# SUSTech CS324 2024 Spring Assignment 2 Report

12110304 Chunhui XU

May 12, 2024

## 1 Part 1 Task 1 & 2

### 1.1 Simple Introduction

In these tasks, I need to create a simple MLP by `PyTorch`. And compare the training situation with the realization by `NumPy`.

The MLP structure just as which in Assignment 1, and I'll compare `torch` and `numpy` implementation in `make_moon` and two other datasets.

### 1.2 `torch` Feature Analysis

In torch, we could use the pre-defined utils.

For exmaple: common layers, such as linear layer and ReLU layer, can simply use `nn.Linear` and `nn.ReLU`, they have there own `forward()` functions. Once they are initialization with `nn.Sequential()`, we can directly get output from input by passing it through each layer one by one.

Use `nn.CrossEntropyLoss()` to compute the Cross Entropy loss of the model. Through `torch.optim`, we can adjust parameters in layers, implement backward propagation.

### 1.3 Global Settings

#### 1.3.1 Default Parameters

- `dnn_hidden_units`: [20], comma separated list of number of units in each hidden layer
- `learning_rate`: $1e^{-2}$, learning rate for optimization
- `max_steps`: 1500, number of steps to run trainer
- `eval_freq`: 10, frequency of evaluation on the test set
- `batch_size`: 800, batch size of single train batch

#### 1.3.2 MLP Structure

1. 2 input
2. $2 \rightarrow 20$ Linear Layer
3. $20 \rightarrow 20$ ReLU Layer
4. $20 \rightarrow 2$ Linear Layer
5. $2 \rightarrow 2$ Softmax Layer
6. One hot output and Cross Entropy loss function
7. Gradient descent strategy: BGD idea actually, refer to 1.5.3.

#### 1.3.3 Dataset Generation

I use code to visually demonstrate the generation method of the dataset.

```
from sklearn.preprocessing import OneHotEncoder
from sklearn.datasets import *

x, y = [], [] # input and label
if type_id == 1:
    self.type_name = 'moon'
    x, y = make_moons(n_samples=1000, noise=0.05)
elif type_id == 2:
    self.type_name = 'blobs'
    x, y = make_blobs(n_samples=1000, centers=2, cluster_std=3)
elif type_id == 3:
    self.type_name = 'gaussian_quantiles'
    x, y = make_gaussian_quantiles(n_samples=1000, n_classes=2)
y = OneHotEncoder(sparse_output=False).fit_transform(y.reshape(-1, 1))
```

### 1.4 Result Visualization

There are 3 figures, fig 1 for `make_moon`, fig 6 for `make_blobs`, fig 7 for `make_gaussian_quantiles`,

### 1.5 Result Analysis

#### 1.5.1 Overall Analysis

Through different images for several different datasets, it can be observed that despite the different implementation approaches of the two models, as long as the model architecture remains constant, their characteristic curves exhibit similarities, both in terms of accuracy and loss. This mutual approve both of my implementations within this simple MLP structure are reasonable.

#### 1.5.2 Dataset Analysis

Simple analysis: The model has different performance in three data sets: the data set of blobs is widely separated, and it is easy to achieve high performance; moon is second. For gaussian quantiles, intuitively speaking, the data distribution of it cannot be divided by a simple decision boundary, and accordingly the accuracy increases relatively slowly with training.

#### 1.5.3 Loss Analysis

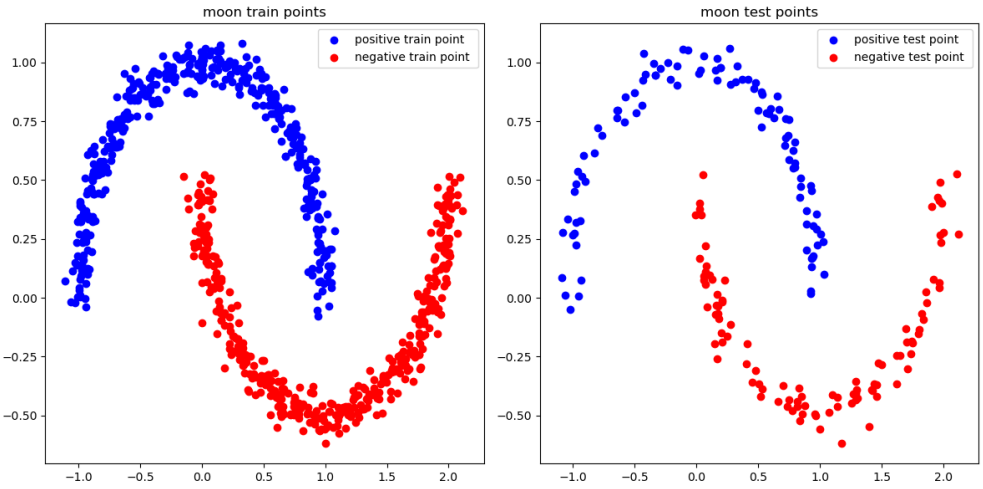
Although I utilized `optim.SGD()` as the optimizer, I actually realize the BGD. I did not perform any data splitting during input; instead, I fed all inputs as a whole to the MLP for computation. Therefore, in practice, what I achieved was finding the global optimum through gradient descent, which is the concept of Batch Gradient Descent (BGD). From this, it can be inferred that the descent of the training loss curve is smooth, devoid of any fluctuations.

## 2 Part 1 Task 3

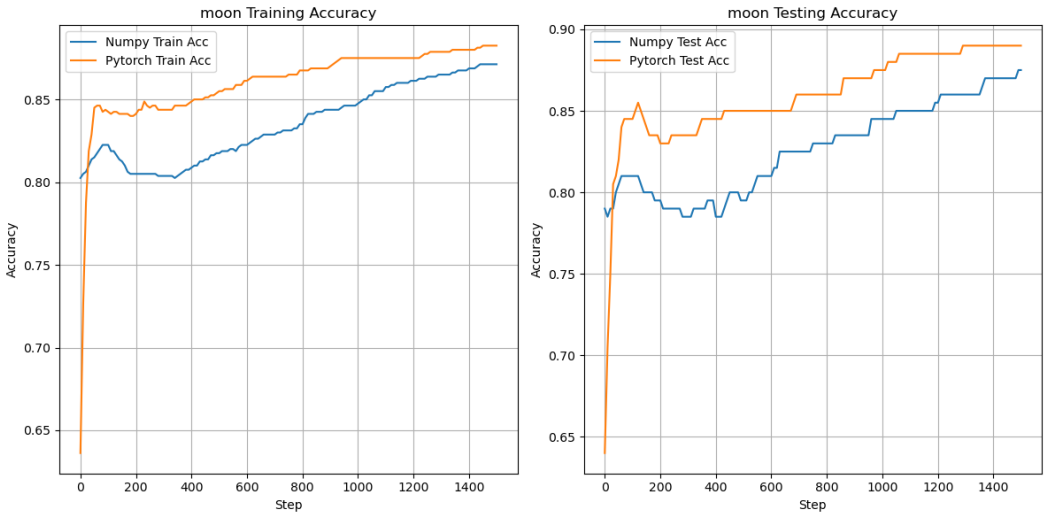
### 2.1 Global Settings

#### 2.1.1 Default Parameters

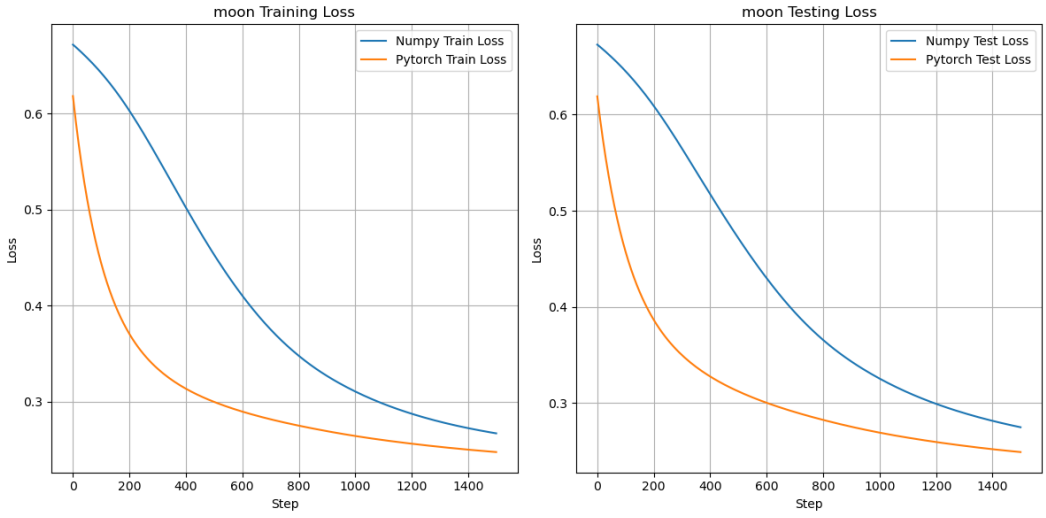
- `dnn_hidden_units`: $[512, 64]$, comma separated list of number of units in each hidden layer
- `learning_rate`: $1e^{-3}$, learning rate for optimization
- `max_epochs`: 150, number of steps to run trainer
- `eval_freq`: 1, frequency of evaluation on the test set
- `batch_size`: 32, batch size of single train batch




(a) Blobs Data




(b) Blobs Accuracy




(c) Blobs Loss

Figure 1: Blobs numpy & torch

### 2.1.2 MLP Structure

1. $3 \times 32 \times 32$ input
2. $3072 \rightarrow 512$ Linear Layer
3. $512 \rightarrow 512$ ReLU Layer
4. $512 \rightarrow 64$ Linear Layer
5. $64 \rightarrow 64$ ReLU Layer
6. $64 \rightarrow 10$ Linear Layer
7. $10 \rightarrow 10$ Softmax Layer
8. 10 output

## 2.2 Result Visualization

Fig 2 show the accuracy and loss in CIFAR-10 dataset with Adam optimizer, while fig 8 is SGD.

