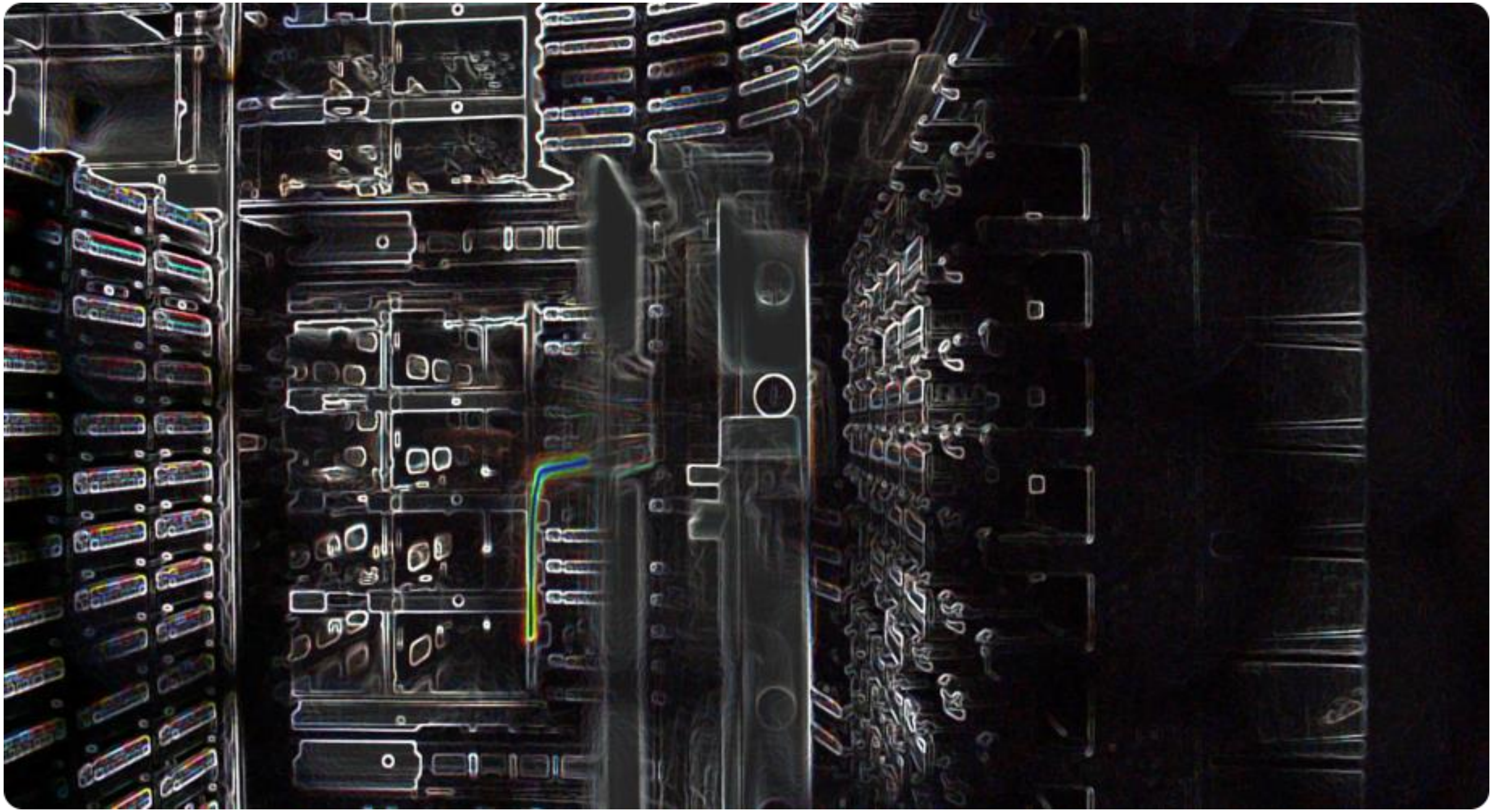


Deep learning

Deep learning with Keras

Today’s program

14:00-14:30 What is ML / DL? What is a neuron?

14:30-15:30 Hands on: building a neuron from scratch

15:30-15:45 How do neural networks work?

15:45-16:15 Break

16:15-16:30 How do neural networks work? (continued)

16:30-17:15 Hands on: building a network from scratch

17:15-18:00 Loss function & updating weights

18:00-19:00 Diner

19:00-19:45 Hands on: the XOR problem

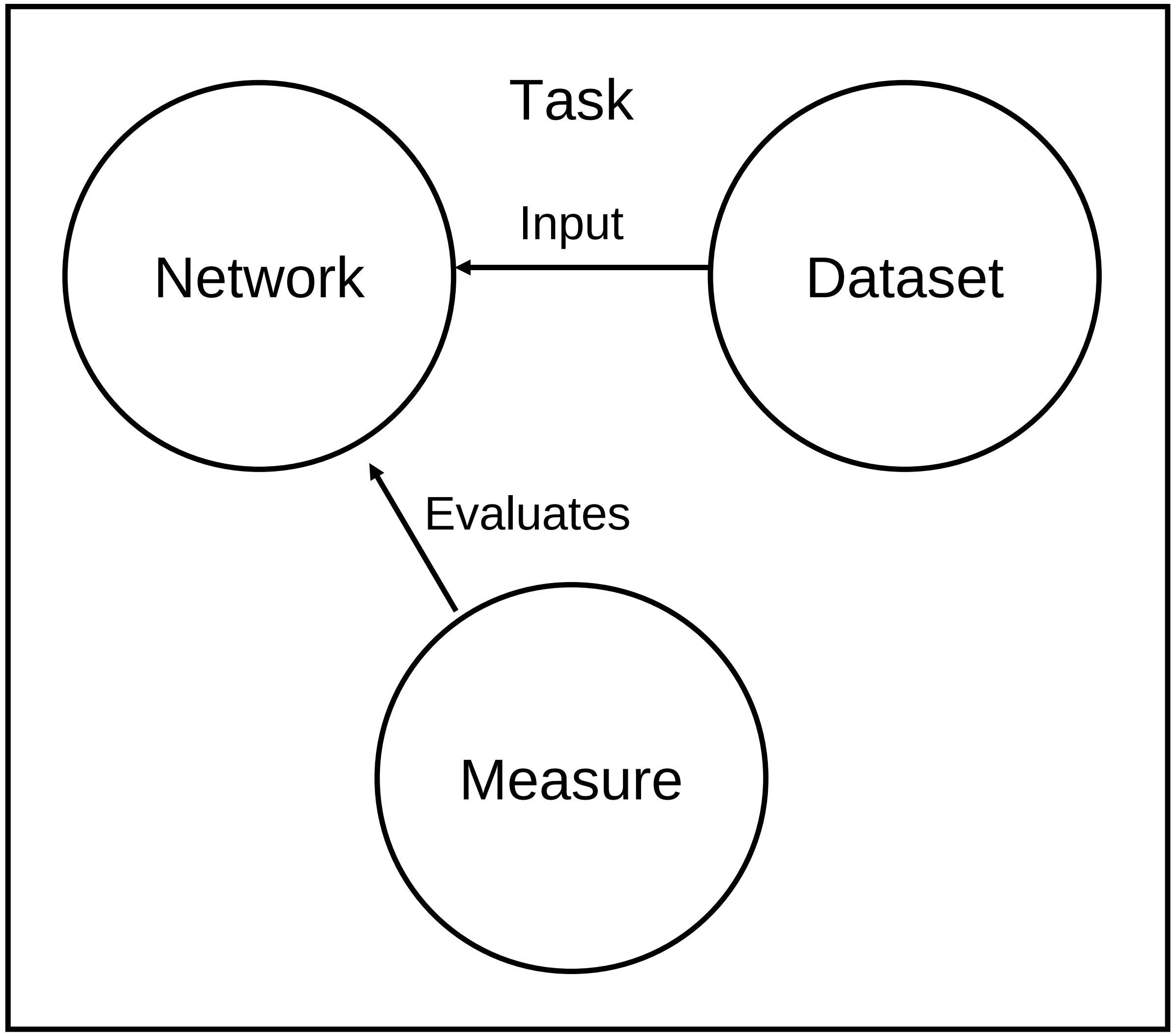
19:45-20:00 Dataset splitting & Performance evaluation

20:00-21:00 Hands on: Keras on fashion Mnist

If we finish early: Machine learning tasks (classification vs regression)

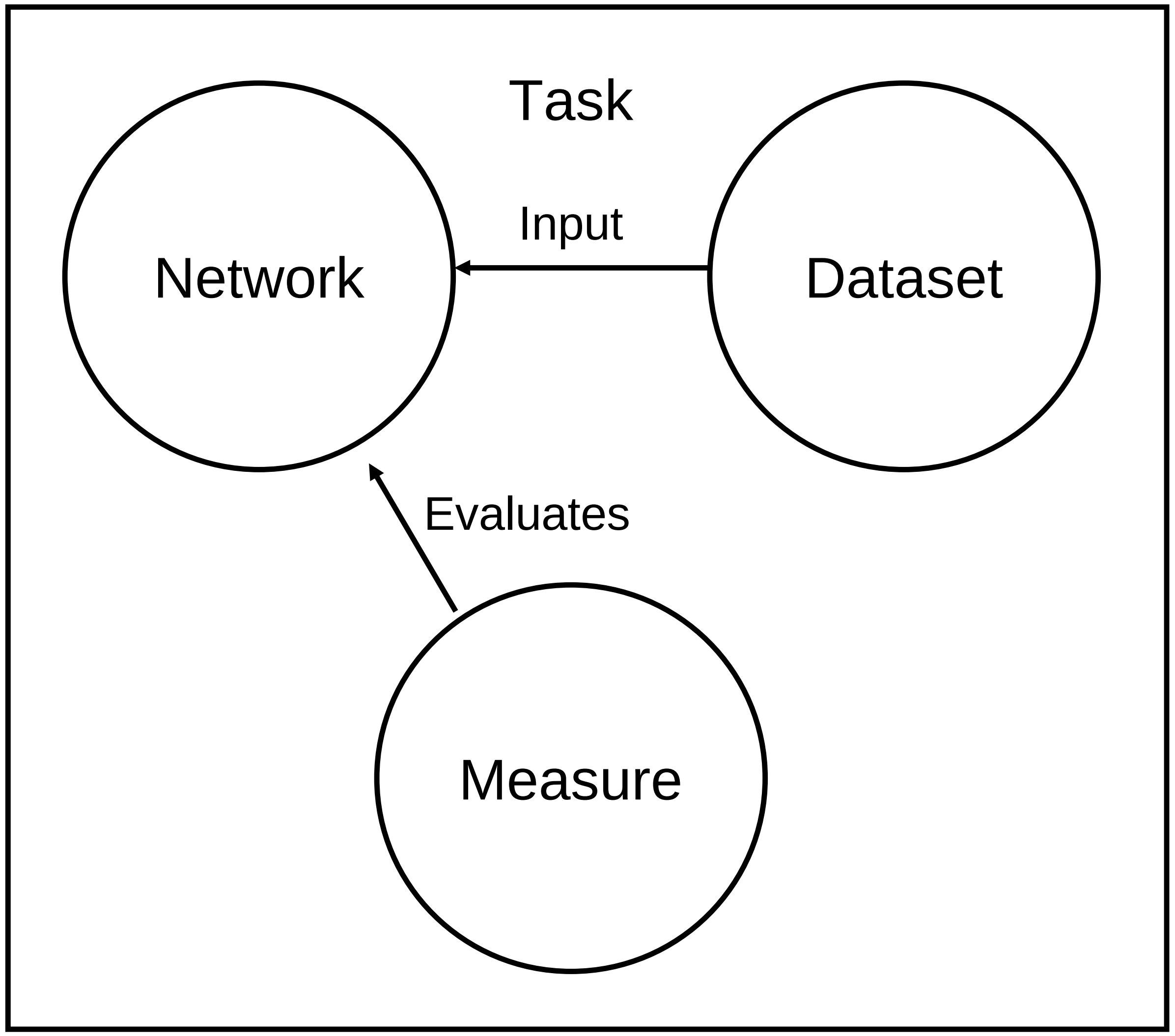
A step back

- What do we want to achieve with our neural networks?
- We want to **input data** and get out some **meaningful result.**
- In machine learning this problem is formulated so that we have a **task** which we want to perform.
  - Regression
  - Classification
  - Clustering



