CS324: Deep Learning

Assignment 1

1. Part I: the perceptron

1.1 Task 1

To generate the required dataset, two sets of points matching a two-dimensional Gaussian distribution, of size 100, are generated using the *random.multivariate_normal* function. Different Gaussian distributions can be generated by changing the mean and co-variance. This is then divided into a training set and a test set.

为了生成要求的数据集，使用 random.multivariate_normal 函数生成两个符合二维高斯分布的，大小为 100 的点集合。通过更改均值和协方差可以生成不同的高斯分布。之后将其划分为训练集和测试集。

```
# generate dataset
mean1, mean2, = [10, 16], [-6, -3]
cov1 = [[3, 0], [0, 4]]
cov2 = [[2, 0], [0, 5]]
x1, y1 = random.multivariate_normal(mean1, cov1, 100).T
x2, y2 = random.multivariate_normal(mean2, cov2, 100).T
# split into train set and test set
train_data = np.array([[x1[i], y1[i]] for i in range(80)] + [[x2[i], y2[i]] for i in range(80)])
train_label = np.array([1 for i in range(80)] + [-1 for i in range(80)])
test_data = np.array([[x1[i], y1[i]] for i in range(80, 100)] + [[x2[i], y2[i]] for i in range(80, 100)])
test_label = np.array([1 for i in range(20)] + [-1 for i in range(20)])
```

1.2 Task 2

First implement the Perceptron class. In the *__init__* method, my code will initialise the weights, number of inputs, maximum training rounds and learning rate of the perceptron. In the *forward* method, it will use *np.dot* to calculate $y = w \cdot x$, and then use *np.sign* to obtain the positivity or negativity of the result as the predicted value output. In the *train* method, in each round of training, it first disrupts the order of the data and then checks the result of multiplying the predicted value by the label and updates the weights if they are less than zero.

首先实现 Perceptron 类。在__init__方法中，初始化感知机的权重、输入数量、最大训练轮次和学习率。在 forward 方法中，使用 np.dot 计算 $y = w \cdot x$，再使用 np.sign 获取结果的正负性作为预测值输出。在 train 方法中，每轮训练先打乱数据顺序，再检查预测值乘标签的结果，如果小于零则更新权重。

```python
def __init__(self, n_inputs, max_epochs=1e2, learning_rate=1e-2):
    """..."""
    self.w = np.array([0, 0])
    self.n = n_inputs
    self.epoch = max_epochs
    self.rate = learning_rate

 montpelllier
def forward(self, input_data):
    """..."""
    label = np.sign(np.dot(self.w, np.transpose(input_data)))
    return label

 montpelllier
def train(self, training_inputs, labels):
    """..."""
    for _ in range(int(self.epoch)):
        tmp = np.column_stack((training_inputs, labels))
        np.random.shuffle(tmp)
        training_inputs = tmp[:, :-1]
        labels = tmp[:, -1]
        predict = self.forward(training_inputs)
        if np.dot(predict, labels) <= 0:
            self.w = self.w + self.rate * np.dot(labels, training_inputs)
```

1.3 Task 3

To get the training results and calculate the accuracy, we first need to create a Perceptron object and call its train method. After training, the forward function is called to obtain the predictions of the perceptron on the test set and then the number of identical values is counted to calculate the accuracy.

为了得到训练结果并计算准确率，首先我们需要创建一个 Perceptron 对象，并调用它的 train 方法。完成训练后，调用 forward 函数获得感知机对测试集的预测结果，然后统计出其与真实值相同的个数来计算准确率。

```
res = perceptron.forward(test_data)
cnt = 0
for label, predict in zip(test_label, res):
    if label == predict:
        cnt += 1
print("accuracy: ", cnt / len(test_data))
```

1.4 Task 4

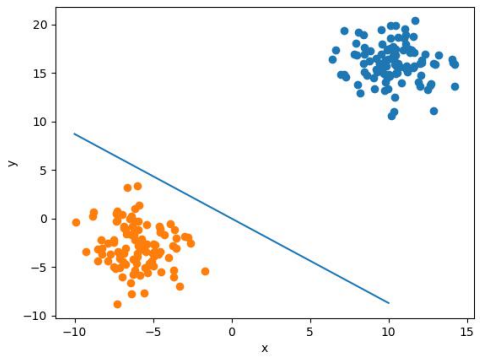
Finally we can generate different data sets by changing the mean and covariance matrices several times. In addition, with the help of the *matplotlib* library, we can easily draw all the generated points and use straight lines to represent the weights of the perceptron, presenting the experimental results more intuitively and facilitating analysis.

