MASTER OF SCIENCE
IN ENGINEERING

Week 2:
Learning and Optimisation

TSM_DeLearn

Illustrated with Binary Classification of MNIST Digits

Jean Hennebert
Martin Melchior

Overview

Recap from Last Week, Plan for This Week

MNIST Dataset

Data Preparation

Model and Cost

Gradient Descent

Stochastic Gradient Descent

Recap from Last Week

From Machine Learning to Deep Learning

Successful Applications of Deep Learning

Perceptron, Learning Rule

Example Application Domains

Lip reading, lip sync from audio

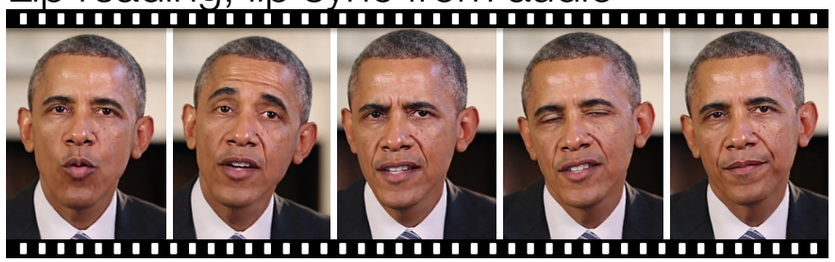
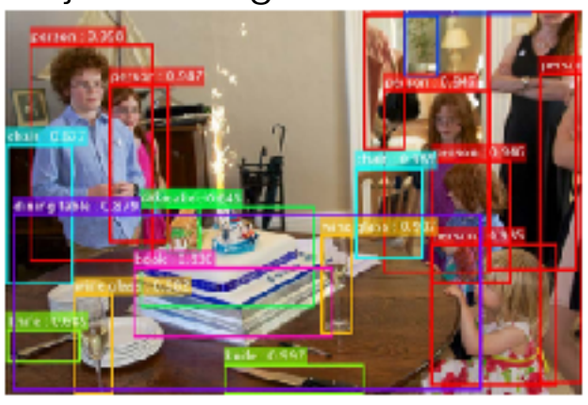


Image Tagging

Object Recognition

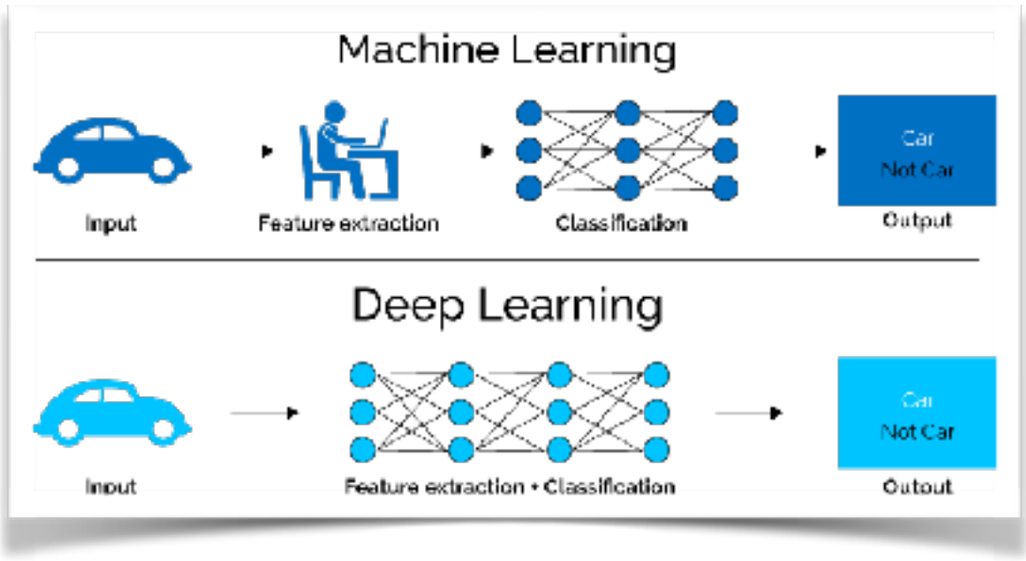


Question / Answering

Self-Driving Cars

Robot Control

From Machine Learning to Deep Learning



Deep Learning can be considered as a sub-discipline of machine learning. But generally, it requires less feature engineering than standard machine learning. The machines finds the features automatically as part of the learning.

Larger quantities of data
text, audio, images, videos, ...

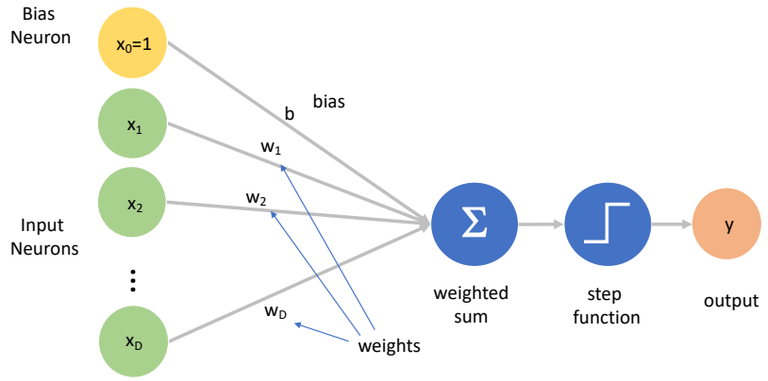
New algorithms
DBN, RBM, CNN ...

Deep Learning

Better computer performance
GPU, distributed computing ...

Perceptron, MLP

- Artificial Neuron (Perceptron, LTU)
  - Rosenblatt's Perceptron with weights and bias parameters
- Perceptron Learning Algorithm
  - Defines a learning scheme for a single perceptron or single layer perceptron.
  - Convergence only for linearly separable problems.
- Not all Problems are Linearly Separable
  - XOR, etc.
- Multi-Layer Perceptron (MLP)
  - Networks with perceptrons as elementary building brick can represent a much larger family of functions.
- However: How can general MLP's be learned?

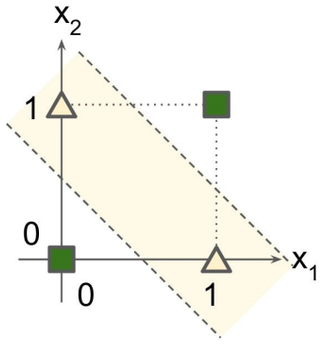


Learning algorithm:

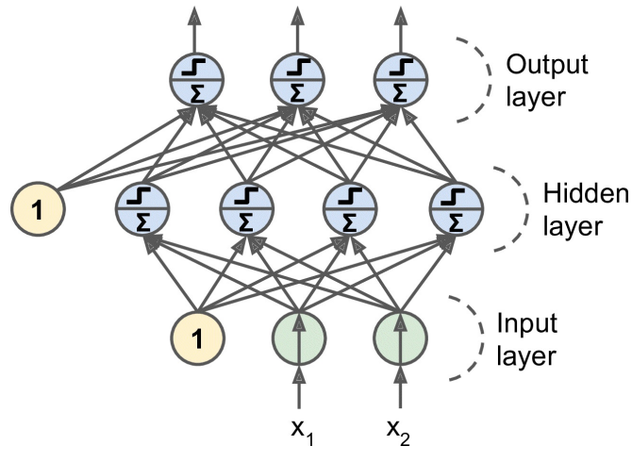
1. Initialise parameters — zero or small random number
2. Iterate by updating weights according to
   - A. Pick sample — $(\mathbf{x}^{(i)}, y^{(i)})$
   - B. Compute predicted value: — $\hat{y}^{(i)} = H(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$
   - C. Parameter update rule: — only update if $\hat{y}^{(i)} \neq y^{(i)}$:

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \cdot (\hat{y}^{(i)} - y^{(i)})\mathbf{x}^{(i)} \\ b &\leftarrow b - \alpha \cdot (\hat{y}^{(i)} - y^{(i)}) \end{aligned}$$

this term is zero or one!



XOR problem



Plan for this week

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

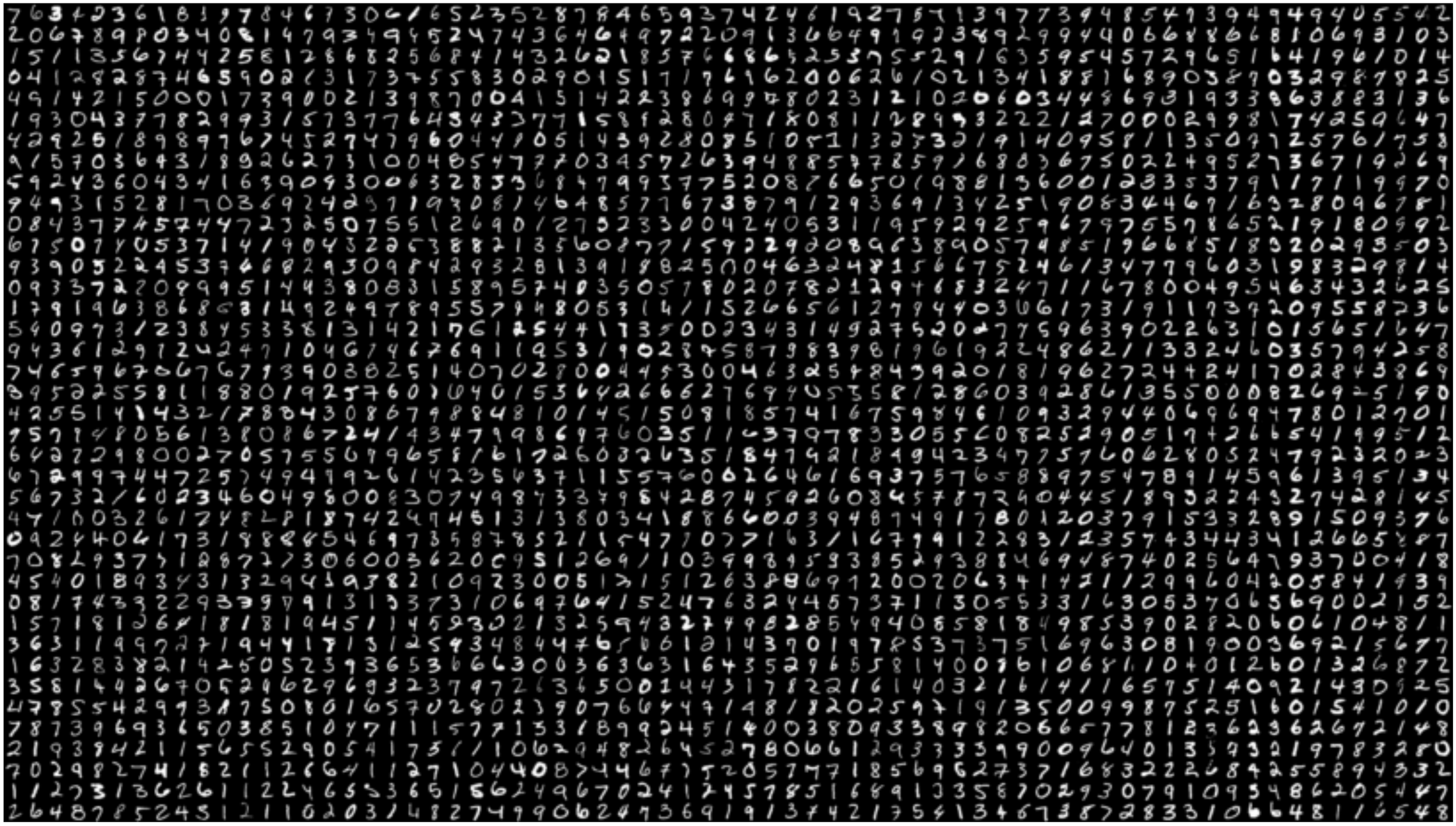
MNIST Dataset

Original and Lightweight MNIST Dataset

Public dataset widely used for testing and illustration

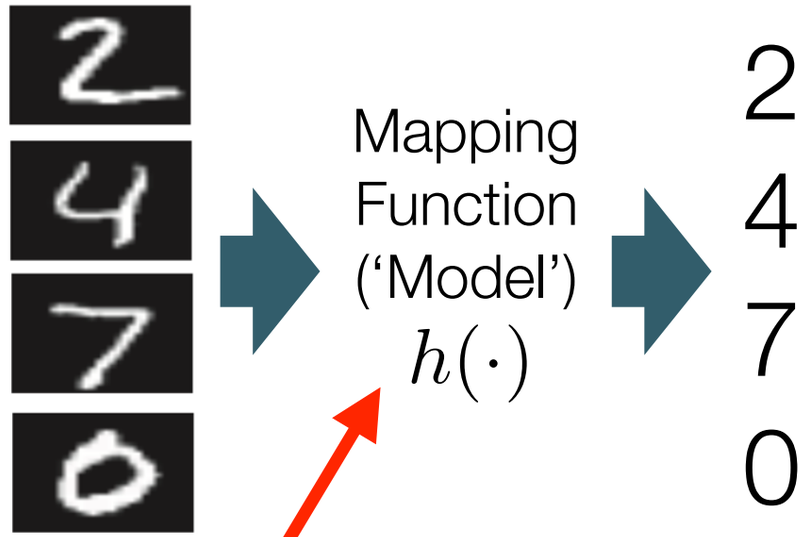
- **Original MNIST**
  - 70’000 images
  - 28x28-pixel
  - Available for download from mldata.org
    (scikit-learn, eras, tensor flow provide utilities to download it)

- **Lightweight MNIST**
  - 1’797 images
  - 8x8-pixel
  - Installed as part of scikit-learn library



MNIST Classification Problem

- General problem to solve:
  - Identify digit shown in the image.
  - Classification problem with 10 classes.



- Simplified problem considered first (for illustration purpose):
  - Identify whether the digit '5' is shown in the image.
  - Binary classification problem.

Is there a general scheme that can be used to train the model from the data?

MNIST Image (reduced version of scikit-learn)

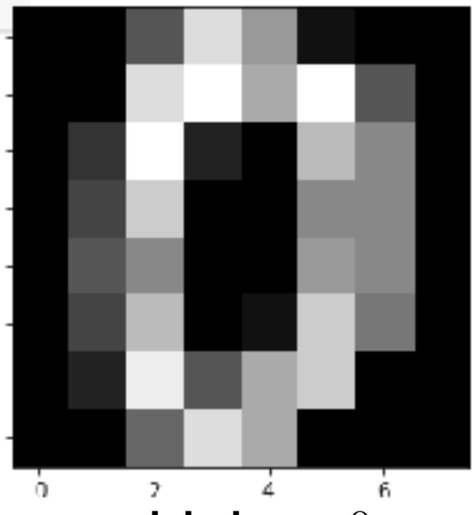
jupyter notebook (using the Lightweight MNIST)

```
from sklearn.datasets import load_digits
digits = load_digits()
x = digits.data
y = digits.target
print("Image Data Shape" , x.shape)
print("Label Data Shape", y.shape)

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

import matplotlib.pyplot as plt
%matplotlib inline
import numpy as np
plt.imshow(image, cmap=plt.cm.gray)
```

Loads all 1797 images and labels

A single image

```
Image Data Shape (1797, 64)
Label Data Shape (1797,)
[[  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.]]
```



grayscale values $0 \leq x_k \leq 16$

label $y = 0$

Build a model for mapping the array with 8x8=64 input values (greyscale value per pixel) to the labels

$$\hat{y} = h(\mathbf{x})$$

Use a learning algorithm that also works for not linearly separable problems and for multiplayer networks.

