CS324 Assignemnt 1 Report

SID：12011725

Name：彭英智

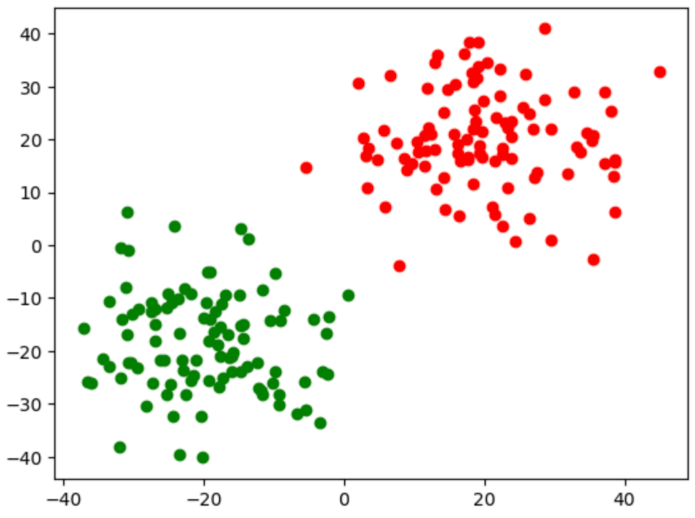
Part I: the perceptron (20 points)

Task 1

Use these code to generate the dataset.

```
gd_1 = torch.normal(20, 10, (100,2))
gd_2 = torch.normal(-20, 10, (100,2))
# swich to numpy array
gd_1_n = gd_1.numpy()
gd_2_n = gd_2.numpy()
# extract the point importation
gd_1_n_x = gd_1_n[:,0]
gd_1_n_y = gd_1_n[:,1]
gd_2_n_x = gd_2_n[:,0]
gd_2_n_y = gd_2_n[:,1]
# draw graph
plt.scatter(gd_1_n_x, gd_1_n_y, c='r')
plt.scatter(gd_2_n_x, gd_2_n_y, c='g')
```

```
Out[8]: <matplotlib.collections.PathCollection at 0x12fc0d9d0>
```



Task 2 & Task 3

Use these code to generate train / test data

```
gd_1_n = np.insert(gd_1_n, 2, 1, axis=1)
gd_2_n = np.insert(gd_2_n, 2, -1, axis=1)
gd_n_comb = np.concatenate((gd_1_n, gd_2_n), axis=0)
np.random.seed(116)
np.random.shuffle(gd_n_comb)
# 80% training 20% testing
gd_n_learn_input = gd_n_comb[:160,:2]
gd_n_learn_label = gd_n_comb[:160,-1]
gd_n_test_input = gd_n_comb[160:,:2]
gd_n_test_label = gd_n_comb[160:,-1]
```

The code of perceptron is shown below.

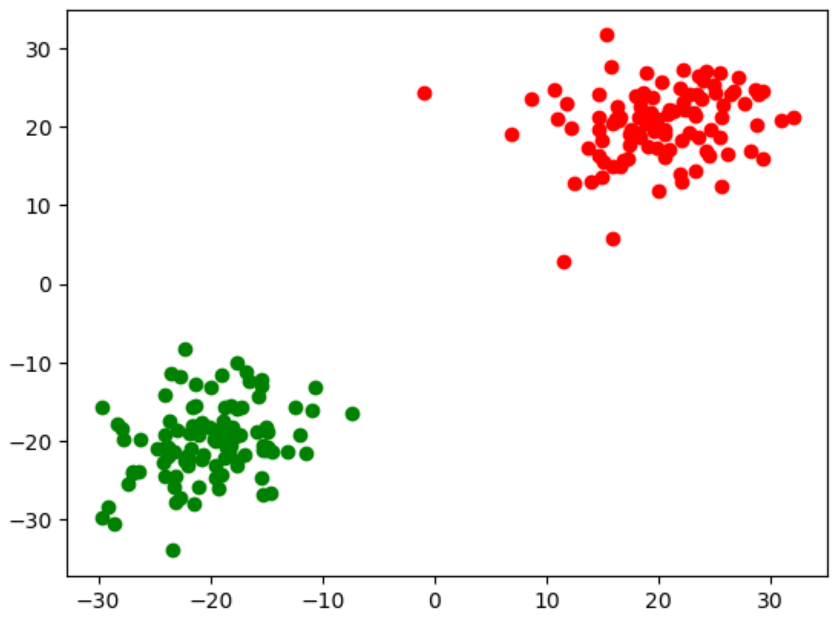
```
In [5]: class Perceptron(object):

            def __init__(self, n_inputs, max_epochs=1e2, learning_rate=1e-2):
                """
                Initializes perceptron object.
                Args:
                    n_inputs: number of inputs.
                    max_epochs: maximum number of training cycles.
                    learning_rate: magnitude of weight changes at each training cycle
                """
                self.weight = np.zeros([1, n_inputs])
                self.max_epochs = max_epochs
                self.learning_rate = learning_rate

            def forward(self, input):
                """
                Predict label fropm input
                Args:
                    input: array of dimension equal to n_inputs.
                """
                label = np.matmul(self.weight, input)
                label = np.where(label > 0, 1, -1)
                return label

            def train(self, training_inputs, labels):
                """
                Train the perceptron
                Args:
                    training_inputs: list of numpy arrays of training points.
                    labels: arrays of expected output value for the corresponding point in training_inputs.
                """
                for i in range(int(self.max_epochs)):
                    error = 0
                    for idx in range(training_inputs.shape[0]):
                        if labels[idx] * self.forward(training_inputs[idx]) < 0:
                            error = error + 1
                            self.weight = self.weight + self.learning_rate * labels[idx] * training_inputs[idx]
                    print("accuracy rate of ", i, " training turns is ", 1.0 - error / training_inputs.shape[0])
```

