Instituto Superior Técnico

Master Degree in Electrical and Computer Engineering

Deep Learning

2021/2022 - 2nd Quarter

Group 30

Tiago Franco Pinto nº93187 tiagofrancopinto@tecnico.ulisboa.pt
Tomás Pimenta nº93195 tomas.pimenta@tecnico.ulisboa.pt

Faculty: Mário Figueiredo

16 of November 2021

Question 1

1.1

Using the common rules of differentiation (linearity and exponential), we can easily deduce the sigmoid's derivative as:

$$\sigma'(z) = \frac{d}{dz}\sigma(z) = \frac{d}{dz}(\frac{1}{1+e^{-z}}) = \frac{d}{dz}(1+e^{-z})^{-1} = -(1+e^{-z})^{-2} \times \frac{d}{dz}(1+e^{-z}) =$$

$$= -(1+e^{-z})^{-2} \times \frac{d}{dz}(e^{-z}) = -(1+e^{-z})^{-2} \times e^{-z} \times \frac{d}{dz}(-z) = (1+e^{-z})^{-2} \times e^{-z}$$

$$= \frac{e^{-z}}{(1+e^{-z})^2} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}}{1+e^{-z}} = \frac{1}{1+e^{-z}} \cdot \frac{e^{-z}+1-1}{1+e^{-z}} =$$

$$= \frac{1}{1+e^{-z}} \cdot (\frac{1+e^{-z}}{1+e^{-z}} - \frac{1}{1+e^{-z}}) = \frac{1}{1+e^{-z}} \cdot (1 - \frac{1}{1+e^{-z}}) = \sigma(z)(1-\sigma(z))$$

1.2

$$L(z; y=1) = -log(\sigma(z)) \tag{1}$$

$$\frac{d}{dz}L(z; y=1) = \frac{-\frac{d}{dz}\sigma(z)}{\sigma(z)} = \frac{-\sigma(z)(1-\sigma(z))}{\sigma(z)} = -1+\sigma(z) \tag{2}$$

$$\frac{d}{dz}(-1+\sigma(z)) = \frac{d}{dz}\sigma(z) = \sigma(z)(1-\sigma(z)) \tag{3}$$

The second derivative of the binary logistic loss function is positive in all its domain, because σ is a logistic function defined between 0 and 1. Therefore the logistic loss function graph is convex.

1.3

The entries of a Jacobian Matrix take two forms: one for the diagonal entries and one for the off diagonal entries. The entries for the diagonal entries (j,j) take the following form:

$$\frac{\partial[softmax(z)]_j}{\partial z_j} = \frac{\sum_k e^{z_k}e^{z_j} - e^{z_j}e^{z_j}}{(\sum_k e^{z_k})^2} = \frac{e^{z_j}}{\sum_k e^{z_k}} - \frac{e^{2z_j}}{(\sum_k e^{z_k})^2} = [softmax(z)]_j - [softmax(z)]_j^2$$

On the other hand, the entries for the off diagonal entries (j, k) take the following form:

$$\frac{\partial [softmax(z)]_j}{\partial z_k} = \frac{\sum_k e^{z_k} \cdot 0 - e^{z_j} e^{z_k}}{(\sum_k e^{z_k})^2} = -\frac{e^{z_j}}{\sum_k e^{z_k}} \cdot \frac{e^{z_k}}{\sum_k e^{z_k}} = -[softmax(z)]_j \cdot [softmax(z)]_k$$

We can easily notice that the derivatives always depend on the softmax function itself. Considering $[softmax(z)]$ as $s(z)$ to help the representation, we can write the Jacob Matrix as:

$$J_x(s) = \begin{bmatrix} s_0 - s_0^2 & -s_0 s_1 & ... & -s_0 s_K \\ -s_1 s_0 & s_1 - s_1^2 & ... & -s_1 s_K \\ ... & ... & ... & ... \\ -s_K s_0 & ... & ... & s_K - s_K^2 \end{bmatrix}$$

1.4

The multinomial logistic loss is defined as

$$L(z; y = j) = -log[softmax(z)]_j = log\left[\frac{\sum_k e^{z_k}}{e^{z_j}}\right] \tag{4}$$

The entries of the Jacobian Matrix, just like in the previous question, take two forms: one for the diagonal entries and one for the non diagonal entries.

