Deep Learning Assignment 1

실험 History

시작 parameter 값

```
self.linear1 = nn.Linear(32*32*3, 1024)
self.linear2 = nn.Linear(1024, 512)
self.linear3 = nn.Linear(512, 256)
self.linear4 = nn.Linear(256, 128)
self.linear5 = nn.Linear(128, 64)
self.linear6 = nn.Linear(64, 10)

배치 노멀라이제이션 적용
dropout 0.5 적용

배치사이즈 = 128
lr = 0.001
Adam optimizer Betas = [0.9, 0.999]
lmbd = 0.003

Accuracy: 54.88%
```

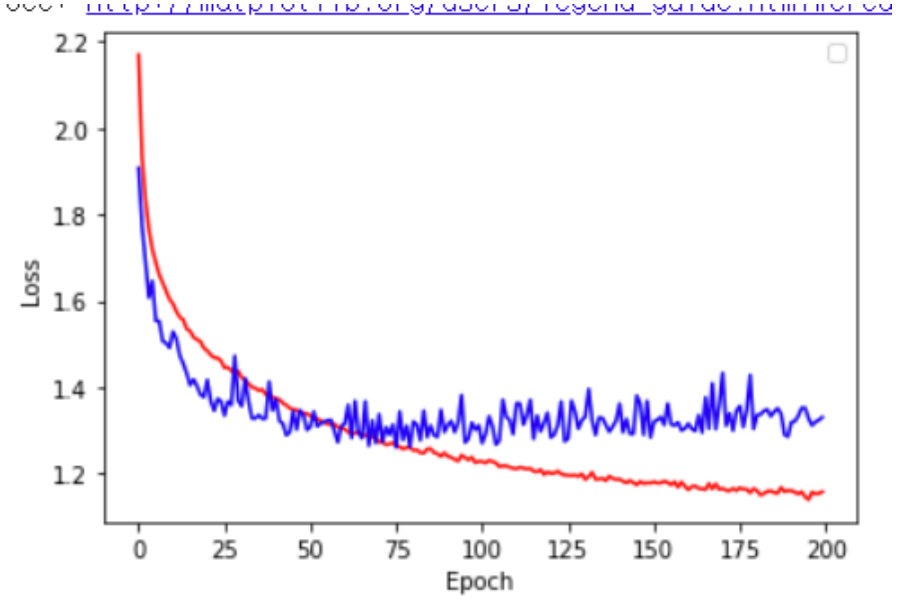
Trial 1

linear 4 256 64

linear 5 64, 32

linear 6 32, 10 로 변경

dropout2 = (0.3)을 추가하여 학습

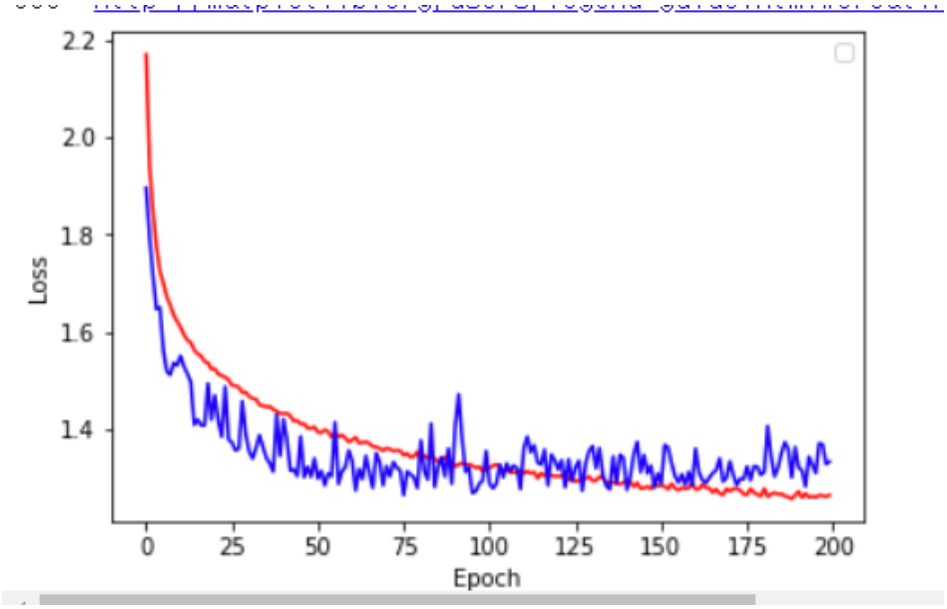


Accuracy: 55.83%

Trial 2

분산값을 줄이기 위해 lmbd 값 조절

lmbd = 0.003 → 0.005



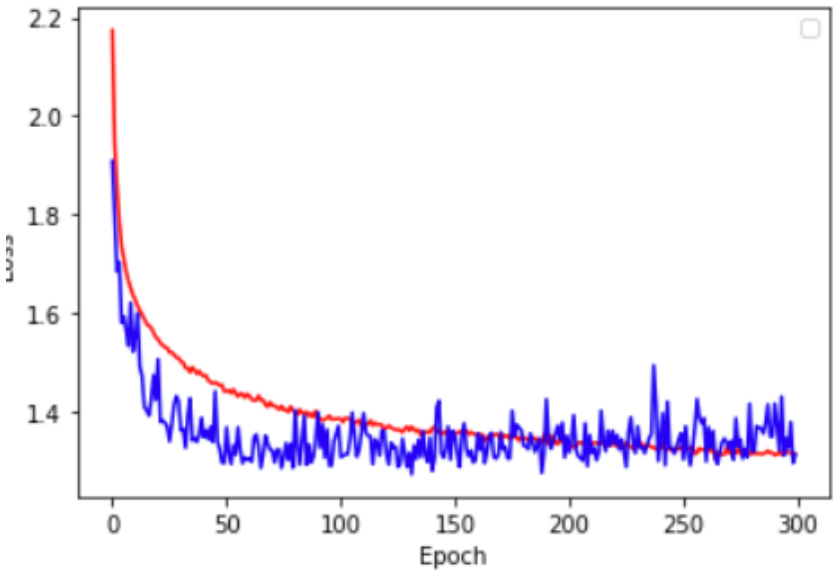
Accuracy: 55.4%

Trial 3

분산값을 더욱 줄이기 위해 lmbd 조절

lmbd = 0.005 → 0.007

epochs = 300



55.38%

분산값은 줄긴 하지만, 학습률도 동시에 떨어짐.

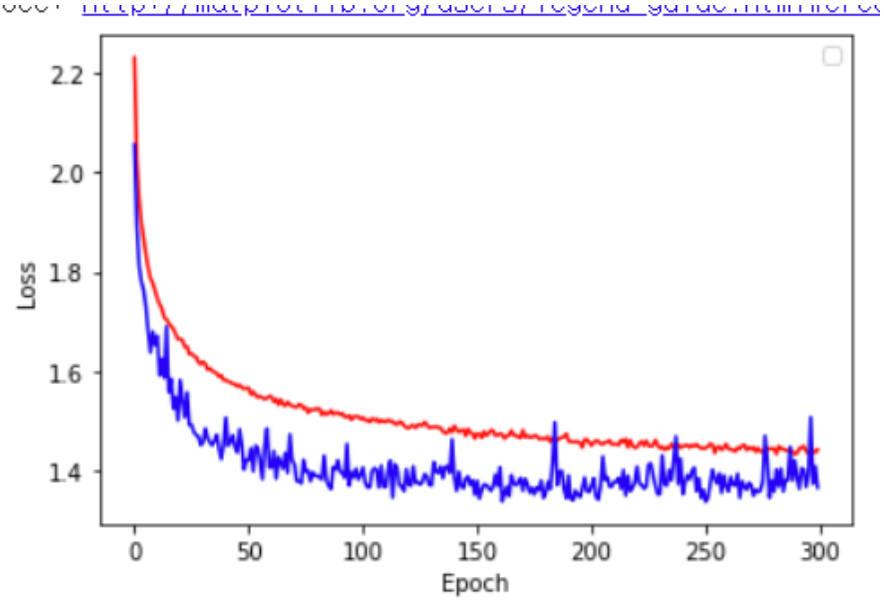
눈에 띄는 정확도 향상은 없었음.

교수님이 수업시간에 레이어가 많을 수록 더 세련된 모델이 나온다고 말씀하셨던 것에서 착안하여, 레이어 수를 늘려보기로 함.

Trial 4

```
def __init__(self, drop_prob):
    super(Classifier, self).__init__()
    self.linear1 = nn.Linear(32*32*3, 512)
    self.linear2 = nn.Linear(512, 256)
    self.linear3 = nn.Linear(256, 128)
    self.linear4 = nn.Linear(128, 64)
    self.linear5 = nn.Linear(64, 32)
    self.linear6 = nn.Linear(32, 16)
    self.linear7 = nn.Linear(16, 10)
```

dropout2 0.2

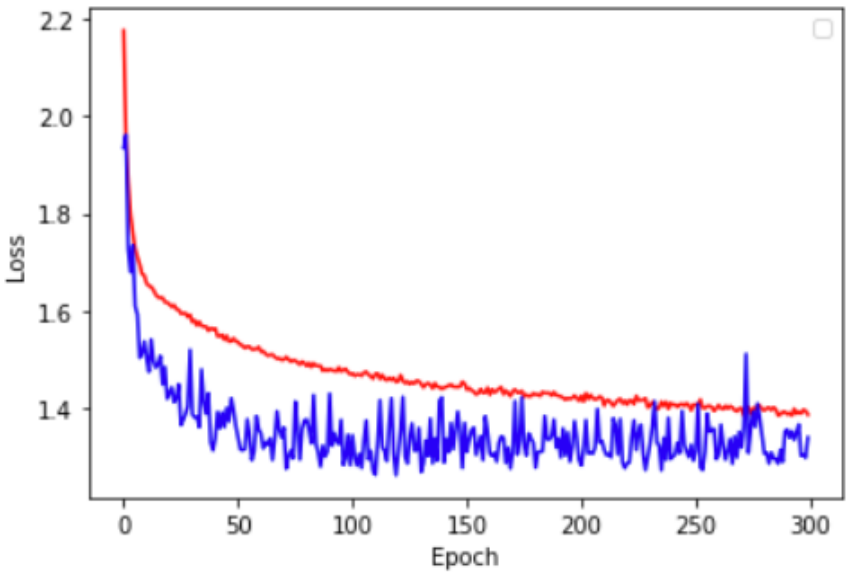


레이어 수만 올리니 LOSS가 증가한 모습

레이어 수와 뉴런 수를 조금 더 늘려보기로 함.