最后，我们可以通过多次更改均值和协方差矩阵来生成不同的数据集。此外，借助 matplotlib 库，我们可以很方便地画出所有生成的点，并利用直线表示感知机的权重，更直观地展示实验结果，并方便进行分析。

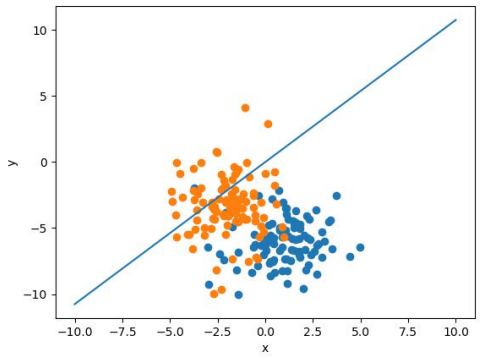
```
# draw
x = np.array([-10, 10])
y = -perceptron.w[0] / perceptron.w[1] * x
plt.xlabel("x")
plt.ylabel("y")
plt.scatter(x1, y1)
plt.scatter(x2, y2)
plt.plot(x, y)
plt.show()
```

Results and Analysis

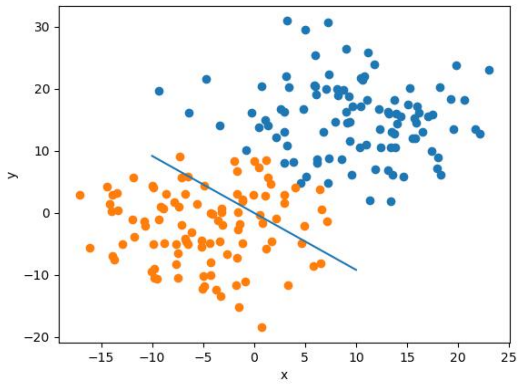
Test ID	Mean1	Mean2	Cov1	Cov2	Accuracy
1	[10, 16]	[-6, -3]	[[3, 0], [0, 4]]	[[2, 0], [0, 5]]	1.0
2	[1, -6]	[-2, -3]	[[3, 0], [0, 4]]	[[2, 0], [0, 5]]	0.625
3	[10, 16]	[-6, -3]	[[40, 0], [0, 34]]	[[38, 0], [0, 35]]	0.825
4	[1, -6]	[-2, -3]	[[40, 0], [0, 34]]	[[38, 0], [0, 35]]	0.65
5	[19, 16]	[5, 6]	[[4, 0], [0, 4]]	[[3, 0], [0, 5]]	0.5



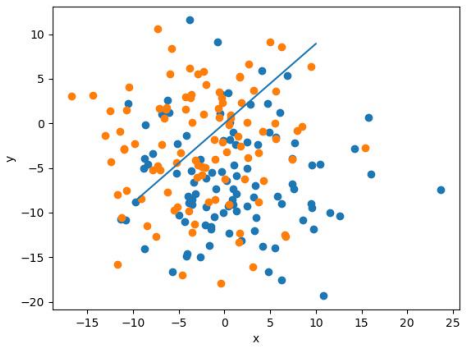
Test 1



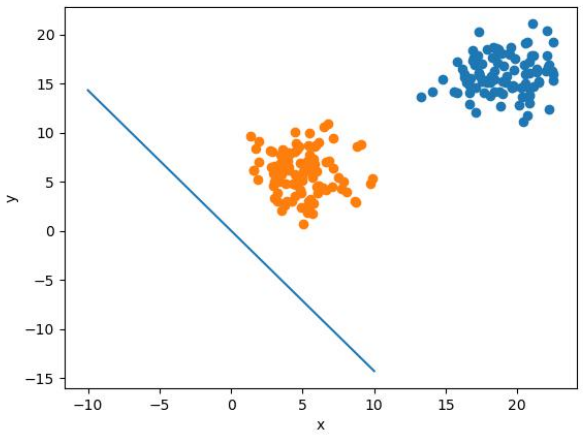
Test 2



Test 3



Test 4



Test 5

In total, five sets of experiments were conducted. In test 1, the two point sets were so dispersed that the perceptron was able to achieve 100% accurate classification. It can be observed that the blue line representing the perceptron in the diagram completely divides the points of the two colors. In Tests 2 and 3, I set the means of the two point sets in one group to be very close and the covariance in the other group to be large enough to result in a large overlap between the two, resulting in a final accuracy of 62.5% and 82.5% respectively. In Test 4, I set the means of the two point sets very close to each other while setting their co-variances large enough to achieve a final accuracy of 65%. In test 5, the perceptron was only 50% accurate, despite the two point sets being

very spread out. This is because the mean points of both datasets are near the straight line at $y = x$ and both lie in the first quadrant. While the perceptron does not set the bias, resulting in the line it draws only passing through the origin and therefore failing to classify them.

