TÉCNICO LISBOA

DEEP STRUCTURED LEARNING

HOMEWORK 1

André Godinho Number: 84006

QUESTION 1

1.

The perceptron is a linear classifier. Therefore, the data must be linearly separable for it to classify correctly.

By analyzing Fig.1 we can observe that the referred data is not linearly separable. Moreover, it's not possible to draw a linear hyperplane between data points.

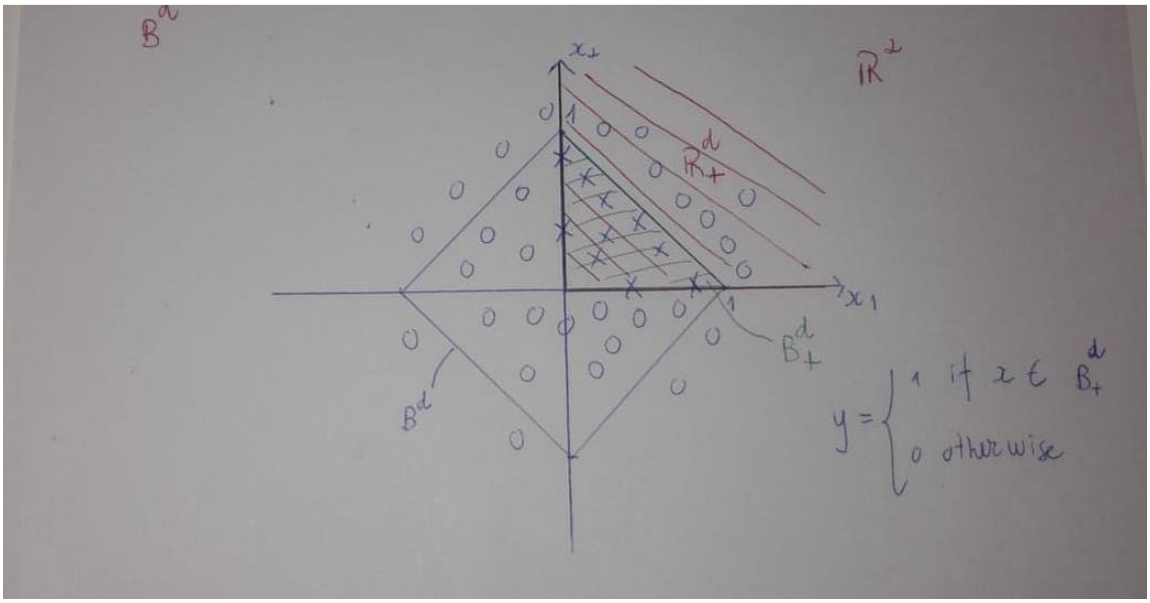

FIG.1 - QUESTION 1.1. REGIONS

In this drawing, the crosses represent the $y = 1$ data points, which belong to B_+^d and the circles represent the $y = 0$ data points and don't belong to the referred region. As we can observe, it is not possible to draw a line to separate data.

2.

We are using Heaviside step function ($H(z)$) to map our datapoints as category 0 or 1.

$$H(z) = \begin{cases} 1, z \geq 0 \\ 0, z < 0 \end{cases}$$

For this exercise we can see that $z = w^T \cdot x + b = \sum_{i=1}^{2}(w_i \cdot x_i) + b$. So, to return the following:

$$H(z) = \begin{cases} 1, x_1 \geq 0 \\ 0, otherwise \end{cases}$$

We need $z = w^T \cdot x + b = x1 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 + b = x1 \Leftrightarrow w = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, b = 0$

3.

We need $z = w^T \cdot x + b = -x1 - x2 + 1 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 + b = -x_1 - x_2 + 1 \Leftrightarrow$

$$\Leftrightarrow w = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, b = 1$$

4.

To perform the task in hands we will define one hidden layer with 3 perceptron. Therefore, we need to define the vector $\boldsymbol{w} \in \mathrm{R}^2$ and bias $\boldsymbol{b}$ for each perceptron.

Perceptron 1

Pretended constraint: $y_1 = \begin{cases} 1, x_1 \geq 0 \\ 0, otherwise \end{cases}$

We need $z = w^T \cdot x + b = x_1 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 + b = x_1 \Leftrightarrow w = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, b = 0.$

Perceptron 2

Pretended constraint: $y_2 = \begin{cases} 1, x_2 \geq 0 \\ 0, otherwise \end{cases}$

We need $z = w^T \cdot x + b = x_2 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 + b = x_2 \Leftrightarrow w = \begin{bmatrix} 0 \\ 1 \end{bmatrix}, b = 0.$

Perceptron 3

Pretended constraint: $y_3 = \begin{cases} 1, x_1 + x_2 \leq 1 \\ 0, otherwise \end{cases}$

We need $z = w^T \cdot x + b = -x_1 - x_2 + 1 \Leftrightarrow w_1 \cdot x_1 + w_2 \cdot x_2 + b = -x_1 - x_2 + 1 \Leftrightarrow$

$$\Leftrightarrow w = \begin{bmatrix} -1 \\ -1 \end{bmatrix}, b = 1$$

Now that we have all perceptron for the needed constraints, we must design a connection between the hidden layer with the 3 perceptron and output so that the classifier only returns 1 if all constraints are met.

