

# Week 2: Learning and Optimisation

TSM\_DeLearn

**Illustrated with Binary Classification of MNIST Digits**

Andreas Fischer  
Klaus Zahn

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

# Plan for this week

- Formalization of Learning as an Optimisation Problem.
- Use optimisation algorithms to solve it:  
Gradient Descent
- Illustrate the procedure with a toy model:  
MNIST digits classification

# Data Preparation

Training and Test Sets  
Data Normalisation



This covers rather some general machine learning methodology!

# Training and Test Set

- Split data into two subsets
  - Training Set : Used for learning the task.
  - Test Set : Used for testing how well the learned model performs.
  - Split ratio: Depends on available data and the specific task to be trained and tested.

Typical ratios:

	<b>Small Datasets</b>	<b>Large Datasets</b>
Train	70-80%	99%
Test	20-30%	1%

Splitting the dataset into three parts (with an additional validation set) will be treated later.

- Make sure that training and test sets have the same characteristics
  - Randomly shuffle the dataset before splitting - unless you are sure that the dataset is already shuffled.
- Keep training set and test set strictly separate!
  - Don't use any information contained in the test set to adjust parameters that are used during learning or that define the model.

# Example: MNIST Dataset

Several libraries have built-in functionality to do the split, e.g. scikit-learn

jupyter notebook (Practical Work 02)

```
from sklearn import model_selection as ms

#define train and test split
x_train, x_test, y_train, y_test = ms.train_test_split(x_sel, y_sel,
                                                    test_size=0.20, random_state=1)

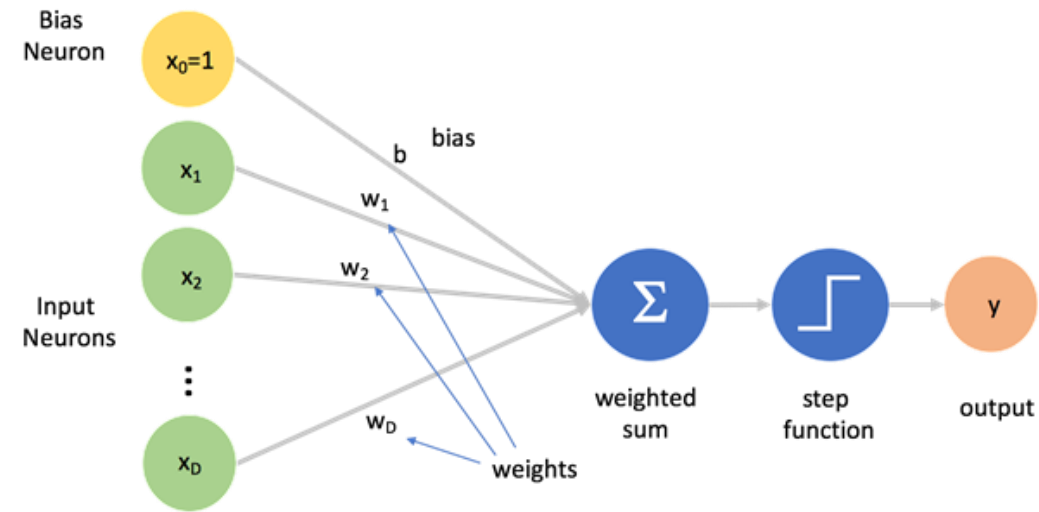
print(x_train.shape, x_test.shape, y_train.shape, y_test.shape)

(12136, 784) (3034, 784) (12136, 1) (3034, 1)
```

Depending on the framework (here sklearn) you may need to do some suitable "data wrangling" (such as the transposition of numpy arrays)

# Why is Feature Scaling beneficial?

$$y = H(z)$$
$$z = \mathbf{w} \cdot \mathbf{x} + b$$



scale of input  
features  $\mathbf{x}$

scale of  
weights  $\mathbf{w}$

$$\mathbf{w} \cdot \mathbf{x} \approx 1$$

# Notations (DL-notations.pdf on Moodle)

## Input features

Single sample  
single feature  $x_k^{(i)}$

Index to enumerate  
the samples in the  
dataset:  $1 \leq i \leq m$

Index to enumerate  
the number of input  
features:  $1 \leq k \leq n_x$

Single sample  
feature vector  $\mathbf{x}^{(i)} = (x_1^{(i)}, \dots, x_{n_x}^{(i)})$

$n_x$ : number input features

## Labels

Single sample  
scalar  $y^{(i)}$   
output/label

Single sample  
output/label  
vector  $\mathbf{y}^{(i)} = (y_1^{(i)}, \dots, y_{n_y}^{(i)})$

$n_y$ : dimension output/label vector

What are the respective ranges of a set of features after "Min-Max Rescaling"?

$$x_k^{/(i)} = \frac{x_k^{(i)} - \min_k}{\max_k - \min_k}$$

$$\min_k = \min_{1 \leq i \leq m} \{x_k^{(i)}\}, \max_k = \max_{1 \leq i \leq m} \{x_k^{(i)}\}$$



# What are the respective ranges of a set of features after "Min-Max Normalisation"?

$$x_k^{/(i)} = 2 \cdot \frac{x_k^{(i)} - \min_k}{\max_k - \min_k} - 1$$

$$\min_k = \min_{1 \leq i \leq m} \{x_k^{(i)}\}, \max_k = \max_{1 \leq i \leq m} \{x_k^{(i)}\}$$

- In general, each feature is rescaled *individually*
- must be computed on the **training set only**.

# What is special for feature scaling of images?

MNIST data range given by integer greyscale values:

- Original MNIST:  
0,...,255
- MNIST light:  
0,...,16

```
img = x[0,:].reshape((8,8))  
print(img)
```

```
[[ 0.  0.  5. 13.  9.  1.  0.  0.]  
 [ 0.  0. 13. 15. 10. 15.  5.  0.]  
 [ 0.  3. 15.  2.  0. 11.  8.  0.]  
 [ 0.  4. 12.  0.  0.  8.  8.  0.]  
 [ 0.  5.  8.  0.  0.  9.  8.  0.]  
 [ 0.  4. 11.  0.  1. 12.  7.  0.]  
 [ 0.  2. 14.  5. 10. 12.  0.  0.]  
 [ 0.  0.  6. 13. 10.  0.  0.  0.]]
```

```
x /= np.max(x)  
print(img)
```

```
[[0.      0.      0.3125 0.8125 0.5625 0.0625 0.      0.      ]  
 [0.      0.      0.8125 0.9375 0.625  0.9375 0.3125 0.      ]  
 [0.      0.1875 0.9375 0.125  0.      0.6875 0.5      0.      ]  
 [0.      0.25   0.75   0.      0.      0.5     0.5     0.      ]  
 [0.      0.3125 0.5     0.      0.      0.5625 0.5     0.      ]  
 [0.      0.25   0.6875 0.      0.0625 0.75   0.4375 0.      ]  
 [0.      0.125  0.875  0.3125 0.625  0.75   0.      0.      ]  
 [0.      0.      0.375  0.8125 0.625  0.      0.      0.      ]]
```

What would be the mean and the variance of features after "z-Normalisation"?

$$x_k^{(i)} = \frac{x_k^{(i)} - \mu_k}{\sigma_k}$$

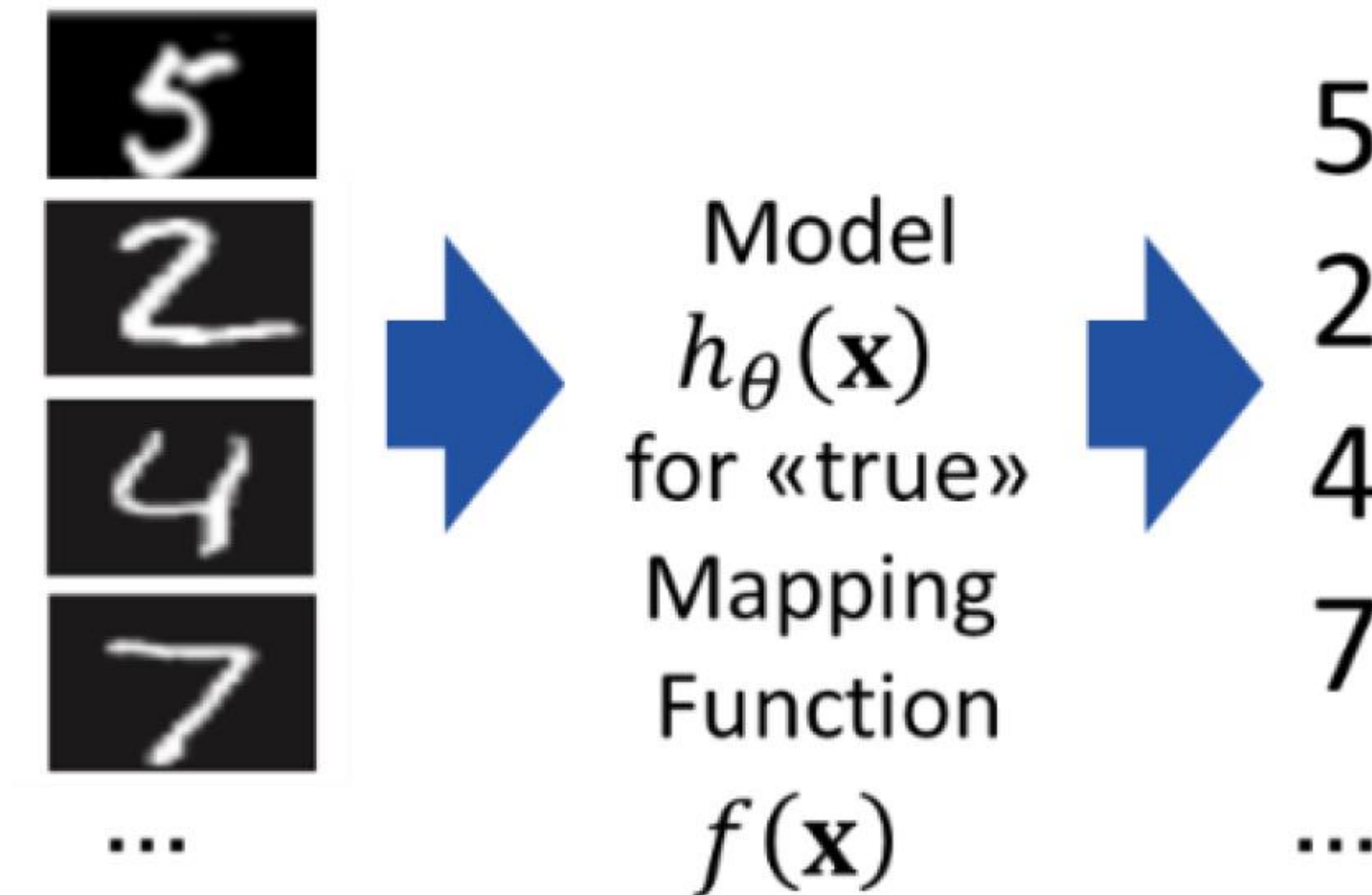
$$\mu_k = \frac{1}{m} \sum_{i=1}^m x_k^{(i)}$$

$$\sigma_k^2 = \frac{1}{m} \sum_{i=1}^m (x_k^{(i)} - \mu_k)^2$$

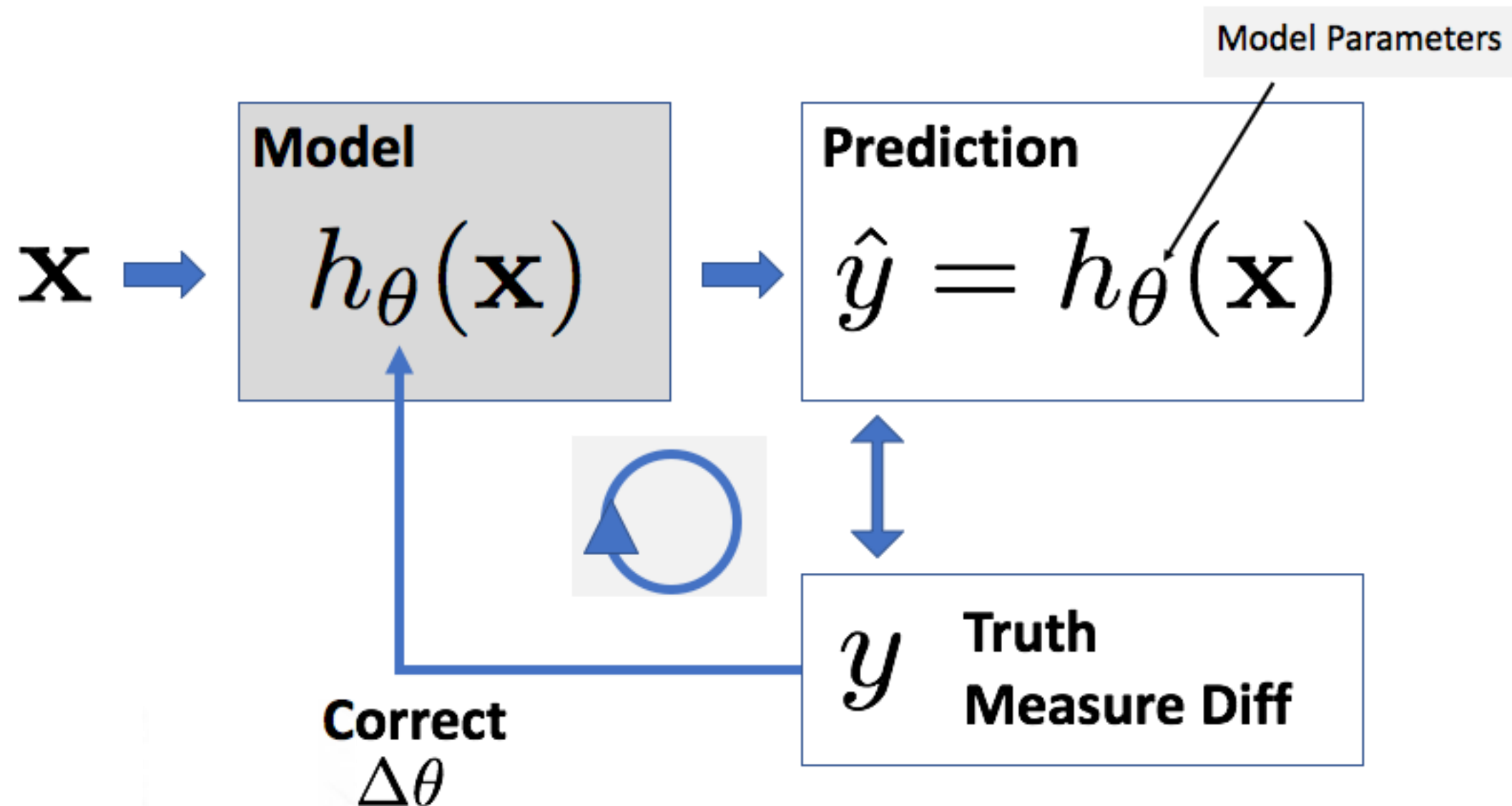
# Generalised Perceptron



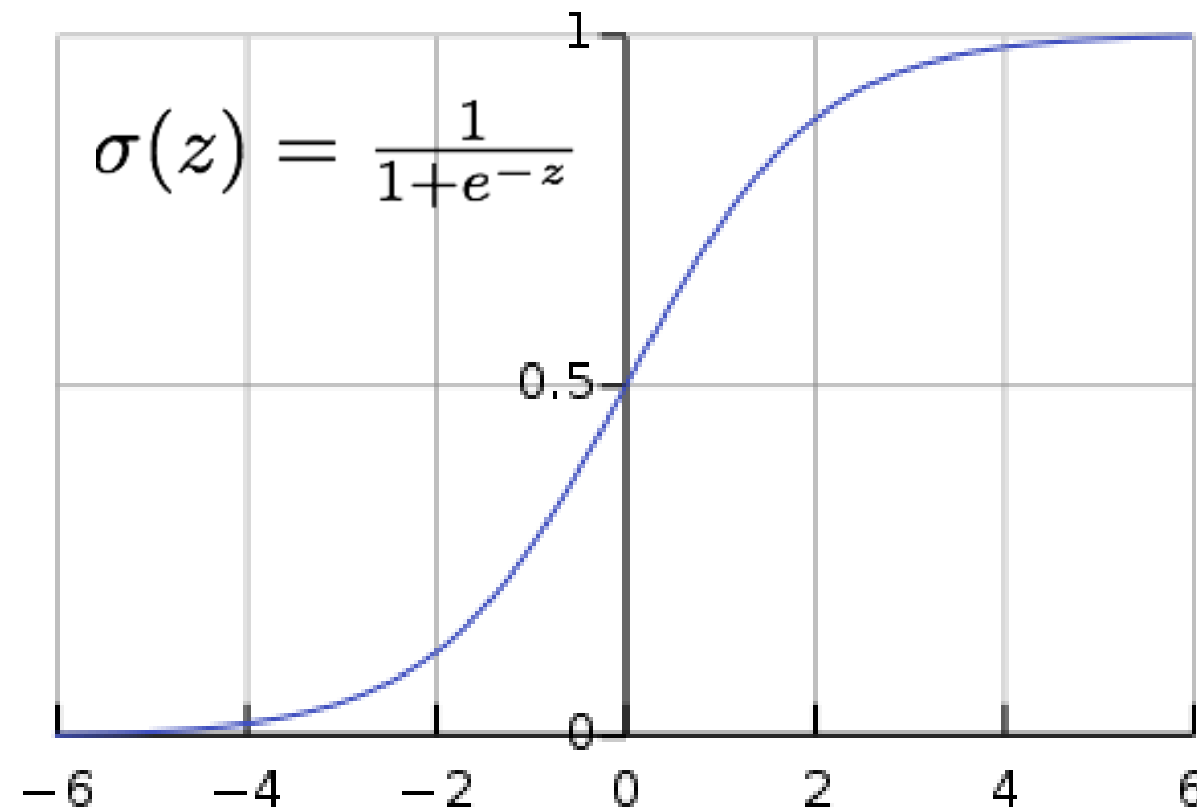
# Recall: Formalization of Learning Procedure



