Deep Learing

Homework 1

Stefanos Zafiris ist1109041
Črtomir Perharič ist1108973

November-December 2023

1 Question 1

1.1 Perceptron

In question one our goal is to classify images of 4 different classes of retinal diseases from the Octmnist dataset. We start by using a simple perceptron. The images were cropped to 28×28 pixels, therefore the input layer will have $784 + 1$ units, where the extra 1 is added for the bias. The accuracy evolution with respect to the epoch number is shown in figure 1. As we can see the accuracy both for the training and validation sets oscillates around 50% by $\approx \pm 15\%$. The final training and validation accuracies were 0.4654 and 0.4610 respectively. The final obtained accuracy of the test set was 0.3422. Since this is a linear classifier and the data is very unlikely linearly separable the low accuracy is not so shocking.

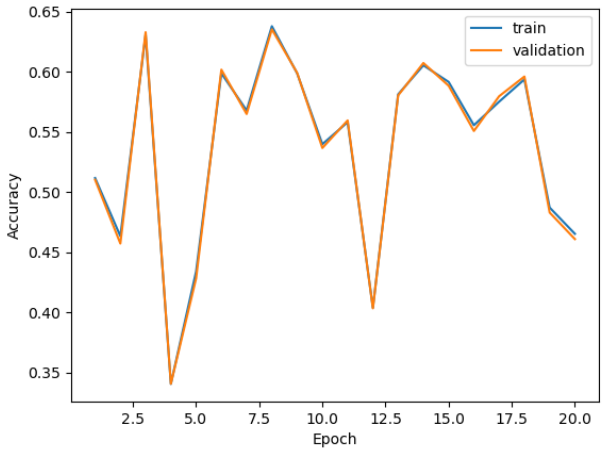

Figure 1: Accuracy with respect to the epoch number for the simple perceptron classifier. The training set is plotted in blue, while the validation set is portrayed in orange.

1.2 Logistic regression

Next we try to classify the same dataset using logistic regression. The accuracies are shown in figures 2 and 3 both for a learning rate $\eta = 0.01$ and $\eta = 0.001$ respectively.

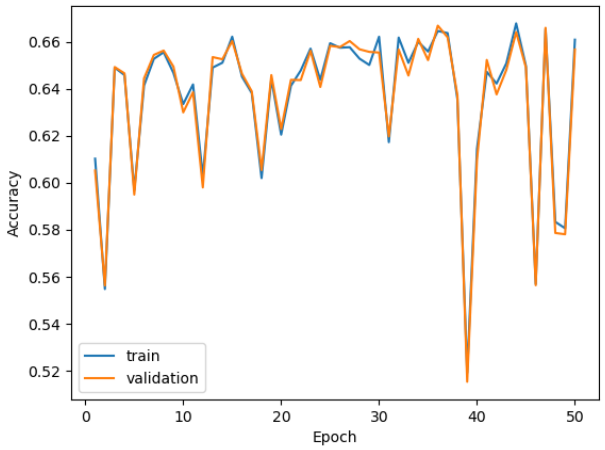

Figure 2: Accuracy with respect to the epoch number for logistic regression with learning rate $\eta = 0.01$. The training set is plotted in blue, while the validation set is portrayed in orange.

From figure 2 we can first see a slight improvement in the accuracy, however, for later iterations oscillations arise yet again. The overall accuracies are definitely higher than for the simple perceptron classifier. The highest value for both the training and validation sets is just over 66%, while the final accuracy of the test set is

$$\mathrm{ACC}_{\mathrm{test}} = 0.5784. \tag{1}$$

This is already much better than the accuracy obtained with the perceptron. Figure 3 shows the accuracy evolution when a learning rate $\eta = 0.001$ is used. In this case the improvement in accuracy with higher epoch numbers is much more obvious. We can clearly see a sharp rise in accuracy at the beginning and a slow plateau, as the epoch number further increases. In average the accuracies are slightly better than the ones shown in figure 2, however, the highest achieved accuracy is not visibly better. The final test set accuracy, however, shows a slight improvement with the value of

