EEE4114F: Introduction to Machine Learning

Neural Networks

Jarryd Son
Dept. Electrical Engineering
University of Cape Town

May 2020

Contents

Neural Networks 1

In this chapter we will look at the ideas that lead to the creation of artificial neural networks and how we can train a basic feedforward fully connected neural network.

1.1 Biological Neurons

Biological neurons, such as those found in our brains, are cells that can process and transmit chemical and electrical signals. Figure 1.1 illustrates the structure of a biological neuron.

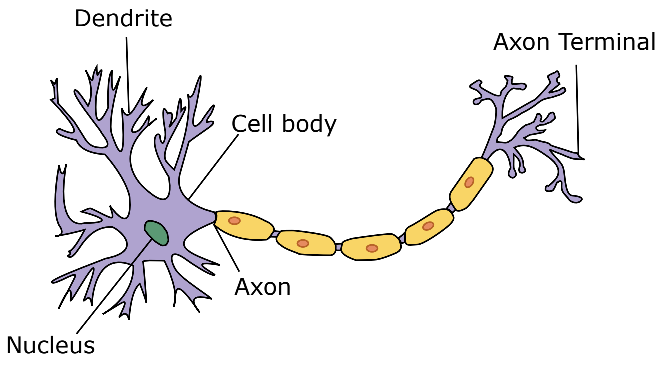

Figure 1.1: An illustration of a biological neuron (modified from [1])

A neuron receives input signals at its dendrites, these signals are chemically processed by the soma (cell body). If the incoming signals exceed a certain threshold the neuron will produce an action potential which can travel along the axon.

A neuron on its own is not significantly useful, but when these simple processing units are interconnected to form extremely large networks they become incredibly powerful [1] Neurons connect to each other via synapses that allow for the transfer of chemical or electrical signals between neurons. These synaptic connections can have varying strengths that modulate how signals are transmitted. Figure 1.2 shows how two biological neurons might be connected.

1: Hopefully I don't need to explain just how powerful the human brain is

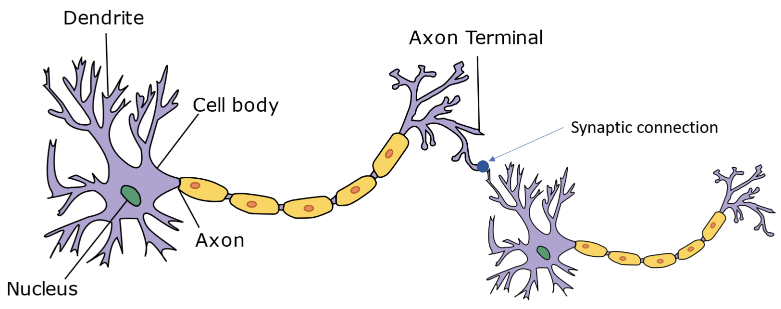

Figure 1.2: Biological neurons connect via synapses that modulate the strength of the signal transmission

In the human brain we have billions of these interconnected simple processing units. This forms the biological inspiration that resulted in the development of artificial neural networks [2].

2: Although inspired by biology, it does not mean the artifical neural networks use biologically plausible mechanisms

1.2 Artificial Neurons

An excellent resource for learning more about neural networks is [2]. The origins of modern artificial neural neurons are most commonly cited to have come from works by McCulloch and Pitts [3]. The general form of these artificial neuron is shown in Figure 1.3.

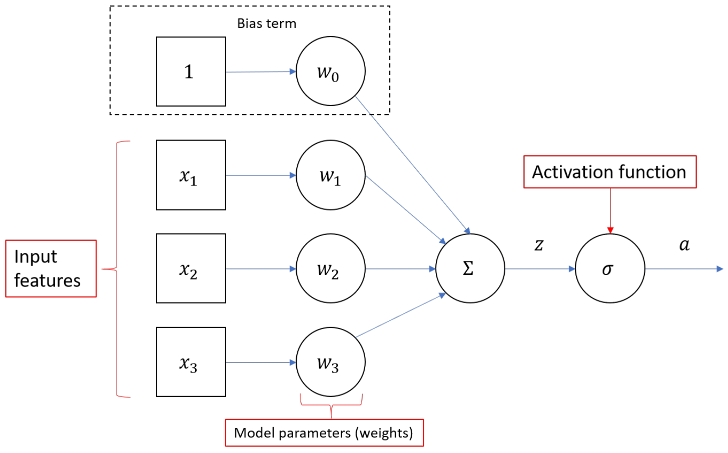

Figure 1.3: An illustration of an artificial neuron based on the MCP neuron

The output activation of a given neuron is given by:

$$a = \sigma\left(\sum_{i=1}^{m} w_i x_i + b\right)$$

Where the activation function, $\sigma(\cdot)$ is a transformation that can be applied to the net input, $z = \sum_{i=1}^{m} w_i x_i + b$. There are various activation functions that can be used [3]. If the activation function is a linear activation such that $\sigma(z) = z$, you could see that this single neuron is the same as a linear regression model.