# Recall: Formalization of Learning Procedure



# Smooth activation function: Sigmoid Function



Properties of the **sigmoid function** will be studied in the exercises.

better suited than 'hard-limit' Heaviside-function:

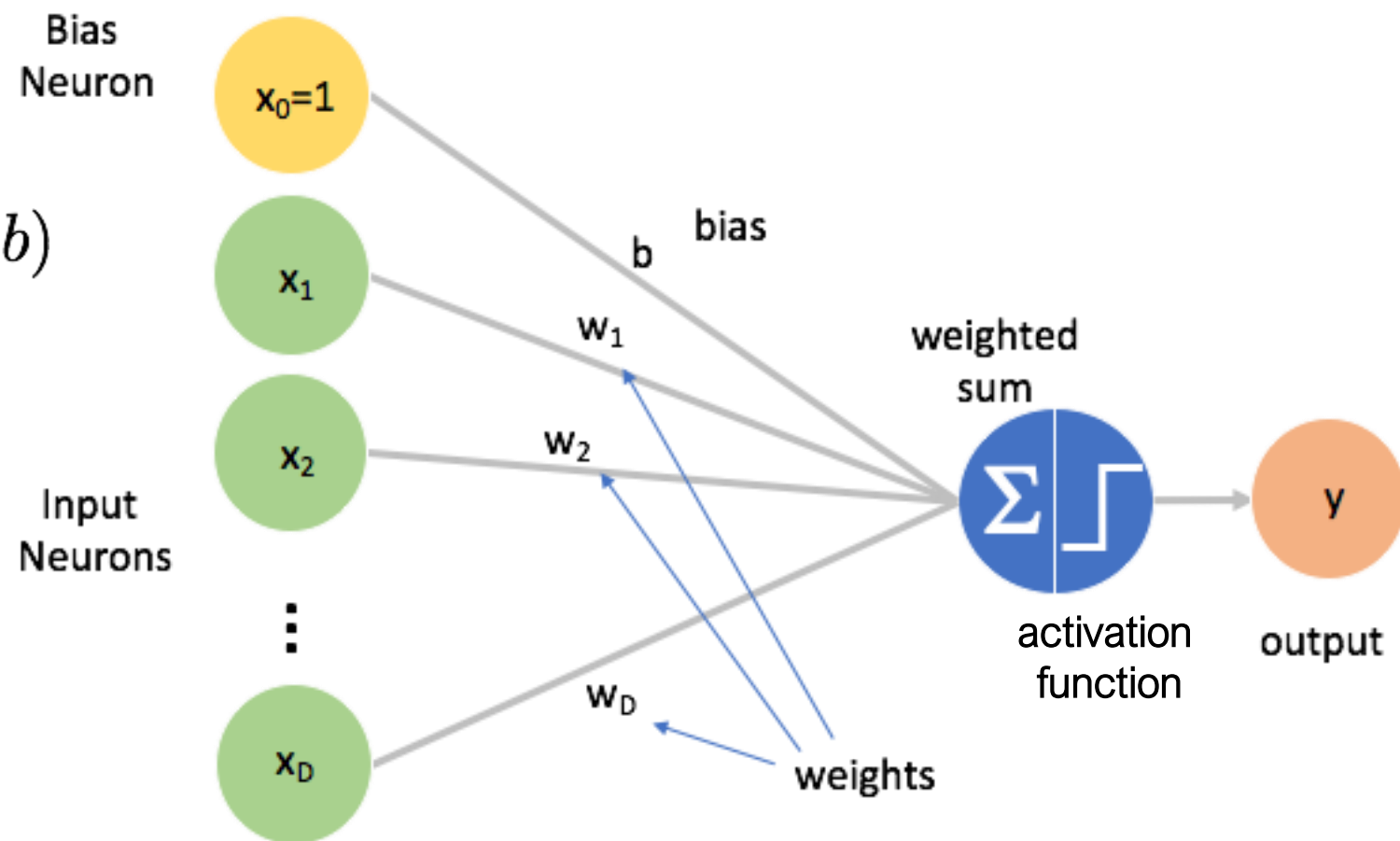
- Differentiable so that optimisation techniques from calculus can be adopted -> GD
- probabilistic interpretation:  
Answer is given in form of a probability

# Generalised Perceptron

$$\hat{y} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

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

$$\theta = (\mathbf{w}, b)$$



## Output

- No longer a 'yes' (1) or 'no' (0), but a numeric value  $\hat{y} \in ]0, 1[$
- Class label can be assigned by the rule:

$$\text{yes} : \hat{y} = h_{\theta}(\mathbf{x}) \geq 0.5$$

$$\text{no} : \hat{y} = h_{\theta}(\mathbf{x}) < 0.5$$

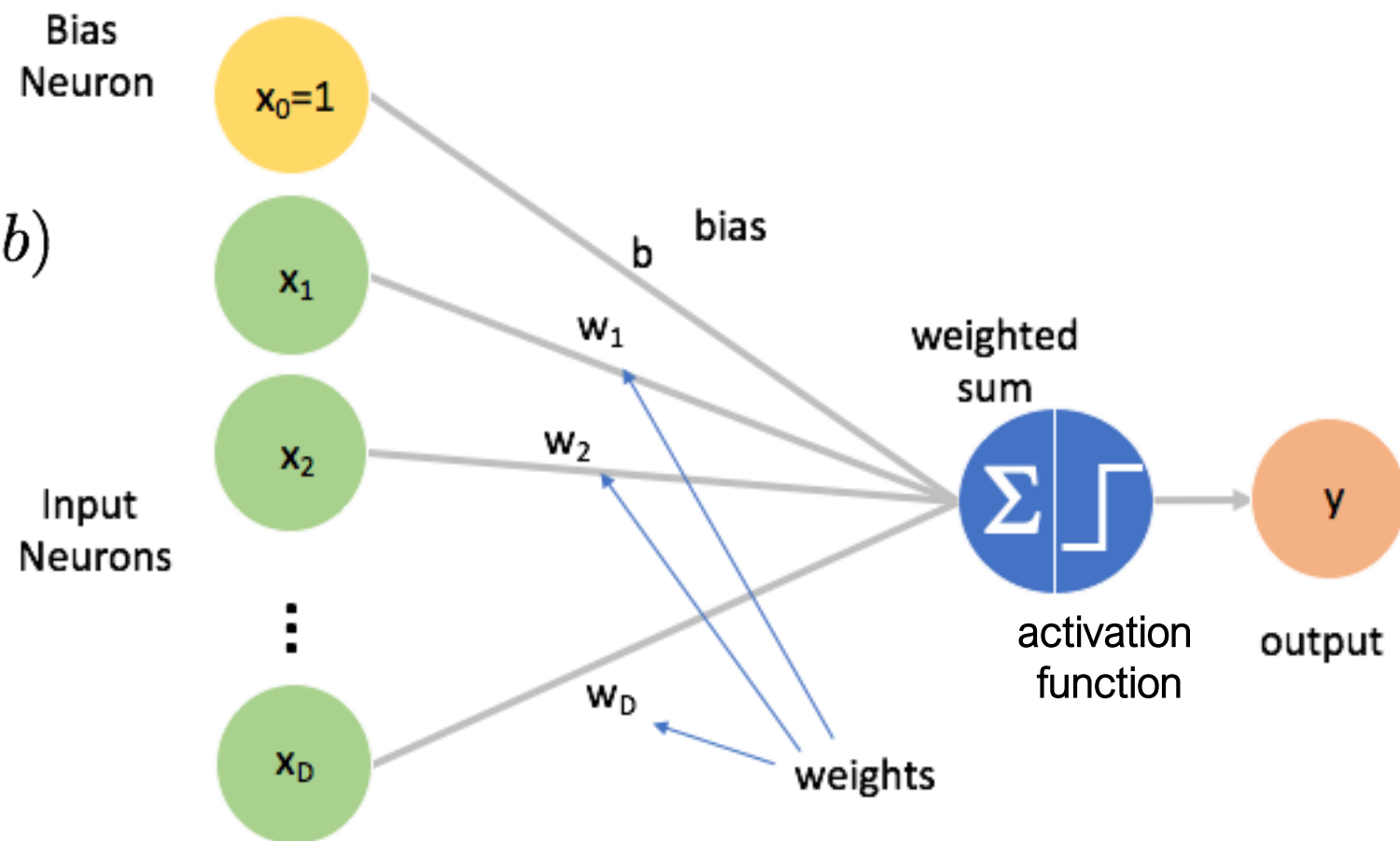


# Generalised Perceptron

$$\hat{y} = h_{\theta}(\mathbf{x}) = \sigma(\mathbf{w} \cdot \mathbf{x} + b)$$

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

$$\theta = (\mathbf{w}, b)$$



optimization of the parameters  $\theta = (\mathbf{w}, b)$  ?

Measure closeness of outcomes  $y^{(i)}$  and the predictions  $\hat{y}^{(i)} = h_{\theta}(\mathbf{x}^{(i)})$   
 -> Cost Function

# How to Choose the Parameters?

- Choose parameters such that the predicted values  $\hat{y}$  are in some notion of distance 'close' to the true outcomes  $y$ .
- The notion of distance is expressed in terms of a cost function. Choose the parameters such that the cost gets smallest.
- Different cost functions lead to different solutions.
- We will discuss two cost functions:
  - (1) Mean Squared Distance between  $y$  and  $\hat{y}$ .
  - (2) Cross-Entropy (distance between probability distributions)

# Mean Square Error Cost Function

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

This is for later math simplifications

This is to normalise the cost regarding the size of training set — so that the cost magnitude of the cost is independent of the sample size  $m$ .

The square makes sure that a positive definite distance measure is obtained — the cost is positive or zero only if all the prediction exactly match the truth.

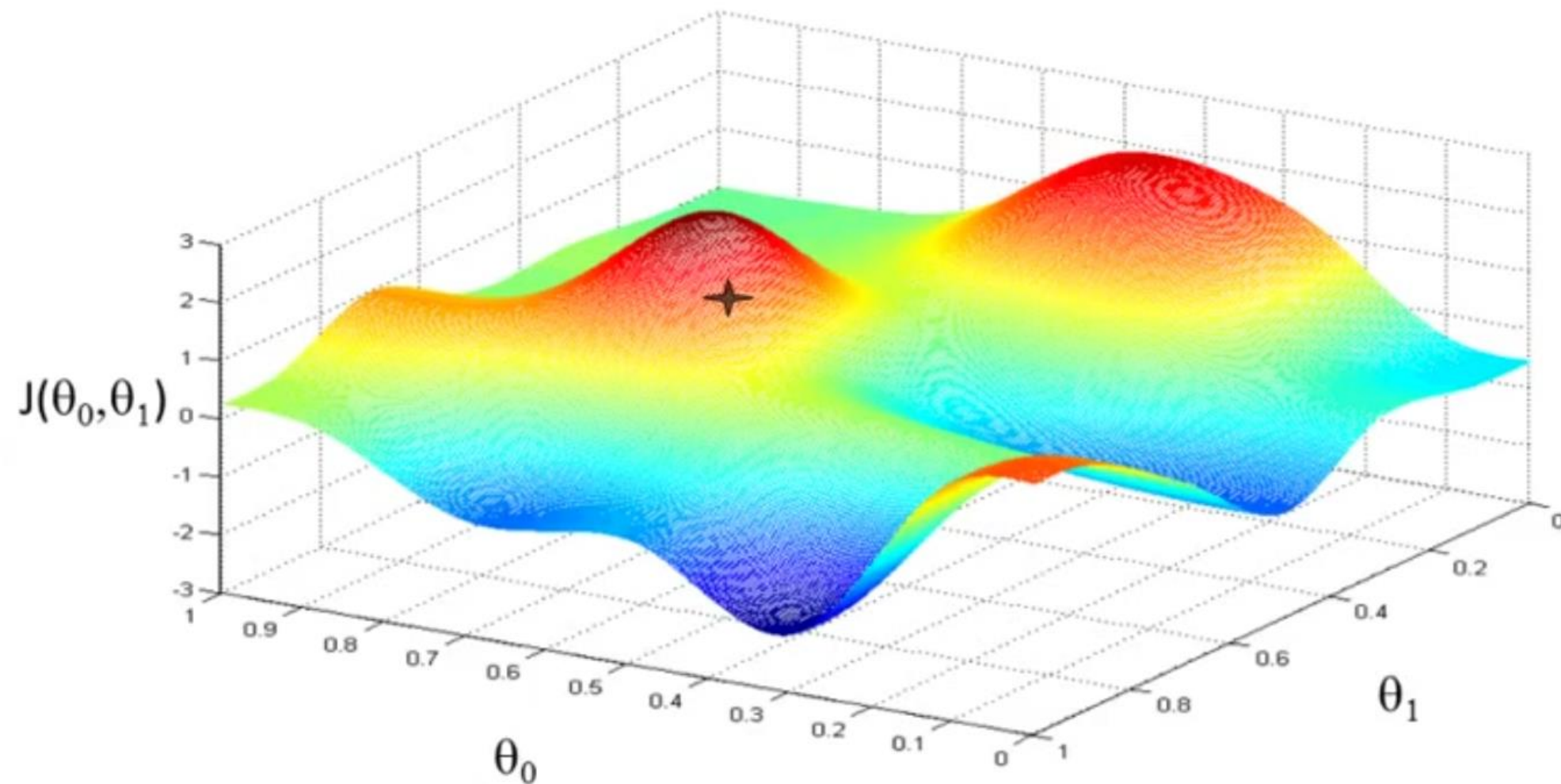
By minimising  $J_{MSE}(\theta)$  with respect to the parameters  $\theta$  we adjust the model to 'best' represent the mapping  $\mathbf{x} \rightarrow y$  seen in the training data.

# Solving the Minimisation Problem

- In general, *no closed-form solution* exists for these kind of optimisation problems — a closed-form solution only exists for very specific cases (e.g. linear regression).
- Thus, *numeric methods* are needed to find a solution for the parameters.
- *Gradient Descent* is a family of optimisation algorithms widely used in practice.

# Gradient Descent Intuition

From the initial position, take steps in the opposite direction of the gradient, "going downhill" until a minimum is reached.



Source: Andrew Ng - Machine Learning class - Stanford

# Basic Principle of Gradient Descent

Minimisation of cost function  $J(\theta)$ :

- 1) Start with some initial value for the parameter vector (for example random or 0):  $\theta_0$
- 2) Iteratively update the parameter vector by
  - (i) Compute the gradient of the cost function  $J(\theta)$  at the last position reached ( $\theta_t$ ):
  - (ii) Step in the negative gradient direction according to

$$\theta_{t+1} = \theta_t - \alpha \cdot \nabla_{\theta} J(\theta_t) \quad (\alpha: \text{learning rate})$$

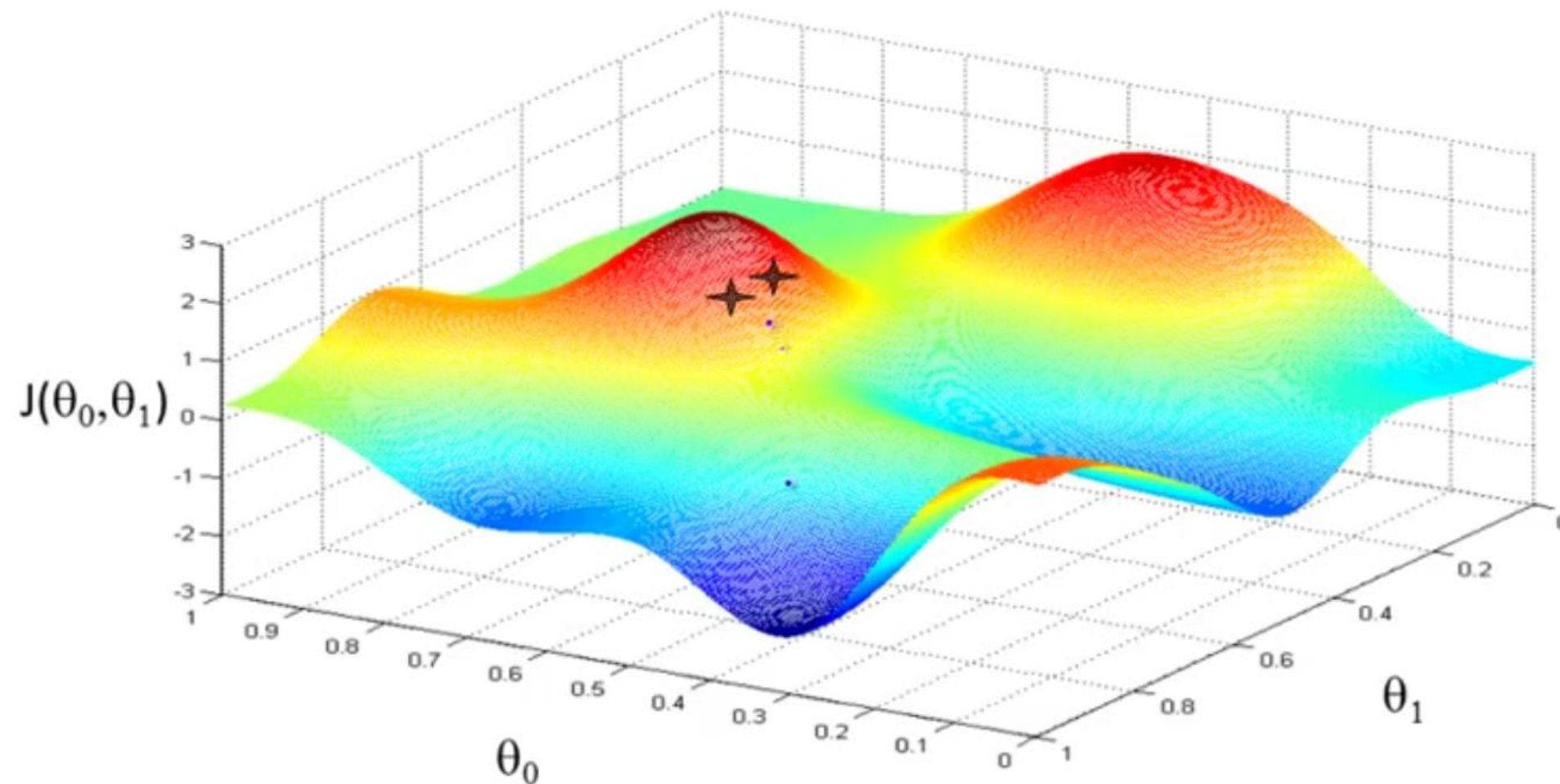
- (3) Stop when change in parameter vector small.

