Deep Learning – Homework 1

Group 40
99238 – Inês Ji | 99314 – Raquel Cardoso

Contribution: both members of the group were involved in the resolution of all questions in the homework.

Question 1

1.a)

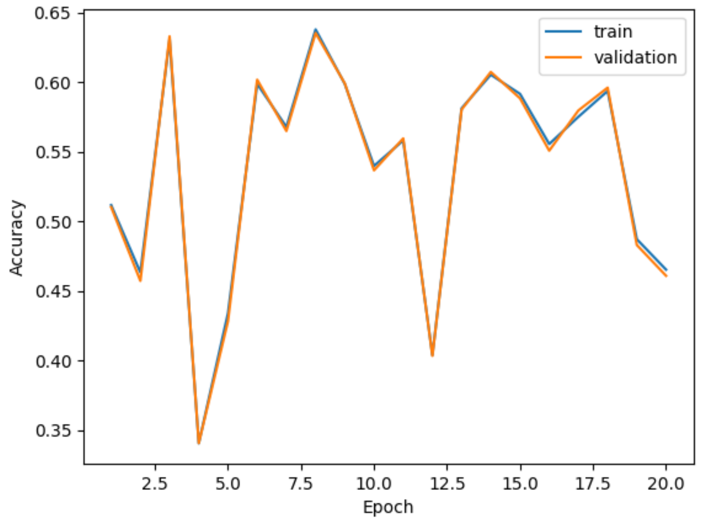


Figure 1 – Percepton: train and validation accuracies as a function of the epoch number

The performances with accuracy as the chosen metric were the following: 0.4654 on the training set, 0.4610 on the validation set and 0.3422 on the test set.

1.b)

- Learning rate (η) = 0.01

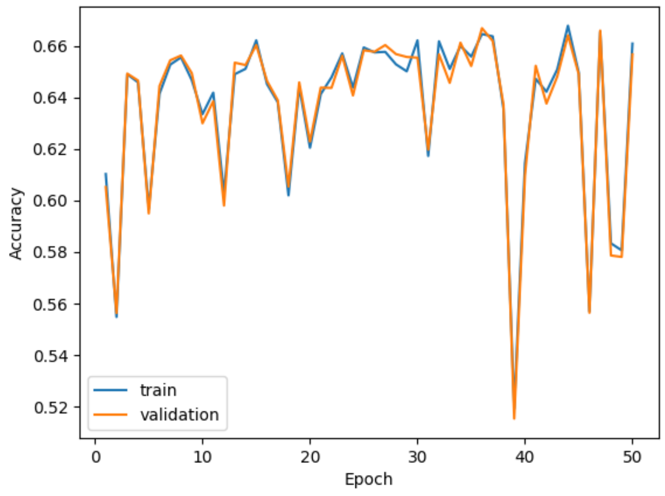


Figure 2 – Logistic Regression: train and validation accuracies as a function of the epoch number (η = 0.01)

The performances with accuracy as the chosen metric were the following: 0.6609 on the training set, 0.6568 on the validation set and 0.5784 on the test set.

- Learning rate (η) = 0.001

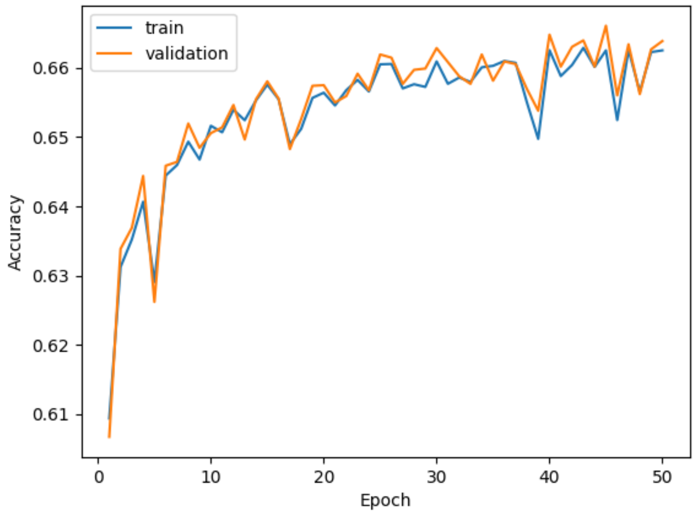


Figure 3 - Logistic Regression: train and validation accuracies as a function of the epoch number (η = 0.01)

The performances with accuracy as the chosen metric were the following: 0.6625 on the training set, 0.6639 on the validation set and 0.5936 on the test set.

Based on the plots obtained with two different learning rates (η = 0.01 and η = 0.001) we can determine that the model with η = 0.001 has better final results and has a more consistent learning process than the one with η = 0.01. This may happen because η = 0.01 may be too big, which can cause the algorithm to overshoot the optimal weights and potentially miss the convergence to the minimum.

2.a)

The claim is true. A logistic regression model calculates the sigmoid function of a linear combination ($b_0+b_1X_1+b_2X_2+\ldots+b_kX_k$) involving the input features (X), which in this case represent pixel values, and learned coefficients (b) during training. This linear nature signifies that it inherently assumes a linear relationship between the pixel values and the output. Moreover, logistic regression has the advantage of being a convex optimization problem, which means it has an unique global minimum during training.

A multi-layer perceptron using ReLU activations – which is a non-linear activation function – possesses the capacity to learn a broader spectrum of complex, non-linear relationships within input features, such as pixel values, capturing possible intricate patterns in the image data. If the activation function was linear, we wouldn't be able to learn those same patterns. However, the expressive power of an MLP with ReLU activations comes at the cost of a non-convex optimization problem, since the ReLU function doesn't have a unique global minimum.

The presence of multiple local minima is an added challenge to the training process, since it necessitates the use of optimization algorithms like stochastic gradient descent to navigate the parameter space in search of a local minimum. This difficulty in training contributed to the limited popularity of Neural Networks in machine learning from 1990 to 2010.

In summary, an MLP with ReLU activations, especially in the context of image data represented by pixel values, demonstrates greater expressiveness than a logistic regression model, as it can adeptly capture intricate patterns. Nonetheless, training logistic regression remains comparatively easier due to its convex optimization nature, offering a unique global minimum during the optimization process.

