Neural Networks and Backpropagation Lecture

Grégoire Payen de La Garanderie

February 4, 2019

1 Introduction

This lecture aims to introduce the mathematical concepts at the core of the backpropagation mechanism that is commonly used to optimise neural networks.

Let us consider a neural network whose last layer is a loss function:

$$\begin{aligned} l\colon \mathbb{R}^n \times \mathbb{R}^m &\to \mathbb{R} \\ \boldsymbol{x}, \boldsymbol{W} &\mapsto l(\boldsymbol{x}, \boldsymbol{W}) \end{aligned}$$

Our neural network l takes an input vector $\boldsymbol{x}$ and a set of weight $\boldsymbol{W}$ and computes the associated loss value. Our goal is to minimise $l(\boldsymbol{x}, \boldsymbol{W})$ *w.r.t.* (with respect to) $\boldsymbol{W}$. In mathematical terms, this is written as:

$$\underset{\boldsymbol{W}}{\text{minimise}} \quad l(\boldsymbol{x}, \boldsymbol{W})$$

Compared to most minimisation problems which feature many constraints, this problem looks deceptively simple however the difficulty of this optimisation resides is the non-linearity of the function l and the large number of dimensions of $\boldsymbol{W}$ (*i.e.* m is a large number, typically a few millions).

The simplest optimisation strategy for a neural network is called *Gradient Descent* and is defined as the following update rule:

$$\boldsymbol{W}_{t+1} \leftarrow \boldsymbol{W}_t - lr \cdot l'_2(\boldsymbol{x}, \boldsymbol{W}_t) \tag{1}$$

where l'_2 in the derivative of l *w.r.t.* $\boldsymbol{W}$ and lr is the learning rate which control the rate of convergence of the algorithm. Figure 1 shows an example of gradient descent along a one-dimensional cost function.

The key question that we will address in this lecture is how to compute the derivative $l'_2(\boldsymbol{x}, \boldsymbol{W})$. This computation is commonly called backpropagation in the deep learning jargon. However this is not a new technique, symbolic automatic differentiation (AD) has been studied since the 1960s, long before they were applied to neural networks in the 1980s. The backpropagation algorithm is a special case of AD that is particularly well suited to large modern deep neural networks.

In the next section, we define function derivatives, both as the refresher of their definition and to define the notation that we use throughout the lecture.

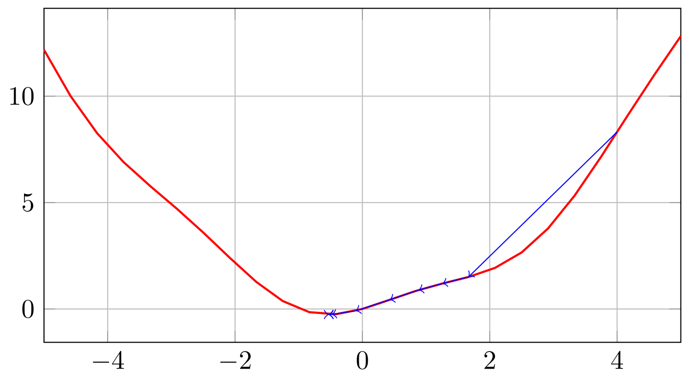

Figure 1: Gradient descent on a one-dimensional function. Red: the cost function to minimise. Blue: Successive gradient updates with $lr = 0.5$.

2 Function Derivative

This section starts by introducing the definition of derivative for a simple real-valued function then extend this definition to vector-valued functions that are typically used in neural networks. We only consider vector-valued functions without loss of generality. Indeed the principles described here for vectors can be trivially generalised to tensors because a tensor of any shape can be flatten into a rank-1 tensor (*i.e.* a vector). We also present the derivative of commonly used functions in deep learning.

Definition 2.1 (Derivative)**.** Let f be a real-valued function:

$$\begin{aligned} f\colon \mathbb{R} &\to \mathbb{R} \\ x &\mapsto f(x) \end{aligned}$$

The derivative of f called f' is a function is defined as:

$$\forall x \in \mathbb{R} \qquad f'(x) = \frac{\mathrm{d}f}{\mathrm{d}x}(x) = \lim_{\delta x \to 0} \frac{f(x+\delta x) - f(x)}{\delta x} \tag{2}$$

Note that this is a very common but slightly confusing and abusive notation. It is important not to forget that the x in $\mathrm{d}x$ or δx is not the same as the x in $f(x)$. An unambiguous equivalent notation would be:

$$\forall a \in \mathbb{R} \qquad f'(a) = \frac{\mathrm{d}f}{\mathrm{d}x}(a) = \lim_{\delta x \to 0} \frac{f(a+\delta x) - f(a)}{\delta x} \tag{3}$$

