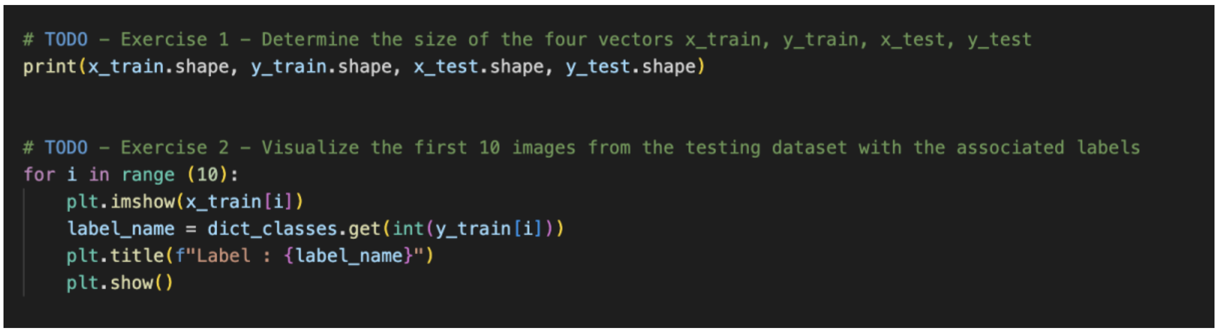
Computer Vision and Deep Learning: Lab 01

“Data Labelling using Linear Classifiers”

Valentin Six

Exercises 1 and 2:

To obtain the size of the four vectors and to display the first 10 images with their corresponding labels, we can use the following code:



```
# TODO - Exercise 1 - Determine the size of the four vectors x_train, y_train, x_test, y_test
print(x_train.shape, y_train.shape, x_test.shape, y_test.shape)


# TODO - Exercise 2 - Visualize the first 10 images from the testing dataset with the associated labels
for i in range (10):
    plt.imshow(x_train[i])
    label_name = dict_classes.get(int(y_train[i]))
    plt.title(f"Label : {label_name}")
    plt.show()
```

Figure 1: Code for vector sizes and labeled image displaying

The displayed images with the corresponding label name look like this:

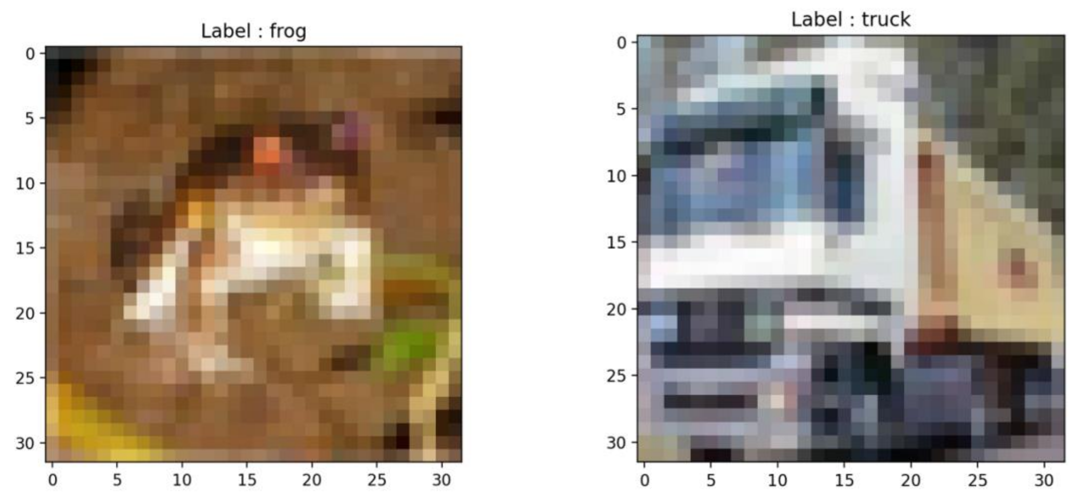
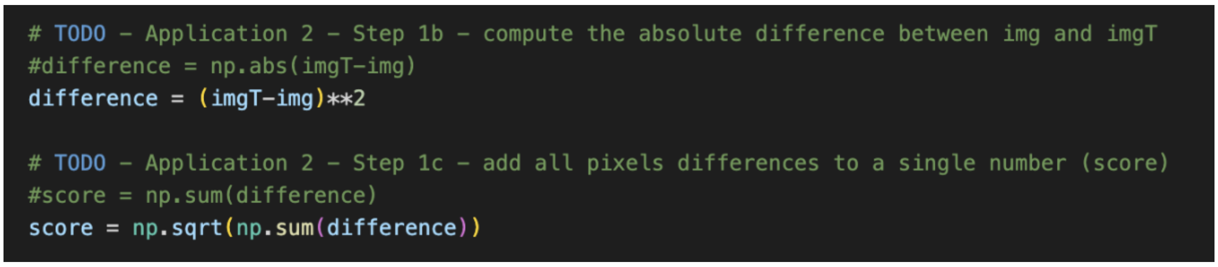


