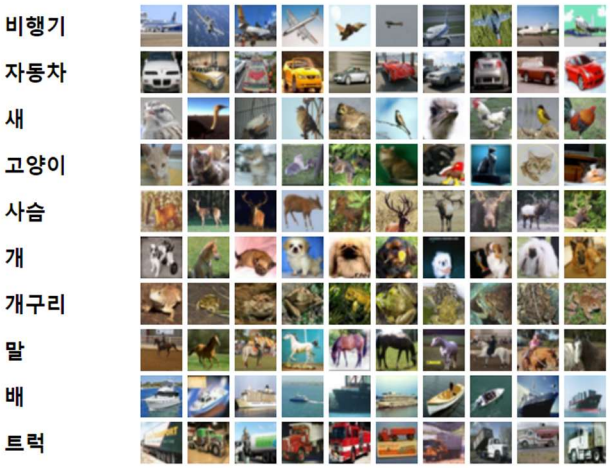
인공지능(딥러닝)개론 HW2 결과보고서

CNN을 통한 CIFAR10 Dataset Classifier Model 만들기

학번: 20180032 이름: 남기동

1. CIFAR10 Dataset

CIFAR10 Dataset이란 32x32 픽셀의 60000개 컬러 이미지로 구성된 데이터로 10개 클래스로 라벨링이 되어있다. 각각의 클래스는 [비행기, 자동차, 새, 고양이, 사슴, 개, 개구리, 말, 배, 트럭]이다.



CIFAR10 Dataset은 MNIST와 함께 머신러닝 연구에 가장 널리 사용되는 dataset중 하나이다.

2. CNN 모델 설명

CNN 모델을 설계하기 위해서는 우선 하이퍼 파라미터를 설정해주어야 하며 몇 개의 Covolution Layer와 몇 개의 Dense Layer로 구성할 것인지 등을 선정할 필요가 있다. 여러 시도 이후에 최종적으로 선택한 모델에 대해서 설명할 것이며, 각각의 부분이 이전 시도와 달라진 부분이 있다고 한다면, 해당 부분을 바꾸었을 때 어떤 효과를 얻을 수 있는지, 그리고 test accuracy에서 어느 정도의 차이가 있었는지에 대해 분석해보고자 한다.

1) Hyper-Parameter

설정해야 할 하이퍼 파라미터는 4개이며 batch_size, max_pool_kernnel, learning_rate, num_epochs이 그것이다. 우선 max_pool_kernel은 이번 과제에서 Covolution Layer에서 Max Pooling을 수행하

게 되는데 그 때 사용하는 Max Pooling의 Kernel 사이즈를 설정하는 하이퍼 파라미터이다. 과제의 제한 조건에 "Max-pooling layer의 kernel size는 2 x 2로 구성"이라는 조건이 있기 때문에 해당 하이퍼 파라미터는 2로 설정하였다. 다음으로 learning_rate는 모델의 gradient에 곱해주는 step size이며 이 수치에 따라 모델이 학습하는 정도가 달라진다. Learning_rate가 지나치게 크면 모델이 제대로 학습하지 못하고 발산할 위험이 있으며 오히려 지나치게 작으면 학습이 더디게 되는 문제가 발생한다. Learning_rate는 0.001이 일반적으로 적절한 수준이기 때문에 0.001로 설정하였다. 다음으로 num_epochs은 학습 횟수를 의미하며 학습이 많아지면 모델의 성능이 좋아지지만 너무 학습이 많아지면 모델이 학습하는데 지나치게 오랜 시간이 걸리는 것과 함께 train data에 대해 overfitting이 발생하여 오히려 test data에 대한 성능이 떨어지는 문제가 발생할 수 있다. 이번 과제에서는 30~50 정도의 num_epochs을 사용하였는데 최종적으로 선택한 모델의 경우 30을 사용하였다. 마지막으로 batch_size는 모델이 학습하는 과정에서 파라미터를 업데이트 할 때 사용할 데이터의 개수를 의미한다. Batch size가 커지면 더 많은 데이터를 동시에 처리하기 때문에 연산이 효율적이고 빨라지며 일반화에 도움이 될 수 있다. 반면 Batch size가 작아지면 연산이 늘어나고 훈련 속도는 감소하지만 local minimum에서 벗어날 수 있는 장점이 있다. 최종 모델에서는 batch_size를 30으로 설정하였다.

```
# Device Configuration
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')

# 최종 클래스와 입력 채널을 설정
num_classes = 10 # CIFAR10은 10개의 클래스로 설정
in_channel = 3   # CIFAR10은 컬러 이미지이기 때문에(RGB) 3개 채널로 입력

# 하이퍼 파라미터 설정
batch_size = 50
max_pool_kernel = 2 # "Max-pooling layer"의 kernel size는 2x2로 구성"
learning_rate = 0.001
num_epochs = 50
```