Example 2.1 (Useful derivatives)**.** Here are some useful derivatives for common activation functions shown in Figure 2:

1. $\mathrm{ReLU}'(x) = \begin{cases} 1, & \text{if } x \geq 0 \\ 0, & \text{otherwise} \end{cases}$

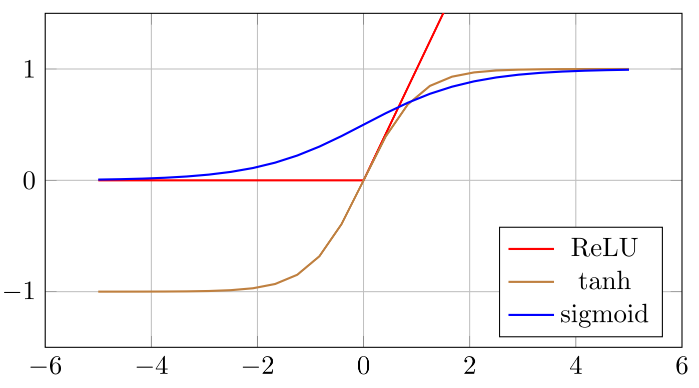

Figure 2: Common activation functions

2. $\tanh'(x) = 1 - \tanh^2(x)$
3. $\text{sigmoid}'(x) = \text{sigmoid}(x) \cdot (1 - \text{sigmoid}(x))$

At first, we extend the definition of a function derivative to functions of two variables in the definition below to introduce the concept of partial derivative. Subsequently we generalise to functions of n variables that are typically used in a neural network.

Definition 2.2 (Partial derivative of a real-valued function of two real-valued variables)**.** Let f be a function of two real-valued variables:

$$\begin{aligned} f\colon \mathbb{R}^2 &\to \mathbb{R} \\ x, y &\mapsto f(x, y) \end{aligned}$$

The function f admits two partial derivatives called $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$ called respectively "partial derivative of f with respect to x" and "partial derivative of f with respect to y" defined as:

$$\forall (x, y) \in \mathbb{R}^2 \qquad \frac{\partial f}{\partial x}(x, y) = \lim_{\delta x \to 0} \frac{f(x + \delta x, y) - f(x, y)}{\delta x} \tag{4a}$$

$$\forall (x, y) \in \mathbb{R}^2 \qquad \frac{\partial f}{\partial y}(x, y) = \lim_{\delta y \to 0} \frac{f(x, y + \delta y) - f(x, y)}{\delta y} \tag{4b}$$

Again, the notation can be confusing and an unambiguous definition would be:

$$\forall (a, b) \in \mathbb{R}^2 \qquad \frac{\partial f}{\partial x}(a, b) = \lim_{\delta x \to 0} \frac{f(a + \delta x, b) - f(a, b)}{\delta x} \tag{5a}$$

$$\forall (a, b) \in \mathbb{R}^2 \qquad \frac{\partial f}{\partial y}(a, b) = \lim_{\delta y \to 0} \frac{f(a, b + \delta y) - f(a, b)}{\delta y} \tag{5b}$$

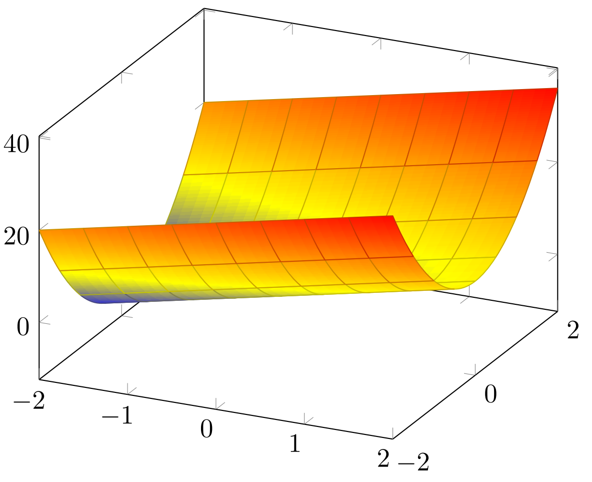

Figure 3: The function $f(x, y) = 4x + 7y^2$

Example 2.2. Let f be $f(x, y) = 4x + 7y^2$ (as shown in Figure 3). Then, we can prove that:

$$\forall (x, y) \in \mathbb{R}^2 \qquad \frac{\partial f}{\partial x}(x, y) = 4 \quad \text{and} \quad \frac{\partial f}{\partial y}(x, y) = 14y \tag{6}$$

In this example, the derivative of f *w.r.t.* x is independent from the value of y and vice-versa. This situation is frequently observed in neural network layers featuring a bias term as we will see later.

