Assignment 2

Abstract

This article explores the pytorch implementation of MLP, and CNN, then tests the superior performance of CNN in image recognition, and also implements PyTorch RNN for palindrome prediction

part1 : PyTorch MLP

implementation of pytorch mlp

data processing : since torch is used, we need to convert the data into tensor first.

The linear layer in the middle of the mlp is generated using nn.Linear().

The relu layer is implemented in foward using nn.functional.relu().

The loss function uses nn.CrossEntropyLoss().

```
TensorDataset and DataLoader fininsh the Data read and batch incoming
The torch.optim method is provided in pytorch to optimize our neural network.
torch.optim is a package that implements various optimization algorithms. In this
code, the Stochastic Gradient Descent (SGD) is chosen.
--optimizer = optim.SGD(model.parameters(), lr=learning_rate)

Training part:
        model.train()
        for x, y in train_loader:
            optimizer.zero_grad()# 梯度清零(Gradient clearing)
            y_pred = model(x)# 利用模型求解预测值(The predicted values are solved
using the model)
            loss = CEL(y_pred, y)# 求解loss(slove loss)
            loss.backward()# 反向传播求解梯度(Back propagation solves the gradient)
            optimizer.step()# 更新权重参数(Update weight parameter)
Evaluate part:
        model.eval()
        with torch.no_grad():
            y_pred = model(X_train)
            loss = CEL(y_pred, y_train)
```

struct of MLP

```
class MLP(nn.Module):

    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 units
in each linear layer
```

```
            n_classes: number of classes of the classification problem (i.e.,
output dimension of the network)
        """
        super(MLP, self).__init__()
        self.n_inputs = n_inputs
        self.n_hidden = n_hidden
        self.n_classes = n_classes
        self.input_layer = nn.Linear(n_inputs, n_hidden[0])
        self.hidden_layers = nn.ModuleList([nn.Linear(n_hidden[i], n_hidden[i +
1]) for i in range(len(n_hidden) - 1)])
        self.output_layer = nn.Linear(n_hidden[-1], n_classes)
    def forward(self, x):
        """
        Predict network output from input by passing it through several layers.
        Args:
            x: input to the network
        Returns:
            out: output of the network
        """

        x = self.input_layer(x)
        x = nn.functional.relu(x)
        for layer in self.hidden_layers:
            x = layer(x)
            x = nn.functional.relu(x)

        out = self.output_layer(x)
        return out
```

Hyperparameter setting

```
DNN_HIDDEN_UNITS_DEFAULT = '20'
LEARNING_RATE_DEFAULT = 1e-2
MAX_EPOCHS_DEFAULT = 1500
EVAL_FREQ_DEFAULT = 10
```

All using stochastic gradient descent

Training and training results

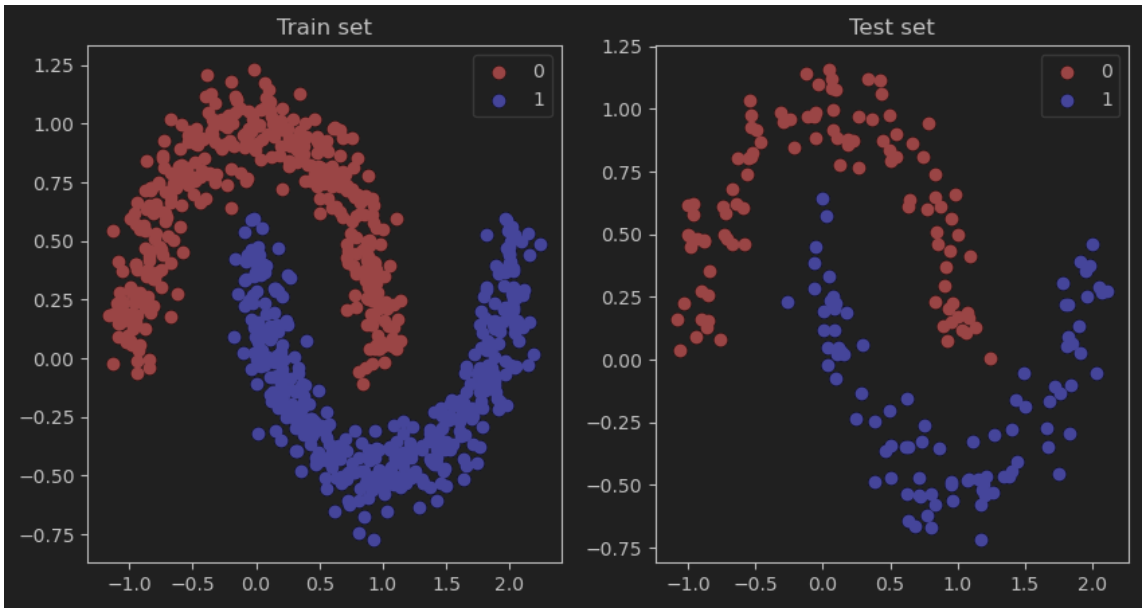
The following calculation uses the cpu, without cuda.

