Homework 1 - Group 6

Afonso Gonçalves João Simões Marcelo Santos

January 12, 2022

Before solving these exercises, we want to recall some derivative rules that will be helpful throughout this document:

- Reciprocal Rule: $\frac{\partial}{\partial x}\frac{1}{f(x)} = \frac{-f'(x)}{f(x)^2}$
- Quocient Rule: $\frac{\partial}{\partial x}\frac{f(x)}{g(x)} = \frac{f'(x)g(x)-f(x)g'(x)}{g(x)^2}$
- Chain Rule: $\frac{\partial}{\partial x}f(g(x)) = f'(g(x)) \cdot g'(x)$

Question 1

1.1

$$\begin{aligned}
\sigma'(z) &= \left(\frac{1}{1+e^{-z}}\right)' = \frac{-(1+e^{-z})'}{(1+e^{-z})^2} = \frac{e^{-z}}{(1+e^{-z})^2} \\
&= \frac{1+e^{-z}-1}{(1+e^{-z})^2} = \frac{1+e^{-z}}{(1+e^{-z})^2} - \frac{1}{(1+e^{-z})^2} \\
&= \frac{1}{1+e^{-z}} - \frac{1}{1+e^{-z}} \cdot \frac{1}{1+e^{-z}} \\
&= \sigma(z) - \sigma(z)\cdot\sigma(z) \\
&= \sigma(z)(1-\sigma(z)) \\
&\quad Q.E.D.
\end{aligned} \tag{1}$$

1.2

We can derive

$$\begin{aligned}(-log(\sigma(z)))' &= -\frac{1}{\sigma(z)}\sigma'(z), \text{ from the Chain Rule (4)} \\ &= -\frac{1}{\sigma(z)}\sigma(z)(1-\sigma(z)) \\ &= \sigma(z) - 1\end{aligned} \tag{2}$$

The second derivative is trivially calculated with:

$$(-log(\sigma(z)))'' = (\sigma(z)-1)' = \sigma'(z) = \sigma(z)(1-\sigma(z))$$

To demonstrate that the binary logistic loss is convex as a function of z, we can recall the following theorem from Convexity and differentiable functions[1]:

> **Theorem 2**: Let $K \subset \mathbb{R}$ be an interval, and let f be a real valued function on K with a continuous second derivative. If f'' is nonnegative everywhere, then f is convex on K.

We know that:

$$\begin{aligned}&0 < \sigma(z) < 1, \forall z \in \mathbb{R} \\ &\Leftrightarrow -1 < -\sigma(z) < 0, \forall z \in \mathbb{R} \\ &\Leftrightarrow 0 < 1-\sigma(z) < 1, \forall z \in \mathbb{R} \\ &\Leftrightarrow 0 < \sigma(z)(1-\sigma(z)) < 1, \forall z \in \mathbb{R} \\ &\Leftrightarrow 0 < (-log(\sigma(z)))'' < 1, \forall z \in \mathbb{R}\end{aligned} \tag{3}$$

Hence, we conclude that the second derivative of $-log(\sigma(z))$ is strictly positive and consequently prove that the binary logistic loss is convex as a function of z.

1.3

We know that the Jacobian Matrix is "the matrix of all the function's first-order partial derivatives"[2], hence we can derive the following:

[1] https://math.ucr.edu/~res/math133/convex-functions.pdf

[2] https://en.wikipedia.org/wiki/Jacobian_matrix_and_determinant

$$J_{jk} = \frac{\partial [softmax(z)]_j}{\partial z_k} \tag{4}$$

$$= \frac{\partial}{\partial z_k} \frac{exp(z_j)}{\sum_{i=1}^{K} exp(z_i)}$$

If $j \neq k$, we have that

$$\frac{\partial}{\partial z_k} \frac{exp(z_j)}{\sum_{i=1}^{K} exp(z_i)} = exp(z_j) \times \frac{\partial}{\partial z_k} \frac{1}{\sum_{i=1}^{K} exp(z_i)}$$

$$= exp(z_j) \times \frac{-\frac{\partial}{\partial z_k}\left(\sum_{i=1}^{K} exp(z_i)\right)}{\left[\sum_{i=1}^{K} exp(z_i)\right]^2}$$

, from the Reciprocal Rule (1)

$$= exp(z_j) \times \frac{-\frac{\partial}{\partial z_k} exp(z_k)}{\left[\sum_{i=1}^{K} exp(z_i)\right]^2} \tag{5}$$

$$= -\frac{exp(z_j)exp(z_k)}{\left[\sum_{i=1}^{K} exp(z_i)\right]^2}$$

$$= -\frac{exp(z_j)}{\sum_{i=1}^{K} exp(z_i)} \frac{exp(z_k)}{\sum_{i=1}^{K} exp(z_i)}$$

$$= -[softmax(z)]_j [softmax(z)]_k$$

If $j = k$, we have that

$$\begin{aligned}
&\frac{\partial}{\partial z_j}\frac{exp(z_j)}{\sum_{i=1}^{K} exp(z_i)} \\
&= \frac{\left(\frac{\partial}{\partial z_j} exp(z_j)\right)\left(\sum_{i=1}^{K} exp(z_i)\right) - exp(z_j)\frac{\partial}{\partial z_j}\left(\sum_{i=1}^{K} exp(z_i)\right)}{\left(\sum_{i=1}^{K} exp(z_i)\right)^2} \\
&\text{, from the Quotient Rule (2)} \\
&= \frac{exp(z_j)\left(\sum_{i=1}^{K} exp(z_i)\right) - exp(z_j)exp(z_j)}{\left(\sum_{i=1}^{K} exp(z_i)\right)^2} \\
&= \frac{exp(z_j)\left(\left(\sum_{i=1}^{K} exp(z_i)\right) - exp(z_j)\right)}{\left(\sum_{i=1}^{K} exp(z_i)\right)^2} \\
&= \frac{exp(z_j)}{\sum_{i=1}^{K} exp(z_i)} \times \frac{\left(\sum_{i=1}^{K} exp(z_i)\right) - exp(z_j)}{\sum_{i=1}^{K} exp(z_i)} \\
&= \frac{exp(z_j)}{\sum_{i=1}^{K} exp(z_i)}\left(1 - \frac{exp(z_j)}{\sum_{i=1}^{K} exp(z_i)}\right) \\
&= [softmax(z)]_j(1 - [softmax(z)]_j)
\end{aligned} \tag{6}$$

We can conclude that the Jacobian matrix is a symmetric matrix with each entry J_{jk} defined as:

$$\begin{aligned}
J_{jk} &= \frac{\partial [softmax(z)]_j}{\partial z_k} \\
&= \begin{cases} -[softmax(z)]_j[softmax(z)]_k, & j \neq k \\ [softmax(z)]_j(1 - [softmax(z)]_j), & j = k \end{cases}
\end{aligned} \tag{7}$$

1.4

Gradient Calculation

The gradient of the given loss function, with respect to z, is given by:

$$\nabla L(z; y=j) = \begin{bmatrix} \frac{\partial}{\partial z_1} L(z; y=j) & \dots & \frac{\partial}{\partial z_K} L(z; y=j) \end{bmatrix}$$

$$= \begin{bmatrix} \frac{\partial}{\partial z_1} - log[softmax(z)]_j & \dots & \frac{\partial}{\partial z_K} - log[softmax(z)]_j \end{bmatrix}$$

$$= \begin{bmatrix} -\frac{\frac{\partial}{\partial z_1}[softmax(z)]_j}{[softmax(z)]_j} & \dots & -\frac{\frac{\partial}{\partial z_K}[softmax(z)]_j}{[softmax(z)]_j} \end{bmatrix}$$

, from the Chain Rule (4)

$$= \frac{-1}{[softmax(z)]_j} \begin{bmatrix} \frac{\partial}{\partial z_1}[softmax(z)]_j & \dots & \frac{\partial}{\partial z_K}[softmax(z)]_j \end{bmatrix} \tag{8}$$

From the previous exercise, we can conclude that

$$J_k = \begin{cases} \frac{-1}{[softmax(z)]_j} \cdot -[softmax(z)]_j[softmax(z)]_k, & k \neq j \\ \frac{-1}{[softmax(z)]_j} \cdot [softmax(z)]_j(1 - [softmax(z)]_j), & k = j \end{cases} \tag{9}$$

$$= \begin{cases} [softmax(z)]_k, & k \neq j \\ [softmax(z)]_j - 1, & k = j \end{cases}$$

Hessian calculation

The Hessian of the Loss function is given as $H = \nabla^2 L(z; y=j)$ and each entry of the Hessian is:

$$
\begin{aligned}
H_{ab} &= \frac{\partial}{\partial z_a}\left(\nabla L(z; y=j)\right)_b \\
&= \begin{cases} \frac{\partial}{\partial z_a}[softmax(z)]_b, & b \neq j \\ \frac{\partial}{\partial z_a}\left([softmax(z)]_j - 1\right), & b = j \end{cases} \\
&= \begin{cases} \frac{\partial}{\partial z_a}[softmax(z)]_b, & b \neq j \\ \frac{\partial}{\partial z_a}[softmax(z)]_j, & b = j \end{cases} \qquad (10) \\
&= \frac{\partial}{\partial z_a}[softmax(z)]_b \\
&= \begin{cases} -[softmax(z)]_a[softmax(z)]_b & , \quad a \neq b \\ [softmax(z)]_a(1-[softmax(z)]_a) & , \quad a = b \end{cases}
\end{aligned}
$$