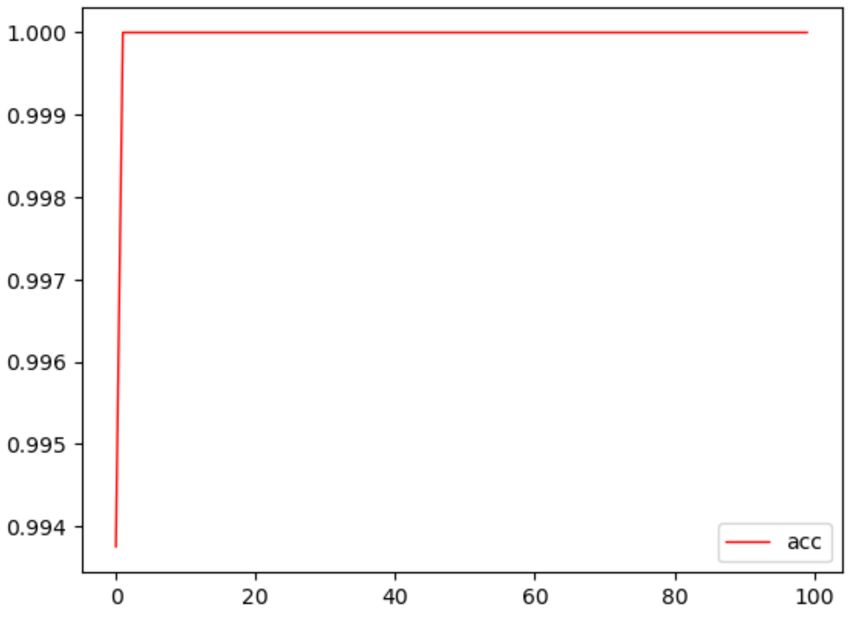
Task 4

The follwing graphs are the distribution graphs of points and the train_accuracy - epoch graphs.

CASE 1 Too Far

```
gd_1_n = np.insert(gd_1_n, 2, 1, axis=1)
gd_2_n = np.insert(gd_2_n, 2, -1, axis=1)
```



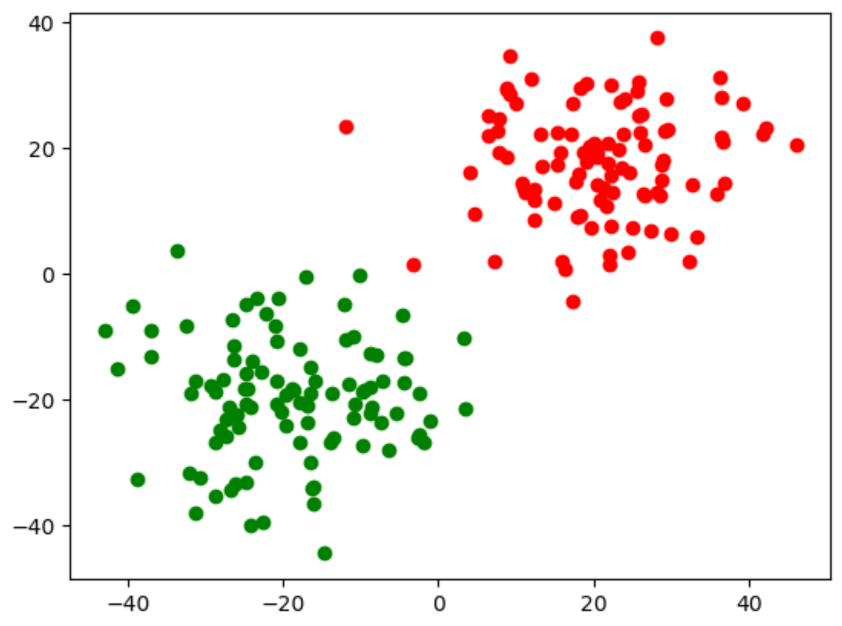


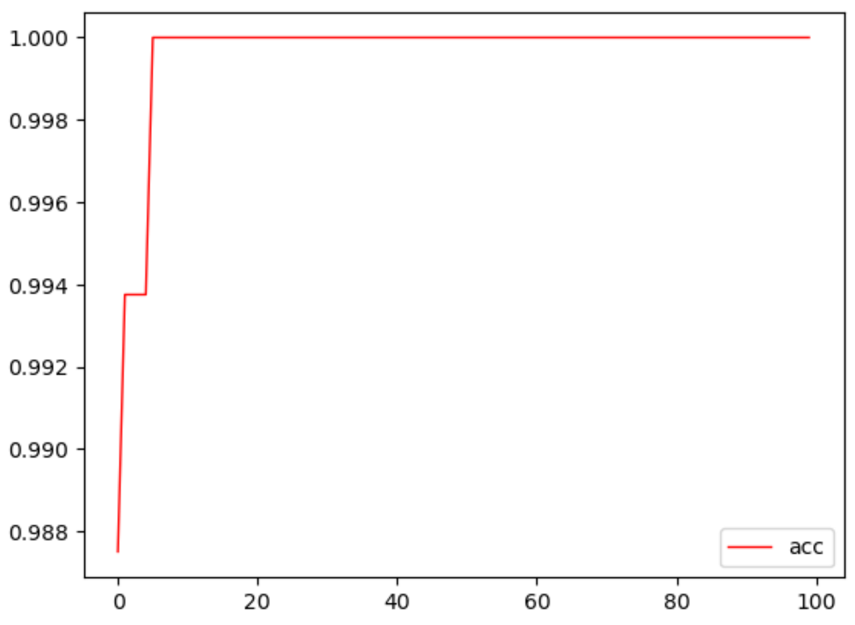
The final test accuracy rate is 1.0

Too early to get convergence

CASE 2

```
gd_1 = torch.normal(20, 10, (100,2))
gd_2 = torch.normal(-20, 10, (100,2))
```