$$\mathrm{ACC}_{\mathrm{test}} = 0.5936. \tag{2}$$

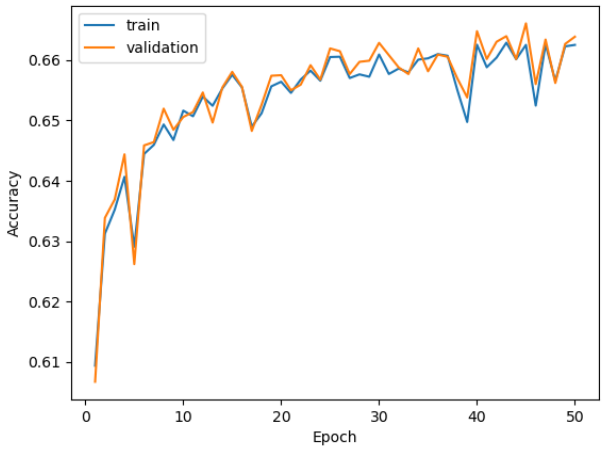

Figure 3: Accuracy with respect to the epoch number for logistic regression with learning rate $\eta = 0.001$. The training set is plotted in blue, while the validation set is portrayed in orange.

1.3 Multi-layer perceptron

1.3.1 Neural Network vs Logistic Regression

The statement given in the homework is in general true. Logistic regression is indeed a convex optimization problem, thus we can find the global minimum easier, since any local minimum will automatically be a global minimum as well. For neural networks, however, this is not true and the optimization gets stuck in local minima.
It is, however, also true, that logistic regression is not as expressive as a multi-layer perceptron (MLP), since the latter is not a linear classifier, i.e. it can approximate non linear functions and extract features. In contrast to linear classifiers, such as logistic regression, features have to be extracted manually.

1.3.2 Results

Lets now look at the results of solving the same problem as before, this time, however, using a one hidden layer MLP. The hidden layer has 200 units with ReLU activation functions. We use the cross-entropy loss function together with a learning rate of $\eta = 0.001$.
The results are shown in figures 4 and 5. From figure 4 we see, that the accuracy improves with epoch iterations as expected. The validation set accuracy is for the most part slightly lower than its training counterpart. Furthermore the accuracy starts to improve slower, as the epoch number increases, which is to be expected as well. The loss-epoch dependence is shown in figure 5. The loss

curve has the typical shape, where it rapidly decreases in the beginning, before it starts to plateau for later iterations. Since the first drop is very big, we also show the log scaled loss function in figure 6, where the plateau is more obvious. Finally we state the final test set accuracy, which was

$$\mathrm{ACC}_{\mathrm{test}} = 0.7561. \tag{3}$$

This is a huge improvement compared to the ones obtained with linear classifiers before. All of the final accuracies are compiled in table 1.

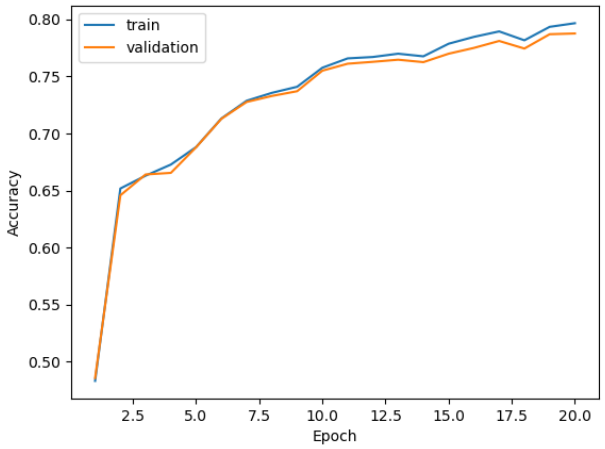

Figure 4: Accuracy evolution with respect to epoch number for MLP classification of retinal diseases. The blue line represents the training set, while the orange represents the validation set.

