Instituto Superior Técnico

Deep Learning 2023

Homework 1 - Development and results

Report

Name	**IST Number**
Davi Giordano Valério	108497
Vicente Lorenzo	92569

Lisbon
2023

INSTITUTO SUPERIOR TÉCNICO

Homework 1 - Development and results

Report

Report of homework 1 to prof. André Martins at the Deep Learning course, at Instituto Superior Técnico, in Lisbon, Portugal.

Lisbon, Portugal
2023

Contents

List of Figures

0. Introduction

This document reports the results of homework 1 from the Deep Learning course at Instituto Superior Técnico. Its main purpose is to implement different linear classifiers for a medical image classification problem, using the OCTMNIST dataset, available at https://medmnist.com/ with the CC BY 4.0 licence.

The dataset contains 109,309 valid optical coherence tomography (OCT) images for retinal diseases and presents a multi-class classification problem.

The members organized the tasks in the project according to Table 1. The difficulty of each task was estimated so as to attribute an equal ammount of complexity to each member of the group (26 points).

In this project, python 3.11 was used.

Item	Who was responsible	Estimated complexity
Solved question 1 item 1.a	Davi Giordano	1
Solved question 1 item 1.b	Davi Giordano	1
Solved question 1 item 2.a	Vicente Lorenzo	3
Solved question 1 item 2.b	Vicente Lorenzo	5
Solved question 2 item 1	Vicente Lorenzo	5
Solved question 2 item 2.a	Vicente Lorenzo	5
Solved question 2 item 2.b	Vicente Lorenzo	3
Solved question 2 item 2.c	Vicente Lorenzo	3
Solved question 3 item 1.a	Davi Giordano	1
Solved question 3 item 1.b	Davi Giordano	3
Solved question 3 item 1.c	Davi Giordano	5
Wrote the report on question 1 item 1.a	Vicente Lorenzo	1
Wrote the report on question 1 item 1.b	Vicente Lorenzo	1
Wrote the report on question 1 item 2.a	Davi Giordano	1
Wrote the report on question 1 item 2.b	Davi Giordano	1
Wrote the report on question 2 item 1	Davi Giordano	1
Wrote the report on question 2 item 2.a	Davi Giordano	1
Wrote the report on question 2 item 2.b	Davi Giordano	1
Wrote the report on question 2 item 2.c	Davi Giordano	1
Wrote the report on question 3 item 1.a	Davi Giordano	3
Wrote the report on question 3 item 1.b	Davi Giordano	3
Wrote the report on question 3 item 1.c	Davi Giordano	3

Table 1: Division of tasks in homework 1

1. Question 1

1.1 Item 1.a

Below, the implementation of the `update_weight` function for the perceptron class can be seen.

```
def update_weight(self, x_i, y_i, **kwargs):
    """
    x_i (n_features): a single training example
    y_i (scalar): the gold label for that example
    other arguments are ignored
    """
    y_hat_i = np.argmax(np.dot(self.W, x_i))
    if y_hat_i != y_i:
        self.W[y_i] += x_i
        self.W[y_hat_i] -= x_i
```

With this implementation, the perceptron was trained on the OCTMNIST dataset for 20 epochs. Its accuracy during training and validation is can be visualized in Figure 1.1. The model showed an accuracy of 0.3422 on the test dataset.

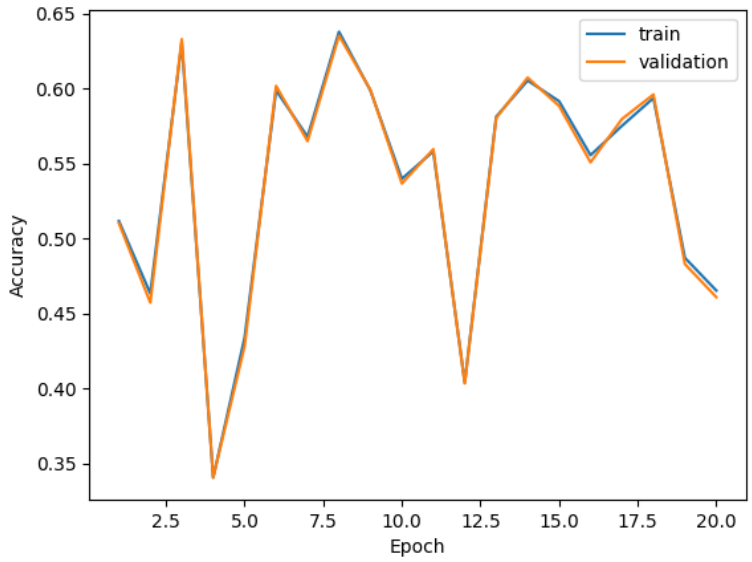


Figura 1.1: Accuracy during training and validation for the perceptron

1.2 Item 1.b

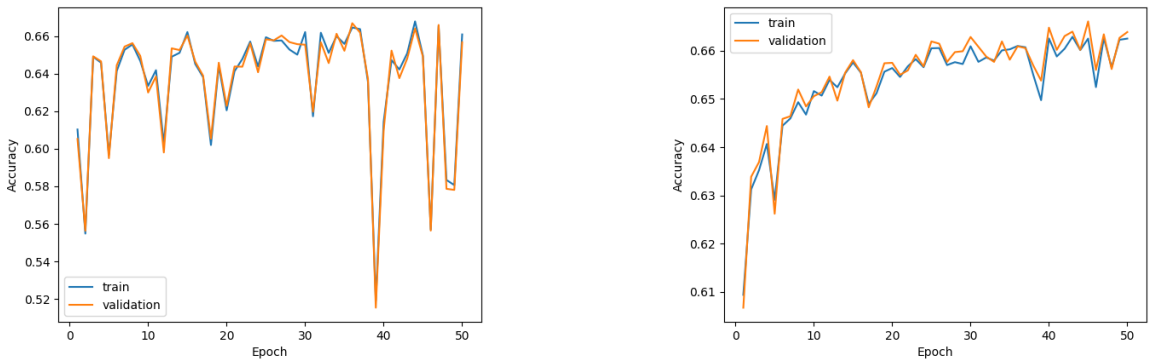
Below, the implementation of the `update_weight` function for the logistic regression class can be seen.

```
def update_weight(self, x_i, y_i, learning_rate=0.001):
    """
    x_i (n_features): a single training example
    y_i: the gold label for that example
    learning_rate (float): keep it at the default value for your plots
    """
    scores = np.dot(self.W, x_i)
    exp_scores = np.exp(scores - np.max(scores))
    probabilities = exp_scores / np.sum(exp_scores)
    y_one_hot = np.zeros(self.W.shape[0])
    y_one_hot[y_i] = 1
    gradient = np.outer(probabilities - y_one_hot, x_i)
    self.W -= learning_rate * gradient
```

With this implementation, the logistic regression model was trained on the OCTMNIST dataset for 50 epochs with a learning rate of 0.01 and of 0.001.