2.b)

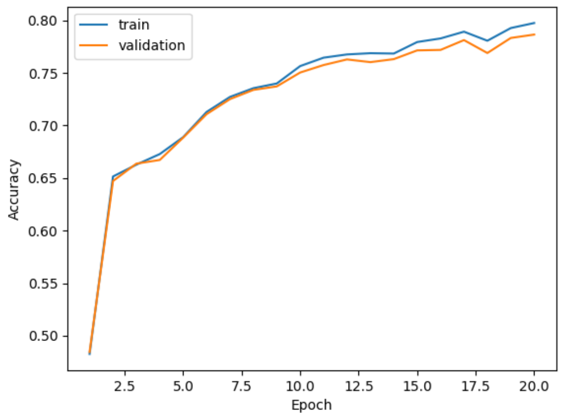


Figure 4 - MLP: train and validation accuracies as a function of the epoch number

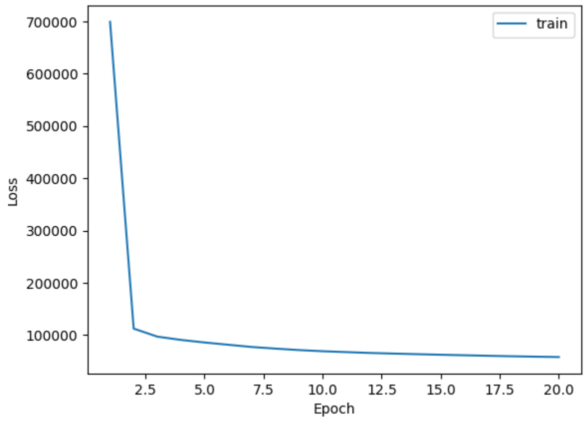


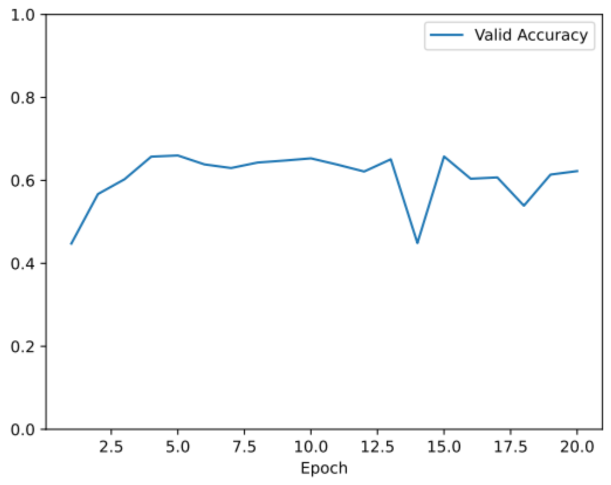
Figure 5 - MLP: train loss as a function of the epoch number

The performances with accuracy as the chosen metric were the following: 0.7975 on the training set, 0.7865 on the validation set and 0.7524 on the test set. The loss value was 58093.7315.

Question 2

1.

- Learning rate (η) = 0.1



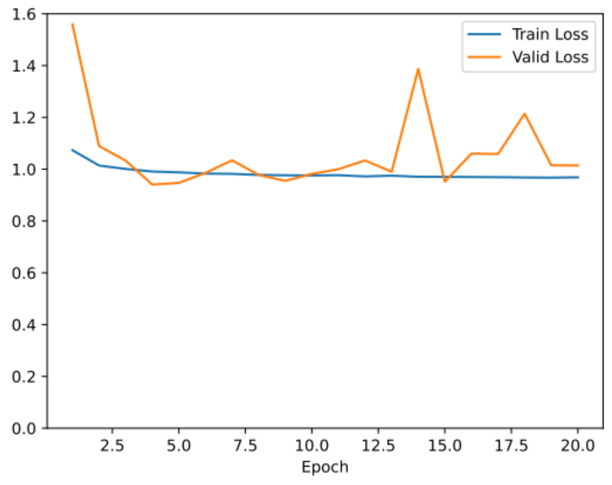
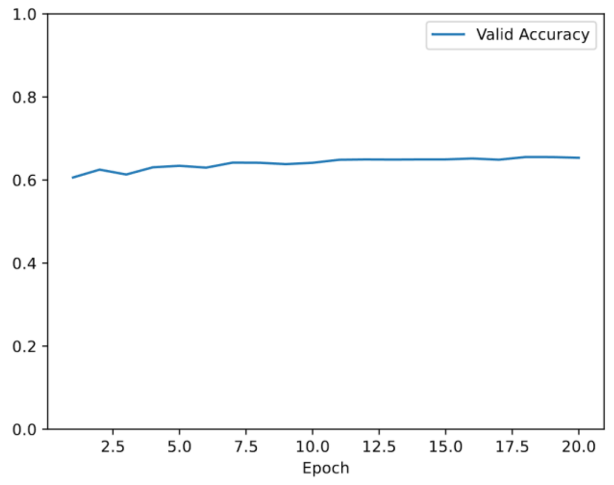


Figure 6 – PyTorch Logistic Regression: validation accuracy as a function of the epoch number (η = 0.1)

Figure 7 - PyTorch Logistic Regression: train and valid loss as a function of the epoch number (η = 0.1)

The final validation accuracy was 0.6224 and the final accuracy on the test set was 0.5577. The training loss value was 0.9687.

- Learning rate (η) = 0.01



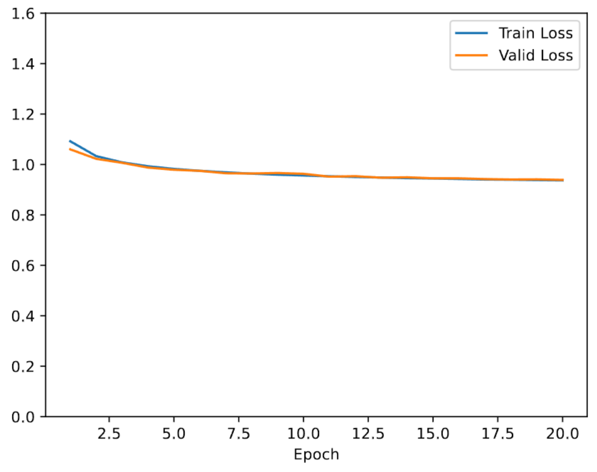


Figure 8 – PyTorch Logistic Regression: validation accuracy as a function of the epoch number (η = 0.01)

Figure 9 – PyTorch Logistic Regression: train and valid loss as a function of the epoch number (η = 0.01)

The final validation accuracy was 0.6535 and the final accuracy on the test set was 0.6200. The training loss value was 0.9370.

