# Human and Artificial Intelligence

Stefano Anzellotti

# Chapter 1

# Introduction

The progress of technology has given the human specie the ability to transform the world at an unprecedented scale. Here in Boston, for example, the "Big Dig" project created 7.8 miles of underground tunnels, using 3.8 million cubic yards of concrete (`https://www.mass.gov/info-details/the-big-dig-project-background`). Human activity has led to transformation at an even larger scale. Many cities in the world expand at a very fast pace (Dubai had a population of under 500,000 in 1990, and of about 3 million today). The widespread use of plastic has led to the formation of a patch of trash in the Pacific Ocean that is twice the size of Texas, with an estimated 1.8 trillion pieces of plastic (`https://theoceancleanup.com/great-pacific-garbage-patch/`). Climate change [Karl and Trenberth, 2003] is increasingly affecting the lives of millions, including here in the USA: a heatwave on the west cost led to temperatures over 100F in Seattle and Portland. Millions of hectares of land burned in fires in California this year, and research suggests that man-made climate change is likely one of the causes [Herring et al., 2015].

Our understanding of the human mind and of society has not kept up with the pace of technological advances. As a consequence, decisions that can have enormous impact - either on society or on our own personal lives - are made with limited knowledge of human decision-making processes and of their shortcomings. Education is likely less efficient than it could be if we had better insight into human learning mechanisms. Social conflict is exacerbated by sterotypes and discrimination, and understanding better the mechanisms through which they arise and are maintained could help to mitigate them.

If we want to use our understanding of the human mind and brain to guide decisions, improve education, and reduce discrimination, we need an understanding that is precise and quantitative, so that we can estimate the consequences of interventions, and calculate the likelihood of different possible outcomes. The aim of this course is to establish the foundations to work towards this kind of quantitative understanding of the mind and brain. There are several approaches to construct quantitative models of the mind and brain, and there will not be time to cover all of them in this course. We will focus on artificial neural networks, because of their potential and flexibility.

**A note on prerequisites.** This course is designed to be accessible to students who do not yet have much background in Mathematics and Computer Science. However, some prior knowledge is expected, such as limits and derivatives, and basics of Python programming. If you are not familiar with limits and derivatives, or if you have never programmed with Python before, you can still take this course, but you will need to learn those foundations - they will not be covered in the course.

# Chapter 2

# History

## 2.1 From artificial neurons to multilayer convolutional networks (1943-1986)

In 1943, McCullough and Pitts published a landmark paper proposing a mathematical model of a neuron [McCulloch and Pitts, 1943]. The second world war was just turning in favor of the Allies. Less than ten years earlier (in 1936), Alan Turing had published his landmark article on computable numbers, in which he envisioned universal computing machines ("Turing machines"). Computers as we know them did not exist: the von Neumann architecture on which they are based would only be introduced in 1945.

In their paper, McCullough and Pitts proposed to model the output of a neuron by taking the neuron's inputs (a vector of numbers), multiplying them by some weights (one number for each of the inputs), summing them, and then applying a nonlinearity. This model is no longer considered an accurate description of biological neurons, but its simplicity makes it possible to build large networks of McCullough and Pitts neurons, and they are still the basis for artificial neural networks used today.

Little more than a decade after McCullough and Pitts' published their paper, neurophysiology research by David Hubel and Thornsten Wiesel led to new insights into the cat visual system that would inspire and transform the subsequent history of artificial intelligence [Hubel and Wiesel, 1959]. Hubel and Wiesel Two key principles of modern neural network architectures derived from the insights obtained in Hubel and Wiesel's work. First, that neurons would be organized in successive processing stages ('layers'). Second that within each layers neurons would receive inputs from only a small neighborhood of the neurons in the previous layer. These two ideas became the foundation for the later development of deep convolutional neural networks.

In the early 1980s, Fukushima developed artificial neural networks built by combining McCullough and Pitts neurons into an architecture inspired by Hubel and Wiesel's work (the Neocognitron, [Fukushima et al., 1983]). The Neocognitron was trained one layer at a time, using a semi-manual procedure in which artificial neurons were trained to respond selectively to specific patterns chosen by the researcher. Training artificial neural networks with many layers was challenging. An important breakthrough occurred in 1986, when David Rumelhart and Jay McClelland developed "backpropagation": an algorithm that could be used to train networks with many layers efficiently. Backpropagation is still at the core of how artificial neural networks are trained today.

By the end of the 1980s, all key theoretical ingredients for artificial neural networks had been developed. However, artificial neural networks did not become widely popular and broadly applied until the mid 2010s. What was the reason for this 30 years gap? To realize the full potential of artificial neural networks, the networks need to be large, and they needed to be trained with many inputs. The relatively small size of the networks used at the time and the lack of large datasets for training severely limited their potential.

## 2.2 Unlocking the potential of deep networks with big data and parallel computing (2007-2012)

Two key ingredients changed the status quo and helped realize the potential of artificial neural networks. First, progress in the engineering of graphic cards for computers led to the development of cards with many compute cores, that could handle efficiently parallel computations. In 2007, new software was introduced to use these graphic cards for purposes other than displaying graphics on a screen (graphic cards with these capabilities are now called General Purpose Graphic Processing Units - GPGPU). Importantly, artificial neural networks can be trained much faster using GPGPUs than using traditional computers, this made it possible to train very large networks.

Second, the diffusion of digital cameras and larger storage devices in the early 2000s, along with the growth of the world wide web, made it possible to create much larger datasets for training artificial neural networks for object recognition. In 2009, the ImageNet database was released, with over 14 million images belonging to over 21 thousand object classes. Together, these advances contributed to the development of the first deep neural network, "AlexNet", in 2012. AlexNet was trained to label objects in pictures, a landmark task in machine learning, researchers had been working on this problem since the 1960s. AlexNet achieved an impressive improvement over previous methods, and within a couple of years since AlexNet's introduction artificial neural networks achieved accuracy comparable to humans for object recognition on the ImageNet dataset.

The remarkable performance of AlexNet was important because it had led to an impressive improvement on a longstanding problem in machine learning, but it was even more important because it revived research on artificial neural networks more broadly. Inspired by this success, new databases were created for a variety of different tasks, from speech recognition to diagnosis from medical images. After over two decades in which artificial neural networks were neglected, large numbers of computer scientists returned to this area of research, leading to fast paced progress in the past decade.

## 2.3 2013 to today: adversarial networks, probabilistic methods, structured representations

Recent research on artificial neural networks has led to the development of a multitude of different types of networks. The field moves so quickly that it is impossible to offer a comprehensive account. Some neural networks have multiple parts that process different kinds of information, that are merged at later stages [Simonyan and Zisserman, 2014]. Other networks contain different components that compete with each other to keep improving ("adversarial" networks, [Goodfellow et al., 2014]). Other networks integrate probabilistic components [Guan et al., 2020], and others use structured representations encoding different objects as nodes of graphs [Li et al., 2017]. While it was impressive at the time, AlexNet is far too simplistic to capture more complex human cognitive abilities. However, the more recent advances in artificial neural networks are promising, and might help us to better understand other aspects of human cognition beyond perception.

# Chapter 3

# Basic architectures

## 3.1 Simple networks

Let's consider a problem in which we have a low-dimensional input, and we want to use it to predict a single output. Let's take as inputs three dimensional real-valued vectors, and as outputs real numbers. Suppose that we have $n$ datapoints for training. Our training dataset will look like this:

$$\left( \begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \end{bmatrix}, y_1 \right), \ldots, \left( \begin{bmatrix} x_{n1} \\ x_{n2} \\ x_{n3} \end{bmatrix}, y_n \right).$$

We can try to predict the output values $y_i$ using a simple neural network. To distinguish the true values of $y_i$ from the predictions, we will call the network's predictions $\hat{y}_i$. We can draw our network as in Figure 3.1.

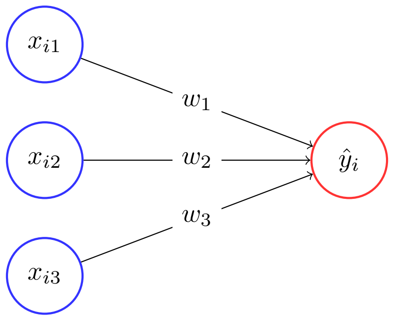


Figure 3.1: Simple network.

where $w_1, w_2, w_3$ are the network's 'weights' or 'parameters'. The network's output is calculated as

$$\hat{y}_i = w_1 \times x_{i1} + w_2 \times x_{i2} + w_3 \times x_{i3}.$$

In the future, for convenience, we will omit the multiplication sign and write

$$\hat{y}_i = w_1 x_{i1} + w_2 x_{i2} + w_3 x_{i3}.$$

Networks like this one, in which the output is a weighted sum of the inputs, are called 'linear'. You can see that the graph of the simplest, one-dimensional case $\hat{y}_i = w_1 x_{i1}$ is a straight line.

**Exercise.** Prove that the network introduced in this section satisfies the two properties of linear functions.

## 3.2 Larger networks

Now, suppose that we wanted to use our inputs to predict more than one output dimension. For example, we might want to predict a person's height and weight based on their age, their sex, and their hair color. Our training data will look like this:

$$\left(\begin{bmatrix} x_{11} \\ x_{12} \\ x_{13} \end{bmatrix}, \begin{bmatrix} y_{11} \\ y_{12} \end{bmatrix}\right), \ldots, \left(\begin{bmatrix} x_{n1} \\ x_{n2} \\ x_{n3} \end{bmatrix}, \begin{bmatrix} y_{n1} \\ y_{n2} \end{bmatrix}\right).$$

And we can draw our network as in as in Figure 3.2.

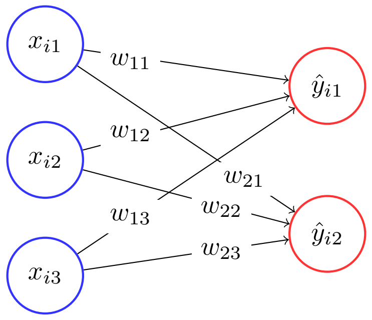


Figure 3.2: Network with two outputs

The predicted outputs are calculated in this way:

$$\begin{aligned} \hat{y}_{i1} &= w_{11}x_{i1} + w_{12}x_{i2} + w_{13}x_{i3}, \\ \hat{y}_{i2} &= w_{21}x_{i1} + w_{22}x_{i2} + w_{23}x_{i3}. \end{aligned}$$

As the networks get larger, it becomes tiresome to write them in this way. We will now introduce a more compact way to write networks, using vectors and matrices.

