컴퓨터비전 Assignment #2

국민대학교

교수님 : 김장호 교수님
학과 : 자동차IT융합학과
학번 : 20183379
이름 : 박제형

[2-1]

이번 과제는 Overfitting현상을 확인하고, 해결하는 문제이다.

Dataset

'CIFAR10' Dataset(32x32 픽셀의 60,000개의 컬러 이미지(Train : 50,000 / Test : 10,000), 각 이미지는 10개의 클래스로 라벨링 되어있음)을 사용하였다.

Overfitting Model

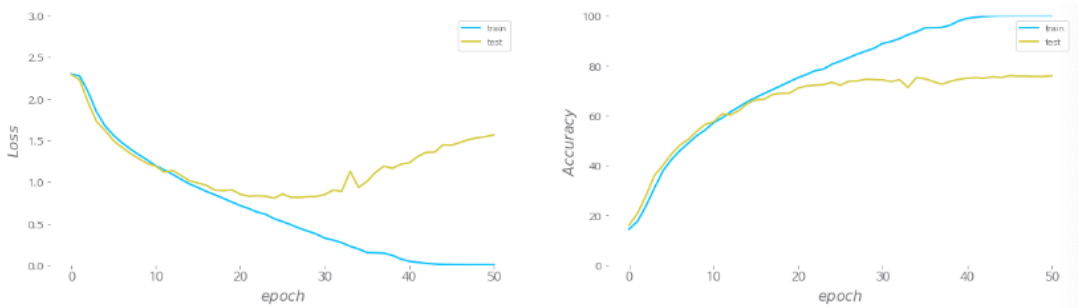
OverFit_model이라는 이름의 클래스로 Overfitting을 만들기 위한 모델을 구현하였다.
기본적인 CNN 모델에 파라미터의 수를 대폭 늘려 Overfit 할 수 있도록 모델을 구성하였다.

```
class OverFit_model(nn.Module):
    def __init__(self):
        super(OverFit_model, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3)
        self.conv2 = nn.Conv2d(64, 128, 3)
        self.conv3 = nn.Conv2d(128, 256, 3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(256, 512, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(512, 1024)
        self.fc2 = nn.Linear(1024, 512)
        self.fc3 = nn.Linear(512, 10)

        self.pool = nn.MaxPool2d(2, 2)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(F.relu(self.conv4(x)))
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

Hyper Parameter

Batch size = 100, epochs = 50
Loss Function : CrossEntropyLoss
Optimizer : SGD (Learning rate = 0.001, momentum = 0.9)

하이퍼 파라미터는 위와 같이 적용하였고, 매 epoch마다 train, test 시 loss와 accuracy를 각각의 List에 append하여 matplotlib.pyplot 라이브러리를 통해서 그래프를 시각화 하였다.
결과는 아래와 같다. (하늘색 : train, 연두색 : test)



수업시간에 배운 것 처럼, epoch이 증가함에 따라 train loss와 accuracy는 잘 수렴하는 반면, test loss와 accuracy는 일정 epoch 이후부터 수렴하지 않는 것을 확인할 수 있다. 전형적인 Overfitting의 그래프 형상이다.

Overfitting 해결

위의 모델을 기반으로, Overfitting 문제를 해결해보겠다.
먼저, Overfitting을 해결할 수 있는 방법으로는 크게 4가지가 있다.

1. **데이터의 양 늘리기**
2. **모델의 복잡도 줄이기**
3. **가중치 규제(Regularization) 적용**
4. **드롭아웃**

위의 네가지 방법을 모두 적용해보고 성능을 비교해보도록 하겠다.

1. 데이터의 양 늘리기

