Department of Mathematics

MM7024 - Linear Algebra and Learning from Data

Prof. Zhaojun Bai Prof. Yishao Zhou Mr. Sebastian Fodor

Project 3:
Essentials of Deep Learning

Author: Joris LIMONIER

joris.limonier.001@student.uni.lu

Due date: October 29, 2020

Contents

List of Figures

Listings

Introduction

In this project we explore the fundamentals of Deep Learning. We use backpropagation alongside with Stochastic Gradient Descent (SGD) to work on an artificial neural network.
First we dive into the theory of backpropagation and SGD, then we implement our own neural network from scratch, playing with the number of neurons per layer, the number of layer, the learning rate and the activation function to see how it affects the end result.
The exploration is done on a toy example where we make up two categories of data and then use a neural network to classify them.
The code is in Python throughout the whole project.

1 Task 1

1.1 Back propagation

Let us define the notation for our network.

- $L \in \mathbb{N}$ is the number of layers of our network (including input and output), each of which have one neuron.
- $w^{(l) \in \mathbb{R}}$ is the weight from the neuron in the $(l-1)^{th}$ layer to the neuron in the l^{th} layer.
- $\sigma : x \mapsto \frac{1}{1-e^{-x}}$ is the activation function (sigmoid).
- $x \in \mathbb{R}$ is the input.
- $z^{(l)} := w^{(l)} a^{(l-1)} + b^{(l)} \in \mathbb{R}$ is the weighted input for the neuron at layer l.
- $a^{(l)} := \begin{cases} x & l = 1 \\ \sigma\left(z^{(l)}\right) & l = 2, \dots, L-1 \end{cases} \in \mathbb{R}$
- $C := \frac{1}{2}\left(y - a^{(L)}\right)^2 \in \mathbb{R}$ is the cost function (*i.e.* loss function).
- η is the learning rate

First we want to compute $\frac{dC}{dw^{(L)}}$ but we cannot do it directly. Indeed, we have to take into consideration that C depends on $a^{(L)}$, which in turn depends on $z^{(L)}$, which finally depends on $w^{(L)}$. Similarly we cannot compute $\frac{dC}{db^{(L)}}$ directly because we have that C depends on $a^{(L)}$, which depends on $z^{(L)}$, which depends on $b^{(L)}$. Using the chain rule, we get the following

$$\begin{cases} \frac{dC}{dw^{(L)}} = \frac{dC}{da^{(L)}} \cdot \frac{da^{(L)}}{dz^{(L)}} \cdot \frac{dz^{(L)}}{dw^{(L)}} \\ \\ \frac{dC}{db^{(L)}} = \frac{dC}{da^{(L)}} \cdot \frac{da^{(L)}}{dz^{(L)}} \cdot \frac{dz^{(L)}}{db^{(L)}} \end{cases} \tag{1}$$

Now we compute each term alone

$$\begin{aligned} \frac{dC}{da^{(L)}} &= \frac{d}{da^{(L)}} \left(\frac{1}{2} \left(y - a^{(L)} \right)^2 \right) \\ &= y - a^{(L)} \end{aligned}$$

$$\begin{aligned} \frac{da^{(L)}}{dz^{(L)}} &= \frac{d}{dz^{(L)}} \left(\sigma\left(z^{(L)} \right) \right) \\ &= \sigma\left(z^{(L)} \right) \left(1 - \sigma\left(z^{(L)} \right) \right) \end{aligned}$$

$$\begin{aligned}\frac{dz^{(L)}}{dw^{(L)}} &= \frac{d}{dw^{(L)}}\left(w^{(L)}a^{(L-1)} + b^{(L)}\right)\\ &= a^{(L-1)}\end{aligned}$$

$$\begin{aligned}\frac{dz^{(L)}}{db^{(L)}} &= \frac{d}{db^{(L)}}\left(w^{(L)}a^{(L-1)} + b^{(L)}\right)\\ &= 1\end{aligned}$$

Let us define $\delta^{(L)}$ as follows

$$\begin{aligned}\delta^{(L)} &:= \frac{dC}{da^{(L)}} \cdot \frac{da^{(L)}}{dz^{(L)}}\\ &= \left[y - a^{(L)}\right] \cdot \left[\sigma\left(z^{(L)}\right)\left(1 - \sigma\left(z^{(L)}\right)\right)\right]\end{aligned}$$

Then equation (1) becomes

$$\begin{cases}\frac{dC}{dw^{(L)}} = \underbrace{\left[y - a^{(L)}\right] \cdot \left[\sigma\left(z^{(L)}\right)\left(1 - \sigma\left(z^{(L)}\right)\right)\right]}_{=\delta^{(L)}} \cdot \left[a^{(L-1)}\right]\\ \frac{dC}{db^{(L)}} = \underbrace{\left[y - a^{(L)}\right] \cdot \left[\sigma\left(z^{(L)}\right)\left(1 - \sigma\left(z^{(L)}\right)\right)\right]}_{=\delta^{(L)}} \cdot [1]\end{cases}$$

$$\implies \begin{cases}\frac{dC}{dw^{(L)}} = \delta^{(L)} \cdot \left[a^{(L-1)}\right]\\ \frac{dC}{db^{(L)}} = \delta^{(L)}\end{cases}$$

Now we want to compute $\frac{dC}{dw^{(L-1)}}$ but once again we cannot do it directly. We have established how changes in $z^{(L)}$ affect C but we have to take into consideration that $z^{(L)}$ depends on $a^{(L-1)}$, which in turn depends on $z^{(L-1)}$, which finally depends on $w^{(L-1)}$. Similarly we cannot compute $\frac{dC}{db^{(L-1)}}$ directly. We know how changes in $z^{(L)}$ affect C but in turn $z^{(L)}$ depends on $a^{(L-1)}$, which depends on $z^{(L-1)}$, which depends on $b^{(L-1)}$. Using the chain rule, makes the relationships more visual.