总共进行了五组实验。在测试 1 中，两个点集非常分散，感知机能够做到 100%地准确分类。可以观察到图中表示感知机的蓝线将两种颜色的点完全分割。在测试 2 和测试 3 中，我将其中一组里的两个点集的均值设置得十分接近，另一组的协方差设置得足够大，最终导致者两者大量重合，使得最终准确率分别仅有 62.5%和 82.5%。而在测试 4 中，我将两个点集的均值设置得十分接近的同时，把他们的协方差设置得足够大，最终准确率仅 65%。在测试 5 中，尽管两个点集十分分散，但是感知机的准确率只有 50%。这是因为两个数据集的均值点都在 $y = x$ 的直线附近，并且都位于第一象限。而该感知机并未设置 bias，导致其划出的直线只能经过原点，故不能将其分类。

2. Part II: the mutli-layer perceptron

2.1 Task 1

In the *Linear* class, the forward propagation, backward propagation and weight update operations of the linear layer are implemented using the forward, backward and update functions respectively. On initialization, weights and biases are generated randomly using the *np.random.randn* and *np.zeros* functions, and stored in the *params* dictionary. Similarly, *self.grads* is initialized to store the gradient values. In forward propagation, the input *x* is multiplied by the weights and bias to obtain the output *out*, and *x* is stored in the instance variable *self.x* for use in backward propagation. In back propagation, the derivative from the previous calculation is accepted as the input *dout*, which is used to calculated *dx*, then return it to the previous layer. The weights and bias gradient values are then stored in the *grads* dictionary during the calculation. Finally in the weight update, the weights and biases are updated directly by subtracting the learning rate *lr* from the parameters and multiplying by the value of the gradient.

As activation functions, the corresponding forward propagation, backward propagation and update parameter methods are implemented in the *ReLU* class. In the *forward* method, a non-negative limit is imposed on each element of the input *x* by the *np.maximum* function, which means only values greater than or equal to 0 are retained and the rest are set to 0. In the *backward* method, values greater than or equal to 0 in *self.x* are set to 1 and values less than or equal to 0 are set to 0. The backpropagation gradient *dx* of the layer is then obtained by multiplying it element by element with *dout* obtained from the previous layer. Since *ReLU* has no parameters to update, the *update* method remains empty.

For the *SoftMax* class, the *forward* method uses the *Gumbel-Max Trick* algorithm, which subtracts the largest element of *x* to avoid overflowing the result. For the *backward* method, since its derivation is more tedious, and its combination with cross-entropy is simpler, the result is calculated directly in the backward function of the *CrossEntropy* class. Therefore, we can directly return the back-propagation gradient *dout* of the previous layer.

Next is the *CrossEntropy* class. In the forward propagation method *forward*, a very small constant

c (here we set as 10^{-7}) is first added to the input *x* to avoid errors in the calculation of $log(0)$. Afterwards *y* is multiplied element by element with the logged *x* and a negative number is taken. Finally the final loss function value *out* is summed using the *np.sum* function, where *x* is the predicted outcome of the model and *y* is the true label of the data. And in backward, the gradient of the cross-entropy loss function with softmax is returned directly, namely $dx = x - y$.

Finally, there is the implementation of the *MLP* class. The *__init__* function is used to initialize the network structure, including setting the loss function to *CrossEntropy*, specifying the *softmax* method and generating the corresponding intermediate linear layers and activation functions based on the input dimension, output dimension and number of hidden layers. The *forward* function calls the corresponding *forward* function layer by layer through a *for* loop to output the prediction, while the *backward* function, in contrast, calls the *backward* function of each layer one by one, starting from the last layer, according to the chain rule, to update the gradient. A *predict* function is also implemented, which is roughly the same as the *forward* function, the only difference being that the *flag* is set to *False* when the *forward* function is called at each layer, in order to only make predictions without interfering with the subsequent backpropagation.

In particular, the *flag* in the *forward* function of all the above classes is used to distinguish whether the purpose is training or just prediction, and thus to decide whether to update the value of *self.x* for backpropagation. In addition, to simplify the code, the *__call__* methods of all the above classes are set to *forward*.

