Lab session #7
Artificial Neural Networks

Dr Zied M'NASRI

Introduction

In this Lab session, we learn how to implement and train an Artificial Neural Networks to make predictions: a) binary classification, b) multiclass classification and c) regression. In this lab session, you will need to use:

1. Python tutorial (see Lab session 1 material)
2. Colab/Jupyter tutorial (see Lab session 2 material)
3. Lecture 7 notes
4. Lab 7 code example (Artificial Neural Networks with Backpropagation)
5. Google's Colab / Jupyter notebbok

Tutorial : Multi-class Classification using MLP with Error Backpropagation

Upload the *UB_AI_Lab7_Example_NN_with_BackPropagation.ipynb* to Google Colab and run it, to understand how ANN are trained with Backpropagation, using a random dataset.

In this tutorial, we use the functions provided in the code example *UB_AI_Lecture7_Example_NN_with_BackPropagation.ipynb* to classify the example Iris dataset.

1. Upload the file *UB_AI_Lab7_Example_NN_with_BackPropagation.ipynb*
   To Google Colab or Jupyter Notebooks
2. Import the dataset Iris from sklearn.datasets

```
from sklearn import datasets
iris = datasets.load_iris()
print("Targets labels are: ",iris.target_names)
print("Target values are: ",np.unique(iris.target))
```

3. Separate Iris data into 2 data frames: X (data) and y (targets).
   Hint! Use the method `pd.dataFrame()` and specify columns

```
X = pd.DataFrame(iris['data'],columns=['sepal length','sepal
width','petal lenghth','petal width'])
y = pd.DataFrame(iris['target'],columns=['target'])
print("Size of input features : ", X.shape)
print("Size of output targets : ", y.shape)
Nfeatures = X.shape[1]
print(f"Number of input features = ",Nfeatures)
Ntargets = y.shape[1]
print("Number of output targets = ",Ntargets)
```

4. Print the following results: Targets labels, Targets values, Size of input features, Size of output targets,

```
Targets labels are:  ['setosa' 'versicolor' 'virginica']
Target values are:  [0 1 2]
Size of input features :  (150, 4)
Size of output targets :  (150, 1)
Number of input features =  4
Number of output targets =  1
```

Figure 1.1. Input and output data

5. Split *X* and *y* into *X_train*, *y_train*, *X_test* and *y_test* using a split ratio 0.3 and `random_state= 42` using the method `train_test_split()`
   Hint! You need to import `train_test_split()` from `sklearn.model_selection`

```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)
for i in range(len(y_train)):
  y_train.iloc[i,0] = int(y_train.iloc[i,0])
```

6. Form a list called dataset_train by concatenating X_train and y_train, and print the first 5 rows of dataset_train
   Hint! Use the method `pd.concat()` with its attribute method `values.tolist()`

```
dataset_train = pd.concat([X_train, y_train],
axis=1).values.tolist()
print(f"dataset_train: {dataset_train[0:5][:]}")
```

7. Do the same for dataset_test with X_test and Y_test

```
dataset_test = pd.concat([X_test, y_test], axis=1).values.tolist()
print(f"dataset_test: {dataset_test[0:5][:]}")
```

```
dataset_train: [[4.6, 3.6, 1.0, 0.2, 0.0], [5.7, 4.4, 1.5, 0.4, 0.0], [6.7, 3.1, 4.4, 1.4, 1.0], [4.8, 3.4, 1.6, 0.2, 0.0], [4.4, 3.2, 1.3, 0.2, 0.0]]
dataset_test: [[6.1, 2.8, 4.7, 1.2, 1.0], [5.7, 3.8, 1.7, 0.3, 0.0], [7.7, 2.6, 6.9, 2.3, 2.0], [6.0, 2.9, 4.5, 1.5, 1.0], [6.8, 2.8, 4.8, 1.4, 1.0]]
```