2) Covolution Layer

```
# 1st Layer
self.layer1 = nn.Sequential(  # 32x32 in_channel==3 --> padding==2 --> 36x36, channel=3 --> cnn 5x5 --> 32x32 out_channel=32
    nn.Conv2d(in_channels=in_channel, out_channels=32, kernel_size=5, stride=1, padding=2), # in_channel==3, 5*5 kernel
    nn.BatchNorm2d(num_features=32),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=max_pool_kernel) # 32 * 32, channel=32 --> 2x2 max_pooling --> 16x16, channel=32
)
# 2nd Layer
self.layer2 = nn.Sequential( # 16x16, in_channel=32 --> padding==2 --> 20x20, channel=32 --> cnn 5x5 --> 16x16, out_channel=64
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=64),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=max_pool_kernel) # 16x16, channel=64 --> 2x2 max_pooling --> 8x8, channel=64
)
# 3rd layer
self.layer3 = nn.Sequential( # 8x8, in_channel=64 --> padding==2 --> 12x12, channel=64 --> cnn 5x5 --> 8x8, out_channel=128
    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=5, stride=1, padding=2),
    nn.BatchNorm2d(num_features=128),
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=max_pool_kernel) # 8x8, channel=128 --> 2x2 max_pooling --> 4x4, channel=128
)
```

CNN 모델은 총 6개의 층으로 구성되어 있다. Input layer은 제외하고 Covolution Layer 3개와 Dense Layer 3개가 연결되어 있다. 그 중 이 파트에서 설명할 부분은 Convolution Layer이다. 첫 번째 층에서 입력 채널은 CIFAR10 Dataset이 32x32 픽셀의 컬러 이미지이기 때문에 RGB에 해당하는 3개이다. 3개의 Covolution Layer는 입력 채널과 출력 채널이 다른 것을 제외하고는 모두 같은 구조를 갖고 있다. 우선 padding을 2로 하였고 stride는 1, Kernel(filter)의 크기는 5x5로 하여 Covolution 연산을 수행한다. Padding을 2로 설정하고 5x5 Kernel을 사용하여 convolution 연산을 수행하면 기존의 이미지의 크기는 변하지 않은 채로 covolution 연산이 이루어진다. Covolution layer 이후에는 Batch Normalization을 이용하여 미니 배치의 입력을 정규화하여 학습을 안정화하며 ReLu(Rectified Linear Unit)을 activation function으로 사용한다. ReLu 함수는 음수는 0으로 만들고 양수는 그대로 반환하는 특징을 갖는다. 다음 층으로 넘어가기 전에 마지막으로 2x2 Max Pooling을 수행하여 입력 이미지의 크기를 줄인다. 2x2 Kernel로 max pooling을 수행하고 나면 이미지의 크기가 반으로 감소한다. 이러한 구조를 통해 3번째 Covolution Layer와 poolig layer를 지나고 났을 때 이미지의 크기는 4x4이며 채널의 수는 128이 된다.