在 Linear 类中，线性层的前向传 s 播、反向传播和权重更新操作分别通过 forward、backward 以及 update 函数实现。初始化时，使用 np.random.randn 和 np.zeros 函数随机生成权重和偏置，并将它们存储在 params 字典中。类似地，self.grads 的初始化用于存储梯度值。在前向传播中，输入 x 乘以权重 weight 和偏置 bias 以获取输出 out，并将 x 存储在实例变量 self.x 中以便在反向传播中使用。在反向传播中，接受上一次计算出的导数，即从输入 dout 计算出 dx 并返回给前一层，并且在计算过程中把权重和偏置的梯度值存在 grads 字典中。最后在权重更新中，直接将参数减去学习率 lr 乘以梯度的值来更新权重和偏置。

作为激活函数，在 ReLU 类中实现了对应的前向传播、反向传播和更新参数方法。在 forward 方法中，通过 np.maximum 函数实现了对输入 x 中每个元素进行非负限制，即仅保留大于等于 0 的值，其余设为 0。在 backward 方法中，先将 self.x 中大于 0 的值置为 1，小于等于 0 的值置为 0。然后用上一层反向传播得到的 dout 和它进行逐元素乘法，即可得到该层的反向传播梯度 dx。由于 ReLU 没有需要更新的参数，因此 update 方法为空。

对于 SoftMax 类， forward 方法使用了 “Gumbel-Max Trick”算法，即减去 x 中最大的元素来避免结果的数值过大导致溢出。对于 backward 方法，由于其求导比较繁琐，而将其与交叉熵结合后的求导较为简单，因此直接在 CrossEntropy 类的 backward 函数中计算出结果，此处就可以直接返回上一层反向传播梯度 dout。

接下来是 CrossEntropy 类。在前向传播方法 forward 中，先对输入 x 加上一个非常小的常数 c（此处取10^{-7}）以避免在计算 $log(0)$时出现错误。之后将 y 与 log 后的 x 进行逐元素乘法并取负数。最后用 np.sum 函数进行求和得到最终损失函数值 out。其中，x 是模型的预测结果，y 是数据的真实标签。而在 backward 中，直接返回交叉熵损失函数与 softmax 的梯度，即 $dx = x - y$。

最后是 MLP 类的实现。__init__函数用于初始化网络结构，包括设定损失函数为

CrossEntropy，指定 softmax 方法以及根据输入维度、输出维度和隐藏层数生成对应的中间线性层和激活函数。其中通过 for 循环能正确解析 n_hidden 参数。Forward 函数通过 for 循环从一层逐层调用对应的 forward 函数输出预测结果。Backward 函数则相反，其根据链式法则从最后一层开始逐个调用每层的 backward 函数，从而更新梯度。此外还实现了 predict 函数，其大致与 forward 函数相同，唯一的区别是在调用每层的 forward 函数将 flag 设为 False，目的是只做预测而不干扰后续的反向传播。

特别地，上述所有类中的 forward 函数中 flag 用去区别目的是训练还是仅预测，从而决定是否更新 self.x 的值来进行反向传播。此外，为了简化代码，上述所有类的__call__方法都设为 forward。

2.2 Task 2

In the *main* function, firstly, all parameters are parsed using the *parser.parse_args()* method, including the number of hidden layers, evaluation frequency, learning rate, maximum number of steps and whether to perform random gradient descent. The requested dataset is then generated using the *make_moons* method and divided into a training set and a test set in an 8:2 ratio. For the labelled data, the *oneHot* function implemented in *numpy* is called to convert it into the form of a unique hot code. The generated data points are then plotted using the *plt.scatter* method. Finally the *train* function is called for training.

In the *train* function, a new *MLP* object is first created to be used as the model. In each round of training, *train_x* is passed as input to the forward method of the model to obtain the predicted value pred. The backward method of *module.loss_fc* is then called to obtain the gradient value and passed to the *module.backward* method for back propagation. When the number of rounds meets the evaluation frequency, the *module.predict* function is called to obtain the prediction results for the training and test sets and the accuracy is calculated using the accuracy method. At the same time *module.loss_fc* is called to calculate the loss function and the results are printed and stored in the list. Finally, the weights are updated with a *for* loop calling the update method layer by layer. After the training is completed, *plt* is used to show the whole training process and the final results.