We can also evaluate the partial derivatives at a specific point:

$$\frac{\partial f}{\partial x}(2, 3) = 4 \quad \text{and} \quad \frac{\partial f}{\partial y}(2, 3) = 14 \cdot 3 = 42 \tag{7}$$

Example 2.3. Let f be $f(x, y) = x \cdot y$ (as shown in Figure 4). We can prove that:

$$\forall (x, y) \in \mathbb{R}^2 \qquad \frac{\partial f}{\partial x}(x, y) = y \quad \text{and} \quad \frac{\partial f}{\partial y}(x, y) = x \tag{8}$$

This example is an illustration of a simple bilinear function (that is a function that is both linear *w.r.t.* to x and linear *w.r.t.* to y). Many functions used in deep learning (*e.g.* fully-connected layers, convolutional layers) are actually multilinear maps *w.r.t.* their input, weight and bias terms.

We can evaluate the partial derivatives at a specific point:

$$\frac{\partial f}{\partial x}(2, 3) = 3 \quad \text{and} \quad \frac{\partial f}{\partial y}(2, 3) = 2 \tag{9}$$

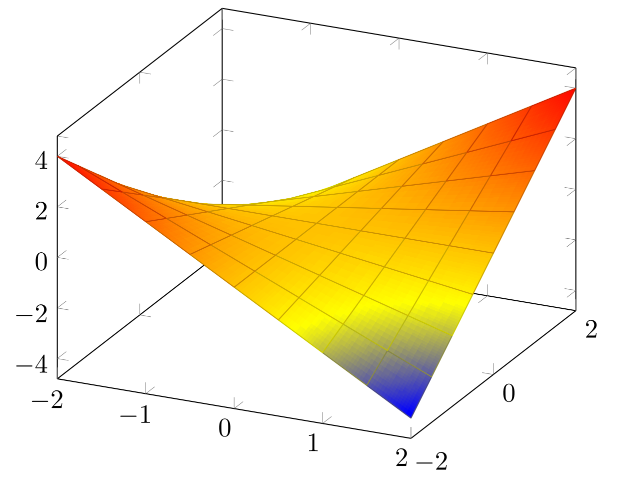

Figure 4: The function $f(x, y) = x \cdot y$

Definition 2.3 (Partial derivative of a real-valued function of n real-valued variables)**.** Let f be a function of n real-valued variables:

$$\begin{aligned} f\colon \mathbb{R}^n &\to \mathbb{R} \\ x_1, \ldots, x_n &\mapsto f(x_1, \ldots, x_n) \end{aligned}$$

The function f admits n partial derivatives called $\frac{\partial f}{\partial x_i}$ for $i \in [1, n]$:

$$\forall (x_1, \ldots, x_n) \in \mathbb{R}^n, i \in [1, n] \qquad \frac{\partial f}{\partial x_i}(x_1, \ldots, x_n) = \lim_{\delta x_i \to 0} \frac{f(x_1, \ldots, x_{i-1}, x_i + \delta x_i, x_{i+1}, \ldots, x_n) - f(x_1, \ldots, x_n)}{\delta x_i} \tag{10}$$

Apart from the verbose notation, this is a straightforward generalisation from a function of 2 variables to a function of n variables.

Example 2.4. Recalling the previous example $f(x, y) = x \cdot y$. We can instead compute a more general form $f(x_1, \ldots, x_n) = \prod_i x_i = x_1 \cdot x_2 \cdot \ldots \cdot x_n$. Then:

$$\forall (x_1, \ldots, x_n) \in \mathbb{R}^n, i \in [1, n] \qquad \frac{\partial f}{\partial x_i}(x_1, \ldots, x_n) = x_1 \times \ldots \times x_{i-1} \times x_{i+1} \times \ldots \times x_n = \prod_{j \neq i} x_j \tag{11}$$

We can evaluate the partial derivatives at a specific point ($n = 3$):

$$\frac{\partial f}{\partial x_1}(3, 7, 4) = 7 \cdot 4 \quad \text{and} \quad \frac{\partial f}{\partial x_2}(3, 7, 4) = 3 \cdot 4 \quad \text{and} \quad \frac{\partial f}{\partial x_2}(3, 7, 4) = 3 \cdot 7 \tag{12}$$

So far, the previous definitions have considered functions of multiple real-valued input variables however they only had a single real-valued output. We can now have a look at functions of multiple inputs and multiple outputs.

