Assignment1 Report

周澳蕾 11811337 （2022.3.10）

Introduction

In assignment 1 of cs324 deep learning, we were asked to implement a perceptron. After constructing dataset in normal distribution, we train it with training set, and test its accuracy on both of the trainning set and testing set. Then we use basic rule perception to implement a MLP, given parameters such as number of hidden layers, number of neurons of each layer, maximum epoch numbers, learning rate and so on. Finally, we use Jupyter notebook to disply the result.

- Perceptron: an algorithm for supervised learning of binary classifiers, which can decide whether of not an input, represented by a vector, belongs to some specific class.
- Multi-layer perceptron(MLP): Combine may perception as Linear hidden (or output) layers of neurons to make a classifier which can classify datasets with more complicated distribution.

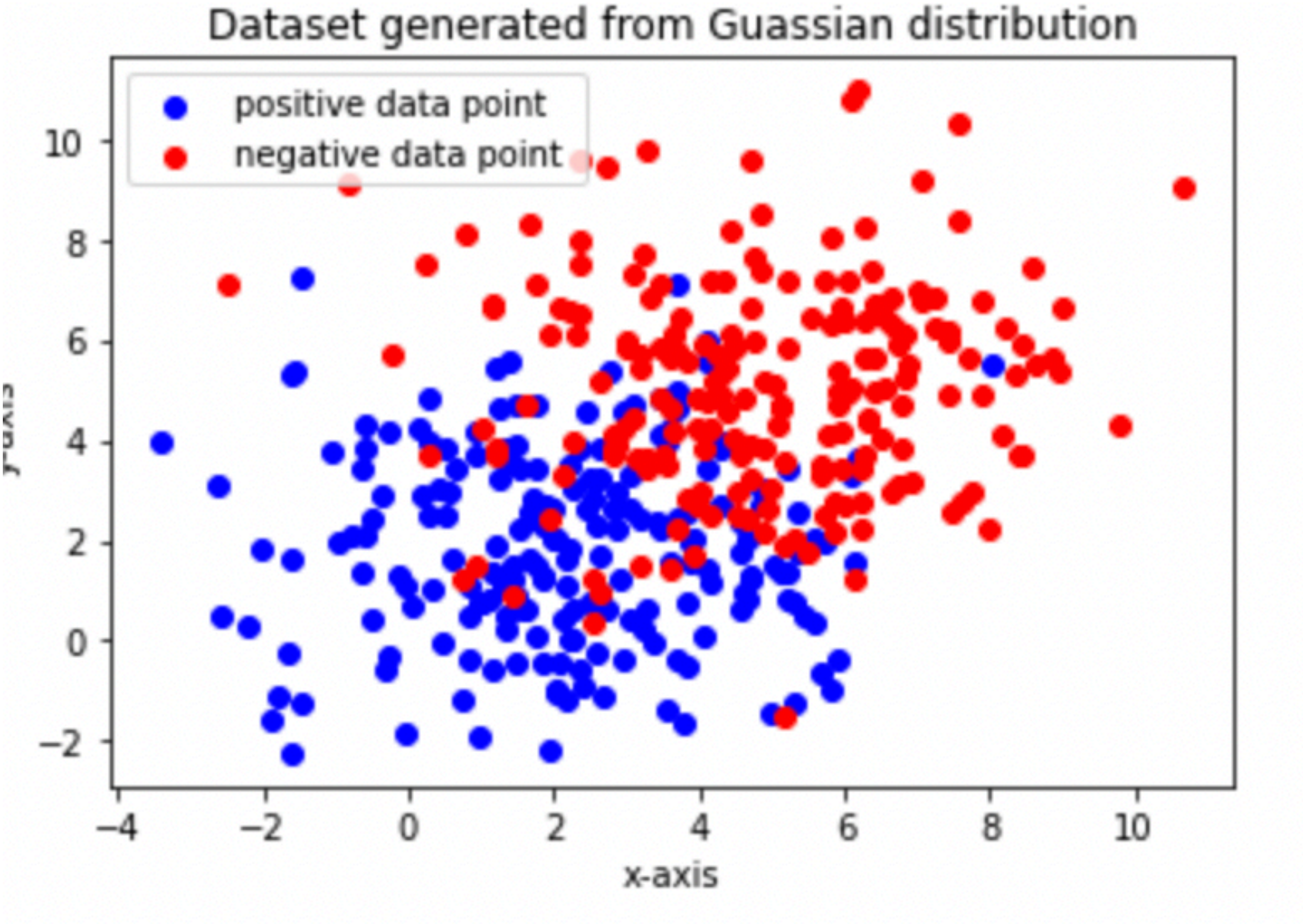
how to run:

Defined in `perceptron.ipynb`/`mlp.ipynb` and the main method in `perceptron.py`/`train_mlp_numpy.py`.