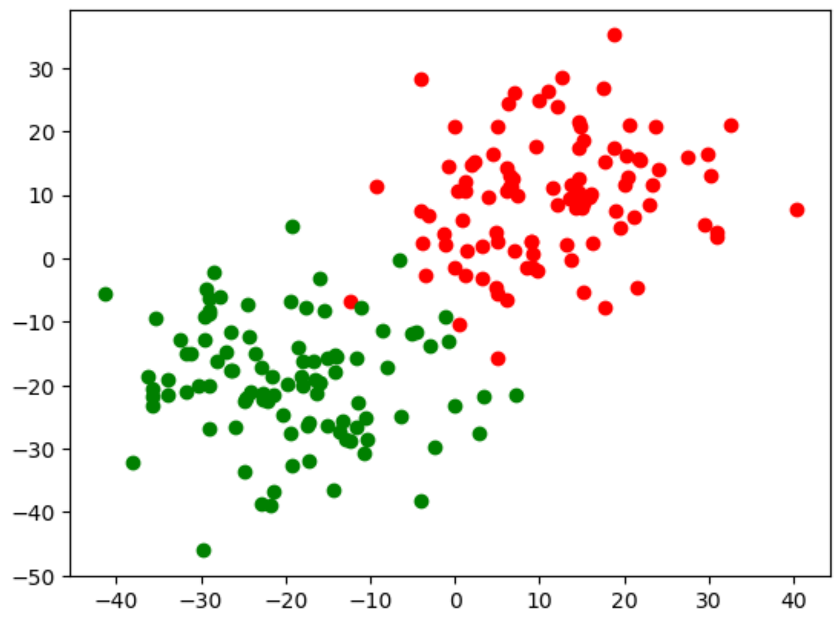


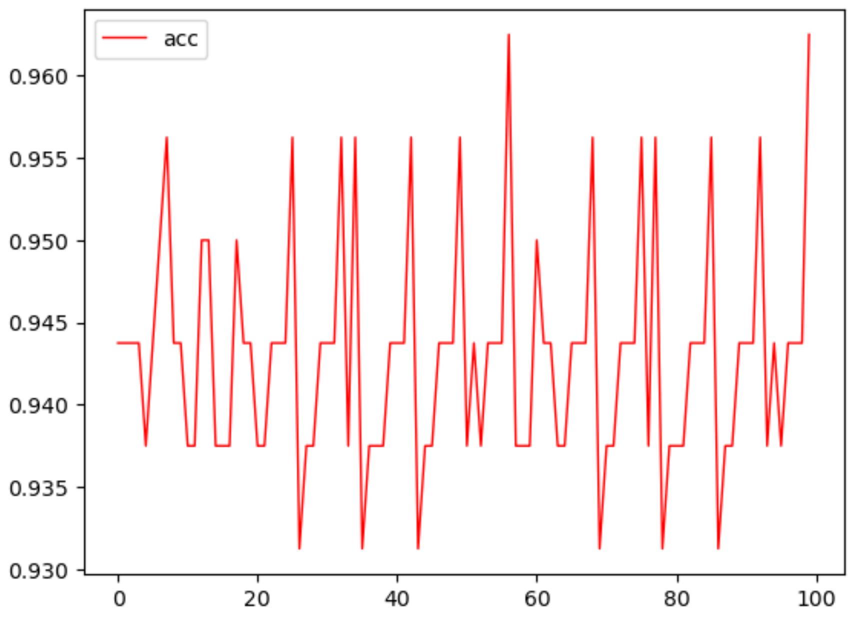


The final test accuracy rate is 1.0

CASE 3

```
gd_1 = torch.normal(10, 10, (100,2))
gd_2 = torch.normal(-20, 10, (100,2))
```