$$\begin{cases}\frac{dC}{dw^{(L-1)}} = \underbrace{\frac{dC}{da^{(L)}} \cdot \frac{da^{(L)}}{dz^{(L)}}}_{=\delta^{(L)}} \cdot \frac{dz^{(L)}}{da^{(L-1)}} \cdot \frac{da^{(L-1)}}{dz^{(L-1)}} \cdot \frac{dz^{(L-1)}}{dw^{(L-1)}}\\ \frac{dC}{db^{(L-1)}} = \underbrace{\frac{dC}{da^{(L)}} \cdot \frac{da^{(L)}}{dz^{(L)}}}_{=\delta^{(L)}} \cdot \frac{dz^{(L)}}{da^{(L-1)}} \cdot \frac{da^{(L-1)}}{dz^{(L-1)}} \cdot \frac{dz^{(L-1)}}{db^{(L-1)}}\end{cases}$$

$$\implies \begin{cases}\frac{dC}{dw^{(L-1)}} = \delta^{(L)} \cdot \frac{dz^{(L)}}{da^{(L-1)}} \cdot \frac{da^{(L-1)}}{dz^{(L-1)}} \cdot \frac{dz^{(L-1)}}{dw^{(L-1)}}\\ \frac{dC}{db^{(L-1)}} = \delta^{(L)} \cdot \frac{dz^{(L)}}{da^{(L-1)}} \cdot \frac{da^{(L-1)}}{dz^{(L-1)}} \cdot \frac{dz^{(L-1)}}{db^{(L-1)}}\end{cases} \tag{2}$$

We know $\delta^{(L)}$ so now we compute the terms we hadn't previously computed.

$$\begin{aligned}\frac{dz^{(L)}}{da^{(L-1)}} &= \frac{d}{da^{(L-1)}}\left(w^{(L)}a^{(L-1)} + b^{(L)}\right)\\ &= w^{(L)}\end{aligned}$$

$$\begin{aligned}\frac{da^{(L-1)}}{dz^{(L-1)}} &= \frac{d}{dz^{(L-1)}}\left(\sigma\left(z^{(L-1)}\right)\right)\\ &= \sigma\left(z^{(L-1)}\right)\left(1-\sigma\left(z^{(L-1)}\right)\right)\end{aligned}$$

$$\begin{aligned}\frac{dz^{(L-1)}}{dw^{(L-1)}} &= \frac{d}{dw^{(L-1)}}\left(w^{(L-1)}a^{(L-2)} + b^{(L-1)}\right)\\ &= a^{(L-2)}\end{aligned}$$

$$\begin{aligned}\frac{dz^{(L-1)}}{db^{(L-1)}} &= \frac{d}{db^{(L-1)}}\left(w^{(L-1)}a^{(L-2)} + b^{(L-1)}\right)\\ &= 1\end{aligned}$$

As before, we can define $\delta^{(L-1)}$ as follows

$$\begin{aligned}\delta^{(L-1)} &:= \frac{dz^{(L)}}{da^{(L-1)}}\cdot\frac{da^{(L-1)}}{dz^{(L-1)}}\\ &= \left[w^{(L)}\right]\cdot\left[\sigma\left(z^{(L-1)}\right)\left(1-\sigma\left(z^{(L-1)}\right)\right)\right]\end{aligned}$$

Then equation (2) becomes

$$\begin{cases}\frac{dC}{dw^{(L-1)}} = \delta^{(L)}\cdot\underbrace{\left[w^{(L)}\right]\cdot\left[\sigma\left(z^{(L-1)}\right)\left(1-\sigma\left(z^{(L-1)}\right)\right)\right]}_{=\delta^{(L-1)}}\cdot\left[a^{(L-2)}\right]\\ \frac{dC}{db^{(L-1)}} = \delta^{(L)}\cdot\underbrace{\left[w^{(L)}\right]\cdot\left[\sigma\left(z^{(L-1)}\right)\left(1-\sigma\left(z^{(L-1)}\right)\right)\right]}_{=\delta^{(L-1)}}\cdot[1]\end{cases}$$

$$\Longrightarrow\begin{cases}\frac{dC}{dw^{(L-1)}} = \delta^{(L)}\cdot\delta^{(L-1)}\cdot\left[a^{(L-2)}\right]\\ \frac{dC}{db^{(L-1)}} = \delta^{(L)}\cdot\delta^{(L-1)}\end{cases}$$

Now provided we have $\delta^{(L)}, \delta^{(L-1)}, \ldots, \delta^{(L-k)}$, $(1 \leq k \leq L-2)$, we want to show that the following property holds

$$\begin{cases}\frac{dC}{dw^{(L-k)}} = \delta^{(L)}\cdot\ldots\cdot\delta^{(L-k)}\cdot\left[a^{(L-(k+1))}\right]\\ \frac{dC}{db^{(L-k)}} = \delta^{(L)}\cdot\ldots\cdot\delta^{(L-k)}\end{cases}$$

We will use a proof by decreasing-induction. The base case has already been proven above, now we focus on the (decreasing-) induction

As previously our goal is to compute $\frac{dC}{dw^{(L-k)}}$ but once again we cannot do it directly. At this point we are able to determine that changes in $z^{(L-(k-1))}$ affect C as follows

$$\frac{dC}{dz^{(L-(k-1))}} = \delta^{(L)} \cdot \ldots \cdot \delta^{(L-(k-1))}$$