Part1: The Perceptron

Task1: Generate Dataset

- Generate two Gaussian distribution (2D) and sample 200 of each of them.
- Input means and variances to two Gaussian distribution to generate dataset using numpy.random.normal package
- The dataset generated with mean = (2, 5); var = (2, 2) are ploted as below.



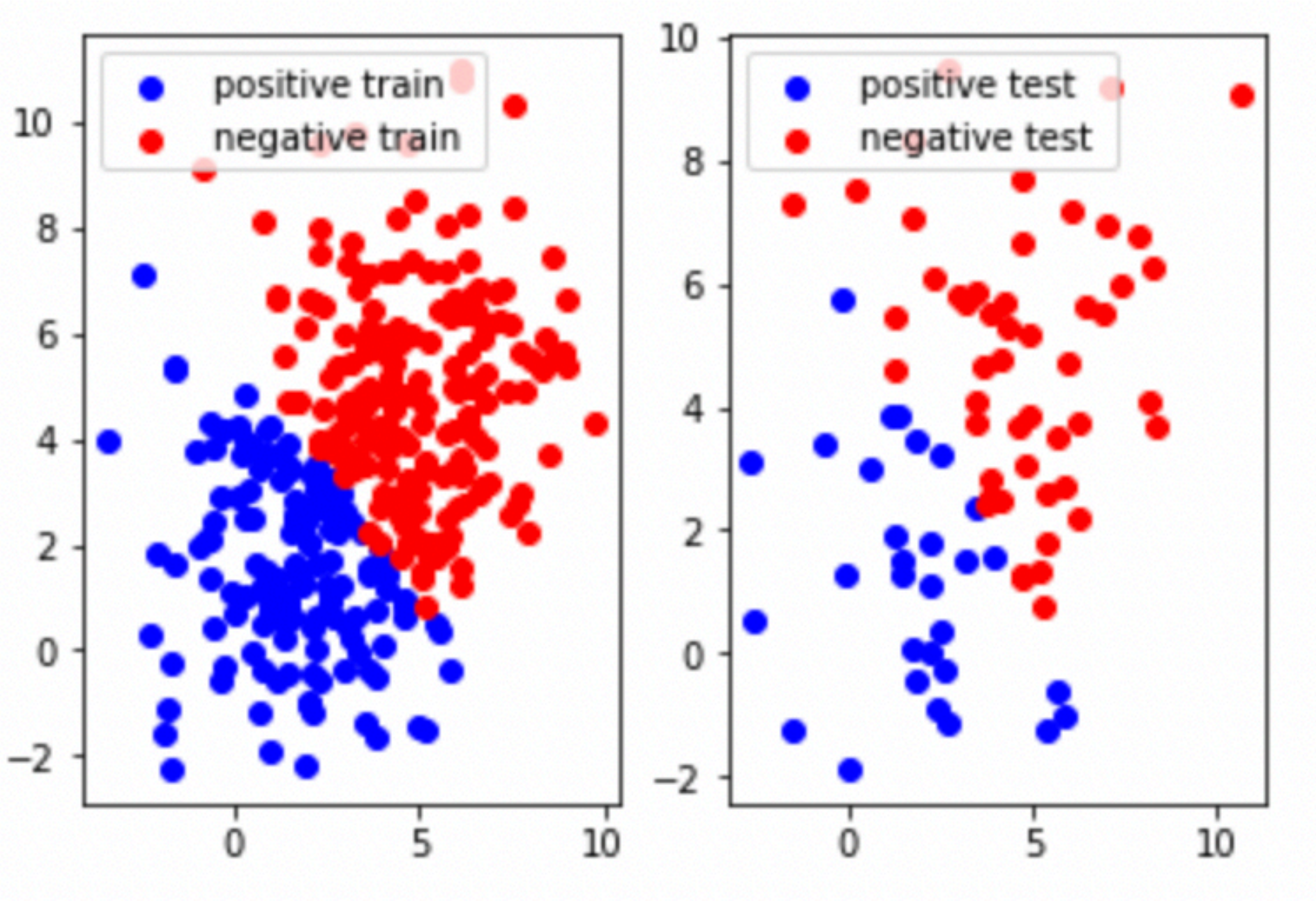
Task 2: Implement Perceptron.py

In task2, we implemented the perceptron following the specs in perceptron.py.

- Impliment the initiation function of class perceptron using parameters: n_inputs, max_epochs, learning_rate. Initiate the weight of the perceptron using all 0.
- Implement the traning method of the perceptron.
    - Use the weight we currently calculated to predict training input data.