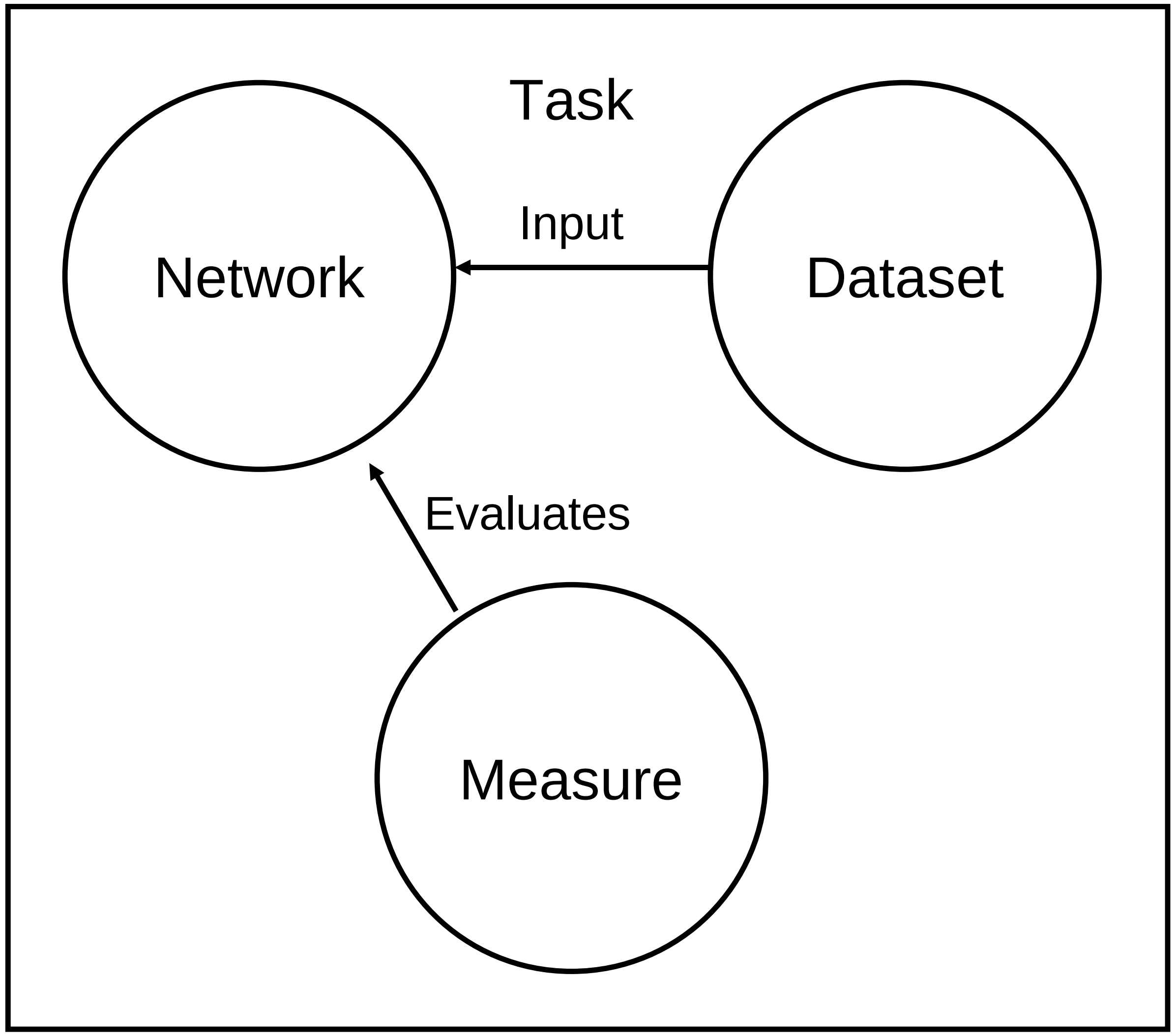
A step back

- Alongside the task we have some input, usually in the form of a **dataset**.
- When the have some **metric** / **measure** which can evaluate how well we are performing the task.
  - Accuracy
  - F1 score
  - Area-under-curve (AUC)



A step back

- Our network should then learn to **map** our **input to** some **output**.
- We thus treat the network as some function f which maps some x to $\hat{y}$ some prediction:
$$f(x) = \hat{y}$$

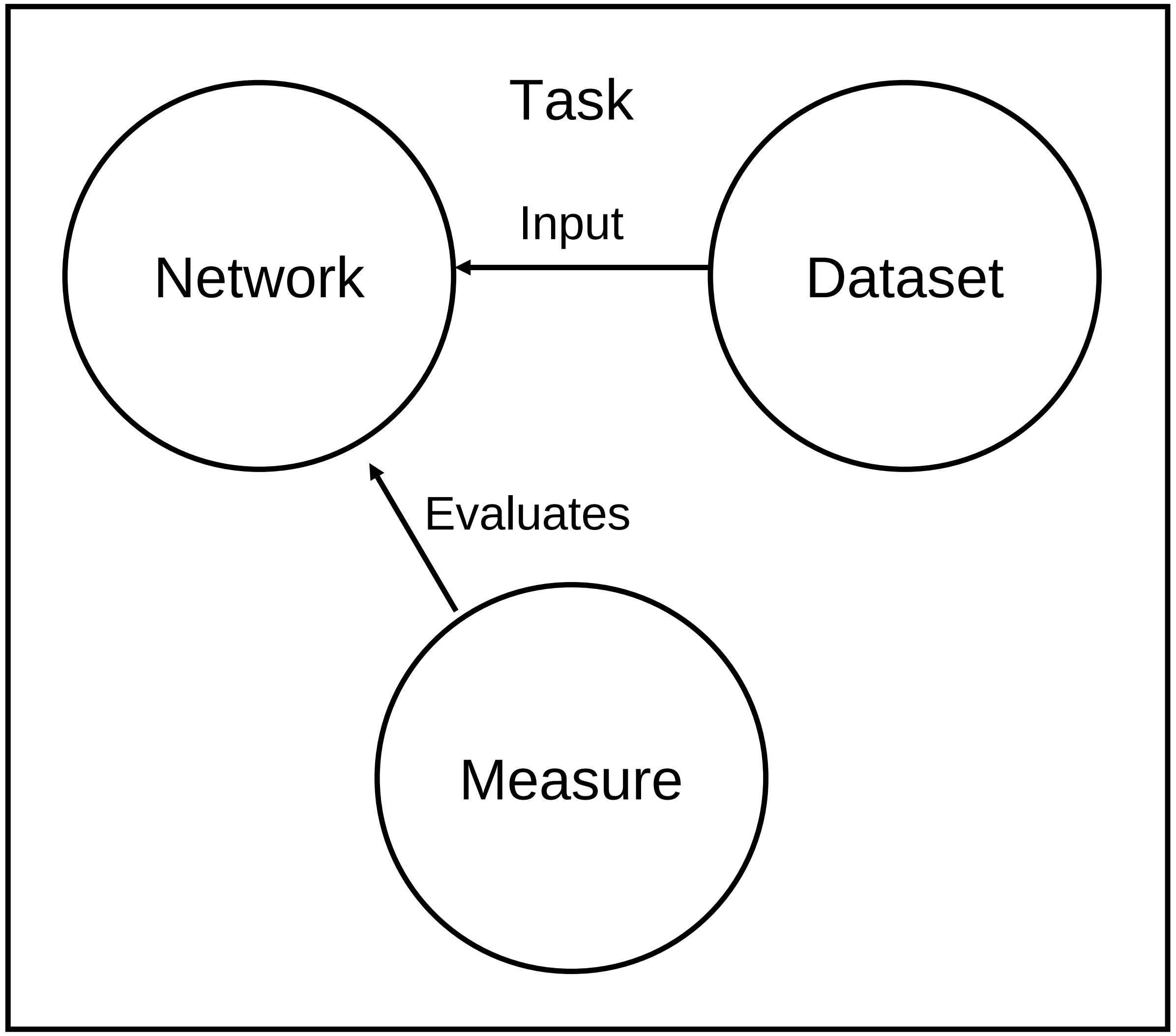


Supervised learning

- We then provide some feedback to our network using **loss**.
- The loss is a function which takes as input the output of the network $\hat{y}$,and for **supervised learning**, the correct output y

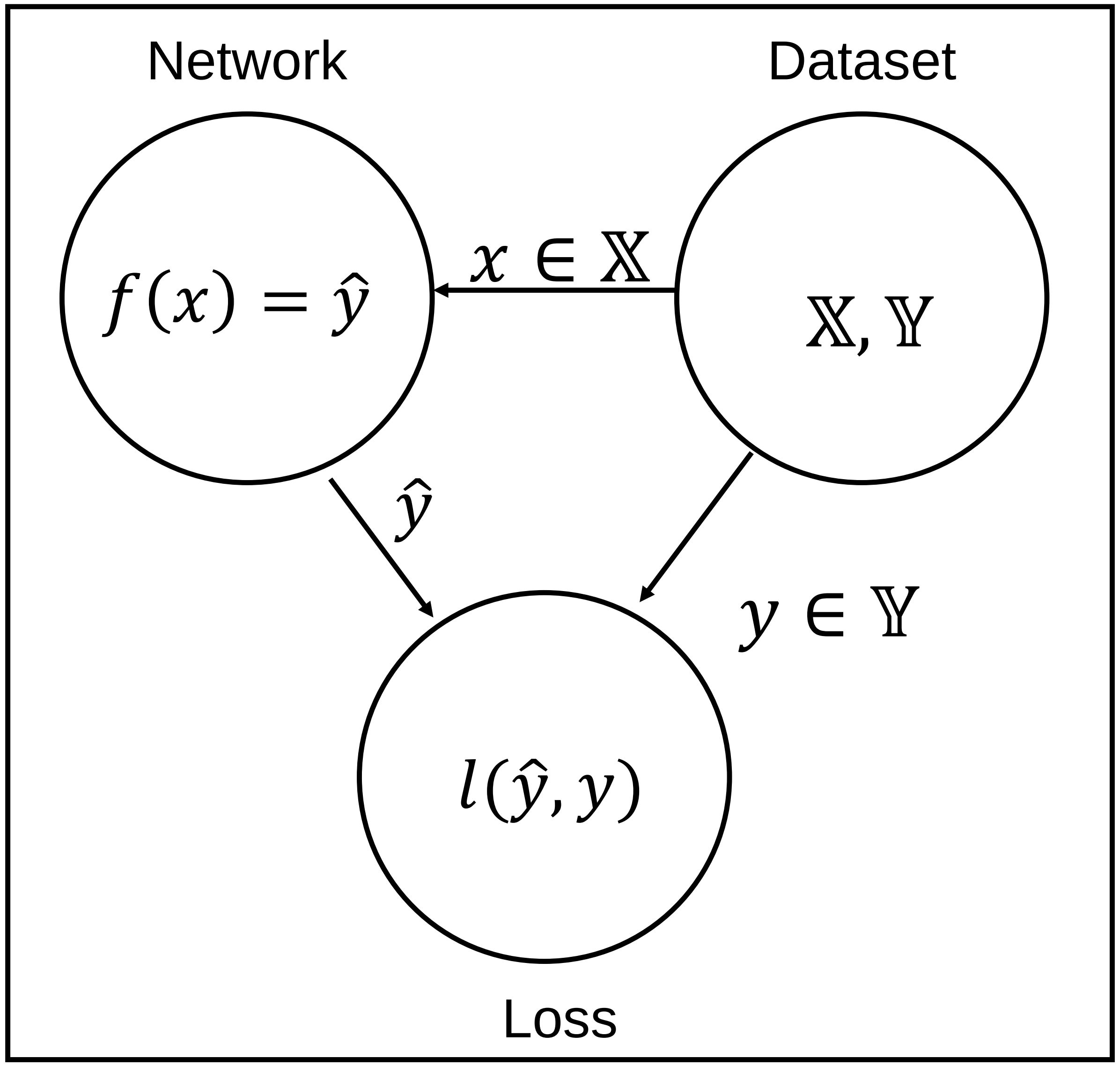
$$l(\hat{y}, y)$$

- The loss outputs a single number which tells us how well we are doing for that example.



Supervised learning

- The loss should be large when we are performing poorly and low (or 0) when we are performing well.
- The loss function is dependent on the task at hand.
- We **do not use the metric** to provide feedback to the network for technical reasons.