Both implementations utilize stochastic gradient descent.

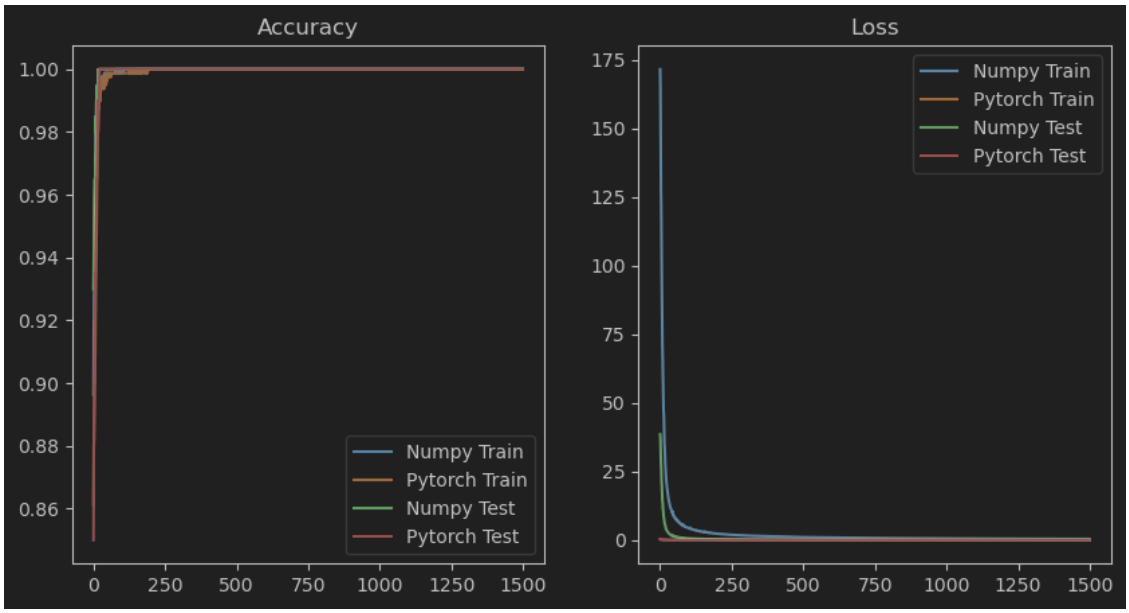
Data set one——make_moons

Data set parameter Settings

```
X, y = make_moons(n_samples=1000, noise=0.1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```



Training result



Step: 1490, Loss: 0.012795103218324005, Accuracy: 1.0(numpy)

Step: 1490 test_Loss: 8.19366323412396e-05 test_Accuracy: 1.0(pytorch)

numpy time: 103.90349674224854

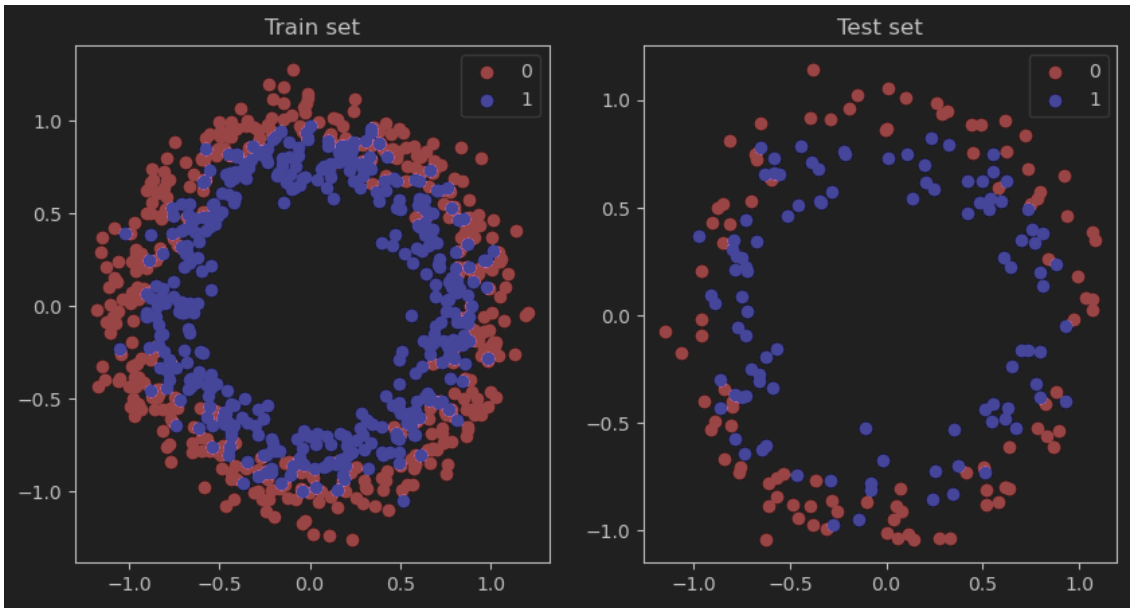
pytorch time: 438.38285779953003

