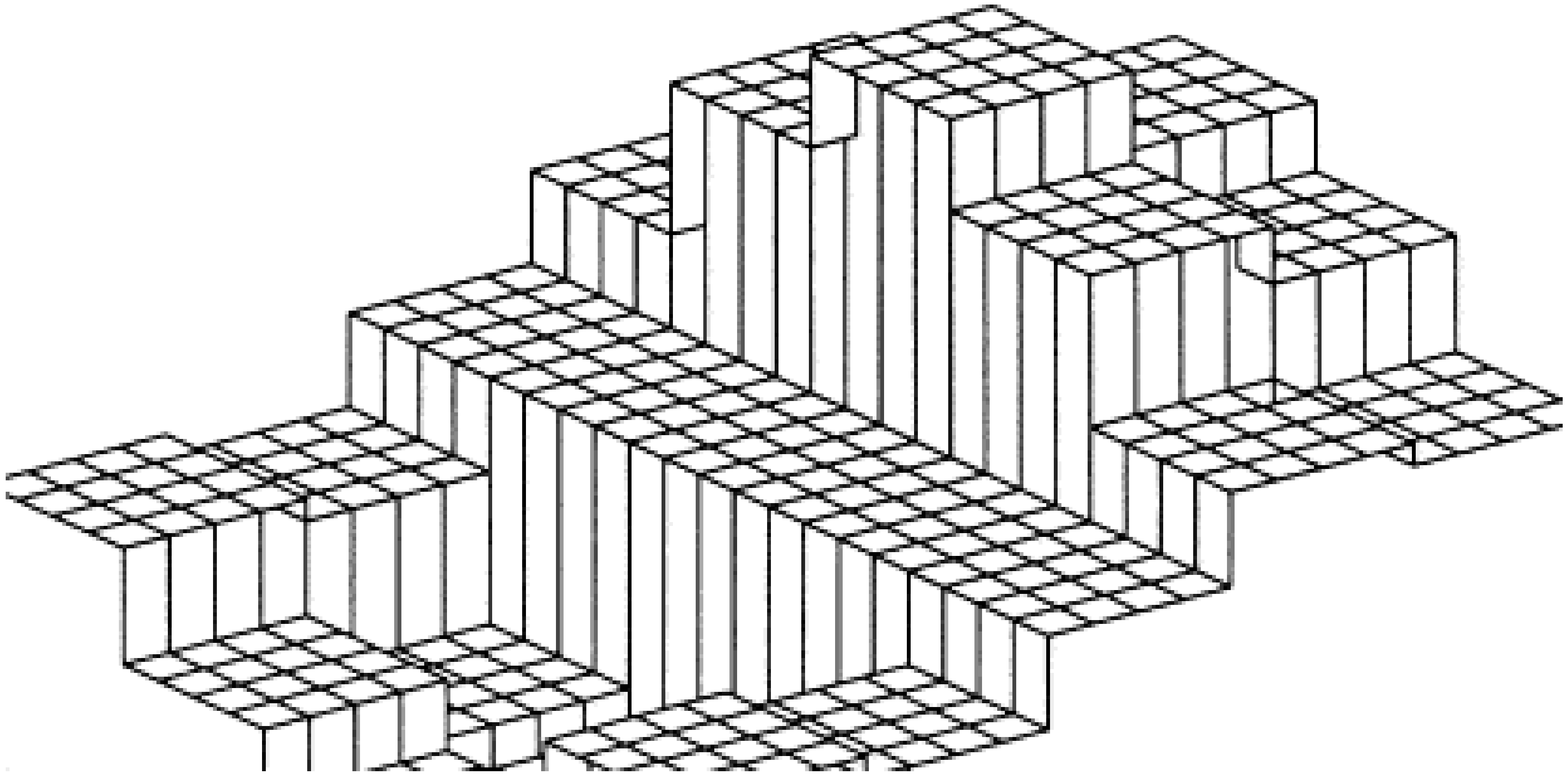


$$f(\mathbf{z})_l = \frac{\exp(z_l)}{\sum_{j=1} \exp(z_j)}$$

Used as last layer in a network for a classification task with *m* classes.
Output vector can be interpreted as probability distribution.

More on activation functions in later weeks.

Universal Function Approximation



Neural Networks as Function Approximators

- Neural networks provide models for predicting an outcome y from an input $\mathbf{x}$. In general, we expect to find only an approximate mapping $\mathbf{x} \rightarrow y$
- The mapping is approximated by searching in a suitable parametric family of functions $h_\theta(\mathbf{x})$ with parameters θ:

$$\mathbf{x} \rightarrow y = h(\mathbf{x}) \approx h_\theta(\mathbf{x}) = \hat{y}$$

  The richness of this family of functions drives the ability to represent more or less complex mappings. This ability is also referred to as *representational capacity*.
- For neural networks, the representational capacity is determined by the number of layers, type of layers, the number of neurons per layer, type of activation function, type of neuron (see later).

Universal Approximation Theorem

Shallow networks (networks with a single hidden layer) provide sufficient capacity for approximating quite general functions!

A result of Cybenko, 1989(1) and Hornik, 1989

A feedforward network with a linear output layer and at least one hidden layer with a non-linear ("squashing") activation function (e.g. sigmoid) can approximate a large class of functions(2) $h : \mathbb{R}^{n_x} \rightarrow \mathbb{R}^{n_y}$ with arbitrary accuracy(3) - provided that the network is given a sufficient number of hidden units and the parameters are suitably chosen.

(1) http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.441.7873&rep=rep1&type=pdf
(2) E.g. any continuous function on compact support.
(3) Accuracy quantified with suitable distance measure (e.g. L^2-distance ~ integrated mean-square distance)

Universal Approximation Theorem

Activity

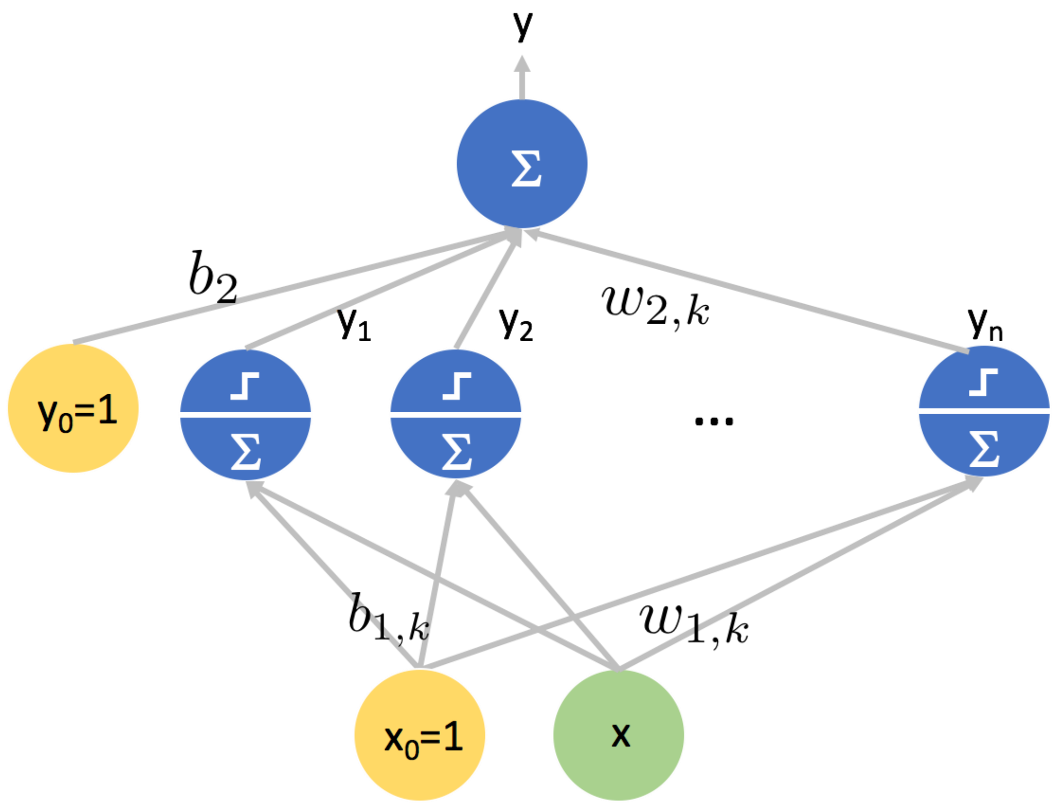
The statement made by the "Universal Approximation Theorem" is strong!
Discuss in groups what it could mean.

Possible (further) points to discuss:

- Are all problems solved? What points are missing for a practical applications?
- It makes a statement about functions.
  But we typically start with data.
  What is the connection?

Function Approximation with Sigmoids: 1d

- One-dimensional real-valued input *x* .
- One hidden layer with *n* neurons and sigmoid activation function, weights $\omega_{1,k}$ and bias $b_{2,k}$
- Linear output layer with weights $\omega_{2,k}$ and biases $b_{1,k}$
- The linear output layer leads to a linear combination of sigmoids:



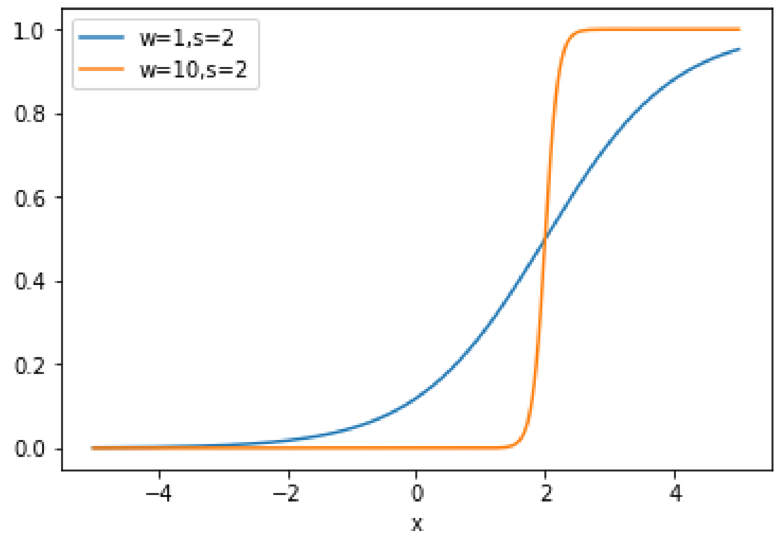
$$h_\theta(x) = \sum_{k=1}^{n} \omega_{2,k} \cdot \sigma(\omega_{1,k} \cdot x + b_{1,k}) + b_2$$

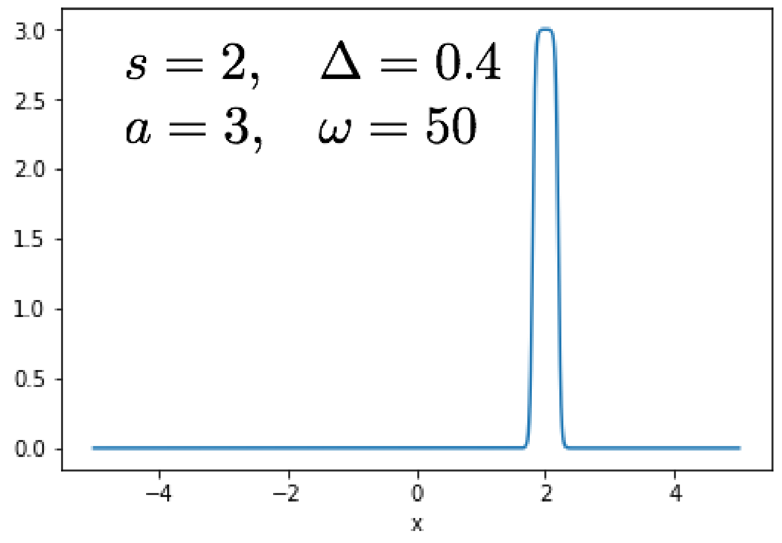
- The theorem states that we can approximate any quite general function (1d input, 1d output) if we choose suitable parameters and a suitable number of neuron *n*.

