조주현
박형민교수님
2021년 11월 19일

인공지능(딥러닝)개론 HW02

HW02의 목표는 오른쪽에 서술한 model structure를 따르는 CIFAR10 classification model을 만들고 test data에서 60%이상의 성능을 보이는 것이다. 이 structure을 구현했을 때 각 layer에서 output shape은 다음과 같다.

Input : [128, 6, 32, 32]
1st layer output : [128, 16, 16, 16]
2nd layer output : [128, 16, 8, 8]
3rd layer output : [128, 32, 4, 4]
* batch size는 128으로 정의하였다.

- 1st layer
  - 5x5 kernel, 6 channel out, Batch normalization
  - Activation : ReLU
  - Max-pooling (2x2 kernel, 2 stride)
- 2nd layer
  - 3x3 kernel, 16 channel out, Batch normalization
  - Activation = ReLU
  - Max-pooling (2x2 kernel, 2 stride)
- 3rd layer
  - 3x3 kernel, 32 channel out, Batch normalization
  - Activation = ReLU
  - Max-pooling (2x2 kernel, 2 stride)
- 4th layer
  - Dense Layer with 120 output
  - Activation : ReLU
- 5th layer
  - Dense Layer with 84 output
  - Activation : ReLU
- Out-layer
  - Dense Layer

1. Library

```
1 import torch
2 import torchvision
3 from torchvision import datasets, models, transforms
4 import torchvision.transforms as transforms
5 from torch.utils.data import DataLoader, Dataset, TensorDataset
6 from torch.utils.data.dataset import random_split
7
8 import torch.nn as nn
9 import torch.nn.functional as F
10 import torch.optim as optim
11
12 from torchsummary import summary as summary_
13 import numpy as np
14 import matplotlib.pyplot as plt
```

가장 먼저 학습에 필요한 library들을 호출하는 것이다, 주로 실습시간에 배운 모듈을 호출했고, line 6, 12는 각각 test, validation set 분리, output shape을 출력하는데 필요한 함수이다.

2. parameter와 dataset, dataloader 선언

```
train_dataset, val_dataset = random_split(datasets, [40000,10000])

val_loader = DataLoader(dataset=val_dataset,
                        batch_size = batch_size,
                        shuffle = False)
```

train_loader와 test_loader는 실습시간 사용한 코드와 동일한 코드를 사용했지만, validation set을 위해 50,000개의 train data중에서 40,000개는 training에, 10,000개는 validation에 이용했다.

3. Structure 구현

```
class ConvNet(nn.Module):
  def __init__(self, num_classes=10):
    super(ConvNet, self).__init__()
    self.layer1 = nn.Sequential(
        nn.Conv2d(3, 6, 5, stride=1, padding = 2, padding_mode = 'reflect'), #input : batch_size = 128, 3 channel, 32x32 pixel
        nn.BatchNorm2d(6),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2) # output : batch_size = 128, 6 channel, 16x16 pixel
    )
    self.layer2 = nn.Sequential(
        nn.Conv2d(6, 16, 3, stride=1, padding=1), # input : batch_size = 128, 6 channel, 16x16 pixel
        nn.BatchNorm2d(16),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2) # ouput : batch_size = 128, 16 channel, 8x8 pixel
    )
    self.layer3 = nn.Sequential(
        nn.Conv2d(16, 32, 3, stride=1, padding=1), # input : batch_size = 128, 16 channel, 8X8 pixel
        nn.BatchNorm2d(32),
        nn.ReLU(),
        nn.MaxPool2d(kernel_size=2, stride=2) # output : batch_size = 128, 32 channel, 4x4 pixel
    )
    self.fc1 = nn.Linear(512, 120) # input : 32 x 4 x 4 = 512
    self.fc2 = nn.Linear(120, 80)
    self.fc3 = nn.Linear(80, num_classes)

  def forward(self, x):
    x = self.layer1(x)
    x = self.layer2(x)
    x = self.layer3(x)

    x = x.reshape(x.size(0),-1)
    x = F.relu(self.fc1(x))
    x = F.relu(self.fc2(x))
    x = F.softmax(self.fc3(x))
    return x
```

CIFAR10 image의 크기는(3, 32, 32) 이므로 train loader에서 layer1에 입력하는 data의 shape은 (128, 3, 32, 32)가 된다. layer1의 output shape(i.e. layer2의 Input shape)은 layer1의 6개 체널, layer1의 마지막 함수은 maxpolling2D에 의해 (128, 6, 16, 16)가 된다. 마찬가지로 layer2의 output shape(i.e. layer3의 Input shape)은 (128, 16, 8, 8), layer3의 output shape는 (128, 16, 8, 8)이 되고, layer3의 그것은 (128, 32, 4, 4)가 된다. Fully connected layer에 들어가는 Input은 32 x 4 x 4 = 128이 되고, fc1의 ouput은 120, fc2는 80, fc3는 10의 크기를 갖는 tensor를 내보낸다.