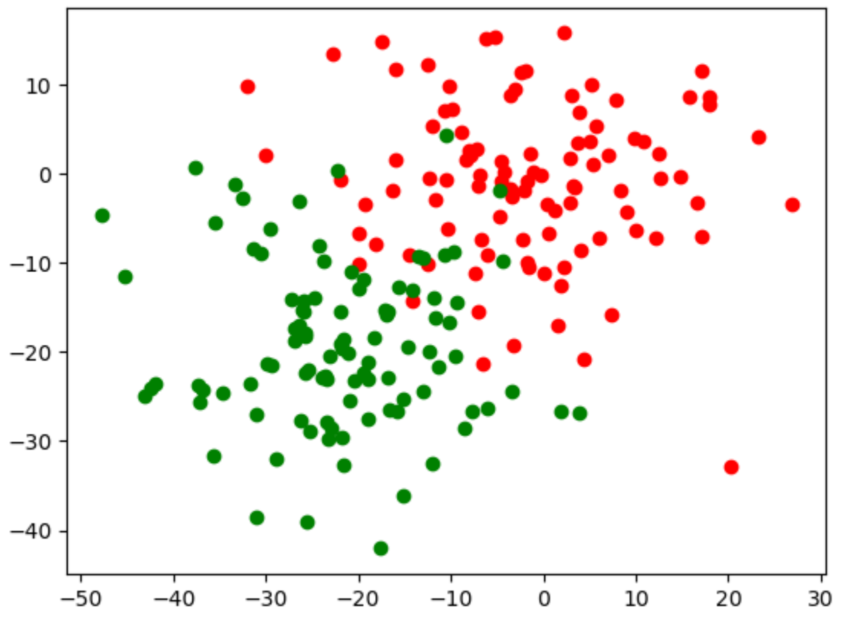


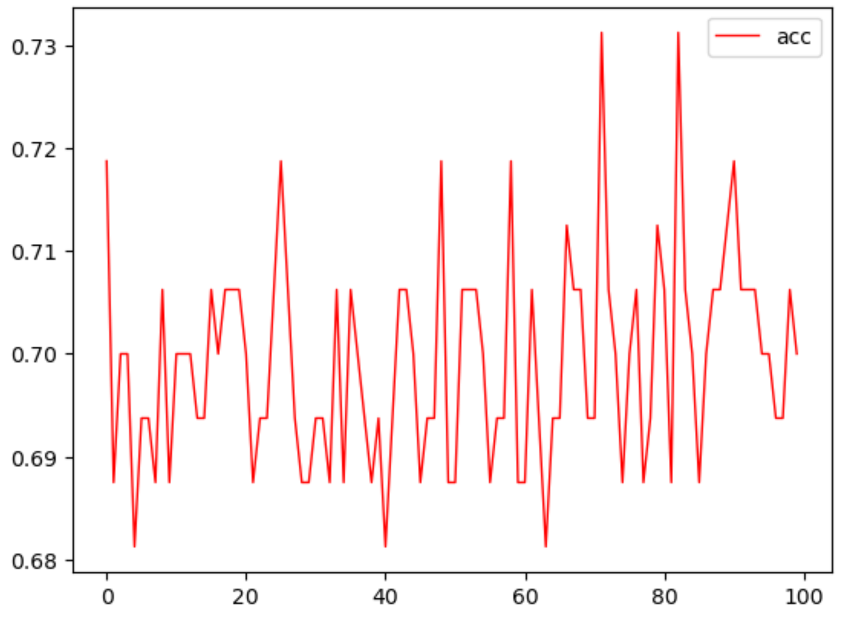
The final test accuracy rate is 0.875

Can't get convergence

CASE 3 Too Close

```
gd_1 = torch.normal(0, 10, (100,2))
gd_2 = torch.normal(-20, 10, (100,2))
```





The final test accuracy rate is 0.525

Can't get convergence

Part II: the mutli-layer perceptron (65 points)

2-1 modules.py

2-1-1 Weight Initialize Method

use two method to initialize the origin weight of cell.

```
def kaiming(m, h):
    return np.random.randn(m, h) * math.sqrt(2. / m)

def normal(loc, scale, in_features, out_features):
    return np.random.normal(loc=loc, scale=scale, size=(in_features, out_features))
```

2-1-2 Linear

```
class Linear(object):
    def __init__(self, in_features, out_features):
        ...
        # two ways to initialize weight
        # self.params['weight'] = normal(0.0, 0.001, in_features, out_features)
        self.params['weight'] = kaiming(in_features, out_features)
        self.params['bias'] = 0

    def forward(self, x):
        self.input = x
        self.output = np.matmul(x, self.params['weight']) + self.params['bias']
        return self.output

    def backward(self, dout):
        self.batch_size = dout.shape[0]
        self.dw = np.mean(np.matmul(self.input.transpose(0, 2, 1), dout), axis=0)  # δw
= δg * x
        self.db = np.mean(dout, axis=0)
        grad = np.matmul(dout, self.params['weight'].T)
        return grad

    # update weight and bias
    def update(self):
        self.params['weight'] = self.params['weight'] - self.dw * LEARNING_RATE_DEFAULT
        self.params['bias'] = self.params['bias'] - self.db * LEARNING_RATE_DEFAULT
        return
```

2-1-3 ReLU

```
class ReLU(object):
    def __init__(self):
        self.out = None

    def forward(self, x):
        self.out = np.where(x > 0, x, 0)
        return self.out

    def backward(self, dout):
        dout_t = torch.tensor(dout)
        out_t = torch.tensor(self.out)

        idx = torch.where(out_t > 0)
        jd = torch.zeros_like(out_t)
        jd[idx] = 1
        j = torch.diag_embed(jd).squeeze()
        assert dout_t.size(-1) == j.size(-2)
        dx_t = torch.matmul(dout_t, j)
        dx = dx_t.numpy()
        return dx

    def update(self):
        return
```

2-1-4 SoftMax & CrossEntropy

Since combining softmax and crossentropy to calculate gradient is much more earlier, so I combine the **backward** method together.

```
class SoftMax(object):
    def __init__(self):
        self.output = None

    def forward(self, x):
        x_max = np.max(x, -1)[:, np.newaxis]
        x = x - x_max
        exp = np.exp(x)
        exp_sum = np.sum(np.exp(x), -1)[:, np.newaxis]
        return exp / exp_sum

    #do nothing just transfer the gradient
    def backward(self, dout):
        return dout

    def update(self):
        return


class CrossEntropy(object):
    def __init__(self):
        self.output = None

    def forward(self, x, y):
        out = -np.sum(y * np.log(x), -1)
        self.output = out
        return x, out

    def backward(self, x, y):
        batch_size = x.shape[0]
        dx = (x - y)
        return dx

    def update(self):
        return
```