Convexity proof

We know that

1. A function f is convex if and only if it has a Positive Semi-Definite Hessian;
2. A symmetric matrix $A \in \mathbb{R}^{n \times n}$ is Positive Semi-Definite if, for any $X \in \mathbb{R}^n, X^TAX \geq 0$;

From the previous exercise, we know that the Hessian of the loss function is symmetric, due to the Commutative property of the multiplication $(-[softmax(z)]_a[softmax(z)]_b = -[softmax(z)]_b[softmax(z)]_a)$. Moreover, we have that

$$
\begin{aligned}
&X^T H X = \\
&= \left(\sum_a [softmax(z)]_a (1 - [softmax(z)]_a) x_a^2 \right) \\
&\quad - \sum_{a \neq b} [softmax(z)]_a [softmax(z)]_b x_a x_b \\
&= \left(\sum_a [softmax(z)]_a x_a^2 - [softmax(z)]_a^2 x_a^2 \right) \\
&\quad - \left[\left(\sum_a [softmax(z)]_a x_a \right) \left(\sum_a [softmax(z)]_a x_a \right) - \sum_a [softmax(z)]_a^2 x_a^2 \right] \\
&= \sum_a [softmax(z)]_a x_a^2 - \sum_a [softmax(z)]_a^2 x_a^2 \\
&\quad - \left(\sum_a [softmax(z)]_a x_a \right)^2 + \sum_a [softmax(z)]_a^2 x_a^2 \\
&= \sum_a [softmax(z)]_a x_a^2 - \left(\sum_a [softmax(z)]_a x_a \right)^2 \\
&= \left(\sum_a [softmax(z)]_a x_a^2 \right) \left(\sum_a [softmax(z)]_a \right) - \left(\sum_a [softmax(z)]_a x_a \right)^2 \\
&= \left(\sum_a \left(\sqrt{[softmax(z)]_a} \cdot x_a \right)^2 \right) \left(\sum_a \sqrt{[softmax(z)]_a}^2 \right) - \\
&\quad \left(\sum_a \sqrt{[softmax(z)]_a} \cdot \sqrt{[softmax(z)]_a} \cdot x_a \right)^2 \\
&= \left(\sum_a \left(\sqrt{[softmax(z)]_a} \cdot x_a \right)^2 \right) \left(\sum_a \sqrt{[softmax(z)]_a}^2 \right) - \\
&\quad \left(\sum_a \sqrt{[softmax(z)]_a} \cdot x_a \cdot \sqrt{[softmax(z)]_a} \right)^2
\end{aligned}
\tag{11}
$$

From Cauchy-Schwarz, we derive

$$
\begin{aligned}
&|\langle \vec{u}, \vec{v} \rangle| \leq \|\vec{u}\| \cdot \|\vec{v}\| \\
&\Leftrightarrow \left| \sum_i u_i v_i \right| \leq \sqrt{\sum_i u_i^2} \cdot \sqrt{\sum_i v_i^2} \\
&\Leftrightarrow \left| \sum_i u_i v_i \right|^2 \leq \sqrt{\sum_i u_i^2}^2 \cdot \sqrt{\sum_i v_i^2}^2 \\
&\Leftrightarrow \left(\sum_i u_i v_i \right)^2 \leq \left(\sum_i u_i^2 \right) \left(\sum_i v_i^2 \right) \\
&\Leftrightarrow 0 \leq \left(\sum_i u_i^2 \right) \left(\sum_i v_i^2 \right) - \left(\sum_i u_i v_i \right)^2
\end{aligned}
\tag{12}
$$

Therefore, we conclude that

$$
\begin{aligned}
X^T H X = &\left(\sum_a \left(\sqrt{[softmax(z)]_a} \cdot x_a \right)^2 \right) \left(\sum_a \sqrt{[softmax(z)]_a}^2 \right) - \\
&\left(\sum_a \sqrt{[softmax(z)]_a} \cdot x_a \cdot \sqrt{[softmax(z)]_a} \right)^2
\end{aligned}
\tag{13}
$$

$$\geq 0 \quad \text{by Cauchy-Schwarz}$$

i.e., H is Positive Semi-Definite.
From 1), we prove that $L(z; y = j)$ is convex with respect to z.

1.5

Convexity Proof

We know that the multinomial logistic loss is $L : \mathbb{R}^m \to \mathbb{R}$ and, from the previous exercise, we know that L is convex w.r.t. z. Moreover, we know that z is an affine function of W and b: $z = W\phi(x) + b$.

We can express $L(W, b)$ as a the composition of an affine map $z : \mathbb{R}^n \to \mathbb{R}^m$ with a convex function $L : \mathbb{R}^m \to \mathbb{R}$: $L(W, b) = L(z(W, b)) = L(W\phi(x) + b)$. From the hint given above, we can conclude that the multinomial logistic loss is convex with respect to (W, b).

Generalization

From "Convexity Theory and Gradient Methods - Angelia Nedić"[3] and "Convex Optimization — Boyd & Vandenberghe"[4], we know that

> The composition of two functions
>
> $$f(x) = h(g(x)) = h(g_1(x), g_2(x), ..., g_p(x))$$
>
> $$h : \mathbb{R}^p \to \mathbb{R}\ ,\ g : \mathbb{R}^m \to \mathbb{R}^p$$
>
> is convex if and only if
>
> - 1. each g_i is convex, h is convex and nondecreasing in each argument
> - 2. each g_i is concave, h is convex and nonincreasing in each argument

When applied to this case, the multinomial logistic loss function is denoted by h, and $z(W, b)$ is denoted by g.

We cannot generalize that the multinomial logistic loss is convex with respect to W and b: If $z(W, b)$ is not convex w.r.t. (W, b), then the composition will not be convex w.r.t. (W, b) and consequently a local minimum will not necessarily be a global minimum.

For example, if $z(W, b) = sin(\|W\|) \cdot b$, z is not convex w.r.t. (W, b), thus the loss function will not be either.

[3]`https://www.ima.umn.edu/materials/2013-2014/ND5.27-6.13.14/20872/IMAConvexity2014-p2.pdf`

[4]`https://web.stanford.edu/class/ee364a/lectures/functions.pdf`

Question 2

2.1

First, let's prove that L is convez w.r.t z:

The first derivative of L w.r.t. to z is:

$$\begin{aligned}
\nabla_z L(z; y) &= \nabla_z \frac{1}{2} \|z - y\|_2^2 \\
&= \frac{1}{2} \nabla_z \|z - y\|_2^2 \\
&= \frac{1}{2} \nabla_z (z^T z - 2y^T z + \|y\|^2) \\
&= \frac{2z - 2y^T}{2} \\
&= z - y^T
\end{aligned} \tag{14}$$

Thus, the second derivative is:

$$\begin{aligned}
\nabla_z^2 L(z; y) &= \nabla_z (z - y^T) \\
&= I
\end{aligned} \tag{15}$$

Since the second derivative of L w.r.t. z is strictly positive, we know that L is convex w.r.t. z.

We also know that, by definition, z is an affine function of (W, b):

$$z(W, b) = W^T \phi(x) + b$$

.

Thus, from the hint given in question 1.5, we conclude that $L(z; y)$ is the composition of an affine function z with a convex function $L(z; y)$, thus we prove that $L(W, b) = L(z(W, b))$ is convex w.r.t (W, b).

2.2.a

In this exercise, we are training Linear Regression model to predict the price of a property, given the features describing it. We will use the Stochastic Gradient Descent to approximate the model parameters to the minimizer of the loss function:

$$W^{k+1} = W^k - \eta_k \nabla_W (L(W^k; (x_t, y_t))) \tag{16}$$

The loss function we aim to minimize is given in the following code snippet:

```python
def evaluate(self, X, y):
    """
    return the mean squared error between the
    model's predictions for X
    and the ground truth y values
    """
    yhat = self.predict(X) # phi(X)W
    error = yhat - y

    # sum(error_i * error_i) -> squared error norm
    squared_error = np.dot(error, error)

    mean_squared_error = squared_error / y.shape[0]
    return np.sqrt(mean_squared_error)
```

