

HW 05 – REPORT

소속 : 정보컴퓨터공학부

학번 : 201924437

이름 : 김윤하

1. 서론

1-1. 실습 목표

이번 실습의 목표는 딥러닝 기법(Deep Learning)을 활용하여 신경망을 설계하고 구현하는 방법을 학습하는 것입니다. 특히, Convolution Neural Networks (CNNs)를 중심으로 다양한 레이어와 그 특성을 이해하고, 이를 바탕으로 주어진 데이터셋을 활용한 이미지 분류 모델을 구축하는 것입니다. 이번 과제를 통해서, CNN의 기본 개념부터 실제 구현까지의 전 과정을 경험하며, 신경망의 성능을 향상시키기 위한 다양한 기술들을 실습할 수 있습니다.

1-2. 이론적 배경

1-2-1. 신경망 (Neural Networks)

신경망은 뇌의 신경 구조를 모방하여 만든 기계 학습 모델로, 입력 데이터와 출력 데이터를 연결하는 비선형 함수들을 층층이 쌓아 학습을 통해 패턴을 인식하고 예측하는 데 사용됩니다. 신경망은 입력층, 은닉층, 출력층으로 구성되며, 각 층은 여러 개의 뉴런으로 이루어져 있습니다. 뉴런들은 가중치와 편향을 통해 서로 연결되며, 활성화 함수(Activation Function)를 통해 비선형성을 부여할 수 있습니다.

1-2-2. 합성곱 신경망

CNN은 이미지 데이터와 같은 고차원 데이터를 처리하기 위해 설계된 신경망의 일종입니다. 주요 특징으로는 합성곱 레이어(Convolutional Layer), 풀링 레이어(Pooling Layer), 완전 연결 레이어(Fully Connected Layer) 등이 있습니다.

- 합성곱 레이어 (Convolutional Layer) : 입력 이미지에서 특징을 추출하기 위해 필터(커널)를 사용하여 합성곱 연산을 수행합니다. 필터는 가중치 행렬로, 학습을 통해 최적화됩니다.
- 풀링 레이어(Pooling Layer) : 합성곱 레이어의 출력을 다운샘플링하여 계산량을 줄이고, 특징의 위치 불변성을 유지합니다. 주로 최대 풀링(Max Pooling)과 평균 풀링(Average Pooling)을 사용합니다.
- 활성화 함수(Activation Function) : 각 뉴런의 출력값에 비선형성을 부여하여 모델이 복잡한 데이터 패턴을 학습할 수 있도록 합니다. 대표적인 활성화 함수로는 ReLU (Rectified Linear Unit), Sigmoid, Tanh 등이 있습니다.

1-2-3. 최적화 기법

CNN을 학습하기 위해서는 손실 함수를 최소화하는 방향으로 가중치와 편향을 조정하는 최적화

기법이 필요합니다. 대표적인 최적화 기법으로는 확률적 경사 하강법(Stochastic Gradient Descent, SGD), Adam, RMSprop 등이 있습니다. 이번 과제에서는 SGD와 모멘텀을 사용하여 모델을 최적화합니다.

1-2-4. 정규화 기법

모델의 일반화 성능을 높이기 위해 정규화 기법을 사용합니다. 대표적인 정규화 기법으로는 드롭아웃(Dropout), 배치 정규화(Batch Normalization) 등이 있습니다. 정규화 기법은 과적합을 방지하고, 모델의 학습 속도를 향상시키는 데 도움이 됩니다.