- Learning rate (η) = 0.001

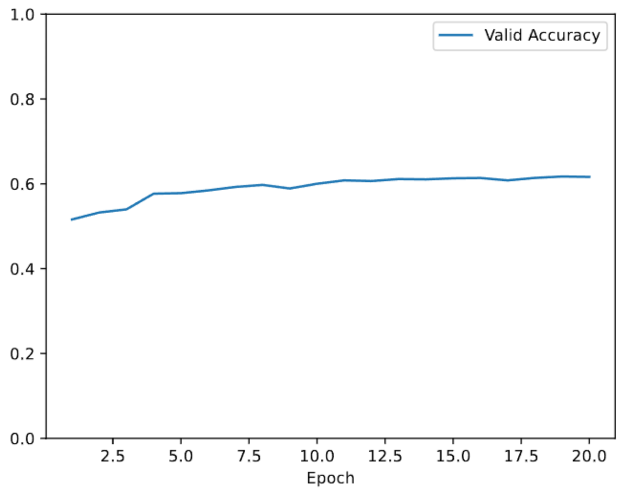


Figure 10 – PyTorch Logistic Regression: validation accuracy as a function of the epoch number (η = 0.001)

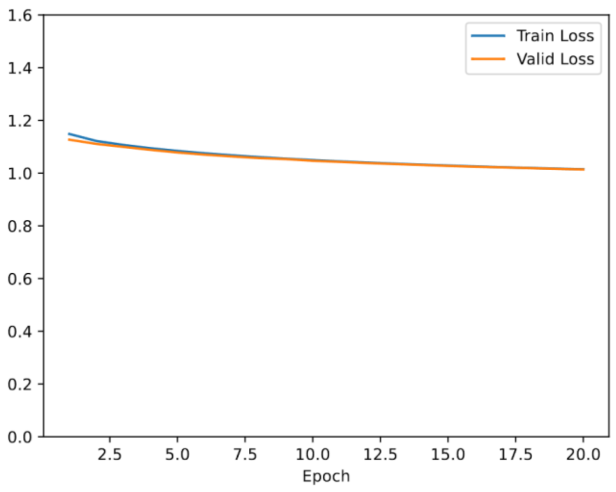


Figure 11 – PyTorch Logistic Regression: train and valid loss as a function of the epoch number (η = 0.001)

The final validation accuracy was 0.6163 and the final accuracy on the test set was 0.6503. The training loss value was 1.0145.

In terms of higher final validation accuracy, the best configuration is the one with learning rate = 0.01 with final accuracy on the test set = 0.6200.

2.a)

- Batch size = 16

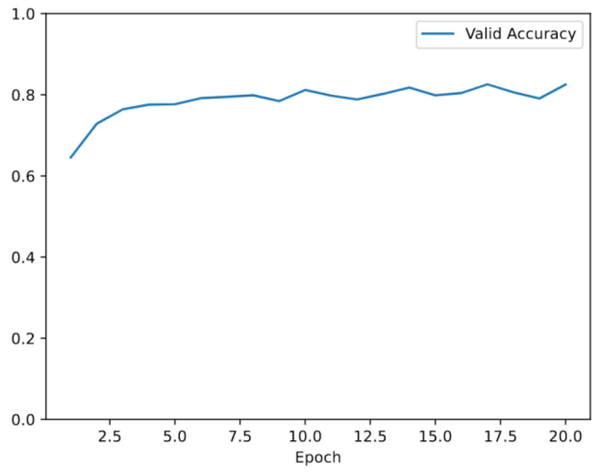


Figure 12 – PyTorch MLP: validation accuracy as a function of the epoch number (batch size = 16)

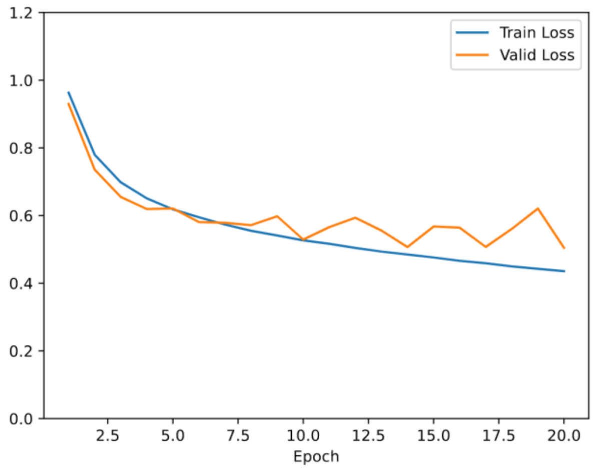


Figure 13 – PyTorch MLP: train and valid loss as a function of the epoch number (batch size = 16)

The final validation accuracy was 0.8253 and the final accuracy on the test set was 0.7505. The training loss value was 0.4358. Real time = 1m50.673s.

- Batch size = 1024

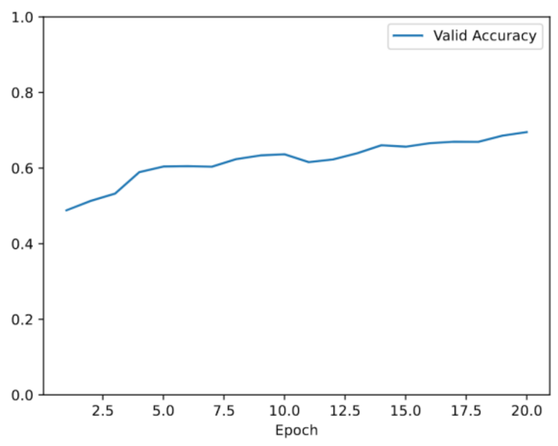


Figure 14 – PyTorch MLP: validation accuracy as a function of the epoch number (batch size = 1024)

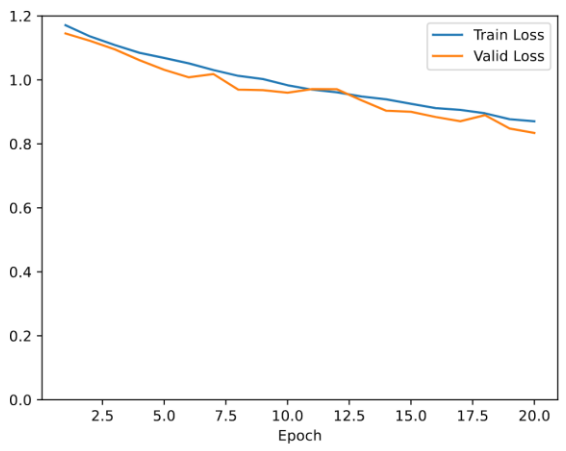


Figure 14 – PyTorch MLP: train and valid loss as a function of the epoch number (batch size = 1024)

The final validation accuracy was 0.6953 and the final accuracy on the test set was 0.7316. The training loss value was 0.8706. Real time = 22.545s.

The best test accuracy is 0.7505 on the configuration with batch size = 16. The performance is better in all parameters in this configuration and in terms of time, the difference between the two is 1m28.128s, being the batch size = 16 the slower one. This is in concordance to the general rule that a higher batch size leads to faster training because of modern hardware vectorization and parallelization capacities. About the performance, smaller batch sizes such as 16 have some advantages that can justify its better performance: they generalize better because they introduce some noise to the training data which prevents overfitting, they also converge faster during training, which makes the model adapt more quickly.

