<CIFAR10 인식 정확도 챌린지>

중간 대체 과제

과목 : 심층학습

교수님 : 양희경 교수님

학과 : 휴먼지능정보공학과

학번 : 201810800

이름 : 이혜인

제출일 : 2020.05.10

목차

- 참고사항

1. 최종 정확도

2. Test 결과 일지

3. 최종 선택한 방법

4. 최종 코드

5. 결론 및 느낀점

● 참고사항

네트워크 상 문제로 인하여 해당 모델 파일을 Colab 에서 다운받았습니다. 따라서 해당 모델(.pkl)파일이 Colab 에서는 실행이 되지만, AWS 상에서는 CPU 로 변경시켜주어야 합니다. 참고 부탁드립니다!!

+만약 AWS 에서 실행시키고 싶으시다면,

```
model = torch.load('./my_model.pkl', map_location='cpu')
model
```

이 코드를 이용하면 실행이 됩니다.

1. 최종 정확도

최종 정확도- Accuracy of Test Data: 77.5040054321289

```
model.eval()
ComputeAccr(test_loader, model)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:6: UserWarning: vo
latile was removed and now has no effect. Use `with torch.no_grad():` instead.
```

```
Accuracy of Test Data: 77.5040054321289
```

➔ 같은 코드를 test 했을 때, 가장 높게 나온 정확도는 77.50400543212289이다. 이는 test할 때마다 약간의 오차가 존재하였다.

2. Test 결과 일지

본 test는 '[04실습] 심층 신경망 훈련'을 커스터마이징(Customizing)하여 '[04] 심층 신경망 훈련'에서 배운 다양한 훈련 방법을 조합하여 가능한 만큼 test 세트에 대한 정확도를 높이는 방식으로 진행되었다.

진행 방식은 다음과 같다.

1. Activation Function, Loss Function, Optimizer 등 여러 함수의 조합을 통해 최적의 경우를 찾는다.
2. batch size, learning rate, epoch 수, step size, gamma 등 여러 parameter들을 수정해가면서 최적의 경우를 찾는다.
3. 실험 중반부 이후부터는 epoch에 따른 Accuracy(정확도)값을 그래프로 출력한다.

➔ 다음 실험은 약 100차례에 가까운 실험을 통해 결과를 얻을 수 있었다.

가장 먼저 '[04실습] 심층 신경망 훈련'에서 모든 코드를 가져온 다음, batch_size는 32, learning_rate는 0.002, epoch는 20으로 고정시켜서 실험을 진행하였다.

1. dropout, Weight initialization, Batch normalization, SGD optimizer, ReLU Activation Function, learning rate decay를 선택하였다.

   ➔ 정확도는 약 66.796를 가진다.

2. optimizer를 Adam으로 변경하였다.

   ➔ 정확도는 약 71.033를 가진다.

   ➔ 이 이후부터는 Adam optimizer를 사용하여 실험을 진행하였다.

3. data augmentation 추가하였다.

   ➔ 정확도는 약 27.754을 가진다.

➔ 따라서 data augmentation은 이 이후로는 사용하지 않았다.

4. Weight initialization을 제거하였다.

➔ 정확도는 약 74.198와 약75.120의 값을 가진다.

➔ 따라서 이후부터 Weight initialization을 사용하지 않고 실험을 진행하였다.

다음으로는 Activation Function만을 변경시키면서 실험을 진행하였다. 위의 실험은 모두 ReLU를 사용한 경우이다.

1. LeakyReLU

➔ 정확도는 약 74.569를 가진다.

➔ ReLU와 큰 차이를 보이지는 않았다.

2. PReLU

➔ 총 3차례의 test를 한 결과 약 75.941, 76.312, 75.851의 정확도를 가진다.

➔ 이전 Activation Function보다 높은 정확도를 나타내었다.

3. RReLU

➔ 총 3차례의 test를 한 결과 약 8.964, 74.599, 75.050의 정확도를 가진다.

➔ 꽤 큰 오차가 발생하였다.

다음은 Optimizer만 변경시키며 실험을 진행하였다. 이 때, Activation Function은 가장 높은 정확도를 보여준 PReLU를 사용하였다. 이 전 실험은 모두 Adam으로 진행한 결과이다.

1. Adagrad

➔ 약 77.544의 정확도를 가진다.

2. RMSprop

➔ 약 75.941의 정확도를 가진다.