Trial 5

```
super(DrasaNet, self).__init__()
self.linear1 = nn.Linear(32*32*3, 2048)
self.linear2 = nn.Linear(2048, 1024)
self.linear3 = nn.Linear(1024, 512)
self.linear4 = nn.Linear(512, 256)
self.linear5 = nn.Linear(256, 128)
self.linear6 = nn.Linear(128, 64)
self.linear7 = nn.Linear(64, 32)
self.linear8 = nn.Linear(32, 10)

self.bn1 = nn.BatchNorm1d(2048)
self.bn2 = nn.BatchNorm1d(1024)
self.bn3 = nn.BatchNorm1d(512)
self.bn4 = nn.BatchNorm1d(256)
self.bn5 = nn.BatchNorm1d(128)
self.bn6 = nn.BatchNorm1d(64)
self.bn7 = nn.BatchNorm1d(32)


self.dropout = nn.Dropout(drop_prob)
self.dropout2 = nn.Dropout(0.3)
self.activation = nn.ReLU()
```



Accuracy: 56.16%

정확도가 상승하긴 했음.

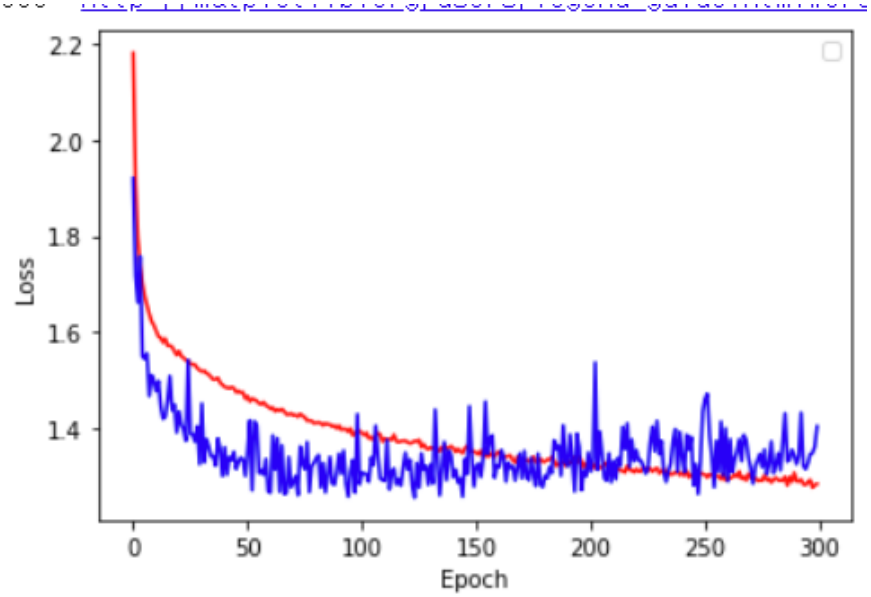
LOSS를 더 줄여야 하지 않을까 생각.

중간 뉴런을 더 늘려보기로 함.

Trial 6

중간뉴런을 512로 조금 더 늘려봤음.

```
def __init__(self, drop_prob):
    super(Classifier, self).__init__()
    self.linear1 = nn.Linear(32*32*3, 2048)
    self.linear2 = nn.Linear(2048, 1024)
    self.linear3 = nn.Linear(1024, 512)
    self.linear4 = nn.Linear(512, 512)
    self.linear5 = nn.Linear(512, 256)
    self.linear6 = nn.Linear(256, 128)
    self.linear7 = nn.Linear(128, 32)
    self.linear8 = nn.Linear(32, 10)

    self.bn1 = nn.BatchNorm1d(2048)
    self.bn2 = nn.BatchNorm1d(1024)
    self.bn3 = nn.BatchNorm1d(512)
    self.bn4 = nn.BatchNorm1d(512)
    self.bn5 = nn.BatchNorm1d(256)
    self.bn6 = nn.BatchNorm1d(128)
    self.bn7 = nn.BatchNorm1d(32)
```



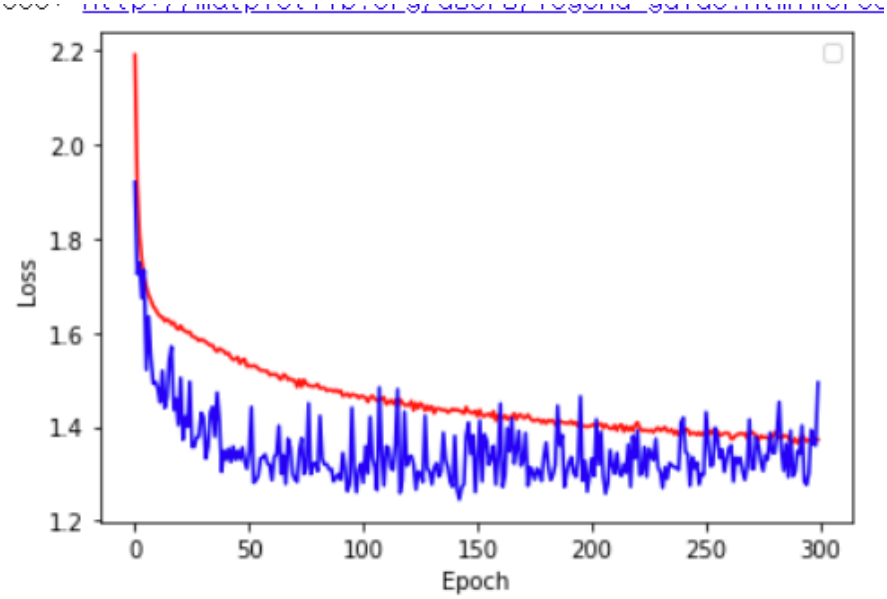
LOSS를 줄이는 것에 성공적 300정도에서 1.28이 나옴.

200에서는 1.32 확실히 LOSS가 줄었음.

여기서 분산값을 더 줄이기위해 정규화 값을 더 세게 줘보기로 함.

lambda 0.007에서 → lambda 0.01

Trial 7



람다값이 높아지니 학습이 잘 안되는 문제가 발생.

