Deep Learning Assignment 2 Report

姓名：邱逸伦

学号：12013006

Note: all the codes are implemented in Jupyter Notebook files, which means they can be run directly if all the packages are installed. I also check the environment in the codes so all the Jupyter Notebook files can be run in both GPU environment (cuda) and CPU environment (very slow).

Part 1

Task 1&2

Import the necessary libraries

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

import torch
import argparse
import torchvision
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt
from sklearn import datasets
from sklearn import model_selection
from torchvision import transforms
from torch.utils.data import DataLoader
```

Use default arguments as shown below, in task 1&2, I will use these arguments to train the MLP.

```
DNN_HIDDEN_UNITS_DEFAULT = '20'
LEARNING_RATE_DEFAULT = 1e-2
MAX_EPOCHS_DEFAULT = 200
EVAL_FREQ_DEFAULT = 40

parser = argparse.ArgumentParser()
```

```
parser.add_argument('--dnn_hidden_units', type=str, default=DNN_HIDDEN_UNITS_DEFAULT,
                    help='Comma separated list of number of units in each hidden layer')
parser.add_argument('--learning_rate', type=float, default=LEARNING_RATE_DEFAULT,
                    help='Learning rate')
parser.add_argument('--max_epochs', type=int, default=MAX_EPOCHS_DEFAULT,
                    help='Number of epochs to run trainer.')
parser.add_argument('--eval_freq', type=int, default=EVAL_FREQ_DEFAULT,
                    help='Frequency of evaluation on the test set')
FLAGS, unparsed = parser.parse_known_args()
dnn_hidden_units = FLAGS.dnn_hidden_units
n_hidden = dnn_hidden_units.split(",")
n_hidden = [int(x) for x in n_hidden]
learning_rate = FLAGS.learning_rate
max_epochs = FLAGS.max_epochs
eval_freq = FLAGS.eval_freq

DATASET = None  # denote the current dataset
FRAMEWORK = None  # denote the current framework
```

Some global methods. `draw` can plot the accuracy chart after finishing training the MLP. `one_hot` can convert the input to the one-hot vector. `numpy_acc` and `pytorch_acc` are used for calculating the accuracy of different frameworks.

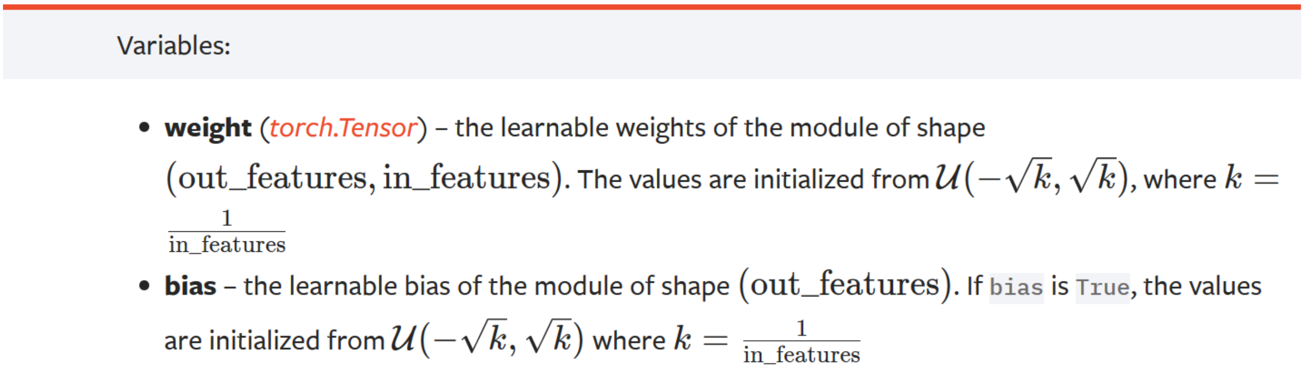
```
def draw(train_loss, test_loss, train_acc, test_acc, epochs):
    plt.figure()
    plt.subplot(1, 2, 1)
    plt.plot(epochs, train_loss, color='red', alpha=0.8, linewidth=1, label='Train Loss')
    plt.plot(epochs, test_loss, color='blue', alpha=0.8, linewidth=1, label='Test Loss')
    plt.legend(loc='upper right')
    plt.title(f'Loss Chart {DATASET} {FRAMEWORK}', fontdict={'weight':'normal','size': 8})
    plt.subplot(1, 2, 2)
    plt.plot(epochs, train_acc, color='red', alpha=0.8, linewidth=1, label='Train Accuracy')
    plt.plot(epochs, test_acc, color='blue', alpha=0.8, linewidth=1, label='Test Accuracy')
    plt.ylim([0, 1.1])
    plt.legend(loc='lower right')
    plt.title(f'Accuracy Chart {DATASET} {FRAMEWORK}', fontdict={'weight':'normal','size': 8})
    plt.show()


def one_hot(input, num_classes=2):
    return np.eye(num_classes)[input]


def numpy_acc(predictions, targets):
    predicted_labels = np.argmax(predictions, axis=1)
    true_labels = np.argmax(targets, axis=1)
    return np.mean(predicted_labels == true_labels)


def pytorch_acc(predictions, targets):
    predictions, targets = predictions.detach().numpy(), targets.detach().numpy()
    predicted_labels = np.argmax(predictions, axis=1)
    true_labels = np.argmax(targets, axis=1)
    return np.mean(predicted_labels == true_labels)
```

Both PyTorch MLP and Numpy MLP use the **SGD** optimizer.

MLP Numpy Implementation and Training

In this task, the `Linear` class updates the initialization method so that the initialization method is the same as the default initialization of the PyTorch framework. Other methods are the same as the implementation of Assignment 1.

Now the initialization method of the Linear layer is shown below.



Variables:

- **weight** (*torch.Tensor*) – the learnable weights of the module of shape $(\text{out\_features}, \text{in\_features})$. The values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$, where $k = \frac{1}{\text{in\_features}}$
- **bias** – the learnable bias of the module of shape (out_features). If `bias` is `True`, the values are initialized from $\mathcal{U}(-\sqrt{k}, \sqrt{k})$ where $k = \frac{1}{\text{in\_features}}$

```
class Linear(object):
    def __init__(self, in_features, out_features):
        k = 1 / in_features
        self.params = {'weight': np.random.uniform(-np.sqrt(k), np.sqrt(k), size=(in_features, 
out_features)),
                       'bias': np.random.uniform(-np.sqrt(k), np.sqrt(k), size=out_features)}
        self.grads = {'weight': np.zeros_like(self.params['weight']),
                      'bias': np.zeros_like(self.params['bias'])}
        self.x = None

    def forward(self, x):
        out = np.dot(x, self.params['weight']) + self.params['bias']
        self.x = x
        return out

    def backward(self, dout):
        n = dout.shape[0]
        dx = np.dot(dout, self.params['weight'].T)
        self.grads['weight'] = np.dot(self.x.T, dout) / n
        self.grads['bias'] = np.sum(dout, axis=0) / n
        return dx


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

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

    def backward(self, dout):
        dx = dout * (self.x > 0)
        return dx


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

    def forward(self, x):
        exp_vals = np.exp(x - np.max(x, axis=1, keepdims=True))
        self.out = exp_vals / np.sum(exp_vals, axis=1, keepdims=True)
        return self.out

    def backward(self, dout):
        dx = -self.out[:, :, None] * self.out[:, None, :]
        dx[:, np.arange(self.out.shape[1]), np.arange(self.out.shape[1])] = self.out * (1 -
self.out)
        d_result = np.matmul(dout[:, None, :], dx)
        d_result.resize((d_result.shape[-3], d_result.shape[-1]))
        return d_result


class CrossEntropy(object):
    def __init__(self):
        self.pd = None
        self.gt = None
        self.batch_size = None

    def forward(self, pd, gt):
        self.batch_size = pd.shape[0]
        self.pd = pd
        self.gt = gt
        eps = 1e-9
        loss = - np.sum(np.multiply(gt, np.log(pd + eps))) / self.batch_size
        return loss

    def backward(self):
        eps = 1e-9
        d_loss = - self.gt / (self.pd + eps)
        return d_loss


class NumpyMLP(object):

    def __init__(self, n_inputs, n_hidden, n_classes):
        self.layers = []
        in_features = n_inputs
        for out_features in n_hidden:
            self.layers.append(Linear(in_features, out_features))
            self.layers.append(ReLU())
            in_features = out_features
        self.layers.append(Linear(in_features, n_classes))
        self.layers.append(SoftMax())

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

    def backward(self, dout):
        for layer in reversed(self.layers):
            dout = layer.backward(dout)
    def update(self, lr):
        for layer in self.layers:
            if isinstance(layer, Linear):
                layer.params['weight'] -= lr * layer.grads['weight']
                layer.params['bias'] -= lr * layer.grads['bias']
                layer.grads['weight'] = np.zeros_like(layer.params['weight'])
                layer.grads['bias'] = np.zeros_like(layer.params['bias'])
```

```
class NumpySGD(object):
    def __init__(self, mlp):
        self.mlp = mlp

    def optimize(self, inputs, labels, lr, max_epochs, eval_freq):
        x_train, x_test, y_train, y_test = model_selection.train_test_split(inputs, labels,
shuffle=True, test_size=0.2)
        y_train, y_test = one_hot(y_train), one_hot(y_test)
        criterion = CrossEntropy()
        train_loss = []
        best_train_loss = 1e5
        train_acc = []
        best_train_acc = 0
        test_loss = []
        best_test_loss = 1e5
        test_acc = []
        best_test_acc = 0
        epochs = []
        n_samples = x_train.shape[0]
        for epoch in range(max_epochs):
            epochs.append(epoch)
            total_loss = 0
            total_acc = 0
            shuffled_indices = np.random.permutation(n_samples)
            for i in range(n_samples):
                idx = shuffled_indices[i]
                x = x_train[idx]
                y = y_train[idx]
                x, y = x[np.newaxis, :], y[np.newaxis, :]
                pd = self.mlp.forward(x)
                loss = criterion.forward(pd, y)
                total_loss += loss
                d_loss = criterion.backward()
                self.mlp.backward(d_loss)
                self.mlp.update(lr)
                acc = numpy_acc(pd, y)
                total_acc += acc
            avg_loss = total_loss / n_samples
            avg_acc = total_acc / n_samples
            train_loss.append(avg_loss)
            train_acc.append(avg_acc)
            if best_train_loss > avg_loss:
                best_train_loss = avg_loss
            if best_train_acc < avg_acc:
                best_train_acc = avg_acc
            pd_test = self.mlp.forward(x_test)
            total_loss = criterion.forward(pd_test, y_test)
            total_acc = numpy_acc(pd_test, y_test)
            test_loss.append(total_loss)
            test_acc.append(total_acc)
            if best_test_loss > total_loss:
                best_test_loss = total_loss
            if best_test_acc < total_acc:
                best_test_acc = total_acc
            if epoch % eval_freq == 0 or epoch == (max_epochs - 1):
                print(f'SGD Epoch {epoch}:')
                print(f'Average Train Loss: {avg_loss} | '
                      f'Average Train Accuracy: {avg_acc}')
                print(f'Average Test Loss: {total_loss} | '
                      f'Average Test Accuracy: {total_acc}')
                print()
        draw(train_loss, test_loss, train_acc, test_acc, epochs)
        print(f'Best Train Loss: {best_train_loss} | '
              f'Best Train Accuracy: {best_train_acc}\n'
              f'Best Test Loss: {best_test_loss} | '
              f'Best Test Accuracy: {best_test_acc}')


def numpy_train(inputs, labels):
    mlp = NumpyMLP(2, n_hidden, 2)
    optimizer = NumpySGD(mlp)
    optimizer.optimize(inputs, labels, learning_rate, max_epochs, eval_freq)
```

MLP PyTorch Implementation and Training

In this task, I use `nn.Sequential` as a container to store PyTorch layers. In __init__ method, all layers implemented by PyTorch are added to the container.

```
class PyTorchMLP(nn.Module):

    def __init__(self, n_inputs, n_hidden, n_classes):
        super(PyTorchMLP, self).__init__()
        self.layers = nn.Sequential()
        in_features = n_inputs
        for (i, out_features) in enumerate(n_hidden):
            self.layers.add_module('linear' + str(i), nn.Linear(in_features, out_features))
            self.layers.add_module('relu' + str(i), nn.ReLU())
            in_features = out_features
        self.layers.add_module('linear' + str(len(n_hidden)), nn.Linear(in_features, n_classes))

    def forward(self, x):
        out = self.layers(x)
        return out


pytorch_mlp = PyTorchMLP(2, n_hidden, 2)
pytorch_mlp
```

The framework of PyTorchMLP is shown below.

```
PyTorchMLP(
  (layers): Sequential(
    (linear0): Linear(in_features=2, out_features=20, bias=True)
    (relu0): ReLU()
    (linear1): Linear(in_features=20, out_features=2, bias=True)
  )
)
```

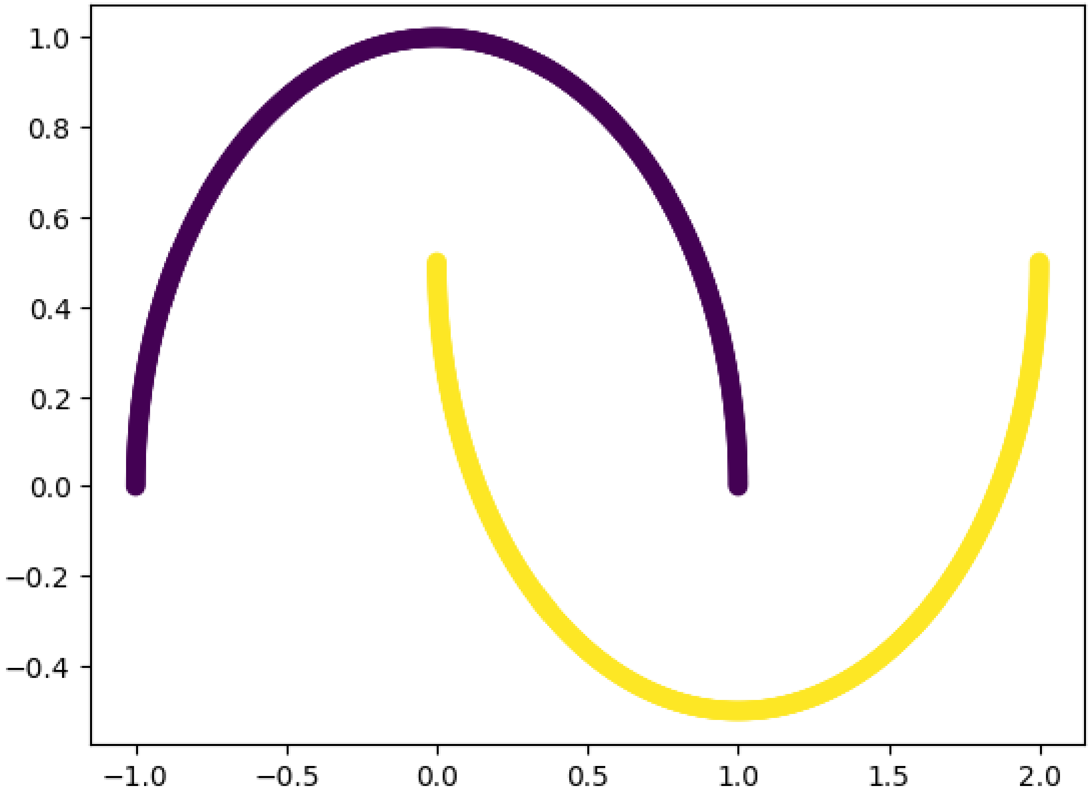
The method `pytorch_train` can train the model and plot the result.

```
def pytorch_train(inputs, labels):
    mlp = PyTorchMLP(2, n_hidden, 2)
    criterion = nn.CrossEntropyLoss()
    x_train, x_test, y_train, y_test = model_selection.train_test_split(inputs, labels,
shuffle=True, test_size=0.2)
    y_train, y_test = one_hot(y_train), one_hot(y_test)
    x_test, y_test = torch.from_numpy(x_test).float(), torch.from_numpy(y_test).float()
    train_loss = []
    best_train_loss = 1e5
    train_acc = []
    best_train_acc = 0
    test_loss = []
    best_test_loss = 1e5
    test_acc = []
    best_test_acc = 0
    epochs = []
    n_samples = x_train.shape[0]
    for epoch in range(max_epochs):
        epochs.append(epoch)
        total_loss = 0
        total_acc = 0
        shuffled_indices = np.random.permutation(n_samples)
        for i in range(n_samples):
            idx = shuffled_indices[i]
            x = x_train[idx]
            y = y_train[idx]
            x, y = x[np.newaxis, :], y[np.newaxis, :]
            x, y = torch.from_numpy(x).float(), torch.from_numpy(y).float()
            pd = mlp(x)
            loss = criterion(pd, y)
            total_loss += loss
            mlp.zero_grad()
            loss.backward()
            for f in mlp.parameters():
                f.data.sub_(f.grad.data * learning_rate)
            acc = pytorch_acc(pd, y)
            total_acc += acc
        avg_loss = total_loss / n_samples
        avg_acc = total_acc / n_samples
        train_loss.append(avg_loss.detach().numpy())
        train_acc.append(avg_acc)
        if best_train_loss > avg_loss:
            best_train_loss = avg_loss
        if best_train_acc < avg_acc:
            best_train_acc = avg_acc
        pd_test = mlp(x_test)
        with torch.no_grad():
            loss_test = criterion(pd_test, y_test)
            acc_test = pytorch_acc(pd_test, y_test)
        test_loss.append(loss_test.detach().numpy())
        test_acc.append(acc_test)
        if best_test_loss > loss_test:
            best_test_loss = loss_test
        if best_test_acc < acc_test:
            best_test_acc = acc_test
        if epoch % eval_freq == 0 or epoch == (max_epochs - 1):
            print(f'SGD Epoch {epoch}:')
            print(f'Average Train Loss: {avg_loss} | '
                  f'Average Train Accuracy: {avg_acc}')
            print(f'Average Test Loss: {loss_test} | '
                  f'Average Test Accuracy: {acc_test}')
            print()
    draw(train_loss, test_loss, train_acc, test_acc, epochs)
    print(f'Best Train Loss: {best_train_loss} | '
          f'Best Train Accuracy: {best_train_acc}\n'
          f'Best Test Loss: {best_test_loss} | '
          f'Best Test Accuracy: {best_test_acc}')
```

In task 1&2, I test `make_moons` dataset and 3 other datasets: `make_blobs`, `make_circles`, and `make_classification`. All these datasets can be found in https://scikit-learn.org/stable/modules/classes.html#module-sklearn.datasets.

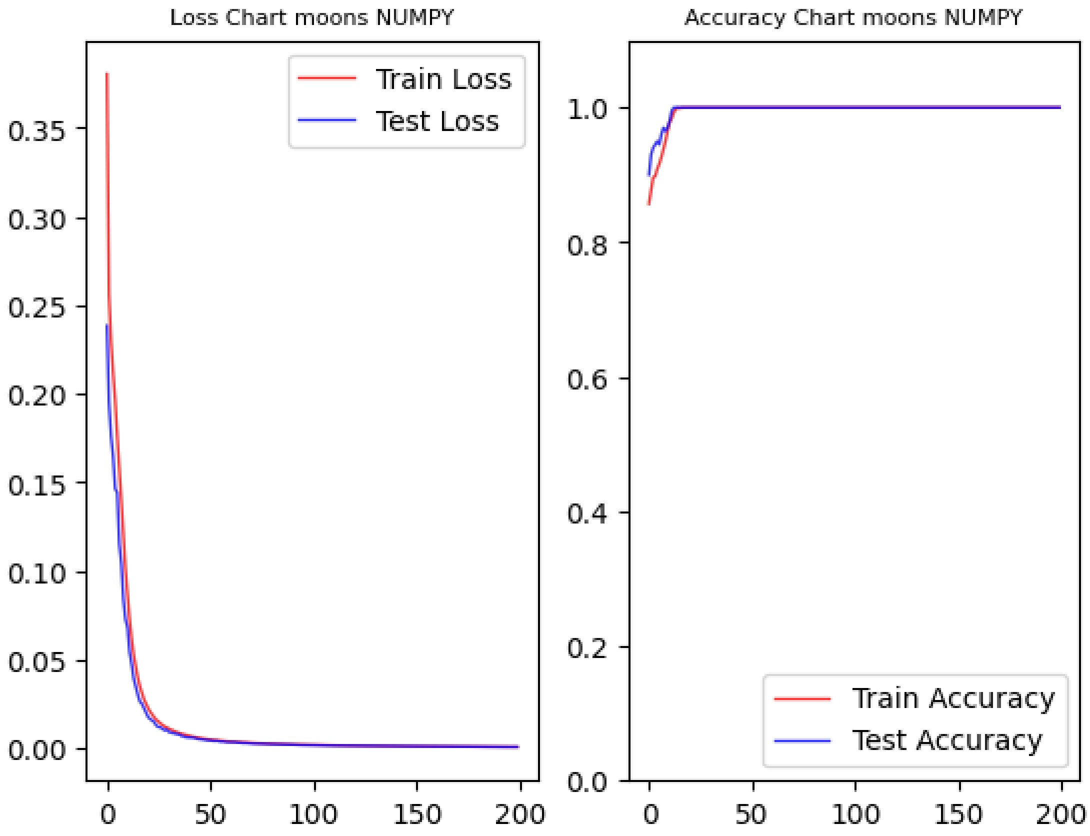
make_moons dataset