We can do it by applying the following:

Pretended constraint: $y = \begin{cases} 1, y_1 + y_2 + y_3 \geq 3 \\ 0, otherwise \end{cases}$

We need,

$$z = w^T \cdot x + b = y_1 + y_2 + y_3 - 3 \Leftrightarrow w_1 \cdot y_1 + w_2 \cdot y_2 + w_3 \cdot y_3 + b = y_1 + y_2 + y_3 - 3$$

$$\Leftrightarrow w = \begin{bmatrix} 1 \\ 1 \\ 1 \end{bmatrix}, b = -3$$

Thus, output will return 1 if all the entries are nonnegative and together, they sum to less than or equal to 1. In other words, if it the datapoint $\mathbf{x} \in \mathrm{R}^2$ belongs to $\boldsymbol{B}_+^d$.

5.

You always need one constraint that consists on all the entries sum to less than or equal to 1. In other words,

$$y = \begin{cases} 1, \sum_{i=1}^{d} x_i \leq 1 \\ 0, otherwise \end{cases}$$

This constraint is associated to one perceptron.

Besides this one, you need as much constraints as size ***d*** (space dimension) to satisfy that all entries are nonnegative.

$$y = \begin{cases} 1, x_i \geq 0 \; \forall \, i: 1, \dots, \boldsymbol{d} \\ 0, otherwise \end{cases}$$

Each one of these constraints are associated to one perceptron.

Thus, you need to increase the number of hidden units as you increase ***d*** by the same rate. In $\mathbf{R^2}$ you need 3 perceptron, in $\mathbf{R^3}$ you need 4 perceptron, in $\mathbf{R^4}$ you need 5 perceptron, etc. In other words,

$$\text{Number of perceptron} = \text{space dimension} + 1$$

QUESTION 2

1.(A).

Using binary pixel as feature representation allows figure-ground separation. Moreover, it's possible to create a two-valued binary image such that we classify into object and background pixels. By classifying pixels that correspond to the foreground above a threshold as "1" and as background if they are below this threshold.

1.(B).

```
class Perceptron(LinearModel):
    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
        """
        # Question 2.1 b

        score = np.dot(self.W, x_i.T)
        predicted_label = score.argmax(axis = 0)
```

```
        if predicted_label != y_i:
            self.W[predicted_label] = self.W[predicted_label] - x_i
            self.W[y_i] = self.W[y_i] + x_i
        return
```

It was implemented the *update_weight* method of the **Perceptron** class. The model was trained with 20 epochs on the training set and we also can observe the accuracies as a function of the epoch number in Fig.2.

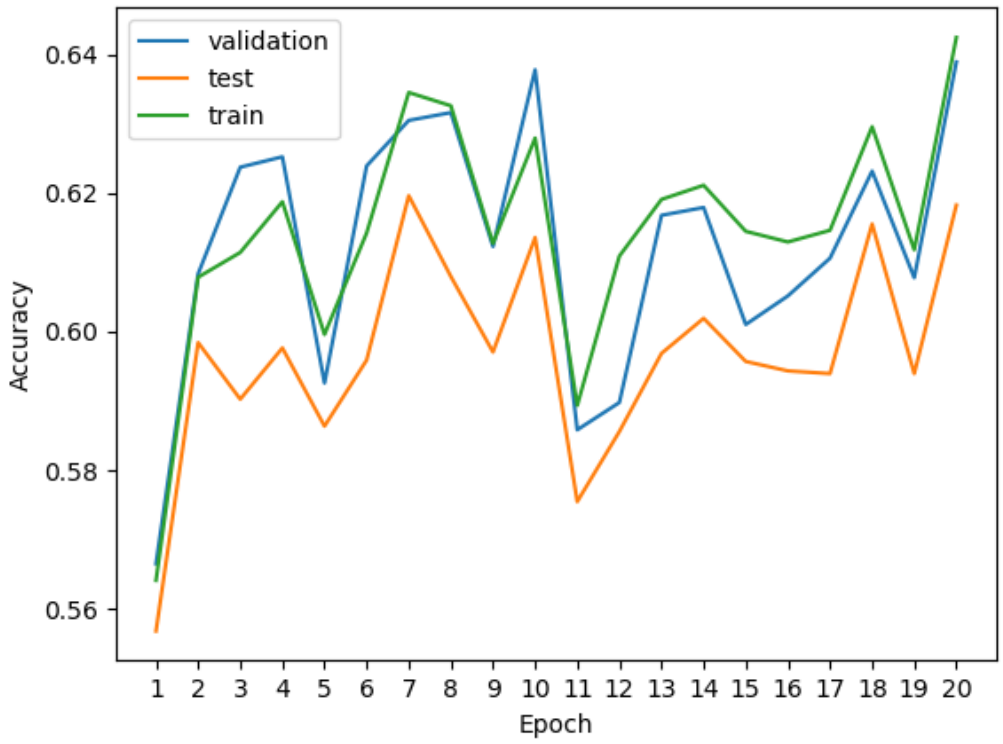

FIG.2 - ACCURACY OF THE PERCEPTRON IN TRAIN, VALIDATION AND TEST SET WITH 20 EPOCHS

TABLE 1 – ACCURACY RESULTS WITH 20 EPOCHS

	Train set	Validation set	Test set
Accuracy	64,2%	63,9%	61,8%

As expected, we can observe that the accuracy was higher in the train set comparing to the validation and test set and it was the lowest on the test set.

The accuracy oscillated throughout the 20 epochs, however, there was an increase of accuracy comparing the first 2 epochs and the last one because the model. On the first epoch the accuracy was almost 10% lower than the final results in all data sets (underfitting).