➔ Adagrad가 가장 높은 정확도를 나타난다는 것을 알 수 있다.

하지만 여기까지의 실험이 모두 이전 값들이 적재되어 나타난 경우라는 것을 알게 되었고, 적재 없이 순수 값 만을 검출하기 위해 실험을 다시 진행하였다. 다시 진행한 실험의 결과는 대부분 2~3정도의 오차를 보였고, 이 가운데 일부 실험만 확인해 보려고 한다.

가장 먼저 Activation Function을 ELU로 설정하고, optimizer를 변경시켰다.

1. SGD

   ➔ 약 70.943의 정확도를 가진다.

2. Adam

   ➔ 약 73.597의 정확도를 가진다.

ReLU를 Activation Function으로 설정

1. Adam

   ➔ 약 75.190의 정확도를 가진다.

PReLU가 Activation Function인 경우에는

1. Adam

   ➔ 약 75.961의 정확도를 가진다.

   ➔ 이와 같은 실험을 통해 최종적인 Function을 Activation Function은 PReLU로 정하였고, Optimizer는 Adam Optimizer로 정하였다.

다음으로 lr_scheduler를 변경하면서 실험을 진행하였다. 이전 실험의 lr_scheduler는 StepLR을 사용하였고, 가장 높은 정확도는 약 77.043의 정확도를 가졌다.

1. MultiStepLR

   ➔ schedule = lr_scheduler.MultiStepLR(optimizer, milestones=[15000, 175

000], gamma=-0.2) 와 같이 설정하였다.

➔ 약 76.963의 정확도를 가진다.

2. ExponentialLR

➔ schedule = lr_scheduler.ExponentialLR(optimizer=optimizer,gamma=0.2)와 같이 설정하였다.

➔ 76.692의 정확도를 가진다.

➔ 이를 통해 최종 lr_scheduler로 StepLR을 사용하기로 결정하였다.

다음으로는 parameter들을 변경하며 실험을 진행하였다. 가장 먼저 batch_size를 수정하면서 진행하였다.

1. batch_size가 32일 때

➔ 가장 높은 정확도는 약 76.372였다.

2. batch_size가 64일 때

➔ 약 75.620의 정확도를 가진다.

3. Batch_size가 128일때

➔ 약 75.661의 정확도를 가진다.

➔ 이를 통해 batch_size는 32로 고정하였다.

다음으로 Dropout 값에 따른 최대값을 찾아보았다.

1. Dropout이 0.3일 때

➔ 약 76.051의 정확도를 가진다.

2. Dropout이 0.35일 때

➔ 약 76.682의 정확도를 가진다.

3. Dropout이 0.4일 때

   ➔ 약 75.681의 정확도를 가진다.

4. Dropout이 0.45일 때

   ➔ 약 74,709의 정확도를 가진다.

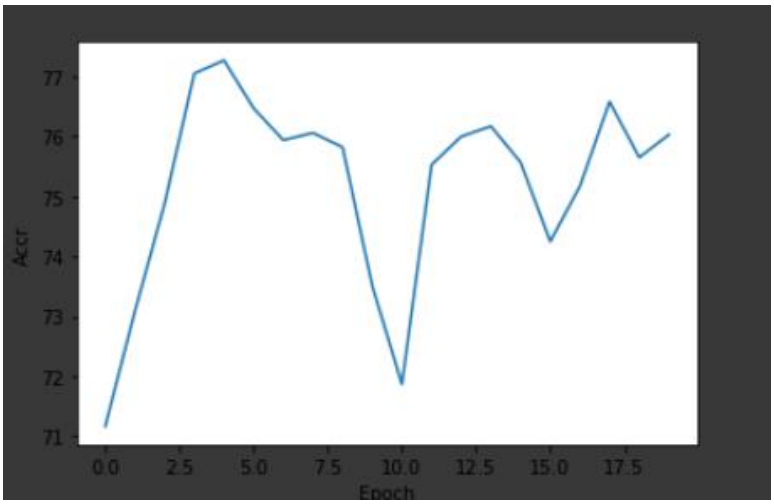
   ➔ 최종적으로 Dropout은 0.35로 결정하였다.

다음으로 Step_size 값에 따른 최대값을 찾아보았다.

1. Step_size가 10일 때

   ➔ 약 76.041의 정확도를 가진다.

2. Step_size가 20일 때



   ➔ 다음과 같이 Epoch에 따른 정확도를 그래프를 통해 확인할 수 있다.