    - If the prediction is different from its label, then we adjust our weight vector by adding the multiplication of the leraning rate and the difference of the label and the prediction.
    - Continuing training all input training data points for max_epochs' time. Recording the training and testing accuracy in each epoch and return its list.
- Implement the forward functions of the perceptron by calculating the sign of np.dot(self.weight, np.array([1, *input])).

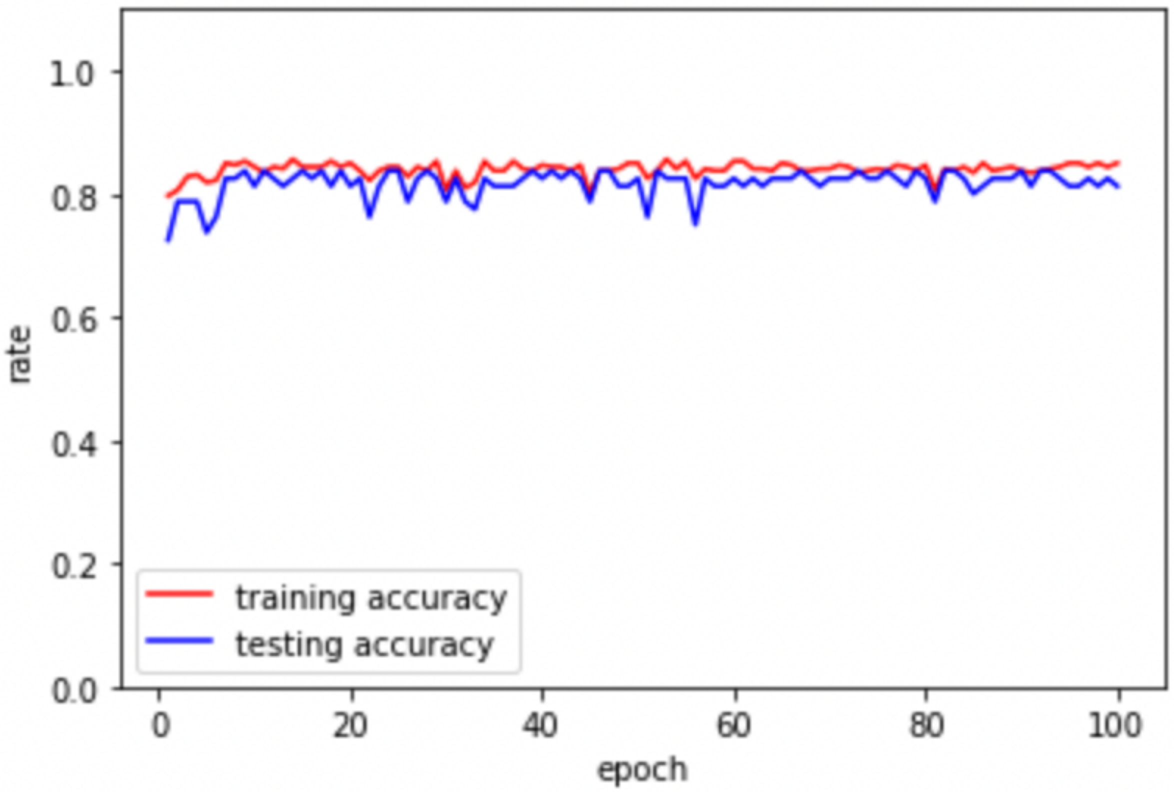
Task 3: Train and Calculate Accuracy

In task 3, I trained the perceptron on the training data (320 points) and test in on the remaining 80 test points. The prediction on training set and testing set are shown as below.