Example: Linear Regression

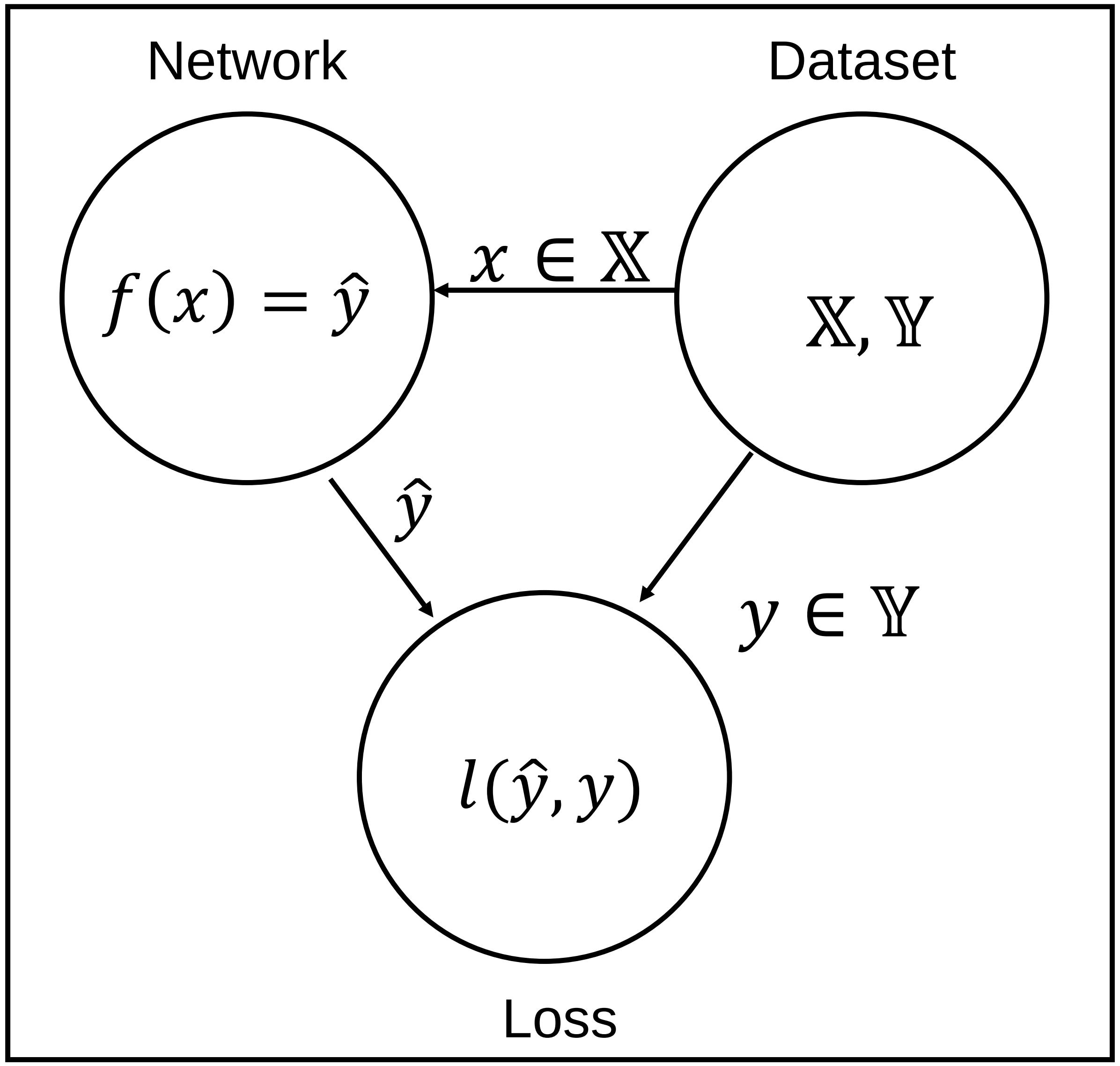
- For example, if we map our inputs directly to an output using:

$$f(x) = w^T x + b$$

- And use this loss:

$$l(\hat{y}, y) = \frac{1}{2}(\hat{y} - y)^2$$

- we get linear regression.



Understanding loss

3 data points

$$\{(x^1, y^1), (x^2, y^2), (x^3, y^3)\}$$

$$x^1 = (1,1) \quad y^1 = 2$$

$$x^2 = (1,0) \quad y^2 = 1$$

$$x^3 = (0,0) \quad y^3 = 0$$

$$\text{Loss} = l(\hat{y}, y) = |\hat{y} - y|$$

$$Total\ loss = \sum_{i=1}^{3} l(\hat{y}^i, y^i) = 0.5$$

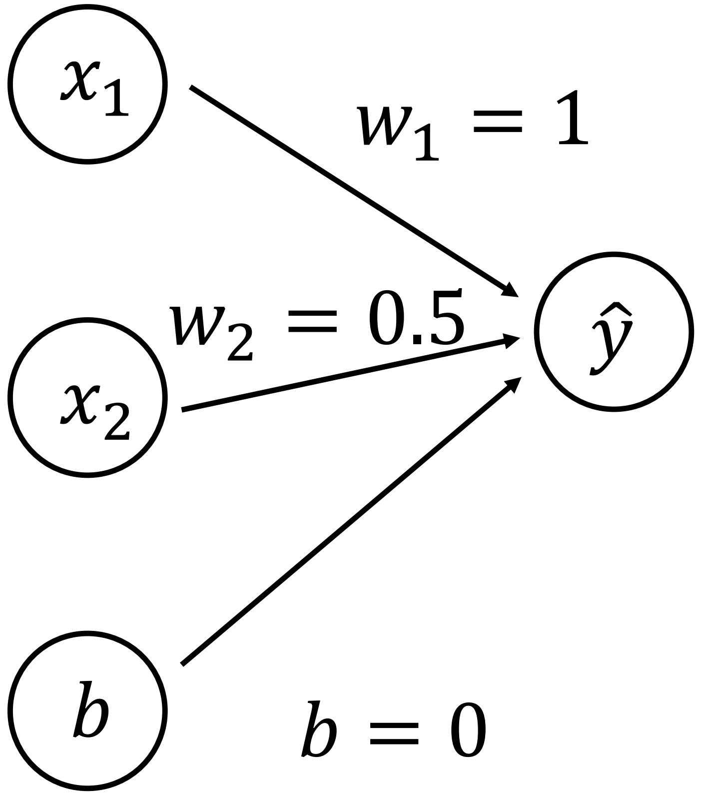
Network predictions

$$\hat{y} = f(x) = w_1 x_1 + w_2 x_2 + b$$

$$x^1 = (1,1) \quad f(x^1) = \hat{y}^1 = 1.5 \quad l(\hat{y}^1, y^1) = 0.5$$

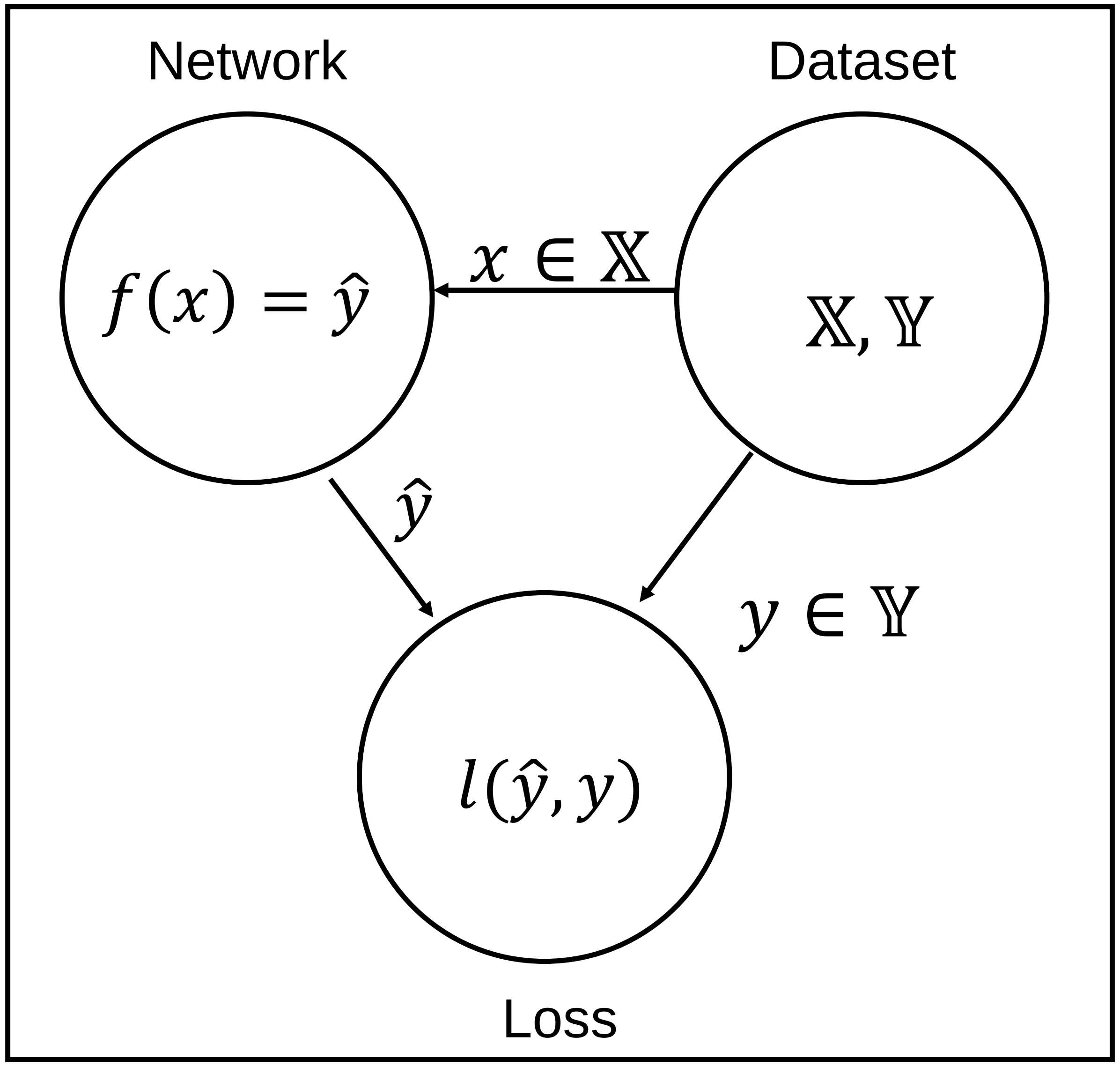
$$x^2 = (1,0) \quad f(x^2) = \hat{y}^2 = 1 \quad l(\hat{y}^2, y^2) = 0$$

$$x^3 = (0,0) \quad f(x^3) = \hat{y}^3 = 0 \quad l(\hat{y}^3, y^3) = 0$$



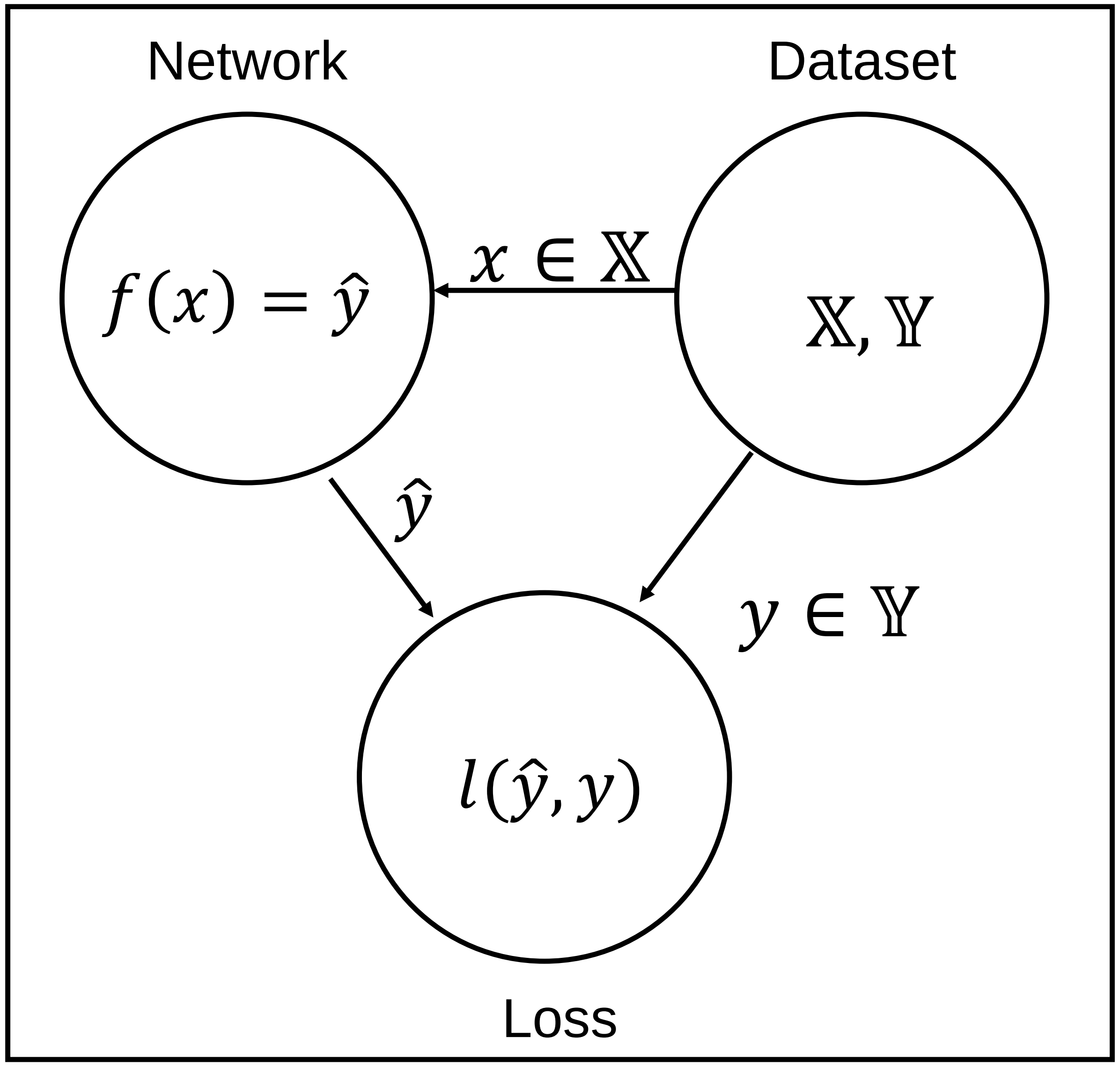
Minimising loss

- We then seek to minimise the loss over the whole dataset.
- To minimise the loss we adjust the **weights** of the network so that the loss decreases.
- We then seek to find the weights of the network which minimises the loss.



