# Assignment1

Name: Yubin Hu

ID: 11712121

# 1 Part I: the perceptron

## Code

In [1]:
```
import numpy as np
import matplotlib.pyplot as plt

def gen_gaussian_distribution(size, mean=None, cov=None):
    if not mean:
        mean = np.random.randn(2)
    if not cov:
        cov = np.eye(2)
    data = np.random.multivariate_normal(mean, cov, size)
    return data

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.n_inputs = n_inputs
        self.max_epochs = max_epochs
        self.learning_rate = learning_rate
        self.weights = np.zeros(self.n_inputs)
        self.bias = 0


    def forward(self, input):
        """
        Predict label from input
        Args:
            input: array of dimension equal to n_inputs.
        """
        sum = np.sign(np.dot(input, self.weights))
        label = np.where(sum > 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 poi
        """
        train_size = len(training_inputs)
        epochs = 0
        while epochs < self.max_epochs:
            epochs += 1
```

```
            for i in range(train_size):
                if np.any(labels[i] * (np.dot(self.weights, training_inputs[i
                    self.weights = self.weights + (self.learning_rate * label
                    self.bias = self.bias + self.learning_rate * labels[i]

    def score(self, test_inputs, test_labels):
        pred_arr = np.where(self.forward(test_inputs) > 0, 1, -1)
        true_size = len(np.where(pred_arr == test_labels)[0])
        return true_size / len(test_labels)


def main():
    p = Perceptron(2)

    """
    gen dataset
    """
    data_size = 100
    train_size = 80
    x1 = gen_gaussian_distribution(data_size, [5, 5])
    x2 = gen_gaussian_distribution(data_size, [-5, -5])
    y1 = a_label = np.ones(data_size, dtype=np.int16)
    y2 = -y1
    x_train = np.concatenate((x1[:train_size], x2[:train_size]), axis=0)
    y_train = np.concatenate((y1[:train_size], y2[:train_size]), axis=0)
    x_test = np.concatenate((x1[train_size:], x2[train_size:]), axis=0)
    y_test = np.concatenate((y1[train_size:], y2[train_size:]), axis=0)

    """
    train model
    """
    p.train(x_train, y_train)

    """
    test model
    """
    acc = p.score(x_test, y_test)


if __name__ == "__main__":
    main()
```

## 1.1 Task 1

> Generate a dataset of points in R2. To do this, define two Gaussian distributions and sample 100 points from each. Your dataset should then contain a total of 200 points, 100 from each distribution. Keep 80 points per distribution as the training (160 in total), 20 for the test (40 in total).

