Deep Learning - Homework 1

99222 - Frederico Silva, 99326 - Sebastião Carvalho

December 13, 2023

Contents

1 Question 1

Medical image classification with linear classifiers and neural networks.

1.1 Question 1.1

1.1.1 Question 1.1 a)

Answer After running the code, the following plot was generated:

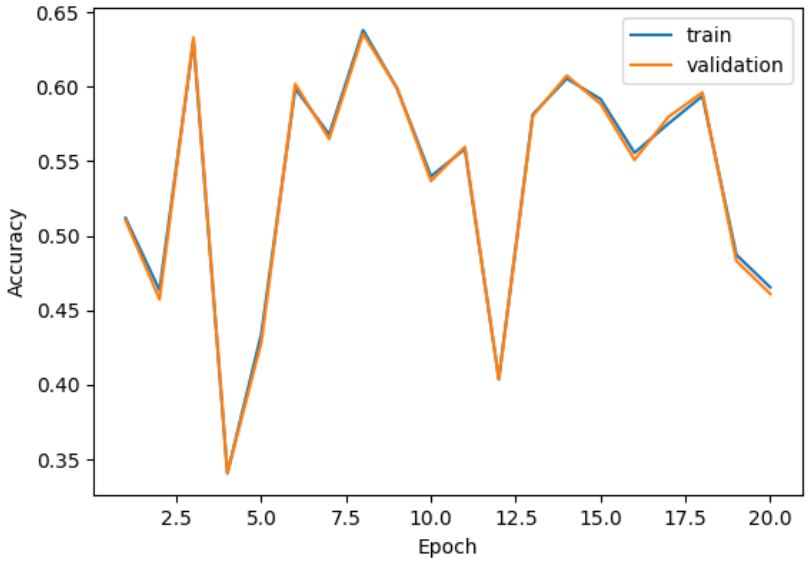

Figure 1: Perceptron Training and Validation Accuracy

The final test accuracy was 0.3422.

1.1.2 Question 1.1 b)

Answer After running the code, the following plots were generated for learning rates $\eta = 0.01$ and $\eta = 0.001$, respectively:

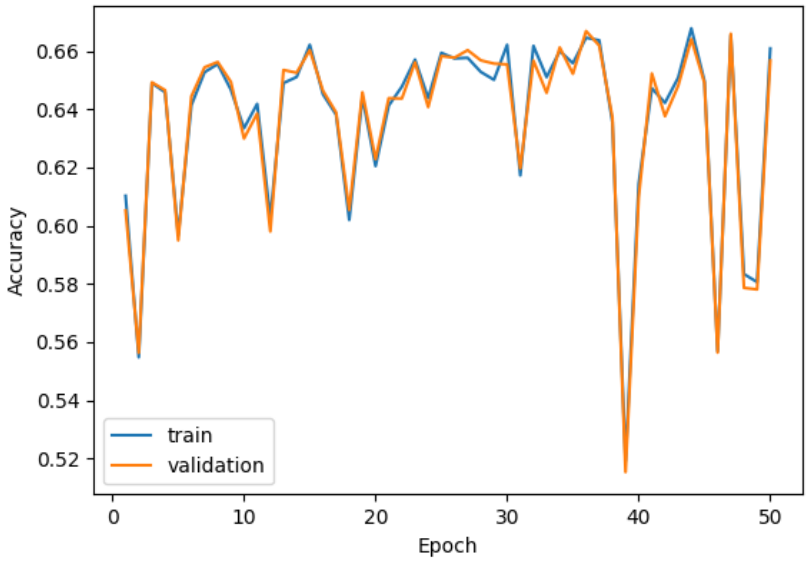

Figure 2: Logistic Regression Accuracy with Learning Rate $\eta = 0.01$

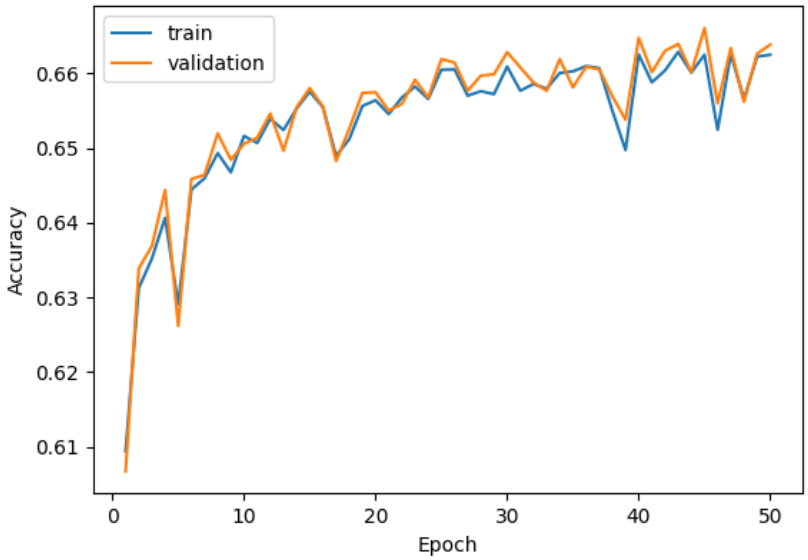

Figure 3: Logistic Regression Accuracy with Learning Rate $\eta = 0.001$

The final test accuracy was 0.5784 and 0.5936 for $\eta = 0.01$ and $\eta = 0.001$ respectively.
Comparing both charts, we can see that for $\eta = 0.001$, the accuracy increases more slowly, but it reaches a higher value than for $\eta = 0.01$. This is because for $\eta = 0.01$, the algorithm is taking too big steps, and it is not able to converge to a better solution. This can be seen in the chart for $\eta = 0.01$, near epoch 40, when the accuracy drops to an all time low.
Since the learning rate is too small for $\eta = 0.001$, the algorithm takes longer to reach higher accuracies, but it's also less oscillatory, and it is able to converge to a better solution.