Figure 1.2. First 5 rows of the training and test sets

8. In Section “Network initialisation”:
   a. Input the number of nodes in the hidden layer
   b. Set n_inputs as the number of features as the length of X_train[0]
   c. Set n_outputs as the length of the value counts in y_train
      Hint! Use the attribute method `.value_count()`
   d. Initialize the network by calling the function `initialize_network( )` (defined in the example)
   e. Print the initialized network

```
# Network initialisation
n_hidden = int(input('Enter the number of nodes in the hidden 
layer: '))
n_inputs = len(dataset_train[0]) - 1
n_outputs = len(y_train.value_counts())
print(f"number of input features = {n_inputs}")
print(f"number of output labels = {n_outputs}")
print(f"number of nodes in the hidden layer = {n_hidden}")
network = initialize_network(n_inputs, n_hidden, n_outputs)
print(f"Initial network weights: {network}")
```

```
Enter the number of nodes in the hidden layer: 2
number of input features = 4
number of output labels = 3
number of nodes in the hidden layer = 2
[[{'weights': [0.4896935204622582, 0.029574963966907064, 0.04348729035652743, 0.703382088603836, 0.9831877173096739]}
```

Figure 1.3. Network initialization

9. In Section “Network training”:
   a. Enter the learning rate (between 0 and 1)
   b. Enter the number of epochs
   c. Print the parameters above, plus the initial weights
   d. Train the networks using the function `train_network()` mentioned in the example code

```
# Network training
sum_error = 0
l_rate = float(input('Enter the learning rate (between 0 and 1):
'))
n_epoch = int(input('Enter the number of epochs: '))
print(f"Selected learning rate = {l_rate}")
print(f"Selected number of epochs = {n_epoch}")
train_network(network, dataset_train, l_rate, n_epoch, n_outputs)
```

10. In section “Test making predictions with the network”:
    a. Replace the test dataset by dataset_test (created above)
    b. Print the target and the predicted values

```
# Test making predictions with the network
print(f"final networks: {network}")
target = prediction = np.zeros(len(dataset_test))
for i in range(len(dataset_test)):
  row = dataset_test[i][:]
  print(row)
  target[i]=int(dataset_test[i][-1])
  prediction[i] = predict(network, row)

print("targets= ", target)
print("predictions= ", prediction)
```

```
targets=  [2. 0. 2. 2. 2. 0. 1. 2. 2. 2. 2. 0. 0. 0. 0. 2. 2. 2. 2. 2. 0. 2. 0. 2.
 2. 2. 2. 2. 0. 0.]
predictions=  [2. 0. 2. 2. 2. 0. 1. 2. 2. 2. 2. 0. 0. 0. 0. 2. 2. 2. 2. 2. 0. 2. 0. 2.
 2. 2. 2. 2. 0. 0.]
```

Figure 1.4. Targets and predictions after test

11. In the section "Evaluate NN model", use the method `classification_report()` to calculate the evaluation metrics. Print the classification report

```
# Evaluate NN model
#from sklearn import metrics
from sklearn.metrics import confusion_matrix, classification_report
print(f"Confusion matrix: \n {confusion_matrix(target,
prediction)}")
print(f"Classification_report: \n {classification_report(target,
prediction)}")
```

```
Confusion matrix:
 [[10  0  0]
 [ 0  1  0]
 [ 0  0 19]]
Classification_report:
               precision    recall  f1-score   support

         0.0       1.00      1.00      1.00        10
         1.0       1.00      1.00      1.00         1
         2.0       1.00      1.00      1.00        19

    accuracy                           1.00        30
   macro avg       1.00      1.00      1.00        30
weighted avg       1.00      1.00      1.00        30
```

Figure 1.5. Evaluation metrics on the test set

Exercises

Exercise #1- Binary classification using MLP with predefined functions and grid search

In this exercise, we aim at using predefined functions in Python libraries (Scikit learn and Pandas) to: a) Preprocess data, b) Initialize and train an MLP, c) Improve the evaluation metrics using a grid search to select the best performing parameters.

1. Upload the file *play_tennis.csv* (see lab 5 or lab 6) and print the dataset header

	Day	Outlook	Temperature	Humidity	Wind	PlayTennis
0	D1	Sunny	85	85	Weak	No
1	D2	Sunny	80	90	Strong	Yes
2	D3	Overcast	83	78	Weak	Yes
3	D4	Rain	70	96	Weak	Yes
4	D5	Rain	68	80	Weak	No

Figure 2.1. play_tennis dataset header

2. Replace the values of the features Outlook and Wind to numerical, as follows (see lab 6):
   Outlook → Sunny = 0, Overcast = 0.5, Rain = 1
   Wind → Weak = 0, Strong = 1
   PlayTennis → No =0, Yes = 1
3. Normalise the values of Temperature using the min-max normalization
   Hint! Use the following formula:

```
dataset['featureName'] =(dataset['featureName'] - min(dataset['
featureName ']))/(max(dataset[' featureName ']) - min(dataset['
featureName ']))
```