To check if the accuracy increases with the number of epochs it makes sense to analyze with a higher number of iterations.

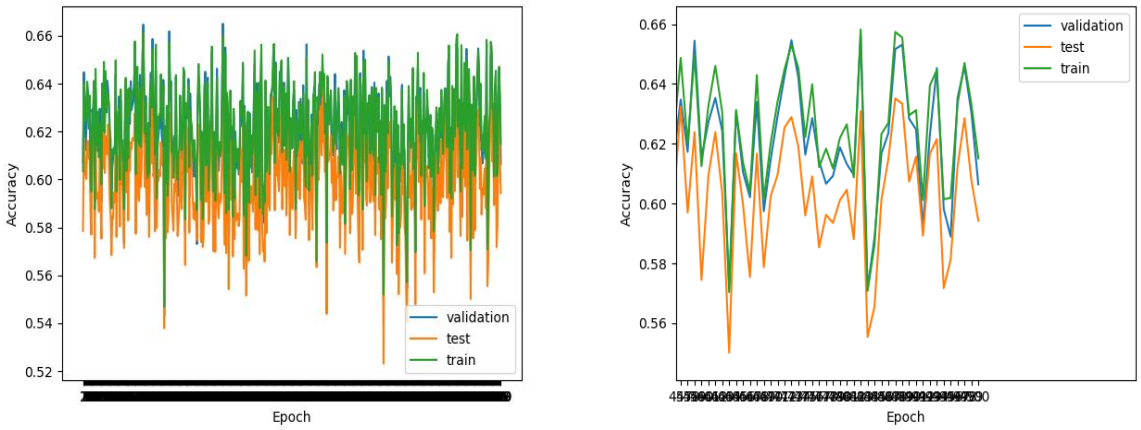

FIG.3 - ACCURACY OF THE TRAIN, VALIDATION AND TEST SET WITH 500 EPOCHS

TABEL 2 – ACCURACY RESULTS WITH 500 EPOCHS

	Train set	Validation set	Test set
Accuracy	61,5%	60,6%	59,4%

As we can observe, the oscillated tendency keeps happening throughout all the number of epochs.

2.(A).

```
def custom_features(x_i):
    """
    x_i (n_features, )
    returns (???, ): It's up to you to define an interesting feature
        representation. One idea: pairwise pixel features (see the handou
t).
    """
    # Q2.2 a
    x_i = x_i.reshape((1,128)) # re-shape as an array of [1x128]
    matrix = x_i.T*x_i # [128x128]

    # calculate upper triangular matrix (w/ diagonal)
    x_i = matrix[np.triu_indices(len(matrix),k=0)]
    print(x_i.shape)
    return x_i # (8256,)
```

For this task, we take advantage of the matrix multiplication of a feature vector by itself (creating a [128x128] matrix). After doing so, we extract the pairwise elements of this matrix by extracting the upper triangular matrix (with the diagonal). We could have also

done the lower triangular matrix (with the diagonal) since the matrix is symmetric. Thus, we obtain a new feature with 8256 features.

Using pixel pairwise combination leads to better results because samples of the same class with pixels shifted a bit to any direction will not have a drastic impact in the training of the model because we don't have a 16x8 feature binary pixel representation of the image. Instead, we now have combinations of pixels which work better for these shifts. Besides, it can grasp combination of close bits much better since those adjacent bits will be featured as 1.