1.2 Question 1.2

1.2.1 Question 1.2 a)

Answer The answer to this question mainly focus on two topics: how complex the models are, meaning their expressiveness, and how easy they are to train.
Regarding the expressiveness, logistic regression is a linear model that learns a single decision boundary to separate classes. When using pixel values as features, it treats each pixel independently and cannot capture complex patterns such as shapes or textures that might be essential for tasks like image classification.
On the other hand, a multi-layer perceptron can learn non-linear decision boundaries due to its hidden layers and non-linear activation functions like ReLU. Each layer can transform the feature space in a way that makes the data linearly separable by subsequent layers, allowing the MLP to capture complex patterns and relationships in the data.
When it comes to how easy it is to train this models, logistic regression, with its convex cost function, offers a straightforward training guaranteed to reach the global minimum, given enough time and proper learning hyperparameters. However, when talking of a multi-layer perceptron, the presence of multiple layers and non-linearities in an MLP makes the optimization landscape non-convex. There can be multiple local minima, saddle points, and plateaus. Finding the global minimum is not guaranteed, making the training process, most of the times, more complex.
In short, the claim is true. A logistic regression model is less expressive than a multi-layer perceptron with ReLU activations because it can only represent linear relationships, whereas MLPs can

capture non-linearities. Logistic regression models are easier to train because they involve convex optimization, unlike the non-convex problem of training MLPs.

1.2.2 Question 1.2 b)

Answer After running the code, the following plots was generated:

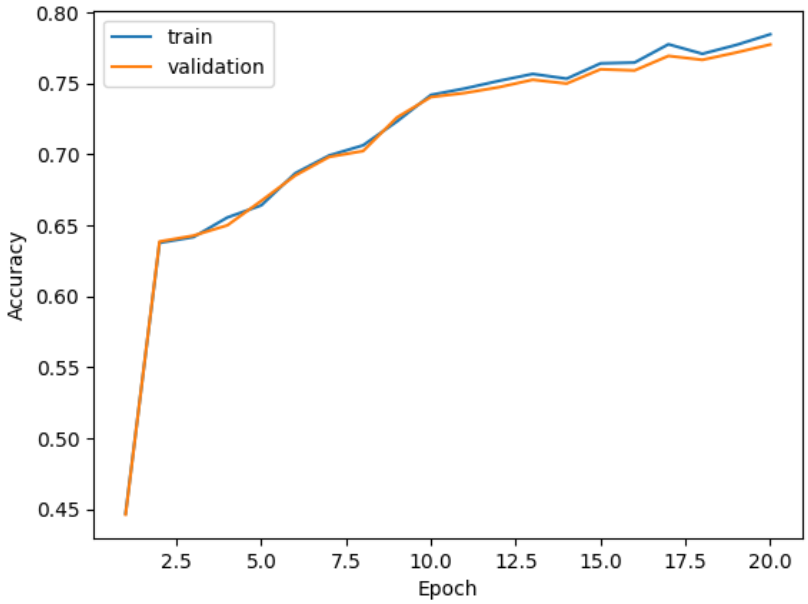

Figure 4: MLP Accuracy with Learning Rate $\eta = 0.001$

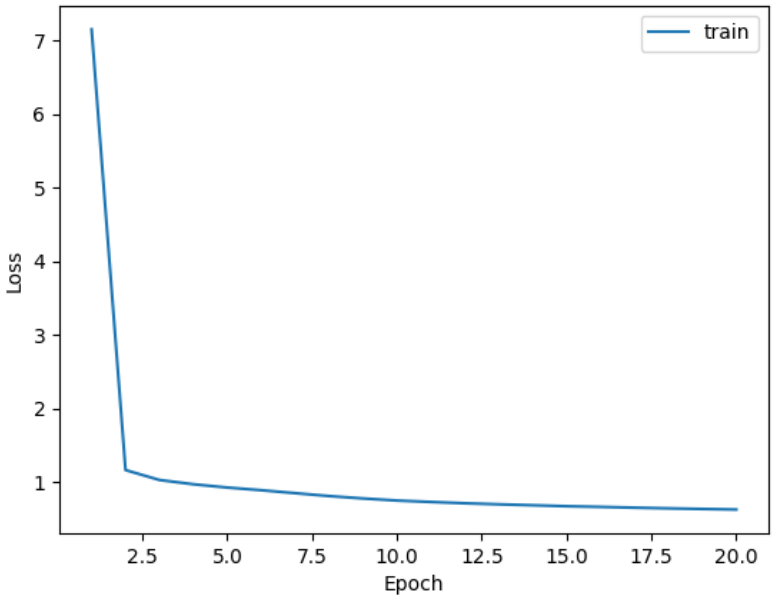

Figure 5: MLP Loss with Learning Rate $\eta = 0.001$

The final test accuracy was 0.7505.

2 Question 2

Medical image classification with an autodiff toolkit.

2.1 Question 2.1

Answer After running the code, the following plots were generated:

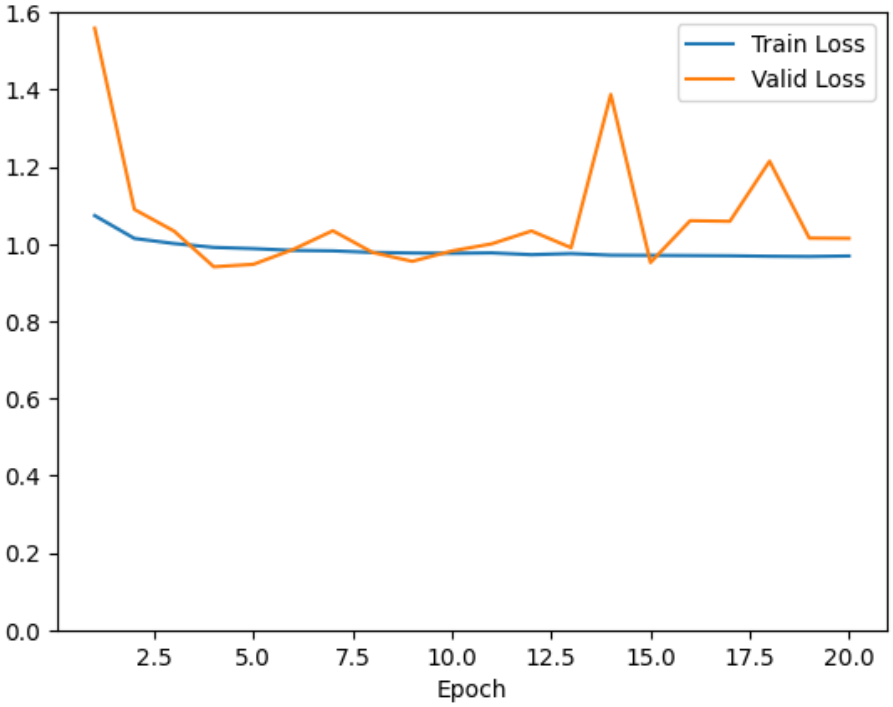

Figure 6: Training and validation loss for $\eta = 0.1$

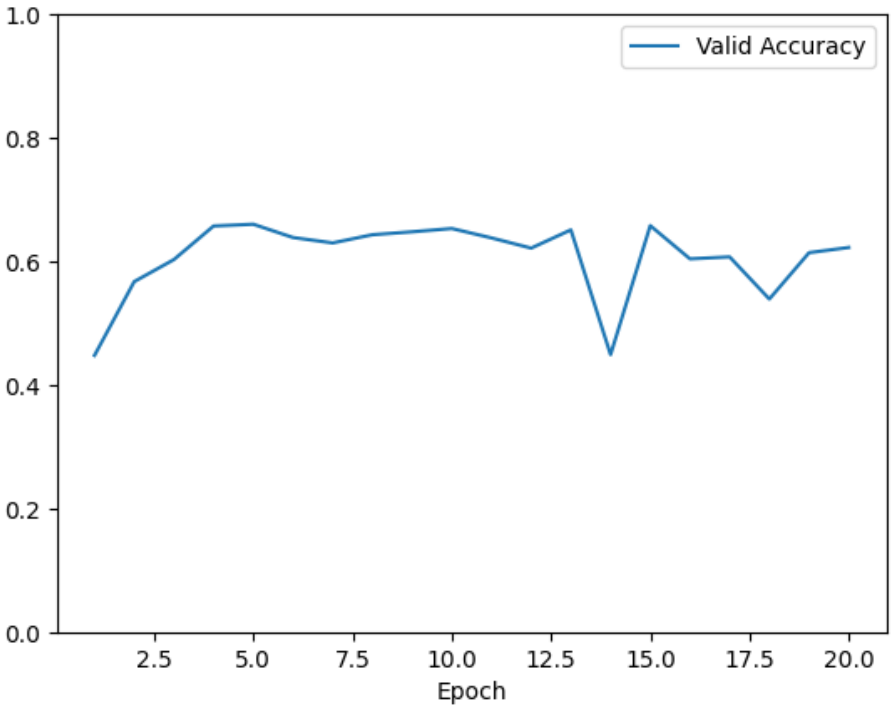

Figure 7: Validation accuracy for $\eta = 0.1$

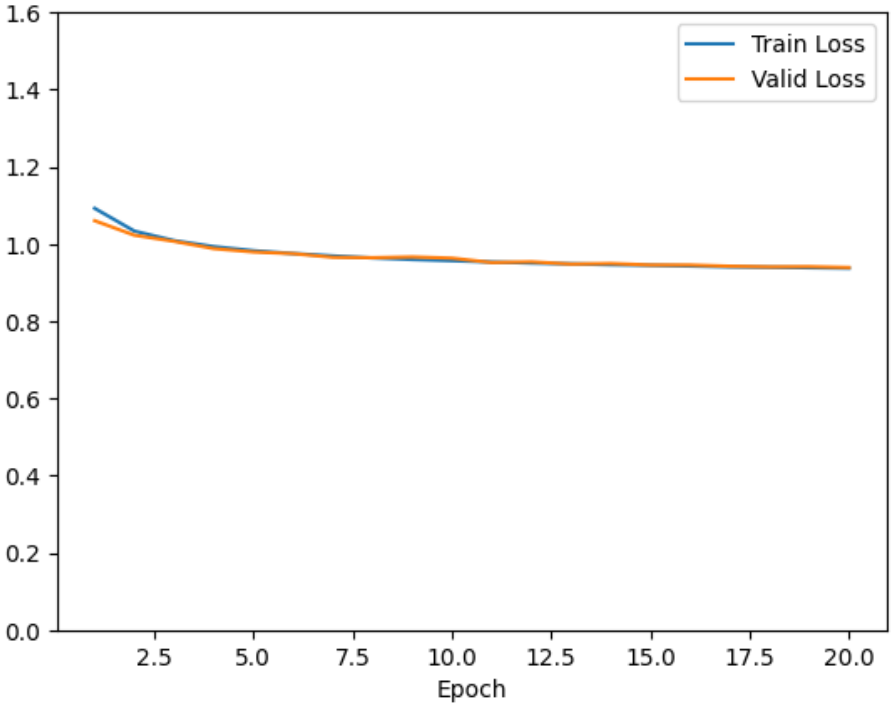

Figure 8: Training and validation loss for $\eta = 0.01$

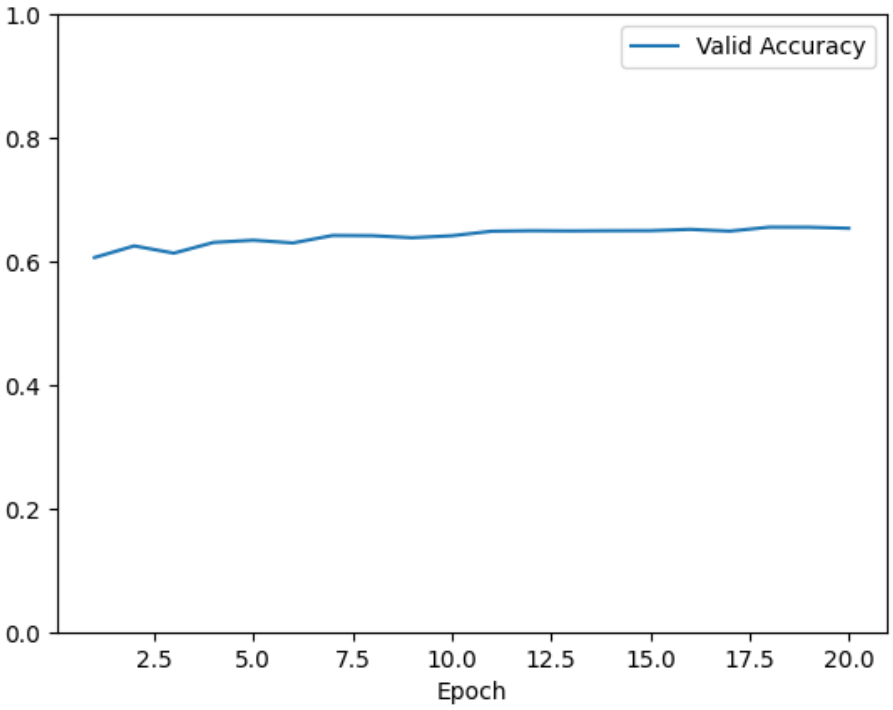

Figure 9: Validation accuracy for $\eta = 0.01$

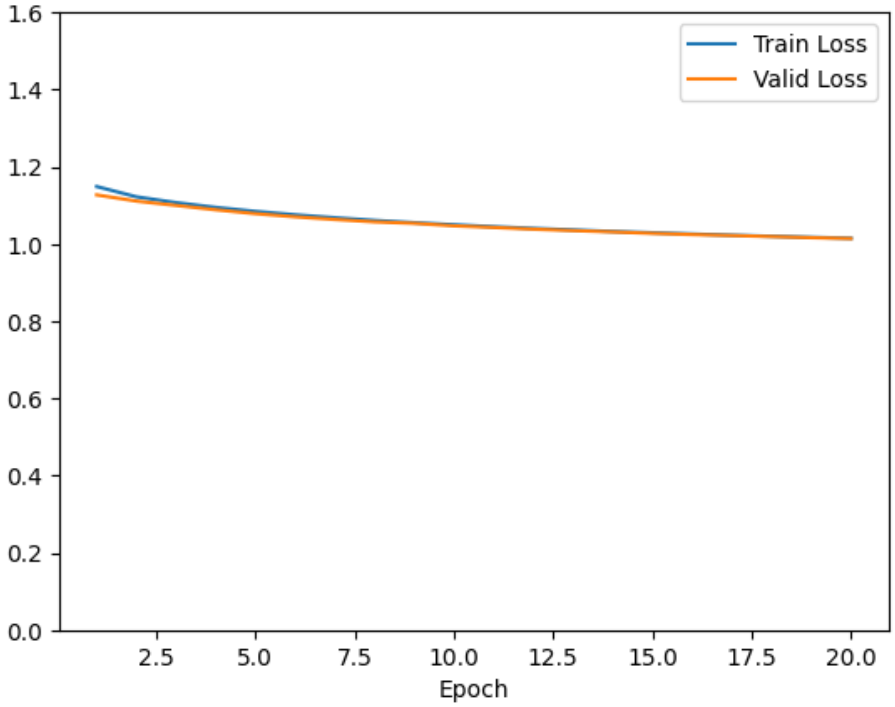

