MASTER OF SCIENCE
IN ENGINEERING

Week 3:
Shallow Networks

TSM_DeLearn

Jean Hennebert
Martin Melchior

Overview

Recap from Last Week

Multi-Classification with Softmax

Adding Hidden Layers

Activation Functions

Universal Function Representation

Overfitting

Recap from Last Week

Important Points to Resume from Last Week

- Learning as Optimisation Problem illustrated with MNIST
- Model :
  - Generalised Perceptron, Logistic Regression
- Cost based on notion of distance
  - Cross-Entropy (for classification problem)
  - Quadratic (for regression problems)
- Gradient descent for minimising the cost function
  - Generally applicable for smooth models and activation functions
  - Guaranteed to work for convex (bowl-like) cost functions
  - Learning rate drives the speed of convergence
- Optimisation schemes
  - Batch Gradient Descent (BGD)
  - Stochastic Gradient Descent (SGD)
  - Mini-Batch Gradient Descent (MBGD)

Gradient Descent Update Rules and Cost Function

Cost (Cross-Entropy, Mean Square Error)

$$J_{\mathrm{CE}}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\log\left(p(y^{(i)}|\mathbf{x}^{(i)},\theta)\right)$$

$$J_{MSE}(\theta) = \frac{1}{2m}\sum_{i=1}^{m}\left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\right)^2$$

Gradient Descent

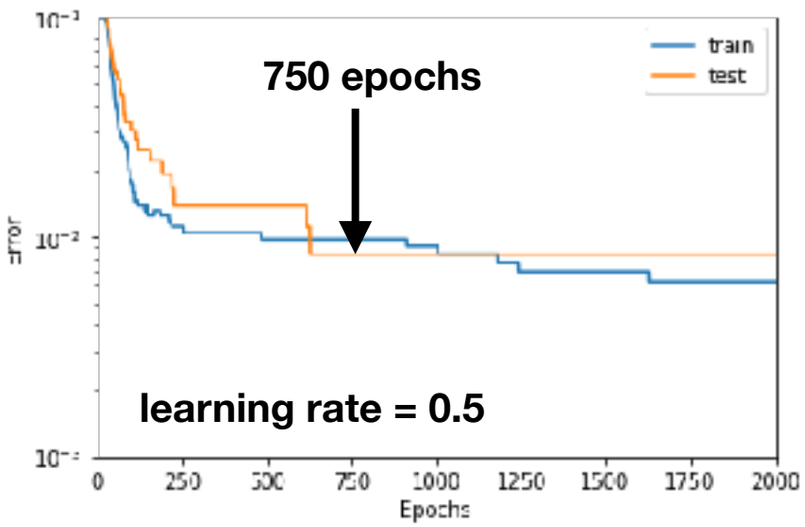
$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_\theta J(\theta_t)$$

Gradient Descent for Perceptron and Cross Entropy Cost

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} - \alpha\sum_{j=1}^{b}\left(h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)}\right)\mathbf{x}^{(i_j)}\\ b &\leftarrow b - \alpha\sum_{j=1}^{b}\left(h_\theta(\mathbf{x}^{(i_j)}) - y^{(i_j)}\right)\end{aligned}$$

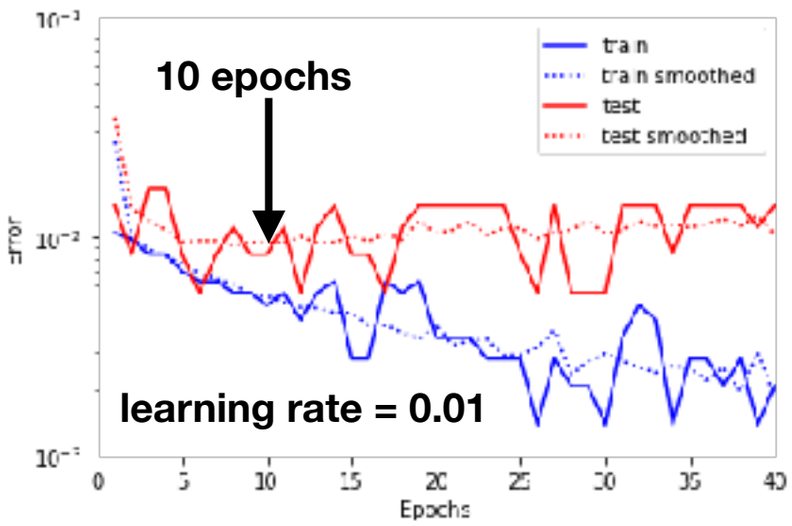
Comparison of Gradient Descent Schemes

BGD



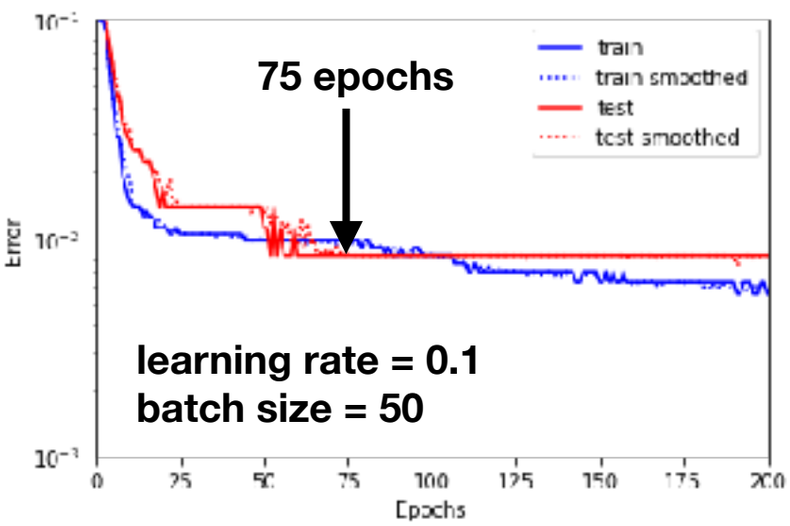
- Smooth, strictly decreasing curves.
- Cost curves strictly decreasing. Can reach minimum.
- Slow for large m. Many epochs needed.
- Learning rate can be large(r).
- No out-of-core support - all data in RAM.
- Easy to parallelise.

SGD



- Wiggling curves, need smoothing.
- Cost curve not necessarily decreasing. Wiggles around minimum.
- Fast for large m. Only few epochs needed.
- Learning rate must be small(er),
- Out-of-core support - not all data to be kept in RAM of a single machine.
- Not easy to parallelise.

MBGD



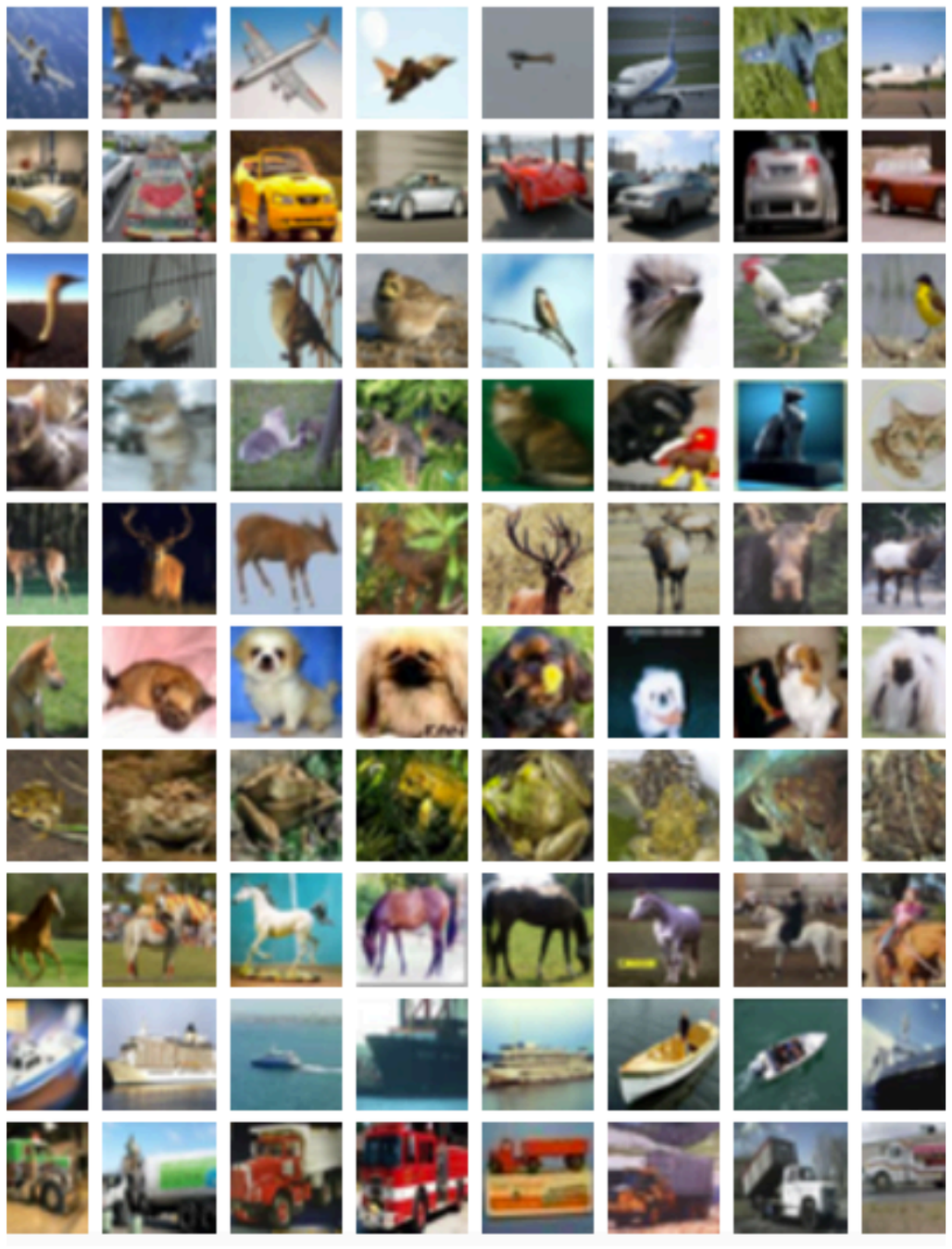
- Slightly wiggling curves.
- Cost curve typically, but not necessarily decreasing. Wiggles slightly around minimum.
- Fast for large m. Much less epochs than BGD but more than SGD needed.
- Medium learning rate.
- Out-of-core support - not all data to be kept in RAM of a single machine.
- Easy to parallelise.

see also http://www.holehouse.org/mlclass/17_Large_Scale_Machine_Learning.html

Plan for this week

- Learn more ingredients needed for general multi-class classification and regression tasks (softmax, other types of activation functions).
- Progress to study richer models by adding hidden layers and analyse the performance achieved for MNIST.
- Understand the result that we can represent almost any function with shallow networks (Universal Representation Theorem). Thus: Do we need deep architectures at all?
- Explain overfitting and generalisation error. Introduce the tools to study and optimise them: Model selection and evaluation (performance measures).

Softmax for Multi-Class



Binary Classification (recap)

- Class labels: 1 — ‘yes’, 0 — ‘no’
- Model:

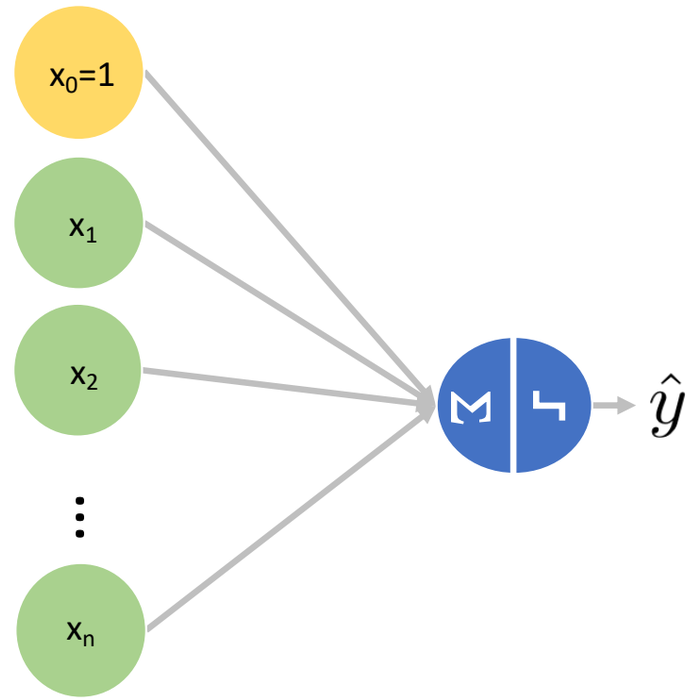
$$h_\theta(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b), \quad \sigma(z) = \frac{1}{1 + e^{-z}}$$

- Parameters: $(\mathbf{w}, b)$
- Predicted class:

$$\hat{y}(\mathbf{x}) = round(h_\theta(\mathbf{x})) \in \{0, 1\}$$

- Probabilistic interpretation:

$$\begin{aligned} p(y = 1|\mathbf{x}, \theta) &= h_\theta(\mathbf{x}) \\ p(y = 0|\mathbf{x}, \theta) &= 1 - h_\theta(\mathbf{x}) \end{aligned}$$



Binary Classification Extended to Multi-Class

See Exercise 4 in PW02

- Class labels (*K* classes) : $0 \leq l < K$
- Model consisting of *K* independent binary classifications:

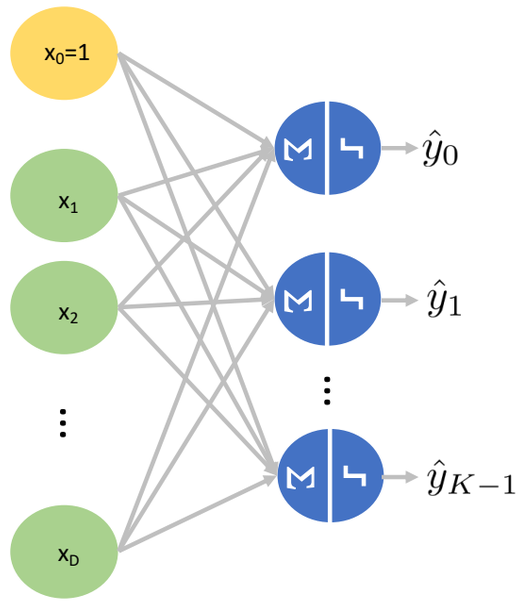
$$h_{\theta_l}(\mathbf{x}) = \sigma(\mathbf{w}_l \cdot \mathbf{x} + b_l)$$