- We can see intuitively training set and testing set are mostly separated correctly. We can see the classification hyperplane of our perceptron are linear in the two dimensional space.

The accuracy of the perceptron calculated while training are ploted as below:



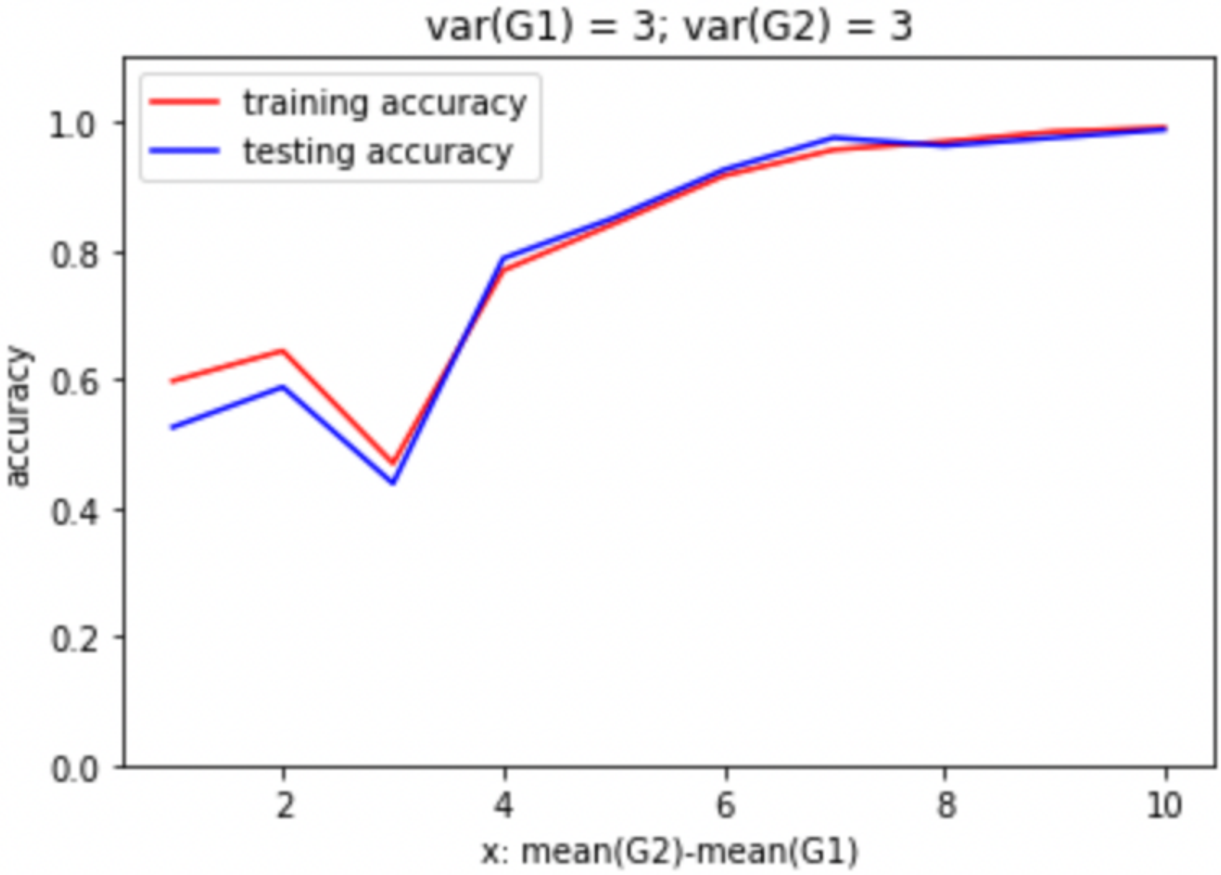
```
Final test accuracy: 0.8125
```

- We can see both of the training accuracy and testing accuracy quickly coverge to approximately 0.8 with the epoch increasing.
- The final test accuracy fianally reached 0.8125 with the default argument input. The perceptron can get much higher accuracy if we define other distribution.