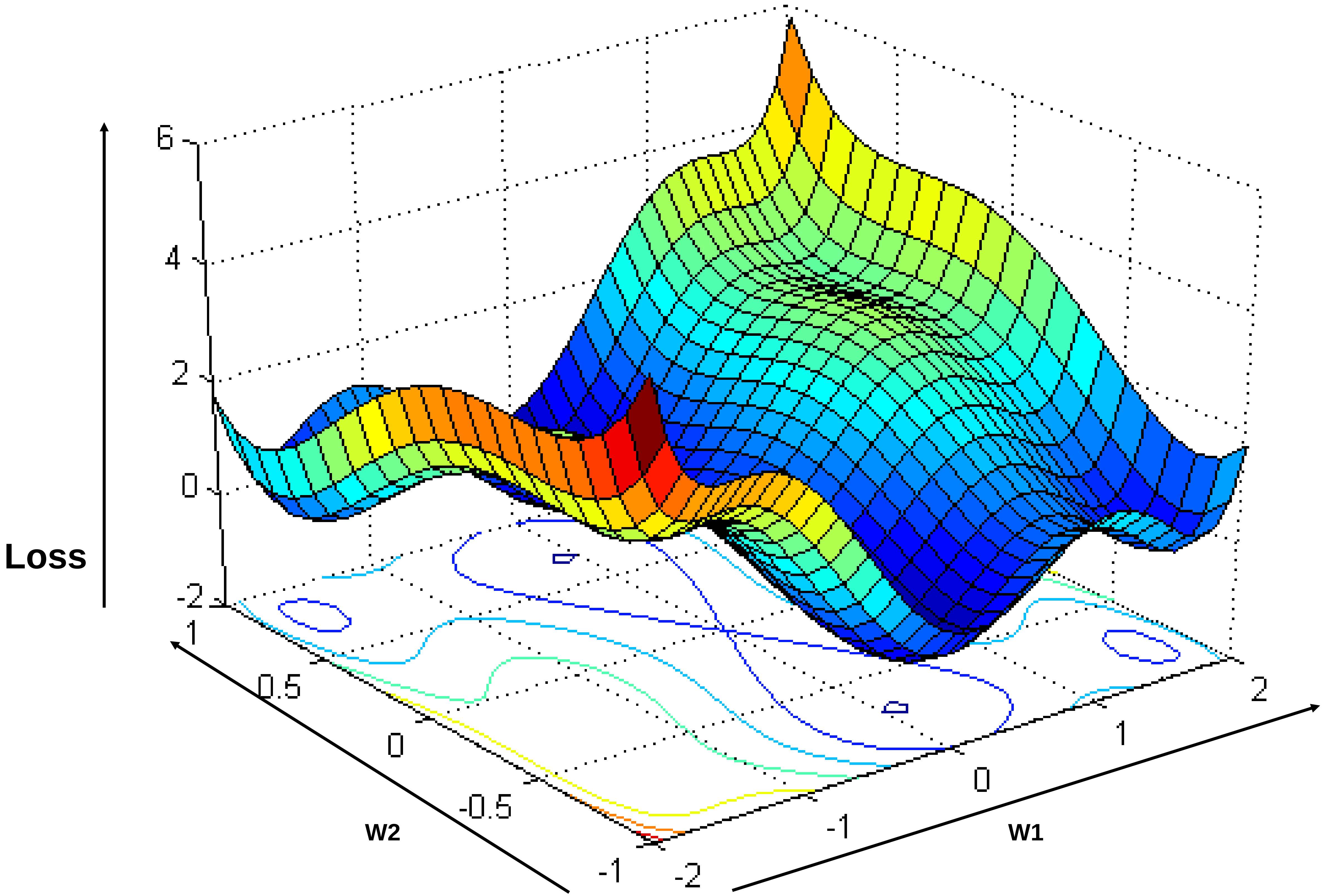
Updating weights

- How do we systematically update the weights to reduce loss?
- If our loss and network are made by using differentiable functions, we simply **differentiate the total loss w.r.t. the weights** and update the weights with that information.
- There are lots of finer details to this but the important part to know is that **each weight contributes to the total loss**.
- This means that our loss function over the whole dataset is high-dimensional function, as it is a function of every weight in the network.



$$Total\ loss = J(\boldsymbol{w})$$

All the weights in a single vector



Source: [Introduction to loss functions]

Updating weights

(batch) Gradient descent

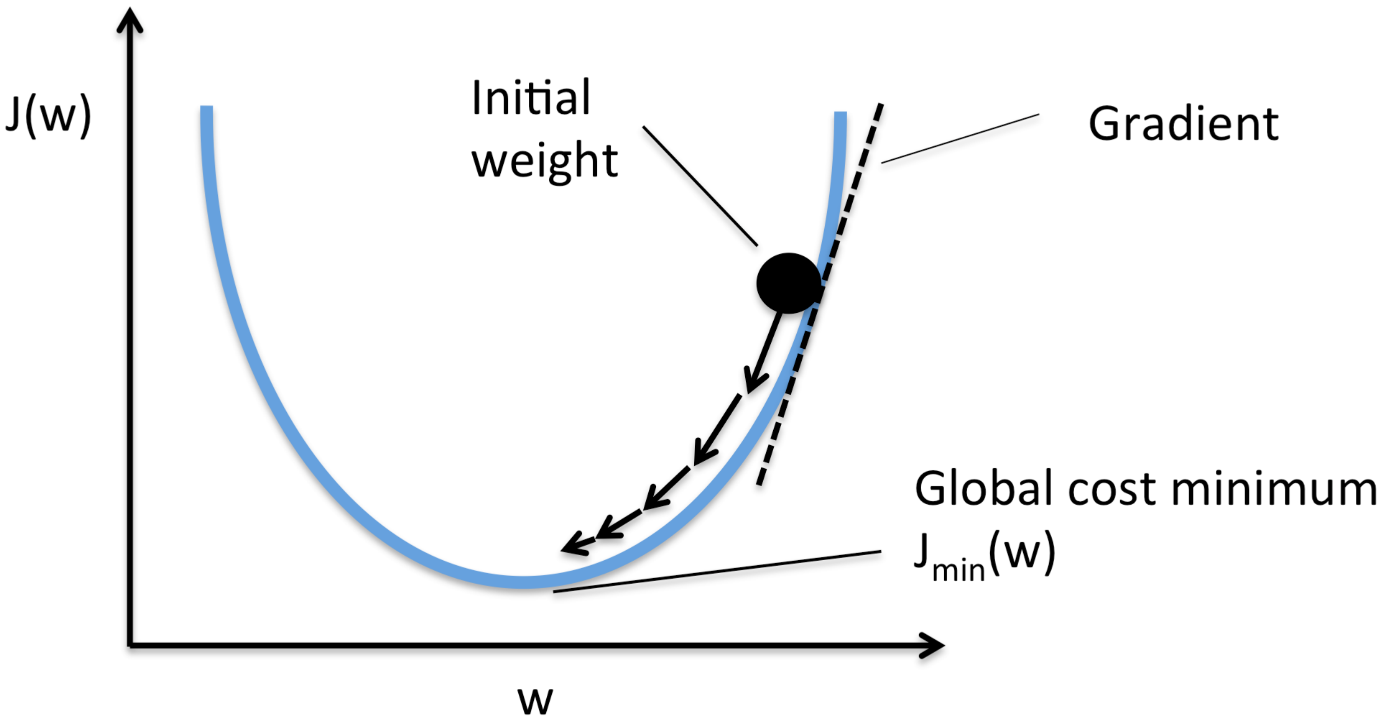
- We apply this procedure a few times, in each iteration we update the weights s.t.

$$w_i := w_i - \mu \frac{\partial J(w)}{\partial w_i}$$

  Where μ is typically a value in (0;1] called the **learning rate**

- This algorithm is called **(batch) gradient descent** (GD).

- The loss (and therefore the gradient) is computed over **all elements** in the dataset.

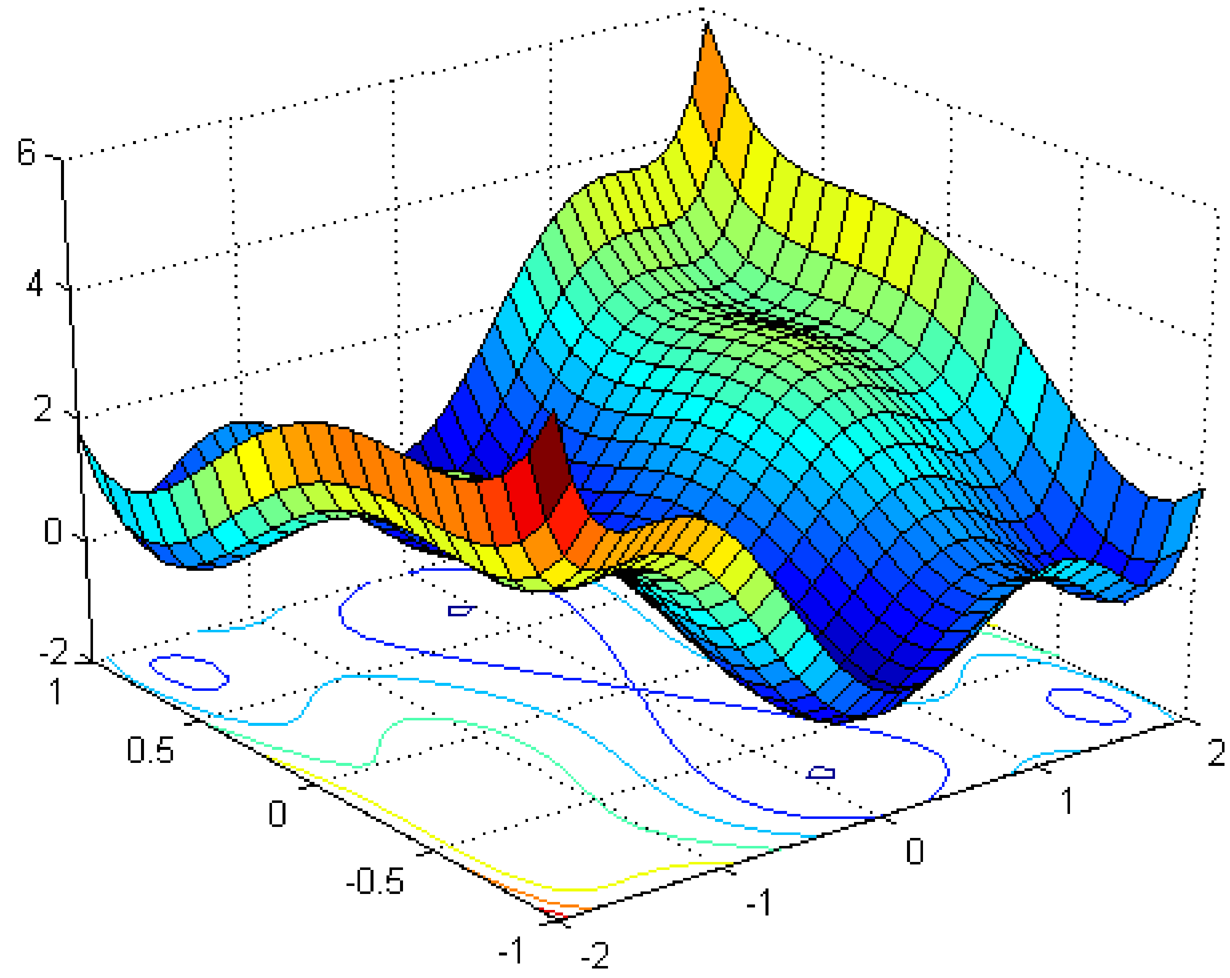


Source: [Gradient optimization]()

Updating weights

(Batch) Gradient descent

- We are not guaranteed to find a global minimum of the loss function using GD.
- In practice, it tends to find pretty good local minima.



