

Obligatorisk oppgave 1 i MAT1110, V26

Øyvind Ryan

2026-02-12

Table of contents

1	Backpropagation and the chain rule	2
1.1	Introduction	2
1.2	Implementation	8
1.3	Block matrices	11

Denne obligatoriske oppgaven tar for seg en viktig anvendelse av kjerneregelen innen maskinlæring. Det er mye tekst og mange oppgaver her, og det er frivillig hvor mye du gjør og leser - det eneste som er obligatorisk er at du faktisk gjør en innlevering i canvas (fristen er 26. februar kl. 1430). Du får tilbakemelding fra gruppelærer på det du leverer inn, og alle innleveringer blir godkjent.

Legg merke til at de fleste av oppgavene i denne teksten er betydelig enklere å løse, sammenlignet med det å lese og forstå hele teksten (som er mer vanskelig enn stoffet ellers i kurset, da det dukker opp en del tekniske ting).

Vi anbefaler derfor at du starter med å løse noen av oppgavene, spesielt oppgave 1.1, 1.2, 1.3, 1.6, 1.7 (i denne rekkefølgen). Hvis dette går greit kan du forsøke å lese mer av teksten, samt se på de resterende oppgavene (1.4, 1.5, 1.8). Hvis også dette går greit kan du forsøke å lese resten av teksten.

Oppgaven er på engelsk. Du bestemmer selv om du skriver på engelsk eller norsk.

På kurssidene kan du finne en jupyter notebook for pythonkoden i denne obligatoriske oppgaven, der du kan kjøre og eksperimentere med koden i oppgaven. Denne koden er noe mer avansert enn det dere støtte på i IN1900.

1 Backpropagation and the chain rule

1.1 Introduction

In machine learning one tries to find a function which approximates a given set of measurement data in a best possible way. For this to be possible computationally, we require that the candidate functions have a given form. If the measurement data are $\mathbf{x}_i \in \mathbb{R}^n$, $\mathbf{y}_i \in \mathbb{R}^n$, we would like to find an $\mathbf{F}$ on the given form so that “the total error”

$$\sum_{i=1}^m |\mathbf{F}(\mathbf{x}_i) - \mathbf{y}_i|^2$$

in the measurement data is as small as possible. We assume that the functions on the given form are uniquely described by a set of *parameters* $w_1, w_2, \dots$, and we then write

$$\mathbf{F}_{w_1, w_2, \dots} : \mathbb{R}^n \rightarrow \mathbb{R}^n$$

for $\mathbf{F}$. Our question becomes: Which choice of parameters gives a function $\mathbf{F}_{w_1, w_2, \dots}$ with total error

$$f(w_1, w_2, \dots) = \sum_{i=1}^m |\mathbf{F}_{w_1, w_2, \dots}(\mathbf{x}_i) - \mathbf{y}_i|^2. \quad (1)$$

as small as possible? This is a minimisation problem where $w_1, w_2, \dots$ are the variables (not $\mathbf{x}_i$ and $\mathbf{y}_i$, these are given measurement data!). The f in Equation 1 is also called the *loss function*.

Finding the $w_1, w_2, \dots$ which give the minimum in Equation 1 turns out to be difficult. To find good approximations to this minimum can be simpler: From analysis in several variables we know that the *gradient* of a function $f : \mathbb{R}^n \rightarrow \mathbb{R}$,

$$\nabla f = \left(\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, \dots, \frac{\partial f}{\partial x_n} \right),$$

points in the direction where f increases the most. Therefore, the function will decrease the most if we move in the direction of the negative gradient, and this is exactly what we will do: We compute the gradient of f given by Equation 1, and follow this a little, to reach a point where f has a smaller value.