3) Dense Layer

```
# 4th Layer(1st Fully_connected)
self.fc1 = nn.Linear(in_features=4 * 4 * 128, out_features= 100)   # 2048 --> 100
self.dropout1 = nn.Dropout(p=0.3)
# 5th Layer(2nd Fully_connected)
self.fc2 = nn.Linear(in_features=100, out_features=50)            #  100 -->  50
self.dropout2 = nn.Dropout(p=0.3)
# Output Layer(3rd Fully_connected)
self.fc3 = nn.Linear(in_features=50, out_features=num_classes)    #   50 -->  10
```

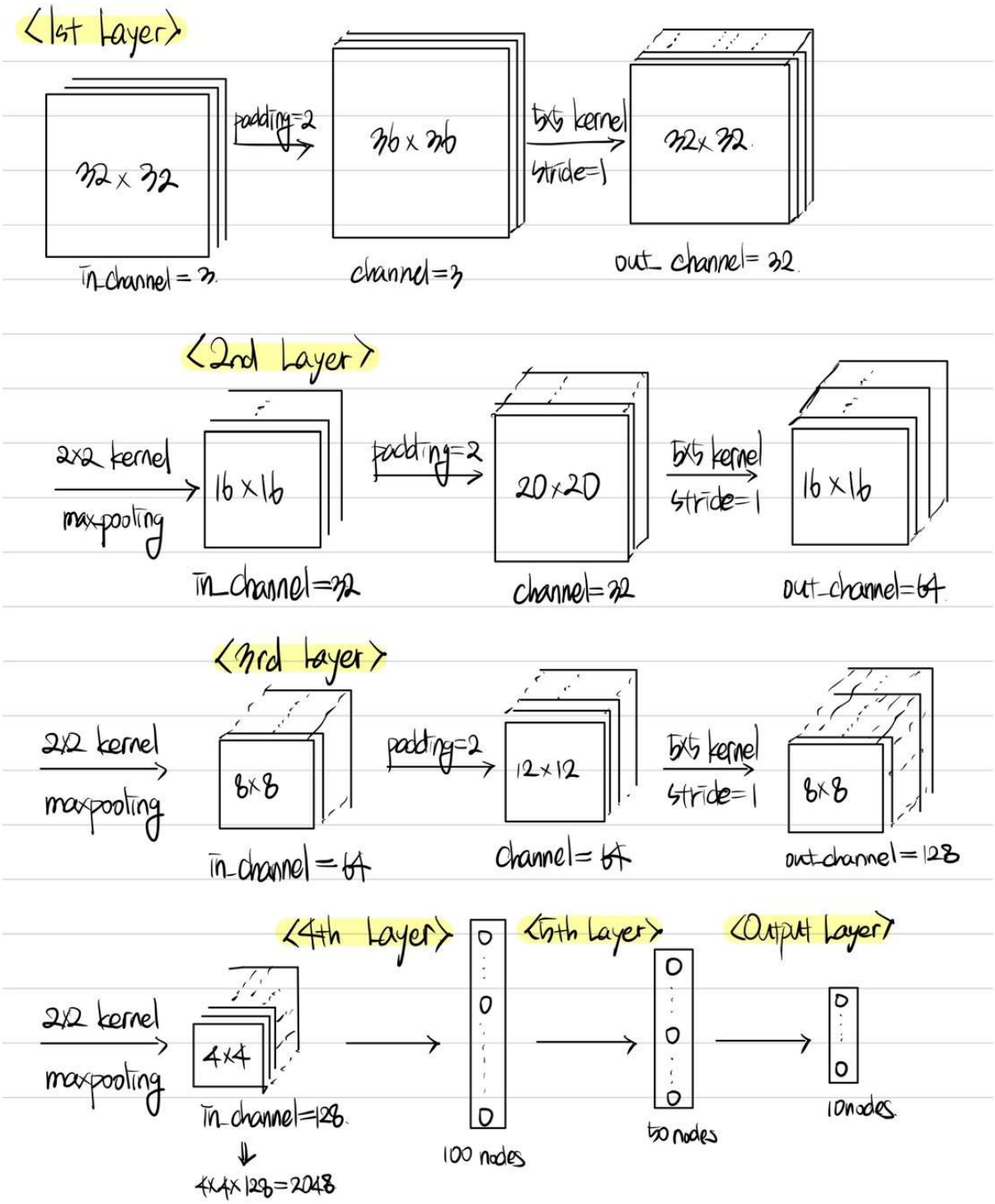
Fully Connected Layer라고도 하는 Dense Layer는 네트워크의 최종 출력을 생성하는데 사용되며, 기존의 4x4 사이즈의 이미지와 128개의 채널을 Dense Layer에 넣을 수 있도록 1차원으로 flatten하고 3개의 Dense Layer을 지나게 된다. 각각의 Dense Layer 사이에는 확률 p를 설정하여 드롭아웃을 수행한다.

```
x = x.reshape(x.size(0),-1) # fully_connected에 넣을 수 있도록 flatten

x = F.relu(self.fc1(x))
x = self.dropout1(x)  # 드롭아웃 적용
x = F.relu(self.fc2(x))
x = self.dropout2(x)  # 드롭아웃 적용
x = self.fc3(x)
```