Figure 2: example of displayed labeled images (frog and truck images)

Exercise 3:

We can change the way we measure the "distance" from one image to another.
In the following code (see Figure , the difference and score with L1 distance have been commented and we added the formulas to compute the L2 distance.



```
# TODO - Application 2 - Step 1b - compute the absolute difference between img and imgT
#difference = np.abs(imgT-img)
difference = (imgT-img)**2

# TODO - Application 2 - Step 1c - add all pixels differences to a single number (score)
#score = np.sum(difference)
score = np.sqrt(np.sum(difference))
```

Figure 3: Code to compute the scores using L1 or L2 distance

Let’s look at the accuracies achieved with the two different distances:

With L2 distance we get an accuracy of 35,5 %.
With L1 distance we get an accuracy of 36,5 %.

We obtain similar accuracies with both distances, L1 being a little more adequate for our use case.

Exercise 4:

Nearest Neighbors	K=3	K=5	K=10	K=20	K=50
Accuracy	35,5 %	38 %	39 %	38,5 %	35 %

Table 1: System accuracy for various numbers of nearest neighbors, with L1 distance

We can note that the prediction time is stable: increasing the number of nearest neighbors does not require more computing as the number of neighbors is negligeable compared to the other operations in the code.
If we were to use K=1, our KNN Classifier would just be a NN classifier: our prediction would be the label of the nearest neighbor, just like in the previous “*predictLabelNN*” program.

Exercise 5:

Nearest Neighbors	K=3	K=5	K=10	K=20	K=50
Accuracy	34,5 %	36 %	38 %	35,5 %	35,5 %

Table 2: System accuracy for various numbers of nearest neighbors, with L2 distance