首先是 main 函数。先利用 parser.parse_args()方法解析所有参数，包括隐藏层数、评估频率、学习率、最大步数和是否进行随机梯度下降。之后用 make_moons 方法生成要求的数据集，并按照 8:2 的比列划分为训练集和测试集。对于标签数据，调用用 numpy 实现的 oneHot 函数来将其转换成独热码的形式。然后使用 plt.scatter 方法绘制生成的数据点。最后调用 train 函数进行训练。

在 train 函数中，先新建 MLP 对象用作模型。在每轮训练中，将 train_x 作为输入传入模型的 forward 方法，得到预测值 pred。再调用 module.loss_fc 的 backward 方法获得梯度值，并将其传入 module.backward 方法来进行反向传播。当轮次数满足评估频率时，调用 module.predict 函数获取训练集和测试集的预测结果，并用 accuracy 方法计算准确率。同时调用 module.loss_fc 计算损失函数，并打印结果并存入 list 中。最后用 for 循环逐层调用 update 方法更新权重。训练完成后使用 plt 展示整个训练过程和最终结果。

```python
def oneHot(labels, dim=None):
    if dim is None:
        dim = np.max(labels) + 1
    one_hot = np.zeros((len(labels), dim))
    one_hot[np.arange(len(labels)), labels] = 1
    return one_hot
```

```python
def accuracy(predictions, labels):
    """..."""
    correct = 0
    for i, pred in enumerate(predictions):
        if np.argmax(pred) == np.argmax(labels[i]):
            correct += 1
    return correct / len(predictions)
```

```python
def main():
    """..."""
    # handle arguments
    args = parser.parse_args()
    dim_hidden = list(map(int, args.dnn_hidden_units.split(',')))
    freq = args.eval_freq
    lr = args.learning_rate
    max_step = args.max_steps
    sgd = args.sgd
    # generate dataset
    size = 1000
    data, label = make_moons(n_samples=size, noise=0.05)
    # split into train set and test set
    bound = int(0.8 * size)
    train_data, test_data = data[:bound], data[bound:]
    train_label, test_label = oneHot(label[:bound]), oneHot(label[bound:])
    train_data, test_data = np.array(train_data), np.array(test_data)
    # draw
    plt.scatter(data[:, 0], data[:, 1], s=10, c=label)
    plt.show()
    # train
    train(max_step, dim_hidden, freq, lr, sgd, (train_data, train_label), (test_data, test_label))
```

2.3 Task 3

In Jupyter Notebook, execute the ***%run train_mlp_numpy.py*** command to get the results of running with the default parameters.