We will require our candidate function to be on the following form.

$$\mathbf{F}_{\mathbf{w}^{(1)}, \mathbf{b}^{(1)}, \mathbf{w}^{(2)}, \mathbf{b}^{(2)}, \dots} = \mathbf{F}_1(\mathbf{F}_2(\dots \mathbf{F}_r(\mathbf{x}))) \quad (2)$$

where all $\mathbf{w}^{(k)}$, $\mathbf{b}^{(k)}$ are vectors of length n , and where

$$\mathbf{F}_k(\mathbf{x}) = \sigma(W^{(k)}\mathbf{x} + \mathbf{b}^{(k)}),$$

with

$$W^{(k)} = \begin{pmatrix} w_1^{(k)} & w_n^{(k)} & w_{n-1}^{(k)} & \cdots & w_2^{(k)} \\ w_2^{(k)} & w_1^{(k)} & w_n^{(k)} & \cdots & w_3^{(k)} \\ w_3^{(k)} & w_2^{(k)} & w_1^{(k)} & \cdots & w_4^{(k)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_n^{(k)} & w_{n-1}^{(k)} & w_{n-2}^{(k)} & \cdots & w_1^{(k)} \end{pmatrix}. \quad (3)$$

Some comments are in order:

- $W^{(k)}$ is a special type of matrix which is constant along diagonals. Such matrices are called *circulant Toeplitz matrices*, and have a series of nice properties which can be useful in machine learning. There also exist effective algorithms to multiply with such matrices, something which is exploited in the function `cconv`, to be defined later. `cconv` takes two vectors $\mathbf{w}$ and $\mathbf{x}$ as input, and returns $W\mathbf{x}$, where W is the circulant Toeplitz matrix with first column being $\mathbf{w}$.
- σ above is the (component-wise) *activation function*. This is first defined as a map from $\mathbb{R}$ to $\mathbb{R}$ by

$$\sigma(x) = 1/(1 + e^{-x}), \quad (4)$$

and extended component-wise to a map from $\mathbb{R}^n$ to $\mathbb{R}^n$ by $\sigma \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} = \begin{pmatrix} \sigma(x_1) \\ \sigma(x_2) \\ \vdots \\ \sigma(x_n) \end{pmatrix}$.

- Equation 2 is also called a *Neural Network* (NN), and the $\mathbf{F}_k$'s are called its *layers*. We see that the parameters of the candidate functions are split in groups, as dictated by the vectors $\mathbf{w}^{(i)}$, $\mathbf{b}^{(i)}$. When the $W^{(i)}$ have the form in Equation 3 (as we require), the neural network is called a *Convolutional Neural Network* (CNN). In a general neural network the $W^{(i)}$ can have other forms as well, and it can be the case that some of the $W^{(i)}$ are circulant Toeplitz matrices, while others are not.

Since Equation 2 is a composition of many functions, we must use the *chain rule* in several variables to find the gradient of Equation 1. For this we need to find the *Jacobians* of the “inner functions” in Equation 2. The Jacobian of $\mathbf{F}(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}^m$, denoted $\mathbf{F}'(\mathbf{x})$, is defined as the $m \times n$ -matrix

$$\begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_m}{\partial x_1} & \frac{\partial F_m}{\partial x_2} & \cdots & \frac{\partial F_m}{\partial x_n} \end{pmatrix}$$

where $F_1, F_2, \dots, F_m$ are the component functions of $\mathbf{F}$. The chain rule can now be formulated as follows..

Theorem 1.1. *Let $\mathbf{F} : \mathbb{R}^k \rightarrow \mathbb{R}^m$, $\mathbf{G} : \mathbb{R}^n \rightarrow \mathbb{R}^k$ be differentiable functions. Then the composite function $\mathbf{H}(\mathbf{x}) = \mathbf{F}(\mathbf{G}(\mathbf{x}))$ is also a differentiable function, and*

$$\mathbf{H}'(\mathbf{x}) = \mathbf{F}'(\mathbf{G}(\mathbf{x}))\mathbf{G}'(\mathbf{x}).$$

We thus obtain the Jacobian of a composite function by multiplying the Jacobians of the involved functions¹. In particular we need the Jacobian of the activation function, so let us start with this.

Exercise 1.1. Recall the definition of the activation function σ in Equation 4. Show that $\sigma'(x) = \sigma(x)(1 - \sigma(x))$.