**A bit of Linear Algebra**

**Linear functions.** We say that a function $f : \mathbb{R}^n \to \mathbb{R}^m$ is 'linear' if and only if it has these two properties:

1. $f(x + y) = f(x) + f(y) \quad \forall x, y \in \mathbb{R}^n$

2. $f(\lambda x) = \lambda f(x) \quad \forall x \in \mathbb{R}^n, \forall \lambda \in \mathbb{R}$

Note that $\forall$ means 'for all' and $\in$ means 'in'.

**Vectors.** For this class, we will think of vectors as list of numbers, arranged vertically by convention. You can also think of vectors as point in a space that has one dimension for each number in the list. The numbers in the list are the coordinates of the point along the different dimensions.

**Dot product.** If we have two vectors of the same size, we can calculate their dot product. The dot product is calculated by computing the product between the numbers in the first dimension, then adding the product between the numbers in the second dimension, and so on. For example:

$$\begin{bmatrix} a \\ b \end{bmatrix} \cdot \begin{bmatrix} c \\ d \end{bmatrix} = ac + bd.$$

**Matrices.** A 'matrix' is a rectangular arrangement of numbers. For example:

$$\begin{bmatrix} 3 & -1 & 2 \\ 0.35 & 100 & -7 \end{bmatrix}$$

is a 2 by 3 matrix. By convention, when we report the size of a matrix, the first number denotes the number of rows, and the second number denotes the number of columns.

**Product between a matrix and a vector.** We can describe our network as a product between a matrix and a vector. The number of rows of the matrix must be equal to the size of the output. The number of columns of the matrix must be equal to the size of the input. Consider this matrix of weights:

$$\begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix}$$

and this input:

$$\begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{bmatrix}.$$

The network of the example in this section can be written as

$$\begin{bmatrix} \hat{y}_{i1} \\ \hat{y}_{i2} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{bmatrix}$$

where each output is calculated as the dot product between its corresponding row of the matrix and the input.

**Exercise.** Consider the weight matrix for a linear network with three inputs and two outputs:

$$W = \begin{bmatrix} 1 & 3 & -2 \\ 2 & -1 & 1 \end{bmatrix}$$

and the input vector

$$\mathbf{x} = \begin{bmatrix} 2 \\ 1 \\ 1 \end{bmatrix}.$$

Calculate the output

$$\hat{\mathbf{y}} = W\mathbf{x}.$$

## 3.3 Multilayer networks

So far, we have talked about networks that map the inputs directly onto the outputs. In this section, we introduce 'deeper' networks, with a 'hidden' layer. In Figure **??** you can find an example of a network with a hidden layer:

We can use matrices to write multilayer networks with more compact notation. Let's consider the matrices

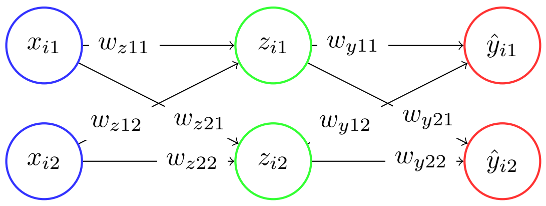


Figure 3.3: Network with one hidden layer

$$W_z = \begin{bmatrix} w_{z11} & w_{z12} \\ w_{z21} & w_{z22} \end{bmatrix}$$
$$W_y = \begin{bmatrix} w_{y11} & w_{y12} \\ w_{y21} & w_{y22} \end{bmatrix}.$$

Let's write the input as

$$\mathbf{x} = \begin{bmatrix} x_{i1} \\ x_{i2} \end{bmatrix}.$$

We can write the network with one hidden layer as

$$\mathbf{z} = W_z \mathbf{x}$$
$$\hat{\mathbf{y}} = W_y \mathbf{z}$$

or in one line

$$\hat{\mathbf{y}} = W_y(W_z \mathbf{x}).$$

Suppose that we wanted to create a network with many hidden layers. The input $\mathbf{x}$ would be mapped to responses in the first hidden layer $\mathbf{z}_1$ using weights $W_{z_1}$, then responses in the first hidden layer would be mapped to responses in the second hidden layer $\mathbf{z}_2$ using weights $W_{z_2}$ and so on, until finally the responses in the last hidden layer $\mathbf{z}_n$ would be mapped to the predictions $\hat{\mathbf{y}}$ using weights $W_y$. We can write this as

$$\hat{\mathbf{y}} = W_y(W_{z_n}(\cdots(W_{z_2}(W_{z_1}\mathbf{x}))\cdots)).$$

**Exercise.** Consider the weight matrices for a linear network with one hidden layer:

$$W_z = \begin{bmatrix} -1 & 2 \\ 2 & 3 \end{bmatrix}, \quad W_y = \begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix}$$

and the input vector

$$\mathbf{x} = \begin{bmatrix} 1 \\ -1 \end{bmatrix}.$$

Calculate the output

$$\hat{\mathbf{y}} = W_y(W_z \mathbf{x}).$$

## 3.4 Adding a constant

You can think about linear neural networks as a generalization of simple linear regression $\hat{y} = ax$. Sometimes, when the value of the input is 0, the correct value of the output is different from 0. However, when the input is 0, the model $\hat{y} = ax$ always predicts that the output will be 0, regardless of the value of $a$. To enable our model to produce non-zero outputs even when the input is 0, we can add a constant $b$. Our model then becomes $\hat{y} = ax + b$. We can do the same thing for neural networks, adding a constant to each output dimension. For example:

$$\begin{bmatrix} \hat{y}_{i1} \\ \hat{y}_{i2} \end{bmatrix} = \begin{bmatrix} w_{11} & w_{12} & w_{13} \\ w_{21} & w_{22} & w_{23} \end{bmatrix} \begin{bmatrix} x_{i1} \\ x_{i2} \\ x_{i3} \end{bmatrix} + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix}$$

which we can also write as

$$\hat{\mathbf{y}} = W\mathbf{x} + \mathbf{b}.$$

In the field of neural networks, this constant is often called the 'bias'.

## 3.5 Nonlinear networks

Sometimes, the input-output relationships we want to model are not well captured by linear networks. For example, consider the relationship between the tension of a string and the pitch of the sound it produces (you might be familiar with this if you play the guitar). The pitch (or 'frequency') $f$ is proportional to the square root of the string tension $T$ [1].

If we want to model nonlinear relationships, linear networks are not enough. However, if we use many nodes that each use very simple nonlinear functions (also called 'activation functions'), we can approximate well very complex nonlinear functions. This result is known as the 'universal approximator theorem' (Cybenko 1989, Hornik 1991).

**A bit of Topology**

**Bounded sets.** We say that a set is 'bounded' if there is a finite distance $d$ such that all points in the set are closer than $d$.

**Closed sets.** We say that a set is 'closed' if, for every sequence of points in the set that get closer and closer (and arbitrarily close) to another point, the point they get closer and closer to is also in the set.

**Compact sets.** We say that a set is 'compact' if it is bounded and closed.

**Continuous functions.** We say that a function $f$ is 'continuous' if: for each input $x$, and for each sequence of inputs $x_1, x_2, \ldots$ that get closer and closer to $x$, the values $f(x_1), f(x_2), \ldots$ get closer and closer to $f(x)$.

**Universal approximation theorem.** A feed-forward network with a single hidden layer containing a finite number of nodes with sigmoid activation functions can approximate arbitrary well real-valued continuous functions on compact subsets of $\mathbb{R}^n$.

Note: more accurate definitions of closed sets and continuous functions can be given using the notion of limit, which we do not discuss in this course. An extension of the theorem to ReLU activation functions and multilayer networks has been proven by Lu and colleagues in 2017 [Lu et al., 2017]

[1]This was noted by Vincenzo Galilei, and it is one of the earliest known mathematical descriptions of a nonlinear phenomenon.

A classic type of nonlinearity is the sigmoid $\phi(x) = \tanh(x)$. A more popular type of nonlinearity used in today's artificial neural network is the 'Rectified Linear Unit' (ReLU). ReLU transforms negative numbers into zero, and keeps positive numbers the same. Moving forward, we will use $\phi$ to refer to the ReLU activation function:

$$\phi(x) = \begin{cases} 0 \text{ if } x \leq 0, \\ x \text{ otherwise.} \end{cases}$$

When a layer has multiple outputs, sometimes we want to apply the activation function to all of the outputs. Consider as an example the output of a hidden layer

$$\mathbf{u} = \begin{bmatrix} u_1 \\ u_2 \end{bmatrix}.$$

We will use $\Phi$ to denote the function that applies $\phi$ to each output:

$$\Phi(\mathbf{u}) = \begin{bmatrix} \phi(u_1) \\ \phi(u_2) \end{bmatrix}.$$

This example uses a two dimensional output, but the same can be done for outputs of any dimension.

By convention, people think of a nonlinear layer as being composed of a linear transformation that maps a vector $\mathbf{x}$ to a vector $\mathbf{u} = W_z\mathbf{x}$), followed by a nonlinear activation function that maps $\mathbf{u}$ to $\mathbf{z} = \Phi(\mathbf{u})$. We can write a nonlinear layer in one line as

$$\mathbf{z} = \Phi(W_z\mathbf{x}).$$

If we have a nonlinear network with many hidden layers, we can write it as

$$\hat{\mathbf{y}} = W_y(\Phi(W_{z_n}(\cdots(\Phi(W_{z_2}(\Phi(W_{z_1}\mathbf{x})))) \cdots))).$$

## 3.6 Networks for classification

In some cases, we want to predict categorical outputs (a 'classification' problem). For example, we might want to determine whether a CT scan shows a medically significant finding (i.e. a tumor). For this problem, we would like a network that has two-dimensional outputs, such that the value for the first dimension is the probability that the scan is normal, and the value for the second dimension is the probability that the scan shows a tumor. For example, an output of $\begin{bmatrix} 0.8 \\ 0.2 \end{bmatrix}$ would indicate that the network assigns a probability 0.8 to the scan being normal, and a probability of 0.2 to the scan showing a tumor.

Probabilities need to be between 0 and 1, and the sum of the probability of all possible outcomes needs to be 1 (see the box on Probability). However, the neural networks we discussed so far can have outputs larger than one, and there is no guarantee that the outputs for the different dimensions will add up to 1. How do we get a network to produce outputs that are probabilities?