# Characteristics of Gradient Descent

- The GD learning principle does not depend on the type of model and can be applied if gradients can be computed (e.g., also works for deep neural networks).
- Learning principle works ‘locally’ (by using local function properties) — designed to find local but not necessarily global minimum.
- In addition, the position of the final minimum approached may depend very sensitively upon the starting position
- The learning schema can get stuck in critical points where gradient is zero. Without additional information, it cannot distinguish between local minima, local maxima or saddle points.
- GD is iterative and iteratively approaches a critical point and may fluctuate around it.
- Thus, it is not guaranteed to converge e.g., if learning rate is chosen too large.



# Sensitivity upon the starting position



The path to the (local) minimum depends on the initial conditions. A slight difference in the initial parameter values, may lead to another minimum, a local minimum which is above the global minimum reached in the previous run.

Source: Andrew Ng - Machine Learning class - Stanford



# Gradient of the Mean Square Cost

$$\frac{\partial}{\partial \theta_k} J_{MSE}(\boldsymbol{\theta}) \quad \text{given} \quad J_{MSE}(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

Focus on one term:

$$\frac{\partial}{\partial \theta_k} (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2 =$$

# Gradient of the Mean Square Cost

$$\frac{\partial}{\partial \theta_k} J_{MSE}(\boldsymbol{\theta}) \quad \text{given} \quad J_{MSE}(\boldsymbol{\theta}) = \frac{1}{2m} \sum_{i=1}^m (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

Focus on one term:

$$\frac{\partial}{\partial \theta_k} (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})^2 = 2 \cdot (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot \frac{\partial}{\partial \theta_k} h_{\theta}(\mathbf{x}^{(i)}) =^{(1)}$$

$$= 2 \cdot (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot (1 - h_{\theta}(\mathbf{x}^{(i)})) \cdot h_{\theta}(\mathbf{x}^{(i)}) \cdot \frac{\partial}{\partial \theta_k} (\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$$

$$=^{(1)} \text{ we used: } \sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

# Gradient of the Mean Square Cost

Require: 
$$\frac{\partial}{\partial \theta_k} (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) = \begin{cases} x_k^{(i)} & (\theta_k = w_k) \\ 1 & (\theta_k = b) \end{cases}$$

Putting everything together:

$$\nabla_{\mathbf{w}} J_{MSE}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^m \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \cdot (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}$$
$$\nabla_b J_{MSE}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^m \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \cdot (\hat{y}^{(i)} - y^{(i)})$$

# Gradient of the Mean Square Cost (Generalised Perceptron)

$$\nabla_{\mathbf{w}} J_{MSE}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^m \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \cdot (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}$$

$$\nabla_b J_{MSE}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^m \hat{y}^{(i)} \cdot (1 - \hat{y}^{(i)}) \cdot (\hat{y}^{(i)} - y^{(i)})$$

Average  
over the  
training  
set.

Measure of uncertainty:  
Small if  $\hat{y}^{(i)}$  is close to  
zero or close to 1 (i.e. if  
the prediction is clear)  
and large if  $\hat{y}^{(i)}$  is around  
0.5 (high uncertainty)

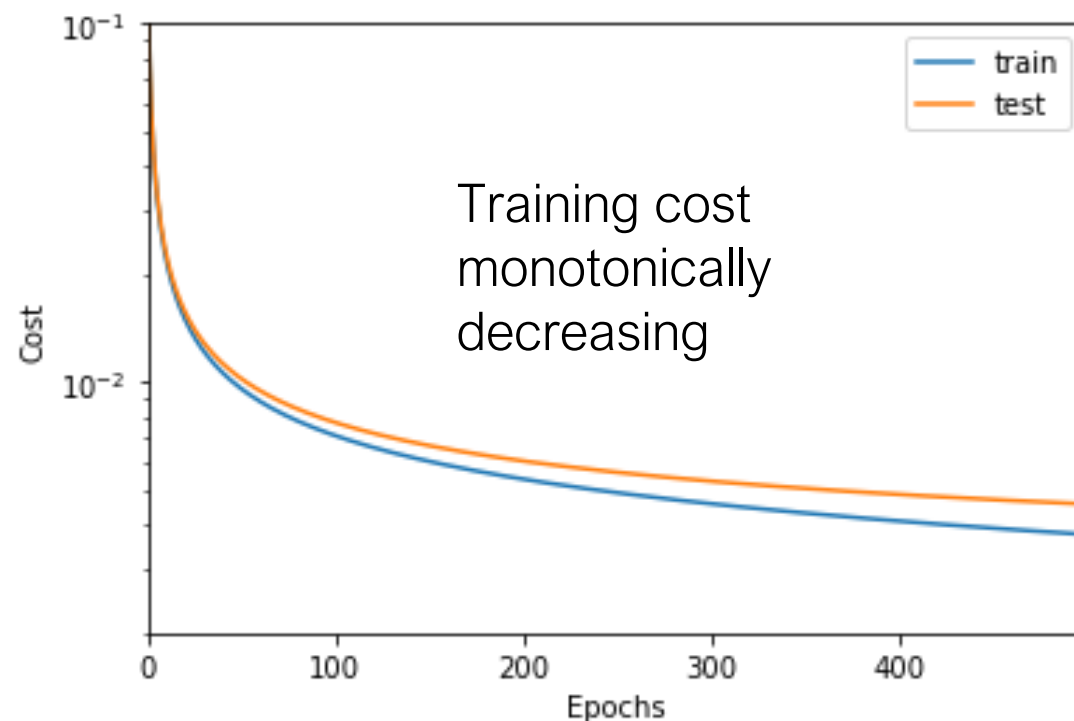
Error Signal:  
contributes more for  
samples with  
mismatch. Similar to  
Perceptron Learning  
Rule but  $\hat{y}^{(i)}$  is  
different.

# Binary Classification on MNIST Dataset

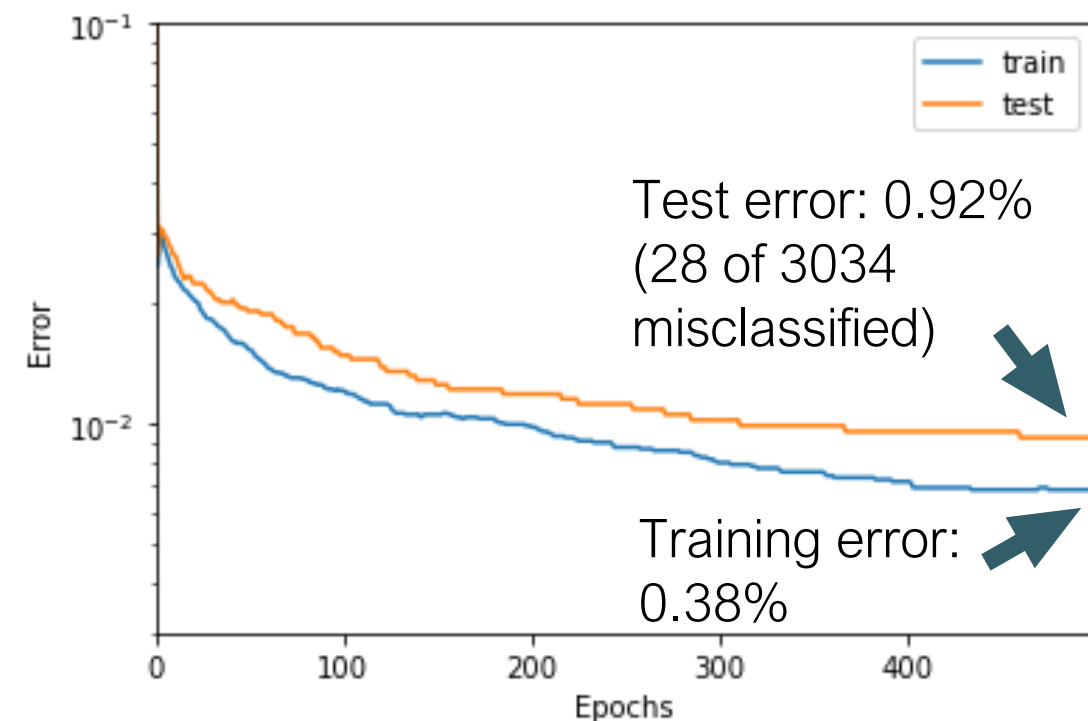
- Single LTU
- Sigmoid activation function
- MSE cost function will be used.
- Split ratio of training and test set sizes is 80% to 20
- Min-Max-Rescaling.
- Weights and bias initialized to zero
- Learning rate  $\alpha=0.5$
- Total of 500 GD optimisation steps (so-called epochs)

# Results for (1,7)-MNIST Digits

MSE cost function  $J_{MSE}(\theta)$



Error / Performance Measure



Fraction of misclassified samples as **performance measure**:

$$\text{error} = \frac{|\{\hat{y} \neq y\}|}{m_{\text{test}}}$$

with  $\hat{y} = \text{round}(h_{\theta}(\mathbf{x}))$

(  $m_{\text{test}} = \# \text{ test samples}$  )

# Results for (1,7)-MNIST

Mis-classified images  
of the test set.

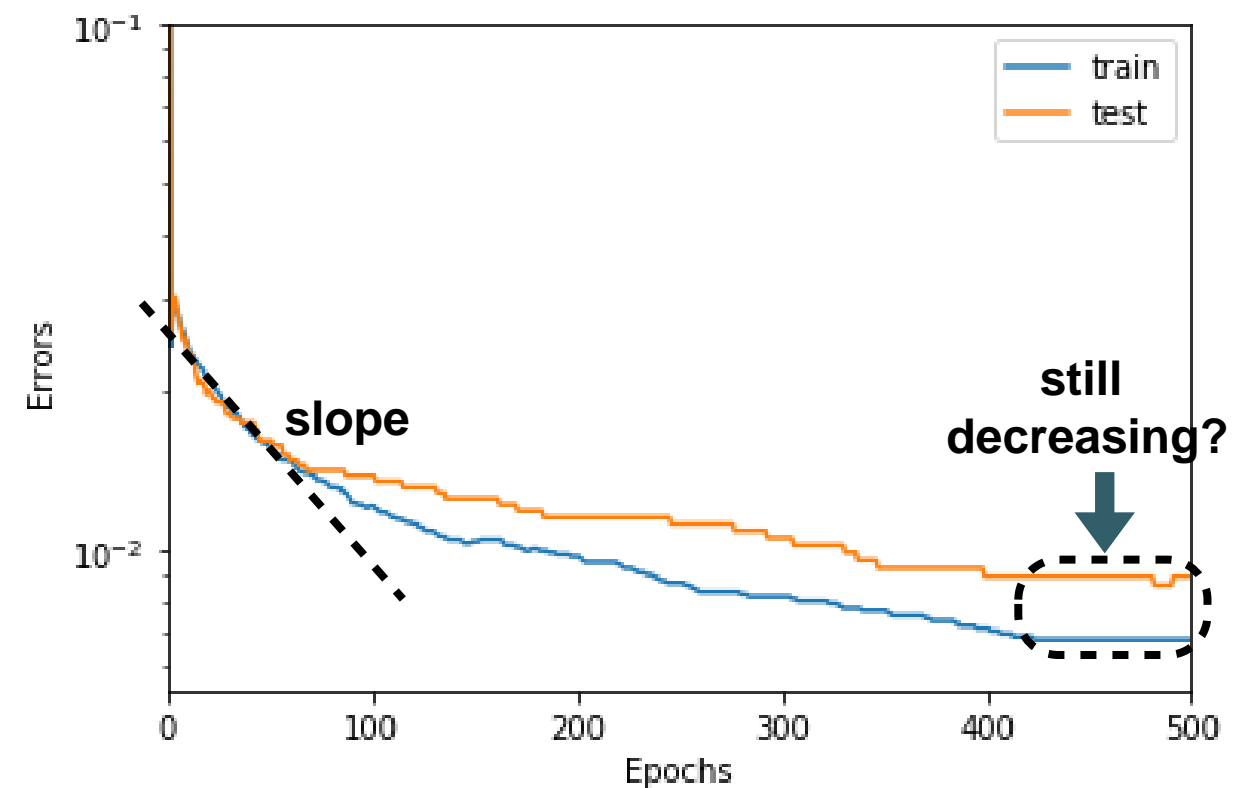


The corresponding  
label value indicates  
the ground truth.

1	7	1	7	1	7	7
1	1	7	7	7	7	7
7	7	1	1	7	7	7
1	7	7	7	7	7	1

# Questions – Possible Improvements

- What is the effect of different learning rates?
- What is the effect of more training epochs?
- What influence has the choice of the parameter initialisation?
- Are there other i.e., better cost functions.



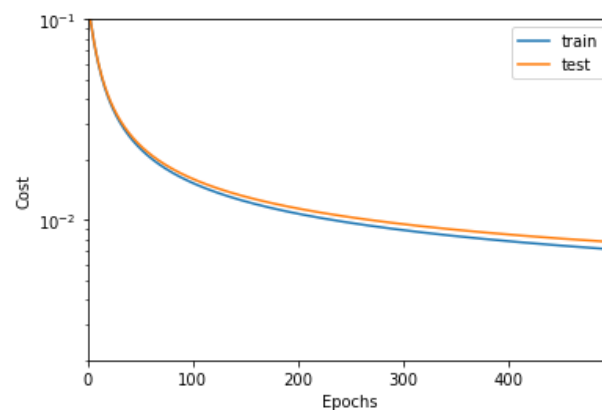


# Effect of Different Learning Rates

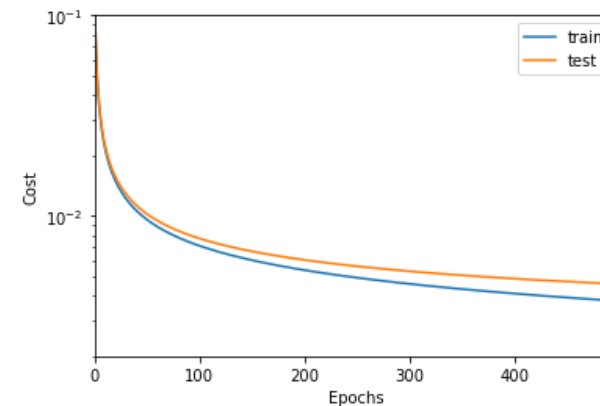
Running batch gradient descent over 500 epochs with different learning rates...

Learning rates here atypically large — usually much smaller rates are chosen.

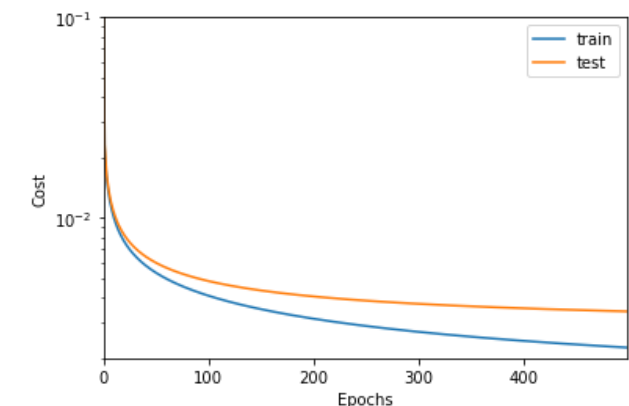
**learning rate 0.1**



**learning rate 0.5**

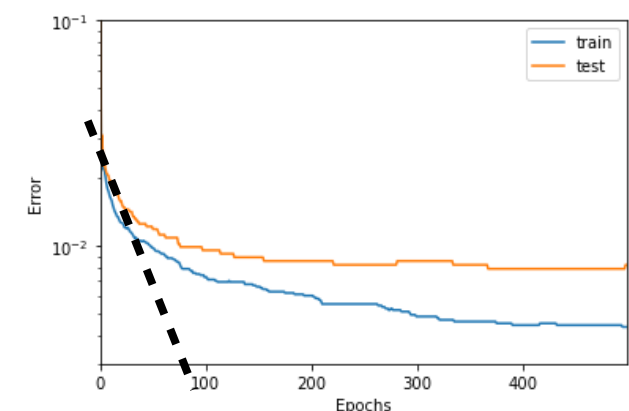
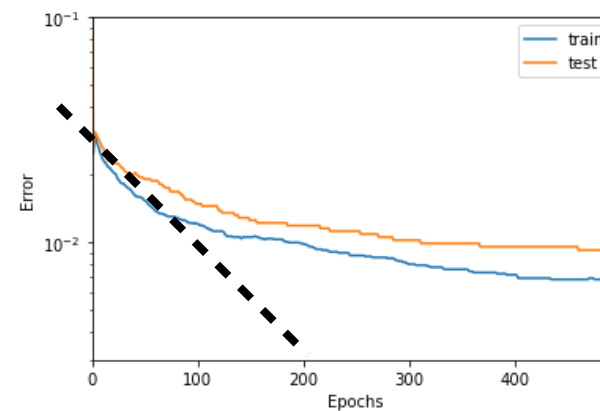
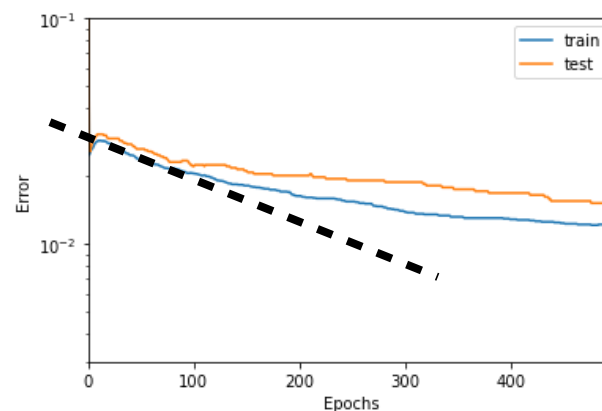


**learning rate 2.0**



Error rate starts at ~50% (not seen on the plot) — corresponds to error rate of dummy predictor.