	Day	Outlook	Temperature	Humidity	Wind	PlayTennis
0	D1	0.0	0.833333	0.738095	0	0
1	D2	0.0	0.714286	0.857143	1	1
2	D3	0.5	0.785714	0.571429	0	1
3	D4	1.0	0.476190	1.000000	0	1
4	D5	1.0	0.428571	0.619048	0	0

Figure 2.2. Normalized dataset

4. Declare X and y as the feature columns and the target column, respectively

Hint! Use the attribute `datasetName.iloc` (see Lab 6)

5. Split X and y into X_train, X_test, y_train, y_test using a test/train ratio = 0.3 (and <0.5 in any case). Use random_state = 42. Print the shape of each subset.

```
Size of training features :  (279, 4)
Size of training targets :  (279,)
Size of test features :  (120, 4)
Size of test targets :  (120,)
```

Figure 2.3. Training and test subsets

6. Define an MLP classifier using the method `MLPClassifier()` with the following parameters:

```
hidden_layer_sizes=(200,100),
max_iter = 300,
activation = 'relu',
solver = 'sgd',
learning_rate='constant',
learning_rate_init=0.001,
batch_size='auto',
tol=0.00001,
verbose=True
```

Hint! Import MLPClassifier from sklearn.neural_network

7. Train the MLP classifier and print the trained weights
Hint! Use the method `MLPClassifierName.fit()` to train the classifier.
The weights are stored in `MLPClassifierName.coefs_`

```
MLP classifer weights = [array([[ 0.07255927, -0.02182532, -0.05824862,  0.07662956,  0.09351607,
        -0.12959692, -0.12025699, -0.16760501,  0.06665141,  0.09019231,
         0.12430765, -0.00388537,  0.12044874,  0.12309126, -0.06039614,
        -0.08944968,  0.11498595, -0.0411311 , -0.054184  , -0.13232408,
        -0.13813591,  0.09055127, -0.04877255,  0.11799414,  0.10812991,
         0.06079494,  0.05857149,  0.0789826 ,  0.15238499,  0.11716647,
```

Figure 2.4. MLP classifer weights

8. Test the trained model MLPClassifierName on X_test
Hint! Use the method `MLPClassfierName.predict()`

9. Calculate the accuracy and print the classification report
Hint! Import and use the methods `accuracy_score()` and `classification_report()` from `sklearn.metrics`

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00        55
           1       0.54      1.00      0.70        65

    accuracy                           0.54       120
   macro avg       0.27      0.50      0.35       120
weighted avg       0.29      0.54      0.38       120
```

Figure 2.5. Classification report

10. Plot the loss curve

    Hint! The loss curve is stored in `MLPClassifierName.loss_curve_`

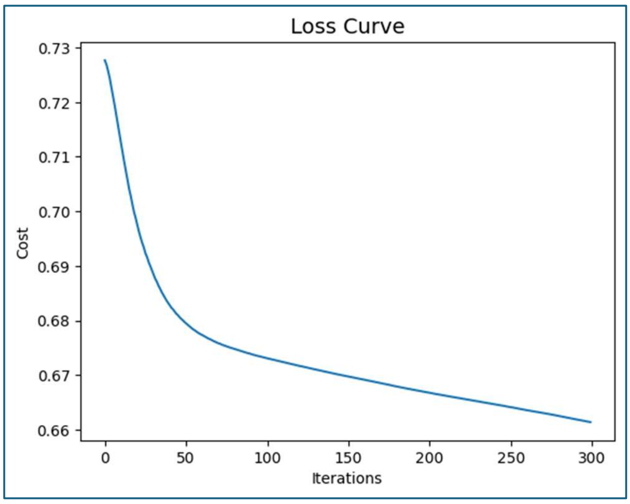

Figure 2.6. Loss curve

11. To improve the performance of the MLP classifier, we can try different configurations using a grid search. Define the grid parameters as follows:

```
param_grid = {'hidden_layer_sizes': [(150,100,50), (120,80,40),
(100,50,30)],'max_iter': [50, 100, 150],'activation': ['tanh', 'relu'],
     'solver': ['sgd', 'adam'],'alpha':
[0.0001,0.05],'learning_rate': ['constant','adaptive'],}
```

12. Define a gridName model using the method `GridSearchCV()` with the following parameters:

    `N_jobs = -1, cv =5`

13. Train the gridName model with X_train and y_train
14. Print the gridName model best parameters, found after training

    Hint! The best parameters are stored in `gridName.best_parameters_`

15. Make predictions on the test set using the gridName model.
16. Print the accuracy and the classification report. What do you notice about the models performance?

