00000_Exercise_09

January 25, 2024

1 Programming Machine Learning Lab

2 Exercise 8

General Instructions:

1. You need to submit the PDF as well as the filled notebook file.
2. Name your submissions by prefixing your matriculation number to the filename. Example, if your MR is 12345 then rename the files as **"12345_Exercise_9.xxx"**
3. Complete all your tasks and then do a clean run before generating the final PDF. (*Clear All Ouputs* and *Run All* commands in Jupyter notebook)

Exercise Specific instructions::

1. You are allowed to use only NumPy and Pandas (unless stated otherwise). You can use any library for visualizations.

2.0.1 Part 1

In this part, we will code a perceptron. It is simply a single node neural network which processes weighted inputs and performs binary classification.

- Read up on perceptron algorithm. (https://en.wikipedia.org/wiki/Perceptron#Learning_algorithm_for_a_ layer_perceptron).
- Create an object class caled perceptron.
- Train your perceptron using the following different datasets and report the test losses.
- Create an animation of how the decision boundary is updated over the iterations. *You can use any library for this viualization*
- We will use toy datasets for the problem. Set aside 20% of samples from each dataset for testing.
  - **Xlin_sep.npy** and **ylin_sep.npy**. This dataset is linearly separable. Run your algorithm for this data, and you should achieve 100% train and test accuracies!
  - **Xlinnoise_sep.npy** and **ylinnoise_sep.npy**. This dataset is not linearly separable and contains noise. Run your algorithm for this data and observe what happens to the decision boundary in the animation. You should get a test accuracy over 80%.
  - **circles_x.npy** and **circles_y.npy**. This dataset is non-linear. Devise a strategy to make the dataset separable linearly. *(Hint: Polynomial Features)*. Plot the decision boundary showing how the two classes are separated.

[1]:
```python
import numpy as np

class Perceptron:
    def __init__(self, learning_rate=0.1, n_iter=1000, random_state=1):
        self.learning_rate = learning_rate
        self.n_iter = n_iter
        self.random_state = random_state

    def fit(self, X, y):
        rgen = np.random.RandomState(self.random_state)
        self.weights = rgen.normal(loc=0.0, scale=0.01, size=X.shape[1] + 1)
        self.errors_ = []

        for _ in range(self.n_iter):
            errors = 0
            for xi, target in zip(X, y):
                update = self.learning_rate * (target - self.predict(xi))
                self.weights[1:] += update * xi
                self.weights[0] += update
                errors += int(update != 0.0)
            self.errors_.append(errors)
        return self

    def net_input(self, X):
        return np.dot(X, self.weights[1:]) + self.weights[0]

    def predict(self, X):
        return np.where(self.net_input(X) >= 0.0, 1, -1)
```

[3]:
```python
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
from sklearn.preprocessing import PolynomialFeatures
from matplotlib import pyplot as plt



def train_and_evaluate(X_train, X_test, y_train, y_test):
    perceptron = Perceptron()
    perceptron.fit(X_train, y_train)
    y_pred_train = perceptron.predict(X_train)
    y_pred_test = perceptron.predict(X_test)
    print("Train accuracy:", accuracy_score(y_train, y_pred_train))
    print("Test accuracy:", accuracy_score(y_test, y_pred_test))
    return perceptron

X_lin_sep = np.load('Xlin_sep.npy')
```

```
y_lin_sep = np.load('ylin_sep.npy')
X_train, X_test, y_train, y_test = train_test_split(X_lin_sep, y_lin_sep,␣
  ↪test_size=0.2, random_state=42)
perceptron_lin_sep = train_and_evaluate(X_train, X_test, y_train, y_test)

#linearly non-separable dataset with noise
X_linnoise_sep = np.load('Xlinnoise_sep.npy')
y_linnoise_sep = np.load('ylinnoise_sep.npy')
X_train, X_test, y_train, y_test = train_test_split(X_linnoise_sep,␣
  ↪y_linnoise_sep, test_size=0.2, random_state=42)
perceptron_linnoise_sep = train_and_evaluate(X_train, X_test, y_train, y_test)

#circular dataset, which is not linearly separable
circles_x = np.load('circles_x.npy')
circles_y = np.load('circles_y.npy')
circles_y[circles_y == 0] = -1  # Assuming original labels in "circles_y.npy"␣
  ↪are 0 and 1 instead of -1 and 1

poly_features = PolynomialFeatures(degree=2)
circles_x_poly = poly_features.fit_transform(circles_x)

X_train, X_test, y_train, y_test = train_test_split(circles_x_poly, circles_y,␣
  ↪test_size=0.2, random_state=42)
perceptron_circles = train_and_evaluate(X_train, X_test, y_train, y_test)
```

```
Train accuracy: 1.0
Test accuracy: 1.0
Train accuracy: 0.85625
Test accuracy: 0.775
Train accuracy: 1.0
Test accuracy: 1.0
```