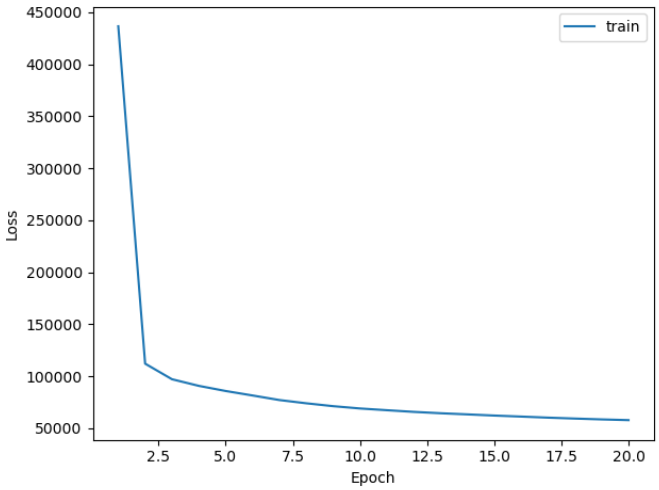

Figure 5: Cross-entropy loss evolution with respect to epoch number for MLP classification of retinal diseases.

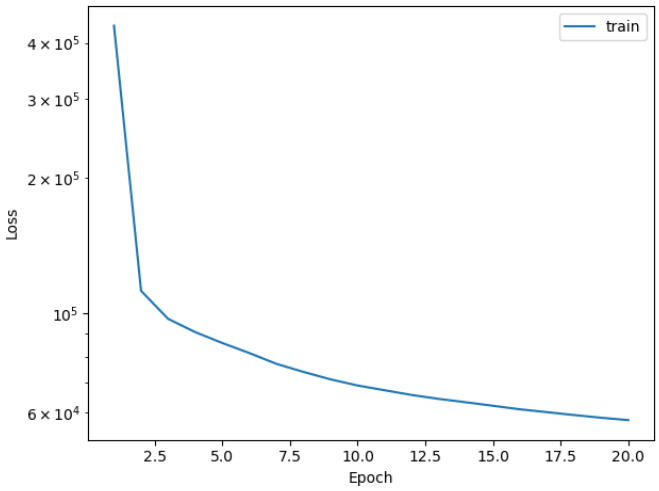

Figure 6: Figure 5 in logarithmic scale.

Classifier	**Accuracy**
perceptron	0.3422
logistic regression with $\eta = 0.01$	0.5784
logistic regression with $\eta = 0.001$	0.5936
MLP	0.7561

Table 1: Accuracies of the test set for all classifiers in question 1.

2 Question 2

2.1 Logistic regression with pytorch

In this section the goal is to implement a linear model with logistic regression model in pytorch to a given skeleton code. The logistic regression will use stochastic gradient descent as training algorithm with batch size of 16. The epochs are set to 20. To compare performance the learning rate on the validation data is tuned to have the following values $\{0.001, 0.01, 0.1\}$.

Learning rate	Final validation accuracy
0.1	0.6224
0.01	**0.6535**
0.001	0.6163

Table 2: Final validation accuracy on different configurations

Based on table 2, after executing the three configurations, we can see that the one with the learning rate set to 0.01 performed the best when it comes to final validation accuracy. A learning rate of 0.01 appears to be the sweet spot between all three configurations when it comes to final validation accuracy, since it moves quicker than 0.01 to the optimal value, but not so quickly that it misses the optimal value, as 0.1 does. Interpolation could yield a better performance, but the difference would be small. The training-validation loss (left) and validation accuracy (right) plots for configuration with learning rate set to 0.01 is showcased in figure 7. The final accuracy on the test set is