Data Preparation

Training and Test Sets
Data Normalisation

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:

	Small Datasets	**Large Datasets**
Train	70-80%	99%
Test	20-30%	1%

- 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 (using the reduced version from scikit-learn)

```
from sklearn.datasets import load_digits
from sklearn.model_selection import train_test_split

digits = load_digits()
x = digits.data
y = digits.target
x_train, x_test, y_train, y_test = train_test_split(x, y, test_size=0.25, random_state=0)

print(len(x),len(x_train),len(x_test))
```

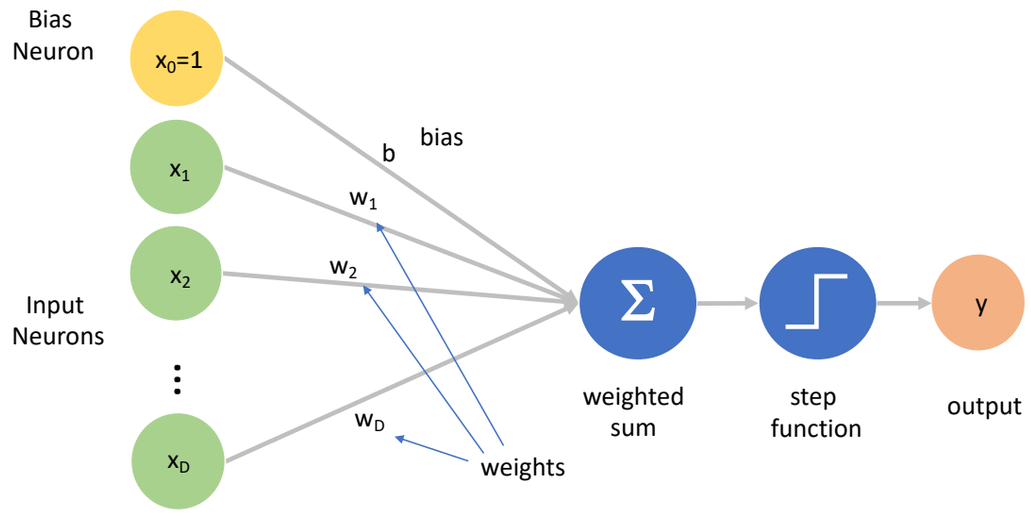
```
1797 1347 450
```

Data Normalisation, Feature Scaling

- Data contains features on different scales — numeric values may range over different orders of magnitude.
- For numerical stability, it's indispensable to bring the numeric values on similar scales. It improves the convergence properties of the learning algorithm.
- Normalisation (also known as feature scaling) is the process that brings the different inputs to a similar range and importance.
- In general, it applies to both, input data and output data.
- Different schemes available:
  - z-normalisation
  - min/max rescaling or normalisation

Why is Input Normalisation Important?

Weighted Sum: $\mathbf{z} = \mathbf{W} \cdot \mathbf{x} + \mathbf{b}$



Step function (such as the Heaviside function) is not dependent on scale, i.e. all the information on the scale should be absorbed when computing $\mathbf{z}$.

Two options:

(A) Adjust the weights to compensate for the scale observed in the inputs

scale of input features *x*

scale of the weights *w* such that $x \cdot w \sim 1$

(B) Normalise the inputs to make them scale-independent — optimisation algorithm can concentrate on finding the weights without taking care of the scale

scale of input features *x*

scale of the weights *w* can be initialised independently while still obtaining $x \cdot w \sim 1$

It turns out that with option (B) the optimisation algorithm works better(*) and the rule for initialising the weights is simpler.

(*) After normalisation the contours of the function to optimise are closer to circles.

z-normalisation

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

Shifting and **rescaling** the data so that a zero mean and a unit-variance is obtained.

where

$$\begin{aligned} \mu_k &= \frac{i}{N} \sum_{k=1}^{N} x_k^{(i)} \\ \sigma_k^2 &= \frac{i}{N} \sum_{k=1}^{N} \left(x_k^{(i)} - \mu_k \right)^2 \end{aligned}$$

Mean and standard deviation for the *k*-th component.

Remarks:

- The parameters μ_k, σ_k should be **computed on the training set** only. Computing them on the whole data set (including the test set) is considered as cheating.
- Normalisation of **image data**: Image data are normalised typically not per pixel. Mean and standard deviation are computed over all the pixels.

Min-Max Rescaling and Normalisation

Min-Max Rescaling

$$x_k'^{(i)} = \frac{x_k^{(i)} - \min_j \left(x_k^{(j)} \right)}{\max_j \left(x_k^{(j)} \right) - \min_j \left(x_k^{(j)} \right)}$$

Features are rescaled in the [0,1] range.

Min-Max Normalisation

$$x_k'^{(i)} = 2 \cdot \frac{x_k^{(i)} - \min_j \left(x_k^{(j)} \right)}{\max_j \left(x_k^{(j)} \right) - \min_j \left(x_k^{(j)} \right)} - 1$$

Features are rescaled and centred to [-1,1] range.

The same important remarks as on previous slide also holds here:

- image data: min/max computed over all pixels
- min/max computed on the basis of training set

Scaling and Centering

In general, normalisation consists of two steps :

- **Scaling**: Bring the data to the same scale.
  - Improves the convergence speed and accuracy of the learning algorithms.
- **Centering**: Balance the data around 0.
  - Improves the robustness of the learning algorithms. This is particularly important for deep learning networks.
  - Probably also improves the convergence speed and accuracy of the learning algorithms.

Sometimes, scaling (e.g. min-max-rescaling) is sufficient.

Remark: Normalisation of input data is closely related to batch normalisation (discussed in the scope of ***regularisation*** in week 5).

Normalisation for MNIST

MNIST data range given by integer greyscale values:

- Original MNIST: 0,…,255
- Lightweight MNIST: 0,…,16

Min-Max-Scaling

We only deal with a shallow network. Hence, we don't care about centering.

Computes the min and max over all the pixels.

```
import numpy as np
xmin = np.min(x_train)
xmax = np.max(x_train)
print(xmin, xmax)

x_train = x_train / xmax
x_test = x_test / xmax
```

```
0.0 16.0
```

```
print(x_train[0,:].reshape((8,8)))
```

```
[[ 0.      0.1875  0.8125  1.      0.5625  0.      0.      0.    ]
 [ 0.      0.625   0.9375  0.8125  0.9375  0.125   0.      0.    ]
 [ 0.      0.9375  0.25    0.25    1.      0.0625  0.      0.    ]
 [ 0.      0.      0.      0.3125  1.      0.125   0.      0.    ]
 [ 0.      0.      0.0625  0.875   0.8125  0.      0.      0.    ]
 [ 0.      0.      0.625   1.      0.3125  0.      0.      0.    ]
 [ 0.      0.25    1.      0.8125  0.5     0.625   0.5625  0.0625]
 [ 0.      0.125   1.      1.      0.875   0.75    0.5625  0.0625]]
```

Several libraries have built-in functionality to do normalisation: see e.g. http://scikit-learn.org/stable/modules/preprocessing.html#preprocessing-normalization.

Notations Time …

The notations can become quite intricate. We follow quite closely the notations of Andrew Ng on Coursera. You can find a summary in DL-notations.pdf on Moodle.

m	number of samples in the input dataset
n_x	number of input features, dimension of the input feature vector
$\mathbf{x}$	input feature vector of dimension n_x
x_k	k-th component of the input feature vector $(1 \leq k \leq n_x)$
$\mathbf{x}^{(i)}$	input feature vector of the i-th training sample $(1 \leq i \leq m)$
$x_k^{(i)}$	k-th component of the input feature vector of the i-th training sample
y	scalar output variable, also called target output or label
$\mathbf{y}$	output vector (or target output vector) of dimension n_y vector
y_k	k-th component of the output vector (or target output vector) $(1 \leq k \leq n_y)$
$\widehat{y}$	predicted output, as computed by the mapping function
$\widehat{\mathbf{y}}$	predicted output vector, as computed by the mapping function
$(\mathbf{x}, \mathbf{y})$	input sample (pair of input feature vector and corresponding label vector)
$(\mathbf{x}^{(i)}, \mathbf{y}^{(i)})$	i-th input sample of the input dataset $(1 \leq i \leq m)$

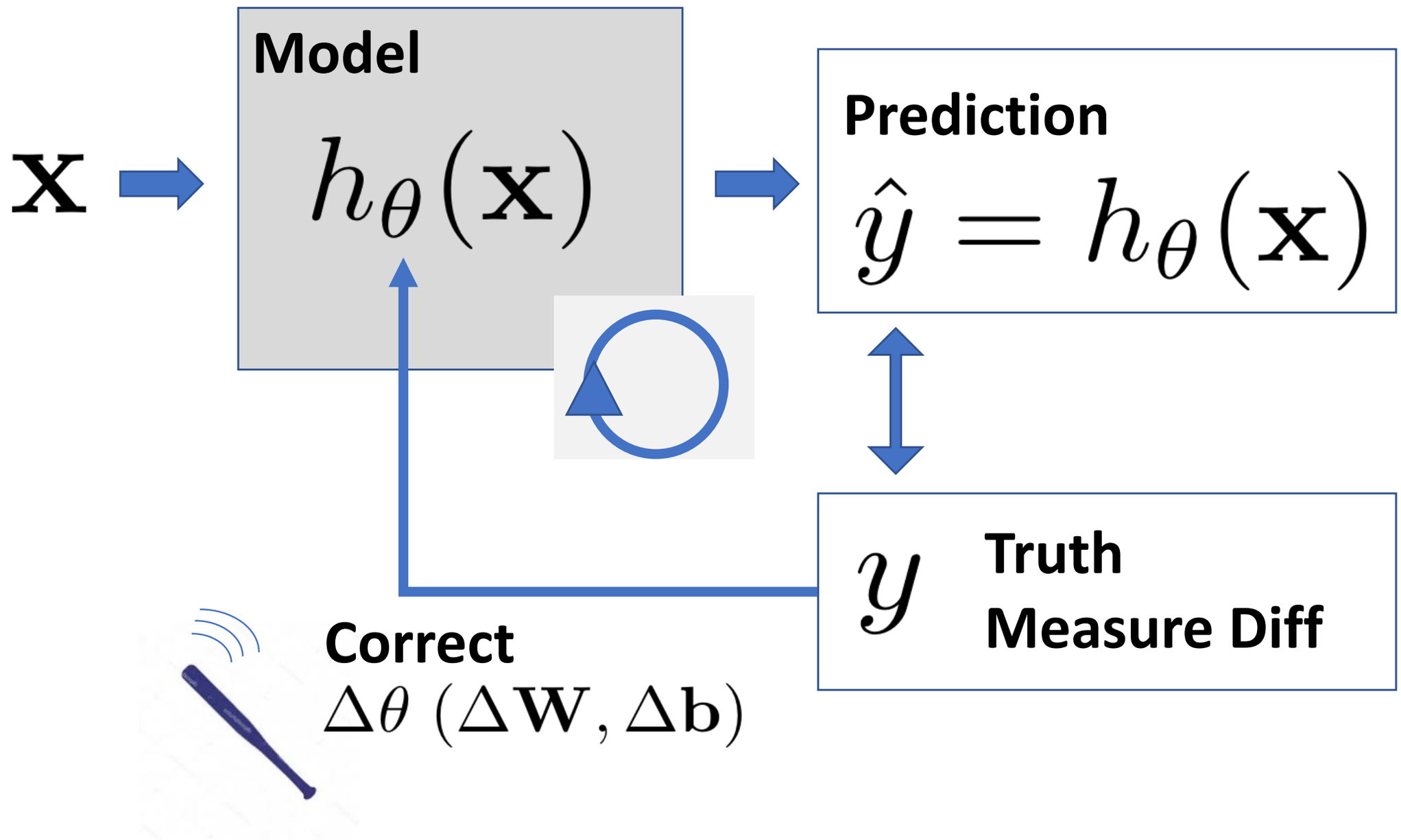
to be continued — UFFF !*$^^…

Model and Cost

Generalised Perceptron Model

Cost Function

Training a Model



Training a Model

Rosenblatt Perceptron of Last Week

Perceptron

Model

$\mathbf{x}$ → $h_\theta(\mathbf{x})$ →

Prediction

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

y **Truth**
Measure Diff

number of correctly classified samples

Correct
$\Delta\theta\ (\Delta\mathbf{W}, \Delta\mathbf{b})$

Perceptron learning rule

Training a Model

This Week

Generalised Perceptron, MLP

$\mathbf{x}$

Model

$h_\theta(\mathbf{x})$

Prediction

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

y **Truth**
Measure Diff

Cost Function

Correct
$\Delta\theta\ (\Delta\mathbf{W}, \Delta\mathbf{b})$

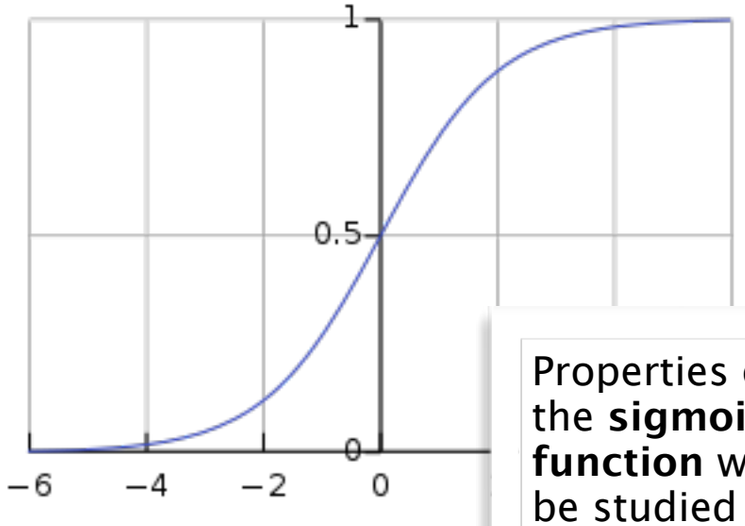
Gradient Descent

“Generalised” Perceptron

Generalised from Rosenblatt’s Perceptron.

1) **Smooth activation function: Sigmoid Function** $\sigma(z) = \frac{1}{1+e^{-z}}$
(replaces the ‘hard-limit’ Heaviside-function)

- Allows for a probabilistic interpretation:
Answer is given in form of a probability
of seeing the class or not (and not just a ‘yes/no’).
- Differentiable so that optimisation techniques
from calculus can be adopted.



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

2) **Gradient Descent Optimisation Algorithm used for Learning**

- Applicable also for training datasets that are not linearly separable and for MLPs

Model for Simplified (Binary) MNIST Classification

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

$\sigma(z) = \dfrac{1}{1 + e^{-z}}$ sigmoid activation function

$\theta = (\mathbf{w}, b)$ model parameters: weights and bias

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:

$$\begin{aligned} \text{yes} &: \quad \hat{y} = h_\theta(\mathbf{x}) \geq 0.5 \\ \text{no} &: \quad \hat{y} = h_\theta(\mathbf{x}) < 0.5 \end{aligned}$$

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 small / smaller / smallest.
- Different cost functions lead to different solutions.
- We will discuss two cost functions:

  (1) Mean Quadratic 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 and 0 only if all the prediction exactly match the truth.

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

Actually, there is another cost function better suited for classification problems, the cross entropy cost which introduce later.

Solving the Minimisation Problem

- There is no closed-form solution to the minimisation problem with the model $h_\theta(\mathbf{x})$ and the cost function $J_{MSE}(\theta)$ as given by the perceptron.
- 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).
- In general, *numeric methods* are needed to find a solution for the parameters θ. Typically, these work iteratively where a solution is iteratively approached.
- *Gradient Descent* is a family of optimisation algorithms widely used in practice.

Imagine Different Minimisation Strategies

Activity

- In groups of 2, address the following points through a short discussion:
  - Think about different strategies for how to find the minimum?
  - Imagine that you move on a landscape and try to reach the lowest point.
  - Will your strategies lead to the lowest point?

Batch Gradient Descent

Characteristics

Principles

Mathematical Formulation

Geometric Intuition about Gradient Descent

The **gradient**, a mathematical construct, gives the direction steepest ascent!

- Cost function defines a manifold in the parameter space.
- Imagine you are a hiker that walks on this manifold and you want to always go downhill — but you cannot see but only feel directions with the tip of your feet.
- Iteratively do the following until you reach **a** minimum:
  - Locally tip with your feet to identify the direction where it is going down steepest.
  - Make a step in this direction proportional to the slope.