2. 본론

2-1. Question 1

이 문제의 경우, 주어진 three-layer ConvNet을 완성하는 것을 목표로 하였습니다. 문제에서 주어진 5가지 네트워크 구조를 구현하기 위해 각 단계에 걸쳐 소스 코드를 완성하였습니다. 다음은 해당 코드의 원문입니다.

```
### Question 1
#####
# TODO: Implement the forward pass for the three-layer ConvNet. #
#####
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# 1. Convolutional layer (with bias) with 32 5x5 filters, with zero-padding of 2
x = F.conv2d(x, conv_w1, bias=conv_b1, padding=2)
# 2. ReLU
x = F.relu(x)

# 3. Convolutional layer (with bias) with 16 3x3 filters, with zero-padding of 1
x = F.conv2d(x, conv_w2, bias=conv_b2, padding=1)
# 4. ReLU
x = F.relu(x)

# 5. Fully-connected layer (with bias) to compute scores for 10 classes
# + Flatten 적용
x = flatten(x)
scores = x.mm(fc_w) + fc_b

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#####
#                                     END OF YOUR CODE                                     #
#####
```

그림 1 : Question 1

주어진 조건 대로, x를 입력 받아 three-layer convolution network를 수행하는 것을 볼 수 있습니다. Conv2d 함수를 통해 합성곱 연산을 수행하며, 활성화 함수로 ReLU 함수를 적용시킵니다. 또한 주어진 파라미터를 사용하여 각 필터를 적용시켜주고 이후 fully-connected layer까지 적용시킵니다. 또한, fc_b인 Bias를 더해주어 scores로 리턴해주는 함수를 완성하였습니다.

2-2. Question 2

이 문제의 경우, 'random_weight' 및 'zero-weight' 함수를 활용해 컨볼루션 레이어와 fully-connected 레이어의 가중치 및 편향을 초기화하였습니다. 다음은 소스 코드의 원문입니다.

```

### Question 2
#####
# TODO: Initialize the parameters of a three-layer ConvNet.
#####
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# TODO
# Initialize your weight matrices using the `random_weight` function defined above,
# and you should initialize your bias vectors using the `zero_weight` function above of a three-layer ConvNet.
conv_w1 = random_weight((channel_1, 3, 5, 5))
conv_b1 = zero_weight((channel_1,))
conv_w2 = random_weight((channel_2, channel_1, 3, 3))
conv_b2 = zero_weight((channel_2,))

fc_w = random_weight((channel_2 * 32 * 32, 10))
fc_b = zero_weight((10,))

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#####
#                                     END OF YOUR CODE
#####

```

그림 2 : Question 2

위 코드를 살펴보면, 주어진 각 변수들에 대해 random_weight와 zero_weight 함수들을 사용하여 conv_w1, conv_b1, conv_w2, conv_b2들을 계산해 주었습니다. 이후 fc_w에서 크기가 (channel_2 * 32 * 32, 10)인 벡터를 통해 무작위 초기화를 수행해주었습니다.

<pre> Iteration 0, loss = 3.7973 Checking accuracy on the val set Got 112 / 1000 correct (11.20%) Iteration 100, loss = 1.7770 Checking accuracy on the val set Got 335 / 1000 correct (33.50%) Iteration 200, loss = 1.8842 Checking accuracy on the val set Got 377 / 1000 correct (37.70%) Iteration 300, loss = 1.7832 Checking accuracy on the val set Got 392 / 1000 correct (39.20%) Iteration 400, loss = 1.7476 Checking accuracy on the val set Got 427 / 1000 correct (42.70%) Iteration 500, loss = 1.5956 Checking accuracy on the val set Got 457 / 1000 correct (45.70%) Iteration 600, loss = 1.5171 Checking accuracy on the val set Got 465 / 1000 correct (46.50%) Iteration 700, loss = 1.5930 Checking accuracy on the val set Got 463 / 1000 correct (46.30%) </pre> <p>그림 3 : Question 2 의 결과</p>	<p>이후, 문제에서 주어진 'train_part2' 함수를 호출해 초기화된 모델과 파라미터를 사용해 학습을 진행하였습니다. 학습은 여러 반복을 걸쳤으며 각 반복마다 loss와 정확도를 출력한 결과, 학습이 진행될 때마다 정확도가 향상되는 것을 확인하였습니다.</p> <p>옆의 결과를 확인하면, 마지막 반복에서 검증 세트에 대한 정확도가 약 46.3%로 측정된 것을 확인할 수 있습니다. 또한, 초기 손실은 높았지만 학습이 진행됨에 따라 정확도가 향상되었습니다.</p>
--	---

출력 결과를 통해 작성한 코드가 문제없이 잘 동작함을 확인할 수 있었습니다.