```
inputs, labels = datasets.make_moons(n_samples=1000, shuffle=True)
plt.scatter(inputs[:, 0], inputs[:, 1], c=labels)
plt.show()
DATASET = "moons"
```



```
FRAMEWORK = "NUMPY"
numpy_train(inputs, labels)
```

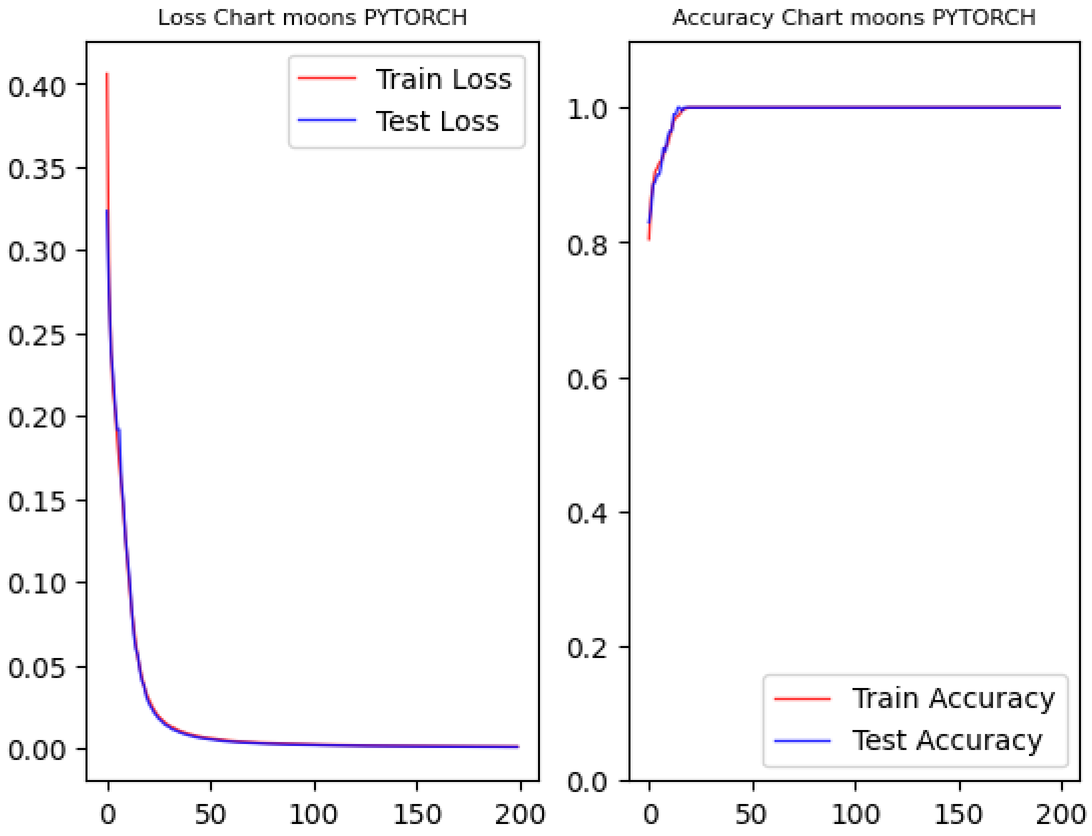
```
SGD Epoch 0:
Average Train Loss: 0.3719298144823714 | Average Train Accuracy: 0.84
Average Test Loss: 0.2933044605294045 | Average Test Accuracy: 0.855

SGD Epoch 40:
Average Train Loss: 0.008089377698839887 | Average Train Accuracy: 1.0
Average Test Loss: 0.008471399137188257 | Average Test Accuracy: 1.0

SGD Epoch 80:
Average Train Loss: 0.002758009660743689 | Average Train Accuracy: 1.0
Average Test Loss: 0.002940851750078023 | Average Test Accuracy: 1.0

SGD Epoch 120:
Average Train Loss: 0.0015850267855487235 | Average Train Accuracy: 1.0
Average Test Loss: 0.0016984141464052803 | Average Test Accuracy: 1.0

SGD Epoch 160:
Average Train Loss: 0.0010913521575490037 | Average Train Accuracy: 1.0
Average Test Loss: 0.0011871034323700197 | Average Test Accuracy: 1.0

SGD Epoch 199:
Average Train Loss: 0.0008298629278206548 | Average Train Accuracy: 1.0
Average Test Loss: 0.0009102350629405885 | Average Test Accuracy: 1.0
```



```
Best Train Loss: 0.0008298629278206548 | Best Train Accuracy: 1.0
Best Test Loss: 0.0009102350629405885 | Best Test Accuracy: 1.0
```

```
FRAMEWORK = "PYTORCH"
pytorch_train(inputs, labels)
```

```
SGD Epoch 0:
Average Train Loss: 0.3785879611968994 | Average Train Accuracy: 0.8475
Average Test Loss: 0.27407553791999817 | Average Test Accuracy: 0.875

SGD Epoch 40:
Average Train Loss: 0.006052356213331225 | Average Train Accuracy: 1.0
Average Test Loss: 0.00528438668698072 | Average Test Accuracy: 1.0

SGD Epoch 80:
Average Train Loss: 0.0021823514252901077 | Average Train Accuracy: 1.0
Average Test Loss: 0.0018738254439085722 | Average Test Accuracy: 1.0

SGD Epoch 120:
Average Train Loss: 0.0012723281979560852 | Average Train Accuracy: 1.0
Average Test Loss: 0.0010898143518716097 | Average Test Accuracy: 1.0

SGD Epoch 160:
Average Train Loss: 0.0008812767337076366 | Average Train Accuracy: 1.0
Average Test Loss: 0.0007546330452896655 | Average Test Accuracy: 1.0

SGD Epoch 199:
Average Train Loss: 0.0006725723505951464 | Average Train Accuracy: 1.0
Average Test Loss: 0.0005744317313656211 | Average Test Accuracy: 1.0
```

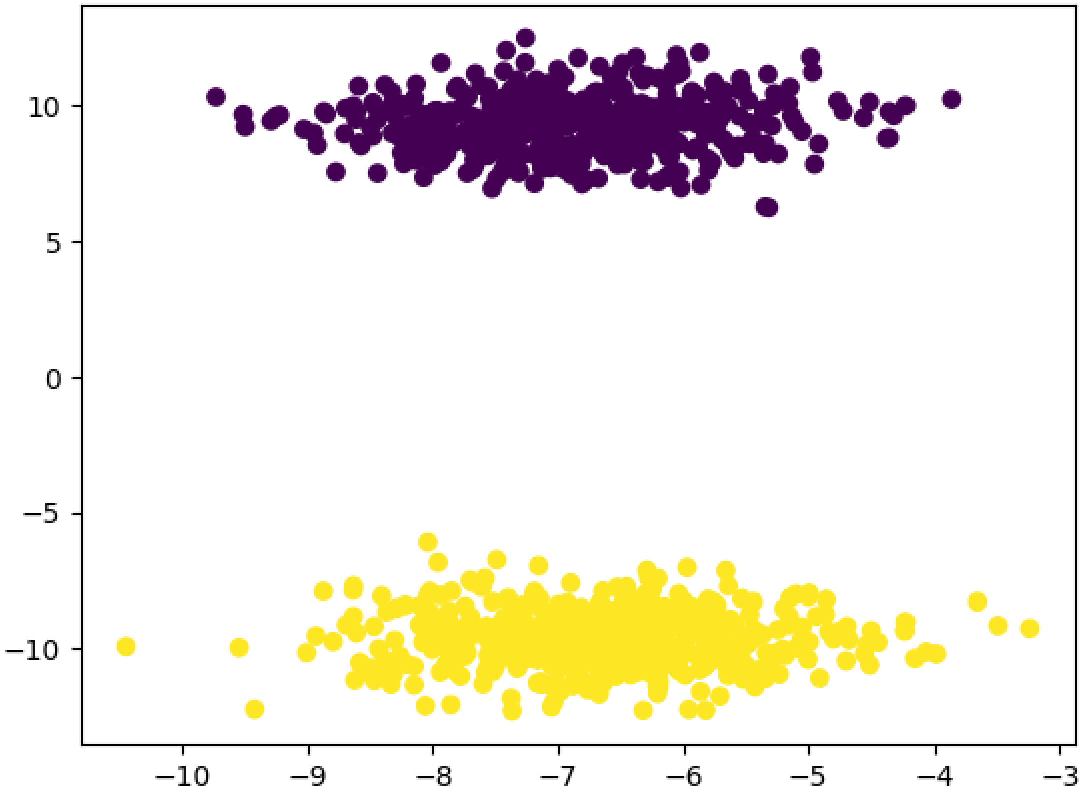


```
Best Train Loss: 0.0006725723505951464 | Best Train Accuracy: 1.0
Best Test Loss: 0.0005744317313656211 | Best Test Accuracy: 1.0
```

Using **make_moons** dataset, we can find that the training loss and testing loss drop to the bottom and the accuracy can reach 100% because the dataset is easy to be separated. Both Numpy and PyTorch show the same tendency.

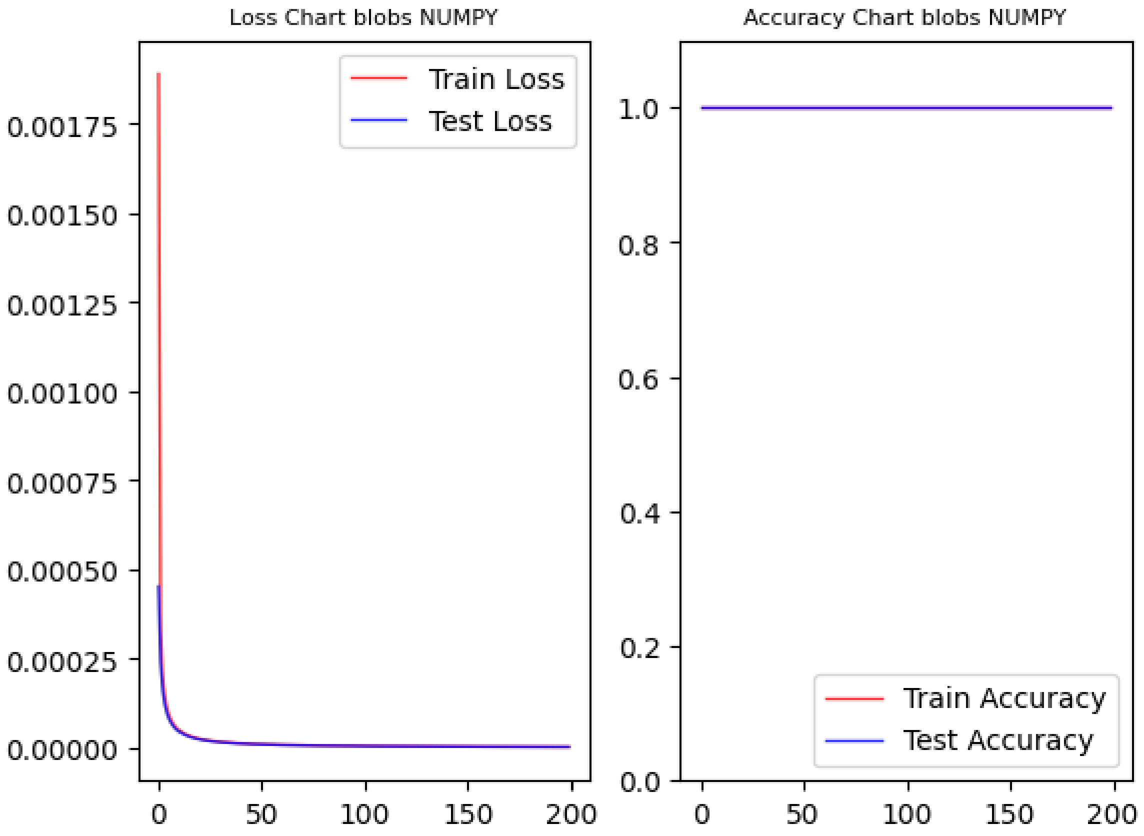
make_blobs dataset

```
inputs, labels = datasets.make_blobs(n_samples=1000, centers=2, n_features=2)
plt.scatter(inputs[:, 0], inputs[:, 1], c=labels)
plt.show()
DATASET = "blobs"
```



```
FRAMEWORK = "NUMPY"
numpy_train(inputs, labels)
```

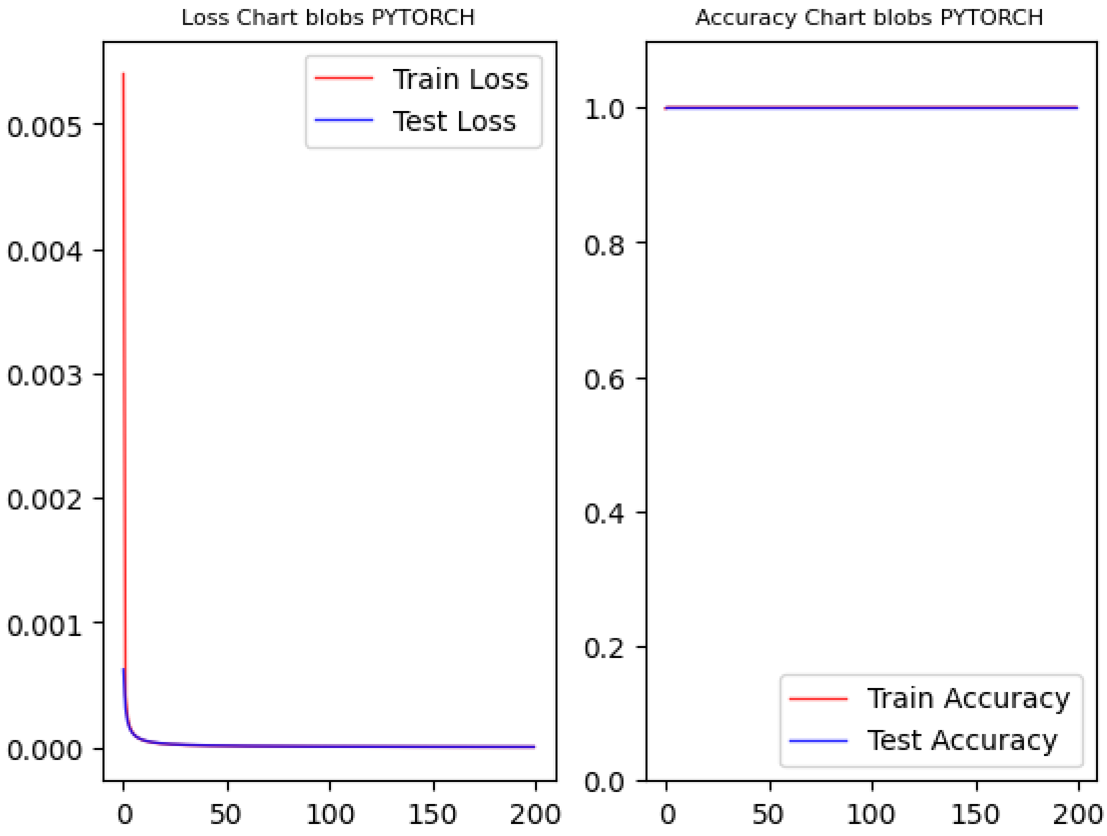
```
SGD Epoch 0:
Average Train Loss: 0.0018897718183546615 | Average Train Accuracy: 1.0
Average Test Loss: 0.0004516992956648647 | Average Test Accuracy: 1.0

SGD Epoch 40:
Average Train Loss: 1.253801184723641e-05 | Average Train Accuracy: 1.0
Average Test Loss: 1.1727658993337538e-05 | Average Test Accuracy: 1.0

SGD Epoch 80:
Average Train Loss: 6.232625788364318e-06 | Average Train Accuracy: 1.0
Average Test Loss: 5.849153950994355e-06 | Average Test Accuracy: 1.0

SGD Epoch 120:
Average Train Loss: 4.131346462952181e-06 | Average Train Accuracy: 1.0
Average Test Loss: 3.881733181164156e-06 | Average Test Accuracy: 1.0

SGD Epoch 160:
Average Train Loss: 3.0836076623684158e-06 | Average Train Accuracy: 1.0
Average Test Loss: 2.8996688618687792e-06 | Average Test Accuracy: 1.0

SGD Epoch 199:
Average Train Loss: 2.4694811967319023e-06 | Average Train Accuracy: 1.0
Average Test Loss: 2.3237849794853655e-06 | Average Test Accuracy: 1.0
```



```
Best Train Loss: 2.4694811967319023e-06 | Best Train Accuracy: 1.0
Best Test Loss: 2.3237849794853655e-06 | Best Test Accuracy: 1.0
```

```
FRAMEWORK = "PYTORCH"
pytorch_train(inputs, labels)
```

```
SGD Epoch 0:
Average Train Loss: 0.005400040186941624 | Average Train Accuracy: 0.99875
Average Test Loss: 0.0006236682529561222 | Average Test Accuracy: 1.0

