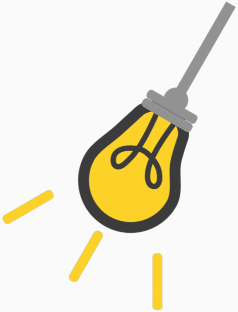
浙江大学爱丁堡大学联合学院
ZJU-UoE Institute

Lecture 11 - Introduction to neural networks

Nicola Romanò - nicola.romano@ed.ac.uk

- Describe artificial neural networks (ANNs).
- Explain the learning process of ANNs.
- Explain the concept of gradient descent.



Introduction

Artificial neural networks

An artificial neural network (ANN) is a supervised computing algorithm made up of nodes (**neurons**) that loosely resemble biological neurons.

Artificial neural networks

An artificial neural network (ANN) is a supervised computing algorithm made up of nodes (**neurons**) that loosely resemble biological neurons.

Each neuron takes in a number of inputs, performs some calculation on these inputs and outputs anothe value.

The connections between neurons (**edges**) are weighted, so that different inputs might influence the results in different ways.

Typically, neural networks consists of layers of these nodes.

Why using neural networks?

- Used in many field - adaptable to many problems
- Sufficiently complex networks can approximate any function
- Image analysis / computer vision has vastly benefitted from ANN (specifically CNN) as they can extract complex information from images
- Downside: often network computation is difficult to interpret

Why using neural networks?

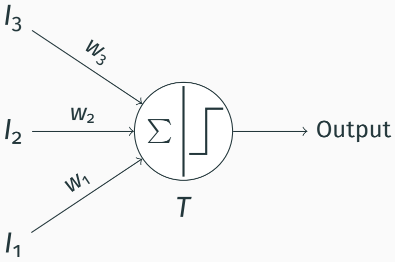
- Used in many field - adaptable to many problems
- Sufficiently complex networks can approximate any function
- Image analysis / computer vision has vastly benefitted from ANN (specifically CNN) as they can extract complex information from images
- Downside: often network computation is difficult to interpret

Today we will introduce **shallow networks**, and will move onto **deep networks** in the next lectures.

Single layer ANN

The McCulloch-Pitts Neuron

- Linear threshold unit (LTU)
- The first type of artificial neuron developed in 1943 by McCulloch and Pitts
- Little resemblance to biological neurons
- Only very simple (binary) operation possible
- Inputs can only be 0 or 1, weights could be +1 or -1 (excitatory or inhibitory)
- A simple **threshold** *T* decides the binary output.



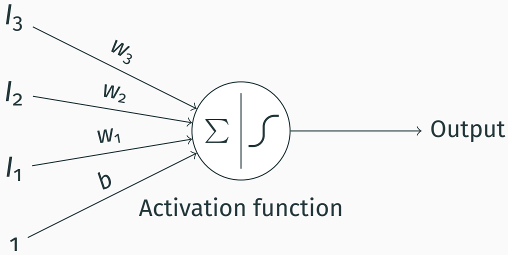
The perceptron

- A much more useful/powerful ANN
- Developed by Frank Rosenblatt in 1945
- Used as a binary classifier

The perceptron

- A much more useful/powerful ANN
- Developed by Frank Rosenblatt in 1945
- Used as a binary classifier
- Learns: $$f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$$

  x: vector of input features
  w: vector of weights
  b: bias

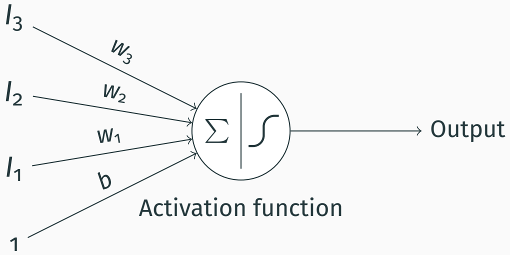


The perceptron

- A much more useful/powerful ANN
- Developed by Frank Rosenblatt in 1945
- Used as a binary classifier
- Learns: $f(\mathbf{x}) = \begin{cases} 1 & \text{if } \mathbf{w} \cdot \mathbf{x} + b > 0, \\ 0 & \text{otherwise} \end{cases}$