2.b)

- Learning rate (η) = 1

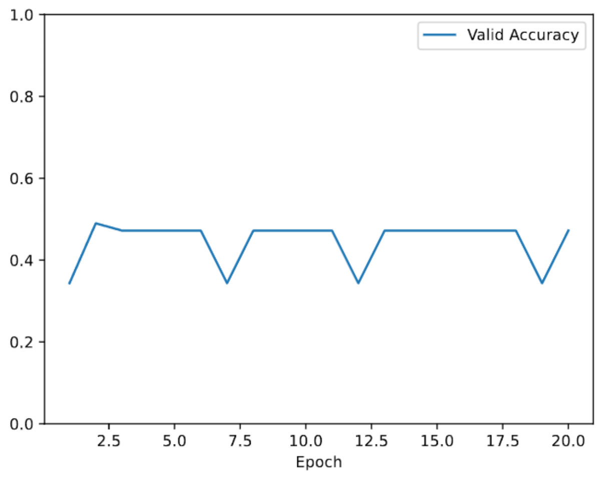


Figure 16 – PyTorch MLP: validation accuracy as a function of the epoch number (η = 1)

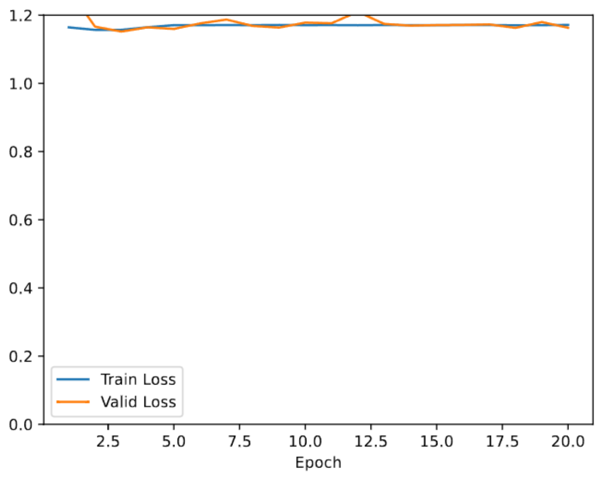


Figure 17 – PyTorch MLP: train and valid loss as a function of the epoch number (η = 1)

The best validation accuracy was 0.4898 (epoch number 2), the final validation accuracy was 0.4721 and the final accuracy on the test set was 0.4726. The training loss value was 1.1714.

- Learning rate (η) = 0.1

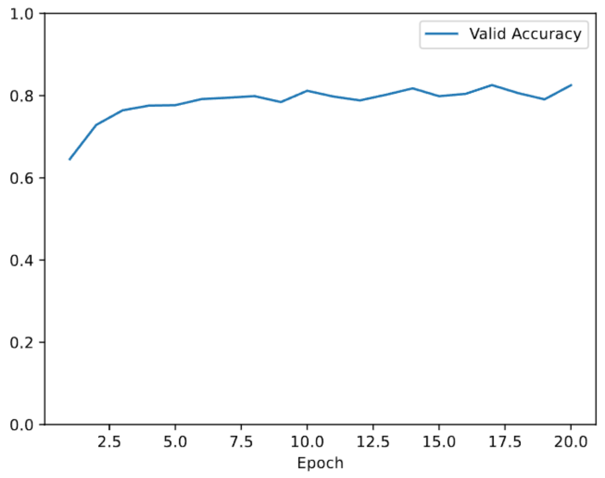


Figure 18 – PyTorch MLP: validation accuracy as a function of the epoch number (η = 0.1)

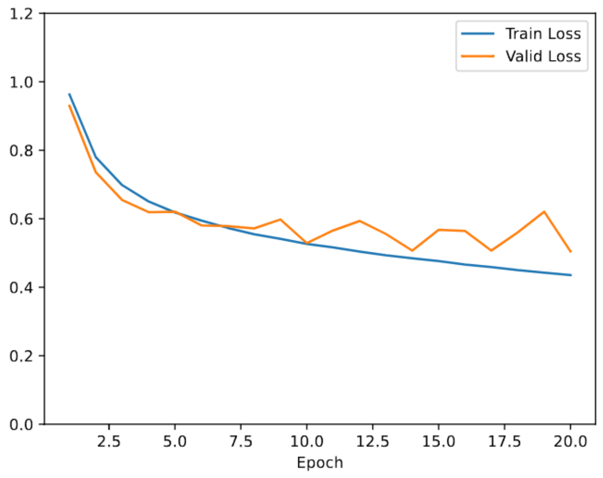


Figure 19 – PyTorch MLP: train and valid loss as a function of the epoch number (η = 0.1)

The best validation accuracy was 0.8258 (epoch number 17), the final validation accuracy was 0.8253 and the final accuracy on the test set was 0.7505. The training loss value was 0.4358.

- Learning rate (η) = 0.01

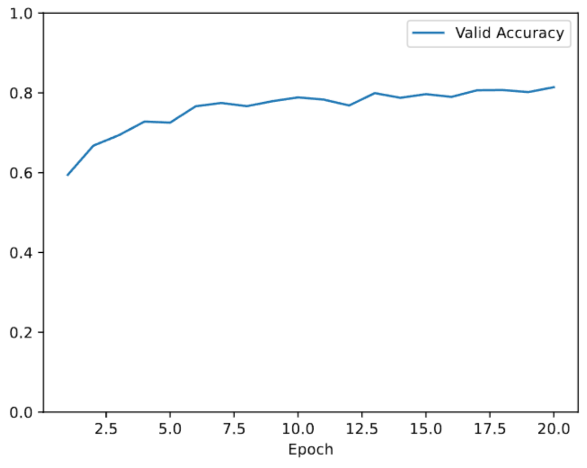


Figure 20 – PyTorch MLP: validation accuracy as a function of the epoch number (η = 0.01)

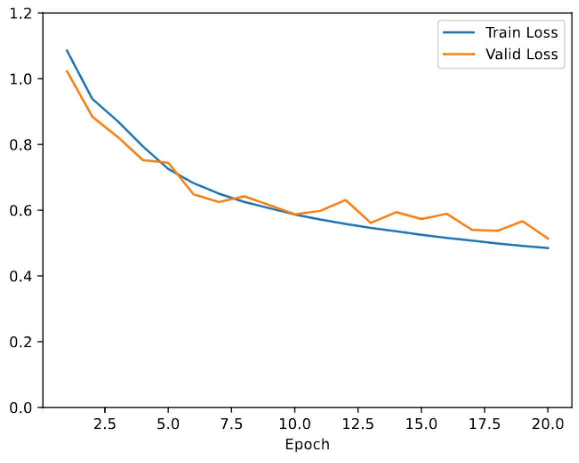


Figure 21 – PyTorch MLP: train and valid loss as a function of the epoch number (η = 0.01)

The final (and best) validation accuracy was 0.8143 and the final accuracy on the test set was 0.7637. The training loss value was 0.4848.