Definition 2.4 (Jacobian Matrix)**.** Let f be a function defined as:

$$\begin{aligned} f\colon \mathbb{R}^n &\to \mathbb{R}^m \\ x_1, \dots, x_n &\mapsto (y_1, \dots, y_m) = f(x_1, \dots, x_n) \end{aligned}$$

This function takes multiple values $x_1, \dots, x_n$ as an input and returns multiple values $y_1, \dots, y_n$. For instance, you can think of the multiple x_is as the pixel values of an image and the y_is as output classification probabilities or the feature map of an intermediate layer.

To simplify the notation, we introduce f_i as the i-th component of the output of f, *i.e.* $y_i = f_i(x_1, \dots, x_n)$. There are m such functions, one for each of our m outputs. Each of the component function f_i admits n partial derivatives called $\frac{\partial f_i}{\partial x_j}$. $\frac{\partial f_i}{\partial x_j}$ can be called the partial derivative of the i-th output of f *w.r.t.* the j-th input of f. They are $n \times m$ such partial derivatives for all combinations of inputs and outputs. We can pack all those derivatives in a matrix often referred as the Jacobian of f defined as:

$$\boldsymbol{J}_f = \frac{\mathrm{d}f}{\mathrm{d}\boldsymbol{x}} = \left[\frac{\partial f_i}{\partial x_j}\right] = \begin{bmatrix} \frac{\partial f_1}{\partial x_1} & \frac{\partial f_1}{\partial x_2} & \cdots & \frac{\partial f_1}{\partial x_n} \\ \frac{\partial f_2}{\partial x_1} & \frac{\partial f_2}{\partial x_2} & \cdots & \frac{\partial f_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1} & \frac{\partial f_m}{\partial x_2} & \cdots & \frac{\partial f_m}{\partial x_n} \end{bmatrix} \tag{13}$$

The representation of the derivative of f as a matrix is convenient because it behaves in the same way as the derivative of a real-valued function as shown below.

This function f of multiple inputs and outputs can be represented using a vectorial notation:

$$\forall \boldsymbol{x} \in \mathbb{R}^n, \boldsymbol{y} \in \mathbb{R}^m \qquad \boldsymbol{y} = f(\boldsymbol{x}) \tag{14}$$

where $\boldsymbol{x}$ and $\boldsymbol{y}$ are vectors.

In the next two examples, we show examples of functions commonly used in neural networks and their derivative.

Example 2.5 (Linear Map)**.** A linear map is the mathematical formulation of a fully-connected linear layer in a neural network (where $\boldsymbol{x}$ is the layer input, $\boldsymbol{W}$ the weight matrix and $\boldsymbol{b}$ the bias term). Let f be a linear map defined as:

$$\begin{aligned} f\colon \mathbb{R}^n &\to \mathbb{R}^m \\ \boldsymbol{x} &\mapsto \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \end{aligned}$$

where $\boldsymbol{W}$ is a $m \times n$ matrix and $\boldsymbol{b} \in \mathbb{R}^m$.

The derivative of f (*i.e.* the Jacobian matrix) is:

$$\forall \boldsymbol{x} \in \mathbb{R}^n \qquad f'(\boldsymbol{x}) = \boldsymbol{W} \tag{15}$$

That is, the Jacobian of the function f is the weight matrix $\boldsymbol{W}$. For a linear map, the derivative is constant and does not depend on the value of $\boldsymbol{x}$. This is obviously not always true for all functions.

This example encompasses not only fully-connected layers but convolutional layers as well because convolutions are special form of linear map. The matrix $\boldsymbol{W}$ of a convolution is a Toeplitz matrix defined by the convolution kernel.

Example 2.6 (Softmax)**.** The Softmax function is often used as the last layer of a classification neural network. It normalises an un-normalised vector into a probability distribution (ensuring that the sum of all probabilities is 1). The softmax function is defined as:

$$\begin{aligned} \text{softmax} \colon \mathbb{R}^n &\to \mathbb{R}^n \\ \boldsymbol{x} &\mapsto \boldsymbol{y} = \left(\frac{e^{x_i}}{\sum_j e^{x_j}} \right)_{i \in [1,n]} \end{aligned}$$

We can show that its derivative is:

$$\forall i, j, \boldsymbol{x} \qquad \frac{\partial\, \text{softmax}_i}{\partial \boldsymbol{x}_j}(\boldsymbol{x}) = \text{softmax}_i(\boldsymbol{x}) \cdot (\delta_{ij} - \text{softmax}_j(\boldsymbol{x})) \tag{16}$$