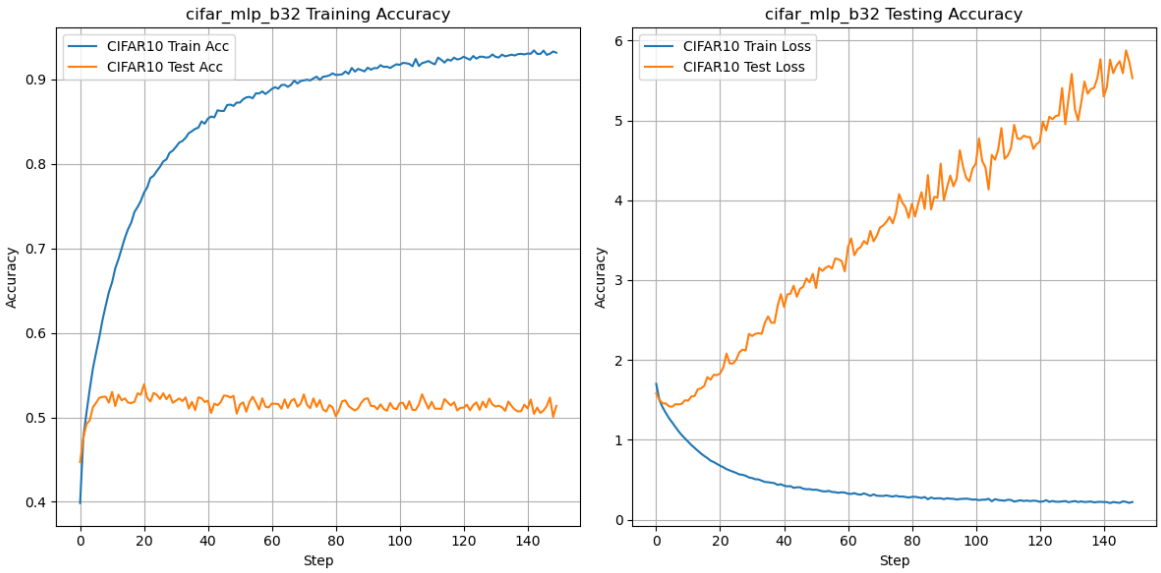



Figure 2: MLP CIFAR-10 Curve

## 2.3 Result Analysis

### 2.3.1 MLP Analysis

In the MLP, I choose $3 \times 32 \times 32$ as input. Because for CIFAR-10 picture, it has $32 \times 32$ as the pixel width and height. In the picture, each pixel has red, green, blue 3 channels. Therefore, I got the 3072 feature inputs.

In hidden layer, I set two hidden layers, there size is 512, 64. As a simple dataset, I don't want to add too complex hidden layers for MLP, so I choose these hidden layer size.

Finally, I got output of size 10, indicating the possibility of each class in CIFAR-10, and I choose the max one as the final classification result.

#### 2.3.2 Feature Analysis

Overall, we can find a very interesting situation: as the number of training times continues to increase, the accuracy of the training set quickly reached 90%, while the accuracy of the test set stagnated at 50% after a brief rise. Finally, it has been unable to rise. And by carefully observing the Loss curve, we will also find an additional fact: the Loss of the test set is even rising!

This leads to an obvious concept: overfitting. With continuous training, MLP overfits the training data, and its parameters are constantly customized according to the training data, but it loses generalization and cannot well extract some general features in the CIFAR-10 data set.

So why is there such a big deviation in the performance of MLP on CIFAR-10 compared to the classification problem of Assignment1? Here are a few possible reasons I guess:

- **Fully connected features**: MLP is a fully connected structure, which means that everything in the image data will affect the final result. The image data may contain edges, interference, and other data that are not helpful or even disruptive to the analysis results, thus causing a lot of problems in the MLP learning process and overfitting.
- **The structure of hidden layers is too simple**: I only set up two hidden layers [512, 64]. Such a simple structure cannot extract the deep features of the image well, which leads to a decrease in the generalization ability of the model and leads to overfitting.

In short, although MLP classification was tried on CIFAR-10, due to some objective limitations, its actual effect was not very satisfactory. In the Part 2, I will try to use CNN to complete the CIFAR-10 image classification problem.

# 3 Part 2 Task 1

## 3.1 Simple Introduction

In this task, I need to create a simple CNN. The structure is listed in lecture slide. I analyze the layers feature and compute the input and output shape in this part.

## 3.2 CNN Structure

The structure Defined by lecture slide, fig 3:

For pooling layer, $k$ stand for kernel size, $s$ stand for stride, $p$ stand for padding.

## 3.3 Input Output Size

- The convolution layer change the channel number.
- The height and width computation for pooling layer is:

$$H_{out} = \left\lfloor \frac{H_{in} - k_h + 2 \times p}{s} \right\rfloor + 1$$

$$W_{out} = \left\lfloor \frac{W_{in} - k_w + 2 \times p}{s} \right\rfloor + 1$$

- The ReLU layer does not change the shape of data.

And table 1 shows the final output for each layer which change the data shape:

# 4 Part 2 Task 2

## 4.1 Simple Introduction

In this part, I will train my CNN on CIFAR-10 dataset with default parameters, and compare the different result with changing parameters following the single variable principle.

1. conv layer: k=3x3, s=1, p=1, in=3, out=64
2. maxpool: k=3x3, s=2, p=1, in=64, out=64
3. conv layer: k=3x3, s=1, p=1, in=64, out=128
4. maxpool: k=3x3, s=2, p=1, in=128, out=128
5. conv layer: k=3x3, s=1, p=1, in=128, out=256
6. conv layer: k=3x3, s=1, p=1, in=256, out=256
7. maxpool: k=3x3, s=2, p=1, in=256, out=256
8. conv layer: k=3x3, s=1, p=1, in=256, out=512
9. conv layer: k=3x3, s=1, p=1, in=512, out=512
10. maxpool: k=3x3, s=2, p=1, in=512, out=512
11. conv layer: k=3x3, s=1, p=1, in=512, out=512
12. conv layer: k=3x3, s=1, p=1, in=512, out=512
13. maxpool: k=3x3, s=2, p=1, in=512, out=512
14. linear, in=512, out=10

Figure 3: CNN Structure Detail

Table 1: Output of each layer in CNN

| **Layer** | **Output Size** |
|---|---|
| Conv2d | $32 \times 32 \times 64$ |
| MaxPool | $16 \times 16 \times 64$ |
| Conv2d | $16 \times 16 \times 128$ |
| MaxPool | $8 \times 8 \times 128$ |
| Conv2d | $8 \times 8 \times 256$ |
| Conv2d | $8 \times 8 \times 256$ |
| MaxPool | $4 \times 4 \times 256$ |
| Conv2d | $4 \times 4 \times 512$ |
| Conv2d | $4 \times 4 \times 512$ |
| MaxPool | $2 \times 2 \times 512$ |
| Conv2d | $2 \times 2 \times 512$ |
| Conv2d | $2 \times 2 \times 512$ |
| MaxPool | $1 \times 1 \times 512$ |
| Flatten | 512 |
| Linear | 10 |

## 4.2 Default Parameters

- `optimizer_type`: 'ADAM' (indicating `torch.optim.Adam`), type of optimizer for the model training
- `learning_rate`: $1e^{-4}$, learning rate for optimization
- `max_epochs`: 200, number of epochs to run trainer
- `eval_freq`: 1, frequency of evaluation on the test set
- `batch_size`: 32, batch size of single train batch

## 4.3 Result Visualization

Actually, in the beginning of this part, I train 1000+ epochs to get the result. But I found that the training accuracy will converge after 200 epochs. So follow TA's explanation, I use 200 epoch for the rest to get results faster.