The trick is to transform the outputs of a standard network with a function called 'softmax'. Softmax transforms an output with 2 possible categories $\hat{\mathbf{y}} = \begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \end{bmatrix}$ into a new transformed vector

$$\text{softmax}(\hat{\mathbf{y}}) = \begin{bmatrix} \frac{e^{\hat{y}_1}}{e^{\hat{y}_1}+e^{\hat{y}_2}} \\ \frac{e^{\hat{y}_2}}{e^{\hat{y}_1}+e^{\hat{y}_2}} \end{bmatrix}.$$

Note that $e^x$ is positive for all $x \in \mathbb{R}$, so even if $\hat{y}_1$ or $\hat{y}_2$ are negative, the versions transformed by softmax are greater or equal than zero. In addition,

$$\frac{e^{\hat{y}_1}}{e^{\hat{y}_1} + e^{\hat{y}_2}} + \frac{e^{\hat{y}_2}}{e^{\hat{y}_1} + e^{\hat{y}_2}} = \frac{e^{\hat{y}_1} + e^{\hat{y}_2}}{e^{\hat{y}_1} + e^{\hat{y}_2}} = 1.$$

So, softmax transforms any real multidimensional vector into a vector that satisfies the properties of probability. The softmax can be generalized to outputs with more than 2 dimensions. If we are trying to classify inputs into many different possible categories (i.e. taking a picture of a dog and determining which of 50 different breeds it belongs to), we can transform each output $\hat{y}_i$ into

$$\text{softmax}(\hat{y}_i) = \frac{e^{\hat{y}_i}}{\sum_{j=1}^{n} e^{\hat{y}_j}},$$

where $n$ is the number of output dimensions. You might wonder why this transformation has this peculiar name, 'softmax'. I will leave it up to you to try to find the answer.

## Artificial networks vs the human brain

**Neurons.** Neurons in the human brain are very different from the neurons in artificial neural networks. Neurons in artificial neural networks usually all use the same nonlinearities (i.e. ReLU, sigmoid) or very similar nonlinearities (i.e. leaky ReLU). By contrast, neurons in the human brain receive their inputs on dendritic trees whose shape determines how different inputs are integrated. The morphology and branching of dendritic trees varies from neuron to neuron depending on their function (see [Ascoli et al., 2007]). The information processing that occurs within a human neuron is likely far more sophisticated than the information processing done by a typical artificial neuron.
**Connections.** In the simple neural networks we discussed so far, each layer is only connected to the following layer in the hierarchy. By contrast, the human brain has a large number of long-range connections between early layers and later layers that 'bypass' intermediate layers (see [Van Essen et al., 1992]). Artificial Intelligence researchers are developing artificial networks with increasingly complex connections.

# Chapter 4

# Loss functions

In the previous section, we learned about artificial neural networks with multiple outputs and with multiple layers. We learned about linear and nonlinear networks, and we learned to calculate their outputs if we are given the weights. However, if we want to predict some outputs given some inputs, how can we find out what the right weights should be? The first step is to be able to tell, given some weights, whether they are doing well or poorly at predicting the correct outputs. To do this, we use a 'loss function'.

## 4.1 Mean square error loss - unidimensional outputs

A very popular type of loss function is the mean square error (MSE) loss. This loss is used when we are trying to predict a variable that has continuous values, such as the future value of stocks, or the amount of activity in a brain region. Let's consider a set of datapoints: $(\mathbf{x_1}, y_1), \ldots, (\mathbf{x_n}, y_n)$, and the network predictions for these datapoints: $\hat{y}_1, \ldots, \hat{y}_n$. We start from the first datapoint, and we calculate the square of the error, that is, the square of the difference between the network prediction and the correct output: $(\hat{y}_1 - y_1)^2$. We calculate the square error for each datapoint, and then we take the mean:

$$\text{MSEloss} = \frac{(\hat{y}_1 - y_1)^2 + \cdots + (\hat{y}_n - y_n)^2}{n}.$$

For convenience, we can also write this as

$$\text{MSEloss} = \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)^2}{n}.$$

You might wonder why should we take the squares of the errors. Wouldn't it be simpler to just calculate the mean of the errors without the squares?

$$\text{aSimplerLoss} = \frac{\sum_{i=1}^{n} (\hat{y}_i - y_i)}{n}$$

It turns out that this would not work very well. Consider this example: imagine that a network's prediction was higher by 2 than the correct output for one datapoint, and lower by 2 than the correct output for another datapoint. If we used the simpler loss, these two errors would cancel out! The simpler loss would not be able to distinguish between this network and a network that got both datapoints exactly right.

You might still be curious about why we summed the squares of the errors, instead of just taking the absolute value:

$$\text{aSomewhatSimplerLoss} = \frac{\sum_{i=1}^{n} |\hat{y}_i - y_i|}{n}.$$

There is a reason for this as well, and it has to do with the fact that taking the squares makes the loss 'differentiable'. We will talk more about this in the section about how the networks actually learn (gradient descent).

**Exercise.** Consider the network $\hat{y} = W\mathbf{x}$, where

$$W = \begin{bmatrix} -1 & 2 & 1 \end{bmatrix}.$$

Given a training dataset

$$\left( \begin{bmatrix} 2 \\ 1 \\ 3 \end{bmatrix}, 2 \right), \left( \begin{bmatrix} 5 \\ 2 \\ 2 \end{bmatrix}, 0 \right).$$

calculate the mean square error loss.

## 4.2 Mean square error loss - multidimensional outputs

What if our network has multidimensional outputs? Let's start from an intuition about the MSE loss for one-dimensional outputs, and use it to generalize the idea to the case with multiple outputs. First, let's rewrite the MSE loss in this way:

$$\text{MSEloss} = \frac{\sum_{i=1}^{n} |\hat{y}_i - y_i|^2}{n}$$

double check that it is actually the same thing! Next, note that if we plot $\hat{y}_i$ and $y_i$ on a line, $|\hat{y}_i - y_i|$ is actually the distance between them. It makes sense, the loss will be larger if the prediction is 'far' from the correct value. We can use this idea to generalize the MSE loss to the case with multidimensional outputs. Given a correct output $\mathbf{y}_i$ and a predicted output $\hat{\mathbf{y}}_i$, we will replace $|\hat{y}_i - y_i|$ with the (euclidean) distance $d(\hat{\mathbf{y}}_i, \mathbf{y}_i)$. For a network with a number $m$ of outputs, the distance can be calculated as

$$d(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \sqrt{(\hat{y}_{i1} - y_{i1})^2 + \cdots + (\hat{y}_{im} - y_{im})^2}$$

(this can be obtained by applying Pythagora's theorem, try it as an exercise!). In the one-dimensional case, we had calculated the mean of the squares of $|\hat{y}_i - y_i|$. In the multidimensional case, we calculate the mean of the squares of the distances.

## 4.3 Cross-entropy loss

We have seen how we can use the softmax function to make networks that generate probabilities as outputs. But how can we evaluate how well a given choice of the weights performs? We can write a loss function for the classification problem: the 'cross entropy loss'. Given a set of labels $\mathbf{y}_1, \ldots, \mathbf{y_n}$ and the corresponding network predictions $\hat{\mathbf{y}}_1, \ldots, \hat{\mathbf{y}}_\mathbf{n}$, the cross entropy loss is given by

$$\text{CEloss} = -\sum_{i=1}^{n} \sum_{j=1}^{m} y_{ij} \log \hat{y}_{ij},$$

where $y_{ij}$ is the value of the label for datapoint $i$ and dimension $j$, $n$ is the total number of datapoints, and $m$ is the total number of dimensions.

At first glance, this loss function might look a bit mysterious and complicated. However, it turns out that it is quite intuitive and that it is exactly what we need - let's take a closer look. Let's first consider the cross entropy loss for one single sample (let's say that it's the sample number 1):

$$-\sum_{j=1}^{m} y_{1j} \log \hat{y}_{1j}.$$

Since the values $y_{1j}$ are the true labels, $y_{1j} = 1$ if and only if $j$ is the correct class. Otherwise, $y_{1j} = 0$. Let's remember that in a typical classification problem, one input belongs to only one correct class (we can call it $\bar{j}$). Therefore, all the products $y_{1j} \log \hat{y}_{1j}$ except one are zero (if the true class is $\bar{j}$, the true probability that the input belongs to any other class is zero). The only one among these products that is not zero is the one for which $j = \bar{j}$. That is, $y_{1\bar{j}} \log \hat{y}_{1\bar{j}}$. Therefore:

$$-\sum_{j=1}^{m} y_{1j} \log \hat{y}_{1j} = -y_{1\bar{j}} \log \hat{y}_{1\bar{j}}.$$

Now, let's remember that for the correct class $\bar{j}$, we have that $y_{1\bar{j}} = 1$ (the true probability that the input belongs to class $\bar{j}$ is 1). Therefore:

$$-y_{1\bar{j}} \log \hat{y}_{1\bar{j}} = -\log \hat{y}_{1\bar{j}}.$$

Things are now much simpler than when we started! Let's keep in mind that $\hat{y}_{1\bar{j}}$ is the output of softmax, and therefore it is some number between 0 and 1. The logarithm of 1 is 0, and the logarithm becomes negative for numbers smaller then 1, approaching negative infinity when the numbers approach 0. If the output of the network $\hat{y}_{1\bar{j}}$ is 1, that is the correct answer (because $\bar{j}$ is the correct class). In that case, the loss ($-\log \hat{y}_{1\bar{j}}$) is zero: that makes sense, the network's answer is perfect. However, if $\hat{y}_{1\bar{j}}$ is far from 1 and closer to 0, the $\log \hat{y}_{1\bar{j}}$ is a large negative number, and the loss $-\log \hat{y}_{1\bar{j}}$ is a large positive number. That's why we needed that minus sign in the cross entropy loss: loss function should give you larger numbers when the network is not performing well. Together with the minus sign, the logarithm does exactly what we needed: when the network's output is closer to the correct answer, the loss is smaller, when the network's output is farther from the correct answer, the loss is larger. The value $-\log \hat{y}_{i\bar{j}}$ is then calculated for each sample $i$, and they are all summed together to obtain the overall loss for a set of samples.

**Exercise.** Show that the cross entropy loss is zero if $y_{ij} = \hat{y}_{ij} \ \ \forall i, j$, and that otherwise it is strictly greater than zero.

**A bit of Probability**

**Probability and events.** Intuitively, probability tells us how likely different events are. But can we state a bit more precisely what is a probability? To do that, we need to clarify what is an 'event'. Consider rolling a dice. We might want to ask what is the probability of getting a specific number, i.e. a six. So, to start, we will need an event for each of the 6 numbers on the dice. However, we might also want to ask how likely it is that we will get either a six or a five. So, we will need more events, one for each combination of the possible outcomes.
In the case of the dice, the set of all events is the set of all possible subsets of outcomes. However, sometimes we want to use probability for phenomena whose outcomes are not discrete, but continuous. For example, we might want to ask 'what is the probability that I will finish my homework before 6pm?'. You might finish your homework at 5:50pm, or at 5:52pm, or at 5:52 and 31 seconds... each of these is a different possible outcome, and there is a continuum of them.
In this case, having an event for every possible subset of outcomes leads to some odd issues (you can look up Vitali sets if you are curious, we will not get into more detail here). To avoid these issues, mathematicians use only some of the subsets of outcomes, the 'well behaved' ones. The set of these 'well-behaved' subsets of outcomes is called a $\sigma$-algebra (we will denote it with $F$). I will not report here a formal definition of $\sigma$-algebra (it is a bit too detailed for this course), but you can find it on Wikipedia if you are interested. In conclusion, an event is a subset of the possible outcomes, that is 'well behaved' (in other words, it is in the $\sigma$-algebra). In practice, you don't really need to worry about whether a subset of outcomes is 'well behaved', encountering a subset of outcomes that is not 'well behaved' is very rare.