SGD Epoch 40:
Average Train Loss: 1.3443022908177227e-05 | Average Train Accuracy: 1.0
Average Test Loss: 1.4979317711549811e-05 | Average Test Accuracy: 1.0

SGD Epoch 80:
Average Train Loss: 6.617953204113292e-06 | Average Train Accuracy: 1.0
Average Test Loss: 7.588705557282083e-06 | Average Test Accuracy: 1.0

SGD Epoch 120:
Average Train Loss: 4.364206233731238e-06 | Average Train Accuracy: 1.0
Average Test Loss: 5.093735126138199e-06 | Average Test Accuracy: 1.0

SGD Epoch 160:
Average Train Loss: 3.2463403840665706e-06 | Average Train Accuracy: 1.0
Average Test Loss: 3.835511506622424e-06 | Average Test Accuracy: 1.0

SGD Epoch 199:
Average Train Loss: 2.5953168005798943e-06 | Average Train Accuracy: 1.0
Average Test Loss: 3.092852466579643e-06 | Average Test Accuracy: 1.0
```

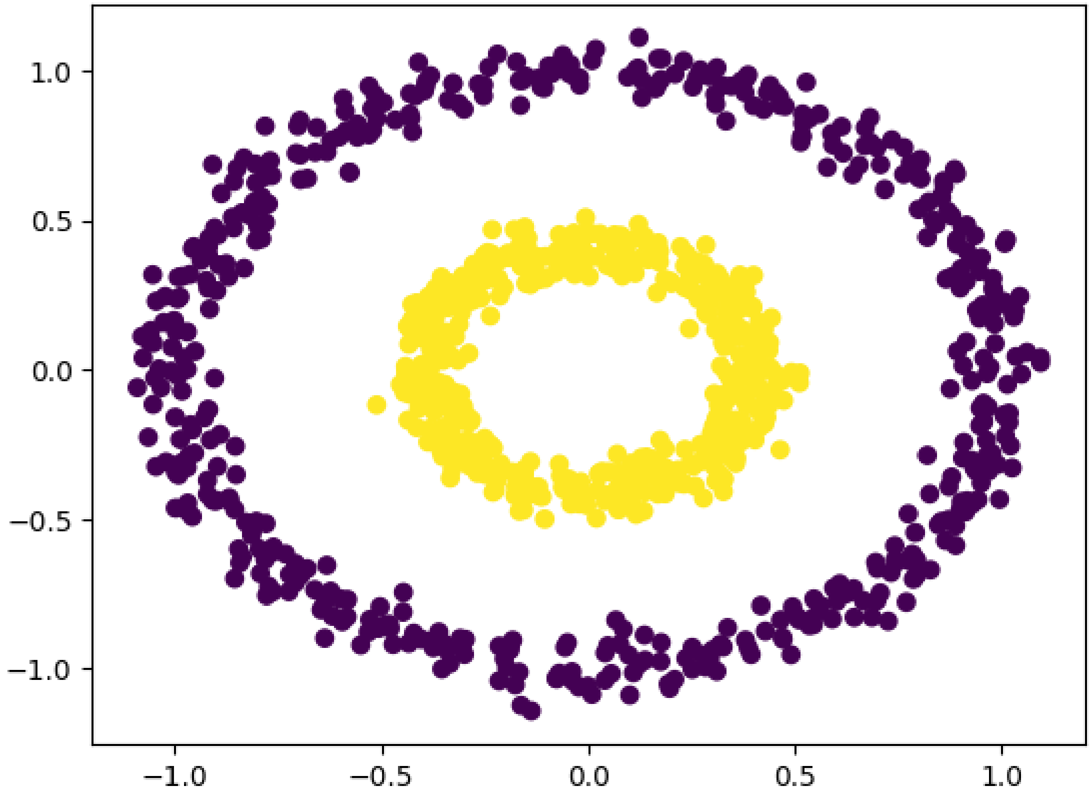


```
Best Train Loss: 2.5953168005798943e-06 | Best Train Accuracy: 1.0
Best Test Loss: 3.092852466579643e-06 | Best Test Accuracy: 1.0
```

Using **make_blobs** dataset, we can find that the training loss and testing loss drop to the bottom and the accuracy can reach 100% because the dataset is easy to be separated. Both Numpy and PyTorch show the same tendency.

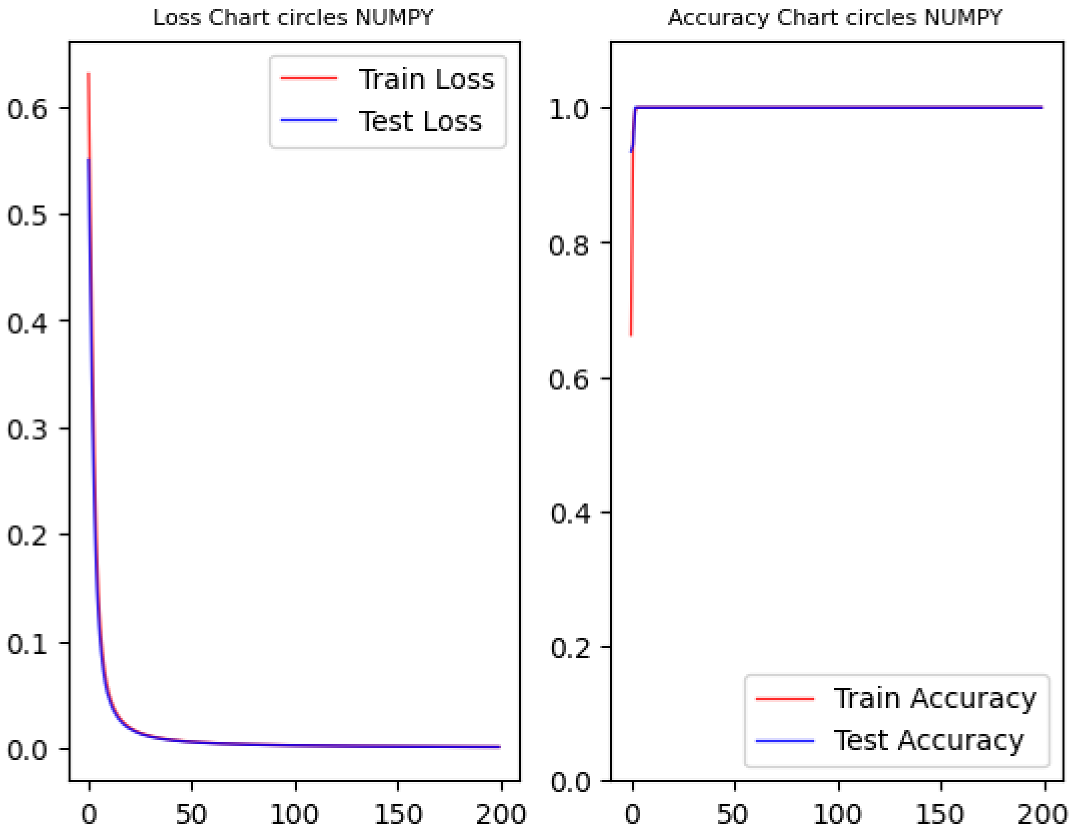
make_circles dataset

```
inputs, labels = datasets.make_circles(n_samples=1000, factor=0.4, noise=0.05)
plt.scatter(inputs[:, 0], inputs[:, 1], c=labels)
plt.show()
DATASET = "circles"
```



```
FRAMEWORK = "NUMPY"
numpy_train(inputs, labels)
```

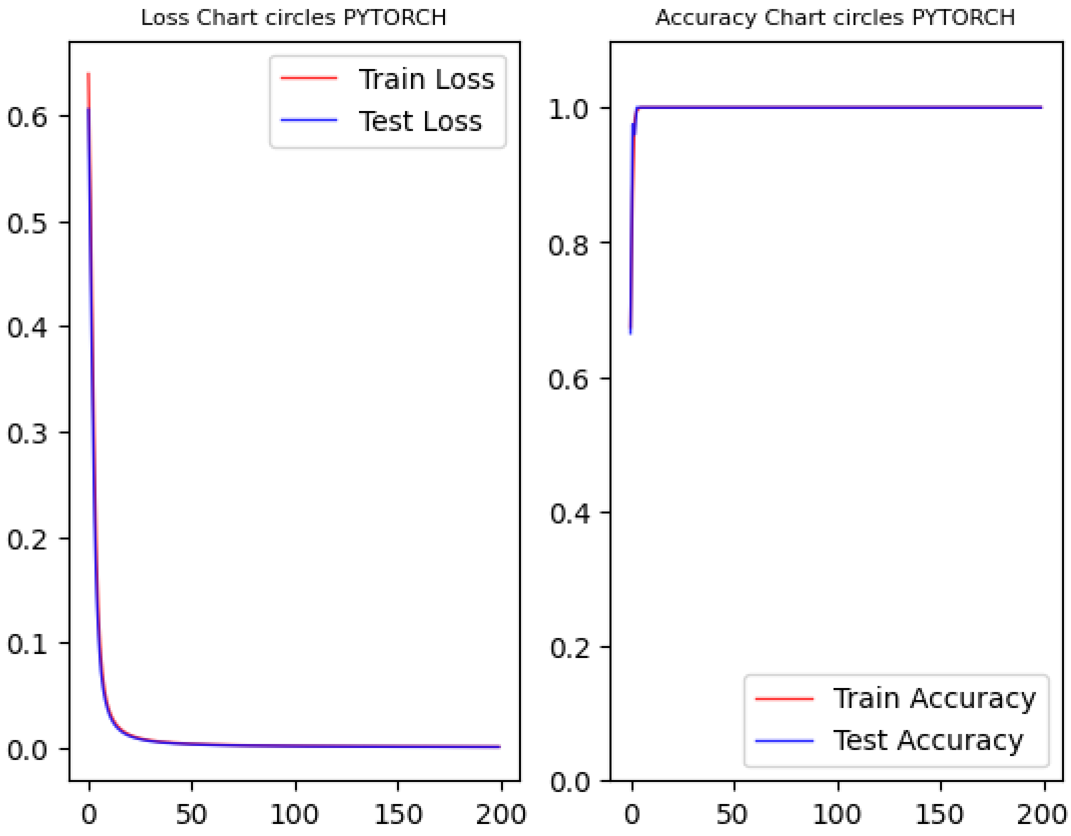
```
SGD Epoch 0:
Average Train Loss: 0.6299879277930301 | Average Train Accuracy: 0.6625
Average Test Loss: 0.5495707778703349 | Average Test Accuracy: 0.935

SGD Epoch 40:
Average Train Loss: 0.007927860146189594 | Average Train Accuracy: 1.0
Average Test Loss: 0.007515641638149795 | Average Test Accuracy: 1.0

SGD Epoch 80:
Average Train Loss: 0.0034358253186707144 | Average Train Accuracy: 1.0
Average Test Loss: 0.003287308547227337 | Average Test Accuracy: 1.0

SGD Epoch 120:
Average Train Loss: 0.0021533530666012343 | Average Train Accuracy: 1.0
Average Test Loss: 0.0020706164310241525 | Average Test Accuracy: 1.0

SGD Epoch 160:
Average Train Loss: 0.0015566008021002714 | Average Train Accuracy: 1.0
Average Test Loss: 0.001500440288456692 | Average Test Accuracy: 1.0

SGD Epoch 199:
Average Train Loss: 0.0012208813533739259 | Average Train Accuracy: 1.0
Average Test Loss: 0.0011772783378801274 | Average Test Accuracy: 1.0
```



```
Best Train Loss: 0.0012208813533739259 | Best Train Accuracy: 1.0
Best Test Loss: 0.0011772783378801274 | Best Test Accuracy: 1.0
```

```
FRAMEWORK = "PYTORCH"
pytorch_train(inputs, labels)
```

```
SGD Epoch 0:
Average Train Loss: 0.6389740109443665 | Average Train Accuracy: 0.67375
Average Test Loss: 0.6052425503730774 | Average Test Accuracy: 0.665

SGD Epoch 40:
Average Train Loss: 0.004631287418305874 | Average Train Accuracy: 1.0
Average Test Loss: 0.004072312265634537 | Average Test Accuracy: 1.0

SGD Epoch 80:
Average Train Loss: 0.001933015533722937 | Average Train Accuracy: 1.0
Average Test Loss: 0.0016458051977679133 | Average Test Accuracy: 1.0

SGD Epoch 120:
Average Train Loss: 0.0011895872885361314 | Average Train Accuracy: 1.0
Average Test Loss: 0.0009886743500828743 | Average Test Accuracy: 1.0

SGD Epoch 160:
Average Train Loss: 0.0008497959934175014 | Average Train Accuracy: 1.0
Average Test Loss: 0.0006950797978788614 | Average Test Accuracy: 1.0

SGD Epoch 199:
Average Train Loss: 0.0006609730189666152 | Average Train Accuracy: 1.0
Average Test Loss: 0.0005348223494365811 | Average Test Accuracy: 1.0
```

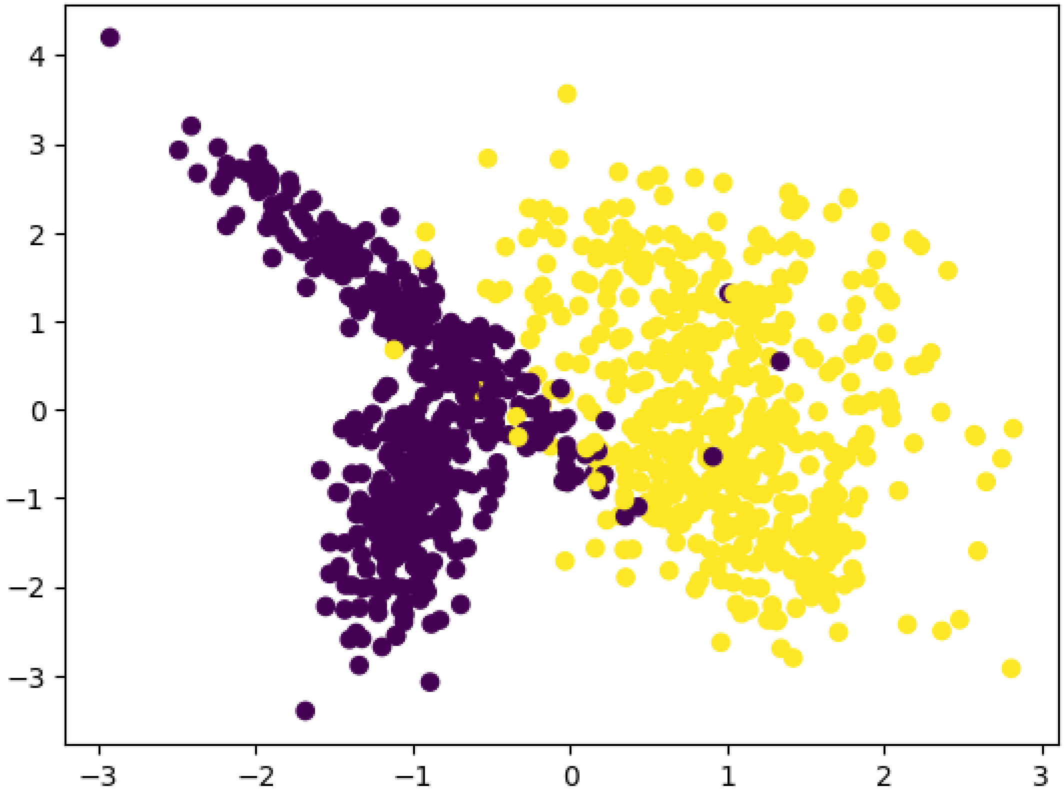


```
Best Train Loss: 0.0006609730189666152 | Best Train Accuracy: 1.0
Best Test Loss: 0.0005348223494365811 | Best Test Accuracy: 1.0
```

Using **make_circles** dataset, we can find that the training loss and testing loss drop to the bottom and the accuracy can reach 100% because the dataset is easy to be separated. Both Numpy and PyTorch show the same tendency.

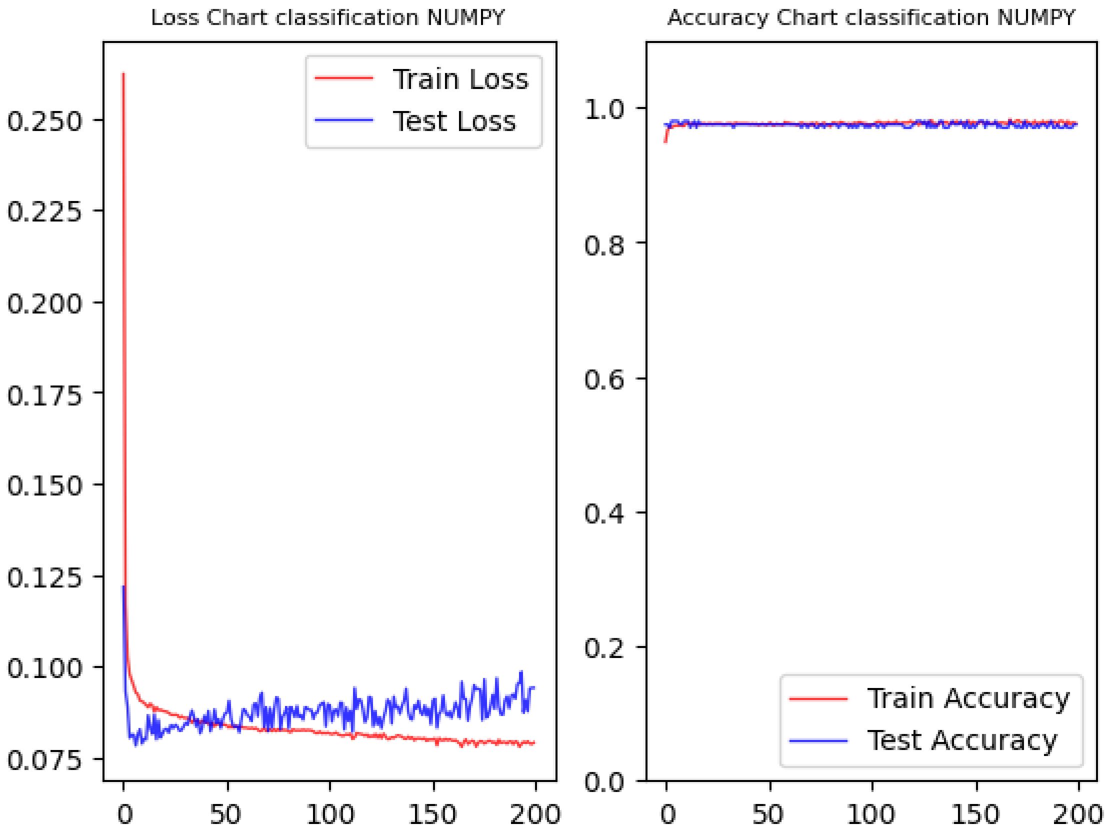
make_classification dataset

```
inputs, labels = datasets.make_classification(n_samples=1000, n_features=2, n_informative=2,
n_redundant=0,
                                              n_repeated=0, hypercube=True)
plt.scatter(inputs[:, 0], inputs[:, 1], c=labels)
plt.show()
DATASET = "classification"
```