```
transform_train = transforms.Compose([
    transforms.RandomCrop(32, padding=4),
    transforms.RandomHorizontalFlip(),
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])

transform_test = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.4914, 0.4822, 0.4465), (0.2023, 0.1994, 0.2010)),
])
```

pytorch의 torchvision.trasnforms library를 통해 Data augmentation을 하였다. (OverFit_model 모델 학습 시에는 ToTensor(), Normalize()만 사용하였다.)
RandomCrop(32, padding=4) : padding을 해준 후 Random으로 32x32 size로 crop 해준다.
RandomHorizontalFlip() : 특정 확률로 이미지를 수평으로 뒤집는다(flip). 파라미터를 설정해주지 않았으므로 기본 값인 0.5의 확률로 뒤집는다.

2. 모델의 복잡도 줄이기

먼저, 위에서 만든 OverFit_model 모델은 overfitting을 위해 과도하게 파라미터를 많이 사용하였다.
CIFAR10 dataset은 크지 않은 크기의 비교적 단순한 image이기 때문에 적은 파라미터 수로도 좋은 성능을 보일 수 있다.
따라서 아래와 같이 Solution_model을 통해 모델을 수정하였다.

```
class Solution_model(nn.Module):
    def __init__(self):
        super(Solution_model, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 3)
        self.bn1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.bn2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(16, 64, 3)
        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)
        self.pool = nn.MaxPool2d(2, 2)


    def forward(self, x):
        x = self.pool(self.conv1(x))
        x = self.pool(self.conv2(F.relu(self.bn1(x))))
        x = self.pool(self.conv3(F.relu(self.bn2(x))))
        x = x.view(-1, self.num_flat_features(x))
        x = self.fc1(x)
        x = self.fc2(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

3. 가중치 규제(Regularization) 적용

모델의 weight의 제곱합을 패널티 텀으로 주어, overfitting을 방지할 수 있는 L2 Regularization을 적용해보았다. Pytorch의 torch.optim 라이브러리를 통해 optimizer를 선언하는데, 이 때 weight_decay value를 넣어줌으로써 규제를 적용할 수 있다. 이 값은 L2 Regularization 식의 lambda를 의미한다.

```
optimizer = torch.optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9, weight_decay=1e-4)
```