- Parameters for *K* classes: $\theta_1 = (\mathbf{w}_1, b_1), \ldots, \theta_{K-1} = (\mathbf{w}_{K-1}, b_{K-1})$
- Predicted class:

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

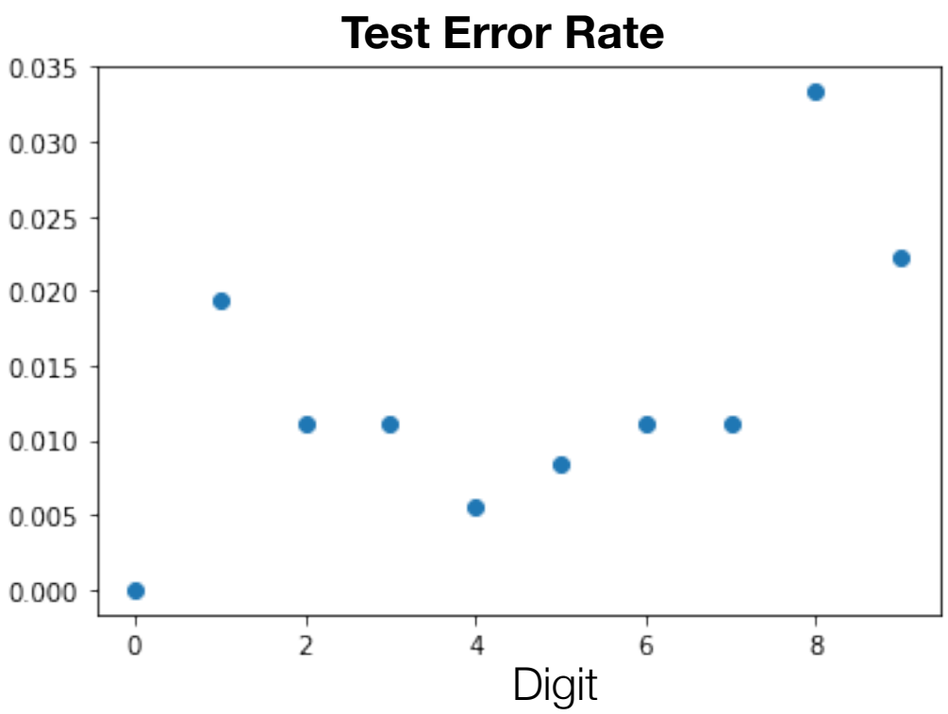
- Probabilistic interpretation: Only for each binary output independently.

$$\begin{aligned} p(y_k = k|\mathbf{x}, \theta) &= h_{\theta_k}(\mathbf{x}) \\ p(y_k \neq k|\mathbf{x}, \theta) &= 1 - h_{\theta_k}(\mathbf{x}) \end{aligned}$$



Model not trained to provide normed probabilities for all the classes — the system is not forced to decide between one of the classes!

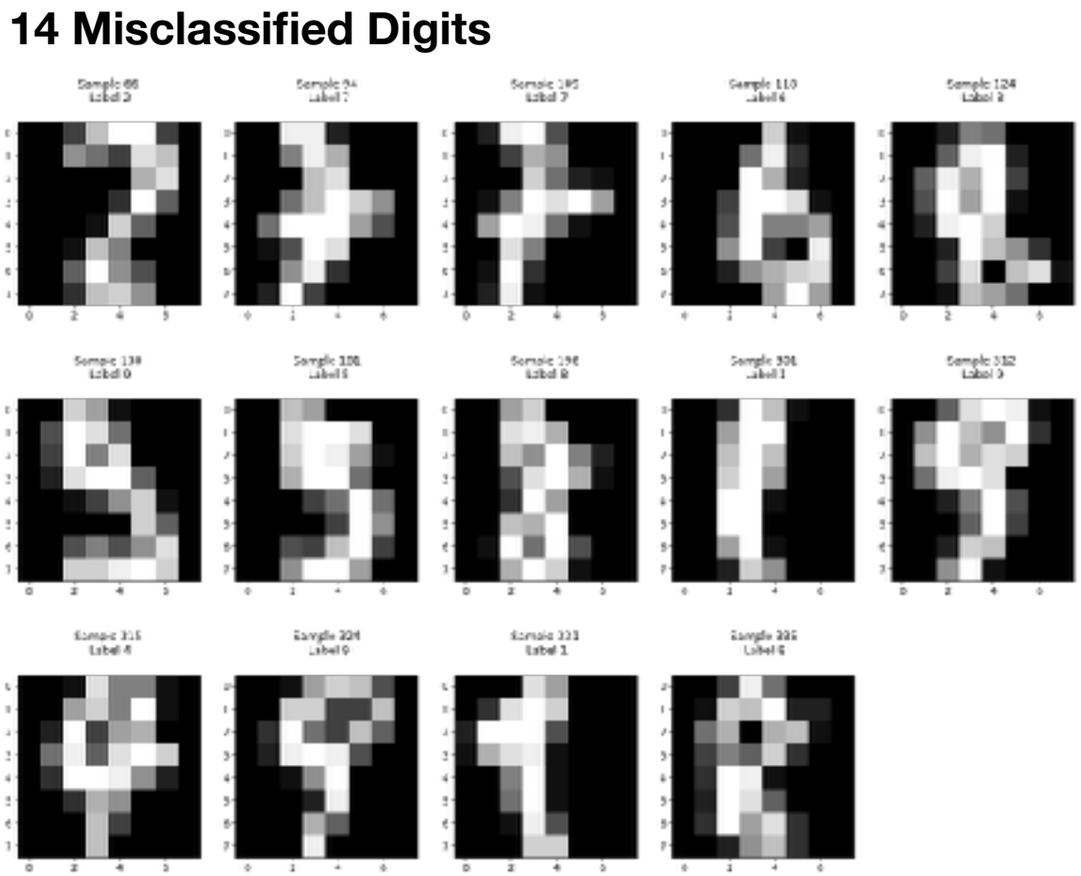
Results for MNIST Lightweight

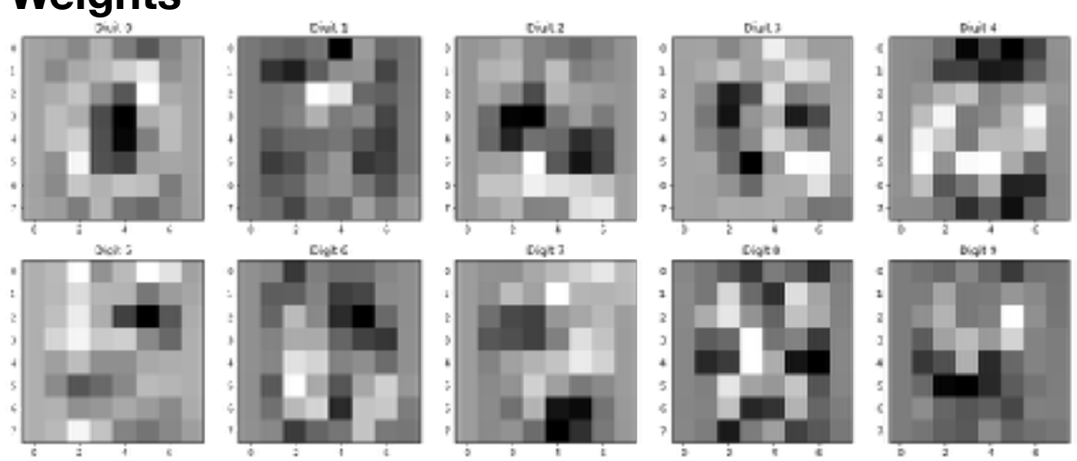


Digits Classification

As prediction pick the digit with the highest probability (resulting by training each digit *independently* in a binary classifications).

Resulting Test Error Rate: 3.89%





Multi-Classification with Softmax

- Extend the model to directly return normed probabilities:

$$h_{\theta,l}(\mathbf{x}) = \frac{\exp(z_l)}{\sum_{j=1}^{k} \exp(z_j)} \text{ where } z_j = \mathbf{w}_j \cdot \mathbf{x} + b_j$$

  i.e. yet *K* different outputs — but which are dependent ($0 \leq l < K$).

- The function

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

  is called *softmax.* It peaks at the largest z_l and smoothly approximates

$$\max\{z_1, \ldots, z_{K-1}\}$$

  if one element is much larger than all the others (e.g. $z_1 \gg z_k, 2 \leq l < K$).

- Parameters: $\theta = ((\mathbf{w}_1, b_1), \ldots, (\mathbf{w}_{K-1}, b_{K-1})) = (\mathbf{W}, \mathbf{b})$

Cost Function with Softmax

- Predicted class:

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

- Probabilistic interpretation: Provides a probability distribution for the different classes

$$p(y = l|\mathbf{x}, \theta) = h_{\theta,l}(\mathbf{x})$$

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

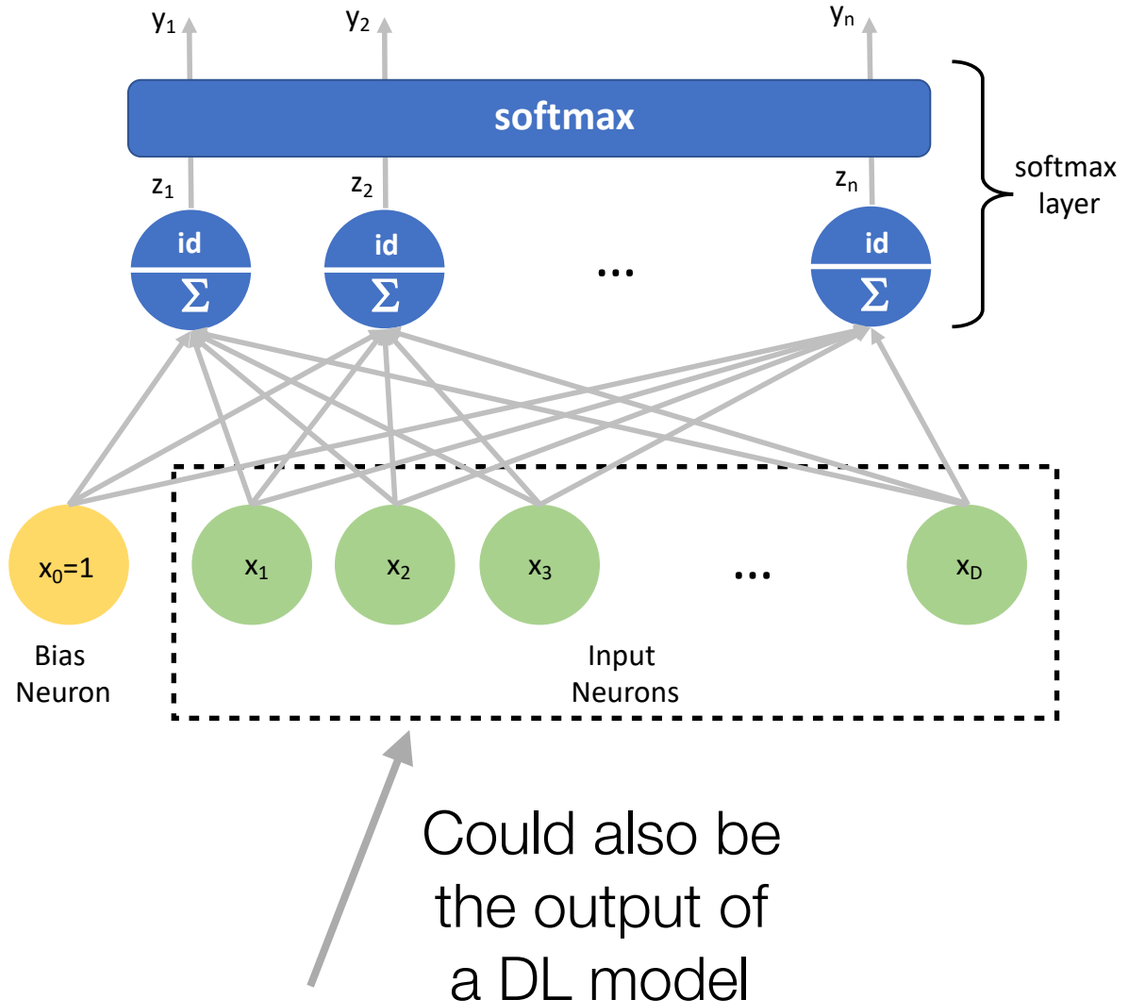
$$\begin{aligned} J_{\mathrm{CE}}(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}\log(p(y^{(i)}|\mathbf{x}^{(i)}, \theta)) \\ &= -\frac{1}{m}\sum_{i=1}^{m}\log(h_{\theta,y^{(i)}}(\mathbf{x}^{(i)}))) \end{aligned}$$

Softmax Layer

- The mapping provided by the softmax function

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

  is a specific form of neuron with *m* inputs and an *m* outputs.
- The inputs $z_1,\ldots, z_n$ are computed by a weighted sum of $x_1,\ldots, x_D$ plus bias.
- The outputs $y_1,\ldots, y_n$ can be interpreted as (normed) probabilities associated with the different classes.
- These ingredients form a *softmax layer.* It is typically used as final layer in classification problems - also with deep neural nets.