```
FRAMEWORK = "NUMPY"
numpy_train(inputs, labels)
```

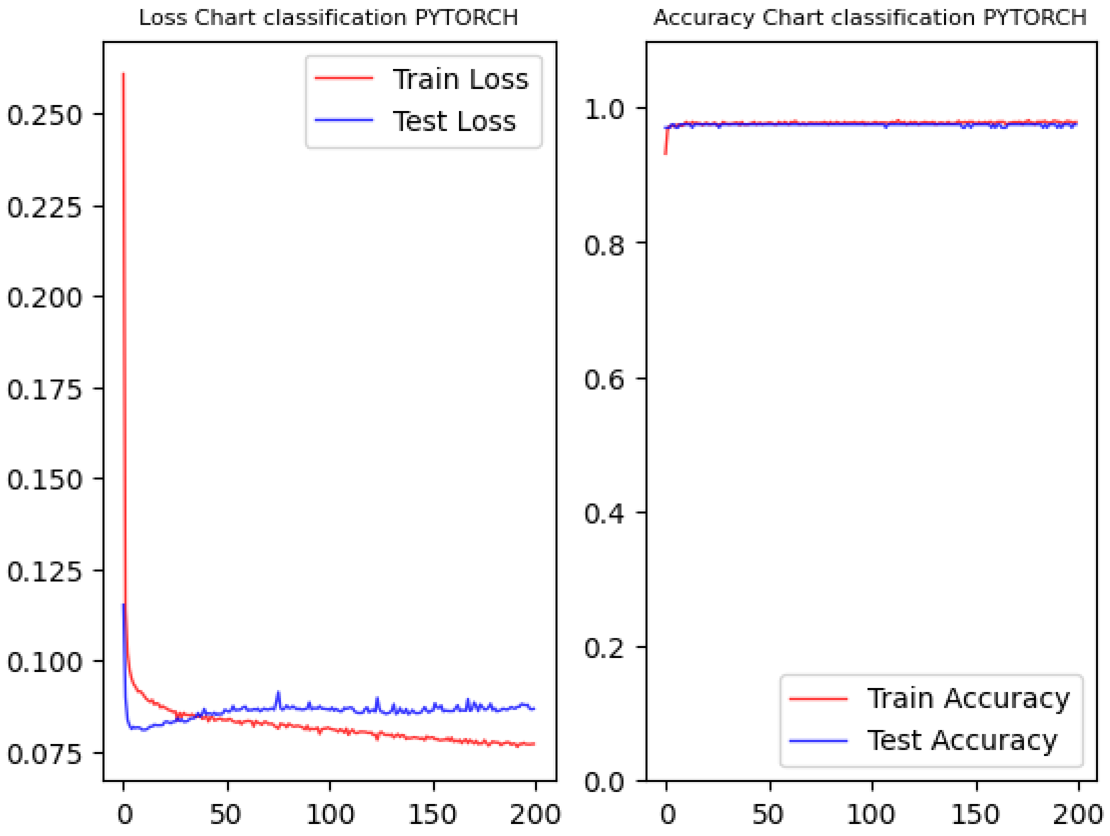
```
SGD Epoch 0:
Average Train Loss: 0.26220999082886104 | Average Train Accuracy: 0.94875
Average Test Loss: 0.12164281037764244 | Average Test Accuracy: 0.975

SGD Epoch 40:
Average Train Loss: 0.0845688339369296 | Average Train Accuracy: 0.97625
Average Test Loss: 0.08452599543953408 | Average Test Accuracy: 0.975

SGD Epoch 80:
Average Train Loss: 0.08209323341367311 | Average Train Accuracy: 0.97125
Average Test Loss: 0.08297062728515141 | Average Test Accuracy: 0.975

SGD Epoch 120:
Average Train Loss: 0.0809786337386276 | Average Train Accuracy: 0.9775
Average Test Loss: 0.09133954712947233 | Average Test Accuracy: 0.975

SGD Epoch 160:
Average Train Loss: 0.07956255821709204 | Average Train Accuracy: 0.9775
Average Test Loss: 0.08767695057692915 | Average Test Accuracy: 0.98

SGD Epoch 199:
Average Train Loss: 0.0790571905273088 | Average Train Accuracy: 0.97625
Average Test Loss: 0.0940700464549425 | Average Test Accuracy: 0.975
```



```
Best Train Loss: 0.07784277106537366 | Best Train Accuracy: 0.9825
Best Test Loss: 0.07820621669268285 | Best Test Accuracy: 0.98
```

```
FRAMEWORK = "PYTORCH"
pytorch_train(inputs, labels)
```

```
SGD Epoch 0:
Average Train Loss: 0.26082754135131836 | Average Train Accuracy: 0.93125
Average Test Loss: 0.11507034301757812 | Average Test Accuracy: 0.97

SGD Epoch 40:
Average Train Loss: 0.08474588394165039 | Average Train Accuracy: 0.975
Average Test Loss: 0.08517061918973923 | Average Test Accuracy: 0.975

SGD Epoch 80:
Average Train Loss: 0.08254282176494598 | Average Train Accuracy: 0.97875
Average Test Loss: 0.08686026930809021 | Average Test Accuracy: 0.975

SGD Epoch 120:
Average Train Loss: 0.08020398765802383 | Average Train Accuracy: 0.97625
Average Test Loss: 0.08620160818099976 | Average Test Accuracy: 0.975

SGD Epoch 160:
Average Train Loss: 0.0776338055729866 | Average Train Accuracy: 0.9775
Average Test Loss: 0.08601728081703186 | Average Test Accuracy: 0.97

SGD Epoch 199:
Average Train Loss: 0.076995350420475 | Average Train Accuracy: 0.97875
Average Test Loss: 0.08666889369487762 | Average Test Accuracy: 0.975
```



```
Best Train Loss: 0.0761747807264328 | Best Train Accuracy: 0.98125
Best Test Loss: 0.0808667466044426 | Best Test Accuracy: 0.975
```

Using **make_classification** dataset, we can find that both Numpy and PyTorch show the same tendency. The accuracies of these two frameworks converge to about 97% because this dataset is not very easy to separate different labels.

Task 3

In this task, I use MLP to classify CIFAR10. The CIFAR10 dataset is loaded by `torchvision.dataset.CIFAR10`. To train the model better, I apply `transforms.Normalize((0.5, 0.5, 0.5), (0.5, 0.5, 0.5))` to the dataset and set the size of batch 512. The whole dataset can be separated to the train dataset and test dataset.

```
batch_size = 512
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5, 0.5, 0.5),
(0.5, 0.5, 0.5))])
train_dataset = torchvision.datasets.CIFAR10(root='data', train=True, download=True,
transform=transform)
train_loader = DataLoader(train_dataset, batch_size, shuffle=True, num_workers=2)
test_dataset = torchvision.datasets.CIFAR10(root='data', train=False, download=False,
transform=transform)
test_loader = DataLoader(test_dataset, batch_size, shuffle=False, num_workers=2)
```

```
class MLP(nn.Module):
    def __init__(self, input_size, hidden_size, num_classes):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(input_size, hidden_size)
        self.relu1 = nn.ReLU()
        self.fc2 = nn.Linear(hidden_size, hidden_size)
        self.relu2 = nn.ReLU()
        self.fc3 = nn.Linear(hidden_size, num_classes)

    def forward(self, x):
        out = self.fc1(x)
        out = self.relu1(out)
        out = self.fc2(out)
        out = self.relu2(out)
        out = self.fc3(out)
        return out
```

To accelerate the training speed, I use GPU to train the model.

```
cuda_id = 0
device_name = "cuda:{}".format(cuda_id) if torch.cuda.is_available() else "cpu"
device = torch.device(device_name)

input_size = 3072
hidden_size = 512
num_classes = 10
model = MLP(input_size, hidden_size, num_classes)
model = model.to(device)
model
```

The framework of MLP is shown below.

```
MLP(
  (fc1): Linear(in_features=3072, out_features=512, bias=True)
  (relu1): ReLU()
  (fc2): Linear(in_features=512, out_features=512, bias=True)
  (relu2): ReLU()
  (fc3): Linear(in_features=512, out_features=10, bias=True)
)
```

The `test` method can test the accuracy of model in the test set.

```
def test(model, loader):
    model.eval()
    correct = 0
    total = 0
    for images, labels in loader:
        images = images.reshape(-1, input_size)
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print('Test Accuracy of the model on the {} test images: {:.2f} %'.format(total, 100 *
correct / total))
    return 100 * correct / total
```

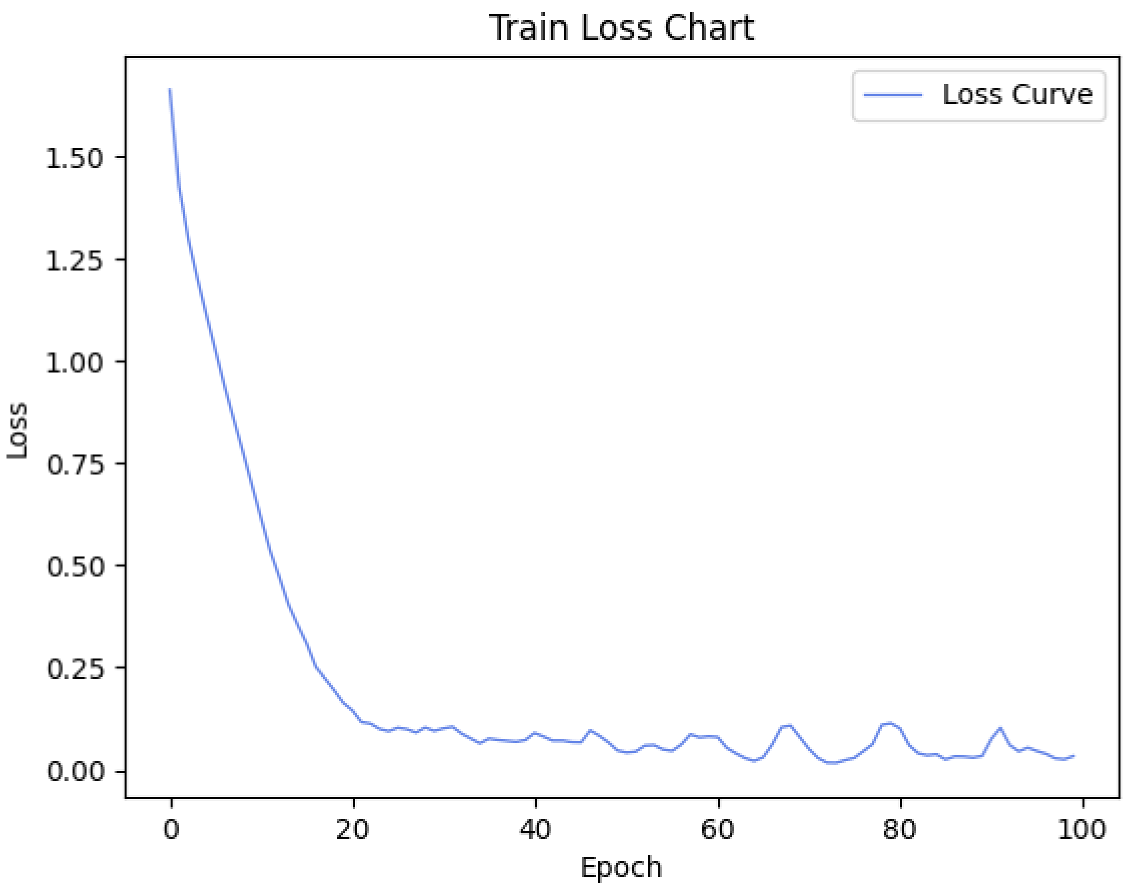
In this task, I use `CrossEntropyLoss` as the loss function and `Adam` as the optimizer. The learning rate is 0.001, and the number of epochs is 100 (the model has already converged in epoch 100).

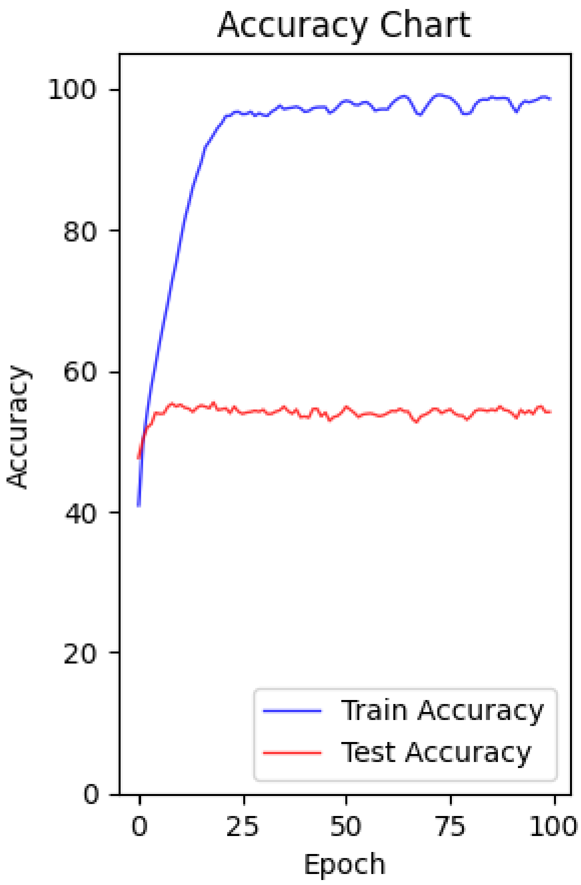
```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.001)

epochs = 100
losses = []
train_accs = []
test_accs = []
for epoch in range(epochs):
    model.train()
    total_loss = 0
    total_correct = 0
    for i, (images, labels) in enumerate(train_loader):
        images = images.reshape(-1, input_size)
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        _, predicted = torch.max(outputs.data, 1)
        correct = (predicted == labels).sum().item()
        total_loss += loss.item()
        total_correct += correct
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    losses.append(total_loss / len(train_loader))
    acc = total_correct / (batch_size * len(train_loader)) * 100
    train_accs.append(acc)
    print('Epoch [{}/{}], Loss: {:.4f}, Accuracy: {:.2f}%'
          .format(epoch + 1, epochs, total_loss / len(train_loader), acc))
    test_acc = test(model, test_loader)
    test_accs.append(test_acc)

# The following codes will print
plt.plot([i for i in range(epochs)], losses, '-', color='#4169E1', alpha=0.8, linewidth=1,
label="Loss Curve")
plt.legend(loc="upper right")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Train Loss Chart")
plt.show()

plt.figure()
plt.subplot(1, 2, 1)
plt.plot([i for i in range(epochs)], train_accs, '-', color='blue', alpha=0.8, linewidth=1,
label="Train Accuracy")
plt.legend(loc="lower right")
plt.plot([i for i in range(epochs)], test_accs, '-', color='red', alpha=0.8, linewidth=1,
label="Test Accuracy")
plt.legend(loc="lower right")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.ylim([0, 105])
plt.title("Accuracy Chart")
plt.show()
```

The training result is shown below.

```
Epoch [1/100], Loss: 1.6620, Accuracy: 40.84%
```

```
Test Accuracy of the model on the 10000 test images: 47.59 %
Epoch [2/100], Loss: 1.4237, Accuracy: 49.88%
Test Accuracy of the model on the 10000 test images: 50.26 %
Epoch [3/100], Loss: 1.2984, Accuracy: 54.14%
Test Accuracy of the model on the 10000 test images: 51.93 %
Epoch [4/100], Loss: 1.2013, Accuracy: 57.76%
Test Accuracy of the model on the 10000 test images: 52.39 %
Epoch [5/100], Loss: 1.1112, Accuracy: 60.78%
Test Accuracy of the model on the 10000 test images: 54.01 %
Epoch [6/100], Loss: 1.0240, Accuracy: 63.90%
Test Accuracy of the model on the 10000 test images: 53.85 %
Epoch [7/100], Loss: 0.9365, Accuracy: 66.85%
Test Accuracy of the model on the 10000 test images: 53.89 %
...... # the whole result can be seen in part1.ipynb
Epoch [98/100], Loss: 0.0288, Accuracy: 98.77%
Test Accuracy of the model on the 10000 test images: 54.92 %
Epoch [99/100], Loss: 0.0264, Accuracy: 98.79%
Test Accuracy of the model on the 10000 test images: 54.09 %
Epoch [100/100], Loss: 0.0343, Accuracy: 98.52%
Test Accuracy of the model on the 10000 test images: 54.12 %
```





The model has already converged in epoch 100. The accuracy is approximately **54%**. Even though the model shows an almost perfect performance in the training dataset, the model is not good in the test dataset and the accuracy of testing rises hardly any during training. This shows that the model is not robust in the test dataset. The MLP model can not extract the nature of a complex figure, so it is not suitable for this task.

Part 2

Import the necessary libraries.

```
import torch
import torchvision
import torch.nn as nn
import matplotlib.pyplot as plt
from torchvision import transforms
from torch.utils.data import DataLoader
```

I encapsulate a `draw` method, this method can plot and show the loss chart and accuracy chart.

```
def draw(epochs, losses, train_accs, test_accs):
    plt.plot([i for i in range(epochs)], losses, '-', color='#4169E1', alpha=0.8, linewidth=1, 
label="Loss Curve")
    plt.legend(loc="upper right")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Train Loss Chart")
    plt.show()

    plt.figure()
    plt.subplot(1, 2, 1)
    plt.plot([i for i in range(epochs)], train_accs, '-', color='blue', alpha=0.8, linewidth=1,
label="Train Accuracy")
    plt.legend(loc="lower right")
    plt.plot([i for i in range(epochs)], test_accs, '-', color='red', alpha=0.8, linewidth=1,
label="Test Accuracy")
    plt.legend(loc="lower right")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.ylim([0, 105])
    plt.title("Accuracy Chart")
    plt.show()
```

The following codes show the PyTorch structure of the model in this task. The model is a version of reduced VGG. The `ConvBlock` class consists of a `nn.Conv2` block, a `nn.BatchNorm2d` block, and a `nn.ReLU` block. The `CNN` class will use `ConvBlock` and `nn.MaxPool2d` module. After several convolution and max pooling operations, the output should be reshaped and be passed to a `nn.Linear` module to obtain the final result.

```
class ConvBlock(nn.Module):
    def __init__(self, in_channels, out_channels, kernel_size=3, stride=1, padding=1):
        super(ConvBlock, self).__init__()
        self.conv = nn.Conv2d(in_channels, out_channels, kernel_size=kernel_size, stride=stride,
padding=padding)
        self.bn = nn.BatchNorm2d(out_channels)
        self.relu = nn.ReLU(inplace=True)

    def forward(self, x):
        x = self.conv(x)
        x = self.bn(x)
        out = self.relu(x)
        return out


class CNN(nn.Module):

    def __init__(self, n_channels, n_classes):
        super(CNN, self).__init__()
        self.conv1 = ConvBlock(n_channels, 64, kernel_size=3, stride=1, padding=1)
        self.pool2 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv3 = ConvBlock(64, 128, kernel_size=3, stride=1, padding=1)
        self.pool4 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv5 = ConvBlock(128, 256, kernel_size=3, stride=1, padding=1)
        self.conv6 = ConvBlock(256, 256, kernel_size=3, stride=1, padding=1)
        self.pool7 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv8 = ConvBlock(256, 512, kernel_size=3, stride=1, padding=1)
        self.conv9 = ConvBlock(512, 512, kernel_size=3, stride=1, padding=1)
        self.pool10 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.conv11 = ConvBlock(512, 512, kernel_size=3, stride=1, padding=1)
        self.conv12 = ConvBlock(512, 512, kernel_size=3, stride=1, padding=1)
        self.pool13 = nn.MaxPool2d(kernel_size=3, stride=2, padding=1)
        self.fc14 = nn.Linear(512, n_classes)
    def forward(self, x):
        x = self.conv1(x)
        x = self.pool2(x)
        x = self.conv3(x)
        x = self.pool4(x)
        x = self.conv5(x)
        x = self.conv6(x)
        x = self.pool7(x)
        x = self.conv8(x)
        x = self.conv9(x)
        x = self.pool10(x)
        x = self.conv11(x)
        x = self.conv12(x)
        x = self.pool13(x)
        x = x.view(x.size(0), -1)
        out = self.fc14(x)
        return out
```

To accelerate the speed of the training, I use Huawei Cloud GPU resources in this part. The program will try to determine whether the cuda is available or not. If the cuda is available, the program will use GPU to accelerate. It will task about 1min to finish an epoch.

