



HW 05 - REPORT

소속 : 정보컴퓨터공학부

학번 : 202155592

이름 : 이지수

서론

실습 목표

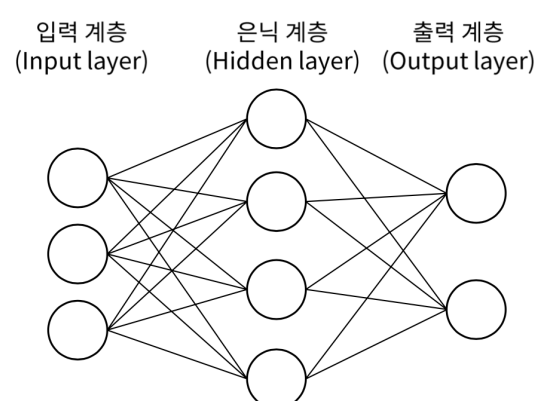
1. Barebones PyTorch를 사용하여 PyTorch 텐서를 직접 다루는 방법을 익힌다.
2. `nn.Module` 을 사용하여 신경망 아키텍처를 정의하는 방법을 익힌다.
3. `nn.Sequential` 을 사용하여 linear feed-forward 네트워크를 정의하는 방법을 익힌다.
4. architectures, hyperparameters, loss functions, optimizers 등을 변경하여 모델의 정확도를 향상시킬 수 있다.

이론적 배경

1. PyTorch

- 딥러닝과 기계 학습을 위한 오픈 소스 프레임워크이다.
- Python 기반의 과학 계산 패키지인 토치(Torch)를 기반으로 개발되었다.
- 텐서 연산과 자동 미분을 위한 기능을 제공하여 신경망 모델의 구성, 학습, 추론을 쉽게 구현할 수 있다.
- 파이썬 라이브러리(Numpy, Scipy, Cython)와 호환성이 높다.
- Define by Run 방식을 기반으로 실시간 결과값을 시각화한다.

2. Neural network



- 뉴런과 layer: 인공 신경망은 입력을 받아 여러 layer를 거쳐 출력을 생성하는 뉴런의 집합으로 구성된다. 각 layer는 가중치와 bias를 가지며 활성화 함수를 통해 비선형성을 추가한다.
- 순전파 (Forward Propagation): 입력 데이터가 네트워크를 통과하면서 각 layer에서 계산된 값을 다음 레이어로 전달하는 과정이다.
- 역전파 (Backward Propagation): 출력과 실제 값 사이의 오차를 최소화하기 위해 가중치를 업데이트하는 과정이다. 손실 함수의 기울기를 계산하여 가중치를 조정한다.

3. Regularization

- 드롭아웃 (Dropout): 학습 중 무작위로 뉴런을 비활성화하여 과적합을 방지하는 기법으로, 모델이 특정 뉴런에 과도하게 의존하지 않도록 한다. 드롭아웃은 학습 과정에서만 활성화되며 평가 시에는 모든 뉴런을 사용한다.
- 배치 정규화 (Batch Normalization): 미니배치 단위로 입력을 정규화하여 학습 속도를 높이고 초기화에 덜 민감하게 만드는 기법이다. 이는 각 층의 입력을 정규화하여 학습이 안정적으로 진행되도록 도와준다.

4. Stochastic gradient descent (SGD)

- 한 번에 전체 dataset을 사용하는 대신, 예를 들어 32개의 이미지나 32개의 샘플을 한 batch로 사용하여 훈련한다.
- 전체 dataset을 학습한 횟수를 epoch라고 한다. 일반적으로 여러 번의 epoch를 수행해야 한다.
- 각 batch에서 자동으로 기울기를 계산하기 위해 역전파를 사용한다.

5. Training a convolutional neural network

- Data preprocessing
 - 학습 데이터를 인위적으로 늘려 데이터를 증강시킬 수 있다.
- Choose your architecture

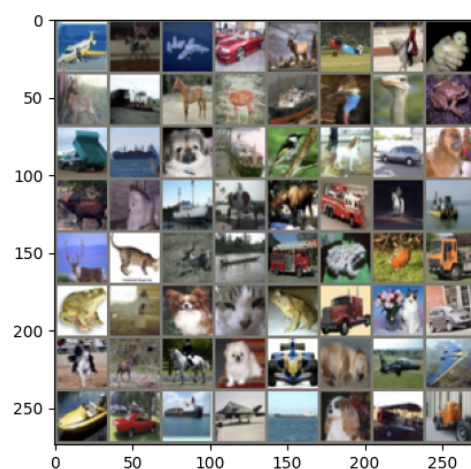


- Initialize your weights
 - Set the weights to small random numbers
 - Set the bias to zero (or small nonzero)
- Overfit a small portion of the data and find a learning rate

본론

Part I. Preparation

- 준비 단계에서는 CIFAR-10 dataset을 로드하고 시각화한다. 각 이미지를 PyTorch tensor로 변환하고 normalize 한 후 dataset을 training, validation, test set으로 나눈다. 64개의 이미지를 시각화하면 아래와 같다.