It can be observed that both the PyTorch and NumPy implementations of the MLP exhibit excellent performance on the current dataset and are able to converge rapidly.

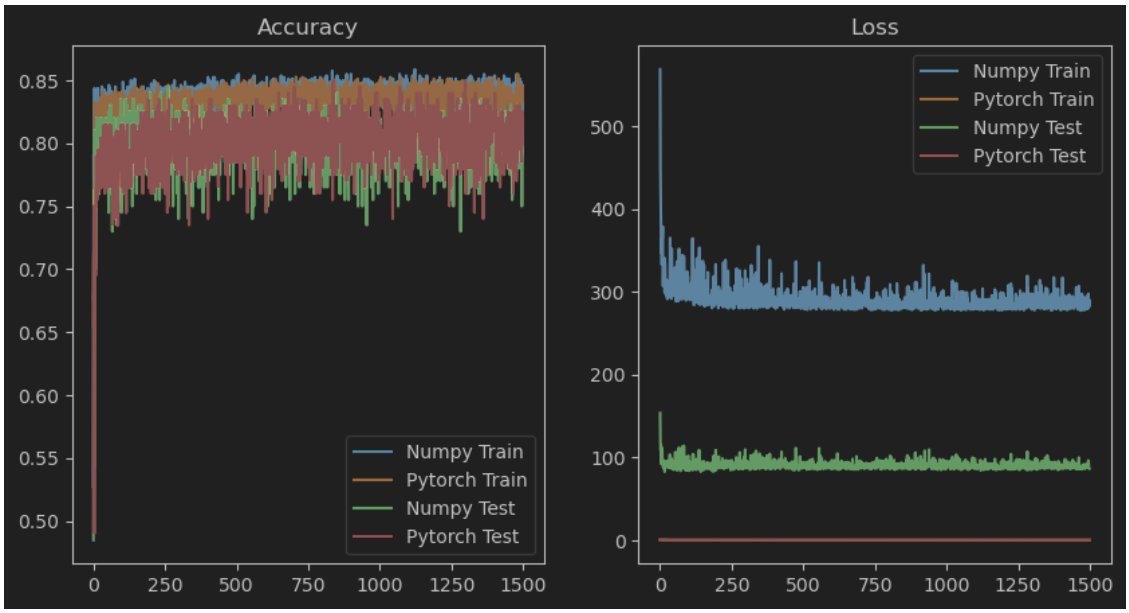
Data set two——make_circles

Data set parameter Settings

```
X, y = make_circles(n_samples=1000, noise=0.1)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```



Training result



Step: 1490, Loss: 91.03325931666727, Accuracy: 0.795(numpy)

Step: 1490 test_Loss: 0.4534014165401459 test_Accuracy: 0.8199999928474426(pytroch)