```
cuda_id = 0
device_name = "cuda:{}".format(cuda_id) if torch.cuda.is_available() else "cpu"
device = torch.device(device_name)
```

In this task, we set the input size as 3 and the number class as 10.

```
input_size = 3
num_classes = 10
model = CNN(input_size, num_classes)
model = model.to(device)
model
```

The framework of the model is shown below.

```
CNN(
  (conv1): ConvBlock(
    (conv): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (pool2): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (conv3): ConvBlock(
    (conv): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (pool4): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (conv5): ConvBlock(
    (conv): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (conv6): ConvBlock(
    (conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (pool7): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (conv8): ConvBlock(
    (conv): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (conv9): ConvBlock(
    (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (pool10): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (conv11): ConvBlock(
    (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (conv12): ConvBlock(
    (conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (relu): ReLU(inplace=True)
  )
  (pool13): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)
  (fc14): Linear(in_features=512, out_features=10, bias=True)
)
```

The image shape of the output is calculated by the following equations in the convolutional layer. W means the width, H means the height, F means the kernel size, P means the padding size, and S means the stride.

$$W_{out} = (W_{in} - F_w + 2P_w)/S_w + 1$$

$$H_{out} = (H_{in} - F_h + 2P_h)/S_h + 1$$

The image shape of the output is calculated by the following equations in the pooling layer. W means the width, H means the height, F means the kernel size, P means the padding size, and S means the stride.

$$W_{out} = floor((W_{in} - F_w + 2P_w)/S_w) + 1$$

$$H_{out} = floor((H_{in} - F_h + 2P_h)/S_h) + 1$$

Note: since the dilation here is 1, I just eliminate it. The ceil_mode of the pooling is false, so the value of the division will be rounded down.

The input shape and output shape are shown below. In this case, batch_size is **32**.

Layer	Input Shape	Output shape
(conv1): ConvBlock((conv): Conv2d(3, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (relu): ReLU(inplace=True))	(batch_size, 3, 32, 32)	(batch_size, 64, 32, 32)
(pool2): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)	(batch_size, 64, 32, 32)	(batch_size, 64, 16, 16)

(conv3): ConvBlock((conv): Conv2d(64, 128, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(128, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (relu): ReLU(inplace=True))	(batch_size, 64, 16, 16)	(batch_size, 128, 16, 16)
(pool4): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)	(batch_size, 128, 16, 16)	(batch_size, 128, 8, 8)
(conv5): ConvBlock((conv): Conv2d(128, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (relu): ReLU(inplace=True))	(batch_size, 128, 8, 8)	(batch_size, 256, 8, 8)
(conv6): ConvBlock((conv): Conv2d(256, 256, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(256, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (relu): ReLU(inplace=True))	(batch_size, 256, 8, 8)	(batch_size, 256, 8, 8)
(pool7): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)	(batch_size, 256, 8, 8)	(batch_size, 256, 4, 4)
(conv8): ConvBlock((conv): Conv2d(256, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (relu): ReLU(inplace=True))	(batch_size, 256, 4, 4)	(batch_size, 512, 4, 4)
(conv9): ConvBlock((conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (relu): ReLU(inplace=True))	(batch_size, 512, 4, 4)	(batch_size, 512, 4, 4)
(pool10): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)	(batch_size, 512, 4, 4)	(batch_size, 512, 2, 2)
(conv11): ConvBlock((conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)	(batch_size, 512, 2, 2)	(batch_size, 512, 2, 2)

(relu): ReLU(inplace=True))		
(conv12): ConvBlock((conv): Conv2d(512, 512, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1)) (bn): BatchNorm2d(512, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True) (relu): ReLU(inplace=True))	(batch_size, 512, 2, 2)	(batch_size, 512, 2, 2)
(pool13): MaxPool2d(kernel_size=3, stride=2, padding=1, dilation=1, ceil_mode=False)	(batch_size, 512, 2, 2)	(batch_size, 512, 1, 1)
View Operation	(batch_size, 512, 1, 1)	(batch_size, 512)
(fc14): Linear(in_features=512, out_features=10, bias=True)	(batch_size, 512)	(batch_size, 10)

In this task, I set the epoch as 100 because the server's resources are limited and I am almost running out of time. The loss function is `nn.CrossEntropyLoss`.

```
epochs = 100
criterion = nn.CrossEntropyLoss()
```

Here I implement three useful methods. The `train` method is to train the model by enumerating the train dataset. In each iteration, the model will have an input with shape (batch_size, in_channel, image.size), image.size is a tuple. Then the model will have an output with the shape (batch_size, out_channel). Then compute the loss function from the label and the output. After that, the script will use the autograd mechanism in PyTorch to obtain the gradient of each layer and use the optimizer to update the parameters. The `test` model is to test the model by enumerating the test dataset. The mechanism is similar to the `train` method, but if we use a script `model.eval()`, the model will not compute the gradient so the model will just test the result. The `main` method implements a `for-loop-epoch` skeleton and calls the `train` and `test` methods to train and test the model, finally, this method will visualize the result.

```
def train(model, loader, criterion, optimizer):
    model.train()
    total_loss = 0
    total_correct = 0
    for i, (images, labels) in enumerate(loader):
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        loss = criterion(outputs, labels)
        _, predicted = torch.max(outputs.data, 1)
        correct = (predicted == labels).sum().item()
        total_loss += loss.item()
        total_correct += correct
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
    return total_loss / len(train_loader), total_correct / (batch_size * len(train_loader)) *
100
```

```
def test(model, loader):
    model.eval()
    correct = 0
    total = 0
    for images, labels in loader:
        images, labels = images.to(device), labels.to(device)
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    return 100 * correct / total
```

```
def main(epochs, model, train_loader, test_loader, criterion, optimizer):
    losses = []
    train_accs = []
    test_accs = []
    for epoch in range(epochs):
        loss, train_acc = train(model, train_loader, criterion, optimizer)
        test_acc = test(model, test_loader)
        losses.append(loss)
        train_accs.append(train_acc)
        test_accs.append(test_acc)
        print('Epoch [{}/{}], Loss: {:.4f}, Train Accuracy: {:.2f} %, Test Accuracy: {:.2f} %'
              .format(epoch + 1, epochs, loss, train_acc, test_acc))
    draw(epochs, losses, train_accs, test_accs)
```

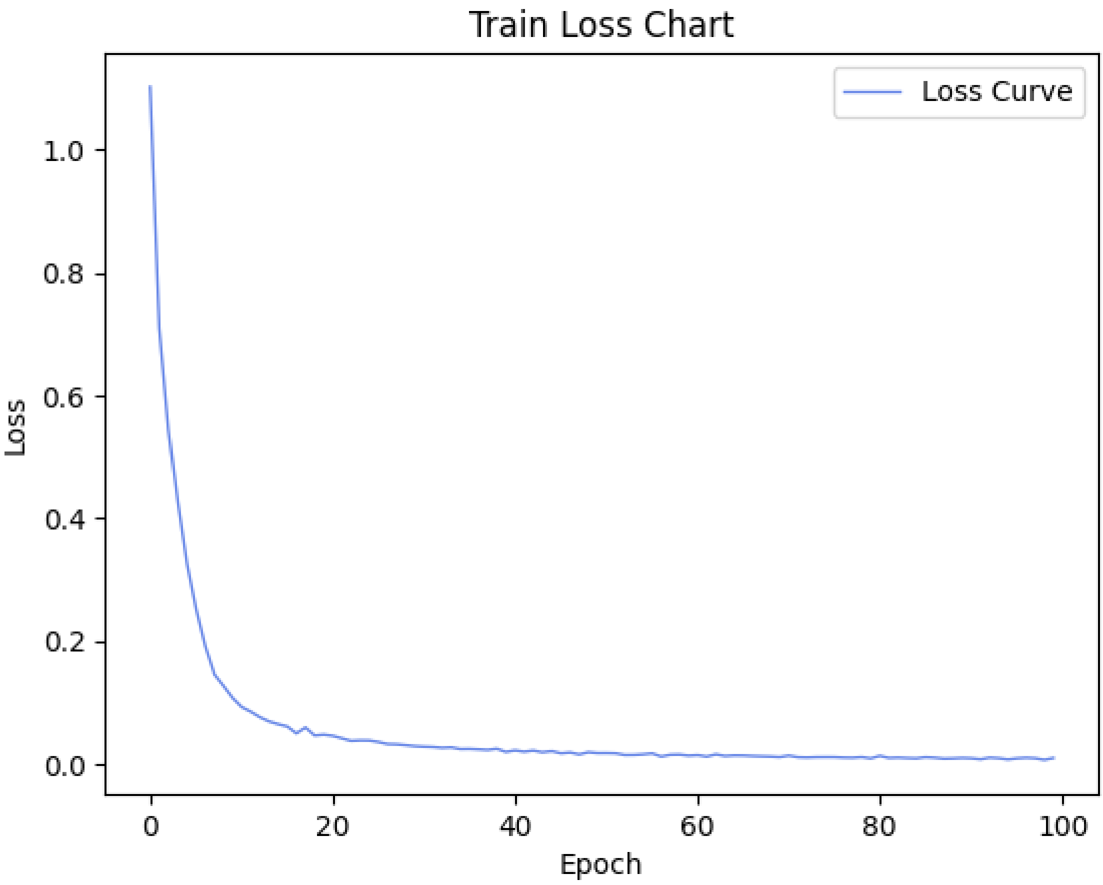
Comparison among different learning rates and optimizers

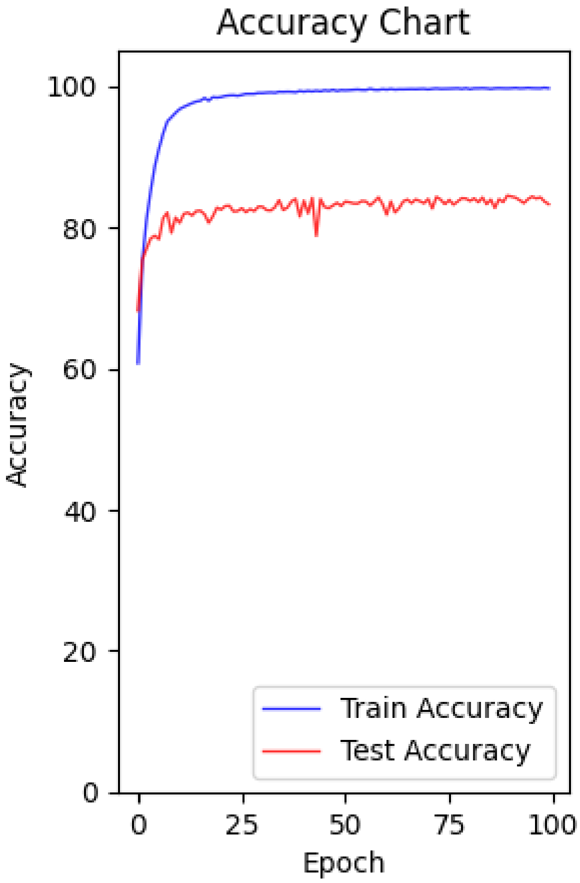
Learning Rate: 1e-4

Optimizer: Adam

```
model = CNN(input_size, num_classes)
model = model.to(device)
lr = 1e-4
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
main(epochs, model, train_loader, test_loader, criterion, optimizer)
```

```
Epoch [1/100], Loss: 1.1024, Train Accuracy: 60.72 %, Test Accuracy: 68.19 %
Epoch [2/100], Loss: 0.7094, Train Accuracy: 75.37 %, Test Accuracy: 75.43 %
Epoch [3/100], Loss: 0.5424, Train Accuracy: 81.18 %, Test Accuracy: 76.99 %
Epoch [4/100], Loss: 0.4251, Train Accuracy: 85.21 %, Test Accuracy: 78.43 %
Epoch [5/100], Loss: 0.3248, Train Accuracy: 88.70 %, Test Accuracy: 78.82 %
Epoch [6/100], Loss: 0.2510, Train Accuracy: 91.20 %, Test Accuracy: 78.31 %
Epoch [7/100], Loss: 0.1909, Train Accuracy: 93.26 %, Test Accuracy: 81.32 %
Epoch [8/100], Loss: 0.1454, Train Accuracy: 94.96 %, Test Accuracy: 82.10 %
...... # the whole result can be seen in part2.ipynb
Epoch [98/100], Loss: 0.0088, Train Accuracy: 99.66 %, Test Accuracy: 84.27 %
Epoch [99/100], Loss: 0.0062, Train Accuracy: 99.76 %, Test Accuracy: 83.67 %
Epoch [100/100], Loss: 0.0092, Train Accuracy: 99.67 %, Test Accuracy: 83.29 %
```





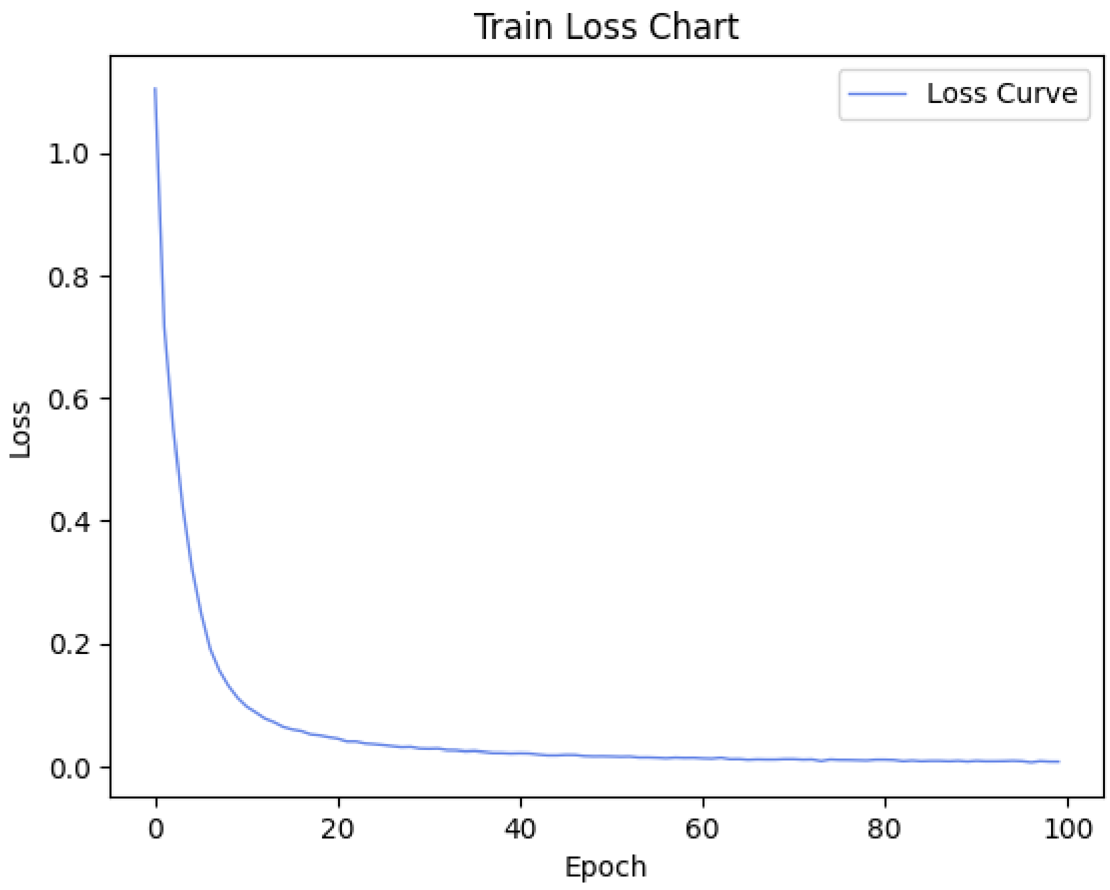
Using the default learning rate and optimizer, we can find that the training loss has a steady descent and finally converges. The training accuracy can reach a peak of 99.76%, and the testing accuracy is larger than 84%. This model shows a better performance than the basic MLP model, which means the model is more capable to classify a complex figure and has a higher robustness. But due to the limit of the GPU resources, the testing accuracy may be larger in an ideal case, but I think the best value it can reach will be smaller than 90% because the training loss is almost 100%, the upside is low.

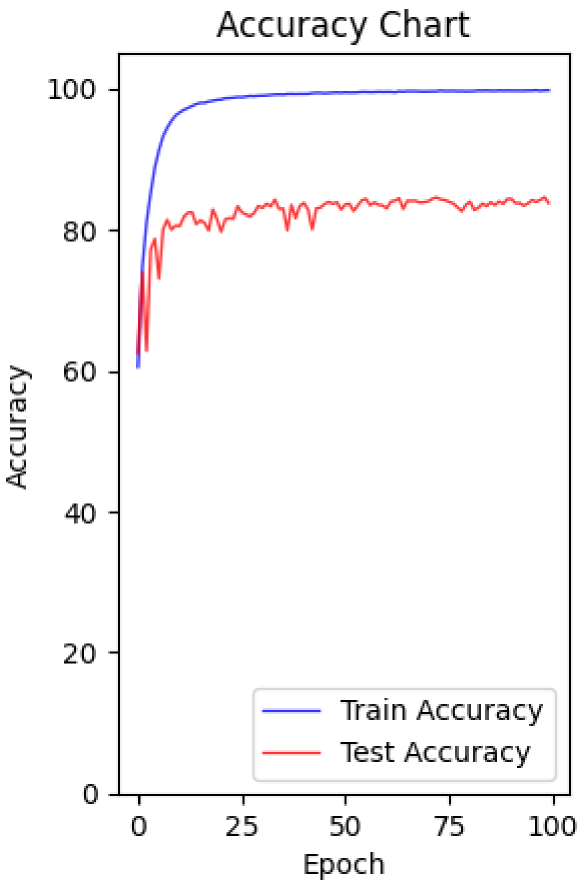
Learning Rate: 1e-4

Optimizer: RMSprop

```
model = CNN(input_size, num_classes)
model = model.to(device)
lr = 1e-4
optimizer = torch.optim.RMSprop(model.parameters(), lr=lr)
main(epochs, model, train_loader, test_loader, criterion, optimizer)
```