Now we have to take it from there and remark that $z^{(L-(k-1))}$ depends on $a^{(L-k)}$, which in turn depends on $z^{(L-k)}$, which finally depends on $w^{(L-k)}$. The same applies for $\frac{dC}{db^{(L-(k-1))}}$. We know how changes in $z^{(L-(k-1))}$ affect C but in turn $z^{(L-(k-1))}$ depends on $a^{(L-k)}$, which depends on $z^{(L-k)}$, which depends on $b^{(L-k)}$. Hence we get

$$\begin{cases} \frac{dC}{dw^{(L-k)}} = \frac{dC}{dz^{(L-(k-1))}} \cdot \underbrace{\frac{dz^{(L-(k-1))}}{da^{(L-k)}} \cdot \frac{da^{(L-k)}}{dz^{(L-k)}}}_{=\delta^{(L-k)}} \cdot \underbrace{\frac{dz^{(L-k)}}{dw^{(L-(k+1))}}}_{=a^{(L-(k+1))}} \\ \frac{dC}{db^{(L-k)}} = \frac{dC}{dz^{(L-(k-1))}} \cdot \underbrace{\frac{dz^{(L-(k-1))}}{da^{(L-k)}} \cdot \frac{da^{(L-k)}}{dz^{(L-k)}}}_{=\delta^{(L-k)}} \cdot \underbrace{\frac{dz^{(L-k)}}{db^{(L-(k+1))}}}_{=1} \end{cases}$$

$$\implies \begin{cases} \frac{dC}{dw^{(L-k)}} = \delta^{(L)} \cdot \ldots \cdot \delta^{(L-(k-1))} \cdot \delta^{(L-k)} \cdot \left[a^{(L-(k+1))}\right] \\ \frac{dC}{db^{(L-k)}} = \delta^{(L)} \cdot \ldots \cdot \delta^{(L-(k-1))} \cdot \delta^{(L-k)} \end{cases}$$

Which proves the property.

1.2 Stochastic Gradient Descent

From here on, the gradient step appears almost "easy", or at least the notation will be much lighter.
With η the learning rate as previously defined, the weights are updated as follows:

$$\begin{cases} w^{(l)} \leftarrow w^{(l)} - \eta \frac{dC}{dw^{(l)}} \\ b^{(l)} \leftarrow b^{(l)} - \eta \frac{dC}{db^{(l)}} \end{cases}$$

$$\implies \begin{cases} w^{(l)} \leftarrow w^{(l)} - \eta \left(\delta^{(L)} \cdot \ldots \cdot \delta^{(l)} \cdot \left[a^{(l-1)}\right]\right) \\ b^{(l)} \leftarrow b^{(l)} - \eta \left(\delta^{(L)} \cdot \ldots \cdot \delta^{(l)}\right) \end{cases}$$

It has been mentioned before but as a reminder, for our induction to work, we use $a^{(1)} = x$.

2 Original code

First we will present the original code, which is not an exact translation of the Matlab code provided. This is what we have done in the first place but it was too much of a hustle to add new layers and change the number of neurons per layers (but it was functional). Instead we use an object-oriented approach with a class *Layer* taking as parameters the number of neurons we wish to create and the number of neurons from the previous layer in order to have matrices with coherent sizes. We have some preparations to do before getting to the actual neural network.

2.1 Preparations

2.1.1 Step 0: Import statements

Before starting, here are the libraries we will use throughout this project.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import time
import winsound
```

Listing 1: Import statements (2-2-3-2)

2.1.2 Step 1: Set up the training data

First we create two categories of points that we would like to classify.

```
# initial training data
m = 5
n = 5
x1 = [0.1, 0.3, 0.1, 0.6, 0.4, 0.6, 0.5, 0.9, 0.4, 0.7]
x2 = [0.1, 0.4, 0.5, 0.9, 0.2, 0.3, 0.6, 0.2, 0.4, 0.6]
X = np.array([x1, x2])
y = np.concatenate([np.ones(m), np.zeros(n), np.zeros(m), np.ones(n
    )])
y = np.reshape(y, (2, m+n))
```

Listing 2: Creating two classes of points (2-2-3-2)

Now we use *matplotlib.pyplot* to plot these data points and verify that we have indeed two classes.

```
plt.figure(figsize=(16,9))
plt.scatter(x1[:m], x2[:m], color="r", marker="o", s=100)
plt.scatter(x1[m:m+n], x2[m:m+n], color="b", marker="x", s=100)
plt.xticks([0, 1])
plt.yticks([0, 1])
```

Listing 3: Displaying the two classes of points (2-2-3-2)

Which produces the following figure.

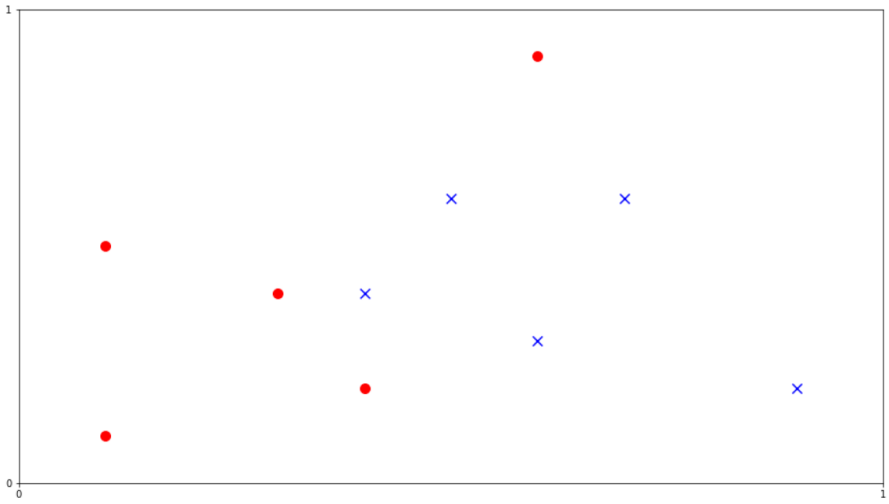

Figure 1: Representation of the data points of the two classes (2-2-3-2)

2.2 The Neural Network

2.2.1 Step 3: Set the learning rate and number of SGD iterations

We do step 3 before step 2 because we will need the learning rate *eta* in the definition of *Layer*. We also define the number of steps for SGD.