The accuracy during training and validation for learning rate equal to 0.01 can be seen in Figure 1.2a. With this configuration, the final test accuracy was 0.5784.

Meanwhile, the accuracy during training and validation for learning rate equal to 0.001 can be seen in Figure 1.2b. With this configuration, the final test accuracy was 0.5936.



(a) Learning rate 0.01

(b) Learning rate 0.001

Figura 1.2: Accuracy during training and validation for the logistic regression

Comparing the accuracies of the two models, it can be seen that model with the lowest learning rate has a more stable training. This means it converges to a final value, whereas the other model diverges.

1.3 Item 2.a

The claim is true. A multi-layer perceptron (MLP) using relu activations is more expressive than a logistic regression, but it is harder to train.

The MLP is more expressive because it can learn non-linear relationships between the pixels of the image, due to the ReLU activation function and the connections between the multiple layers. Thus, its hidden layers can represent more complex patterns. The logistic regression, however, can only learn linear patterns, since it is only capable of fitting a linear hyperplane as decision boundary.

However, the loss function of the Logistic Regression is convex. As a result, the model will reach its global minimum with a sufficient number epochs, which means it is a simple model to train. On the other hand, the MLP has local minima and saddle points, which can make it harder to find the global minimum of the loss function. This makes it harder to train than the Logistic Regression.

1.4 Item 2.b

Below, the implementation of the `MLP` class can be seen.

```
class MLP(object):
    def __init__(self, n_classes, n_features, hidden_size):
        mu, sigma = 0.1, 0.1  # Mean and standard deviation for weight initialization

        # Initialize weights and biases for the hidden layer
        self.W1 = np.random.normal(mu, sigma, (hidden_size, n_features))
        self.b1 = np.zeros(hidden_size)

        # Initialize weights and biases for the output layer
        self.W2 = np.random.normal(mu, sigma, (n_classes, hidden_size))
        self.b2 = np.zeros(n_classes)

    @staticmethod
```

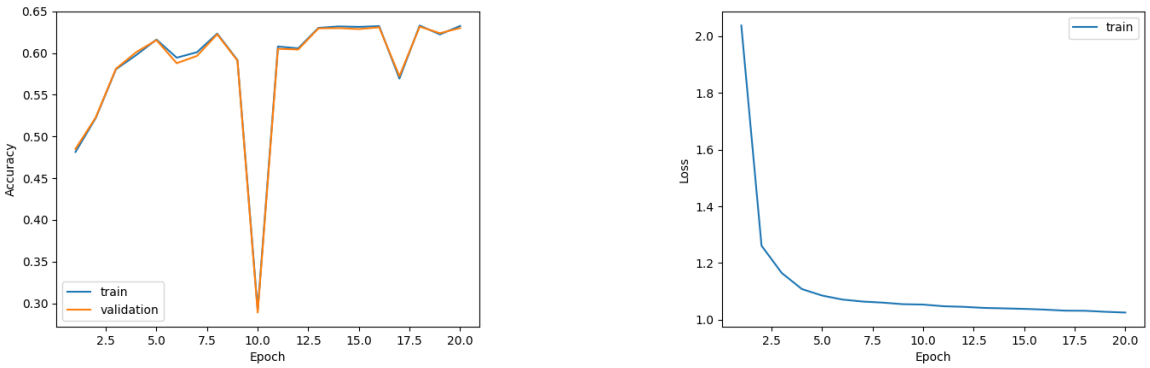
```
    def relu(z):
        # ReLU activation function
        return np.maximum(0, z)

    @staticmethod
    def softmax(z):
        # Adjusted Softmax activation function for single processing
        exp_z = np.exp(z - np.max(z))
        return exp_z / np.sum(exp_z)

    def forward_pass(self, x_i):
        # Forward Pass function for single processing
        # Compute activations for the hidden layer
        z1_i = np.dot(self.W1, x_i) + self.b1
        a1_i = self.relu(z1_i)

        # Compute activations for the output layer
        z2_i = np.dot(self.W2, a1_i) + self.b2
        a2_i = self.softmax(z2_i)
        return a1_i, a2_i

    def forward_pass_batch(self, X):
        # Forward Pass function for batch processing
        # Compute activations for the hidden layer
        z1 = np.dot(X, self.W1.T) + self.b1
        a1 = self.relu(z1)

        # Compute activations for the output layer
        z2 = np.dot(a1, self.W2.T) + self.b2
        a2 = self.softmax(z2)
        return a1, a2

    def predict(self, X):
        # Predict class labels for a batch of inputs
        _, a2 = self.forward_pass_batch(X)
        return np.argmax(a2, axis=1)

    def evaluate(self, X, y):
        """
```

```
        X (n_examples x n_features):
        y (n_examples): gold labels
        """
        y_hat = self.predict(X)
        n_correct = (y == y_hat).sum()
        n_possible = y.shape[0]
        return n_correct / n_possible

    def backward_pass(self, x_i, y_i, a1_i, a2_i, learning_rate):
        # Convert y to one-hot encoding
        y_one_hot = np.zeros_like(a2_i)
        y_one_hot[y_i] = 1

        # Compute gradients for the output layer
        dZ2 = a2_i - y_one_hot
        dW2 = np.outer(dZ2, a1_i)
        db2 = np.sum(dZ2)

        # Compute gradients for the hidden layer
        dA1 = np.dot(dZ2, self.W2)
        dZ1 = dA1 * (a1_i > 0)  # Derivative of ReLU
        dW1 = np.outer(dZ1, x_i)
        db1 = np.sum(dZ1)

        # Update weights and biases
        self.W1 -= learning_rate * dW1
        self.b1 -= learning_rate * db1
        self.W2 -= learning_rate * dW2
        self.b2 -= learning_rate * db2

        # Compute and return the loss
        loss = -np.sum(y_one_hot * np.log(a2_i + 1e-8))
        return loss

    def train_epoch(self, X, y, learning_rate=0.001):
        # Shuffle the dataset
        num_examples = X.shape[0]
        indices = np.arange(num_examples)
        np.random.shuffle(indices)
```

```
        X = X[indices]
        y = y[indices]

        total_loss = 0

        # Stochastic gradient descent by processing each example individually
        for x_i, y_i in zip(X, y):
            # Forward pass for a single example
            a1_i, a2_i = self.forward_pass(x_i)

            # Backward pass and update weights for a single example
            loss = self.backward_pass(x_i, y_i, a1_i, a2_i, learning_rate)
            total_loss += loss

        # Return average loss over the epoch
        return total_loss / num_examples
```

With this implementation, the logistic regression model was trained on the OCTMNIST dataset for 20 epochs with a learning rate of 0.001.