**Probability measures.** A probability is a function that takes an event as input, and produces a number between 0 and 1 as output. If the probability of an event is 0 the event is impossible, if the probability is 1 the event is certain. However, not all functions that take events as inputs and produce numbers between 0 and 1 as outputs are probabilities. A probability also needs to have two more properties:

1. Imagine the space of all possible outcomes (let's call it $\Omega$). One outcome or another must occur. Therefore, $P(\Omega) = 1$.

2. If two events $A$ and $B$ are disjoint (that is, they do not have any outcomes in common, i.e. the event of rolling a 1 or a 2, and the event of rolling a 3 or a 4), the probability of their union (i.e. the event of rolling any number among 1 ,2, 3 and 4) should be the sum of the probabilities of the individual events: $P(A \cup B) = P(A) + P(B)$. This property should also hold for (countably) infinitely many disjoint events: $P(\bigcup_{i=1}^{\infty} A_i) = \sum_{i=1}^{\infty} P(A_i)$.

If a function satisfies these properties, we say that it is a 'probability measure' (a probability measure is a particular case of a 'measure', a more general concept).

**Probability spaces.** A probability space is a triplet $(\Omega, F, P)$, where $\Omega$ is the set of all possible outcomes, $F$ is the set of events, and $P$ is a probability measure.

## Does the brain calculate loss functions?

**Supervised and unsupervised learning.** We discussed how we can use the correct outputs and the outputs predicted by the network to calculate the value of a loss function, that tells us whether or not a neural network with certain weights is producing accurate predictions. However, often, humans do not know what are the correct outputs. When an artificial neural network is trained to recognize objects, it uses a training dataset of images that already have the correct labels (i.e. 'chair', 'flamingo'). By contrast, humans have to learn to recognize objects without being given the labels. How can this be done? Is it still possible to calculate a loss function without labels? A branch of Artificial Intelligence called 'unsupervised learning' studies how we can build artificial systems that learn representations of the world without using any labels. We will discuss unsupervised learning methods (including 'autoencoders') in later sections of these notes. Recent research suggests that artificial neural networks trained with some types of unsupervised approaches are a good model for neural responses in the visual system of humans [Konkle and Alvarez, 2020] and macaques [Zhuang et al., 2020].

**Responses to prediction error.** Is there some evidence that the brain computes predictions? Is there any indication that, like a loss function, the brain is sensitive to the difference between its predictions and its observations? A seminal study [Rao and Ballard, 1999] found that an artificial model that encodes prediction error, exposed to natural images, learns representations that are similar to simple cells in visual cortex. Since then, a large amount of evidence has accumulated showing that neural responses are sensitive to prediction error, in fields as disparate as perception [Huang and Rao, 2011] and social cognition [Koster-Hale and Saxe, 2013].

# Chapter 5

# Learning

In the section on basic network architectures we learned how to calculate a network's outputs given the inputs and the weights, and in the section on loss functions we learned how to evaluate how well a network with given weights is approximating the outputs. In this key section, we will discuss how we can use some observations (the 'training data') to change the weights of the network until it performs well: we will talk about how the network can learn.

Try to imagine the loss function for all possible values of a network's weights. Start from the simpler case in which the network has two weights: you can imagine a 3 dimensional space in which the first two dimension are the values chosen for the weights, and the third dimension is the value of the loss. For each choice of the two weights, there is a corresponding value of the loss given those weights. When you consider all possible weights, you can think of the loss as a 2 dimensional surface in the 3 dimensional space. When we train a neural network, we try to find weights that are associated with a low value of the loss function.

## 5.1 Gradient descent

The most common approach to find weights for which the loss function is low is 'gradient descent'. Don't worry about the name for now - the gist of 'gradient descent' is as follows:

1. First, we pick the values of the weights at random.
2. Next, we find the direction in weight space in which the loss has the steepest decrease.
3. Finally, we move a small step in that direction.
4. This procedure is repeated for many steps.

Imagine you are on a mountain, and you want to go down as quickly as possible (let's assume you can't get hurt no matter how steep the mountain is). You start in a random position on the mountain, find the steepest direction, and move a step downhill. Then, once you arrive in this second position, you find what is the steepest direction there, and move another step downhill. And so on. Eventually you will continue to descend the mountain, finding a position at a low altitude (i.e. with a low loss).

If we want to use gradient descent, we need to be able to find at each position what is the direction in weight space in which the loss has the steepest decrease. If the loss 'plays nice' (if it is 'differentiable'), then we can find this direction calculating the 'gradient': the direction of steepest descent is minus the gradient. But what is the gradient, and how can we calculate it? The gradient of a differentiable function is the direction in weight space along which the function has the steepest increase. It is denoted with a symbol called 'nabla': for instance, if we wanted to write the gradient of a loss function $L$ we would write $\nabla L$. It can be calculated as the vector of the partial derivatives of the function along the different dimensions. For example, if we have a neural network with two weights $w_1, w_2$, we can write the gradient of the loss as

$$\nabla L(w_1, w_2) = \left[\frac{\partial L(w_1,w_2)}{\partial w_1}, \frac{\partial L(w_1,w_2)}{\partial w_2}\right].$$

More generally, if our network has $n$ weights, we have that

$$\nabla L(w_1,\ldots,w_n) = \left[\frac{\partial L(w_1,\ldots,w_n)}{\partial w_1},\cdots,\frac{\partial L(w_1,\ldots,w_n)}{\partial w_n}\right].$$

I could have started this section by telling you what the gradient is, but then it would have been entirely mysterious why we should care about the gradient at all.

To do gradient descent, we also need to choose the size of the step. The step is usually denoted with the greek letter $\eta$ ('eta'), and it is often a small positive number like 0.01 or 0.001. Now we have all the ingredients to describe more precisely the 4 steps we enumerated earlier:

1. Choose random weights $\mathbf{w^0} = (w_1^0,\ldots,w_n^0)$.
2. Calculate the gradient of the loss function in that point: $\nabla L_{|\mathbf{w^0}} = \left[\frac{\partial L(w_1,\ldots,w_n)}{\partial w_1},\cdots,\frac{\partial L(w_1,\ldots,w_n)}{\partial w_n}\right]_{|\mathbf{w^0}}$.
3. Calculate updated weights $\mathbf{w^1} = (w_1^1,\ldots,w_n^1)$. $\mathbf{w^1} = \mathbf{w^0} - \eta\nabla L_{|\mathbf{w^0}}$.
4. Repeat.

**A bit of Calculus**

**Derivatives.** The derivative of a function $f:\mathbb{R}\to\mathbb{R}$ at a point $x$ is the slope of the tangent line to the function's graph at that point (if such a tangent line exists). If it does not exist, we say that the function is non-derivable.
Let's consider a point $x$ and a point $x+h$, where $h$ is a small positive number. Let's also consider $f(x)$ and $f(x+h)$. We can calculate the slope of the line that goes through the points $(x, f(x))$ and $x+h, f(x+h)$ this way:

$$slope = \frac{f(x+h)-f(x)}{x+h-x} = \frac{f(x+h)-f(x)}{h}.$$

As $h$ gets smaller and smaller, the slope of the line going through $x$ and $x+h$ gets closer and closer to the slope of the tangent line.
When the limit exists, we can say that the function $f$ is differentiable at point $x$, and $d_r = d_l$ is its derivative.

**Example.** Let's calculate the derivative of $f(x) = x^2$ in a generic point $\bar{x}$.

$$\lim_{h\to 0}\frac{f(\bar{x}+h)-f(\bar{x})}{h} = \lim_{h\to 0}\frac{(\bar{x}+h)^2-\bar{x}^2}{h} \quad (5.1)$$
$$= \lim_{h\to 0}\frac{\bar{x}^2+2\bar{x}h+h^2-\bar{x}^2}{h} \quad (5.2)$$
$$= \lim_{h\to 0}\frac{2\bar{x}h+h^2}{h} = \lim_{h\to 0}(2\bar{x}+h) = 2\bar{x}. \quad (5.3)$$

One way to think about derivatives is that they tell us how much $f(x)$ will change for a 'small' change in $x$. If the derivative is large, a small change in $x$ leads to a large change in $f(x)$. If the derivative is small, a small change in $x$ leads to a small change in $f(x)$.

**Partial derivatives.** Sometimes we want to study how much a multivariate function ($f:\mathbb{R}^n\to\mathbb{R}$) changes for small changes in the inputs. For example, when we use neural networks we need to study how small changes in a network's weights lead to changes in the loss function, so that we can change weights in the right way to make the loss lower and the network more accurate. Since we are now considering multivariate functions, the inputs have multiple dimensions. It is convenient to study the effect of small changes in each dimension, one dimension at a time. Let's consider a function with two dimensional inputs: $f(x,y)$. We define its partial derivatives as

$$\frac{\partial f(x,y)}{\partial x} = \lim_{h\to 0} \frac{f(x+h,y)-f(x,y)}{h} \tag{5.4}$$

$$\frac{\partial f(x,y)}{\partial y} = \lim_{h\to 0} \frac{f(x,y+h)-f(x,y)}{h}. \tag{5.5}$$

Another way of thinking about the partial derivatives of a function $f(x,y)$ at a point $[\bar{x},\bar{y}]$ is this: imagine you sliced the graph of $f(x,y)$ cutting through $[\bar{x},\bar{y}]$ parallel to the $x,z$ plane. The cross-section of the slice is the graph of a univariate function of $x$, and the derivative of that function at $\bar{x}$ is the partial derivative $\frac{\partial f(x,y)}{\partial x}$ in $[\bar{x},\bar{y}]$. Likewise, if you sliced the graph of $f(x,y)$ cutting through $[\bar{x},\bar{y}]$ parallel to the $y,z$ plane, the cross-section would be the graph of a univariate function of $y$ whose derivative at $\bar{x}$ is the partial derivative $\frac{\partial f(x,y)}{\partial y}$ in $[\bar{x},\bar{y}]$. The definition and these intuitions can be generalized to functions that have inputs with any number of dimensions.