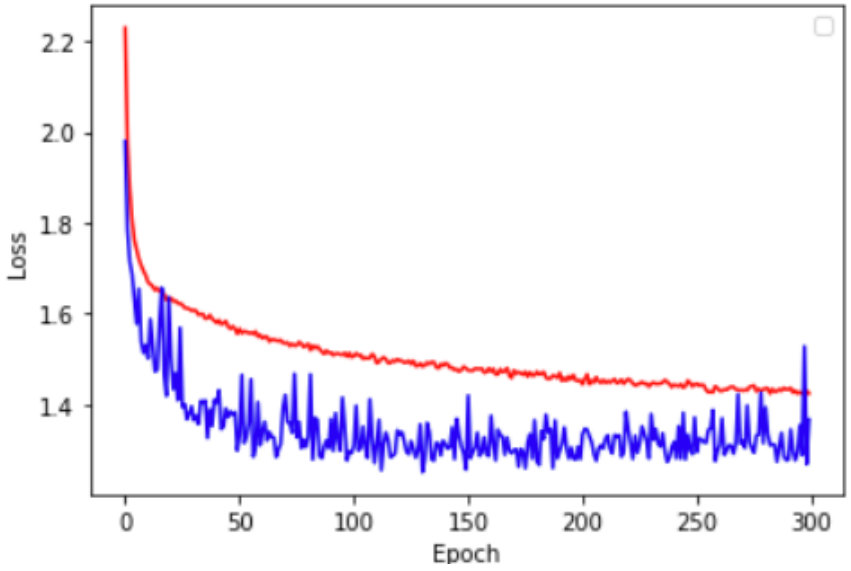
Trial 8

람다 값 대신에 dropout값을 높혀보기로 함

lambda 0.01 → 0.007

dropout 0.55 → 0.6

dropout2 0.3 →0.35



Accuracy: 54.8%

Loss값이 너무 높아지는 문제 발생

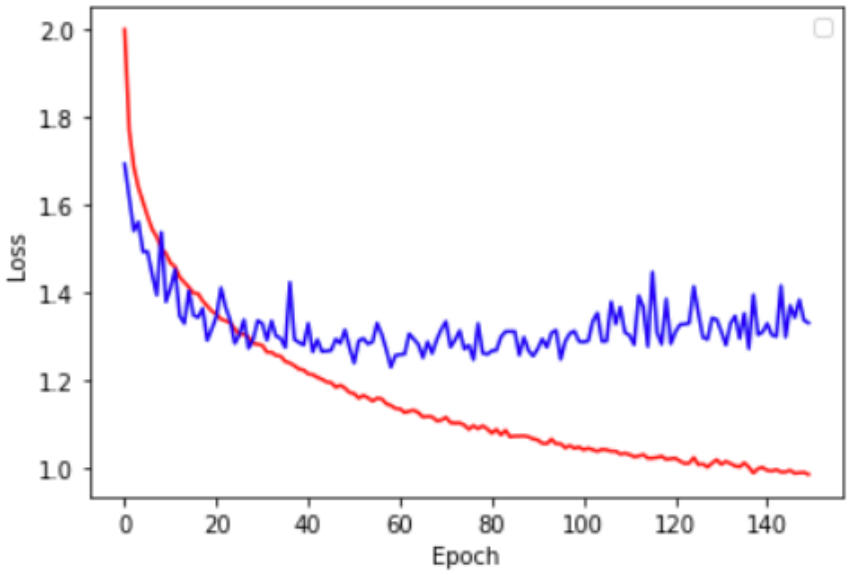
Trial 9

레이어와 뉴런수를 줄여보기로 함

```
self.linear1 = nn.Linear(32*32*3, 512)
self.linear2 = nn.Linear(512, 512)
self.linear3 = nn.Linear(512, 256)
self.linear4 = nn.Linear(256, 64)
self.linear5 = nn.Linear(64, 10)
```

dropout 0.5

dropout2 0.3

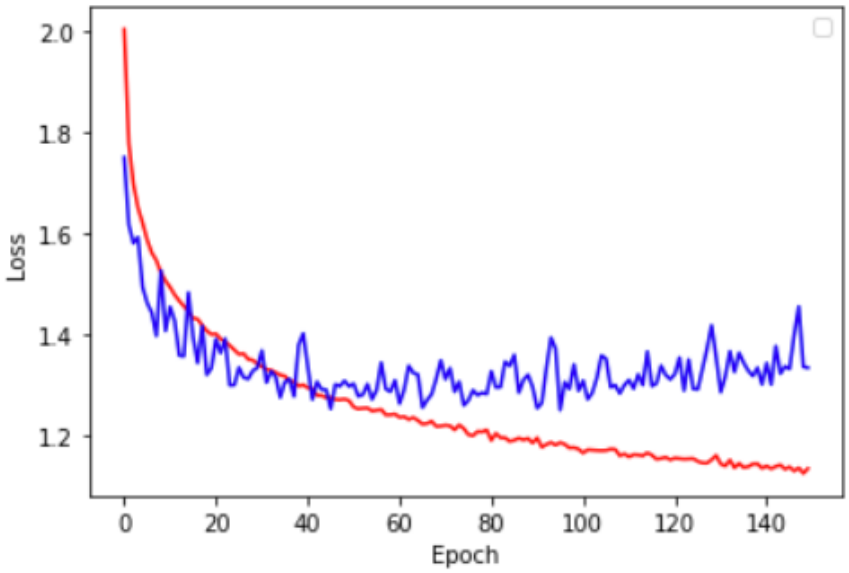


Accuracy: 56.45%

확실히 LOSS값이 줄긴 했음.

Trial 10

분산값 더 줄이기 위해 lambda값 더 올려 0.003 → 0.007



Accuracy: 54.75%

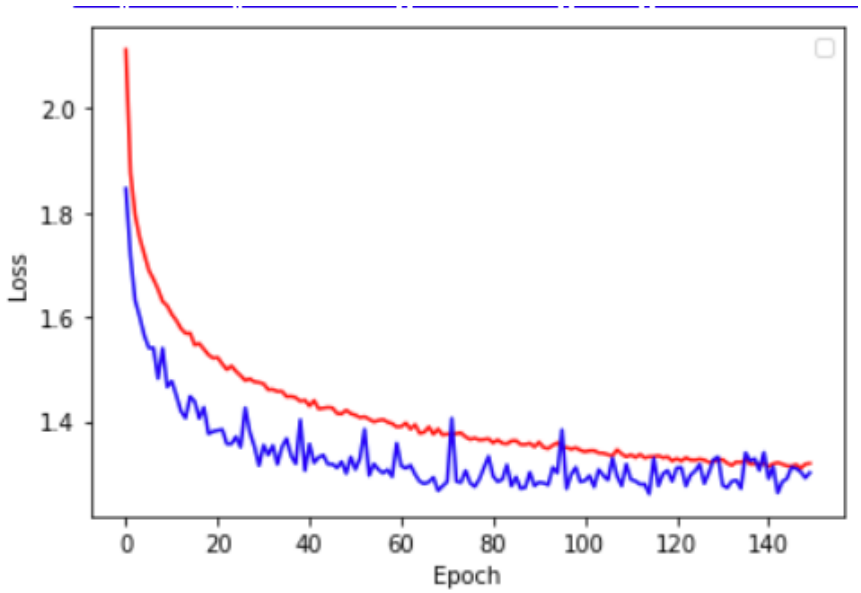
학습이 잘 안됨.

Trial 11

람다값 0.007→ 0.005로 변경 후 dropout을 좀 만져주기로 함

dropout 0.5 → 0.6

dropout2 0.3→ 0.4

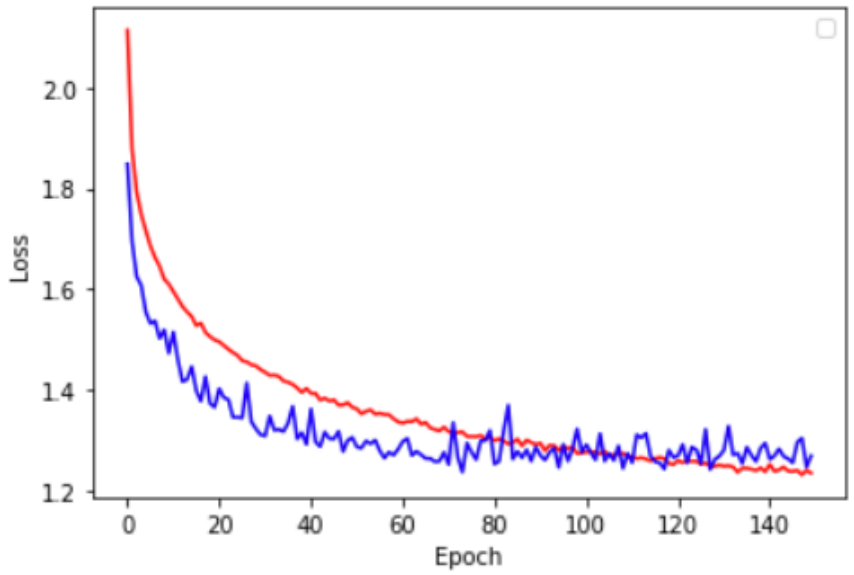


Accuracy: 54.17%

LOSS가 1.31정도까지 내려감

Trial 12

lambda값 lmbd = 0.003 다시 내리고 dropout만 유지



130 LOSS 1.31

Accuracy: 56.04%

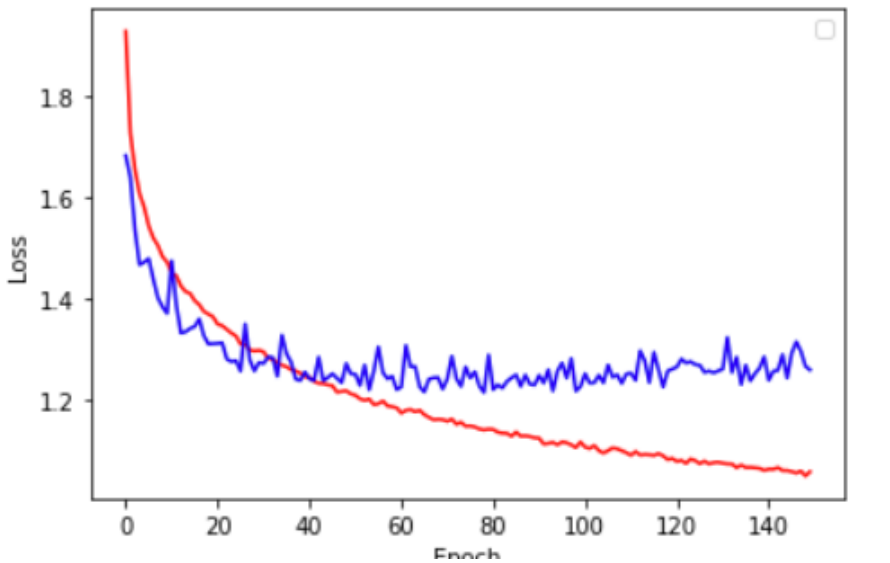
Trial 13

```
self.dropout  = nn.Dropout(0.6)
self.dropout2 = nn.Dropout(0.3)
self.dropout3 = nn.Dropout(0.2)
self.activation = nn.ReLU()
```

dropout3 만들어서 적용

```
self.linear1 = nn.Linear(32*32*3, 512)
self.linear2 = nn.Linear(512, 512)
self.linear3 = nn.Linear(512, 256)
self.linear4 = nn.Linear(256, 64)
self.linear5 = nn.Linear(64, 10)

self.bn1 = nn.BatchNorm1d(512)
self.bn2 = nn.BatchNorm1d(512)
self.bn3 = nn.BatchNorm1d(256)
self.bn4 = nn.BatchNorm1d(64)

self.dropout = nn.Dropout(0.6)
self.dropout2 = nn.Dropout(0.3)
self.dropout3 = nn.Dropout(0.2)
self.activation = nn.ReLU()

forward(self, x):
z1 = self.linear1(x)
z1 = self.bn1(z1)
a1 = self.activation(z1)
a1 = self.dropout(a1)

z2 = self.linear2(a1)
z2 = self.bn2(z2)
a2 = self.activation(z2)
a2 = self.dropout(a2)

z3 = self.linear3(a2)
z3 = self.bn3(z3)
a3 = self.activation(z3)
a3 = self.dropout2(a3)
```

```
See: http://matplotlib.org/users/legend_guide.html#crea
```