```
eta = 0.05
Niter = int(5e5)
```

Listing 4: Defining *eta* and *Niter* (2-2-3-2)

2.2.2 Step 2: Initialize weights and biases

The step title says that we will initialize weights and biases but we will actually do a more. We also define our methods for the forward and backward pass, as well as the gradient step. All this work will be done inside our class *Layer* but first we need to define some methods which will be used.

```
def cost(W2, W3, W4, b2, b3, b4):
    costvec = np.zeros(m+n)
    for i in range(m+n):
        x = np.array([[x1[i]], [x2[i]]])
        a2 = activate(x,W2,b2)
        a3 = activate(a2,W3,b3)
        a4 = activate(a3,W4,b4)
        costvec[i] = np.linalg.norm(np.reshape(y[:,i], (2,1)) - a4)

    return np.linalg.norm(costvec)**2

def sigm(z):
    return 1/(1 + np.exp(-z))

def activate(x, W, b):
    return sigm(W @ x + b)
```

Listing 5: Methods for loss & activation functions (2-2-3-2)

We are now ready to implement *Layer*, alongside methods for forward and backward pass, as well as the gradient step.

```
class Layer:
    def __init__(self, n_neurons, n_inputs):
        self.weights = 0.5*np.random.normal(size=(n_neurons, n_inputs))
        self.biases = 0.5*np.random.normal(size=(n_neurons, 1))

    def forward(self, inputs):
        # forward pass
        self.outputs = activate(inputs, self.weights, self.biases)

    def backward(self, backward_input):
        # backward pass
        self.delta = self.outputs * (1-self.outputs) * (backward_input)

    def gradient_step(self, step):
        # Gradient step
        self.weights -= eta * self.delta @ np.transpose(step)
        self.biases -= eta * self.delta
```

Listing 6: The *Layer* class (2-2-3-2)

2.2.3 Step 4: Back propagation to train the network

We are now ready to proceed with the forward pass, backward pass and gradient steps. We also measure the run time of this code snippet.

```
l2 = Layer(2, 2)
l3 = Layer(3, 2)
l4 = Layer(2, 3)

start_time = time.time()
savecost = pd.Series(np.zeros(Niter)-1)

for counter in range(Niter):
    k = np.random.randint(m+n-1) # choose one point from the sample
    x = np.array([[x1[k]], [x2[k]]])

    # forward pass
    l2.forward(x)
    l3.forward(l2.outputs)
    l4.forward(l3.outputs)

    # backward pass
    l4.backward(l4.outputs - y[:, [k]])
    l3.backward(l4.weights.T @ l4.delta)
    l2.backward(l3.weights.T @ l3.delta)

    # gradient step
    l2.gradient_step(x)
    l3.gradient_step(l2.outputs)
    l4.gradient_step(l3.outputs)

```

```
27      savecost[counter] = cost(l2.weights, l3.weights, l4.weights, l2
    .biases, l3.biases, l4.biases)
```

Listing 7: Forward pass, Backward pass and Gradient steps (2-2-3-2)

2.3 Evaluation and visual representation

2.3.1 Step 5: Show the "convergence" of the cost function

The use of *savecost* in listing 7 also allows us to save the cost on each step as counter iterate increases until *Niter*. Hence we can plot the cost as a function of the number of iterations. Here is the code to plot the figure.

```
plt.figure(figsize=(16,9))
savecost.plot(logy=True)
plt.tick_params(axis='both', which='major', labelsize=20)
plt.ticklabel_format(axis="x", style="sci", scilimits=(0,0))
```

Listing 8: Code to plot the error over the number of iterations (2-2-3-2)

And here is the figure we obtain for 5×10^5 iterations.

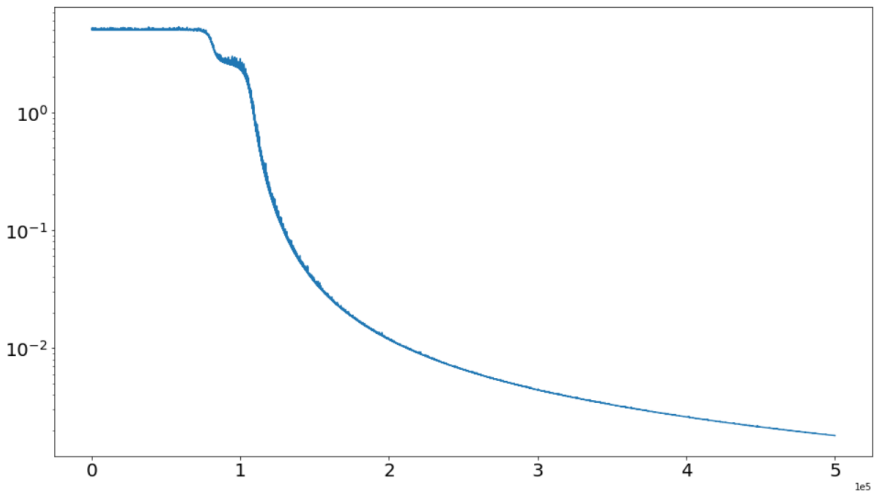

Figure 2: Loss as a function of number of iterations (2-2-3-2)

2.3.2 Step 6: Show the classification by displaying shaded and unshaded regions

Now we want to display the regions of figure 1 in different colors depending upon the class they would be classified into by our Neural Network. What we do here is to compute an $N \times N$ grid over $[0, 1] \times [0, 1]$ and classify each points in the grid. If the points belong to class A, they are colored in a color, if they belong to class B they are colored in another color.