   ➔ 약 77.193의 정확도를 가진다.

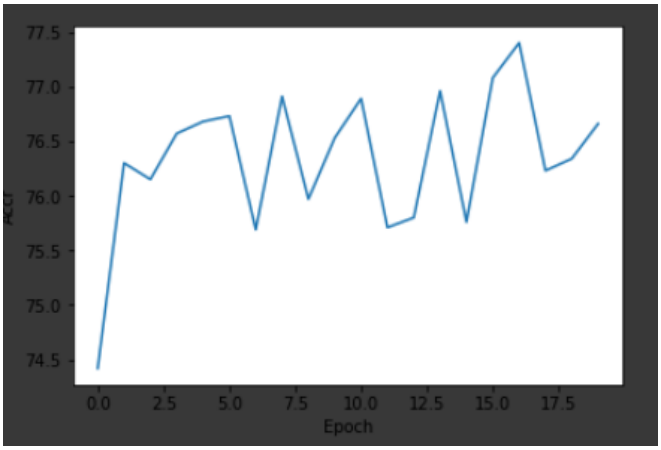
3. Step_size가 30일 때

   ➔ 약 77.083의 정확도를 가진다.

4. Step_size가 40일 때

   ➔ 약 76.762의 정확도를 가진다.

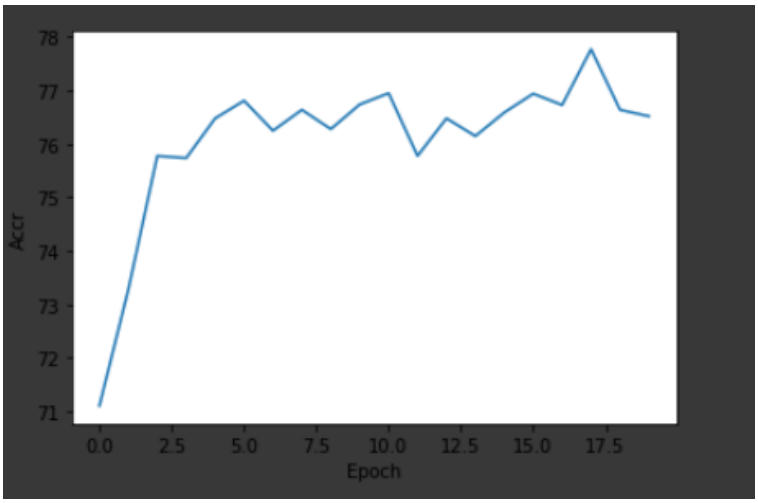
5. Step_size가 50일 때



➔ 다음과 같이 Epoch에 따른 정확도를 그래프를 통해 확인할 수 있다.

➔ 약 76.221의 정확도를 가진다.

6. Step_size가 60일 때



➔ 다음과 같이 Epoch에 따른 정확도를 그래프를 통해 확인할 수 있다.

➔ 약 76.512의 정확도를 가진다.

7. Step_size가 70일 때

➔ 약 75.851의 정확도를 가진다.

8. Step_size가 80일 때

➔ 약 76.131의 정확도를 가진다.

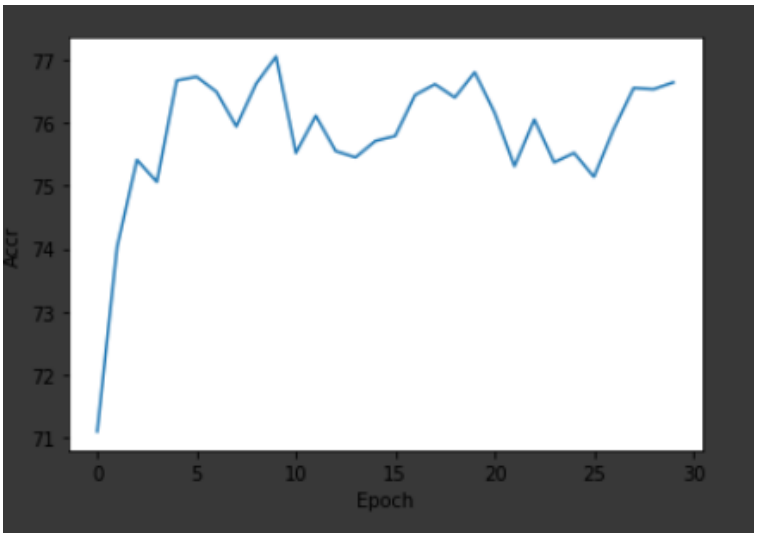
9. Step_size가 100일 때

➔ 약 74.779의 정확도를 가진다.