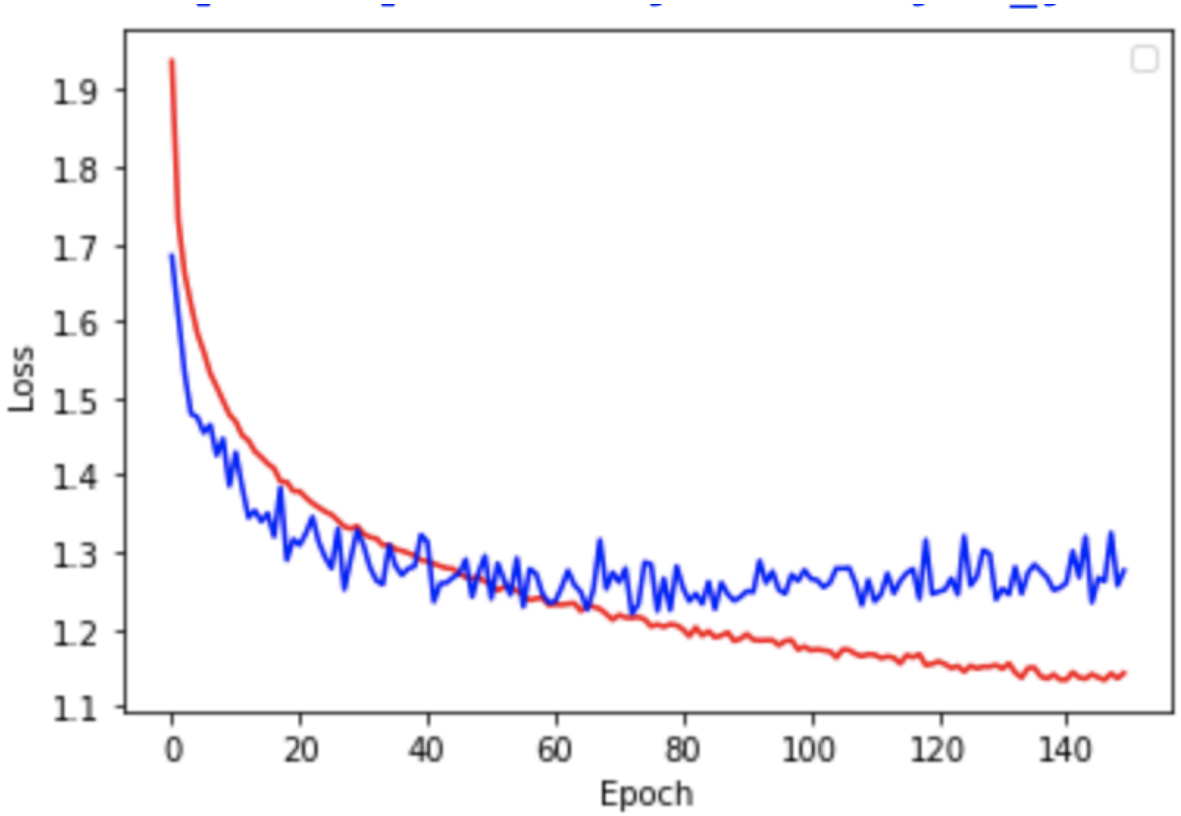



```
Accuracy: 56.49%
```

Trial 14

람다값 올려서 분산값 잡아보기로 함.

0.003→0.005



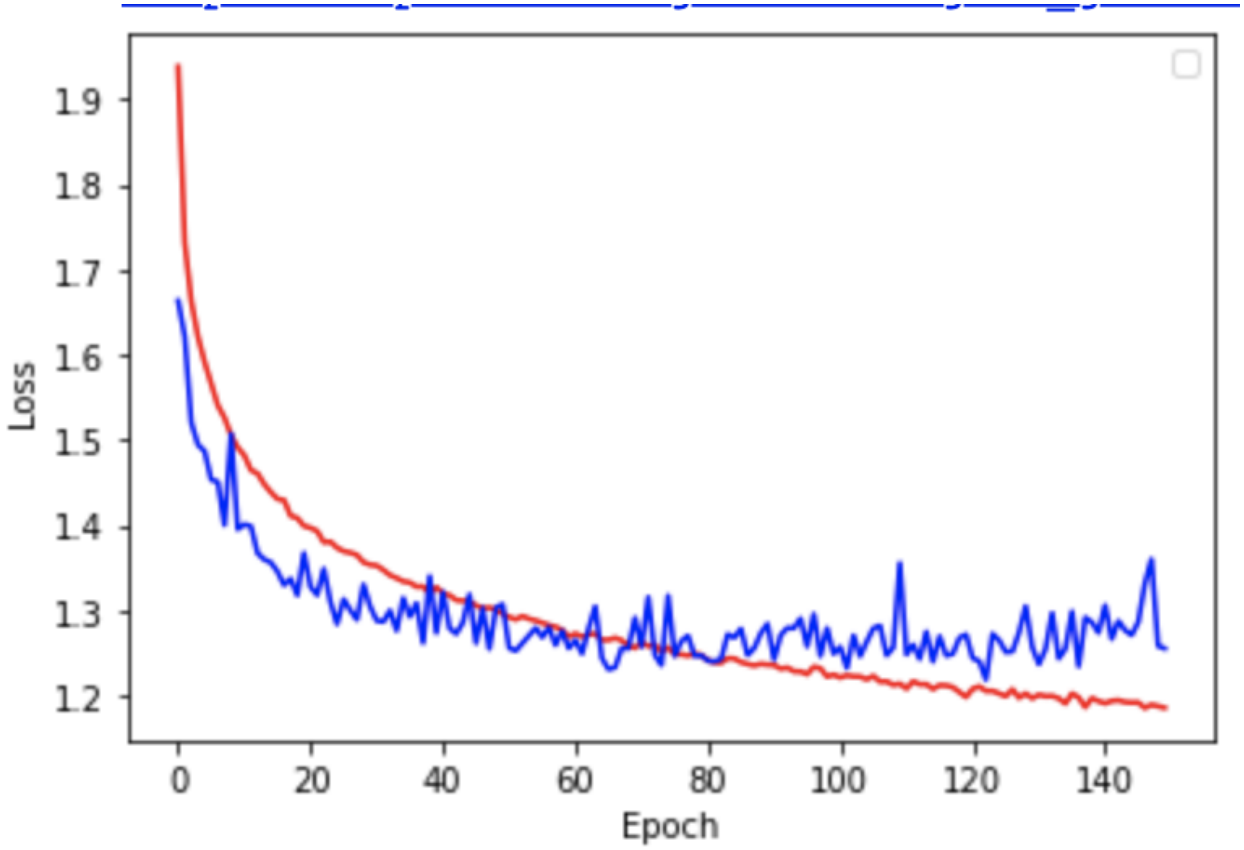
1.141까지는 떨어짐

Accuracy 55.3%

Trial 15

분산값을 더 줄이기 위해

lambda 0.005 → 0.007



Accuracy: 55.68%

확실히 분산이 잡히긴 하는 것 같음.