Dummy Predictor:  
Randomly predict one or the other digit.

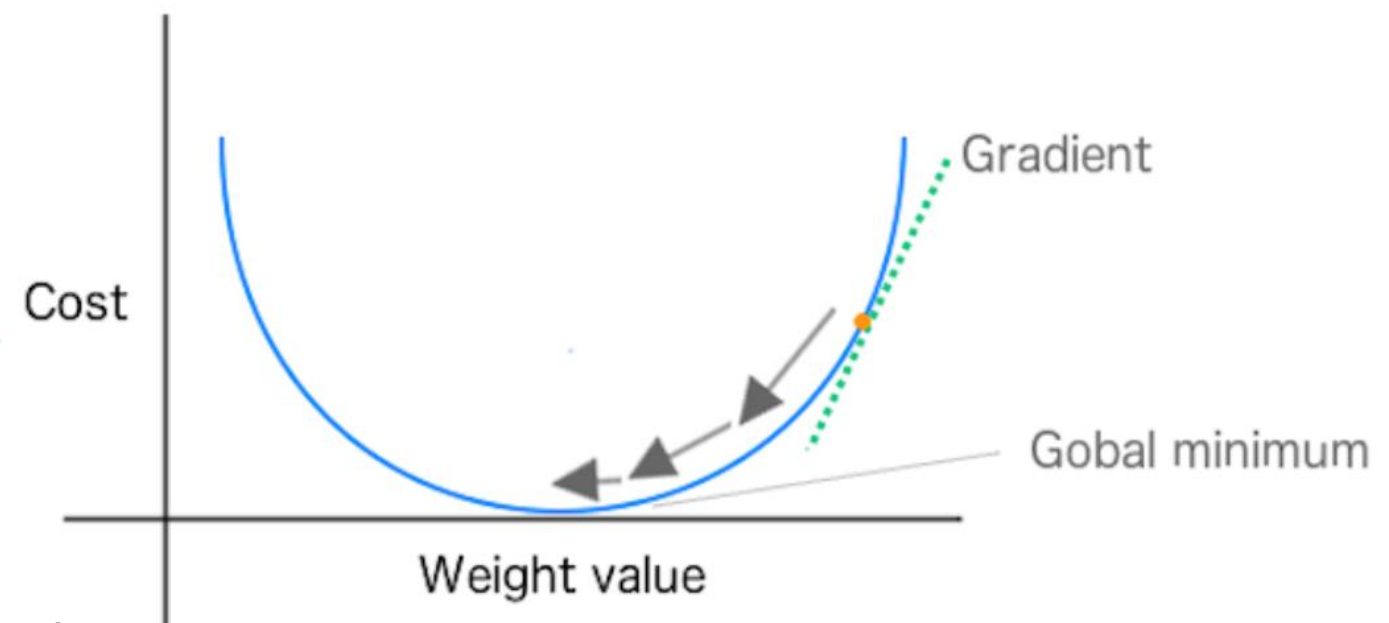
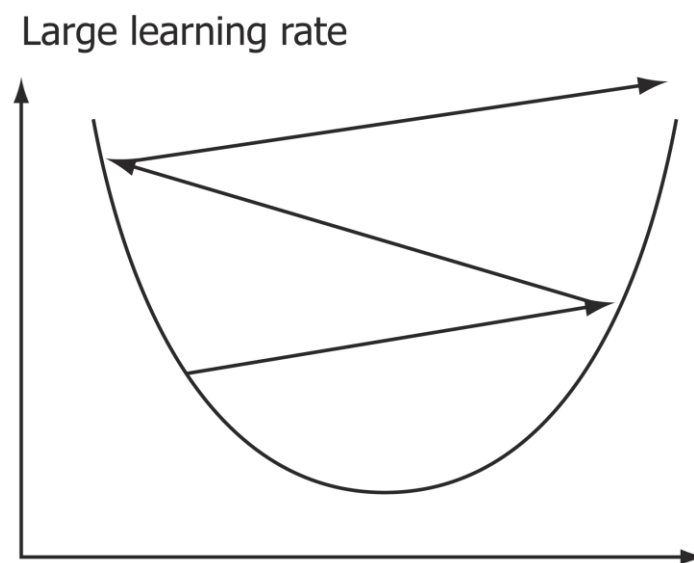


slow decrease in cost and error rates, possibly more epochs are needed.

faster decrease in cost and error rates

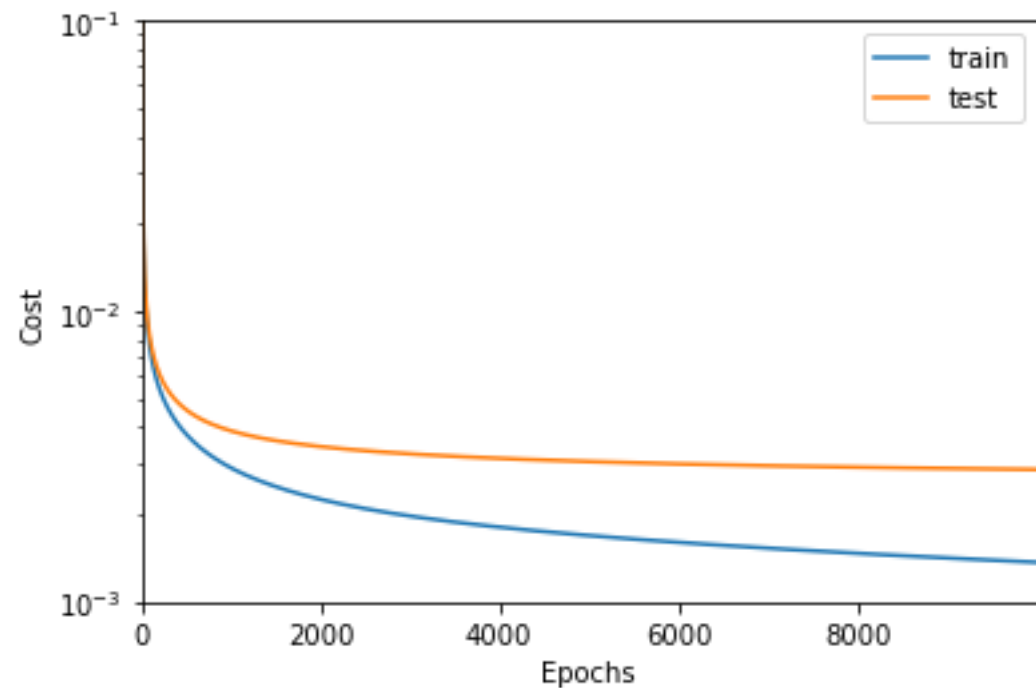
# Learning Rate $\alpha$

- Determines the learning speed
  - If too large: Oscillation around the minimum or even divergence
  - If too small: Slow convergence



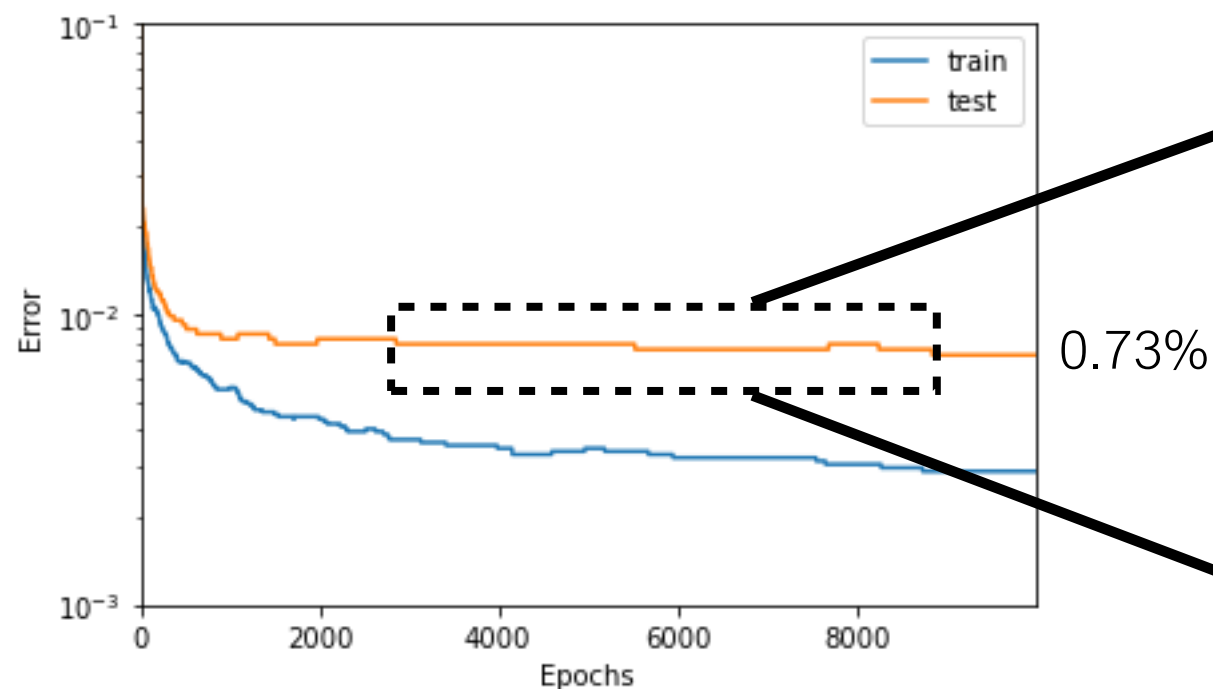
- Needs to be tuned to a given problem
  - An optimal value for a given problem may not work for another problem.
  - Thanks to data normalisation it's largely independent of the scale of the original data.

# Effect of More Epochs



After a few 1'000 epochs the test performance (test error) cannot be further reduced.

After a few 1'000 epochs the model gets overfitted. This important topic will be further covered in weeks 3 and 5).



**22 misclassified test images**



# Issues with MSE Cost and Alternatives

- Recall that the **gradient of the MSE cost** contains in each summand a factor

$$\hat{y}^{(i)}(1 - \hat{y}^{(i)}) \leq 1/4$$

As the model output  $\hat{y}^{(i)}$  gets closer to either 0 or 1 (where the model is very confident in its prediction) this expression and the gradient can get very small. In result, the change in parameters can get very small and training can get stalled or stuck.

- For *classification tasks*, the **cross entropy cost function** (defined below) is better suited and the gradient has nicer properties.

By using the cross entropy cost function, a probabilistic perspective is adopted. Hence we need to grasp some probability theory to properly understand it....

# Motivation of CE via Maximum Likelihood

- Maximum Likelihood is a widely used principle for determining the parameters of a model: Choose the parameters  $\theta$  such that the “likelihood” for observing the given dataset


$$\left\{ (\mathbf{x}^{(i)}, y^{(i)}), i = 1, \dots, m_{\text{train}} \right\}$$

is maximised.

- In mathematical terms:

$$\theta_{MLE} = \operatorname{argmax}_{\theta} p_{\theta}(y|\mathbf{x}) = \operatorname{argmax}_{\theta} \prod_{i=1..m} p_{\theta}(y^{(i)}|\mathbf{x}^{(i)})$$

$$\theta_{MLE} = \operatorname{argmax}_{\theta} \sum_{i=1}^m \log p_{\theta}(y^{(i)}|\mathbf{x}^{(i)}) = \operatorname{argmin}_{\theta} \underbrace{\left[ - \sum_{i=1}^m \log p_{\theta}(y^{(i)}|\mathbf{x}^{(i)}) \right]}_{\text{CE Cost}}$$



log(.) a monotonous function

functional equation for the log(.)

CE Cost

# Formulation of CE Cost for our task

- The quantity  $h_{\theta}(\mathbf{x}^{(i)})$  can be interpreted as the probability for ‘observing’  $y^{(i)} = 1$  given  $\mathbf{x}^{(i)}$ . Similarly,  $1 - h_{\theta}(\mathbf{x}^{(i)})$  is then interpreted as the probability of ‘observing’  $y^{(i)} = 0$  given  $\mathbf{x}^{(i)}$ .

- In maths terms:

$$p_{\theta}(y^{(i)} = 1, \mathbf{x}^{(i)}) = h_{\theta}(\mathbf{x}^{(i)})$$

$$p_{\theta}(y^{(i)} = 0, \mathbf{x}^{(i)}) = 1 - h_{\theta}(\mathbf{x}^{(i)})$$

Thus, the formulation of CE cost involves two terms:

$$J_{CE}(\boldsymbol{\theta}) = -\frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \cdot \log h_{\theta}(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \cdot \log (1 - h_{\theta}(\mathbf{x}^{(i)})) \right]$$

# Cross-Entropy Cost Function

- Cross entropy cost function in general formulation

$$J_{CE}(\boldsymbol{\theta}) = - \sum_{i=1}^m \log p_{\theta}(y^{(i)} | \mathbf{x}^{(i)})$$

- For the binary classification problem:

$$J_{CE}(\boldsymbol{\theta}) = - \frac{1}{m} \sum_{i=1}^m \left[ y^{(i)} \cdot \log h_{\theta}(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \cdot \log (1 - h_{\theta}(\mathbf{x}^{(i)})) \right]$$

# Gradient of the Cross-Entropy Cost

$$\begin{aligned}\frac{\partial}{\partial \theta_k} J_{CE}(\boldsymbol{\theta}) &= -\frac{1}{m} \sum_{i=1}^m \frac{\partial}{\partial \theta_k} \left[ y^{(i)} \cdot \log h_{\theta}(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \cdot \log (1 - h_{\theta}(\mathbf{x}^{(i)})) \right] \\ &= -\frac{\partial}{\partial \theta_k} \left[ y^{(i)} \cdot \log h_{\theta}(\mathbf{x}^{(i)}) + (1 - y^{(i)}) \cdot \log (1 - h_{\theta}(\mathbf{x}^{(i)})) \right] = \\ &= -\left[ \frac{y^{(i)}}{h_{\theta}(\mathbf{x}^{(i)})} \cdot \frac{\partial}{\partial \theta_k} h_{\theta}(\mathbf{x}^{(i)}) + \frac{(1 - y^{(i)})}{1 - h_{\theta}(\mathbf{x}^{(i)})} \cdot \frac{\partial}{\partial \theta_k} [-h_{\theta}(\mathbf{x}^{(i)})] \right] = \\ &= -\left[ \frac{y^{(i)}}{h_{\theta}(\mathbf{x}^{(i)})} - \frac{(1 - y^{(i)})}{1 - h_{\theta}(\mathbf{x}^{(i)})} \right] \cdot \frac{\partial}{\partial \theta_k} h_{\theta}(\mathbf{x}^{(i)}) =\end{aligned}$$



# Gradient of the Cross-Entropy Cost

$$-\left[ \frac{y^{(i)}}{h_{\theta}(\mathbf{x}^{(i)})} - \frac{(1 - y^{(i)})}{1 - h_{\theta}(\mathbf{x}^{(i)})} \right] \cdot \frac{\partial}{\partial \theta_k} h_{\theta}(\mathbf{x}^{(i)}) =^{(1)}$$

$$-\left[ \frac{y^{(i)}}{h_{\theta}(\mathbf{x}^{(i)})} - \frac{(1 - y^{(i)})}{1 - h_{\theta}(\mathbf{x}^{(i)})} \right] \cdot (1 - h_{\theta}(\mathbf{x}^{(i)})) \cdot h_{\theta}(\mathbf{x}^{(i)}) \cdot \frac{\partial}{\partial \theta_k} (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) =$$

$$(h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot \frac{\partial}{\partial \theta_k} (\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$$

recall (MSE cost):  $\frac{\partial}{\partial \theta_k} (\mathbf{w} \cdot \mathbf{x}^{(i)} + b) = \begin{cases} x_k^{(i)} & (\theta_k = w_k) \\ 1 & (\theta_k = b) \end{cases}$

$$=^{(1)} \text{ we used: } \sigma'(z) = \sigma(z) \cdot (1 - \sigma(z))$$

# Gradient of the Cross-Entropy Cost (Generalised Perceptron)

Putting everything together:

$$\begin{aligned}\nabla_{\mathbf{w}} J_{CE}(\mathbf{w}, b) &= \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)} \\ \nabla_b J_{CE}(\mathbf{w}, b) &= \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})\end{aligned}$$

Compact form:

$$\nabla_{\boldsymbol{\theta}} J_{CE}(\mathbf{w}, b) = \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) \cdot \begin{bmatrix} \mathbf{x}^{(i)} \\ 1 \end{bmatrix}$$

Error Signal:  
contributes more for  
samples with mismatch.