Calculation of the Gradient

With

$$\frac{\partial h_{\theta,l}(\mathbf{x})}{\partial z_k} = \delta_{l,k} h_{\theta,l}(\mathbf{x}) - h_{\theta,l}(\mathbf{x}) h_{\theta,k}(\mathbf{x})$$

we obtain

$$\begin{aligned}
\frac{\partial}{\partial \mathbf{w}_j} J_{\mathrm{CE}}(\theta) &= -\frac{1}{m} \sum_{i=1}^{m} \frac{\partial}{\partial \mathbf{w}_j} \log\left(h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)\right) \\
&= -\frac{1}{m} \sum_{i=1}^{m} \frac{1}{h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)} \frac{\partial h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)}{\partial \mathbf{w}_j} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \frac{1}{h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)} \frac{\partial h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)}{\partial z_j} \frac{\partial z_j}{\partial \mathbf{w}_j} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \frac{1}{h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)} \left(\delta_{j,y^{(i)}} h_{\theta,j}\left(\mathbf{x}^{(i)}\right) - h_{\theta,j}\left(\mathbf{x}^{(i)}\right) h_{\theta,y^{(i)}}\left(\mathbf{x}^{(i)}\right)\right) \frac{\partial z_j}{\partial \mathbf{w}_j} \\
&= -\frac{1}{m} \sum_{i=1}^{m} \left(\delta_{j,y^{(i)}} - h_{\theta,j}\left(\mathbf{x}^{(i)}\right)\right) \mathbf{x}^{(i)}
\end{aligned}$$

where we have used the Kronecker symbol

$$\delta_{k,j} = \begin{cases} 1 & (k = j) \\ 0 & (k \neq j) \end{cases}$$

Training with Softmax

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

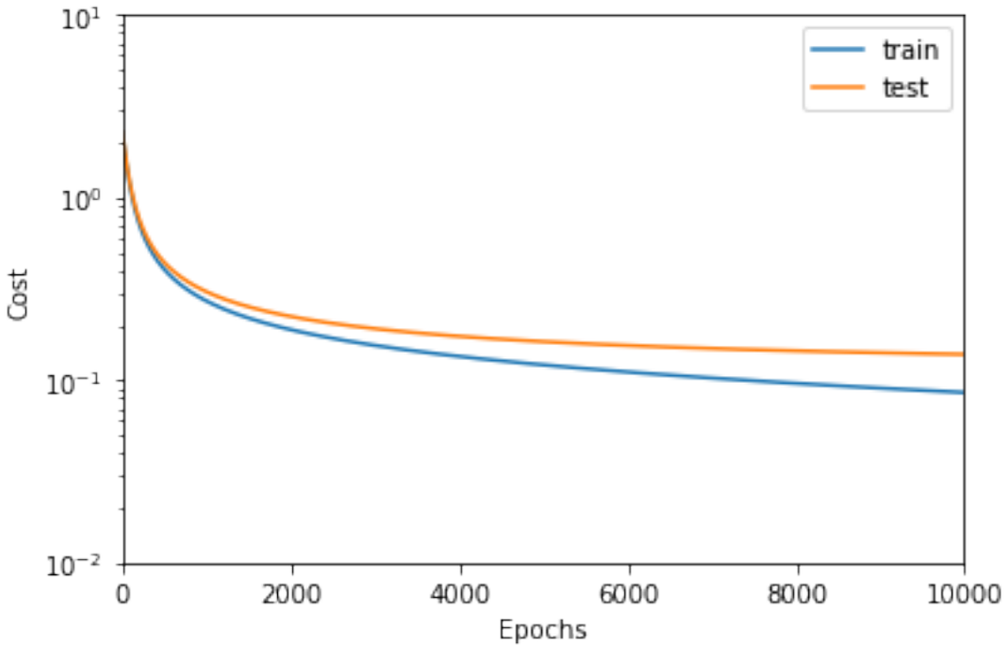
$$\begin{aligned}\frac{\partial}{\partial \mathbf{w}_j} J_{\mathrm{CE}}(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{j,y^{(i)}} - h_{\theta,j}\left(\mathbf{x}^{(i)}\right)\right)\mathbf{x}^{(i)} \\ \frac{\partial}{\partial b_j} J_{\mathrm{CE}}(\theta) &= -\frac{1}{m}\sum_{i=1}^{m}\left(\delta_{j,y^{(i)}} - h_{\theta,j}\left(\mathbf{x}^{(i)}\right)\right)\end{aligned}$$

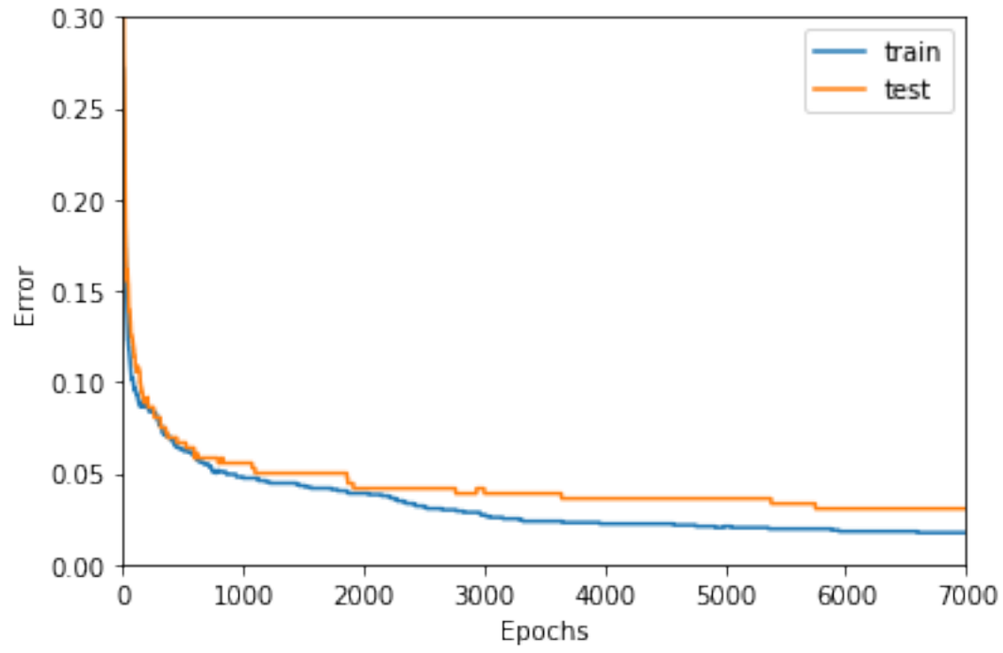
Update rules for the weights and biases of the softmax layer:

$$\mathbf{w}_j \leftarrow \mathbf{w}_j - \alpha\frac{\partial}{\partial \mathbf{w}_j} J_{\mathrm{CE}}(\theta)\ ,\ b_j \leftarrow b_j - \alpha\frac{\partial}{\partial b_j} J_{\mathrm{CE}}(\theta)$$

These rules apply also to softmax layers as last layer of DL models.

Results for Lightweight MNIST



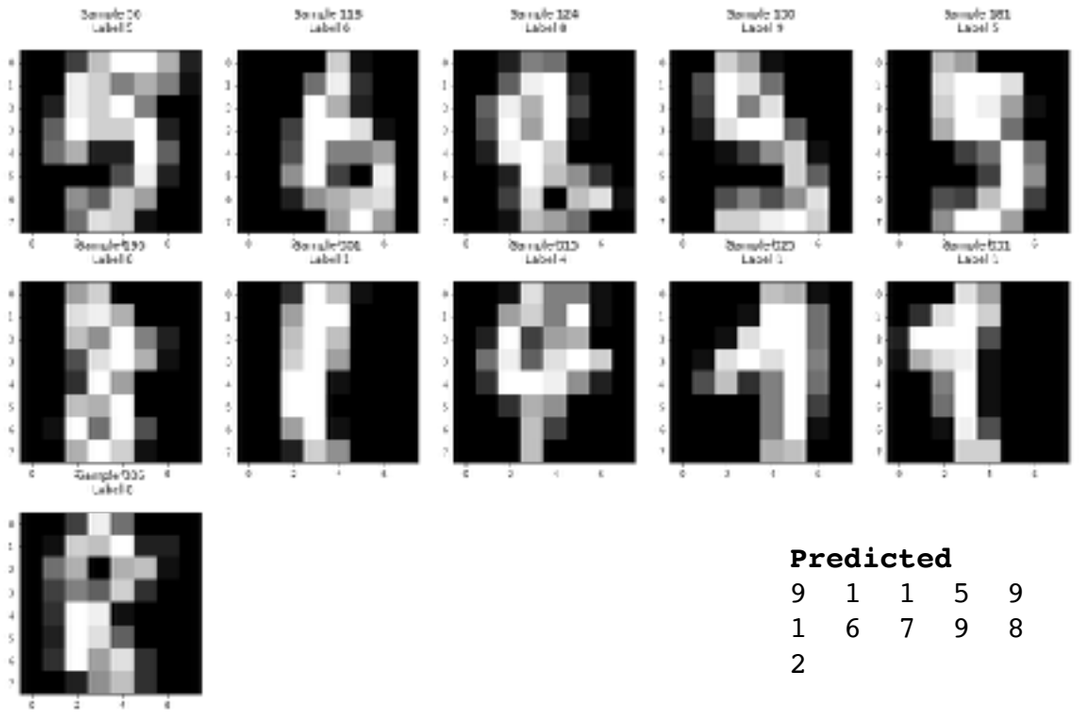


Learning Rate: 0.1

Train Error: 1.32%

Test Error: 3.06%

11 incorrectly classified digits

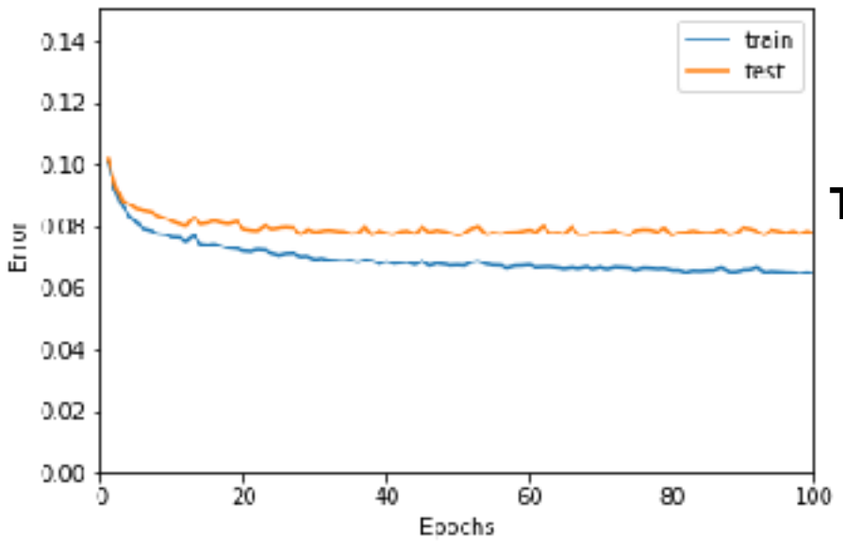


Predicted

9 1 1 5 9
1 6 7 9 8
2

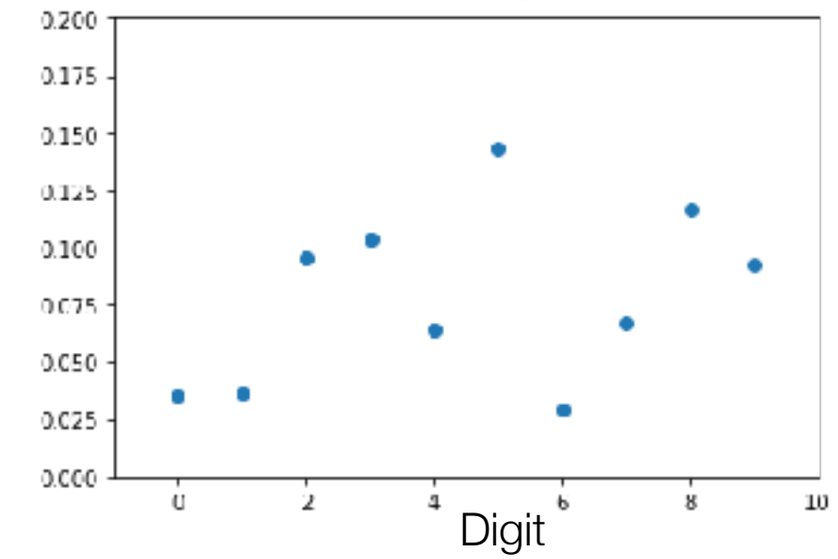
Results for *Original* MNIST: Using MBGD

Error Rate

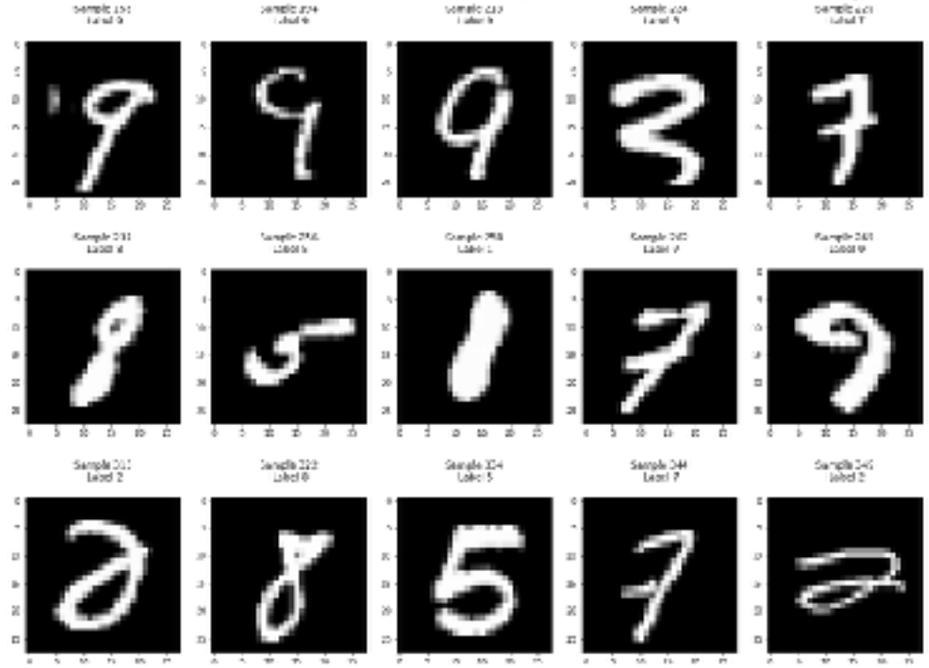


Test Error
7.76%

Error Rates per Digit



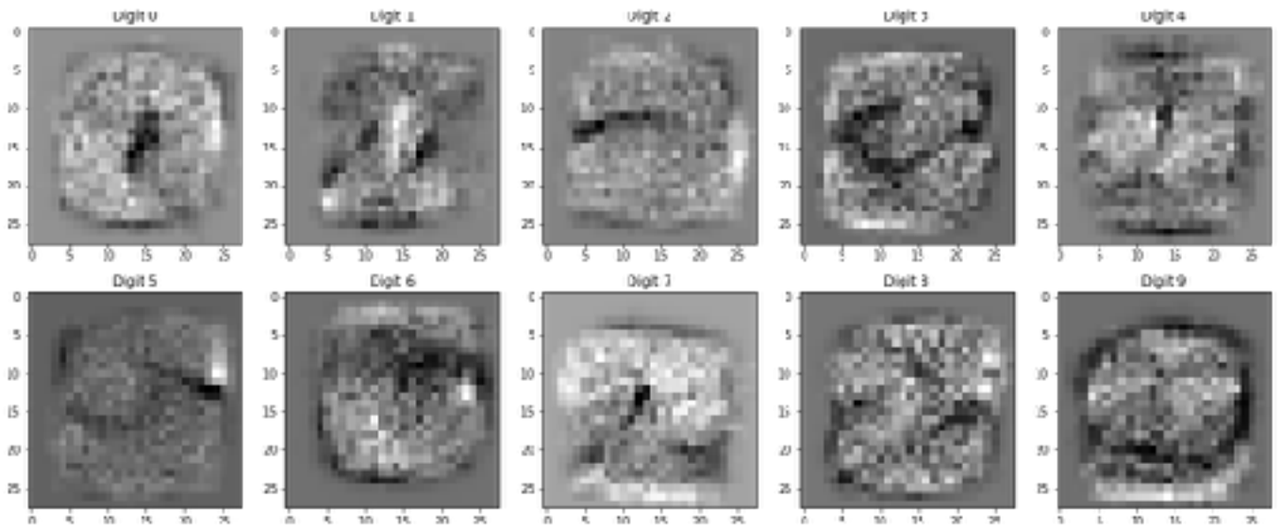
Samples of incorrectly classified digits



```
Ground Truth
9 9 9 3 7
8 5 1 7 9
0 2 8 5 7

Predicted
7 4 4 8 9
1 8 8 2 7
3 3 8 9 4
```