- Learning rate (η) = 0.001

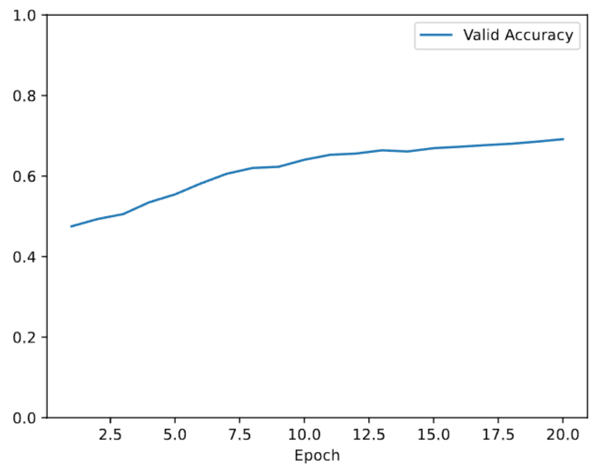


Figure 22 – PyTorch MLP: validation accuracy as a function of the epoch number (η = 0.001)

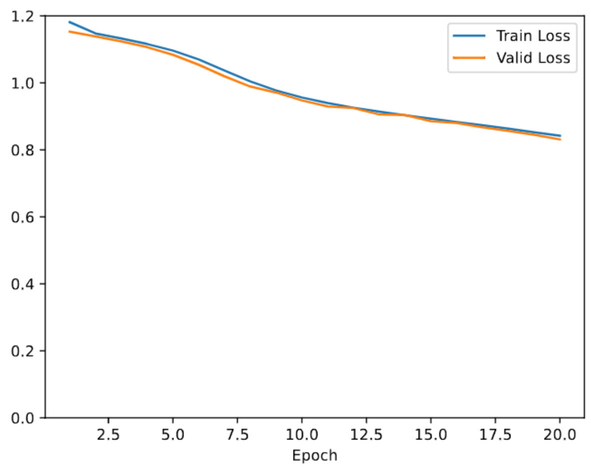


Figure 23 – PyTorch MLP: train and valid loss as a function of the epoch number (η = 0.001)

The final (and best) validation accuracy was 0.6917 and the final accuracy on the test set was 0.7146. The training loss value was 0.8421.

The best test accuracy is 0.7637, in the configuration with learning rate = 0.01.

The best configuration in terms of best validation accuracy, is the one with learning rate = 0.1.

The worst configuration in terms of best validation accuracy, is the one with learning rate = 1.

The difference in performance is extremely noticeable between the configuration with learning rate = 1 and all the other ones, being the first one the worst configuration among the four, as mentioned above. The configurations with learning rate = 0.1 and learning rate = 0.01 have extremely similar performance, the first one has the best validation accuracy and the second one has the best test accuracy. The configuration with learning rate = 0.001 has worse performance than the ones with learning rate = 0.1 and learning rate = 0.01, especially in terms of validation accuracy.

The performance is directly affected by the learning rates. One possible reason for the performance to be bad when the learning rate = 1 is because this particular learning rate is just too big for this model, which causes the algorithm to overshoot the optimal weights and diverge. For learning rates 0.1 or 0.01 the model seems to be a good fit, as the loss plots decrease then begin to stabilize, and they have the best values of validation accuracy and test accuracy. When the learning rate = 0.001 it's possible to notice from the loss plots that the model is clearly underfit: this may be because the network is converging too slowly.

2.c)

- Batch size = 256 | Epochs = 150

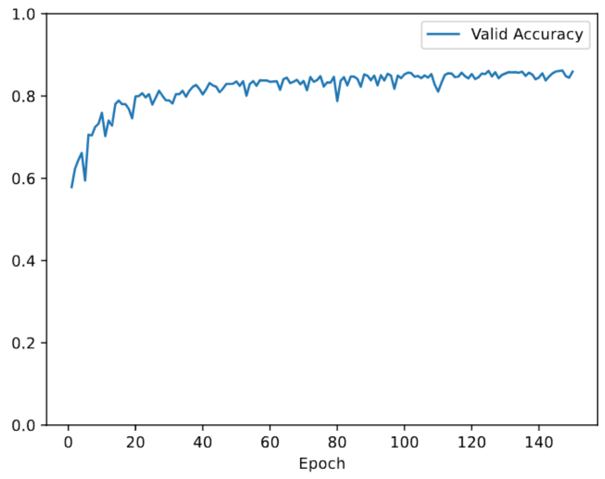


Figure 24 – PyTorch MLP: validation accuracy as a function of the epoch number

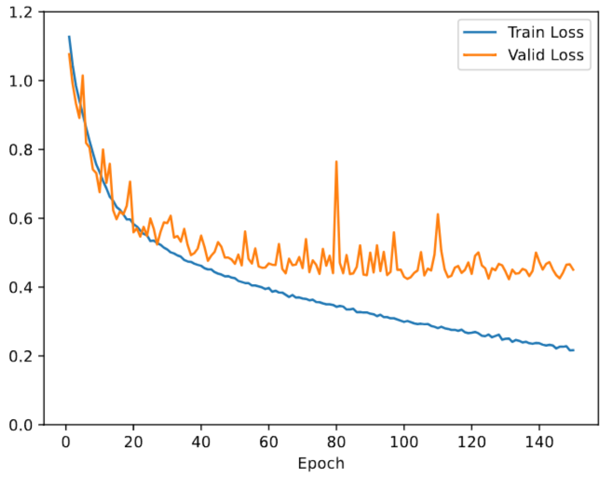


Figure 25 – PyTorch MLP: train and valid loss as a function of the epoch number

The best validation accuracy was 0.8620 (epoch number 147), the final validation accuracy was 0.8593 and the final accuracy on the test set was 0.7637. The training loss value was 0.2166.

Overfitting happens when the model learns the statistical noise or random fluctuations and the significant data (signal), instead of only the significant data, from the training dataset. In this model, there is overfitting because the plot of the training loss keeps decreasing, while the plot of validation loss keeps constantly decreasing and increasing instead of just decreasing to a point of stability.

The gap between the training and validation loss learning curves (generalization gap) is also significant which indicates overfitting, and it should have been minimal to be considered an optimal fit. This is because while a smaller gap would indicate that the model is generalizing well to unseen data, a bigger gap suggests that the model might not generalize well and could be overfitting the training data.