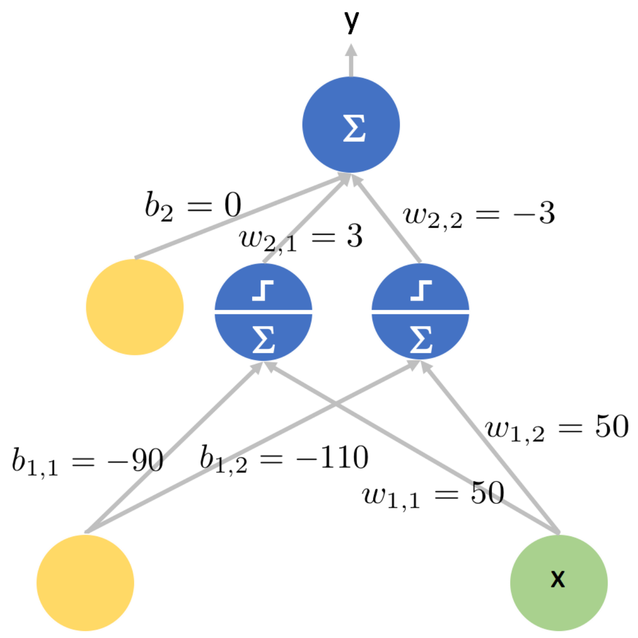
Function Approximation with Sigmoids

Inspired by http://neuralnetworksanddeeplearning.com/chap4.html

For convenience, we re-parametrise the sigmoids with the position of the step $s = -b/\omega$ so that $\sigma(\omega \cdot x + b) = \sigma(\omega \cdot (x - s))$. With the parameter ω we control the slope of the step, i.e. the larger ω the steeper the step as indicated in the figure on the left.





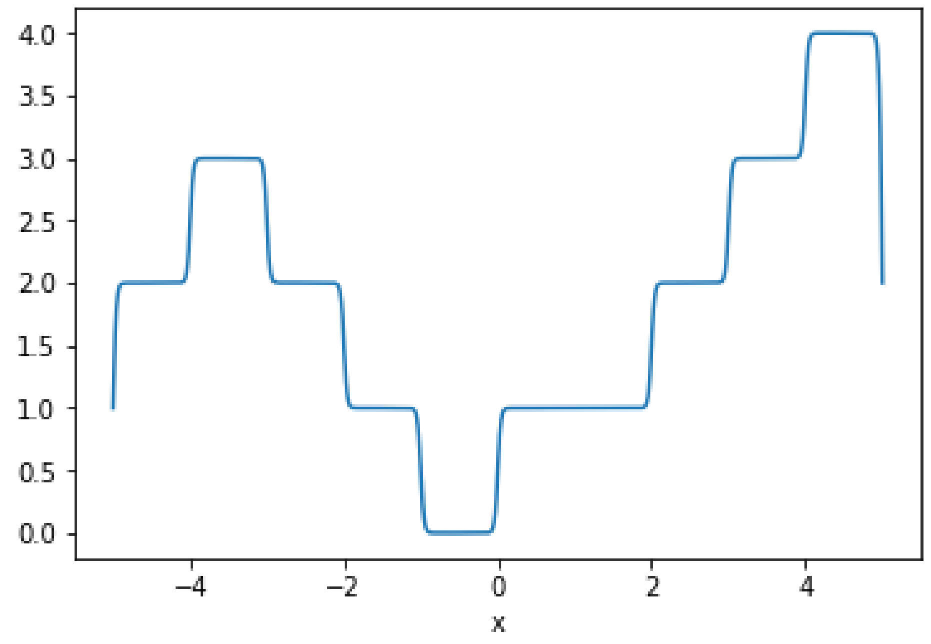


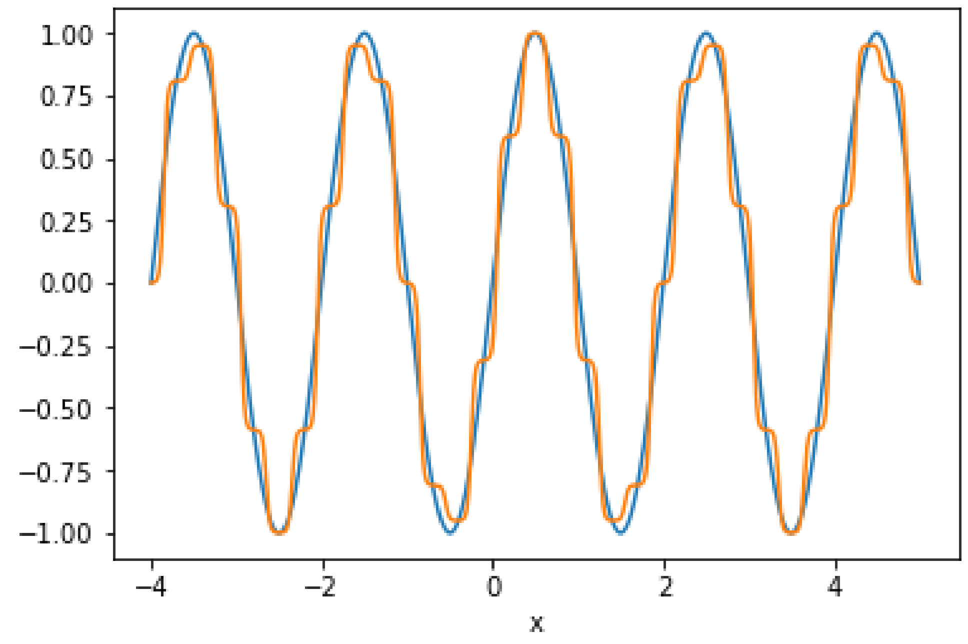
By combining two such step functions we can easily create a peak (see figure on the right and graph) at location s with width : Δ

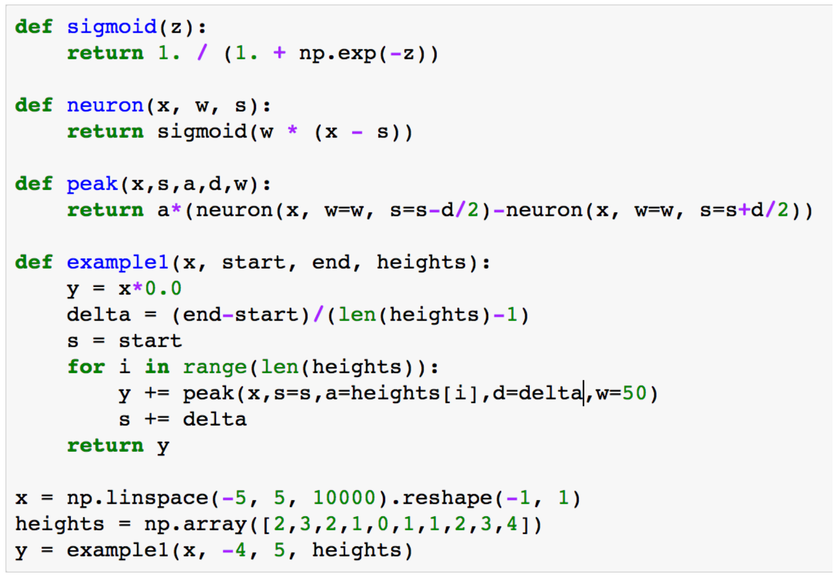
$$\phi(x; s, a, \Delta, \omega) = a\left(\sigma(w \cdot (x - s + \Delta/2)) - \sigma(w \cdot (x - s - \Delta/2))\right)$$

Function Approximation with Sigmoids

Linearly combining such peaks at different locations allows to construct arbitrary step-wise functions. A large class of functions can be approximated arbitrarily close with step functions.







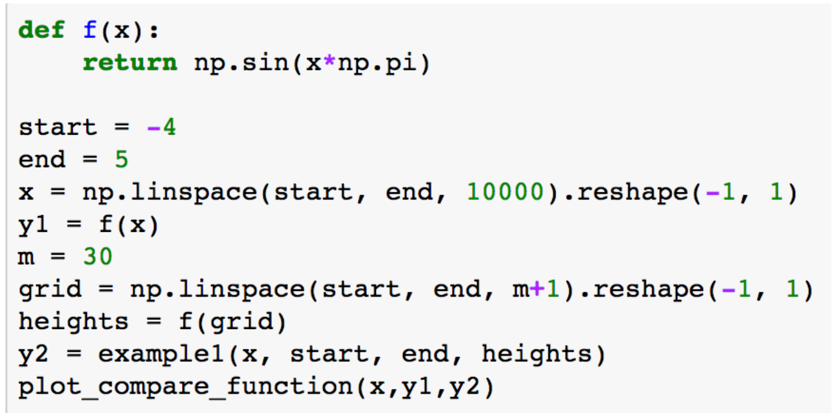
```
def sigmoid(z):
    return 1. / (1. + np.exp(-z))

def neuron(x, w, s):
    return sigmoid(w * (x - s))

def peak(x,s,a,d,w):
    return a*(neuron(x, w=w, s=s-d/2)-neuron(x, w=w, s=s+d/2))

def example1(x, start, end, heights):
    y = x*0.0
    delta = (end-start)/(len(heights)-1)
    s = start
    for i in range(len(heights)):
        y += peak(x,s=s,a=heights[i],d=delta,w=50)
        s += delta
    return y

x = np.linspace(-5, 5, 10000).reshape(-1, 1)
heights = np.array([2,3,2,1,0,1,1,2,3,4])
y = example1(x, -4, 5, heights)
```