Table 2 shows different parameters and corresponding figure.

Table 2: Different Parameters and Result

| Fig ID | Single Variable Content |
|---|---|
| Fig 4 | Default Parameters |
| Fig 9 | Change optimizer to `torch.optim.SGD` |
| Fig 10 | Change optimizer to `torch.optim.RMSprop` |
| Fig 11 | Change learning rate to $1e^{-3}$ |
| Fig 12 | Change learning rate to $5e^{-2}$ |

## 4.4 Result Analysis

### 4.4.1 Overall Analysis

In the following sections, unless specified, the accuracy refers to the test accuracy by default.

Comparing with MLP, we can clearly find that the test accuracy of CNN has been significantly improved. Not only has the accuracy of the training set increased rapidly, but more importantly, the accuracy of the test set has been significantly improved, with the highest level staying at around 85%. This directly shows that compared to MLP, CNN has made very significant progress on the CIFAR-10 data set, or by extension, the task of image classification.

However, by observing the curve, we can easily find that CNN cannot get rid of the problem of over-fitting. As the number of training increases, after a brief decline, the test loss slowly rises. In fact, through training with default parameters, it can be found that this increase still exists after 1000+ epochs.

In general, CNN still has its limitations. If we want to achieve better accuracy, more appropriate structures and better optimization strategies are needed.

### 4.4.2 Optimizer Analysis

In the comparison of optimizers, I found that Adam, RMSprop, as an optimizer, has a very good learning effect, and the accuracy within the specified 200 epochs can reach 85% in the end; in contrast, the final accuracy of SGD is lower. Only at 78-80%.

After search the information, I found that the biggest difference is that both Adam and RMSprop use adaptive learning rate optimization algorithms and can dynamically adjust the learning rate. Such dynamic adjustment can not only ensure the efficiency of back propagation updates, but also prevent it from falling into the trap of local optimality like SGD.

Among them, Adam adjusts the learning rate for each parameter based on the first and second moment estimates (mean and variance) of its gradient. And RMSprop adjusts the learning rate using the moving average of squared gradients, it uses the exponential decay average of squared gradients.

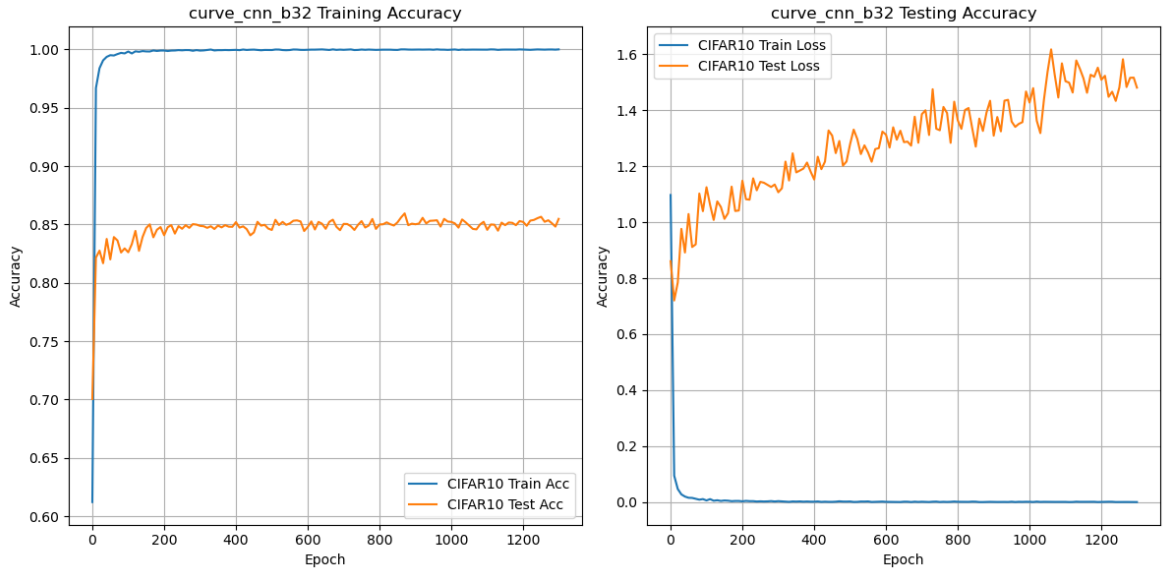
In image classification, a task with deeper networks, an optimizer with more strategies and an adaptive learning rate is more suitable. If we really want to find the advantages of SGD, it may be the train speed is slightly faster.

### 4.4.3 Class Accuracy Analysis

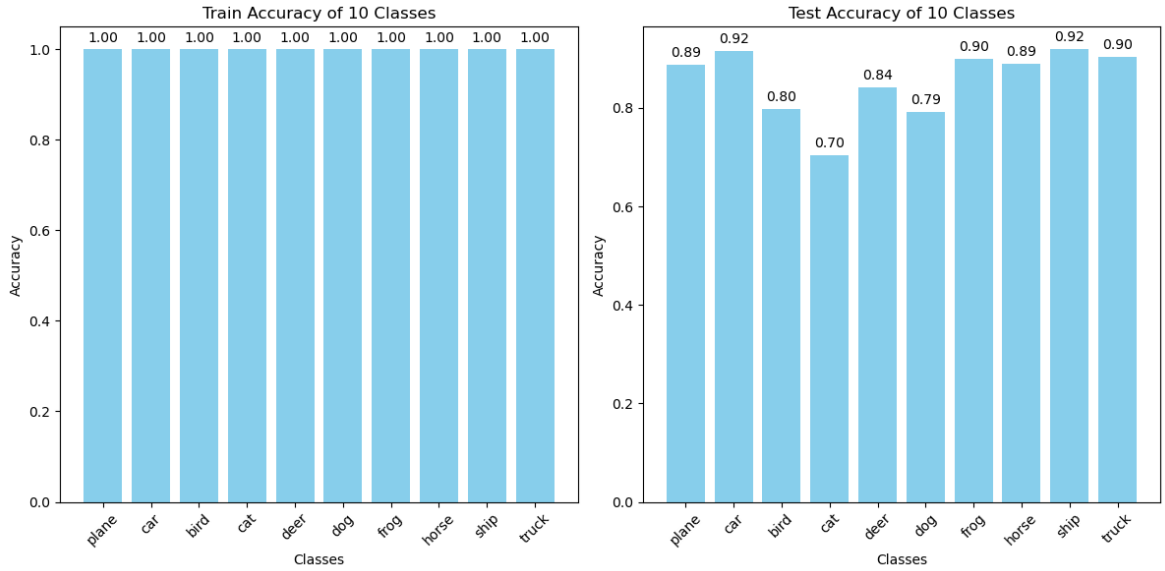
Through the type accuracy I output, we can also find interesting phenomena: the overall classification accuracy of car, ship, truck, etc. is significantly higher, while the overall classification accuracy of dog, cat, bird, etc. is lower.

We can make a bold guess: there are some similarities between animals, and cats and dogs themselves are especially similar. In contrast, frog, as an amphibian, does not seem to be similar so much.

Therefore, we can see that in image classification tasks, our human intuitive feelings are also has hidden relationship with the feature of the data itself.




(a) Accuracy and Loss




(b) Class Accuracy

Figure 4: Default Parameters

#### 4.4.4 Learning Rate Analysis

Comparing the learning rates, we can see that appropriately increasing the learning rate ($1e^{-3}$) can improve learning efficiency and speed up convergence (even with the assistance of Adam Optimizer). However, an inappropriately large learning rate ($5e^{-2}$) makes the upper limit of accuracy convergence lower, and the curve oscillates very violently. The step when adjusting the parameters each time is too large, resulting in continuous missing of the extreme value, and only keeps oscillating near the extreme value, making it impossible to get close.

This is why in practical deep learning, the learning rate should not be too large.

# 5 Part 3 Task 1

## 5.1 Simple Introduction

In this task, I need to create a simple RNN following the given formula. I shall realize the RNN without `torch.nn.RNN` or `torch.nn.LSTM`

## 5.2 Structure Analyze

### 5.2.1 RNN Modules

The requirement give series formula:

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

$$o^{(t)} = \left(W_{ph}h^{(t)} + b_o\right) \tag{2}$$

$$\tilde{y}^{(t)} = \text{softmax}(o^{(t)}) \tag{3}$$

According to these formula, I create 3 linear layer to realize the whole RNN operations:

```
    # w_hx * xt + bh
    self.hx = nn.Linear(input_dim, hidden_dim, bias=True)
    # w_hh * h_(t-1)
    self.hh = nn.Linear(hidden_dim, hidden_dim, bias=False)
    # o_t = w_ph * h_t + b_o
    self.ho = nn.Linear(hidden_dim, output_dim, bias=True)
```

And I will explain the function of them:

1. hx: $1 \to 128$. Linear Layer with bias, implement equation 1 $x^{(t)}$ part, let new input digit to hidden layer size.
2. hh: $128 \to 128$. Linear Layer without bias, implement equation 1 $h^{(t-1)}$ part, let last time $h$ output get a weight in computation of new $h$.
3. ho: $128 \to 10$. Linear Layer with bias, implement equation 2. Get the final predicted number from $[0, 9]$.
4. a softmax layer, implement equation 3

### 5.2.2 Forward Propagation

When doing forward propagation:

```
def forward(self, x):
    batch_size, input_length = x.size(0), x.size(1)
    h_last = torch.zeros(batch_size, self.hidden_dim).to(self.device)
    for t in range(input_length):
        x_cur = x[:, t, :]
        h_last = torch.tanh(self.hx(x_cur) + self.hh(h_last))
    out = self.softmax(self.ho(h_last))
    return out
```

I use a `for` loop to let the $x$ and previous $h$ participant the computation together, implement the RNN computation. And finally get the output.