The component-wise extension of the activation function thus has the diagonal matrix with $\sigma'(x_i)$ on the diagonal as Jacobian.

Given measurement data $\mathbf{x}_i \in \mathbb{R}^n$, $\mathbf{y}_i \in \mathbb{R}^n$, we would like to find vectors $\mathbf{w}^{(i)}$, $\mathbf{b}^{(i)}$ that gives a neural network (Equation 2) with minimum total error

$$f(\mathbf{w}^{(1)}, \mathbf{b}^{(1)}, \mathbf{w}^{(2)}, \mathbf{b}^{(2)}, \dots) = \sum_{i=1}^m |\mathbf{F}_{\mathbf{w}^{(1)}, \mathbf{b}^{(1)}, \mathbf{w}^{(2)}, \mathbf{b}^{(2)}, \dots}(\mathbf{x}_i) - \mathbf{y}_i|^2.$$

When we compute the gradient of this we will write $\mathbf{F}_1 \circ \mathbf{F}_2$ for the composite function $\mathbf{x} \rightarrow \mathbf{F}_1(\mathbf{F}_2(\mathbf{x}))$, and more generally $\mathbf{F}_1 \circ \dots \circ \mathbf{F}_r$ for the composite function $\mathbf{x} \rightarrow \mathbf{F}_1(\mathbf{F}_2(\dots \mathbf{F}_r(\mathbf{x})))$.

Let us now look at the individual layers. To apply the chain rule correctly we need to set up the points where we evaluate the Jacobians. For this we define

$$\begin{aligned} \mathbf{z}^{(2r)} &= \mathbf{x} \\ \mathbf{z}^{(2k-1)} &= W^{(k)} \mathbf{z}^{(2k)} + \mathbf{b}^{(k)} \quad 1 \leq k \leq r \\ \mathbf{z}^{(2k-2)} &= \sigma(\mathbf{z}^{(2k-1)}) \quad 1 \leq k \leq r. \end{aligned} \tag{5}$$

The first layer can be written

$$\mathbf{F}_r : \begin{pmatrix} \mathbf{w}^{(1)} \\ \mathbf{b}^{(1)} \\ \vdots \\ \mathbf{w}^{(r-1)} \\ \mathbf{b}^{(r-1)} \\ \mathbf{w}^{(r)} \\ \mathbf{b}^{(r)} \end{pmatrix} \rightarrow \begin{pmatrix} \mathbf{w}^{(1)} \\ \mathbf{b}^{(1)} \\ \vdots \\ \mathbf{w}^{(r-1)} \\ \mathbf{b}^{(r-1)} \\ \sigma(W^{(r)} \mathbf{x} + \mathbf{b}^{(r)}) \end{pmatrix},$$

We have here re-defined $\mathbf{F}_r$ in the sense that the variables, with the exception of $\mathbf{w}^{(r)}, \mathbf{b}^{(r)}$, are forwarded to the next layer. We continue like this, so that we eliminate the $\mathbf{w}$ and $\mathbf{b}$ -variables, layer by layer. Let us find the Jacobian $\mathbf{F}'_r$, and let us start with the last n component functions, which have the form

$$(\mathbf{w}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{w}^{(r)}, \mathbf{b}^{(r)}) \rightarrow \sigma(\mathbf{G}(\mathbf{w}^{(r)}, \mathbf{b}^{(r)}))$$

with $\mathbf{G}(\mathbf{w}^{(r)}, \mathbf{b}^{(r)}) = W^{(r)} \mathbf{x} + \mathbf{b}^{(r)}$.

¹The requirement of continuous partial derivative is actually stricter than what we need

Exercise 1.2.