numpy time: 95.99722361564636

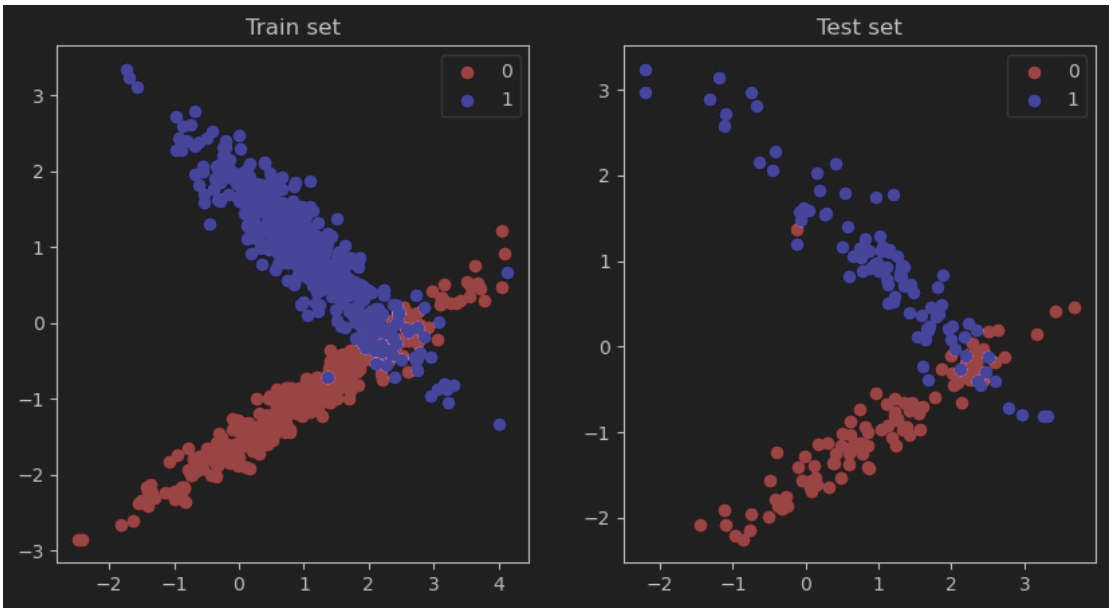
pytorch time: 428.80032873153687

It can be observed that on the current dataset, the performance of the PyTorch-implemented MLP is slightly superior to that of the manually implemented NumPy MLP, with a notably lower loss

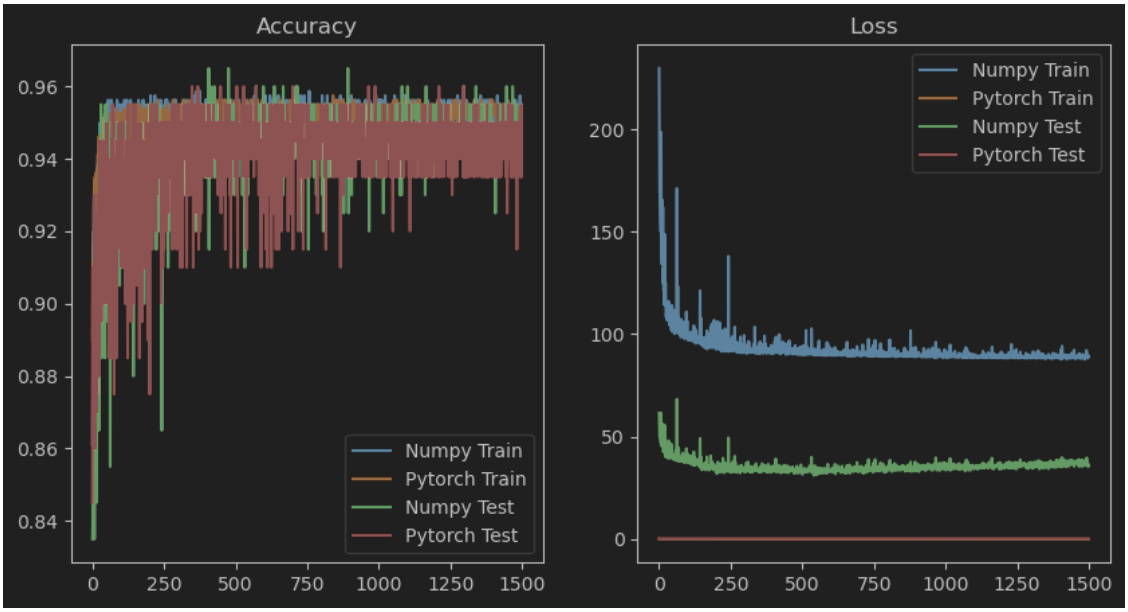
Data set three——make_classification

Data set parameter Settings

```
X, y = make_classification(n_samples=1000, n_features=2, n_informative=2,
n_redundant=0, n_clusters_per_class=1, random_state=42)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
random_state=42)
```