# Update Rules for 'Generalised Perceptron' based on Cross Entropy Cost

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \cdot \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)}) \cdot \mathbf{x}^{(i)}$$
$$b \leftarrow b - \alpha \cdot \frac{1}{m} \sum_{i=1}^m (\hat{y}^{(i)} - y^{(i)})$$

**these formulas  
only work for  
single layer  
perceptrons!**

# Difference between GD update for CE and Perceptron Learning Algorithm?



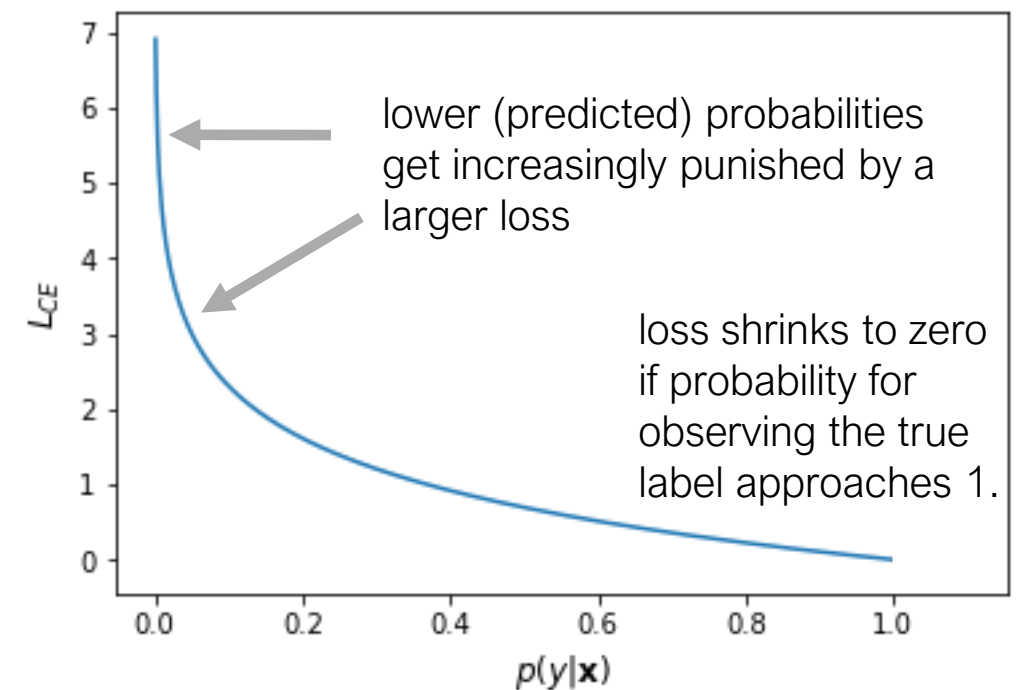
## Activity

In groups of 2, discuss the similarities and differences between the GD update rule for CE and the Perceptron Learning Algorithm

# Differences between MSE and CE Cost GD Update

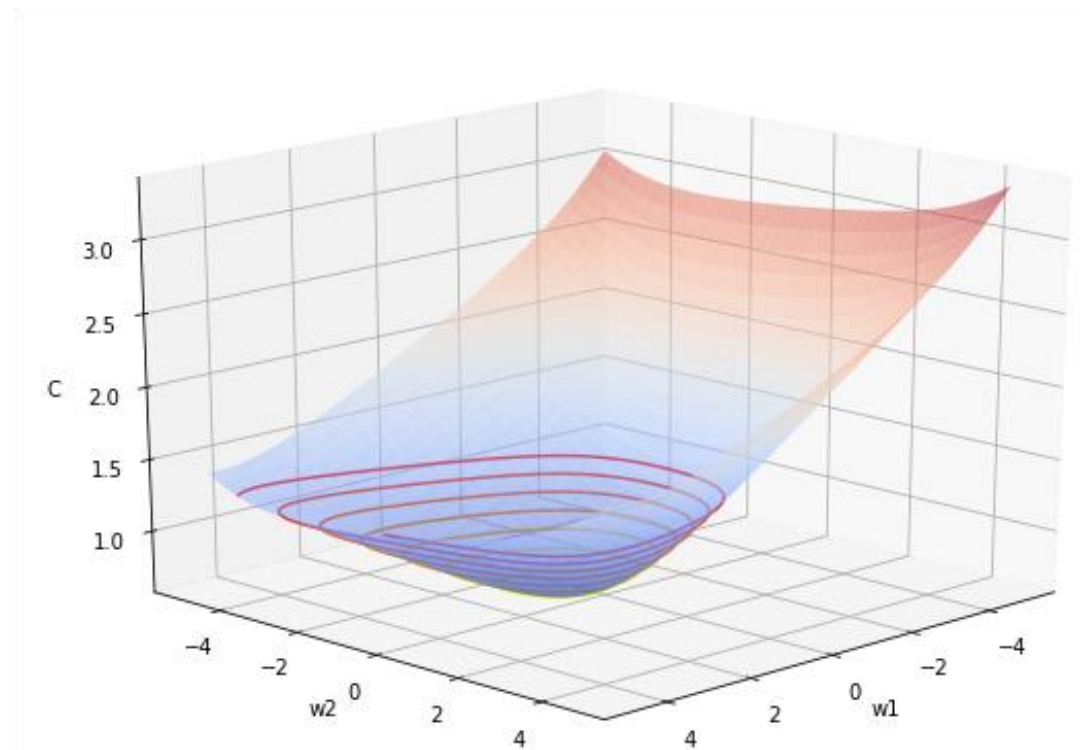
low probability values i.e.,  $p_{\theta}(y|\mathbf{x}) \rightarrow 0$  are increasingly punished for CE cost, which tends to plus infinity, while for MSE cost the values are limited to a maximum of 1

Cross-Entropy Loss

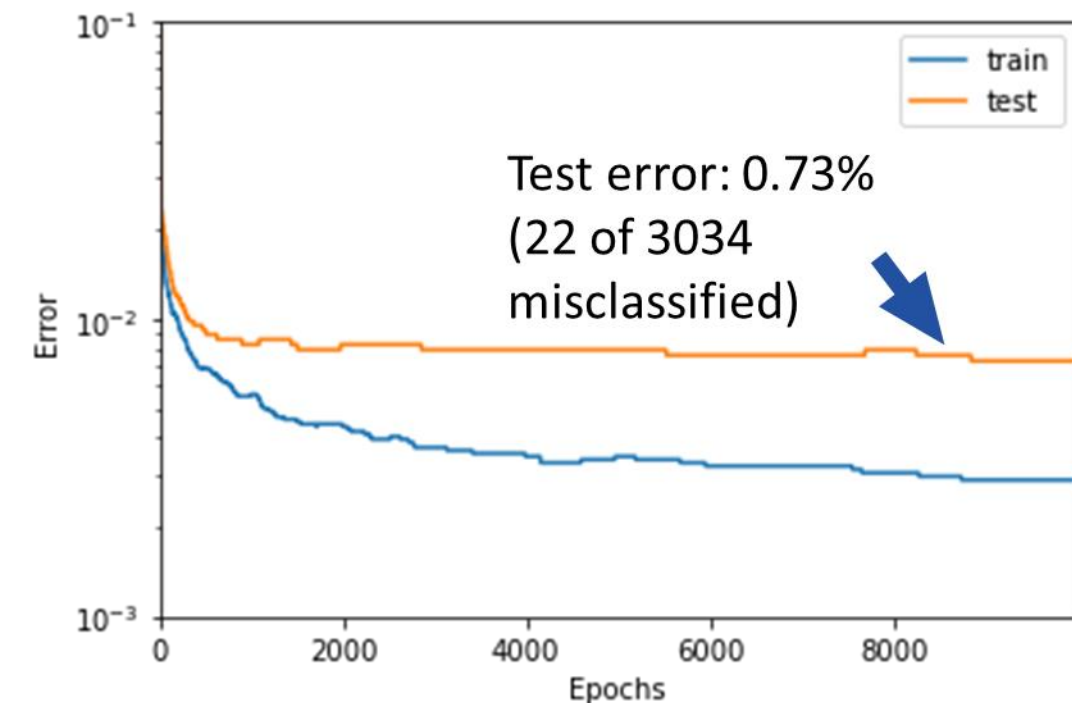
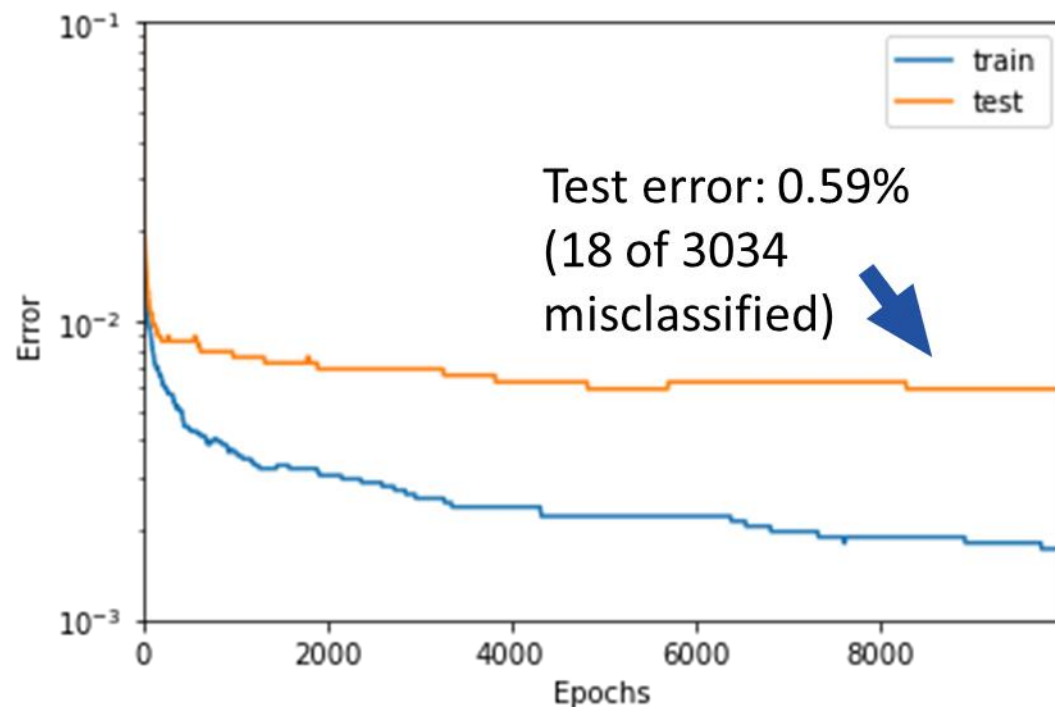
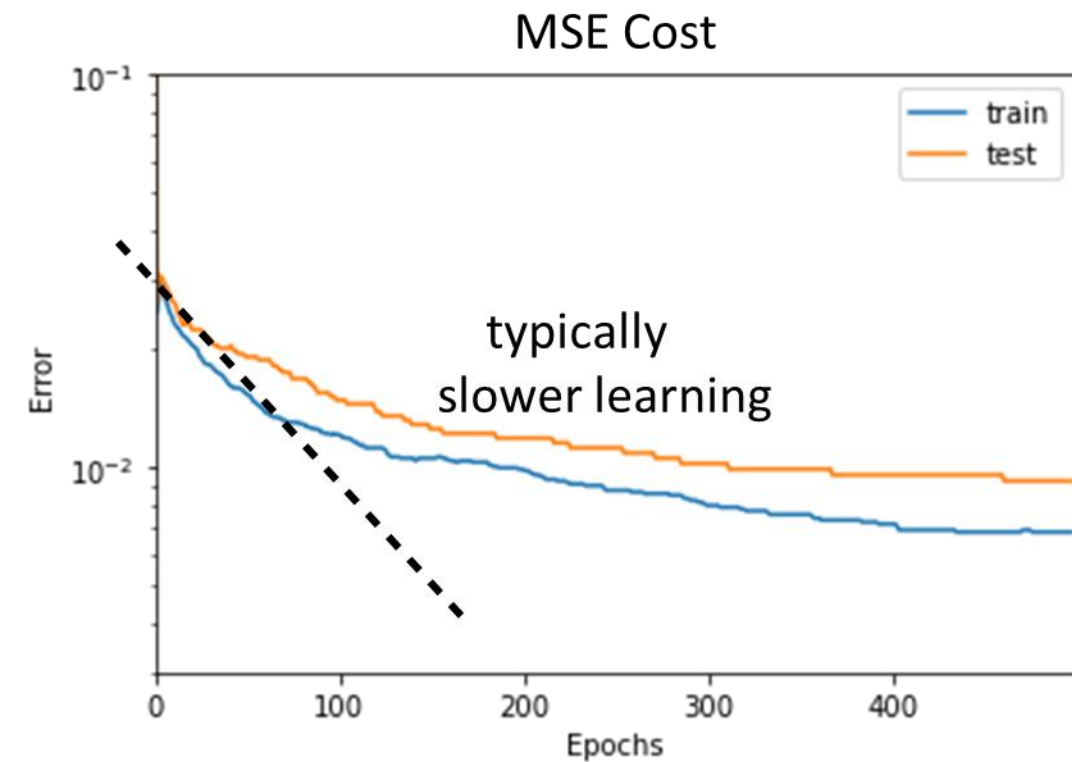
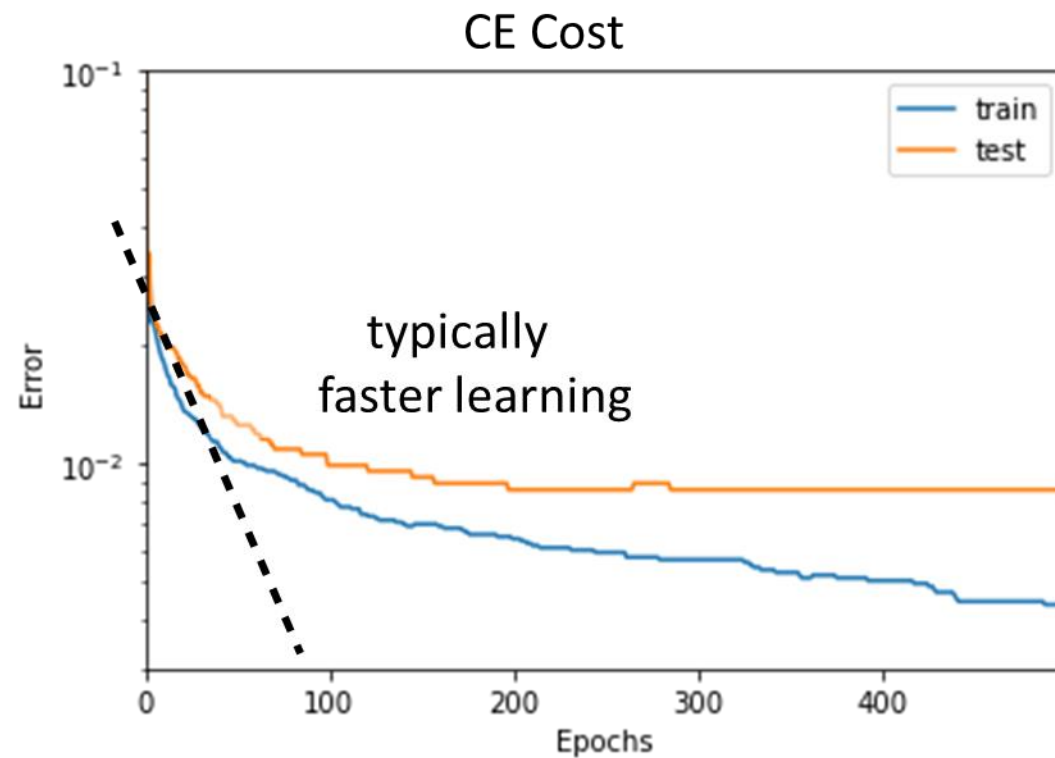


The cross entropy cost function for the generalised perceptron is a convex function, i.e. bowl-shaped.

Therefore, the above scheme is guaranteed to find the global minimum - as long as the learning rate is kept sufficiently small.



# Results for (1,7)-MNIST - MSE vs CE Cost



# Results for Different Digit-Pairs (i,j)

← j →

↑  
↓

	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%	

## Settings

- 5000 epochs
- 0.5 learning rate
- cross entropy cost
- initialised with zero values

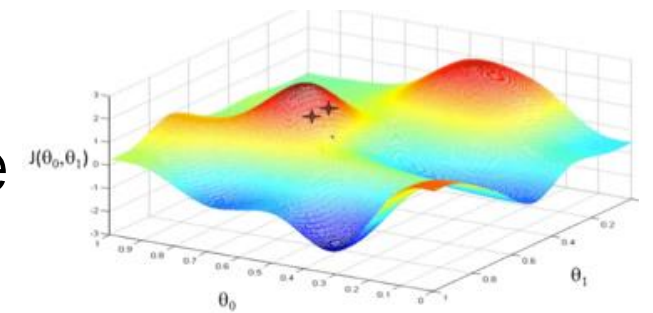
## Results

- Digits '3', '5' and '8' often involved in classifiers with high error rates: (3,5), (3,8), (5,8)
- For digits (5,3) the error rate is largest.
- Digits '0' and '1' are well distinguishable from all the others.



# Random Initial Parameters

- Results from different training runs with different initial parameters allow to judge the error bars in the estimates.



- Normalise (not only rescale) the input data!
- Choose random initial weights at proper scale:  $w_k \sim \frac{1}{\sqrt{n}} \mathcal{N}(0, 1)$

See Xavier  
initialisation  
in week 7

$\Rightarrow$  Logits  $z = \sum_{k=1}^n x_k \cdot w_k + b$  have unit standard deviation ( $\sim 1$ ).

10 runs for digits (5,3) with MSE cost !