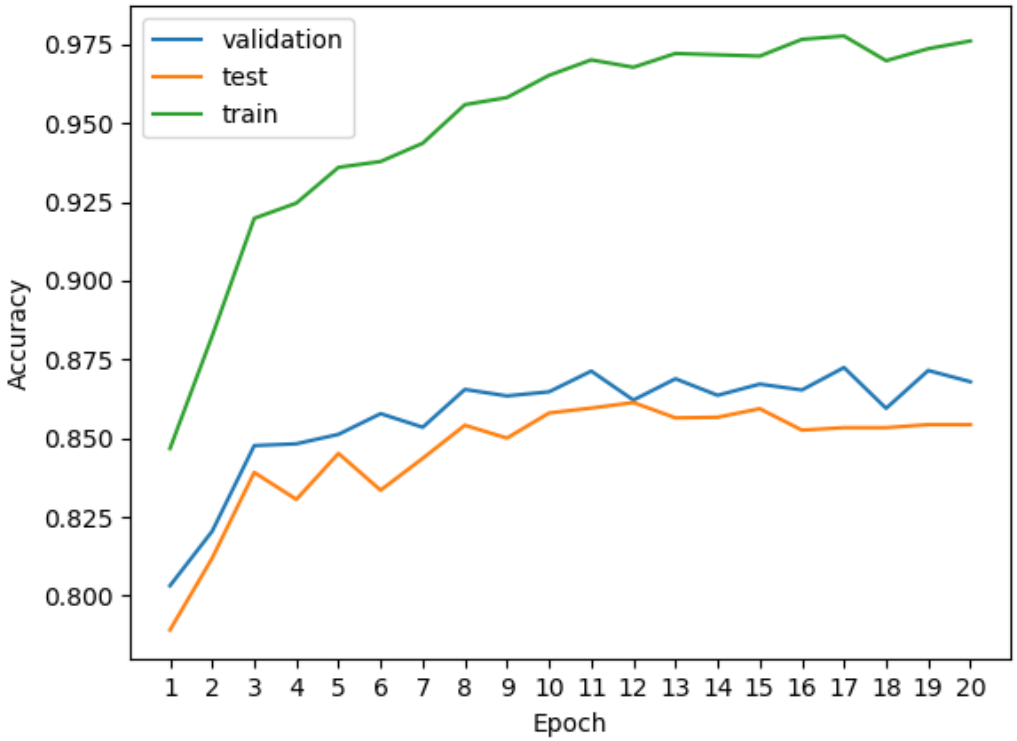

FIG.4 – ACCURACY OF THE TRAIN, VALIDATION AND TEST SET WITH PERCEPTRON, PIXEL PAIRWISE COMBINATION AND 20 EPOCHS

TABLE 3 – ACCURACY RESULTS WITH PERCEPTRON, PIXEL PAIRWISE COMBINATION (PPC) AND 20 EPOCHS

	Train set	Validation set	Test set
Accuracy w/ ppc	97.6%	86,8%	85,4%
Accuracy w/o ppc	64,2%	63,9%	61,8%
Difference	33,4%	22,9%	23,6%

The results are very clear. By **using pixel pairwise combination** in pre-processing the **accuracy** was **33,4% higher in training, 22,9% in validation and 23,6% in test**. We must be careful with overfitting using pixel pairwise combination because the error in train set tends to zero with the number of epochs, the model starts to learn details in data instead of grasping the pattern. To tackle this issue, a good strategy is applying *Early Stopping* in validation.

2.(B).

```
def update_weight(self, x_i, y_i, learning_rate=0.001, l2_penalty=0.0):
        """
        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 plot
s
        l2_penalty (float): BONUS
        """
        # Question 2.2 b
        score = np.dot(self.W, x_i.T)
        predicted_label = score.argmax(axis = 0)

        if predicted_label != y_i:
            exp = np.exp([score])
            zx = np.sum(exp)

            expectation = (exp/zx).T*(x_i.reshape((1,len(x_i))))
            if l2_penalty != 0:
                self.W = self.W - learning_rate*(expectation + l2_penalty
*self.W)
            else:
                self.W = self.W - learning_rate*expectation
            self.W[y_i] = self.W[y_i] + learning_rate*x_i
        return
```

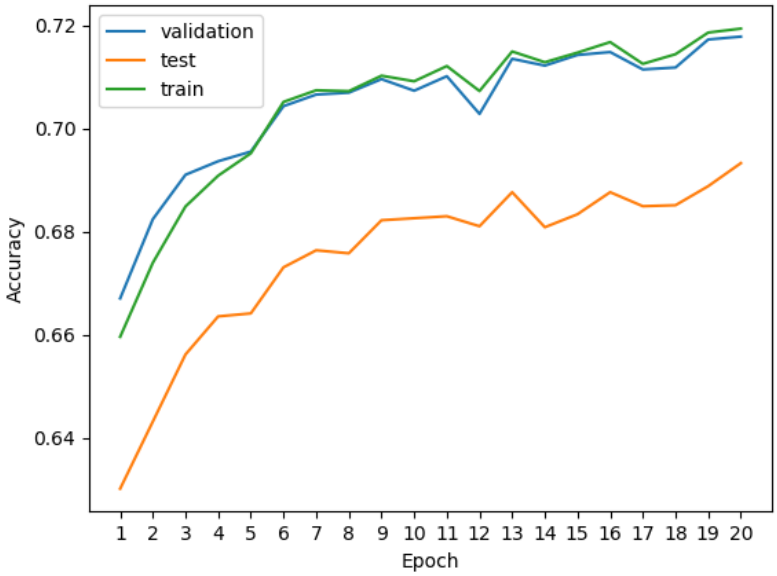

FIG.5 - ACCURACY OF THE TRAIN, VALIDATION AND TEST SET WITH LOGISTIC REGRESSION AND 20 EPOCHS

TABLE 4 – ACCURACY RESULTS IN TRAIN, VALIDATION AND TEST SET WITH LOGISTIC REGRESSION AND 20 EPOCHS

	Train set	Validation set	Test set
Accuracy	71,9%	71,8%	69,3%

Comparing the results of the fig.2, fig.5 and table 2 and table 4 we can observe that the accuracy of the logistic regressions was ~ 10% higher than the perceptron. Besides that, the accuracy did not oscillate as much as the perceptron did and it increased more smoothly throughout the epochs. This is explained because it was used the expectation on the stochastic gradient instead of the prediction when updating weights and the given learning rate.