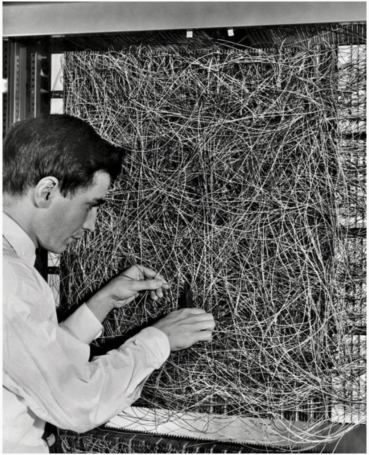
  x: vector of input features
  w: vector of weights
  b: bias

- Includes an **activation function** (e.g. a sigmoid), which can introduce non-linearity in the system, allowing to model complex functions.
- Includes a **bias term**, which allows shifting the activation function.

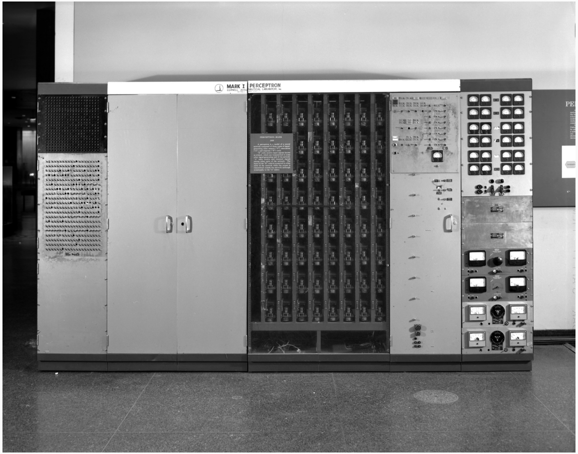


A little historical side note...

The perceptron was built as an actual machine!

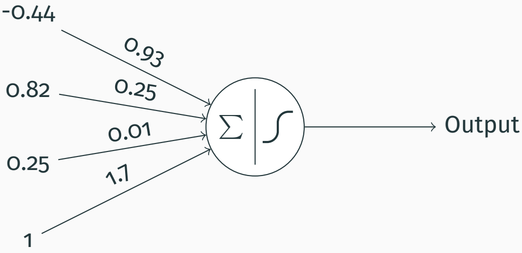


Frank Rosenblatt with a Mark I Perceptron computer in 1960



A Mark I Perceptron computer - National Museum of American History

The calculations performed by a ANN are very simple.

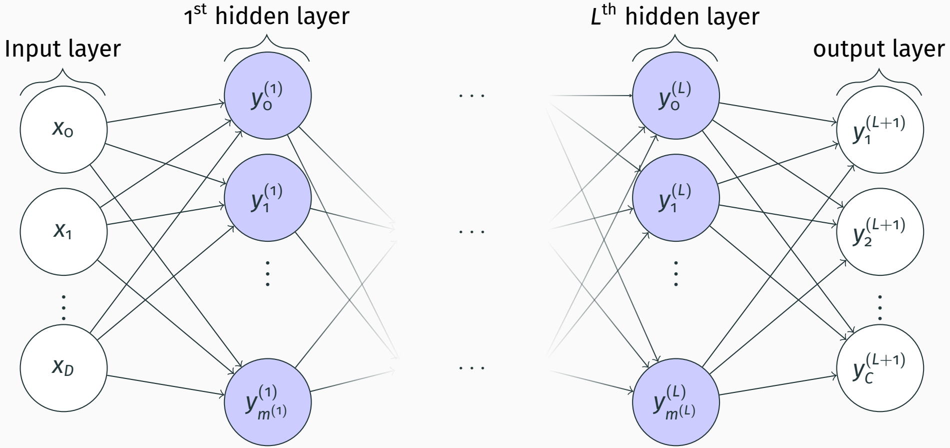


We calculate: $\sum_i (x_i \cdot w_i) + b = 0.25 * 0.01 + 0.82 * 0.25 - 0.44 * 0.93 + 1.7 = \mathbf{1.498}$ and we pass it through the activation function.

For example, using the sigmoid we get $\frac{1}{1+e^{-1.498}} = \mathbf{0.9}$.