- differently initialised weights ( $w_k$  as above), bias always 0
- 5000 epochs, learning rate 0.1

Result: average error  $\bar{e} = 4.1\%$  standard dev.  $\Delta e = 0.07\%$

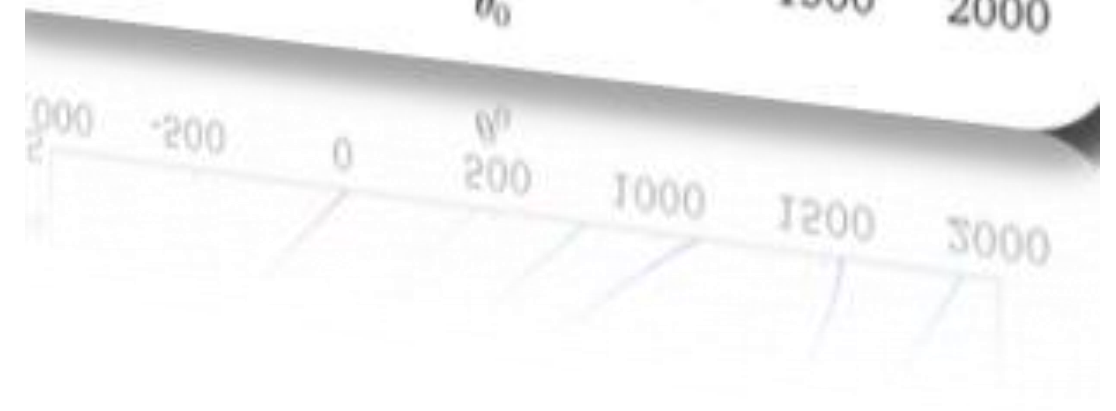
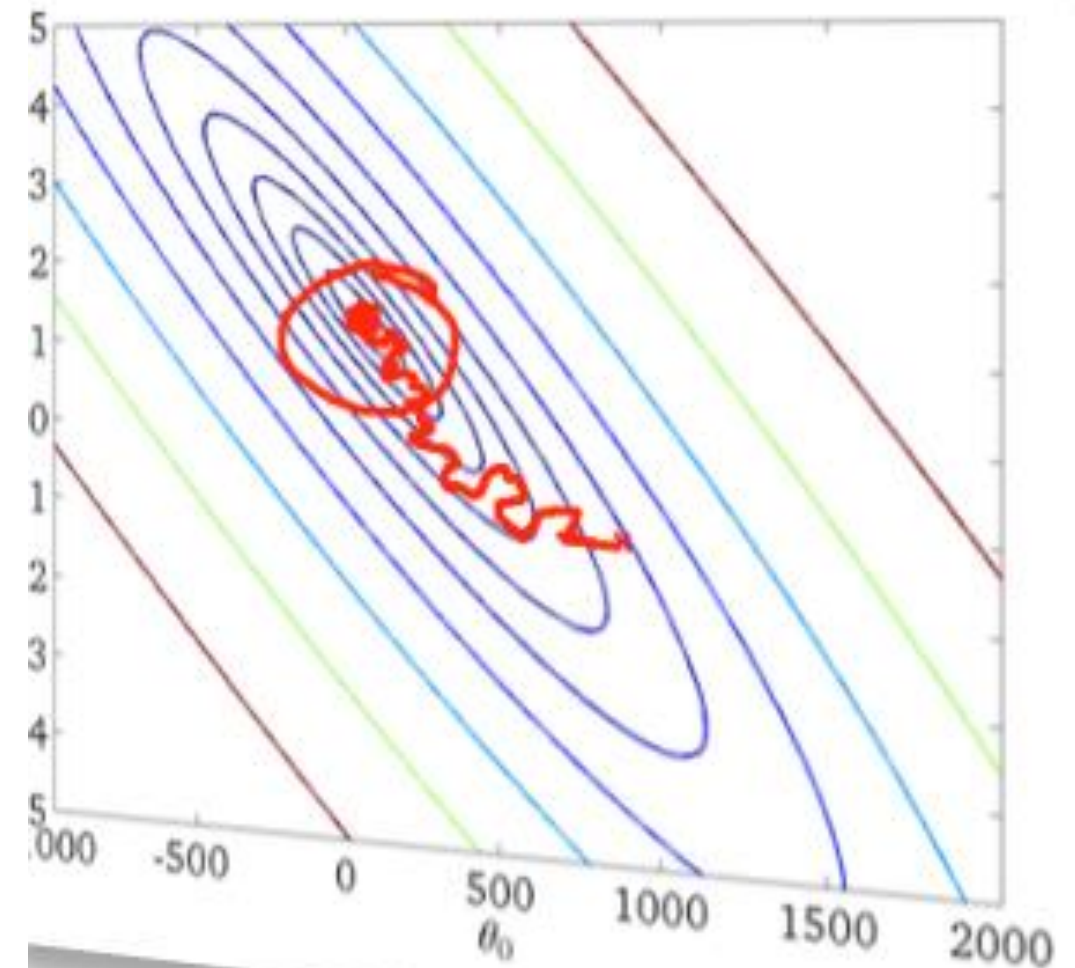
run	1	2	3	4	5	6	7	8	9	10
error rate	4.42%	4.65%	4.50%	4.42%	4.46%	4.46%	4.39%	4.39%	4.42%	4.50%



$\sim 100$  misclassified  
test images



# Stochastic and Mini-Batch Gradient Descent



# Idea behind Stochastic Gradient Descent and Mini-Batch Gradient Descent

- Cost functions ( $J_{\text{CE}}(\theta)$ ,  $J_{\text{MSE}}(\theta)$ ) are expressed as an arithmetic mean over per sample contributions ('loss').

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

- The update rule for the parameters can be defined by including less samples - possibly, the descent direction is captured already from less samples.

$$\Delta \theta = -\alpha \cdot \left( \frac{1}{m} \sum_{i=1}^m \nabla_{\theta} \log \left( p(y^{(i)} | \mathbf{x}^{(i)}, \theta) \right) \right)$$

- **Batch GD**: Averaging over all training samples
- **Mini-Batch GD**: Averaging only over a subset of training samples
- **Stochastic GD**: No averaging, just use a single randomly selected sample
- If not using Batch GD: No guarantee to move in parameter space in a direction that leads to smaller values of the cost function.

Sometimes,  
both called  
Stochastic GD

# Stochastic Gradient Descent (SGD)

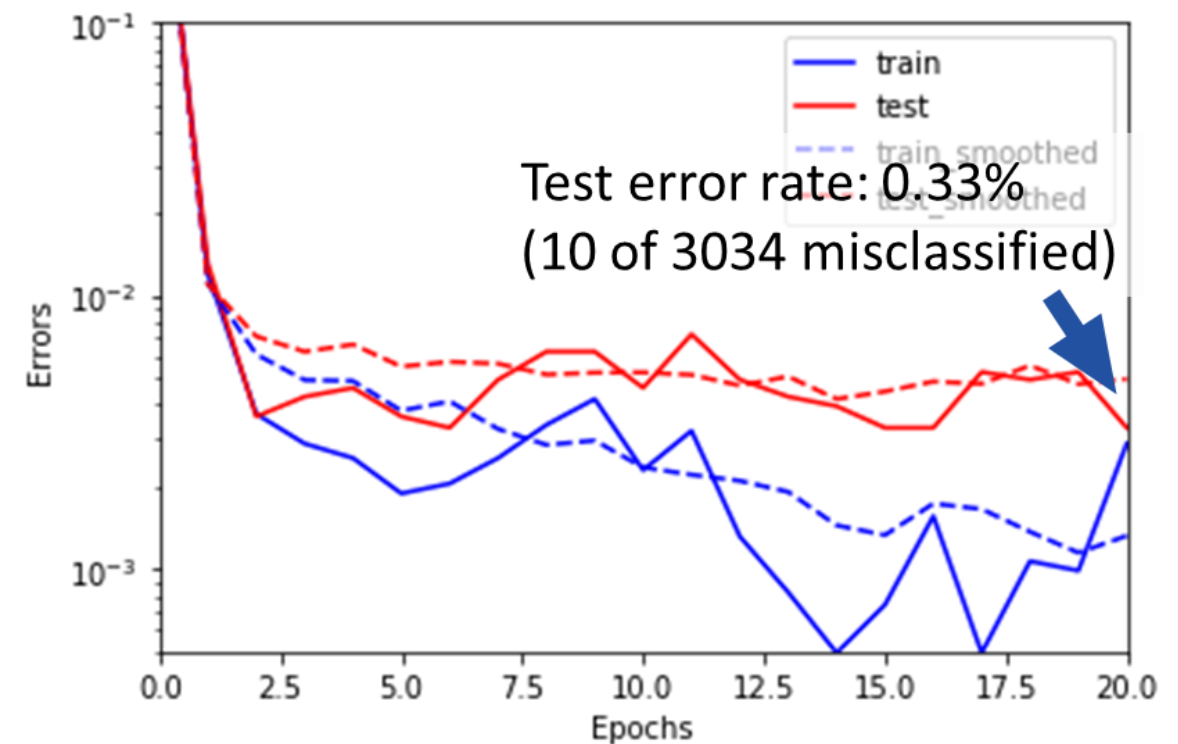
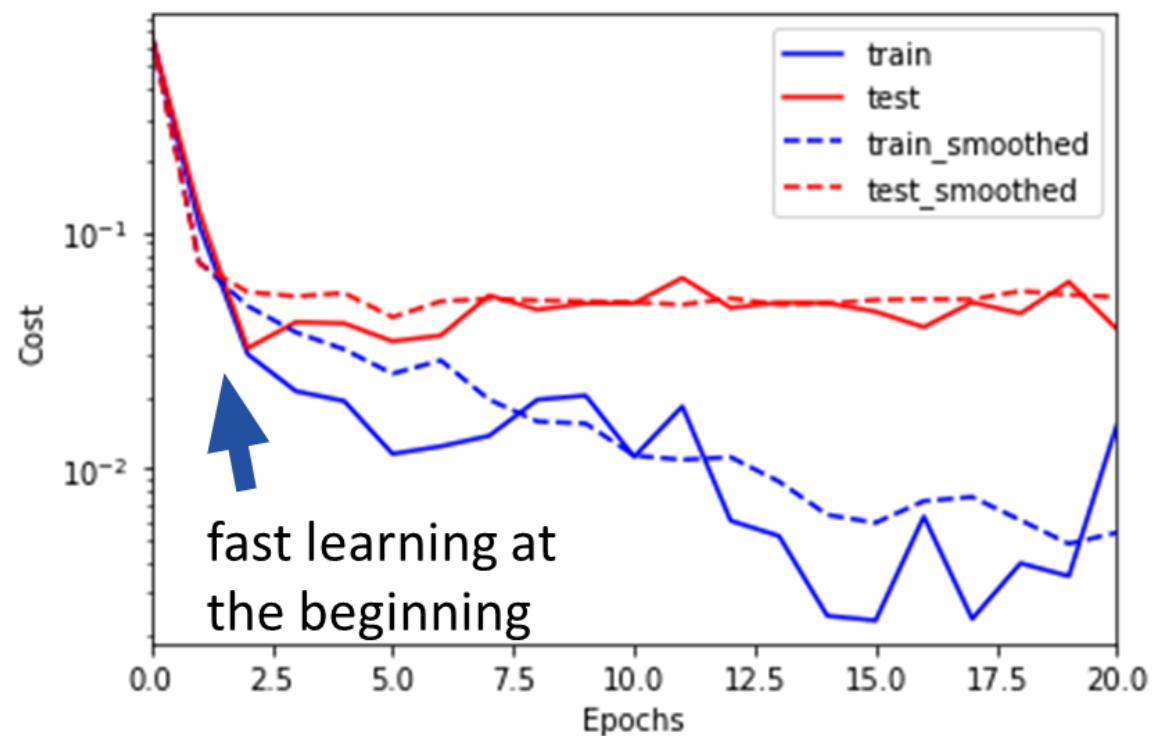
for the simple binary classification example with CE cost

1. Start with some initial value  $\theta_0$  for the parameter vector  $\theta_0 = (\mathbf{w}_0, b_0)$  .  
(e.g., random values or all 0)
2. Iteratively update the parameter vector  $\theta = (\mathbf{w}, b)$  by
  - a. Selecting one training sample randomly  $(\mathbf{x}^{(i)}, y^{(i)})$
  - b. Step in the negative gradient direction according to:
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \cdot (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)}) \cdot \mathbf{x}^{(i)}$$
$$b_{t+1} = b_t - \alpha \cdot (h_{\theta}(\mathbf{x}^{(i)}) - y^{(i)})$$
3. Stop when the change in parameter vector update step  $(\theta_t \rightarrow \theta_{t+1})$  is small

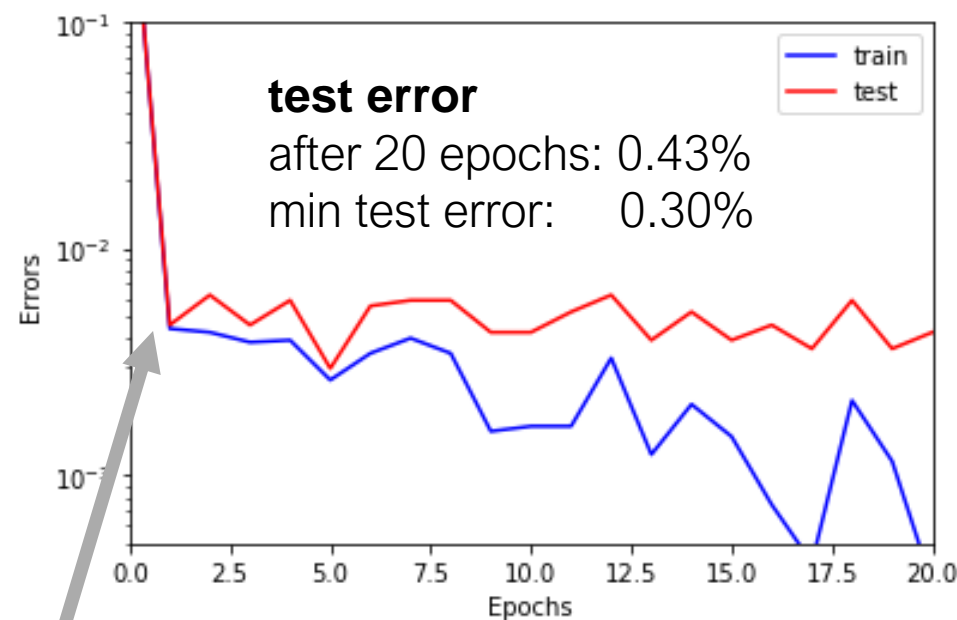
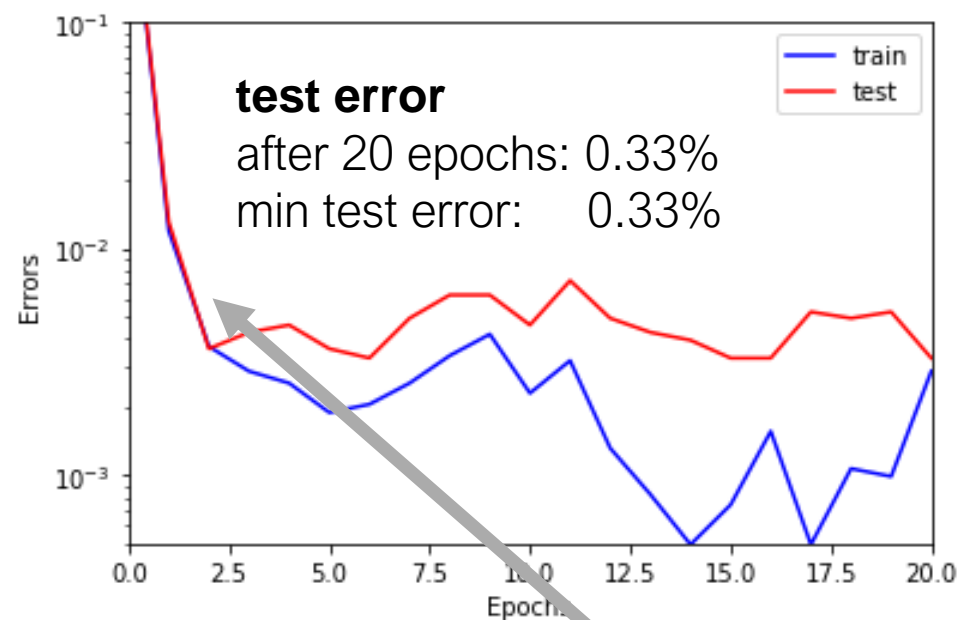
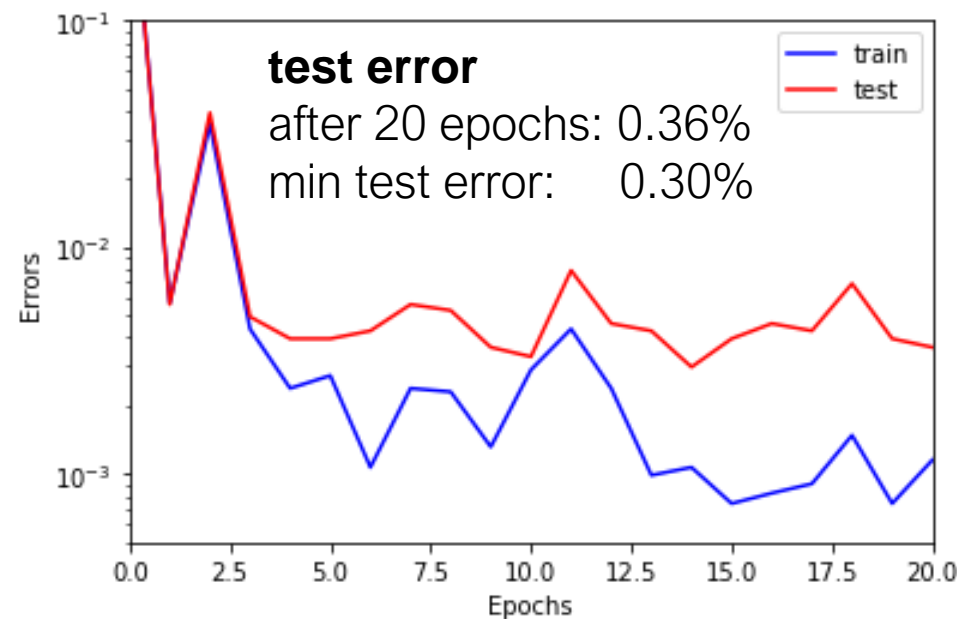
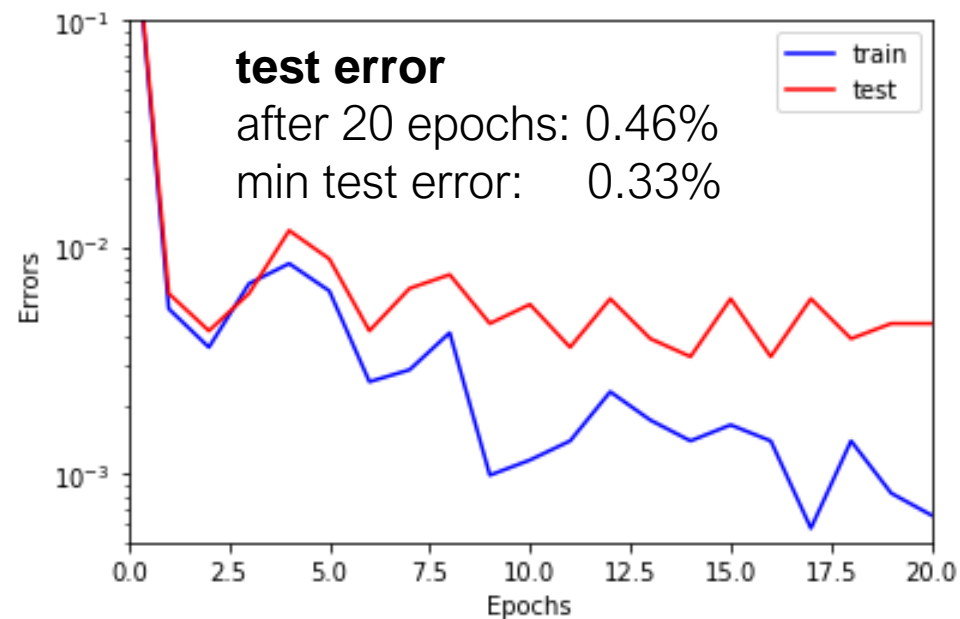
**Convergence** is trickier to observe as we may select favourable and less favourable points in the training set. In practice we may compute an "averaged" cost over the past U updates.