2-1 mlp_numpy.py

Initialization of MLP, generating the input-layer, hidden-layer and output-layer

```
def __init__(self, n_inputs, n_hidden, n_classes):
    self.layers = []
    for index in range(len(n_hidden)):
        if index == 0:
            self.layers.append(Linear(n_inputs, n_hidden[index]))
            self.layers.append(ReLU())
        if index == len(n_hidden)-1:
            self.layers.append(Linear(n_hidden[index], n_classes))
        else:
            self.layers.append(Linear(n_hidden[index], n_hidden[index+1]))
            self.layers.append(ReLU())
    self.layers.append(SoftMax())
```

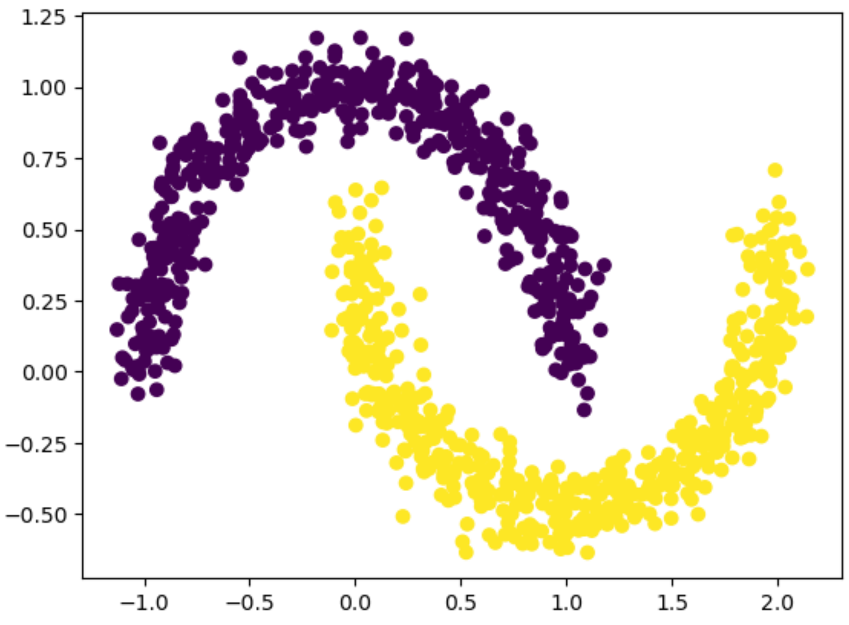
forward and backward

```
def forward(self, x):
    for layer in self.layers:
        x = layer.forward(x)
    out = x
    return out

def backward(self, dout):
    for layer in reversed(self.layers):
        dout = layer.backward(dout)
        layer.update()
    return
```

2-3 Experiment & Analyse

2-3-1 Data Generation

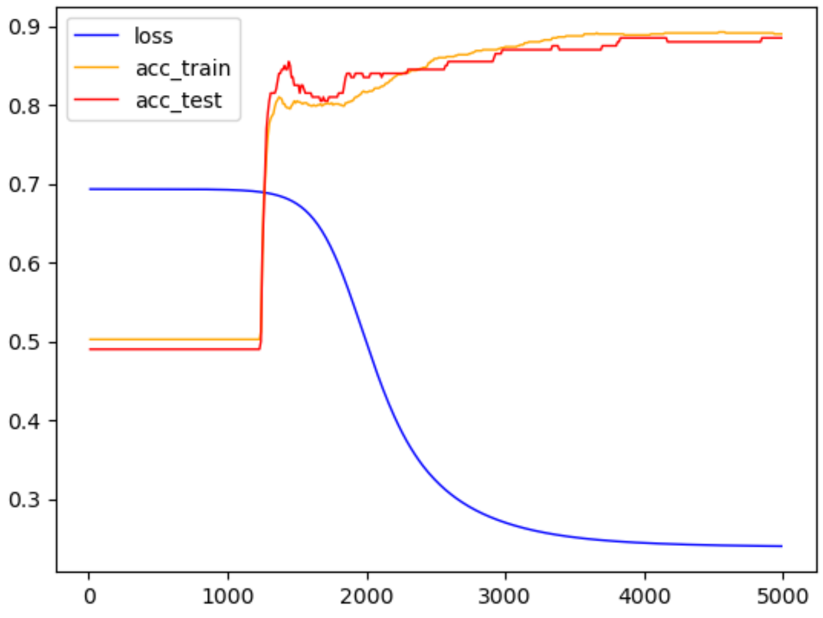
```
x, y = datasets.make_moons(n_samples=1000, noise=0.08, shuffle=True, random_state=None)
x_input = np.expand_dims(x, axis=1)
y_onehot = np.eye(2)[y]
y_onehot = np.expand_dims(y_onehot, axis=1)

x_input_train = x_input[:800,:,:]
y_onehot_train = y_onehot[:800,:]
x_input_test = x_input[800:,:,:]
y_onehot_test = y_onehot[800:,:]
```