$$\text{ACC}_{\text{test}} = 0.6200. \tag{4}$$

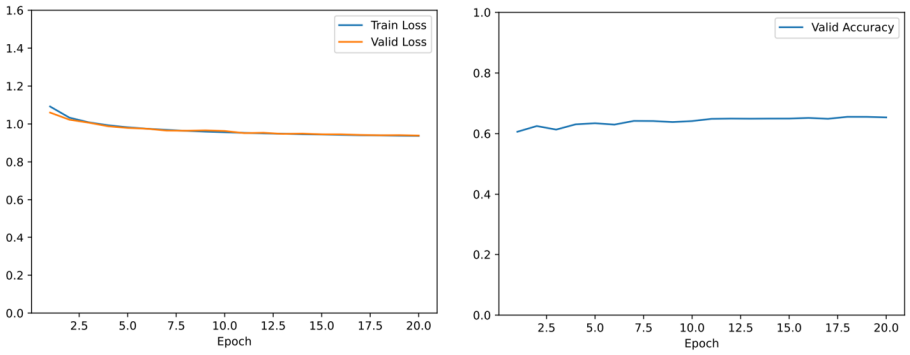

Figure 7: Plots of configuration in which learning rate is set to 0.01, where left is training-validation loss and right is validation accuracy

2.2 Feed-forward neural network with pytorch

In this section the task is to implement a feed-forward neural network using dropout regularization. The implementation is again done in pytorch to the same skeleton code provided by the course staff. The hyperparameters used in this exercise are by default the ones showed in Table 3, unless stated otherwise.

Number of epochs	20
Learning rate	0.1
Hidden size	200
Number of layers	2
Dropout	0.0
Batch size	16
Activation	ReLU
L2 Regularization	0.0
Optimizer	SGD

Table 3: Default hyperparameters

2.2.a Batch size tuning

The goal of this task is to compare the performance of a model with batch sizes of 16 and 1024, while the remaining hyperparameters remain default. The training-validation loss of both configurations is showcased in Figure 8, where the left plot is the configuration with batch size set to 16 and right plot is the configuration with batch size set to 1064.

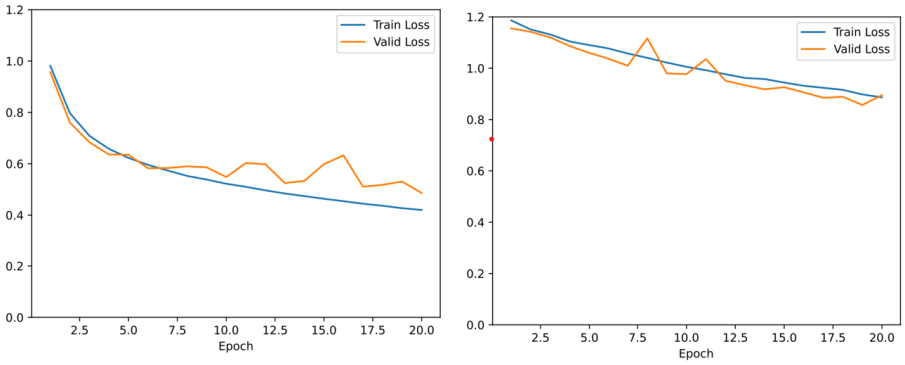

Figure 8: Plots of training-validation loss for configuration with batch size of 16 (left) and batch size of 1064 (right)

According to Figure 8, the configuration with a batch size of 16 performs much better in terms of loss. This is to be expected because with batch size 16, the weights and biases are updated much more frequently (64x) than with batch size 1024, resulting in better capturing of underlying features, especially with

such a low number of epochs. This also means smaller batch sizes are more likely to over fit with larger number of epochs. Because of the before mentioned reasons, the configuration with batch size of 16 performed better when it comes to final test accuracy with

$$\text{ACC}_{\text{test}} = 0.7713. \tag{5}$$

Batch size	Time taken
16	657.34
1024	**78.53**

Table 4: Execution times of both configurations

When analyzing table 4, we can see that the execution time is highly dependent on batch size. This is because as mentioned before, the smaller the batch size the more often the model updates weights and biases per epoch. This means that difference would be even bigger if the model was more complex or the epochs were larger. This clearly demonstrates the trade-off that must be made between model accuracy and execution times, emphasizing the need of determining the ideal batch size.