The accuracy during training and validation for learning rate equal to 0.01 can be seen in Figure 1.3a. In addition, the Loss Function can be visualized in Figure 1.3b. With this model, the final test accuracy was 0.6352.



(a) Accuracy (b) Loss function

Figura 1.3: Accuracy and Loss Function during training and validation for the MLP

2. Question 2

2.1 Item 1

Below, the implementation of `__init__()` and `forwards()` for the `LogisticRegression` using Pytorch can be seen.

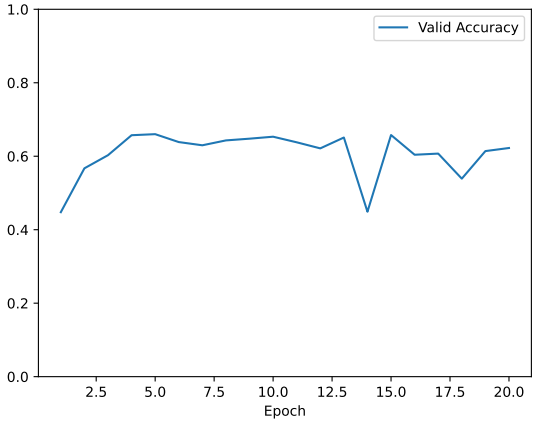
```
class LogisticRegression(nn.Module):

    def __init__(self, n_classes, n_features, **kwargs):
        """
        n_classes (int)
        n_features (int)
        """
        super().__init__()
        self.linear = nn.Linear(n_features, n_classes)

    def forward(self, x, **kwargs):
        """
        x (batch_size x n_features): a batch of training examples
        """
        return self.linear(x)
```

And below, the implementation of the `train_batch()` function:

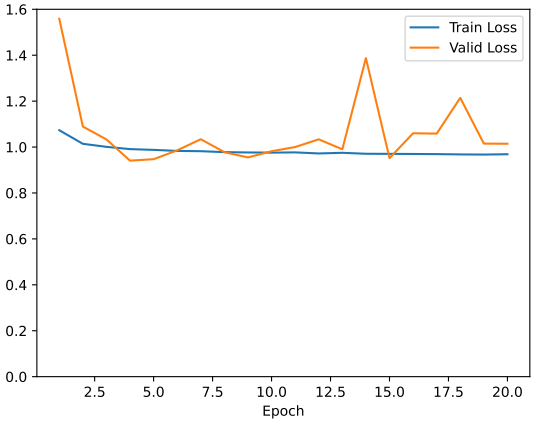
```
def train_batch(X, y, model, optimizer, criterion, **kwargs):
    """
    X (n_examples x n_features)
    y (n_examples): gold labels
    """
    # Zero the gradients before running the backward pass.
    optimizer.zero_grad()

    # Forward pass: Compute predicted y by passing X to the model
    y_hat = model(X)

    # Compute and print loss
    loss = criterion(y_hat, y)

    # Backward pass: compute gradient of the loss with respect to model parameters
```

```
16      loss.backward()
17
18      # Calling the step function on an Optimizer makes an update to its parameters
19      optimizer.step()
20
21      return loss.item()
```

With this implementation, the logistic regression model was trained on the OCTMNIST dataset for 20 epochs with a learning rate of 0.1, 0.01 and of 0.001. The accuracy and the loss function during training and validation for learning rate equal to 0.1, 0.01 and 0.001 can be seen in Figures 2.1, 2.2 and 2.3, respectively. Meanwhile, the final validation accuracies and the test accuracy for each choice can be seen in 2.1

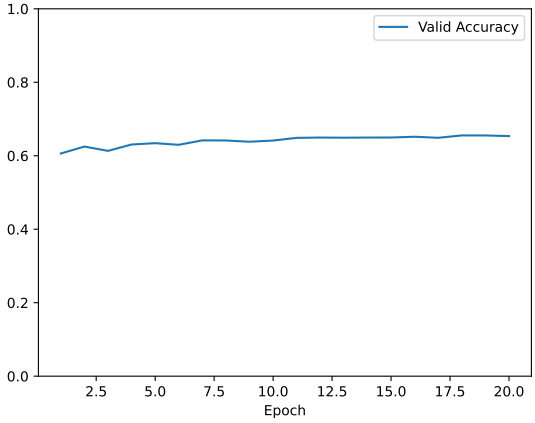


(a) Validation accuracy

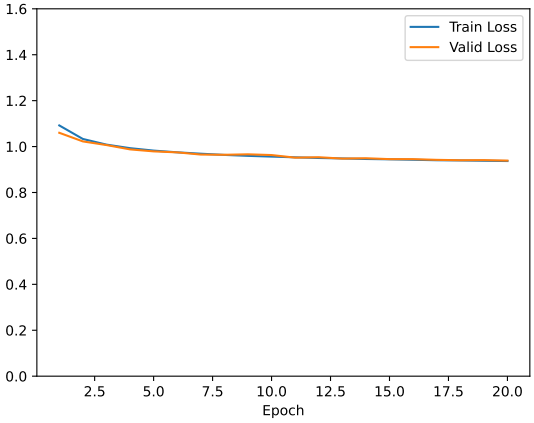


(b) Train and validation loss functions

Figura 2.1: Accuracy and loss training and validation for the logistic regression using learning rate 0.1 with Pytorch