Figure 10: Training and validation loss for $\eta = 0.001$

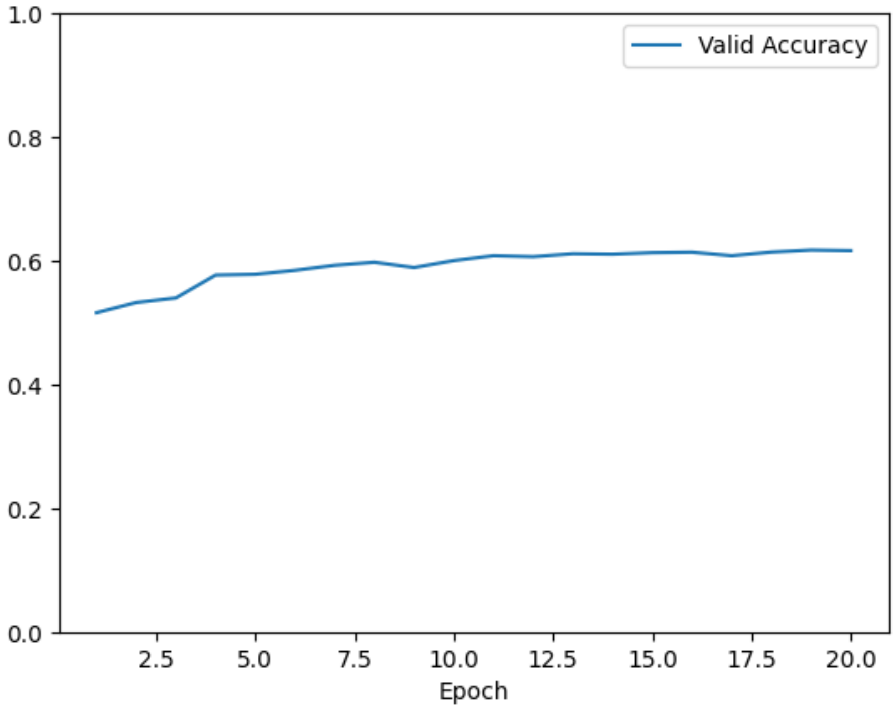

Figure 11: Validation accuracy for $\eta = 0.001$

The test accuracies were 0.5577, 0.6200 and 0.6503 for $\eta = 0.1$, $\eta = 0.01$ and $\eta = 0.001$

respectively.

Looking at the plots, we can see that the best configuration, in terms of final valdiation accuracy, was $\eta = 0.01$.