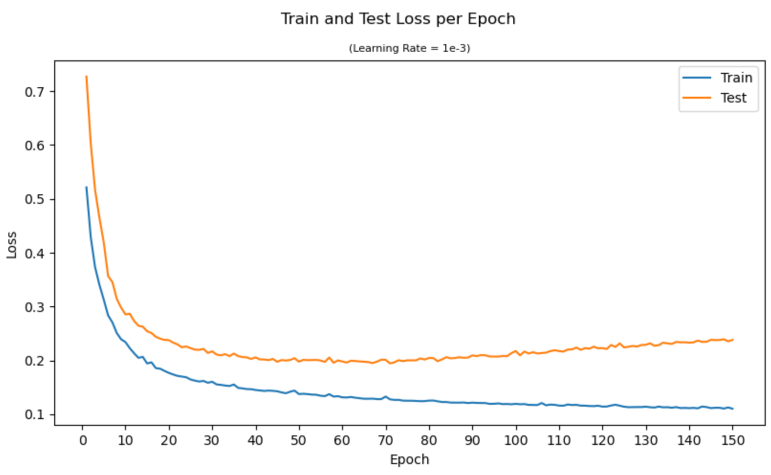
Hence we can describe it as:

$$L(W, \phi(X), Y) = \sqrt{\frac{\|\phi(X)W - Y\|^2}{N}}$$

And its derivative will be:

$$
\begin{aligned}
\nabla_W L(W, \phi(X), Y) &= \nabla_W \sqrt{\frac{\|W\phi(X) - Y\|^2}{N}} \\
&= \frac{1}{\sqrt{N}} \nabla_W \sqrt{\|W\phi(X) - Y\|^2} \\
&= \frac{\nabla_W \|W\phi(X) - Y\|^2}{2\sqrt{N} \times \sqrt{\|W\phi(X) - Y\|^2}} \\
&= \frac{\nabla_W (W^2\phi(X)^2 - 2W\phi(X)Y + Y^2)}{2\sqrt{N \times \|W\phi(X) - Y\|^2}} \\
&= \frac{2\phi(X)^T\phi(X)W - 2\phi(X)^TY}{2\sqrt{N \times \|W\phi(X) - Y\|^2}} \qquad (17) \\
&= \frac{\phi(X)^T\phi(X)W - \phi(X)^TY}{\sqrt{N \times \|W\phi(X) - Y\|^2}} \\
&= \frac{\phi(X)^T(\phi(X)W - Y)}{\sqrt{N \times \|W\phi(X) - Y\|^2}} \\
&= \frac{\phi(X)^T(\hat{Y} - Y)}{\sqrt{N \times \|\hat{Y} - Y\|^2}} \quad (N = 1) \\
&= \frac{\phi(X)^T(\hat{Y} - Y)}{\|\hat{Y} - Y\|}
\end{aligned}
$$

After training this model for 150 epochs, we got the following results.



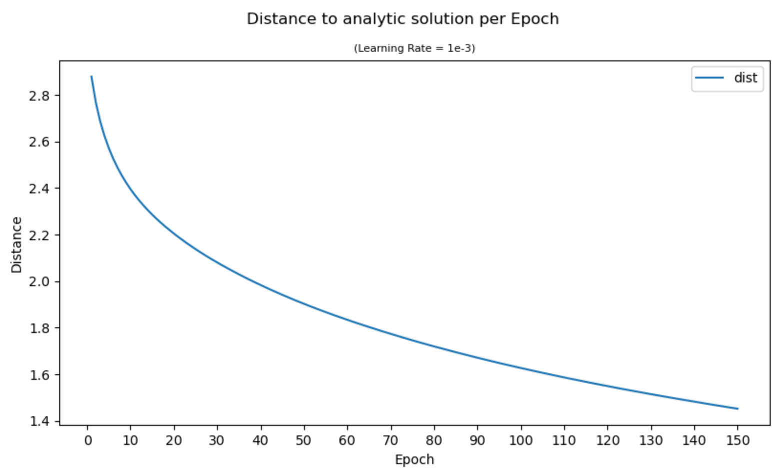
The first graph depicts the loss in each epoch, which indicates how far the model predictions were from the real values. Hence, lower losses mean that the model is more accurate.

We can see that in the beginning of the training, the loss had a value of 0.521 and 0.727 for the train and test sets, respectively, which rapidly decreased to values between 0.2 and 0.3 in the first 10 epochs. After this moment, the training loss kept decreasing, reaching a minimum loss of 0.111 on epoch 150. However, the test loss started increasing, having values from 0.195 at epoch 71 to 0.239, at epoch 148, and finishing the experiment with a value of 0.238.

We can explain this drift between the test and train loss functions because **the model is overfitting the training set**, i.e. the model is adjusting its weights based only on the gradients obtained from the training set. One can use regularization to diminish this problem.

We can also note that the model gets some loss spikes during the experiments. We hypothesise this result comes from the randomness of the stochastic gradient descent. Stochastic optimization algorithms may, sometimes, update the model parameters in the wrong ”direction”, leading to slight increases in the attained loss.

Below, we have the plot of the distance from the analytical solution in each epoch:



The analytical solution corresponds to the model parameters (weights) that minimize the loss function for the training data set. This value is analytically calculated by a closed form expression. We can see the analytical solution as the target parameters we want our model to learn, and the distance between that solution and the model parameters as ”how far are we from the optimized weights”.

The initial parameters start with a distance of 2.879 and monotonically decreases in further epochs, reaching a final distance of 1.452. After experimenting training the model for 1000 epochs, we could see that the distance kept reducing, reaching a minimum 0.669.

From this result, we can infer that the model is correctly learning the parameters that best predict the prices of the desired properties.

2.2.b

Our single layer feedforward network is given by the following expression:

$$\begin{aligned} f(x) = z^{(2)}(x) &= o(W^{(2)} \cdot h(x) + b^{(2)}) = W^{(2)} \cdot h(x) + b^{(2)} \\ &= W^{(2)} \cdot g(z^{(1)}(x)) + b^{(2)} \\ &= W^{(2)} \cdot g(W^{(1)} \cdot x + b^{(1)}) + b^{(2)} \end{aligned} \tag{18}$$

Where:

- $o(x) = x$. This applies for regression networks.
- $g(x) = ReLU(x)$, which is our activation function.

Throughout the training of the model, we are tasked with reducing the loss given by:

$$L(f(x), y) = \frac{1}{2}\|f(x) - y\|^2 = \frac{1}{2}\|z^{(2)}(x) - y\|^2$$

By the gradient rules, we have that:

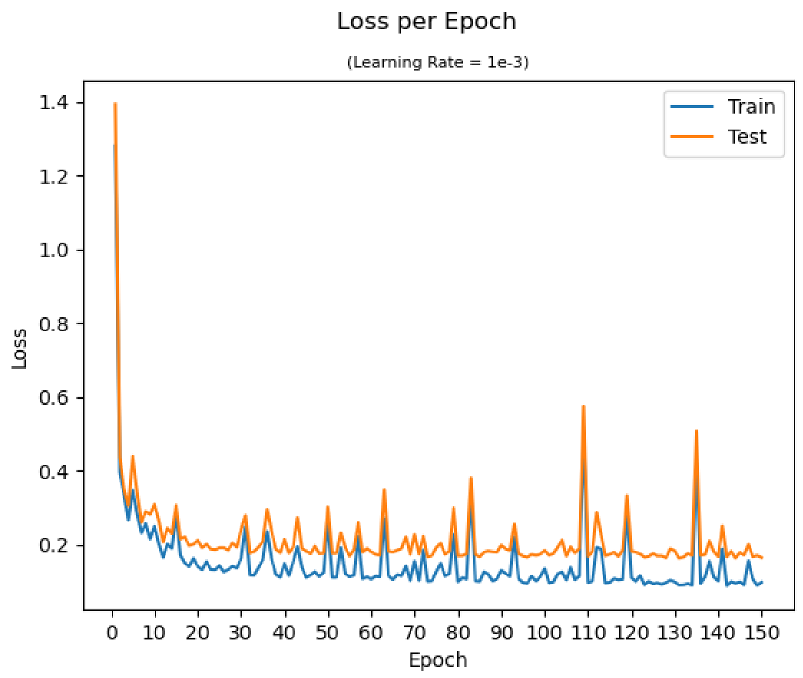
$$\nabla_{f(x)} L(f(x), y) = f(x) - y \nabla_{z^{(2)}} L(f(x), y) = z^{(2)} - y$$

For the hidden layer gradients, we have the following expressions:

$$\nabla_{h^{(l)}} L(f(x), y) = W^{(l+1)^T} \nabla_{z^{(l+1)}} L(f(x), y)$$
$$\nabla_{z^{(l)}} L(f(x), y) = \nabla_{h^{(l)}} L(f(x), y) \odot g'(z^{(l)})$$
$$g'(x) = 1_{x>0}$$
$$\nabla_{W^{(l)} L(f(x), y)} = \nabla_{z^{(l)}} L(f(x), y) h^{(l-1)^T}$$
$$\nabla_{b^{(l)}} L(f(x), y) = \nabla_{z^{(l)}} L(f(x), y)$$

We can apply them throughout all layers, starting from $\nabla_{z^{(2)}} L(f(x), y) = z^{(2)} - y$

The reported loss graph is the following:

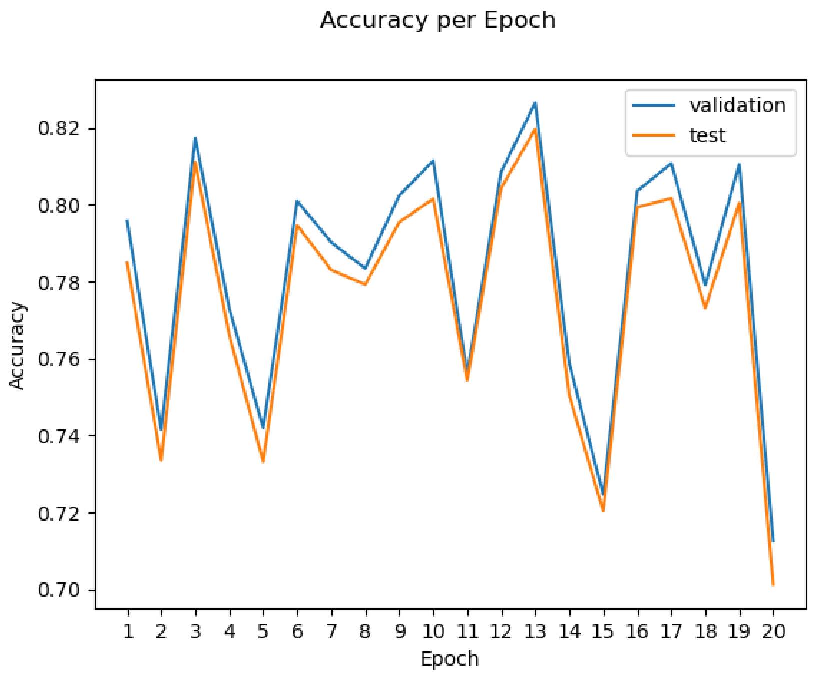


In the initial epochs, the loss values were high, but quickly lowered toward 0.2. In the following epochs, the most significant changes happened in the form of spikes, most likely due to steps in the wrong direction in gradient descent. Compared to the linear model, there was a smaller gap between the training and test sets, hinting that in the neural network overfitting did not occur. Furthermore, we obtained a smaller loss at the final epoch in the neural network model.

Question 3

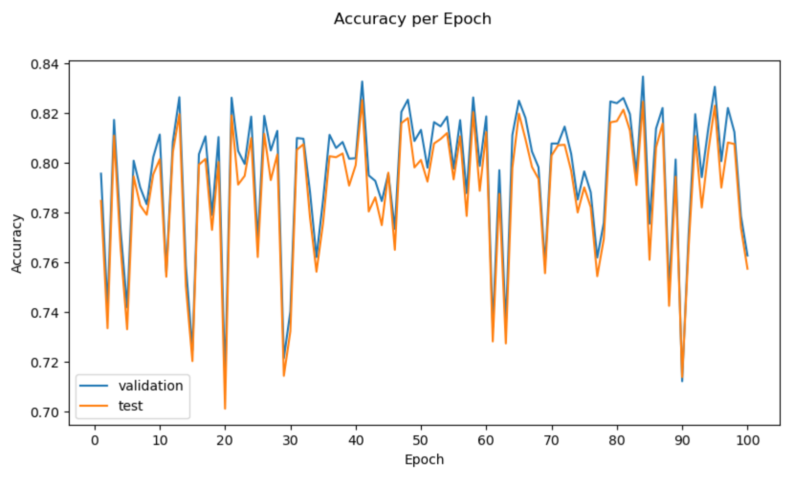
3.1.a

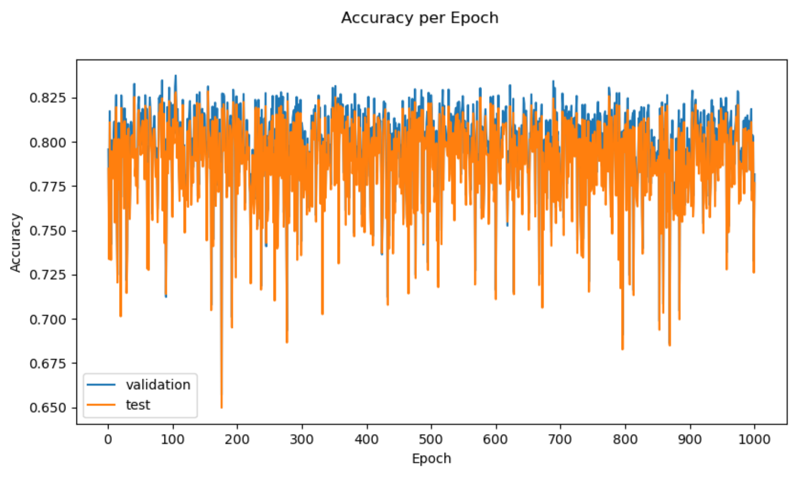
The graph below depicts the accuracy scores attained by the perceptron in each epoch, for the validation and test data sets.



We can see that the accuracy scores vary between 0.7 and 0.8 and that the perceptron has a similar performance for the validation and test sets, however with slightly higher scores for the former (the biggest difference is 0.0113). We believe the perceptron had a similar performance in both data sets since none of them contained data the perceptron trained with.

We can also note that the accuracy score oscillates drastically in each epoch, reaching amplitudes around 10% in some epochs. This result may suggest that the perceptron already reached its maximum accuracy. For this reason, we decided to analyze the accuracy scores the perceptron gets in 100 and 1000 epochs. The obtained results are shown in the graphs below:





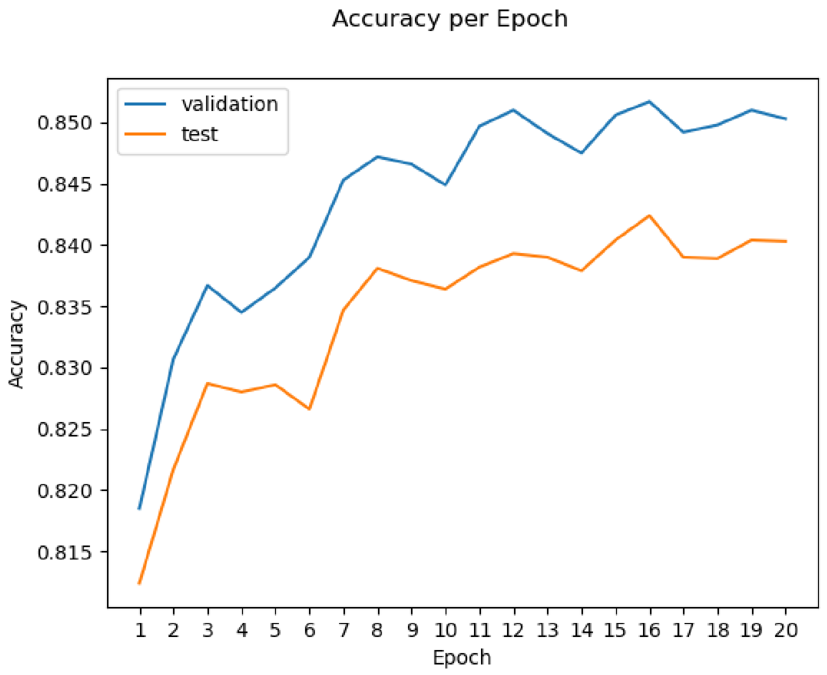
The maximum accuracy obtained in these runs was 0.840, which corroborates the previous idea.

We know that the perceptron is a Linear classifier and is unable to represent non-linearly separable problems. The limited accuracy results obtained in this experiment suggest that the classification problem of this exercise may be too complex to be represented in a single perceptron and that we would need a more complex model, such as a Neural Network to achieve better results. This explains why the perceptron is underfitting the training data and having such low accuracies.

Finally, we verified that adding a learning rate to the perceptron did not change the accuracy results.

3.1.b

After implementing and running the logistic regression model, we obtained the following results:



We can observe that this model achieves accuracy scores similar to the ones attained by the previous model, but with much less oscillation.

We believe the smoother results stem from the gradient that is calculated in the logistic regression, which is given as:

$$\nabla = e_y \phi(x)^T - \sum_y P_w(y'|x) e_{y'} \phi(x)^T$$

This gradient corresponds to a matrix where each column y' equals $\phi(x)$ multiplied by the probability the model assigned to the class y', except for the gold class. Applying this gradient to the weight matrix tunes up the weights of the correct class and tunes down the weights of every incorrect class, according to the probability assigned to that class. This means that the higher the score the model gives to an incorrect class, the higher the gradient that will be applied to the weights of that same class. Contrarily, the perceptron algorithm only updates the weights of the gold and incorrectly predicted classes, leaving the other classes as they were.

3.2.a

Multi-layer perceptrons with non-linear activations are more expressive than a single perceptron because the non-linear activation functions allow to perform non-linear operations on the input data. The combination of these non-linear activations, across different layers, allows to create more complex models that represent more complex problems, hence the increased expressiveness.

This statement is supported by the following theorems:

Theorem (Hornik et al. (1989)): An NN with one hidden layer and a linear output can approximate arbitrarily well any continuous function, given enough hidden units.

Theorem (Montufar et al. (2014)): The number of linear regions carved out by a deep neural network with D inputs, depth L, and K hidden units per layer with ReLU activations is:

$$O\left(\left(\begin{array}{c} K \\ D \end{array}\right)^{D(L-1)} K^D\right)$$

This means that, for a fixed number of hidden units, the networks are exponentially more expressive (Lecture 05 slides[5]).