Basic Principle of Gradient Descent

1) Start with some initial value for the parameter vector (for example random or 0): θ_0

2) Iteratively update the parameter vector by

   (i) Computing the gradient of the cost function $J(\theta)$ at the last position reached (θ_t) : $\nabla_\theta J(\theta_t)$

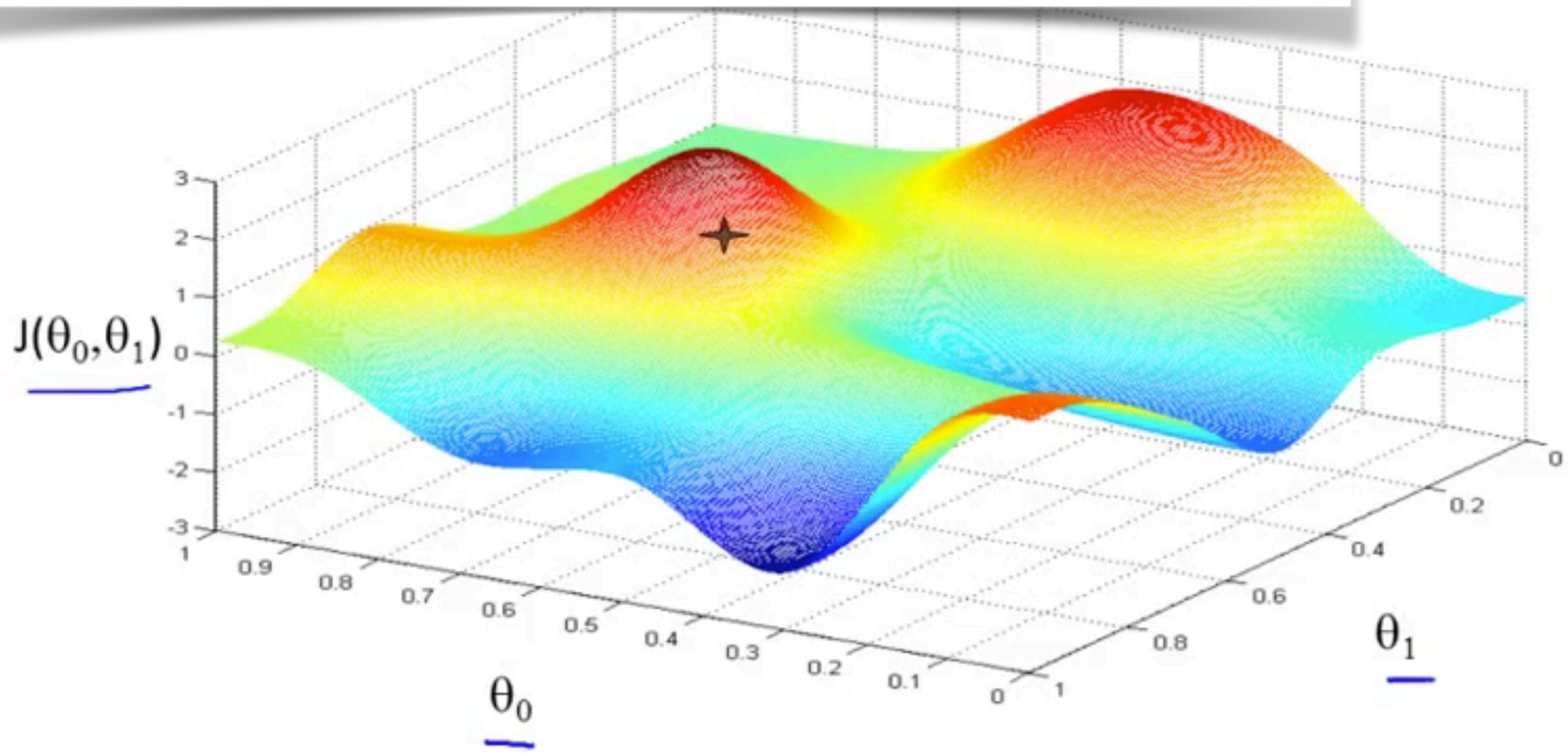
   (ii) Stepping in the negative direction of the gradient according to

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

(3) Stop when change in parameter vector small.

Gradient Descent Intuition

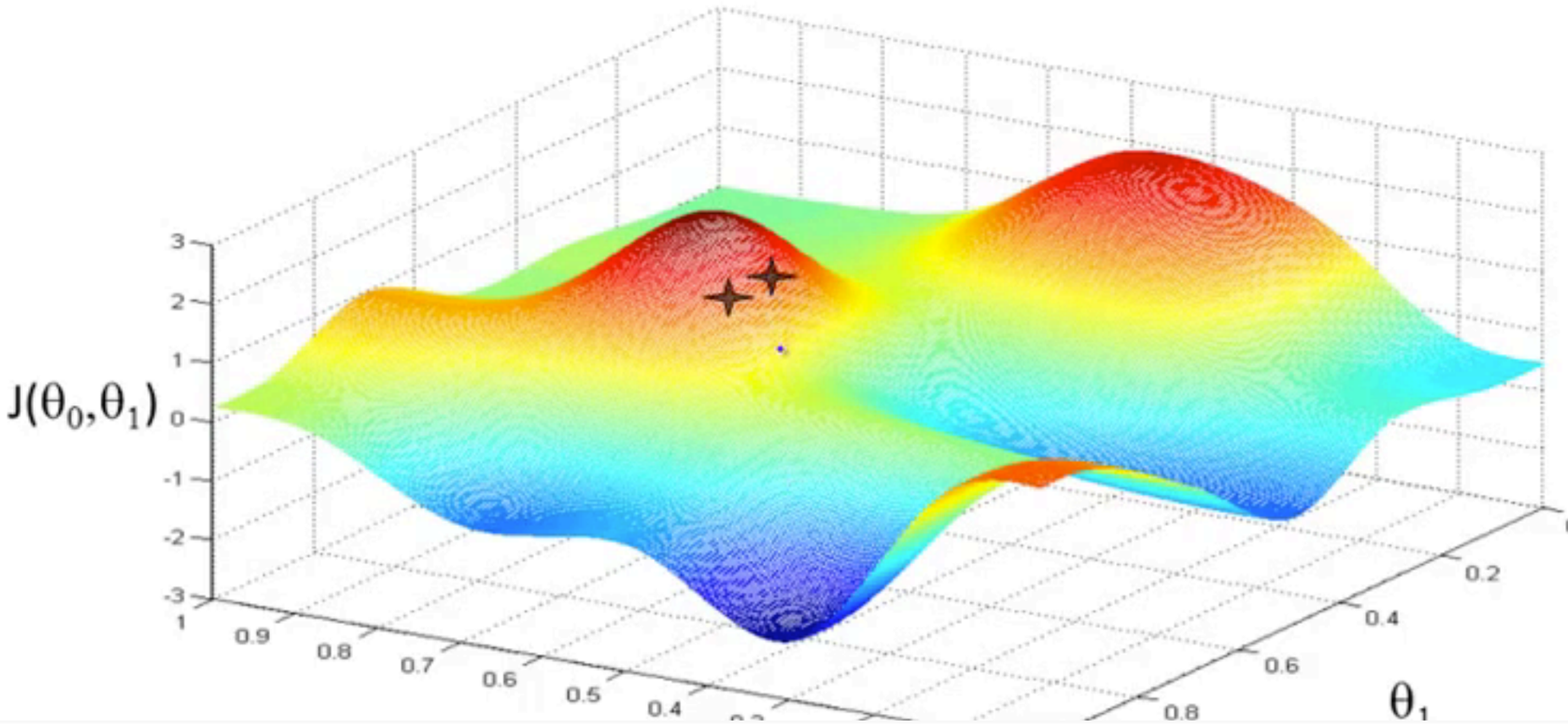
From the initial position, take steps in the opposite direction of the gradient, “going downhill” until a minimum is reached.



Source: Andrew Ng - Machine Learning class - Stanford

Andrew Ng

Gradient Descent Intuition



Source: Andrew Ng - Machine Learning class - Stanford

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.

In the case of a single unit perceptron and a sigmoid activation function, the J() function is convex, i.e. “bowl shaped” with a unique minimum, so we don’t need to care by getting stuck in a local minimum in this case.

Ng

Characteristics of Gradient Descent

- Learning principle does not depend on the type of model $h_\theta(\mathbf{x})$ as long as gradients can be computed (e.g. also works for deep neural networks).
- Learning principle works ‘locally’ and finds local wells — not designed to find global minima.
  - In a well (local minimum), the gradient is zero. There, it cannot distinguish between local minima, local maxima or saddle points (referred to as critical points).
  - Learning schema is iterative — iteratively approaches a critical point. There it fluctuates around the critical point.
  - Generally, the learning schema is not guaranteed to converge (e.g. if learning rate is chosen too large, see later).

How to Compute Gradients?

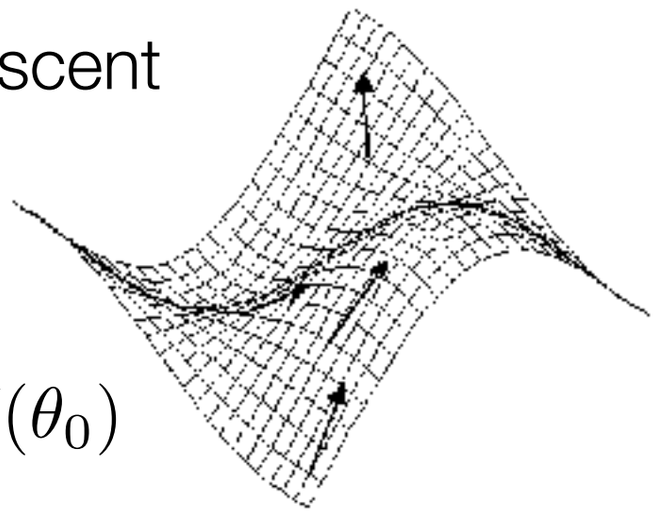
Lots of Calculus Needed !

Partial Derivative of Function $J(\theta) = J(\theta_1, \ldots, \theta_n)$ w.r.t. θ_k :

$$\frac{\partial J}{\partial \theta_k} = \lim_{\epsilon \to 0} \frac{1}{\Delta \theta_k} \left(J(\theta_1, \ldots, \theta_k + \Delta\theta_k, \ldots, \theta_n) - J(\theta_1, \ldots, \theta_k, \ldots, \theta_n) \right)$$

Gradient: $\nabla_\theta J = \frac{\partial J}{\partial \theta} = \begin{pmatrix} \frac{\partial J}{\partial \theta_1} \\ \vdots \\ \frac{\partial J}{\partial \theta_n} \end{pmatrix}$

Direction of Steepest Ascent



Linearisation (Taylor): $J(\theta_0 + \Delta\theta) \approx J(\theta_0) + \Delta\theta \cdot \nabla_\theta J(\theta_0)$

Largest Decrease in direction of $\Delta\theta = -\alpha \nabla_\theta J(\theta_0)$:

$$J(\theta_0 - \alpha \nabla_\theta J(\theta_0)) \approx J(\theta_0) - \alpha \|\nabla_\theta J(\theta_0)\|^2 \quad (\alpha \text{ sufficiently small})$$

Gradient of the Mean Square Cost (Generalised Perceptron)

Lots of Calculus Needed !

$$
\begin{aligned}
\frac{\partial J_{\mathrm{MSE}}(\theta)}{\partial \theta_k} &= \frac{\partial}{\partial \theta_k} \frac{1}{2m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right)^2 \\
&= \frac{1}{2m} \sum_{i=1}^{m} \frac{\partial}{\partial \theta_k} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right)^2 \\
&= \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \frac{\partial h_\theta(\mathbf{x}^{(i)})}{\partial \theta_k} \\
&= \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \sigma'(\mathbf{w} \cdot \mathbf{x}^{(i)} + b) \frac{\partial (\mathbf{w} \cdot \mathbf{x}^{(i)} + b)}{\partial \theta_k}
\end{aligned}
$$

In the third and the fourth line we have used the **chain rule of calculus**.
By using $\sigma'(z) = \sigma(z)(1 - \sigma(z))$ and $\hat{y}^{(i)} = h_\theta(\mathbf{x}^{(i)}) = \sigma(\mathbf{w} \cdot \mathbf{x}^{(i)} + b)$ we obtain

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

Gradient of the Mean Square Cost (Generalised Perceptron)

The gradient of the cost function is the sum over all training samples where each sample contributes in proportion to the difference between true label and its predicted value.

$$\nabla_{\mathbf{w}} J_{\mathrm{MSE}}(\mathbf{w}, b) = \underbrace{\frac{1}{m}\sum_{i=1}^{m}}_{\text{Average over the training set.}} \underbrace{\hat{y}^{(i)}\left(1-\hat{y}^{(i)}\right)}_{\text{Measure for the uncertainty in the prediction: It is small if } \hat{y} \text{ is close to zero or close to 1 and large if } p \text{ is around 0.5, i.e. if the prediction is clear.}} \underbrace{\left(\hat{y}^{(i)}-y^{(i)}\right)}_{\text{Error Signal — contributes more for samples with mismatch. Similar to Perceptron Learning Rule but } \hat{y} \text{ is different.}} \mathbf{x}^{(i)}$$

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

General Gradient Descent Update-Rules

In vector notation:

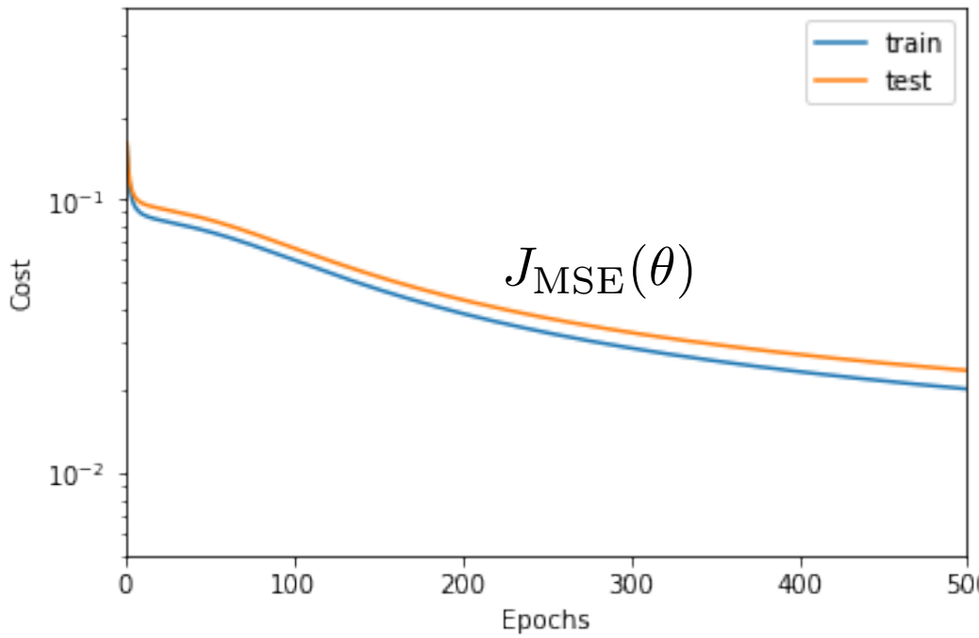
$$\theta \leftarrow \theta - \alpha \nabla_\theta J(\theta)$$

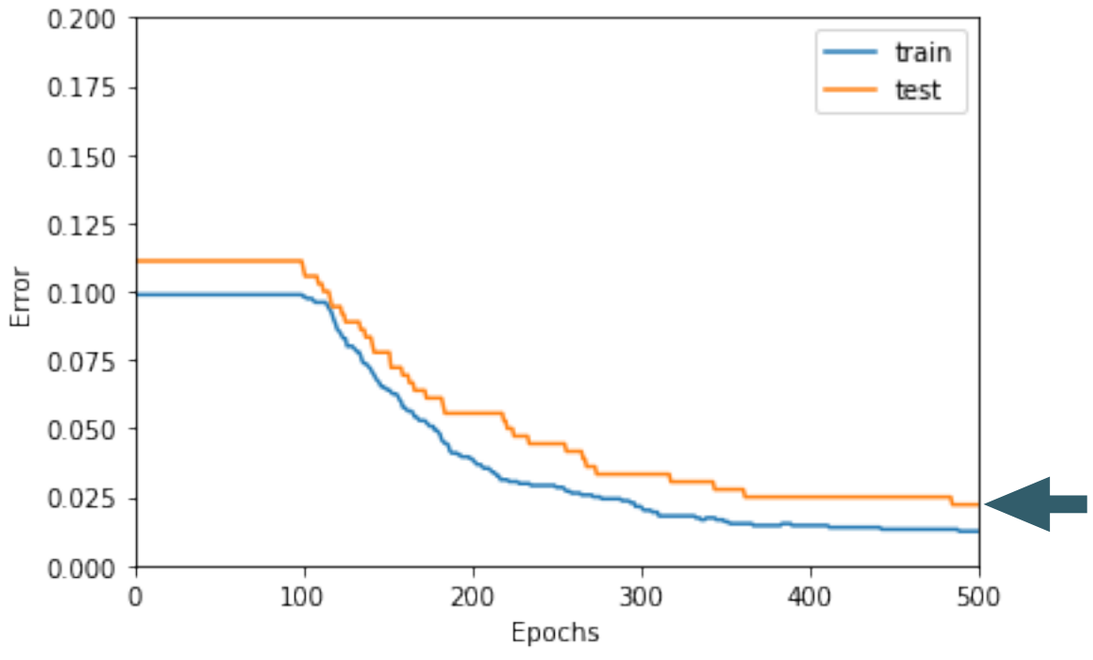
In coordinates:

$$\theta_k \leftarrow \theta_k - \alpha \frac{\partial J(\theta)}{\partial \theta_k}$$

α : learning rate.