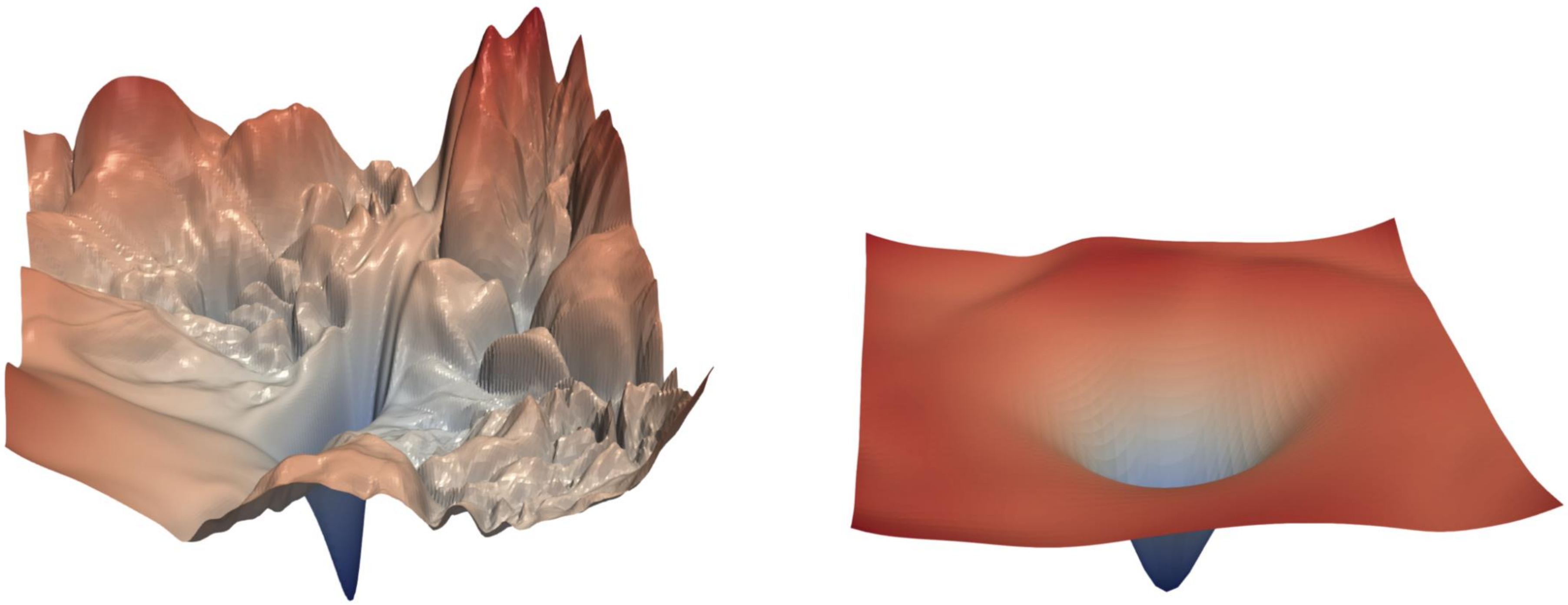
Source: [Introduction to loss functions]

Updating weights

Batch gradient descent is time-consuming! It calculates $J(\boldsymbol{w})$ based on <u>all</u> samples before doing a single weight-update…

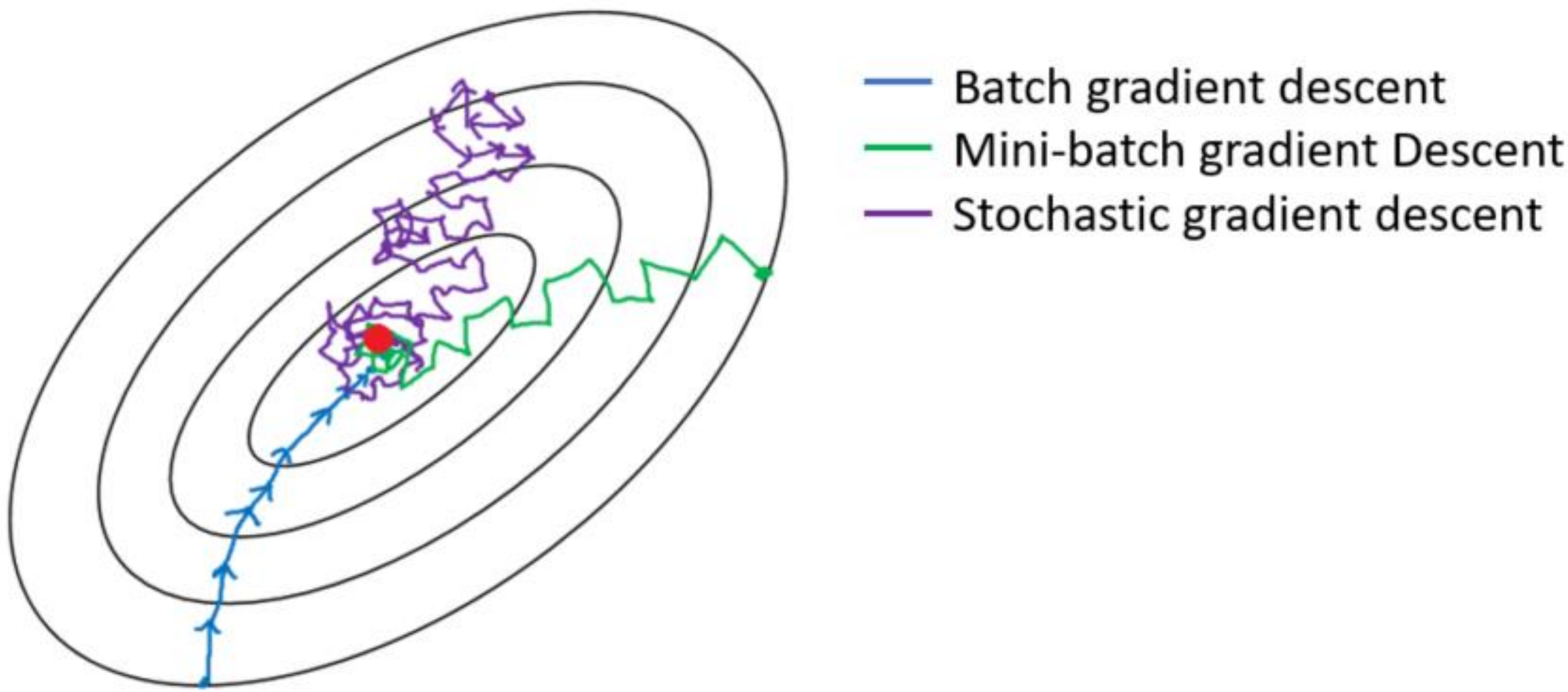
Alternatives

- Use **mini-batch** gradient descent instead (batch size typically 10—100 samples)
- Batch size = 1 is **stochastic** gradient descent



Updating weights

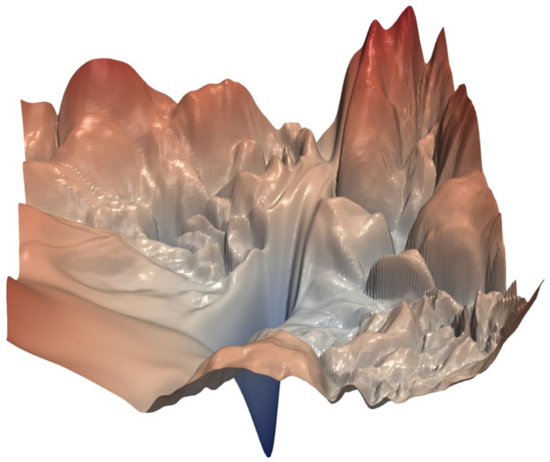
Mini-batch gradient descent

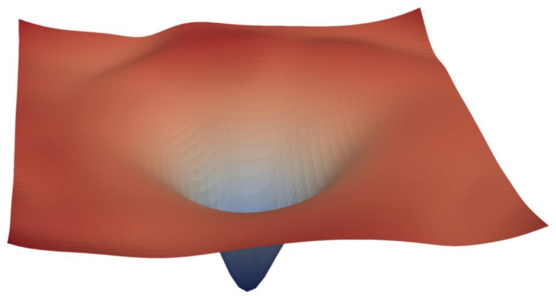


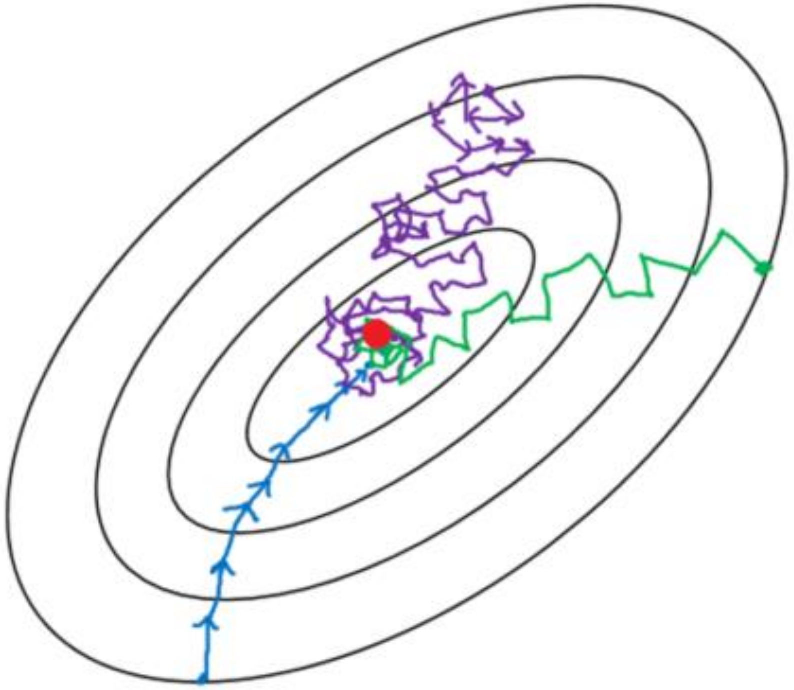
Source: Andrew Ng, deep learning course, Coursera

The problem of small batches

- SGD considers 1 sample per iteration => bad approximation of the real optimization surface
- Thus, gradient direction & size vary substantially between iterations
- Various optimizers mitigate this problem by some form of averaging of direction and / or step size (e.g. Adam)