The Jacobian of the softmax function can be defined as:

$$s'(\boldsymbol{x}) = \begin{bmatrix} s_1(\boldsymbol{x})(1 - s_1(\boldsymbol{x})) & -s_1(\boldsymbol{x})s_2(\boldsymbol{x}) & \dots & -s_1(\boldsymbol{x})s_n(\boldsymbol{x}) \\ s_2(\boldsymbol{x})s_1(\boldsymbol{x})) & s_2(\boldsymbol{x})(1 - s_2(\boldsymbol{x})) & & -s_2(\boldsymbol{x})s_n(\boldsymbol{x}) \\ \vdots & & \ddots & \vdots \\ -s_n(\boldsymbol{x})s_1(\boldsymbol{x}) & -s_n(\boldsymbol{x})s_2(\boldsymbol{x}) & \dots & s_n(\boldsymbol{x})(1 - s_n(\boldsymbol{x})) \end{bmatrix} \tag{17}$$

This Jacobian has a common term $-s_i(\boldsymbol{x})s_j(\boldsymbol{x})$ corresponding to the denominator of the softmax function as well as an extra term along the diagonal $s_i(\boldsymbol{x})$ corresponding to the numerator. The softmax function is an example of a non-linear function. As such, the value of the derivative is non-constant and depends on the value of $\boldsymbol{x}$.

3 Function Composition and Derivative

Theorem 3.1 (Chain Rule)**.** Let f and g be two functions, either real-valued or vector-valued and let h be the composition of the two: $h = f \circ g$. That is:

$$\forall x \qquad h(x) = f(g(x)) \tag{18}$$

For example, suppose we have a function that normalises the intensity of an image $f(x)$, but before that we Gaussian blur the image $g(x)$, we have the composite $f(g(x))$.

Then the derivative of h is such that:

$$\forall x \qquad h'(x) = f'(g(x)) \times g'(x) \tag{19}$$

This is sometime written as:

$$\frac{\mathrm{d}h}{\mathrm{d}x} = \frac{\mathrm{d}f}{\mathrm{d}g} \times \frac{\mathrm{d}g}{\mathrm{d}x} \tag{20}$$

This is known as the chain rule. It works with both real-valued and vector-valude functions. However unlike for real-valued functions, the chain-rule is non-commutative for vector-valued function. That is:

$$\begin{array}{lll} \forall \boldsymbol{x} & h'(\boldsymbol{x}) = f'(g(\boldsymbol{x})) \cdot g'(\boldsymbol{x}) & \textbf{Correct} \\ \forall \boldsymbol{x} & h'(\boldsymbol{x}) = g'(\boldsymbol{x}) \cdot f'(g(\boldsymbol{x})) & \textbf{Very wrong} \end{array}$$

If the dimensions of the input and output vectors of g are mismatching, the Jacobian of g will be a rectangular matrix and $g'(\boldsymbol{x}) \cdot f'(g(\boldsymbol{x}))$ will not even be a valid expression. Therefore be extremely careful that you are applying the chain rule in the right order.

Example 3.1 (A simple neural network)**.** Let l be a neural network constituted of three layers f, g, h:

$$l(\boldsymbol{x}) = (f \circ g \circ h)(\boldsymbol{x}) = f(g(h(\boldsymbol{x}))) \tag{22}$$

This network can be represented visually by:

$$\begin{tikzcd} \boldsymbol{x} \arrow[d] \\ h(\boldsymbol{x}) \arrow[d] \\ (g \circ h)(\boldsymbol{x}) \arrow[d] \\ (f \circ g \circ h)(\boldsymbol{x}) \end{tikzcd}$$

The gradient of l with respect to the input $\boldsymbol{x}$ is:

$$l'(\boldsymbol{x}) = f'((g \circ h)(\boldsymbol{x})) \cdot g'(h(x)) \cdot h'(x) \tag{23}$$

We can see in the above equation that the derivative of each layer depends on the input of its layer. For instance, the derivative of the layer f depends on its input $(g \circ h)(\boldsymbol{x})$.

For the sake of simplicity, we will omit the input of each layer in the notation of the gradients and write:

$$l' = f' \cdot g' \cdot h' \tag{24}$$

We can see that the gradient of a sequential neural network is the product of the gradient of each individual layers (according to the chain rule).

This neural network is not particularly realistic because it does not have any weights. Recall the minimisation goal defined in Section 1, we are interested in minimising the loss *w.r.t.* to the weights, not the input itself. In practice, each layer takes an additional weight parameter. To take this into account, we need one more definition.