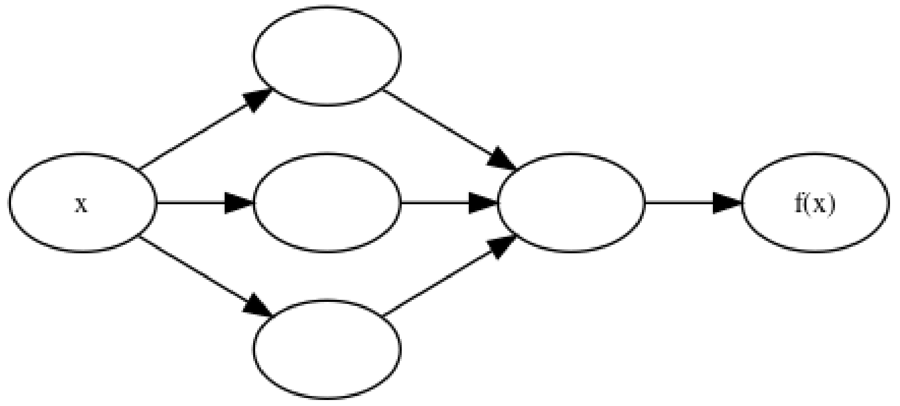
```
def f(x):
    return np.sin(x*np.pi)

start = -4
end = 5
x = np.linspace(start, end, 10000).reshape(-1, 1)
y1 = f(x)
m = 30
grid = np.linspace(start, end, m+1).reshape(-1, 1)
heights = f(grid)
y2 = example1(x, start, end, heights)
plot_compare_function(x,y1,y2)
```

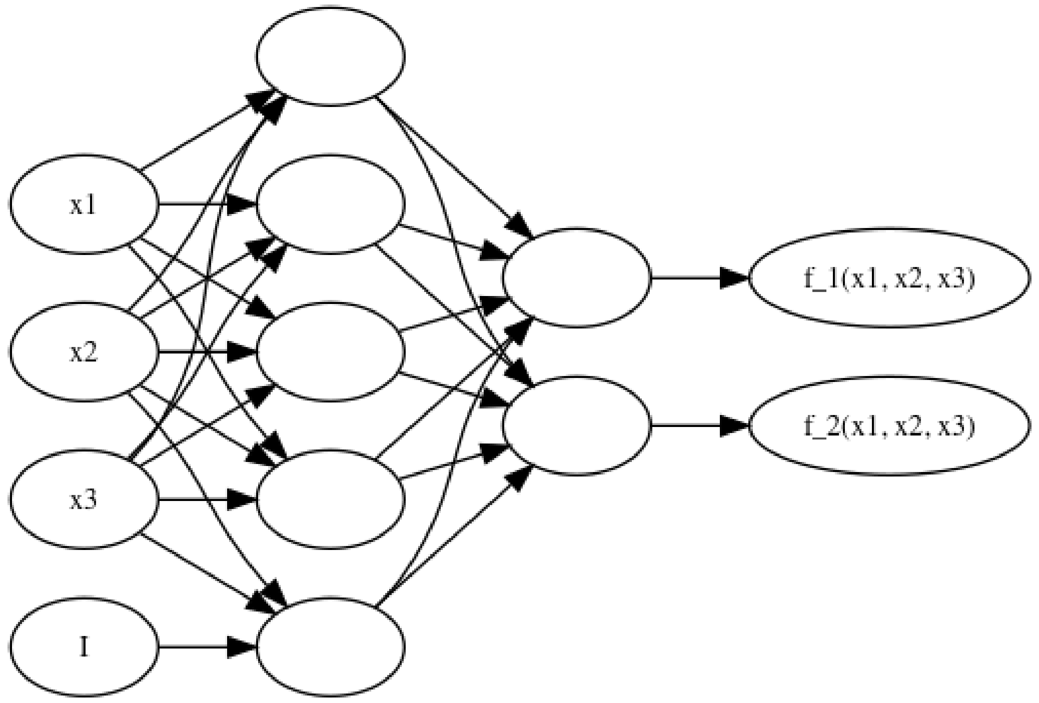
(see Exercise in PW for Week 3 / Exercise 3)

Generalisation to Arbitrary Functions

The network considered so far has the shape as depicted in the upper figure.



The scheme for *1*d input and *1*d output can easily be generalised to *n*d input and *m*d output.

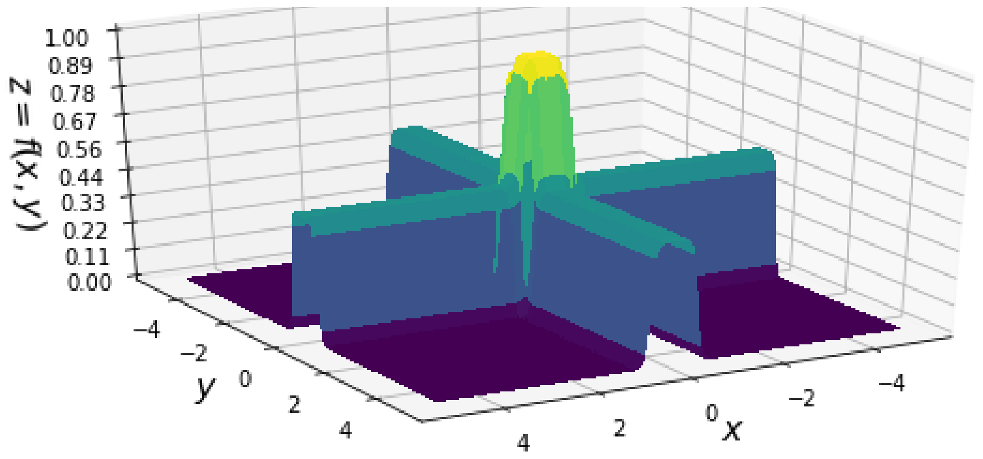


Function Approximation in 2d

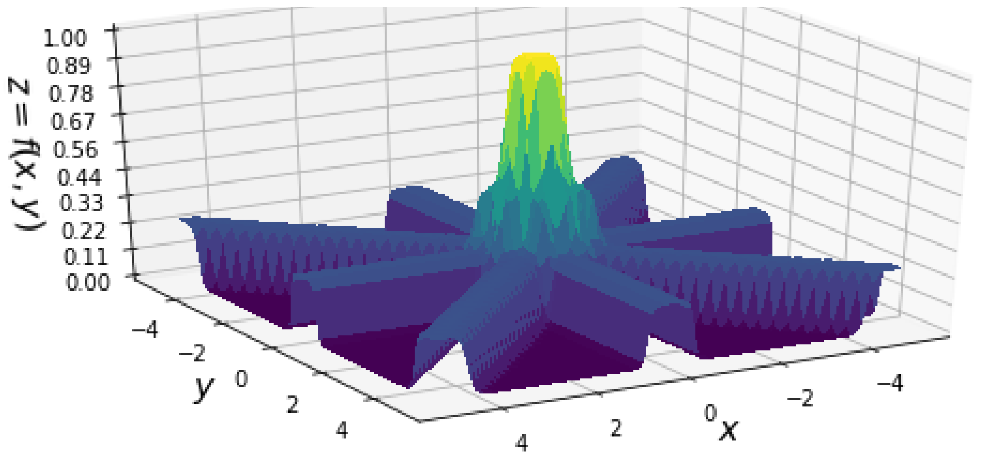
Peak function by linearly super-imposing ridges oriented in different directions - intensity in the center amplified in proportion to the number of ridges involved.

By super-imposing peak functions, “any” function can be approximated.

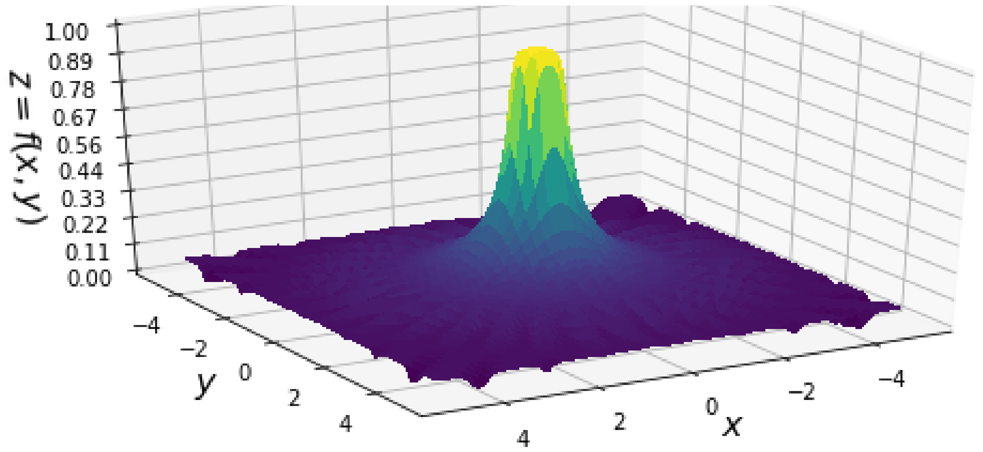
2 ridges, 4 hidden neurons



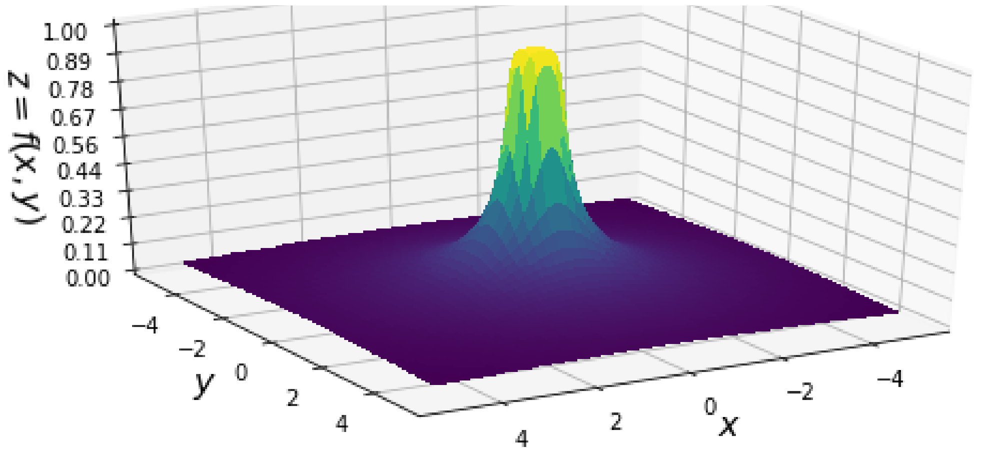
4 ridges, 8 hidden neurons



16 ridges, 32 hidden neurons



64 ridges, 128 hidden neurons



Problems

- For improving the accuracy when modelling with step-functions, more and more neurons are needed and many parameters need to be determined. In problems with high-dimensional input, an exponentially growing number of neurons is needed.