- a. Let w_i, b_i be real numbers, and set $\mathbf{G}(\mathbf{x}) = \begin{pmatrix} w_1 & w_3 & w_2 \\ w_2 & w_1 & w_3 \\ w_3 & w_2 & w_1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix}$. Show that

$$\mathbf{G}'(\mathbf{x}) = \begin{pmatrix} w_1 & w_3 & w_2 \\ w_2 & w_1 & w_3 \\ w_3 & w_2 & w_1 \end{pmatrix}.$$

- b. Let W be any $n \times n$ matrix, and $\mathbf{b} \in \mathbb{R}^n$, and set $\mathbf{G}(\mathbf{x}) = W\mathbf{x} + \mathbf{b}$. Show that $\mathbf{G}'(\mathbf{x}) = W$.

Let instead $\mathbf{G} : \mathbb{R}^{2n} \rightarrow \mathbb{R}^n$ be defined by

$$\mathbf{G}(w_1, \dots, w_n, b_1, \dots, b_n) = \begin{pmatrix} w_1 & w_n & w_{n-1} & \cdots & w_2 \\ w_2 & w_1 & w_n & \cdots & w_3 \\ w_3 & w_2 & w_1 & \cdots & w_4 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_n & w_{n-1} & w_{n-2} & \cdots & w_1 \end{pmatrix} \mathbf{x} + \mathbf{b}.$$

In other words, we now consider w_i and b_i as variables, and $\mathbf{x}$ as a constant.

Exercise 1.3. Show that $\mathbf{G}(w_1, \dots, w_n, b_1, \dots, b_n)$ also can be written as

$$\left(\begin{array}{ccccc|ccccc} x_1 & x_n & x_{n-1} & \cdots & x_2 & 1 & 0 & 0 & \cdots & 0 \\ x_2 & x_1 & x_n & \cdots & x_3 & 0 & 1 & 0 & \cdots & 0 \\ x_3 & x_2 & x_1 & \cdots & x_4 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & x_{n-1} & x_{n-2} & \cdots & x_1 & 0 & 0 & 0 & \cdots & 1 \end{array} \right) \begin{pmatrix} w_1 \\ \vdots \\ w_n \\ b_1 \\ \vdots \\ b_n \end{pmatrix}$$

Hint: Compute and compare the i 'th components of

$$\begin{pmatrix} w_1 & w_n & w_{n-1} & \cdots & w_2 \\ w_2 & w_1 & w_n & \cdots & w_3 \\ w_3 & w_2 & w_1 & \cdots & w_4 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ w_n & w_{n-1} & w_{n-2} & \cdots & w_1 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_n \end{pmatrix} \text{ and } \begin{pmatrix} x_1 & x_n & x_{n-1} & \cdots & x_2 \\ x_2 & x_1 & x_n & \cdots & x_3 \\ x_3 & x_2 & x_1 & \cdots & x_4 \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & x_{n-1} & x_{n-2} & \cdots & x_1 \end{pmatrix} \begin{pmatrix} w_1 \\ w_2 \\ w_3 \\ \vdots \\ w_n \end{pmatrix}$$

It follows from Exercise 1.2 that $\mathbf{G}(w_1, \dots, w_n, b_1, \dots, b_n)$ is linear with Jacobi matrix

$$\mathbf{G}'(w_1, \dots, w_n, b_1, \dots, b_n) = \left(\begin{array}{ccccc|ccccc} x_1 & x_n & x_{n-1} & \cdots & x_2 & 1 & 0 & 0 & \cdots & 0 \\ x_2 & x_1 & x_n & \cdots & x_3 & 0 & 1 & 0 & \cdots & 0 \\ x_3 & x_2 & x_1 & \cdots & x_4 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & x_{n-1} & x_{n-2} & \cdots & x_1 & 0 & 0 & 0 & \cdots & 1 \end{array} \right)$$

For the last part of this project we need the concept of *block matrices*. We give an overview of this in Section 1.3.