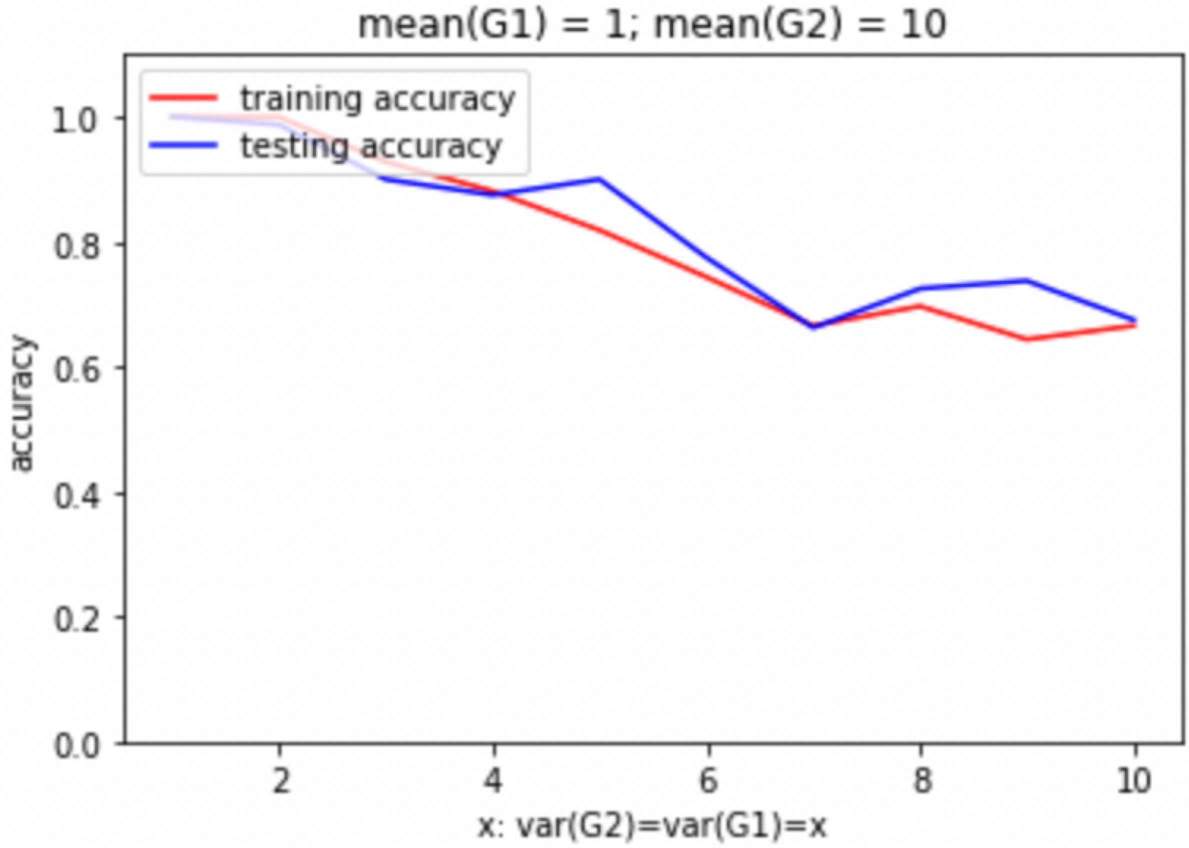
Task 4: Analyze Means and Variances

During repeated test, I find that:

- With the difference of the two means of Gaussian Distribution increasing, the accuracy of the perceptron increases.



- With the variances of the Gaussian distribution increasing, the accuracy of perceptron decreases.



Conclusion:

- the closer of the means of two Gaussian Distribution, the higher the confusion level of the data is.
- the bigger of the variance, the higher the confusion level of the data is.
- More data points are mixed in our two-dimensional space when two means are too close or when the variances are big enough, which means it is more inaccurate for a linear perceptron to be classified.