Weights for the different digits



(# epochs = 100, learning rate = 0.6, batch size = 500, test data size = 10’000)

Results for *Original* MNIST

- MNIST (original dataset):
  - **Test Error: 7.76%**
    (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)

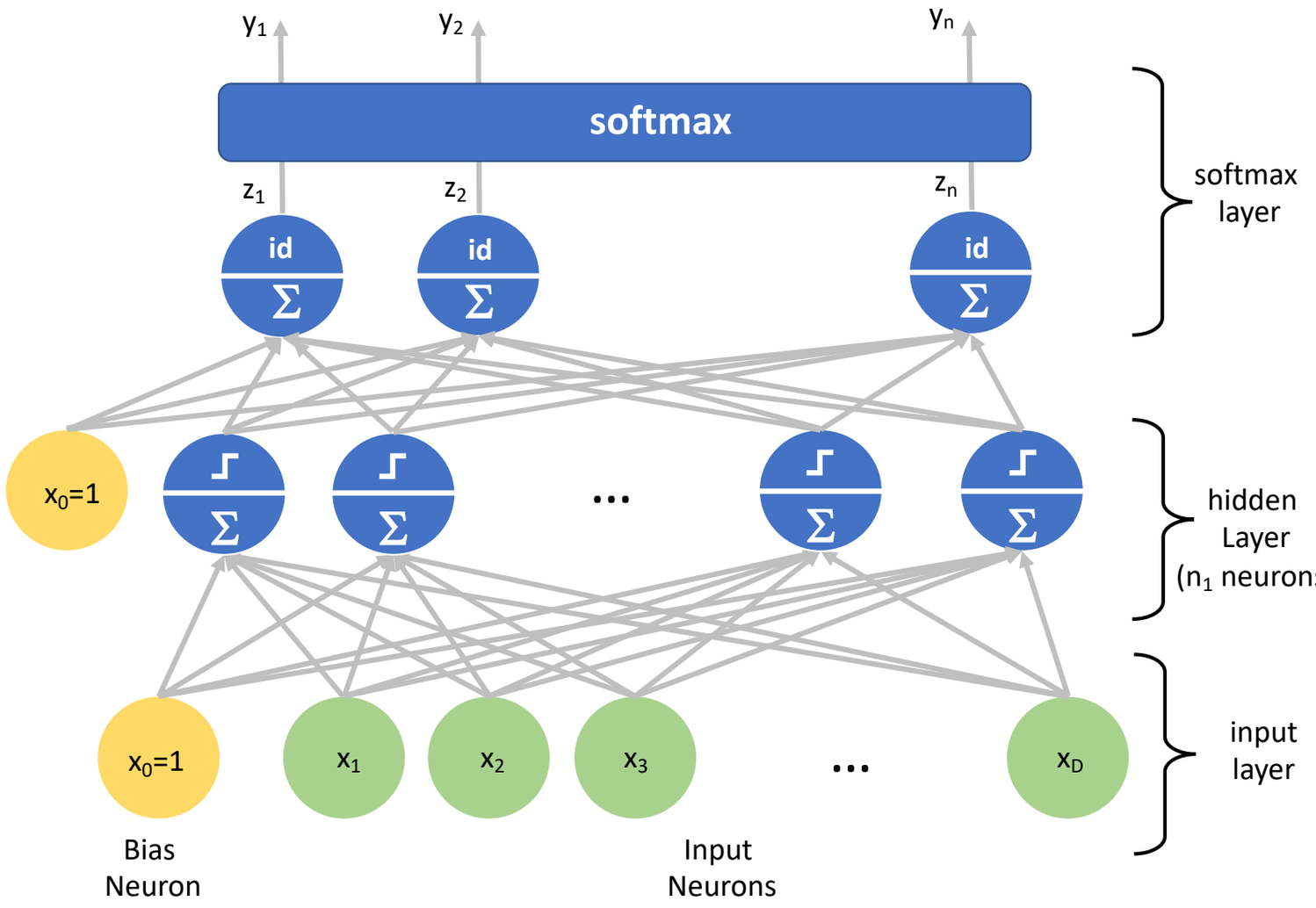
- To achieve this performance, larger models with higher representational capacity are needed
  - (Many) more model parameters
  - Danger of overfitting

Improving the Capacity of Models

Adding Hidden Layers
Role of the Activation Function

Adding One Hidden Layer

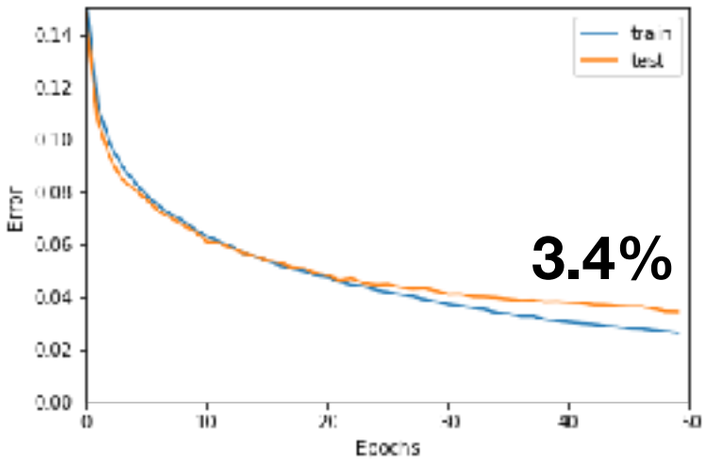
- One additional layer with n_1 neurons, sigmoid activation function.
- "Fully connected": Weights for each connection between
  - input and hidden layer neurons
  - hidden layer and softmax neurons
- Bias for each connection to a bias neuron.

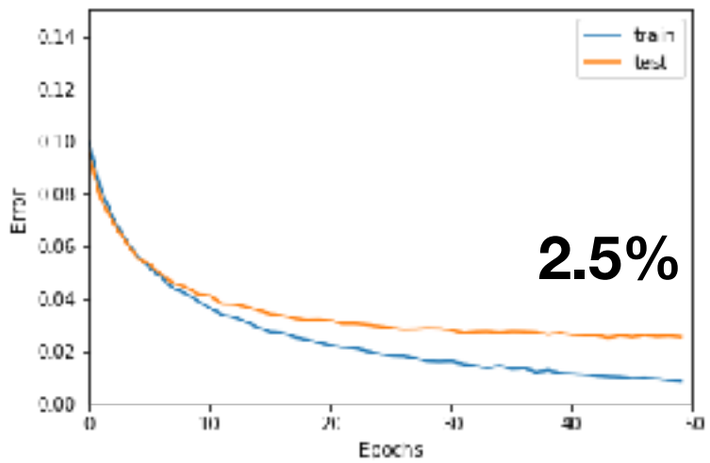


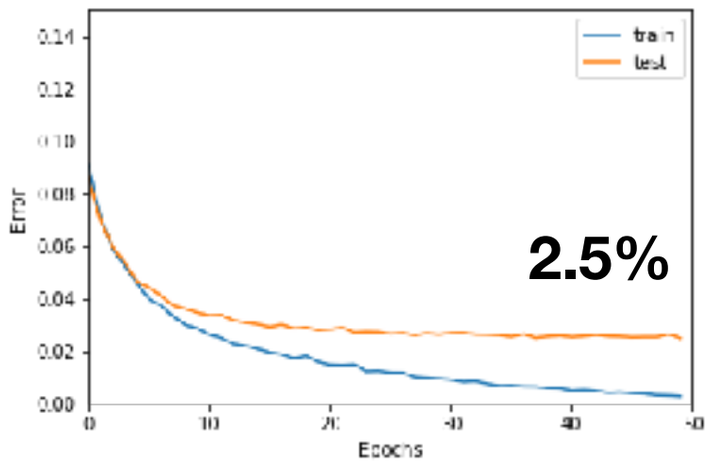
Results for Original MNIST

learning rate 0.1 | **learning rate 0.3** | **learning rate 0.5**

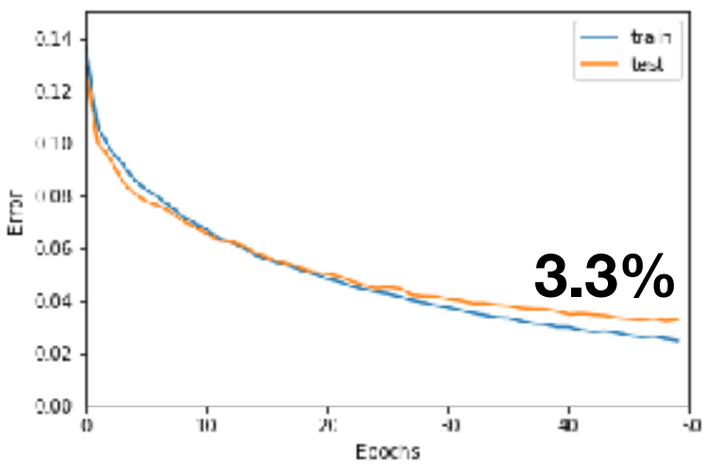
50 units

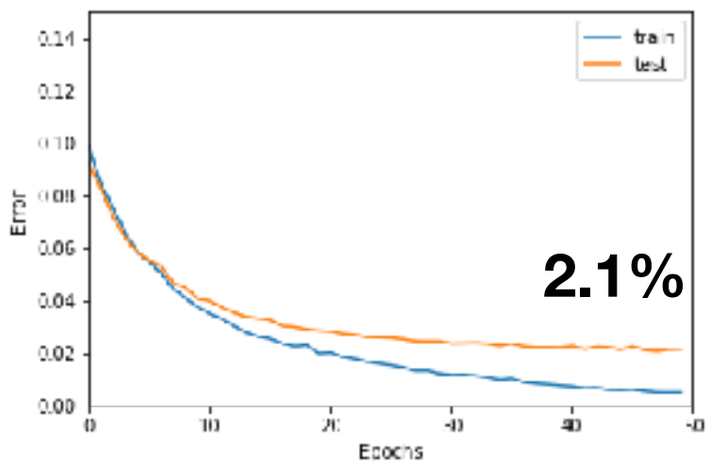


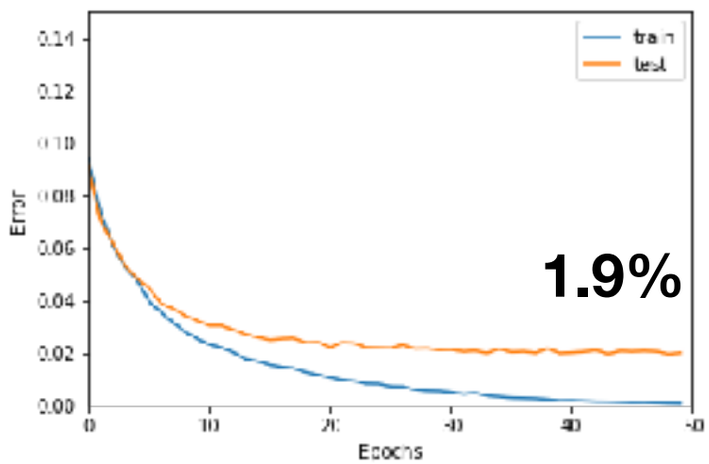




150 units







Interpreting Learning Curves

Discuss in groups whether the plots depicted on the previous slide are consistent.

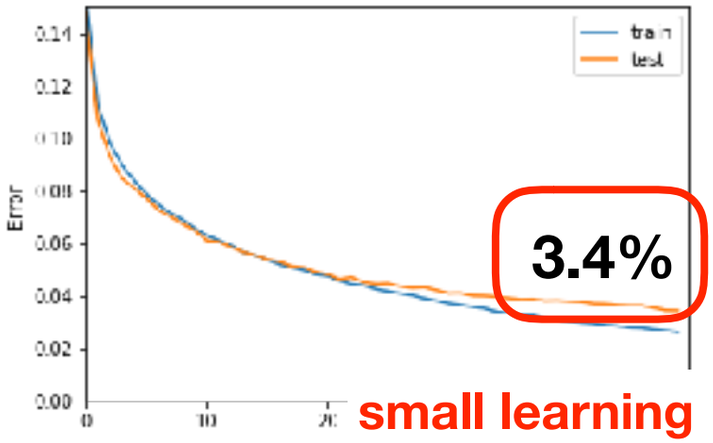
Interpret the change in the curve and the final test error rate

- with changing learning rate (left to right).
- with changing number of neurons (upper/ lower).