- Accordingly, more data sampled on a sufficiently fine grid is needed. With growing dimensionality in the input (e.g. image or audio data) this becomes infeasible. This is known as curse of dimensionality.

- If we just use the available data and interpolate with step-functions between data points we significantly overfit on the training data.

- The theorem does not provide a scheme for efficiently learning the parameters from available limited data.

Bias – Variance
Overfitting

Motivation

“The central challenge in machine learning is that we must perform well on new, previously unseen inputs — not just those on which our model was trained. The ability to perform well on previously unobserved inputs is called generalisation.” (I.Goodfellow et al., *Deep Learning)*

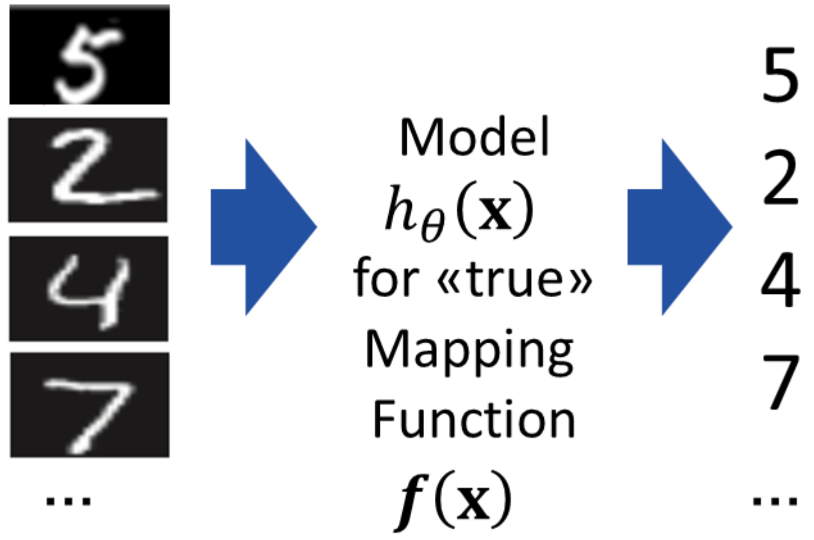
Thus

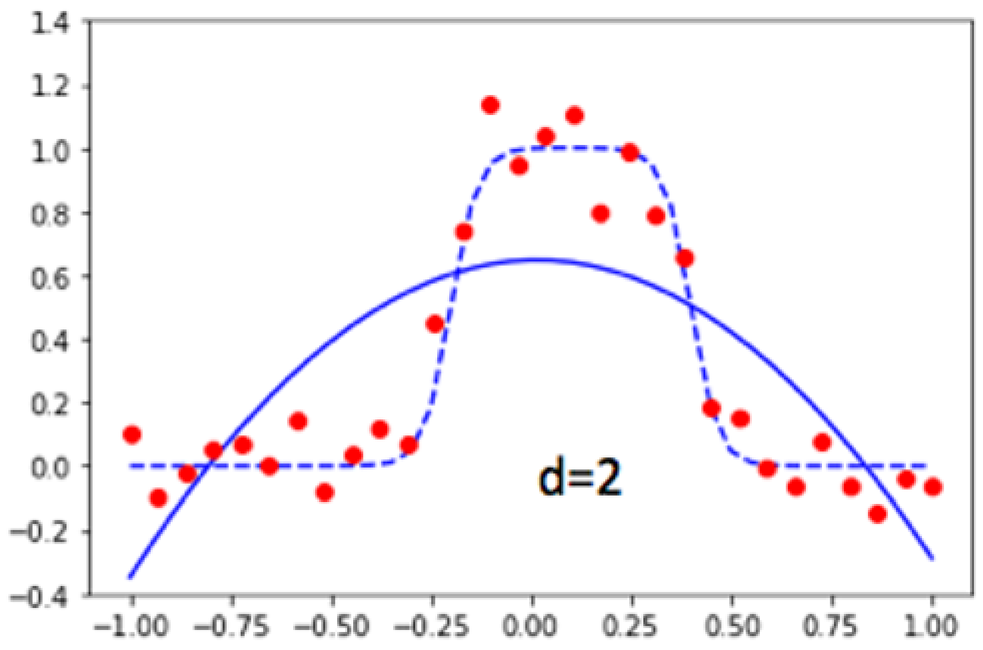
- We are interested in the generalization performance
- How do we estimate the generalization error ?

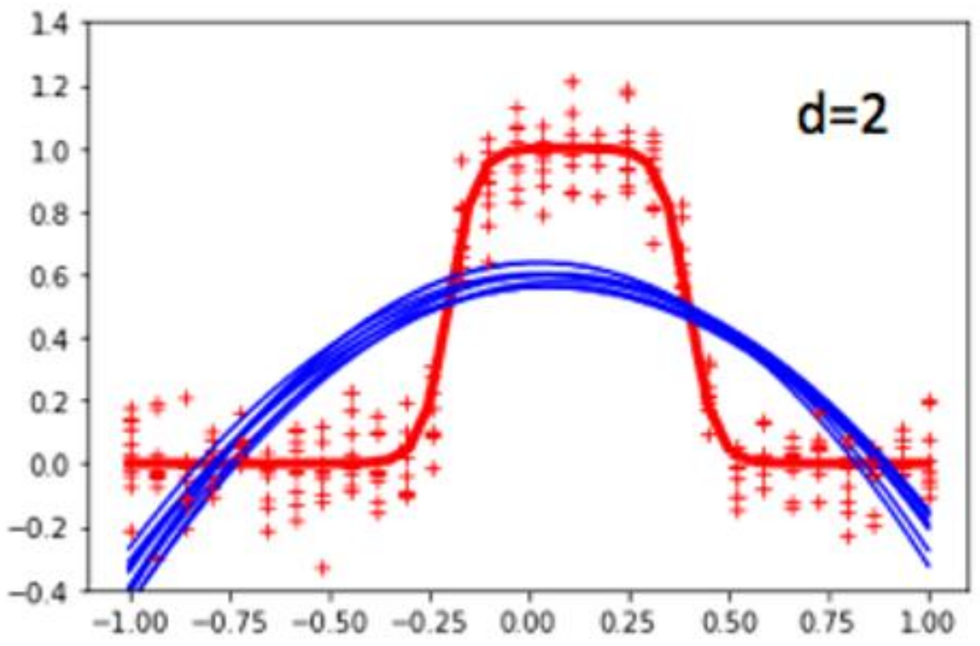
Bias and Variance

$$\mathrm{bias}(h_\theta) = \mathbf{E}\left[h_{\theta,D}\right] - f$$

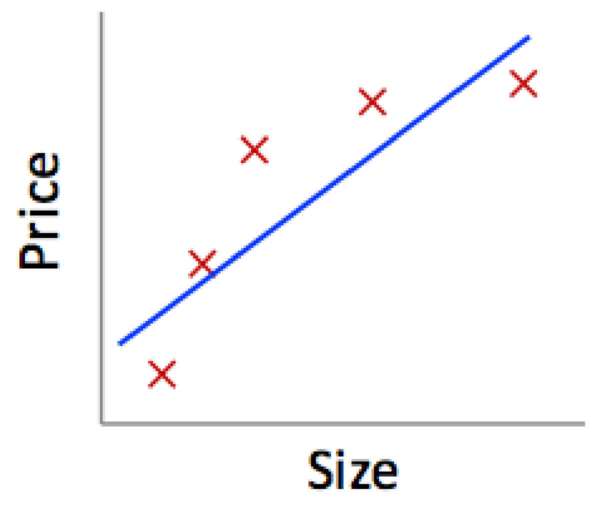
$$\mathrm{Var}(h_\theta) = \mathbf{E}\left[\left(h_{\theta,D} - \mathbf{E}\left[h_{\theta,D}\right]\right)^2\right]$$