2.2.b Learning rate tuning

The objective of this task is to compare the performance of a model with different learning rates: 1, 0.1, 0.01, 0.001, with all other hyperparameters set to default. Table 5 shows how different configurations performed based on accuracy. It's easy to observe from the table, that when the learning rate was the highest (1), the model did the worst, and when it was 0.1, it performed the best. The plots of training-validation loss of the best (lr = 0.1) and worst (lr = 1.0) models, based on accuracy are showcased in Figure 9. The configuration where learning rate was set to 0.1 also performed the best in the final test accuracy by achieving a value of

$$\text{ACC}_{\text{test}} = 0.7713. \tag{6}$$

Learning rate	Best valid. acc.	Last valid. acc.
1	0.4721	0.4721
0.1	**0.8291**	**0.8291**
0.01	0.8147	0.8099
0.001	0.7224	0.7224

Table 5: Tabled values of configurations with different learning rates

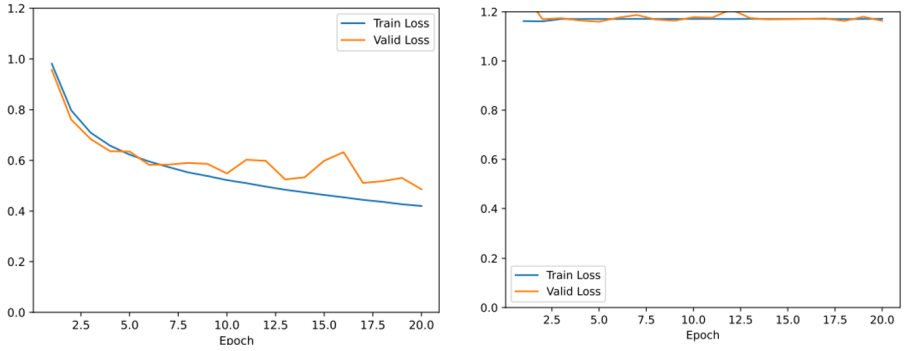

Figure 9: Plots of training-validation loss for a configuration where learning rate is set to 0.1 (left) and the learning rate is set to 1.0 (right)

Based on Table 5 the configurations with learning rate of 0.1 and 0.01 performed really well, indicating that the optimum learning rate is somewhere close to them. With a learning rate of 1.0 the model did a really bad job, indicating that the rate was too high and the model may have oscillated between different minima, never converging to a stable solution. These inferences are also prevalent in Figure 9. The model fared better with a learning rate of 0.001 than with a learning rate of 1.0, but it was still far from the two best. This is due to the fact that the learning rate in this situation is too small for the amount of epochs. With such a low learning rate, the model will take much more updates to converge to a stable solution since the updates are so little that the optimizer cannot make substantial progress towards the optimum solution. It is also more likely to become stuck on a suboptimal local minima. This clearly highlights the importance of selecting a proper learning rate through hyperparameter tuning.

2.2.c Overfitting and regularization

This task composed of two different objectives. The first objective was to use a batch size of 256 with number of epochs set to 150, while the remaining parameters were set to default. This configuration did not perform too well as the model started to over fit at around 50 epochs, as can be seen in Figure 10. In this case early stopping or regularization should be used.

The second objective was to still have a batch size of 256 and number of epochs 150, but this time try two different regularization methods. First regularization method was L2 regularization also known as weight decay, by setting L2 regularization parameter to 0.0001, while all the other hyperparameters are set default. The second regularization method was setting a dropout chance to 0.2, while all the other hyperparameters are set to default. The Best model out of all these configurations in terms of last validation accuracy was dropout regularization and the worst one was the initial model as can be seen in Table 6. Based on these values the training-validation losses were plotted for the best (right)

and worst (left) configurations in Figure 10. In terms of final test accuracy the dropout regularization also performed the best with a value of

$$\text{ACC}_{\text{test}} = 0.7807. \tag{7}$$

Model	Best valid. acc.	Last valid. acc
Initial	**0.8667**	0.8436
L2 Regularization	0.8651	0.8581
Dropout Regularization	0.8649	**0.8629**