Results for Original MNIST

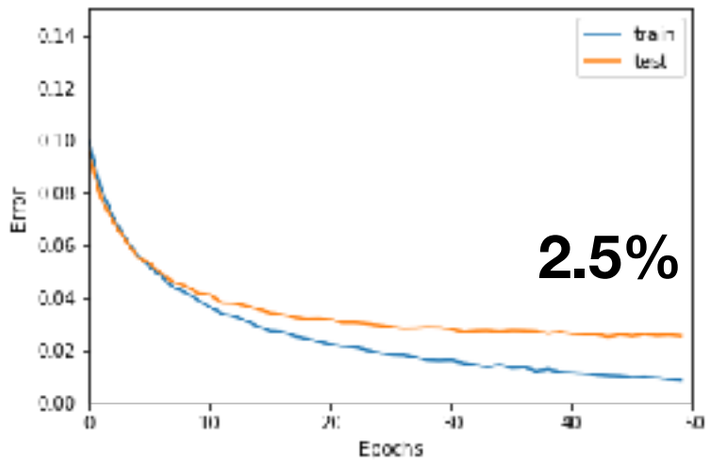
learning rate 0.1 | **learning rate 0.3** | **learning rate 0.5**

50 units

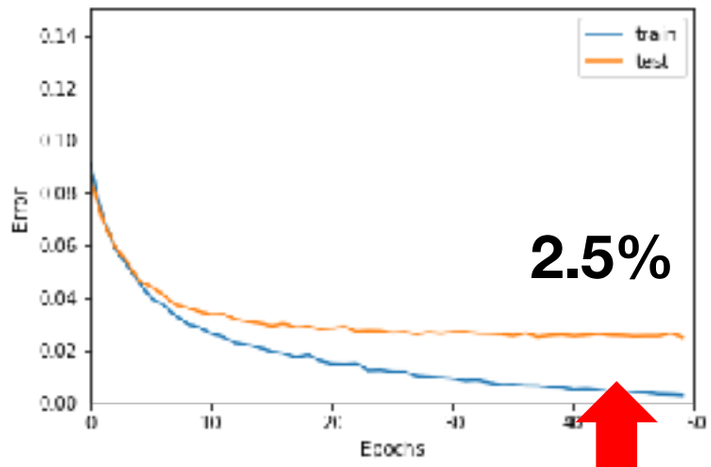


3.4%

small learning rate — still descending!

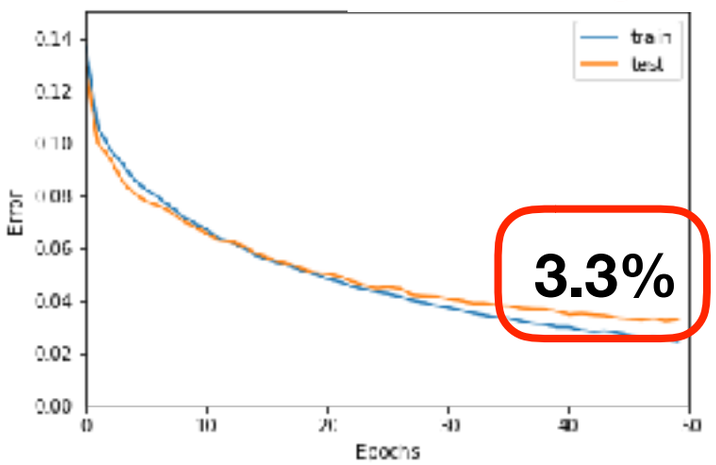


2.5%



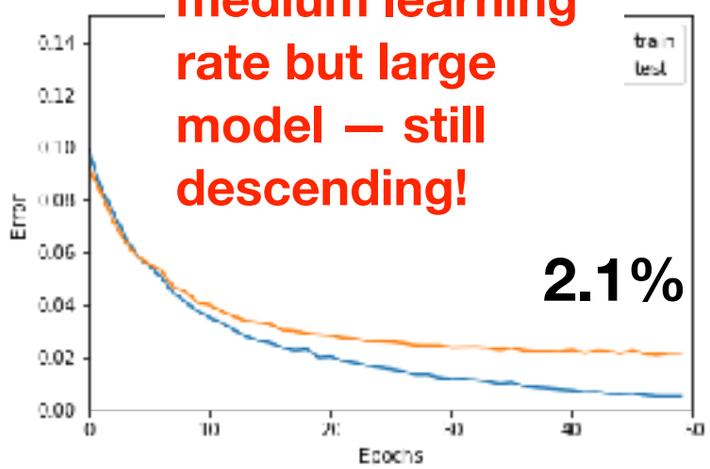
2.5%

150 units



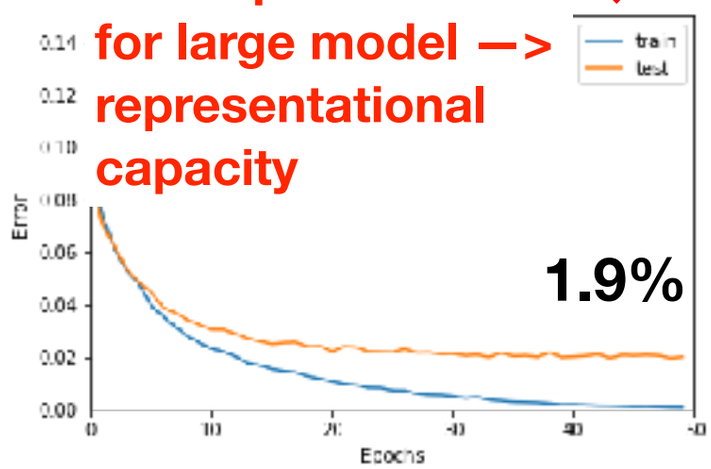
3.3%

medium learning rate but large model — still descending!



2.1%

better performance for large model —> representational capacity

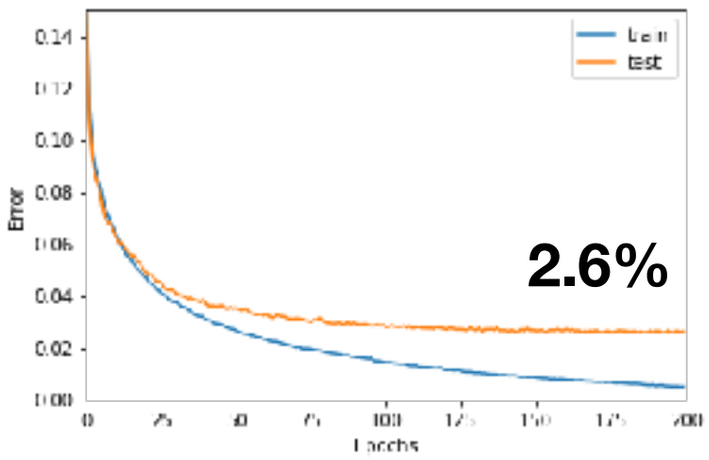


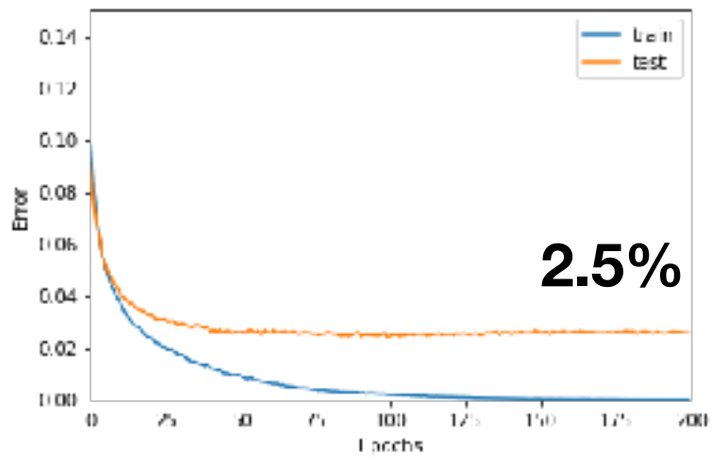
1.9%

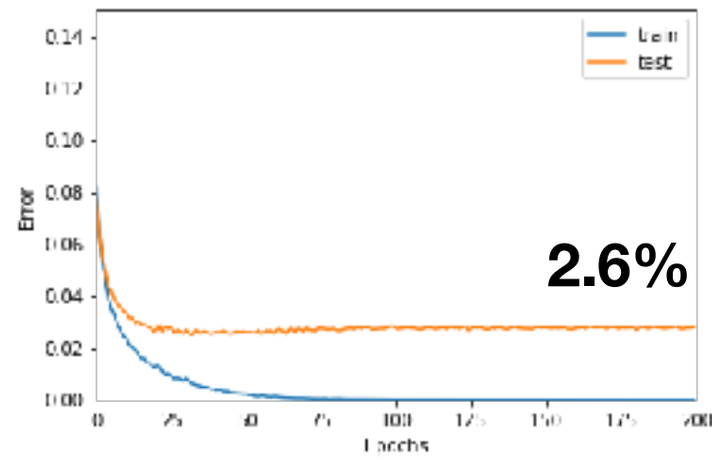
Results for Original MNIST

learning rate 0.1 | **learning rate 0.3** | **learning rate 0.5**

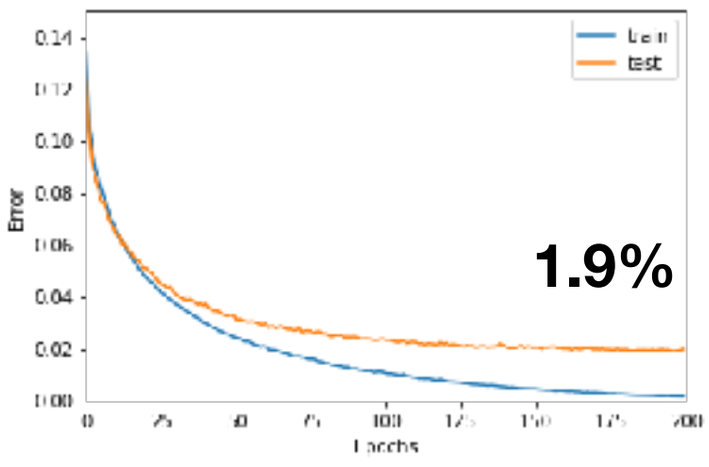
50 units

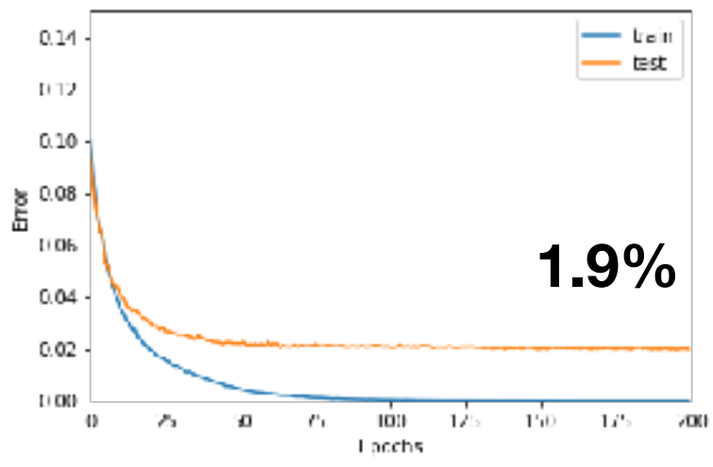


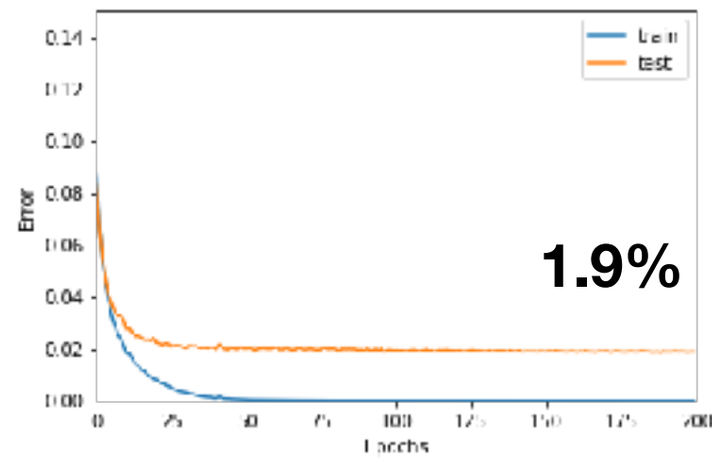




150 units



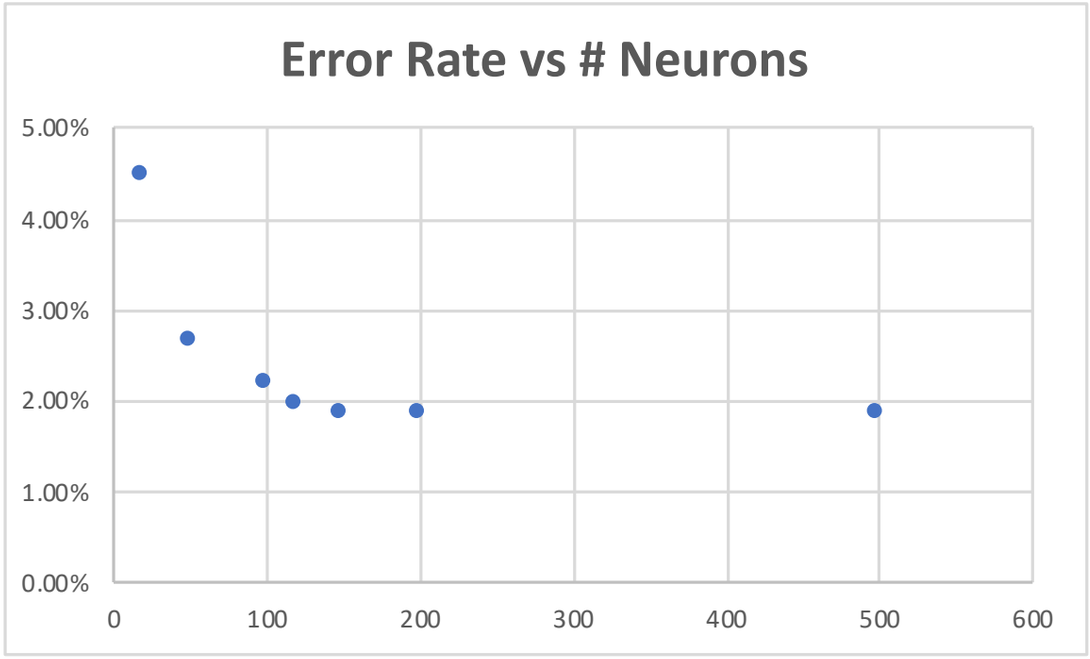




Improvements by Adding One Hidden Layer

- We can reduce the error rate from 7.8% down to ~1.85% by only adding one hidden layer (with sufficiently many neurons).