Training result



Step: 1490, Loss: 42.22639076425106, Accuracy: 0.94

Step: 1490 test_Loss: 0.17298758029937744 test_Accuracy: 0.949999988079071

numpy time: 95.36873149871826

pytorch time: 458.8827908039093

The model works well for this data set as well

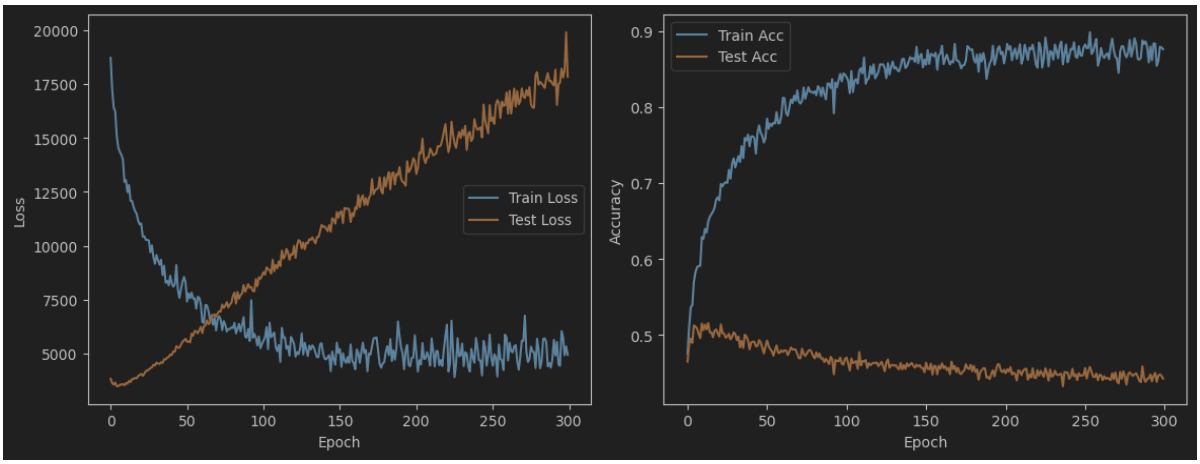
CIFAR10 dataset

the best Hyperparameter setting of testing

using gpu to speed up

```
model = mlp.MLP(n_inputs=3*32*32, n_hidden=[100, 50], n_classes=10).to(device)
LEARNING_RATE_DEFAULT = 1e-2
MAX_EPOCHS_DEFAULT = 300
EVAL_FREQ_DEFAULT = 2
```

Training result



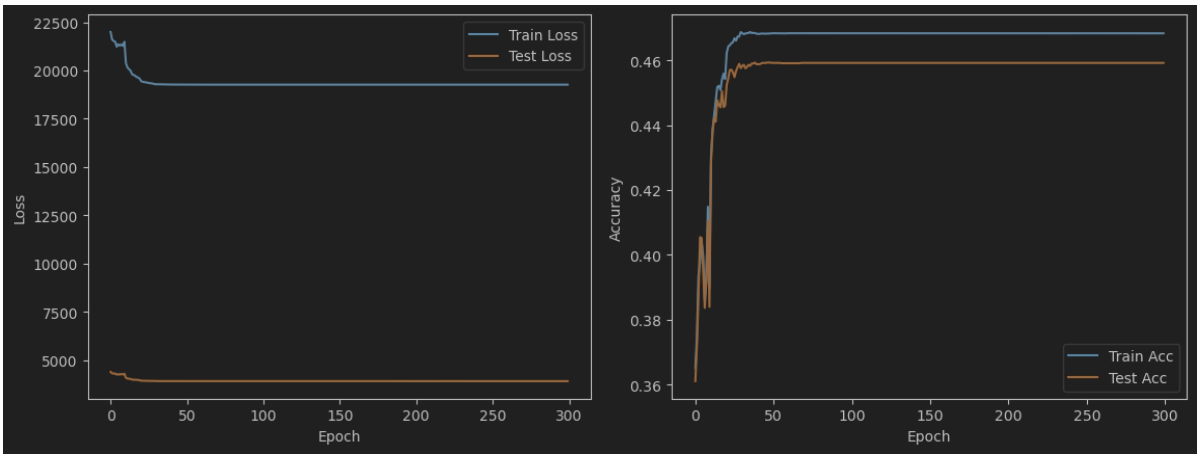
It is evident that the classification performance of MLP on the CIFAR10 dataset is mediocre. While the accuracy on the training set can reach around 88%, it only achieves about 50% on the test set. Additionally, as the number of training epochs increases, overfitting becomes apparent, indicating that the generalization ability of the MLP model for image classification tasks is relatively poor.