**Gradient.** The gradient of a function $f:\mathbb{R}^n \to \mathbb{R}$ (denoted with the symbol $\nabla$) is the vector of partial derivatives:

$$\nabla f(x_1,\dots,x_n) = \left[\tfrac{\partial f(x_1,\dots,x_n)}{\partial x_1},\dots,\tfrac{\partial f(x_1,\dots,x_n)}{\partial x_n}\right].$$

The gradient of a function $f$ at a point $[x_1,\dots,x_n]$ is a vector in input space that denotes the direction in which the function $f$ at that point has the steepest increase.

## 5.2 Backpropagation

When we use networks that have many layers, calculating the gradient of the los function can become computationally intensive. Backpropagation is an algorithm that makes it possible to calculate the gradient more efficiently, saving computation time and making it possible to train larger, deeper networks. We will not delve into the details of backpropagation in these notes.

**Does the brain do gradient descent?**

It seems challenging for the brain to calculate the gradient of a loss function with respect to all the weights. Information about the error of a prediction generated by a brain region would need to be conveyed back to all neurons in remote brain regions that contributed at some point in shaping that output. Is it possible to do something that is just like gradient descent, just like backpropagation, but using local mechanisms so that a neuron can learn using only information about its neighbors? A lot of effort has been devoted to answering this question. In a breakthrough, a new study has found a local learning algorithm equivalent to backpropagation [Song et al., 2020]. However, it is still unknown whether a backpropagation-like learning mechanism is implemented in the brain.

## 5.3 Batches and epochs

When we want to train an artificial neural network in practice, we usually use a 'general purpose graphic processing unit' (GPGPU), because it is faster than a standard CPU for this purpose. The memory of a GPGPU is limited, it is not large enough to hold the entire dataset. For this reason, instead of calculating the loss for gradient descent with the whole dataset, we divide the dataset into 'batches': smaller subsets of the entire dataset (they might contain 16, or 32, or 64 inputs; sometimes even more). We calculate the loss and the gradient for one batch, and update the weights of the network using that gradient. Then we move to the next batch.

The inputs that end up in a batch are often chosen randomly, without repetitions until the entire dataset has been used (although there are exceptions). Once all the elements in the datasets have been used as inputs once, an 'epoch' has been completed. A network is usually trained for several epochs. This might seem counterintuitive: if the network has already 'seen' all the datapoints once (that is, if one epoch has been completed), what is the point of showing them to the network again? The point is that the network does not learn everything that there is to learn from one datapoint the very first time. When the network makes a mistake, it will make a small step in the direction that reduces the loss, but this step might not be large enough to fix the mistake. Even if the step were sufficient to fix the mistake, later batches could lead the network to change in a way that brings the mistake back. With multiple epochs of training, the network can gradually adjust until it makes enough steps to fix many of the mistakes.

# Chapter 6

# Convolutional neural networks

Recognizing objects from pictures has been a landmark problem in the history of Artificial Intelligence. Taking inspiration from the brain, AI researchers introduced a type of neural networks - convolutional neural networks - that is very successful for artificial vision. Convolutional neural networks are now used in many applications outside of the field of vision as well. It took decades to develop neural networks that are accurate at object recogntion. What makes this problem hard? And how do convolutional neural networks solve it?

## 6.1 The problem

Pictures have a large number of pixels. A typical Instagram picture, for example, has 1080x1080 pixes, that is, 1166400 pixels! Neural networks like the ones we discussed so far in this course have a weight for each 'connection' between an input dimension and a dimension in the following layer. However, having a different weight for each pixel leads to a very large number of parameters. When a model has a very large number of parameters as compared to the amount of data available to learn those parameters, the model can suffer from 'overfitting': it can perform well in the training set, but it generalizes poorly to new datapoints outside the training set.

## 6.2 The solution

How can we address the problem of overfitting? We could try to remove some of the parameters. However, removing many parameters randomly would probably lead the network to perform poorly. We would like to find a clever way of reducing the number of parameters that gives us a network that performs well.

If we think about images, there are some similar image parts that can be repeated multiple times. Sometimes an image part (like an eye, or a traffic light) can appear in different locations of a picture. For example, one picture might have a traffic light in the top right corner, and another picture might have one in the bottom left. We can learn weights that, applied to a small patch of an image, detect some pattern - like a traffic light. Then, we can apply those same weights to all different small patches of the image. This way, if the network encounters a new picture in which the traffic light shows up in a new location, it will be able to reuse the weights it learned to recognize the traffic light when it appeared somewhere else. The process of applying the weights to all the different patches of the image is called 'convolution', and it is the reason these networks are called convolutional neural networks. To express this a bit more precisely, 'applying the weights to a patch of the image' means multiplying the value in each pixel in the patch by the corresponding weight in the kernel (the patch and the kernel must have the same size), and summing all the resulting products. Convolutional neural networks typically perform better than fully connected networks on image datasets of usual sizes (up to millions of images).

This design is inspired to seminal work by Hubel and Wiesel on the responses of neurons in primary visual cortex [Hubel and Wiesel, 1959, Hubel and Wiesel, 1962]. Hubel and Wiesel found that neurons in the cat's primary visual cortex responded selectively to oriented lines in specific locations of the visual field.

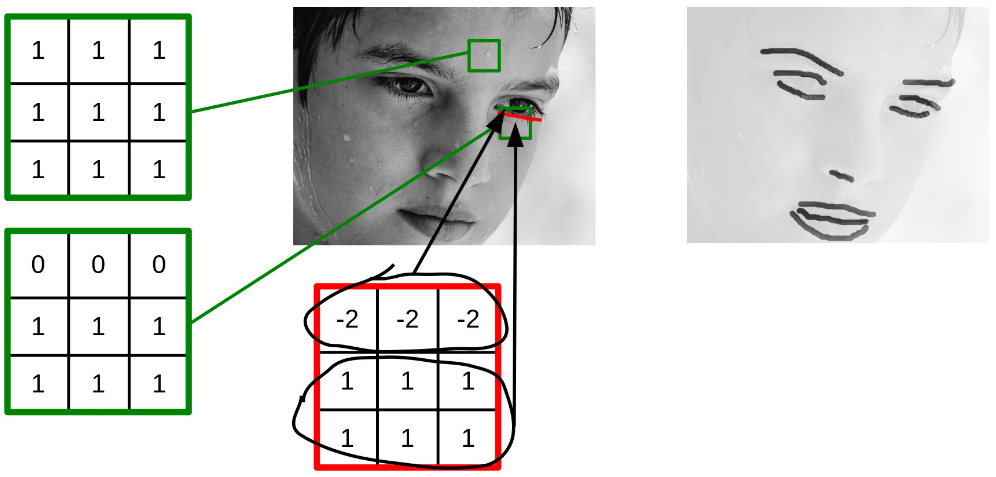


Figure 6.1: Visualization of a kernel that detects horizontal lines. The kernel weights (shown with a red border) return a positive output when there is a horizontal line (e.g. the patch of pixel values on the lower left - with a green border - that has dark pixels with 0 values on top and bright pixels with 1 values on the bottom). By contrast, when applied to a region with uniform pixel values (like the one with a green border on the top left), the output is zero. When applied to all the patches in an image, the kernel can produce as output a channel that tells us how similar a patch of the image is to a horizontal line, effectively 'detecting' the horizontal lines in the image. Original face image from Wilfredor, CC0, via Wikimedia Commons. `https://commons.wikimedia.org/wiki/File:Boy_Face_from_Venezuela.jpg`

For this work Hubel and Wiesel were awarded the Nobel prize for medicine in 1981. Think of the oriented line as the traffic light in the previous example. There are neurons that detect a vertical line in a certain location, and then other neurons that use 'the same weights' to detect also a vertical line, but in a different location. In a convolutional network, the weights used to detect a type of pattern are called a 'kernel'. There can be more than one kernel. For example, for each location in the visual field, we have some vertical line detectors but we also have horizontal line detectors, and diagonal line detectors. Each kernel produces one output when it is applied to one patch of the image. The set of outputs obtained by applying one kernel to all the patches in the image is called a 'channel'. Therefore, for each kernel, there is one output channel.

**Exercise.** Consider an image patch $P$ and a kernel $K$:

$$P = \begin{bmatrix} -1 & 2 \\ 2 & 3 \end{bmatrix}, \quad K = \begin{bmatrix} 2 & 1 \\ -1 & 1 \end{bmatrix}.$$

Calculate the output obtained applying the kernel to the patch.

There are several ways to apply a kernel to an image. Imagine that we start from the top-left corner of the image. One possibility is to apply the kernel to the top-left corner patch, then to slide it by one pixel to the right and apply it to this new patch, and so on. When we went through the entire row, we can slide it down by 1 pixel, and restart from the left, completing another row. In this case, we say that the **'stride'** is 1. However, we could also slide the kernel by 2 pixels at a time (stride 2), and so on.

When an image has color, we can represent it as an input with 3 **channels**. The first channel tells us how much red there is in each pixel, the second how much blue there is, and the third one how much green (RGB coding). How do kernels work when the input has multiple channels? We just need a larger kernel, that has one channel for each input channel. As before, we multiply each element in a patch of the image by each element of the kernel, and we sum them all across locations in the patch and across channels.

**Exercise.** Write an example of a kernel for inputs color images, that detects vertical lines in the blue channel.

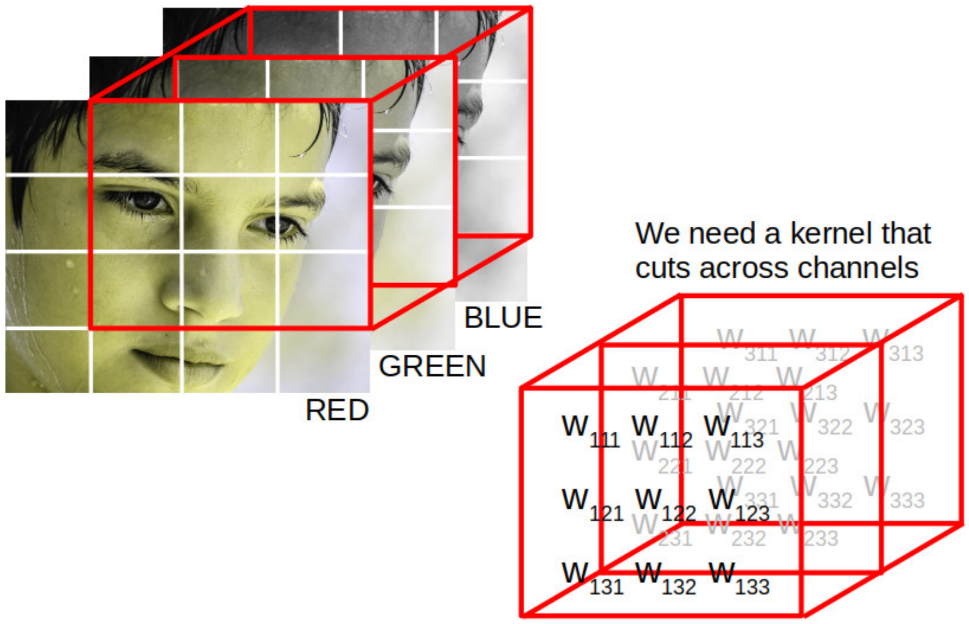


Figure 6.2: Visualization of a kernel for RGB color images. Original face image from Wilfredor, CC0, via Wikimedia Commons. `https://commons.wikimedia.org/wiki/File:Boy_Face_from_Venezuela.jpg`

# Chapter 7

# Comparing artificial networks and the brain

In Neuroscience, we often use artificial neural networks as models of the brain. To assess whether an artificial neural network is a good model of the brain, we compare what happens inside that network to what happens inside the brain. Imagine for example that we want to study vision, and we want to test whether an artificial neural network that recognizes images is a good model of a particular brain region. We can choose a set of images, and show them to the artificial neural network. For each image, we can then obtain the output of each layer of the network. We will call the output of a layer of the network the "representation" of the image in that layer. Similarly, we can show the images to a participant while we are recording their neural responses. For each image, we can obtain the responses in the brain region that we are interested in. These are usually a vector of values: if we use fMRI, the values are the amount of response in different voxels in the region, if we are using monkey physiology, they might be the recordings from different electrodes in the region. We will call these the "representation" of the image in that brain region.