2-3, 2-4. Question 3, 4

해당 문제들의 경우 같은 클래스 내에 구현되었습니다. 세 개의 레이어로 구성된 컨볼루션 신경망을 정의하고 초기화하는 과정이었습니다. 문제에서 주어진 대로 첫 번째와 두 번째 레이어의 커널의 크기, 패딩 등을 지정하였고, 가중치 초기화를 수행하였습니다. 또한 nn.Conv2d와 nn.Linear 함수들을 사용하여 각 ToDo를 해결할 수 있었습니다. 다음은 해당 코드들의 원문입니다.

```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        ### Question 3
        #####
        # TODO: Set up the layers you need for a three-layer ConvNet with the #
        # architecture defined above. #
        #####
        # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

        # TODO:
        # Set up the layers you need for a three-layer ConvNet
        # with the architecture defined above Question1 by using 'nn.Module API'.

        # Q 3-1. 32 필터 (5x5, padding=2) using nn.Conv2d
        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight)

        # Q 3-2. 16 필터 (3x3, padding=1) using nn.Conv2d
        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1)
        nn.init.kaiming_normal_(self.conv2.weight)

        # fully-connected layer
        self.fc = nn.Linear(channel_2 * 32 * 32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight)

        # ReLU 활성화 함수
        self.relu = nn.ReLU(inplace=True)
        # Flatten (1d vector로)
        self.flatten = nn.Flatten()

        # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
        #####
        #                               END OF YOUR CODE                               #
        #####
```

그림 4 : Question 3

자세한 설명은 코드와 주석을 통해 확인할 수 있습니다.

```

def forward(self, x):
    scores = None
    ### Question 4
    #####
    # TODO: Implement the forward function for a 3-layer ConvNet. you #
    # should use the layers you defined in __init__ and specify the #
    # connectivity of those layers in forward() #
    #####
    # *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

    # TODO:
    # Implement the forward function for a 3-layer ConvNet.
    # You should use the layers you defined in __init__ and specify the connectivity of those layers in forward()

    # 첫 번째 컨볼루션 레이어
    x = F.relu(self.conv1(x))
    # 두 번째 컨볼루션 레이어
    x = F.relu(self.conv2(x))
    # Flatten
    x = flatten(x)
    # fully-connected
    scores = self.fc(x)

    # *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
    #####
    #                               END OF YOUR CODE                               #
    #####
    return scores

```

그림 5 : Question 4

4번 문제에서는, 앞에서 정의한 레이어를 이용해 'forward(self, x)' 함수를 구현하였습니다. 두 컨볼루션 레이어를 통과하고 flatten 작업을 수행한 뒤 fully-connected 레이어를 거쳐 scores를 리턴해 주었습니다. 코드를 Run 하면 다음과 같이 의도한 대로 동작한 것을 확인할 수 있었습니다.

```

torch.Size([64, 10])

```

그림 6 : Q3, Q4 결과 화면

2-5. Question 5

이 문제의 경우, CIFAR 데이터 셋에서 3개의 레이어로 이루어진 컨볼루션 신경망을 학습시켰습니다. 모델과 최적화하는 것이 주요 목표였으며, momentum 없는 Stochastic Gradient Descent(SGD) 옵티마이저를 이용하였습니다. 다음은 코드의 원문입니다.

```

### Question 5
#####
# TODO: Instantiate your ThreeLayerConvNet model and a corresponding optimizer #
#####
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# TODO:
# Instantiate your three layer ConvNet model and a corresponding optimizer.
# You should train the model using stochastic gradient descent without momentum.
model = ThreeLayerConvNet(in_channel=3, channel_1=channel_1, channel_2=channel_2, num_classes=10)
optimizer = optim.SGD(model.parameters(), lr=learning_rate)


# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#####
#                               END OF YOUR CODE                               #
#####

```

그림 7 : Question 5

앞서 구현된 ThreeLayerConvNet 모델과 해당 모델에 대한 SGD 옵티마이저를 초기화해 주었습니다. 모델은 3개의 층으로 구성되어 있으며 첫번째 및 두번째는 컨볼루션 레이어이며, 마지막 레이어는 fully-connected layer 입니다.