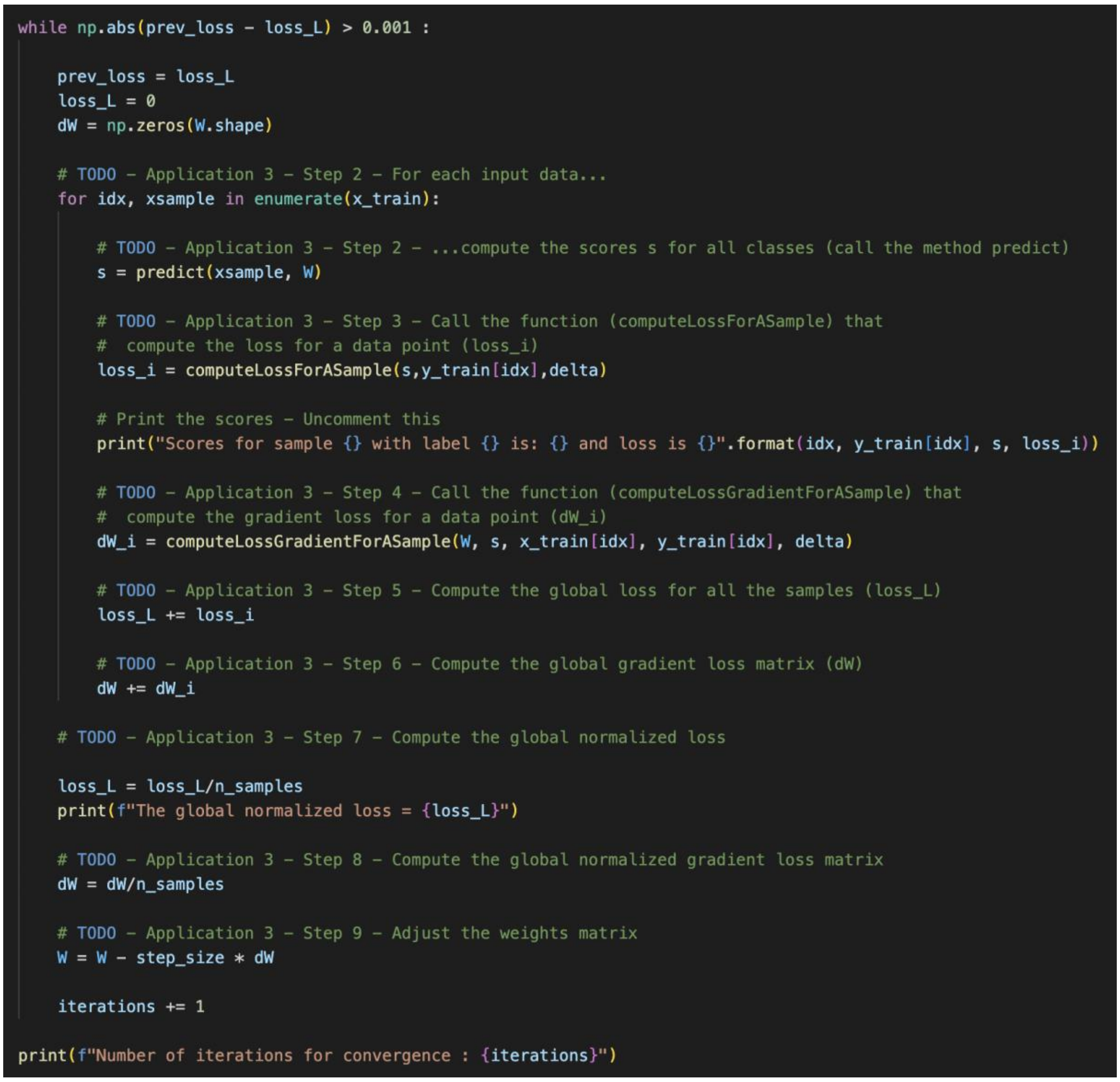
We can observe that the accuracies achieved with the L1 distance and with the L2 distance are quite similar: it seems that we reach maximum accuracy with around K=10 Nearest Neighbors. If we look at less neighbors, we get less information and we have something called “underfitting”; but if we choose more than 10 neighbors, than we are trying to make much more complex relationships and we are “overfitting” the data.

The difference between the use of L1 distance and L2 distance is not that important in our case, but it may be important in other projects where the choice of distance really makes a difference, or if having 0,5 % more accuracy is highly beneficial.

Exercises 6 and 7:

To repeat the process on a new set of weights (a changed W matrix), we nedd to update the value of matrix W and create a while loop: we continue to iterate and update the values of W until our criteria (a certain value of loss variation) is reached. To count the number of iterations needed to reach our criteria, we introduce a variable to which we add "1" every time we enter the loop.

The following code show the corresponding "while" loop:



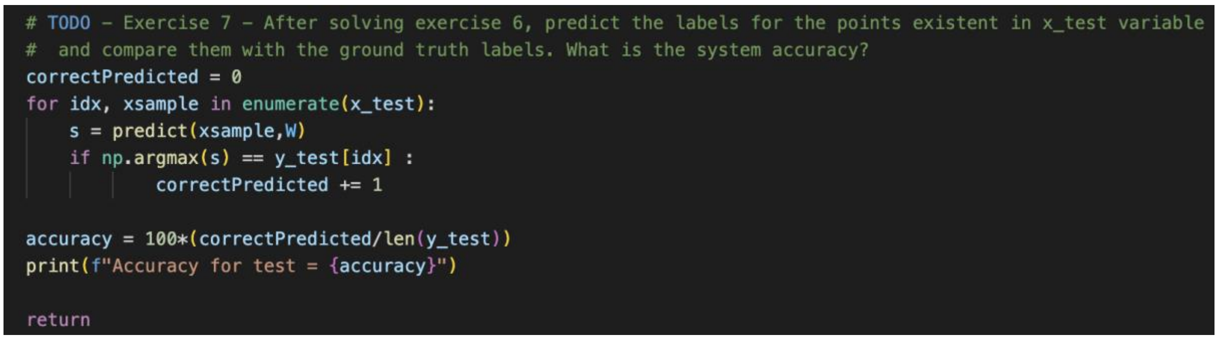
```
while np.abs(prev_loss - loss_L) > 0.001 :

    prev_loss = loss_L
    loss_L = 0
    dW = np.zeros(W.shape)

    # TODO - Application 3 - Step 2 - For each input data...
    for idx, xsample in enumerate(x_train):

        # TODO - Application 3 - Step 2 - ...compute the scores s for all classes (call the method predict)
        s = predict(xsample, W)

        # TODO - Application 3 - Step 3 - Call the function (computeLossForASample) that
        #  compute the loss for a data point (loss_i)
        loss_i = computeLossForASample(s,y_train[idx],delta)

        # Print the scores - Uncomment this
        print("Scores for sample {} with label {} is: {} and loss is {}".format(idx, y_train[idx], s, loss_i))

        # TODO - Application 3 - Step 4 - Call the function (computeLossGradientForASample) that
        #  compute the gradient loss for a data point (dW_i)
        dW_i = computeLossGradientForASample(W, s, x_train[idx], y_train[idx], delta)

        # TODO - Application 3 - Step 5 - Compute the global loss for all the samples (loss_L)
        loss_L += loss_i

        # TODO - Application 3 - Step 6 - Compute the global gradient loss matrix (dW)
        dW += dW_i

    # TODO - Application 3 - Step 7 - Compute the global normalized loss

    loss_L = loss_L/n_samples
    print(f"The global normalized loss = {loss_L}")

    # TODO - Application 3 - Step 8 - Compute the global normalized gradient loss matrix
    dW = dW/n_samples

    # TODO - Application 3 - Step 9 - Adjust the weights matrix
    W = W - step_size * dW

    iterations += 1

print(f"Number of iterations for convergence : {iterations}")
```

Figure 4: Code for the repetition of the process of gradient descent

To reach the convergence criteria (loss variation criteria), the loop is run through 188 times (see *Figure 6* below)

To test our classifier, we must make predictions on the test set and compute the accuracy.

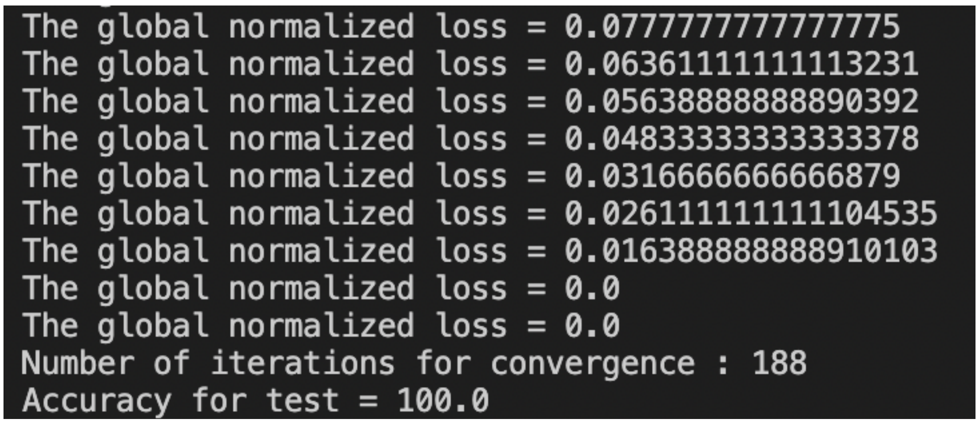
Our code looks like this:



```
# TODO - Exercise 7 - After solving exercise 6, predict the labels for the points existent in x_test variable
#  and compare them with the ground truth labels. What is the system accuracy?
correctPredicted = 0
for idx, xsample in enumerate(x_test):
    s = predict(xsample,W)
    if np.argmax(s) == y_test[idx] :
        correctPredicted += 1

accuracy = 100*(correctPredicted/len(y_test))
print(f"Accuracy for test = {accuracy}")

return
```