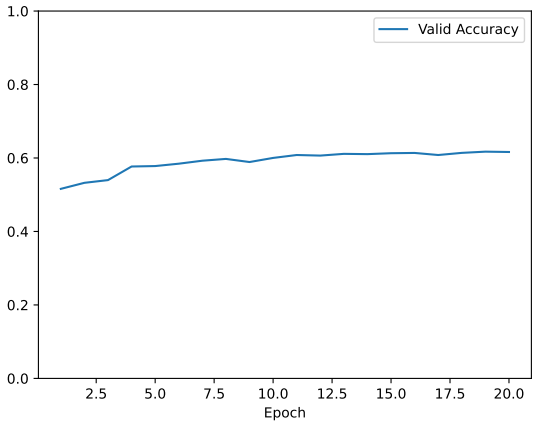


(a) Validation accuracy

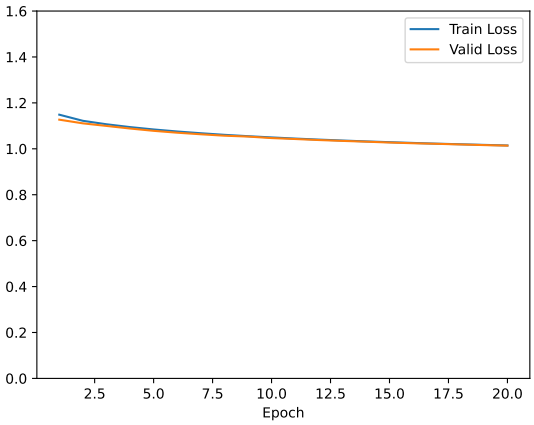


(b) Train and validation loss functions

Figura 2.2: Accuracy and loss training and validation for the logistic regression using learning rate 0.01 with Pytorch



(a) Validation accuracy



(b) Train and validation loss functions

Figura 2.3: Accuracy and loss training and validation for the logistic regression using learning rate 0.001 with Pytorch

Learning rate	Final validation accuracy	Test accuracy
0.1	0.6224	0.5577
0.01	0.6535	0.6200
0.001	0.6163	0.6503

Table 2.1: Results for different accuracy choices

In terms of final validation accuracy, the best model was the one with learning rate 0.01. However, in terms of test accuracy, the best one used learning rate equal to 0.001

2.2 Item 2.a

Below, the implementation of `__init__()` and `forward()` for the `FeedforwardNetwork` class using Pytorch can be seen.

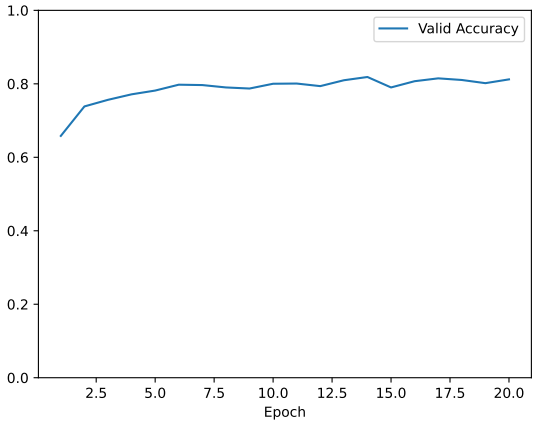
```
class FeedforwardNetwork(nn.Module):
    def __init__(
            self, n_classes, n_features, hidden_size, layers,
            activation_type, dropout, **kwargs):
        """
        n_classes (int)
        n_features (int)
        hidden_size (int)
        layers (int)
        activation_type (str)
        dropout (float): dropout probability
```

```
        """
        super().__init__()

        # Define the activation function
        if activation_type == 'relu':
            activation = nn.ReLU()
        else:
            activation = nn.Tanh()

        # Create a list of modules
        modules = []
        in_features = n_features

        # Add hidden layers
        for _ in range(layers):
            modules.append(nn.Linear(in_features, hidden_size))
            modules.append(activation)
            modules.append(nn.Dropout(dropout))
            in_features = hidden_size

        # Add output layer
        modules.append(nn.Linear(hidden_size, n_classes))

        # Combine all modules into a single sequential model
        self.model = nn.Sequential(*modules)

    def forward(self, x, **kwargs):
        """
        x (batch_size x n_features): a batch of training examples
        """
        # Forward pass through the sequential model
        return self.model(x)
```

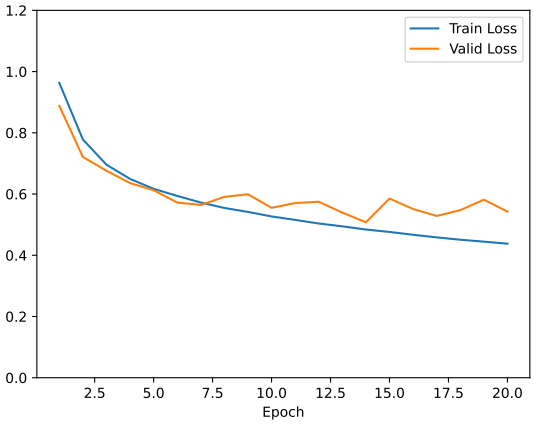
With this implementation, this feedforward neural network was trained on the OCTMNIST dataset with the parameters shown in Table 2.2 and with batch size equal to 16 and to 1024. The training and validations accuracies and the loss function for each batch size can be seen in Figures 2.4 and 2.5, respectively. In addition, the final validation accuracy, the test accuracy and the elapsed time during training for both choices can be seen in Table 2.3

Hyperparameter	Value
Number of Epochs	20
Learning Rate	0.1
Hidden Size	200
Number of Layers	2
Dropout	0.0
Activation	ReLU
L2 Regularization	0.0
Optimizer	SGD

Table 2.2: Default hyperparameters for item a.

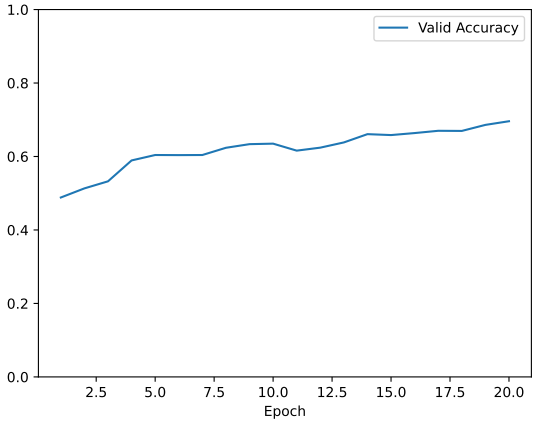


(a) Validation accuracy

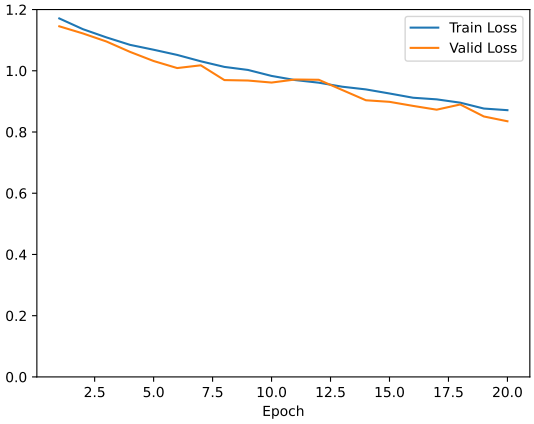


(b) Train and validation loss functions

Figura 2.4: Accuracy and loss training and validation for the MLP using batch size 16 with Pytorch