Other MLP architecture designs and their results

```
class MLP(nn.Module):
    def __init__(self):
        super(MLP, self).__init__()
        self.fc1 = nn.Linear(3*32*32, 100)
        self.fc2 = nn.Linear(100, 50)
        self.fc3 = nn.Linear(50, 10)
        self.relu = nn.ReLU()
        self.dropout = nn.Dropout(0.5)  # Add dropout layer with 50% probability

    def forward(self, x):
        x = x.view(-1, 3*32*32)
        x = self.dropout(self.relu(self.fc1(x)))
        x = self.dropout(self.relu(self.fc2(x)))
        x = self.fc3(x)
        return x

# Learning rate scheduler
scheduler = StepLR(optimizer, step_size=10, gamma=0.1)
```



The use of StepLR and dropout layers resulted in poor performance. Additionally, due to StepLR, the learning rate became quite low in the later stages, rendering the training ineffective towards the end.

conclusion

The dataset is crucial for the accuracy of the trained model, as noise, data types, and data distribution significantly impact the final results. Clearly, MLPs are insufficient for image classification tasks and are also time-consuming.

part2： PyTorch CNN

theory

Convolutional Neural Networks (CNNs) are a type of deep learning model specifically designed to process data with a grid-like topology, such as image data. CNNs excel in computer vision tasks such as image classification, object detection, and segmentation.The key lies in utilizing convolutional layers to extract features, reducing computational load through pooling layers, and ultimately performing classification via fully connected layers.

implementation of pytorch CNN

The architecture is defined as follow

```
class CNN(nn.Module):
    def __init__(self, n_channels, n_classes):
        """
    Initializes CNN object.
    Args:
      n_channels: number of input channels
      n_classes: number of classes of the classification problem
    """
        super(CNN, self).__init__()
        self.conv1 = nn.Sequential(
            nn.Conv2d(n_channels, 64, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(64)
        )
        self.conv2 = nn.Sequential(
            nn.Conv2d(64, 128, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(128)
```

```
        )
        self.conv3 = nn.Sequential(
            nn.Conv2d(128, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256)
        )
        self.conv4 = nn.Sequential(
            nn.Conv2d(256, 256, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(256)
        )
        self.conv5 = nn.Sequential(
            nn.Conv2d(256, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512)
        )
        self.conv6 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512)
        )
        self.conv7 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512)
        )
        self.conv8 = nn.Sequential(
            nn.Conv2d(512, 512, kernel_size=3, stride=1, padding=1),
            nn.BatchNorm2d(512)
        )
        self.fc1 = nn.Linear(512, n_classes)

    def forward(self, x):
        """
    Performs forward pass of the input.

    Args:
      x: input to the network
    Returns:
      out: outputs of the network
    """
        # Define the forward pass
        x = F.relu(self.conv1(x))
        x = F.max_pool2d(x, 3, 2, 1)
        x = F.relu(self.conv2(x))
        x = F.max_pool2d(x, 3, 2, 1)
        x = F.relu(self.conv3(x))
        x = F.relu(self.conv4(x))
        x = F.max_pool2d(x, 3, 2, 1)
        x = F.relu(self.conv5(x))
        x = F.relu(self.conv6(x))
        x = F.max_pool2d(x, 3, 2, 1)
        x = F.relu(self.conv7(x))
        x = F.relu(self.conv8(x))
        x = F.max_pool2d(x, 3, 2, 1)
        x = x.view(x.size(0), -1)
        x = self.fc1(x)

        return x
```

`nn.Sequential` is a subclass of `nn.Module`, which means it inherits all the methods of `nn.Module`. Moreover, when directly using `nn.Sequential`, there is no need to write a `forward` function because it is already implemented internally. This approach has yielded satisfactory results.

Hyperparameter setting

```
# Default constants
LEARNING_RATE_DEFAULT = 1e-4
BATCH_SIZE_DEFAULT = 32
MAX_EPOCHS_DEFAULT = 5000
EVAL_FREQ_DEFAULT = 500
OPTIMIZER_DEFAULT = 'ADAM'
DATA_DIR_DEFAULT = './data'
FLAGS = None
```