```
N = 500
Aval = np.zeros((N, N))
Bval = np.zeros((N, N))

xvals = np.linspace(0, 1, N)
yvals = np.linspace(0, 1, N)
for k1 in range(N):
    xk = xvals[k1]
```

```
    for k2 in range(N):
        yk = yvals[k2]
        xy = np.array([[xk], [yk]])
        a2 = activate(xy,l2.weights, l2.biases)
        a3 = activate(a2,l3.weights, l3.biases)
        a4 = activate(a3,l4.weights, l4.biases)
        Aval[k2,k1] = a4[0]
        Bval[k2,k1] = a4[1]

[X,Y] = np.meshgrid(xvals,yvals)
plt.clf()
Mval = Aval>=Bval
plt.figure(figsize=(16,9))
plt.contourf(X, Y, Mval, cmap="gray", alpha=0.5)
plt.scatter(x1[:m], x2[:m], color="r", marker="o", s=100)
plt.scatter(x1[m:m+n], x2[m:m+n], color="b", marker="x", s=100)
plt.xticks([0, 1])
plt.yticks([0, 1])
```

Listing 9: Showing regions in color depending upon class (2-2-3-2)

Here is the figure with colors depending on the classification by our Neural Network.

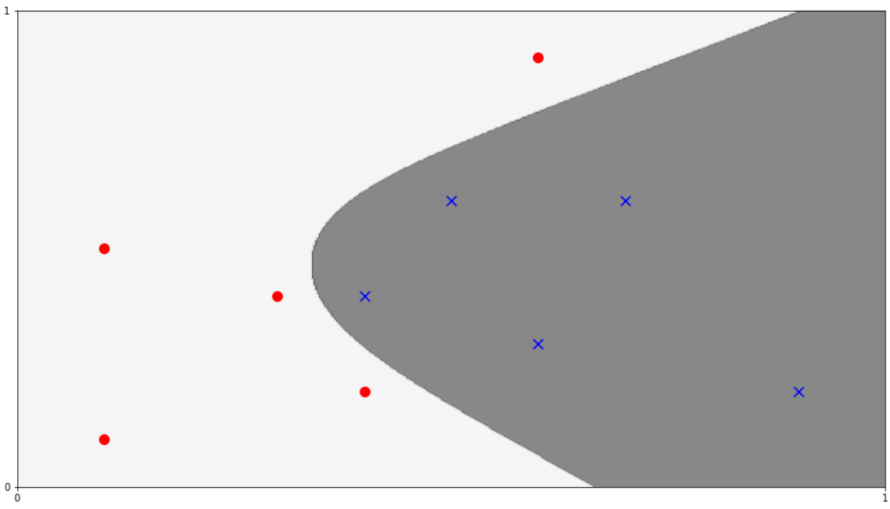

Figure 3: Regions colored depending on classification (2-2-3-2)

3 Task 2

Now we change the number of neurons per layer, the number of layers and the activation function in turn. We want to see how changes in these parameters affect our results.

3.1 Subtask 1

The first change is going from a 2-2-3-2 to a 2-5-5-5-2 architecture, that is 2 neurons in the first layer (input), then 5 neurons in each of layer 2, 3, 4, and 2 neurons in the last layer (output). We use "architecture" in lack of a better word to define the number of layers, together with the number of neurons per layer. Here is how we had to adapt our cost function.

```
def cost(W2, W3, W4, W5, b2, b3, b4, b5):
```

```
    costvec = np.zeros(m+n)
    for i in range(m+n):
        x = np.array([[x1[i]], [x2[i]]])
        a2 = activate(x,W2,b2)
        a3 = activate(a2,W3,b3)
        a4 = activate(a3,W4,b4)
        a5 = activate(a3,W5,b5)
        costvec[i] = np.linalg.norm(np.reshape(y[:,i], (2,1)) - a5)
    return np.linalg.norm(costvec)**2
```

Listing 10: Cost function adapted to the new architecture (2-5-5-5-2)

It would be possible to use an array structure (*i.e.* a list for all weights and another list for all biases), combined with a for loop in order to iterate over all weight matrices and all biases matrices. We haven't done it by lack of time but it would allow for an arbitrary number of layers rather than having to copy-paste lines whenever we add layers.
Here is the modified code for passes (forward and backward) and gradient step. It is adapted to the new architecture.

```
l2 = Layer(5, 2)
l3 = Layer(5, 5)
l4 = Layer(5, 5)
l5 = Layer(2, 5)

start_time = time.time()
savecost = pd.Series(np.zeros(Niter)-1)

for counter in range(Niter):
    k = np.random.randint(m+n-1) # choose one point from the sample
    x = np.array([[x1[k]], [x2[k]]])

    # forward pass
    l2.forward(x)
    l3.forward(l2.outputs)
    l4.forward(l3.outputs)
    l5.forward(l4.outputs)

    # backward pass
    l5.backward(l5.outputs - y[:, [k]])
    l4.backward(l5.weights.T @ l5.delta)
    l3.backward(l4.weights.T @ l4.delta)
    l2.backward(l3.weights.T @ l3.delta)

    # gradient step
    l2.gradient_step(x)
    l3.gradient_step(l2.outputs)
    l4.gradient_step(l3.outputs)
    l5.gradient_step(l3.outputs)