After forward propagation, let the max value of the 10 dimension item become the prediction result.

# 6 Part 3 Task 2

## 6.1 Simple Introduction

In this part, I will train my RNN on palindrome dataset with input length 4 and 19, and let the RNN predict the last number of the palindrome number.

## 6.2 Default Parameters

- `input_length`: 19, length of input sequence
- `input_dim`: 1, dimension of input data
- `num_classes`: 10, number of classes in the classification task
- `num_hidden`: 128, number of hidden units in the neural network
- `batch_size`: 128, batch size for training
- `learning_rate`: 0.001, learning rate for optimization
- `max_epoch`: 1000, maximum number of epochs to train the model
- `max_norm`: 10, maximum norm constraint for gradient clipping
- `data_size`: 1000000, size of the dataset
- `portion_train`: 0.8, portion of the dataset used for training

And I use `torch.optim.lr_scheduler.StepLR()` (and step_size=10, gamma=0.7) for scheduler, but it has some interesting unexpected influence. Let me analysis it in the following part.

## 6.3 Result Visualization

Fig 5 show a $T = 5$ for RNN training, while fig 13 show a $T = 20$.

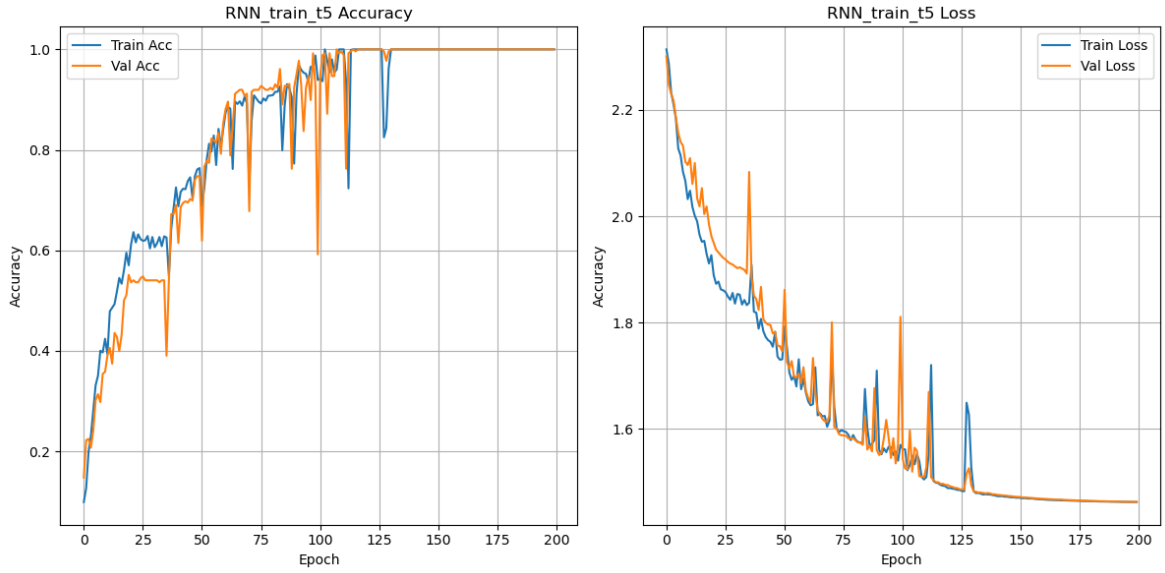
## 6.4 Result Analysis

Through actual operations, I discovered a very strange situation: when training with $T = 5$, 100% accuracy can be achieved without the scheduler. After using the scheduler, this increase becomes very slow; on the contrary, in $WhenT = 20$, not using the scheduler will cause the model to degrade, but using it will keep the model 100% accurate.

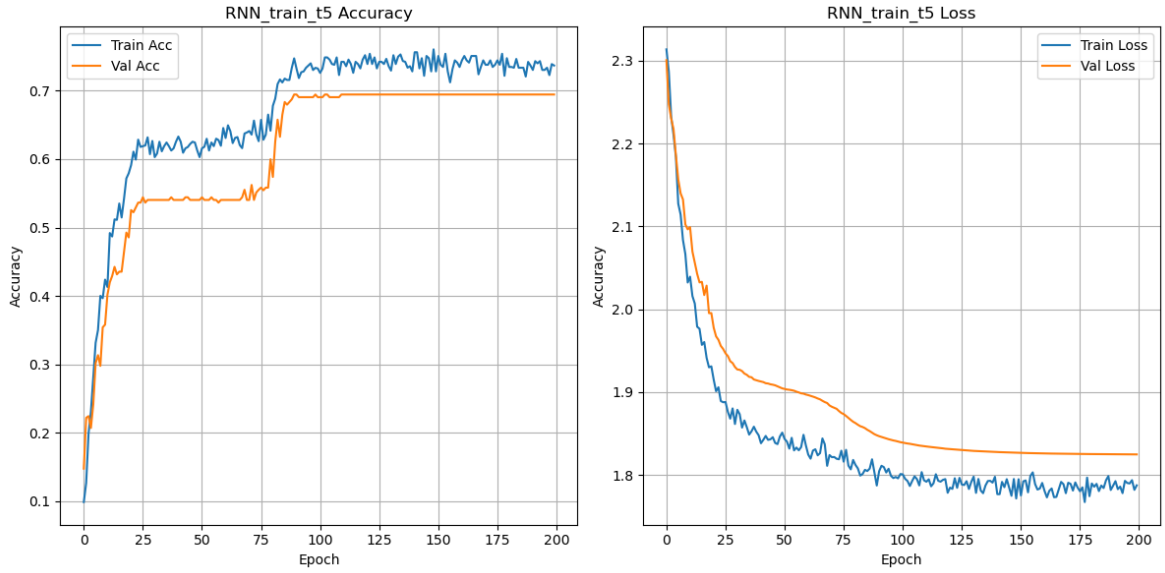
I compare the training success and "failure" curves together and guess some possible reasons.

- First, analyze $T = 5$. We know that `StepLR` can multiply the learning rate with $\gamma$ after a certain steps. This multiplication may cause an exponential explosion after several times, and the learning rate gradually drops to a very small value. Observing the normal curve without scheduler, we can find that the model reaches perfect accuracy after about 150 epochs. Therefore, if a scheduler is used, the model will not be able to effectively complete learning within a certain epoch, resulting in lower accuracy. But the learning rate still increased until the end
- Second, for $T = 20$, why does the accuracy become higher after using the scheduler? Observing the failure curve which without using the scheduler, we can find that the accuracy of the model reached a very high value at the beginning, but as the training continued, the accuracy experienced several sudden drops, and loss also appeared corresponding rising.

  My guess is that during the training process, it achieved very high accuracy at the beginning, but it did not perform a early stop, and gradient explosion occurred later. Although I used




(a) $T = 5$ Normal Curve (without scheduler)




(b) $T = 5$ Fail Curve (with scheduler)

Figure 5: $T = 5$ Curves

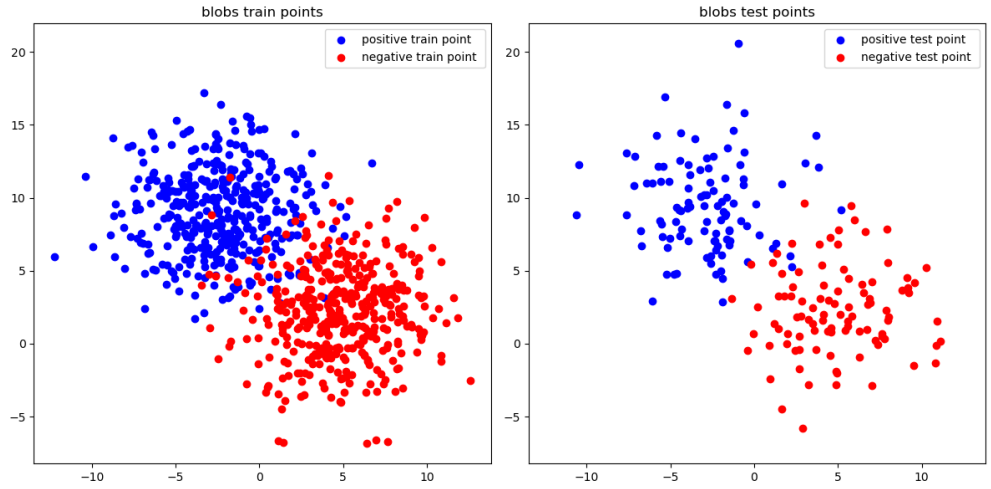
`clip_grad_norm_()` correctly, it may be that the value of norm is too high, making it increasingly difficult to correct gradient explosion problem. Moreover, with the vanishing gradient problem caused by long dependencies of RNN, this makes correct learning more difficult.