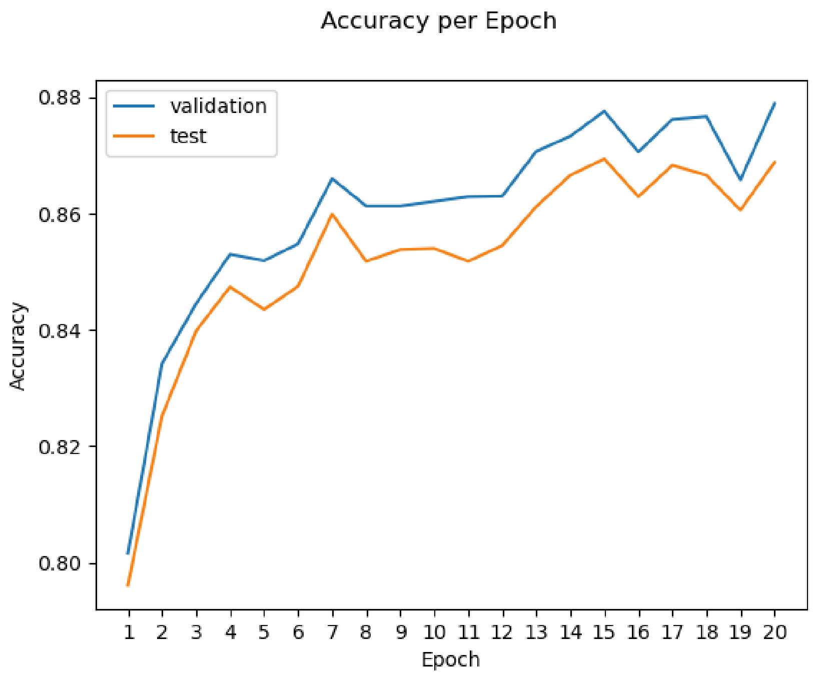
Note that if the activation functions of a multi-layer perceptron are linear, then it is equivalent to a single perceptron and wil not be able to model more complex problems. (With linear activation functions it is possible to represent the entire feedforward process into a single matrix).

The results obtained in 3.1.a hint that the problem may be too complex to be represented by a single perceptron (which follows a linear model). A more complex model, such as a multi-layer perceptron with non-linear activation functions may be able to better model this problem and to consequently have higher accuracy scores for this classification task.

[5]`https://fenix.tecnico.ulisboa.pt/downloadFile/563568428828553/lecture_05.pdf`

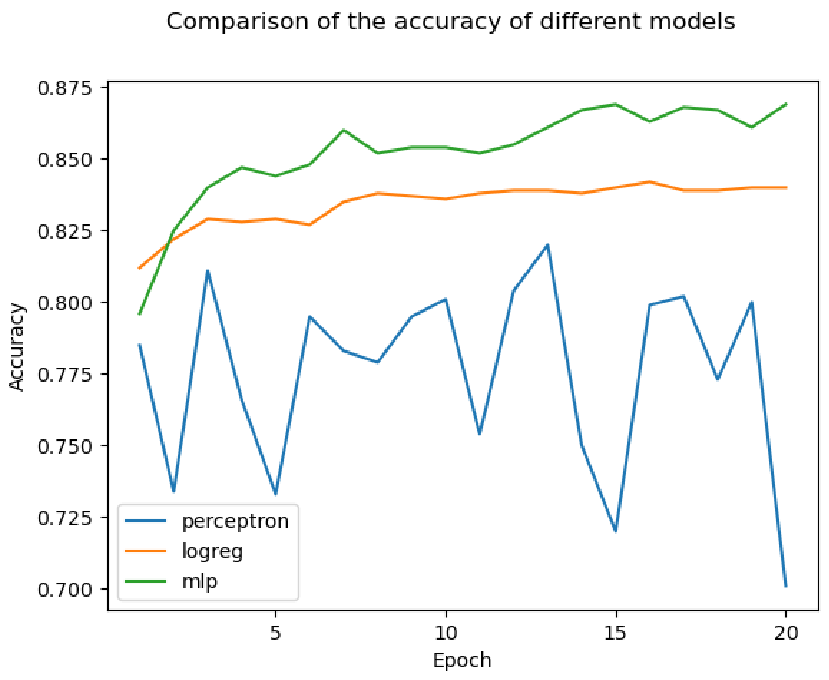
3.2.b

After implementing and executing the MLP, we got the accuracy results depicted below:



This shows that this model attained higher accuracy scores than the perceptron, as expected.

Below we compare the accuracy of the perceptron, the logistic regression and the MLP:

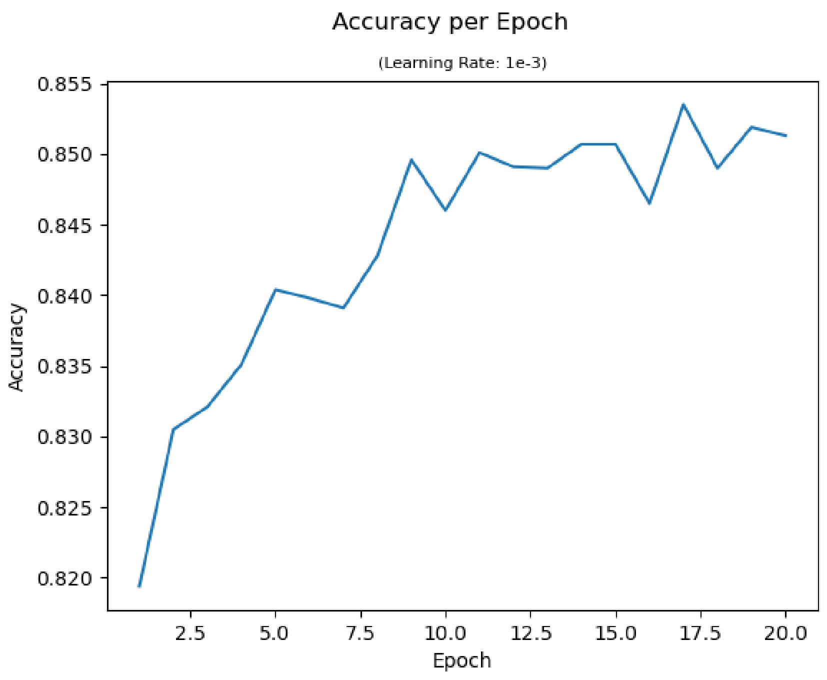


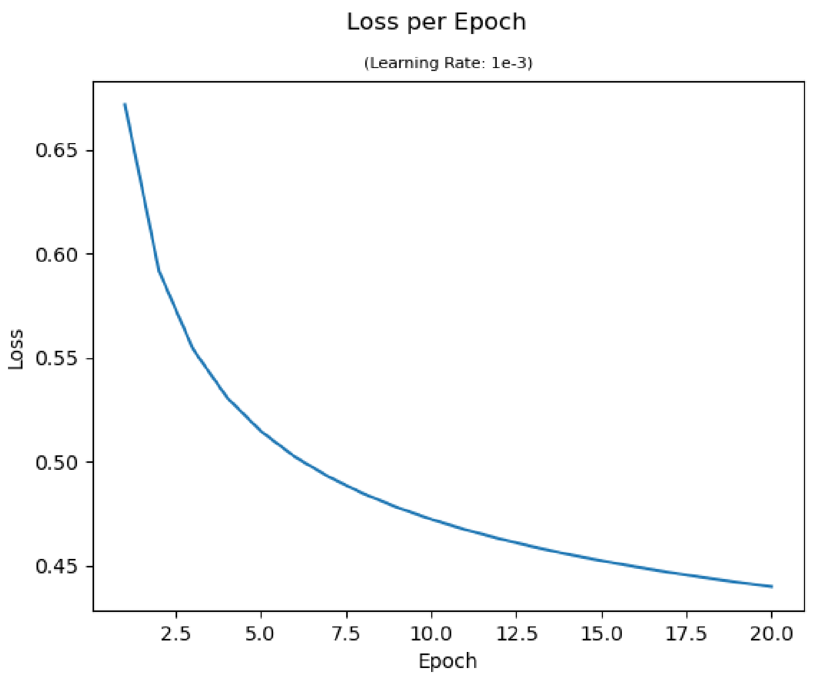
Question 4

4.1

After tuning training the Logistic Regression model for 20 epochs, with learning rates of $\{0.001, 0.01, 0.1\}$, we concluded that the best learning rate for this problem is 0.001. This learning rate granted the lowest loss values and the highest accuracy scores.