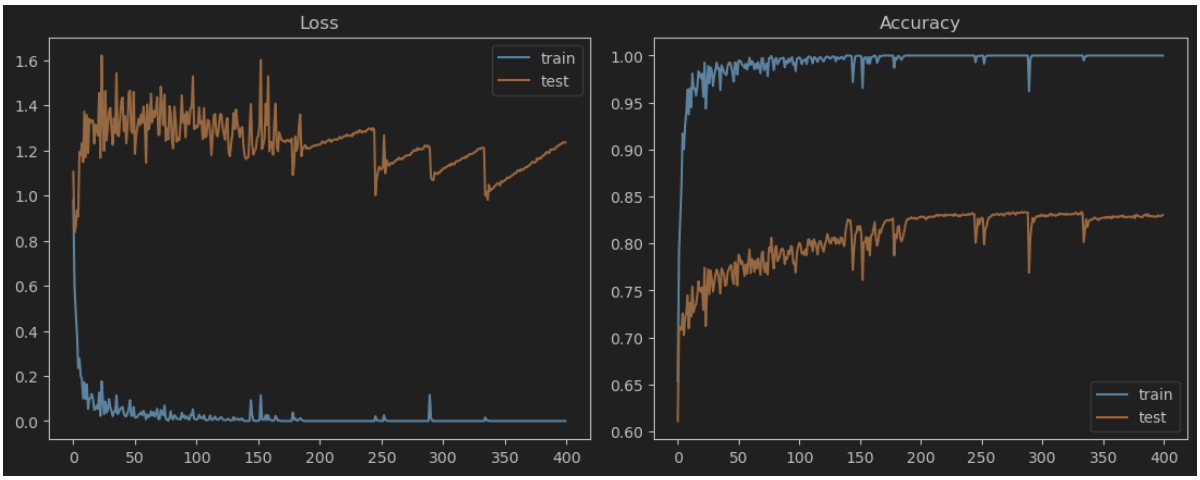
This is the default parameter setting, but according to the actual situation, the accuracy becomes about 0.84 when the hundredth epoch, and there is no significant change in the subsequent years, nor is the Test loss, so the epoch is changed to 400,

Truth setting

```
# Default constants
LEARNING_RATE_DEFAULT = 1e-4
BATCH_SIZE_DEFAULT = 32
MAX_EPOCHS_DEFAULT = 400
EVAL_FREQ_DEFAULT = 2
OPTIMIZER_DEFAULT = 'ADAM'
DATA_DIR_DEFAULT = './data'
FLAGS = None
```

using mini-batch gradient descent

Training result



The running results show that CNNs indeed demonstrate strong performance. For the training set, the accuracy approaches 100% after 50 to 60 epochs, and the accuracy on the test set reaches around 80%. Additionally, based on the loss from the test set, there is no significant overfitting. Overall, the training results are quite satisfactory.

conclusion & unknown

Part3: PyTorch RNN

theory

Recurrent Neural Networks (RNNs) are a type of deep learning model well-suited for handling sequential data, such as time series and text. Unlike traditional neural networks, RNNs capture temporal dependencies in sequences through recurrent connections in their hidden states.

implementation of RNN

```
class VanillaRNN(nn.Module):

    def __init__(self, input_length, input_dim, hidden_dim, output_dim,device):
        super(VanillaRNN, self).__init__()
        # Initialization here ...
        self.hidden_dim = hidden_dim
        self.intput_dim = input_dim
        self.input_length = input_length
        self.output_dim = output_dim
        self.Wx = nn.Linear(input_dim, hidden_dim)
        self.Wh = nn.Linear(hidden_dim, hidden_dim)
        self.Wy = nn.Linear(hidden_dim, output_dim)
    def forward(self, x):
        # Implementation here ...
        h = torch.zeros(x.size(0), self.hidden_dim).to(x.device)
        for i in range(self.input_length):
            h = torch.tanh(self.Wx(x[:, i, :]) + self.Wh(h))
        y = self.Wy(h)
        return nn.functional.softmax(y, dim=1)
```

Modify the code for generating the dataset to ensure consistency in the dataset.

```
# if self.total_len > max_num:
#     print("Warning: total_len is larger than the maximum possible length. ")
#     print("Setting total_len to the maximum possible length. ")
#     print(
#         "Warning: access length of dataset by len(dataset) to get the actual
length. ")
#     self.total_len = 10 ** self.half_length
print("Total length of dataset: ", self.total_len)
self.data = np.random.default_rng().choice(
    max_num, self.total_len, replace=True)
```