    # Monitor progress
    savecost[counter] = cost(l2.weights, l3.weights, l4.weights, l5
   .weights, l2.biases, l3.biases, l4.biases, l5.biases)
```

Listing 11: New code for passes & gradient step (2-5-5-5-2)

And here is the loss as a function of the number of iterations.

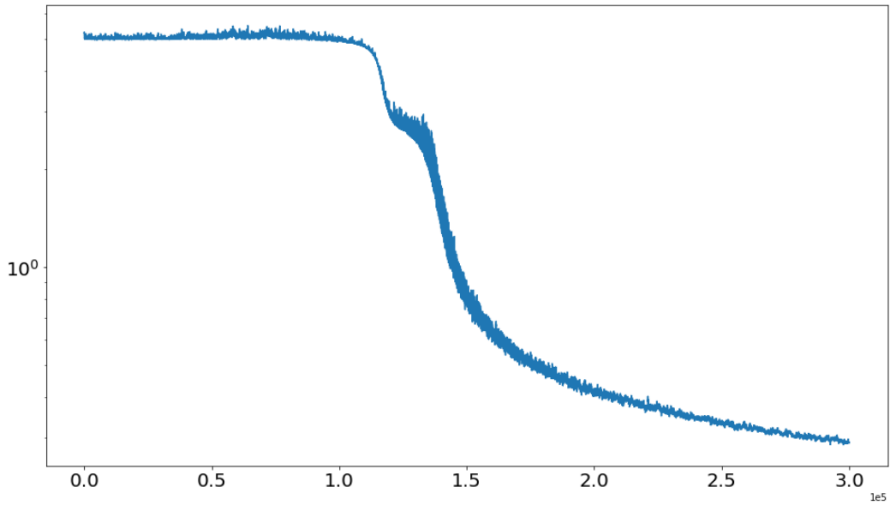

Figure 4: Loss as a function of number of iterations (2-5-5-5-2)

Although it looks similar in shape as our previous result in figure 2, one must look at the scale on the vertical axis. Indeed, 3×10^5 iterations got us to a loss of $\approx 10^{-2} = 0.01$ with the previous architecture, as opposed to $\approx 10^0 = 1$ with the new architecture. It is unclear to us whether cost functions with different architectures are actually comparable. Our intuition tells us that adding layers or neurons will not change the value of the cost function because what is measured is how different the output vector is from the actual result. Therefore the number of weights and biases the input has to go through seems irrelevant, but this is only an intuition, not an actual proof. Here is now what the color as a function of classification gives us.

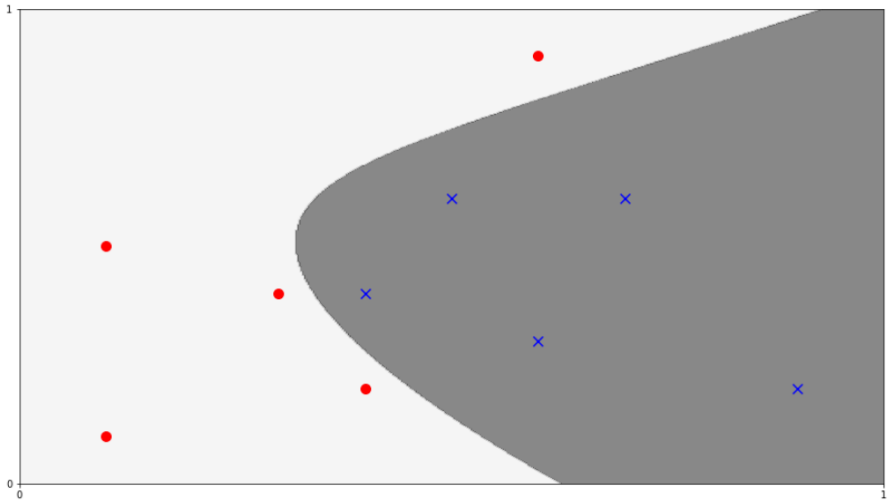

Figure 5: Regions colored depending on classification (2-5-5-5-2)

There is actually a small difference with figure 3 but this could have to do with the number of iterations too so we do not want to jump to conclusion since more than one parameter has been changed.

3.2 Subtask 2

We redefine our activation function and define the ReLU function as follows.

```python
def relu(z):
    return np.maximum(z, 0)