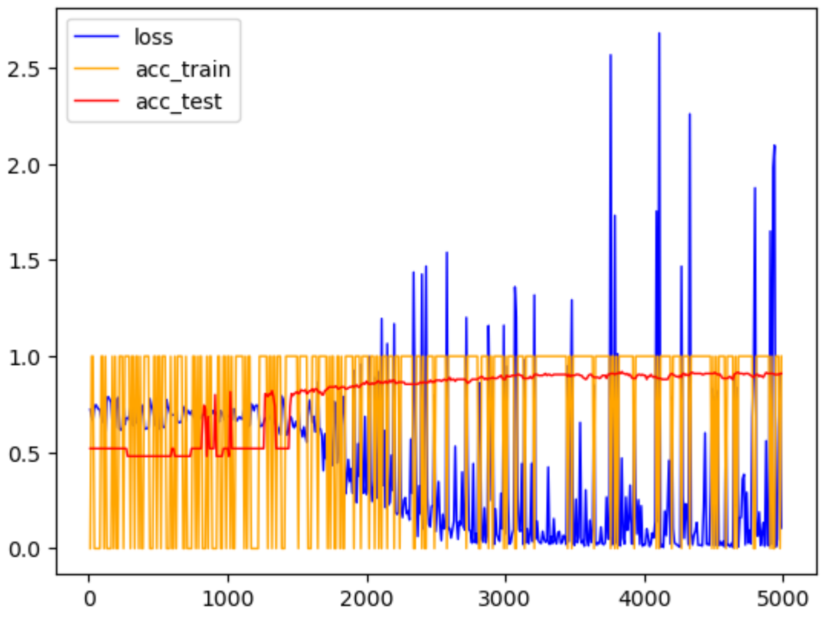
2-3-2 BGD & SGD & Mini-Batch

```
DNN_HIDDEN_UNITS_DEFAULT = [20]
LEARNING_RATE_DEFAULT = 0.01
MAX_EPOCHS_DEFAULT = 5000
EVAL_FREQ_DEFAULT = 10
```