At this point, we face a challenge. We would like to compare the representation of the image in the brain region to the representation of the image in the layers of the artificial neural network. But the former contains neural responses in voxels (or electrodes), while the latter contains the outputs of the units of the artificial neural network. There is not an obvious one-to-one correspondence between voxels and units, so we cannot compare them directly. To solve this problem, researchers typically use one of two strategies: "representational dissimilarity matrices", or "encoding models".

Representation dissimilarity matrices solve the problem as follows. We show several different images to the artificial neural network, and we show the same images to our participants. Then, instead of comparing directly the representations in a layer of the artificial neural network to the representations in a brain region, we first calculate the dissimilarities between the representations of different images in the layer of the artificial neural network. We can arrange them into a square matrix, where the element at position $i, j$ contains the dissimilarity between the representations of image $i$ and image $j$. The size of this matrix will be $n \times n$, where $n$ is the number of images. At this point, we can repeat the same procedure for the representations in the brain region, obtaining another $n \times n$ matrix. This is really helpful, because now we have two matrices of the same size: one for the brain region, and one for the artificial neural network. Therefore, we can compare them directly and see whether the brain region and the artificial neural network "agree" on what images are similar to each other and what images are more different.

**Mean, variance, covariance, correlation**

**Mean.** Let's consider a vector $\mathbf{x} = (x_1, \dots, x_n)$. Its mean is the value $\bar{\mathbf{x}} = \frac{\sum_{i=1}^n x_i}{n}$.

**Variance.** Let's consider a vector $\mathbf{x} = (x_1, \dots, x_n)$. Its variance is the value $\text{var}(\mathbf{x}) = \frac{\sum_{i=1}^n (x_i - \bar{\mathbf{x}})^2}{n}$. When the values in $\mathbf{x}$ are samples from a random variable, we use the sample variance $\text{var}(\mathbf{x}) = \frac{\sum_{i=1}^n (x_i - \bar{\mathbf{x}})^2}{n-1}$, because it converges to the true variance of the random variable when the size of the sample tends to infinity.

**Standard deviation.** The standard deviation $\sigma(\mathbf{x})$ is the square root of the variance:

$$\sigma(\mathbf{x}) = \sqrt{\mathrm{var}(\mathbf{x})}$$

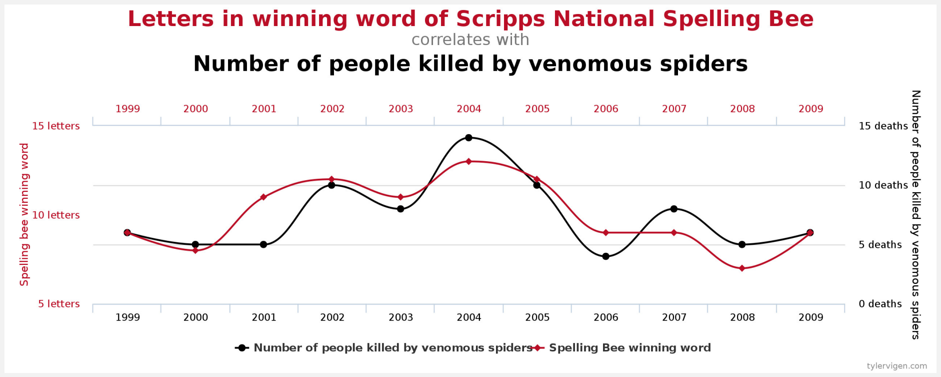
**Covariance.** Let's consider a vector $\mathbf{x} = (x_1, \ldots, x_n)$, and a vector $\mathbf{y} = (y_1, \ldots, y_n)$. The covariance between $\mathbf{x}$ and $\mathbf{y}$ is given by

$$\mathrm{cov}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n}(x_i - \bar{\mathbf{x}})(y_i - \bar{\mathbf{y}})}{n}.$$

Just like in the case of the variance, when the values in $\mathbf{x}$ and $\mathbf{y}$ are samples from random variables, we use the sample covariance

$$\mathrm{cov}(\mathbf{x}, \mathbf{y}) = \frac{\sum_{i=1}^{n}(x_i - \bar{\mathbf{x}})(y_i - \bar{\mathbf{y}})}{n-1}.$$

Note that if $\mathbf{y} = \mathbf{x}$, then $\mathrm{cov}(\mathbf{x}, \mathbf{y}) = \mathrm{var}(\mathbf{x})$. Also note that the covariance is larger if individual values $x_i$ in $\mathbf{x}$ are larger than the mean $\bar{\mathbf{x}}$ when the corresponding values $y_i$ in $\mathbf{y}$ are larger than the mean $\bar{\mathbf{y}}$. In other words, if you plot $\mathbf{x}$ and $\mathbf{y}$, the covariance tends to be larger if $\mathbf{x}$ "goes up" when $\mathbf{y}$ "goes up", and $\mathbf{x}$ "goes down" when $\mathbf{y}$ "goes down". Like the red and black lines below:



Importantly, the covariance is sensitive to the scale of $\mathbf{x}$ and $\mathbf{y}$. A covariance close to zero might mean that $\mathbf{x}$ and $\mathbf{y}$ don't go up and down together, or it might mean that they do go up and down together but they have very small values. To check whether $\mathbf{x}$ and $\mathbf{y}$ go up and down together regardless of whether they have large or small values, we can normalize the covariance by the standard deviation of each vector, thus calculating the correlation.

**Correlation.** Let's consider a vector $\mathbf{x} = (x_1, \ldots, x_n)$, and a vector $\mathbf{y} = (y_1, \ldots, y_n)$. The correlation between $\mathbf{x}$ and $\mathbf{y}$ is given by

$$\mathrm{r}(\mathbf{x}, \mathbf{y}) = \frac{\mathrm{cov}(\mathbf{x}, \mathbf{y})}{\sigma(\mathbf{x})\sigma(\mathbf{y})}.$$

Note that the lowest possible correlation is $-1$, and the highest possible correlation is 1. Also note that the correlation has the following three properties:

$$\begin{aligned}\mathrm{r}(\mathbf{x}, \mathbf{y}) &= \mathrm{r}(\mathbf{y}, \mathbf{x})\\ \mathrm{r}(\mathbf{x}+u, \mathbf{y}) &= \mathrm{r}(\mathbf{x}, \mathbf{y})\,\forall u \in \mathbb{R}\\ \mathrm{r}(\lambda\mathbf{x}, \mathbf{y}) &= \mathrm{r}(\mathbf{x}, \mathbf{y})\,\forall \lambda > 0\end{aligned}$$

Encoding models use a different strategy to solve the same problem.

# Chapter 8

# Appendix: backpropagation

In these notes we will see how to train an artificial neural network with backpropagation.

### 8.0.1 Many inputs to one output

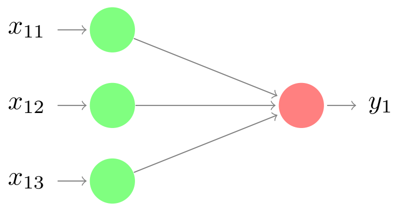
Let's first consider an artificial neural network with $I$ inputs and one output. Let's consider an input vector $x_{11}, \ldots, x_{1I}$, and the corresponding output $\hat{y}_1$. The neural network calculates the output with the following equation:

$$\hat{y}_1 = \phi(w_1 x_{11} + \cdots + w_I x_{1I}) \tag{8.1}$$

where $\phi$ is a differentiable nonlinear function (see Appendix), and $w_1, \ldots, w_I$ are the 'weights' of the network. For convenience, we can rewrite equation 8.1 using vector notation, using the dot product (see Appendix):

$$\hat{y}_1 = \phi(\mathbf{w}^T \mathbf{x}_1) \tag{8.2}$$

where $\mathbf{w}$ is the vector of weights $(w_1, \ldots, w_I)$ and $\mathbf{x}_1$ is the vector of inputs $(x_{11}, \ldots, x_{1I})$.



Suppose that we have made a set of $L$ observations $(\mathbf{x}_1, y_1), \ldots, (\mathbf{x}_L, y_L)$. We want to use these observations to learn weights $\mathbf{w}$ so that when we give the network an input $\mathbf{x}_l$, the output $\hat{y}_l$ is close to the observed $y_l$.

#### Writing a loss function

We can write a **loss function** that tells us how 'distant' the outputs of the network are from the observations:

$$E = (\hat{y}_1 - y_1)^2 + \cdots + (\hat{y}_L - y_L)^2. \tag{8.3}$$

This loss function is called **mean square error** (MSE) loss. For convenience, we can rewrite it as

$$E = \sum_{l=1}^{L} (\hat{y}_l - y_l)^2. \tag{8.4}$$

As we change the weights $\mathbf{w}$ of the network, the outputs $\hat{y}_l$ also change. Therefore, also the loss function $E$ depends on the weights. We can write it more clearly as:

$$E(\mathbf{w}) = \sum_{l=1}^{L} (\phi(\mathbf{w}^T \mathbf{x}_l) - y_l)^2 \tag{8.5}$$

(because $\hat{y}_l = \phi(\mathbf{w}^T \mathbf{x}_l)$).

Since the loss measures how different the outputs are from the observations, we want to find weights $\mathbf{w}$ that make the loss as small as possible.

To make things concrete, imagine there are only two weights $w_1, w_2$. You can visualize all possible choices of $w_1, w_2$ as points on a plane. For each point on the plane, imagine that the loss is a value plotted on a third dimension.