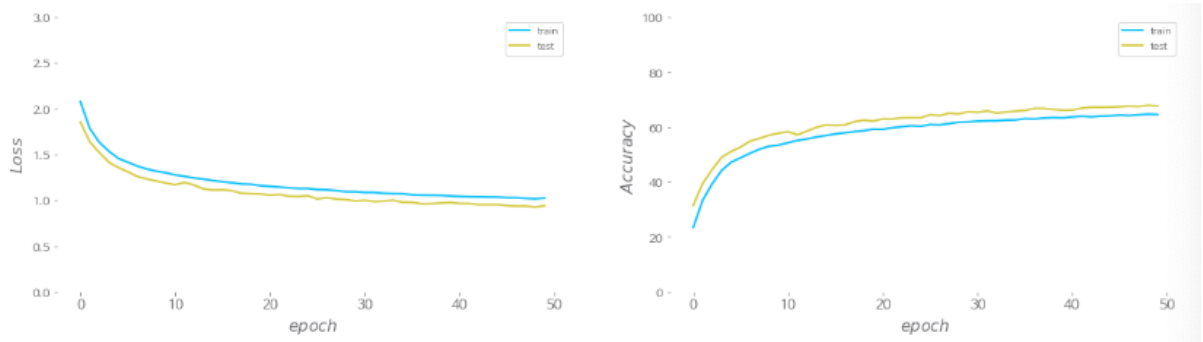
4. 드롭아웃

마지막으로 드롭아웃이다. 드롭아웃을 적용하려다, 수업 중 Batch Normalization이 드롭아웃의 효과도 있다고 들었던 것이 생각났다. 더 공부해보니, Batch Normalization은 가중치의 초기값에 크게 의존하지 않고, 학습을 빨리 진행시킬 수 있을 뿐 아니라 오버피팅을 억제한다는 장점이 있다는 것을 알게 되었고, 드롭아웃 대신 BatchNormalization을 적용하였다.

```
class Solution_model(nn.Module):
    def __init__(self):
        super(Solution_model, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, 3)
        self.bn1 = nn.BatchNorm2d(8)
        self.conv2 = nn.Conv2d(8, 16, 3)
        self.bn2 = nn.BatchNorm2d(16)
        self.conv3 = nn.Conv2d(16, 64, 3)
        self.fc1 = nn.Linear(256, 128)
        self.fc2 = nn.Linear(128, 10)
        self.pool = nn.MaxPool2d(2, 2)


    def forward(self, x):
        x = self.pool(self.conv1(x))
        x = self.pool(self.conv2(F.relu(self.bn1(x))))
        x = self.pool(self.conv3(F.relu(self.bn2(x))))
        x = x.view(-1, self.num_flat_features(x))
        x = self.fc1(x)
        x = self.fc2(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

결과 확인(Solution_model)

이전의 OverFit_model을 학습하였던 것과 같은 Hyper Parameter(batch size=100, epoch=50, ...)을 사용하여, 학습을 시켜보았고, 아래와 같은 결과를 확인할 수 있었다. (하늘색 : train, 연두색 : test)



train, test data set에 대하여 모두 수렴하고 있는 것으로 보아 overfitting 문제를 해결하고 잘 학습이 되고 있음을 확인할 수 있었다.

[2-2]

두 번째 과제는 Baseline 모델 2개와, 직접 제안한 모델을 CIFAR-10 dataset에서 성능 비교 후 결과 분석 및 제안한 모델의 타당성을 서술하는 과제이다.

Hyper Parameter

보다 정확한 성능 비교를 위해 아래와 같이 Hyper Parameter를 고정시키고 모델들을 학습시켜보았다.

- Batch Size = 100
- epoch = 50
- learning rate = 0.001 (Schedular 적용한 모델도 있다.)

모델 설명

Baseline1

Baseline1 모델은 30 channel의
Linear layer 하나로 구성된 모델이다.

```
class Baseline1(nn.Module):
    def __init__(self):
        super(Baseline1, self).__init__()

        self.d1 = nn.Linear(32 * 32 * 3, 30)
        self.d2 = nn.Linear(30, 10)

    def forward(self, x):

        x = x.flatten(start_dim = 1)

        x = self.d1(x)
        x = F.relu(x)

        logits = self.d2(x)
        out = F.softmax(logits, dim=1)
        return out
```

Baseline2

Baseline2 모델은 5x5 size의 kernel을 사용하는
Conv Layer2개와 Linear Layer 3개로 구성된 모델이다.

```
class Baseline2(nn.Module):
    def __init__(self):
        super(Baseline2, self).__init__()
        self.conv1 = nn.Conv2d(3, 6, 5)
        self.pool = nn.MaxPool2d(2, 2)
        self.conv2 = nn.Conv2d(6, 16, 5)
        self.fc1 = nn.Linear(16 * 5 * 5, 120)
        self.fc2 = nn.Linear(120, 84)
        self.fc3 = nn.Linear(84, 10)

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = x.view(-1, 16 * 5 * 5)
        x = F.relu(self.fc1(x))
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x
```

JHNet_1

JHNet_1 모델은 3x3 size을 kernel을 사용하는 Conv layer 4개와, fc layer3개를 사용하여 모델을 구성하였고, 각 레이어의 channel수를 확실히 늘려주었다.
또한 dropout(p=0.5)도 적용하여 많은 파라미터에 대해서 overfitting 하지 않도록 방지하기도 하였다.

```
class JHNet_1(nn.Module):
    def __init__(self):
        super(JHNet_1, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3)
        self.conv2 = nn.Conv2d(64, 64, 3)
        self.conv3 = nn.Conv2d(64, 128, 3, stride=1, padding=1)
        self.conv4 = nn.Conv2d(128, 256, 3, stride=1, padding=1)
        self.fc1 = nn.Linear(256, 512)
        self.fc2 = nn.Linear(512, 512)
        self.fc3 = nn.Linear(512, 10)

        self.pool = nn.MaxPool2d(2, 2)
        self.dropout = nn.Dropout(p=0.5, inplace=False)