Definition 3.1 (Function of multiple vectors)**.** Let f be a function of multiple vectors defined as:

$$\begin{aligned} f\colon \mathbb{R}^s \times \mathbb{R}^t &\to \mathbb{R}^u \\ x, y &\mapsto f(x, y) \end{aligned}$$

Analogously to the definition of a partial derivative of a function of multiple real values, this function can be differentiated with respect to each of its input. We will call f'_i the derivative with respect to the i-th input.

Example 3.2 (Multilinear Map)**.** In the Example 2.5, we have looked at the derivative of a linear map with fixed weights and bias. However this is not the case of a neural network where the weight matrix and bias vector are themselves inputs of the map that can be adjusted by the optimiser.

Let f be a multilinear map defined as:

$$\begin{aligned} f\colon \mathbb{R}^n \times \mathbb{R}^{m\times n} \times \mathbb{R}^m &\to \mathbb{R}^m \\ \boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b} &\mapsto \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b} \end{aligned}$$

where $\boldsymbol{W}$ is a $m \times n$ matrix and $\boldsymbol{b} \in \mathbb{R}^m$.

For a given $\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}$, the derivatives of f are:

$$f'_1(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{W} \tag{25a}$$

$$f'_2(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}) = \boldsymbol{x} \tag{25b}$$

$$f'_3(\boldsymbol{x}, \boldsymbol{W}, \boldsymbol{b}) = (1, \dots, 1) \tag{25c}$$

Example 3.3 (A weighted neural network)**.** Let l be a neural network constituted of three layers f, g, h and weights $\boldsymbol{w_f}$, $\boldsymbol{w_g}$, $\boldsymbol{w_h}$:

$$l(\boldsymbol{x}, \boldsymbol{w_f}, \boldsymbol{w_g}, \boldsymbol{w_h}) = f(g(h(\boldsymbol{x}, \boldsymbol{w_h}), \boldsymbol{w_g}), \boldsymbol{w_f}) \tag{26}$$

$$\begin{tikzcd}
\boldsymbol{x} \arrow[d] & \boldsymbol{w_h} \arrow[dl] \\
h(\boldsymbol{x}, \boldsymbol{w_h}) \arrow[d] & \boldsymbol{w_g} \arrow[dl] \\
g(h(\boldsymbol{x}, \boldsymbol{w_f}), \boldsymbol{w_g}) \arrow[d] & \boldsymbol{w_f} \arrow[dl] \\
f(g(h(\boldsymbol{x}, \boldsymbol{w_h}), \boldsymbol{w_g}), \boldsymbol{w_f}) &
\end{tikzcd}$$

The gradient with respect to the input $\boldsymbol{x}$ is:

$$l'_1 = f'_1 \cdot g'_1 \cdot h'_1 \tag{27}$$

However the gradient w.r.t to the input is rarely used in machine learning. Instead the optimiser relies on the gradient of the weights to update each weight at every iteration. Those gradients can also be computed using the chain rule:

1. w.r.t to $\boldsymbol{w_f}$: $l'_2 = f'_2$
2. w.r.t to $\boldsymbol{w_g}$: $l'_3 = f'_1 \cdot g'_2$
3. w.r.t to $\boldsymbol{w_h}$: $l'_4 = f'_1 \cdot g'_1 \cdot h'_2$

As the output of l is a single real number, each Jacobian l'_1, l'_2, l'_3, l'_4 degenerates to a column vector. This validates that we are allowed to use them inside the update rule:

$$\begin{aligned}
\boldsymbol{w_f}_{t+1} &\leftarrow \boldsymbol{w_f}_t - lr \cdot l'_2(\boldsymbol{x}, \boldsymbol{w_f}_t, \boldsymbol{w_g}_t, \boldsymbol{w_h}_t) \\
\boldsymbol{w_g}_{t+1} &\leftarrow \boldsymbol{w_g}_t - lr \cdot l'_3(\boldsymbol{x}, \boldsymbol{w_f}_t, \boldsymbol{w_g}_t, \boldsymbol{w_h}_t) \\
\boldsymbol{w_h}_{t+1} &\leftarrow \boldsymbol{w_h}_t - lr \cdot l'_4(\boldsymbol{x}, \boldsymbol{w_f}_t, \boldsymbol{w_g}_t, \boldsymbol{w_h}_t)
\end{aligned}$$