Since $\sigma(W^{(r)}\mathbf{x} + \mathbf{b}^{(r)})$ does not depend on $\mathbf{w}^{(1)}, \mathbf{b}^{(1)}, \dots, \mathbf{w}^{(r-1)}, \mathbf{b}^{(r-1)}$, the last n component functions in $\mathbf{F}_r$ will therefore have Jacobian $\left(O \mid U^{(r)} \right)$, where

$$U^{(r)} := \sigma'(\mathbf{z}^{(2r-1)}) \left(\begin{array}{cccc|cccc} x_1 & x_n & \cdots & x_2 & 1 & 0 & \cdots & 0 \\ x_2 & x_1 & \cdots & x_3 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ x_n & x_{n-1} & \cdots & x_1 & 0 & 0 & \cdots & 1 \end{array} \right), \quad (6)$$

which is an $n \times n$ diagonal matrix multiplied with a $n \times (2n)$ matrix (here we have used the block matrix notation from Section 1.3). Since the first component functions are simply the identity mapping, we also get

$$\mathbf{F}'_r = \left(\begin{array}{c|c} I & O \\ \hline O & U^{(r)} \end{array} \right), \quad (7)$$

In the same way we write the other layers $\mathbf{F}_1, \dots, \mathbf{F}_{r-1}$ as

$$\mathbf{F}_k : \begin{pmatrix} \mathbf{w}^{(1)} \\ \mathbf{b}^{(1)} \\ \vdots \\ \mathbf{w}^{(k-1)} \\ \mathbf{b}^{(k-1)} \\ \mathbf{w}^{(k)} \\ \mathbf{b}^{(k)} \\ \mathbf{x} \end{pmatrix} \rightarrow \begin{pmatrix} \mathbf{w}^{(1)} \\ \mathbf{b}^{(1)} \\ \vdots \\ \mathbf{w}^{(k-1)} \\ \mathbf{b}^{(k-1)} \\ \mathbf{b}^{(k-1)} \\ \sigma(W^{(k)}\mathbf{x} + \mathbf{b}^{(k)}) \end{pmatrix},$$

where $1 \leq k \leq r-1$. The only difference from the above is that there are n extra variables x_i here (we shall evaluate in $\mathbf{x} = \mathbf{z}^{(2k-1)}$). This means that the only difference for $\mathbf{F}'_k$, when comparing with $\mathbf{F}'_r$ in Equation 7, is that we get extra columns $\left(\begin{array}{c} O \\ V^{(k)} \end{array} \right)$ on the end, where

$$V^{(k)} := \sigma'(\mathbf{z}^{(2k-1)}) \left(\begin{array}{cccc} w_1^{(k)} & w_n^{(k)} & \cdots & w_2^{(k)} \\ w_2^{(k)} & w_1^{(k)} & \cdots & w_3^{(k)} \\ \vdots & \vdots & \vdots & \vdots \\ w_n^{(k)} & w_{n-1}^{(k)} & \cdots & w_1^{(k)} \end{array} \right) \quad (8)$$

(this is the Jacobian of $\mathbf{x} \rightarrow \sigma(W^{(k)}\mathbf{x} + \mathbf{b})$). The other columns are deduced in the same way as Equation 6, giving $\left(\begin{array}{c|c} I & O \\ \hline O & U^{(k)} \end{array} \right)$, where

$$U^{(k)} := \sigma'(\mathbf{z}^{(2k-1)}) \left(\begin{array}{cccc|cccc} z_1^{(2k)} & z_n^{(2k)} & \cdots & z_2^{(2k)} & 1 & 0 & \cdots & 0 \\ z_2^{(2k)} & z_1^{(2k)} & \cdots & z_3^{(2k)} & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ z_n^{(2k)} & z_{n-1}^{(2k)} & \cdots & z_1^{(2k)} & 0 & 0 & \cdots & 1 \end{array} \right) \quad (9)$$

($\mathbf{x}$ in Equation 6 is exchanged with $\mathbf{z}^{(2k)}$). In other words we get

$$\mathbf{F}'_k = \left(\begin{array}{c|c|c} I & O & O \\ \hline O & U^{(k)} & V^{(k)} \end{array} \right).$$

Now that we have the Jacobian of each individual layer, we can use the chain rule to find the Jacobian of the entire neural network. If we first apply the chain rule to find the Jacobian of $\mathbf{F}_{r-1} \circ \mathbf{F}_r$ we get

$$\left(\begin{array}{c|c|c} I_1 & O & O \\ \hline O & U^{(r-1)} & V^{(r-1)} \end{array} \right) \left(\begin{array}{c|c|c} I_1 & O & O \\ \hline O & I_2 & O \\ \hline O & O & U^{(r)} \end{array} \right). \quad (10)$$

Here we split the identity matrix I into $\left(\begin{array}{c|c} I_1 & O \\ \hline O & I_2 \end{array} \right)$, for the rows to match the columns in the preceding matrix.

