Chapter 1

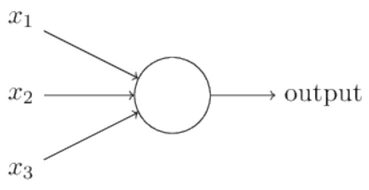
Recognizing Digits

Consider the problem of recognising handwritten digits like the ones below:

Our brain easily recognises the digits but attempting to write a program that does the same thing is really difficult as if we try to make rules we quickly get lost in a morass of exceptions and caveats and special cases.
Neural networks approach the problem in a different way, as long as we provide many examples with the desired outcome, the network is able to infer some rules that properly classify an unseen input. Now we will explore more in depth the core elements of neural networks: the neurons.

1.1 Neurons

Neurons are the basic element of a neural network, they receive one or more binary inputs x_i and output a value between 0 and 1. Each input has a weight w_i that represents the importance of the respective input to the output.



1.1.1 Perceptron

The perceptron developed in the 60s by Rosenblatt was the first kind of neuron. The neuron's output is determined by whether the weighted sum $\sum_i w_i x_i$ is less than or greater than some threshold value b. If we rewrite the weighted sum as the scalar product of the input vector $\mathbf{x}$ and weight vector $\mathbf{w}$ we have:

$$\text{output} = \begin{cases} 0 & \text{if } \mathbf{w}\cdot\mathbf{x} \leq \text{ threshold} \\ 1 & \text{if } \mathbf{w}\cdot\mathbf{x} > \text{ threshold} \end{cases} \tag{1.1}$$

If we move the threshold to the other side of the inequality, and replace it by perceptron's bias $b =$ -threshold we can rewrite the perceptron rule as:

$$\text{output} = \begin{cases} 0 & \text{if } \mathbf{w}\cdot\mathbf{x} + b \leq 0 \\ 1 & \text{if } \mathbf{w}\cdot\mathbf{x} + b > 0 \end{cases} \tag{1.2}$$

To fully understand what this equation is saying, let's make an example, suppose we want to evaluate whether you should buy a certain model of a car (1) or not (0) and suppose you select several attributes and give them a weight depending on how much do you value each attribute: whether it is electric or not (weight 3), whether if it is a convertible or not (weight 5), whether it is less than 40K (weight 5) and whether if it is available in a matte white color (weight 3). If we set the threshold at 9 we see that we should buy the car if it convertible and less than 40K but not if it is only electric and available in a matte white color.

From the previous example we learned that the perceptron is a good decision-making model but changing the weights or the threshold can significantly change the outcome. For instance, lowering the threshold makes the decision more inclined to 1 while increasing the threshold makes the decision more inclined to 0. So we can think the bias ($b =$ threshold) as as a measure of how easy it is to get the perceptron to output a 1.

If we have a two input perceptron with weights -2 for both inputs and 3 as a threshold we replicate the behaviour of a `NAND` gate which has been shown to be functional complete, in other words, any boolean function can be implemented by using a combination of NAND gates. As a consequence, perceptrons are also universal for computation, in other words, they can be as powerful as any other computer.

1.1.2 Sigmoid Neuron

In the previous example we arbitrarily defined some weight and thresholds. What we want to do in a neural network is to give many examples to the network so the network itself can establish which are the weights and biases needed for a correct decision making. What we'd like is that any small change in a weight or bias causes only a small change in the output from the network. It is worth noticing that in perceptron neurons is doesn't matter how much you're above the threshold, if you're above it, the output is 1. What we would like is something that expresses the intuitive statement: the more we are above the threshold the more we are sure of the output. Some might

think a linear function relationship with the output would be a good idea, but unfortunately the output would grow linearly and we normally want to keep things between 0 and 1.
The sigmoid neuron is like the perceptron neuron but applies a sigmoid function to the weighted sum plus bias in order to squish the values in the $[0, 1)$ range and allow comparisons between different weight and biases as any small change in one of them would result in a change in the output. The output of the sigmoid neuron is defined as:

$$\text{output} \quad = \sigma(\mathbf{w} \cdot \mathbf{x} + b) \tag{1.3}$$

where:

$$\sigma(z) = \frac{1}{1 + e^{-z}} \tag{1.4}$$

we notice that when $z \to +\infty$ we have $\sigma(z) = 1$ and when $z = 0$ we have $\sigma(z) = 0$.