在 Jupyter Notebook 中，执行%run train_mlp_numpy.py 命令，获取以默认参数运行的结果。

Results and Analysis

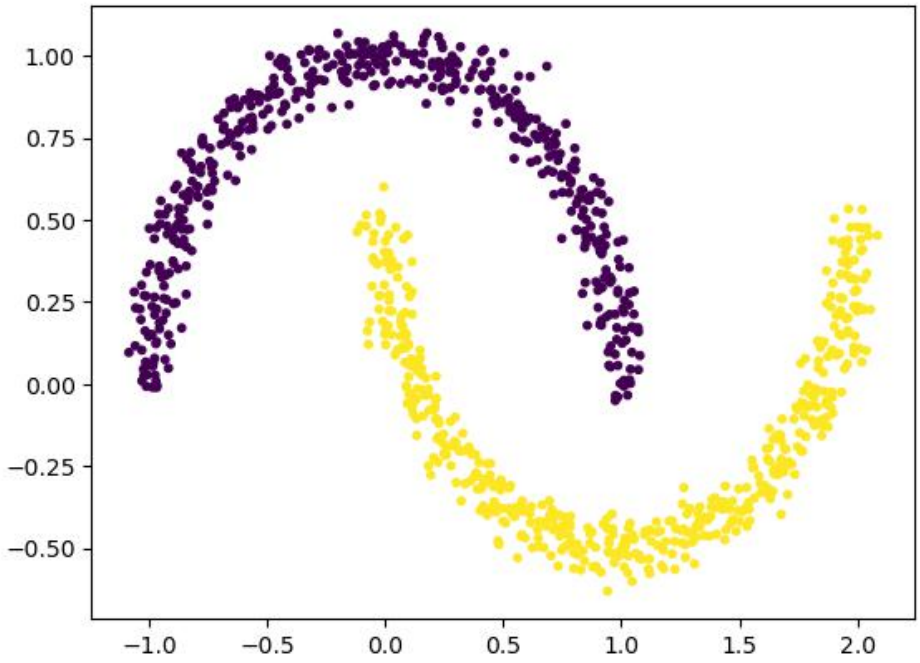


Figure 1

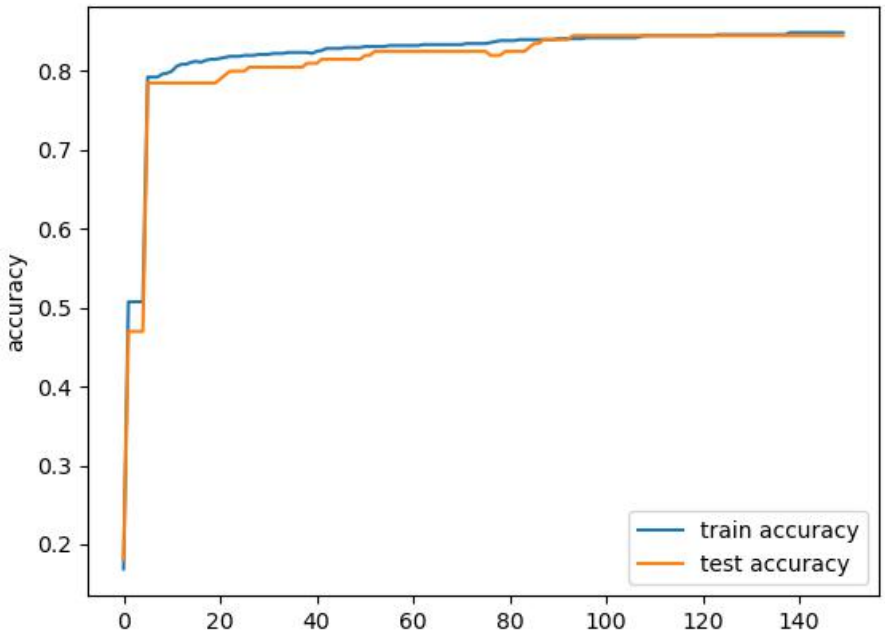


Figure 2

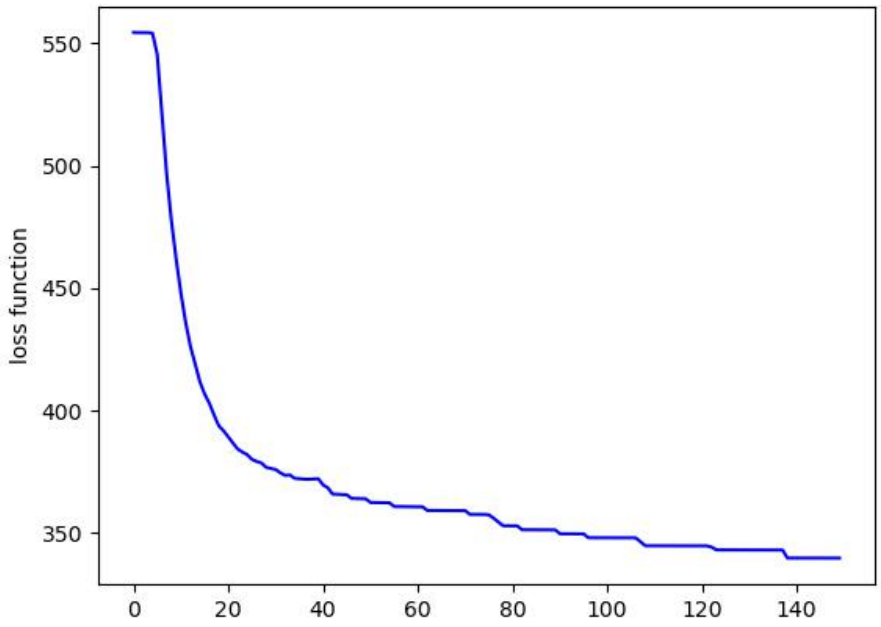


Figure 3

图 1 展示了生成的数据集，图 2 和图 3 分别展示了准确率和损失函数。从图中可以看出，多层感知机的效果不错，最终准确率达到了 84.5%，损失函数也下降到了 340 左右。其中，训练集的准确率略高于测试集的，但两条曲线的形状基本吻合，说明没有发生过拟合。在前 50 轮中，准确率快速上升至 80%左右，但损失函数几乎维持在 554 不变。之后准确率缓慢增长，而损失函数则呈现指数下降的趋势。推测如果继续训练，准确率可能会继续上升。

3. Part III: stochastic gradient descent

3.1 Task 1

In order to implement stochastic gradient descent, a parameter *sgd* is added to the *parser.add_argument* method to indicate whether or not to use stochastic gradient descent. This is then simply determined at each training round, with a random piece of data from the training set being used as input to the model if the SGD is to be enabled; otherwise the entire training set is used as input.