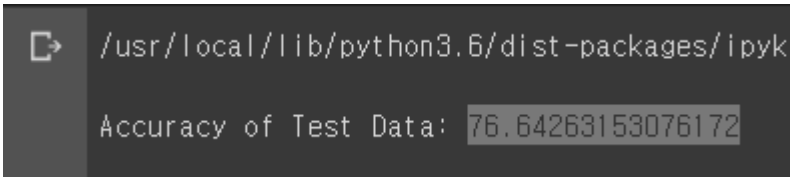
➔ 따라서 가장 높은 경우는 Step_size가 20일 때였다. 따라서 Step_size는 20으로 최종 결정되었다.

다음으로는 epoch를 30으로 정한 후, gamma를 정하였다. 이 때 epoch는 100을 실험해본 결과와 30을 실험해본 결과의 차이가 많이 나지 않아 epoch를 30으로 정하였다.

1. gamma가 0.1일 때



➔ 다음과 같이 Epoch에 따른 정확도를 그래프를 통해 확인할 수 있다.

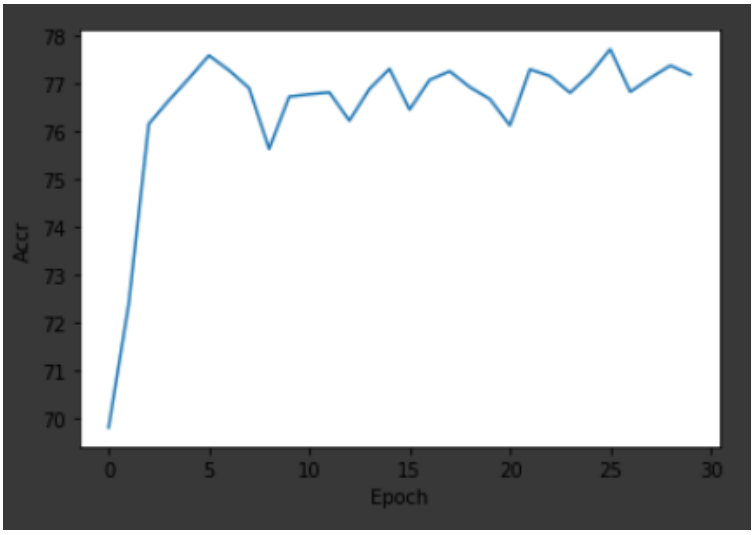


➔ 약 76.642의 정확도를 가진다.

2. gamma가 0.15일 때

➔ 약 76.522의 정확도를 가진다.

3. gamma가 0.2일 때



➔ 다음과 같이 Epoch에 따른 정확도를 그래프를 통해 확인할 수 있다.

➔ 약 77.193의 정확도를 가진다.

➔ 따라서 Gamma는 최종적으로 0.2로 정하게 되었다.

```
1 model.eval()
2 ComputeAccr(test_loader, model)
```

```
/usr/local/lib/python3.6/dist-packages/ipyker

Accuracy of Test Data: 77.19351196289062
```

3. 최종 선택한 방법

최종적으로 선택한 방법은 다음과 같다.