Table 6: Tabled values of configurations with different regularization's

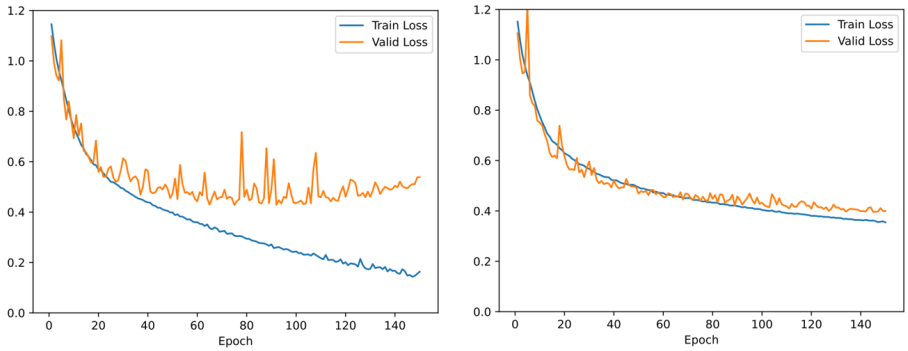

Figure 10: Plots of training-validation loss for a configuration without regularization (left) and where there is dropout regularization (right)

If early stopping had been utilized in the initial model, it would have performed best in terms of validation accuracy, as shown in Table 6. Unfortunately, as previously stated, the model began to overfit, which explains why the last validation accuracy was higher in the models with regularization. Both regularization strategies successfully suspended this overfitting, however dropout regularization performed slightly better in situation. Dropout regularization performed exceptionally well in terms of validation loss, as seen in Figure 10.

In L2 Regularization large weights are penalized by adding a term to the loss function that is proportionate to the total of the squared weights. This pushes the model to find a solution with fewer weights, which can assist to minimize the model's complexity and reduce the likelihood of overfitting. Dropout regularization on the other hand drops neurons from the network at random during training. This means that for each training example, a portion of the network's neurons are temporarily eliminated. This forces the remaining neurons to learn to represent the input data more robustly. This clearly shows why regularization is really important to use, even if has a minor negative effect on best accuracy.

3 Question 3

Lets define a Boolean function of D variables, $f : \{-1, +1\} \rightarrow \{-1, +1\}$:

$$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{D} x_1 \in [A, B] \\ -1 & \text{otherwise} \end{cases}, \tag{8}$$

where A and B are integers, such that $-D \leq A \leq B \leq D$.

3.1 Single perceptron

Suppose we take $D = 2$, meaning $x_1, x_2 \in \{-1, 1\}$. Therefore, A and B are subject to the following constraint:

$$-2 \leq A \leq B \leq 2 \tag{9}$$

Let's now say that $A = 0$ and $B = 1$. If we insert all the possible values of x_1 and x_2 into the function (8), we get the following table 7, where f(x) is the label:

x_1	x_2	$f(x)$
$+1$	$+1$	-1
$+1$	-1	$+1$
-1	$+1$	$+1$
-1	-1	-1

Table 7: Values achieved by inserting x_1 and x_2 into equation (8)

By plotting the values of table 7, we get the following Figure 11:

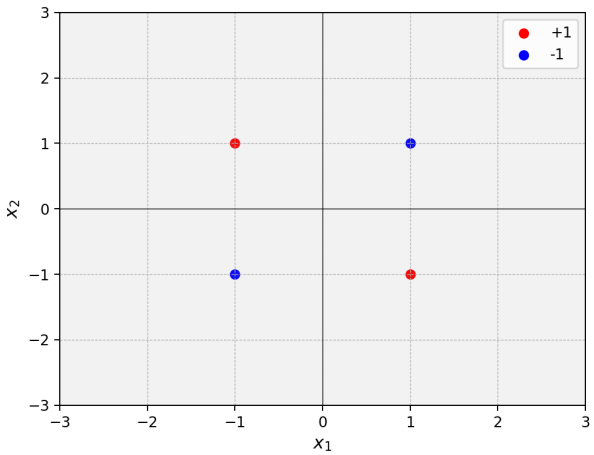

Figure 11: Plot of the values achieved by inserting x_1 and x_2 into equation (8)

Only linearly separable functions can be represented by a single perceptron. We can see from Figure 11 that there is no way to separate the two labels linearly. This means that in this configuration, function (8) cannot be computed with a single percepton, so it cannot be computed with a single percepton in general.