```
Accuracy: 0.78
              precision    recall  f1-score   support

           0       0.77      0.73      0.75        55
           1       0.78      0.82      0.80        65

    accuracy                           0.78       120
   macro avg       0.77      0.77      0.77       120
weighted avg       0.77      0.78      0.77       120
```

Figure 2.7. Classification report after parameters optimisation using grid search

Exercise #2- Regression with MLP and input data standardization

This exercise aims at using MLP for regression, i.e. continuous-valued output prediction. We use also data standardization, i.e. transforming the input data so that it has a zero-and a unit-variance, instead of min-max normalisation. Also we use a different evaluation metric, i.e. the root mean square error, since the output is not categorical.

1. Import the following packages:

```
import numpy as np
from sklearn import datasets
from sklearn.datasets import fetch_california_housing
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.neural_network import MLPRegressor
from sklearn.metrics import mean_squared_error
```

2. Declare housing as the result of `fetch_california_housing()` and X, y as the attributes `housing.data()` and `housing.target()`, respectively. Print X and y and their size.

```
Size of X:  (20640, 8)
[[   8.3252       41.            6.98412698 ...    2.55555556
    37.88       -122.23      ]
 [   8.3014       21.            6.23813708 ...    2.10984183
    37.86       -122.22      ]
 [   7.2574       52.            8.28813559 ...    2.80225989
    37.85       -122.24      ]
 ...
 [   1.7          17.            5.20554273 ...    2.3256351
    39.43       -121.22      ]
 [   1.8672       18.            5.32951289 ...    2.12320917
    39.43       -121.32      ]
 [   2.3886       16.            5.25471698 ...    2.61698113
    39.37       -121.24      ]]
Size of y:  (20640,)
[4.526 3.585 3.521 ... 0.923 0.847 0.894]
```

Figure 3.1. Data and targets and the size of each

3. Transform X and y to X_std and y_std using the mean-variance standardisation formula:

$$z_{std} = \frac{z - \mu}{\sigma}$$

Where μ is the mean value of (z) and σ is its standard deviation
Hint! For each of X and y, mean and standard deviation are obtained using the attribute methods `np.mean(variableName, axis =0)` and `np.std(variableName, axis = 0)`

4. To visualize the effect of standardisation, plot the histograms of y and y_std
   Hint! Use the following methods:
   `sns.set(color_codes=True)`
   `sns.set_style("whitegrid")`
   then for each variable:
   `sns.hisplot`(variableName)
   Finally
   `plot.show()`

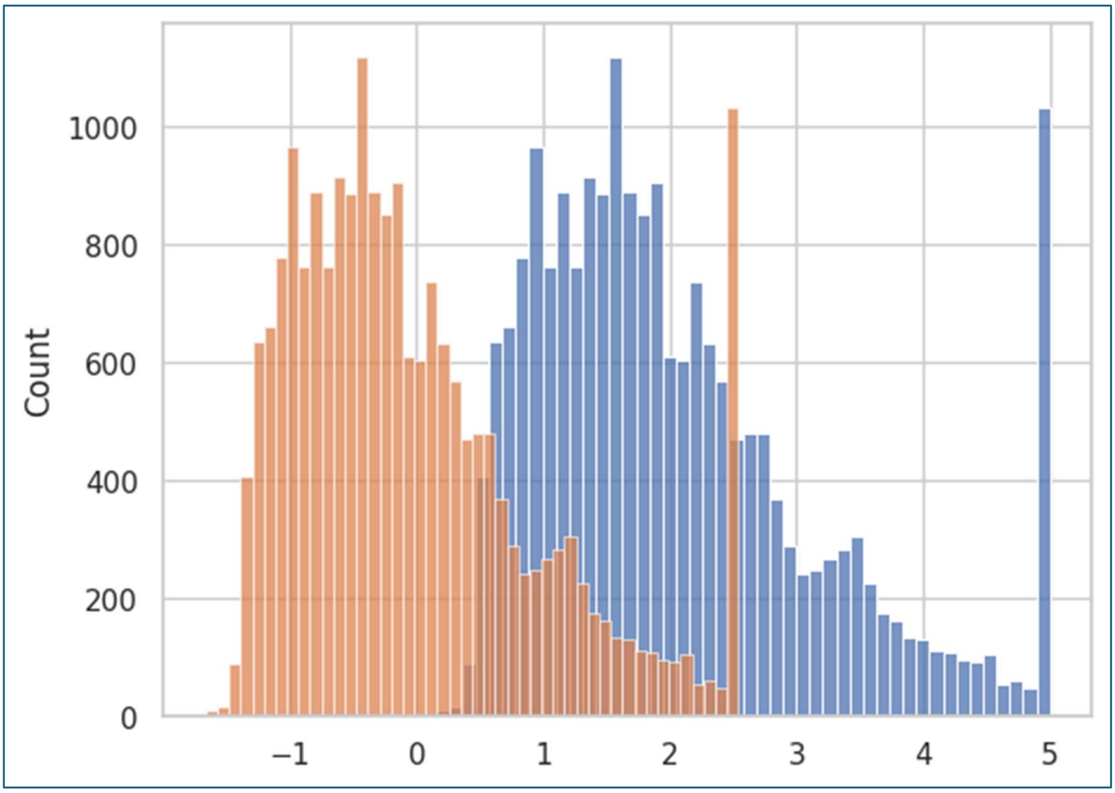

Figure 3.2. Histograms of original (blue) and standardised (amber) targets

5. Split each of X_std and y_std into training and test sets using a test/training ratio = 0.3 and `random_state = 1`. Print the size of each subset