(a) Validation accuracy



(b) Train and validation loss functions

Figura 2.5: Accuracy and loss training and validation for the MLP using batch size 1024 with Pytorch

Batch size	Final validation accuracy	Test accuracy	Elapsed time (s)
16	0.8121	0.7675	146.43
1024	0.6960	0.7353	36.87

Table 2.3: Results for different batch size choices

Since in both cases the same number of epochs was used, the model with batch size 16 had more iterations and thus took more time to train. It is interesting to note that it shows a significantly higher final validation accuracy, when comparing to the model with batch size 1024. However, their test accuracies are similar. This is because models with larger batch sizes tend to be better at generalizing the training data when predicting unseen examples.

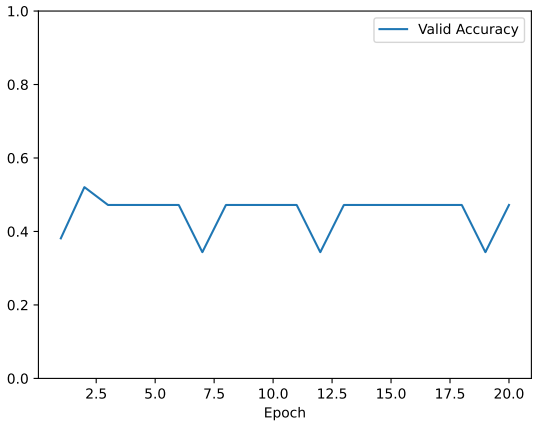
2.3 Item 2.b

Then, the feedforward neural network was trained with the parameters shown in Table 2.4 and with learning rate equal to 1, 0.1, 0.01 and 0.001. In terms of validation accuracy, the worst model had learning rate equal to 1, whereas in the best model it was 0.01.

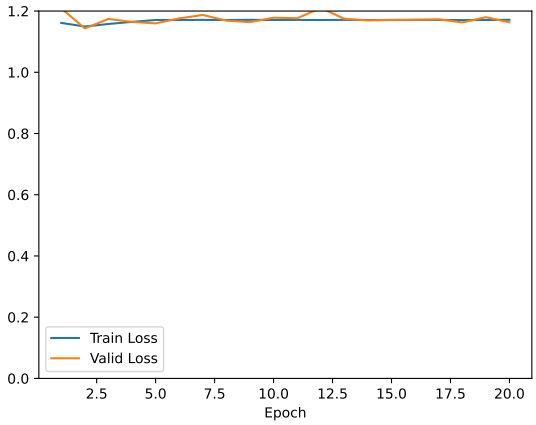
The training and validations accuracies and the loss function for the worst and best model can be seen in Figures 2.6 and 2.7, respectively. In addition, the final validation accuracy, the test accuracy and the elapsed time during training for all models can be seen in Table 2.5

Hyperparameter	**Value**
Number of Epochs	20
Batch Size	16
Hidden Size	200
Number of Layers	2
Dropout	0.0
Activation	ReLU
L2 Regularization	0.0
Optimizer	SGD

Table 2.4: Default hyperparameters for item b.

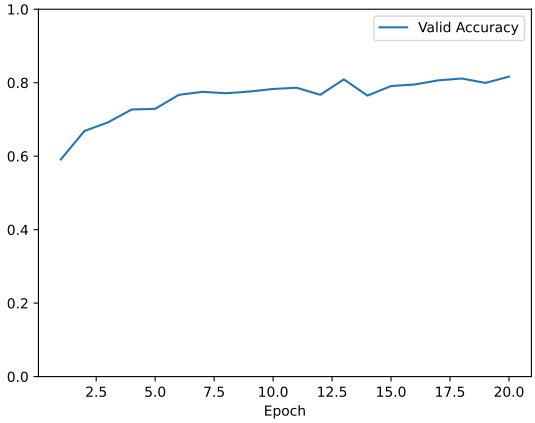


(a) Validation accuracy

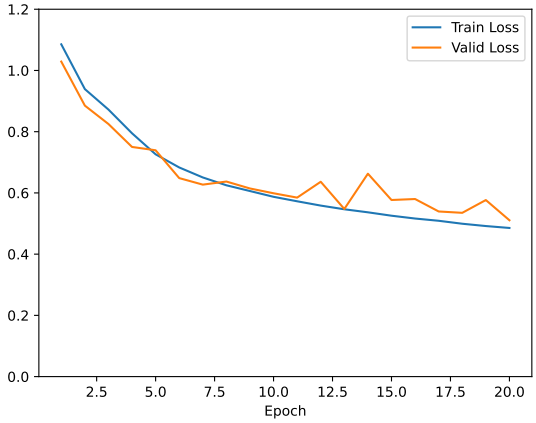


(b) Train and validation loss functions

Figura 2.6: Accuracy and loss training and validation for the MLP learning rate 1.0 with Pytorch



(a) Validation accuracy



(b) Train and validation loss functions

Figura 2.7: Accuracy and loss training and validation for the MLP learning rate 0.01 with Pytorch

Learning rate	Final validation accuracy	Test accuracy	Elapsed time (s)
1	0.4721	0.4726	288.65
0.1	0.8121	0.7675	296.0
0.01	0.8166	0.7637	304.86
0.001	0.6916	0.7108	296.44

Table 2.5: Results for different learning rate choices

With learning rate equal to 1, the model could not reduce its loss function, and the accuracy did not improve. On the contrary, the model with the value equal to 0.01 was able to reduce the loss function and increase the accuracy throughout the epochs. It is interesting to note that the accuracies of the models with learning rate 0.01 and 0.1 wew

very similar. In addition, reducing even more the the hyperparameter cause perfomance to worsen.