Below, we present the accuracy and loss plots for this learning rate:


Accuracy per Epoch
(Learning Rate: 1e-3)
Accuracy
Epoch
0.820
0.825
0.830
0.835
0.840
0.845
0.850
0.855
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

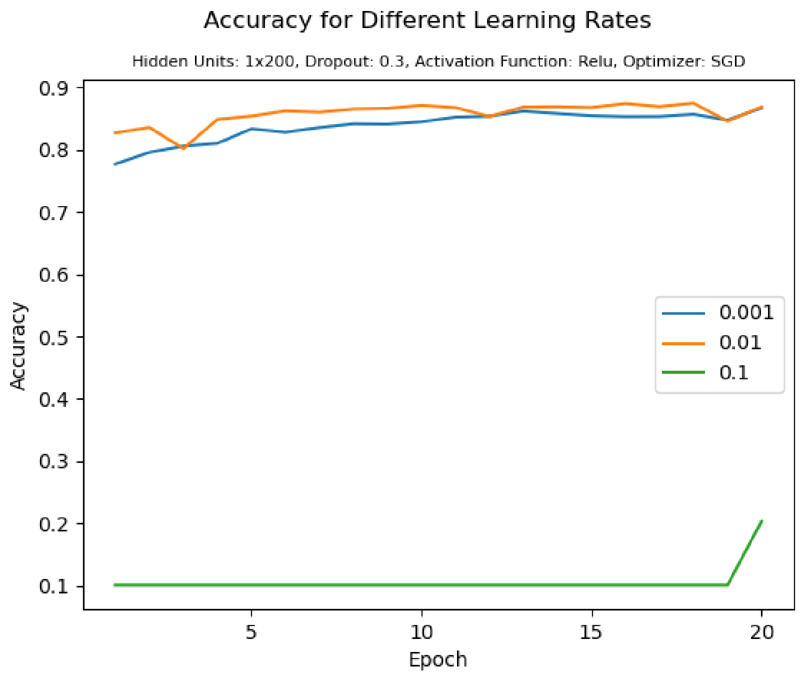

Loss per Epoch
(Learning Rate: 1e-3)
Loss
Epoch
0.45
0.50
0.55
0.60
0.65
2.5
5.0
7.5
10.0
12.5
15.0
17.5
20.0

4.2

For this experiment, we tuned the hyperparameters one by one, while leaving the others at their default value, and compared the different hyperparameter values using the attained accuracy in the validation data set. For each tuned hyperparameter, we generated a plot that helped comparing the results, and a table that frames the accuracy range and the final accuracy.

We discuss our results hyperparameter by hyperparameter:

Learning Rate



Parameters	Min. Acc.	Max. Acc.	Final Acc.
Learning Rate = 0.1	0.101	0.203	0.203
Learning Rate = 0.01	0.802	0.875	0.868
Learning Rate = 0.001	0.777	0.867	0.867

From the plot above, we can see that the configuration with Learning Rate of 0.1 (‘LR=0.1‘) reported accuracy scores much lower than the other two configurations. Moreover, it got a constant accuracy of 0.101 for every epoch except for the last one. We hypothesize that the low accuracy results are due to the learning rate being too high, which may make the model adjusting its parameters too much, in each SGD step. Hence, we decided to test the outlier configuration for

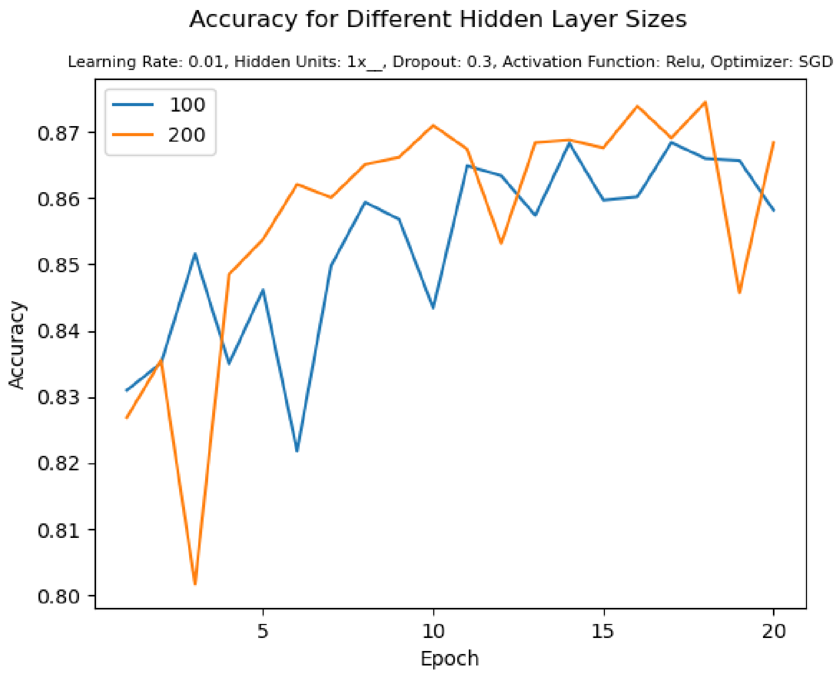
50 epochs, to analyse what happened to the accuracy. We observed that the accuracy varied between 0.101 and 0.231, reaching the final epoch with a value of 0.170 and a final test accuracy of 0.173. From this observation, we can conclude that **the learning rate of 0.1 is too high for this particular problem**, as lower learning rates present better results. We could not find any explanation for having constant accuracy during the first epochs, for this configuration.

When it comes to pick the best Learning Rate, we can observe that the configuration with ‘LR=0.01‘ presented the highest minimum, maximum and final accuracy scores. Moreover, we can observe that, in the first epochs, ‘LR=0.001‘ got lower accuracy results than ‘LR=0.01‘, which suggests that the lower learning rate is learning more slowly, which makes sense. For this reason, **we chose ‘LR=0.01‘ for the final configuration**.

Nonetheless, we cannot say for sure that 0.01 is the best Learning Rate value for this problem, since both ‘LR=0.01‘ and ‘LR=0.001‘ configurations got very similar results. In fact, when experimenting with other configuration combinations, we got higher accuracy results with a lower ‘LR‘:

Parameters	Min. Acc.	Max. Acc.	Final Acc.
Learning Rate = 0.1	0.822	0.868	0.858
Learning Rate = 0.01	0.820	0.878	0.878

Hidden Size

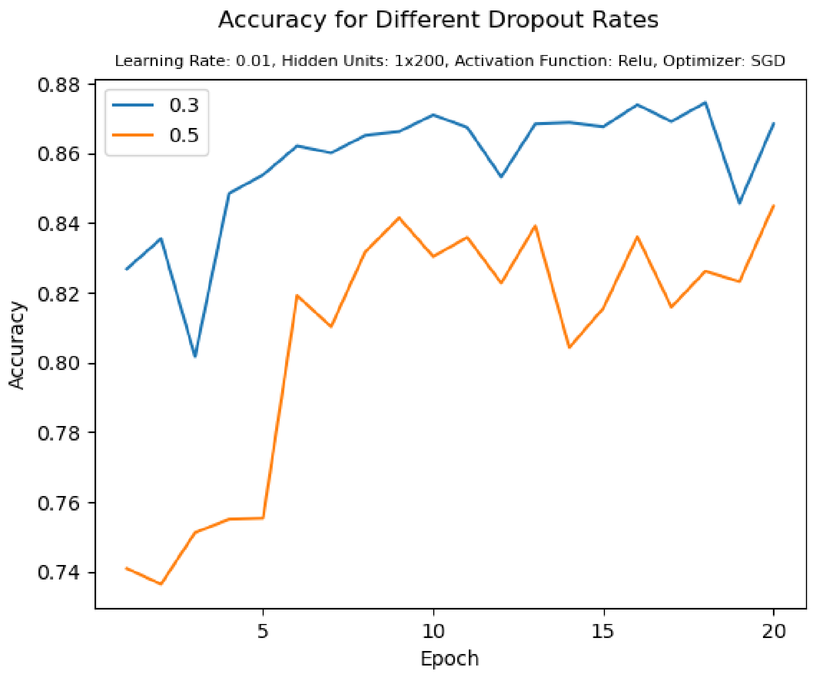


Parameters	Min. Acc.	Max. Acc.	Final Acc.
Hidden Size = 100	0.822	0.868	0.858
Hidden Size = 200	0.802	0.875	0.868

We could not identify a big difference in the performance these configurations, as the one that has the greater accuracy varies in different epochs. We opted to **choose the ‘HS=200‘ configuration as it yielded higher maximum and final accuracy.**

Note that choosing a smaller Hidden Layer would also be a valid option since it would require smaller weight matrices and would need to perform less computations.

Dropout Probability



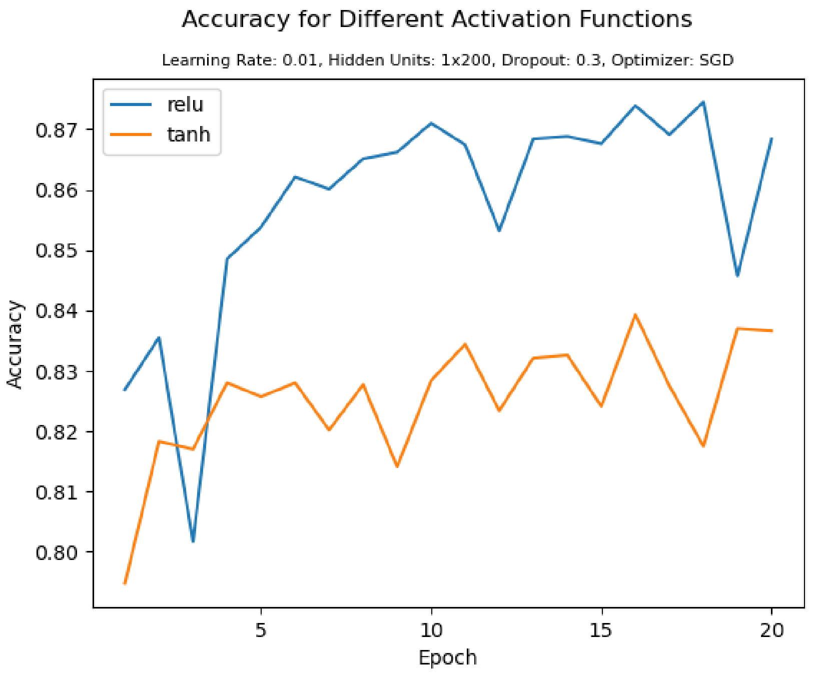
Parameters	Min. Acc.	Max. Acc.	Final Acc.
Dropout = 0.3	0.802	0.875	0.868
Dropout = 0.5	0.741	0.845	0.845

We can observe that the ‘Dropout=0.3‘ configuration has higher accuracies than the ‘Dropout=0.5‘ configuration in every epoch, hence it is a good candidate for the best value for the Dropout.