Figure 5: Code for testing the classifier and computing the accuracy over the test set

At the end, the accuracy achieved is 100 %. It means that in each of the three test samples, our classifier was able to predict the correct label / class.



```
The global normalized loss = 0.07777777777777775
The global normalized loss = 0.06361111111113231
The global normalized loss = 0.05638888888890392
The global normalized loss = 0.04833333333333378
The global normalized loss = 0.0316666666666879
The global normalized loss = 0.02611111111104535
The global normalized loss = 0.01638888888910103
The global normalized loss = 0.0
The global normalized loss = 0.0
Number of iterations for convergence : 188
Accuracy for test = 100.0
```

Figure 6: Accuracy results for the test set

Exercise 8:

We now want to use the famous “Iris Flowers Dataset” to try our brand-new classifier. First, we need to load the data: the initial data is a csv file, easily transformable to a Data Frame, using pandas.

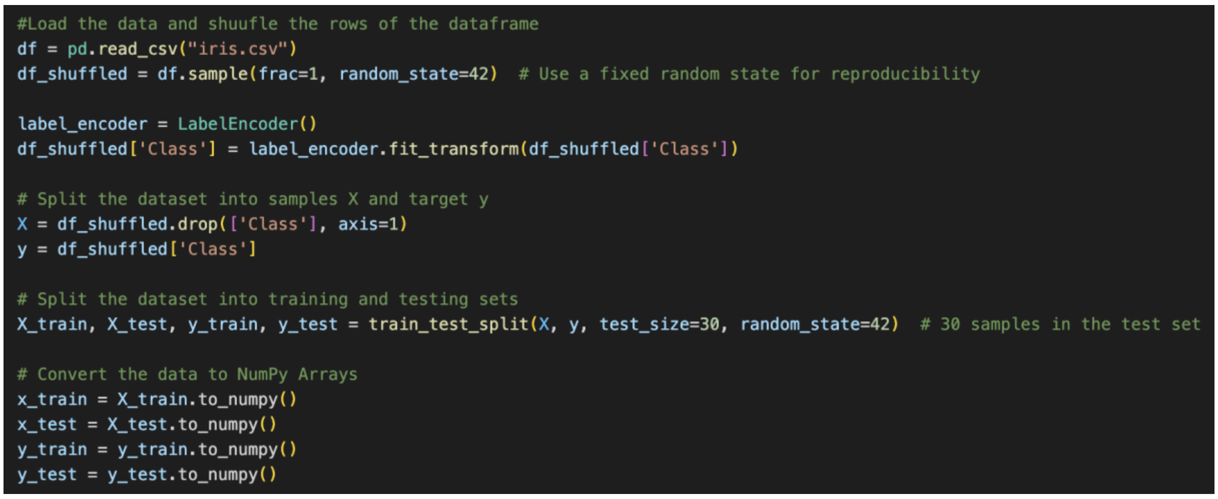
We also shuffle the data so the samples from different labeled to classes are uniformly distributed in the dataset (otherwise the training set might contain more samples from a certain class and same problem for the test set).

One problem we need to solve is the names of the labels: in the original file, each label (or class) is written in text. This means that each element of the target vector, the vector containing the labels, is a string element. It is a problem because we want to compute the loss with respect to those elements; therefor they need to be numerical.

A solution to this issue is to use an "encoder": it helps us convert non-numerical features to numerical ones. This is a good practice in Machine Learning because the numerical values are well easier to deal with than non-numerical values.

Next, we split the data into four different sets: training sets (training samples and training labels) and test sets (test samples and test labels).