For the diagonal entries k=j, thus the gradient is computed the following way:

$$\frac{\partial L(z; y = j)}{\partial z_j} = \frac{\frac{e^{z_j} e^{z_j} - \sum_k e^{z_k} e^{z_j}}{(e^{z_j})^2}}{\frac{\sum_k e^{z_k}}{e^{z_j}}} = \frac{e^{z_j} - \sum_k e^{z_k}}{\sum_k e^{z_k}} = softmax(z)_j - 1 \tag{5}$$

For the non diagonal entries $k \neq j$, thus the gradient is computed the following way:

$$\frac{\partial L(z; y = j)}{\partial z_k} = \frac{\frac{e^{z_k} e^{z_j} - 0}{(e^{z_j})^2}}{\frac{\sum_k e^{z_k}}{e^{z_j}}} = \frac{e^{z_k}}{e^{z_j}} = softmax(z)_k \tag{6}$$

The derivatives of the logistic function depend on the softmax function itself. Considering softmax(z)= s(z) to help with representation, the Jacob Matrix is:

$$J_x(s) = \begin{bmatrix} s_0 - 1 & s_1 & ... & s_K \\ s_0 & s_1 - 1 & ... & s_K \\ ... & ... & ... & ... \\ s_0 & ... & ... & s_K - 1 \end{bmatrix} \tag{7}$$

Just like the Jacobian the Hessian Matrix take two forms, one with diagonal entries and one for non diagonal entries.

For the diagonal entries k=j, the second derivative is computed the following way:

$$\frac{\partial(softmax(z)_j - 1)}{\partial z_j} = \frac{\partial[softmax(z)]_j}{\partial z_j} = [softmax(z)]_j - [softmax(z)]_j^2 \tag{8}$$

For the non diagonal entries $k \neq j$, the second derivative is computed the following way:

$$\frac{\partial[softmax(z)]_j}{\partial z_k} = -[softmax(z)]_j \cdot [softmax(z)]_k \tag{9}$$

This means the Hessian matrix of the multinominal logistic loss is the same as the Jacobian matrix if the softmax function.

$$H_x(s) = \begin{bmatrix} s_0 - s_0^2 & -s_0 s_1 & ... & -s_0 s_K \\ -s_1 s_0 & s_1 - s_1^2 & ... & -s_1 s_K \\ ... & ... & ... & ... \\ -s_K s_0 & ... & ... & s_K - s_K^2 \end{bmatrix}$$

In order to prove the multinomial logistic loss function is convex. The Hessian matrix must be positive semi-definite. Let $\mathbf{H}$ be the Hessian,

$$\mathbf{x}^\intercal \mathbf{H} \mathbf{x} = \sum_{i=0}^{K} \sum_{j=0}^{K} \mathbf{H}_{i,j} x_i x_j,$$

where $\mathbf{x} \in \mathbb{R}^K$. Expanding the expression, leads to

$$(s_0 - s_0^2)x_0^2 - 2\mathrm{s}_0\mathrm{s}_1 x_0 x_1 - 2\mathrm{s}_0\mathrm{s}_1 x_0 x_2 + ... + (\mathrm{s}_1 - \mathrm{s}_1^2)x_K^2,$$

After some algebraic manipulation the expression can be written as

$$(x_0 - x_1)^2 \mathrm{s}_0\mathrm{s}_1 + (x_0 - x_2)^2 \mathrm{s}_0\mathrm{s}_2 + ... + (x_K - x_{K+1})^2 \mathrm{s}_K\mathrm{s}_{K+1}$$

Given that s$(z) \geq 0$, for any $\mathbf{x} \in \mathbb{R}^K$, $\mathbf{x}^\intercal\mathbf{H}\mathbf{x} \geq 0$. All eigenvalues are non-negative ($\lambda_i(\mathbf{H}) \geq 0$), this means the Hessian is a positive semi-definite matrix $\mathbf{H} \succeq 0$ which proves the multinominal logistic loss is convex.