3: More on this later

We can see some of the similarities between these types of artificial neurons and biological neurons

- Dendrites => inputs
- Synapses => weights
- Cell body (soma) => summation and activation function
- Axon => output

This is the most commonly used model, however there are other artificial neuron models such as those found in spiking neural networks that attempt to mimic the timing aspects of biological neurons. Rosenblatt [4] developed an algorithm known as the Perceptron learning rule to automatically learn weights.

Arguably the most significant contribution was the development and use of backpropagation (See Section 15) by Rumelhart et. al [5] [4].

4: Backpropagation had been developed before, but its use was not demonstrated effectively until this paper

Activation Functions

Activation functions are usually non-linear transformations, however some early neural networks such as Adaline [6] use linear activation functions. Some examples of common activation functions are shown in Figure 1.4.

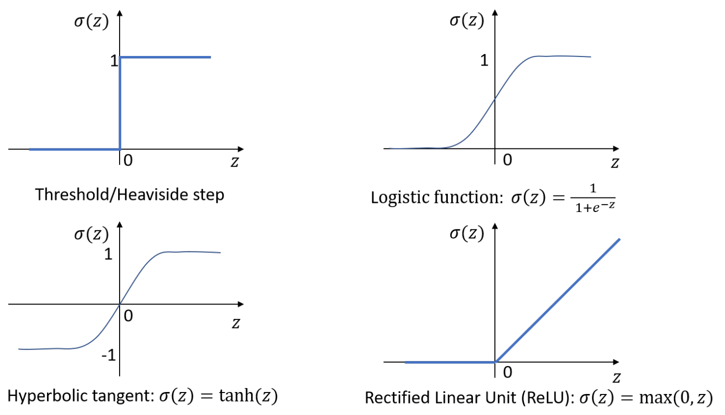

Figure 1.4: Examples of common activation functions. Many older texts will make extensive use of sigmoid activations, although modern neural networks tend to use ReLU. Sigmoid and tanh activations still have a place, and can be found in some recurrent neural network models

The threshold function behaves like a switch. If the weighted input to the neuron is positive it “fires” and produces an activation value of 1. If the weighted input is negative then the neuron is “off” and produces an activation value of 0. The logistic function [5] can be viewed as a stretched version of the threshold function. It can produce continuous values between 0 and 1. The hyperbolic tangent function [6] behaves similarly to the logistic function except that produces continuous values between -1 and 1 [7]. The ReLU function is the most popular currently. It is easy to compute both activations and gradients and can lead to sparsity which is helpful in very large networks.

5: often called a sigmoid function, although this is a more general term that can be used to describe any functions that have the characteristic ‘s’ shape

6: also a sigmoid!

7: LeCun et. al. provide some ideas regarding the choice between logistic and tanh [7]

1.3 Multilayer Artificial Neural Networks

In order to produce more complex behaviour, many neurons are connected together to form networks of neurons [8]. A basic architecture for connecting neurons is to organize them into layers and connect each neuron in one layer to every neuron in the next layer. This would be considered a fully-connected multi-layered feedforward neural network [9] The term fully-connected describes the fact that every neuron in one layer is connected to every neuron in the next. The term feedforward refers to the fact that the signals propagate in the through the network in the forward direction (from input to output)[10].