2.2 Question 2.2

2.2.1 Question 2.2 a)

Answer After running the code, the following plots were generated:

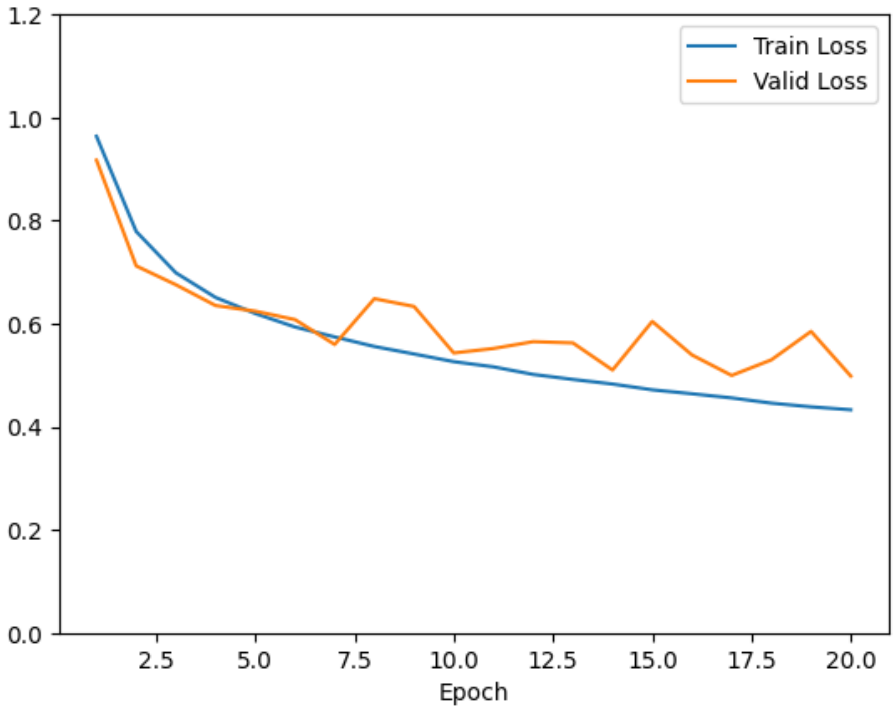

Figure 12: Training and validation loss for batch size of 16

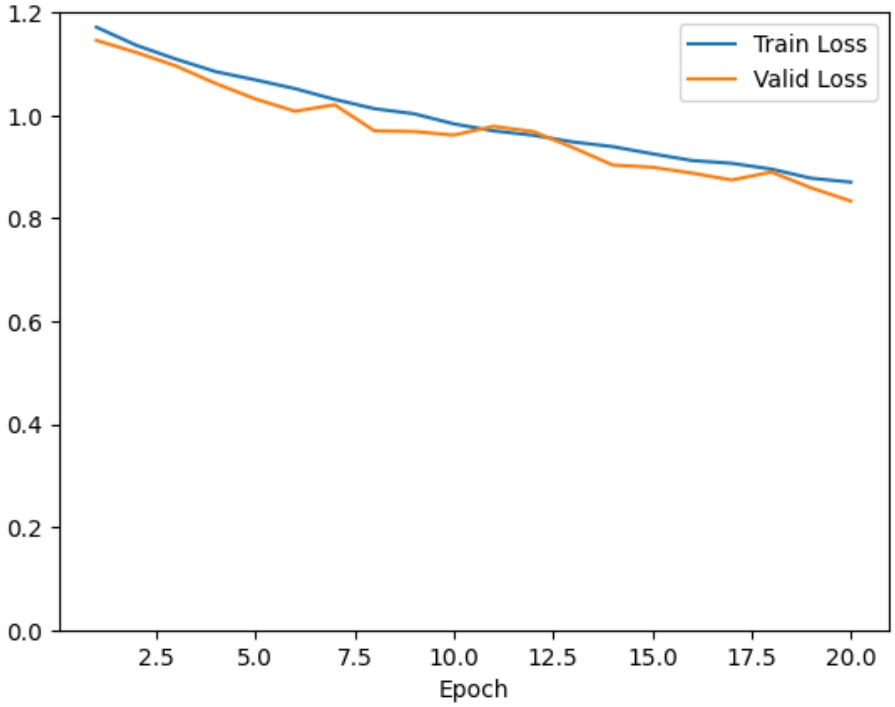

Figure 13: Training and validation loss for batch size of 1024

This exercise explores the trade-off between train time and performance when using different batch sizes. When training with smaller batch sizes, the model updates weights more frequently, since each update is working with less data each time, and thus, takes more time to train when comparing to a bigger batch size. However, this frequent updating for smaller batch sizes can provide a regularizing effect avoiding overfitting. When compared to larger batch sizes, smaller batch sizes are more noisy, since they are more sensitive to the data they are working with. This noise can be positive, since it can help the model to avoid converging to a local minimum and, that way, impoves the chances of finding the global minimum. On the other hand, the noise can also be negative, since it can lead to a slower convergence. On a overall note, smaller batch sizes often lead to a better model generalization and accuracy, despite taking more time to train. This comment can be supported by the plots above, where we can see that the model with batch size 16 has a better performance than the one with batch size 1024.

The best test accuracy was 0.7675 for batch size 16.

2.2.2 Question 2.2 b)

Answer After running the code, the best and worst configurations were for learning rates of 0.1 and 1, respectively:

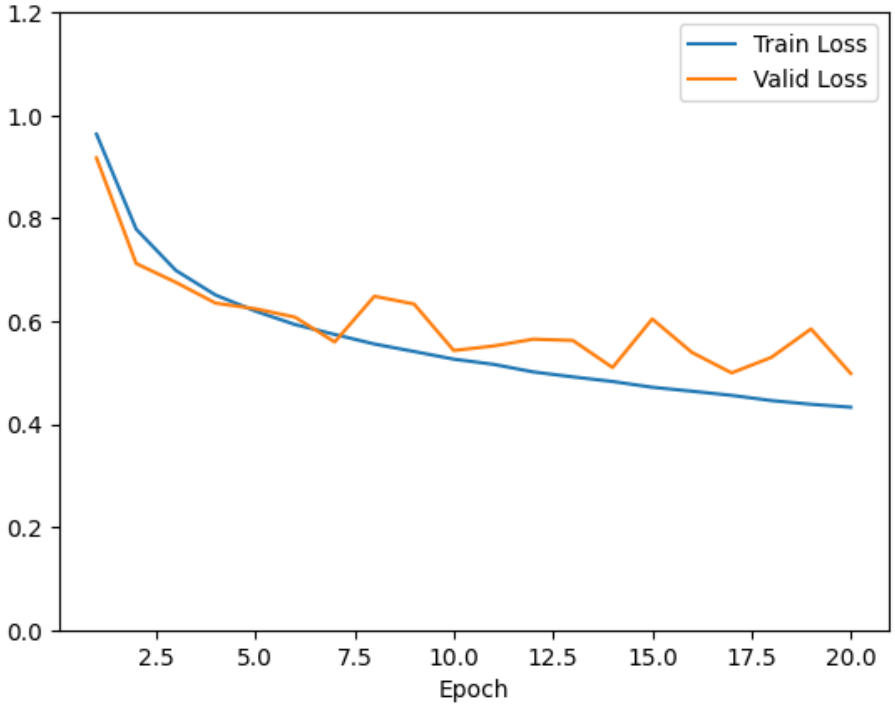

Figure 14: Training and validation loss for learning rate of 0.1

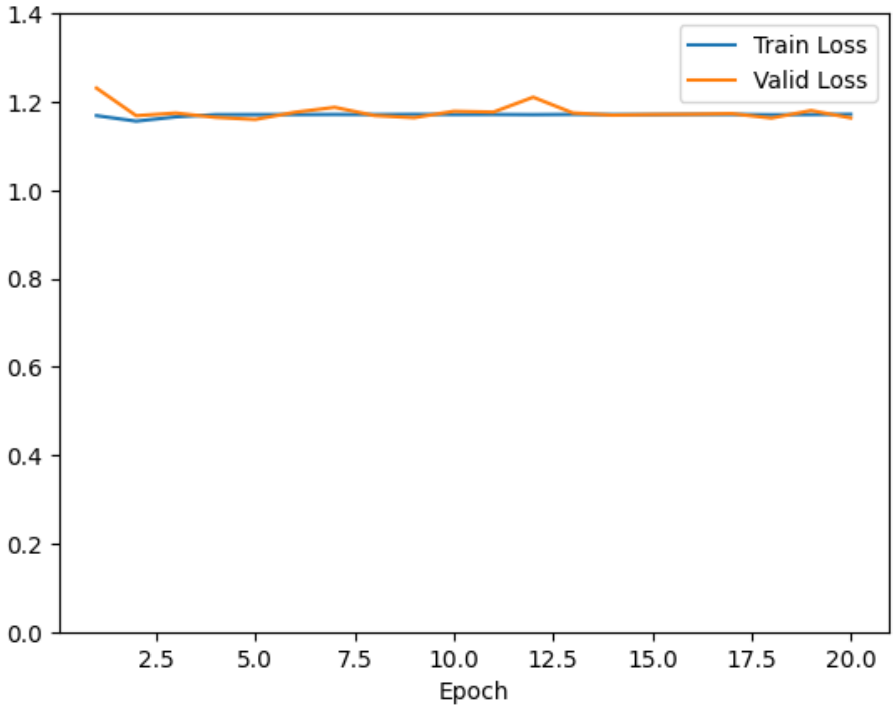

Figure 15: Training and validation loss for learning rate of 1

This exercise explores the trade-off between a high and low learning rate. When using a high

learning rate, the model converges faster, since it is taking bigger steps towards the minimum. However, this can lead to problems, since the model can overshoot the optimal minimum. On the other hand, when using a low learning rate, the model takes smaller steps towards the minimum and thus, it takes more time to converge. This can be a good thing, since it can avoid overshooting the minimum and diverging. Important to note, that for a very low learning rate, the model can get stuck in a local minimum, not achieving the best possible accuracy. For our specific case, the model with learning rate 1 seems to have converged too quickly on a non optimal minimum, since the model does not improve further after the first ephocs, as we can see in the Figure 15. For the learning rates 0.01 and 0.001, since the convergence is slower, the model was not able to reach a good accuracy in 20 epochs. The best test accuracy was 0.7675 for learning rate 0.1, wich offered a good balance between convergence speed and accuracy.

2.2.3 Question 2.2 c)

Answer After running the code, the best configuration was for the model with a dropout probability of 0.2 and the worst one was for the default model.