이후, 'train_part34'를 통해 학습시킨 결과는 다음과 같습니다.

 <pre> Iteration 0, loss = 3.1695 Checking accuracy on validation set Got 140 / 1000 correct (14.00) Iteration 100, loss = 1.9549 Checking accuracy on validation set Got 327 / 1000 correct (32.70) Iteration 200, loss = 1.8358 Checking accuracy on validation set Got 389 / 1000 correct (38.90) Iteration 300, loss = 1.6488 Checking accuracy on validation set Got 439 / 1000 correct (43.90) Iteration 400, loss = 1.5497 Checking accuracy on validation set Got 445 / 1000 correct (44.50) Iteration 500, loss = 1.6882 Checking accuracy on validation set Got 458 / 1000 correct (45.80) Iteration 600, loss = 1.5334 Checking accuracy on validation set Got 467 / 1000 correct (46.70) Iteration 700, loss = 1.4286 Checking accuracy on validation set Got 482 / 1000 correct (48.20) </pre> <p>그림 8 : Question 5의 결과</p>	<p>일정 간격으로 검증 셋(validation set)의 결과를 확인한 결과, 마찬가지로 훈련이 진행됨에 따라 손실이 감소하며 검증 세트에 대한 정확도가 증가하는 것을 확인할 수 있었습니다.</p> <p>또한, 마지막 반복에서는 약 48.2%의 정확도를 가지며 이는 문제의 요건인 45%보다 높으므로, 의도한 대로 모델이 구현된 것을 잘 확인할 수 있었습니다.</p>
---	--

2-6. Question 6

이번 문제의 경우, Sequential API를 사용해 3개의 층으로 이루어진 컨볼루션 신경망을 정의하고 학습하는 과정을 나타내었습니다. 여기서 문제에서 제시한 대로 Nesterov momentum = 0.9인 SGD를 사용하여 optimizer를 구현한 것을 확인할 수 있습니다. nn.Sequential을 사용해 모델을 정의하였습니다.

먼저, 입력 채널의 개수를 3으로 설정하며 출력 채널의 개수는 channel_1으로 설정해 주었습니다. 또한, 커널은 5, 패딩은 2로 조정하여 Conv2d를 적용시키고 활성화 함수로 ReLU를 사용해 주었습니다. 또한, 두 번째 레이어에서 입력 채널의 개수는 channel_1, 출력 채널의 개수는 channel_2, 커널 크기 3, 패딩 1을 적용시켜 마찬가지로 활성화 함수까지 적용시켜 주었습니다. 이후 Flatten을 사용해 평탄화하고 Linear 함수를 이용해 선형 레이어를 생성하였습니다. 또한 앞서 말한대로 0.9의 momentum을 갖는 SGD 방법으로 옵티마이저를 사용하였습니다.

다음은 코드의 원문입니다.


```

### Question 6
#####
# TODO: Rewrite the 2-layer ConvNet with bias from Part III with the
# Sequential API.
#####
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

# TODO:
# To define and train a three-layer ConvNet with the same architecture we used in Part III with the Sequential API.
# You can use the default PyTorch weight initialization. You should optimize your model using stochastic gradient descent with momentum 0.9.

# 2-layer ConvNet
model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2),
    nn.ReLU(),
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1),
    nn.ReLU(),
    Flatten(),
    nn.Linear(channel_2 * 32 * 32, 10)
)

optimizer = optim.SGD(model.parameters(), lr=learning_rate, momentum=0.9)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#####
#                               END OF YOUR CODE
#####

```

그림 9 : Question 6

이후, 'train_part34' 함수를 통해 모델을 학습하였습니다.