1.5

The Multinomial Logistic Loss is given by:

$$L(\mathbf{W}, \mathbf{b}) = -log(softmax(\mathbf{w_j}\phi(\mathbf{x}) + \mathbf{b}) \tag{10}$$

As we showed before, this function is convex with respect to z. We need now to prove that a twice-differentiable convex function of an affine function is a convex function. Considering $f : \mathbb{R}^m \to \mathbb{R}$ is convex and $g : \mathbb{R}^n \rightarrow \mathbb{R}^m, g(u) = Au+c$ is an affine map. This leads us to $h(u) = f(g(u)) = f(Au+b)$, where h is convex in $\mathbb{R}^m$. The gradient of h with respect to u is:

$$\nabla_u h(u) = A^T \nabla_u f(Au+b) \in \mathbb{R}^m$$

and the Hessian of h with respect to u is:

$$\nabla_u^2 h(u) = A^T \nabla_u^2 f(Au+b) A \in \mathbb{R}^m.$$

Knowing that f is a convex function $\nabla_x^2 f(x) A \geq 0$ (it will be a positive semidefinite matrix for every x). Then,

$$\mathbf{z^T} \nabla_u^2 h(u) \mathbf{z} = \mathbf{z^T} A^T \nabla_u^2 f(Au+b)(\mathbf{Az}) = (\mathbf{Az})^T \nabla_u^2 f(Au+b)(\mathbf{Az}) \geq 0$$

,

proving that the hessian is also a positive semidefinite matrix. Given that the multinomial logistic loss is given by 10, we can assume that it is a function of linear affine functions in $(\mathbf{W}, \mathbf{b})$, hence, it is convex.

Generally this isn't true for non linear $\mathbf{z}$ due to the fact that an affine function is a composition of linear functions, and this would lead to h not being affine and thus not verifying the conditions in this proof to be convex.

Question 2

2.1

Considering the squared error loss

$$L(z; y) = \frac{1}{2}\|z-y\|_2^2 = \frac{1}{2}(z-y)^\intercal(z-y),$$

where $z = W\phi(x) + b$, L is convex with respect to (W, b).

The operation $z - y$ can be defined as an affine map, as seen in the previous question. The norm $\|.\|_2$ is a convex function $f : \mathbb{R}^m \rightarrow \mathbb{R}$, thus, the resulting composition is convex in $\mathbb{R}^m$. The square operation is also a convex function $g : \mathbb{R}^n \rightarrow \mathbb{R}$, given that is a quadratic function. The composition $f \circ g$ will also result in a convex function. A product between a convex function and a constant is still a convex function.

2.2 a)

In this problem we aimed to predict the price-tag of the houses present in the Ames Housing dataset. To do so, we trained a simple Linear Regression Model for 150 epochs with a learning rate of 0.001. To evaluate the performance of the model, we plotted the Loss with regard to the epochs.

In the Linear Regression, the prediction process is simply the following:

$$y = Xw + b \tag{11}$$

However, in this case, the biases (b) are already in the input matrix (X) as a column. Regarding the training model, we used the Stochastic Gradient Descent to optimize the weights, thus we updated the weights according to the error of each input. The Stochastic Gradient Model follows

$$W^{(k+1)} = W^{(k)} - \eta \nabla_W L(W^{(k)}; (x_i, y_i)) \tag{12}$$

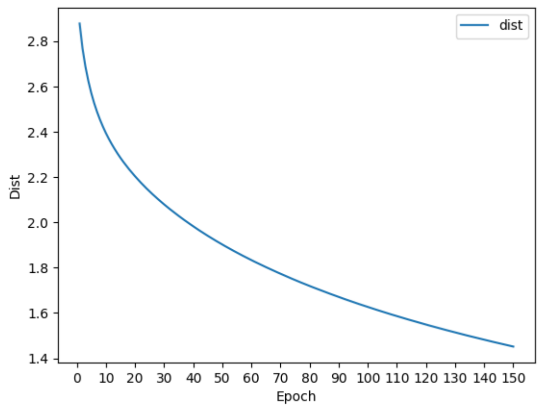
where η is the learning rate and the gradient of the loss is given by:

$$\nabla_W L(W^{(k)}; (x_i, y_i)) = x_i^T (x_i W - y_i). \tag{13}$$

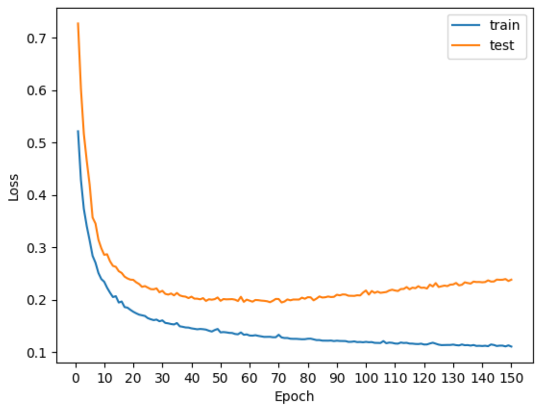
During this training process, the weights were constantly updated, and compared to a weight vector calculated analytically with the following expression (obtained from (11) with a regularization term added):

$$\hat{w} = (X^T X + I)^{-1} X^T y,$$

in which the regularization term (λI) is used to let us invert the matrix $X^T X$, as it is a singular matrix and therefore can't be inverted. This regularization is equivalent to ridge regression, however, we did not include in the loss as it was only used for numerical stability. As we can see in Figure 2 a), as the training process goes on, the weights get closer to the analytical solution (that represents the exact weights that map the training set features to its outputs). And as we can see, this approximation of the vectors is followed by a loss reduction, meaning that the predicted values are getting closer to the true output values.



(a) Distance between weight vectors evolution



(b) Loss evolution

Figure 2: Metrics obtained with Linear Regression

We can also observe in Figure 2 b) that the test's loss at around 70 epochs starts growing. This means that the model isn't generalising well and thus the model is performing better on the training set than on the test set. One way to fix this would be to use regularization methods such as Ridge and Lasso in this model.

2.2 b)

Instead of using a simple Linear Regression model, we now used a Neural Regressor. The implemented Neural Network has as input de features of the Ames Housing Dataset, has 1 hidden layer with 150 nodes (ReLU activation function) and a single output node. The forward propagation in this network is the following (in the given order):

$$z_1 = W_1 x_i + b_1 \tag{14}$$

$$h_1 = ReLU(z1) \tag{15}$$

$$\hat{y} = z2 = W_2 h_1 + b_2 \tag{16}$$

We used the Stochastic Gradient Descent once again, so the weights were updated again according to (12), and the biases were updated according to

$$b^{(k+1)} = b^{(k)} - \eta \nabla_b L(b^{(k)}; (x_i, y_i)). \tag{17}$$

However, this time, we have more than one weight matrix and we have activation functions, make it more complex to compute the desired gradients. To solve this problem, the chain derivative rule is commonly used, so we computed the gradients of the intermediate steps in order to be able to update the weights and biases. The first gradient to compute is ∇_{z_2}, and as it is the output

$$\nabla_{z_2} L(f(x,\theta); y) = \hat{y} - y_i = Xw + b - y_i \tag{18}$$

and amounts to the error. We can then compute the gradients of W_2 and b_2 (and use both (12) and (17) to update it)

$$\begin{cases} \nabla_{W_2} L(f(x,\theta); y) = \nabla_{z_2} L(f(x,\theta); y) h_1^T \\ \nabla_{b_2} L(f(x,\theta); y) \nabla_{z_2} L(f(x,\theta); y) \end{cases} \tag{19}$$