8: hence the term neural networks

9: You might often see these referred to as multilayer perceptron (MLP), however perceptrons are usually considered to have a threshold activation. In general for the multilayer neural networks we are dealing with the activation function could take a variety of forms

10: There are also recurrent neural networks that include feedback connections

Usually the diagrams for individual neurons are simplified and illustrated as neural network as shown in Figure 1.5

Each node represents the activation, a, for the given neuron. Each line connecting neurons has an associated weight denoted by w^l_{jk}, which indicates the weight connecting the j^{th} neuron in previous layer $l-1$ to the k^{th} neuron in the current layer l. The input layer connects the input features to the neurons in the first hidden layer. Hidden layers are the layers that lie between the input and output layers. There could be a

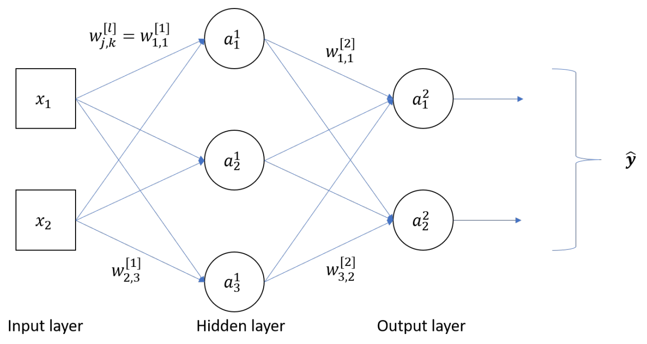

Figure 1.5: An illustration of a fully connected/dense multilayer feedforward artificial neural network.

number of hidden layers which add depth to the model, the number of neurons in a given layer would describe a neural networks width.

You might wonder why depth is important, so take a look at a network with no hidden layers in Figure 1.6.

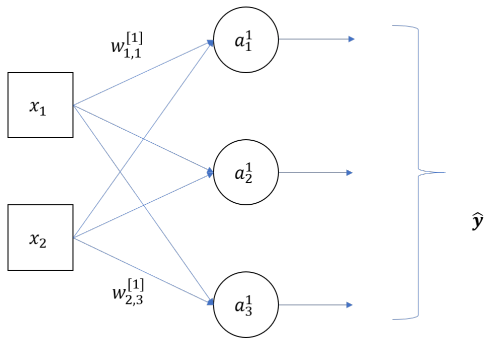

Figure 1.6: Shallow networks with no hidden layers have outputs that depend on linear combinations of the inputs

Consider the output from a shallow network with a non-linear activation function, such as a logistic function. The output would be given by

$$\sigma(\mathbf{z}) = \begin{bmatrix} \frac{1}{1+e^{-z_1}} \\ \frac{1}{1+e^{-z_2}} \\ \frac{1}{1+e^{-z_3}} \end{bmatrix} = \begin{bmatrix} \frac{1}{1+e^{-(w_{11}x_1+w_{21}x_2)}} \\ \frac{1}{1+e^{-(w_{12}x_1+w_{22}x_2)}} \\ \frac{1}{1+e^{-(w_{13}x_1+w_{23}x_2)}} \end{bmatrix}$$

Even though the output is non-linear it is still dependant on a linear combination of the input features [11]. The result is that this model can only produce linear decision boundaries [12].

11: These are considered Generalized Linear Models

12: if we are thinking about classification tasks - which is what logistic regression is used for

Now look back at what happens when you add an additional layer $l = 2$ e.g. in Figure 1.5. For just one of the neurons in the output layer

$$\sigma(\mathbf{z}^2) = \frac{1}{1+e^{-z_1^2}} = \frac{1}{1+e^{-(w_{11}^2\sigma(w_{11}x_1+w_{21}x_2)+\ldots)}}$$

Now it is dependent on a non-linear combination of the inputs and it can learn non-linear decision boundaries [13].

13: We will see this in action in later lectures

I prefer to keep all of the processes separate in order to illustrate the operation of a multilayer neural network as shown in Figure 1.7

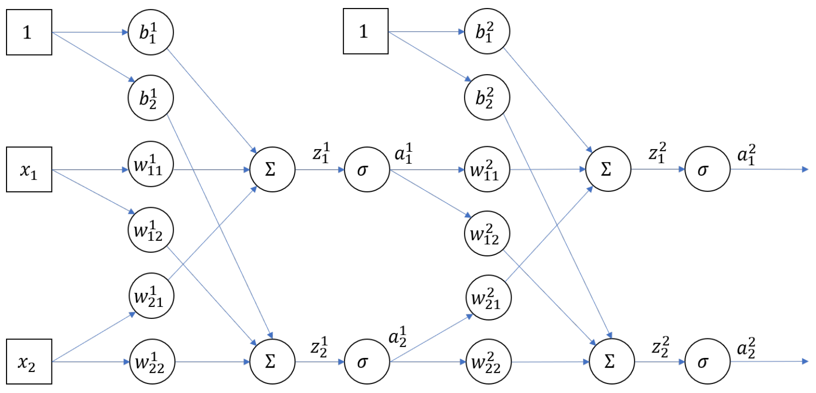

Figure 1.7: A more detailed illustration of a neural network that explicitly shows all of the components