```
Epoch [1/100], Loss: 1.1033, Train Accuracy: 60.50 %, Test Accuracy: 62.44 %
Epoch [2/100], Loss: 0.7160, Train Accuracy: 74.79 %, Test Accuracy: 73.96 %
Epoch [3/100], Loss: 0.5476, Train Accuracy: 81.06 %, Test Accuracy: 62.78 %
Epoch [4/100], Loss: 0.4200, Train Accuracy: 85.18 %, Test Accuracy: 77.20 %
Epoch [5/100], Loss: 0.3216, Train Accuracy: 88.79 %, Test Accuracy: 78.71 %
Epoch [6/100], Loss: 0.2487, Train Accuracy: 91.25 %, Test Accuracy: 73.06 %
Epoch [7/100], Loss: 0.1906, Train Accuracy: 93.23 %, Test Accuracy: 80.08 %
Epoch [8/100], Loss: 0.1556, Train Accuracy: 94.43 %, Test Accuracy: 81.42 %
...... # the whole result can be seen in part2.ipynb
Epoch [98/100], Loss: 0.0086, Train Accuracy: 99.66 %, Test Accuracy: 84.20 %
Epoch [99/100], Loss: 0.0078, Train Accuracy: 99.73 %, Test Accuracy: 84.55 %
Epoch [100/100], Loss: 0.0077, Train Accuracy: 99.74 %, Test Accuracy: 83.72 %
```





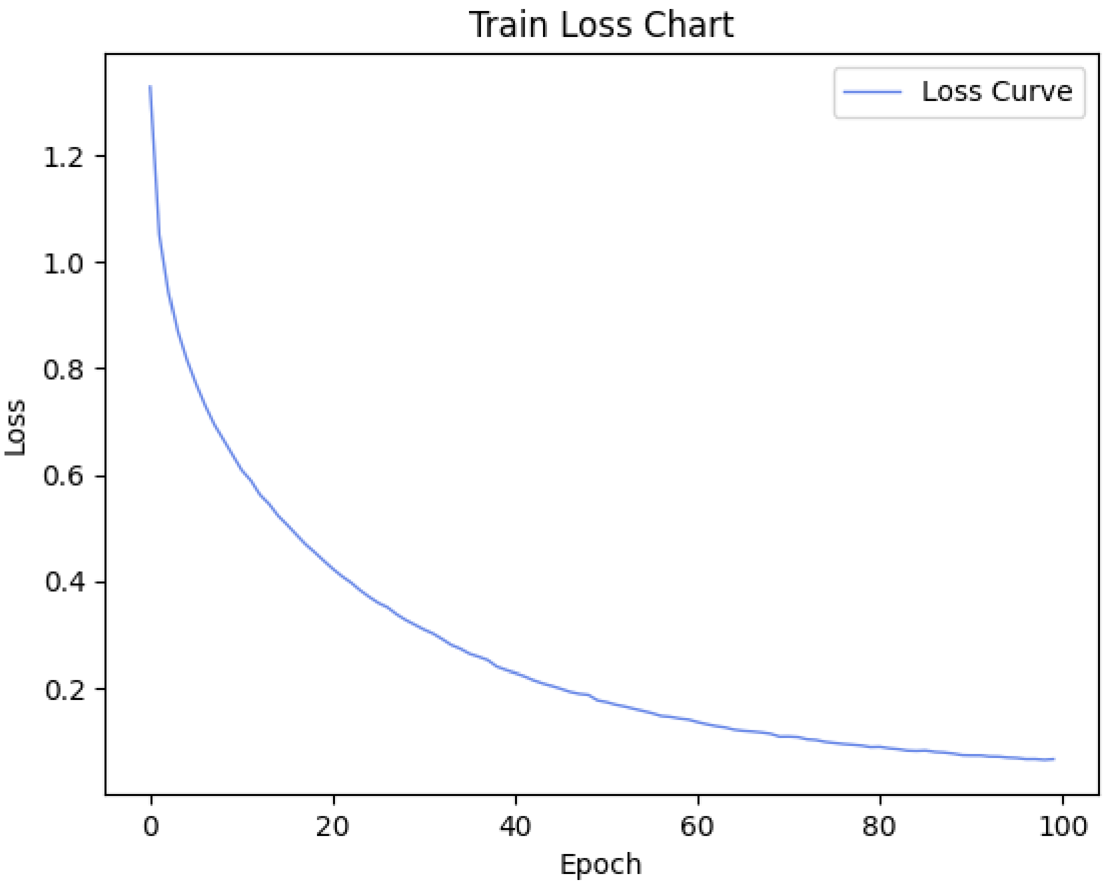
Using the RMSprop optimizer, we can find both the Adam and RMSprop optimizers can have a testing accuracy of 84%. But the convergence speed of the RMSprop optimizer is slower. The difference between these two optimizers can show that both the Adam optimizer and the RMSprop optimizer are useful in classifying the CIFAR10 dataset, but the Adam dataset can help the model converge faster.

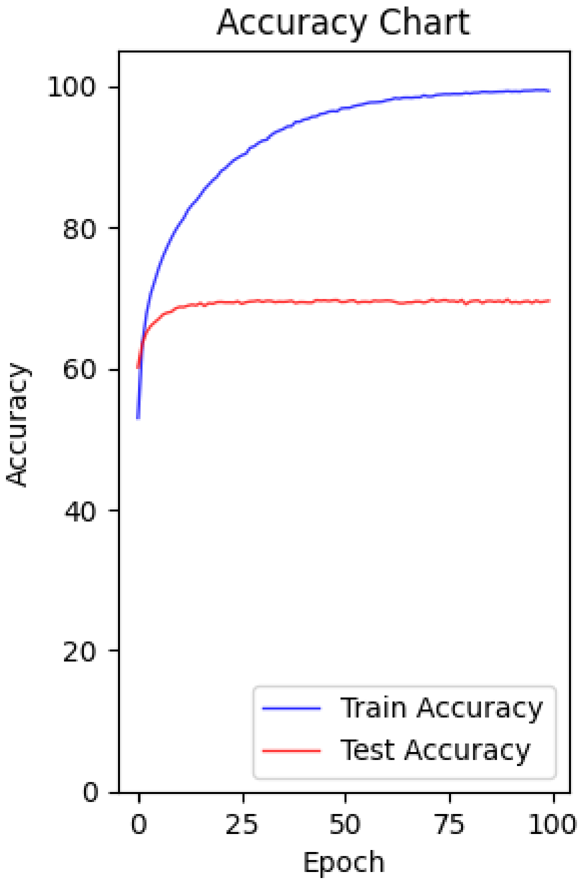
Learning Rate: 1e-4

Optimizer: Adagrad

```
model = CNN(input_size, num_classes)
model = model.to(device)
lr = 1e-4
optimizer = torch.optim.Adagrad(model.parameters(), lr=lr)
main(epochs, model, train_loader, test_loader, criterion, optimizer)
```

```
Epoch [1/100], Loss: 1.3277, Train Accuracy: 52.89 %, Test Accuracy: 60.09 %
Epoch [2/100], Loss: 1.0508, Train Accuracy: 63.55 %, Test Accuracy: 63.47 %
Epoch [3/100], Loss: 0.9426, Train Accuracy: 67.78 %, Test Accuracy: 65.10 %
Epoch [4/100], Loss: 0.8685, Train Accuracy: 70.55 %, Test Accuracy: 65.98 %
Epoch [5/100], Loss: 0.8148, Train Accuracy: 72.41 %, Test Accuracy: 66.51 %
Epoch [6/100], Loss: 0.7697, Train Accuracy: 74.32 %, Test Accuracy: 67.02 %
Epoch [7/100], Loss: 0.7293, Train Accuracy: 75.90 %, Test Accuracy: 67.62 %
Epoch [8/100], Loss: 0.6937, Train Accuracy: 77.24 %, Test Accuracy: 67.86 %
...... # the whole result can be seen in part2.ipynb
Epoch [98/100], Loss: 0.0663, Train Accuracy: 99.39 %, Test Accuracy: 69.35 %
Epoch [99/100], Loss: 0.0648, Train Accuracy: 99.41 %, Test Accuracy: 69.46 %
Epoch [100/100], Loss: 0.0661, Train Accuracy: 99.29 %, Test Accuracy: 69.55 %
```





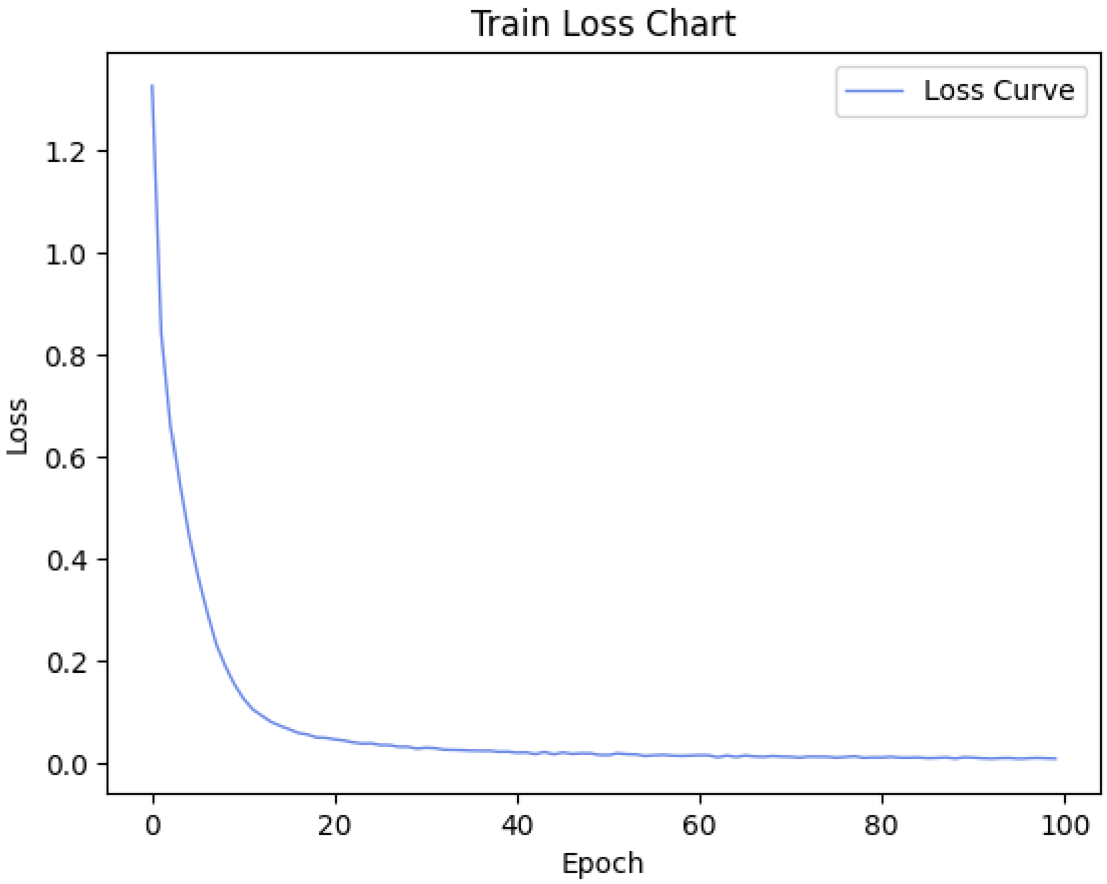
From the accuracy chart, we can find that even though the training accuracy goes up gradually, the testing accuracy just rises a little and continues vibrating between 69% and 70%. The Adagrad optimizer is not as good as the Adam optimizer and the RMSprop optimizer. The training accuracy has reached a peak of 99%, I do not think increasing the number of epochs can contribute to the tetsing accuracy.

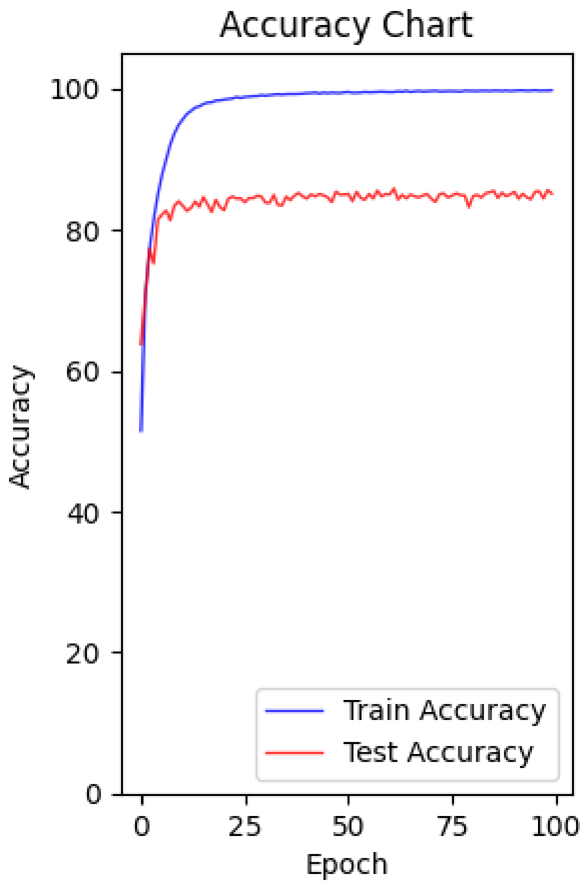
Learning Rate: 1e-3

Optimizer: Adam

```
model = CNN(input_size, num_classes)
model = model.to(device)
lr = 1e-3
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
main(epochs, model, train_loader, test_loader, criterion, optimizer)
```

```
Epoch [1/100], Loss: 1.3272, Train Accuracy: 51.44 %, Test Accuracy: 63.73 %
Epoch [2/100], Loss: 0.8422, Train Accuracy: 70.90 %, Test Accuracy: 70.63 %
Epoch [3/100], Loss: 0.6610, Train Accuracy: 77.32 %, Test Accuracy: 77.28 %
Epoch [4/100], Loss: 0.5464, Train Accuracy: 81.43 %, Test Accuracy: 75.21 %
Epoch [5/100], Loss: 0.4453, Train Accuracy: 84.93 %, Test Accuracy: 81.48 %
Epoch [6/100], Loss: 0.3637, Train Accuracy: 87.71 %, Test Accuracy: 82.09 %
Epoch [7/100], Loss: 0.2935, Train Accuracy: 89.90 %, Test Accuracy: 82.65 %
Epoch [8/100], Loss: 0.2308, Train Accuracy: 92.05 %, Test Accuracy: 81.35 %
...... # the whole result can be seen in part2.ipynb
Epoch [98/100], Loss: 0.0092, Train Accuracy: 99.70 %, Test Accuracy: 84.36 %
Epoch [99/100], Loss: 0.0084, Train Accuracy: 99.69 %, Test Accuracy: 85.57 %
Epoch [100/100], Loss: 0.0078, Train Accuracy: 99.74 %, Test Accuracy: 85.07 %
```



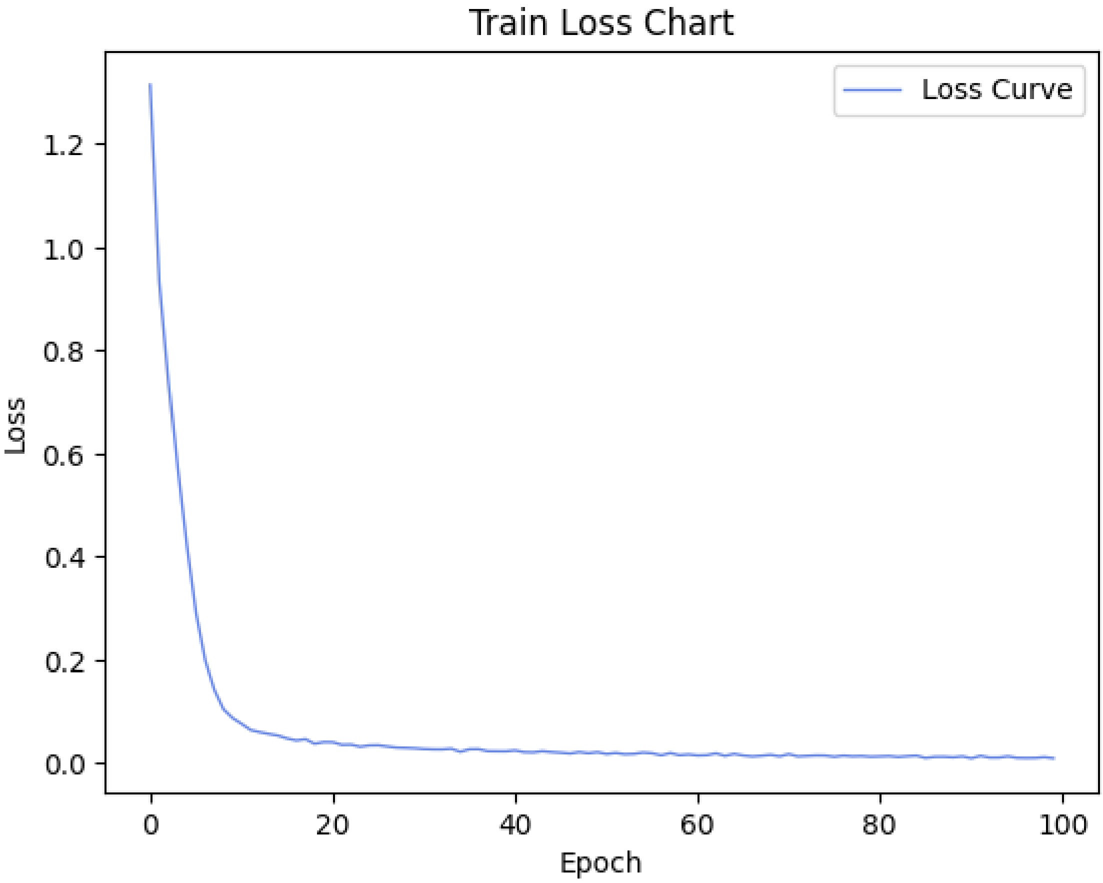


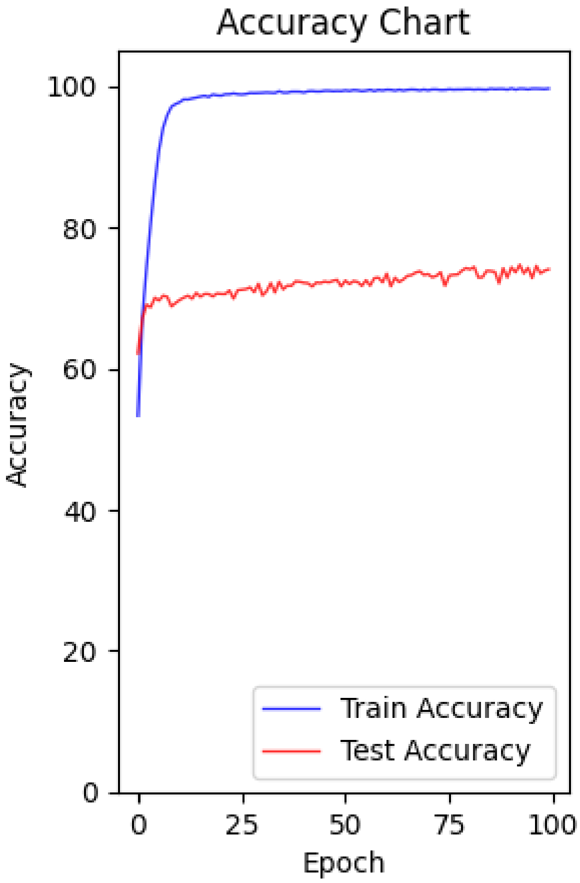
Learning Rate: 1e-5

Optimizer: Adam

```
model = CNN(input_size, num_classes)
model = model.to(device)
lr = 1e-5
optimizer = torch.optim.Adam(model.parameters(), lr=lr)
main(epochs, model, train_loader, test_loader, criterion, optimizer)
```

```
Epoch [1/100], Loss: 1.3145, Train Accuracy: 53.26 %, Test Accuracy: 62.11 %
Epoch [2/100], Loss: 0.9334, Train Accuracy: 67.34 %, Test Accuracy: 67.07 %
Epoch [3/100], Loss: 0.7388, Train Accuracy: 74.37 %, Test Accuracy: 69.03 %
Epoch [4/100], Loss: 0.5696, Train Accuracy: 80.62 %, Test Accuracy: 68.66 %
Epoch [5/100], Loss: 0.4193, Train Accuracy: 86.28 %, Test Accuracy: 70.02 %
Epoch [6/100], Loss: 0.2895, Train Accuracy: 90.97 %, Test Accuracy: 69.61 %
Epoch [7/100], Loss: 0.1987, Train Accuracy: 94.08 %, Test Accuracy: 70.31 %
Epoch [8/100], Loss: 0.1422, Train Accuracy: 95.87 %, Test Accuracy: 70.14 %
...... # the whole result can be seen in part2.ipynb
Epoch [98/100], Loss: 0.0104, Train Accuracy: 99.60 %, Test Accuracy: 73.49 %
Epoch [99/100], Loss: 0.0120, Train Accuracy: 99.57 %, Test Accuracy: 73.85 %
Epoch [100/100], Loss: 0.0098, Train Accuracy: 99.63 %, Test Accuracy: 74.05 %
```





Using a larger learning rate 1e-3 and the Adam optimizer, the model converges slightly faster, but the value is almost the same as the default learning rate; using a smaller learning rate 1e-5, the testing accuracy has not reached a peak and has not converged because the update is too small to have the model get better in a short time. But I think maybe it can still have a close value of the test accuracy by using the learning rate of 1e-3 and 1e-4, but I do not think it can reach the value of 85% because the training accuracy seems to already reach the peak.

The largest testing accuracy of this model is approximately 85%. The accuracy of the random selection should be 10%. The testing accuracy is limited by the model itself since I have tested the different learning rates and different optimizers, and the results show that the model has reached its best.

Part 3

Import the necessary libraries.