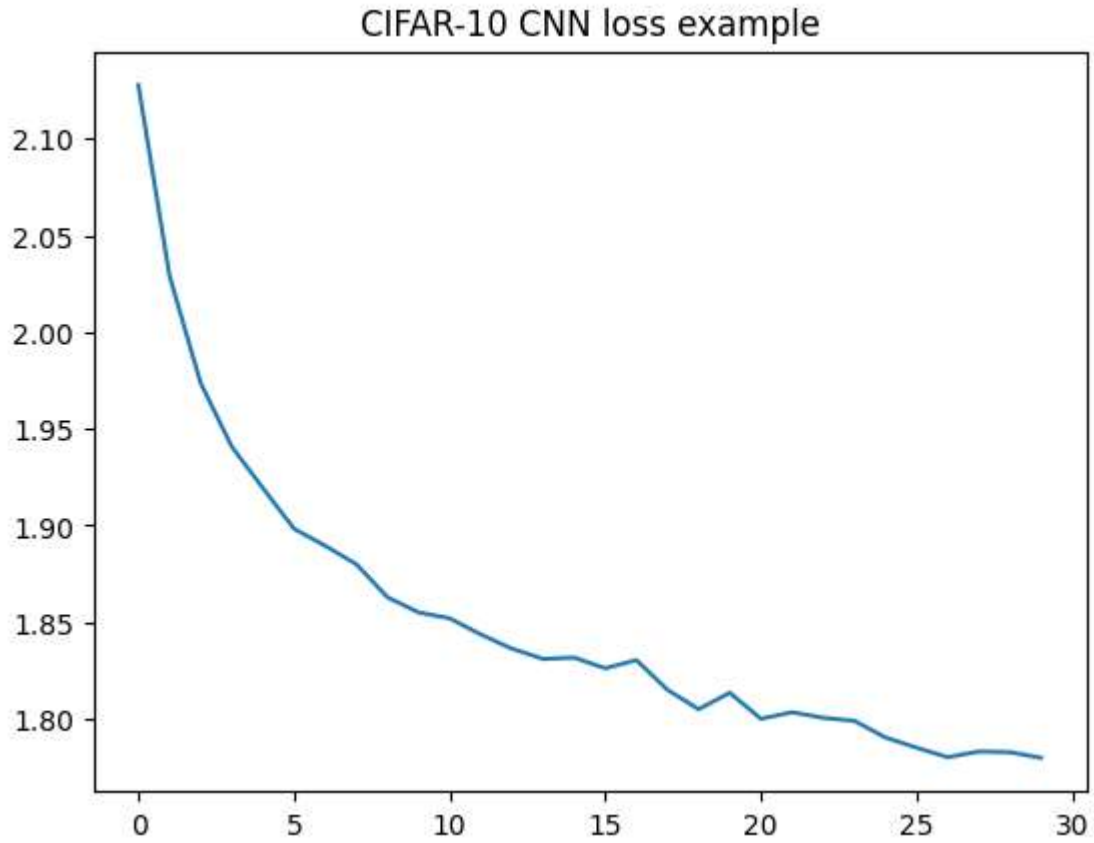
활성화 함수로는 각각의 Dense Layer을 지나고 ReLu를 사용해주었다.