Trial 16

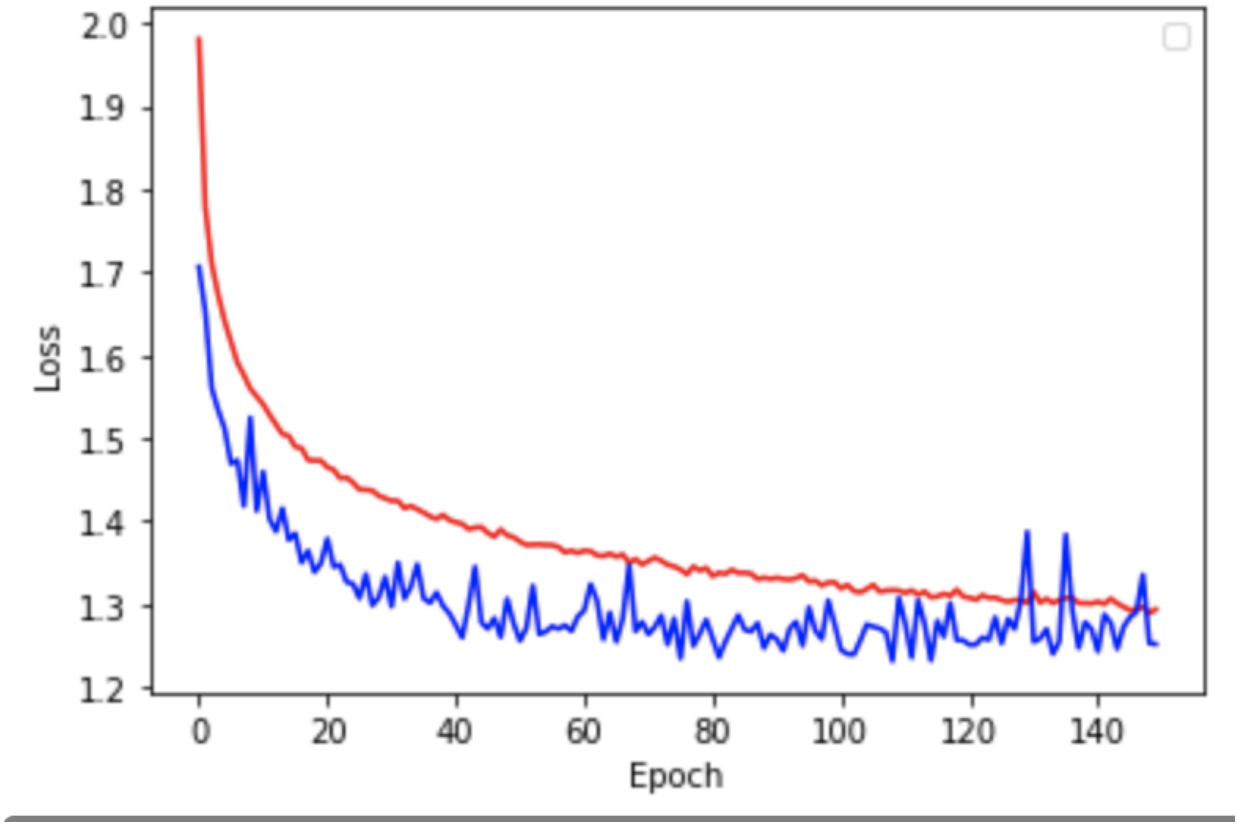
dropout 늘려보자

0.6 → 0.65

0.3 → 0.35

0.2 → 0.25

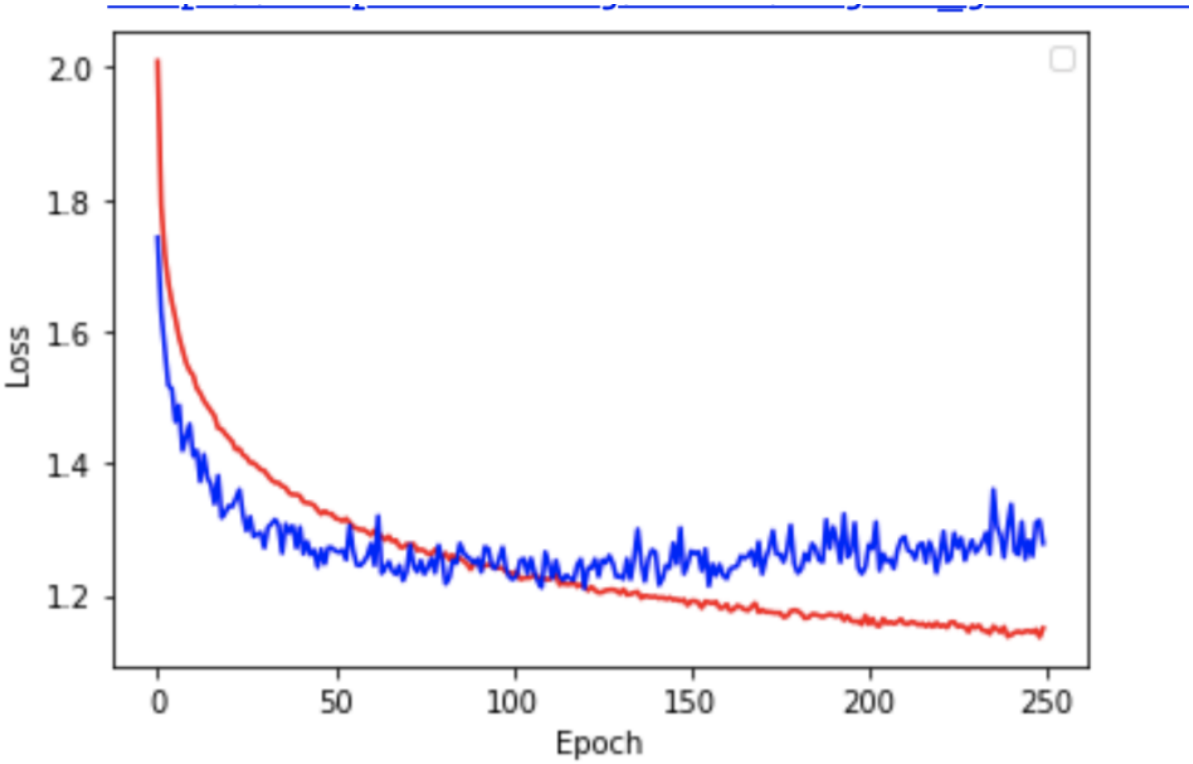
```
self.dropout = nn.Dropout(0.65)
self.dropout2 = nn.Dropout(0.35)
self.dropout3 = nn.Dropout(0.25)
self.activation = nn.ReLU()
```



Accuracy: 55.75%

Trial 17

lr 0.001 to 0.0007

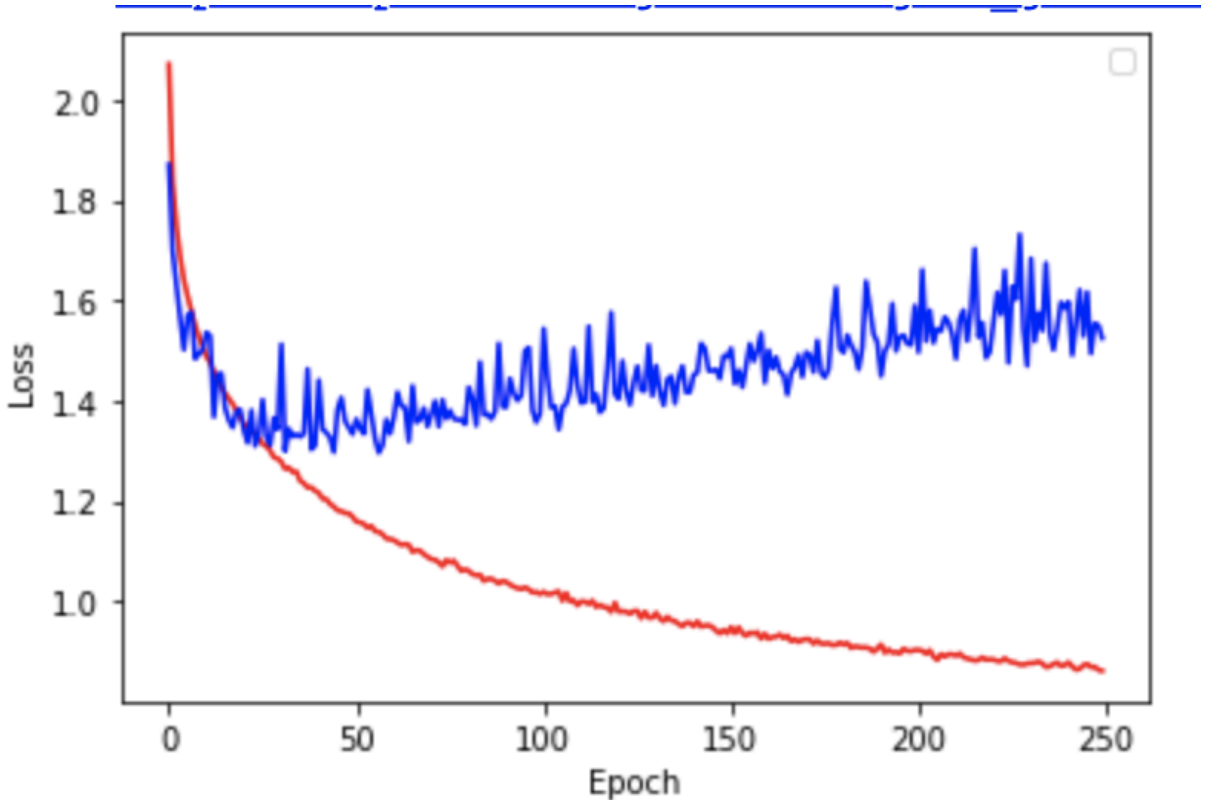


Accuracy: 54.97%

Trial 18

초기 레이어 뉴런수를 줄여보기로 함

```
def __init__(self):
    super(Classifier, self).__init__()
    self.linear1 = nn.Linear(32*32*3, 256)
    self.linear2 = nn.Linear(256, 512)
    self.linear3 = nn.Linear(512, 256)
    self.linear4 = nn.Linear(256, 64)
    self.linear5 = nn.Linear(64, 32)
    self.linear6 = nn.Linear(32, 10)

    self.bn1 = nn.BatchNorm1d(256)
    self.bn2 = nn.BatchNorm1d(512)
    self.bn3 = nn.BatchNorm1d(256)
    self.bn4 = nn.BatchNorm1d(64)
    self.bn5 = nn.BatchNorm1d(32)

    self.dropout1 = nn.Dropout(0.65)
    self.dropout2 = nn.Dropout(0.35)
    self.dropout3 = nn.Dropout(0.25)
    self.activation = nn.ReLU()
```