<pre> ➡ Iteration 0, loss = 2.2957 Checking accuracy on validation set Got 146 / 1000 correct (14.60) Iteration 100, loss = 1.7203 Checking accuracy on validation set Got 432 / 1000 correct (43.20) Iteration 200, loss = 1.4814 Checking accuracy on validation set Got 479 / 1000 correct (47.90) Iteration 300, loss = 1.4325 Checking accuracy on validation set Got 493 / 1000 correct (49.30) Iteration 400, loss = 1.4825 Checking accuracy on validation set Got 475 / 1000 correct (47.50) Iteration 500, loss = 1.3282 Checking accuracy on validation set Got 527 / 1000 correct (52.70) Iteration 600, loss = 1.0656 Checking accuracy on validation set Got 552 / 1000 correct (55.20) Iteration 700, loss = 1.2445 Checking accuracy on validation set Got 581 / 1000 correct (58.10) </pre>	<p>학습 결과, 마찬가지로 학습이 진행됨에 따라 손실이 감소하고 검증 세트에 대해 정확도가 증가하는 것을 확인하였습니다. 또한 마지막 반복에서는 문제에서 제시한 대로 55% 이상의 정확도를 얻었습니다.</p>
<p>그림 10 : Question 5의 결과 화면</p>	

해당 과정을 통해, Sequential API를 사용하여 간단하고 효율적으로 컨볼루션 신경망을 정의하고 훈련하는 방법을 배울 수 있었습니다.

2-7. Question 7

마지막으로, 이번 문제에서는 70% 이상의 정확도를 갖는 multi-layer ConvNet 모델을 구현하는 것이 목표였습니다. 문제에서 제시했던 대로 'Filter size', 'Number of filters', 'Pooling vs Strided Convolution', 'Batch normalization', 'Global Average Pooling', 'Regularization'등을 고려하여 실습을 진행하였습니다.

모델은 세 개의 Convolution Layer를 가지며, 마지막으로 하나의 fully-connected layer를 갖도록 설계하였습니다. 각 레이어에 대한 상세 설계 내용은 다음과 같습니다.

1st Convolution Layer	2nd Convolution Layer	3rd Convolution Layer	Fully-connected Layer
입력 채널 : 3(RGB) 출력 채널 : 32 커널 크기 : 3x3 패딩 : 1	입력 채널 : 32 출력 채널 : 64 커널 크기 : 3x3 패딩 : 1	입력 채널 : 64 출력 채널 : 128 커널 크기 : 3x3 패딩 : 1	입력 노드 수 : $128 * 4 * 4$ (3rd Convolution 레이어와 맞춤) 출력 노드 수 : 512 (최종 출력 노드 수 : 10)

모델 구조를 설계할 때에는 다음과 같은 요소들을 고려하였습니다.

- 작은 필터 크기 : 3x3의 필터를 사용하여 연산량을 줄이면서 세밀한 이미지의 특징을 효과적으로 학습할 수 있도록 설정하였습니다.
- 활성화 함수 : ReLU 함수를 사용하여 계산 효율성이 높으면서도 그래디언트 손실 문제를 완화하였습니다.
- Max Pooling : 크기를 축소시키고 연산량을 감소시키는 Max Pooling을 각 Convolutional Layer 후에 적용하였습니다.
- Fully-connected Layer : 마지막에 두 개의 레이어를 씌워 특성을 결합하며 최종 클래스를 예측하도록 설계하였습니다.

위 내용을 기반으로 작성된 소스 코드 원문은 다음과 같습니다.

```
# *****START OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****

class CustomCNN(nn.Module):
    def __init__(self):
        super(CustomCNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.conv3 = nn.Conv2d(64, 128, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(128 * 4 * 4, 512)
        self.fc2 = nn.Linear(512, 10)
```

```

def forward(self, x):
    x = F.relu(self.conv1(x))
    x = F.max_pool2d(x, 2)
    x = F.relu(self.conv2(x))
    x = F.max_pool2d(x, 2)
    x = F.relu(self.conv3(x))
    x = F.max_pool2d(x, 2)
    x = x.view(-1, 128 * 4 * 4)
    x = F.relu(self.fc1(x))
    x = self.fc2(x)
    return x

model = CustomCNN()

# Define your optimizer
optimizer = optim.Adam(model.parameters(), lr=0.001)

# *****END OF YOUR CODE (DO NOT DELETE/MODIFY THIS LINE)*****
#####
#                                     END OF YOUR CODE                                     #
#####