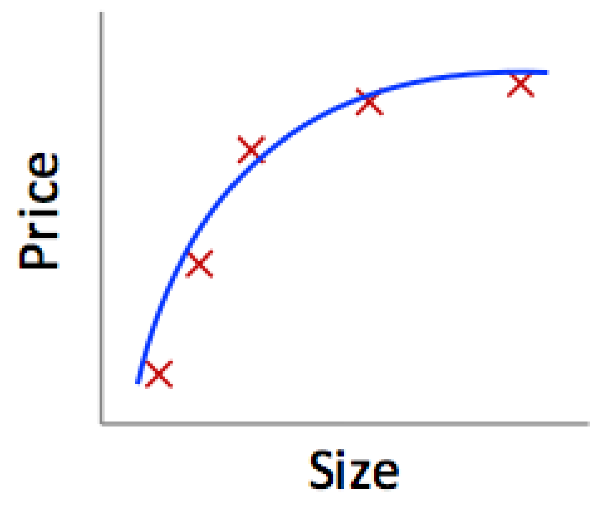




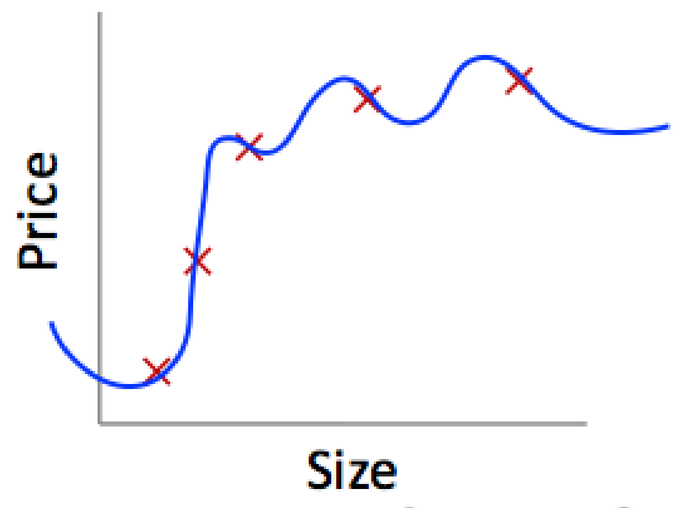
Guiding Principle



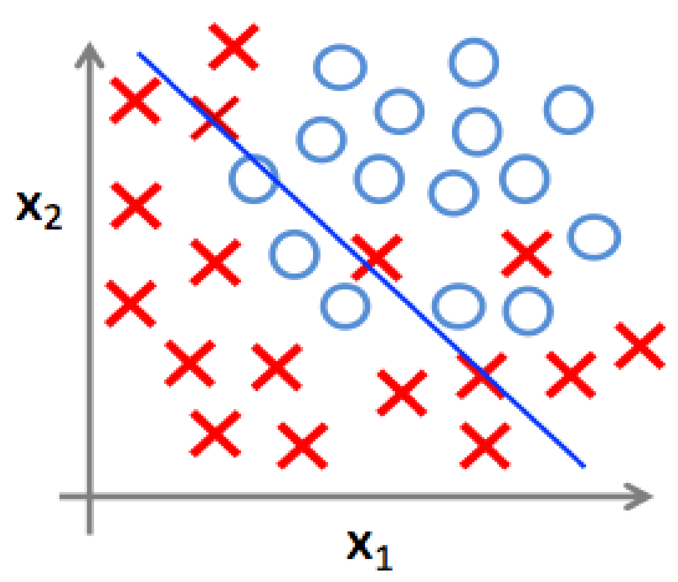
$$\theta_0 + \theta_1 x$$



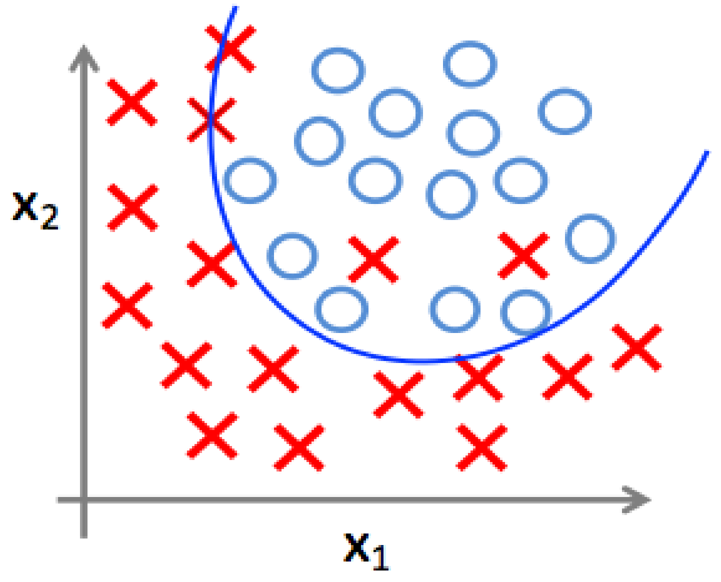
$$\theta_0 + \theta_1 x + \theta_2 x^2$$



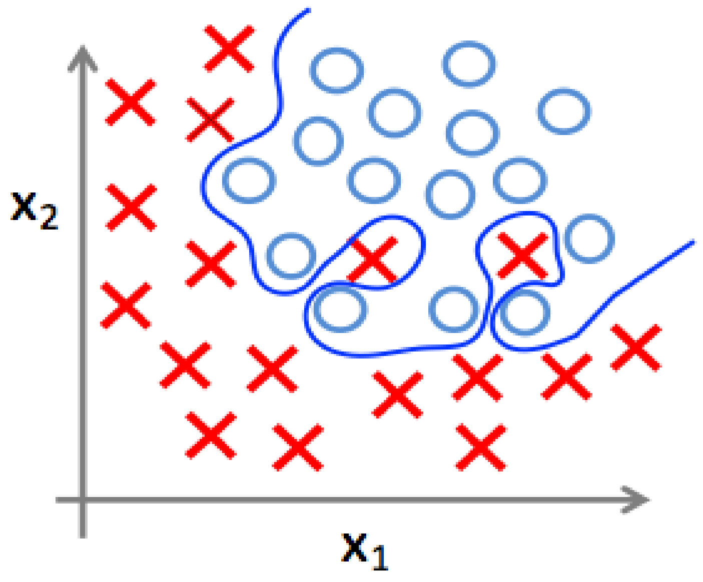
$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$



Underfit
"high bias"



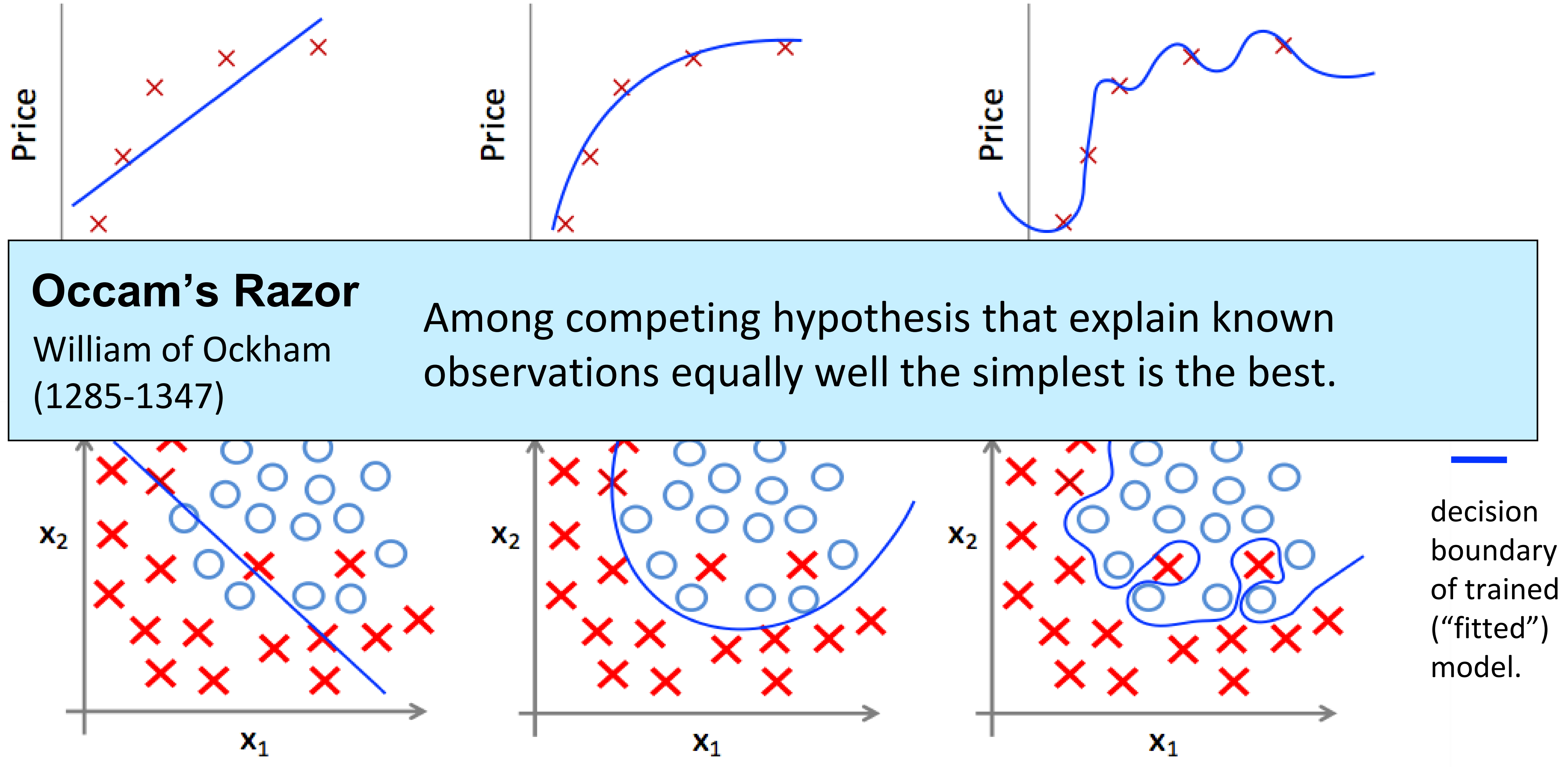
Good Fit
"just right"



Overfit
"high variance"

— decision boundary of trained ("fitted") model.

Guiding Principle



Occam's Razor

William of Ockham (1285-1347)

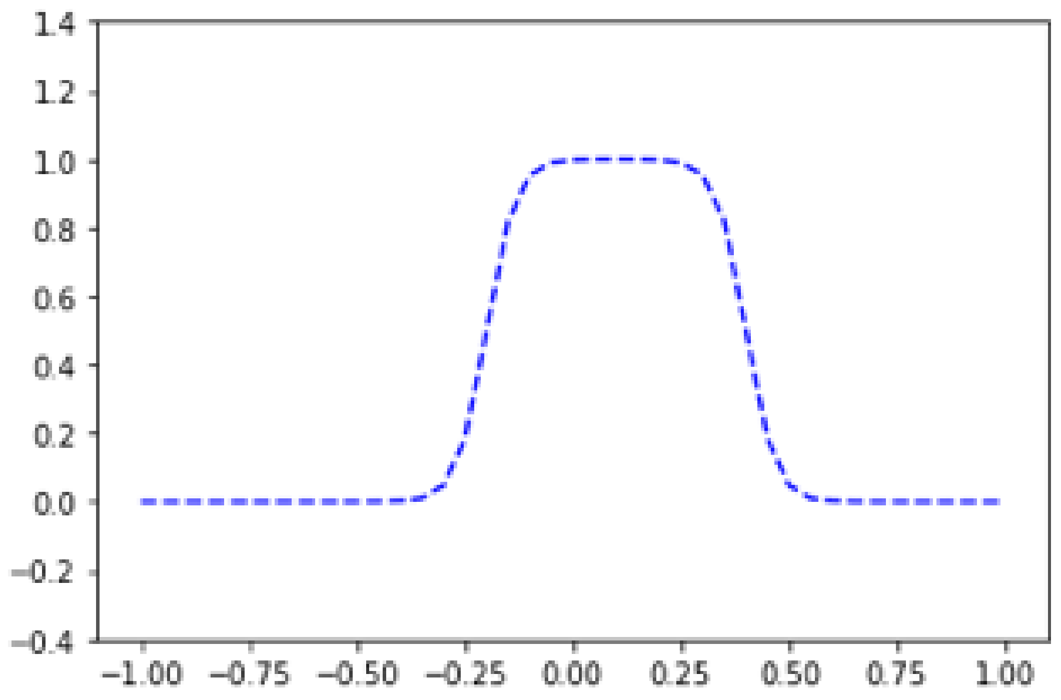
Among competing hypothesis that explain known observations equally well the simplest is the best.