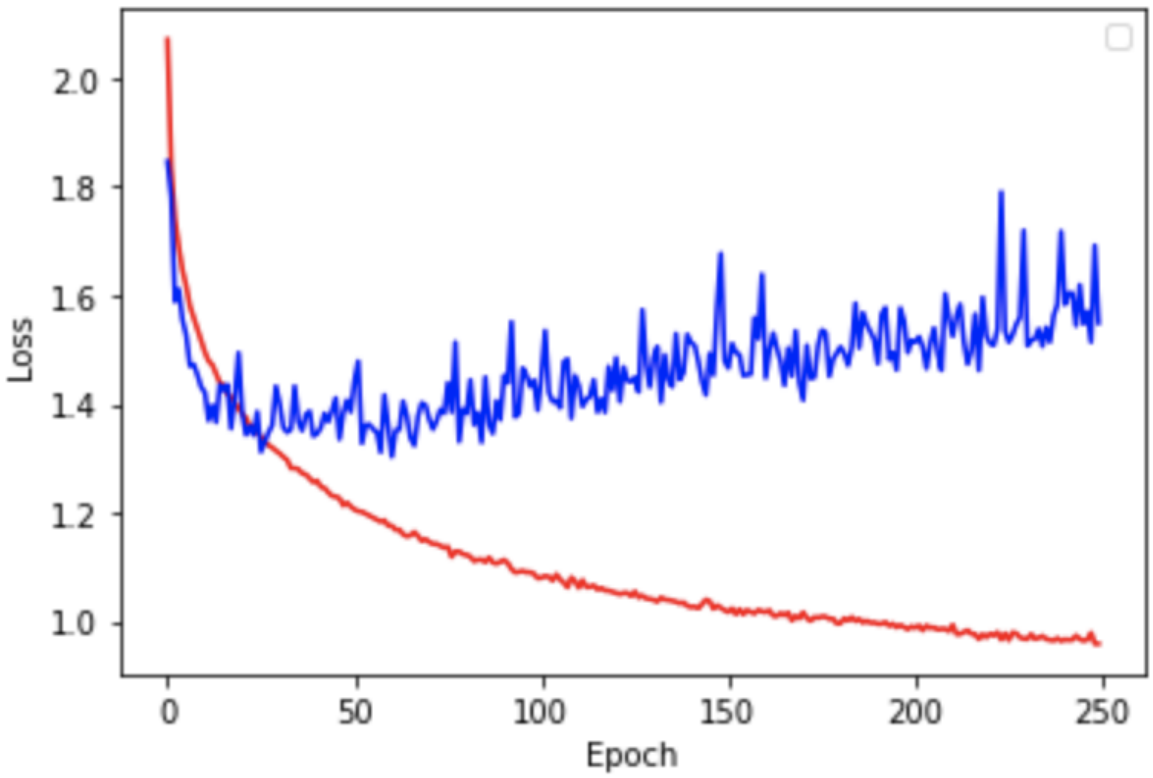
Accuracy: 51.23%

cost 값 자체는 잘 내려가는데, 분산값이 너무 높음.

람다를 더 많이 주기로 결정

Trial 19

lambda값을 0.005→ 0.01



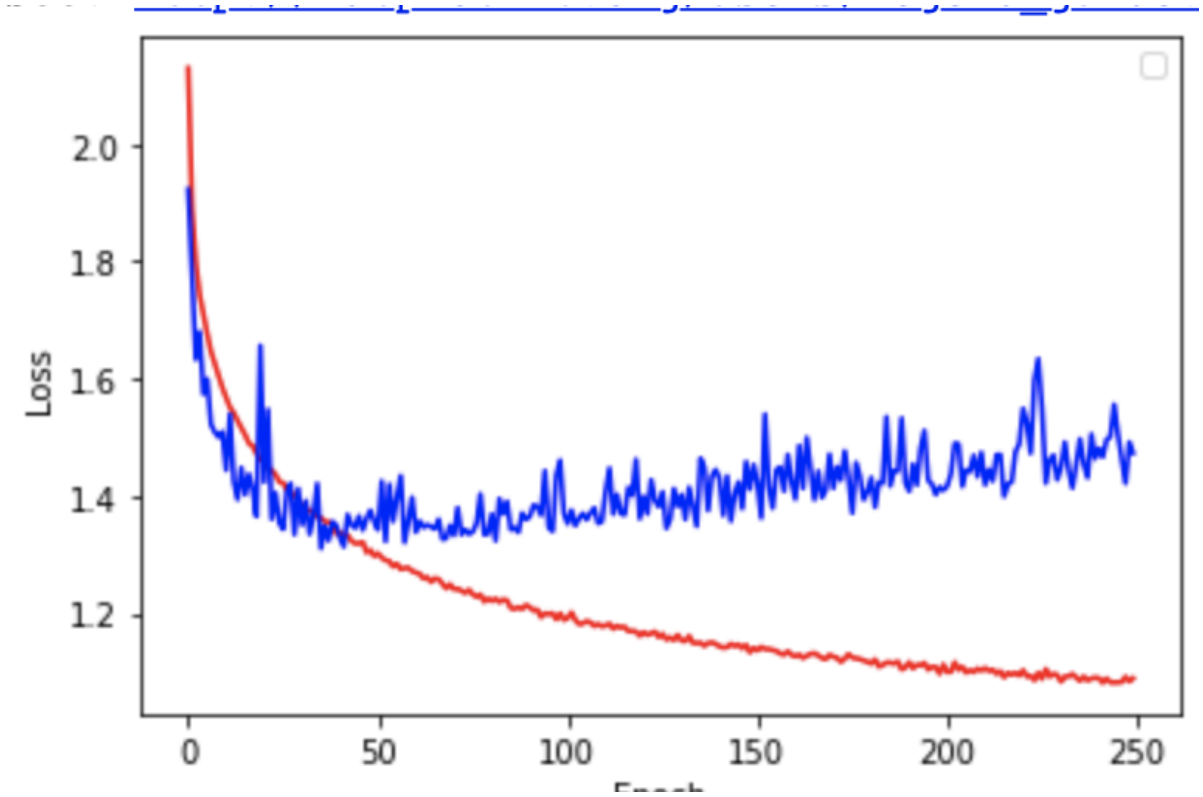
람다를 많이 줘도 정규화 효과가 미미해보임

Trial 20

```
self.dropout1 = nn.Dropout(0.65)
self.dropout2 = nn.Dropout(0.35)
self.dropout3 = nn.Dropout(0.25)
self.activation = nn.ReLU()
```

```
self.dropout1 = nn.Dropout(0.7)
self.dropout2 = nn.Dropout(0.4)
self.dropout3 = nn.Dropout(0.3)
self.activation = nn.ReLU()
```

드롭아웃 증가



Epoch: 250 / 250, cost: 1.089613914489746

학습률이 떨어지고, 여전히 분산값이 높음

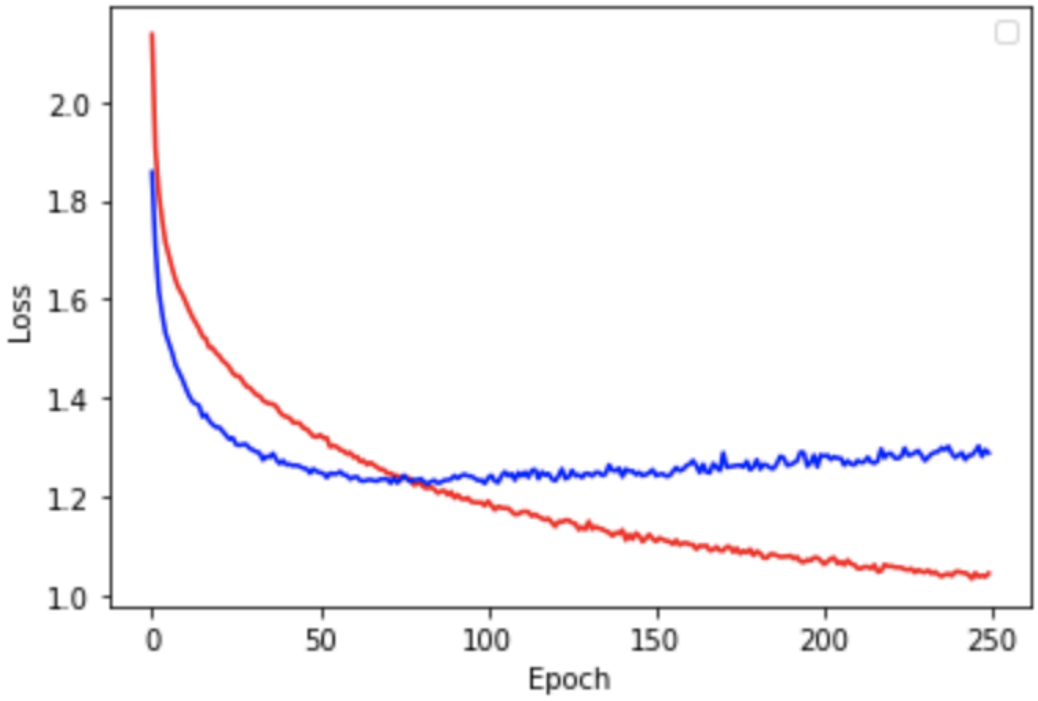
Trial 21

시작은 뉴런수를 많이 주고 중간 뉴런을 확 낮추어보았음.

```
def __init__(self):
    super(Classifier, self).__init__()
    self.linear1 = nn.Linear(32*32*3, 1024)
    self.linear2 = nn.Linear(1024, 512)
    self.linear3 = nn.Linear(512, 64)
    self.linear4 = nn.Linear(64, 64)
    self.linear5 = nn.Linear(64, 32)
    self.linear6 = nn.Linear(32, 10)

    self.bn1 = nn.BatchNorm1d(1024)
    self.bn2 = nn.BatchNorm1d(512)
    self.bn3 = nn.BatchNorm1d(64)
    self.bn4 = nn.BatchNorm1d(64)
    self.bn5 = nn.BatchNorm1d(32)

    # dropout 0.6

    self.dropout1 = nn.Dropout(0.6)
    self.dropout2 = nn.Dropout(0.4)
    self.dropout3 = nn.Dropout(0.25)
    self.activation = nn.ReLU()
```