## Gradient descent

To find the point where the loss is lowest, we can start from some random point, and go downhill along the direction of steepest descent. This direction is called the **gradient**, and this strategy is called **gradient descent**.

The loss's gradient can be calculated as the vector of the derivatives of the loss function along the weights:

$$\nabla E(\mathbf{w}) = \left( \frac{\partial E}{\partial w_1}, \cdots, \frac{\partial E}{\partial w_I} \right). \tag{8.6}$$

To do gradient descent, we need to calculate these $\frac{\partial E}{\partial w_{i^*}}$. We can use Equation 8.5 to write $E$ as a function of $\mathbf{w}$:

$$\frac{\partial E}{\partial w_{i^*}} = \frac{\partial}{\partial w_{i^*}} \sum_{l=1}^{L} (\phi(\mathbf{w}^T \mathbf{x}_l) - y_l)^2 \tag{8.7}$$

and thanks to the linearity of the differential operator:

$$\frac{\partial E}{\partial w_{i^*}} = \sum_{l=1}^{L} \frac{\partial}{\partial w_{i^*}} (\phi(\mathbf{w}^T \mathbf{x}_l) - y_l)^2. \tag{8.8}$$

So all we really have to do is calculate $\frac{\partial}{\partial w_{i^*}} (\phi(\mathbf{w}^T \mathbf{x}_l) - y_l)^2$. To do this, we can use the chain rule of derivation (see Appendix):

$$\frac{\partial}{\partial w_{i^*}} (\phi(\mathbf{w}^T \mathbf{x}_l) - y_l)^2 = 2(\phi(\mathbf{w}^T \mathbf{x}_l) - y_l)\phi'(\mathbf{w}^T \mathbf{x}_l) x_{li^*}. \tag{8.9}$$

In the end, each component $i^*$ of the gradient is given by:

$$\frac{\partial E}{\partial w_{i^*}} = \sum_{l=1}^{L} 2(\phi(\mathbf{w}^T \mathbf{x}_l) - y_l)\phi'(\mathbf{w}^T \mathbf{x}_l) x_{li^*}. \tag{8.10}$$

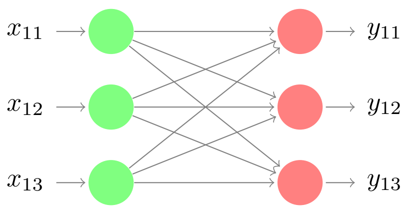
To look for weights $\mathbf{w}$ for which the loss function $E(\mathbf{w})$ is low, we can start from some initial random weights $\mathbf{w}^0$, and update the weights with the equation

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t) \tag{8.11}$$

where $\eta$ is a parameter that regulates how fast we are moving in the direction of the gradient with each update.

### 8.0.2 Many inputs to many outputs

Let's now consider a case in which both the input and the output are multidimensional. Given an $I$ dimensional input $\mathbf{x}_1 = x_{11}, \ldots, x_{1I}$ the network generates a $J$ dimensional output $\hat{\mathbf{y}}_1 = \hat{y}_{11}, \ldots, \hat{y}_{1J}$.



Each output dimension $\hat{y}_j$ is calculated with the following equation:

$$\hat{y}_{1j} = \phi(w_{1j}x_{11} + \cdots + w_{Ij}x_{1I}). \tag{8.12}$$

We call $w_{ij}$ the weight from input node $i$ to output node $j$. Note that each output dimension transforms the inputs using different weights, this is why now the weights have an index $j$ in addition to the index for the number of the input dimension.

For convenience, we can rewrite equation 8.12 using the dot product:

$$\hat{y}_{1j} = \phi(\mathbf{w}_j^T \mathbf{x}_1). \tag{8.13}$$

#### Writing a loss function for multivariate outputs

To write a loss function in the case of multivariate outputs, we can use the intuition that the loss should capture how 'distant' the predictions are from the observations. We can calculate the **Euclidean distance** between a prediction $\hat{\mathbf{y}}_1$ and an observation $\mathbf{y}_1$ with this equation:

$$d(\hat{\mathbf{y}}_1, \mathbf{y}_1) = \sqrt{\sum_{j=1}^{J} (\hat{y}_{1j} - y_{1j})^2}. \tag{8.14}$$

Note that if $J = 2$ this is just Pythagora's theorem!

For convenience, we usually use the square of the distance to build our loss:

$$d(\hat{\mathbf{y}}_1, \mathbf{y}_1)^2 = \sum_{j=1}^{J} (\hat{y}_{1j} - y_{1j})^2. \tag{8.15}$$

We can make our loss for multivariate outputs summing for all observations how far the predictions are from the observations:

$$E = \sum_{l=1}^{L} \left( \sum_{j=1}^{J} (\hat{y}_{lj} - y_{lj})^2 \right). \tag{8.16}$$

Compare equation 8.16 with equation 8.4, and notice that the latter is a particular case of the former when $J = 1$.

We can write the loss explicitly as a function of the weights $\mathbf{w}_1, \ldots, \mathbf{w}_J$:

$$E(\mathbf{w}_1, \ldots, \mathbf{w}_J) = \sum_{l=1}^{L} \left( \sum_{j=1}^{J} (\phi(\mathbf{w}_j^T \mathbf{x}_l) - y_{lj})^2 \right). \tag{8.17}$$

**Gradient descent for multivariate outputs**

Like in the case of the one-dimensional output, we want to find weights that minimize the loss, and to do it we can use gradient descent. For each weight $w_{i^*j^*}$ we can calculate the partial derivative

$$\frac{\partial E}{\partial w_{i^*j^*}} = \frac{\partial}{\partial w_{i^*j^*}} \sum_{l=1}^{L} \left( \sum_{j=1}^{J} (\phi(\mathbf{w}_j^T \mathbf{x}_l) - y_{lj})^2 \right). \tag{8.18}$$

Using the linearity of the differential operator we can write that

$$\frac{\partial E}{\partial w_{i^*j^*}} = \sum_{l=1}^{L} \left( \sum_{j=1}^{J} \frac{\partial}{\partial w_{i^*j^*}} (\phi(\mathbf{w}_j^T \mathbf{x}_l) - y_{lj})^2 \right). \tag{8.19}$$

We also note that when $j \neq j^*$, then $\phi(\mathbf{w}_j^T \mathbf{x}_l) - y_{lj})^2$ is constant in $w_{i^*j^*}$. Therefore

$$\frac{\partial E}{\partial w_{i^*j^*}} = \sum_{l=1}^{L} \left( \frac{\partial}{\partial w_{i^*j^*}} (\phi(\mathbf{w}_{j^*}^T \mathbf{x}_l) - y_{lj^*})^2 \right). \tag{8.20}$$

So all we really have to do is calculate $\frac{\partial}{\partial w_{i^*j^*}} (\phi(\mathbf{w}_{j^*}^T \mathbf{x}_l) - y_{lj^*})^2$. To do this, we can use the chain rule of derivation (see Appendix):

$$\frac{\partial}{\partial w_{i^*j^*}} (\phi(\mathbf{w}_{j^*}^T \mathbf{x}_l) - y_{lj^*})^2 = 2(\phi(\mathbf{w_{j^*}}^T \mathbf{x}_l) - y_{lj^*})\phi'(\mathbf{w_{j^*}}^T \mathbf{x}_l) x_{li^*}. \tag{8.21}$$

In the end, each component $i^*, j^*$ of the gradient is given by:

$$\frac{\partial E}{\partial w_{i^*j^*}} = \sum_{l=1}^{L} 2(\phi(\mathbf{w_{j^*}}^T \mathbf{x}_l) - y_{lj^*})\phi'(\mathbf{w_{j^*}}^T \mathbf{x}_l) x_{li^*}. \tag{8.22}$$

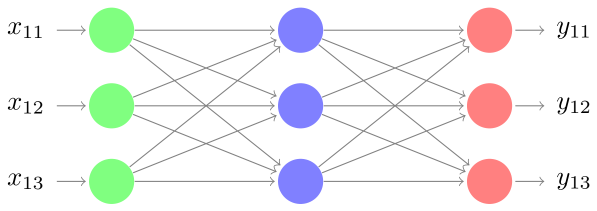
As in the single-output case, we can start from some initial random weights $\mathbf{w}^0$, and update the weights with the equation

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t) \tag{8.23}$$

where $\eta$ is a parameter that regulates how fast we are moving in the direction of the gradient with each update.

## 8.0.3 Many layers

Now we are ready to see what happens when we have more than one layer. We will consider the case with two layers as an example. In this case, the 'trick' of backpropagation becomes clear. Given an $I$ dimensional input $\mathbf{x}_1 = x_{11}, \ldots, x_{1I}$ the network generates a $J$ dimensional hidden state $\mathbf{z}_1 = z_{11}, \ldots, z_{1J}$. In turn, from the $J$ dimensional hidden state $\mathbf{z}_1$, the network generates a $K$ dimensional output $\hat{\mathbf{y}}_1 = \hat{y}_{11}, \ldots, \hat{y}_{1K}$



Elements of the hidden state are calculated as

$$z_{1j} = \phi(\mathbf{w}_j^T \mathbf{x}_1). \tag{8.24}$$

Elements of the output are calculated as

$$\hat{y}_{1k} = \phi(\mathbf{w}_k^T \mathbf{z}_1). \tag{8.25}$$

**Gradient descent for a two layer network**

We need to calculate the gradient of the loss function $E = \sum_{l=1}^{L}\left(\sum_{k=1}^{K}(\hat{y}_{lk} - y_{lk})^2\right)$ with respect to all the weights: both the weights between the hidden layer and the output, and the weights between the input and the hidden layer. We call $w_{ij}$ the weight from input node $i$ to hidden node $j$, and $w_{jk}$ the weight from hidden node $j$ to output node $k$. In other words, $\mathbf{w}_j = (w_{1j}, \ldots, w_{Ij})$ and $\mathbf{w}_k = (w_{jk}, \ldots, w_{Jk})$.

First we calculate

$$\begin{aligned}
\frac{\partial E}{\partial w_{j^*k^*}} &= \frac{\partial}{\partial w_{j^*k^*}} \sum_{l=1}^{L}\left(\sum_{k=1}^{K}(\hat{y}_{lk} - y_{lk})^2\right) && (8.26)\\
&= \frac{\partial}{\partial w_{j^*k^*}} \sum_{l=1}^{L}\left(\sum_{k=1}^{K}(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})^2\right) && (8.27)\\
&= \sum_{l=1}^{L}\left(\sum_{k=1}^{K}\frac{\partial}{\partial w_{j^*k^*}}(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})^2\right) && (8.28)\\
&= \sum_{l=1}^{L} 2(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})\phi'(\mathbf{w}_k^T\mathbf{z}_l)z_{lj^*}. && (8.29)
\end{aligned}$$

where 8.27 is obtained by replacing $\hat{y}_{lk}$ with 8.25, 8.28 follows from the linearity of the differential operator, and 8.29 is obtained applying the formula for the derivation of composite functions (see Appendix).