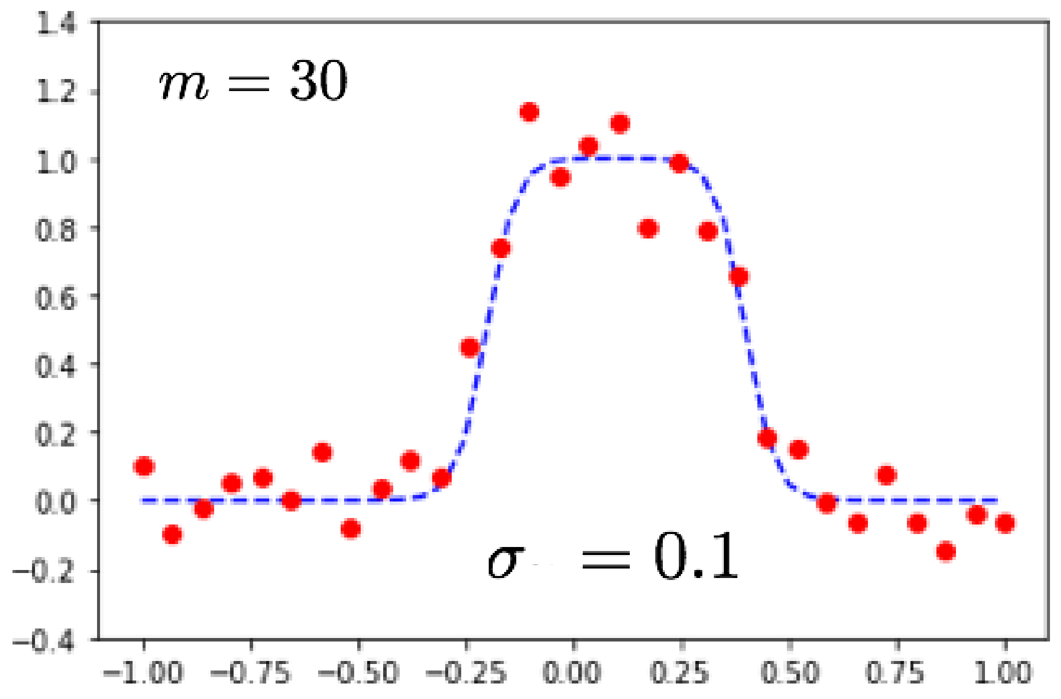
Underfit
"high bias"

Good Fit
"just right"

Overfit
"high variance"

Illustrative Example





Underlying Structure

unknown, assumed to be given by mapping

$$x \rightarrow y = f(x)$$

Regression Problem with
1d input / 1d output

Observed Data - Training Set (●)

noisy, Gaussian-distributed:

$$x \rightarrow y = f(x) + \sigma \cdot \mathcal{N}(0,1)$$

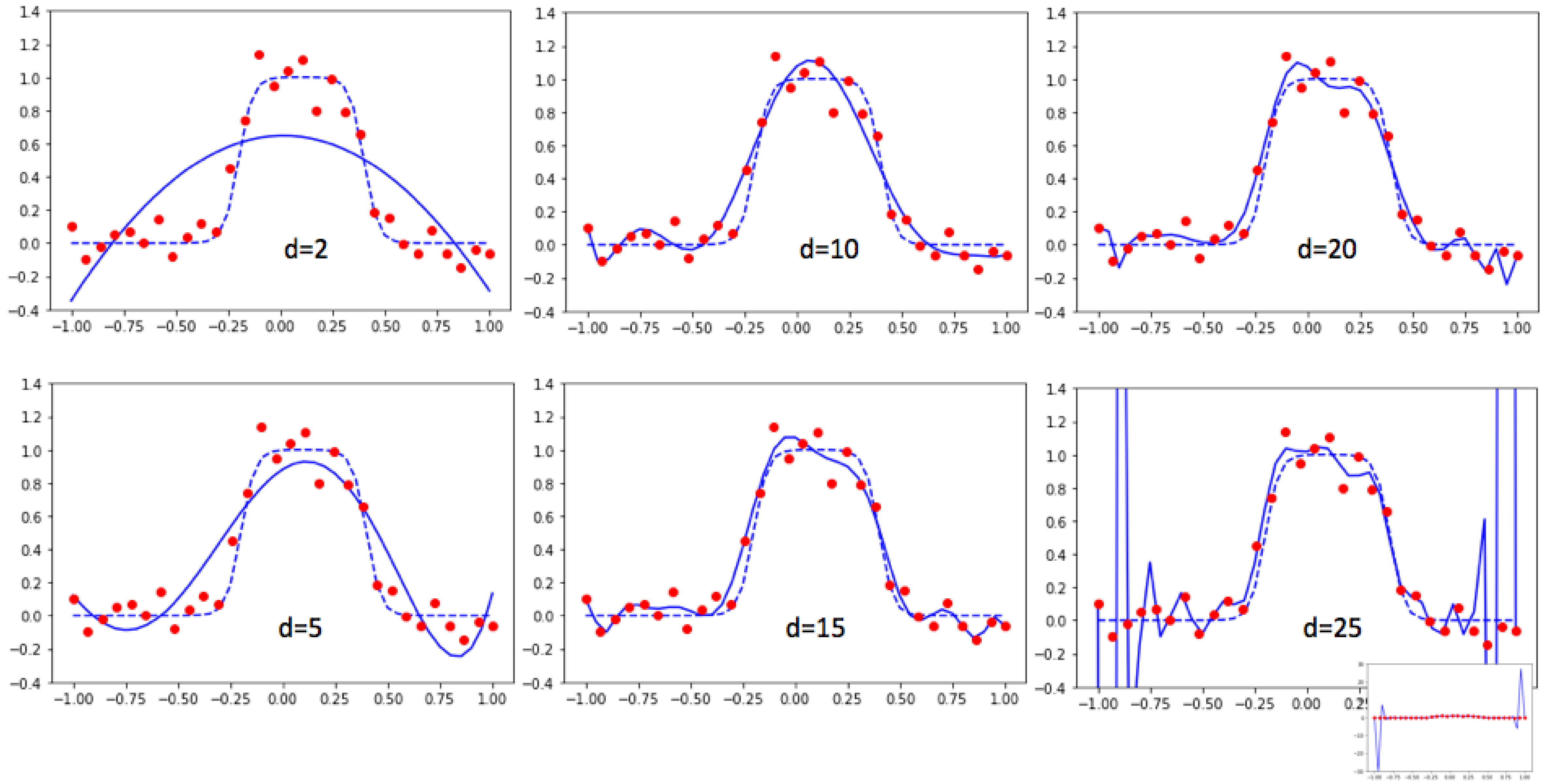
Model

to be learned from the data

$$h_\theta(\mathbf{x}) = \theta_0 + \theta_1 \cdot x + \theta_2 \cdot x^2 + .. + \theta_d \cdot x^d$$

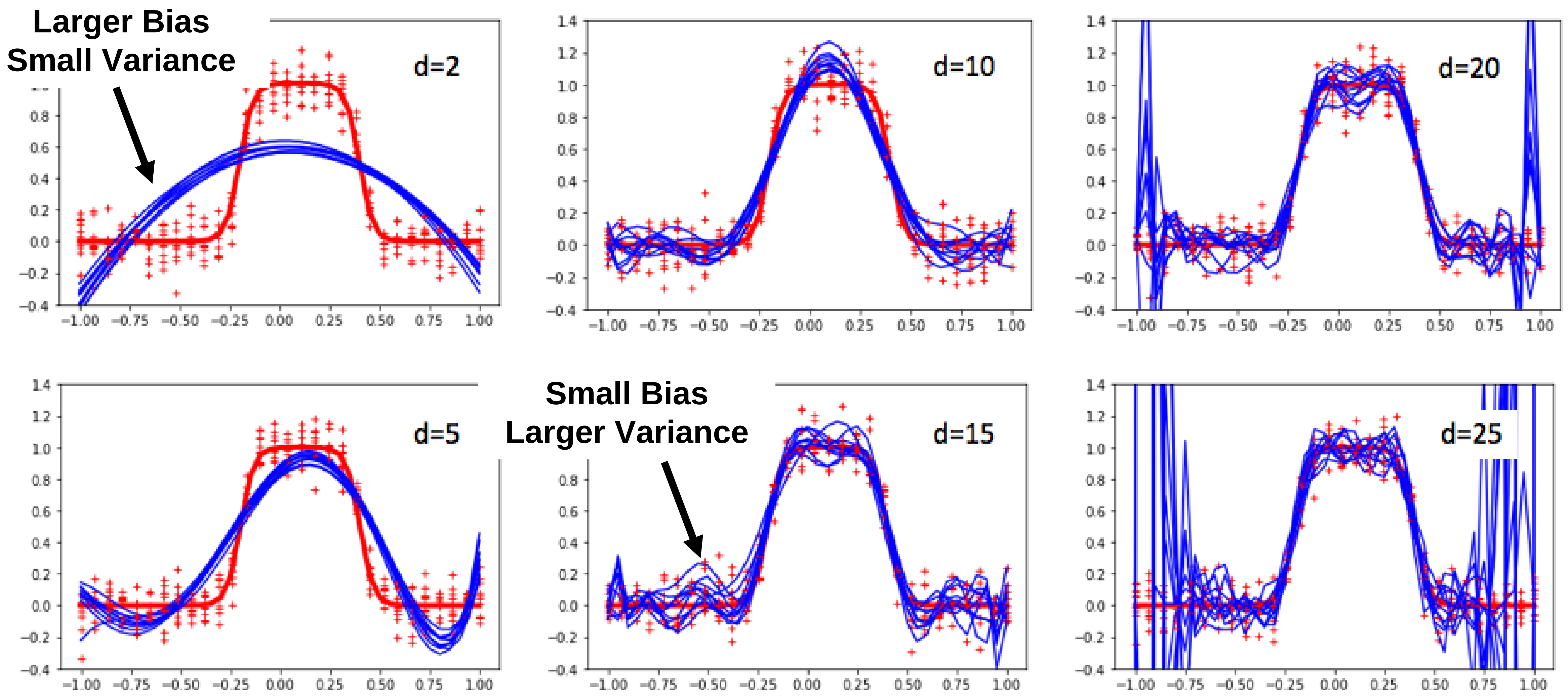
Illustrative Example

Results obtained by minimising MSE cost with different model complexities *d*.