지금까지 인풋 노멀라이제이션을 적용하지 않았다는 것을 깨닫고 적용

See: http://matplotlib.org/users/legend_guide.html#



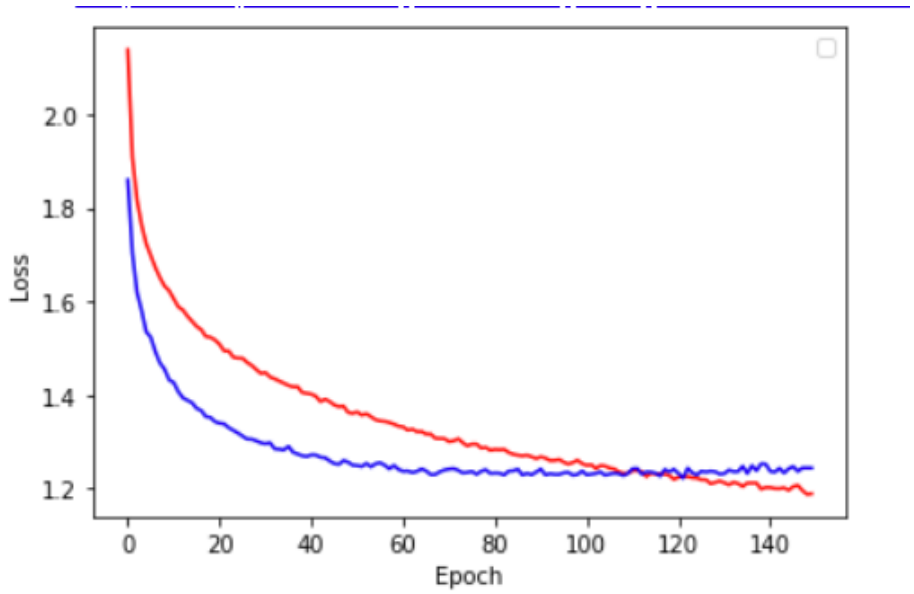
Accuracy: 57.86%

인풋 노멀라이제이션과, 중간뉴런 수를 줄이니 확실히 정확도가 잘 나오는 것을 확인.

Trial 22

분산값을 줄이기 위해 람다값 증가

lambda 0.003→0.005

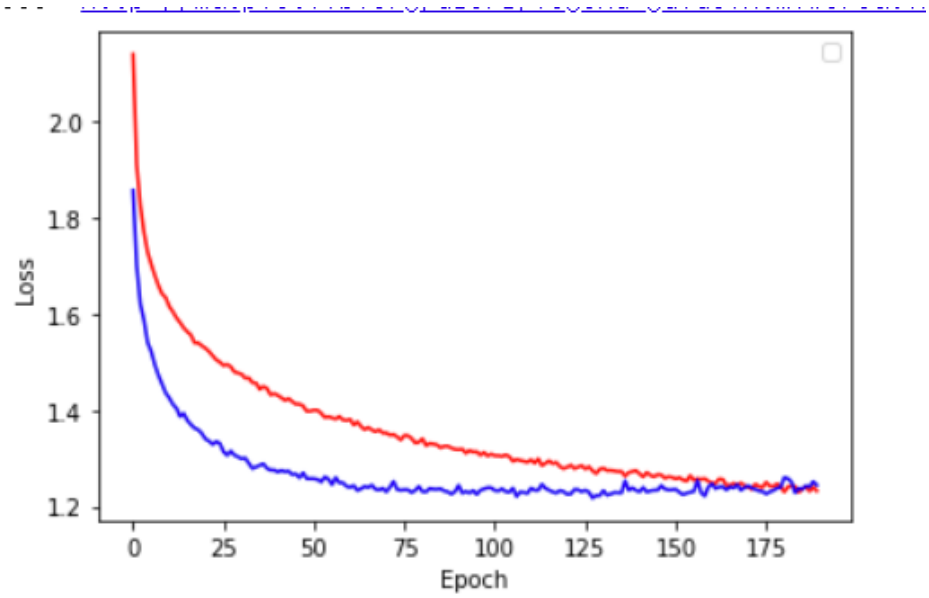


Accuracy: 58.0%

Cost값이 높아지지만 분산값이 확실히 줄어듦을 확인.

Trial 23

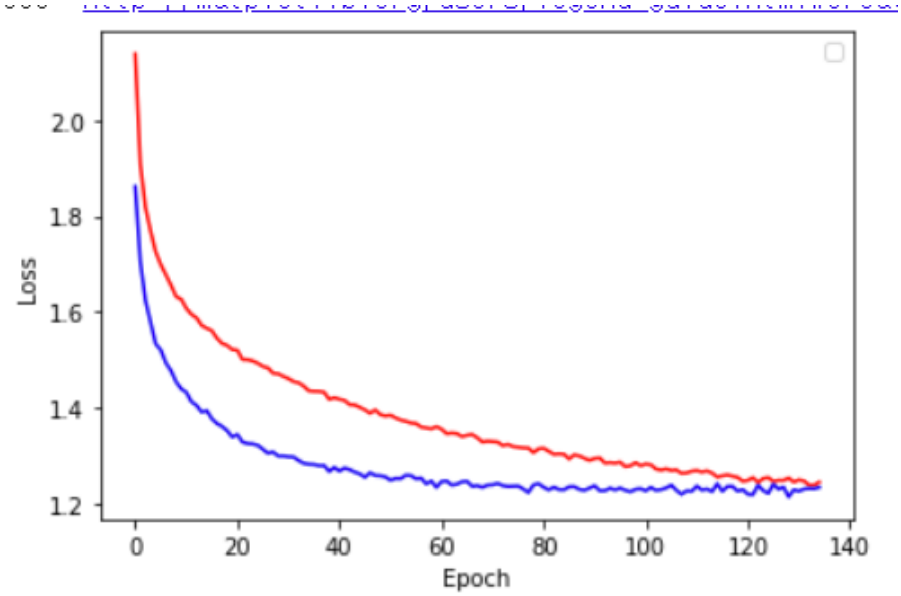
lambda 0.005 → 0.007



Accuracy: 57.73%

Trial 24

0.007 → lambda 0.006

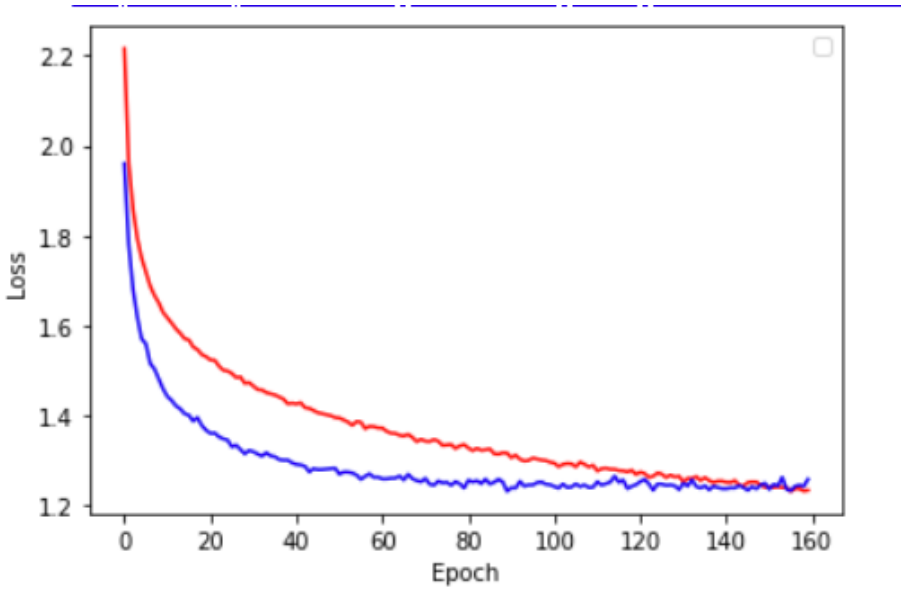


Accuracy: 57.53%

Trial 25

레이어 하나만 더 늘려보기로 함

```
super(Classifier, self).__init__()
self.linear1 = nn.Linear(32*32*3, 1024)
self.linear2 = nn.Linear(1024, 512)
self.linear3 = nn.Linear(512, 64)
self.linear4 = nn.Linear(64, 64)
self.linear5 = nn.Linear(64, 32)
self.linear6 = nn.Linear(32, 16)
self.linear7 = nn.Linear(16, 10)
```



Accuracy: 58.08%

코드 설명

```
torch.manual_seed(0)
torch.cuda.manual_seed(0)
torch.cuda.manual_seed_all(0)
```