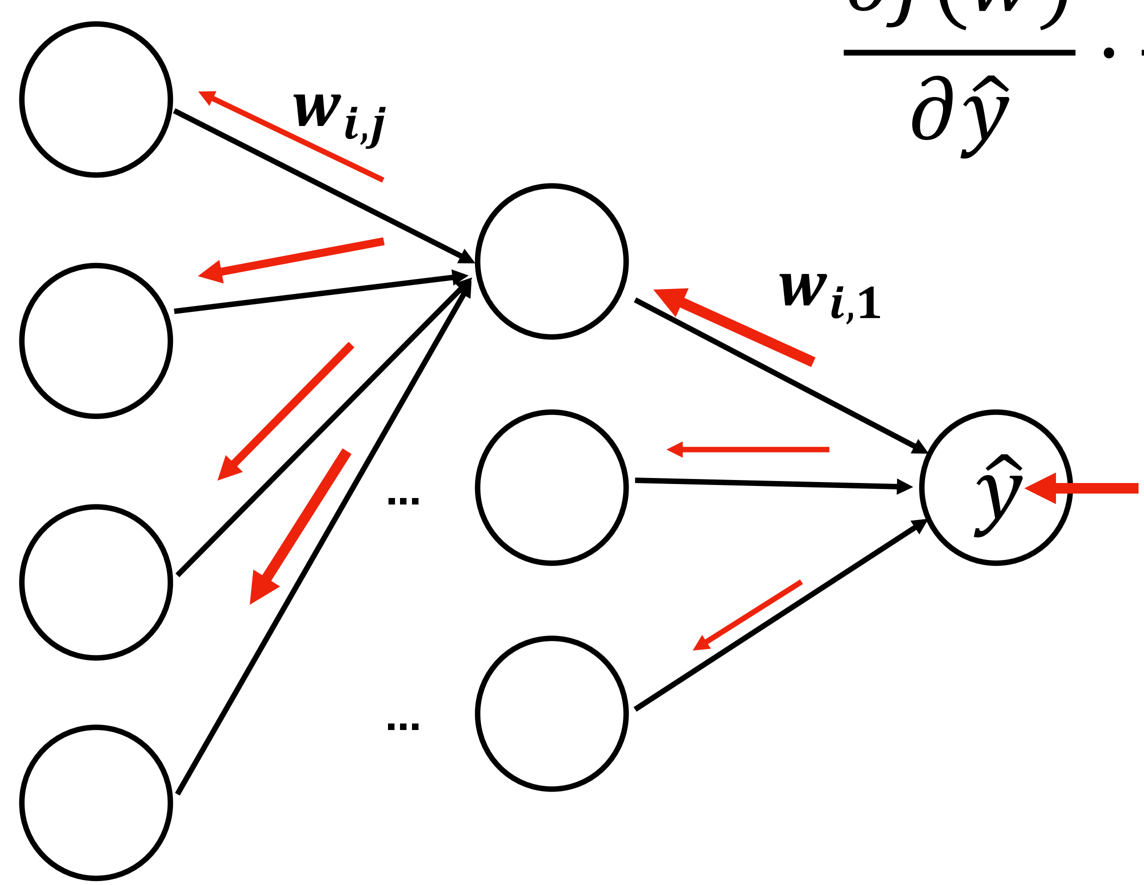
Applying GD in DL

Backpropagation

$$\frac{\partial J(w)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial a_i} \cdot \frac{\partial a_i}{\partial w_{i,j}}$$

- In the context of DL we need to compute the gradient for each layer.
- We do this by applying the **chain rule** of derivatives.
- This algorithm is known as **backpropagation**.

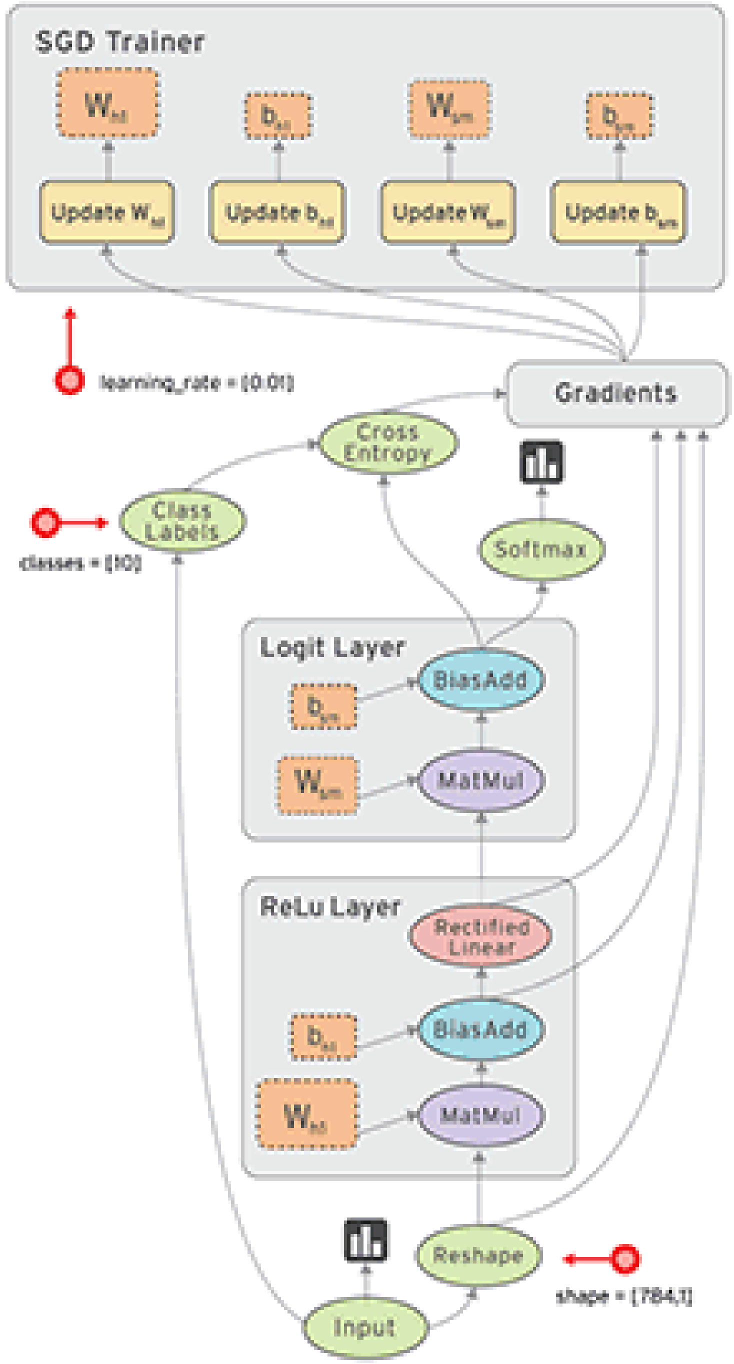
$$\frac{\partial J(w)}{\partial \hat{y}} \cdot \frac{\partial \hat{y}}{\partial w_i}$$



$$\frac{\partial J(w)}{\partial \hat{y}}$$

Keras -> TensorFlow

- We do not need to worry at all about updating these weights and differentiating since this is done by the framework (TensorFlow).
- It is still important to know what is happening when you need to debug your network.



Source:
https://www.tensorflow.org/guide/graphs

Creating neural networks

- Many frameworks exist; **TensorFlow**, **CNTK**, **Torch**, **Keras**, **Theano**, **Caffe**, ...

- We will use **Keras** (https://keras.io/)

- Keras used to call TensorFlow as a ‘backend’, but is now fully integrated in TensorFlow.

Hands-on

Go to https://https://jupyter.lisa.surfsara.nl:8000/

Notebook: 02a-keras-on-xor.ipynb

19:00-19:45

Today’s program

14:00-14:30 What is ML / DL? What is a neuron?

14:30-15:30 Hands on: building a neuron from scratch

15:30-15:45 How do neural networks work?

15:45-16:15 Break

16:15-16:30 How do neural networks work? (continued)

16:30-17:15 Hands on: building a network from scratch

17:15-18:00 Loss function & updating weights

18:00-19:00 Diner

19:00-19:45 Hands on: the XOR problem

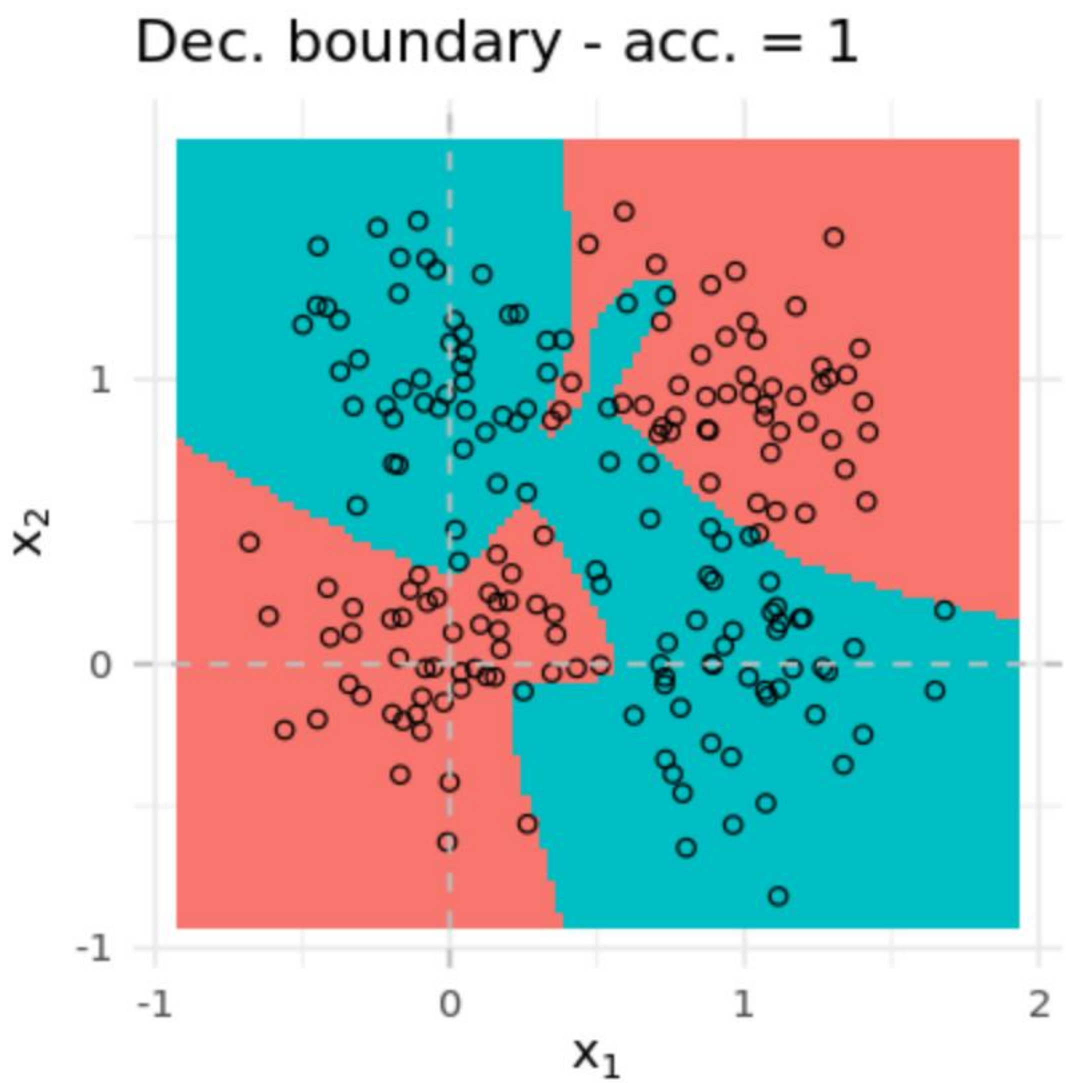
19:45-20:00 Dataset splitting & Performance evaluation

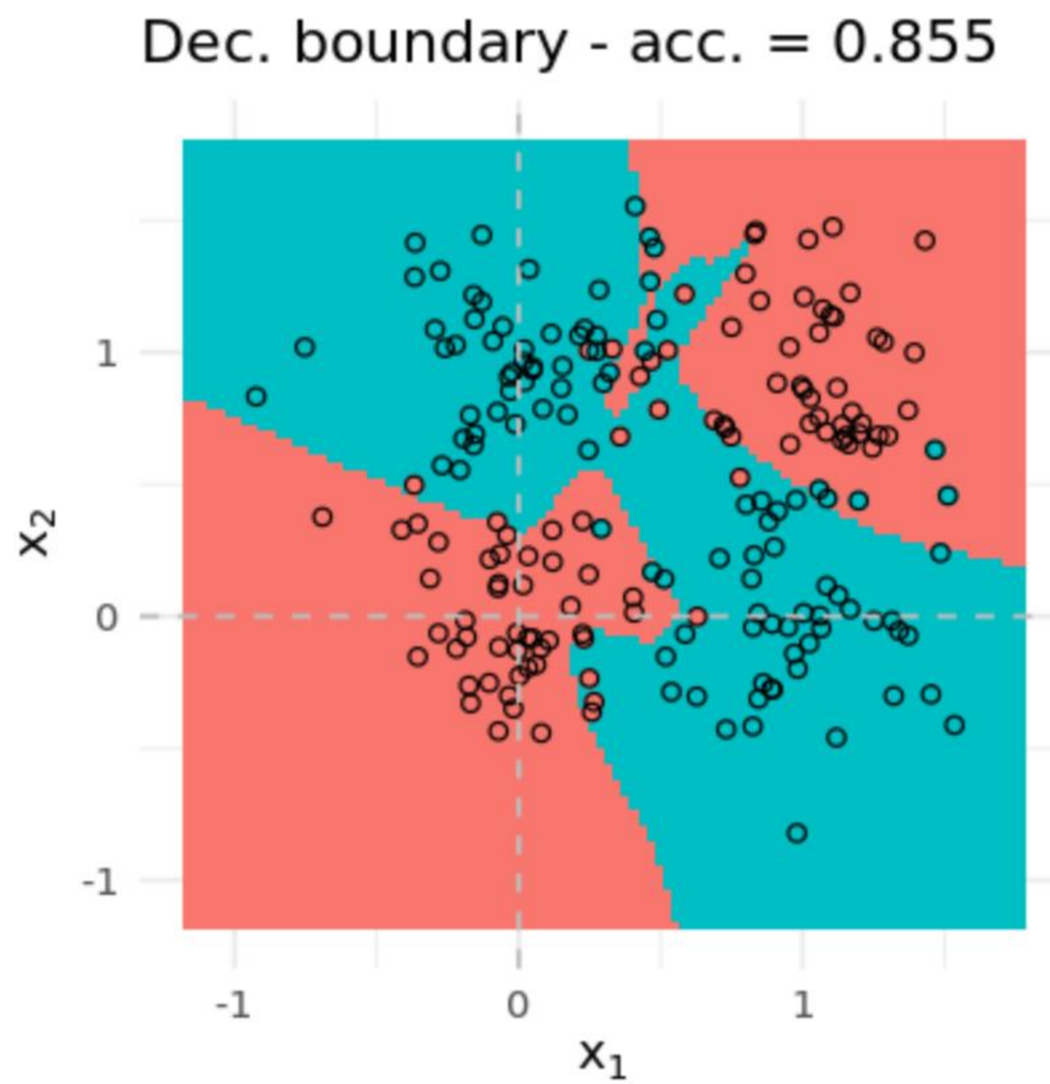
20:00-21:00 Hands on: Keras on fashion Mnist