In order to train a neural network we can use a gradient descent method such as the one we used to train our linear regression models. The process to train a neural network can be described as follows:

1. Provide data to the input layer and propagate the activations forward to generate an output.
2. Determine the error using an appropriate objective/cost on the network's output compared to a desired target.
3. Calculate the partial derivative (gradient) of the objective function with respect to each weight/parameter by backpropagating the error (see Section 15).

As with the linear regression models we can repeat this process for multiple epochs until the desired performance is achieved or until you give up and move on. The training process can be split into two phases: the forward pass (forward propagation) and the backward pass (backpropagation)

Forward Pass

From the diagram in Figure 1.7 we can derive the equations that describe the forward propagation of the activations for a given input.

For the net inputs in the hidden layer ($l = 1$)

$$\begin{aligned} z_1^1 &= w_{11}^1 x_1^1 + w_{21}^1 x_2^1 + b_1^1 \\ z_2^1 &= w_{12}^1 x_1^1 + w_{22}^1 x_2^1 + b_2^1 \end{aligned}$$

This can also be written in a matrix form [14], which is more compact to read and beneficial for computation [15]

14: Notice the transpose in this case needed to ensure the correct result

15: We can use hardware dedicated to matrix operations i.e. GPUs to accelerate performance

$$\mathbf{z}^1 = (W^1)^T \cdot \mathbf{x} + \mathbf{b}^1$$

$$\begin{bmatrix} z_1^1 \\ z_2^1 \end{bmatrix} = \begin{bmatrix} w_{11}^1 & w_{21}^1 \\ w_{12}^1 & w_{22}^1 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} + \begin{bmatrix} b_1^1 \\ b_2^1 \end{bmatrix}$$

The activations would be given by

$$\mathbf{a}^1 = \sigma\left(\mathbf{z}^1\right) = \begin{bmatrix} \sigma\left(z_1^1\right) \\ \sigma\left(z_2^1\right) \end{bmatrix}$$

For the output layer ($l = 2$) the same applies except that the input is now the activations from the previous layer.

$$\mathbf{z}^2 = (W^2)^T \cdot \mathbf{a}^1 + \mathbf{b}^2$$
$$\begin{bmatrix} z_1^2 \\ z_2^2 \end{bmatrix} = \begin{bmatrix} w_{11}^2 & w_{21}^2 \\ w_{12}^2 & w_{22}^2 \end{bmatrix} \cdot \begin{bmatrix} a_1^1 \\ a_2^1 \end{bmatrix} + \begin{bmatrix} b_1^2 \\ b_2^2 \end{bmatrix}$$

The activations would be as follows:

$$\mathbf{a}^2 = \sigma\left(\mathbf{z}^2\right) = \begin{bmatrix} \sigma\left(z_1^2\right) \\ \sigma\left(z_2^2\right) \end{bmatrix}$$

In this example the activations $\mathbf{a}^2$ would be the output predictions $\hat{\boldsymbol{y}}$

Once the output has been determined you can calculate the error, for example, the squared error loss:

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

Now we need to determine the gradients for each weight.

Backpropagation

Backpropagation is an efficient algorithm for determining the partial derivative of the objective function with respect to a given weight. This was an important development, as it allowed for the training of neural networks with many layers [16]. It sounds fancy but it is really just a nice application of the chain rule.

16: Prior to the adoption of backpropagation, neural networks were severly limited and struggled to compete at the time.

Just to recap the chain rule:

$$\frac{d}{dx}\left[f(g(h(x)))\right] = \frac{df}{dg} \cdot \frac{dg}{dh} \cdot \frac{dh}{dx}$$

For multiple variables:

$$\frac{d}{dx}\left[f(g(x),(h(x))\right] = \frac{\partial f}{\partial g} \cdot \frac{dg}{dx} + \frac{\partial f}{\partial h} \cdot \frac{dh}{dx}$$

To start let's look at a single weight w_{11}^1. The goal is to determine the gradient of the cost function with respect to this weight $\frac{\partial J}{\partial w_{11}^1}$ Using the simplified graph in Figure 1.8 we can look at how chain rule might apply.