Part 2: The Mutli-layer Perceptron

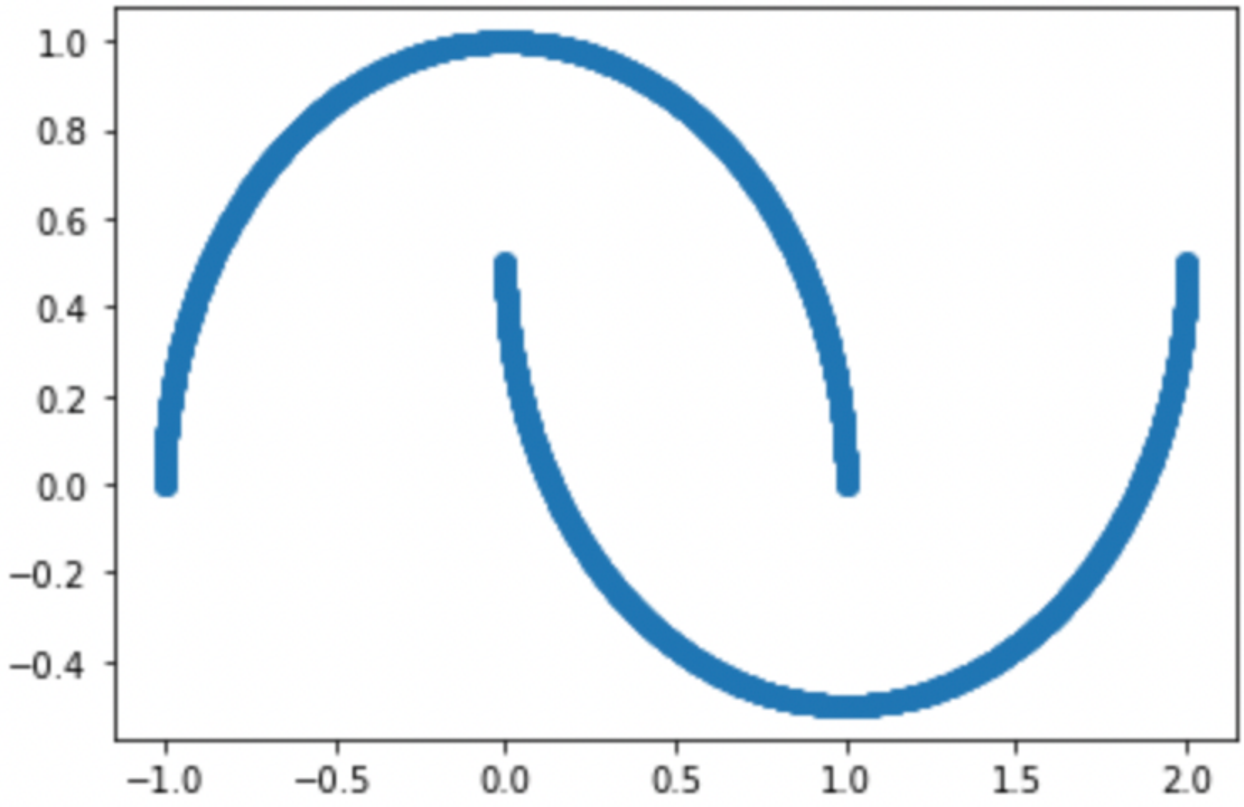
Task 1: Implement module.py, mlp_numpy.py

- In `module.py`, I implement **Linear** (the perceptron):
  - **Implement initialization function**: Initialize weight using Gaussian distribution. Initialize bias using 0. Initialize x as None to record the input data points.
  - **Implement forward function**: return the dot production of weight and x, which means convert data x into a new dimension space. At the same time recordthe input x in self.x.
  - **Implement backward function**: Using the input dout, calculate the gradient of bias (dout) and the graadient of weight (dot product of x and dout), record it the Linear object. Return the gradient of dx.
  - **Implement train function**: Update the weight matrix and bias vectors using the gradient recorded in the object. Then reset the gradient.
  - Implement the forward function **ReLu, SoftMax, CrossExtropy** according to their formula definition. Implement their backward function using the gradients.