1.1.3 Generic neuron

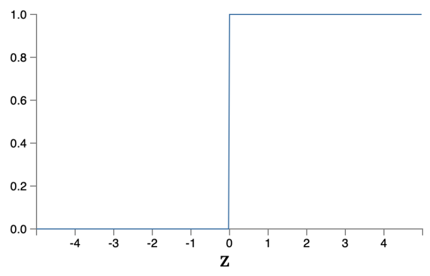
A generic neuron is a neuron where the output is defined as:

$$\text{output} \quad = f(\mathbf{w} \cdot \mathbf{x} + b) \tag{1.5}$$

where f is called the *activation function*. The crucial fact about the σ function is not its definition

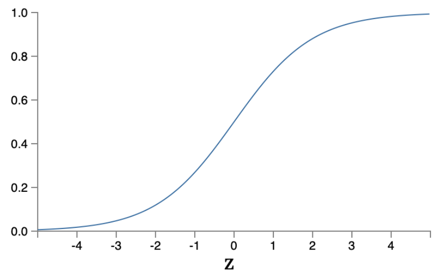
Perceptron Neuron

$f =$ step function (0)



Sigmoid Neuron

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



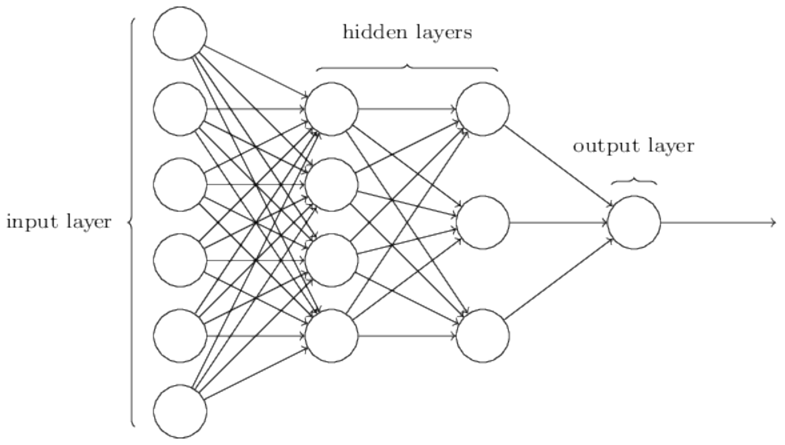
but the smoothness it has, this means that small changes of the weights Δw_i and in the biases Δb_j will produce a small change in the output Δoutput. Calculus tells us that Δoutput is a linear function of the changes Δw_i and Δb_j.

$$\Delta\text{output} \approx \sum_i \frac{\partial\,\text{output}}{\partial w_i} \Delta w_i + \frac{\partial\,\text{output}}{\partial b} \Delta b \tag{1.6}$$

1.2 Networks

We saw the basic unit of neural networks, the neurons. Now we will see how neurons are organized in order to perform complex tasks such as digit recognition.

Neurons are organized in three different kind of layers: input layer, hidden layers and output layer. The *input layer* is usually represented as the leftmost layer in the graphical representation of a neural network and represents the information we have to solve the problem, in case of our digit recognition problem, all the pixel intensities of the image. The *output layer* is usually represented as the rightmost layer in the graphical representation of a neural network and represents the output given by the network based on the inputs, it can be a single neuron (e.g. yes or no) or several neurons (e.g. 10 numbers). The layers that are not input or output are called *hidden layers* and are neurons that perform some kind of decision making, it can be a single layer or multiple layers. Networks with multiple layers of sigmoid neurons are sometimes called **multilayer perceptrons** or **MLPs**, despite being made up of sigmoid neurons, not perceptrons.



RM In a network we have as many biases as the number of neurons in the output layer, but a number of weights that depends on the topology of the network, i.e. how many hidden layers and how many neurons there are in each hidden layer and in the input layer.

Networks where information is always fed forward and never fed back are called **feedforward neural networks**. This means there are no loops in the network. If we had loops, there would be situations where the input given to the *sigma* function depended on the output. There are however other models of artificial neural networks in which feedback loops are possible. These models are called **recurrent neural networks** and in these models loops don't cause problems since a neuron's output only affects its input at some later time and not instantaneously.

In the digit recognition example if we have 32 x 32 grayscale images we have 1024 inputs in the range [0,1] expressing 0-white and 1-black. The network should have 10 output neurons, one for each number and the number predicted by the network is the one with highest value.

1.3 Learning

As we said before neural networks learn through examples, what we'd like now is an algorithm that finds weights and biases in a way that so the output from the network approximates the real result. In the case of digit recognition we want that based on a 32 x 32 image we approximate the correct 10 dimensional output vector $\mathbf{y}$ that contains a 1 in the position of the digit it represents (e.g. $\mathbf{y}(\mathbf{x}) = (0, 0, 1, 0, 0, 0, 1, 0, 0, 0)^T$ represents number 2).