4 Backpropagation

4.1 Differentiation and Computational Graph

Example 4.1 (Basic neural network computation order)**.** Recall the previous example of a three layers neural network. Let l be a neural network constituted of three layers f, g, h:

$$l(\boldsymbol{x}) = (f \circ g \circ h)(\boldsymbol{x}) = f(g(h(\boldsymbol{x}))) \tag{29}$$

We have shown that the gradient can be computed as:

$$l' = f' \cdot g' \cdot h' \tag{30}$$

Computationally, this gradient can be computed in two different ways:

$$l' = (f' \cdot g') \cdot h' \tag{31a}$$

$$l' = f' \cdot (g' \cdot h') \tag{31b}$$

The first one performs the multiplication between f' and g' first (as shown in Figure 5(a)) while the second performs the multiplication between g' and h' first (as shown in Figure 5(b)). Both of those ways are mathematically equivalent and correct however one of them is not practical from a computational point of view with neural network.

$$\begin{tikzcd}
\boldsymbol{x} \arrow[d] \arrow[dr] & \\
h(\boldsymbol{x}) \arrow[d] \arrow[dr] & h'(\boldsymbol{x}) \arrow[d] \\
(g \circ h)(\boldsymbol{x}) \arrow[d] \arrow[dr] & g'(\ldots) \cdot h'(\boldsymbol{x}) \arrow[d] \\
(f \circ g \circ h)(\boldsymbol{x}) & \boxed{f'(\ldots) \cdot g'(\ldots) \cdot h'(\boldsymbol{x})}
\end{tikzcd}$$

(a) Forward propagation using Equation 31a

$$\begin{tikzcd}
\boldsymbol{x} \arrow[d] \arrow[dr] & \\
h(\boldsymbol{x}) \arrow[d] \arrow[dr] & \boxed{f'(\ldots) \cdot g'(\ldots) \cdot h'(\boldsymbol{x})} \\
(g \circ h)(\boldsymbol{x}) \arrow[d] \arrow[dr] & f'(\ldots) \cdot g'(\ldots) \arrow[u] \\
(f \circ g \circ h)(\boldsymbol{x}) & f'(\ldots) \arrow[u]
\end{tikzcd}$$

(b) Backpropagation using Equation 31b

Figure 5: Computational graphs for forward and backward propagation. The final result in each graph has been framed.

To understand the problem, we need to look at the dimension of each of the layers. During training, the final layer of a neural network (f here) is the loss function which outputs a single real-value therefore its output has a dimension of 1. In contrast the input and the intermediate layers are much larger. For instance, assuming that this network is processing an image of size 1000×1000, this image has $1,0000,000$ pixels therefore the input layer has a dimension of $1,000,000$. Further assuming that the intermediate layers have the same size. We have:

1. $f \colon \mathbb{R}^{1,000,000} \to \mathbb{R}$
2. $g \colon \mathbb{R}^{1,000,000} \to \mathbb{R}^{1,000,000}$
3. $h \colon \mathbb{R}^{1,000,000} \to \mathbb{R}^{1,000,000}$

On one hand, the Jacobian of g is a $1,000,000 \times 1,000,000$ matrix. This matrix would require 1TB of memory. Therefore it is not actually possible to compute and store the Jacobian of g. The Jacobian of h and $g \circ h$ have exactly the same size and are too large to be stored. This means that it is unrealistic to compute $g' \cdot h'$ with modern technologies.

On the other hand, the Jacobian of f is a $1 \times 1,000,000$ matrix. To store this matrix, we would need about 1Mb of memory. The Jacobian of $f \circ g$ is also a $1 \times 1,000,000$ matrix. This is the intuition behind the backpropagation algorithm. It is unrealistic to compute the gradients starting from the input because the intermediate Jacobian matrices would be too large however if we start from the loss value, all the intermediate matrices are kept small.

4.2 Implementation of the Chain Rule

In the previous example, we have shown that it is better to work our way backward, computing f' then $f' \cdot g'$ and finally $f' \cdot g' \cdot h'$. If the intermediate layers are of size $1,000,0000$, each of those derivative is a matrix of size $1 \times 1,000,000$. The simplest way to compute $f' \cdot g'$ is to compute f' and g' and multiply them together however this does not work because the Jacobian g' is also a $1,000,000 \times 1,000,000$ matrix. In practice, the Jacobian of most operations (*e.g.* a convolution) is very sparse and a lot of redundant operations and memory can be saved by computing the derivative g' and the multiplication together.