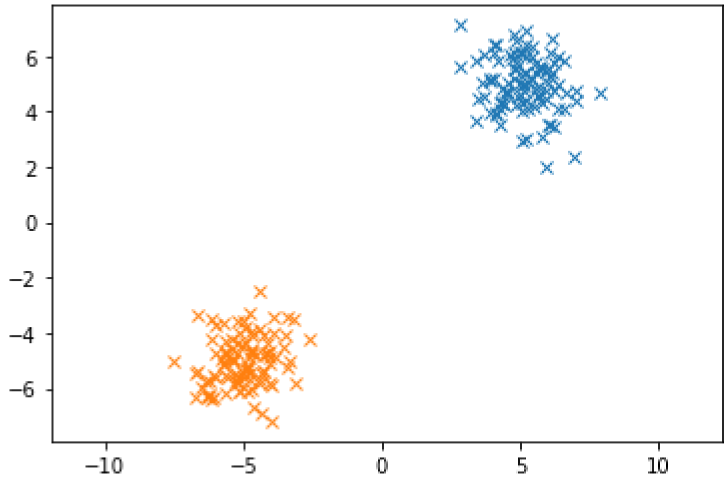
In [2]:

```
"""
gen dataset
"""
data_size = 100
train_size = 80
x1 = gen_gaussian_distribution(data_size, [5, 5])
x2 = gen_gaussian_distribution(data_size, [-5, -5])
y1 = a_label = np.ones(data_size, dtype=np.int16)
y2 = -y1
x_train = np.concatenate((x1[:train_size], x2[:train_size]), axis=0)
y_train = np.concatenate((y1[:train_size], y2[:train_size]), axis=0)
x_test = np.concatenate((x1[train_size:], x2[train_size:]), axis=0)
```

```
y_test = np.concatenate((y1[train_size:], y2[train_size:]), axis=0)


# plt
plt.plot(x1[:,0], x1[:,1], 'x')
plt.plot(x2[:,0], x2[:,1], 'x')
plt.axis('equal')
plt.savefig('./img/fig1.png')
plt.show()
```

```
---------------------------------------------------------------------------
FileNotFoundError                         Traceback (most recent call last)
<ipython-input-2-6ff8b82d0010> in <module>
     18 plt.plot(x2[:,0], x2[:,1], 'x')
     19 plt.axis('equal')
---> 20 plt.savefig('./img/fig1.png')
     21 plt.show()

~/.local/lib/python3.7/site-packages/matplotlib/pyplot.py in savefig(*args, **
kwargs)
    857 def savefig(*args, **kwargs):
    858     fig = gcf()
--> 859     res = fig.savefig(*args, **kwargs)
    860     fig.canvas.draw_idle()   # need this if 'transparent=True' to rese
t colors
    861     return res

~/.local/lib/python3.7/site-packages/matplotlib/figure.py in savefig(self, fna
me, transparent, **kwargs)
   2309                 patch.set_edgecolor('none')
   2310 
-> 2311         self.canvas.print_figure(fname, **kwargs)
   2312 
   2313         if transparent:

~/.local/lib/python3.7/site-packages/matplotlib/backend_bases.py in print_figu
re(self, filename, dpi, facecolor, edgecolor, orientation, format, bbox_inche
s, pad_inches, bbox_extra_artists, backend, **kwargs)
   2215                     orientation=orientation,
   2216                     bbox_inches_restore=_bbox_inches_restore,
-> 2217                     **kwargs)
   2218             finally:
   2219                 if bbox_inches and restore_bbox:

~/.local/lib/python3.7/site-packages/matplotlib/backend_bases.py in wrapper(*a
rgs, **kwargs)
   1637             kwargs.pop(arg)
   1638 
-> 1639         return func(*args, **kwargs)
   1640 
   1641     return wrapper

~/.local/lib/python3.7/site-packages/matplotlib/backends/backend_agg.py in pri
nt_png(self, filename_or_obj, metadata, pil_kwargs, *args)
    510         mpl.image.imsave(
    511             filename_or_obj, self.buffer_rgba(), format="png", origin=
"upper",
--> 512             dpi=self.figure.dpi, metadata=metadata, pil_kwargs=pil_kwa
rgs)
    513 
    514     def print_to_buffer(self):

~/.local/lib/python3.7/site-packages/matplotlib/image.py in imsave(fname, arr,
vmin, vmax, cmap, format, origin, dpi, metadata, pil_kwargs)
   1609         pil_kwargs.setdefault("format", format)
   1610         pil_kwargs.setdefault("dpi", (dpi, dpi))
-> 1611         image.save(fname, **pil_kwargs)
   1612 
```

```
   1613

~/.local/lib/python3.7/site-packages/PIL/Image.py in save(self, fp, format, **
params)
   2159                 fp = builtins.open(filename, "r+b")
   2160             else:
-> 2161                 fp = builtins.open(filename, "w+b")
   2162
   2163         try:

FileNotFoundError: [Errno 2] No such file or directory: './img/fig1.png'
```




We set `mean1 = [5, 5]` and `mean2 = [−5, −5]` , generate `cov1` and `cov2` with `np.eye(2)` which returns a 2-D array with ones on the diagonal and zeros elsewhere.

## 1.2 Task 2

> Implement the perceptron following the specs in perceptron.py and the pseudocode in perceptronslides.pdf.

## 1.3 Task 3

> Train the perceptron on the training data (160 points) and test in on the remaining 40 test points. Compute the classification accuracy on the test set.

In [3]:

```
p = Perceptron(2)

"""
gen dataset
"""
data_size = 100
train_size = 80
x1 = gen_gaussian_distribution(data_size, [5, 5])
x2 = gen_gaussian_distribution(data_size, [-5, -5])
y1 = a_label = np.ones(data_size, dtype=np.int16)
y2 = -y1
x_train = np.concatenate((x1[:train_size], x2[:train_size]), axis=0)
y_train = np.concatenate((y1[:train_size], y2[:train_size]), axis=0)
x_test = np.concatenate((x1[train_size:], x2[train_size:]), axis=0)
y_test = np.concatenate((y1[train_size:], y2[train_size:]), axis=0)

"""
train model
"""
p.train(x_train, y_train)
```

```
"""
test model
"""
acc = p.score(x_test, y_test)

print(f'Perceptron test accuracy: {acc * 100}%')
```

```
Perceptron test accuracy: 100.0%
```

## 1.4 Task 4

> Experiment with different sets of points (generated as described in Task 1). What happens during the training if the means of the two Gaussians are too close and/or if their variance is too high?