4. Loss and Optimizer

```
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(model.parameters(), lr=0.0005)
```

Loss function은 cross entropy, optimizer는 Adam을 사용하였고 Learning rate는 0.0005로 설정했다.

5. Training

```
17 # Train
18 for epoch in range(num_epochs):
19   correct = 0
20   for i, (images, labels) in enumerate(train_loader):
21
22     # Assign Tensors to Configured Device
23     images = images.to(device)
24     labels = labels.to(device)
25
26     # Forward Propagation
27     outputs = model(images)
28
29     # Get Loss, Compute Gradient, Update Parameters
30     loss_value = criterion(outputs, labels)
31     train_loss_value += loss_value.item()
32     optimizer.zero_grad()
33     loss_value.backward()
34     optimizer.step()
35
36     # evaluate loss and accuracy of train data set
37     _, predicted = torch.max(outputs, 1)
38     correct += (predicted == labels).sum().item()
39
40   train_acc_value = correct / (len(train_loader) * batch_size) * 100
41   train_loss_value = train_loss_value / len(train_loader)
42   train_acc_list.append(train_acc_value)
43   train_loss_list.append(train_loss_value)
44   # Print Loss for Tracking Training
45   print('\n')
46   print('Completed training Epoch', epoch + 1,
47         '\nTraining Accuracy: %.2f%%' %(train_acc_value),
48         '\nTraining Loss: %.4f' %train_loss_value)
```

model과 loss function, optimizer가 모두 선언되었으니 training이 가능하다. 18~34 까지 학습이 진행되고, 그 이후에 train accuracy와 loss값을 계산하고, 매 epoch마다 그를 출력하였다. 37~48 에서처럼 validation과 test의 accuracy와 loss값을 계산하고, 위에서 계산한 모든 값들을 각각 list에 append해 저장해 주었다. 이 list로는 후에 learning curve를 그릴 때 사용한다.

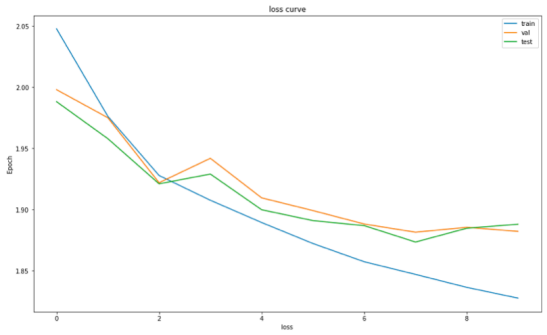
6. Model save

```
89   if epoch+1 == num_epochs:
90     torch.save(model.state_dict(), 'model.pth')
91   else:
92     torch.save(model.state_dict(), 'model-{:02d}_epochs.pth'.format(epoch+1))
```

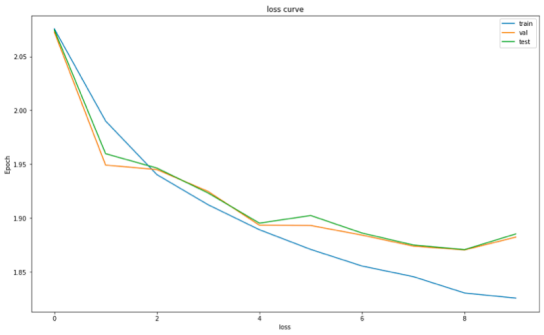
매 epoch마다 해당 epoch을 파일명으로 하는 모델을 저장하고, 학습이 끝나면 model.pth라는 파일명으로 모델을 저장한다.

7. Discussion

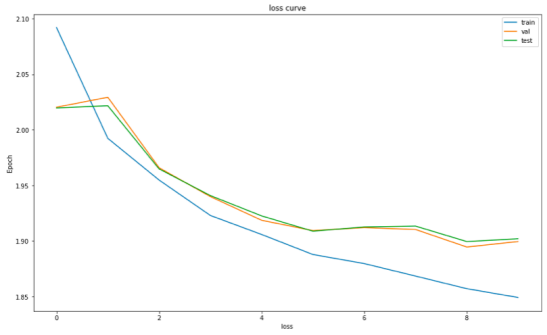
먼저 parameter중 batch_size와 epoch을 각각 128, 30으로 설정한 이유는 아래와 같다.