Exercise 1.4. Show that Equation 10 equals

$$\left(\begin{array}{c|c|c} I_1 & O & O \\ \hline O & U^{(r-1)} & V^{(r-1)}U^{(r)} \end{array} \right)$$

The Jacobian of $\mathbf{F}_{r-2} \circ \mathbf{F}_{r-1} \circ \mathbf{F}_r$ now becomes

$$\left(\begin{array}{c|c|c} I_1 & O & O \\ \hline O & U^{(r-2)} & V^{(r-2)} \end{array} \right) \left(\begin{array}{c|c|c|c} I_1 & O & O & O \\ \hline O & I_2 & O & O \\ \hline O & O & U^{(r-1)} & V^{(r-1)}U^{(r)} \end{array} \right), \quad (11)$$

Exercise 1.5. Show that Equation 11 becomes

$$\left(\begin{array}{c|c|c|c} I_1 & O & O & O \\ \hline O & U^{(r-2)} & V^{(r-2)}U^{(r-1)} & V^{(r-2)}V^{(r-1)}U^{(r)} \end{array} \right)$$

If we continue this process to include all layers, we get that the k 'th part in the end becomes.

$$V^{(1)} \dots V^{(k-1)}U^{(k)}. \quad (12)$$

What remains now is to also apply the Jacobian (=gradient) of the outer function.

Exercise 1.6.

a. Let $g(\mathbf{x}) = |\mathbf{H}(\mathbf{x}) - \mathbf{y}|^2$, where $\mathbf{H}$ is a differentiable function, and $\mathbf{y} \in \mathbb{R}^n$. Show that

$$\nabla g(\mathbf{x}) = 2(\mathbf{H}(\mathbf{x}) - \mathbf{y})^T \mathbf{H}'(\mathbf{x}).$$

- b. Let $g(\mathbf{x}) = \sum_{i=1}^m |\mathbf{H}_i(\mathbf{x}) - \mathbf{y}_i|^2$, where the $\mathbf{H}_i$ are differentiable functions, and $\mathbf{y}_i \in \mathbb{R}^n$. Show that

$$\nabla g(\mathbf{x}) = \sum_{i=1}^m 2(\mathbf{H}_i(\mathbf{x}) - \mathbf{y}_i)^T \mathbf{H}'_i(\mathbf{x}).$$

If we combine Equation 12 with Exercise 1.6, the k 'th part of the gradient for the whole neural network becomes (in the case of one measurement point)

$$2(\mathbf{z}_0 - \mathbf{y})^T V^{(1)} \dots V^{(k-1)} U^{(k)}. \quad (13)$$

The first half of this matrix are the partial derivatives for the $w_i^{(k)}$ -variables, the second half those of the $b_i^{(k)}$ variables.

1.2 Implementation

Let us take a look at how all this can be implemented. The algorithm for following the gradient to get closer to a minimum of the loss function is called *backpropagation*. The algorithm takes the measurements $\mathbf{x}_i$ and $\mathbf{y}_i$, the number of layers r , and the number of iterations `its`, as input, and returns the updated parameters $\mathbf{w}^{(i)}$ and $\mathbf{b}^{(i)}$, after this number of iterations. As before we assume that all matrices are $n \times n$, all vectors have length n , and that there are m measurement points, which we will randomly generate.