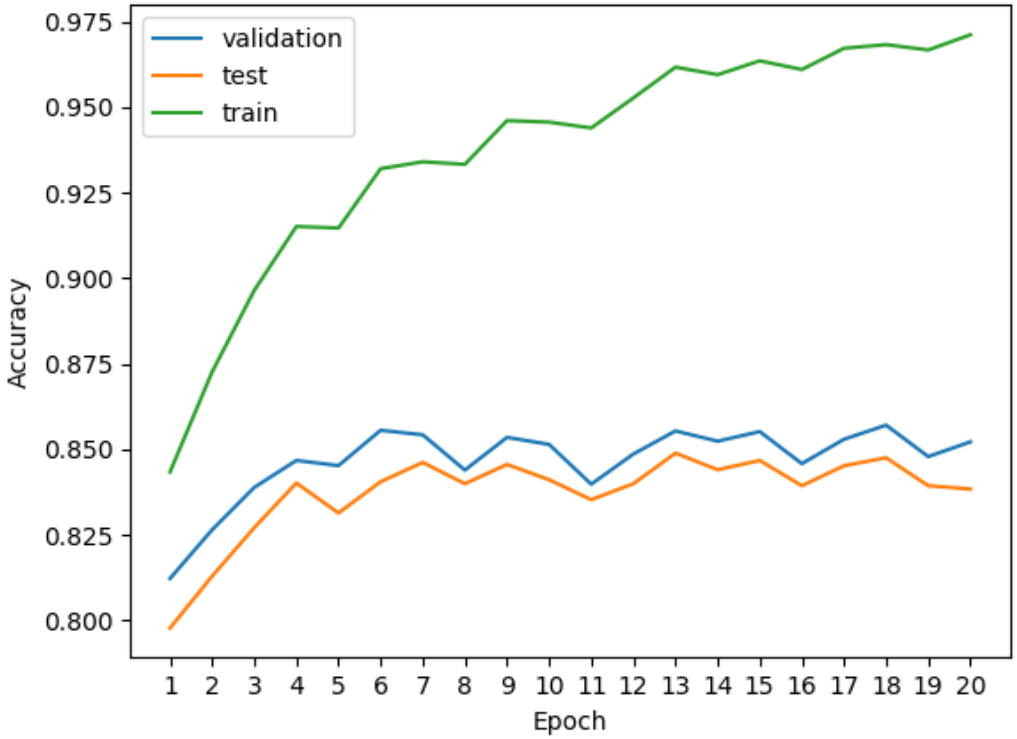

FIG.6 - ACCURACY OF THE TRAIN, VALIDATION AND TEST SET WITH LOGISTIC REGRESSION, PIXEL PAIRWISE COMBINATION AND 20 EPOCHS

TABLE 5 - ACCURACY RESULTS WITH LOGISTIC REGRESSION, PIXEL PAIRWISE COMBINATION (PPC) AND 20 EPOCHS

	Train set	Validation set	Test set
Accuracy w/ ppc	97.1%	85,2%	83,8%
Accuracy w/o ppc	71,9%	71,8%	69,3%
Difference	25,2%	13,4%	14,5%

By **using pixel pairwise combination** in pre-processing the **accuracy** was **25,2% higher in training, 13,4% in validation and 14,5% in test.** As said before the model tend to overfit the data with this number of epochs as we can see by the difference between the accuracy in the validation and train set.

TABLE 6 – COMPARING THE RESULTS BETWEEN PERCEPTRON AND LOGISTIC REGRESSION WITH PIXEL PAIRWISE COMBINATION AND 20 EPOCHS

	Train set	Validation set	Test set
Accuracy w/ perceptron and ppc	97.6%	86,8%	85,4%
Accuracy w/ Log. Regression and ppc	97.1%	85,2%	83,8%
Difference	0.5%	+1.6%	1.6%

The winner was the perceptron because it had a slightly higher accuracy in validation set (+1.6%). We conclude by saying that the pixel pairwise combination had a clear impact on improving the accuracy of both classifiers.

2.(c).

```
def update_weight(self, x_i, y_i, learning_rate=0.001, l2_penalty=0.0):
        """
        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 plot
s
        l2_penalty (float): BONUS
        """
        # Question 2.2 b
        score = np.dot(self.W, x_i.T)
        predicted_label = score.argmax(axis = 0)

        if predicted_label != y_i:
            exp = np.exp([score])
            zx = np.sum(exp)

            expectation = (exp/zx).T*(x_i.reshape((1,len(x_i))))
            if l2_penalty != 0:
                self.W = self.W - learning_rate*(expectation + l2_penalty
*self.W)
            else:
                self.W = self.W - learning_rate*expectation
            self.W[y_i] = self.W[y_i] + learning_rate*x_i
        return
```

TABLE 7 – VARIATION OF ACCURACY WITH LOGSTIC REGRESSION WITH THE L2 REGULARIZER

	Train set	Validation set	Test set
L2_penalty = 0.0001	71,5%	71.4%	68.8%
L2_penalty = 0.001	71,8%	71,5%	69.0%
L2_penalty = 0.01	70.0%	70.1%	67.3%
L2_penalty = 0.1	62.6%	63%	61%
L2_penalty = 1	53.7%	54.1%	53.1%