Batch_size : 32

Learning_rate : 0.002

Num_epoch = 30

Dropout : 0.35

Activation Function : PReLU

Loss Function : CrossEntropyLoss

Optimizer : Adam optimizer

Scheduler : StepLR

➔ Step_size : 20, gamma : 0.2

4. 최종 코드

코드는 다음과 같다.(정확도 그래프를 추가한 코드)

```
[245]  1 import numpy as np
       2 import torch
       3 import torch.nn as nn
       4 import torch.optim as optim
       5 import torch.nn.init as init
       6 import torchvision.datasets as dset
       7 import torchvision.transforms as transforms
       8 from torch.utils.data import DataLoader
       9 from torch.autograd import Variable
      10 import matplotlib.pyplot as plt
      11
      12 # (8) learning rate decay
      13 from torch.optim import lr_scheduler
      14
      15 batch_size=32
      16 learning_rate=0.002 #0.01
      17 num_epoch=30
```

```
[246]  1 cifar_train=dset.CIFAR10("CIFAR10/", train=True, transform=transforms.ToTensor(), target_transform=None, download=True)
       2 cifar_test=dset.CIFAR10("CIFAR10/", train=False, transform=transforms.ToTensor(), target_transform=None, download=True)

   Files already downloaded and verified
   Files already downloaded and verified
```

```
[248]  1 def ComputeAccr(dloader, imodel):
       2     correct = 0
       3     total = 0
       4
       5     for j, [imgs, labels] in enumerate(dloader): # batch_size 만큼
       6         img = Variable(imgs, volatile = True).cuda() # x
       7         #label = Variable(labels) # y
       8         label = Variable(labels).cuda()
       9         # .cuda() : GPU에 로드되기 위함. 만약 CPU로 설정되어 있다면 에러남
      10
      11         output = imodel.forward(img) #forward prop.
      12         _, output_index = torch.max(output, 1)
      13
      14         total += label.size(0)
      15         correct += (output_index == label).sum().float()
      16     print ("Accuracy of Test Data: {}".format(100*correct/total))
```

```
[249]  1 def ComputeAccr2(dloader, imodel):
       2     correct = 0
       3     total = 0
       4
       5     for j, [imgs, labels] in enumerate(dloader): # batch_size 만큼
       6         img = Variable(imgs, volatile = True).cuda() # x
       7         #label = Variable(labels) # y
       8         label = Variable(labels).cuda()
       9         # .cuda() : GPU에 로드되기 위함. 만약 CPU로 설정되어 있다면 에러남
      10
      11         output = imodel.forward(img) #forward prop.
      12         _, output_index = torch.max(output, 1)
      13
      14         total += label.size(0)
      15         correct += (output_index == label).sum().float()
      16     return format(100*correct/total)
```