```
Size of X_std_train:  (14448, 8)
Size of X_std_test:  (6192, 8)
Size of y_std_train:  (14448,)
Size of y_std_test:  (6192,)
```

Figure 3.3. Size of each split

6. Initialize an MLP regressor using the following parameters:

```
activation='relu',
hidden_layer_sizes=(10, 100),
alpha=0.001,
random_state=20,
early_stopping=False
```

   Hint! Use the built-in function `MLPRegressor()`
7. Train the model using X_std_train and y_std_train
   Hint! Use the buit-in function `mlpRegressorName.fit()`
8. Plot the training loss curve
   Hint! When calling the function `plt.plot()`, use the attribute `mlpRegressorName.loss_curve_`

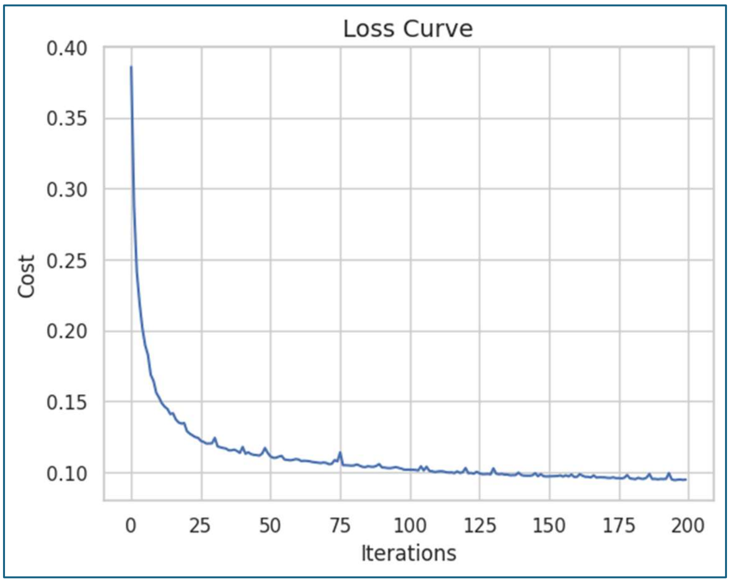

Figure 3.4. Loss curve on the training set

9. Make predictions on the test set using X_std_test
   Hint! Use the function `mlpRegressorName.predict()`
10. Transform the predicted values into the real range of the targets
   Hint! Use the reverse transform, i.e `z = z_std*np.std(z)+np.mean(z)`

```
Standardised predictions:  [ 1.56297565 -1.24122407  0.28785319 ... -0.76120006 -0.41375626
  0.14063342]
Transformed predictions:  [3.87211986 0.63627471 2.40072008 ... 1.19018795 1.59111315 2.23083904]
```

Figure 3.5. Standardised and transformed predictions

11. To evaluate the predicted values, calculate the Root Mean Square Error (RMSE), and its relative value (Relative RMSE) , between the actual targets (y_test) and the transformed predictions.
   Hint! Use the function `mean_squared_error()` to calculate MSE and `np.sqrt()` for the square root.