- In `mlp_numpy.py`, I implement **MLP**:
  - **Implement initialization function**: Initialize a list of Linears representing all hidden layer and output layer whose shapes are defined by input parameters. Initialize list of reLu for each layers.
  - **Implement forward method**: Let input vector x go through forward methods of all hidden layers (Linear.forward and ReLu.forward), finally input x into output layer (Linear) and return the SoftMax(out).
  - **Implement backward method**: Let input vector dout go backward of output layers and hidden layers in inverse order.
  - **Implement train method**: let all hidden layers and the output layer to update their weight and bias.

Task 2: Generate Dataset

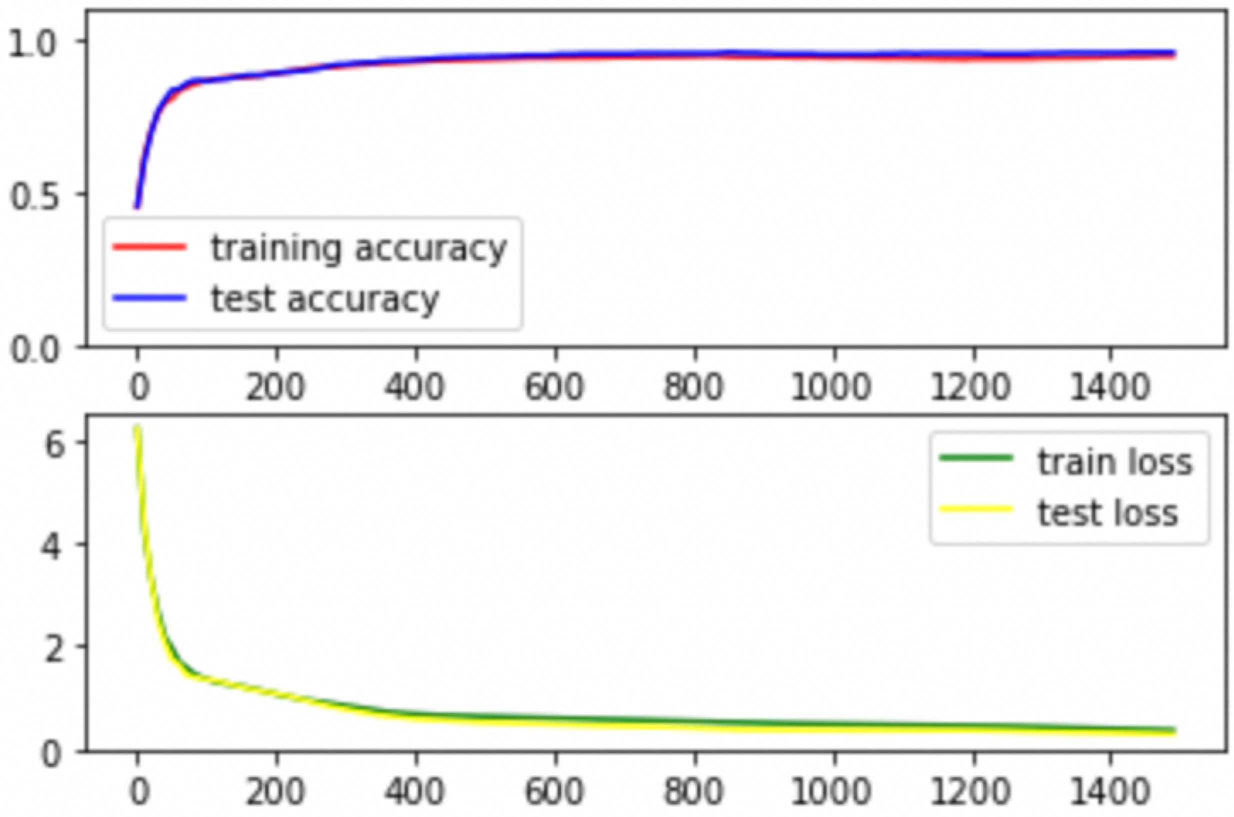
- Generate shuffled dataset using `make_moon` method in sklearn package and sample 2000 samples. The data points are ploted as follows.



- Split the dataset into 70% training set and 30% testing set.

Task 3: Batch gradient Descent, Train, Analysis

- Implement batch gradient descent, which means update the accumulated weight and bias of all layers each epoch.
- Use default parameters and train the mlp in batch gradient descent, we plot the training accuracy, training loss, testing accuracy, testing loss as follows.