The use of the scheduler may have corrected this unexpected: after the epoch rises and before the gradient explosion occurs, the learning rate has been reduced to a very low value, simulating a early stop, so the impact of the gradient explosion has become very small, make it getting rid of significantly affect the entire model.

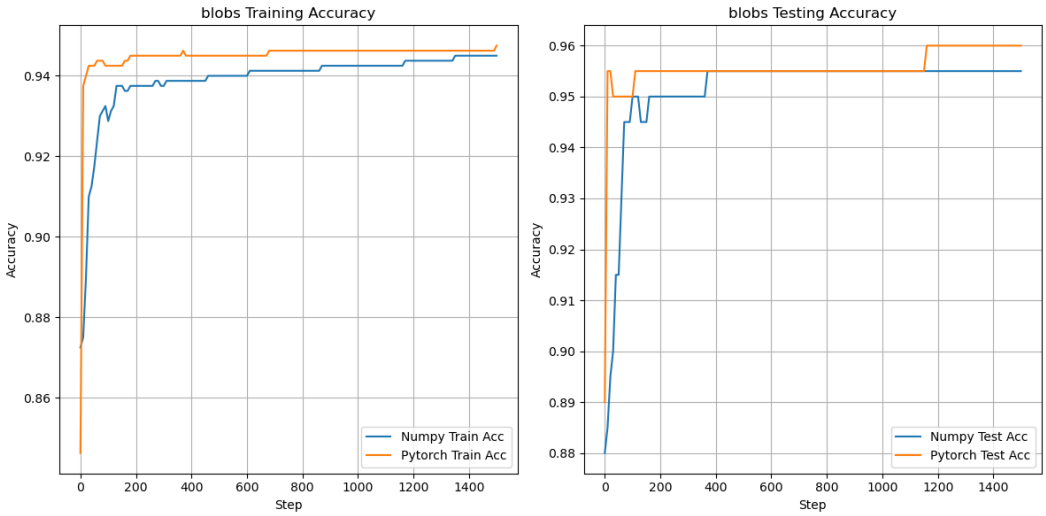
Of course, the above is just guess which is not rigorous. Usually, in general, we will find that when the input length is low, RNN can eventually achieve a higher result, but once the input length becomes longer, its accuracy is very easy to decline. As the training continued, the model may seriously degrade. This can reflect a characteristic of RNN: the model's ability of memory is not very good. Once the input dependencies become longer, the problem of vanishing gradient occurs during training, which leads to poor learning results. The model will become very poor.

So in comparison, maybe using LSTM is a good choice.