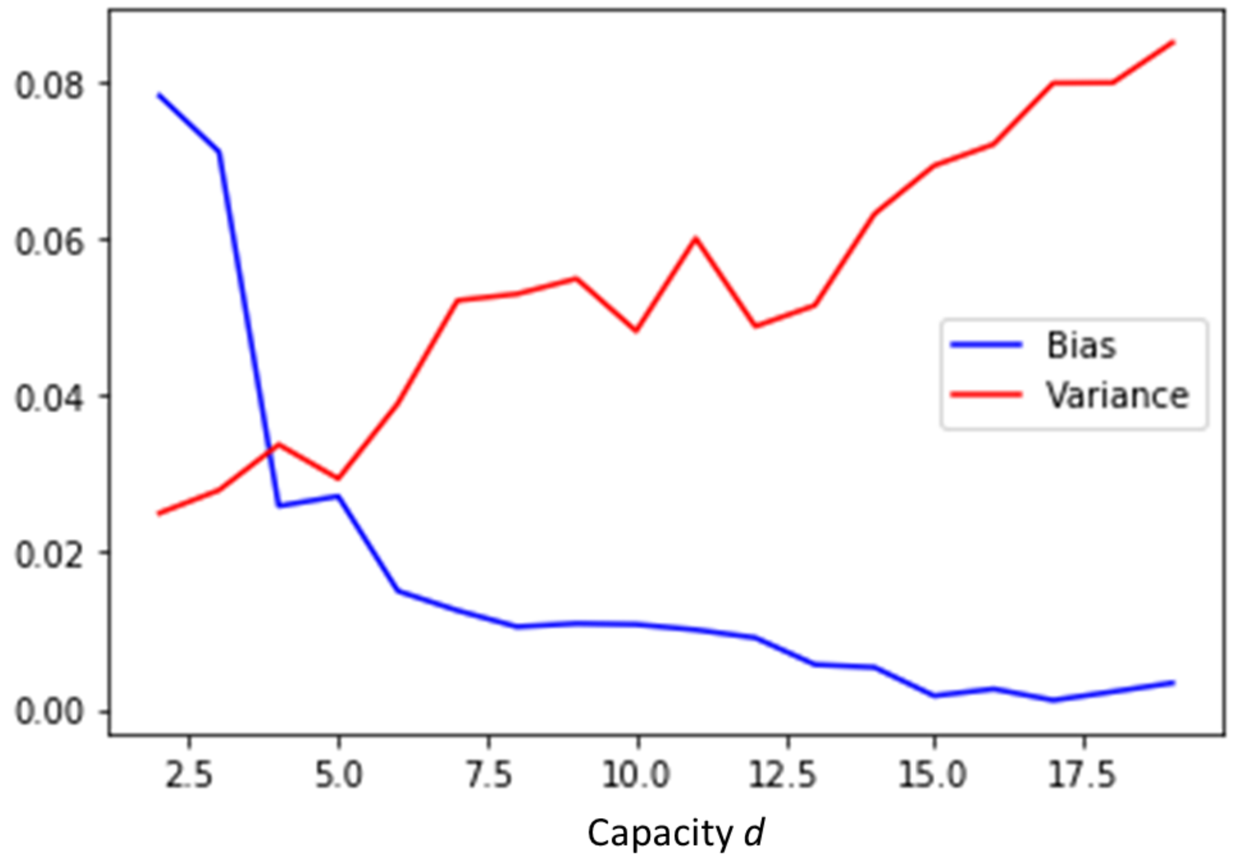


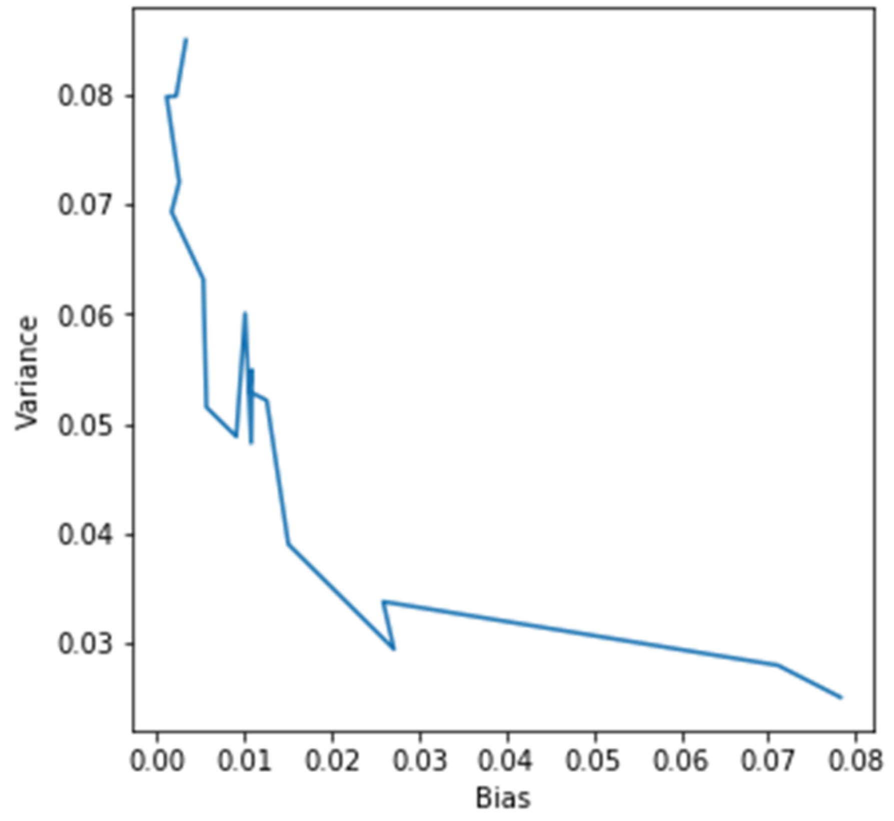
Illustrative Example

Independently generated training sets (with m=30 each



Bias-Variance Tradeoff

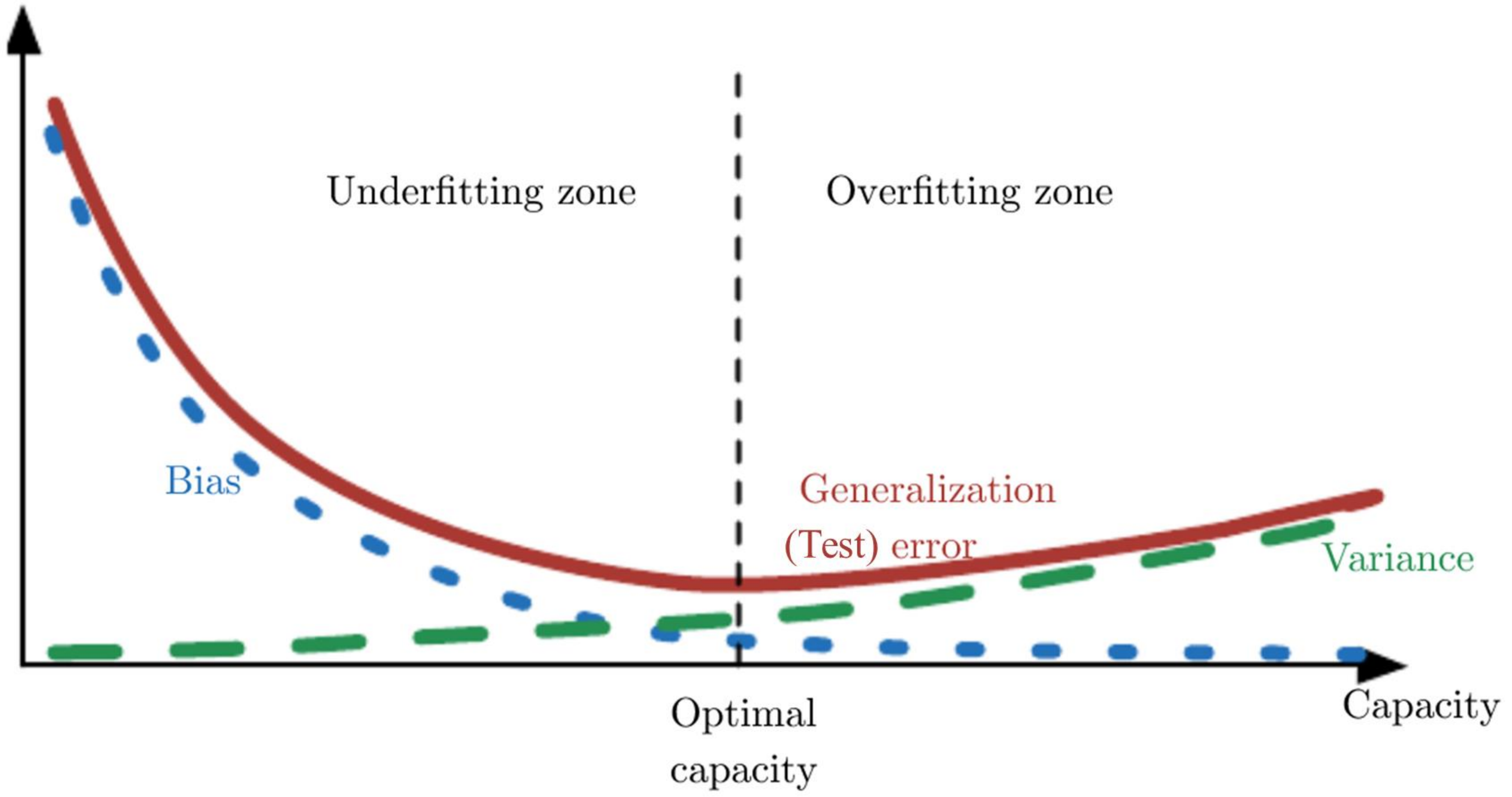




Generalization error:

$$\mathrm{MSE} = \mathbf{E}\left[\left(h_{\theta,D} - f\right)^2\right] = \mathrm{bias}(h_\theta)^2 + \mathrm{Var}(h_\theta)$$

Bias-Variance Tradeoff



$$\mathrm{MSE} = \mathbf{E}\left[\left(h_{\theta,D} - f\right)^2\right] = \mathrm{bias}(h_\theta)^2 + \mathrm{Var}(h_\theta) + \sigma^2$$