```

그림 11 : Question 7

이후 Optimizer의 경우 Adam Optimizer와 손실 함수는 Cross-Entropy Loss를 사용하여 10 epochs를 수행한 결과는 다음과 같습니다. 아래는 첫 번째, 아홉 번째, 열 번째 epoch를 수행한 결과입니다.

<p>Iteration 0, loss = 2.3021 Checking accuracy on validation set Got 102 / 1000 correct (10.20)</p> <p>Iteration 100, loss = 1.4976 Checking accuracy on validation set Got 440 / 1000 correct (44.00)</p> <p>Iteration 200, loss = 1.2266 Checking accuracy on validation set Got 518 / 1000 correct (51.80)</p> <p>Iteration 300, loss = 1.4547 Checking accuracy on validation set Got 557 / 1000 correct (55.70)</p> <p>Iteration 400, loss = 1.1322 Checking accuracy on validation set Got 551 / 1000 correct (55.10)</p> <p>Iteration 500, loss = 1.1598 Checking accuracy on validation set Got 591 / 1000 correct (59.10)</p> <p>Iteration 600, loss = 1.1566 Checking accuracy on validation set Got 621 / 1000 correct (62.10)</p> <p>Iteration 700, loss = 0.8802 Checking accuracy on validation set Got 650 / 1000 correct (65.00)</p>	<p>Iteration 0, loss = 0.0552 Checking accuracy on validation set Got 763 / 1000 correct (76.30)</p> <p>Iteration 100, loss = 0.0765 Checking accuracy on validation set Got 774 / 1000 correct (77.40)</p> <p>Iteration 200, loss = 0.0956 Checking accuracy on validation set Got 778 / 1000 correct (77.80)</p> <p>Iteration 300, loss = 0.1044 Checking accuracy on validation set Got 771 / 1000 correct (77.10)</p> <p>Iteration 400, loss = 0.1882 Checking accuracy on validation set Got 759 / 1000 correct (75.90)</p> <p>Iteration 500, loss = 0.0746 Checking accuracy on validation set Got 760 / 1000 correct (76.00)</p> <p>Iteration 600, loss = 0.2899 Checking accuracy on validation set Got 772 / 1000 correct (77.20)</p> <p>Iteration 700, loss = 0.2407 Checking accuracy on validation set Got 789 / 1000 correct (78.90)</p>	<p>Iteration 0, loss = 0.0633 Checking accuracy on validation set Got 763 / 1000 correct (76.30)</p> <p>Iteration 100, loss = 0.1333 Checking accuracy on validation set Got 764 / 1000 correct (76.40)</p> <p>Iteration 200, loss = 0.0307 Checking accuracy on validation set Got 767 / 1000 correct (76.70)</p> <p>Iteration 300, loss = 0.1200 Checking accuracy on validation set Got 772 / 1000 correct (77.20)</p> <p>Iteration 400, loss = 0.0900 Checking accuracy on validation set Got 772 / 1000 correct (77.20)</p> <p>Iteration 500, loss = 0.0781 Checking accuracy on validation set Got 770 / 1000 correct (77.00)</p> <p>Iteration 600, loss = 0.0531 Checking accuracy on validation set Got 767 / 1000 correct (76.70)</p> <p>Iteration 700, loss = 0.1627 Checking accuracy on validation set Got 781 / 1000 correct (78.10)</p>
---	---	---

위를 살펴보면, 9 번째 epoch의 'Iteration 700'에서 78.9%의 정확도를 달성한 것을 확인할 수 있습니다.