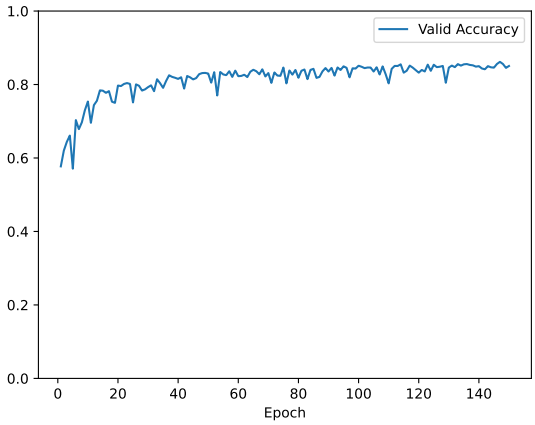
2.4 Item 2.c

To conclude the study, the feedforward neural network was trained with the parameters shown in Table 2.6. Then, two other models were trained: one with the L2 regularization parameter set to 0.0001 and other with the previous hyperparameters, but with dropout probability of 0.2. The aim of these three different tests was to assess overfitting and the impact of these two techniques in avoiding it. In terms of validation accuracy, the worst model was the the one with regularization, while the best one was the one with dropout.

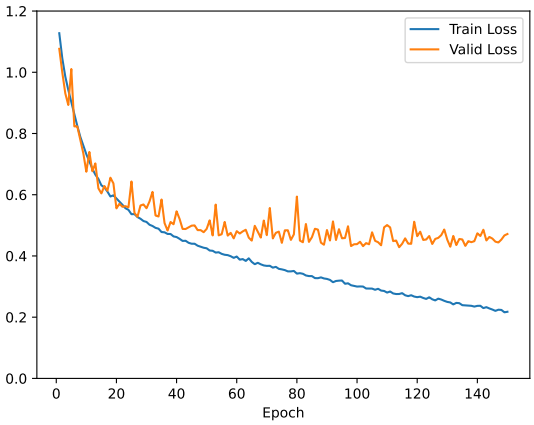
The accuracy and loss function for the default model can be seen in 2.8, and the plots for the worst (with regularization) and best (with dropout) model can be seen in Figures 2.9 and 2.10, respectively. In addition, the final validation accuracy, the test accuracy and the elapsed time during training for all the three models can be seen in Table 2.7

Hyperparameter	**Value**
Number of Epochs	150
Learning Rate	0.1
Batch Size	256
Hidden Size	200
Number of Layers	2
Dropout	0.0
Activation	ReLU
L2 Regularization	0.0
Optimizer	SGD

Table 2.6: Default hyperparameters for item c.

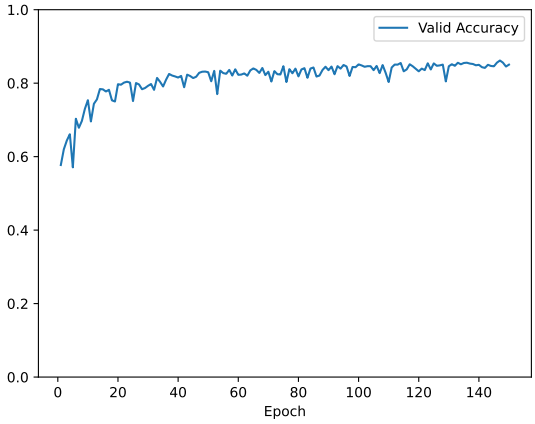


(a) Validation accuracy

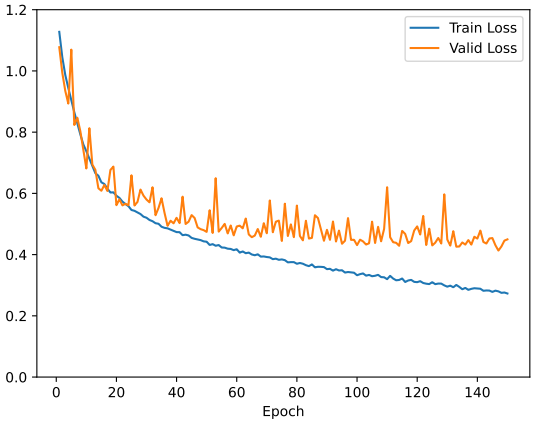


(b) Train and validation loss functions

Figura 2.8: Accuracy and loss training and validation for the default MLP for item c

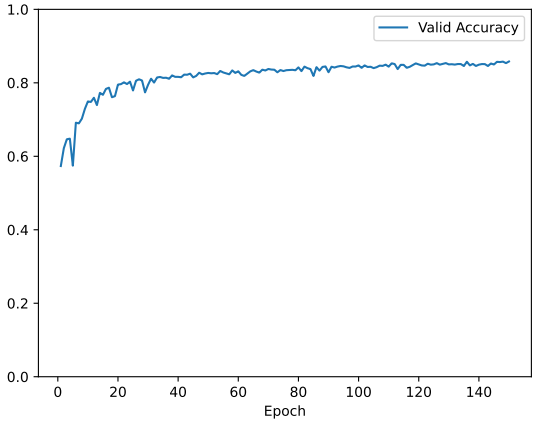


(a) Validation accuracy

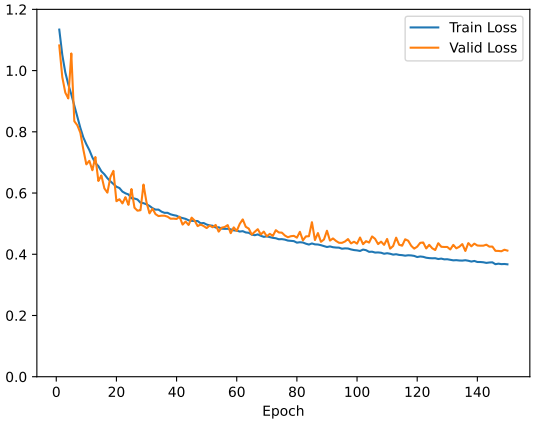


(b) Train and validation loss functions

Figura 2.9: Accuracy and loss training and validation for the MLP with regularization



(a) Validation accuracy



(b) Train and validation loss functions

Figura 2.10: Accuracy and loss training and validation for the MLP with dropout

Model	Final validation accuracy	Test accuracy	Elapsed time (s)
Default	0.8572	0.7561	295.62
With regularization	0.8504	0.7486	366.40
With dropout	0.8581	0.7921	372.57

Table 2.7: Results for different strategies to reduce overfitting

It is possible to infer that the default model was overfit, since the validation loss stops decreasing, while the train loss continues to do so (Figure 2.8). The aplication of the regularization technique was not enough to prevent the overfitting of the model, since the same behaviour can be observed (Figure 2.9). It is also interesting to note that the model with regularization had a worse accuracy than the default model, which was not expected.