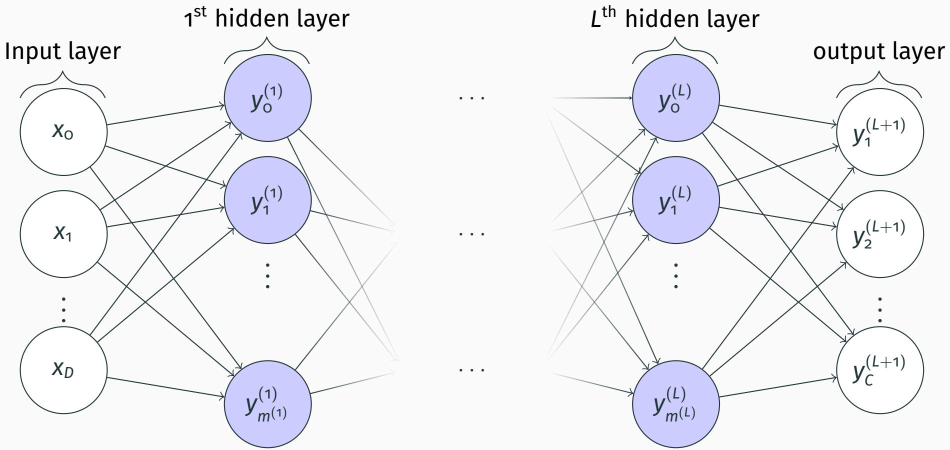
Multi-layer perceptrons

We can extend the perceptron to a **multi-layer perceptron** (MLP).



Multi-layer perceptrons

We can extend the perceptron to a **multi-layer perceptron** (MLP).



MLPs have one or more **hidden layers** that are connected to the input layer. By increasing the complexity of the network, it can perform much more complex tasks.

Forward propagation in an MLP

Forward propagation in an MLP is similar to what we just saw for a single neuron.

Forward propagation in an MLP

Forward propagation in an MLP is similar to what we just saw for a single neuron.

- We start from the first hidden layer

Forward propagation in an MLP

Forward propagation in an MLP is similar to what we just saw for a single neuron.

- We start from the first hidden layer
- We calculate the output of each neuron as the weighted sum of the inputs plus the bias, then pass it through the activation function.

Forward propagation in an MLP

Forward propagation in an MLP is similar to what we just saw for a single neuron.

- We start from the first hidden layer
- We calculate the output of each neuron as the weighted sum of the inputs plus the bias, then pass it through the activation function.
- The output of this hidden layer is used as input for the next hidden layer.

Forward propagation in an MLP

Forward propagation in an MLP is similar to what we just saw for a single neuron.

- We start from the first hidden layer
- We calculate the output of each neuron as the weighted sum of the inputs plus the bias, then pass it through the activation function.
- The output of this hidden layer is used as input for the next hidden layer.
- We repeat this process until we reach the output layer.

Activation function

Several activation functions are used in ANNs. They are used to introduce non-linearity in the ANN.

Some of the most common are:

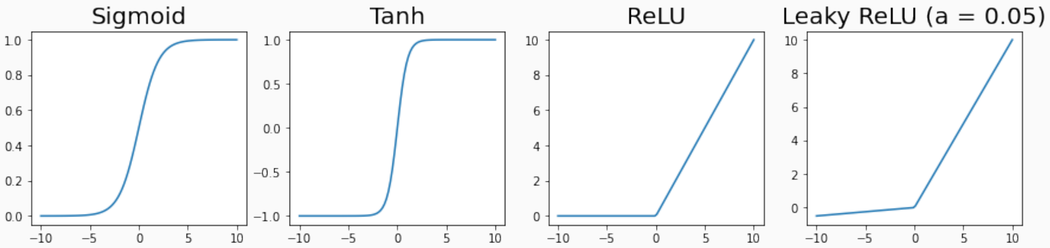
- Sigmoid
- Hyperbolic tangent (tanh)
- Rectified linear unit (ReLU)
- Leaky ReLU

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

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

$$\mathrm{ReLU}(x) = \max(0, x)$$

$$\mathrm{ReLU}(x, a) = \max(a \cdot x, x)$$

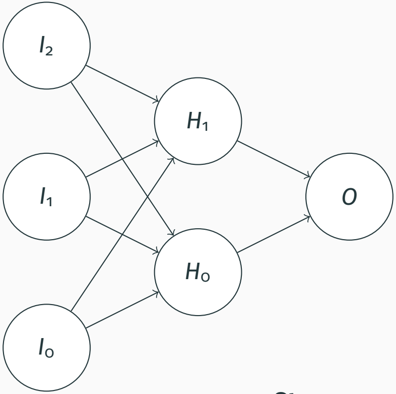


Optimization (how does the ANN learn?)