We can expand $\frac{\partial J}{\partial w_{11}^1}$, as J is composed of multiple functions if you follow the graph.

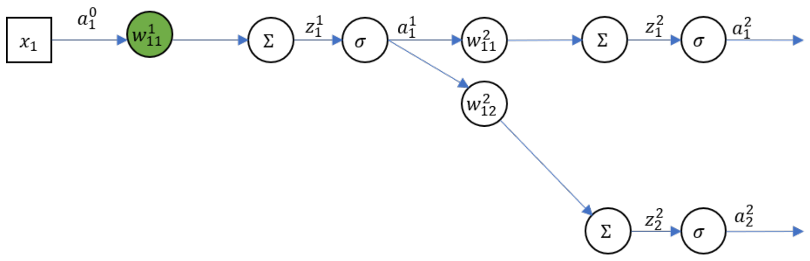

Figure 1.8: A simplified illustration of a neural network to make it easier to see backpropagation in action

$$\frac{\partial J}{\partial w_{11}^1} = \frac{\partial J}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial a_1^1} \cdot \frac{\partial a_1^1}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{11}^1} + \frac{\partial J}{\partial a_2^2} \cdot \frac{\partial a_2^2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial a_1^1} \cdot \frac{\partial a_1^1}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{11}^1} \tag{1.1}$$

This is quite cumbersome, but it is easy to follow if you read it along with the graph starting from right to left. We can also find a more general form that applies for any weight, but let's stick with this single weight for now and break it down.

$\frac{\partial J}{\partial a_1^2}$ should be easy to find because it is the output layer and therefore we can directly compute the gradient, as we know the output from the forward pass. For example, if $J = \frac{1}{2}(\hat{\boldsymbol{y}} - \boldsymbol{y})^2$ then

$$\frac{\partial J}{\partial a_1^2} = a_1^2 - y_1$$

.

This is dependant on the chosen objective function. The next term is given by

$$\frac{\partial a_1^2}{\partial z_1^2} = \sigma'(z_1^2) \tag{1.2}$$

We are assuming the activation function is differentiable or at least piecewise differentiable. Considering that $z_1^2 = w_{11}^2 a_1^1 + w_{21}^2 a_2^1 + b_1^2$

$$\frac{\partial z_1^2}{\partial a_1^1} = w_{11}^2 \tag{1.3}$$

$$\frac{\partial a_1^1}{\partial z_1^1} = \sigma'(z_1^1) \tag{1.4}$$

Considering that $z_1^1 = w_{11}^1 a_1^0 + w_{21}^1 a_2^0 + b_1^1 = w_{11}^1 x_1 + w_{21}^1 x_2 + b_1^1$

$$\frac{\partial z_1^1}{\partial w_{11}^1} = a_1^0 = x_1 \tag{1.5}$$

$$\tag{1.6}$$

The first term in 1.1 comes to

$$\frac{\partial J}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial a_1^1} \cdot \frac{\partial a_1^1}{\partial z_1^1} \cdot \frac{\partial z_1^1}{\partial w_{11}^1} = (a_1^2 - y_1)\sigma'(z_1^2)w_{11}^2\sigma'(z_1^1)x_1 \tag{1.7}$$

This is all still rather messy, but if you look at the gradients for multiple weights a few patterns emerge. The term $\frac{\partial J}{\partial z_j^l}$ appears quite often an appears for multiple different weights. This is often called the error for the j^{th} neuron in the l^{th} layer and is denoted by.

$$\frac{\partial J}{\partial z_j^l} = \delta_j^l$$

For the first neuron in the output layer we can see this is given by

$$\begin{aligned}\delta_1^2 = \frac{\partial J}{\partial z_1^2} &= \frac{\partial J}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} &(1.8)\\ &= \frac{\partial J}{\partial a_1^2}\sigma'(z_1^2) &(1.9)\end{aligned}$$

Where, as mentioned before, $\frac{\partial J}{\partial a_1^2}$ is easy to compute at the output layer, however it depends on the chosen objective function. More generally for any output neuron:

$$\boldsymbol{\delta}^L = \nabla_a J \odot \sigma'(\boldsymbol{z}^L) \tag{1.10}$$

where

$$\nabla_a J = \begin{bmatrix} \frac{\partial J}{\partial a_1^L} \\ \vdots \\ \frac{\partial J}{\partial a_k^L} \end{bmatrix}$$

The $\odot$ operator is the Hadamard product and indicates element-wise multiplication.