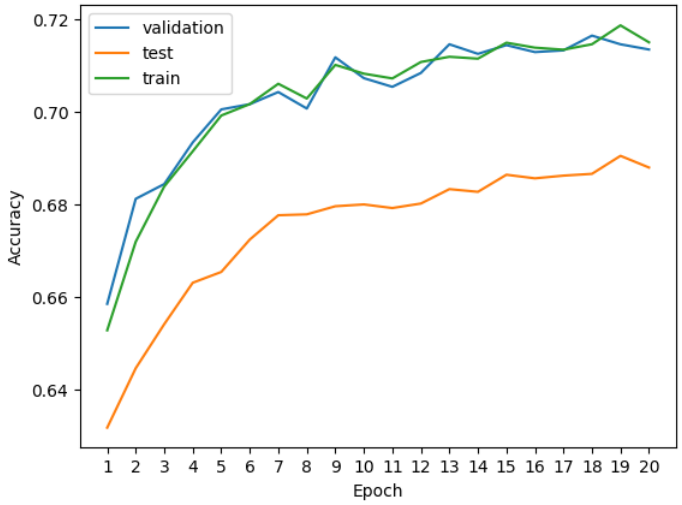

FIG.7 – ACCURACY WITH LOGISTIC REGRESSION AND L2_REGULARIZER = 0.0001

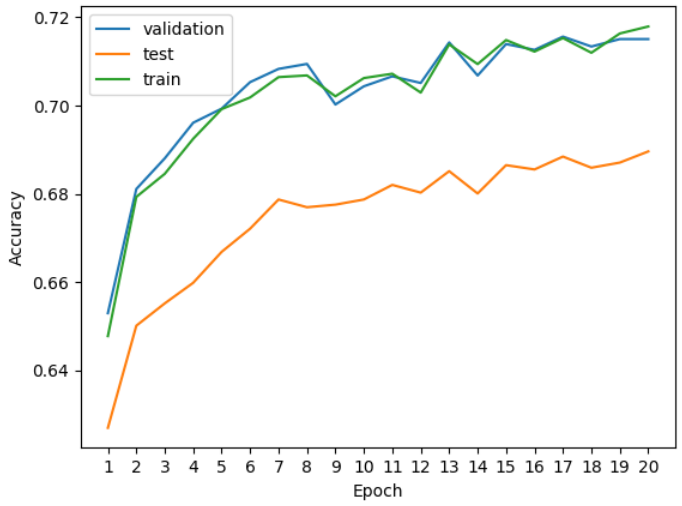

FIG.8 - ACCURACY WITH LOGISTIC REGRESSION AND L2_REGULARIZER = 0.001

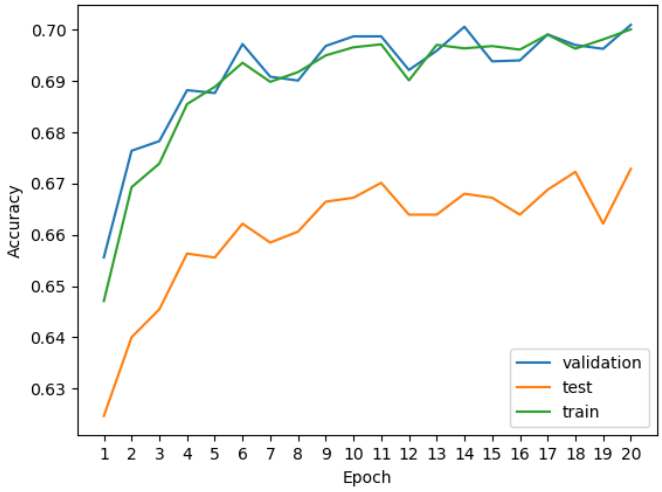

FIG.9 - ACCURACY WITH LOGISTIC REGRESSION AND L2_REGULARIZER = 0.01

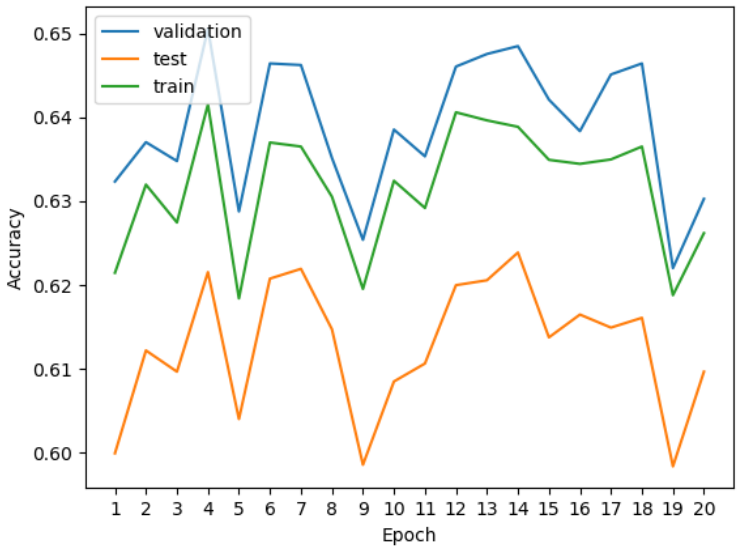

FIG.10 - ACCURACY WITH LOGISTIC REGRESSION AND L2_REGULARIZER = 0.1

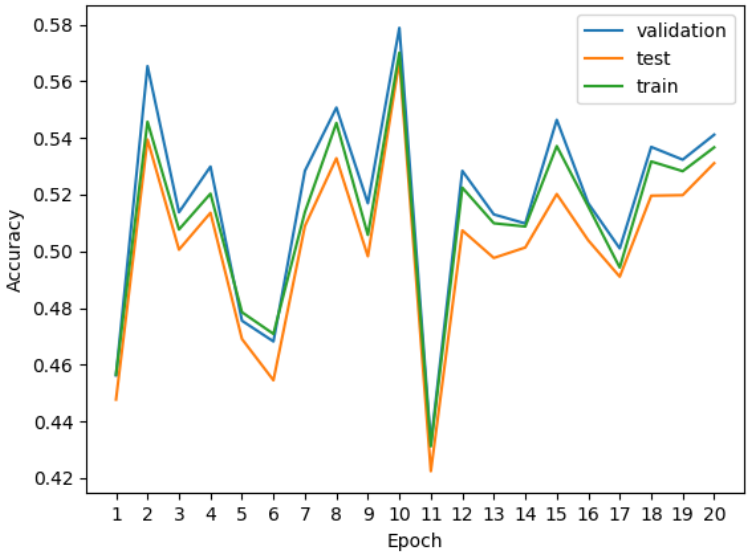

FIG.11 - ACCURACY WITH LOGISTIC REGRESSION AND L2_REGULARIZER = 0.0001

According to table 8 we can observe that the accuracy decreases with the increase of l2_penalty. This happens because with its increase, the l_2 regularization has a higher and higher weight in the optimization function (sum of loss function and the regularization) which leads to the loss function having a smaller contribution for the optimization function. Thus, it gets less minimized and the accuracy tends to decrease.