4) 각 Layer를 통과한 시점에서의 output의 shaep(size)



6) Learning Curve와 Test Accuracy

아래는 설계한 CNN 모델로 epoch이 30일 때 학습함에 따라 감소하는 Loss를 나타낸 그림이다.



그림을 보면 Loss가 epoch이 증가함에 따라 지속적으로 감소하는 것을 확인할 수 있지만, epoch이 증가함에 따라 연산의 양이 많아져서 지나치게 오랜 시간이 걸리는 문제가 있다. 또한 Loss가 감소하는 것이 단순히 모델의 성능의 향상으로 직결되는 것이 아니라, 지나치게 많은 학습량은 모델이 학습 데이터에 과적합되는 문제가 발생하여 오히려 일반화 성능이 떨어져 테스트 데이터로 측정한 분류 정확도가 떨어질 수 있다. 따라서 epoch을 30으로 설정하였으며 최종적인 분류 정확도(Accuracy)는 약 78%였다.

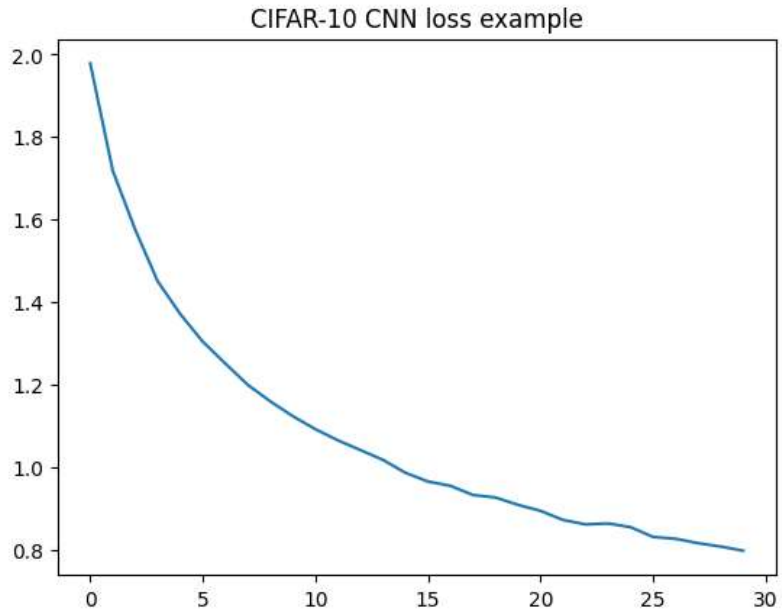
```
Accuracy of the network on the 10000 test images 78.34%
```

3. Accuracy를 올리기 위한 방법

1) batch_size의 감소

하이퍼 파라미터 중 하나인 Batch size는 모델이 학습하는 과정에서 파라미터를 업데이트 할 때 사용할 데이터의 개수를 의미한다. Batch size가 커지면 더 많은 데이터를 동시에 처리하기 때문에 연산이 효율적이고 빨라지며 일반화에 도움이 될 수 있다. 하지만 위의 사진은 완성된 모델에

batch size를 256으로 설정했을 때의 loss 그래프이다. Batch size를 30으로 줄이고 나서 더 작은 배치를 사용하여 모델이 많은 데이터 샘플을 보고 학습하여 Loss도 더 감소하고 일반화 성능도 향상되어 높은 정확도를 얻을 수 있었다.



2) Covolution Layer의 Channel 확대

또한 기존의 covolution layer에서 output channel을 늘리는 것도 모델의 성능을 향상시키는데 도움이 되었다. 기존에 사용했던 모델은 아래와 같다.