If we finish early: Machine learning tasks (classification vs regression)

Evaluating performance

- Exercise 5's decision boundary **does not generalise**
- Use a **test set** to estimate performance on unseen examples

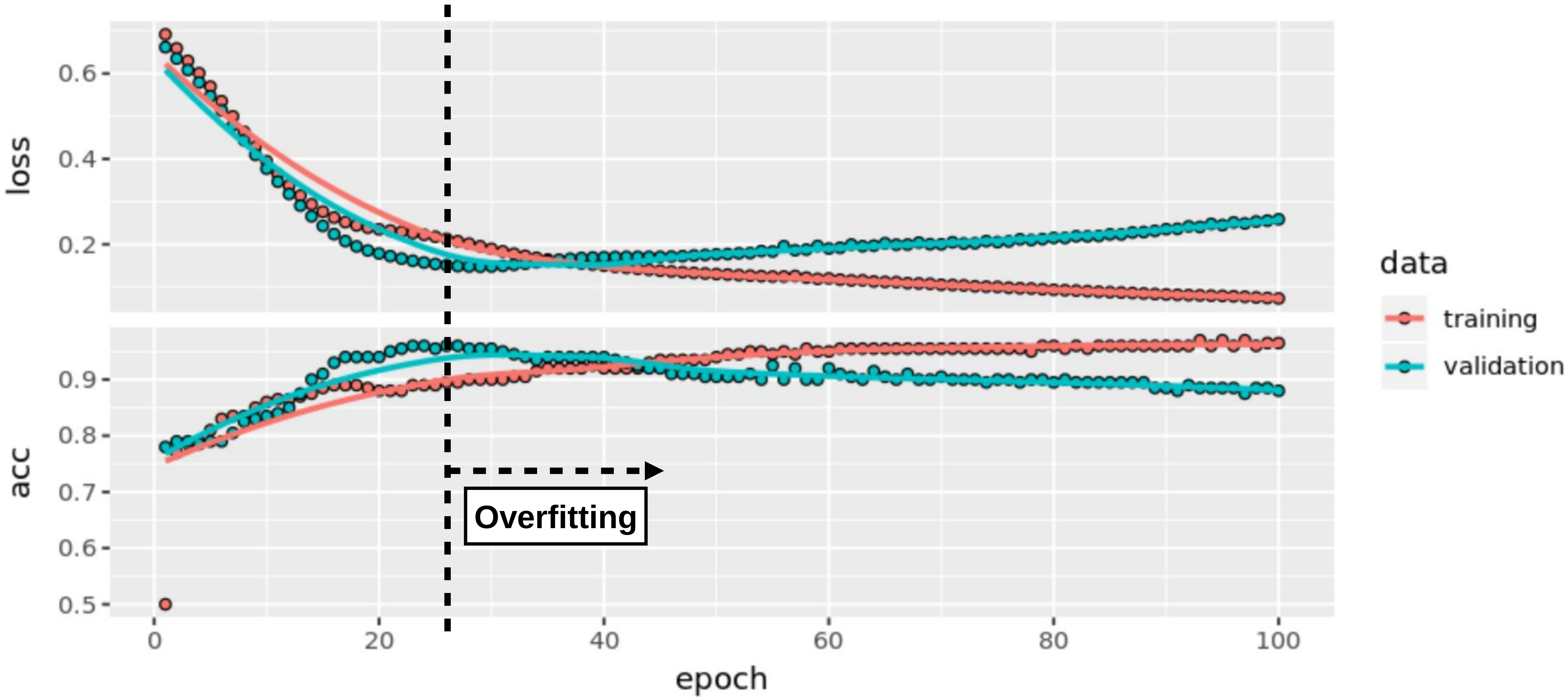




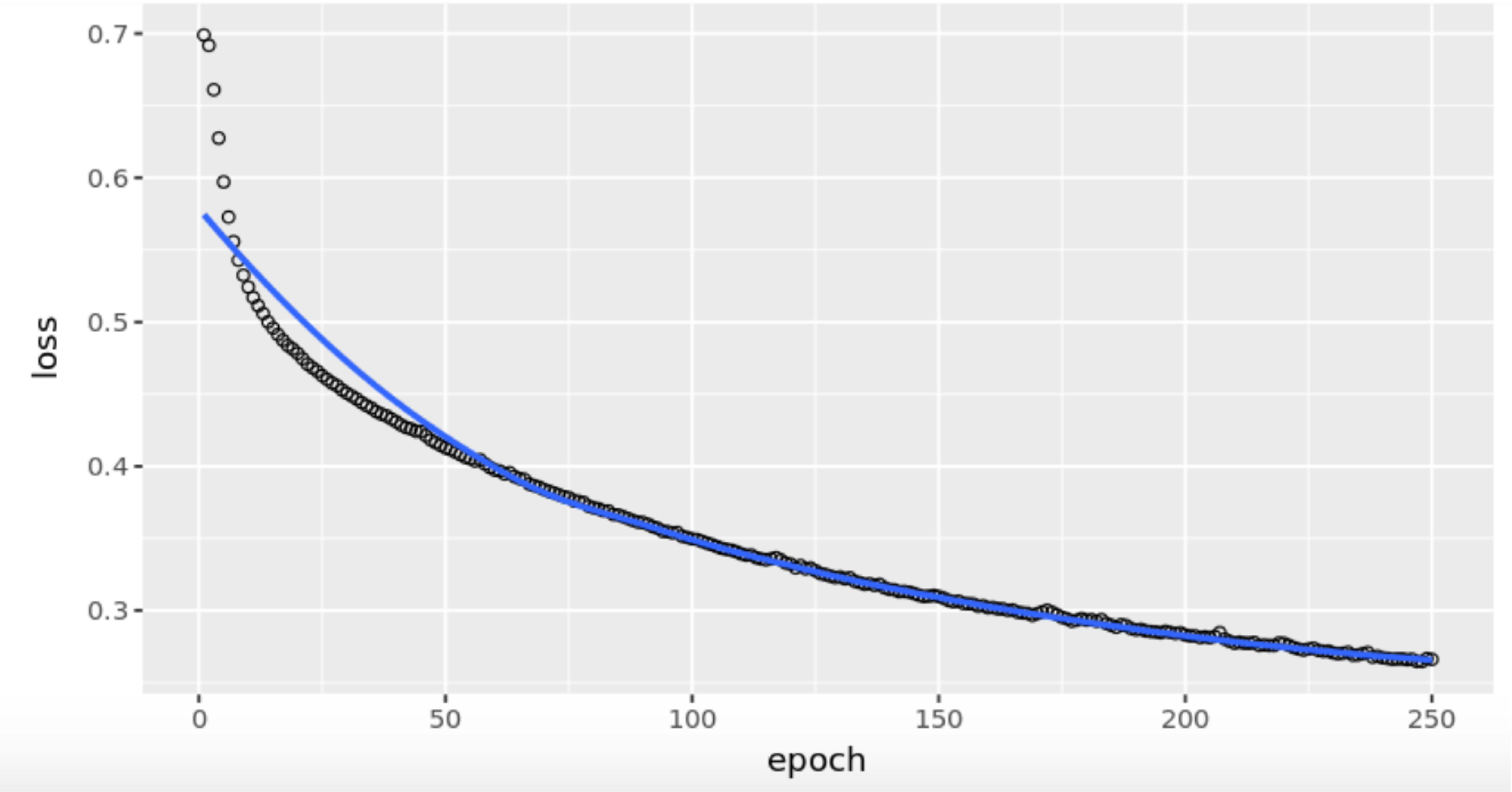
Evaluating performance

- **Hyperparameters** affect performance
  - learning rate
  - batch size
  - # layers
  - # neurons
- We need to test combinations manually
- We test different hyperparameters on a **validation set**