    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))
        x = self.pool(F.relu(self.conv2(x)))
        x = self.pool(F.relu(self.conv3(x)))
        x = self.pool(self.dropout(F.relu(self.conv4(x))))
        x = x.view(-1, self.num_flat_features(x))
        x = F.relu(self.fc1(x))
        x = self.dropout(x)
        x = F.relu(self.fc2(x))
        x = self.fc3(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

JHNet_2

JHNet_2 모델은 JHNet_1 모델과 구조는 유사하다. 3x3 kernel의 Conv layer 4개와, 2개의 fc layer를 사용하였다.
이번에는 Dropout 대신 batch normalization을 사용하였다. batch norm layer는 activation function인 relu 함수 앞으로 배치하였다.

추가로 이번 모델은 학습 과정(epoch)에 따라 learning rate를 조정하는 scheduler를 사용하였다. 여러 종류의 scheduler 중 CosineAnnealingLR 을 사용하였다.

```
class JHNet_2(nn.Module):
    def __init__(self):
        super(JHNet_2, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3)
        self.conv2 = nn.Conv2d(64, 128, 3)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(128, 256, 3)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(256, 512, 3)
        self.bn3 = nn.BatchNorm2d(256)
        self.fc1 = nn.Linear(2048, 512)
        self.fc2 = nn.Linear(512, 10)
        self.pool = nn.MaxPool2d(2, 2)


    def forward(self, x):
        x = self.conv1(x)
        x = self.pool(self.conv2(F.relu(self.bn1(x))))
        x = self.pool(self.conv3(F.relu(self.bn2(x))))
        x = self.pool(self.conv4(F.relu(self.bn3(x))))
        x = x.view(-1, self.num_flat_features(x))
        x = self.fc1(x)
        x = self.fc2(x)
        return x

    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

```
schedular = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max = num_epochs, eta_min=0) # schedular add
```

JHNet_3

마지막으로 JHNet_3 모델이다. 과제는 2개의 모델을 제안하는 것이지만, 과제를 하던 중 ResNet에 대한 특성을 적용해보고 싶다는 생각이 들어 하나의 모델을 더 만들어 보았다.

JHNet_3 모델은 JHNet_2와 비슷하게 3x3 kernel의 Conv layer 4개와, 2개의 fc layer를 사용하였다.
추가로 ResNet의 특성인 이전의 값(identity)을 더해주는 부분을 forward 함수에서 구현하였다.

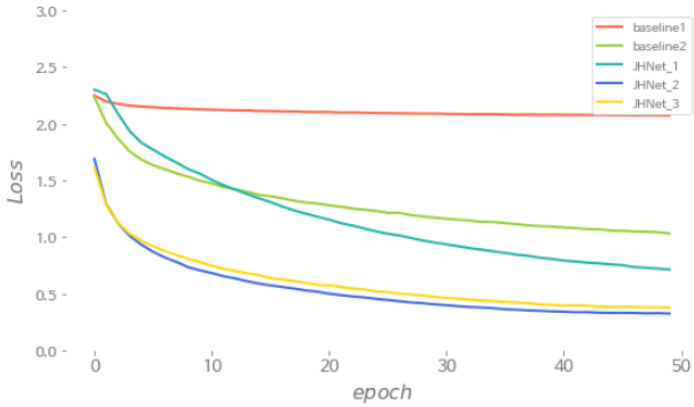
이 모델을 학습 할 때에도 마찬가지로 CosineAnnealingLR scheduler를 사용하여 epoch에 따라 learning rate가 조정될 수 있도록 하였다.

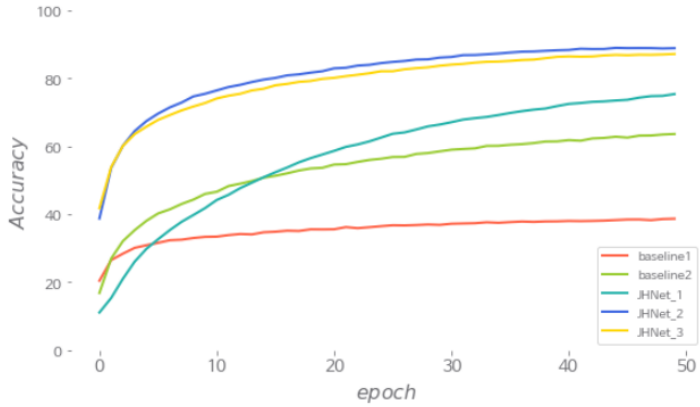
```
class JHNet_3(nn.Module):
    def __init__(self):
        super(JHNet_3, self).__init__()
        self.conv1 = nn.Conv2d(3, 64, 3)
        self.conv2 = nn.Conv2d(64, 128, 3, padding=1)
        self.bn1 = nn.BatchNorm2d(64)
        self.conv3 = nn.Conv2d(128, 256, 3, padding=1)
        self.bn2 = nn.BatchNorm2d(128)
        self.conv4 = nn.Conv2d(256, 64, 3, padding=1)
        self.bn3 = nn.BatchNorm2d(256)
        self.fc1 = nn.Linear(3136, 512)
        self.fc2 = nn.Linear(512, 10)
        self.pool = nn.MaxPool2d(2, 2)