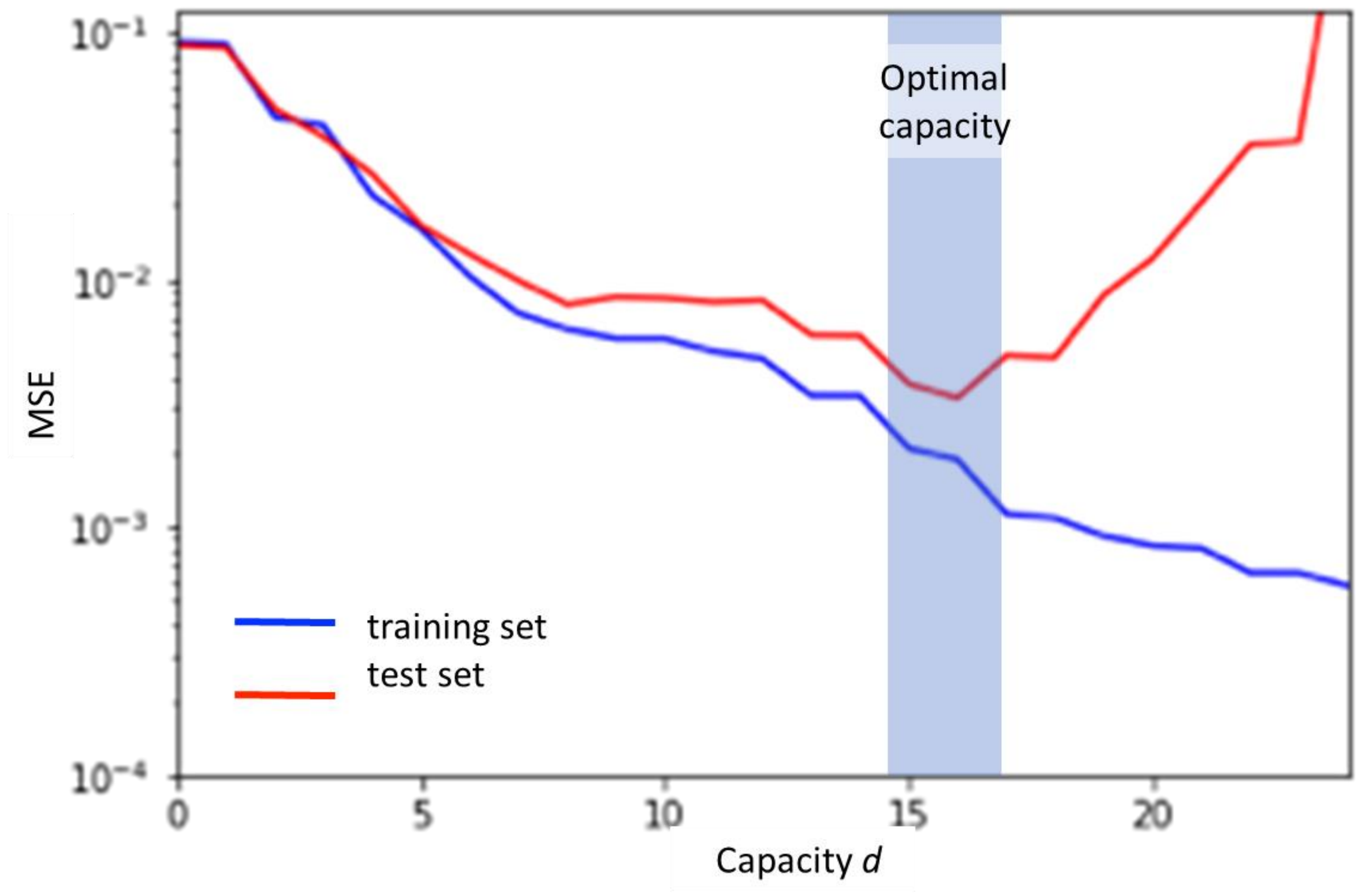
Characterisation of Overfitting

Overfitting occurs when the learned hypothesis (trained model) fits the training data set very well - but fails to generalise to new examples.

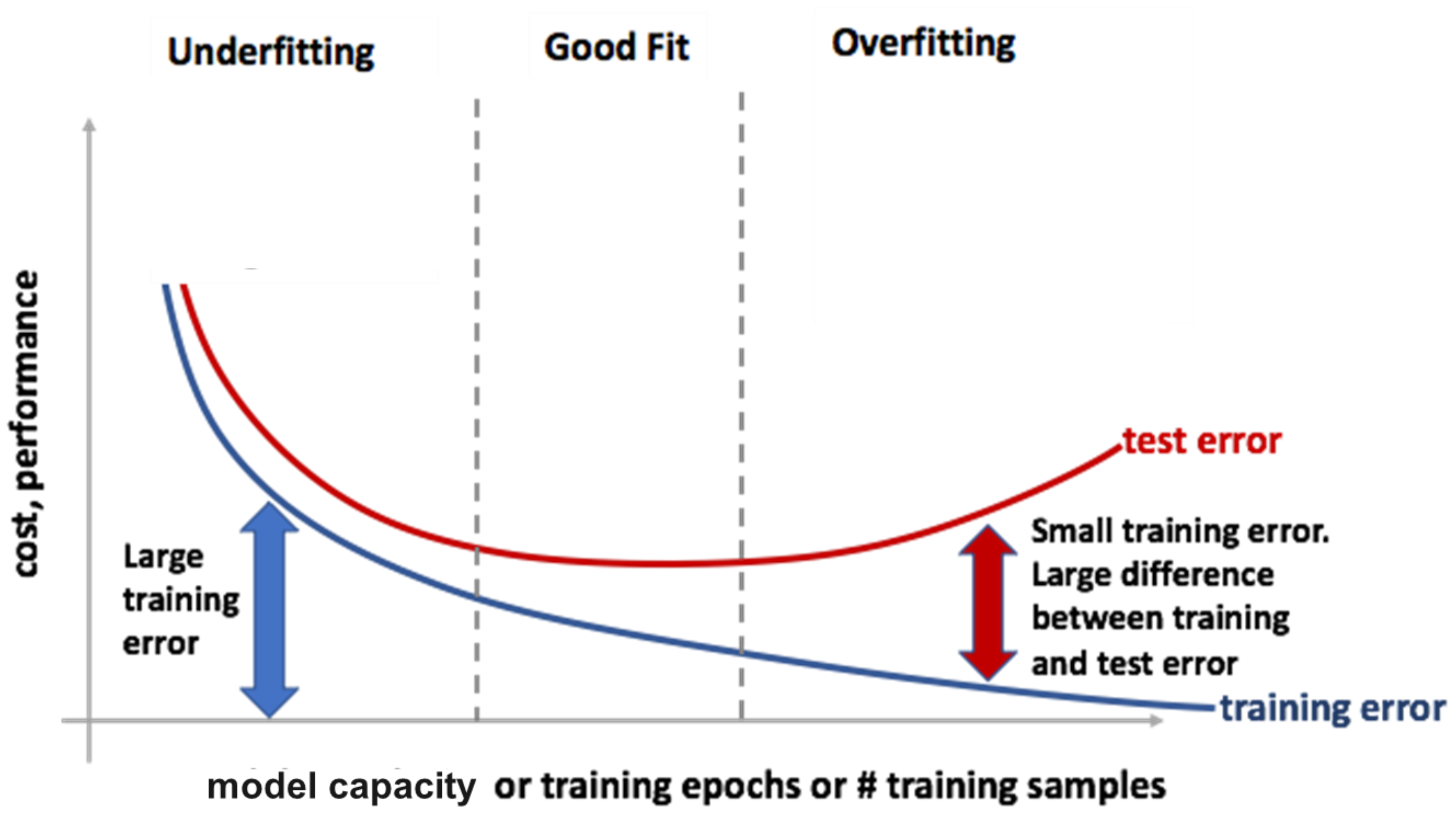
Overfitting can occur when

- the number of parameters of the model is too large, i.e. the model is too “flexible”
- the training set is too small in comparison with the dimensionality of the input data (which introduces sampling noise)
- the training set is too noisy

How to Examine Overfitting?

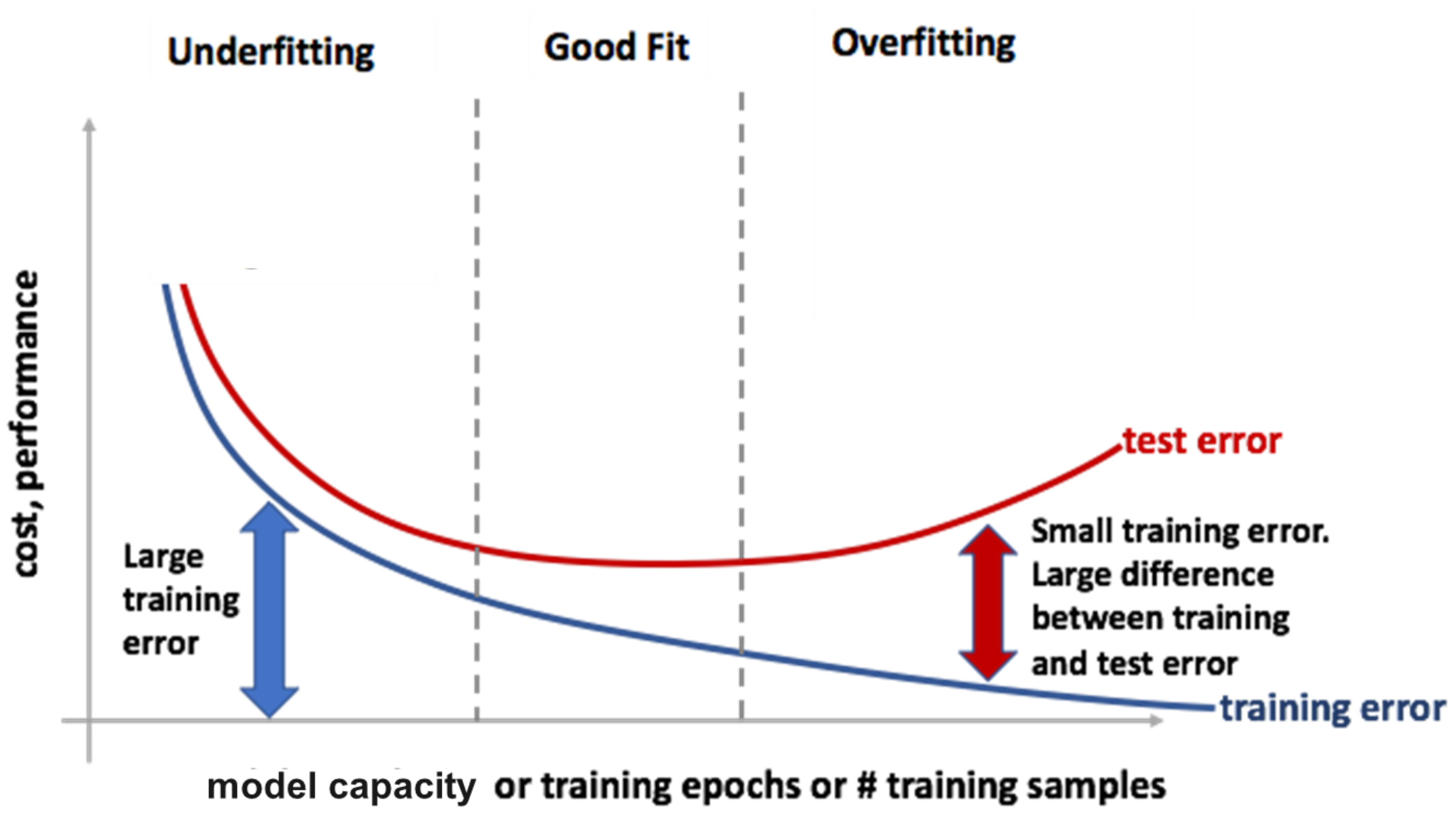


How to Examine Overfitting?


Underfitting
Good Fit
Overfitting
cost, performance
Large training error
test error
Small training error. Large difference between training and test error
training error
model capacity or training epochs or # training samples

- for increasing model complexities
- for increasing training set sizes
- for increasing number of training epochs

How to Examine Overfitting?


Underfitting
Good Fit
Overfitting
cost, performance
Large training error
test error
Small training error. Large difference between training and test error
training error
model capacity or training epochs or # training samples

A machine learning algorithm to perform well should be able to make

1. training error (“bias error”) small
2. gap between training and test error (“variance error”) small
3. bias and variance error of comparable magnitude