- Batch size = 256 | Epochs = 150 | L2 regularization = 0.0001

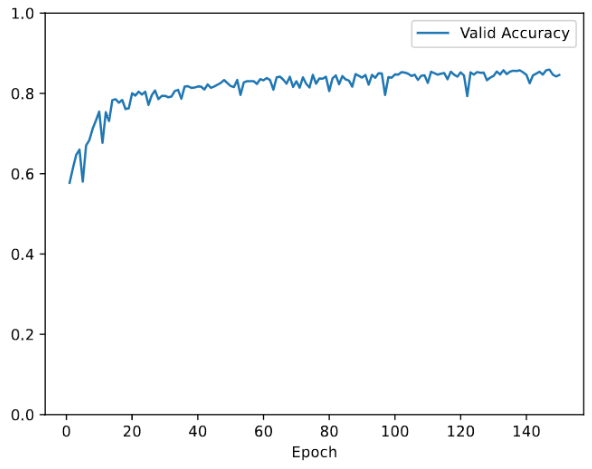


Figure 26 – PyTorch MLP: validation accuracy as a function of the epoch number

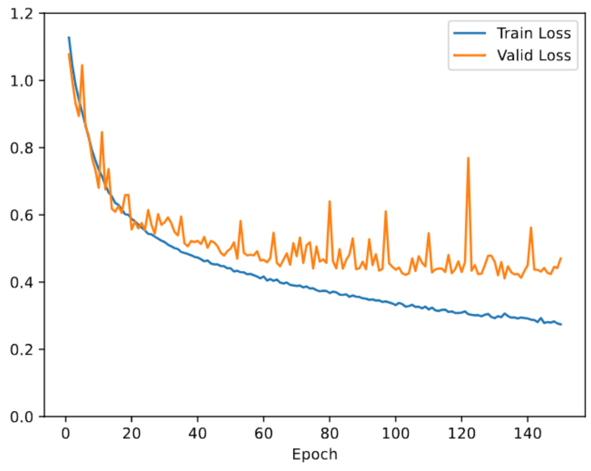


Figure 27 – PyTorch MLP: train and valid loss as a function of the epoch number

The best validation accuracy was 0.8595 (epoch number 147), the final validation accuracy was 0.8458, and the final accuracy on the test set was 0.7694. The training loss value was 0.2742.

- Batch size = 256 | Epochs = 150 | Dropout = 0.2

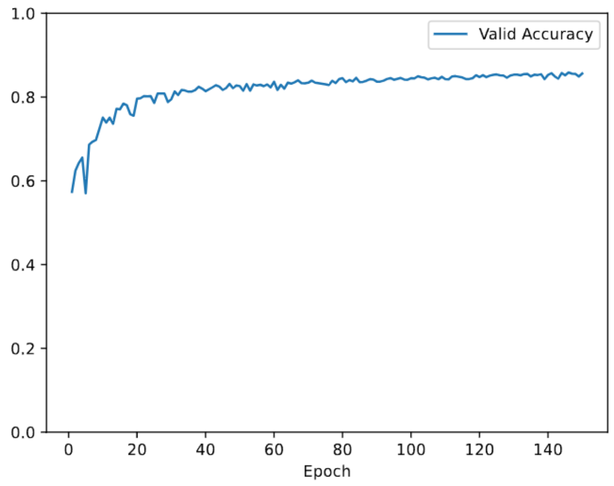


Figure 28 – PyTorch MLP: validation accuracy as a function of the epoch number

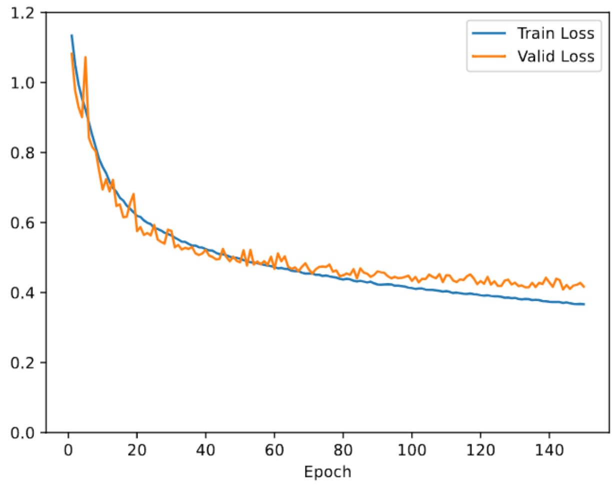


Figure 29 – PyTorch MLP: train and valid loss as a function of the epoch number

The best validation accuracy was 0.8588 (epoch number 146), the final validation accuracy was 0.8563 and the final accuracy on the test set was 0.7845. The training loss value was 0.3666.

The best configuration in terms of best validation accuracy, is the one without any technique.

The worst configuration in terms of best validation accuracy, is the one with dropout = 0.2.

The best test accuracy is 0.7845 in the configuration with dropout = 0.2.

Both L2 Regularization and Dropout probability are both techniques used to prevent overfitting. However, as we can see by comparing Figure 27 with 29, the configuration with dropout shows minimal overfitting (as the generalization gap is small), whereas with L2 Regularization while it reduced the overfitting in comparison with the default configuration, it wasn't nearly as effective as the one with the dropout (the generalization gap is still quite big). L2 Regularization imposes a penalty on large weights, so the neural network uses all features in the training data. Dropout probability, as the name suggests, is a technique in which there's a chance that random weights are set to 0. This prevents overfitting by introducing some randomness in the learning process.

The dropout having worse validation accuracy, but a better test accuracy may be because during training, it sets random weights to 0, which can cause a reduced capacity in learning but at the same time it effectively reduces the model's reliance on specific features which in turn might help the model's generalization ability and performance on unseen data (test set).

And, although they are mutually exclusive in this exercise they can also be used together.

Question 3

a)

Q3

1.

(a) The function Receives as input a vector of dimension D with both the input values and the output value belonging to $\{-1,1\}$

$f: \{-1,+1\}^D \rightarrow \{-1,+1\}$

$f(x) = \begin{cases} 1 & \text{if } \sum_{i=1}^{D} x_i \in [A,B] \\ -1 & \text{otherwise} \end{cases}$ in which A e B are integers such that $-D \leq A \leq B \leq D$

To show that the function above cannot be computed with a single perceptron we just need to show a case example in which, in the graphic, all the points that Require Output 1 can't be separated from all the points that require -1 by an hyperplane.

This is because single perceptrons can only learn linear separable functions and can't handle non-linear ones.

If we choose dimension 2 $(D=2)$ and $A=-1$, $B=1$, we can enumerate all possible inputs and their outputs:

- For $x = [-1\ -1]^T$ the output is -1

  $\sum_{i=1}^{2} x_i = -2 \qquad -2 \notin [-1,1]$

  $f(x) = -1$

- For $x = [1\ 1]^T$ the output is -1

  $\sum_{i=1}^{2} x_i = 2 \qquad 2 \notin [-1,1]$

  $f(x) = -1$

- For $x = [1\ -1]^T$ the output is 1

  $\sum_{i=1}^{2} x_i = 0 \qquad 0 \in [-1,1]$

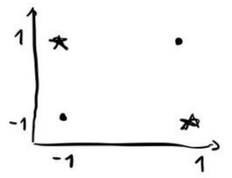
  $f(x) = 1$

- For $x = [-1\ 1]^T$ the output is 1

  $\sum_{i=1}^{2} x_i = 0 \qquad 0 \in [-1,1]$

  $f(x) = 1$

Since $D=2$, an hyperplane is just a line.