Continuing the back-propagation trough the intermediate values, we obtain the following gradients

$$\nabla_{h_1} L(f(x,\theta); y) = W_2^T \nabla_{z_2} L(f(x,\theta); y) \tag{20}$$

$$\nabla_{z_1} L(f(x,\theta); y) = \nabla_{h_1} L(f(x,\theta); y) \odot g'(z_1) = \nabla_{h_1} L(f(x,\theta); y) \odot ReLU'(z_1), \tag{21}$$

where the derivative of the ReLU function is 0 for negative values and 1 for positive values. Knowing the gradient with respect to z_1, we can update the weights and biases with the gradients obtained just like in (19):

$$\begin{cases} \nabla_{W_1} L(f(x,\theta); y) = \nabla_{z_1} L(f(x,\theta); y) x_i^T \\ \nabla_{b_1} L(f(x,\theta); y) = \nabla_{z_1} L(f(x,\theta); y) \end{cases} \tag{22}$$

The results we obtained are in Figure 3.

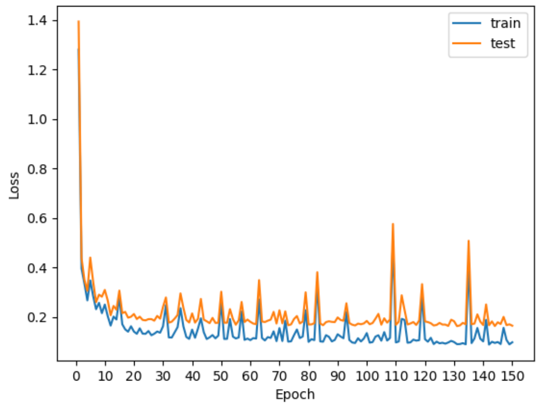


Figure 3: Loss obtained with Neural Network

As we can see, in both models reach losses between 0.1 and 0.2 in the training set, with the loss of the previous model being slightly lower. However, this new model performs better on the test set, as it's loss doesn't start increasing in the middle of the training. The main difference between this two models is that the Neural Regressor introduces non-linearity, and it is proving to be more efficient than the linear method. This tells us that the relation between the output and the features is not completely linear, and that is why the second model generalises better.

The other difference between this two models that is noticeable is that the first model as a smooth curve and the second one has a lot of spikes. As we introduced non-linearity, the SGD - Stochastic Gradient Descent - (that updates the weights each time for each sample's loss) may face an 'unlucky' sample where the loss is high, thus the spikes.

Question 3

3.1 a)

In this exercise we aimed to implement a model to a linear classifier for a simple image classification problem. We used the perceptron as the first model to solve this problem. The prediction, in both 3.1 exercises is done using

$$y = argmax(WX^T) \tag{23}$$

As we're solving a multiclass problem, the weights of the perceptron are updated in the following way: first we predict the output ($\hat{y}$) given one input, using (23) and then, if $\hat{y} \neq y$

$$\begin{cases} W_{y_i}^{(k+1)} = W_{y_i}^{(k)} - x_i \\ W_{\hat{y}}^{(k+1)} = W_{\hat{y}}^{(k)} + x_i \end{cases} \tag{24}$$

The accuracy we obtained is observable in Figure 4

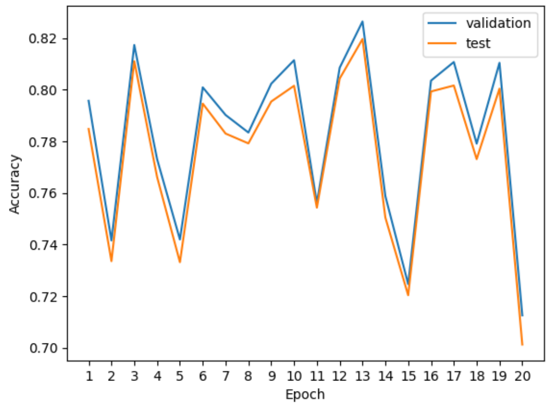


Figure 4: Perceptron Accuracy Evolution

As we can see, this model behaved poorly, not being able to converge. This is a great indicator that this problem is not linearly separable, as the perceptron can't solve these. The reason for the perceptron to behave like this is because it lacks non-linear activation functions.