Results for MNIST Lightweight





- Inputs rescaled to [0,1]
- Weights and bias initialised to zero
- Learning rate: 0.5
- 500 Iterations (‘epochs’)

- Training cost monotonically decreasing.
- Error / Performance measure:

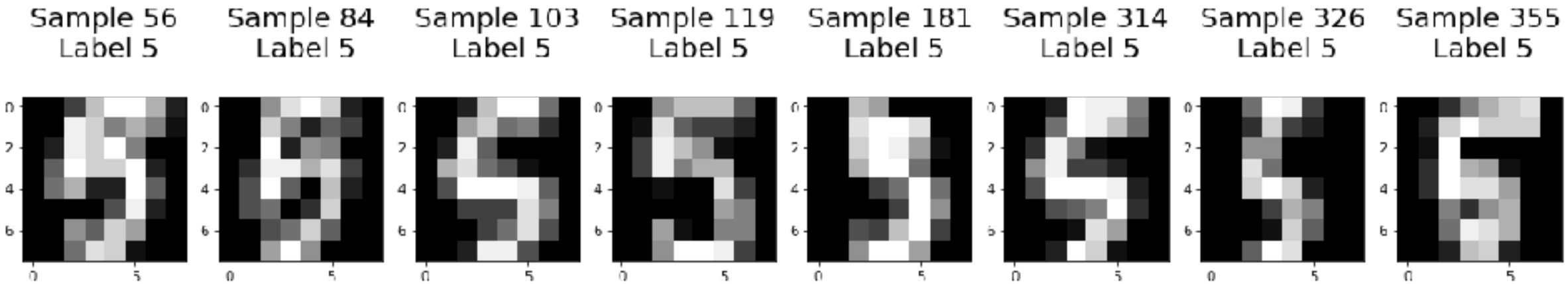
$$\mathrm{error} = \frac{|\{\hat{y} \neq y\}|}{m_{\mathrm{test}}}$$

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

- After 500 epochs:
  - Training error: 1.2%
  - Test error: 2.2%
    (8 samples misclassified)

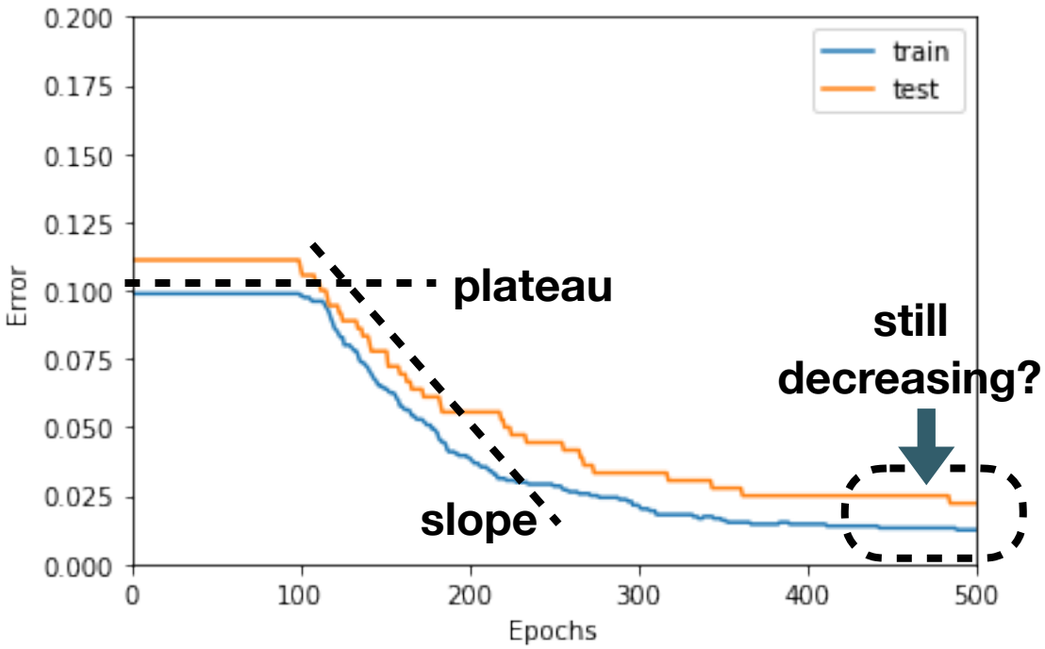
Results for MNIST Lightweight

Mis-classified images of the test set:



Possible Improvements

- Try different learning rates
  - Learning rate needs to be tuned to the problem at hand.
- Run more epochs
  - Have we arrived at the bottom of the valley?
- More care when initialising parameters and preparing the input data
  - Do the initial weights properly match the distribution of the input data?
- Other cost function
  - Are there other cost functions that better capture the differences between predictions and labels?

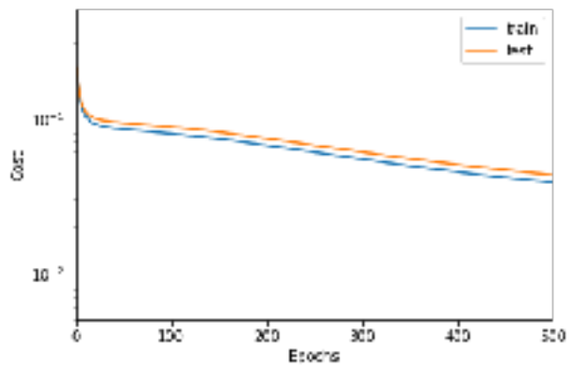


Effect of Different Learning Rates

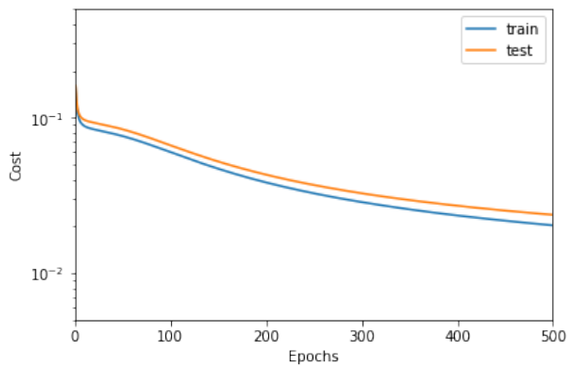
Running batch gradient descent over 500 epochs with different learning rates…

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

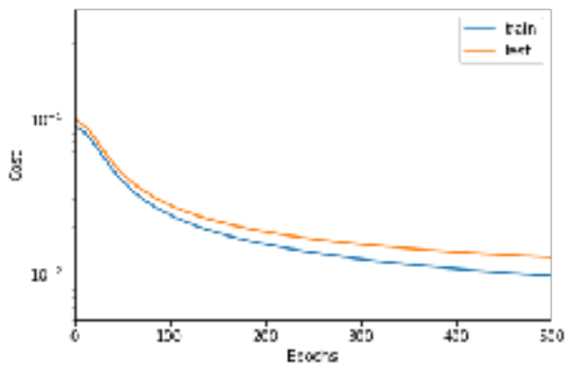
learning rate 0.2



learning rate 0.5

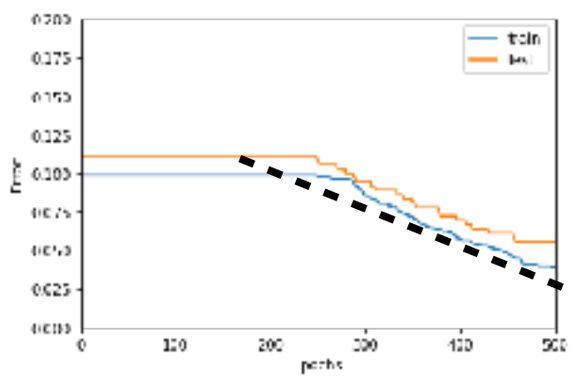


learning rate 2.0

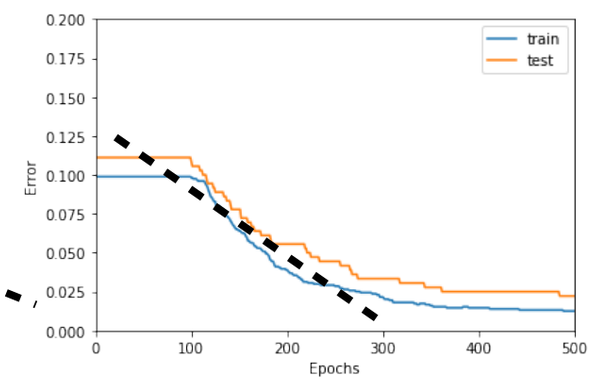


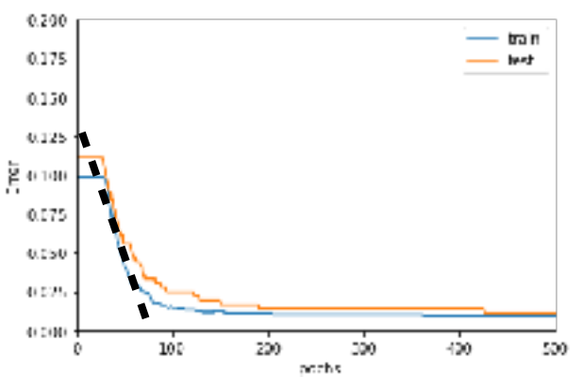
Error rate starts at 10% — corresponds to error rate of dummy predictor.

Dummy Predictor:
None of the digits is a ‘5’.



slow decrease in cost and error rates
more epochs would be needed here !



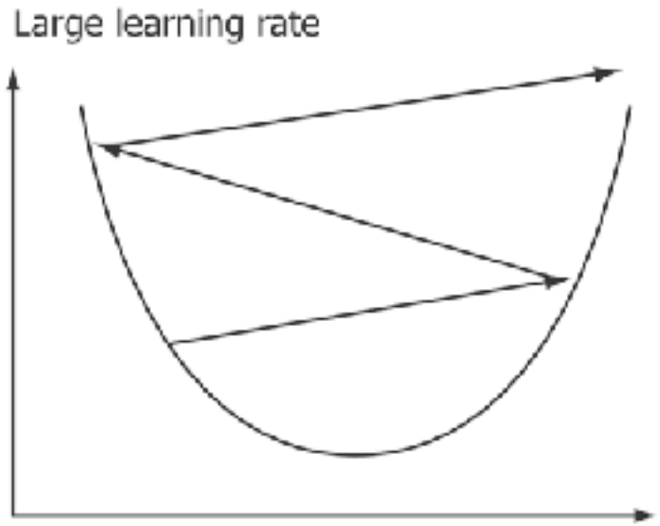


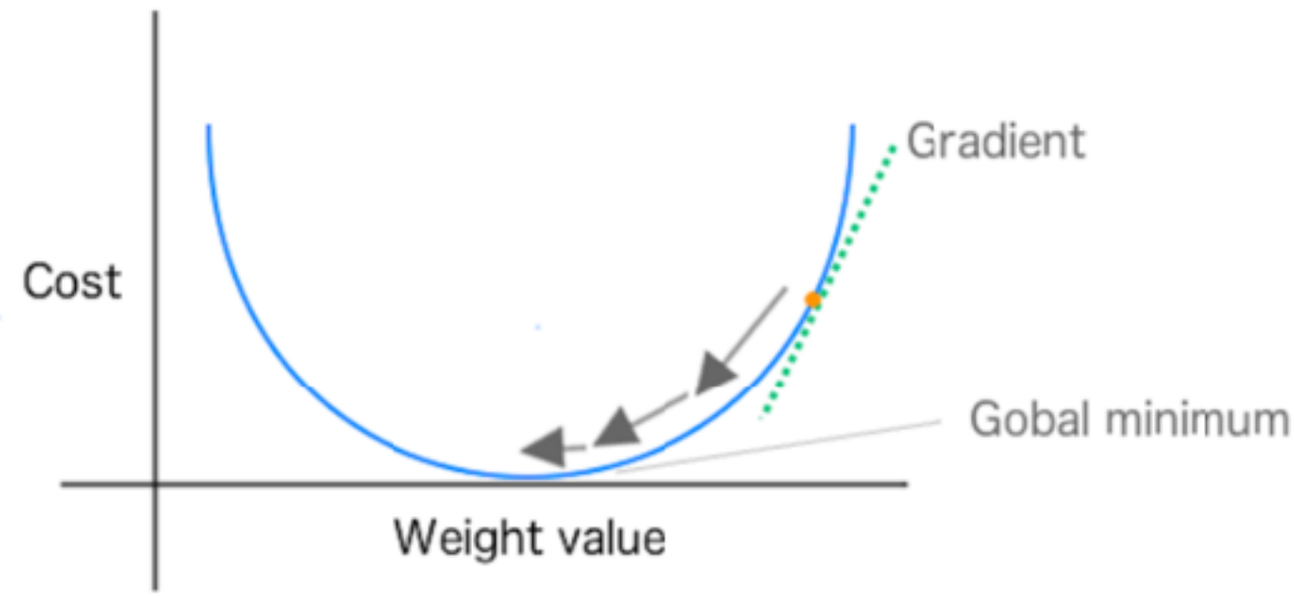
faster decrease in cost and error rates

Further reading: https://www.jeremyjordan.me/nn-learning-rate/

Learning Rate α

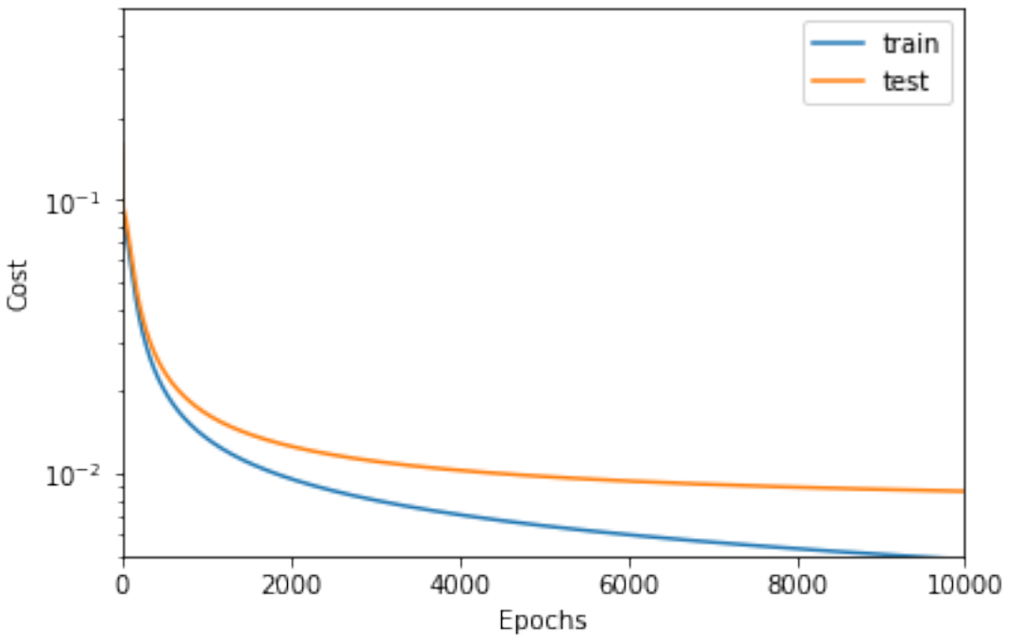
- Determines the learning speed
  - If too large: Oscillation around the minimum or even divergence
  - If too small: Slow convergence





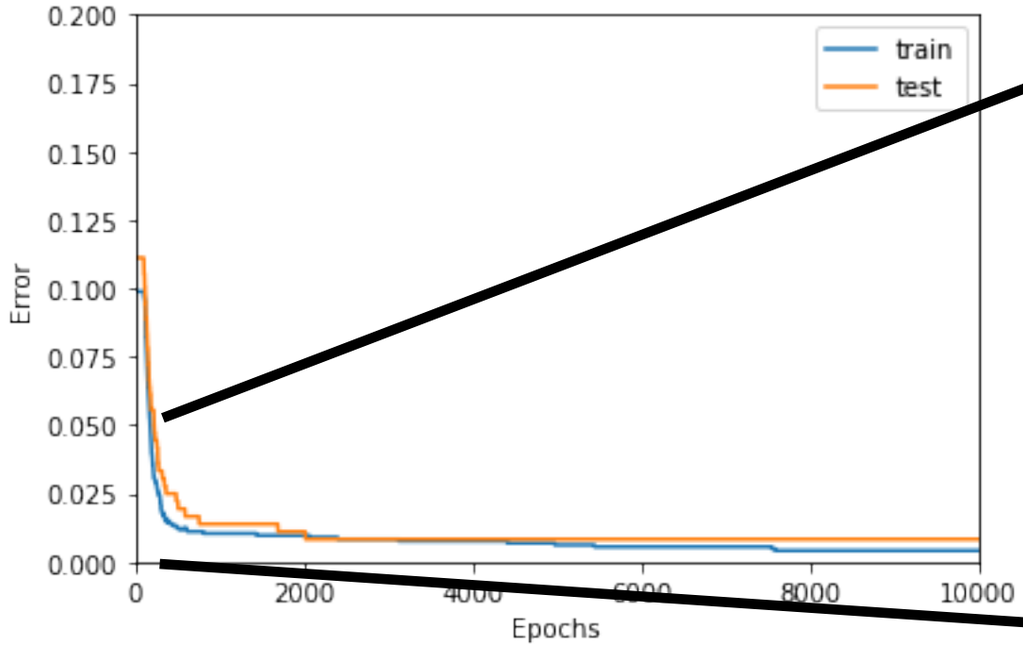
- Needs to be tuned to a given problem
  - An optimal value for a given problem may not work for another problem.
  - Thanks to data normalisation it's largely independent of the scale of the original data.