For the first neuron in the hidden layer we need to realise the signal travels along two paths going forwards which results in two components for $\frac{\partial J}{\partial z_1^1}$.

$$\begin{aligned}\delta_1^1 = \frac{\partial J}{\partial z_1^1} &= \frac{\partial J}{\partial a_1^2} \cdot \frac{\partial a_1^2}{\partial z_1^2} \cdot \frac{\partial z_1^2}{\partial a_1^1} \cdot \frac{\partial a_1^1}{\partial z_1^1} + \frac{\partial J}{\partial a_2^2} \cdot \frac{\partial a_2^2}{\partial z_2^2} \cdot \frac{\partial z_2^2}{\partial a_1^1} \cdot \frac{\partial a_1^1}{\partial z_1^1} &(1.11)\\ &= \delta_1^2 w_{11}^2 \sigma'(z_1^1) + \delta_2^2 w_{12}^2 \sigma'(z_1^1) &(1.12)\end{aligned}$$

This can be written more generally for all neurons in a given layer, l, and in matrix form as

$$\boldsymbol{\delta}^l = ((W^{l+1})^T \boldsymbol{\delta}^{l+1}) \odot \sigma'(\boldsymbol{z}^l) \tag{1.13}$$

The important thing to notice here is that the error for a given neuron only depends on the errors from neurons in the next layer. As we calculate errors for one layer we can then determine the errors in the previous layers.

Lastly we need to relate these error terms to parameter gradients so that we can perform parameter updates. Looking at equation 1.1 again you might have noticed that $\frac{\partial z_1^1}{\partial w_{11}^1}$ is common to both terms and evaluates to a_1^0 or more generally a_k^{l-1}. You should also notice that equation 1.1 is therefore $\delta_1^1 a_1^0$ or more generally the gradient for any weight is given by:

$$\frac{\partial J}{\partial w_{jk}^l} = a_k^{l-1}\delta_j^l \tag{1.14}$$

We know the activations from the forward pass and we can determine the errors in each neuron with a backward pass. Therefore we can compute any gradient in the neural network.

For the biases you can think of them as weights with connected to an activation of 1 (see Figure 1.3) In which case you can use equation 1.14 and have an activation of 1, resulting in:

$$\frac{\partial J}{\partial b_j^l} = \delta_j^l \tag{1.15}$$

1.4 Parameter updates

Lastly once we know the gradients for our model parameters we can update them in an attempt to optimize the loss function. We can use the same gradient descent techniques that we used in linear regression, except the loss functions are no longer convex (we may run into local minima, resulting in suboptimal results).

$$W^l := W^l - \eta\frac{\partial J}{\partial W^l} \tag{1.16}$$

$$\boldsymbol{b}^l := \boldsymbol{b}^l - \eta\frac{\partial J}{\partial \boldsymbol{b}^l} \tag{1.17}$$

Where η is the learning weight. As before we should iterate over multiple examples to gradually move down the loss curve towards some minima.

1.5 Worked Example

See the slides and videos for a worked example.2

Bibliography

[1] Wikimedia Commons. *Neuron Hand-tuned*. 2009. URL: `https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg` (cited on page 1).

[2] Michael Nielsen. *Neural Networks and Deep Learning*. Determination Press, 2015 (cited on page 2).

[3] Warren S McCulloch and Walter Pitts. ‘A logical calculus of the ideas immanent in nervous activity’. In: *The bulletin of mathematical biophysics* 5.4 (1943), pp. 115–133 (cited on page 2).

[4] Frank Rosenblatt. ‘The perceptron: a probabilistic model for information storage and organization in the brain.’ In: *Psychological review* 65.6 (1958), p. 386 (cited on page 2).

[5] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. ‘Learning representations by back-propagating errors’. In: *nature* 323.6088 (1986), pp. 533–536 (cited on page 2).

[6] Bernard Widrow. *An Adaptive ‘Adaline’ Neuron Using Chemical ‘Memistors’*. Tech. rep. 1553-2. Standford Electronics Laboratories, Stanford University, Oct. 1960 (cited on page 3).

[7] Yann A LeCun et al. ‘Efficient backprop’. In: *Neural networks: Tricks of the trade*. Springer, 2012, pp. 9–48 (cited on page 3).