- According to the training accuracy and testing accuracy with epoch as the x-axis. We find that both the training accuracy and testing accuracy gradually approach 1.
- According to the training loss and testing loss with epoch as the x-axis. We find that both the training loss and testing loss gradually approach 0.

Part 3: Stochastic Gradient Descent

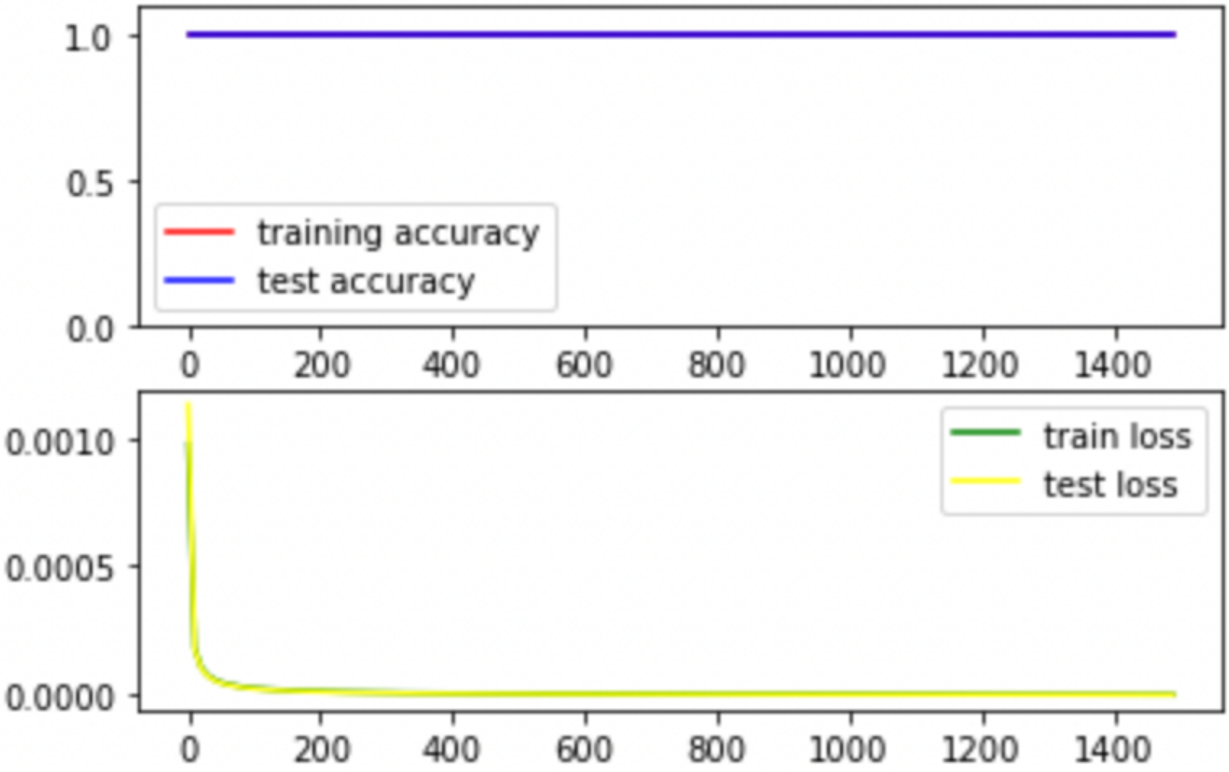
Task 1: Implement Stochastic Gradient Descent

- Implement stochastic gradient descent, which means update the weight and bias of all layers after training each single data point x.

Task 2: Train, Analysis

- We repeat the training process of MLP with the same dataset while using stochastic gradient descent, which means we update our weights in Linear layers of MLP every time we forward a new data point, rather than update weights after all data in the dataset is calculated.

- The graph of training and testing accuracy and loss is also shown below.



Compared with the MLP using batch gradient descent with the same dataset, we find that:

- the accuracy and loss reach the same value after enough epoch
    - accuracy: 1
    - loss: 0
- MLP prediction coverage quickly when using stochastic gradient descent than using batch gradient descent

done