To quantify how well we're approximating the the correct output vector, we need to define a cost function. We now define the most used cost function, the quadratic cost function also known as **mean squared error or MSE** :

$$C(\mathbf{w}, \mathbf{b}) = \frac{1}{2n} \sum_x \|\mathbf{y}(x) - \text{output}(x)\|^2. \tag{1.7}$$

If we inspect this definition we see that if our training algorithm does a good job approximating the output vector it can find weights and a bias so that $C(\mathbf{w}, \mathbf{b}) \simeq 0$.

1.3.1 Stochastic Gradient Descent

What we want to find a set of weights and biases which make the cost as small as possible, to do that we use an algorithm called stochastic gradient descent or **SGD** for short.
The gradient descent algorithm is based on a simple idea, the gradient of a function indicates the direction of its steepest ascent, if we take some steps in the opposite direction, we should be going to the minimum in the fastest way possible. This algorithm is very useful as the analytical computation of the minimum results unfeasible with functions of many variables.

Calculus tells us that:

$$\Delta C \approx \frac{\partial C}{\partial w_1} \Delta w_1 + \ldots \frac{\partial C}{\partial w_n} \Delta w_n + \frac{\partial C}{\partial b_1} \Delta b_1 + \ldots + \frac{\partial C}{\partial b_m} \Delta b_m \tag{1.8}$$

being the gradient vector ∇C defined as:

$$\nabla C = \left(\frac{\partial C}{\partial w_1}, \ldots, \frac{\partial C}{\partial w_n}, \frac{\partial C}{\partial b_1}, \ldots, \frac{\partial C}{\partial b_m} \right) \tag{1.9}$$

we can write that how much the cost function changes can be approximated based on the gradient of each of the parameters $\mathbf{p} = (\mathbf{w}, \mathbf{b})$ (weights and biases) and how much we vary them:

$$\Delta C \approx \nabla C \cdot \Delta \mathbf{p} \tag{1.10}$$

as we said we want to take some small steps in direction towards the minimum so we define:

$$\Delta \mathbf{p} = -\eta \nabla C \tag{1.11}$$

the parameter η is a small, positive parameter called **learning rate** and can be constant or vary according to how much we've learned.

If we replace $\Delta\mathbf{p}$ in equation 1.10 we obtain:

$$\Delta C \approx -\eta \nabla C \cdot \nabla C = -\eta \|\nabla C\|^2 \tag{1.12}$$

this equations makes evident an important property, given the fact that η is positive and $\|\nabla C\|^2 \geq 0$ this guarantees that $\Delta C \leq 0$, in other words if we change the parameters like in the equation 1.11 the cost function will always decrease, never increase.

To compute the new value of the parameters we simply do what is called the **gradient descent update rule** :

$$\Delta\mathbf{p} = -\eta \nabla C \rightarrow p' = p - \eta \nabla C \tag{1.13}$$

The rule doesn't always work as gradient descent might not find the global minimum of C but a local one. Despite this, gradient descent performs extremely well in practice, however when the number of training inputs is very large, computing the gradient can take a long time as it is the average of all the gradients separately for each training input.

Stochastic gradient descent solves this problem by estimating the gradient ∇C by computing ∇C_x for a small sample of k randomly chosen training inputs called **mini-batch**. The average over the small sample can quickly get a good estimate of the true gradient without the need of going through all inputs.

$$\nabla C = \frac{1}{n} \sum_x \nabla C_x \approx \frac{1}{k} \sum_{j=1}^{k} \nabla C_{X_j} \tag{1.14}$$

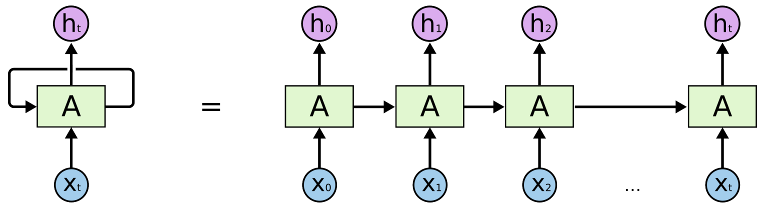
Then stochastic gradient descent works by picking out a randomly chosen mini-batch of training inputs, and train with those, then we pick out another randomly chosen mini-batch and train with those too and so on until we've exhausted the training inputs, when that happens, it is said to complete an **epoch** of training. At that point we start over with a new training epoch.
Stochastic gradient descents works just like election polls, by using a small sample we can approximate the behaviour of the whole population (the cost function over all the inputs).