Finally, we convert our sets to NumPy Arrays, so that we can use the same code as before.



```
#Load the data and shuufle the rows of the dataframe
df = pd.read_csv("iris.csv")
df_shuffled = df.sample(frac=1, random_state=42)  # Use a fixed random state for reproducibility

label_encoder = LabelEncoder()
df_shuffled['Class'] = label_encoder.fit_transform(df_shuffled['Class'])

# Split the dataset into samples X and target y
X = df_shuffled.drop(['Class'], axis=1)
y = df_shuffled['Class']

# Split the dataset into training and testing sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=30, random_state=42)  # 30 samples in the test set

# Convert the data to NumPy Arrays
x_train = X_train.to_numpy()
x_test = X_test.to_numpy()
y_train = y_train.to_numpy()
y_test = y_test.to_numpy()
```

Figure 7: Code for loading, shuffling, and formatting the data

We can try to answer different questions to evaluate the performance of our classifier on this dataset:

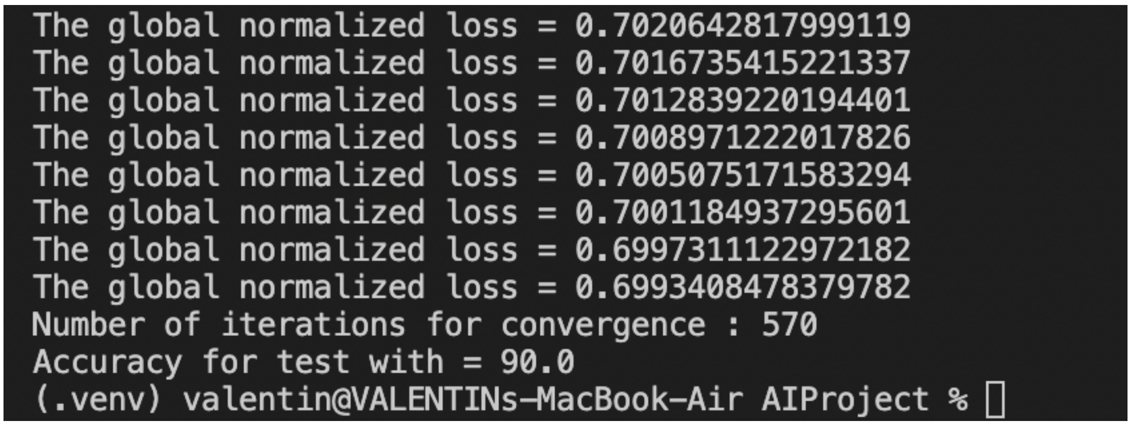
1) What is the optimal value for the weights adjustment step to obtain the maximum testing accuracy?

If the value for the adjustment step (or learning rate) test is too high, the loss function will explode, and the model will not be able to learn (quite bad for us). In the opposite case, if the value for the adjustment step is too low, the learning process of the model will be too slow (which is not so bad but still annoying).

In our case, after trying multiple learning rates (or adjustment steps), the optimal value is around 0.001.

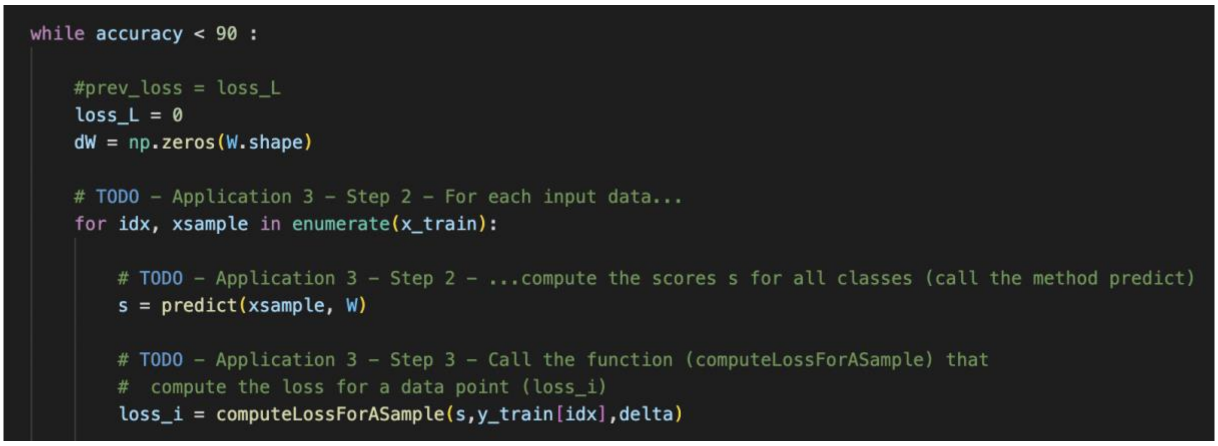
2) What is the minimum number of steps necessary to train the system to obtain an accuracy superior to 90%?