We cogitate that this may happen because a higher Dropout probabilities will deactivate more units during the feedforward, which may lead to a slower learning curve.

After testing the configuration ‘Dropout=0.5‘ for 50 epochs, we observed that the maximum and final accuracy values were 0.856 and 0.848, respectively. This is still lower than the values got by the other configuration.

Activation Function



Parameters	Min. Acc.	Max. Acc.	Final Acc.
Activation = ‘relu‘	0.802	0.875	0.868
Activation = ‘tanh‘	0.795	0.839	0.837

We can observe that, with the exception of epoch 3, the ‘Activation=relu‘ configuration got higher accuracy results than the ‘Activation=tanh‘ activation. Moreover, the gap between the accuracy of both configurations is larger than the one in other hyperparameters. This indicates that the ‘relu‘ Activation function might be the best suited for this problem.

After researching about this topic, we found that

”Due to vanishing gradient problem,

...

> sigmoid and tanh functions are avoided;
>
> ...
>
> The ReLU function is the most widely used function and performs better than other activation functions in most of the cases" Activation Functions in Neural Networks[6]

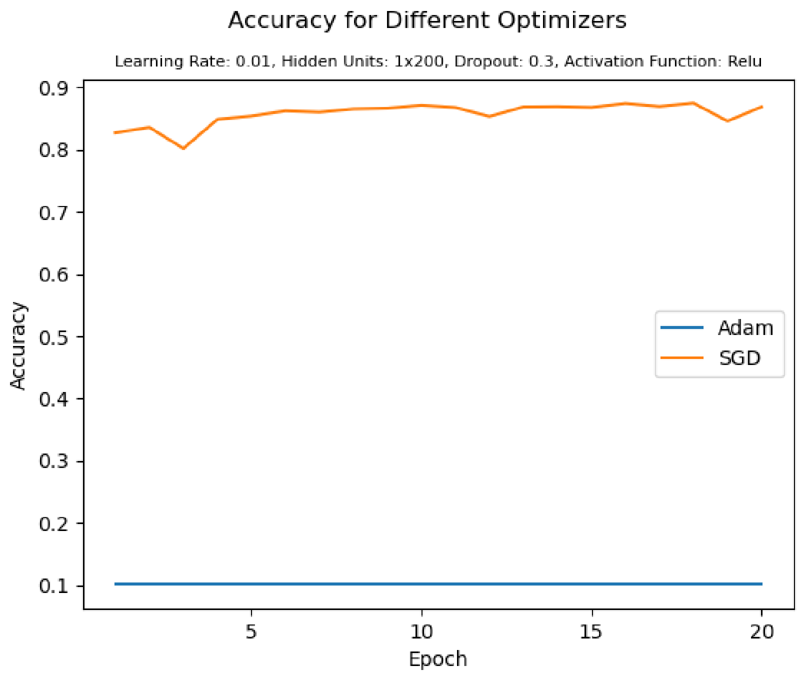
and that

> "A property of the tanh function is that it can only attain a gradient of 1, only when the value of the input is 0, that is when x is zero. This makes the tanh function produce some dead neurons during computation.
>
> ...
>
> This limitation of the tanh function spurred further research in activation functions to resolve the problem, and it birthed the rectified linear unit (ReLU) activation function." Activation Functions: Comparison of Trends in Practice and Research for Deep Learning[7]

These conclusions suggest that the 'ReLU' activation function is more likely to yield better accuracy scores, which is verified in the results of this experiment.

[6] `https://www.ijeast.com/papers/310-316,Tesma412,IJEAST.pdf`
[7] `https://arxiv.org/pdf/1811.03378.pdf`

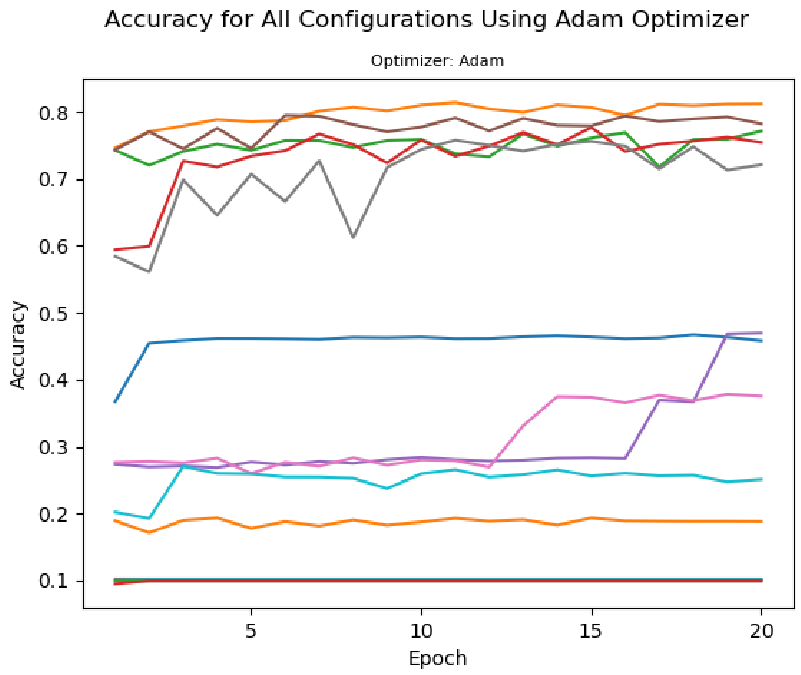
Optimizer



Parameters	Min. Acc.	Max. Acc.	Final Acc.
Optimizer = ‘Adam‘	0.101	0.101	0.101
Activation = ‘SGD‘	0.802	0.875	0.868

The first interesting result is that the accuracy of the ‘Adam‘ optimizer is constant during every epoch and has much lower accuracy compared to ‘SGD‘. This may hint that ‘Adam‘ may not be the best optimizer for this problem, however we cannot generalize this result for other scenarios.

We further investigated this result and launched several tests, one for each hyperparameter combination that used the ‘Adam‘ optimizer, and got the following results:



This exercise showed that some configurations still had low accuracy scores, but 5 configurations managed to have accuracy scores above 0.500. The best registered configuration for the ‘Adam‘ optimizer had ‘LR=0.001‘, ‘HS=100‘, ‘Dropout=0.3‘, ‘Activation=tanh‘, ‘Optimizer=adam‘, and ‘HL=1‘. This configuration got accuracy values ranging between 0.746 and 0.814, with a final accuracy of 0.812.

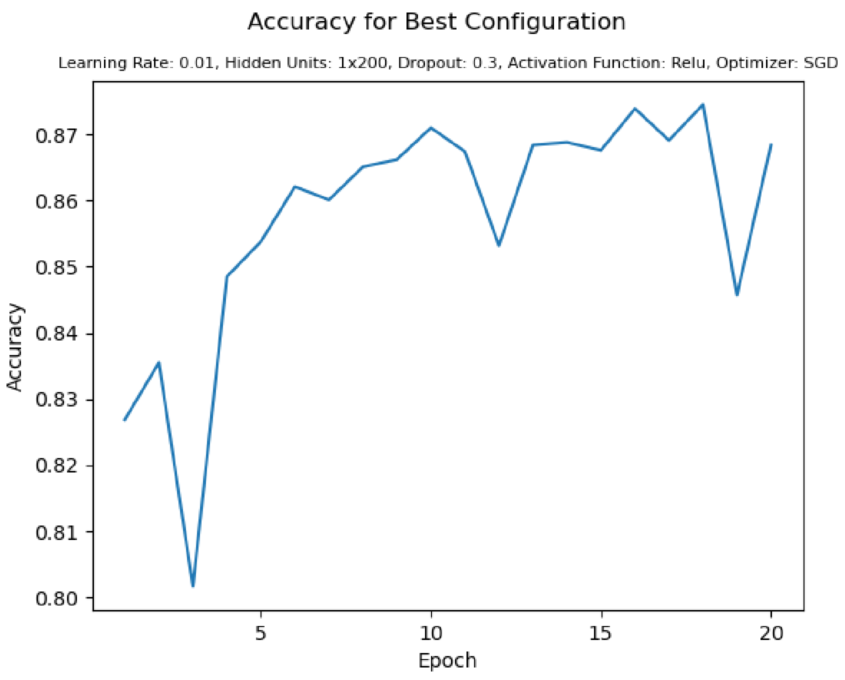
Interestingly enough, we could also notice that, from the top 10 configurations that use the ‘Adam‘ optimizer, only one of them used the ‘ReLU‘ activation function, bein ranked in 3rd place with accuracy values ranging between 0.718 and 0.771, with a final accuracy of 0.771.