QUESTION 3

1.

Multi-layer perceptron can learn internal representations and avoid manual feature engineering because the information on the dataset might be enough to represent the classification scenario, so that the MLP won't miss out on the features and learn correctly how to classify the input.

In image recognition for example, the MLP recognizes low-level features such as textures and edges in the picture and then aggregates this information into mid-level and high-level features, such as patterns and objects, with the features becoming further refined as we go on the model's depth.

2.

```
class MLP(object):
    # Q3. This MLP skeleton code allows the MLP to be used in place of th
e
    # linear models with no changes to the training loop or evaluation co
de
    # in main().
    def __init__(self, n_classes, n_features, hidden_size):
        # Initialize an MLP with a single hidden layer.
        # softmax layer must have the same number of nodes of the output
layer
        # # hidden units = # classes
        self.W1 = np.random.rand(n_features, n_classes)
        self.b1 = np.zeros((1, n_classes))
        self.W2 = np.random.rand(n_classes,n_classes)
        self.b2 = np.zeros((1,n_classes))
```

First of all, a few words regarding the instantiation of the MLP class. In this exercise the objective is to implement a multi-layer perceptron with a single hidden layer. Therefore, there will be two matrices of weights and two 1D array of biases. The first matrix and bias will be applied to the input before the activation function and the second matrix and bias will be applied to the output of the activation function.

```
def linear_activation(self,x):
        return x

    def softmax(self,x):
        exps = np.exp(x - np.max(x, axis=1, keepdims=True))
        return exps/np.sum(exps, axis=1, keepdims=True)
```

In what concerns activation functions, we are going to use linear activation in the hidden layer and SoftMax for the output layer. It's important to refer that on the first trial it was used sigmoid as the activation function, but it had poor performance because the

manipulated values in the hidden layer became rather small within few epochs, which lead to poor impact on the matrices of weights.

```
def forward_propagation(self, X):
        z1 = np.dot(X,self.W1) + self.b1 # [n_samples x 26]
        self.a1 = self.linear_activation(z1) # [n_samples x 26]
        z2 = np.dot(self.a1, self.W2) + self.b2 # [n_samples x 26]
        output = self.softmax(z2) # [n_samples x 26]
        return output
```

The forward propagation method is invoked during training and during prediction. The input consists in the whole dataset inserted (train, validation or test set). The output obtained will be a matrix with dimensions *n_samples* x 26 (one-hot format), which will be tested for accuracy and used for backpropagation in train.

Also, we have stored the values after the activation function in the attribute *self.a1* to enable quicker access to these values in backpropagation.

```
def predict(self, input):
        # Compute the forward pass of the network. At prediction time, th
ere is
        # no need to save the values of hidden nodes, whereas this is req
uired
        # at training time.
        output = self.forward_propagation(input)
        output = np.argmax(output,axis = 1)
        return output
```

Since the vector *y* programmed is an 1D array we need to decode the one hot format for 1D as well to calculate the accuracy. For backpropagation this will not happen.

```
def linearact_derivative(self,x):
        return 1

   def softmax_derivative(self, output, y):
        n_samples = y.shape[0]
        y_onehot = np.zeros((len(y),output.shape[1]))
        y_onehot[np.arange(n_samples),y] = 1
        res = output - y_onehot
        return res/n_samples

    def back_propagation(self, X,y,output, learning_rate):
        dL_dz2 = self.softmax_derivative(output,y) # [n_samples x 26]
        dz2_dw2 = self.a1 # [n_samples x 26]
        dL_dw2 = np.dot(dz2_dw2.T,dL_dz2) # [26x26]

        dL_da1 = np.dot(dL_dz2,self.W2.T) # [n_samples x 26]
```

```
        da1_dz1 = self.linearact_derivative(self.a1) # [n_samples x 26]
        dL_dz1 = dL_da1 * da1_dz1 # [n_samples x 26]
        dL_dw1 = np.dot(X.T, dL_dz1) # [128 x 26]

        self.W2 -= learning_rate * dL_dw2
        self.b2 -= learning_rate * np.sum(dL_dz2, axis=0)
        self.W1 -= learning_rate * dL_dw1
        self.b1 -= learning_rate * np.sum(dL_dz1, axis=0)

    def train_epoch(self, X, y, learning_rate = 0.5):
        pred = self.forward_propagation(X)
        self.back_propagation(X, y, pred, learning_rate)
```

To train the MLP, we calculate the output of the model and use it for backpropagation. By applying the chain rule, it was possible to tune the weights for the desired task.

It is important to refer that we **tuned the learning rate for some trials** and obtained decent results with the value $\boldsymbol{learning\ rate = 0,5}$.