BGD

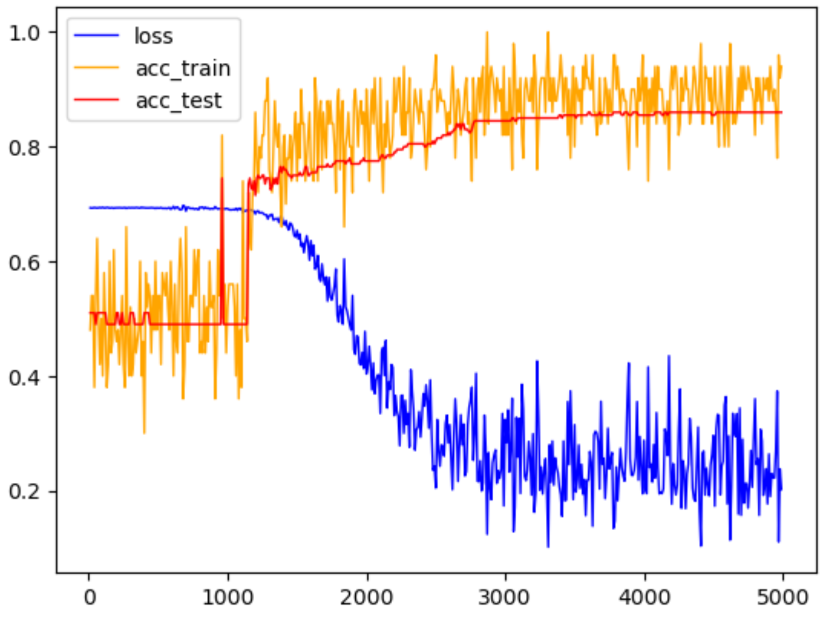


SGD

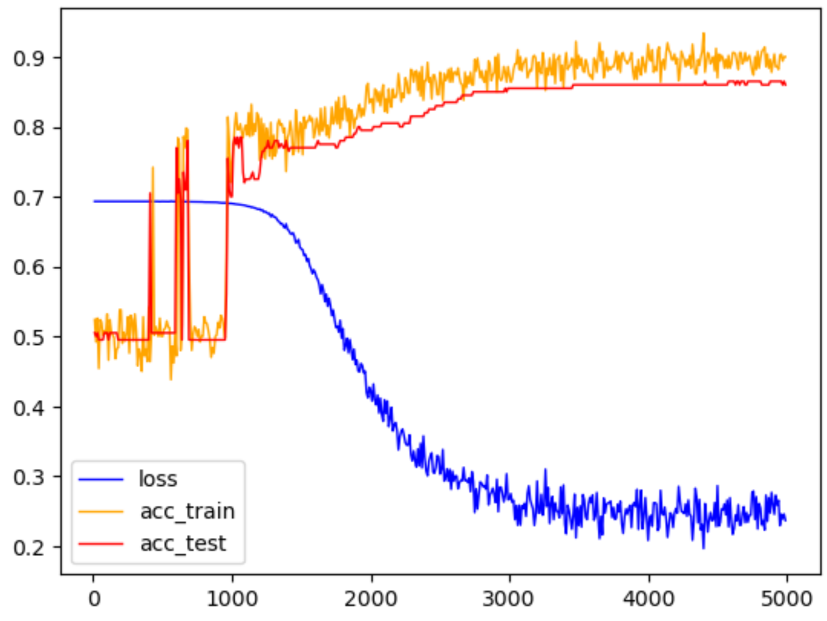


Mini-Batch

Batch_Size = 50



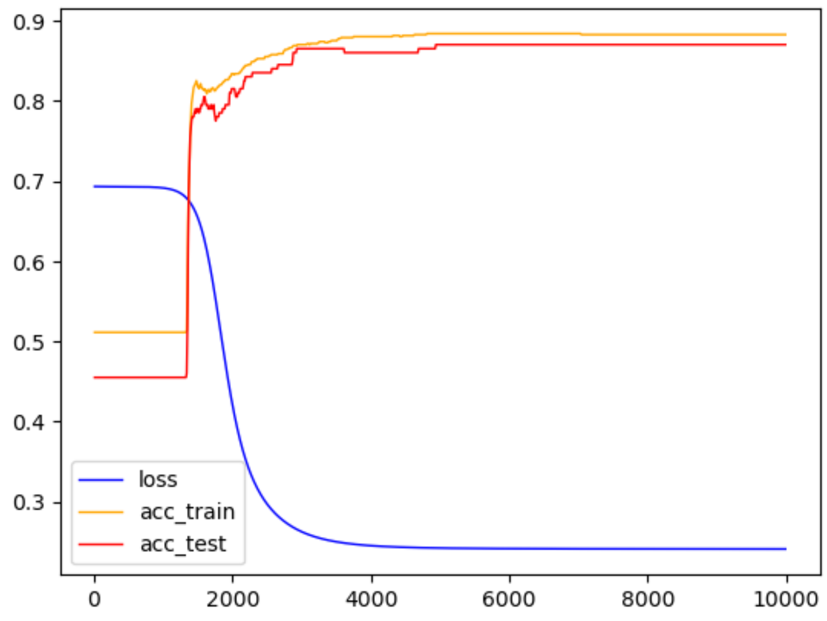
Batch_Size = 500



2-3-3 Analyse of different hidden layer

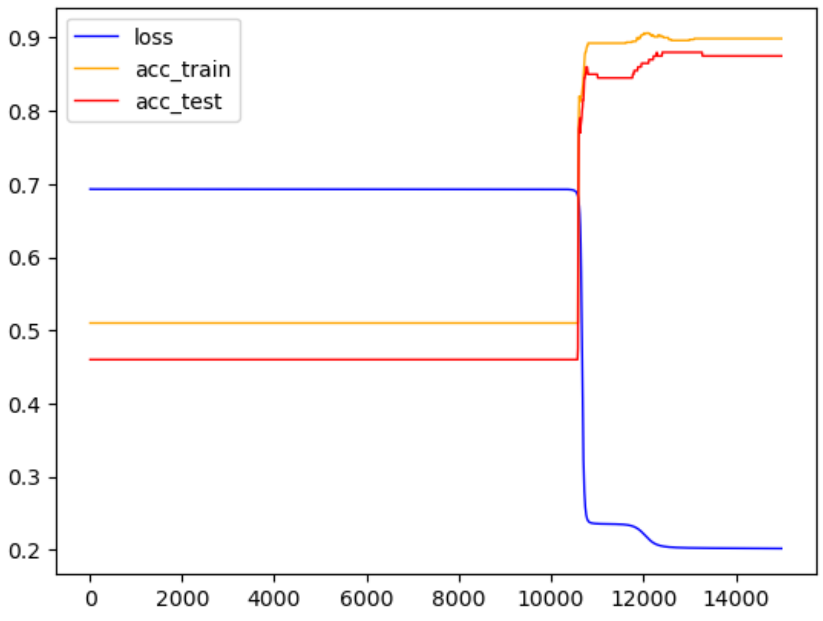
One Hidden Layer (BGD)

```
DNN_HIDDEN_UNITS_DEFAULT = [20]
LEARNING_RATE_DEFAULT = 1e-2
MAX_EPOCHS_DEFAULT = 10000
Initialization Method: Normal
```



Two Hidden Layers (BGD)

```
DNN_HIDDEN_UNITS_DEFAULT = [20, 20]
LEARNING_RATE_DEFAULT = 1e-1
MAX_EPOCHS_DEFAULT = 15000
Initialization Method: Normal
```

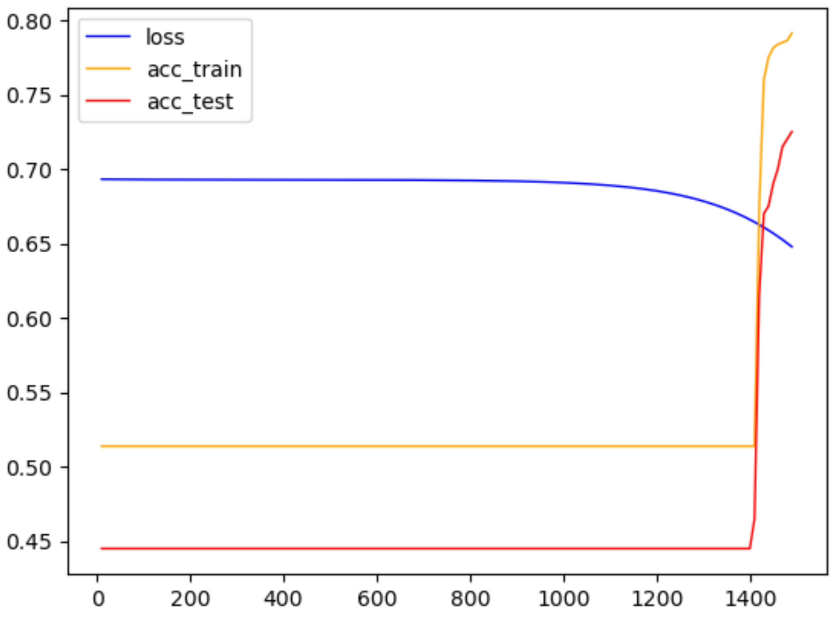


more hidden layers needs more training epoch.