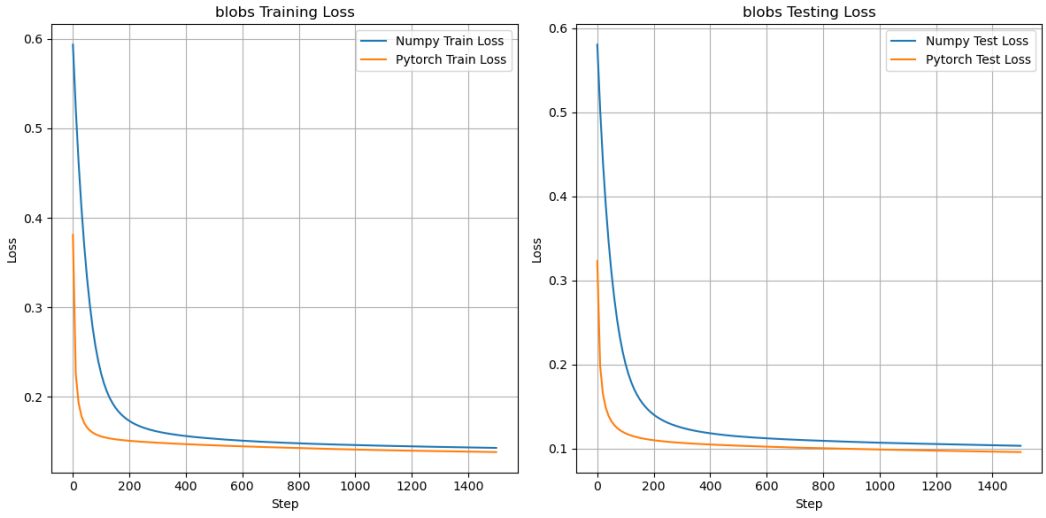
# A Extra Pictures




(a) Moon Data

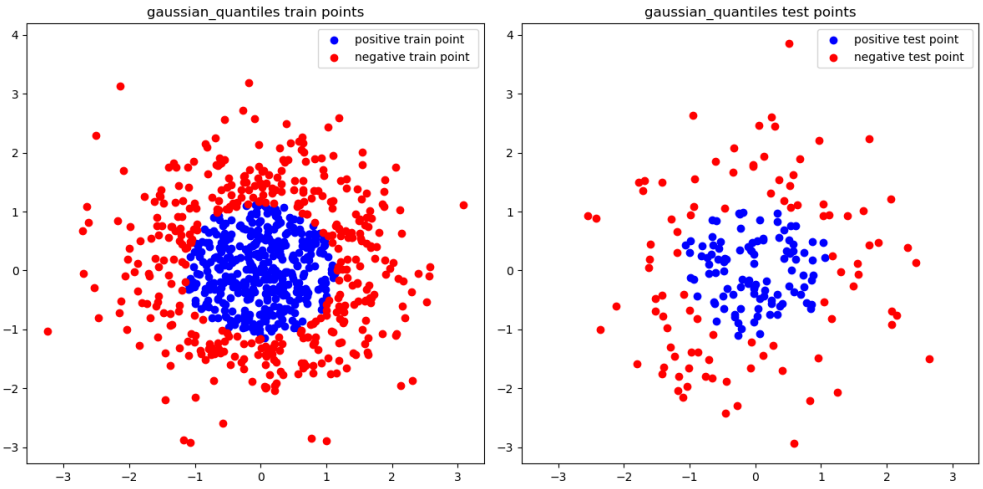



(b) Moon Accuracy

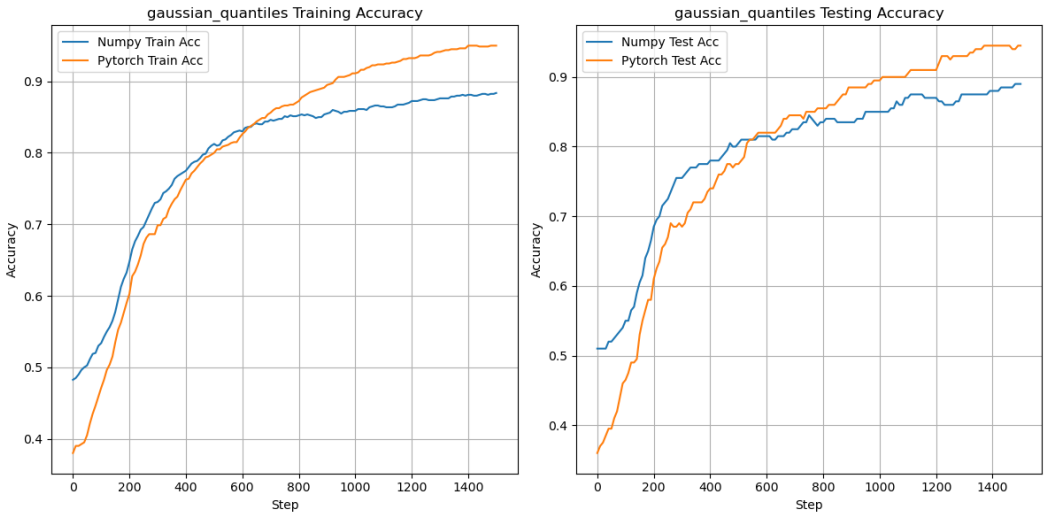



(c) Moon Loss

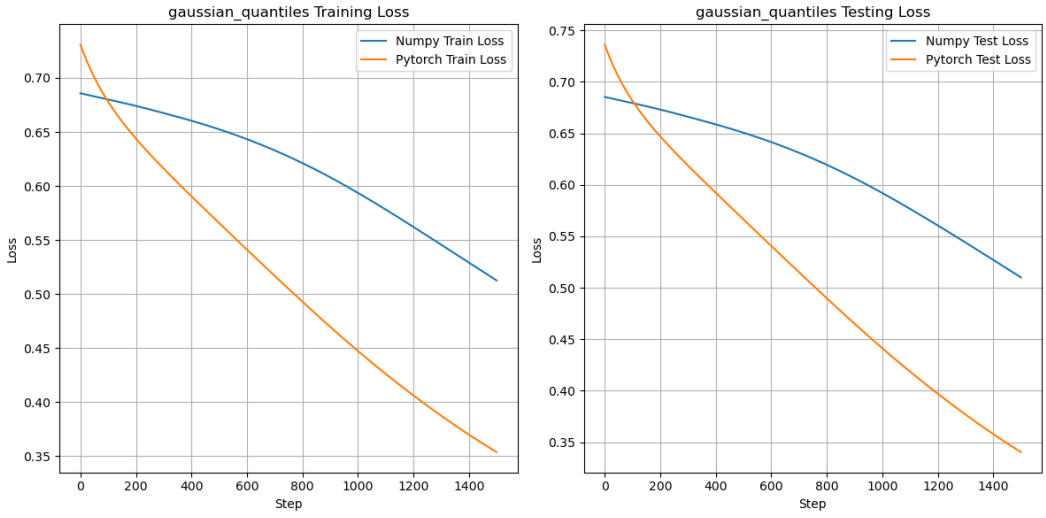
Figure 6: Moon numpy & torch




(a) Gaussian Quantiles Data




(b) Gaussian Quantiles Accuracy




(c) Gaussian Quantiles Loss

Figure 7: Gaussian Quantiles numpy & torch

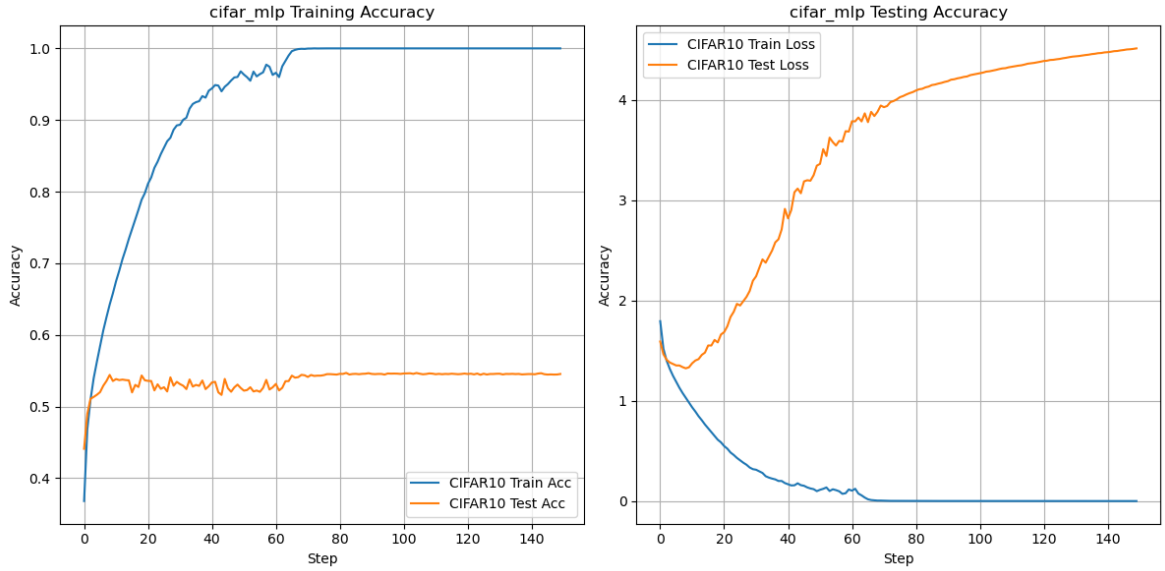
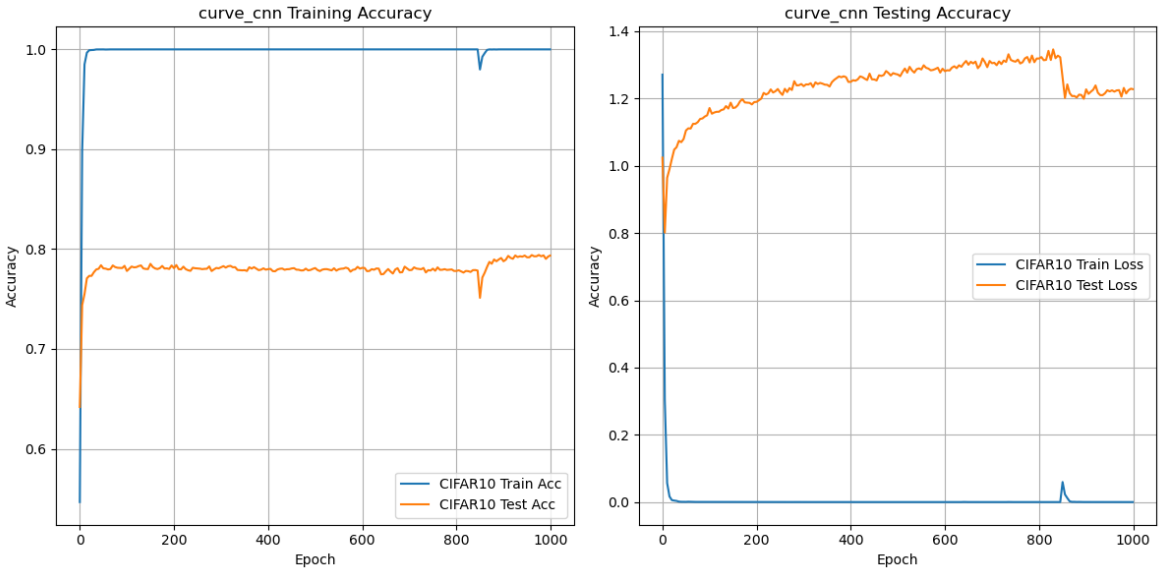
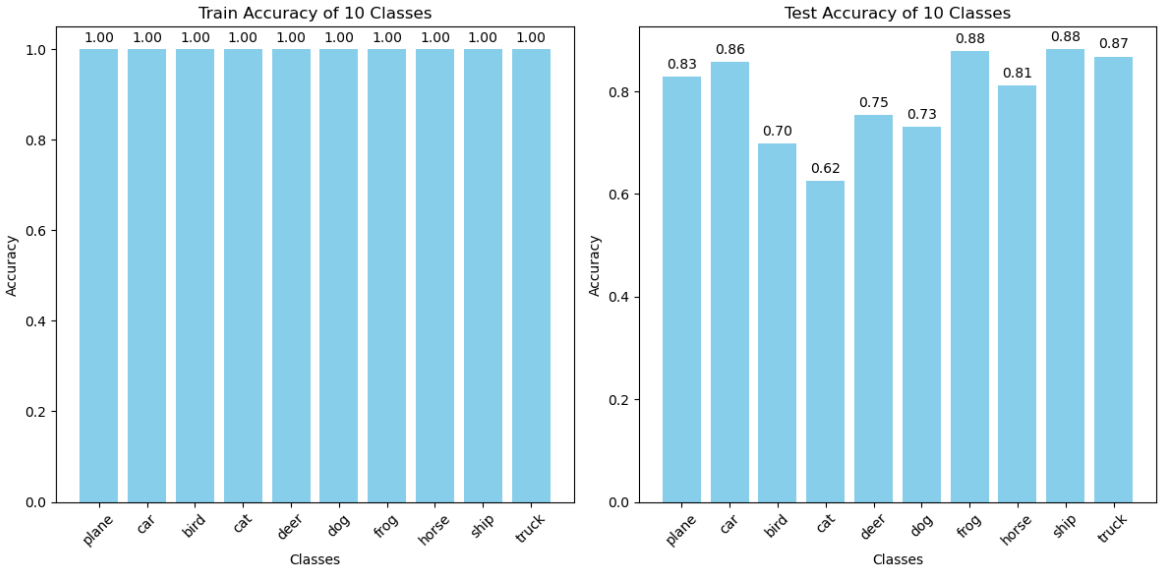