- By increasing number of neurons in the hidden layer the performance becomes better - reaching at ~120 neurons a plateau of ~2% error rate (minimum at 200 neurons with 1.85% error rate).



- Plain mini-brach gradient descent (no enhanced optimisation schemes), executed with tensorflow.
- Learning rate = 0.5
- Batch size = 100
- 200 Epochs

Open Points

- What improvements are necessary to further boost the performance from ~1.85% down to ~0.2% test error?
  - Better strategy for choosing/evolving the learning rate and number of epochs —> hyper-parameter tuning in week 4
  - Improved optimisation schemes see week 6
  - Convolutional Neural Nets —> week 8 and following
- Just adding more hidden layers? —> Performance cannot be improved much for MNIST.
  - Possible explanation: Correlation between pixels is captured already with a single layer.
- How do the formulas look like when applying gradient descent to networks with hidden layer(s)?
  - See backprop in week 4

Role of Activation Function

Non-Linearities Crucial for Sufficient Representational Capacity

- Single neuron in an MLP

$$h(\mathbf{x}) = f(\mathbf{w} \cdot \mathbf{x} + b)$$

  - Affine transformation of the input (linear + bias)
  - Activation function, in most cases non-linear

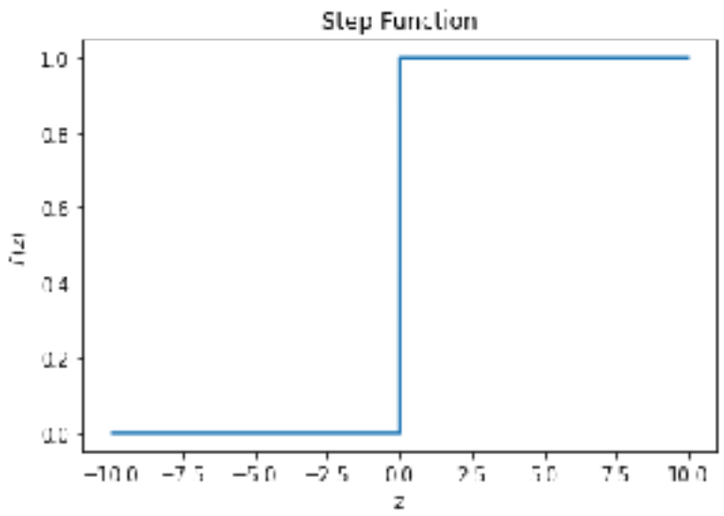
- Non-linearities in the mapping between input and output of a neural network are crucial for gaining sufficient power for learning a task with sufficient accuracy.
- If the activation function is linear for all neurons in a neural network, the mapping function for the neural network is linear and the representational capacity very limited.

Robustness and Performance of Learning Algorithm

- The choice of activation functions has an impact on the robustness and performance of the learning algorithm — different activation functions have been introduced to improve the robustness and performance of the learning algorithm.

Activation Functions (1)

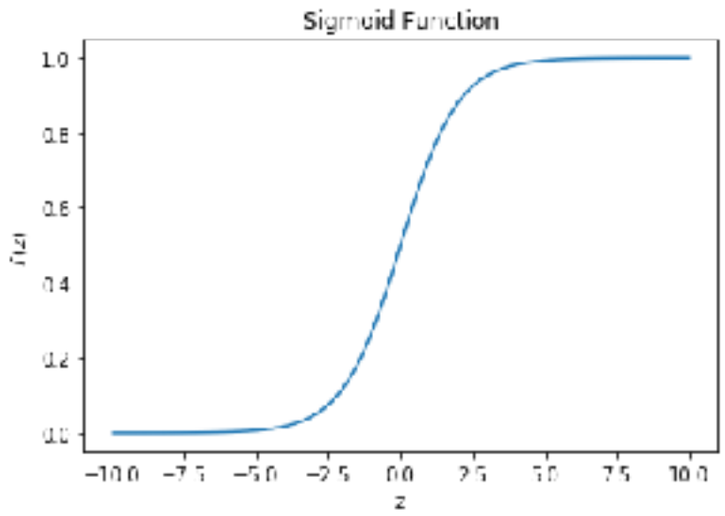
Heaviside



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

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

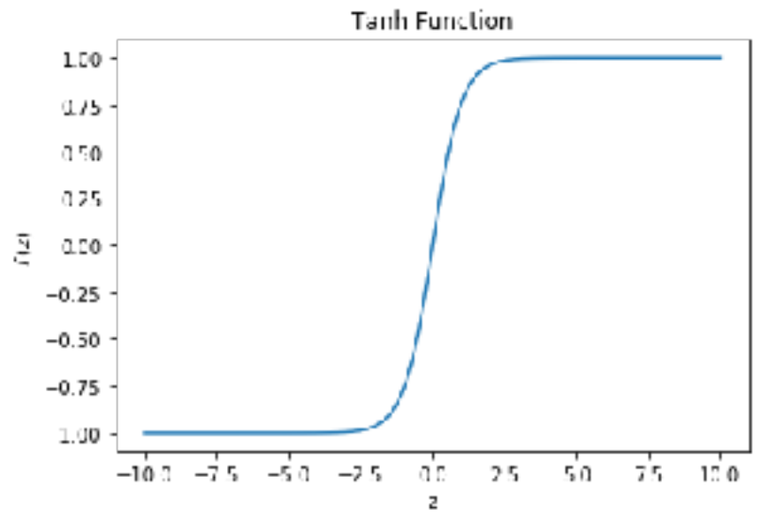
Sigmoid



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

Most commonly used in textbooks and in illustrative examples. In practice, typically in output layer in binary classification.
Smooth, i.e. gradient descent works.
But saturation regions leading to vanishing gradients. See Week 5.

Tanh



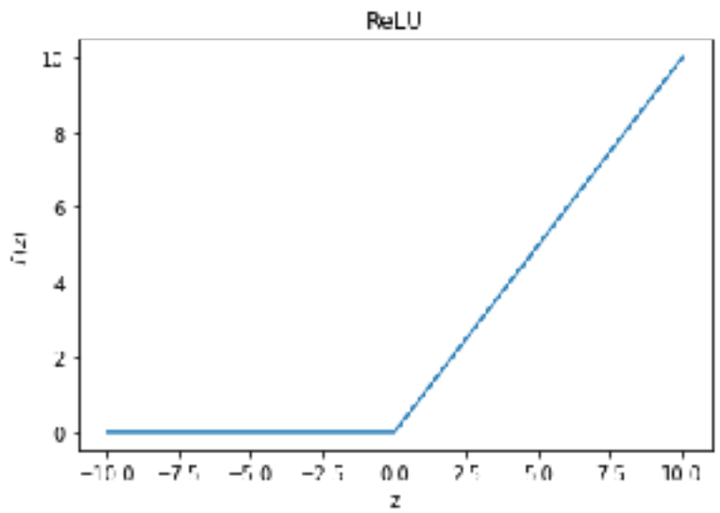
$$f(z) = \frac{\exp(z)-\exp(-z)}{\exp(z)+\exp(-z)}$$

Often preferred over sigmoid since the output is centred around 0.

Smooth, i.e. gradient descent works.
But saturation regions leading to vanishing gradients. See Week 5.

Activation Functions (2)

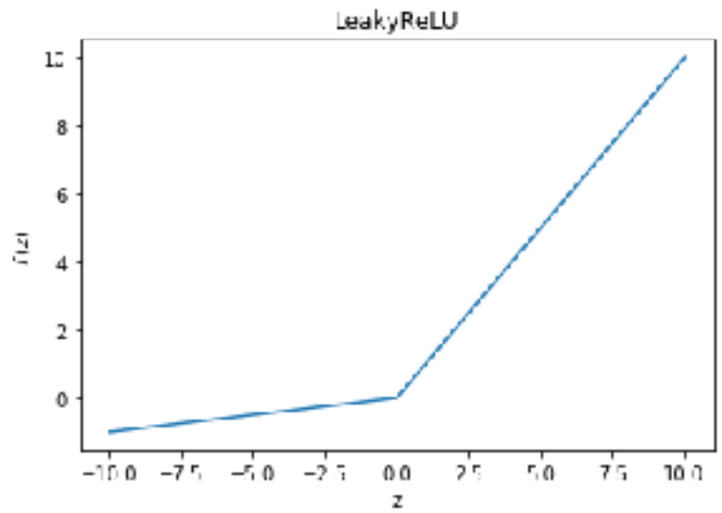
Recti-Linear Unit



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

Increasingly used as de facto standard.
Introduced to alleviate the vanishing gradient problem (see Week 5).
However, suffer from dying units problem for z<0 where the activation and the gradient is 0.

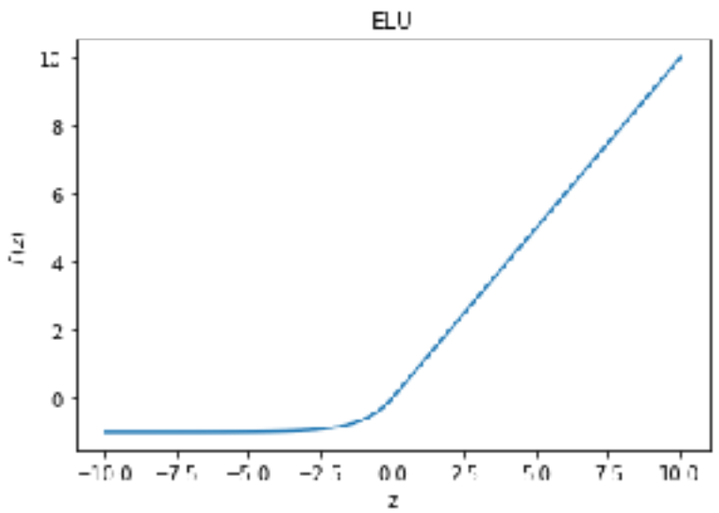
Leaky Recti-Linear Unit



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

Alleviates both, the vanishing gradient problem and the dying units problem (see Week 5).
Uses a small hyper-parameter
$0 < \alpha < 1$
which makes sure that the unit never dies (typical default: $\alpha = 0.01$)

Exponential Linear Unit

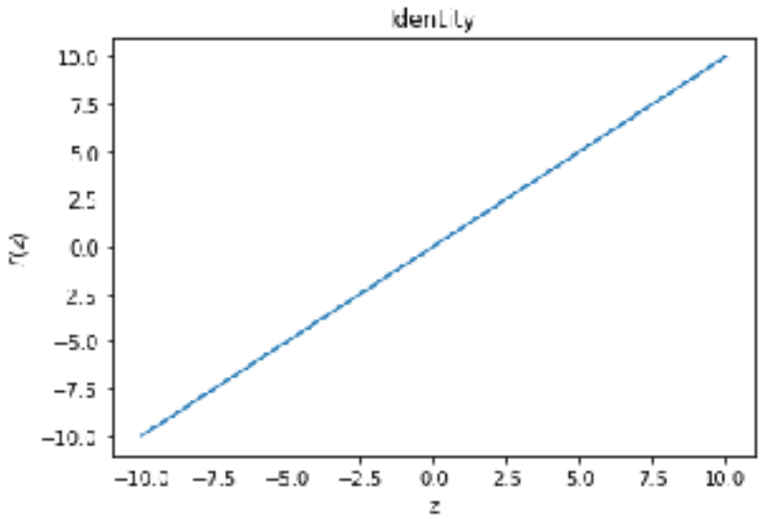


$$f(z) = \begin{cases} \alpha\left(\exp(z) - 1\right) & (z < 0) \\ z & (z \geq 0) \end{cases}$$

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

Activation Functions (3)

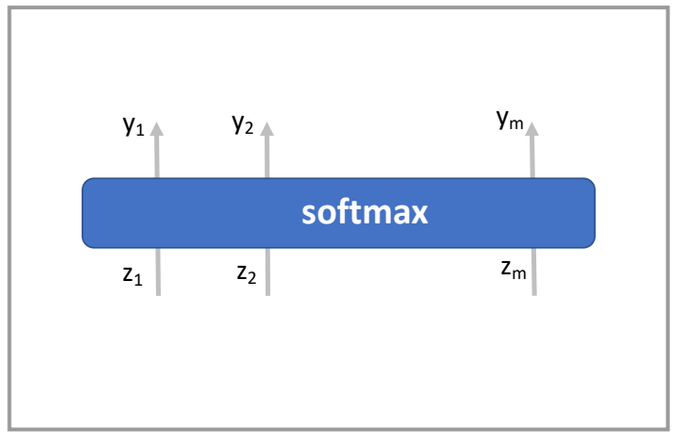
Identity



$$f(z) = z$$

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.

Universal Function Representation

copyright www.alamy.com

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:

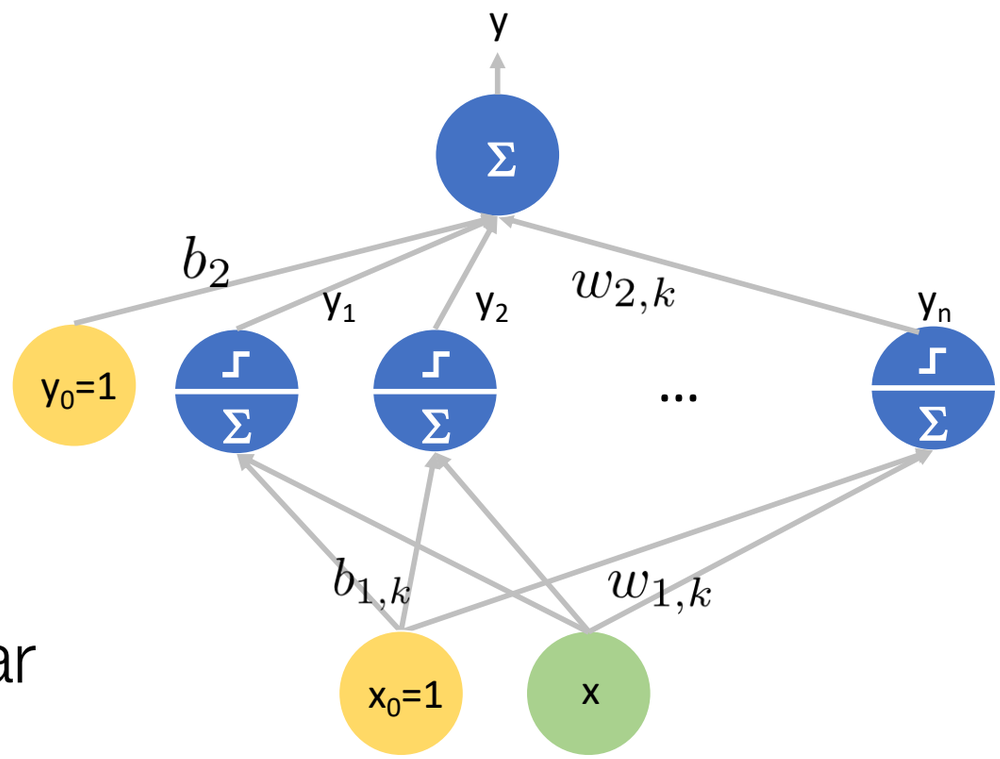
- Are all problems solved? What points are missing for a practical applications?
- It makes a statement about functions.
  But we typically start with data.
  What is the connection?