➔ ComputeAccr2 는 그래프를 추가하기 위해 추가한 함수이다.

```
[250]  1 # === 3. 데이터 로드함수 ===
       2 train_loader = torch.utils.data.DataLoader(list(cifar_train)[:], batch_size=batch_size, shuffle=True, num_workers=2, drop_last=True)
       3 test_loader = torch.utils.data.DataLoader(cifar_test, batch_size=batch_size, shuffle=False, num_workers=2, drop_last=True)
       4
       5 # === 4. 모델 선언 ===
       6 class CNN(nn.Module):
       7     def __init__(self):
       8         super(CNN,self).__init__()
       9         self.layer=nn.Sequential(
      10             nn.Conv2d(3,16,3,padding=1),#
      11             #nn.ReLU(),
      12             #nn.LeakyReLU(),
      13             nn.PReLU(),
      14             #nn.ReLU6(),
      15             #nn.RReLU(),
      16             #nn.ELU(),
      17             nn.Dropout2d(0.35),   # (2) drop out
      18             nn.BatchNorm2d(16),  # (6) Batch normalization
      19             nn.Conv2d(16,32,3,padding=1),#
      20             #nn.ReLU(),
      21             #nn.LeakyReLU(),
      22             nn.PReLU(),
      23             #nn.ReLU6(),
      24             #nn.RReLU(),
      25             #nn.ELU(),
      26             nn.Dropout2d(0.35),
      27             nn.BatchNorm2d(32),
      28             nn.MaxPool2d(2,2),#
      29             nn.Conv2d(32,64,3,padding=1),#
      30             #nn.ReLU(),
      31             #nn.LeakyReLU(),
      32             nn.PReLU(),
      33             #nn.ReLU6(),
      34             #nn.RReLU(),
      35             #nn.ELU(),
      36             nn.Dropout2d(0.35),
      37             nn.BatchNorm2d(64),
      38             nn.MaxPool2d(2,2)#
      38             nn.MaxPool2d(2,2)#
      39         )
      40         self.fc_layer=nn.Sequential(
      41             nn.Linear(64*8*8, 100),
      42             #nn.ReLU(),
      43             #nn.LeakyReLU(),
      44             nn.PReLU(),
      45             #nn.ReLU6(),
      46             #nn.RReLU(),
      47             #nn.ELU(),
      48             nn.Dropout2d(0.35),
      49             nn.BatchNorm1d(100),
      50             nn.Linear(100,10)
      51         )
      52         #(3) weight initialization
      53         # for m in self.modules():
      54         #     if isinstance(m, nn.Conv2d):
      55         #         init.kaiming_normal(m.weight.data) #RELU 일 때
      56         #         m.bias.data.fill_(0)
      57         #     if isinstance(m, nn.Linear):
      58         #         init.kaiming_normal(m.weight.data)
      59         #         m.bias.data.fill_(0)
      60
      61     def forward(self,x):
      62         out=self.layer(x)
      63         out=out.view(batch_size, -1)
      64         out=self.fc_layer(out)
      65
      66         return out
      67 model=CNN().cuda()
```

```
[251]  1 # === 5. loss, optimizer ===
       2 loss_func=nn.CrossEntropyLoss()
       3 #optimizer=torch.optim.SGD(model.parameters(), lr=learning_rate)
       4 optimizer=torch.optim.Adam(model.parameters(), lr=learning_rate)
       5 #(6) Adam optimizer
       6 #optimizer=torch.optim.Rprop(model.parameters(), lr=learning_rate)
       7 #optimizer=torch.optim.Adagrad(model.parameters(), lr=learning_rate)
       8 #optimizer=torch.optim.Adamax(model.parameters(), lr=learning_rate)
       9 #optimizer=torch.optim.ASGD(model.parameters(), lr=learning_rate)
      10 schedule = lr_scheduler.StepLR(optimizer, step_size=20, gamma=0.2)
      11 # (7)learning rate
      12 #schedule = lr_scheduler.MultiStepLR(optimizer, milestones=[15000, 175000], gamma=-0.2)
      13 #schedule = lr_scheduler.ExponentialLR(optimizer=optimizer,gamma=0.2)
      14 accr=[]
      15 epoch=[]
      16 model.train()
      17 # === 6. 학습 ===
      18 for i in range(num_epoch):
      19     for j, [image, label] in enumerate(train_loader):
      20         x=Variable(image).cuda()
      21         y_=Variable(label).cuda()
      22 
      23         optimizer.zero_grad()
      24         output=model.forward(x)
      25         loss=loss_func(output, y_)
      26         loss.backward()
      27         optimizer.step()
      28 
      29         if j%10000==0:
      30             print(j, loss)
      31     model.eval()
      32     accr.append(float(ComputeAccr2(test_loader,model)))
      33     model.train()
      34     epoch.append(i)
```

```
0 tensor(2.4603, device='cuda:0', grad_fn=<NllLossBackward>)
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:6: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.

0 tensor(0.7030, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.3495, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.3652, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.3404, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.3372, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.2311, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.1463, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0396, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.1644, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0984, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0084, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0248, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.1026, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0414, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0181, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0400, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0240, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0053, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0461, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0057, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0106, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0696, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0038, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0044, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0189, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0368, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0317, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.0010, device='cuda:0', grad_fn=<NllLossBackward>)
0 tensor(0.1210, device='cuda:0', grad_fn=<NllLossBackward>)
```

➔ 다음은 학습이 진행되는 상황을 나타낸 그림이다.

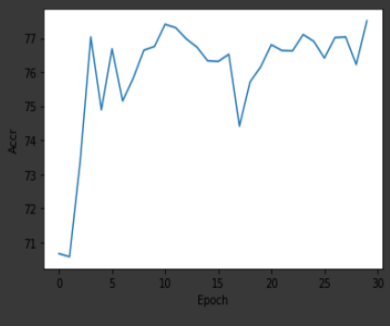
```
1 model.eval()
2 ComputeAccr(test_loader, model)
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:6: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.

Accuracy of Test Data: 77.5040054321289
```

```
[253] 1 model.eval()
      2 ComputeAccr2(test_loader, model)
      3 plt.plot(epoch,accr)
      4 plt.xlabel("Epoch")
      5 plt.ylabel("Accr")
      6 plt.show()
```

```
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:6: UserWarning: volatile was removed and now has no effect. Use `with torch.no_grad():` instead.
```



➔ 이를 통해 최종 정확도는 약 77.504 가 나온 것을 확인할 수 있다.

➔ 다만 이는 test 를 반복하면서 약간의 오차가 존재할 수 있다.