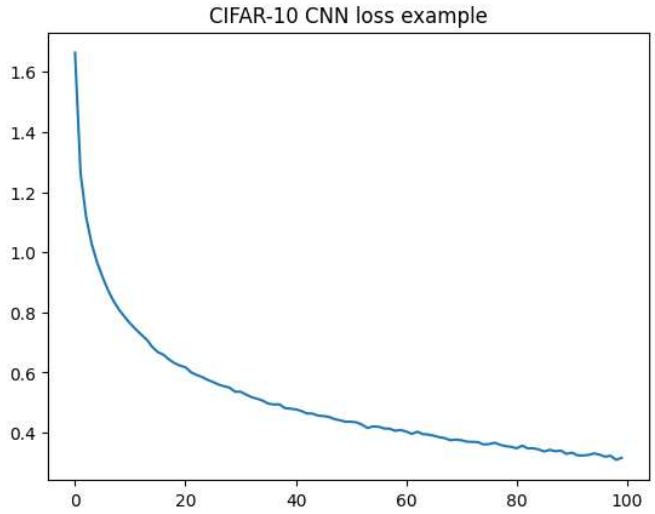
```
# 1st Layer
self.layer1 = nn.Sequential(  # 32x32 in_channel==3 --> padding==2 --> 36x36, channel=3 --> cnn 5x5 --> 32x32 out_channel=16
    nn.Conv2d(in_channels=in_channel, out_channels=32, kernel_size=5, stride=1, padding=2), # in_channel==3, 5*5 kernel
    nn.BatchNorm2d(num_features=32), # convolution의 output channel 그대로
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=max_pool_kernel) # 32 * 32, channel=16 --> 2x2 max_pooling --> 16x16, channel=16
)
# 2nd Layer
self.layer2 = nn.Sequential( # 16x16, in_channel=16 --> padding==2 --> 20x20, channel=16 --> cnn 5x5 --> 16x16, out_channel=32
    nn.Conv2d(in_channels=32, out_channels=64, kernel_size=5, stride=1, padding=2), # 1층의 output인 16을 in으로,
    nn.BatchNorm2d(num_features=64), # out_channel 그대로
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=max_pool_kernel) # 16x16, channel=32 --> 2x2 max_pooling --> 8x8, channel=32
)
# 3rd layer
self.layer3 = nn.Sequential( # 8x8, in_channel=32 --> padding==2 --> 12x12, channel=32 --> cnn 5x5 --> 8x8, out_channel=64
    nn.Conv2d(in_channels=64, out_channels=128, kernel_size=5, stride=1, padding=2), # 1층의 output인 16을 in으로,
    nn.BatchNorm2d(num_features=128), # out_channel 그대로
    nn.ReLU(),
    nn.MaxPool2d(kernel_size=max_pool_kernel) # 8x8, channel=64 --> 2x2 max_pooling --> 4x4, channel=64
)
```

애초에 3개의 convolution layer을 지나고 2x2 max pooling도 3번을 거치기 때문에 32x32 크기의 이미지는 Covolution Layer을 지나고 4x4로 감소한다. 그러한 상황에서 연산에 사용되는 feature의 수를 늘리기 위해서는 out_channel을 늘려야 한다. 기존에 비해 out_channel을 2배 향상시킴으로써 모델을 기존보다 복잡하게 만들고 연산의 양을 늘려서 학습의 효율을 높이고자 하였다. 그리

고 이렇게 함으로써 정확도가 올라가는 효과를 얻었다.

3) Epoch을 줄이고 Dropout을 적용하여 일반화 성능 향상

드롭 아웃은 정해진 p확률만큼 학습 중에 뉴런의 연결을 제거, 혹은 비활성화하여 모델의 과적합을 방지하고 일반화 성능을 향상시키기 위해 사용하는 방법이다. Dense Layer에서는 모든 뉴런이 연결이 되는데 연산의 양이 증가함에 따라 모델이 학습 데이터에 대해서 과적합(Overfitting)되는 문제가 발생할 수 있다.



이 사진은 dropout을 적용하지 않았고 epoch이 100이었을 때의 모델의 loss를 나타낸 그래프이다. Epoch이 증가함에 따라 loss가 감소하여 마지막에는 0.3 수준에 도달하는데 이는 현재 최종적으로 선택한 모델보다 낮은 손실을 기록하는 것이다. 하지만 이 경우의 정확도는 오히려 약 63% 정도를 기록했다. 따라서 적정 수준으로 epoch을 조절하고 dropout을 적용하여 모델의 일반화 성능을 향상시키고자 하였다.