Figure 8: MLP CIFAR-10 Curve with SGD

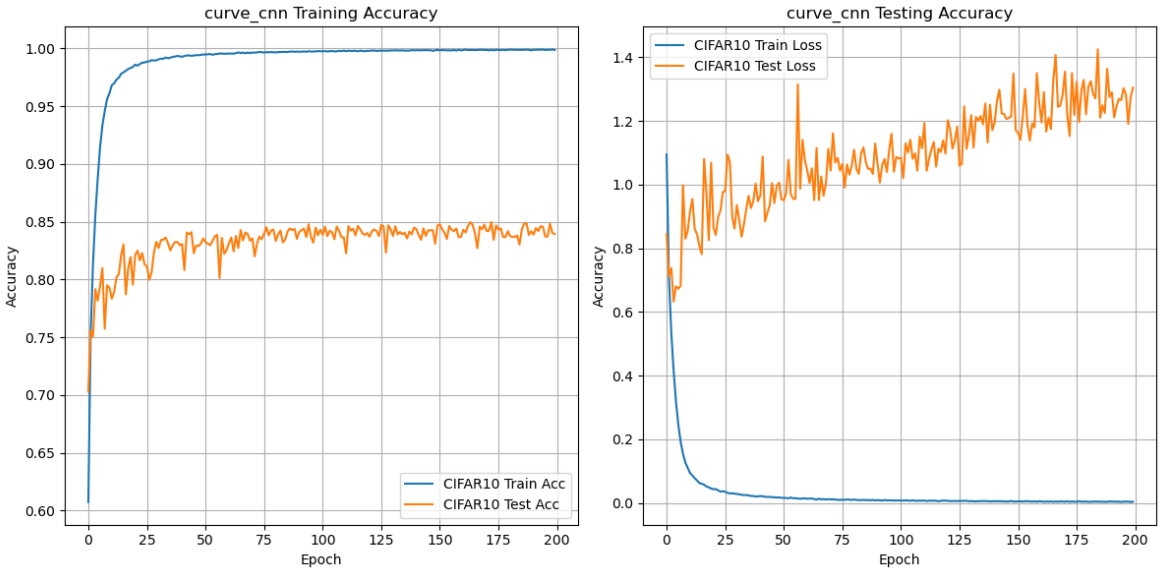



(a) Accuracy and Loss

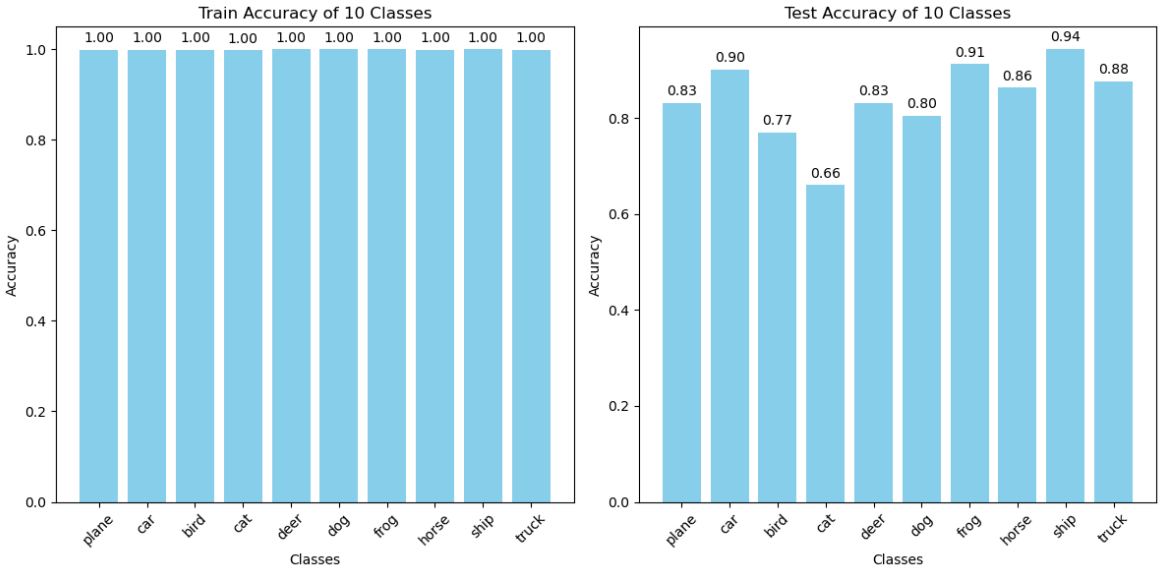



(b) Class Accuracy

Figure 9: Optimizer SGD

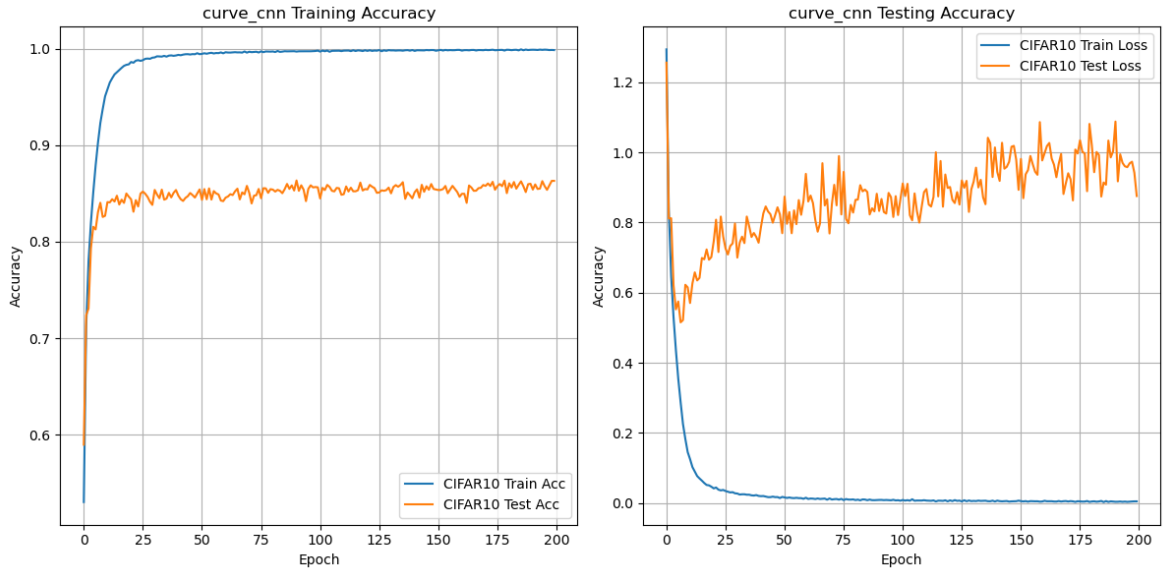



(a) Accuracy and Loss

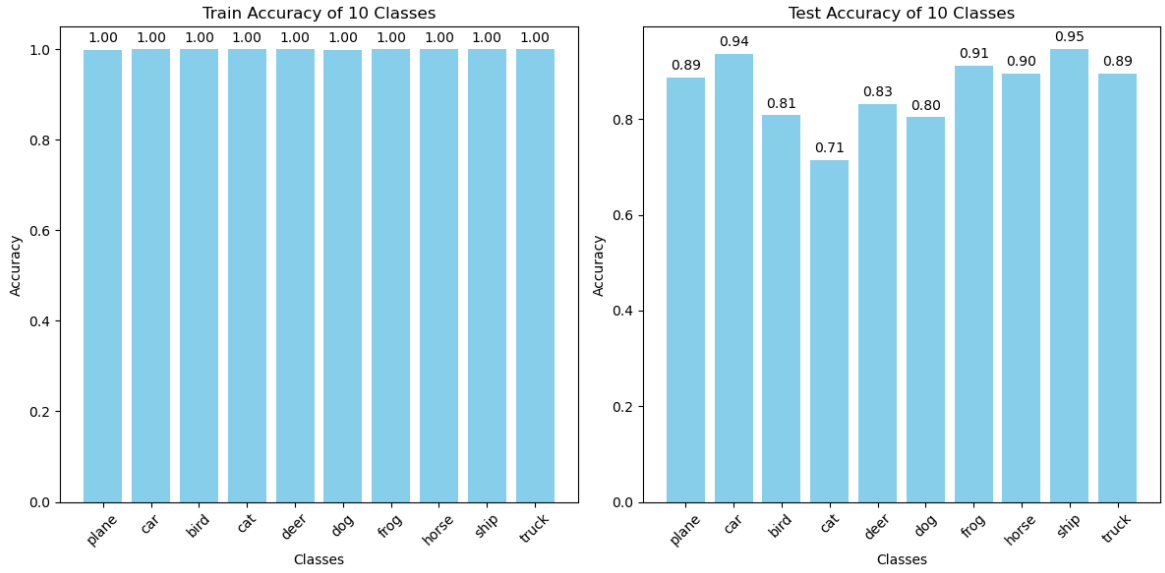



(b) Class Accuracy

Figure 10: Optimizer RMS

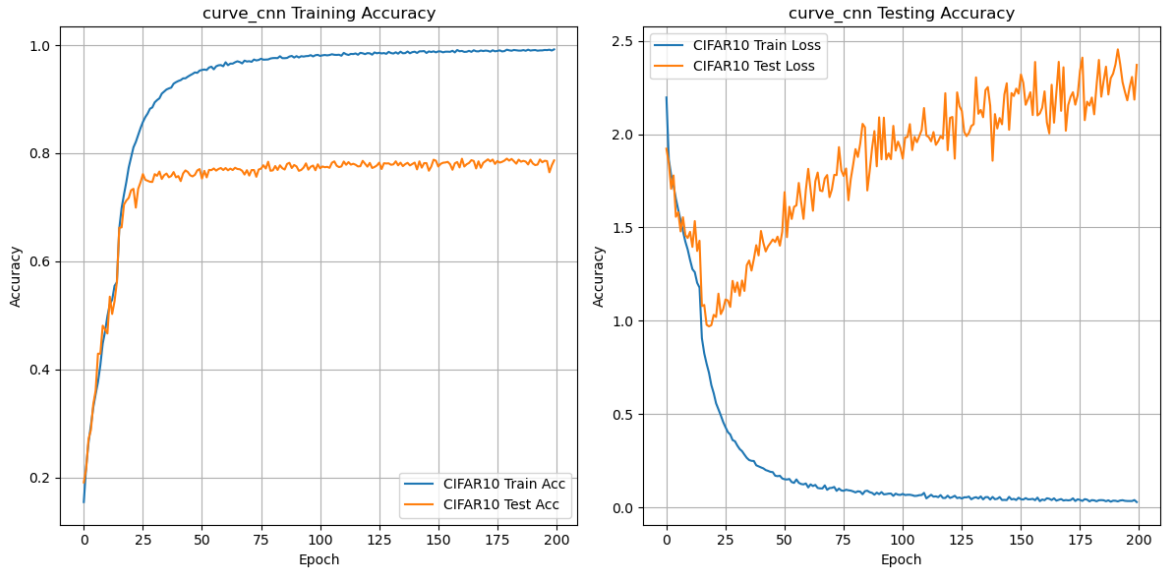