```python
import sys
import math
import torch
import numpy as np
import torch.nn as nn
import matplotlib.pyplot as plt
import torch.utils.data as data
```

Construct the palindrome dataset as the `PalindromeDataset` class. (the codes are given)

```python
class PalindromeDataset(data.Dataset):

    def __init__(self, seq_length):
        self.seq_length = seq_length

    def __len__(self):
        return sys.maxsize

    def __getitem__(self, idx):
        full_palindrome = self.generate_palindrome()
        return full_palindrome[0:-1], int(full_palindrome[-1])

    def generate_palindrome(self):
        left = [np.random.randint(0, 10) for _ in range(math.ceil(self.seq_length / 2))]
        left = np.asarray(left, dtype=np.float32)
        right = np.flip(left, 0) if self.seq_length % 2 == 0 else np.flip(left[:-1], 0)
        return np.concatenate((left, right))
```

Implement the `VanillaRNN` class.

All parameters such as W_{hx}, W_hh, W_{ph}, b_h, and b_o are defined using `nn.Parameter` so that they can rely on PyTorch's automatic differentiation. The size of these parameters can be seen below.

To calculate $h^{(t)}$, we need a for-loop to traverse the value of h from 0 to t. The value of h can be computed using the equation below.

$$h^{(t)} = tanh(W_{hx}x^{(t)} + W_{hh}h^{(t-1)} + b_h)$$

The initial value of h is a zero-tensor.

After getting the value of h in time t, we can now calculate the value we need to obtain by the following equations.

$$o^{(t)} = W_{ph}h^{(t)} + b_o$$

$$\hat{y}^{(t)} = softmax(o^{(t)})$$

Finally, we compute the cross-entropy loss over the last time-step by $Loss = -\Sigma_{k=1}^{K} y_k log(\hat{y}_k^{(t)})$, where k runs over the classes (K = 10 digits in total).

The codes can bee seen below.

```
class VanillaRNN(nn.Module):

    def __init__(self, seq_length, input_dim, hidden_dim, output_dim, batch_size):
        super(VanillaRNN, self).__init__()
        self.seq_length = seq_length
        self.input_dim = input_dim
        self.hidden_dim = hidden_dim
        self.output_dim = output_dim
        self.batch_size = batch_size
        self.Whx = nn.Parameter(torch.randn(input_dim, hidden_dim))
        self.Whh = nn.Parameter(torch.randn(hidden_dim, hidden_dim))
        self.Wph = nn.Parameter(torch.randn(hidden_dim, output_dim))
        self.bh = nn.Parameter(torch.zeros(1, hidden_dim))
        self.bo = nn.Parameter(torch.zeros(1, output_dim))

    def forward(self, x):
        h = torch.zeros(self.batch_size, self.hidden_dim)
        for t in range(self.seq_length):
            h = torch.tanh(torch.mm(x[:, t].reshape(-1, 1), self.Whx) + torch.mm(h, self.Whh) +
self.bh)
        output = torch.mm(h, self.Wph) + self.bo
        return output
```

The hyperparameters are set as follows. The size of the batch is 128 and the number of total steps is 100000. The dimension of the hidden layer is 128.

```
steps = 100000
batch_size = 128
input_dim = 1
hidden_dim = 128
output_dim = 10
```

The `draw` is a useful method to draw the chart of the result.

```
def draw(steps, accs, losses):
    plt.plot([i for i in range(steps)], losses, '-', color='#4169E1', alpha=0.8, linewidth=1,
label="Loss Curve")
    plt.legend(loc="upper right")
    plt.xlabel("Epoch")
    plt.ylabel("Loss")
    plt.title("Loss Chart")
    plt.show()
    plt.plot([i for i in range(steps)], accs, '-', color='#4169E1', alpha=0.8, linewidth=1,
label="Accuracy Curve")
    plt.legend(loc="lower right")
    plt.xlabel("Epoch")
    plt.ylabel("Accuracy")
    plt.title("Accuracy Chart")
    plt.show()
```

The `train` method encapsulates the process of training of `VanillaRNN`. This method will receive an argument `length` defining the length of the sequence. Then the method will first construct the model and the dataset using the classes we defined above, then use `torch.optim.RMSprop` as the optimizer and `nn.CrossEntropyLoss` as the loss function. After that, the method will loop all steps to train the model. In this process, the method will first try to deal with exploding gradients using `torch.util.clip_grad_norm_`, then the method will generate the outputs using the model, compute the loss between the targets and the ouputs, finally compute the loss and use the optimizer to update the model's parameters. The rest codes are useful in printing information and drawing the chart of the result.

```
def train(length):
    losses, accs = [], []
    model = VanillaRNN(length - 1, input_dim, hidden_dim, output_dim, batch_size)
    dataset = PalindromeDataset(length)
    data_loader = data.DataLoader(dataset, batch_size)
    optimizer = torch.optim.RMSprop(model.parameters(), lr=0.001)
    criterion = nn.CrossEntropyLoss()
    for step, (batch_inputs, batch_targets) in enumerate(data_loader):
        torch.nn.utils.clip_grad_norm_(model.parameters(), max_norm=10.0)
        batch_outputs = model(batch_inputs)
        loss = criterion(batch_outputs, batch_targets)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()
        acc = np.mean(np.argmax(batch_outputs.detach().numpy(), axis=1) ==
batch_targets.detach().numpy())
        losses.append(loss.item())
        accs.append(acc)
        if step % 2000 == 0:
            print('Step: ', step, 'Loss: ', loss.item(), 'Accuracy: ', acc)
        if step == steps - 1:
            print('Step: ', step, 'Loss: ', loss.item(), 'Accuracy: ', acc)
            break
    draw(steps, accs, losses)
```

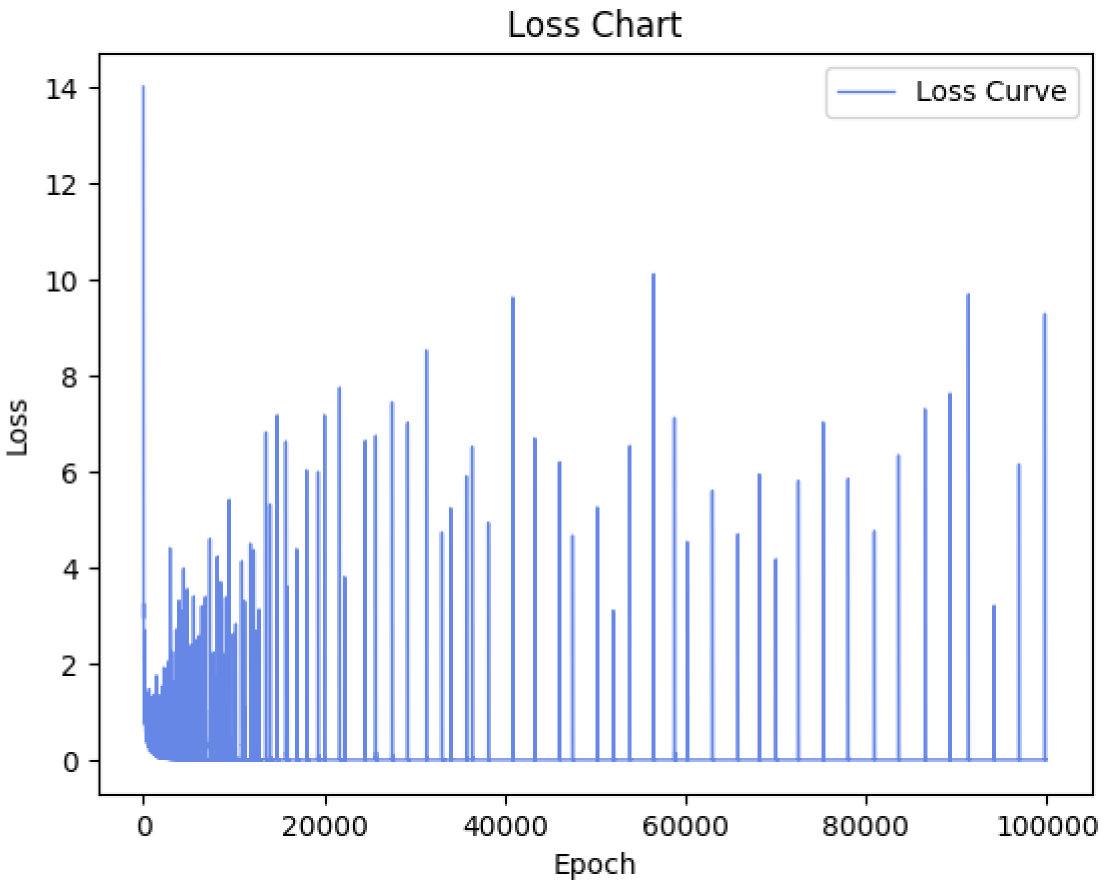
In the following parts, I will note the sequence length as T.

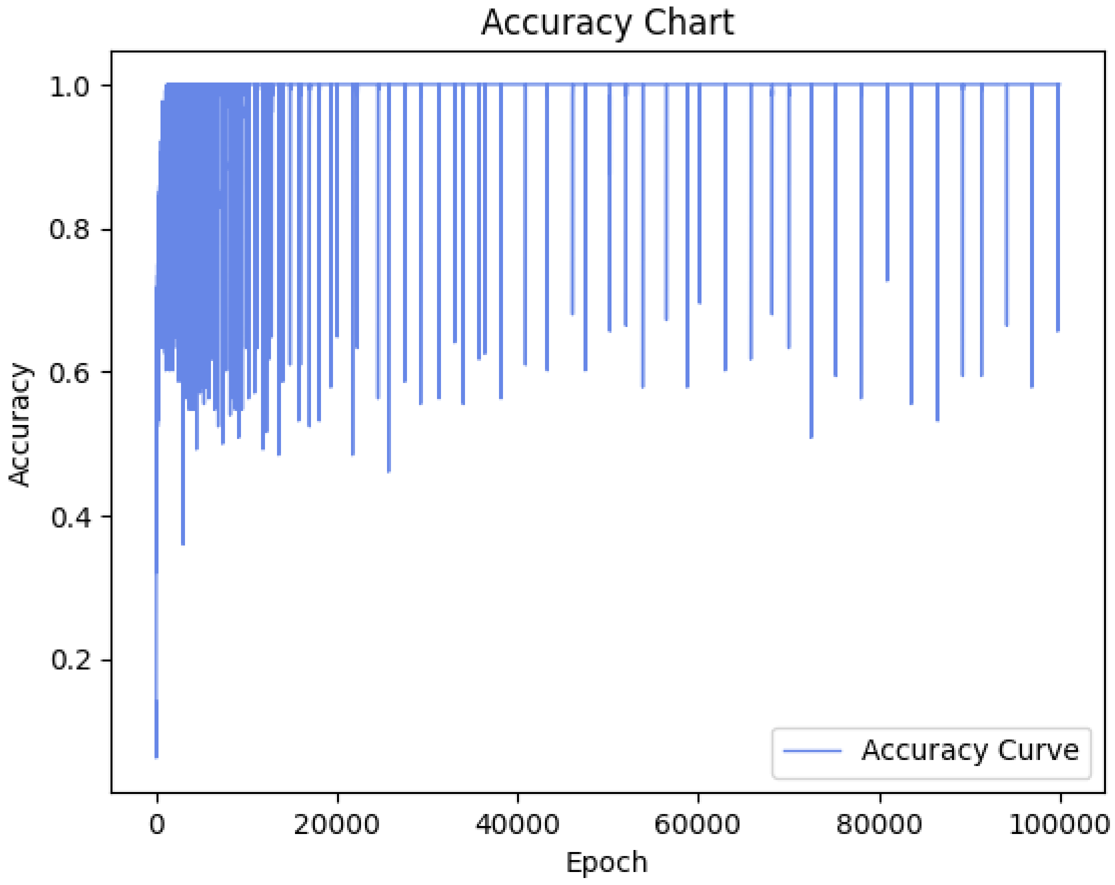
T = 5

```
length = 5
train(length)
```

```
Step:  0 Loss:  14.01941204071045 Accuracy:  0.140625
Step:  2000 Loss:  0.0743178129196167 Accuracy:  0.9921875
Step:  4000 Loss:  0.020959846675395966 Accuracy:  1.0
Step:  6000 Loss:  0.009927574545145035 Accuracy:  1.0
...... # the whole result can be seen in part3.ipynb
Step:  98000 Loss:  4.819460627913941e-06 Accuracy:  1.0
Step:  99999 Loss:  0.0002640431048348546 Accuracy:  1.0
```

The accuracy chart shows perfect accuracy with $T = 5$. The model converges quickly and has an accuracy of 100% most time. However, as we can see in the accuracy chart, RNN has a significant prediction fluctuation in short sequences, which may be because it fails to capture the underlying features of the data during training. Due to the short length of the sequence, the variation in input data is relatively large, making it difficult for the weight parameters in the RNN model to learn the essential features of the data accurately. However, since the sequence is too short, the overall performance can still predict correctly.


Loss Chart
Loss Curve
Loss
14
12
10
8
6
4
2
0
Epoch
0
20000
40000
60000
80000
100000

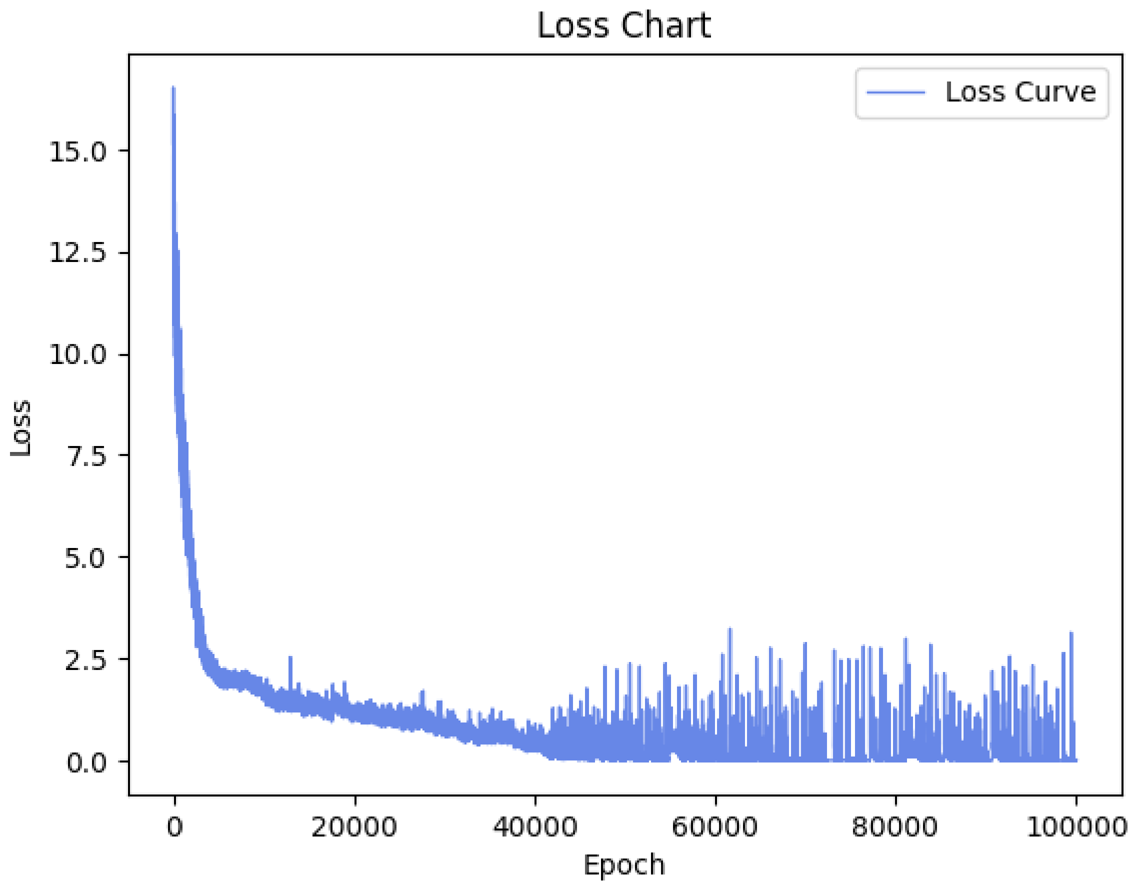

Accuracy Chart
Accuracy Curve
Accuracy
1.0
0.8
0.6
0.4
0.2
Epoch
0
20000
40000
60000
80000
100000

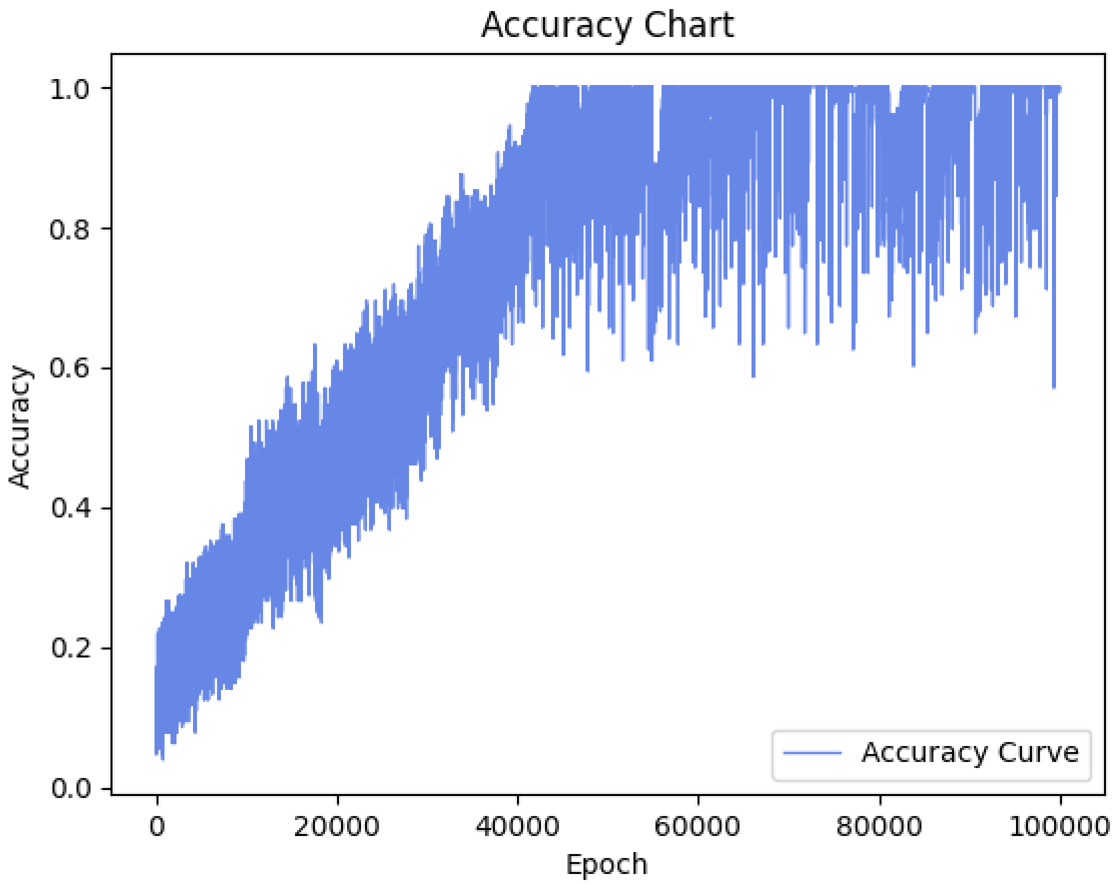
T = 10