”Best” Configuration

After analyzing all the hyperparameters, we hypothesize the best configuration is:

Parameter	Value
Learning Rate	0.01
Hidden Size	200
Dropout	0.3
Activation Function	ReLu
Optimizer	SGD

After running this configuration we got the following result:



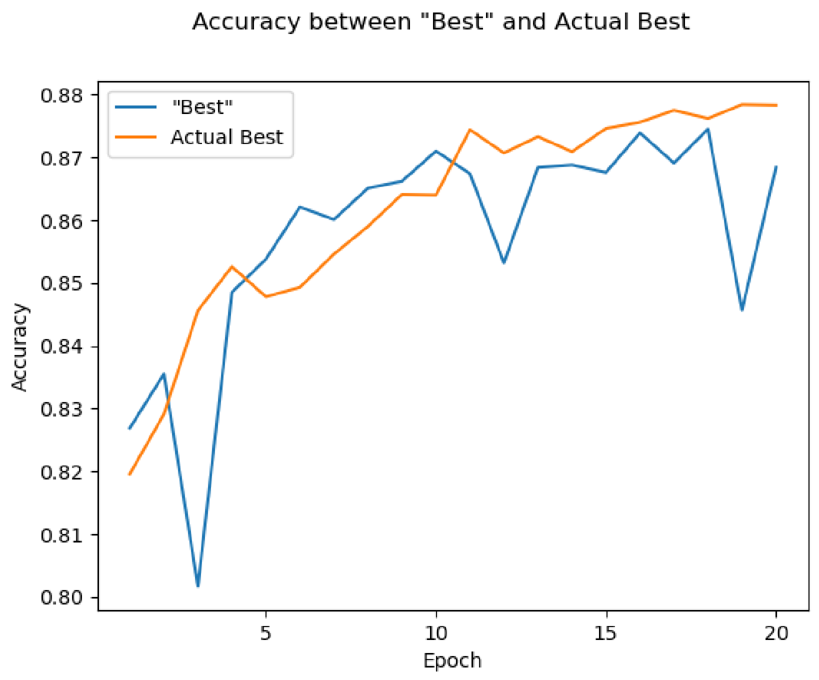
Since the best hyperparameters coincided always with the default parameters, the best configuration ended up being the default one. For this reason, the accuracy obtained in this experiment is the same as the accuracy we got in every other test, with the default value for the tuned hyperparameter.

However, the extra experiments made when tuning the Learning Rate and the Optimizer suggested that we could have better accuracy scores for other hyperparamenter combinations. For this reason, we tested the accuracy for every combination and exported those values to a file (available in the **data/** directory). We also developed a helper script (**graph.py**) to easily visualize this information.

We concluded that the best configuration for this problem and for this number of epochs is:

Parameter	"Best"	Actual Best
Learning Rate	0.01	**0.001**
Hidden Size	200	**100**
Dropout	0.3	0.3
Activation Function	ReLU	ReLU
Optimizer	SGD	SGD

We can compare the accuracies of both configurations in the following graph:



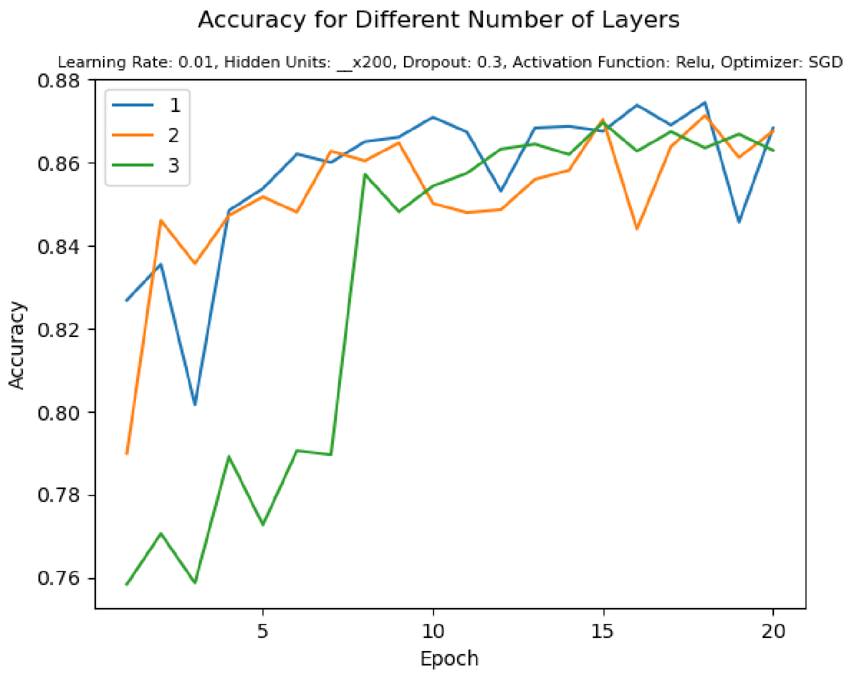
Parameters	Min. Acc.	Max. Acc.	Final Acc.
”Best”	0.802	0.875	0.868
Actual Best	0.820	0.878	0.878

We can see that this configuration got an accuracy 1% higher in the ”Actual Best” configuration.

The fact that the best configuration does not correspond to the combination of the best hyperparameter individually may indicate that the hyperparameters influence each other. This hypothesis is supported by the results obtained when experimenting different configurations for the ‘Adam‘ optimizer.

4.3

After executing the ”Best” configuration with 1, 2 and 3 hidden layers, we got the following results:



Parameters	Min. Acc.	Max. Acc.	Final Acc.
1 Layer	0.802	0.875	0.868
2 Layers	0.790	0.871	0.868
3 Layers	0.758	0.870	0.863

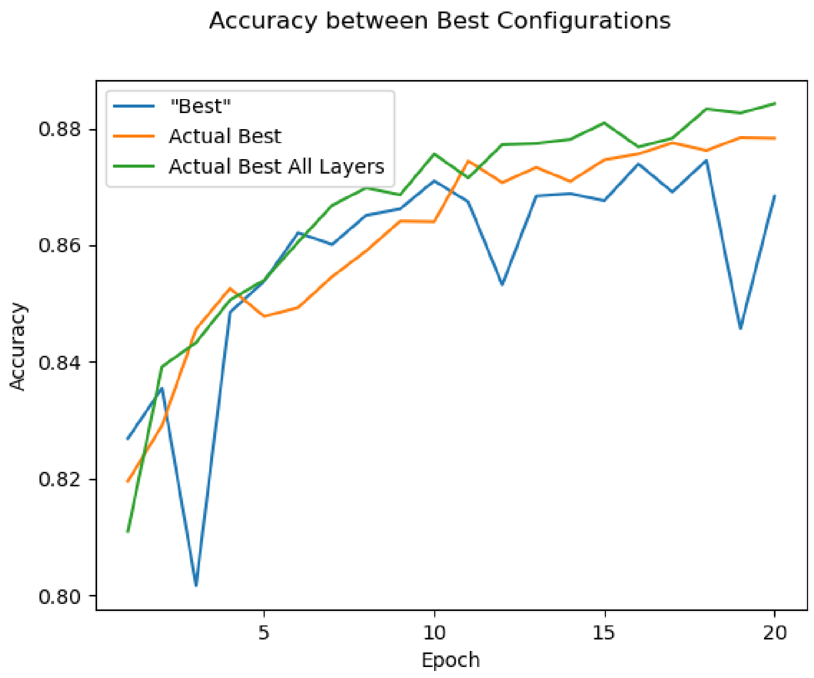
Although having the configuration with three layers took longer to learn, all of them finished the experiment with a similar accuracy score. Moreover, every configuration is the higher scorer in, at least, one epoch.

If we were to choose the best, **we would pick the configuration with one layer for three reasons**: First, its accuracy scores varied between the larger of the minimum and the larger of the maximum; Second, having fewer layers leads to fewer computations and possibly faster results; and Third, it is the top scorer in more than 50% of the epochs.

The results gotten in the previous question made us look for the best configuration considering different network depths. We called it ‘Actual Best All Layers‘ and below we can see its configuration:

Parameter	”Best”	Actual Best	Actual Best All Layers
Learning Rate	0.01	**0.001**	**0.001**
Hidden Size	200	**100**	200
Dropout	0.3	0.3	0.3
Activation Function	ReLU	ReLU	ReLU
Optimizer	SGD	SGD	SGD
Hidden Layers	1	1	**2**

We got the following results:



Parameters	Min. Acc.	Max. Acc.	Final Acc.
”Best”	0.802	0.875	0.868
Actual Best	0.820	0.878	0.878
Actual Best All Layers	0.811	0.884	0.884

We can see that this last configuration has higher accuracy scores in almost every epoch.