Now we need to calculate the partial derivatives with respect to the weights $w_{i^*j^*}$. Using the linearity of the differential operator we obtain

$$\frac{\partial E}{\partial w_{i^*j^*}} = \sum_{l=1}^{L}\left(\sum_{k=1}^{K}\frac{\partial}{\partial w_{i^*j^*}}(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})^2\right) \tag{8.30}$$

and applying the formula for the derivative of composite multivariate functions (see Appendix) we have

$$\begin{aligned}
\frac{\partial}{\partial w_{i^*j^*}}(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})^2 &= \sum_{j=1}^{J}\frac{\partial}{\partial z_{lj}}(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})^2\frac{\partial z_{lj}}{\partial w_{i^*j^*}} && (8.31)\\
&= \frac{\partial}{\partial z_{lj^*}}(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})^2\frac{\partial z_{lj^*}}{\partial w_{i^*j^*}} && (8.32)
\end{aligned}$$

because when $j \neq j^*$, $z_{lj}$ is constant in $w_{i^*j^*}$ and therefore $\frac{\partial z_{lj}}{\partial w_{i^*j^*}} = 0$.
To conlcude,

$$\frac{\partial}{\partial z_{lj^*}}(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})^2 = 2(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})\phi'(\mathbf{w}_k^T\mathbf{z}_l)w_{j^*k} \tag{8.33}$$

and

$$\begin{aligned}
\frac{\partial z_{lj^*}}{\partial w_{i^*j^*}} &= \frac{\partial}{\partial w_{i^*j^*}}\phi(\mathbf{w}_j^T\mathbf{x}_l) && (8.34)\\
&= \phi'(\mathbf{w}_j^T\mathbf{x}_l)x_{li^*} && (8.35)
\end{aligned}$$

therefore

$$\frac{\partial E}{\partial w_{i^*j^*}} = \sum_{l=1}^{L}\left(\sum_{k=1}^{K}2(\phi(\mathbf{w}_k^T\mathbf{z}_l) - y_{lk})\phi'(\mathbf{w}_k^T\mathbf{z}_l)w_{j^*k}\phi'(\mathbf{w}_j^T\mathbf{x}_l)x_{li^*}\right). \tag{8.36}$$

Let's compare the equation 8.29 for the weights $w_{j^*k^*}$ and equation 8.36 for the weights $w_{i^*j^*}$:

$$\frac{\partial E}{\partial w_{j^*k^*}} = \sum_{l=1}^{L} 2(\phi(\mathbf{w}_k^T \mathbf{z}_l) - y_{lk})\phi'(\mathbf{w}_k^T \mathbf{z}_l) z_{lj^*}. \tag{8.37}$$

$$\frac{\partial E}{\partial w_{i^*j^*}} = \sum_{l=1}^{L} \left( \sum_{k=1}^{K} 2(\phi(\mathbf{w}_k^T \mathbf{z}_l) - y_{lk})\phi'(\mathbf{w}_k^T \mathbf{z}_l) w_{j^*k} \phi'(\mathbf{w}_j^T \mathbf{x}_l) x_{li^*} \right). \tag{8.38}$$

The error is 'propagating back' to earlier layers through the networks weights $w_{j^*k}$.

As usual, to look for good weights we can start from some initial random weights $\mathbf{w}^0$, and update the weights with the equation

$$\mathbf{w}^{t+1} = \mathbf{w}^t - \eta \nabla E(\mathbf{w}^t) \tag{8.39}$$

where $\mathbf{w}$ is a vector containing all the weights, and $\eta$ is a parameter that regulates how fast we are moving in the direction of the gradient with each update.

### Generalizing to more than three layers

Looking back to what we have written so far, we can notice a pattern. Suppose we have many hidden layers $\mathbf{z}^1, \ldots, \mathbf{z}^n$, so that

$$z_{j^1}^1 = \phi(\mathbf{w}_{j^1}^{1T} \mathbf{x}) \tag{8.40}$$

$$\cdots \tag{8.41}$$

$$z_{j^n}^n = \phi(\mathbf{w}_{j^n}^{nT} \mathbf{z}^{(n-1)}) \tag{8.42}$$

$$y_k = \phi(\mathbf{w}_k^{(n+1)T} \mathbf{z}^n). \tag{8.43}$$

To calculate the gradient of the loss function $E$ with respect to the weights we can start from the last layer and compute:

$$\frac{\partial E}{\partial w_{j^n k}} \tag{8.44}$$

$$\frac{\partial E}{\partial z_{j^n}^n} \qquad \frac{\partial E}{\partial w_{j^{n-1}j^n}} = \frac{\partial E}{\partial z_{j^n}^n} \frac{\partial z_{j^n}^n}{\partial w_{j^{n-1}j^n}} \tag{8.45}$$

$$\frac{\partial E}{\partial z_{j^{n-1}}^{n-1}} = \sum_{j^n} \frac{\partial E}{\partial z_{j^n}^n} \frac{\partial z_{j^n}^n}{\partial z_{j^{n-1}}^{n-1}} \qquad \frac{\partial E}{\partial w_{j^{n-2}j^{n-1}}} = \frac{\partial E}{\partial z_{j^{n-1}}^{n-1}} \frac{\partial z_{j^{n-1}}^{n-1}}{\partial w_{j^{n-2}j^{n-1}}} \tag{8.46}$$

$$\frac{\partial E}{\partial z_{j^{n-2}}^{n-2}} = \sum_{j^{n-1}} \frac{\partial E}{\partial z_{j^{n-1}}^{n-1}} \frac{\partial z_{j^{n-1}}^{n-1}}{\partial z_{j^{n-2}}^{n-2}} \qquad \frac{\partial E}{\partial w_{j^{n-3}j^{n-2}}} = \frac{\partial E}{\partial z_{j^{n-2}}^{n-2}} \frac{\partial z_{j^{n-2}}^{n-2}}{\partial w_{j^{n-3}j^{n-2}}} \tag{8.47}$$

$$\cdots \tag{8.48}$$

As we can see, we can reuse a lot of calculations.

## Backpropagation's foundations: chain rules of derivation

### Chain rule of derivation

Consider a composite function $f(g(x))$. The chain rule of derivation states that we can calculate its derivative as

$$\frac{d}{dx} f(g(x)) = \frac{df(g(x))}{dg(x)} \frac{dg(x)}{dx}. \tag{8.49}$$

### Chain rule of derivation for multivariate functions

$$\frac{d}{dx} f(g_1(x), \ldots, g_n(x)) = \sum_{i=1}^{n} \frac{df(g(x))}{dg_i(x)} \frac{dg_i(x)}{dx}. \tag{8.50}$$

# Bibliography

[Ascoli et al., 2007] Ascoli, G. A., Donohue, D. E., and Halavi, M. (2007). Neuromorpho. org: a central resource for neuronal morphologies. *Journal of Neuroscience*, 27(35):9247–9251.

[Fukushima et al., 1983] Fukushima, K., Miyake, S., and Ito, T. (1983). Neocognitron: A neural network model for a mechanism of visual pattern recognition. *IEEE transactions on systems, man, and cybernetics*, (5):826–834.

[Goodfellow et al., 2014] Goodfellow, I., Pouget-Abadie, J., Mirza, M., Xu, B., Warde-Farley, D., Ozair, S., Courville, A., and Bengio, Y. (2014). Generative adversarial nets. *Advances in neural information processing systems*, 27.

[Guan et al., 2020] Guan, J., Yuan, Y., Kitani, K. M., and Rhinehart, N. (2020). Generative hybrid representations for activity forecasting with no-regret learning. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 173–182.

[Herring et al., 2015] Herring, S. C., Hoerling, M. P., Kossin, J. P., Peterson, T. C., and Stott, P. A. (2015). Explaining extreme events of 2014 from a climate perspective. *Bulletin of the American Meteorological Society*, 96(12):S1–S172.

[Huang and Rao, 2011] Huang, Y. and Rao, R. P. (2011). Predictive coding. *Wiley Interdisciplinary Reviews: Cognitive Science*, 2(5):580–593.

[Hubel and Wiesel, 1959] Hubel, D. H. and Wiesel, T. N. (1959). Receptive fields of single neurones in the cat's striate cortex. *The Journal of physiology*, 148(3):574.

[Hubel and Wiesel, 1962] Hubel, D. H. and Wiesel, T. N. (1962). Receptive fields, binocular interaction and functional architecture in the cat's visual cortex. *The Journal of physiology*, 160(1):106.

[Karl and Trenberth, 2003] Karl, T. R. and Trenberth, K. E. (2003). Modern global climate change. *science*, 302(5651):1719–1723.

[Konkle and Alvarez, 2020] Konkle, T. and Alvarez, G. A. (2020). Instance-level contrastive learning yields human brain-like representation without category-supervision. *bioRxiv*.

[Koster-Hale and Saxe, 2013] Koster-Hale, J. and Saxe, R. (2013). Theory of mind: a neural prediction problem. *Neuron*, 79(5):836–848.

[Li et al., 2017] Li, R., Tapaswi, M., Liao, R., Jia, J., Urtasun, R., and Fidler, S. (2017). Situation recognition with graph neural networks. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4173–4182.

[Lu et al., 2017] Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. (2017). The expressive power of neural networks: A view from the width. In *Advances in neural information processing systems*, pages 6231–6239.

[McCulloch and Pitts, 1943] McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

[Rao and Ballard, 1999] Rao, R. P. and Ballard, D. H. (1999). Predictive coding in the visual cortex: a functional interpretation of some extra-classical receptive-field effects. *Nature neuroscience*, 2(1):79–87.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Two-stream convolutional networks for action recognition in videos. *arXiv preprint arXiv:1406.2199*.

[Song et al., 2020] Song, Y., Lukasiewicz, T., Xu, Z., and Bogacz, R. (2020). Can the brain do backpropagation?—exact implementation of backpropagation in predictive coding networks. *Advances in Neural Information Processing Systems*, 33.

[Van Essen et al., 1992] Van Essen, D. C., Anderson, C. H., and Felleman, D. J. (1992). Information processing in the primate visual system: an integrated systems perspective. *Science*, 255(5043):419–423.

[Zhuang et al., 2020] Zhuang, C., Yan, S., Nayebi, A., Schrimpf, M., Frank, M., DiCarlo, J., and Yamins, D. (2020). Unsupervised neural network models of the ventral visual stream. *bioRxiv*.