3.1 b)

As the perceptron proved itself useless, we tried a new model: a logistic regression. A logistic regression differs from the perceptron as it has a non-linear activation function. When updating the weights, this model uses the softmax function to map the scores to probabilities. Then, it computes a one hot encoded vector of the expected label (y_i), where the column with the index that matches the label is 1 and the other elements of the row are 0. Finally, the weights are updated according to (12) where the gradient is given by:

$$\nabla_W L(W^{(k)}; (x_i, y_i)) = -(y_{onehot} - probabilities)x_i, \tag{25}$$

knowing that the probabilties are given by the softmax function (as it is a multiclass problem):

$$probabilities = \frac{e^{z_i}}{\sum_k e^{z_k}} = softmax(z_i) \tag{26}$$

The results we got are visible in Figure 5.

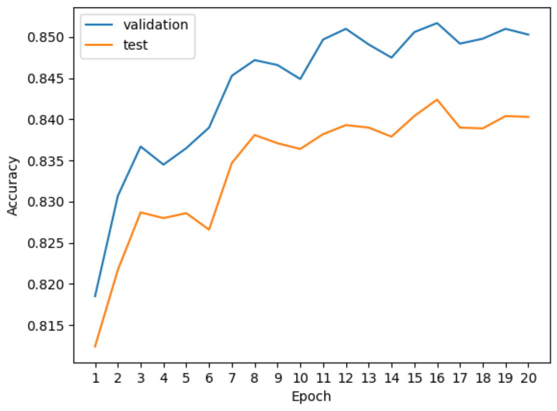


Figure 5: Logistic Regression Accuracy Evolution

As we can see, this model behave significantly better. However, the accuracy on the test set was noticeably lower than on the training set, leading us to the conclusion that the model didn't generalise that well.

3.2 a)

As we saw in 3.1 a), the perceptron algorithm fails to converge, highly suggesting that the problem in our hands is not linearly separable. On the other hand, a MLP with non-linear activation functions has the ability to respond to the inputs in a way that doesn't vary linearly, allowing itself to predict more efficiently non-linearly separable problems. Usually this non-linear activation-functions are used in intermediate hidden layers (tanh, ReLU, i.e.) or even in the output layer, to map the output to different classes (sigmoid, softmax, i.e).

However, if the activation functions of a MLP were linear functions instead of non-linear, it would behave like a single layer perceptron, as summing the MLP layers would result in another linear function, thus making the model unable to correctly predict non-linearly separable problems.

3.2 b)

In the last model of this exercise, we implemented a multi-layer perceptron with a single hidden layer (with ReLU activation functions and 200 hidden units).The forward propagation used was similar to the one used in exercise 2.2 b), however, now the output layer has a softmax activation function. This means that the forward propagation is given by

$$z_1 = W_1 x_i + b_1 \tag{27}$$

$$h_1 = ReLU(z1) \tag{28}$$

$$z2 = W_2 h_1 + b_2 \tag{29}$$

$$\hat{y} = h2 = softmax(z2) \tag{30}$$

Regarding the back-propagation, we start with the following:

$$\nabla_{z_2} L(f(x,\theta); y) = h2 - y_{onehot} \tag{31}$$

Then we can compute the gradients needed by the SGD to update the parameters W_2 and b_2.

$$\begin{cases} \nabla_{W_2} L(f(x,\theta); y) = \nabla_{z_2} L(f(x,\theta); y) h_1^T \\ \nabla_{b_2} L(f(x,\theta); y) \nabla_{z_2} L(f(x,\theta); y) \end{cases} \tag{32}$$

After that we need to compute the gradients necessary to update W_1 and b_1, so we proceed like this

$$\nabla_{h_1} L(f(x,\theta); y) = W_2^T \nabla_{z_2} L(f(x,\theta); y) \tag{33}$$

$$\nabla_{z_1} L(f(x,\theta); y) = \nabla_{h_1} L(f(x,\theta); y) \odot g'(z_1) = \nabla_{h_1} L(f(x,\theta); y) \odot ReLU'(z_1), \tag{34}$$

$$\begin{cases} \nabla_{W_1} L(f(x,\theta); y) = \nabla_{z_1} L(f(x,\theta); y) x_i^T \\ \nabla_{b_1} L(f(x,\theta); y) = \nabla_{z_1} L(f(x,\theta); y) \end{cases} \tag{35}$$

We can use both (12) and (17) to update the weights and bias.