Backpropagation

Once the forward propagation is complete, we can start **backpropagation**.

This is the process of improving the weights of each node to minimise the error in the output of the network.



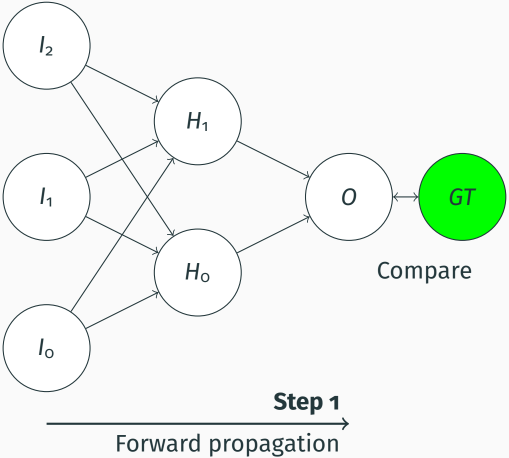
Step 1

Forward propagation

Backpropagation

Once the forward propagation is complete, we can start **backpropagation**.

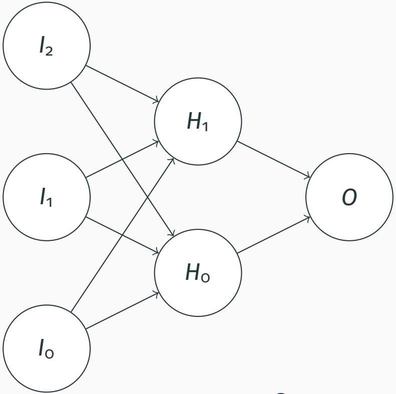
This is the process of improving the weights of each node to minimise the error in the output of the network.



Backpropagation

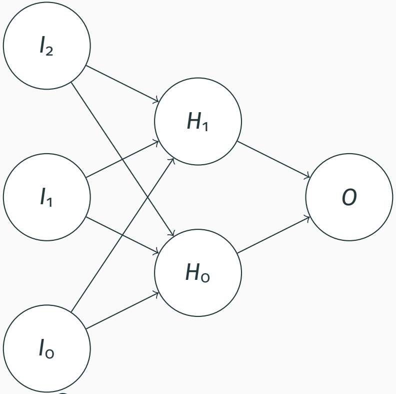
Once the forward propagation is complete, we can start **backpropagation**.

This is the process of improving the weights of each node to minimise the error in the output of the network.



Step 1

Forward propagation



Step 2

Back propagation

The training process

During training:

- We initialize the weights of the network with random numbers.

The training process

During training:

- We initialize the weights of the network with random numbers.
- We pass data and ground truth through the ANN and do a forward pass.

The training process

During training:

- We initialize the weights of the network with random numbers.
- We pass data and ground truth through the ANN and do a forward pass.
- We calculate the error in the output layer. This is done with a **loss function** (or **cost function**).

During training:

- We initialize the weights of the network with random numbers.
- We pass data and ground truth through the ANN and do a forward pass.
- We calculate the error in the output layer. This is done with a **loss function** (or **cost function**).
- We now use an **optimizer** to update the weights to minimise the loss.

During training:

- We initialize the weights of the network with random numbers.
- We pass data and ground truth through the ANN and do a forward pass.
- We calculate the error in the output layer. This is done with a **loss function** (or **cost function**).
- We now use an **optimizer** to update the weights to minimise the loss.
- We run a forward pass again and repeat the process until the loss is small enough or we reach a maximum number of iterations.

During training:

- We initialize the weights of the network with random numbers.
- We pass data and ground truth through the ANN and do a forward pass.
- We calculate the error in the output layer. This is done with a **loss function** (or **cost function**).
- We now use an **optimizer** to update the weights to minimise the loss.
- We run a forward pass again and repeat the process until the loss is small enough or we reach a maximum number of iterations.