2.0.2 Part 2

In this part, we will create a feed-forward neural network

- Load the MNIST classification dataset using sklearn. Split the data into train and test datasets (80-20 split).
- Implement a neural network with forward propagation and backpropagation **from scratch**.
- Use Stochastic Gradient Descent as the optimizer and Cross-entropy as Loss.
- You model class should be flexible in terms of
  - Number of layers
  - Number of hidder parameters.
  - Activation function for each layer (SoftMax, ReLU or tanh)
- Now create a training function that takes the neural network and training data as inputs and updates the weights of the network. This function should also take in the learning rate, number of epochs, and batchsize as input.
- Try out different hyperparameters to train your model and try to achieve >90% test accuracy.

Hints: - Flatten the MNIST data from 2D to 1D. - Use *He weights initialization* for weights. *The He initialization calculates the starting weights as randomly generated matrices using a Gaussian probability distribution with a mean of 0.0 and a standard deviation of sqrt(2/n), where n is the number of inputs to the layer.*

```
[2]: ### Write your code here
```

2.0.3 Part 3

MLPClassifier

In this part, we will use the same dataset from Part 2 and implement a multi-layer perceptron using sklearn. - Import the necessary classes and perform a 5-fold cross-validation by defining a hyperparameter grid for the MLP classifier. - You need to read about the hyperparameters supported by the function and define a grid for them. - Perform a random search on the grid that you have chosen. - Report a single test accuracy with the best found hyperparameters

Note: you can use any sklearn function for this and the next part

```
[3]: ### Write your code here
     from sklearn.neural_network import MLPClassifier
```

MLPRegressor

In this part, we would repeat the all steps taken for MLPClassifier. However, we will try to learn a regression model using MLPRegressor instead. In the end calculate the accuracy of MLPRegressor by using the *test_accuracy_regressor* function provided.

Note: The target output needs to be numerical in this case.

```
[4]: ### Write your code here
     from sklearn.neural_network import MLPRegressor
```

```
[1]: ## Sample code
     import numpy as np
     y_true = np.arange(10)
     y_pred = np.array([-0.1,1.2,1.9,2.9,4.5,6,20,4.5,6.9,8.5])

     def test_accuracy_regressor(y_true,y_pred):
         ### the function assumes both inputs to be 1-D arrays
         assert y_true.shape==y_pred.shape, f"y_true and y_pred needs to be of same␣
      ↪shape, but found y_true: {y_true.shape} and y_pred:{y_pred.shape}"
         assert len(y_pred.shape)==1, f'inputs should be 1-D, but found them as␣
      ↪{len(y_pred.shape)}-D'

         return np.sum((np.round(y_pred,0).astype(int))==y_true)/y_pred.size

     test_accuracy_regressor(y_true,y_pred)
```

```
[1]: 0.5
```

Comment on the performance of MLP Regressor vs MLP Classifier