이후 test_set으로 모델을 돌려본 결과, 다음과 같이 75.83%의 정확도를 도출할 수 있었습니다.

```

Checking accuracy on test set
Got 7583 / 10000 correct (75.83)

```

3. 결론

이번 실습을 통해 CIFAR-10 데이터 세트를 활용하여 다양한 Convolutional Neural Network(CNN) 아키텍처를 설계하였으며, 이를 통해 모델의 성능을 효율적으로 개선하는 과정을 탐구하였습니다. 실습을 통해 tree-layer CNN 구조를 시작으로 여러 가지 아키텍처와 하이퍼파라미터 조합을 실험하였으며, 최종적으로 70% 이상의 Validation set 정확도를 달성하는 모델을 구현할 수 있었습니다.

기본 CNN을 설계할 때에는, 단순한 2-layer CNN 및 3-layer CNN을 구현하여 CIFAR-10 데이터 세트를 분류하는 기본 구조를 학습하였습니다. 첫 번째 실험에서는 5x5와 3x3 필터를 사용하여 두 개의 Convolutional Layer를 구성하였으며, 학습 과정에서 필터 크기와 레이어 수가 모델의 성능에 미치는 영향을 확인할 수 있었습니다. 이를 통해 간단한 CNN 모델도 적절한 하이퍼파라미터 설정을 통해 어느 정도의 성능을 낼 수 있음을 이해할 수 있었습니다.

이후 더 복잡한 구조의 CNN 모델을 설계하였습니다. 3 개의 Convolutional Layer를 통해 32, 64, 128개의 필터를 적용하였고, 각 레이어 후에 Max Pooling과 ReLU 활성화 함수를 추가하여 비선형성과 공간 축소를 동시에 달성하였습니다. 이는 CIFAR-10 이미지의 다양한 특징을 학습하는 데에 도움을 주었습니다. 특히, 다양한 필터의 수와 커널의 크기를 실험하며 각 레이어의 출력을 조정하는 방법들을 학습할 수 있었습니다.

또한, 이후 최적화 기법을 비교하며 실습한 내용들을 통해 SGD는 단순하지만 학습 속도가 느리며 Adam은 학습율을 자동으로 조정해주며 빠른 수렴을 시켜주는 효과가 있음을 확인하였습니다. 이로써 최적화 기법의 선택이 모델의 성능 및 학습 효율성에 영향을 미칠 수 있다는 점을 확인할 수 있었습니다.

이번 실습을 통해 CNN 기본 구조와 동작 원리를 이해하고, CIFAR-10과 같은 복잡한 데이터 세트에서 CNN을 효과적으로 설계하고 학습시키는 방법을 알게 되었습니다. 기본적인 2-Layer CNN을 시작으로 여러 CNN 모델의 성능을 개선할 수 있었습니다. 이를 통해 아키텍처를 설계하고(적절한 필터 크기, 레이어 수, 활성화 함수의 선택이 모델의 성능에 큰 영향을 미치며 더 깊고 복잡한 모델이 데이터의 다양한 패턴을 학습하는 데에 유리), 하이퍼파라미터를 튜닝하고(최적화 기법과 하이퍼파라미터 설정은 모델의 학습 속도와 성능에 중요한 역할을 함), 모델의 일반화(데이터 증강과 정규화 기법을 통해 모델의 일반화 성능을 높이고 과적합을 방지)를 수행하는 방법들을 익혔습니다. 최종적으로는 CNN 모델을 설계하고 최적화하는 과정을 통해 딥러닝 모델이 데이터의 복잡한 패턴을 학습하는 방법과 그 성능을 향상시키는 다양한 기법을 종합적으로 이해할 수 있었습니다. 이후 이러한 기초적인 CNN 모델 설계에 대한 지식을 바탕으로, 향후에 더 복잡한 데이터 세트의 모델들을 다루게 된다면 다양한 아키텍처를 적용하고, 하이퍼파라미터들을 튜닝하고, 모델을 일반화하여 효율적이며 좋은 성능을 내는 적절한 모델을 설계할 수 있을 것입니다.