# SGD - Results for (1,7)-MNIST

learning rate 0.5



# SGD - Results for (1,7)-MNIST



**fast learning at  
the beginning!**

Learning curves  
for several runs  
with the same  
settings show  
different results.

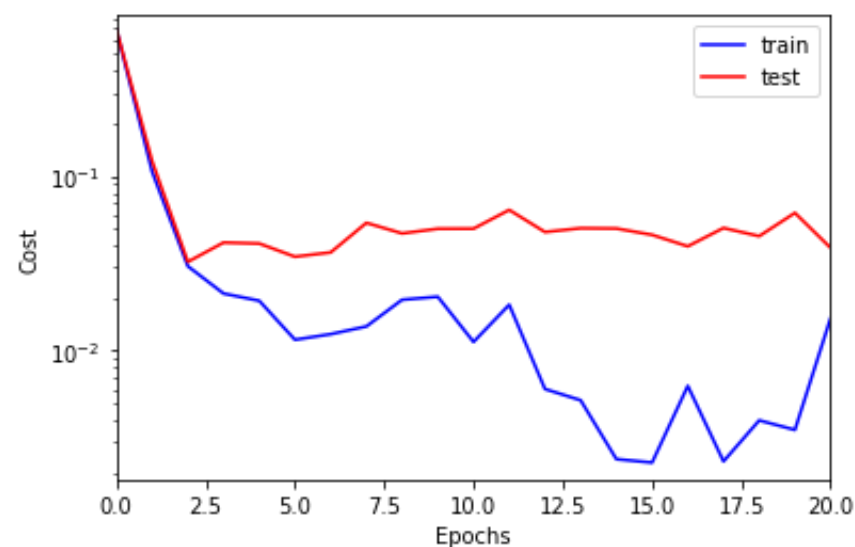
Why?

Results have a  
stochastic nature,  
interpret them  
accordingly!

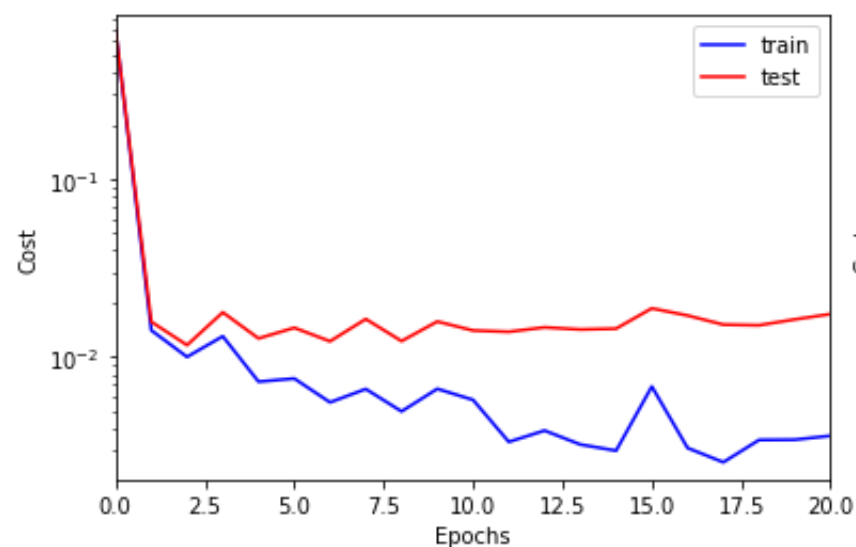
# SGD - Learning Rate

**Reduce learning rate  
when applying SGD!**

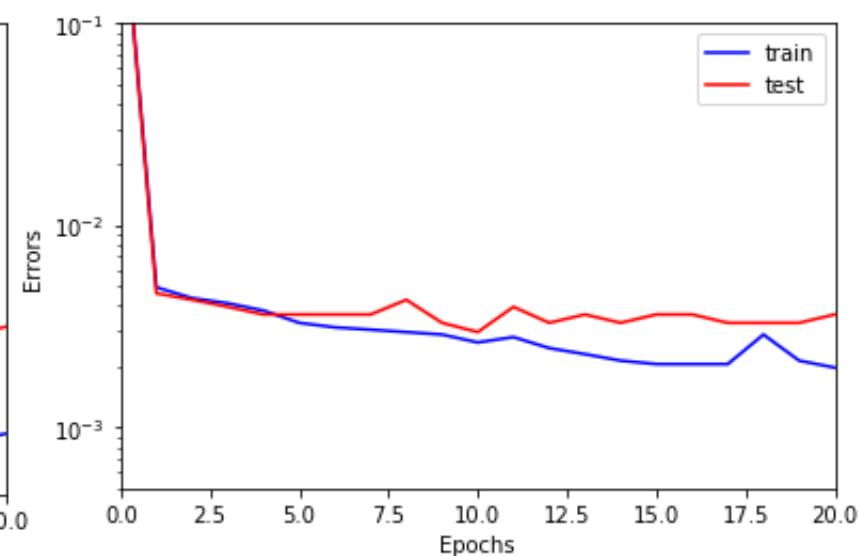
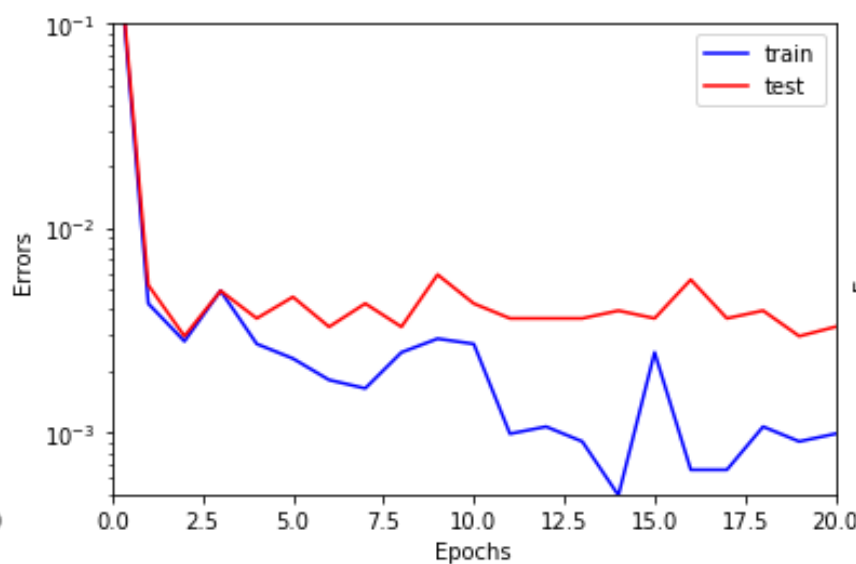
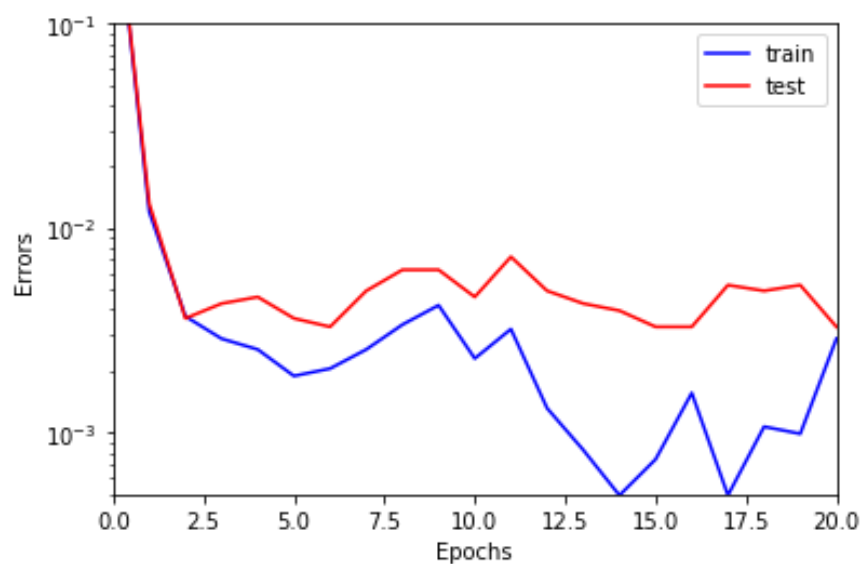
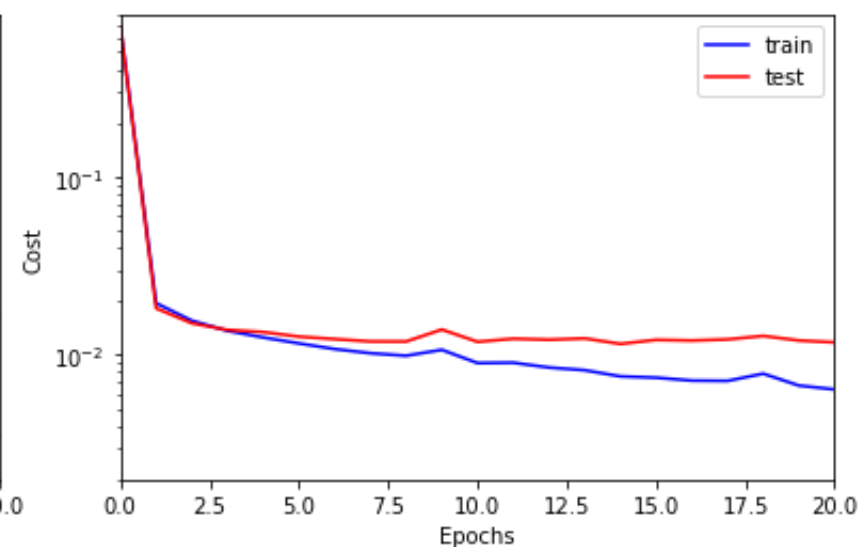
learning rate = 0.5



learning rate = 0.1



learning rate = 0.02





# Characteristics of SGD

## General Characteristics

- SGD tends to move in direction of a local minimum, but not always.
- It never converges like batch gradient descent does but ends up fluctuating around the local minimum. If we are close enough to the minimum this is not a problem.
- It allows for escaping local minima, thus has a regularizing effect. We will address this important issue later.
- The learning principle is generalizable to many other “hypothesis families” (same as BGD in this respect).

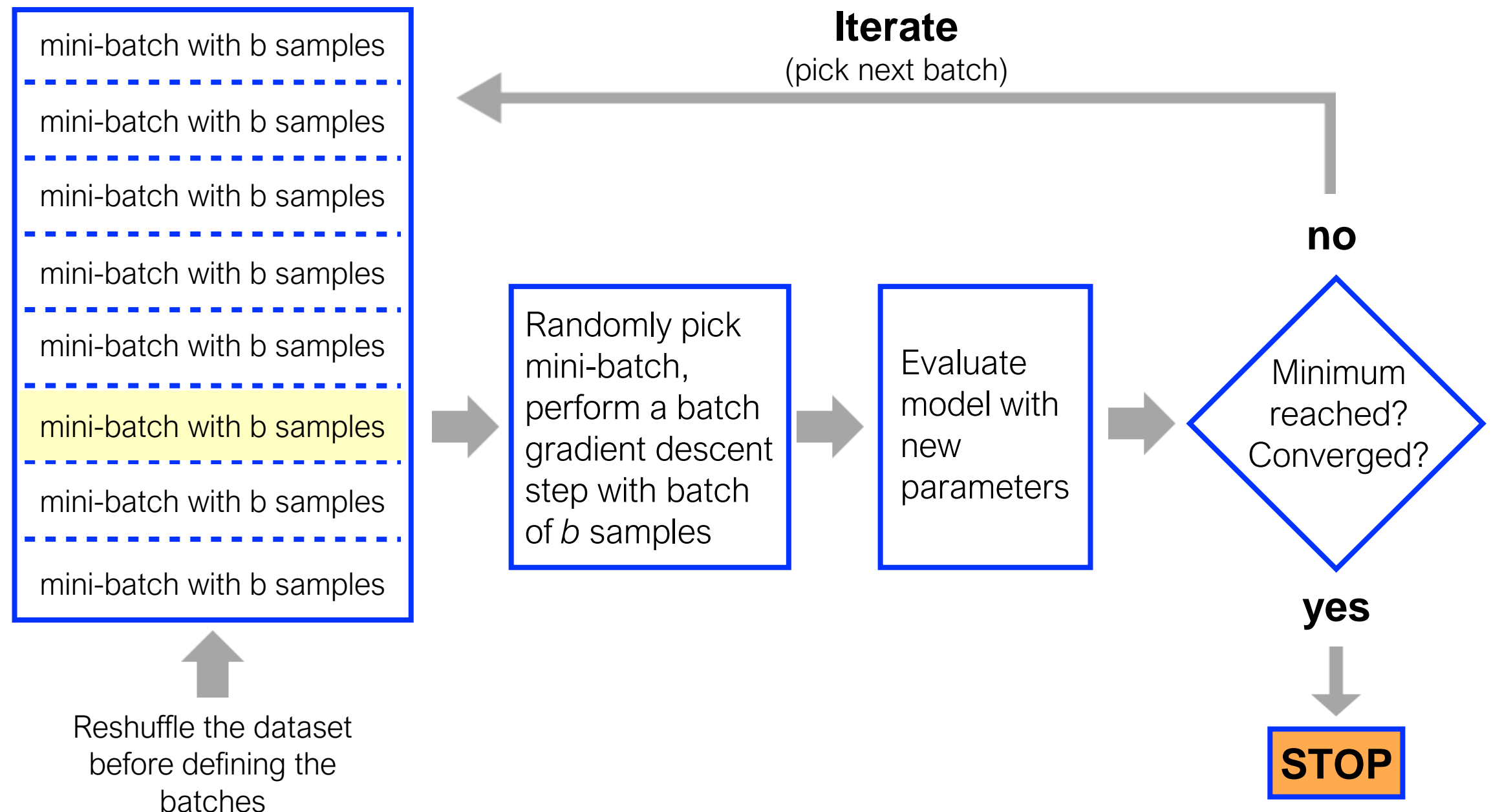
## Advantages

- SGD needs less epochs since parameters are updated for each training sample.
- It can handle very large sets of data and is great for learning on huge datasets that do not fit in memory (“out-of-core learning”).
- It allows for incremental learning (“online learning”) i.e., on-the-fly adjustment of the model parameters on new incoming data.

## Disadvantage

- SGD cannot easily be parallelised.