When dropout was used, on the other hand, the validation loss continued to decrease together with the training function. And, as a result, its accuracy is higher than the two previous models.

3. Question 3

This exercise will focus in the manual design of a multilayer perceptron that computes a Boolean function of D variables, $f : \{-1, +1\}^D \rightarrow \{-1, +1\}$, defined as:

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

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

3.1 Item 1.a

To show that for some choice of interval $[A, B]$ this function cannot be computed with a single perceptron, one can take the counter-example of $D = 2$ and $[A, B] = [0, 0]$.

In this case, the space of $\mathbf{x} = [x1, x2]$ is the set $\{[-1, -1], [-1, 1], [1, -1], [-1, -1]\}$, and the values that $\sum_{i=1}^{2} x_i$ can assume are $\{-2, 0, +2\}$.

As a result, if $[A, B] = [0, 0]$, for each possible value of $\mathbf{x}$, $f(\mathbf{x})$:

- $f([-1, -1]) = -1$
- $f([-1, 1]) = 0$
- $f([1, -1]) = 0$
- $f([1, 1]) = -1$

And, if these values are plotted in a chart (Figure 3.1), it is possible to note that the points are not separable by a line. As a result, a single perceptron cannot learn this specific function.

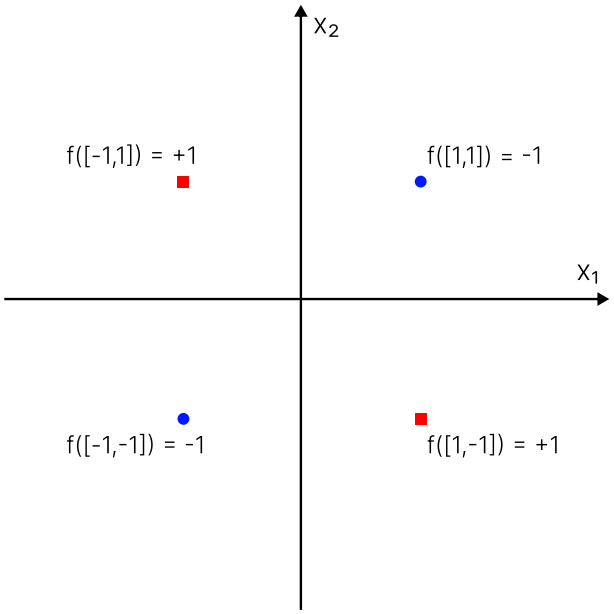


Figura 3.1: Plot of the specified function

3.2 Item 1.b

This item will show that it is possible to implement the previous function with a multilayer perceptron with one hidden layer with two units. An illustration of this network is shown in Figure 3.2, whre the first layer is the input layer with dimension D.

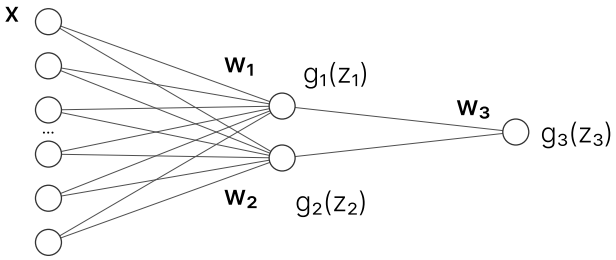


Figura 3.2: Illustration of proposed network

All units of the network use hard threshold activations $g : \mathbb{R} \to \{-1, +1\}$ with

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

To learn the specified function for any D and choice of interval $[A, B]$, one must take into considerations the possible configurations of the function. For this, $\sum_{i=1}^{D} x_i$ will be reffered to as S.

Depending on the choice of $[A, B]$, it separates the values of S in two or three regions. In the case of two regions, the space is linearly separable and can be learned by a simple perceptron. However, if it separates in three regions, as ilustrated in figure 3.3, a MLP will be needed. In this exercise, the second case will be the focus.

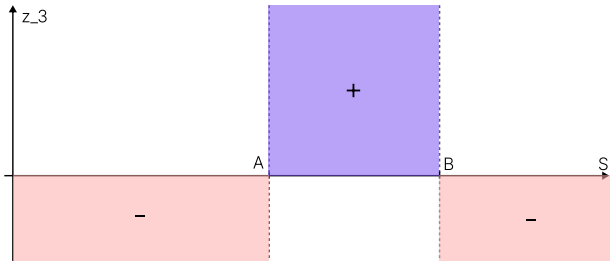


Figura 3.3: Illustration of three regions

Take as an example $D = 4$. The space of S becomes $\{-4, -2, 0, +2, +4\}$. If $[A, B] = [0, 2]$, the space is divided into three regions, since $f(x) = 1$ for $S \in \{-4, -2, +4\}$ and $f(x) = -1$ for $S \in \{0, +2\}$.

With this in mind, the MLP will be design as follows. The first unit in the hidden layer will designed to be activated if $S \geq A$. For this, its weights will be set to 2 for all inputs and its bias to $-(2A - 1)$. With its activation function, it will output 1 if $2S \geq 2A - 1$, that is, $S \geq A - 0.5$. The 0.5 was added to ensure the unit still fires in the case of small perturbations, and the values were multiplied by 2 to turn them into integers, as required.

In a similar way, the second unit will be activated if $S \leq B$. For this, set its weights to -2 for all inputs and its bias to $2B + 1$

Finally, the output neuron must fire if both the hidden units were activated, which means S is bigger or equal to A and smaller or equal to B. For this, set the weight to 2 for both units and the bias to -3.

To show that this strategy works, an example with $D = 5$ can be analyzed. In table 3.1, the values of the hidden inputs are shown for the possible combinations of $\mathbf{x}$ that have different values of S (permutations were omited). And in table 3.2, the output value is shown, for each case from the previous table. It is then possible to see that the proposed MLP correctly maps a non-trivial example of the function.