Using a standard learning-rate of 0.001 if we loop the iterations on the accuracy, using a while loop, we can obtain the following results:



```
The global normalized loss = 0.7020642817999119
The global normalized loss = 0.7016735415221337
The global normalized loss = 0.7012839220194401
The global normalized loss = 0.7008971222017826
The global normalized loss = 0.7005075171583294
The global normalized loss = 0.7001184937295601
The global normalized loss = 0.6997311122972182
The global normalized loss = 0.6993408478379782
Number of iterations for convergence : 570
Accuracy for test with = 90.0
(.venv) valentin@VALENTINs-MacBook-Air AIProject % 
```

Figure 8: Number of iterations of the loop to achieve 90% accuracy



```
while accuracy < 90 :

    #prev_loss = loss_L
    loss_L = 0
    dW = np.zeros(W.shape)

    # TODO - Application 3 - Step 2 - For each input data...
    for idx, xsample in enumerate(x_train):

        # TODO - Application 3 - Step 2 - ...compute the scores s for all classes (call the method predict)
        s = predict(xsample, W)

        # TODO - Application 3 - Step 3 - Call the function (computeLossForASample) that
        #  compute the loss for a data point (loss_i)
        loss_i = computeLossForASample(s,y_train[idx],delta)
```

Figure 9: Code for the loop on the accuracy; similar code to Figure 4

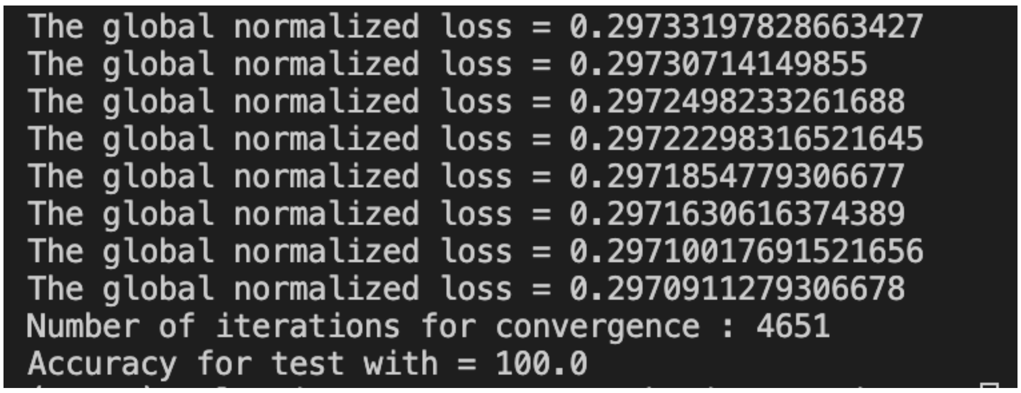
3) Is the system influenced by the random initialization of the weights matrix? Justify your answer?

Yes, the system will be influenced by the random initialization of the weights matrix. There are multiple reasons for this claim: the speed of convergence will change depending on the initial matrix; usually the more Asymetrix the initial weights matrix is, the faster the algorithm will converge (random initialization helps in this case as it avoids symmetry and repetitions).

We have to keep in mind that weights initialization is a complex problem, and that multiple solution are available to us: for example, we could use the famous Xavier initialization.

4) Can this system reach a 100% of accuracy in the testing stage?

Because we only have 150 observations in the dataset and because our test set is so small (30 values for testing), it is quite likely that we can achieve 100% accuracy. If we give the system enough time, with a reasonably small learning rate (or step) and a strict precision condition to stop the loop, we can achieve the following results:



```
The global normalized loss = 0.29733197828663427
The global normalized loss = 0.29730714149855
The global normalized loss = 0.2972498233261688
The global normalized loss = 0.29722298316521645
The global normalized loss = 0.2971854779306677
The global normalized loss = 0.2971630616374389
The global normalized loss = 0.29710017691521656
The global normalized loss = 0.2970911279306678
Number of iterations for convergence : 4651
Accuracy for test with = 100.0
```

Figure 10: Full accuracy on the test set, with high number of iterations