★ = 1
● = -1

As we can see there is no line that can be used to separate the negative instances from the positive ones.

As such this is a case example of the function in which the data can't be linearly separable.

Since there are cases in which the function is not linearly separable, then a single perceptron can't compute this function.

b)

(b) Since we have as input a vector of dimension D and a multilayer perceptron with a single hidden layer with 2 hidden units:

$$\underset{W^{T(1)}}{2\overset{D}{\left[\quad\right]}}\ \underset{x}{{}_{D}\overset{1}{\left[\quad\right]}} + \underset{b^{(1)}}{2\overset{1}{\left[\quad\right]}} = \underset{z^{(1)}}{2\overset{1}{\left[\quad\right]}} \qquad\qquad \underset{W^{T(2)}}{1\overset{2}{\left[\quad\right]}}\ \underset{h^{(1)}}{{}_{2}\overset{1}{\left[\quad\right]}} + \underset{b^{(2)}}{1\overset{1}{\left[\quad\right]}} = \underset{z^{(2)}}{1\overset{1}{\left[\quad\right]}}$$

$$z^{(1)} = W^{T(1)}x + b^{(1)} \qquad\qquad z^{(2)} = W^{T(2)}h^{(1)} + b^{(2)}$$

$$h^{(1)} = \operatorname{sign}(z^{(1)}) \qquad\qquad h^{(2)} = \operatorname{sign}(z^{(2)})$$

To show that the function can be computed with a multilayer perceptron we need to ensure that when the input $\sum_{i=1}^{D} x_i \in [A,B]$, the perceptron outputs 1, and -1 otherwise.

Since we want to ensure the resulting network is robust to infinitesimal perturbation of the inputs, the boundary will be $[A-\varepsilon, B+\varepsilon]$ instead ($\varepsilon \in \mathbb{Q}$ is a constant which represents an increase to the boundary such that it envelops the possible infinitesimal perturbation of the inputs)

Since we have 2 hidden units, they can be used to learn if the sum of x belongs to the boundary:

- One to check if $\sum_{i=1}^{D} x_i$ is bigger than $A-\varepsilon$ (hidden unit 1)
- Other to check if $\sum_{i=1}^{D} x$ is smaller than $B+\varepsilon$ (hidden unit 2)

The hidden unit uses sign(z) as activation function, so the possible outputs of each hidden unit are 1 or -1 and since we want to be able to distinguish when the output belongs to the boundary or not, each hidden unit should output the same value (1 or -1) depending if its condition is true or not.
If the condition is true, the value should be positive so sign(+) outputs 1 (since $f(x)=1$ if $\sum_{i=1}^{D} x \in [A,B]$) otherwise should be negative, sign(-) = -1.

$$\begin{cases} \sum_{i=1}^{D} x_i \geq A-\varepsilon \\ \sum_{i=1}^{D} x_i \leq B+\varepsilon \end{cases} \Leftrightarrow \begin{cases} \sum_{i=1}^{D} x_i - A + \varepsilon \geq 0 \\ \sum_{i=1}^{D} x_i - B - \varepsilon \leq 0 \end{cases} \Leftrightarrow \begin{cases} \sum_{i=1}^{D} x_i - A + \varepsilon \geq 0 \\ -\sum_{i=1}^{D} x_i + B + \varepsilon \geq 0 \end{cases} \Leftrightarrow$$

(if the condition is true, the output needs to be positive)

$$\varepsilon = \frac{\varepsilon_1 \in \mathbb{Z}}{\varepsilon_2 \in \mathbb{Z}}$$

Any rational number can be converted to integer

$$\Leftrightarrow \begin{cases} \sum_{i=1}^{D} x_i - A + \frac{\varepsilon_1}{\varepsilon_2} \geq 0 \\ -\sum_{i=1}^{D} x_i + B + \frac{\varepsilon_1}{\varepsilon_2} \geq 0 \end{cases} \Leftrightarrow \begin{cases} \sum_{i=1}^{D} x_i \varepsilon_2 - A\varepsilon_2 + \varepsilon_1 \geq 0 \\ -\sum_{i=1}^{D} x_i \varepsilon_2 + B\varepsilon_2 + \varepsilon_1 \geq 0 \end{cases}$$

Hidden unit 1: $z = \underbrace{\varepsilon_2}_{w} \underbrace{\sum_{i=1}^{D} x_i}_{x} + \underbrace{\varepsilon_1 - A\varepsilon_2}_{b}$ Hidden unit 2: $z = \underbrace{-\varepsilon_2}_{w} \underbrace{\sum_{i=1}^{D} x_i}_{x} + \underbrace{\varepsilon_1 + B\varepsilon_2}_{b}$

$$W^{T(1)} = \begin{bmatrix} \varepsilon_2 \dots \varepsilon_2 \\ -\varepsilon_2 \dots -\varepsilon_2 \end{bmatrix} \qquad b^{(1)} = \begin{bmatrix} \varepsilon_1 - A\varepsilon_2 \\ \varepsilon_1 + B\varepsilon_2 \end{bmatrix}$$

For the output layer, it receives the sums of the hidden unit's outputs (considering all w = 1 and b = 0)

This sum can be either:

(a) 2 if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, B+\varepsilon]$

(b) 0 if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \notin]-\infty, B+\varepsilon]$ OR $\sum_{i=1}^{D} x_i \notin [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \in]-\infty, B+\varepsilon]$

The output layer activation function is sign(z), so we need (b) to become a negative number such that not belonging to the boundary outputs -1 (because $\text{sign}(0) = 1$ and $f(x) = -1$ if $\sum_{x=1}^{D} x_i \notin [A, B]$)

Bias needs to be either -1 or -2 (so sign(a) is still +1).