(a) Accuracy and Loss

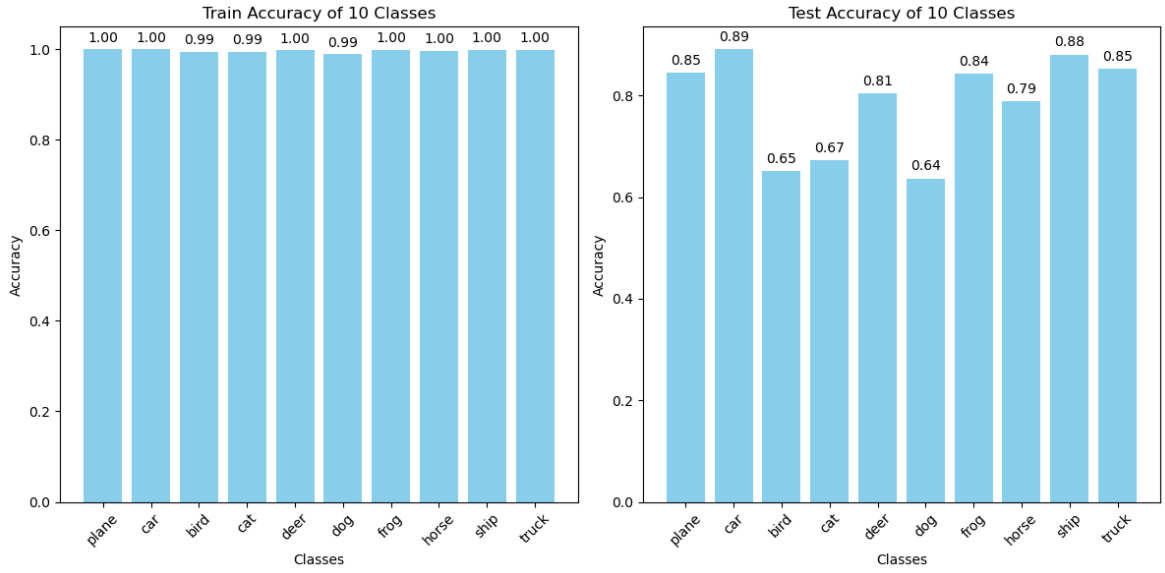



(b) Class Accuracy

Figure 11: Learning Rate $1e^{-3}$

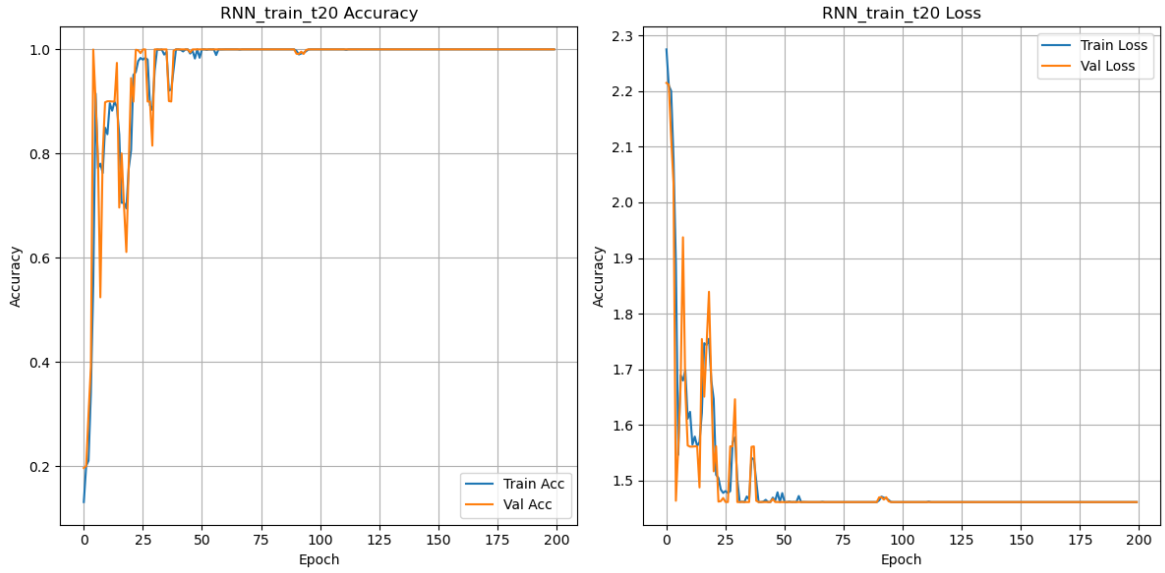



(a) Accuracy and Loss

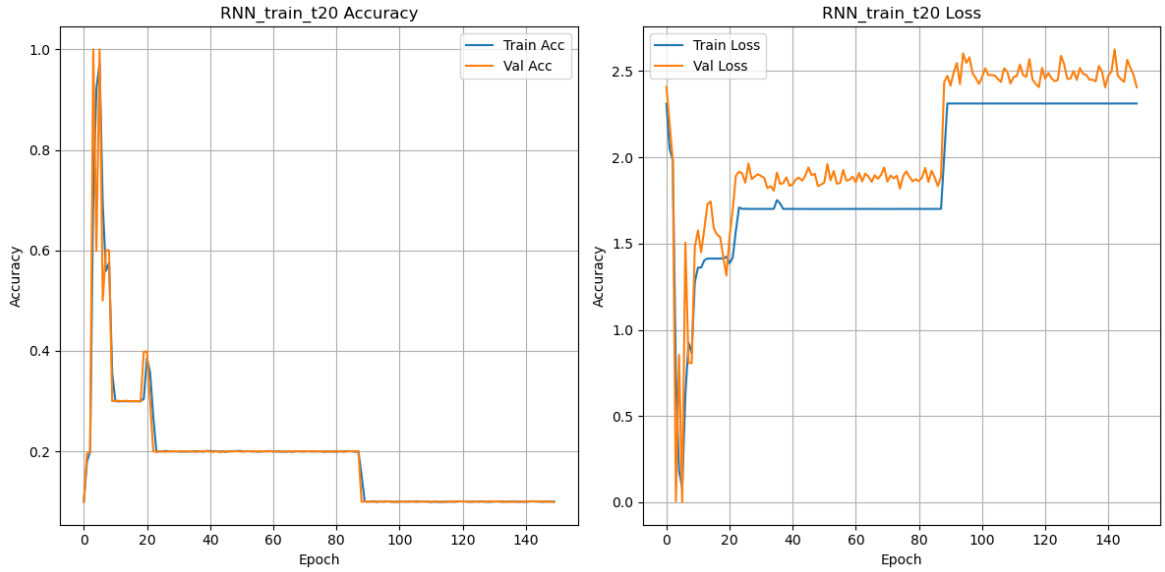



(b) Class Accuracy

Figure 12: Learning Rate $5e^{-2}$




(a) $T = 20$ Normal Curve (with scheduler)




(b) $T = 20$ Fail Curve (without scheduler)

Figure 13: $T = 20$ Curve