    def forward(self, x):
        x = self.pool(self.conv1(x))
        out = self.conv2(F.relu(self.bn1(x)))
        out = self.conv3(F.relu(self.bn2(out)))
        out = self.conv4(F.relu(self.bn3(out)))
        x = out + x
        x = self.pool(x)
        x = x.view(-1, self.num_flat_features(x))
        x = self.fc1(x)
        x = self.fc2(x)
        return x



    def num_flat_features(self, x):
        size = x.size()[1:]  # all dimensions except the batch dimension
        num_features = 1
        for s in size:
            num_features *= s
        return num_features
```

```
schedular = torch.optim.lr_scheduler.CosineAnnealingLR(optimizer, T_max = num_epochs, eta_min=0) # schedular add
```

성능 비교

매 epoch마다 train, test 시 loss와 accuracy를 각각의 List에 append하여 matplotlib.pyplot 라이브러리를 통해서 그래프를 시각화 하였다.





50 epoch으로 학습시킨 후 각 모델들의 loss과 accuracy를 비교하여 보았을 때,

- Baseline1 모델은 Linear Layer1개로 구성된 모델로 아주 천천히 학습이 되고 있음을 볼 수 있고,
- Baseline2 모델은 2개의 Conv Layer와 3개의 Linear Layer로 Baseline1 모델보다 빠르게 loss가 떨어지며 학습이 되는 것을 확인할 수 있다.
- 직접 제안한 모델인 JHNet_1 모델은 Baseline2 모델보다 훨씬 많은 파라미터 수를 갖는 모델로, 보다 빠르게 수렴하며 학습이 되는 것을 확인할 수 있다.
- JHNet_2 모델은 Dropout 대신 Batch Norm을 적용하였고, scheduler를 사용하여 학습시킨 모델로, 눈에 띄게 빠른 속도로 수렴하고 학습이 잘 이루어 지는 것을 확인할 수 있다.
- 마지막으로 JHNet_3 모델은 ResNet의 특성을 적용한 모델로, JHNet_2와 거의 비슷하게 수렴하는 것을 확인할 수 있다. JHNet_3이 Layer의 깊이가 깊지 않기 때문에 Backpropagation시 gradient vanishing 문제가 크게 발생하지 않고, 그렇기 때문에 gradient vanising에 대한 해결책으로 나온 방안중 하나인 ResNet의 특성을 적용했음에도 성능면에서 큰 효과를 볼 수 없지 않았나 생각이 든다.

또한 모든 모델들이 완전히 수렴한 것이 아니었기 때문에 epoch을 늘렸을 때 어떤 결과가 나오는지 확인해보고, 또 layer의 깊이를 깊게 쌓았을 때 ResNet의 효과를 볼 수 있는지 확인해봐야겠다는 생각이 들었다. 이러한 내용으로 추가적인 학습을 해 볼 생각이다.