We start by importing the packages we need:

```
import numpy as np
```

We also need functions that implement the activation function, as well as multiplication with its Jacobian, and a function that implements circular convolution.

```
def sigma(x):
    return 1/(1+np.exp(-x))

def sigmader(x):
    return sigma(x)*(1-sigma(x))

def cconv(x, y):
    X = np.fft.fft(x)
    Y = np.fft.fft(y)
    Z = X*Y
    return np.fft.ifft(Z).real
```

We also need code that computes the current value of the lossfunction:

```

def loss(x, y, w, b):
    n, m = np.shape(x)
    n, r = np.shape(w)
    loss = 0
    for t in range(m):          # Iterate through the measurements
        z = np.zeros(n)
        for k in range(r,0,-1): # iterate through the layers
            z = cconv(w[:,k-1], z) + b[:,k-1]
            z = sigma(z)
        loss += sum(abs(z-y[:,t])**2)
    return loss

```

We now have all we need to implement the backpropagation function:

```

def backpropagation(x, y, r, its):
    valp = 0
    lrate = 0.01 # Learning rate
    n, m = np.shape(x)
    w = np.random.rand(n, r) # Randomly generated initial parameters
    b = np.random.rand(n, r) # Randomly generated initial parameters
    for it in range(its):
        dw = np.zeros_like(w)
        db = np.zeros_like(b)
        for t in range(m):
            z = np.zeros((n,2*r+1))
            z[:,2*r] = x[:,t]
            for k in range(r,0,-1):
                z[:, 2*k-1] = cconv(w[:,k-1], z[:,2*k]) + b[:,k-1]
                z[:, 2*k-2] = sigma(z[:,2*k-1])

            temp = 2*(z[:,0]-y[:,t])
            for k in range(1,r+1):
                temp = temp*sigmader(z[:,2*k-1])
                dw[:,k-1] += cconv( np.flip(z[:,2*k]), temp)
                db[:,k-1] += temp
                temp = cconv( np.flip(w[:,k-1]), temp)
        w = w - lrate*dw
        b = b - lrate*db
        print(loss(x, y, w, b))
    return w, b

```

This code requires several comments. The implementation has randomly generated initial

values for the weights $\mathbf{w}^{(k)}$ and $\mathbf{b}^{(k)}$. These are stored in $n \times r$ -matrices, where each column represents a layer. The outer **for**-loop in the code goes through the iterations. Each iteration follows the gradient a small bit from the current point to reach a new point. In the next iteration the gradient at the new point is computed, and this is followed a little bit from the new point, and so on. The update of the point by following the gradient a small bit is captured by the two last lines in the outermost **for**-loop. There the *hyperparameter* `lr` is seen, which captures the distance to follow the gradient. In machine learning this is also called the *learning rate*, and the algorithm is highly sensitive to this parameter: Following the gradient for a little while corresponds to using a first order approximation to the function. But this approximation is good only near the point we evaluate in. We should therefore not follow the gradient too far. We should not follow it too short either - that would render the algorithm ineffective. In practice a compromise is struck. An algorithm can also set the learning rate adaptively, as some points may allow for following the gradient longer than others.

The next **for**-loop goes through the m data points. For each of them the vectors $\mathbf{z}^{(k)}$ in Equation 5 are computed. This is called the *forward pass* of the algorithm, and is done by applying the activation function and multiplication with $W^{(k)}$ in alternating order (line 14-15). Once this has been done, the expression Equation 13 for the different parts of the gradient is computed. This is also called the *backward pass* of the algorithm since, as we will see, the order of the computations will be reversed when compared to the forward pass. Let us look at why this is the case.

Exercise 1.7. Assume we would like to compute the expression $\mathbf{x}^T A_1 A_2 \cdots A_n$ on a computer, where $\mathbf{x} \in \mathbb{R}^n$ and the A_i are $n \times n$ -matrices. Why is it smart to compute this expression from left to right, rather than from right to left.