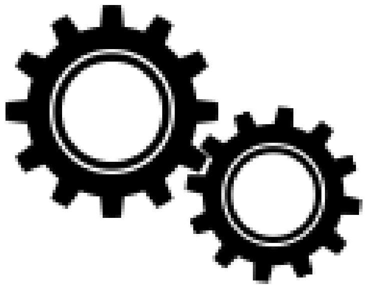
Early stopping

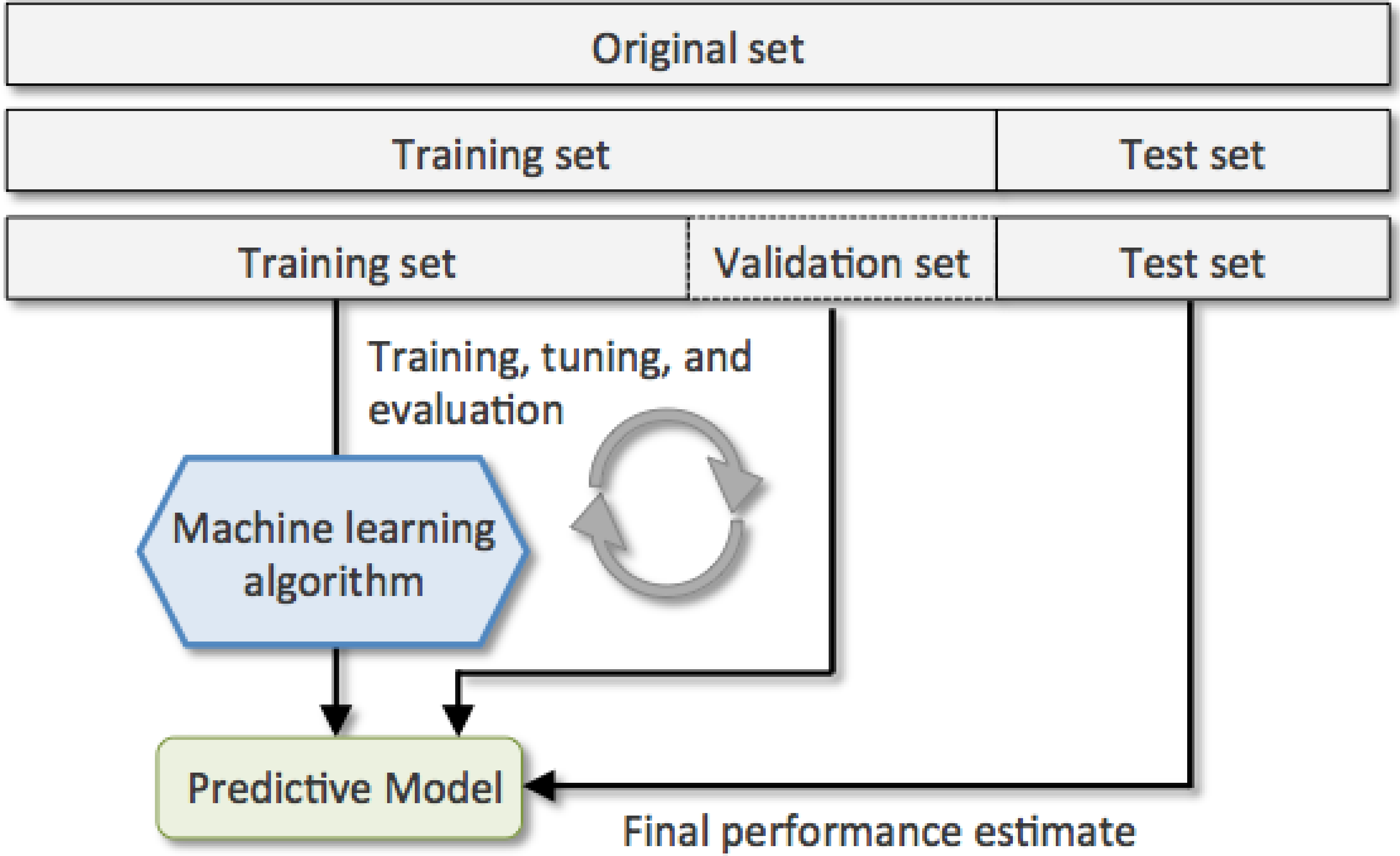


Underfitting



Training, validation, testing

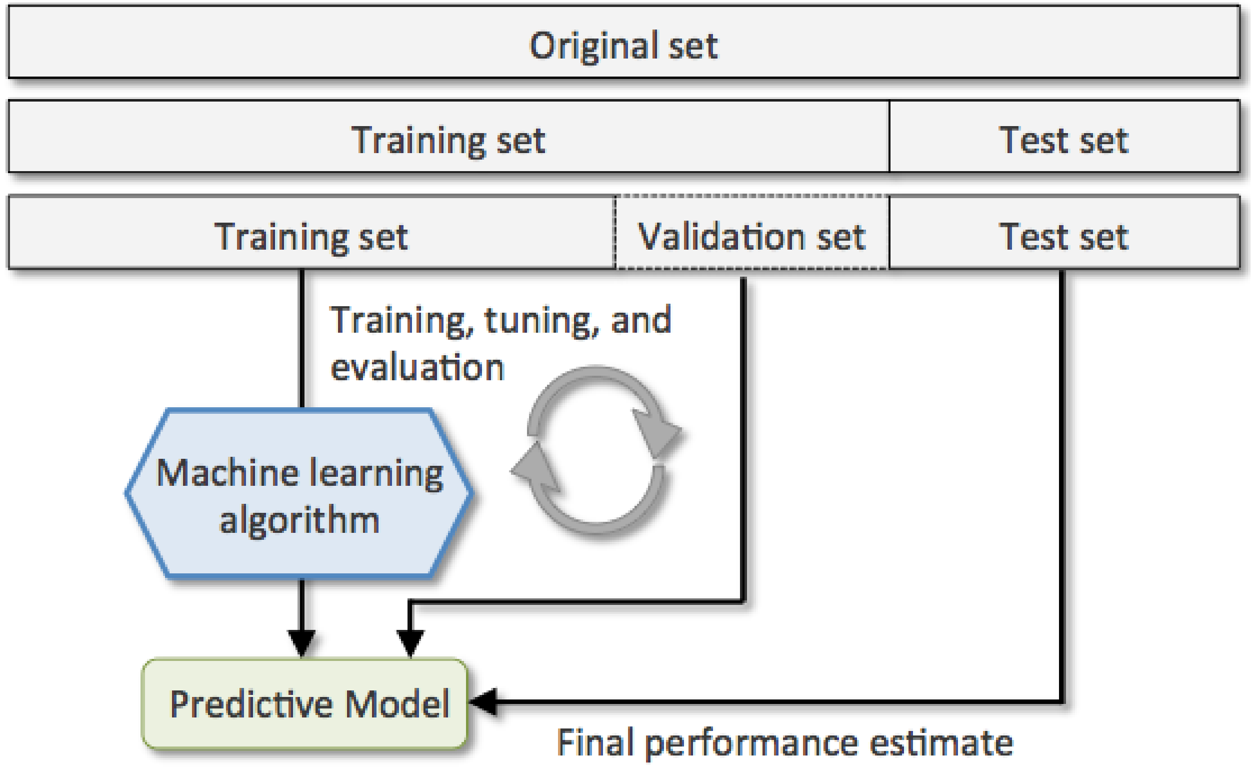
Training set	Validation set	Test set
Model training	Model selection	Model testing
		
Best model parameters Weights	**Best hyperparameters** Learning rate #neurons #layers	**Final model** Accuracy Sensitivity/specificity



Source: [Cross Validation & Ensembling]

Evaluating performance

- We generate the validation and test sets ourselves by splitting the original data set.
- Typically, people split their train/val/test set as 70/10/20.



Source: [Cross Validation & Ensembling]

Overfitting / underfitting

- In the previous notebook we did not expect our model to generalise well. In this case our model was **overfitting** the data.

- When we overfit, we see a **low training error** but **high validation / test error**.

- Similarly, if we see a **high training error**, we might be **underfitting** the data. We need to increase model capacity.

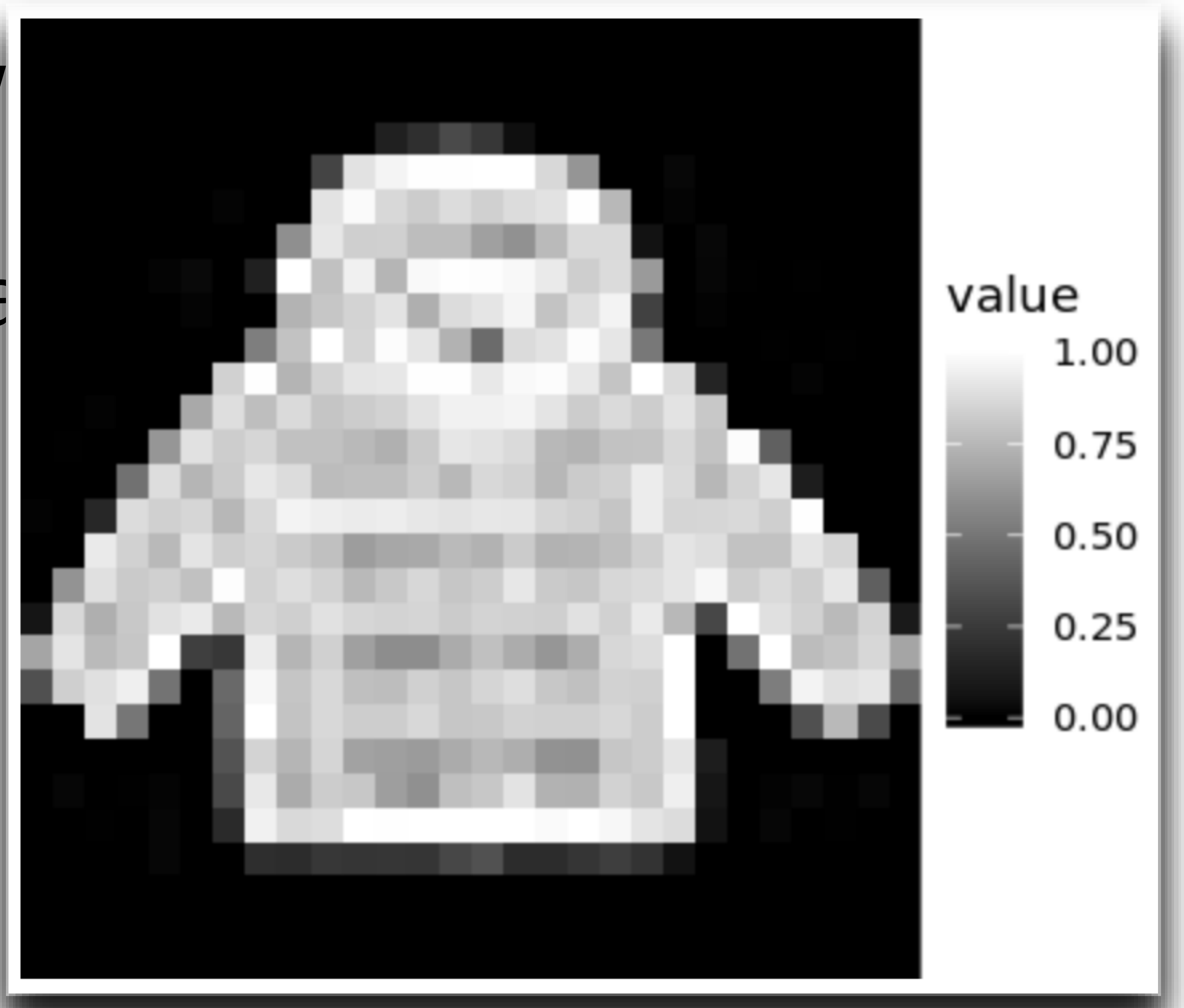
- We can test if we are underfitting by adding additional layers / more neurons and see if the training error goes down.

Training process

1. Load the data
2. Split the data into a training, validation and test set
3. Normalise the data
4. Build a simple, initial model
5. Improve the model such that it has sufficient capacity
6. Perform early stopping and evaluate the model on the test set

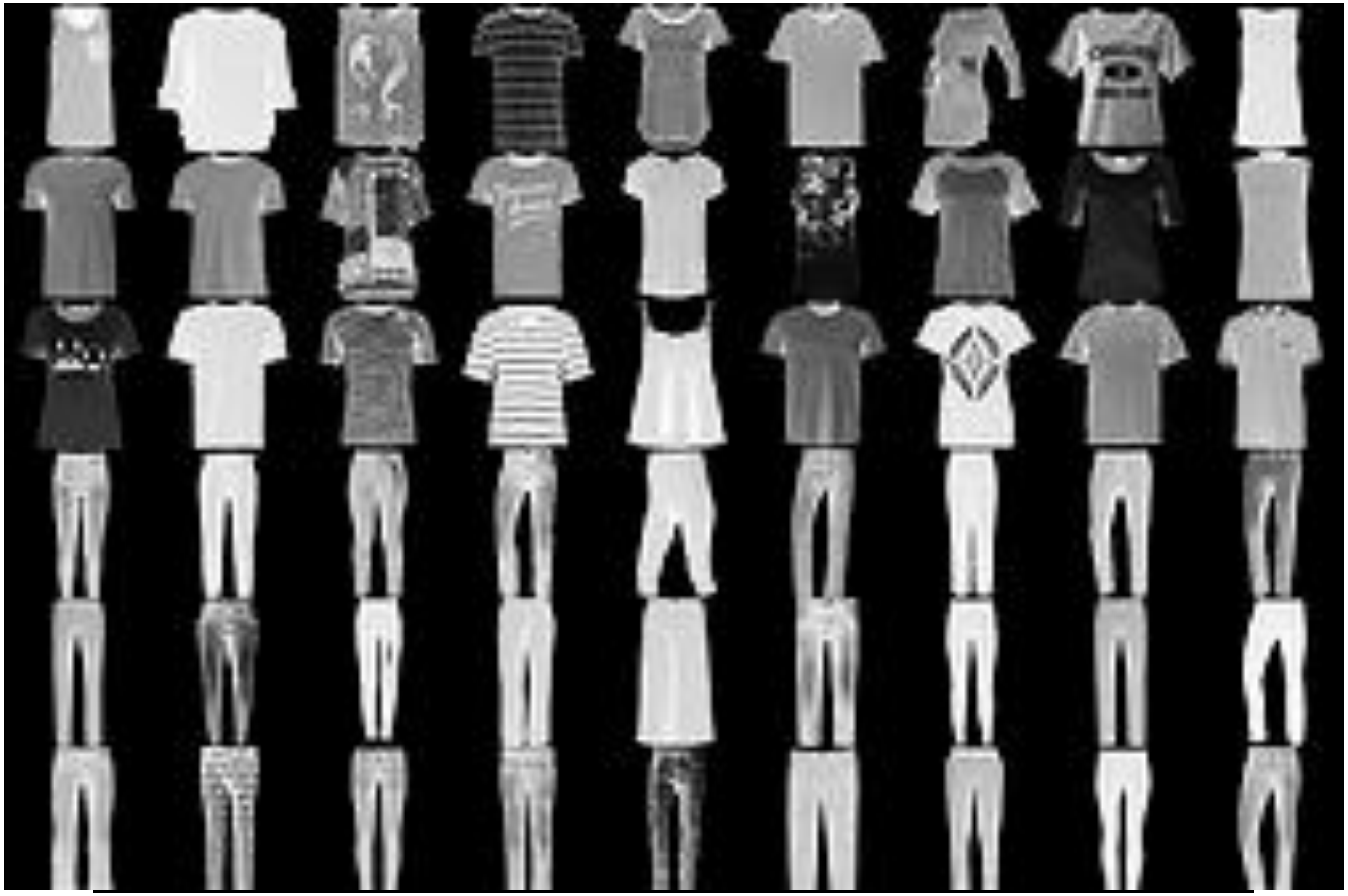
Normalisation

- Features need to have the same range
- Usually betw
- Normalise ba



The problem

Because MNIST is too easy



Hands-on

Go to https://https://jupyter.lisa.surfsara.nl:8000/

Notebook: 02b-keras-on-fashion-mnist.ipynb

20:00-21:00