동일한 조건에서 실험을 진행하기 위해 random seed값 설정

```
class Classifier(nn.Module):
    # 모델의 코드는 여기서 작성해주세요

    def __init__(self):
        super(Classifier, self).__init__()
        self.linear1 = nn.Linear(32*32*3, 1024)
        self.linear2 = nn.Linear(1024, 512)
        self.linear3 = nn.Linear(512, 64)
        self.linear4 = nn.Linear(64, 64)
        self.linear5 = nn.Linear(64, 32)
        self.linear6 = nn.Linear(32, 16)
        self.linear7 = nn.Linear(16, 10)

        self.bn1 = nn.BatchNorm1d(1024)
        self.bn2 = nn.BatchNorm1d(512)
        self.bn3 = nn.BatchNorm1d(64)
        self.bn4 = nn.BatchNorm1d(64)
```

```
        self.bn5 = nn.BatchNorm1d(32)
        self.bn6 = nn.BatchNorm1d(16)

        self.activation = nn.ReLU()
        self.dropout1 = nn.Dropout(0.6)
        self.dropout2 = nn.Dropout(0.4)
        self.dropout3 = nn.Dropout(0.25)

    def forward(self, x):
        z1 = self.linear1(x)
        z1 = self.bn1(z1)
        a1 = self.activation(z1)
        a1 = self.dropout1(a1)

        z2 = self.linear2(a1)
        z2 = self.bn2(z2)
        a2 = self.activation(z2)
        a2 = self.dropout1(a2)

        z3 = self.linear3(a2)
        z3 = self.bn3(z3)
        a3 = self.activation(z3)
        a3 = self.dropout2(a3)

        z4 = self.linear4(a3)
        z4 = self.bn4(z4)
        a4 = self.activation(z4)
        a4 = self.dropout2(a4)

        z5 = self.linear5(a4)
        z5 = self.bn5(z5)
        a5 = self.activation(z5)
        a5 = self.dropout3(a5)

        z6 = self.linear6(a5)
        z6 = self.bn6(z6)
        a6 = self.activation(z6)

        z7 = self.linear7(a6)

        return z7
```

인풋 레이어 포함 총 7개의 레이어를 사용하는 모델, batch normalization 을 적용시켰으며, activation function으로는 ReLU를 채택하였음.

dropout을 3개로 나누어 레이어별로 차등적으로 dropout 시켰음.

레이어6 에서는 뉴런수가 적어 dropout을 적용시키지 않았음.

```
if __name__ == "__main__":
    # 학습코드는 모두 여기서 작성해주세요
    if torch.cuda.is_available():
      device = torch.device('cuda')
    else:
```

```
      device = torch.device('cpu')

    transform = transforms.Compose(
      [transforms.ToTensor(),
       transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))])

    train_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                                                 train=True,
                                                 transform=transform,
                                                 download=True)
    test_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                                                train=False,
                                                transform=transform,
                                                download=True)
    batch_size = 256

    train_dataloader = torch.utils.data.DataLoader(train_dataset, batch_size=batch_size, shuffle=True)
    test_dataloader = torch.utils.data.DataLoader(test_dataset,  batch_size=batch_size)

    model = Classifier()
    model = model.to(device).train()

    optimizer = optim.Adam(params=model.parameters(), lr = 0.0007, betas=(0.9, 0.999)) #  set optimizer
    criterion = nn.CrossEntropyLoss()
```

조교님께서 수업시간에 제시해주신 최적의 input normalization 값을 활용하여 normalize하였음.

batch size를 256으로 하여 학습이 더 빠르게 진행될 수 있도록 함

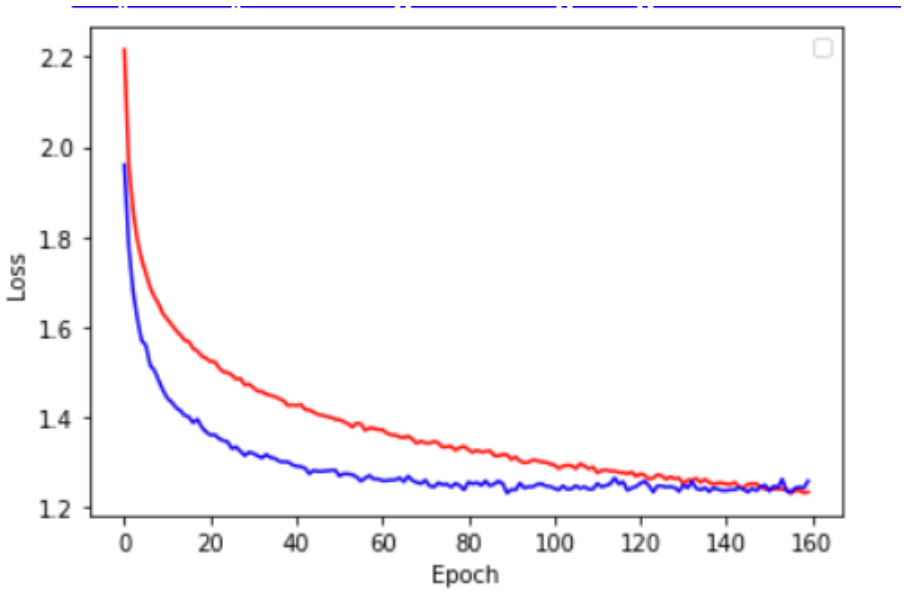
optimizer로는 Adam을 사용, 학습률 0.0007, betas는 교수님께서 수업시간에 제시해주신 최적의 값으로 세팅.

Cost function으로는 CrossEntropy를 사용

```
# continued from __main__
    epochs = 160
    lmbd = 0.006

    train_avg_costs = []
    test_avg_costs = []

    test_total_batch = len(test_dataloader)
    total_batch_num = len(train_dataloader)
```

분산값이 가장 적어지는 지점을 기준으로 하기 위해 epoch를 160으로 세팅

정규화 람다 값은 실험을 통해서 0.006이 가장 잘 맞음을 확인하였음.




```
# continued from __main__
    for epoch in range(epochs):
      avg_cost = 0
      model.train()

      for b_x, b_y in train_dataloader:
        b_x = b_x.view(-1, 32*32*3).to(device) # 1 step 마다 batch size만큼의 데이터 사용
        logits = model.forward(b_x)
        loss = criterion(logits, b_y.to(device))

        reg = model.linear1.weight.pow(2.0).sum()
        reg += model.linear2.weight.pow(2.0).sum()
        reg += model.linear3.weight.pow(2.0).sum()

        loss += ((lmbd * reg) / (len(b_x) * 2))

        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        avg_cost += loss / total_batch_num
      train_avg_costs.append(avg_cost.detach().cpu().numpy())
      print('Epoch: {} / {}, cost: {}'.format(epoch + 1, epochs, avg_cost))

      test_avg_cost = 0
      model.eval()
      for b_x, b_y in test_dataloader:
        b_x = b_x.view(-1, 32*32*3).to(device)
        with torch.no_grad():
          logits = model(b_x)
          test_loss = criterion(logits, b_y.to(device))
        test_avg_cost += test_loss / test_total_batch
      test_avg_costs.append(test_avg_cost.detach().cpu().numpy())

    torch.save(model.state_dict(), 'model.pt')  # 학습된 모델을 저장하는 코드입니다.
```

b_x를 torch.Size([batch_size, 32*32*3])로 view를 변경. 256은 batch size만큼의 크기.

forward로 나온 output값을 nn.CrossEntropyLoss를 활용해 label값과 비교.

원래 로직이라면 output을 softmax 취해주고 오차값 구해주어야 함. 하지만 pytorch에서는 CrossEntropyLoss 함수 내에서 softmax를 취해주기 때문에 안해주어도 무방함.

Frobenius norm을 활용하여 각 레이어 별 weight들의 제곱의 합을 구하고, 각 제곱의 합을 더해줌.

모든 레이어별 제곱의 합을

$$+\frac{\lambda}{2m}\sum_{l=1}^{L}\left\|W^{[l]}\right\|_F^2$$

이 식과 같이 람다를 곱해주고 2*batch_size로 나눠줌.

그리고 back propagation을 하며 각 parameter값을 업데이트.

그래프를 그리기 위해 avg_cost값을 계산해서 cache해둠.

한 train epoch가 끝날 때 마다, test(dev) dataset을 활용하여 avg_cost을 계산해줌.

이는 train avg_cost와 test avg_cost를 비교하여 나중에 그래프를 그릴 때 활용됨.

```
import matplotlib.pyplot as plt
import numpy as np

epoch = range(epochs)
line1 = plt.plot(epoch, train_avg_costs, 'r-')
line2 = plt.plot(epoch, test_avg_costs, 'b-')
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend([line1, line2], ['train_avg_cost', 'test_avg_cost'])
plt.show()
```

그래프를 그리는 코드

```
# 학습된 모델의 성능을 평가하는 코드입니다.
# 아래의 코드로 평가를 진행할 예정이므로 아래의 코드가 정상 동작 해야하며, 제출전 모델의 성능을 확인하시면 됩니다.

device = torch.device("cuda:0" if torch.cuda.is_available() else "cpu")
```

```
transform = transforms.Compose(
      [transforms.ToTensor(),
       transforms.Normalize((0.4914, 0.4822, 0.4465), (0.247, 0.243, 0.261))])

test_dataset = torchvision.datasets.CIFAR10(root="CIFAR10/",
                                            train=False,
                                            transform=transform,
                                            download=True)


test_dataloader = torch.utils.data.DataLoader(test_dataset, batch_size=10000)

classifier = Classifier().to(device)
classifier.load_state_dict(torch.load('model.pt'))
classifier.eval()


for data, label in test_dataloader:
    data = data.view(-1, 32 * 32 * 3).to(device)

    with torch.no_grad():
        logits = classifier(data)

        pred = torch.argmax(logits, dim=1)

        total = len(label)
        correct = torch.eq(pred, label.to(device)).sum()

        print("Accuracy on test set : {:.4f}%".format(100 * correct / total))
```

test dataset을 새로 load하기 때문에 input normalization을 다시 적용시켜주었음.