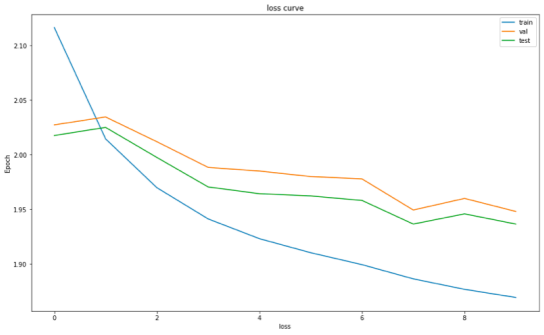
Batch = 32



Batch = 64

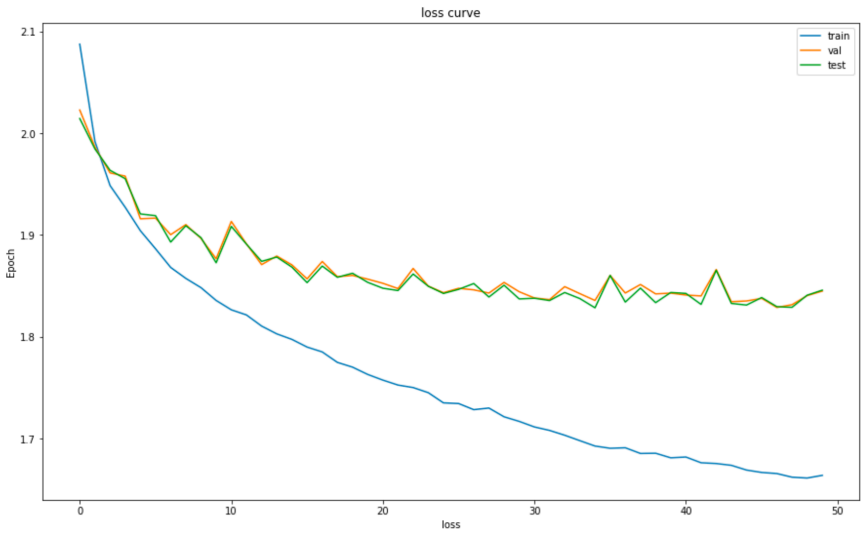


Batch = 128



Batch = 256

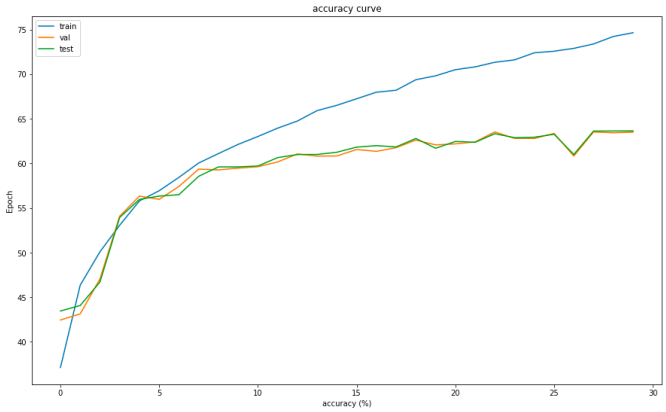
파란색이 train, 주황색이 validation, 초록색이 test dataset에 대한 loss curve이다. 위 4개의 curve를 비교했을 때 32와 64는 128에 비해 overfitting이 심하다고 판단했고, 256의 경우 train이 validation에 잘 적용되지 않는다 판단해서, batch_size는 128로 결정했다.



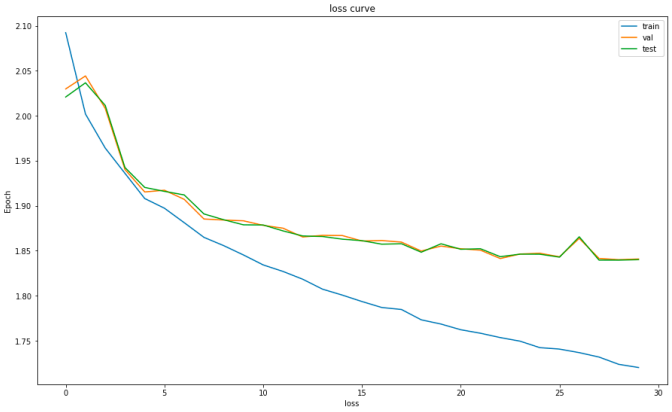
해당 모델을 50 epoch가량 돌린 결과이다. 그래프를 보면 epoch이 30을 넘게 되면 overfitting이 되어 loss값이 줄어들지 않는 것을 볼 수 있다. 따라서 epoch을 30으로 고정하였다.

위 두 그래프를 해석할 때에는 test_loss_curve는 고려하지 않았다. 이는 현실세계에서는 test_data가 존재하지 않는 점을 반영하기 위함이다.

마지막으로 adam optimizer의 learning rate의 경우, 경험적으로 0.001은 너무 크고, 0.0001은 작다고 생각해 그 사이값인 0.0005를 사용하였다.



Accuracy curve



Loss curve

더욱 좋은 모델을 만들기 위해서는 model layer의 complexity를 증가시켜야 할것이라 판단된다. train_acc_curve를 보았을 때 30 epoch을 돌았음에도 accuracy가 75% 근처에 머무른 것과, 약 20 epoch 이후에 test나 validation의 accuracy 증가나, loss의 감소가 보이지 않는것으로 보아 data에 대한 underfitting임을 알 수 있다.

또한 transform 함수에서 ToTensor() 외에 Normalize()함수의 사용, cnn layer의 complexity 증가후 dropout 적용 혹은 weight regularization등을 사용해 더욱 정확한 model을 구현할 수 있을 것이다.