3.2 Multi-layer perceptron with hard threshold activation functions

We now add a 2-unit hidden layer with hard threshold activations

$$g(z) = \text{sign}(z) = \begin{cases} +1 & \text{if } z \geq 0 \\ -1 & \text{otherwise} \end{cases} . \tag{10}$$

Let's first introduce some notation, so we can follow along easier. We denote the preactivation sum as z_i, where $i \in \{1, 2, 3\}$. Furthermore, the output of each unit is denoted by $h_i = g(z_i)$, the weights by $W_i^{(j)}$ and the biases by $b_i^{(j)}$, where i and j correspond to the unit and layer respectively.
Now after some trial and error we find, that one possible network configuration, which successfully computes the function f is

$$\begin{aligned}
\boldsymbol{W}_1^{(1)} &= \underbrace{\begin{bmatrix} 2 & 2 & \dots & 2 \end{bmatrix}}_{D}^{\mathrm{T}}, \\
\boldsymbol{W}_2^{(1)} &= \underbrace{\begin{bmatrix} -2 & -2 & \dots & -2 \end{bmatrix}}_{D}^{\mathrm{T}}, \\
b_1^{(1)} &= -2A + 1, \\
b_2^{(1)} &= 2B + 1, \\
\boldsymbol{W}_1^{(2)} &= \begin{bmatrix} 1 & 1 \end{bmatrix}^{\mathrm{T}}, \\
b_1^{(2)} &= -1.
\end{aligned}$$

Let's prove this. For the given weights we can evaluate the preactivation sum as

$$\boldsymbol{z} = W^{(1)}\boldsymbol{x} + \boldsymbol{b}^{(1)} = \begin{bmatrix} \boldsymbol{W}_1^{(1)} & \boldsymbol{W}_2^{(1)} \end{bmatrix}^{\mathrm{T}} \boldsymbol{x} + \begin{bmatrix} -2A + 1 \\ 2B + 1 \end{bmatrix} = \begin{bmatrix} 2(\sum_i^D x_i - A) + 1 \\ 2(B - \sum_i^D x_i) + 1 \end{bmatrix}.$$

We should now think about what the outputs of $\boldsymbol{h}$ will be, with regard to different inputs. Therefore we separately analyze three different intervals:

1. $\sum_i^D x_i < A$ will result in $h_1 = g(z_1) = g(-|z_1|) = -1$ and $h_2 = 1$, since $B - \sum_i^D x_i \geq A - \sum_i^D x_i > 0$ for $B \geq A$.

2. $A \leq \sum_i^D x_i \leq B$ will result in $h_1 = 1$ and $h_2 = 1$.

3. $\sum_i^D x_i > B$ will result in $h_1 = 1$ (since $B \geq A$) and $h_2 = g(-|z_2|) = -1$.

The output of the second layer will be

$$z_3 = \begin{bmatrix} h_1 & h_2 \end{bmatrix} \boldsymbol{W_1^{(2)}} - 1.$$

The compiled outputs for the three different regimes presented above, together with the desired outputs given by f can be seen in table 8. As we can see the MLP indeed successfully computes the function f.

interval	h_1	h_2	$b_1^{(2)}$	$\hat{y} = z_3$	$f(\boldsymbol{x})$
$A \leq \sum_i x_i \leq B$	1	1	-1	1	1
$\sum_i x_i < A$	-1	1	-1	-1	-1
$\sum_i x_i > B$	1	-1	-1	-1	-1

Table 8: Predicted outputs of the 1 hidden layer MLP with hard threshold activations, together with desired outputs. The table should be read as: sum the columns 2-4 of each row, to get z_3 and compare it to the desired function value in the last column.