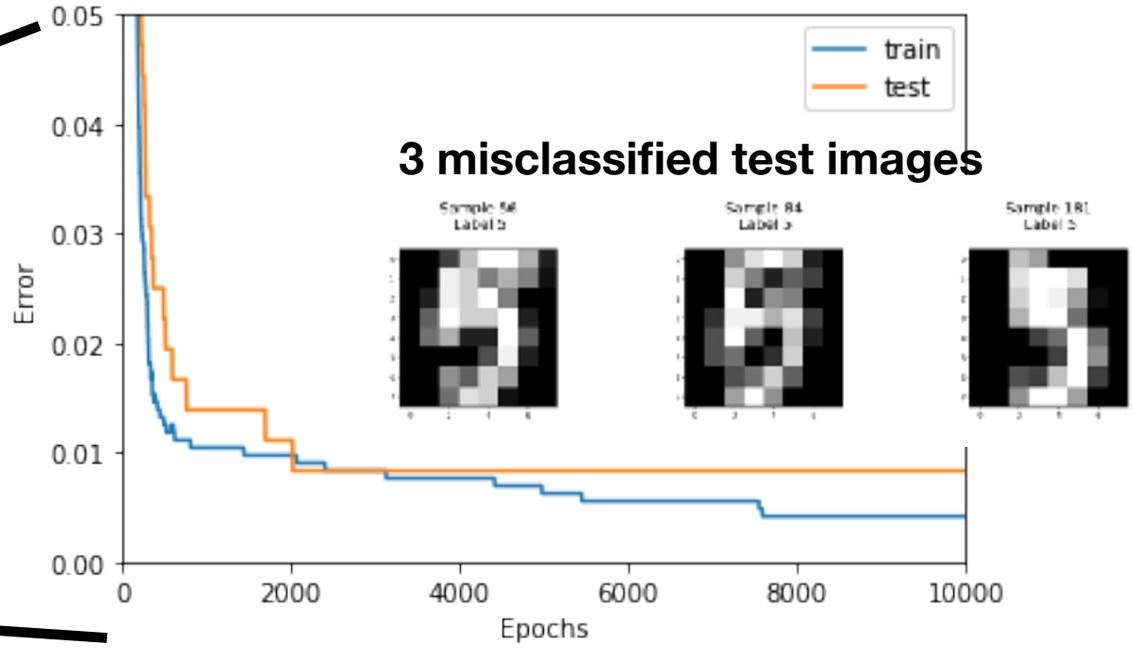
Effect of More Epochs



After 2’000 epochs the test performance (test error) cannot be further reduced.

After 2’000 epochs the model gets overfitted. This important topic will be further covered in weeks 3 and 5).





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 this expression 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.
- To understand the cross entropy cost function, a probabilistic consideration is needed — or stated differently, by using the cross entropy cost function, a probabilistic perspective is adopted.

Probabilistic Interpretation of the Model Output

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

- In maths terms:

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

- Since $p(y|\mathbf{x})$ depends on θ we sometimes indicate that explicitly by writing:

$$p(y|\mathbf{x}, \theta) \quad \text{or} \quad p_\theta(y|\mathbf{x})$$

Cross-Entropy Loss Function

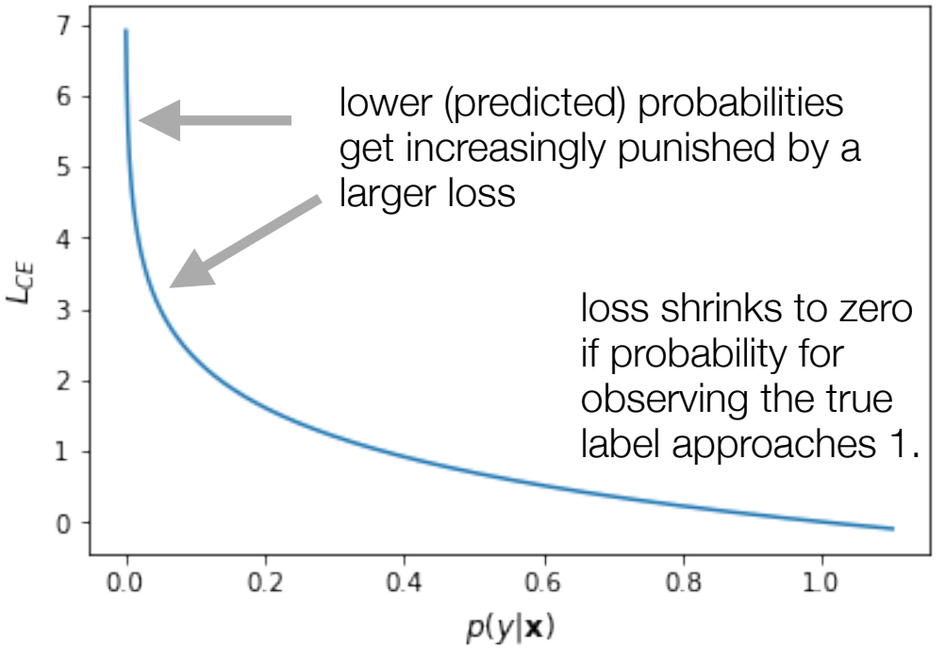
- Suited for classification problems.
- Based on model predicting the probability for observing class y given $\mathbf{x}$ denoted by $p_\theta(y|\mathbf{x})$.
- Cross-Entropy Loss defined by

$$L_{\mathrm{CE}}((\mathbf{x}, y), \theta) = -\log(p_\theta(y|\mathbf{x}))$$

where y is the true label for $\mathbf{x}$.

- Example: Binary classification

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



$$L_{\mathrm{CE}}((\mathbf{x}, y), \theta) = -y \log(h_\theta(\mathbf{x})) - (1 - y) \log(h_\theta(\mathbf{x}))$$

Cross-Entropy Cost Function

- Cross entropy cost function : Averaged cross entropy loss

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

- For the binary classification problem:

$$J_{\mathrm{CE}}(\theta) = -\frac{1}{m}\sum_{i=1}^{m}\Big(y^{(i)}\log(h_\theta(\mathbf{x}^{(i)})) + (1-y^{(i)})\log(1-h_\theta(\mathbf{x}^{(i)}))\Big)$$

- In the given single layer perceptron problem the cross-entropy cost function is a *convex function*. As a result, the function has a single local minimum (a single critical point) which is the global minimum.

$\Rightarrow$ As long as we choose the learning rate sufficiently small, we will always get to the global minimum with gradient descent.

From Maximum Likelihood to Cross Entropy

- Maximum Likelihood is a widely used principle for determining the parameters of a model: Choose the parameters θ such that the probability for observing the given dataset

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

is maximised.

- In mathematical terms:

$$\begin{aligned}
\hat{\theta} &= \underset{\theta \in \mathbb{R}^n}{arg\,max} \prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}, \theta) \\
&= \underset{\theta \in \mathbb{R}^n}{arg\,max} \log \left(\prod_{i=1}^{m} p(y^{(i)}|\mathbf{x}^{(i)}, \theta)\right) \\
&= \underset{\theta \in \mathbb{R}^n}{arg\,max} \left(\sum_{i=1}^{m} \log p(y^{(i)}|\mathbf{x}^{(i)}, \theta)\right) \\
&= \underset{\theta \in \mathbb{R}^n}{arg\,min} \left(-\sum_{i=1}^{m} \log p(y^{(i)}|\mathbf{x}^{(i)}, \theta)\right)
\end{aligned}$$

Cross-Entropy Cost

Gradient of the Cross-Entropy Loss (Generalised Perceptron)

The calculation of the update rule for the cross-entropy cost function is more complicated. We compute it here for the binary classification case. We first state a result we have computed for the MSE cost:

$$\nabla h_\theta(\mathbf{x}) = h_\theta(\mathbf{x}) \cdot (1 - h_\theta(\mathbf{x})) \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \qquad (*)$$

Now we can compute the gradient of the CE-loss:

$$\begin{aligned}
-\nabla L_{\mathrm{CE}}(\theta) &= \nabla \left(y \log(h_\theta(\mathbf{x})) + (1-y) \log(1 - h_\theta(\mathbf{x})) \right) \\
&= \frac{y}{h_\theta(\mathbf{x})} \nabla h_\theta(\mathbf{x}) - \frac{1-y}{1-h_\theta(\mathbf{x})} \nabla h_\theta(\mathbf{x}) \\
&= \left(\frac{y}{h_\theta(\mathbf{x})} - \frac{1-y}{1-h_\theta(\mathbf{x})} \right) \nabla h_\theta(\mathbf{x}) \\
&\overset{(*)}{=} (y(1 - h_\theta(\mathbf{x})) - (1-y) h_\theta(\mathbf{x})) \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix} \\
&= (y - h_\theta(\mathbf{x})) \begin{pmatrix} \mathbf{x} \\ 1 \end{pmatrix}
\end{aligned}$$

Gradient of the Cross-Entropy Cost (Generalised Perceptron)

The gradient of the cost function is the gradient per sample (‘loss’) summed over all training samples:

$$\nabla J_{\mathrm{CE}}(\theta) = \frac{1}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \begin{pmatrix} \mathbf{x}^{(i)} \\ 1 \end{pmatrix}$$

Predicted Probability (for seeing label y=1)

Label

Error Signal: Contributes more for samples with a mismatch.

Very similar to the perceptron update rule! What are the differences?

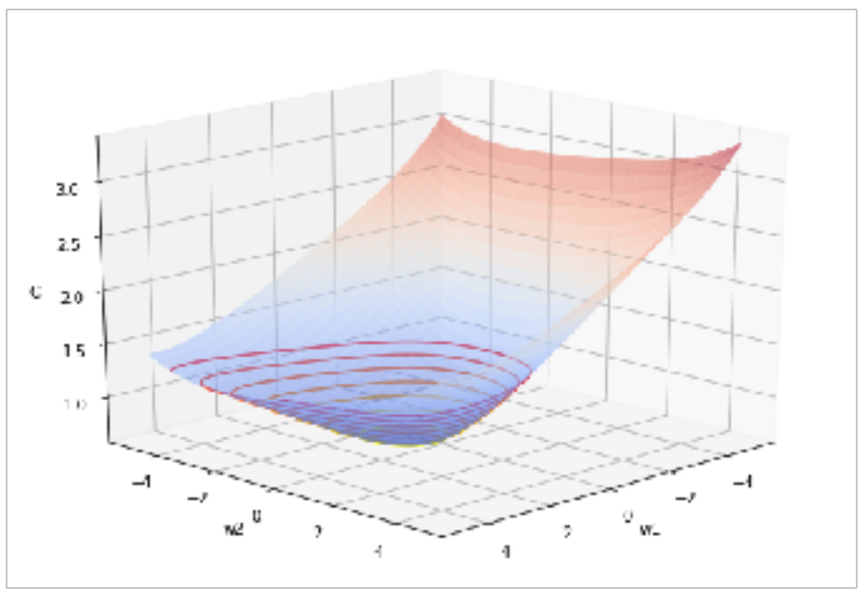
Update Rules for Generalised Perceptron based on Cross Entropy Cost

$$\mathbf{w} \leftarrow \mathbf{w} - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \mathbf{x}^{(i)}$$

$$b \leftarrow b - \frac{\alpha}{m} \sum_{i=1}^{m} \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right)$$

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.



Note on Entropy and Cross-Entropy

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

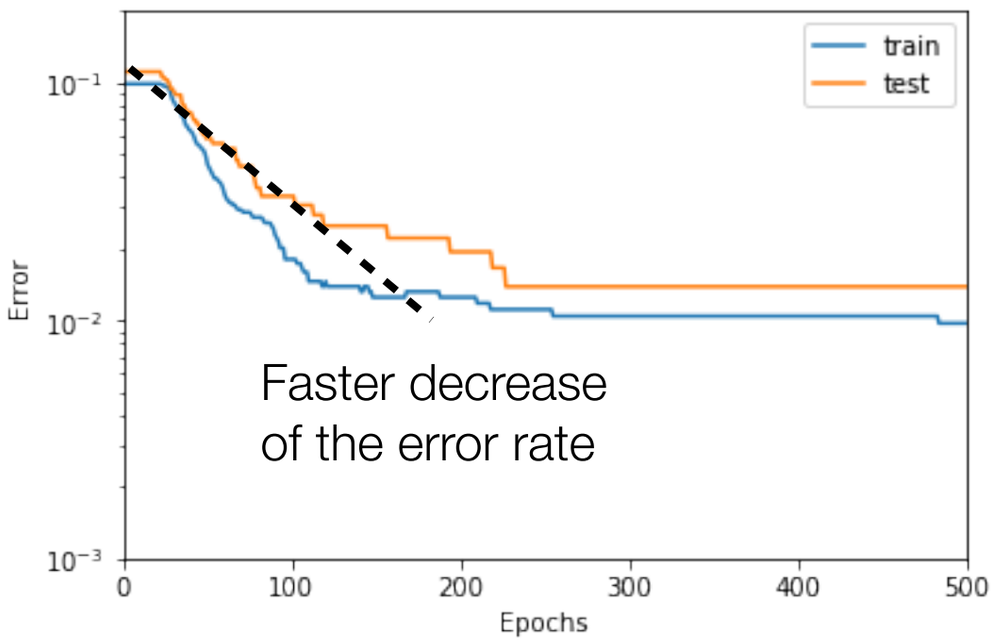
$$H(p) = -\sum_x p(x) \ln p(x) = E_{X \sim p} [-\ln p(X)]$$