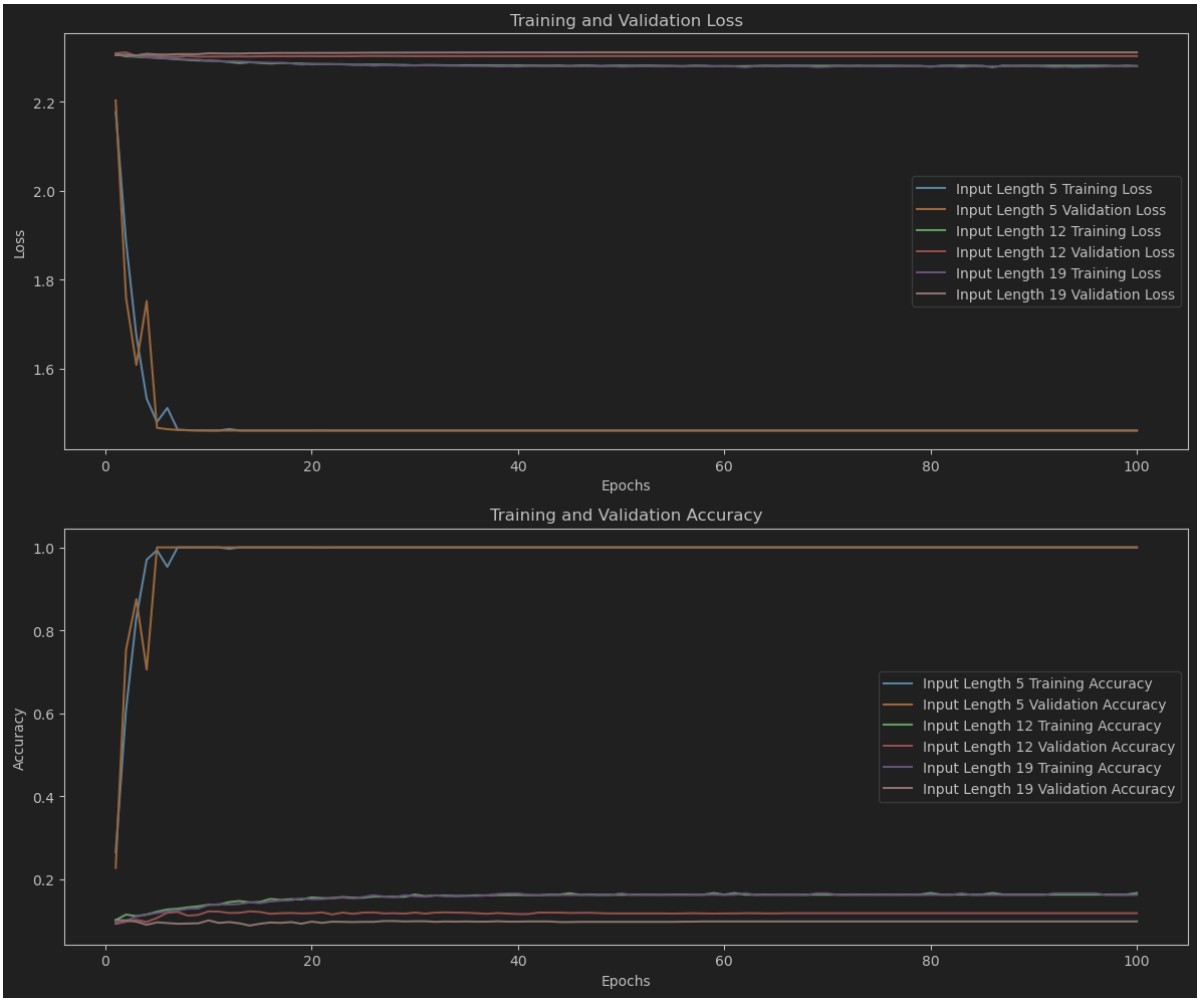
After testing with default parameters, the updated parameters are as follows:

```
configs = [
    {'input_length': 5, 'input_dim': 1, 'num_classes': 10, 'num_hidden': 128,
     'batch_size': 32, 'learning_rate': 0.001, 'max_epoch': 100,
     'max_norm': 10.0, 'data_size': 10000, 'portion_train': 0.8},

    {'input_length': 12, 'input_dim': 1, 'num_classes': 10, 'num_hidden': 128,
     'batch_size': 32, 'learning_rate': 0.001, 'max_epoch': 100,
     'max_norm': 10.0, 'data_size': 10000, 'portion_train': 0.8},

    {'input_length': 19, 'input_dim': 1, 'num_classes': 10, 'num_hidden': 128,
     'batch_size': 32, 'learning_rate': 0.001, 'max_epoch': 100,
     'max_norm': 10.0, 'data_size': 10000, 'portion_train': 0.8}
]
```

Training result



conclusion

The performance with T=5 is significantly better than that with T=12 and T=19, and the improvement is considerable.

it is obviously that the prediction accuracy of the RNN become lower as we challenge it with longer and longer palindromes