In terms of complexity it is preferable to compute the quantities from left to right, i.e., in order opposite to the forward pass. When computing Equation 13 from left to right, $2(\mathbf{z}_0 - \mathbf{y})^T V^{(1)} \cdots V^{(k)}$ can be obtained from $2(\mathbf{z}_0 - \mathbf{y})^T V^{(1)} \cdots V^{(k-1)}$ by multiplying with $V^{(k)}$ to the right. Our implementation exploits this by storing the $2(\mathbf{z}_0 - \mathbf{y})^T V^{(1)} \cdots V^{(k)}$ in the temporary variable `temp`. According to Equation 8, this can be done by

- first multiplying with $\sigma'(\mathbf{z}^{(2k-1)})$ to the right (first line in the innermost **for**-loop),
- then multiplying with $W^{(k)}$ to the right (circular convolution with the reverse of $\mathbf{w}^{(k)}$. Last line in the innermost **for**-loop).

To get the partial derivatives w.r.t. the $\mathbf{w}^{(k)}$ and the $\mathbf{b}^{(k)}$ (the k 'th part of the gradient), we multiply with $U^{(k)}$ to the right. According to Equation 9 this can be done by first multiplying with $\sigma'(\mathbf{z}^{(2k-1)})$ to the right (first line in the innermost **for**-loop). After this,

- storing this gives partial derivatives w.r.t. the $\mathbf{b}^{(k)}$ (third line in the innermost **for**-loop),
- circular convolution with the reverse of $\mathbf{z}^{(2k)}$ gives partial derivatives w.r.t. the $\mathbf{w}^{(k)}$ (second line in the innermost **for**-loop),

After the innermost `for`-loop the gradient at the current point (captured by the variables `dw` and `db`) has been updated with the contribution from one more data point.

Code for invoking backpropagation can now be as follows.

```
m = 10    # Number of data points
n = 10    # dimension of the data
its = 10   # Number of iterations
r = 10    # Number of layers

x = np.random.rand(n,m) # Randomly generated measurements
y = np.random.rand(n,m) # Randomly generated measurements

w, b = backpropagation(x, y, r, its)
```

```
28.34182645953592
27.817693818045484
27.034113240230276
25.77890099152559
23.598003381886258
19.5564221725827
12.857791682497254
8.173844551601723
7.964980053850136
7.926560587008899
```

Exercise 1.8. If you run the code you will see that the loss function usually decreases in the iterations. Is this always the case? If you increase the learning rate to 0.1, will the loss function decrease when you run the code? If the loss function increases in some iterations, can you think of a way to adjust the learning rate so that the loss function always decreases?

1.3 Block matrices

If we have four matrices A , B , C , D , we write

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right) \quad (14)$$

for the matrix where the components in these four matrices are placed over, under, to the left, or to the right of each other as indicated. Lines are used to separate the blocks. We call this a *block matrix*. Equation 14 only gives meaning if

- A and B have equally many rows,

- C and D have equally many rows,

for then these can be put next to each other. In the same way A and C must have equally many columns, B and D equally many columns. As an example, if

$$A = \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \quad B = \begin{pmatrix} 2 & 2 \\ 2 & 2 \end{pmatrix} \quad C = \begin{pmatrix} 3 & 3 & 3 \\ 3 & 3 & 3 \\ 3 & 3 & 3 \end{pmatrix} \quad D = \begin{pmatrix} 4 & 4 \\ 4 & 4 \\ 4 & 4 \end{pmatrix}$$

then

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right) = \begin{pmatrix} 1 & 1 & 1 & 2 & 2 \\ 1 & 1 & 1 & 2 & 2 \\ 3 & 3 & 3 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 \\ 3 & 3 & 3 & 4 & 4 \end{pmatrix}$$

It is easy to generalise to block matrices with more than four blocks. The product of two block matrices $\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right)$ and $\left(\begin{array}{c|c} E & F \\ \hline G & H \end{array} \right)$ give meaning if

- the number of columns in A equals the number of rows in E ,
- the number of columns in B equals the number of rows in G .

Block matrices can then be multiplied the way we are used to for ordinary matrices:

$$\left(\begin{array}{c|c} A & B \\ \hline C & D \end{array} \right) \left(\begin{array}{c|c} E & F \\ \hline G & H \end{array} \right) = \left(\begin{array}{c|c} AE + BG & AF + BH \\ \hline CE + DG & CF + DH \end{array} \right)$$