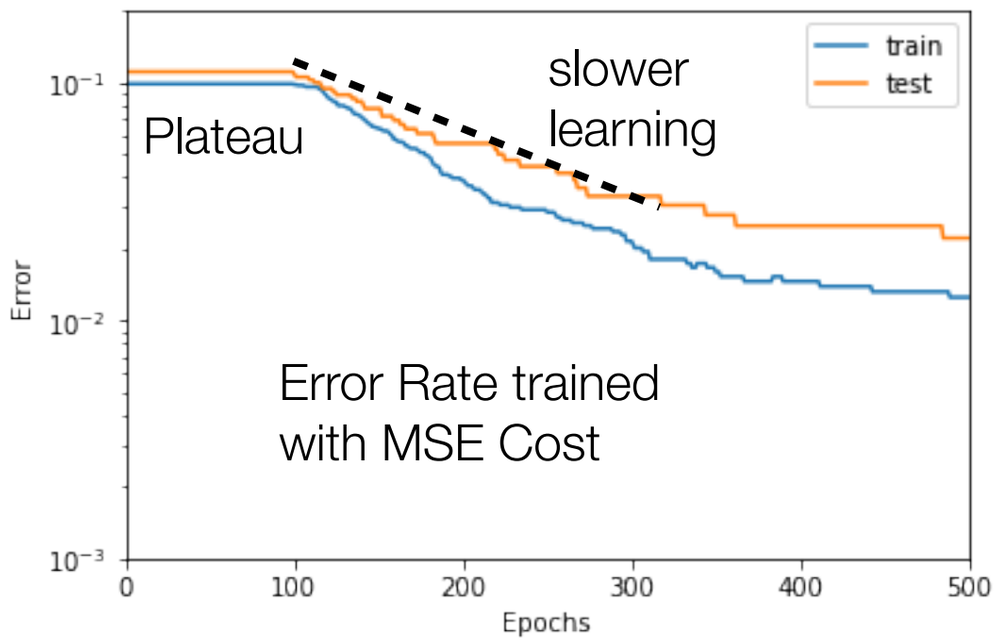
Basic Idea of Information Theory
The more one knows about a topic, the less new information one is apt to get about it. If an event is very probable, it is no surprise when it happens and thus provides little new information. Inversely, if the event was improbable, it is much more informative that the event happened. Therefore, the information content is an increasing function 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**)

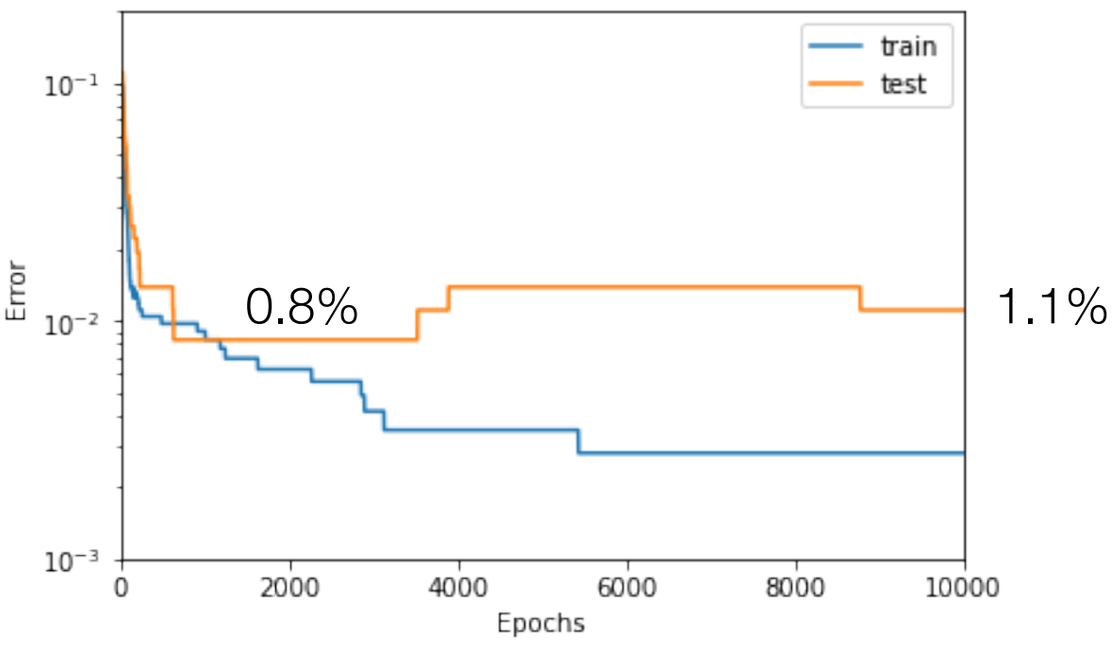
- **Cross-Entropy** measures the information of one distribution w.r.t. to another

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

Results for MNIST Lightweight

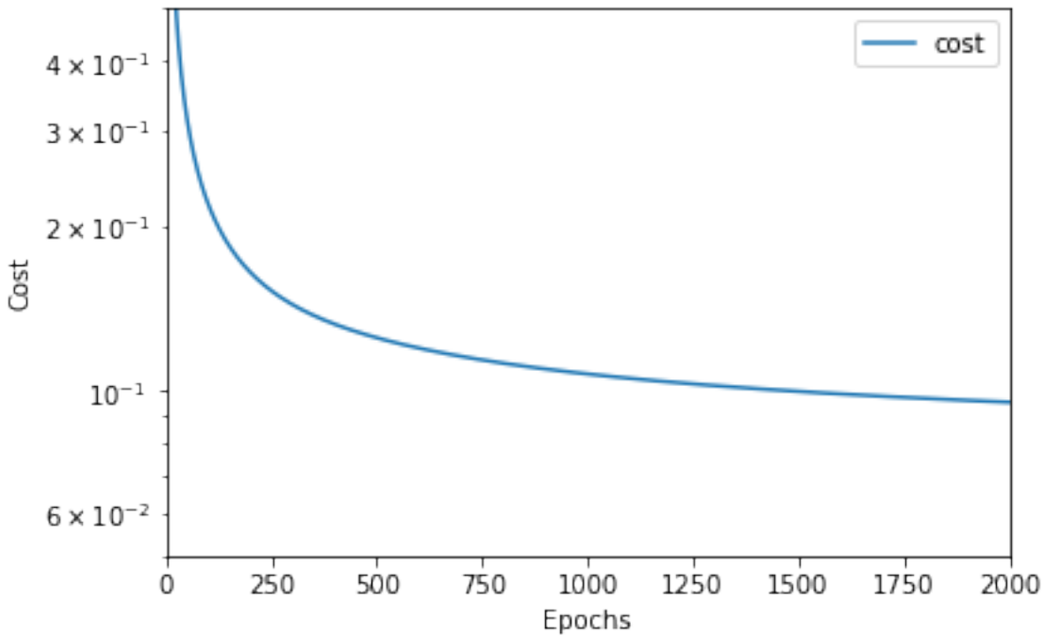


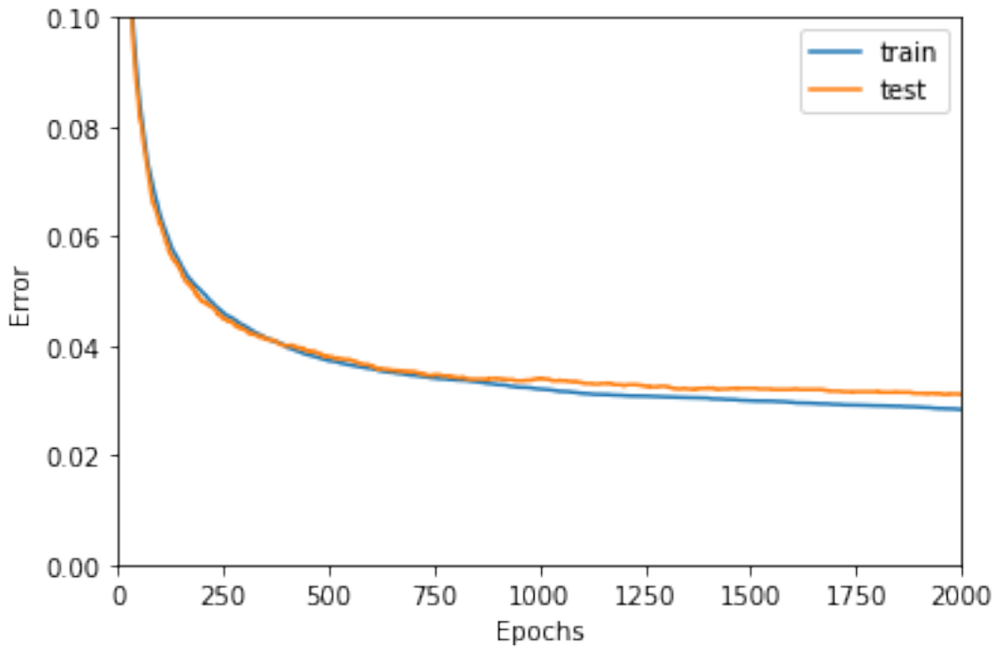




- Same settings as with MSE cost:
  - Inputs rescaled to [0,1]
  - Weights and bias initialised to zero
  - Learning rate: 0.5
- CE Cost performs better:
  - After 500 epochs: 1.4% (MSE: 2.2%)
  - After 2000 epochs: 0.8% (MSE: 1.1%)
  - Misclassifications after 500 epochs: 5

Results for Original MNIST



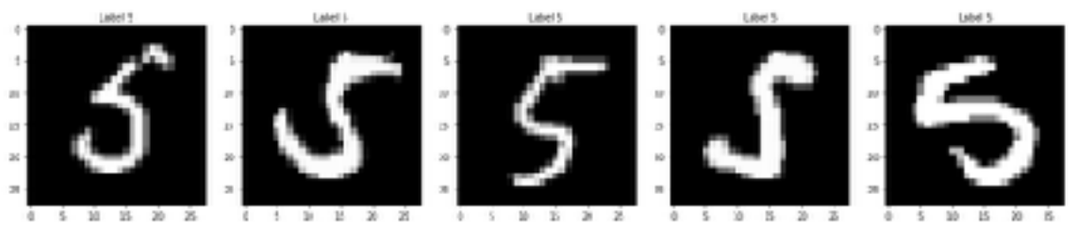



```
from sklearn.datasets import fetch_mldata
from sklearn.model_selection import train_test_split
mnist = fetch_mldata('MNIST original')
x, y = mnist['data'], np.array(mnist['target'], dtype='int')

x_train0, x_test0, y_train, y_test = train_test_split(x, y, test_size=10000, random_state=0)
xmax = np.max(x_train0)
xmin = np.min(x_train0)
x_train = x_train0 / xmax
x_test = x_test0 / xmax
```

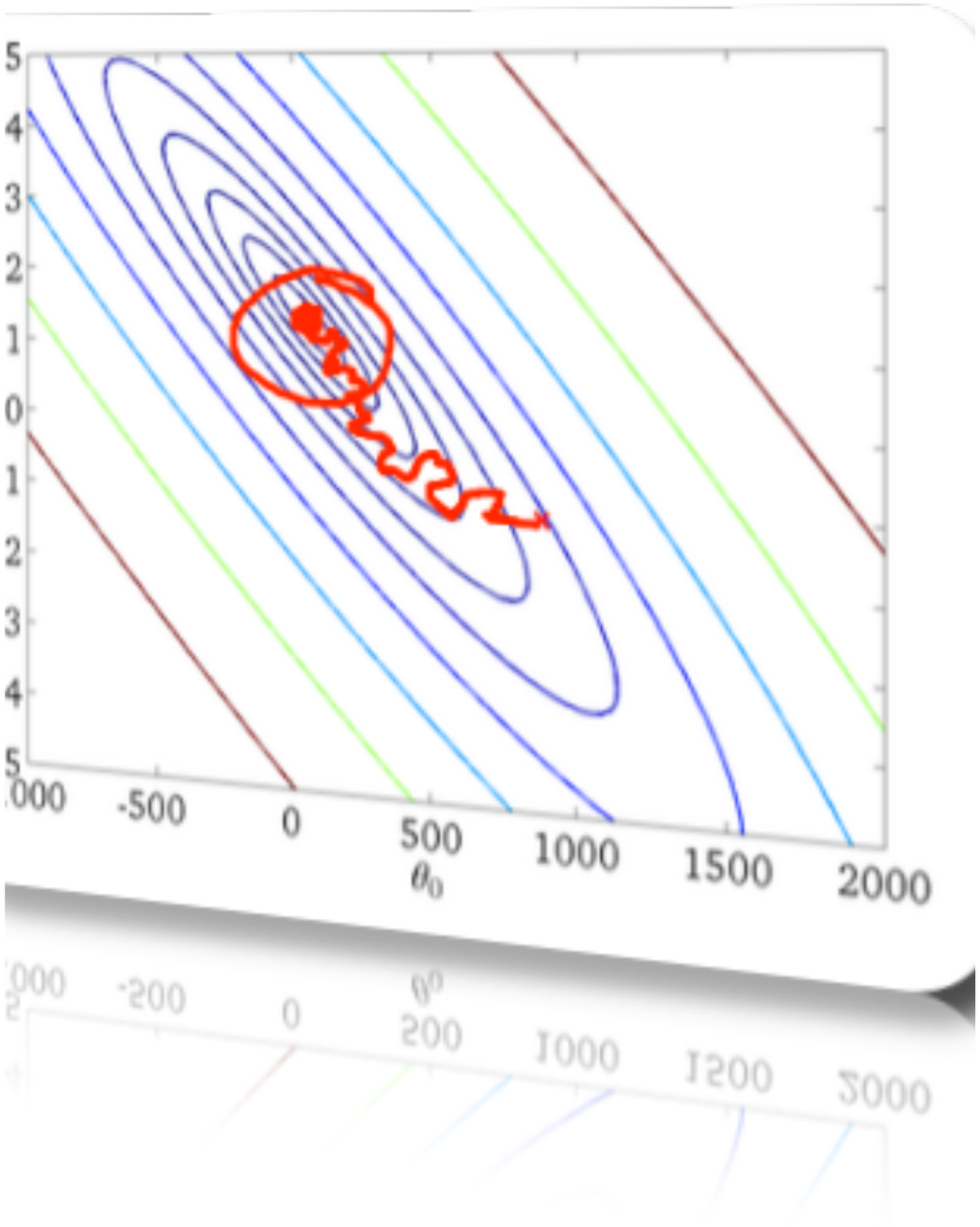
- Learning rate: 1.0
- 2000 epochs

- Training error: 2.84%
- Test error: 3.11% (311 / 10’000)

Some images not identified as ‘5’:



Stochastic Gradient Descent



Motivation for Stochastic Gradient Descent

- So far, we have computed the gradient of the cost function defined on all the training set. This procedure is referred to as ***Batch Gradient Descent***.
- This can be costly: It involves the evaluation of the model and its derivatives for all the samples in the training set.
- This is not an issue for small models like the single (layer) perceptron — but for deep neural networks and if the training dataset is large this becomes a problem.
- Idea of ***Stochastic Gradient Descent*** of ***Mini-Batch Gradient Descent***: Use less training samples to define a single update for moving in parameter space.

Idea behind Stochastic Gradient Descent and Mini-Batch Gradient Descent

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

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

- The update rule for the parameters can be defined by including the contributions from less samples (here binary classification example).

$$\begin{pmatrix} \boldsymbol{\Delta \mathbf{w}} \\ \Delta b \end{pmatrix} = -\alpha \frac{1}{m}\sum_{i=1}^{m} \Big(h_\theta(\mathbf{x}^{(i)}) - y^{(i)}\Big) \begin{pmatrix} \mathbf{x}^{(i)} \\ 1 \end{pmatrix}$$

  - **Batch Gradient Descent**: Summation over all the training set.
  - **Mini-Batch Gradient Descent**: Summation only over subset of the training samples
  - **Stochastic Gradient Descent**: No summation , just a single sample

- However: No guarantee to move in parameter space in a direction that leads to smaller values of the cost function.