During training:

- We initialize the weights of the network with random numbers.
- We pass data and ground truth through the ANN and do a forward pass.
- We calculate the error in the output layer. This is done with a **loss function** (or **cost function**).
- We now use an **optimizer** to update the weights to minimise the loss.
- We run a forward pass again and repeat the process until the loss is small enough or we reach a maximum number of iterations.

One of the most common optimizers is **gradient descent** (or some of its variations).

Gradient descent

- **Gradient descent** works by calculating how much the loss changes depending on each weight.

Gradient descent

- **Gradient descent** works by calculating how much the loss changes depending on each weight.
- It does so by calculating the partial derivative of the loss with respect to each weight (the gradient!)

Gradient descent

- **Gradient descent** works by calculating how much the loss changes depending on each weight.
- It does so by calculating the partial derivative of the loss with respect to each weight (the gradient!)
- It then takes a step in the direction of the gradient to go towards the minimum of the loss.

Gradient descent

- **Gradient descent** works by calculating how much the loss changes depending on each weight.
- It does so by calculating the partial derivative of the loss with respect to each weight (the gradient!)
- It then takes a step in the direction of the gradient to go towards the minimum of the loss.
- The bigger the loss, the bigger the step it takes.

Gradient descent

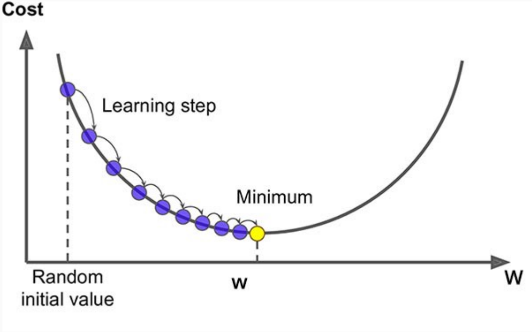
- **Gradient descent** works by calculating how much the loss changes depending on each weight.
- It does so by calculating the partial derivative of the loss with respect to each weight (the gradient!)
- It then takes a step in the direction of the gradient to go towards the minimum of the loss.
- The bigger the loss, the bigger the step it takes.
- We control this using a parameter called **learning rate** (α).

Gradient descent

- **Gradient descent** works by calculating how much the loss changes depending on each weight.
- It does so by calculating the partial derivative of the loss with respect to each weight (the gradient!)
- It then takes a step in the direction of the gradient to go towards the minimum of the loss.
- The bigger the loss, the bigger the step it takes.
- We control this using a parameter called **learning rate** (α).

Gradient descent

- **Gradient descent** works by calculating how much the loss changes depending on each weight.
- It does so by calculating the partial derivative of the loss with respect to each weight (the gradient!)
- It then takes a step in the direction of the gradient to go towards the minimum of the loss.
- The bigger the loss, the bigger the step it takes.
- We control this using a parameter called **learning rate** (α).

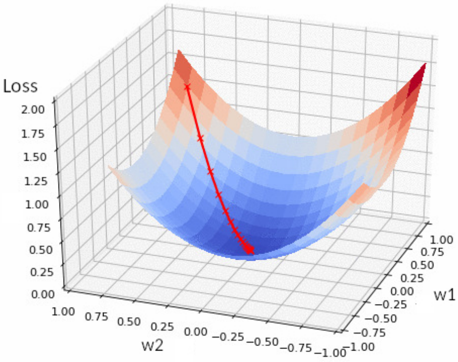


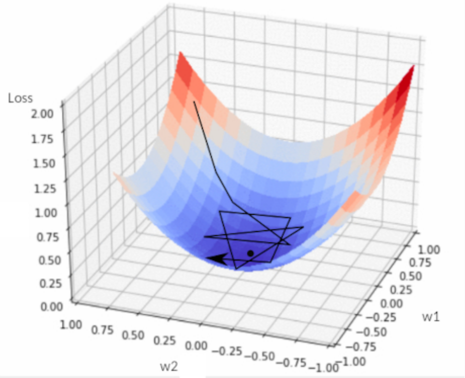
$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla J(\mathbf{w})$$

$\leftarrow$ While the image shows a single weight, in reality we need to do this for the (very) large number of parameters in the network!

Learning rate

The choice of learning rate is key!





Next lecture...

Having a general understanding of ANNs, we will look at more complex networks and start talking about deep learning!