In [4]:

```
for _ in range(10):
    p = Perceptron(2)

    """
    gen dataset
    """
    data_size = 100
    train_size = 80
    x1 = gen_gaussian_distribution(data_size, [1, 1])
    x2 = gen_gaussian_distribution(data_size, [1, 1])
    y1 = a_label = np.ones(data_size, dtype=np.int16)
    y2 = -y1
    x_train = np.concatenate((x1[:train_size], x2[:train_size]), axis=0)
    y_train = np.concatenate((y1[:train_size], y2[:train_size]), axis=0)
    x_test = np.concatenate((x1[train_size:], x2[train_size:]), axis=0)
    y_test = np.concatenate((y1[train_size:], y2[train_size:]), axis=0)

    """
    train model
    """
    p.train(x_train, y_train)

    """
    test model
    """
    acc = p.score(x_test, y_test)

    print(f'Perceptron test accuracy: {acc * 100}%')
```

```
Perceptron test accuracy: 47.5%
Perceptron test accuracy: 52.5%
Perceptron test accuracy: 50.0%
Perceptron test accuracy: 60.0%
Perceptron test accuracy: 57.49999999999999%
Perceptron test accuracy: 45.0%
Perceptron test accuracy: 55.00000000000001%
Perceptron test accuracy: 47.5%
Perceptron test accuracy: 52.5%
Perceptron test accuracy: 57.49999999999999%
```

We run 10 times for the close Gaussians([1, 1], [1, 1]), and accuracy is lower then 50%.

In [5]:

```
for _ in range(10):
    p = Perceptron(2)

    """
    gen dataset
```

```
    """
    data_size = 100
    train_size = 80
    x1 = gen_gaussian_distribution(data_size, [-5, -5])
    x2 = gen_gaussian_distribution(data_size, [1, 1])
    y1 = a_label = np.ones(data_size, dtype=np.int16)
    y2 = -y1
    x_train = np.concatenate((x1[:train_size], x2[:train_size]), axis=0)
    y_train = np.concatenate((y1[:train_size], y2[:train_size]), axis=0)
    x_test = np.concatenate((x1[train_size:], x2[train_size:]), axis=0)
    y_test = np.concatenate((y1[train_size:], y2[train_size:]), axis=0)

    """
    train model
    """
    p.train(x_train, y_train)

    """
    test model
    """
    acc = p.score(x_test, y_test)

    print(f'Perceptron test accuracy: {acc * 100}%')
```

```
Perceptron test accuracy: 85.0%
Perceptron test accuracy: 97.5%
Perceptron test accuracy: 97.5%
Perceptron test accuracy: 97.5%
Perceptron test accuracy: 97.5%
Perceptron test accuracy: 97.5%
Perceptron test accuracy: 97.5%
Perceptron test accuracy: 100.0%
Perceptron test accuracy: 92.5%
Perceptron test accuracy: 90.0%
```

We run 10 times for the Gaussians with high variance([-5, -5], [1, 1]), and accuracy is higher then the close Gaussians, over 90%.

# 2 Part II: the mutli-layer perceptron

## 2.1 Task 1

Implement the MLP architecture by completing the files mlp_numpy.py and modules.py.

In [ ]:

```
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from modules import *

class MLP(object):

    def __init__(self, n_inputs, n_hidden, n_classes):
        """
        Initializes multi-layer perceptron object.
        Args:
            n_inputs: number of inputs (i.e., dimension of an input vector).
            n_hidden: list of integers, where each integer is the number of u
            n_classes: number of classes of the classification problem (i.e.,
        """
        self.n_inputs = n_inputs
        self.n_hidden = n_hidden
        self.n_classes = n_classes
```

```
        self.layers = []
        n_pre = n_inputs
        for n_units in n_hidden:
            self.layers.append(Linear(n_pre, n_units))
            self.layers.append(ReLU())
            pre = n_units
        self.layers.append(Linear(pre, n_classes))
        self.layers.append(SoftMax())

    def forward(self, x):
        """
        Predict network output from input by passing it through several layer
        Args:
            x: input to the network
        Returns:
            out: output of the network
        """
        out = x
        for layer in self.layers:
            out = layer.forward(out)
        return out

    def backward(self, dout):
        """
        Performs backward propagation pass given the loss gradients.
        Args:
            dout: gradients of the loss
        """
        for layer in self.layers[::-1]:
            dout = layer.backward(dout)
        return dout


if __name__ == '__main__':
    pass
```