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:

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

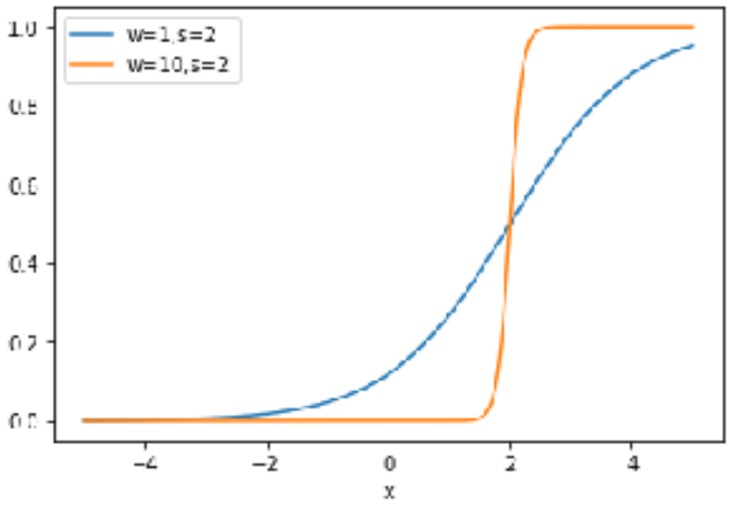
- The theorem states that we can approximate any quite general function if we choose suitable parameters and a suitable number of neuron *n*.

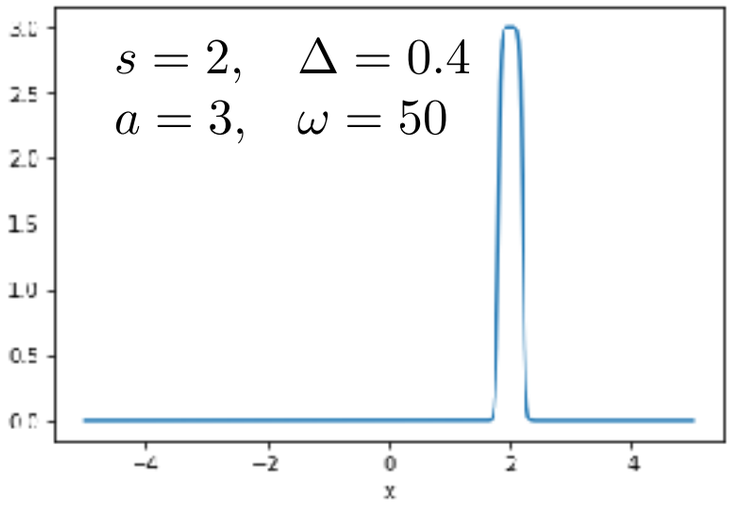


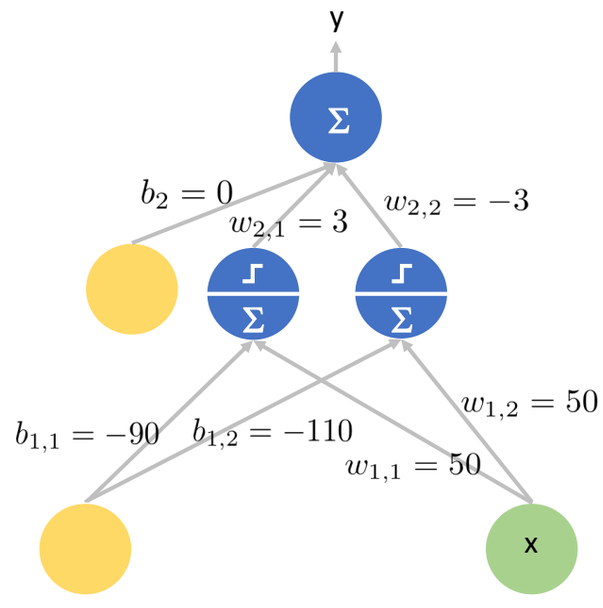
Function Approximation with Sigmoids

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.





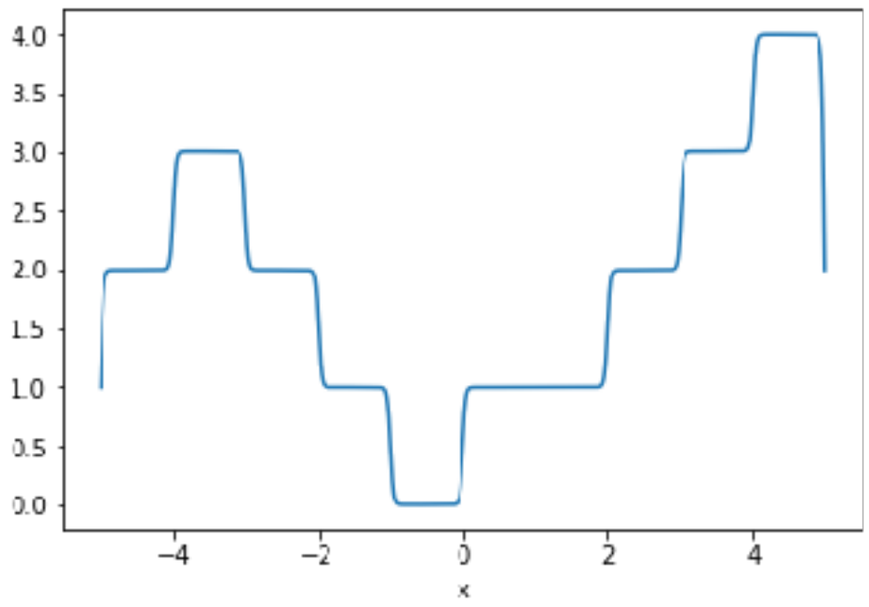


By combining two such step functions we can easily create a peak (see figure on the right and graph) at location s with width Δ :

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

Function Approximation with Sigmoids

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.



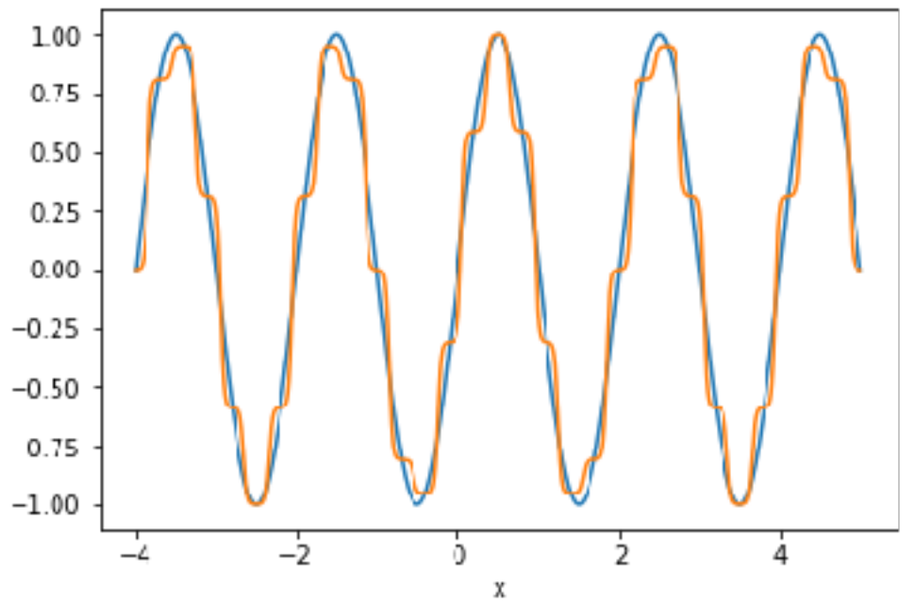

```
def sigmoid(z):
    return 1. / (1. + np.exp(-z))

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)
```



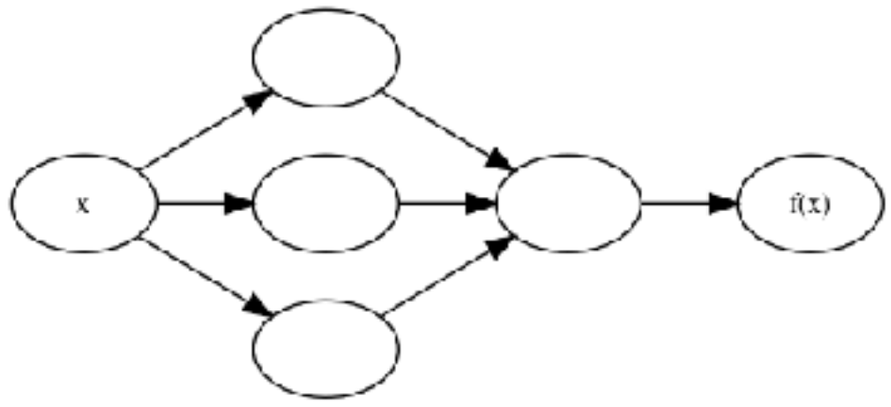

```
def f(x):
    return np.sin(x*np.pi)

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)
```

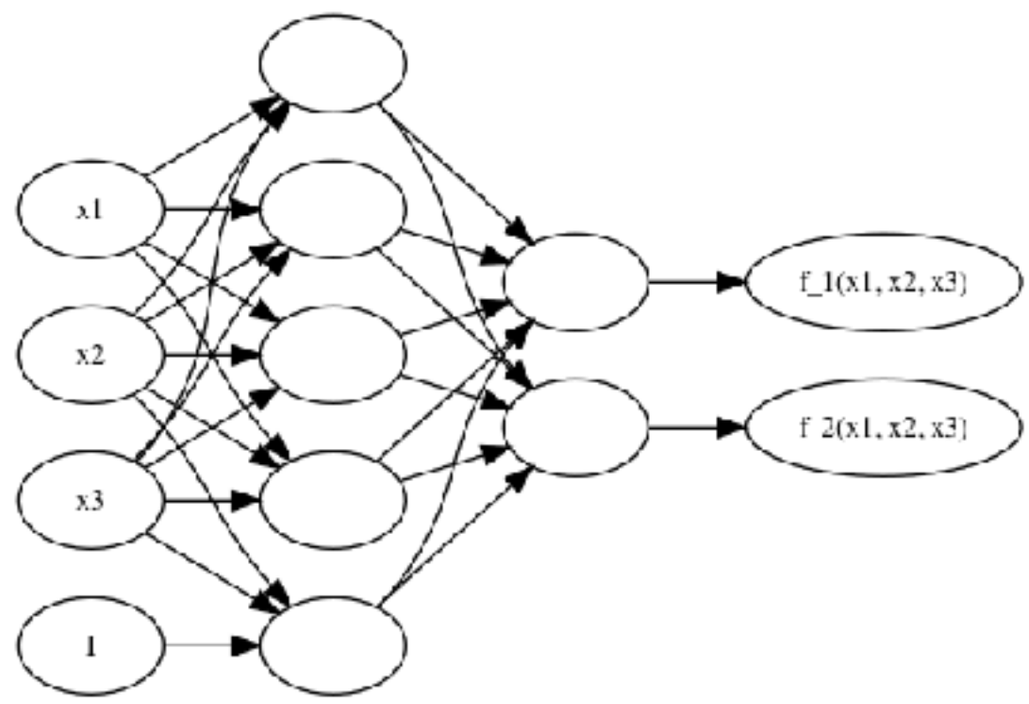
(see Exercise in PW for Week 3 / Exercise 3)

Generalisation to Arbitrary Functions

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



The scheme for 1-dim input and 1-dim output can be easily generalised to n-dim input and m-dim output as illustrated in the lower figure.



Limitations

- Approximation scheme seems to work well for functions with bounded support. What about functions with unbounded support?
  Examples:

$$h(x) = x^2, h(x_1, x_2) = x_1 \cdot x_2$$

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

Problems

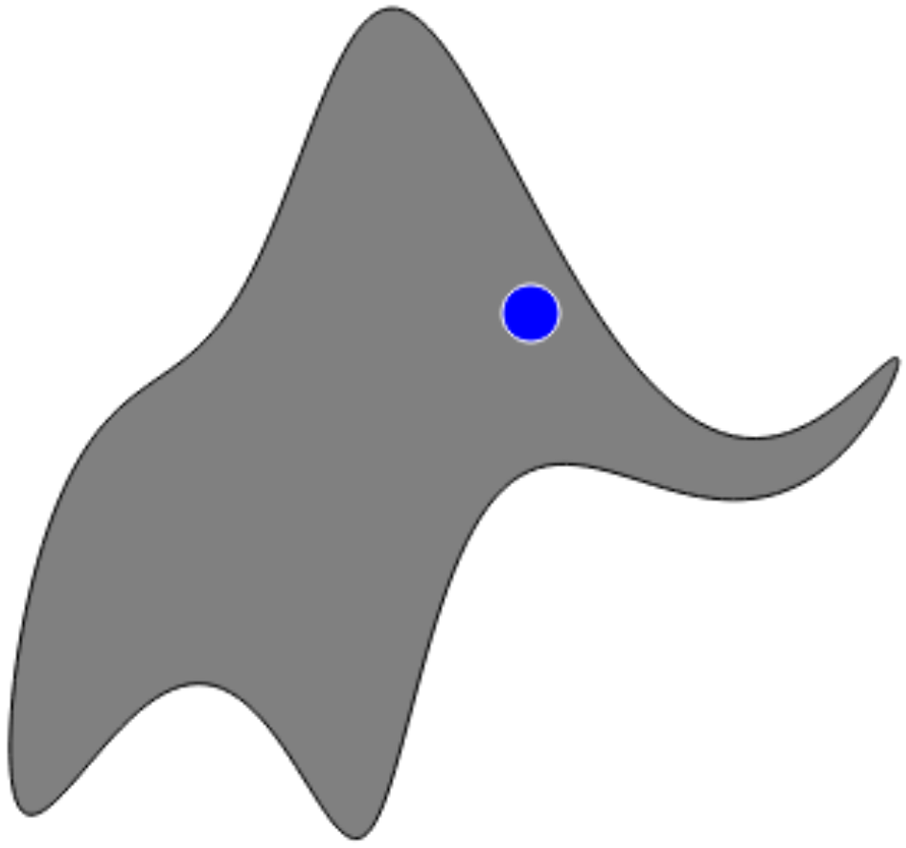
- For improving the accuracy when modelling with step-functions, more and more neurons are needed and many parameters need to be determined. In problems with high-dimensional input, an **exponentially growing number of neurons** is needed (see REF).