def activate(x, W, b):
    return relu(W @ x + b)
```

Listing 12: Definition of the relu method (ReLU)

We also adjust our learnign rate to 0.0025 as required in this subtask. Here is what the loss looks like now.

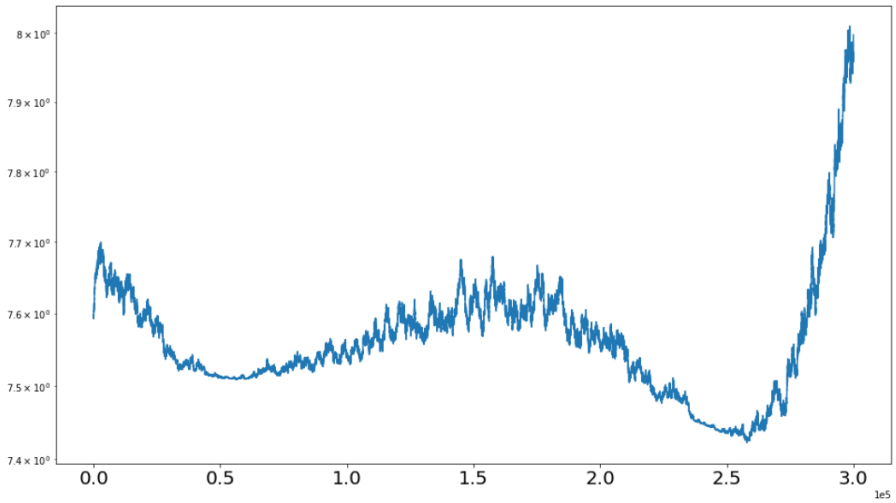

Figure 6: Loss as a function of number of iterations (ReLU)

Obviously something is wrong here, the loss goes up as we increase the number of iterations. This is very likely to be due to our step size being too big. It seems that our gradient steps somehow "overshoot" and end up missing the minimum that SGD looks for. A solution would be to reduce the learning rate or to set the number of iterations at $\approx 2.5 \times 10^5$. However both of these methods require to re-run the algorithm, which can be costly and long in real life situations. Ideally we would like to be able to anticipate these scenarios and tune our learning rate before pushing the "on" button. However we don't know whether this is actually feasible.
Because the cost increases sharply, we obtain a completely deficient classifier, as is shown by the following.

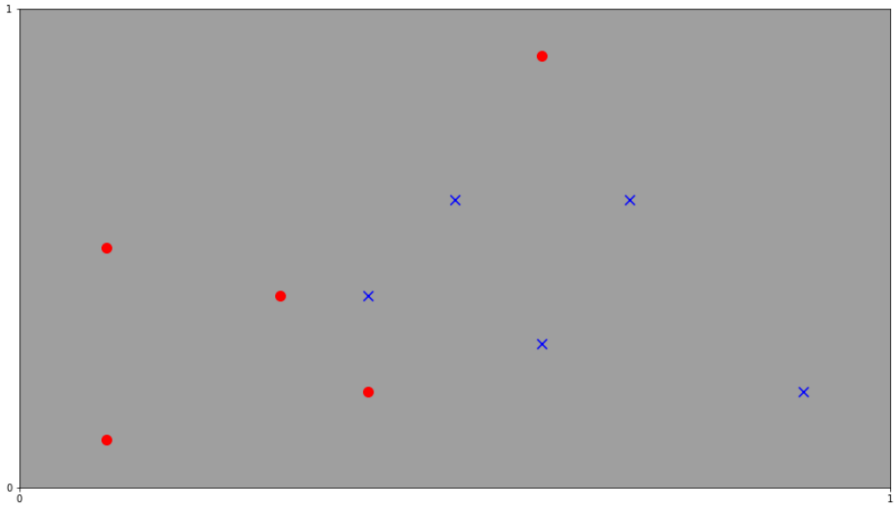

Figure 7: Regions colored depending on classification (ReLU)

Our classifier classifies all points as blue, which is obviously not what we expected from it.

3.3 Subtask 3

This time we use a network architecture of (2-5-5-5-2) and we use the ReLU activation function but we use the new points provided. Here is what those new points look like.

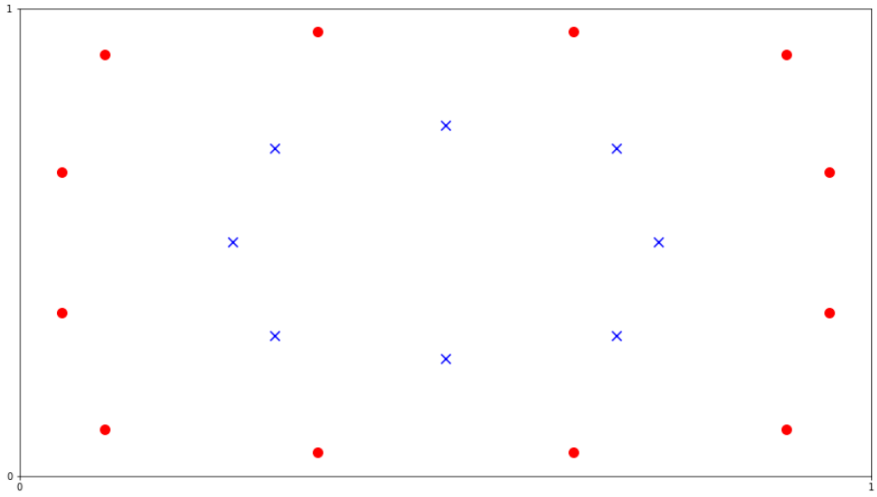

Figure 8: Representation of the data points of the two classes (new points)

We can see that these points have a different shape. They look like they are on the contour of two ellipsoids one inside of the other.
This is what the loss looks like for the new data.

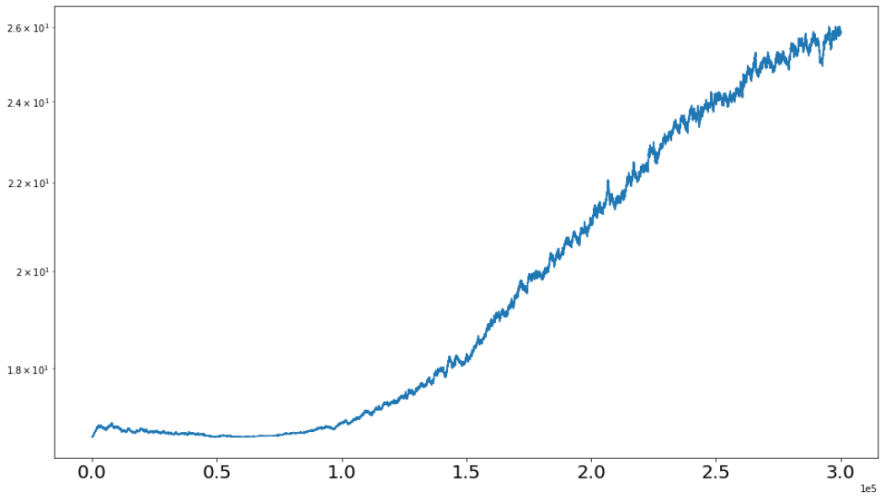

Figure 9: Loss as a function of number of iterations (new points)

The loss is once again going up and our classifier completely fails as shown here.

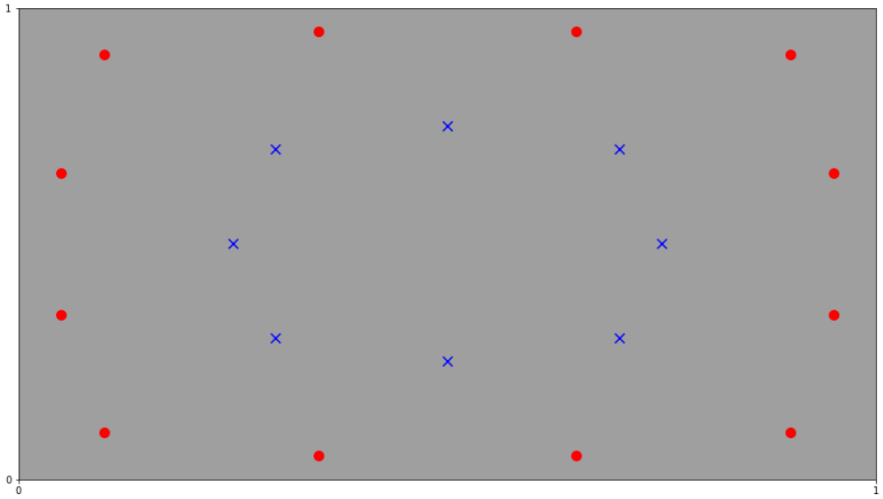

Figure 10: Regions colored depending on classification (new points)

3.4 Exploration post-subtask 3

One workaround comes to mind when tackling task 3. These points are in circle-like shapes (ellipsoidal shapes) around the points $(x, y) = (0.5, 0.5)$. So we could try centering the coordinates and switching to polar coordinates. Deducing 0.5 from both the x and the y coordinate gives us ellipsoids around the origin. Now we switch to polar coordinates by applying the following changes to $x1$ and $x2$.