Lastly we check, that this MLP is robust to small perturbations. Lets denote the whole function describing the network as $h(\boldsymbol{x})$, where

$$h(\boldsymbol{x}) = g\left(W^{(1)}\boldsymbol{x} + \begin{bmatrix} -2A+1 \\ 2B+1 \end{bmatrix}\right)^{\mathrm{T}} \begin{bmatrix} 1 \\ 1 \end{bmatrix} - 1 \tag{11}$$

Now we perform a small perturbation $h(\boldsymbol{x}+t\boldsymbol{v})$, where $\boldsymbol{v} \in \mathbb{R}^D$. For the weights given above, we can express the activation of the hidden layer by

$$g(z) = g\left(\begin{bmatrix} 2(\sum_i x_i + t\sum_i v_i - A) + 1 \\ 2(-\sum_i x_i - t\sum_i v_i + B) + 1 \end{bmatrix}\right)$$

Since all of the non perturbative parts in the argument of g are integers, a small perturbation $\ll 1$ will not change the activation function output. We can show this properly by performing a Taylor expansion of g

$$g(z + \epsilon) = g(z); \epsilon \ll z,$$

which gives an exact solution in first order, since all the derivatives are equal to 0. This of course is not true at $g(0)$, as the derivative at that point diverges, however, given the choice of our parameters, we have not made it possible for the argument to ever be 0 as $x_i, A, B \in \mathbb{Z}$.

3.3 Multi-layer perceptron with ReLU activation functions

In this section we will switch the hidden layer activation functions with ReLU

$$g(z) = \mathrm{ReLU}(z) = \max\{0, z\}. \tag{12}$$

We will only mention the weights and biases that we change, as the others will stay the same as in the previous subsection. These are

$$\begin{aligned} \boldsymbol{W}_1^{(2)} &= \begin{bmatrix} -1 & -1 \end{bmatrix}^{\mathrm{T}}, \\ b_1^{(2)} &= 2B - 2A + 2. \end{aligned}$$

Furthermore, we now add a hard threshold activation function for the output layer $\hat{y} = h_3 = g(z_3)$, which we did not, but could have used in the previous problem as well. Once more we analyze the three different intervals

1. $\sum_i x_i < A$ will result in $h_1 = 0$ and $h_2 = 2(B - \sum_i x_i) + 1$.

2. $A \leq \sum_i x_i \leq B$ will result in $h_1 = 2(\sum_i x_i - A) + 1$ and $h_2 = 2(B - \sum_i x_i) + 1$.

3. $\sum_i x_i > B$ will result in $h_1 = 2(\sum_i x_i - A) + 1$ and $h_2 = 0$.

Lets again compile the network outputs, together with the desired results given by f (table 9). As we can see the MLP yet again successfully computes the function f.

interval	h_1	h_2	z_3	h_3	$f(\boldsymbol{x})$
$A \leq \sum_i x_i \leq B$	$2(\sum_i x_i - A) + 1$	$2(B - \sum_i x_i) + 1$	0	1	1
$\sum_i x_i < A$	0	$2(B - \sum_i x_i) + 1$	$1 + 2(A - \sum_i x_i)$	-1	-1
$\sum_i x_i > B$	$2(\sum_i x_i - A) + 1$	0	$1 + 2(B - \sum_i x_i)$	-1	-1

Table 9: Predicted outputs of the 1 hidden layer MLP with ReLU activations, together with desired outputs.

Finally we should again check that the MLP is invariant under small perturbations. Since the first part of the network remains unchanged, we can copy the expression for the hidden layer activation

$$g(z) = g\left(\begin{bmatrix} 2(\sum_i x_i + t\sum_i v_i - A) + 1 \\ 2(-\sum_i x_i - t\sum_i v_i + B) + 1 \end{bmatrix}\right),$$

where g is now the ReLU activation function. We can again perform a Taylor expansion

$$g(z + \epsilon) = \begin{cases} g(z); & \text{if } z < 0 \\ g(z) + \epsilon; & \text{if } z > 0 \end{cases} = \begin{cases} 0; & \text{if } z < 0 \\ z + \epsilon; & \text{if } z > 0 \end{cases}.$$

Not that we have once again made sure, that the argument of the activation function without any perturbations cannot be zero. Since z is still an integer,

we can disregard the ϵ term in the $\lim_{\epsilon \to 0}$. The only time where we have to be careful, is if the sum is inside the closed interval $[A, B]$. However, since in that case two perturbation terms with opposite signs arise, they cancel out. Thus we are left with the original network transfer function $h(\boldsymbol{x}) = f(\boldsymbol{x})$.