In [ ]:

```
import numpy as np

class Linear(object):
    def __init__(self, in_features, out_features):
        """
        Module initialisation.
        Args:
            in_features: input dimension
            out_features: output dimension
        TODO:
        1) Initialize weights self.params['weight'] using normal distribution
        std = 0.0001.
        2) Initialize biases self.params['bias'] with 0.
        3) Initialize gradients with zeros.
        """
        mean = 0
        std = 0.0001
        size = (in_features, out_features)
        self.params = {}

        self.params['weight'] = np.random.normal(mean, std, size)
        self.params['bias'] = np.zeros(out_features)
        self.gradients = {}

    def forward(self, x):
        """
        Forward pass (i.e., compute output from input).
```

```
        Args:
            x: input to the module
        Returns:
            out: output of the module
        Hint: Similarly to pytorch, you can store the computed values inside 
        and use them in the backward pass computation. This is true for *all*
        """
        self.x = x
        w = self.params['weight']
        b = self.params['bias']
        out = np.dot(x, w) + b
        self.out = out
        return out

    def backward(self, dout):
        """
        Backward pass (i.e., compute gradient).
        Args:
            dout: gradients of the previous module
        Returns:
            dx: gradients with respect to the input of the module
        TODO:
        Implement backward pass of the module. Store gradient of the loss wit
        layer parameters in self.grads['weight'] and self.grads['bias'].
        """
        # TODO
        self.gradients['weight'] = np.dot(np.transpose(self.x), dout)  / self
        self.gradients['bias'] = np.mean(dout, axis=0)
        dx = np.dot(dout, np.transpose(self.params['weight']))
        return dx

class ReLU(object):
    def forward(self, x):
        """
        Forward pass.
        Args:
            x: input to the module
        Returns:
            out: output of the module
        """
        self.x = x;
        out = np.maximum(x, 0)
        return out

    def backward(self, dout):
        """
        Backward pass.
        Args:
            dout: gradients of the previous module
        Returns:
            dx: gradients with respect to the input of the module
        """
        dx = np.where(self.x > 0, dout, 0)
        return dx

class SoftMax(object):
    def exp_normalize(self, x):
        b = x.max()
        y = np.exp(x - b)
        return y / np.reshape(y.sum(axis=1), (-1, 1))

    def forward(self, x):
        """
        Forward pass.
        Args:
```

```
            x: input to the module
        Returns:
            out: output of the module

        TODO:
        Implement forward pass of the module.
        To stabilize computation you should use the so-called Max Trick
        https://timvieira.github.io/blog/post/2014/02/11/exp-normalize-trick/

        """
        self.x = x
        out = self.exp_normalize(x)
        self.out = out
        return out

    def backward(self, dout):
        """
        Backward pass.
        Args:
            dout: gradients of the previous module
        Returns:
            dx: gradients with respect to the input of the module
        """
        # TODO
        dx = (dout - np.reshape(np.sum(dout * self.out, 1), [-1, 1])) * self.

class CrossEntropy(object):
    def forward(self, x, y):
        """
        Forward pass.
        Args:
            x: input to the module
            y: labels of the input
        Returns:
            out: cross entropy loss
        """
        # TODO
        out = np.sum(- np.log(np.maximum(x, 1e-8)) * y) / x.shape[0]
        return out

    def backward(self, x, y):
        """
        Backward pass.
        Args:
            x: input to the module
            y: labels of the input
        Returns:
            dx: gradient of the loss with respect to the input x.
        """
        dx = - y / (np.maximum(x, 1e-8))
        return dx


if __name__ == "__main__":
    softmax = SoftMax()

    crossentropy = CrossEntropy()
```

## 2.2 Task 2

Implement training and testing script in train mlp numpy.py. (Please keep 80% of the dataset for training and the remaining 20% for testing. Note that this is a random split of 80% and

20% )

In [3]:
```
def generate_dataset():
    data_size = 1000
    train_size = 800
    x, y = datasets.make_moons(data_size, shuffle=True, noise=None)
    # one hot
    data_shape = 2
    y = np.eye(data_shape)[y.reshape(-1)]
    train_x = x[:train_size]
    train_y = y[:train_size]
    test_x = x[train_size:]
    test_y = y[train_size:]

    return train_x, train_y, test_x, test_y
```

## 2.3 Task 3

Using the default values of the parameters, report the results of your experiments using a jupyter notebook where you show the accuracy curves for both training and test data.

# 3 Part III: stochastic gradient descent

## 3.1 Task 1

Modify the train method in train mlp numpy.py to accept a parameter that allows the user to specify if the training has to be performed using batch gradient descent (which you should have implemented in Part II) or stochastic gradient descent.

## 3.2 Task 2

Using the default values of the parameters, report the results of your experiments using a jupyter notebook where you show the accuracy curves for both training and test data.

## How to run code

for SGD, default is BGD

```
python3  train_mlp_numpy.py --grdient_descent SGD
```