➔ 또한, 다음 그래프를 통해 Epoch 30 정도에서 최대값을 갖는다는 것을 확인할 수 있다.

```
1 model = torch.load('./my_model.pkl')
2 model
```

```
CNN(
  (layer): Sequential(
    (0): Conv2d(3, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): PReLU(num_parameters=1)
    (2): Dropout2d(p=0, inplace=35)
    (3): BatchNorm2d(16, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Conv2d(16, 32, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (5): PReLU(num_parameters=1)
    (6): Dropout2d(p=0, inplace=35)
    (7): BatchNorm2d(32, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(32, 64, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): PReLU(num_parameters=1)
    (11): Dropout2d(p=0, inplace=35)
    (12): BatchNorm2d(64, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (13): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (fc_layer): Sequential(
    (0): Linear(in_features=4096, out_features=100, bias=True)
    (1): PReLU(num_parameters=1)
    (2): Dropout2d(p=0, inplace=35)
    (3): BatchNorm1d(100, eps=1e-05, momentum=0.1, affine=True, track_running_stats=True)
    (4): Linear(in_features=100, out_features=10, bias=True)
  )
)
```

➔ 또한, 다음 사진은 모델(.pkl) 파일을 다운로드 한 후, Colab 에 다시 업로드를 시켜서 실행시킨 결과이다. 이를 통해 해당 모델이 어떻게 설계되었는지 알 수 있다.

```
In [2]: model = torch.load('./my_model.pkl', map_location='cpu')
        model

Out[2]: OrderedDict([('layer.0.weight', tensor([[[[-0.0529, -0.0208, -0.1419],
                                                   [-0.3604,  0.1421,  0.8869],
                                                   [-0.1462,  0.0600,  0.3303]],

                                                  [[-0.3633, -0.0394,  0.2400],
                                                   [-0.5646,  0.2772,  0.8743],
                                                   [-0.3665,  0.0879,  0.5940]],

                                                  [[-0.2081, -0.1096,  0.2484],
                                                   [-0.7351, -0.2884,  0.7863],
                                                   [-0.5053, -0.4730,  0.2867]]],


                                                 [[[-0.4070, -0.2016, -0.6631],
                                                   [ 0.2603,  0.3365, -0.4306],
                                                   [ 0.9128,  0.4593, -0.3408]],

                                                  [[-0.3918,  0.4106,  0.1415],
                                                   [ 0.0031,  0.3485, -0.0875],
                                                   [ 0.3023,  0.0709, -0.5889]],

                                                  [[-0.4537,  0.6481,  0.8651],
                                                   [-0.4373,  0.3599,  0.0505],
                                                   [-0.2280, -0.5029, -0.4547]]],


                                                 [[[ 0.0945, -0.1221, -0.1856],
                                                   [-0.1142, -0.5638, -0.2144],
                                                   [ 0.0592, -0.6711, -0.6441]],

                                                  [[ 0.3297,  0.1548,  0.4654],
                                                   [ 0.3501,  0.0823,  0.2731],
                                                   [ 0.4812, -0.0869,  0.3426]],

                                                  [[-0.6839, -0.7020,  0.1232],
                                                   [-0.9552, -0.9167, -0.0558],
                                                   [-0.3642, -0.4037,  0.4471]]],
```

➔ 이는 AWS 에서 실행시킨 결과이다.

5. 결론 및 느낀점

이번 실습을 통해 Colab 에 대해 처음 알았고, GPU 성능이 얼마나 좋은 지 알 수 있었다. 또한, 만약 AWS 만을 이용해서 실습을 진행했으면 완성하지 못했을 수도 있다는 것을 알 수 있었다. 그리고 정확도가 내가 똑같은 코드를 돌린다 해서 모두 같은 결과가 나오지 않는다는 것과 함수와 변수에 따라 정확도가 많은 영향을 받는다는 것을 알 수 있었던 실습이었다.

비록 시간도 오래 걸리고 많은 시행착오를 겪으면서 많은 시도 끝에 낸 결과라서 더 기억에 남고, 인상적이었던 실습이었다. 다만, 마지막에 AWS 를 통해 실습 코드와 결과, 모델(.pkl)을 얻어야 했는데, 네트워크가 끊기면서 다시 실행해야하는 등의 문제로 인하여 Colab 에서 이를 얻을 수밖에 없었다. 이렇게 얻은 pkl 파일은 Colab 에서는 실행이 되지만 AWS 에서는 GPU 를 사용하지 않아

```
model = torch.load('./my_model2.pkl', map_location='cpu')
```

해당 코드로 GPU 를 CPU 로 변경시켜주면 결과가 나온다.

이번 실습을 통해 한 차례 인공지능에 대해 더 잘 알 수 있었고, 앞으로도 다양한 실습을 통해서 많은 인공지능 관련 지식을 배울 수 있었으면 좋겠다.