```
length = 10
train(length)
```

```
Step:  0 Loss:  15.132143020629883 Accuracy:  0.0859375
Step:  2000 Loss:  4.471063137054443 Accuracy:  0.1953125
Step:  4000 Loss:  2.3331708908081055 Accuracy:  0.1640625
Step:  6000 Loss:  1.9909831285476685 Accuracy:  0.2109375
...... # the whole result can be seen in part3.ipynb
Step:  98000 Loss:  0.0031557297334074974 Accuracy:  1.0
Step:  99999 Loss:  0.0013669375330209732 Accuracy:  1.0
```

RNN's accuracy in predicting palindromic strings shows a stable increase, and compared to $T = 5$, the model exhibits a more stable performance. The accuracy of the predictions shows an overall upward trend and converges near 100%, with less fluctuation than $T = 5$. This indicates that when the sequence length is suitable, the model can learn the underlying features of the data more accurately, resulting in more stable and excellent prediction performance.



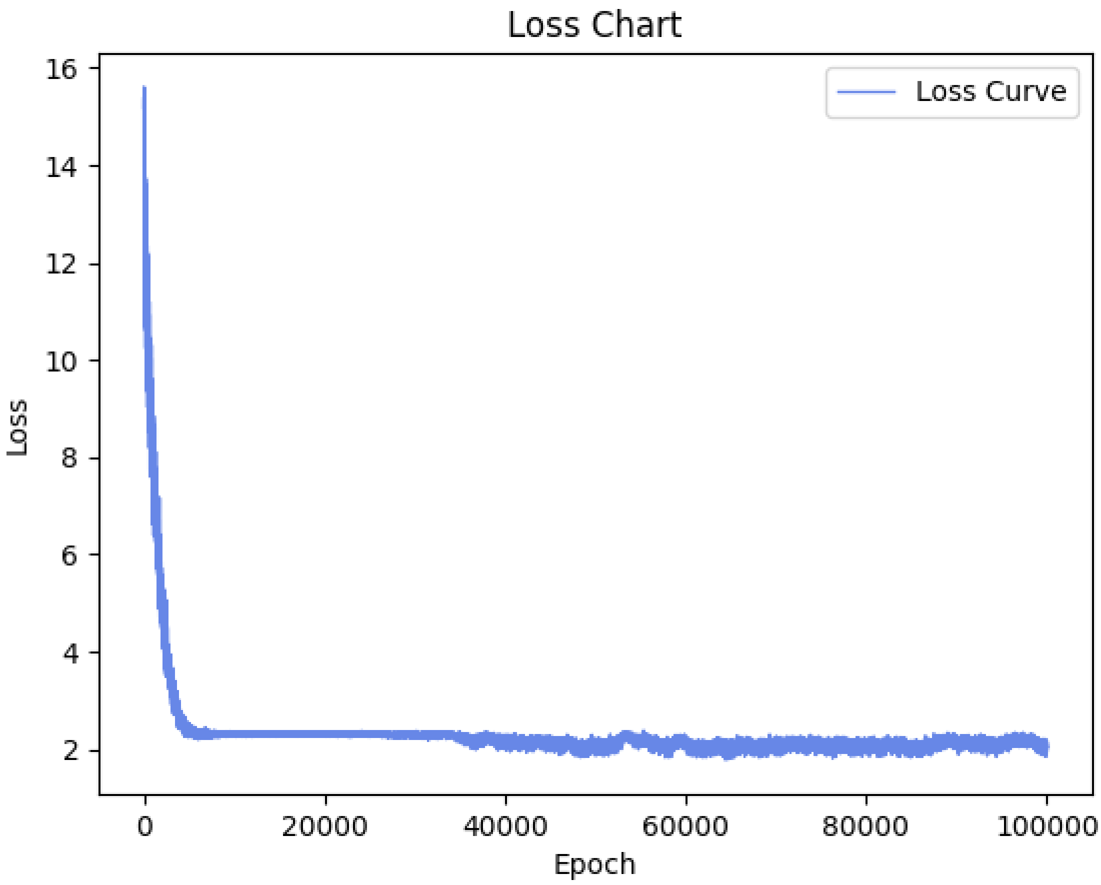


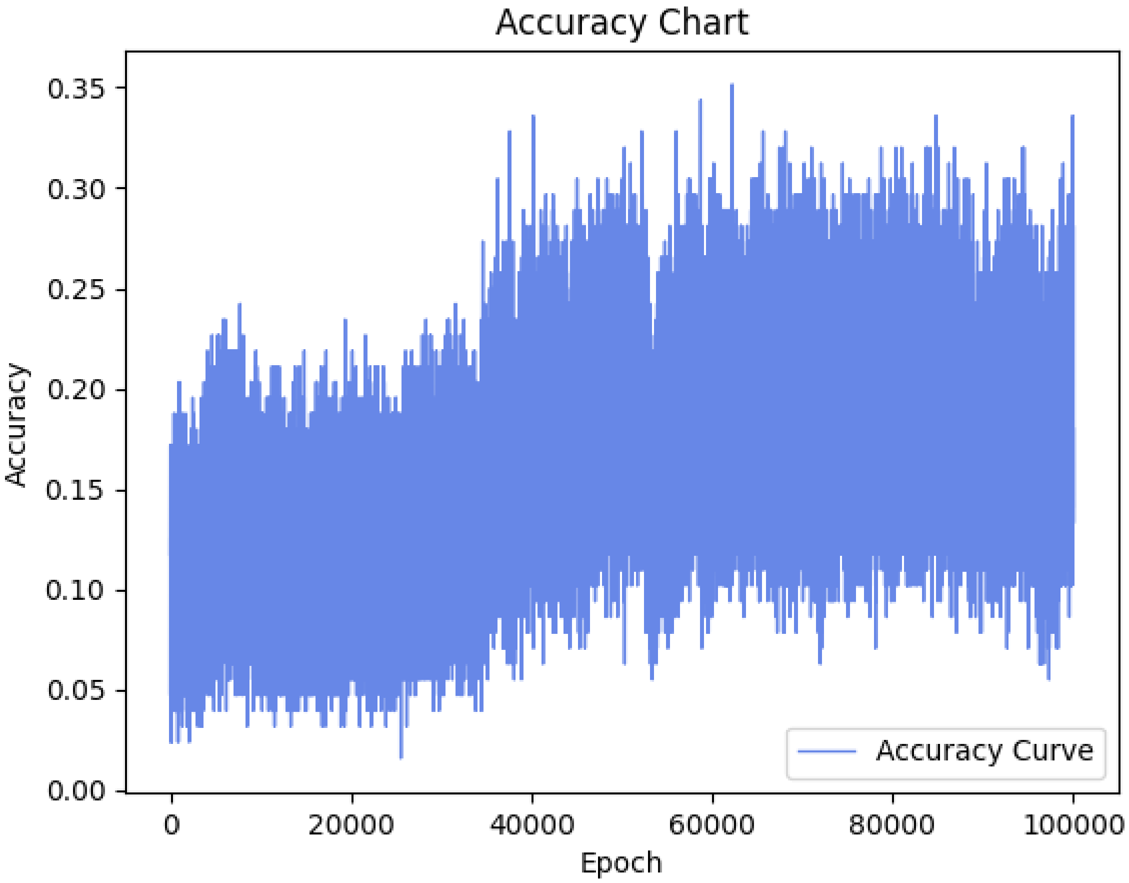
T = 15

```
length = 15
train(length)
```

```
Step:  0 Loss:  15.176674842834473 Accuracy:  0.1171875
Step:  2000 Loss:  5.657275676727295 Accuracy:  0.046875
Step:  4000 Loss:  2.7474279403686523 Accuracy:  0.1171875
Step:  6000 Loss:  2.390328884124756 Accuracy:  0.0859375
...... # the whole result can be seen in part3.ipynb
Step:  98000 Loss:  2.143059015274048 Accuracy:  0.140625
Step:  99999 Loss:  2.0372674465179443 Accuracy:  0.1796875
```

Although the loss function's value steadily decreases and converges during the training, RNN's accuracy in prediction can only reach around 18%. This is because its main drawback is the long-term dependency problem. When it needs to process distant information, the backpropagation algorithm calculates based on the chain rule, and the gradient is continuously multiplied by the same weight matrix, leading to gradient vanishing or explosion. In addition, while processing sequences, RNN accumulates previous information without focusing more attention on the most critical parts of the sequence. This may cause the model to forget relevant information when processing sequences, particularly apparent in handling long sequences. Therefore, the model's prediction performance is not satisfactory when $T = 15$, but as we can see from the figure, maybe the model can still learn a little more if the number of steps is larger enough.


Loss Chart
Loss Curve
Loss
16
14
12
10
8
6
4
2
0
20000
40000
60000
80000
100000
Epoch

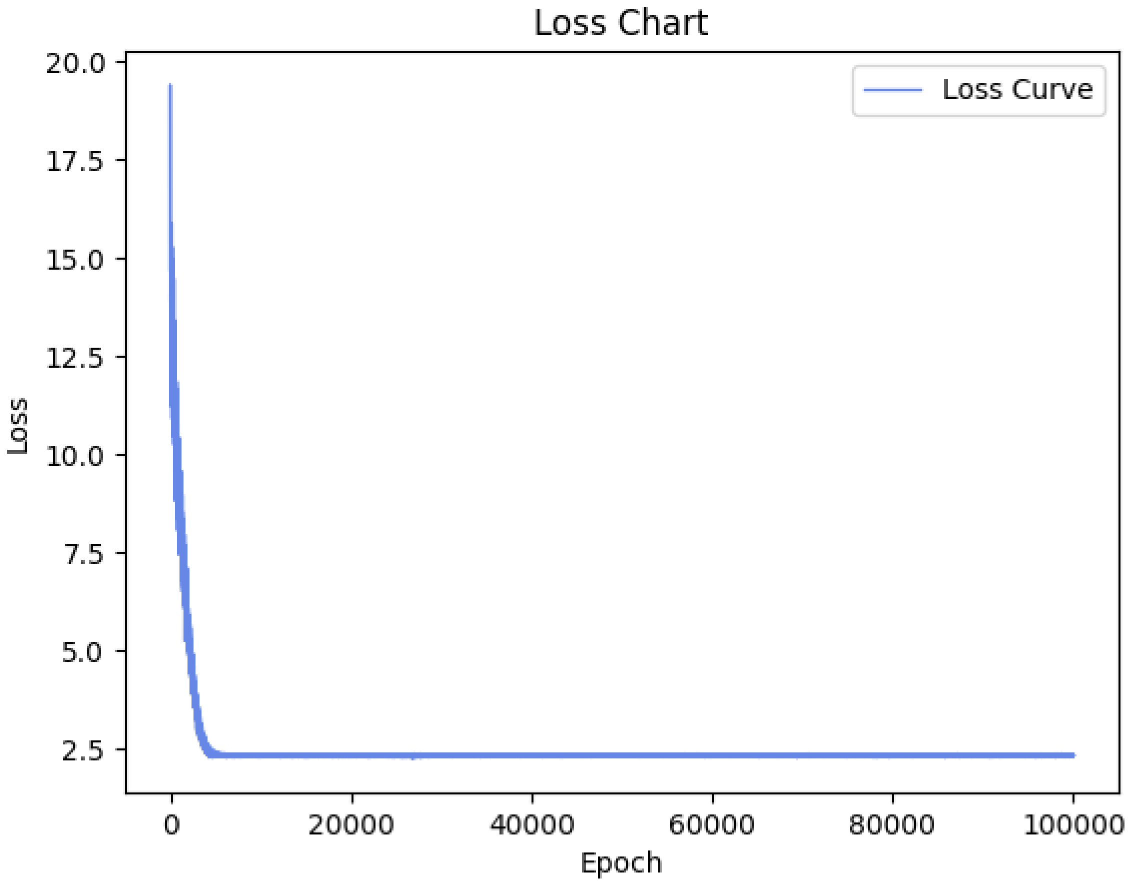

Accuracy Chart
Accuracy Curve
Accuracy
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
20000
40000
60000
80000
100000
Epoch

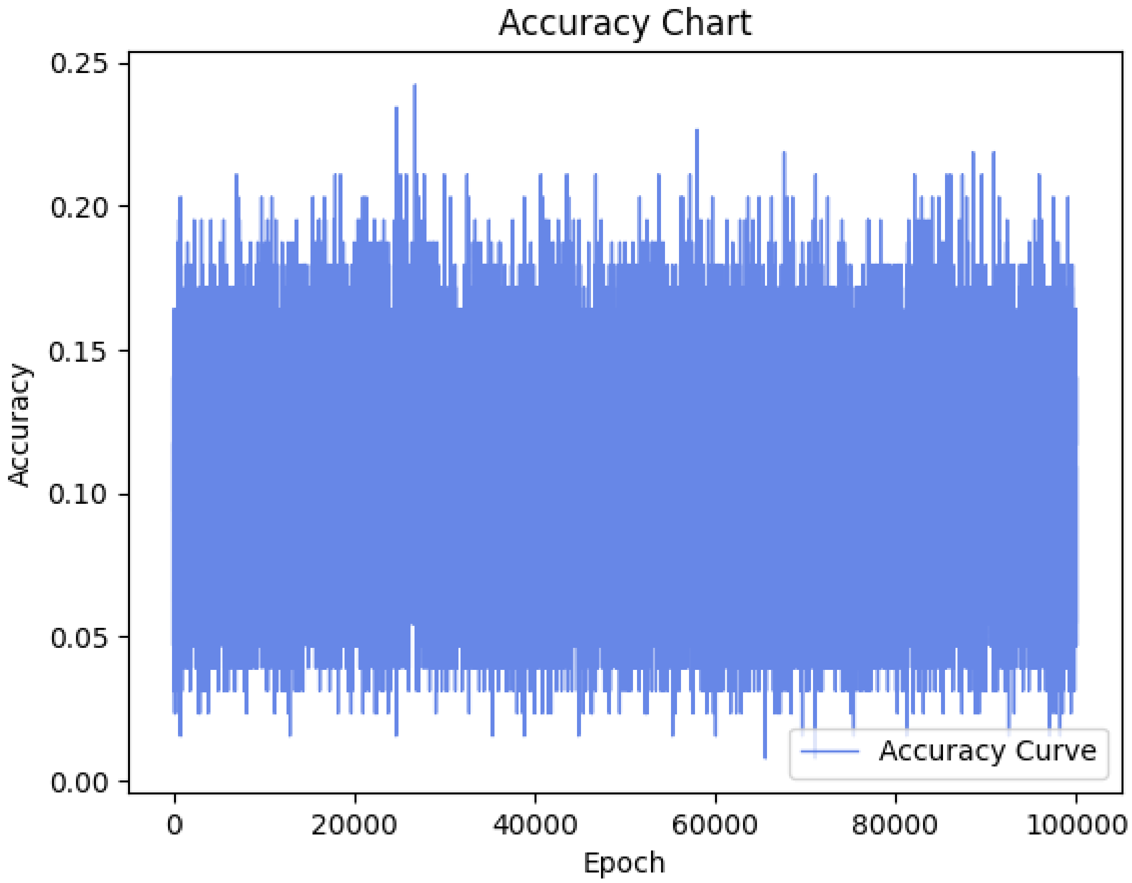
T = 20

```
length = 20
train(length)
```

```
Step:  0 Loss:  14.70294189453125 Accuracy:  0.1171875
Step:  2000 Loss:  5.732685565948486 Accuracy:  0.109375
Step:  4000 Loss:  2.510740280151367 Accuracy:  0.0859375
Step:  6000 Loss:  2.2793822288513184 Accuracy:  0.140625
...... # the whole result can be seen in part3.ipynb
Step:  98000 Loss:  2.3579070568084717 Accuracy:  0.078125
Step:  99999 Loss:  2.307896614074707 Accuracy:  0.1171875
```

When $T = 20$, the value of the loss function decreases during the training, but the model's accuracy in predictions remains only around 12%. The reason is the same as when T=15; the model's prediction performance is bad when dealing with long sequences.





Conclusion:

RNN performs poorly when dealing with short-time or long-time sequences. When the sequence is too short, the model finds it challenging to learn its underlying features, resulting in significant performance fluctuations despite good prediction accuracy. When the sequence is too long, the model encounters such problems as gradient vanishing and gradient explosion, making it unable to capture the features of a long sequence.

To correctly handle time-series data, better models such as LSTM are required instead of RNN.