Chapter 2

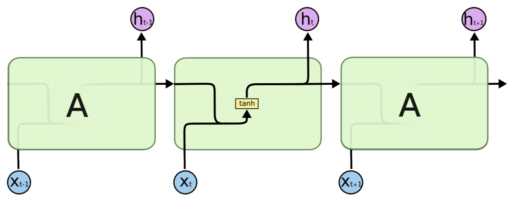
Recurrent Neural Networks

As we said before, recurrent neural networks are just neural networks with loops in them. The loops in its architecture helps them to persist information, in other words, instead of producing a guess at each time instant from scratch, they can use past information to produce a better guess. This persistence of information has made them really useful in applications such as speech recognition, language modelling, translation, text autocomplete and so on.

Having a loop may have complicated but it is actually not, this is due to the fact that the loop acts in separate time instants. Actually, a recurrent neural network can be thought of as having multiple copies of the same network, each passing its output as an input for the next network.



Normal RNNs have a simple blocks with a tanh activation function, however these kind of networks have a problem, even though they are able to persist information, when the gap between the relevant information for prediction needed is big, RNNs cannot learn to use the past information.

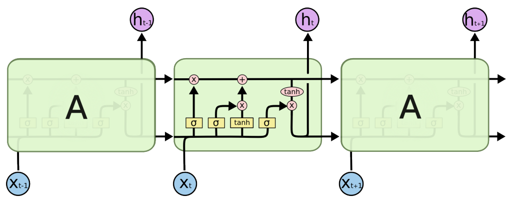


In other words, RNNs are capable of handling dependencies as long as the gap doesn't become too long. This problem is called the *short memory problem* or *long-term dependency problem.*

2.1 LSTM

There are some applications where long term dependencies need to be managed and standard RNN were not able to do so. It wasn't until *Hochreiter & Schmidhuber* (1997) developed LSTMs (Long Short Term Memory networks) explicitly designed to avoid the long-term dependency problem that the problem was finally solved.

This kind of network works very similarly to standard RNN, the difference is that instead of using a simple module using a tanh activation function, each module has a relatively complicated structure with four interacting layers specially crafted to handle the long-term dependency problem.



The LSTM module has the ability to remove or add information to the cell state, or let information through. This decisions are taken by structures called gates.
The decision to let information through or not is made by a sigmoid layer f_t called the **forget gate layer**. A value of 1 means completely keep this while a 0 represents completely get rid of this. To calculate its value it looks at h_{t1} and x_t and outputs a number in the interval $[0, 1]$.

$$f_t = \sigma(W_f \dot{[} h_{t-1}, x_t] + b_f) \tag{2.1}$$

We have other two layers, a sigmoid layer i_t called the **input gate layer** that decides which values well update, and a tanh layer $\tilde{C}_t$ that creates a vector of new candidate values that could be added to the state.

$$i_t = \sigma\left(W_i \cdot [h_{t-1}, x_t] + b_i\right) \tag{2.2}$$

$$\tilde{C}_t = \tanh\left(W_C \cdot [h_{t-1}, x_t] + b_C\right) \tag{2.3}$$

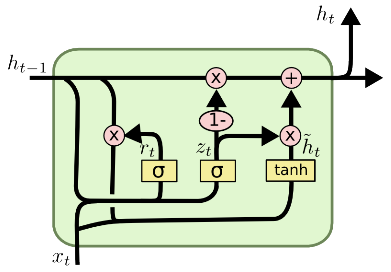
Finally we have an **output gate layer** that based on our cell state outputs a result, this result is filtered.

$$o_t = \sigma\left(W_o\left[h_{t-1}, x_t\right] + b_o\right) \tag{2.4}$$

$$h_t = o_t * \tanh\left(C_t\right) \tag{2.5}$$

2.2 GRU

The Gated Recurrent Unit, or GRU is a modified version of the LSTM introduced by Cho, et al. (2014). Between the modifications, it combines the forget and input gates into a single update gate, it also merges the cell state and hidden state, and makes some other changes.
This model has been growing in popularity thanks to its good performance and simplicity being simpler than standard LSTMs.



$$z_t = \sigma \left(W_z \cdot [h_{t-1}, x_t] \right)$$
$$r_t = \sigma \left(W_r \cdot [h_{t-1}, x_t] \right)$$
$$\tilde{h}_t = \tanh \left(W \cdot [r_t * h_{t-1}, x_t] \right)$$
$$h_t = (1 - z_t) * h_{t-1} + z_t * \tilde{h}_t$$