Stochastic Gradient Descent (SGD)

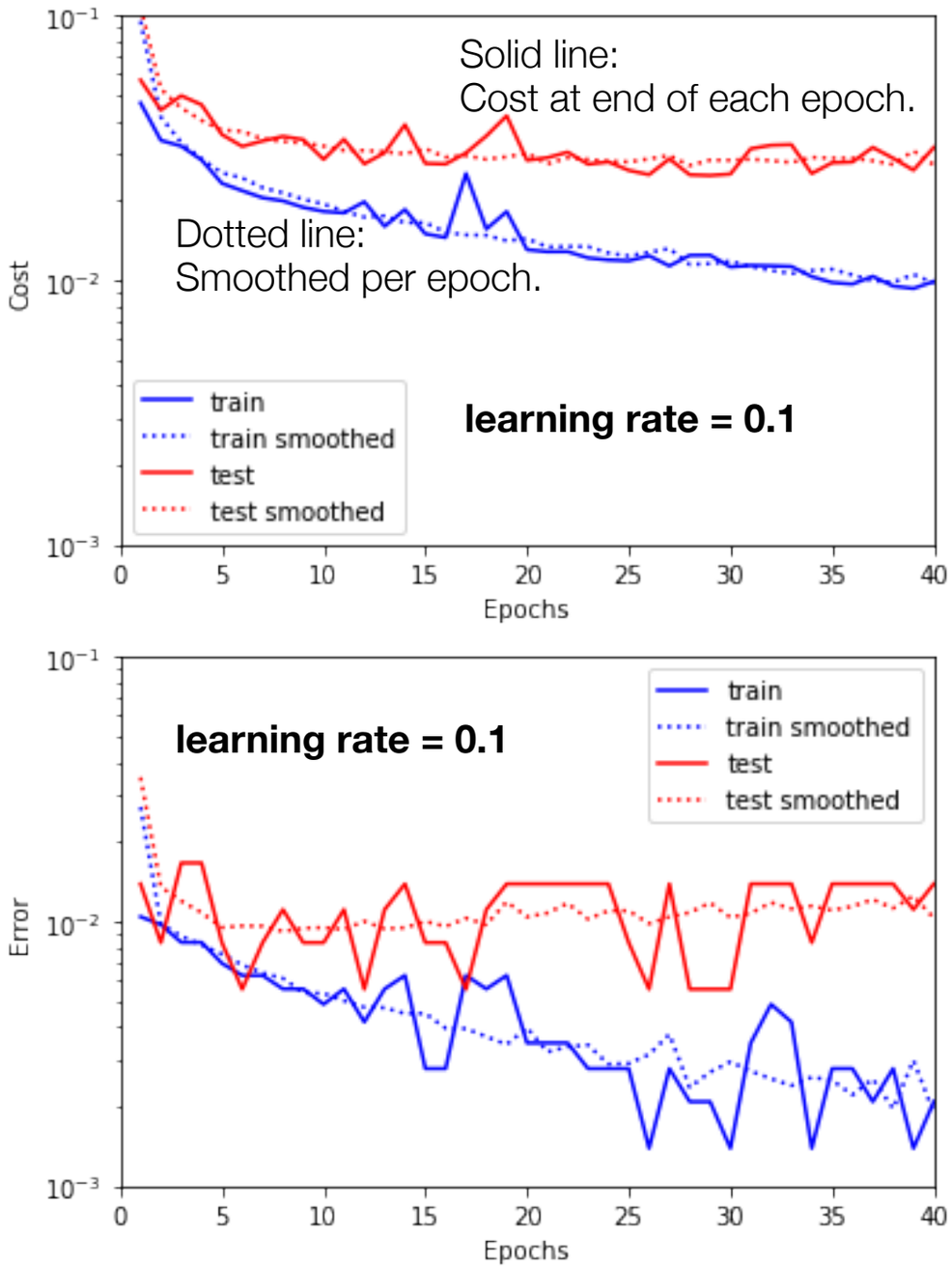
1. Start with some initial θ's (for example random or 0)
   Here: $\theta = (b, \mathbf{w})$
2. Select *one* training sample randomly $(\mathbf{x}^{(i)}, y^{(i)})$
   and perform the update of the θ's with this example

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} - \alpha \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \mathbf{x}^{(i)} \\ b &\leftarrow b - \alpha \left(h_\theta(\mathbf{x}^{(i)}) - y^{(i)} \right) \end{aligned}$$

3. Loop in step 2 until convergence.

Convergence is trickier to observe as we may select favorable and less favorable points in the training set.
In practice we may compute an "averaged" cost function over the past U updates. Typically U<=T and U<<T if the total number of updates T is really large.

Results for Lightweight MNIST (SGD)



Cost is not monotonically decreasing.

Values for the cost and error rates (both training and test data) fluctuate.

Reduce learning rate!

40 epochs have been computed to reach an test error rate of ~1.1%.
Fewer steps needed than with BGD.

one epoch = one pass through the training data
= 1347 update steps

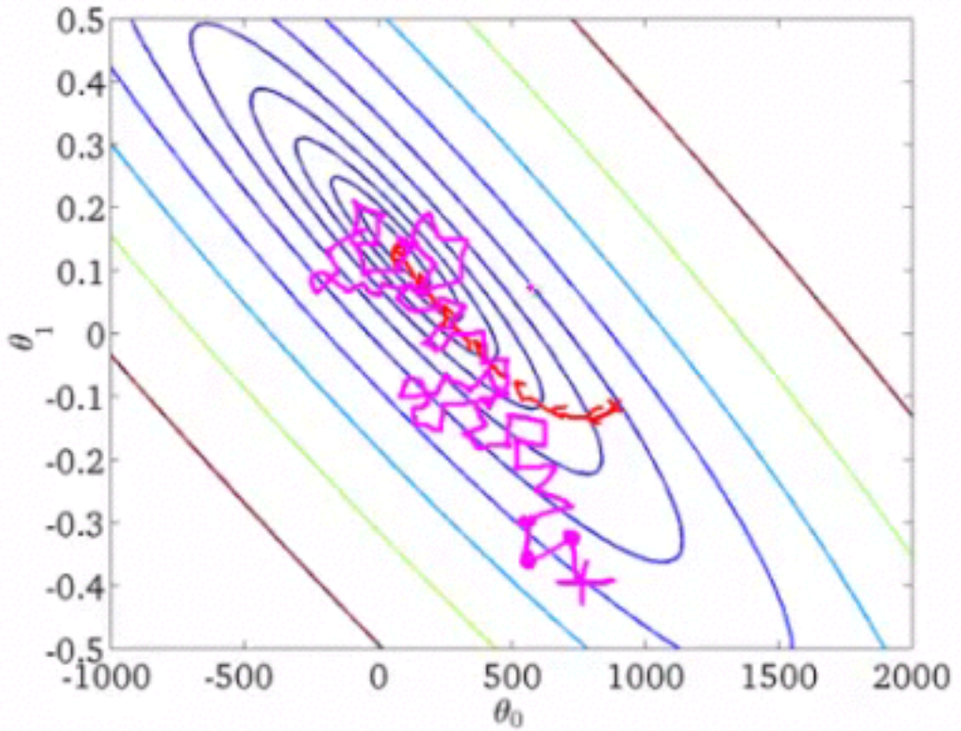
(for comparison with batch gradient descent)

Random motion in parameter space leads to uncertainty in error rate and cost.

Characteristics of SGD

General Characteristics

- Tends to move in direction of the global minimum, but not always.
- Never converges like batch gradient descent does, but ends up wandering around close to the global minimum — not a problem in practice as long as we are close to the minimum. Regularising effect!
- As batch gradient descent, the learning principle is generalisable to many other “hypothesis families”.



Advantages

- Faster since parameters are updated for each training sample.
- It can handle very large sets of data - 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.

Online learning is great for systems that receive data as a continuous flow!

Disadvantage

- Not easily parallelised.

Mini-Batch Gradient Descent (MBGD)

Find a good compromise between

Batch Gradient Descent (BGD) ⟺ Stochastic Gradient Descent (SGD)

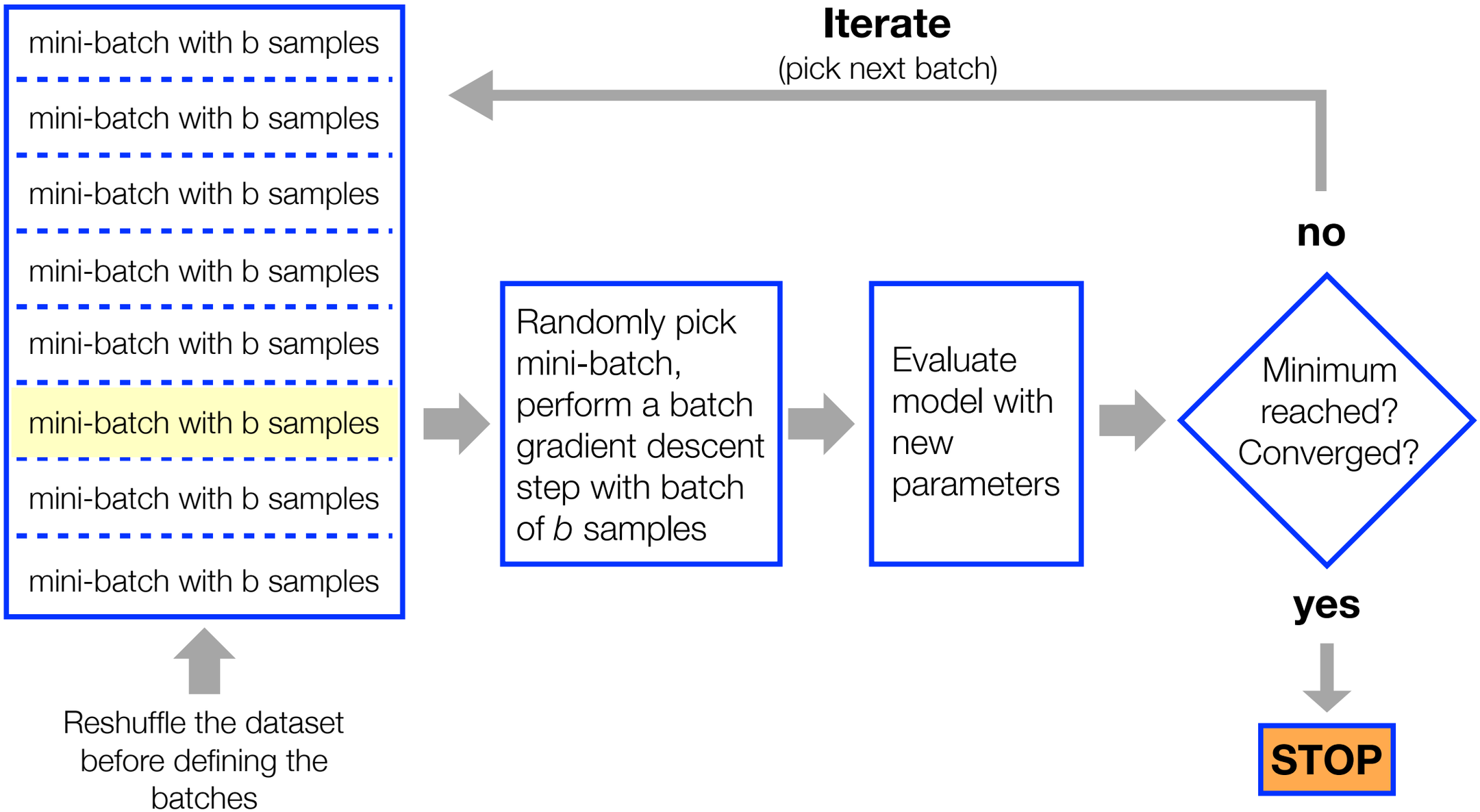
Advantage

- Level of noise seen in the learning curve of stochastic gradient descent is reduced
- Yet allows for vectorised implementations, potentially more efficient (pipelining)
- Straightforward distribution of computational load by computing mini-batches on different machines or cores - i.e. adopting a map/reduce pattern:
  - map = distribute the mini-batches (e.g. each with 50 samples)
  - reduce = update parameters with the results of the mini-batches

Disadvantage

- Batch size needs to be optimised

Mini-Batch Gradient Descent Principle



Mini-Batch Gradient Descent (MBGD)

1. Start with some initial θ's (for example random or 0)
   Here: $\theta = (b, \mathbf{w})$
2. Randomly select mini-batch of *b* training sample with indices $i_1, \ldots, i_b$, i.e.

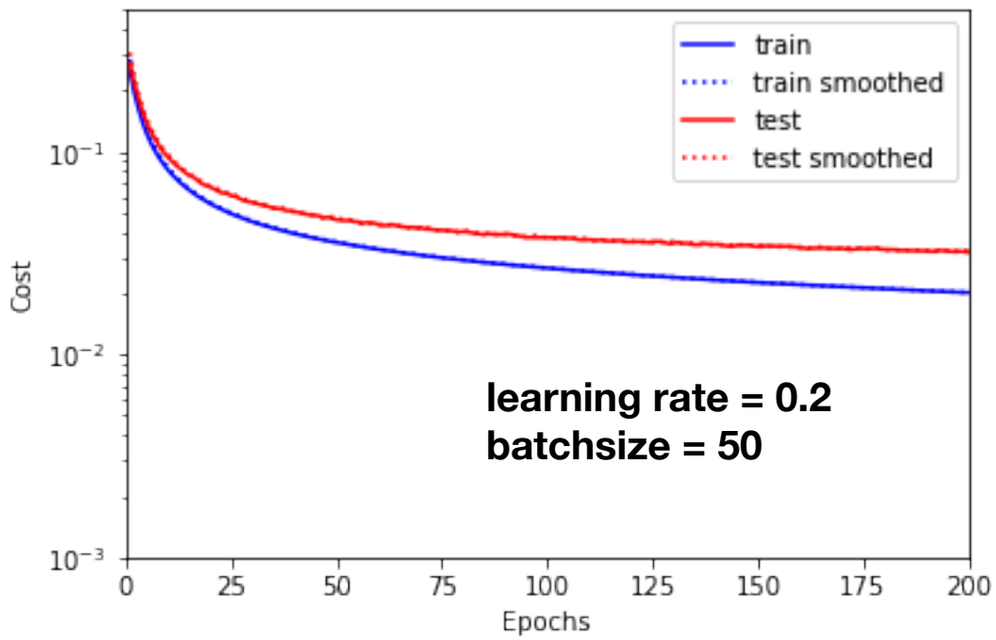
   $$(\mathbf{x}^{(i_1)}, y^{(i_1)}), \ldots, (\mathbf{x}^{(i_b)}, y^{(i_b)})$$

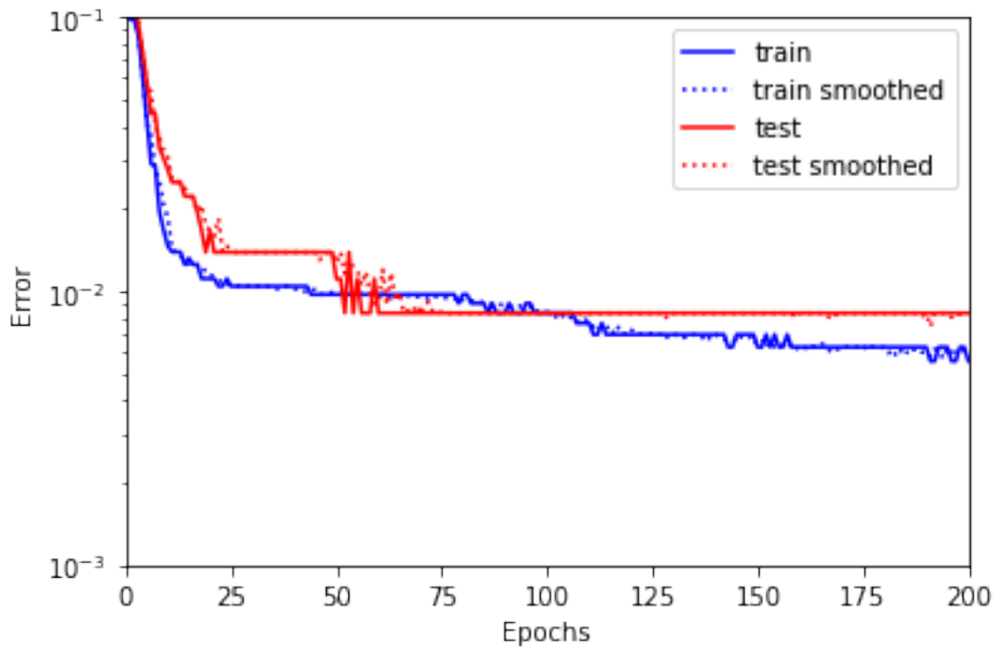
   and perform the update of the 's with this example

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

3. Loop in step 2 until convergence.

Results for Lightweight MNIST





Cost is much smoother than with SGD, typically monotonically decreasing.

Values for the cost and error rates (both training and test data) fluctuate.

Reduce learning rate!

Test error rate of ~0.8% reached after ~60 epochs.

As compared with SGD much less wiggling in the error rate due to random motions in parameter space.