```
# centering the data
x1 -= 0.5
x2 -= 0.5
# conversion to polar coordinates
x1 = np.sqrt(x1**2+x2**2)
x2 = np.arctan2(x2,x1)
```

Listing 13: Conversion to polar coordinates (polar)

Here is what our loss function looks like.

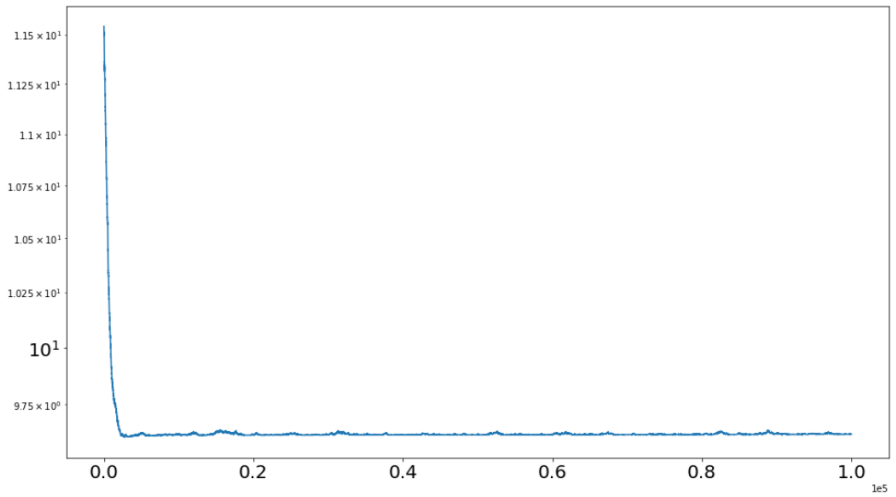

Figure 11: Loss as a function of number of iterations (polar)

And here is what the separation line looks like.

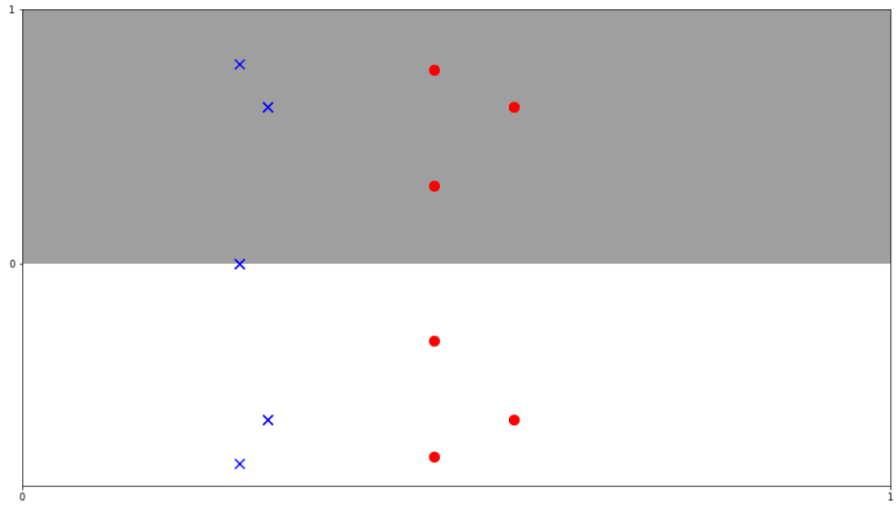

Figure 12: Regions colored depending on classification (polar)

This is not what we wanted but we wanted a separation line perpendicular to the one we obtained. This might be due to an error in our code or maybe changing the architecture of the network would solve it. It would be an interesting rabbit-hole to go down to but we will not push it further here.

Conclusion

In this project we saw that different architectures can lead to different results, even with the same data. Also, having hyperparameters optimised for one data set does not guarantee they will be optimised for another data set. This is why hyperparameter-tuning requires a attention in industry applications.
We have also experimented with different activation functions (namely sigmoid and ReLU) and this project brought to our attention the fact that the choice of one activation function over another can change the results of our classifier.

Bibliography

3Blue1Brown (n.d.). URL: https://www.youtube.com/watch?v=aircAruvnKk&list=PLZHQObOWTQDNU6R1_67000Dx_ZCJB-3pi&ab_channel=3Blue1Brown.

Brownlee, Jason (Oct. 2016). *How to Code a Neural Network with Backpropagation In Python (from scratch)*. URL: https://machinelearningmastery.com/implement-backpropagation-algorithm-scratch-python/.

Nielsen, Michael A. (2015). 'Neural Networks and Deep Learning'. In: URL: http://neuralnetworksanddeeplearning.com.

sentdex (n.d.). URL: https://www.youtube.com/playlist?list=PLQVvvaa0QuDcjD5BAw2DxE6OF2tius3V3.

Strang, Gilbert (2019). *Linear algebra and learning from data*. Wellesley-Cambridge Press. ISBN: 9780692196380.