- 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**.

Overfitting and Generalisation

Typical Learning Curves for training and test data
Factors triggering overfitting
Example

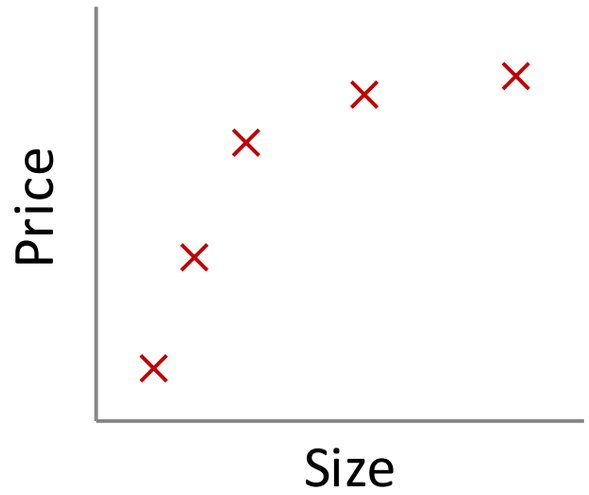


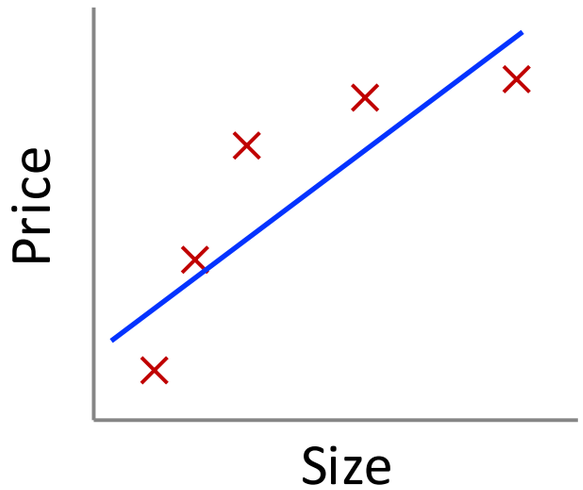
"A turning point in Freeman Dyson's life occurred during a meeting in the Spring of 1953 when Enrico Fermi criticized the complexity of Dyson's model by quoting Johnny von Neumann [1] `With four parameters I can fit an elephant, and with five I can make him wiggle his trunk'"

http://theoval.cmp.uea.ac.uk/~gcc/projects/elephant/

What is Overfitting?

Consider the data points with house prices vs size depicted on the right. What is a suitable model for prediction? The three models below (linear, quadratic, 4th order polynomial) can fit the given training data points with different accuracy (leading to different cost).

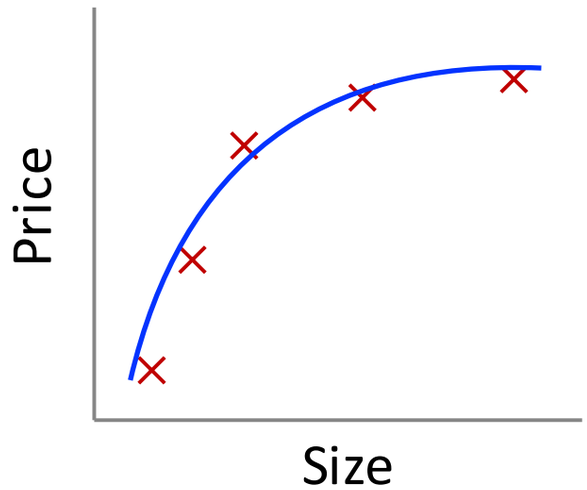




$$\theta_0 + \theta_1 x$$

Underfitting - High Bias

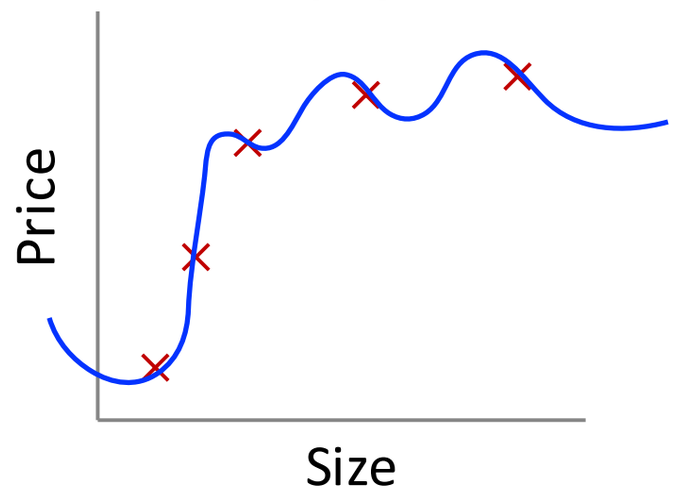
Strong bias in the way the data deviate from the linear model.



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

Good Fit - “Just Right”

The model hypothesis seems to capture just right the underlying structure in the data.

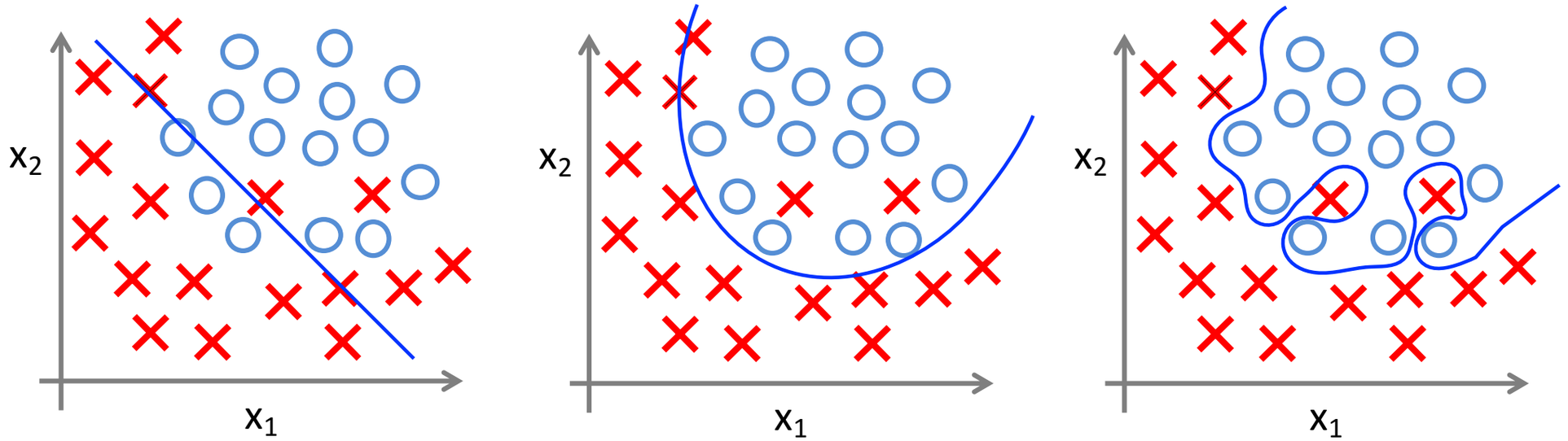


$$\theta_0 + \theta_1 x + \theta_2 x^2 + \theta_3 x^3 + \theta_4 x^4$$

Overfitting - High Variance

The model matches data points perfectly; but too well — given the number of data points; also seems to capture statistical fluctuations.

Overfitting - Binary Classification Example



$h_\theta(x) = g(\theta_0 + \theta_1 x_1 + \theta_2 x_2)$

(g = sigmoid function)

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_1^2 + \theta_4 x_2^2 + \theta_5 x_1 x_2)$

$g(\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^2 x_2 + \theta_4 x_1^2 x_2^2 + \theta_5 x_1^2 x_2^3 + \theta_6 x_1^3 x_2 + \dots)$

Underfit	Good fit	Overfit
"high bias"	"just right"	"high variance"

Occam's Razor "Among competing hypothesis the simplest is the best."

William of Ockham (1285-1347)

Definition 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

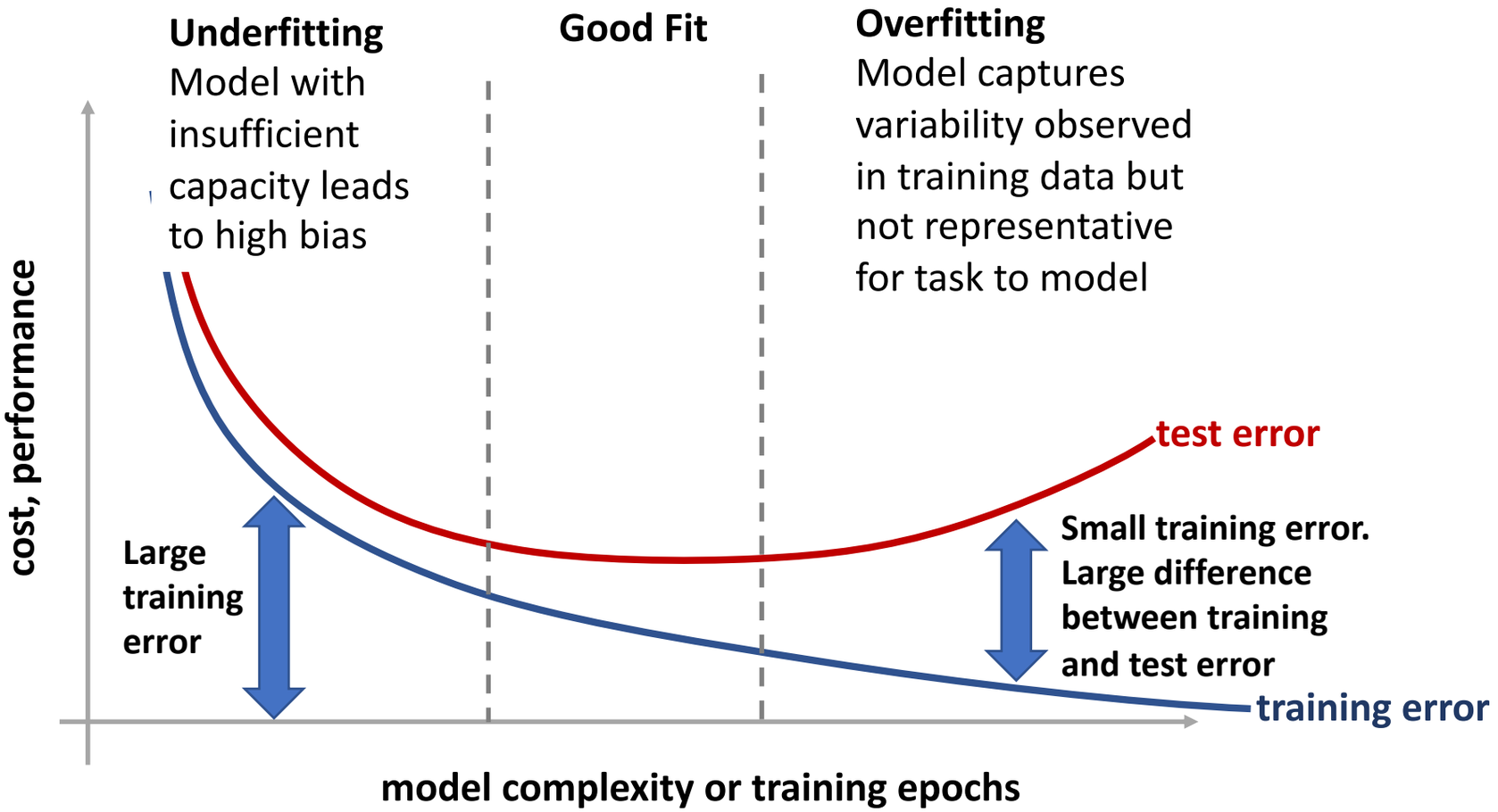
- the size of the training set is too noisy or too small in comparison with the dimension of the input variable (which introduces sampling noise).
- the number of parameters of the model is too large, i.e. the model is too “flexible”.

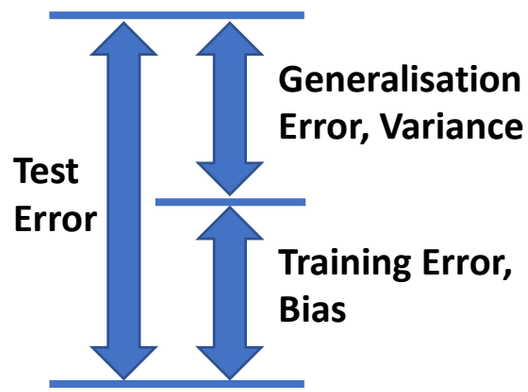
Then, the model is likely to detect patterns in the noise itself. These patterns will not generalise to new examples.

How to Examine Overfitting?

Evaluate the performance on the training and the test set

- for different models (with different model complexity) or
- after different number of training epochs (‘early stopping’)
- by using larger training sets





Outlook

- Overfitting and Generalisation (cont'd)
- Model Selection
- Performance Measures
- Computational Graphs
- Back-propagation