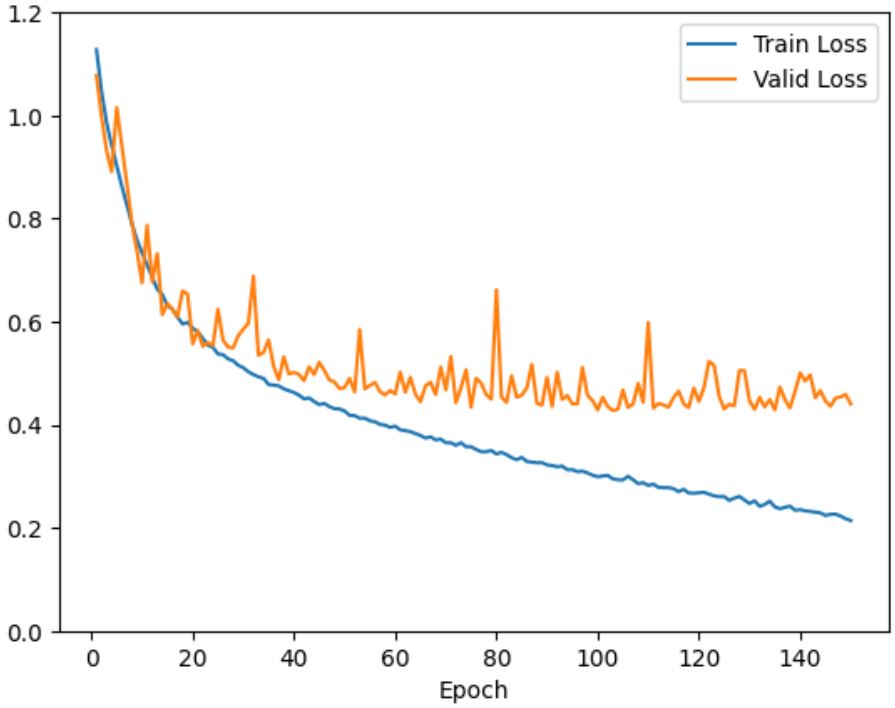

Figure 16: Training and validation loss for default model

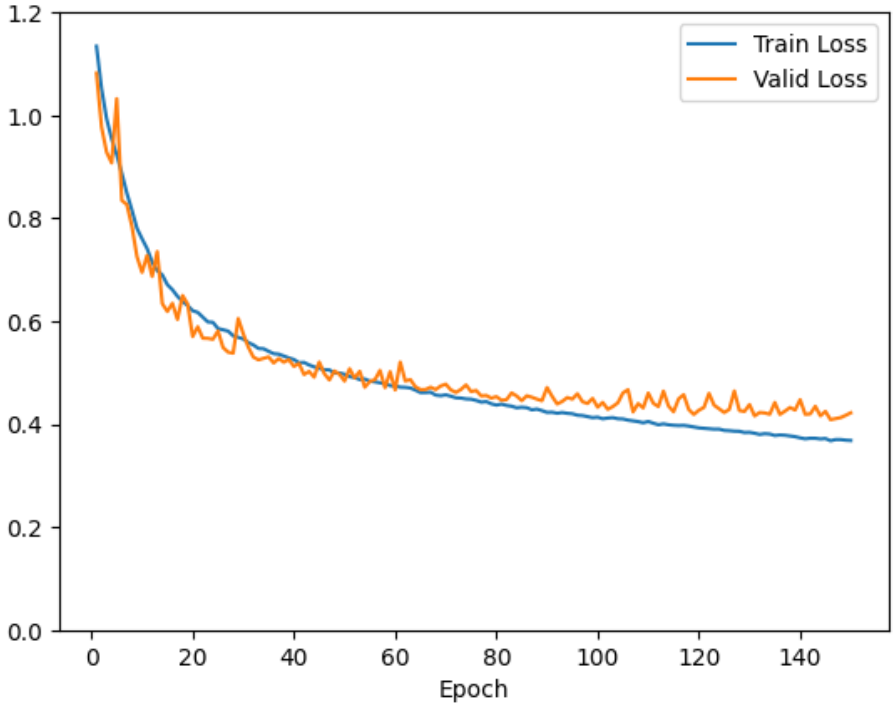

Figure 17: Training and validation loss for model with dropout of 0.2

For the base model proposed in this exercise, although the validation loss starts to slow down its decrease, there are no clear signs of overfitting.

In this exercise, we explored two different techniques to avoid overfitting. We started by testing the model by applying L2 regularization, which adds a penalty term to the loss function, forcing the model to learn smaller weights, encouraging the model to find simpler solutions and, that way, reducing the impact of noisy or irrelevant features. After that we tested applying a dropout, which is a technique that randomly drops a proportion of the network's nodes during each training epoch, forcing the network to avoid relying too heavily on any one feature. The L2 regularization, only managed to a slight improvement in the validation loss's stability. On the other hand, the dropout technique, managed to significantly improve the model's performance, as we can see in the Figure 17. The best test accuracy was 0.7825 for the model with dropout probability of 0.2.

3 Question 3

3.1 Question 3.1

3.1.1 Question 3.1 a)

Answer To demonstrate that the specified Boolean function cannot be computed by a single perceptron, let's consider a simple case where $D = 2$, $A = -1$, and $B = 1$. The function f is defined as:

$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{D} x_i \in [-1, 1], \\ -1 & \text{otherwise} \end{cases}$$

In this setup:

- For $x = (+1, +1)$, the sum $\sum x_i = 2$. Since 2 is not in the range [-1, 1], $f(x) = -1$.

- For $x = (-1, -1)$, the sum $\sum x_i = -2$. Since -2 is also not in the range [-1, 1], $f(x) = -1$.
- For $x = (-1, +1)$ or $x = (+1, -1)$, the sum $\sum x_i = 0$. This falls within the range [-1, 1], so $f(x) = 1$ for these inputs.

The visual representation of the points can be seen in Figure 18. The red points represent the inputs that should be classified as $+1$ and the blue points represent the inputs that should be classified as -1.

The critical point here is that a single perceptron is fundamentally a linear classifier, which means it can only separate data points using a straight line in the feature space. However, in this example, there is no straight line that can separate these points accordingly in a 2D space, to satisfy the function f.

This example thus serves as a counter-example proving that the given function cannot generally be computed with a single perceptron, as it requires a non-linear decision boundary which a single perceptron cannot provide.

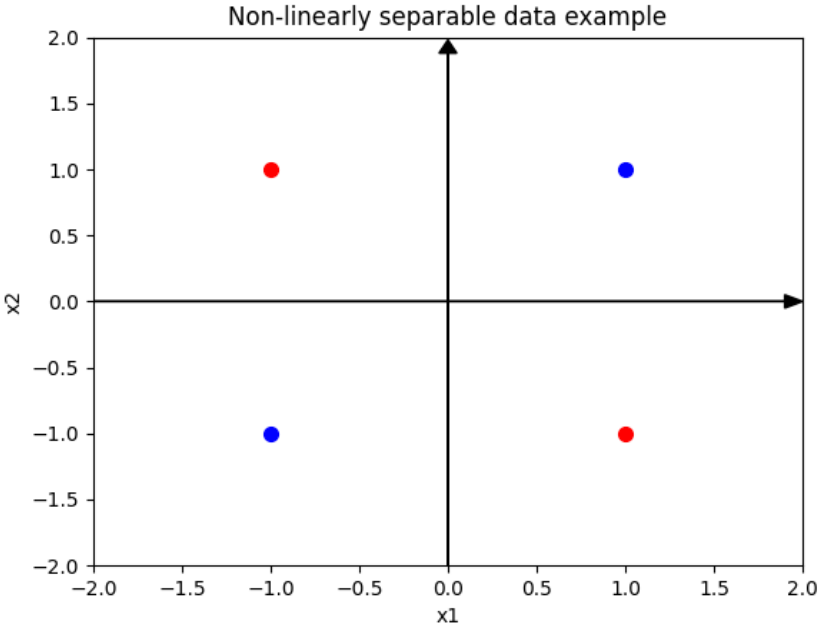

Figure 18: Classification of points using the function f

3.1.2 Question 3.1 b)

Answer Firstly, we will start by defining the weights and biases of the network. We will use the notation $W^{(l)}$ and $b^{(l)}$ to represent the weights and the biases, respectively, of the l-th layer. Consider now:

$W^{(1)} = \begin{bmatrix} 1 & \dots & 1 \\ -1 & \dots & -1 \end{bmatrix}$, where $W^{(1)}$ is a matrix of size 2 x D, and D is the size of the input vector.