The results we got are displayed in Figure 6.

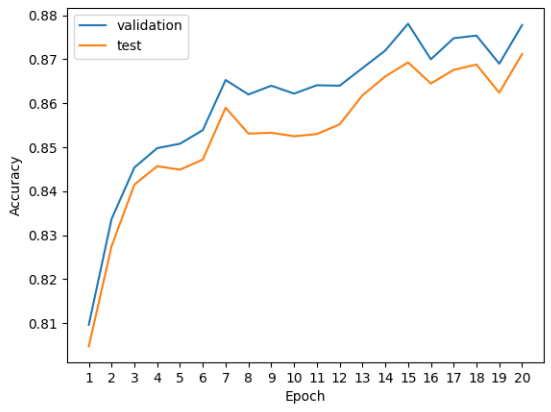


Figure 6: MLP Accuracy Evolution

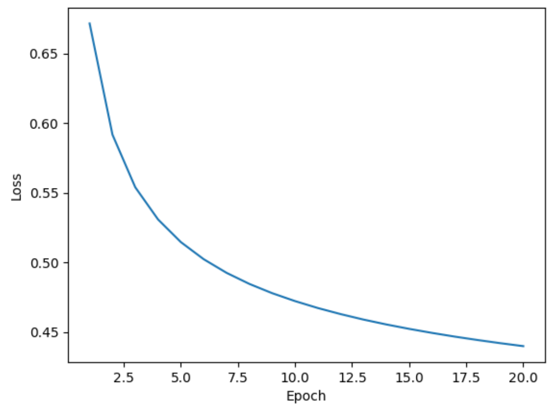
By observing Figure 6 we can conclude that this model is the best one. It converges to best accuracy and the performance on the test set is closer to the training set, meaning that it generalises better. This MLP has non-linear activation functions, unlike the perceptron and just like the logistic regression, however, it is a more complex net, fitting the problem better.

Question 4

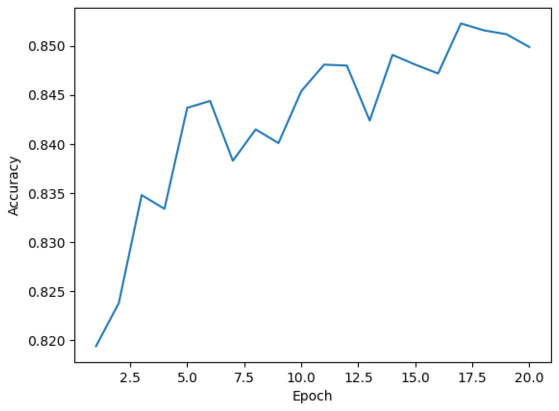
4.1

In this exercise, instead of computing the gradients by ourselves, we used the Pytorch library to solve the same problem as in Question 3.1 b). We used Logistic Regression models with 0.1, 0.01 and 0.001 learning rates, leaving the remaining values default. When initialising the model, we defined a single linear layer (meaning that there are no hidden layers, only the input and output layers). Even thought we used a Logistic Regression model, we did not implement a softmax activation function because the loss function was already set as cross-entropy, and this function doesn't expect normalised values. When looking at the validation set accuracy graphs, we quicly noticed that only the model with 0.001 learning rate converged, and therefore it is the best model.

The results we obtained for this model are the one in Figure 7.