$W^{T(2)} = [1 \ \ 1] \quad b^{(2)} = [-1]$

(a) $z = 2 - 1 = 1$
(b) $z = 0 - 1 = -1$

Then, instead we have as $z^{(2)}$:

(a) 1 if $\sum_{i=1}^{D} x_i \in [A, B]$

(b) -1 if $\sum_{i=1}^{D} x_i \in [A-\epsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \notin]-\infty, B+\epsilon]$ **OR** $\sum_{i=1}^{D} x_i \notin [A-\epsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \in]-\infty, B+\epsilon]$

$g(a) = \text{sign}(1) = 1$
$g(b) = \text{sign}(-1) = -1$
which matches
$f(x) = 1$ if $\sum_{i=1}^{D} x_i \in [A, B]$
$f(x) = -1$ if otherwise

As such the function f can be computed with a multilayer perceptron with a single hidden layer with two hidden units and sign(z) as activation function for both hidden and output layers.

c)

(c) Since we have as input a vector of dimension D and a multilayer perceptron with a single hidden layer with 2 hidden units:

$$2\underset{W^{T(1)}}{\overset{D}{\left[\qquad\right]}}D\underset{x}{\overset{1}{\left[\quad\right]}} + 2\underset{b^{(1)}}{\overset{1}{\left[\quad\right]}} = 2\underset{z^{(1)}}{\overset{1}{\left[\quad\right]}} \qquad 1\underset{W^{T(2)}}{\overset{2}{\left[\qquad\right]}}2\underset{h^{(1)}}{\overset{1}{\left[\quad\right]}} + 1\underset{b^{(2)}}{\overset{1}{\left[\quad\right]}} = 1\underset{z^{(2)}}{\overset{1}{\left[\quad\right]}}$$

$$z^{(1)} = W^{T(1)}x + b^{(1)} \qquad z^{(2)} = W^{T(2)}h^{(1)} + b^{(2)}$$
$$h^{(1)} = \mathrm{ReLU}(z^{(1)}) \qquad h^{(2)} = \mathrm{sign}(z^{(2)})$$

To show that the function can be computed with a multilayer perceptron we need to ensure that when the input $\sum_{i=1}^{D} x_i \in [A,B]$, the perceptron outputs 1, and -1 otherwise.

Since we want to ensure the resulting network is robust to infinitesimal perturbation of the inputs, the boundary will be $[A-\varepsilon, B+\varepsilon]$ instead ($\varepsilon \in \mathbb{Q}$ is a constant which represents an increase to the boundary such that it envelops the possible infinitesimal perturbation of the inputs)

Since we have 2 hidden units, they can be used to learn if the sum of x belongs to the boundary:

- One to check if $\sum_{i=1}^{D} x_i$ is bigger than $A-\varepsilon$ (hidden unit 1)
- Other to check if $\sum_{i=1}^{D} x$ is smaller than $B+\varepsilon$ (hidden unit 2)

The activation function used in the hidden units is ReLU.

Relu(- or 0) = 0

ReLU(+) = +

As such we want the conditions to be true if the output in the hidden units is a negative number (- to represent negative numbers and + to represent positive numbers). This is to apply weight -1 before the output layer so we can convert the + to - and mantain the 0 as 0 to use g(z) after.

(a) − if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, B+\varepsilon]$

(b) + if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \notin]-\infty, B+\varepsilon]$ OR $\sum_{i=1}^{D} x_i \notin [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \in]-\infty, B+\varepsilon]$

(so the output is negative if the condition is true)

$$\begin{cases}\sum_{i=1}^{D} x_i \geq A-\varepsilon \\ \sum_{i=1}^{D} x_i \leq B+\varepsilon\end{cases} \Leftrightarrow \begin{cases}\sum_{i=1}^{D} x_i - A + \varepsilon \geq 0 \\ \sum_{i=1}^{D} x_i - B - \varepsilon \leq 0\end{cases} \Leftrightarrow \begin{cases}-\sum_{i=1}^{D} x_i + A - \varepsilon \leq 0 \\ \sum_{i=1}^{D} x_i - B - \varepsilon \leq 0\end{cases}$$

Any rational number can be converted to integer

$\varepsilon = \frac{\varepsilon_1}{\varepsilon_2}$ $\varepsilon_1 \in \mathbb{Z}$, $\varepsilon_2 \in \mathbb{Z}$

$\Leftrightarrow$

$$\begin{cases}-\sum_{i=1}^{D} x_i + A - \frac{\varepsilon_1}{\varepsilon_2} \leq 0 \\ \sum_{i=1}^{D} x_i - B - \frac{\varepsilon_1}{\varepsilon_2} \leq 0\end{cases} \Leftrightarrow \begin{cases}-\sum_{i=1}^{D} x_i \varepsilon_2 + A\varepsilon_2 - \varepsilon_1 \leq 0 \\ \sum_{i=1}^{D} x_i \varepsilon_2 + (-B\varepsilon_2 - \varepsilon_1) \leq 0\end{cases}$$

Hidden unit 1: $z = \underbrace{-\varepsilon_2}_{w} \underbrace{\sum_{i=1}^{D} x_i}_{x} + \underbrace{A\varepsilon_2 - \varepsilon_1}_{b}$ Hidden unit 2: $z = \underbrace{\varepsilon_2}_{w} \underbrace{\sum_{i=1}^{D} x_i}_{x} + \underbrace{(-B\varepsilon_2 - \varepsilon_1)}_{b}$

$$W^{T(1)} = \begin{bmatrix} -\varepsilon_2 \dots -\varepsilon_2 \\ \varepsilon_2 \dots \varepsilon_2 \end{bmatrix} \quad b^{(1)} = \begin{bmatrix} -\varepsilon_1 + A\varepsilon_2 \\ -\varepsilon_1 - B\varepsilon_2 \end{bmatrix}$$

For the output layer, it receives the sums of the hidden unit's outputs (considering all $w = 1$ and $b = 0$)

This sum can be either:

(a) 0 if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, B+\varepsilon]$

(b) $+$ (positive numbers) if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \notin]-\infty, B+\varepsilon]$ OR $\sum_{i=1}^{D} x_i \notin [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \in]-\infty, B+\varepsilon]$

Since we know $\text{sign}(0) = 1$, only b doesn't output the correct result according to $f(x)$.

To fix this, we apply weight -1 to all the outputs from the hidden units:

(a) $z = 0 \times (-1) = 0$

(b) $z = (+) \times (-1) = -$

$$W^{T(2)} = [-1 \ -1] \quad b^{(2)} = [\ 0\ ]$$

Now we have:

(a) 0 if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, B+\varepsilon]$

(b) $-$ (negative numbers) if $\sum_{i=1}^{D} x_i \in [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \notin]-\infty, B+\varepsilon]$ OR $\sum_{i=1}^{D} x_i \notin [A-\varepsilon, +\infty[$ and $\sum_{i=1}^{D} x_i \in]-\infty, B+\varepsilon]$

$g(a) = \text{sign}(0) = 1$

$g(b) = \text{sign}(-) = -1$

which matches $f(x) = 1$ if $\sum_{i=1}^{D} x_i \in [A, B]$
$f(x) = -1$ if otherwise

As such the function f can be computed with a multilayer perceptron with a single hidden layer with two hidden units and $\text{ReLU}(z)$ as activation function for the hidden layer and $\text{sign}(z)$ in the output layer.