为了实现随机梯度下降，需要先用 parser.add_argument 方法添加一条参数 sgd 用来表示是否使用随机梯度下降方法。之后只需要在每轮训练时进行判断，如果要启用 SGD，则从训练集中随机选择一条数据作为训练数据输入模型；否则将整个训练集作为输入。

```python
for t in range(epoch):
    if sgd:
        rand_i = np.random.randint(len(train_x))
        x = train_x[rand_i:rand_i + 1]
        y = train_y[rand_i:rand_i + 1]
    else:
        x = train_x
        y = train_y
```

3.2 Task 2

Similarly, to run the stochastic gradient descent method with Jupyter Notebook, simply execute the command ***%run train_mlp_numpy.py --sgd True***.

同理，如果想用 Jupyter Notebook 运行随机梯度下降方法，只需执行命令%run train_mlp_numpy.py --sgd True。

Results and Analysis

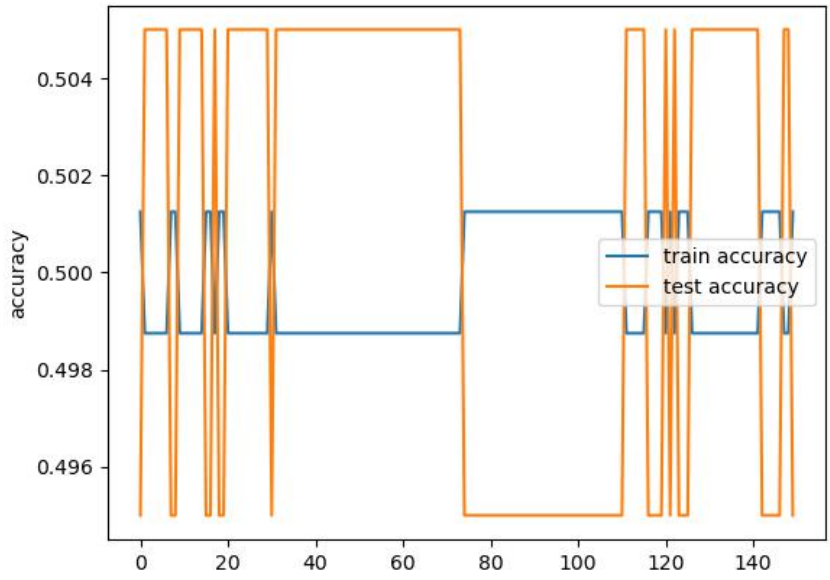


Figure 1

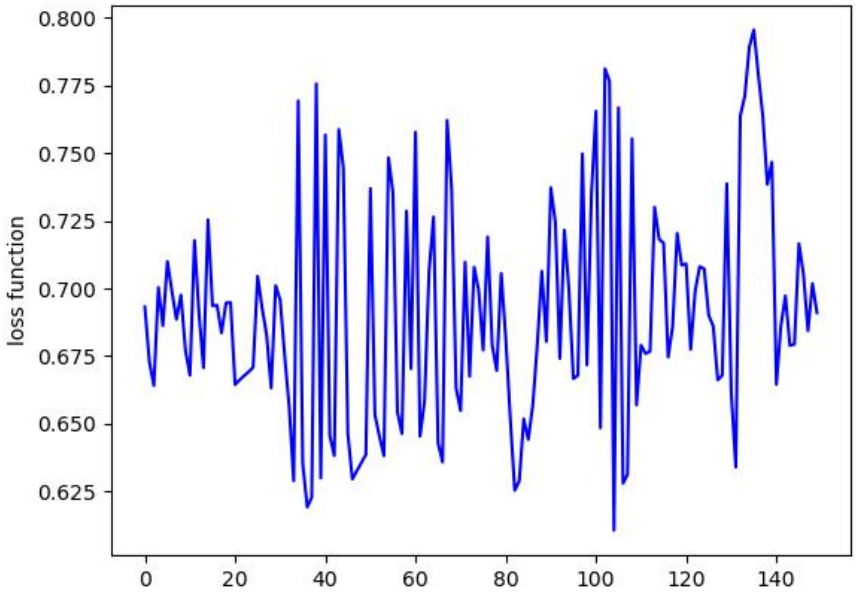


Figure 2

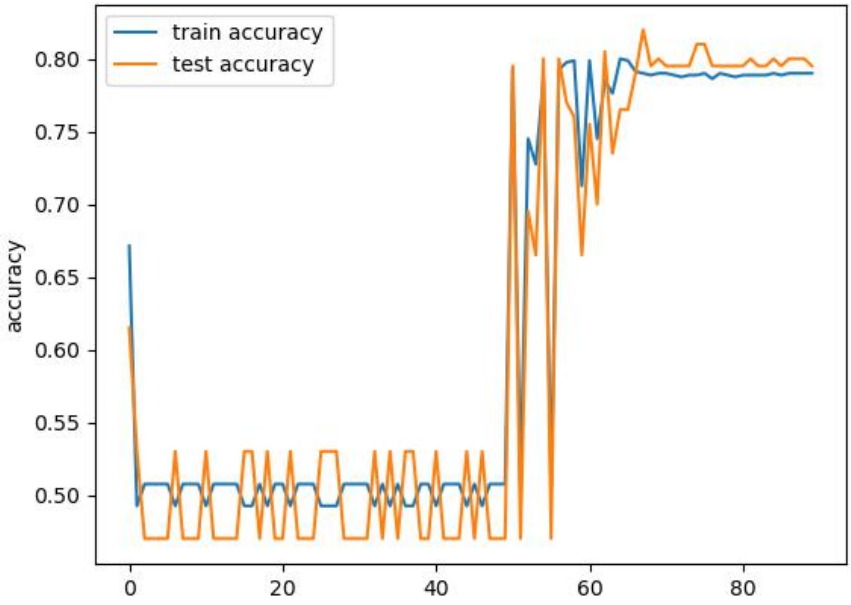


Figure 3

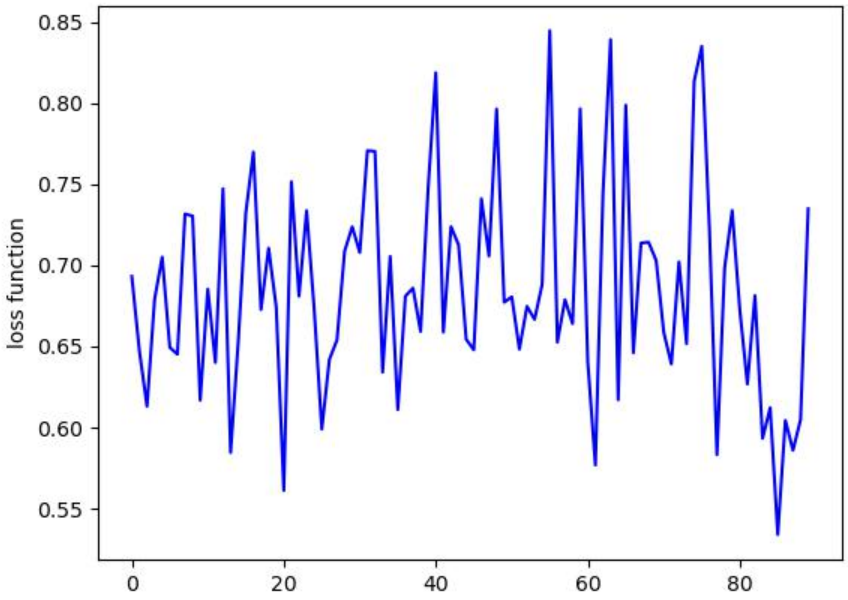


Figure 4

Figure 1 and 2 show the results obtained by running SGD with the default parameters. As can be seen from the above figure, SGD training is not very effective when compared to batch gradient descent (BGD), with the accuracy hovering around 50% and the loss function fluctuating dramatically all the time, with no downward trend. However, at the same time, the time cost of its training dropped dramatically, so this result may be due to too few training rounds. The maximum number of steps and the evaluation frequency are adjusted to 45000 and 500 respectively and run again to obtain Figures 3 and 4. It is easy to see that although the accuracy still fluctuates around 50% until 25000 rounds, it rises rapidly to around 80% after this time and takes much less time than the BGD. The loss function fluctuates more dramatically, but also shows a slow decline after 25000 rounds.

图 1 图 2 是用默认参数运行 SGD 得到的结果。从上图中可以看出，与批量梯度下降（BGD）比较，SGD 训练的效果并不好，准确率在 50%左右徘徊，损失函数也一直在剧烈波动，并没有下降的趋势。但同时，其训练的时间成本大幅下降，因此这个结果可能是训练轮次太少导致。将最大步数和评估频率分别调至 45000 和 500，再次运行得到图 3 和图 4。不难看出，尽管在 25000 轮前准确率依然在 50%左右波动，但这次之后迅速上升至 80%左右，且用时也远小于 BGD。损失函数的波动更为剧烈，但同样在 25000 轮后呈现缓慢下降的趋势。