This leads us to the usual way of defining forward and backward functions in most deep learning frameworks. The forward function takes an input and apply the function to it. The backward function takes the input and the gradient of the loss *w.r.t.* the function's output. The backward function returns the gradient of the loss *w.r.t.* the function's input (that is, by the chain rule, the product of the gradient *w.r.t.* to its output and the Jacobian of the function *w.r.t.* to its input).

Algorithm 1 Sample forward function

```
1: function FORWARD(input)
2:     return $f$(input)
3: end function
```

Algorithm 2 Sample backward function

```
1: function BACKWARD(input,grad_output)
2:     return grad_output $\cdot f'$(input)
3: end function
```

Example 4.2 (Convolution implementation)**.** Let us define a convolution between a 1D input $\boldsymbol{x} \in \mathbb{R}^n$ and a small kernel $\boldsymbol{h} \in \mathbb{R}^3$:

$$\begin{aligned} f \colon \mathbb{R}^n &\to \mathbb{R}^n \\ \boldsymbol{x} &\mapsto \boldsymbol{y} = \left(\sum_{k \in [-1,1]} x_{i+k} h_{k+2} \right)_{i \in [1,\ldots,n]} \end{aligned}$$

In the sum, for the sake of simplicity, we assume that $x_{i+j} = 0$ if $i + j < 1$ or $i + j > n$.

The partial derivatives of f are:

$$\forall i,j \in [1,n] \qquad \frac{\partial f_i}{\partial x_j} = \begin{cases} h_{j-i+2}, & \text{if } j-i \in [-1,1] \\ 0, & \text{otherwise} \end{cases} \tag{32}$$

The Jacobian matrix is a band matrix:

$$\forall \boldsymbol{x} \in \mathbb{R}^n \qquad f'(\boldsymbol{x}) = \begin{bmatrix} h_2 & h_3 & 0 & & \dots & & 0 \\ h_1 & h_2 & h_3 & & & & \\ 0 & h_1 & & \ddots & \ddots & & \vdots \\ & & \ddots & \ddots & \ddots & & \\ \vdots & & \ddots & \ddots & & h_3 & 0 \\ & & & & h_1 & h_2 & h_3 \\ 0 & & \dots & & 0 & h_1 & h_2 \end{bmatrix} \tag{33}$$

This Jacobian is a sparse matrix called a Toeplitz matrix which has mostly zeros everywhere apart from a narrow diagonal band of width 3.

Here is a naive implementation of the backward function which computes the Jacobian first then multiply it with the output gradient:

Algorithm 3 Naive convolution backward function

```
function ConvBackward(input,grad_output)
    #Compute the Jacobian matrix
    $J \leftarrow$ zero matrix of size $n \times n$
    for $i \in [1,n]$ do
        for $j \in [1,n]$ do
            $k \leftarrow j - i$
            if $k \in [-1,1]$ then
                $J_{i,j} \leftarrow h_{k+2}$
            end if
        end for
    end for

    #Multiply the Jacobian with grad_output
    $v \leftarrow$ zero vector of size $n$
    for $i \in [1,n]$ do
        for $j \in [1,n]$ do
            $v_j \leftarrow v_j + \text{grad\_output}_i \cdot J_{i,j}$
        end for
    end for
    return $v$
end function
```

The algorithm has a computational and spatial complexity of $\mathcal{O}(n^2)$. As we have seen, this is not practical for large arrays and images. We would need several terabytes of memory and several years to compute a single iteration. In contrast, we can exploit the sparsity of the Jacobian to reduce the complexity to $\mathcal{O}(n)$ in the following algorithm.

Algorithm 4 Optimised convolution backward function

```
1:  function ConvBackward(input,grad_output)
2:      $v \leftarrow$ zero vector of size $n$
3:      for $k \in [-1, -1]$ do
4:          for $j \in [1, n]$ do
5:              $i \leftarrow j - k$
6:              if $i \in [1, n]$ then
7:                  $v_j \leftarrow v_j + \text{grad\_output}_i \cdot h_{k+2}$
8:              end if
9:          end for
10:     end for
11:     return $v$
12: end function
```

5 Conclusion

We have seen that there are multiple possible algorithms to compute the gradient of a neural network however a naive approach would have terrible computational and spatial complexity. The backpropagation algorithm provides an efficient way of computing the gradients. Besides for most operations, it is a bad idea to directly compute the Jacobian matrix because it can be very large. It is often much more effective to simplify and compute the differentiation of a function and multiplication of the chain rule together.

While it is often unnecessary to understand how backpropagation works to understand the overall architecture of a neural network; it is a requirement in order to implement new layers, choose adequate architecture to keep the memory requirements down as well as avoid numerous pitfalls related to numerical accuracy and convergence.