$b^{(1)} = \begin{bmatrix} -A \\ B \end{bmatrix}$, where A is the lower bound of the sum of the input vector, and B is the upper bound of the sum of the input vector.

The idea behind the weights and biases of the first layer is that we want to verify if the sum of the input vector is within the range [A, B]. However, we have to do this individually, computing

the lower bound condition for the first hidden unit, and the upper bound condition for the second hidden unit.

That is why we have the weights of the first layer as all 1's for the first row and all -1's for the second row, and the biases as -A and B.

If $Z^{(1)}$ is the pre-activation of the first layer, we have that $Z^{(1)} = W^{(1)}X + b^{(1)}$. The first hidden unit's output will be $g((\sum_{i=1}^{D} x_i) - A)$, and it will be 1 if the sum of the input vector is greater than or equal to A, and -1 otherwise. The second hidden unit's output will be $g(B - (\sum_{i=1}^{D} x_i))$, and it will be 1 if the sum of the input vector is less than or equal to B, and -1 otherwise.

This means the first hidden unit's output is 1 if the sum of the input vector respects the lower bound, and the second hidden unit's output is 1 if the sum of the input vector respects the upper bound.

$W^{(2)} = \begin{bmatrix} 1 & 1 \end{bmatrix}$.

$b^{(2)} = \begin{bmatrix} -1 \end{bmatrix}$.

$W^{(2)}$ and $b^{(2)}$ are used to compute an AND function of the two hidden units. We use this since the output of the hidden units is either -1 or 1, and computing the AND we can verify if they respect both the lower bound and upper bound, and thus belong to the range $[A, B]$.

With this, if $h(x)$ is the resulting function of the network, we have that $h(x) = 1$ if the sum of the input vector is within the range $[A, B]$, and -1 otherwise, and, thus, $h(x) = f(x)$, and we prove this neural network computes $f(x)$.

Now we need to make sure our network is robust to infinitesimal perturbation of the inputs. As it is right now, the network is not robust to infinitesimal perturbation of the inputs. If we have an input vector $X = \begin{bmatrix} 1 \end{bmatrix}$, and we perturb it to make it $X' = \begin{bmatrix} 1 + \epsilon \end{bmatrix}$, with $\epsilon \in \mathbb{R}$ and $\epsilon > 0$. Since ϵ is a really small number, we have that $h(X) = 1$ and $h(X') = -1$, and thus the current network is not robust to infinitesimal perturbation of the inputs.

To prevent this from happening, we need to change the interval of values we want to accept from $[A, B]$ to $[A - \epsilon, B + \epsilon]$. Since the input vector contains only integers, we can say that the sum of the input vector is also an integer, and thus we need an ϵ that is smaller than 1.

This means we have to create new inequations for the lower bound and upper bound conditions, and thus we have to change the weights and biases of the first layer.

The first hidden unit needs to represent the following inequation:

$$(\sum_{i=1}^{D} x_i) \geq A - \epsilon \iff (\sum_{i=1}^{D} x_i) - A + \epsilon \geq 0$$

The second hidden unit needs to represent the following inequation:

$$(\sum_{i=1}^{D} x_i) \leq B + \epsilon \iff B - (\sum_{i=1}^{D} x_i) - \epsilon \geq 0$$

Since $\epsilon \in \mathbb{R}$ and $\epsilon > 0$, we can say that $\epsilon = k1/k2$, where $k1, k2 \in \mathbb{Z}$, and $k1 > 0, k2 > 0$ and $k1 < k2 (\epsilon < 1)$. This way we can rewrite the inequations as:

$$(\sum_{i=1}^{D} x_i) - A + k1/k2 \geq 0 \iff k2(\sum_{i=1}^{D} x_i) - k2A + k1 \geq 0$$

$$B - (\sum_{i=1}^{D} x_i) - k1/k2 \geq 0 \iff k2B - k2(\sum_{i=1}^{D} x_i) - k1 \geq 0$$

This way, we can rewrite the weights and biases of the first layer as:

$$W^{(1)} = \begin{bmatrix} k2 & ... & k2 \\ -k2 & ... & -k2 \end{bmatrix}$$

$$b^{(1)} = \begin{bmatrix} -k2A + k1 \\ k2B - k1 \end{bmatrix}$$

With this, we have that the first hidden unit's output will be $g(k2(\sum_{i=1}^{D} x_i) - k2A + k1)$, and it will be 1 if the sum of the input vector is greater than or equal to A - ϵ, and -1 otherwise. The second hidden unit's output will be $g(k2B - k2(\sum_{i=1}^{D} x_i) - k1)$, and it will be 1 if the sum of the input vector is less than or equal to B + ϵ, and -1 otherwise.

A possible set of values for $k1$ and $k2$ is $k1 = 1$ and $k2 = 100$. With this, $\epsilon = 0.01$, and we accept values in the range $[A - \epsilon, B + \epsilon] = [A - 0.01, B + 0.01] = [A - 0.01, B + 0.01]$.

This means the first hidden unit's output is 1 if the sum of the input vector respects the lower bound, and the second hidden unit's output is 1 if the sum of the input vector respects the upper bound.

With this, we have that $h(x) = f(x)$, and the network is robust to infinitesimal perturbation of the inputs, since it now accepts inputs that are within the range $[A - \epsilon, B + \epsilon]$.

3.1.3 Question 3.1 c)