Input		Unit 1: is S>=A?		Unit 1: is S<=B?	
[x1, x2, x3, x4, x5]	S	$2S-(2A-1)$	$H1$	$-2S+(2B+1)$	H2
[-1, -1, -1, -1, -1]	-5	-11	-1	17	1
[-1, -1, -1, -1, 1]	-4	-9	-1	15	1
[-1, -1, -1, 1, 1]	-1	-3	-1	9	1
[-1, -1, 1, 1, 1]	1	1	1	5	1
[-1, 1, 1, 1, 1]	3	5	1	1	1
[1, 1, 1, 1, 1]	5	9	1	-3	-1

Table 3.1: Example of hidden units for D = 5

Hidden units		Output: is S>=A and S<=B?	
$H1$	H2	$2H1+2H2-3$	$f(x)$
-1	1	-3	-1
-1	1	-3	-1
-1	1	-3	-1
1	1	1	1
1	1	1	1
1	-1	-3	-1

Table 3.2: Example of output for D = 5

3.3 Item 1.c

In this exercise, the previous $f(x)$ will be implemented in a MLP ilustrated in Figure 3.4 with one hidden layer and two hidden units that use the ReLU activation function.

$$g(z) = \text{ReLU}(z) = \begin{cases} z & \text{if } z \geq 0 \\ 0 & \text{otherwise} \end{cases}$$

And the output layer still uses the hard threshold activation function:

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

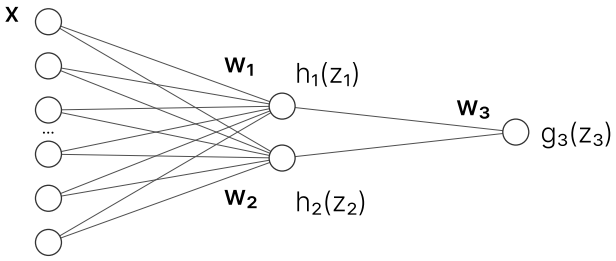


Figura 3.4: Illustration of proposed network

Again, the focus of this solution will be when the choice of $[A, B]$ divides the problem in three regions, as shown in Figure 3.3.

To solve this problem, the aim will be find the parameters of z_3 that make it positive for $S \in [A, B]$, where

$$z_3 = w_{31}h_1(z_1) + w_{32}h_2(z_2) + b_3$$

and $S = \sum_{i=1}^{D} x_i$; $h_1(z_1)$ and $h_2(z_2)$ are the outputs of the hidden units and b_3 is the bias of the output unit.

One possible approach is to try to construct the function shown in Figure 3.5 only using the ReLU function, that is ilustrated in 3.6

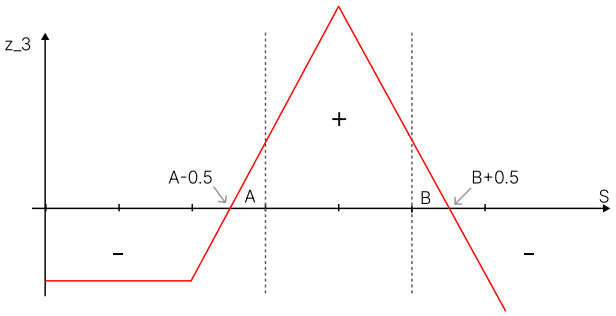


Figura 3.5: Illustration of goal function

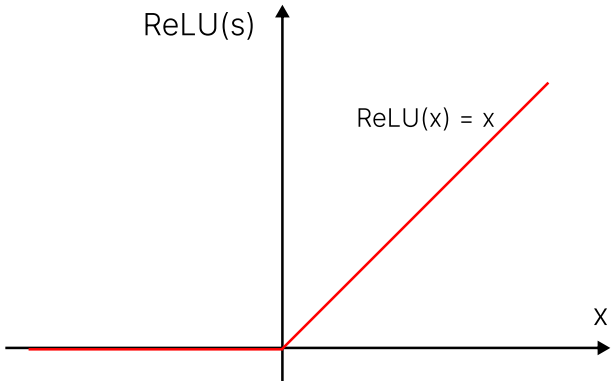


Figura 3.6: Illustration of the ReLU function

With this rationale, it is possible to sum the following functions, illustrated in figure 3.7, to construct the complete desired z_3

- $b_3 = -1$ (dotted blue)
- $z_3 = 2 \cdot S$ for $S >= A - 1$ (solid blue)
- $z_3 = 4 \cdot S$ for $S >= \frac{(A+B)}{2}$ (dash-dot blue)

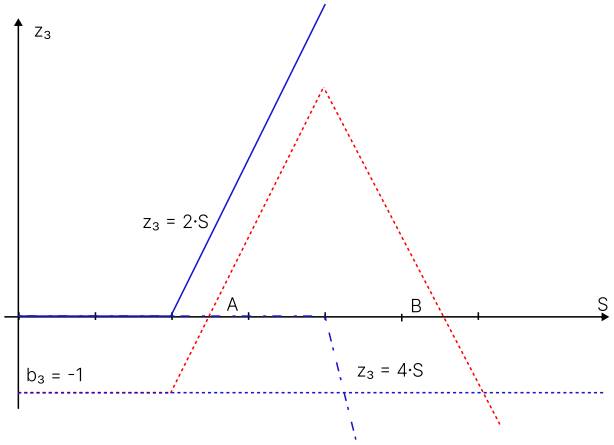


Figura 3.7: Illustration componentes of the function

And, to write these functions using the ReLU activations, one possible definition of the parameters is:

Parameter	Value
$\mathbf{w}_1$	$\mathbf{1}$
b_1	$-(A-1)$
$\mathbf{w}_2$	$\mathbf{1}$
b_2	$-(A+B)/2$
w_{31}	2
w_{32}	-4
b_3	-1

Table 3.3: Parameters choice

To show that this strategy works, an example with $D = 5$ can be analyzed. In table 3.4, the values of the hidden inputs are shown for the possible combinations of $\mathbf{x}$ that have different values of S (permutations were omited). And in table 3.5, the output value is shown, for each case from the previous table. It is then possible to see that the proposed MLP correctly maps a non-trivial example of the function by using the ReLU as the activation function in the hidden layer.

Input $[x1, x2, x3, x4, x5]$	S	H1: $S-(A-1)$ if $S >= (A-1)$	H2: $S-(A+B)/2$ if $S >= (A+B)/2$
[-1, -1, -1, -1, -1]	-5	0	0
[-1, -1, -1, -1, 1]	-3	0	0
[-1, -1, -1, 1, 1]	-1	0	0
[-1, -1, 1, 1, 1]	1	1	0
[-1, 1, 1, 1, 1]	3	3	1
[1, 1, 1, 1, 1]	5	5	3

Table 3.4: Example of hidden layers for D = 5

Hidden layer		z_3	g_3
$H1$	$H2$	$2H1 - 4H2 - 1$	$sign(z_3)$
0	0	-1	-1
0	0	-1	-1
0	0	-1	-1
1	0	1	1
3	1	1	1
5	3	-3	-1

Table 3.5: Example of output for D = 5