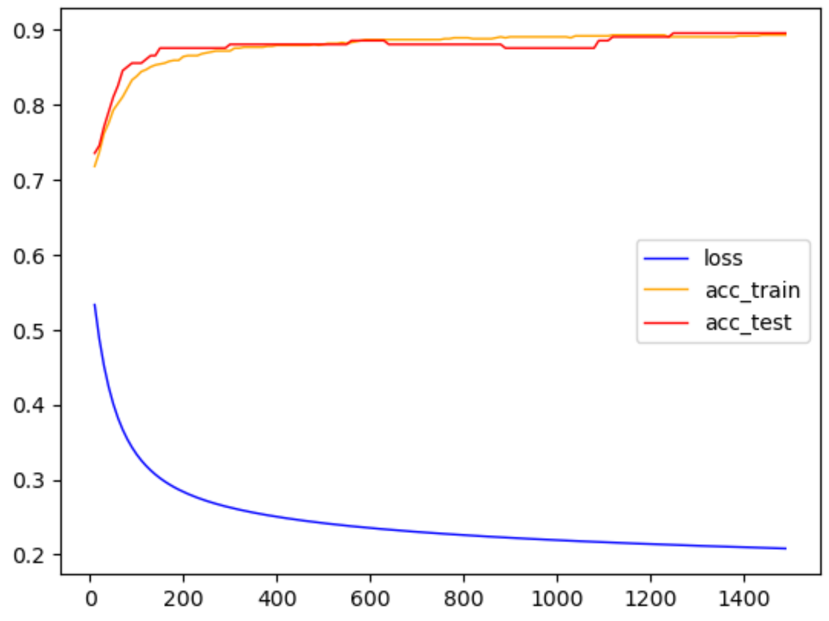
2-3-4 Analyse of different Initialize Method of w

```
DNN_HIDDEN_UNITS_DEFAULT = [20]
LEARNING_RATE_DEFAULT = 0.01
MAX_EPOCHS_DEFAULT = 1500
EVAL_FREQ_DEFAULT = 10
```

Using normalize to initial weights, in 1500 steps the model can't get convergence. (BGD)

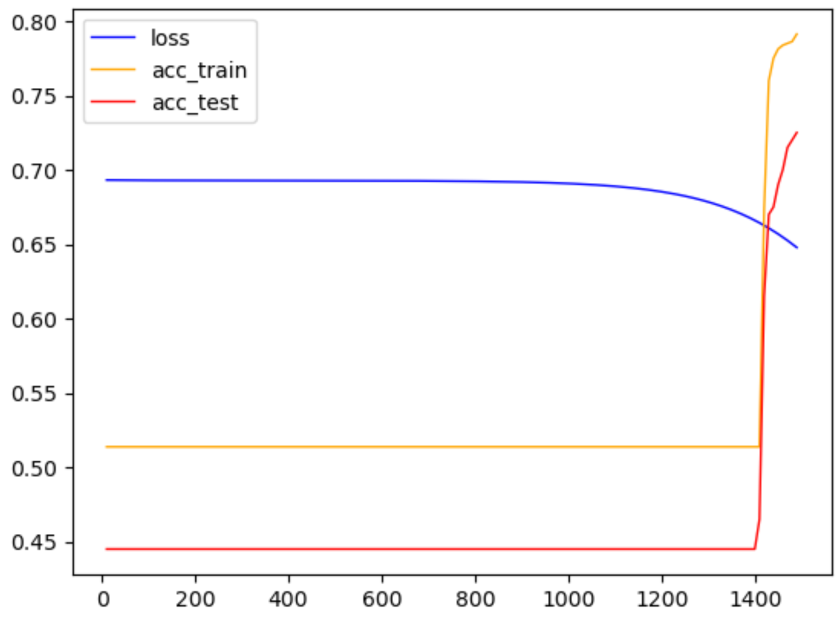


Using kaiming method, the model can quickly get convergence. (BGD)

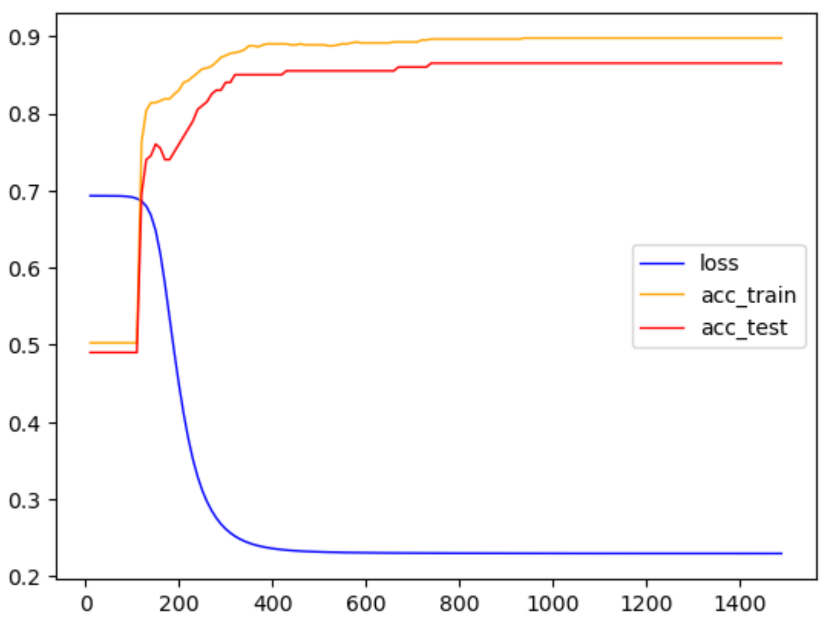


2-3-5 Analyse of different Learning rate

Default Learing Rate = 1e-2



Learing Rate = 1e-1



When learing rate increase, the model is easier to get convergence.