# Mini-Batch Gradient Descent Principle





# Mini-Batch Gradient Descent (MBGD)

for the simple binary classification example with CE cost

1. Start with some initial value  $\theta_0$  for the parameter vector  $\theta_0 = (\mathbf{w}_0, b_0)$  .  
(e.g., random values or all 0)
2. Iteratively update the parameter vector  $\theta = (\mathbf{w}, b)$  by
  - a. Selecting one mini batch with indices  $i_1, i_2, i_3, \dots$  i.e.,  
 $(\mathbf{x}^{(i_1)}, y^{(i_1)}), (\mathbf{x}^{(i_2)}, y^{(i_2)}), (\mathbf{x}^{(i_3)}, y^{(i_3)}), \dots$  randomly
  - b. Step in the negative gradient direction according to:

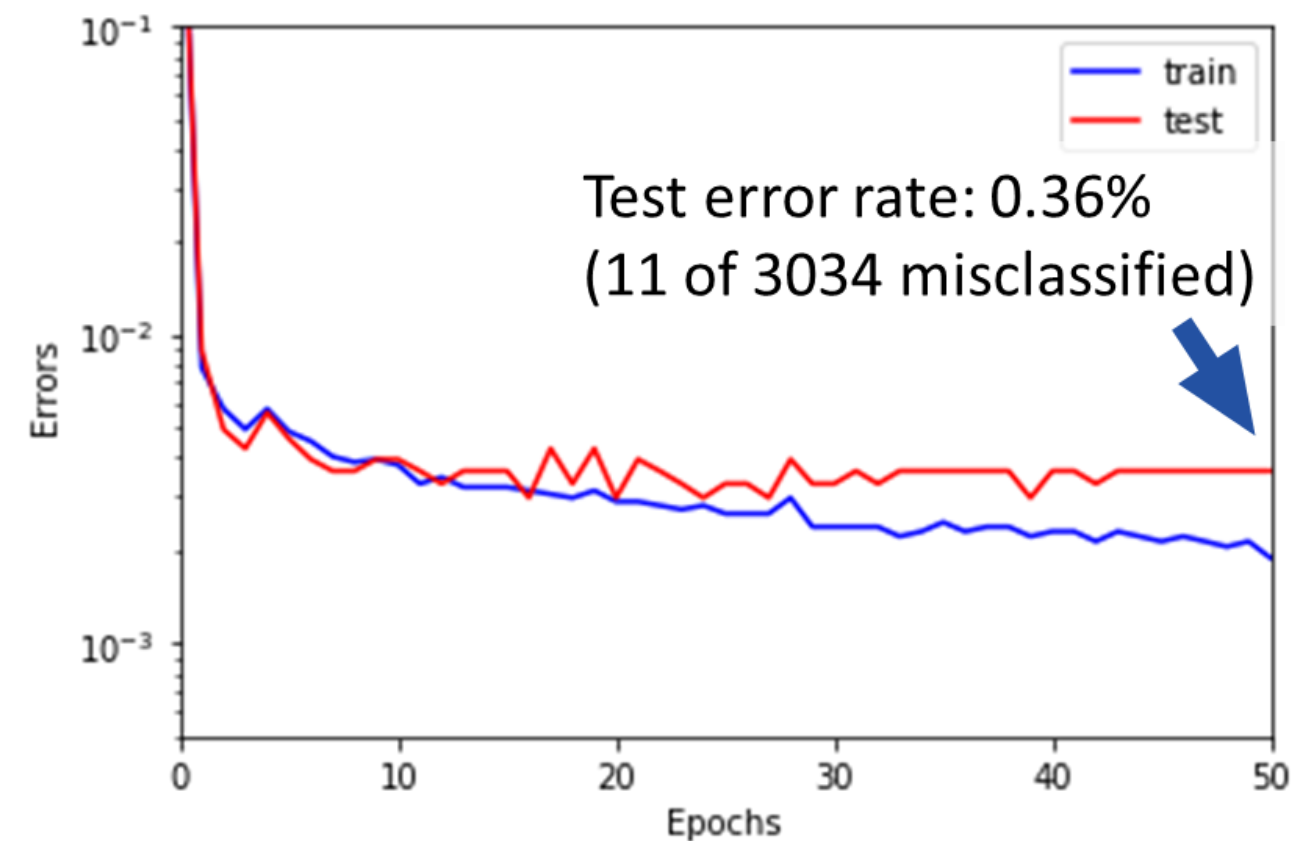
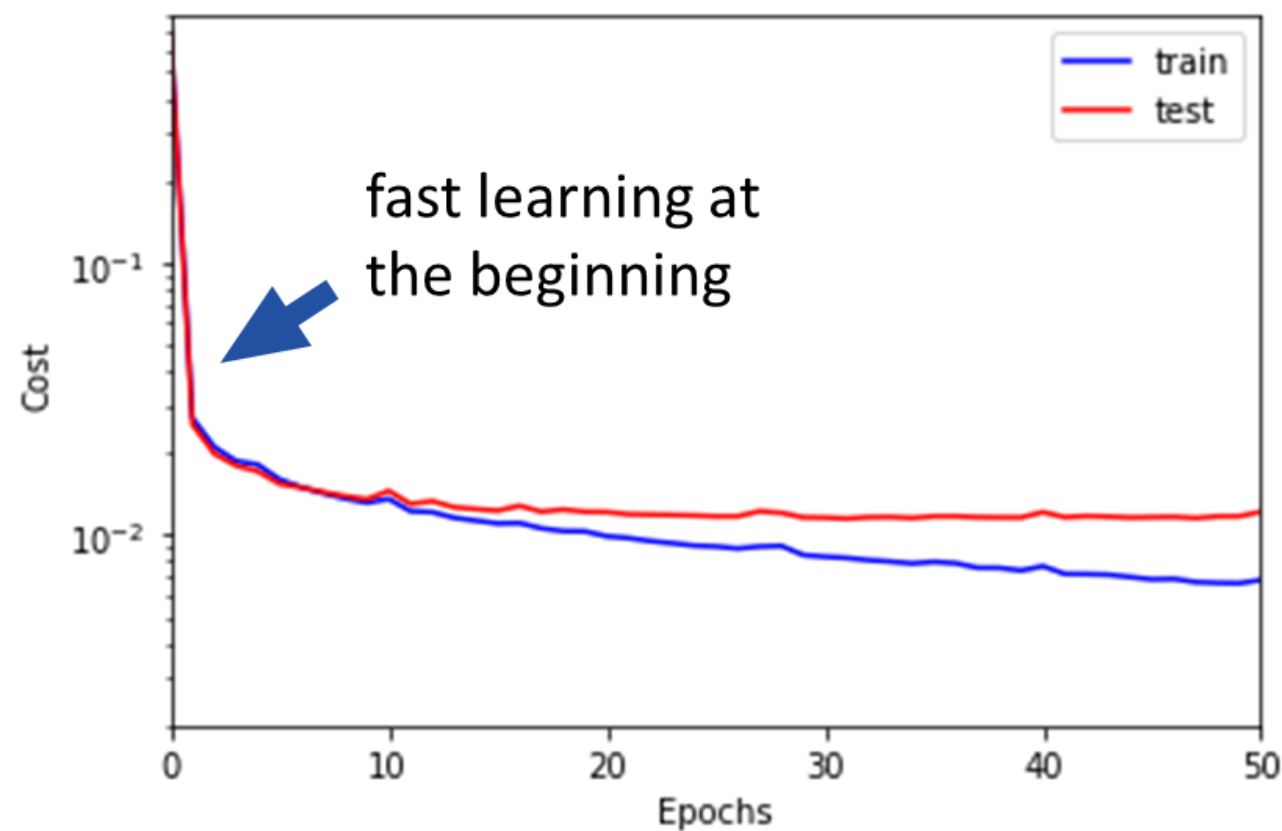
$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \cdot \frac{1}{b} \sum_{j=1}^b (h_{\theta}(\mathbf{x}^{(i_j)}) - y^{(i_j)}) \cdot \mathbf{x}^{(i_j)}$$

$$b_{t+1} = b_t - \alpha \cdot \frac{1}{b} \sum_{j=1}^b (h_{\theta}(\mathbf{x}^{(i_j)}) - y^{(i_j)})$$

3. Stop when the change in parameter vector update step ( $\theta_t \rightarrow \theta_{t+1}$ ) is small

# MBGD - Results for (1,7)-MNIST

learning rate 0.5, batch size = 64



# Characteristics of MBGD

## General Characteristics

- MBGD tends to move in direction of a local minimum, but not always.
- It ends up wandering closely around the local minimum.
- It allows for escaping local minima, thus has a regularizing effect. We will address this important issue later.
- As for BGD, the learning principle is generalisable to many other “hypothesis families”.

## Advantages

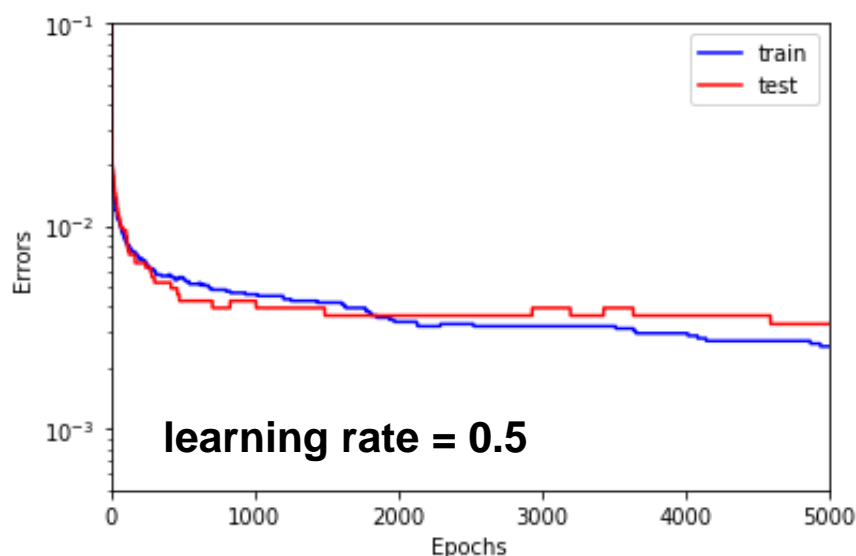
- MBGD is faster than BGD since parameters are updated for each mini-batch.
- It can handle very large sets of data and is great for learning on huge datasets that do not fit in memory (“out-of-core learning”).
- It allows for incremental learning (‘online learning’) i.e., on-the-fly adjustment of the model parameters on new incoming data.
- It can be parallelised on GPU and in HPC..

## Disadvantages

- We have an additional hyper-parameter, the batch size, that needs to be optimised.

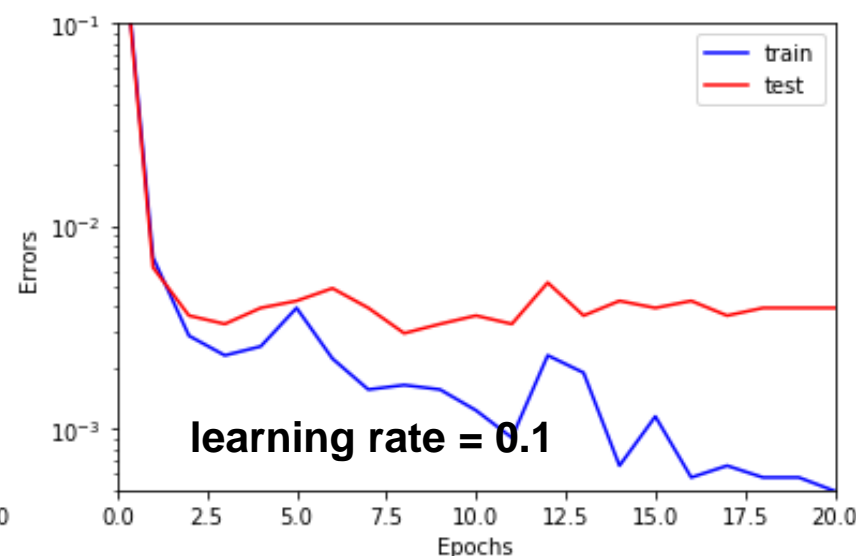
# Comparison of Gradient Descent Schemes

## BGD



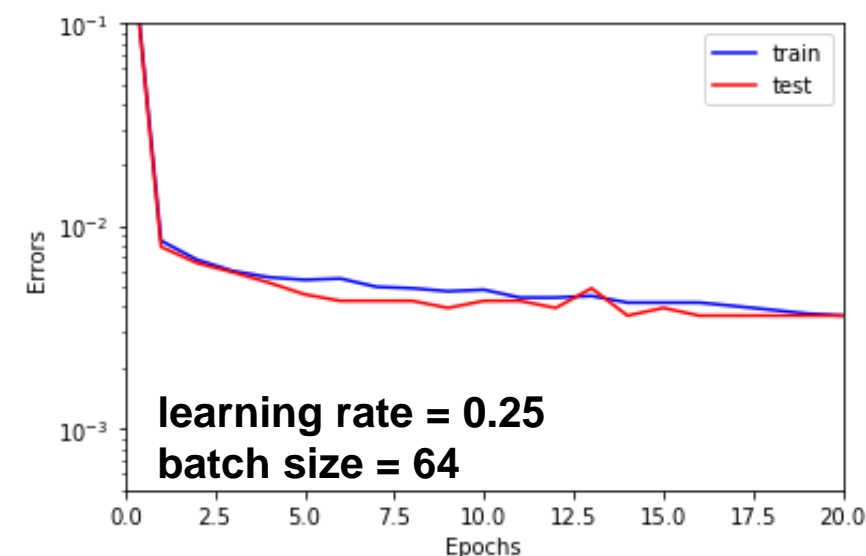
- Smooth.  
Not wiggling.
- Strictly decreasing cost.
- Many epochs needed.
- Choose larger learning rate.
- No out-of-core support - all data in RAM ( $\sim m$ ).
- Easy to parallelise.

## SGD



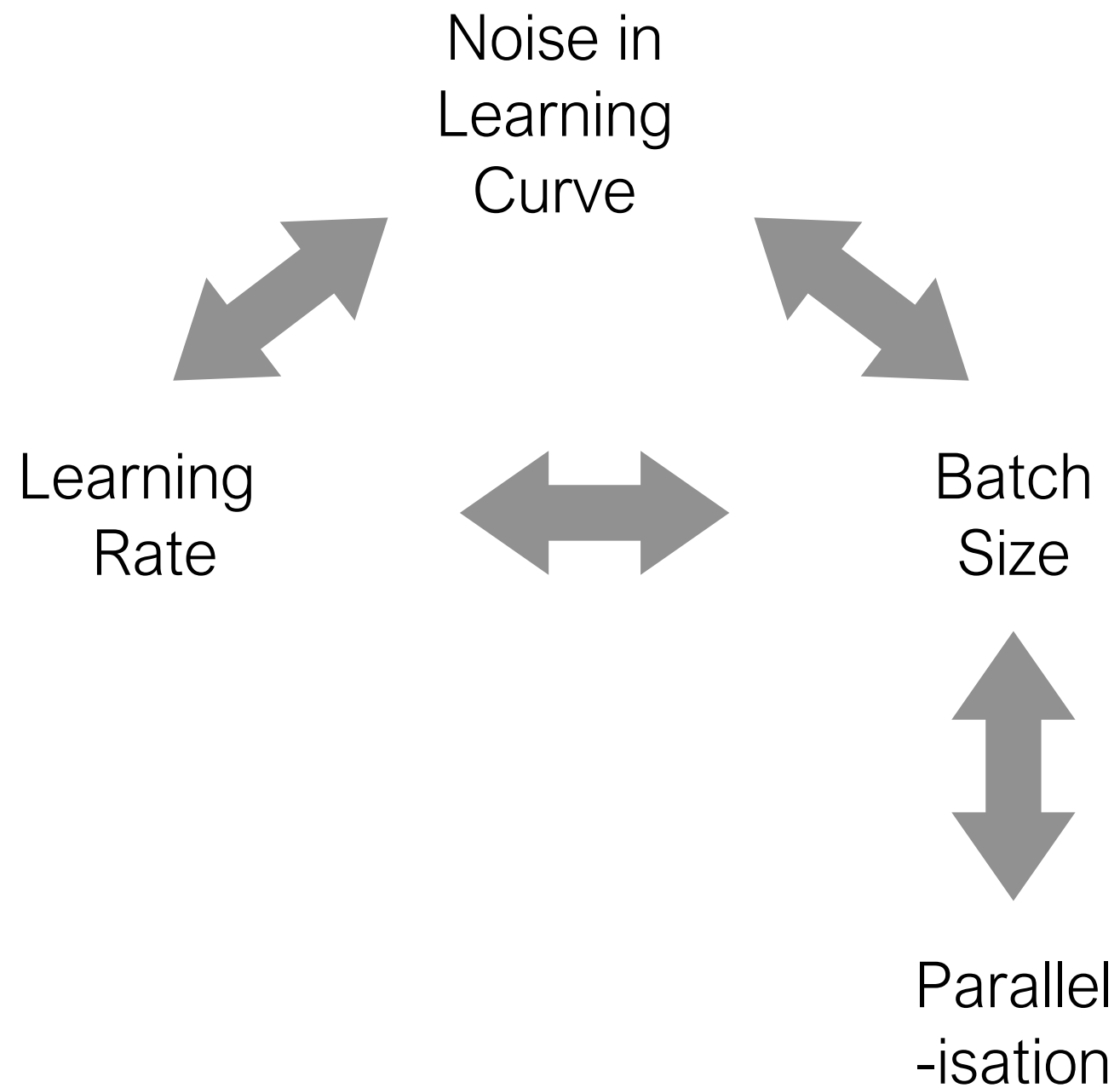
- Wiggling, needs smoothing.  
Wiggles around minimum.
- Not necessarily decreasing cost.
- Few epochs needed.
- Choose smaller learning rate.
- Out-of-core support - not all data to be kept in RAM of a single machine.
- Not easy to parallelise.

## MBGD



- Slightly wiggling.  
Wiggles around minimum.
- Typically decreasing cost.
- Less epochs than BGD, more than SGD needed.
- Choose medium learning rate (dependent on model)
- Out-of-core support - not all data to be kept in RAM of a single machine.
- Easy to parallelise.

# Learning Rate / Batchsize / Noise in Results





# Note on Entropy and Cross-Entropy

- Probability distributions encode information: For a distribution  $p(x)$  it is measured with **Shannon's Entropy** as

$$H(p) = - \sum_x p(x) \ln p(x) = E_{X \sim p} [-\ln p(X)]$$

## Basic Idea of Information Theory

The more one knows about a topic, the less new information one is apt to get about it. If an event is very probable, it is no surprise when it happens and thus provides little new information. Inversely, if the event was improbable, it is much more informative that the event happened. Therefore, the information content is an increasing function [log] of the inverse of the probability of the event ( $1/p$ ). Now, if more events may happen, entropy measures the average information content you can expect to get if one of the events actually happens. This implies that casting a die has more entropy than tossing a coin because each outcome of the die has smaller probability than each outcome of the coin. (**Source: Wikipedia**)

- Cross-Entropy** measures the information of one distribution w.r.t. to another

$$H(p, q) = - \sum_x p(x) \ln q(x) = E_{X \sim p} [-\ln q(X)]$$

Also related to  
Kullback-Leibler  
Divergence

- Cross-Entropy Cost** measures the information of the predicted distribution w.r.t. distribution of the samples used for training.