(a) Logistic Regression Loss



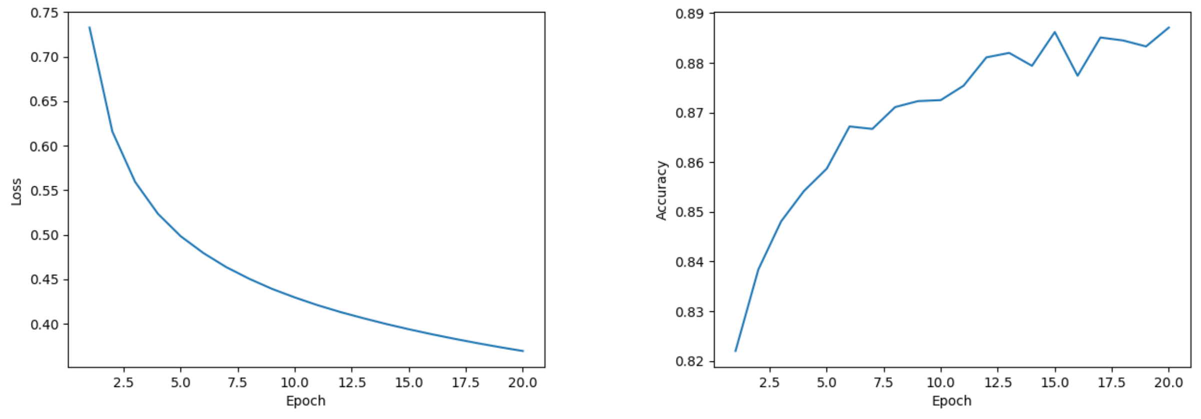
(b) Logistic Regression Accuracy

Figure 7: Metrics of the Logistic Regression (learning rate = 0.001)

The final accuracy on the test set was 0.8388.

4.2

In this exercise, we also used Pytorch, but now we used it to solve the problem in Question 3.2 b). However, we now used numerous tuning hyperparameters: learning rate, hidden layer size, dropout probability, activation function and optimizer. We always let the default values on except for one value, in order to test the influence of each hyperparameter. We used only one hidden layer, with ReLU activation functions. We did not include the softmax due to the same reason as before. When tuning the hyperparameters heuristically, by comparing the validation accuracy curves we noticed that the best configuration for the network was the one with a learning rate of 0.001. The results of this configuration are in Figure 8.



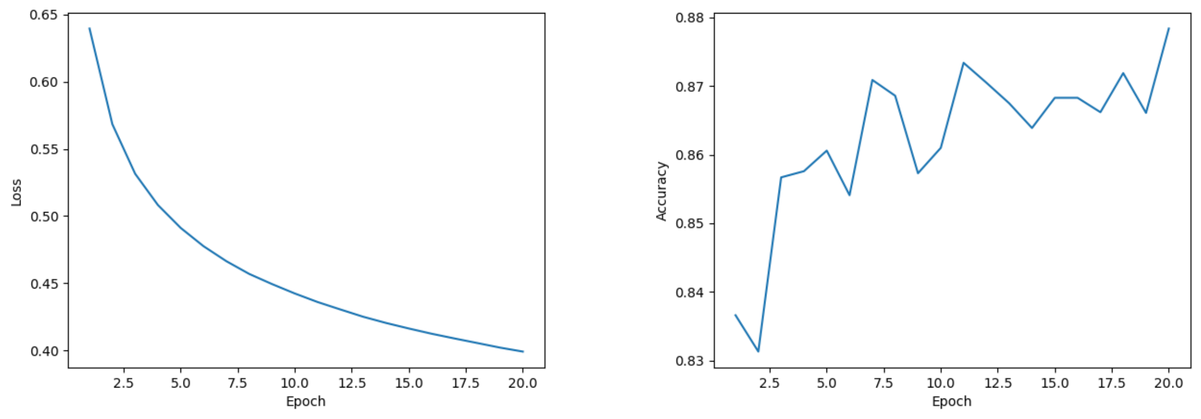
(a) MLP Loss (b) MLP Accuracy

Figure 8: Metrics of the MLP (learning rate = 0.001)

The final value for the accuracy on the test set was 0.8793.

4.3

In the final exercise, we repeated the same exercise as the one before, however, we used the default values with 2 or 3 hidden layers. We heuristically tuned this hyperparameter by comparing the accuracies on the validation set, and although the results we got for both of them were similar, the 2 hidden layers configuration got the edge. The results are in Figure 9.



(a) MLP Loss (b) MLP Accuracy

Figure 9: Metrics of the MLP (2 hidden layers)

The final test set accuracy was 0.8745.