Results

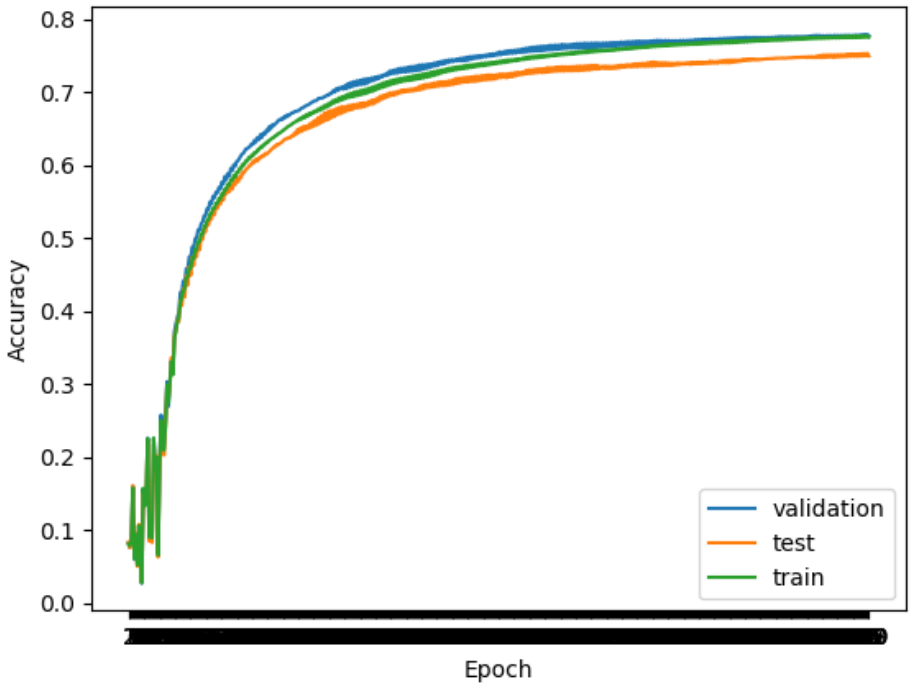

FIG.12 – ACCURACY OF TRAIN, VALIDATION AND TEST SET FOR THE MULTI-LAYER PERCEPTRON WITH 500 EPOCHS

TABLE 8 - ACCURACY RESULTS IN TRAIN, VALIDATION AND TEST SET WITH MULTI-LAYER PERCEPTRON AND 500 EPOCHS

	Train set	Validation set	Test set
Accuracy	77,5%	77,7%	74,9%

This model takes a bit more time to start converging because it needs to learn the weights that minimize the negative log-likelihood cost function.

The results of the accuracy of the train and validation set were very similar which around (77%) is good and the test set was close to 75%.

3.

This question is not well implemented, however I will still the code on the paper. The code file name is hw1q3.py.

```
    def __init__(self, n_classes, n_features, hidden_size):
        # Initialize an MLP with a single hidden layer.
        # softmax layer must have the same number of nodes of the output
layer
        # # hidden units = # classes
        self.hidden_size = hidden_size
        for i in range(self.hidden_size+1):
            try:
                self.W.append(np.random.rand(n_classes,n_classes))
                self.b.append(np.zeros((1,n_classes)))
            except:
                self.W=[]
                self.b=[]
                self.W.append(np.random.rand(n_features,n_classes))
                self.b.append(np.ones((1,n_classes)))
```

Now, our number of layers is a hyperparameter which will decide the number of different matrices of weights and bias.

```
def forward_propagation(self, X):
        self.a = []
        for i in range(self.hidden_size):
            try:
                z = np.dot(self.a[i-
1], self.W[i]) + self.b[i] # [n_samples x 26]
            except:
                z = np.dot(X,self.W[i]) + self.b[i] # [n_samples x 26]
            self.a.append(self.linear_activation(z))
        z = np.dot(self.a[self.hidden_size -
1],self.W[self.hidden_size]) + self.b[self.hidden_size]
        output = self.softmax(z)
        return output
```

The forward propagation is different as well. The *try* is the code that will be run throughout the inner layers and the *except* for the first inner layer. In the end of the loop, the output layer is calculated.

```
def back_propagation(self, X,y,output, learning_rate):
        dLdwlist = []
        dLdblist = []

        for i in range(self.hidden_size, 0, -1):
            if i == self.hidden_size: # output backprop
                dL_dzhs = self.softmax_derivative(output,y) # [n_samples]
 x 26
                dL_dwhs = np.dot(self.a[i-1].T,dL_dzhs) # [26 x 26]
                dLdwlist.insert(0,dL_dwhs)
                dLdblist.insert(0,np.sum(dL_dzhs, axis = 0))
                chain = dL_dzhs # [n_samples x 26]
            else:
                dL_dai = np.dot(chain,self.W[i+1].T)
                dai_dzi = self.linearact_derivative(self.a[0])
                chain = dL_dai * dai_dzi
                dL_dwi = np.dot(self.a[i].T, chain)
                dLdwlist.insert(0, dL_dwi)
                dLdblist.insert(0,np.sum(chain, axis = 0))

        dL_dai = np.dot(chain,self.W[1].T)
        dai_dzi = self.linearact_derivative(self.a[0])
        chain = dL_dai * dai_dzi
        dL_dwi = np.dot(X.T, chain)
        dLdwlist.insert(0, dL_dwi)
        dLdblist.insert(0,np.sum(chain, axis = 0))

        for i in range(len(self.W)):
            self.W[i] -= learning_rate * dLdwlist[i]
            self.b[i] -= learning_rate * dLdblist[i]
```

The problems arise in the backpropagation function. Two auxiliary lists were created to store the values to update the values of weights and biases. Afterwards, a loop starts iterating from the output layer to the inner layers until it achieves the first layer, which will be done after the loop. In the end, update the weights with the values stored in the auxiliary lists.