Characteristics of MBGD

General Characteristics

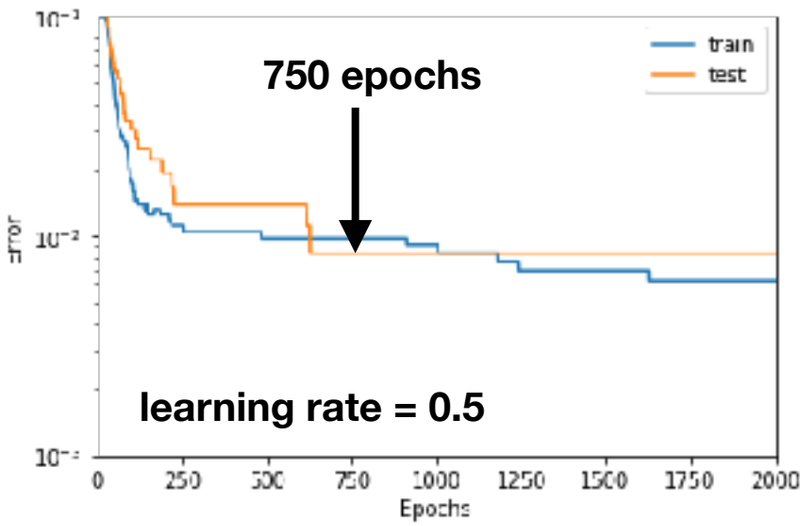
- Tends to move in direction of the global minimum, but not always.
- Ends up wandering closely around to the global minimum. Regularising effect!
- As batch gradient descent, the learning principle is generalisable to many other “hypothesis families”.

Advantages

- Faster since parameters are updated for each mini-batch.
- It can handle very large sets of data - 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.
- Easily parallelised on GPU and in HPC.

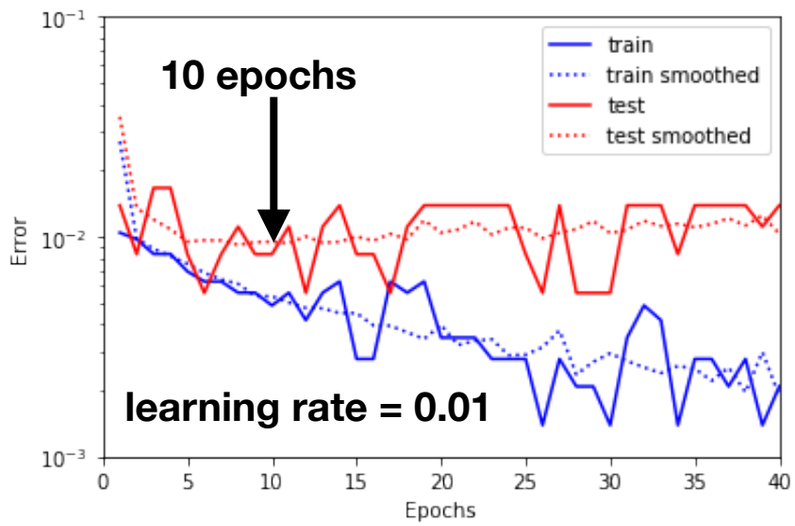
Comparison of Gradient Descent Schemes

BGD



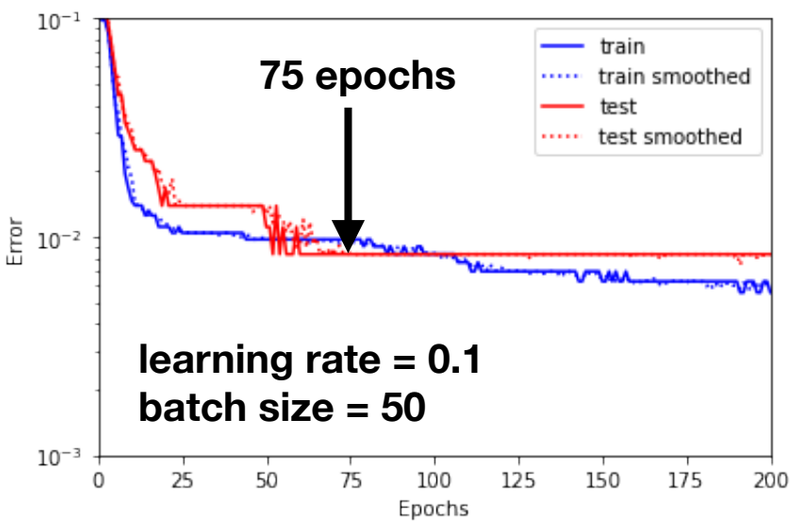
- Smooth, strictly decreasing curves.
- Cost curves strictly decreasing. Can reach minimum.
- Slow for large m. Many epochs needed.
- Learning rate can be large(r).
- No out-of-core support - all data in RAM.
- Easy to parallelise.

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



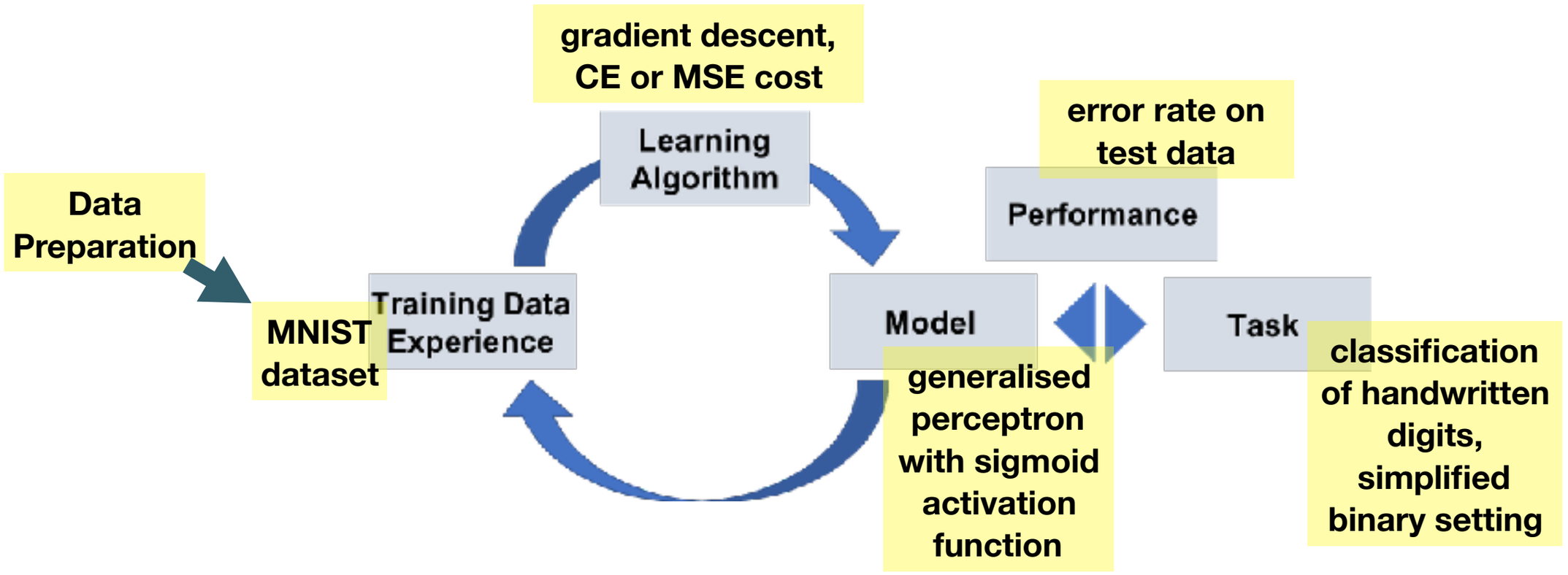
- Slightly wiggling curves.
- Cost curve typically, but not necessarily decreasing. Wiggles slightly around minimum.
- Fast for large m. Much less epochs than BGD but more than SGD needed.
- Medium learning rate.
- Out-of-core support - not all data to be kept in RAM of a single machine.
- Easy to parallelise.

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

Characteristics of Gradient Descent

- Can handle very large sets of data — particularly, the SGD and MBG.
- Allows for incremental learning, i.e. on-the-fly adaptation of the model on new incoming data.
- Learning principle is generalisable to many other forms of models $h_\theta(\mathbf{x})$ such as deep neural networks.
- It works ‘locally’ and finds local critical points (local minima, maxima, saddle points). It is not designed to find global minima in general.

Summary - Ingredients of ML Algorithms



Questions / Outlook

- How to evaluate the performance (‘convergence’) of the different schemes? When to stop?

- How to best choose
  - the learning rate?
  - the mini-batch size?

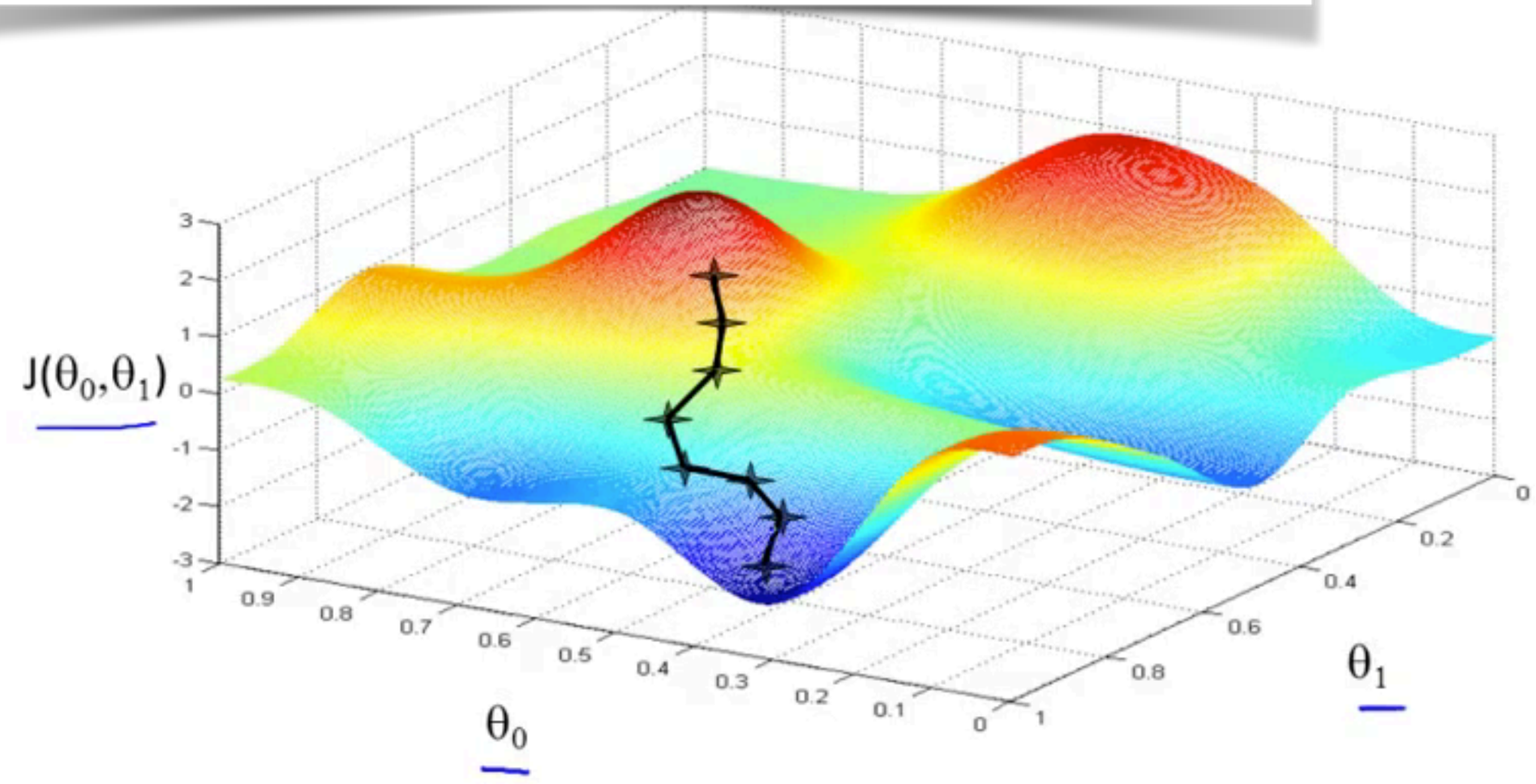
  … and how to evaluate that systematically?

- How can we improve the performance?

Backup

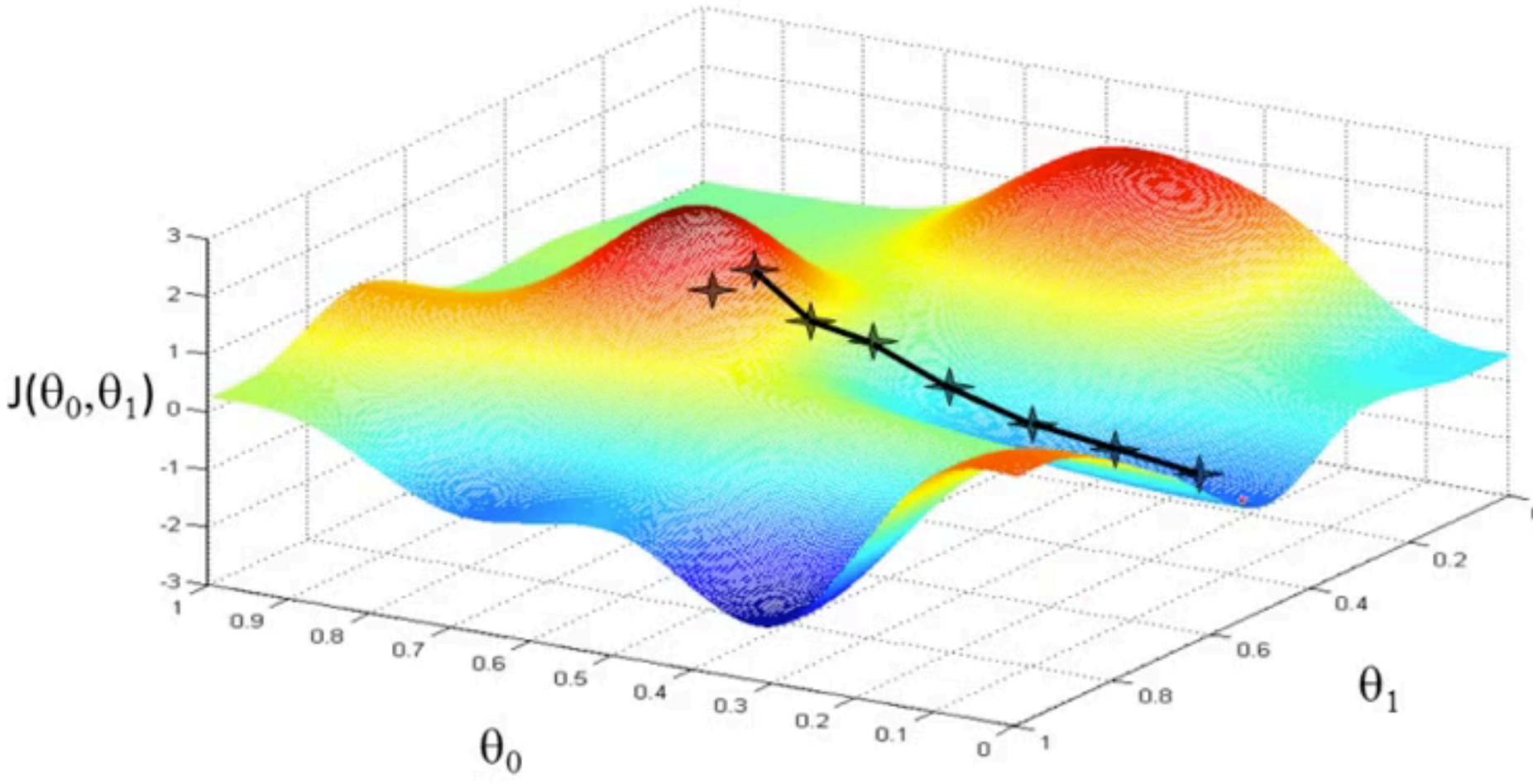
Gradient Descent Intuition

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

Gradient Descent Intuition



Source: Andrew Ng - Machine Learning class - Stanford

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.

In the case of a single unit perceptron and a sigmoid activation function, the J() function is convex, i.e. “bowl shaped” with a unique minimum, so we don’t need to care by getting stuck in a local minimum in this case.