Part II. Barebones PyTorch

- `flatten` 함수는 입력 tensor를 flatten한다. `N` (batch 크기)을 제외한 나머지 차원들을 하나의 벡터로 변환한다.


- `test_flatten` 함수는 `flatten` 함수를 테스트한다. $2 \times 1 \times 3 \times 2$ 크기의 tensor를 2×6 크기로 flatten한다.
- `two_layer_fc` 함수는 two-layer fully-connected ReLU network의 forward pass를 수행한다. 입력 tensor를 flatten 한 후, 두 개의 FC layer를 거쳐 최종 score를 계산한다.
- `three_layer_convnet` 함수는 three-layer convolutional network의 forward pass를 수행한다. 입력 이미지에 대해 두 개의 convolutional layer를 거친 후, 출력 tensor를 flatten하고 FC layer를 통해 최종 클래스 score를 계산한다.

```
# Question 1
def three_layer_convnet(x, params):
    conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b = params
    scores = None

    x = F.conv2d(x, conv_w1, conv_b1, padding=2) # 첫 번째 convolutional layer (1)
    x = F.relu(x) # ReLU (2)
    x = F.conv2d(x, conv_w2, conv_b2, padding=1) # 두 번째 convolutional layer (3)
    x = F.relu(x) # ReLU (4)
    x = flatten(x)
    scores = x.mm(fc_w) + fc_b # FC Layer (5)

    return scores
```

`three_layer_convnet_test` 함수를 실행한 결과는 아래와 같다.

 torch.Size([64, 10])

- `random_weight` 와 `zero_weight` 함수는 가중치를 초기화하는 역할을 한다. `random_weight` 는 Kaiming normalization을 사용하여 가중치를 랜덤으로 생성하고, `zero_weight` 는 가중치를 0으로 초기화한다.
- `check_accuracy_part2` 함수는 모델의 정확도를 평가한다.
- `train_part2` 함수는 모델을 학습시키고, 학습 중 손실과 정확도를 출력한다.
- two-layer fully-connected network를 학습시키기 위해 가중치를 초기화하고 학습을 수행한다. iteration이 증가할수록 손실은 낮아지고 정확도는 높아지는 것을 볼 수 있다.
- three-layer convolutional network의 파라미터를 초기화하고, 이를 사용하여 모델을 학습한다.

```
# Question 2
learning_rate = 3e-3

channel_1 = 32
channel_2 = 16

conv_w1 = None
conv_b1 = None
conv_w2 = None
conv_b2 = None
fc_w = None
fc_b = None

conv_w1 = random_weight((channel_1, 3, 5, 5)) # 첫 번째 convolutional layer의 가중치 초기화 [out_channel, in_channel, height, width]
conv_b1 = zero_weight((channel_1, )) # 첫 번째 convolutional layer의 bias 초기화 [out_channel]
conv_w2 = random_weight((channel_2, channel_1, 3, 3)) # 두 번째 convolutional layer의 가중치 초기화 [out_channel, in_channel, height, width]
conv_b2 = zero_weight((channel_2, )) # 두 번째 convolutional layer의 bias 초기화 [out_channel]
fc_w = random_weight((channel_2 * 32 * 32, 10)) # FC layer의 가중치 초기화 [out_channel, in_channel]
fc_b = zero_weight((10, )) # FC layer의 bias 초기화 [out_channel]

params = [conv_w1, conv_b1, conv_w2, conv_b2, fc_w, fc_b]
train_part2(three_layer_convnet, params, learning_rate)
```

실행 결과는 아래와 같다.

```
↔ Iteration 0, loss = 2.8110
Checking accuracy on the val set
Got 88 / 1000 correct (8.80%)

Iteration 100, loss = 1.8940
Checking accuracy on the val set
Got 330 / 1000 correct (33.00%)

Iteration 200, loss = 1.8199
Checking accuracy on the val set
Got 398 / 1000 correct (39.80%)

Iteration 300, loss = 1.7527
Checking accuracy on the val set
Got 404 / 1000 correct (40.40%)

Iteration 400, loss = 1.6794
Checking accuracy on the val set
Got 436 / 1000 correct (43.60%)

Iteration 500, loss = 1.5040
Checking accuracy on the val set
Got 449 / 1000 correct (44.90%)

Iteration 600, loss = 1.5112
Checking accuracy on the val set
Got 457 / 1000 correct (45.70%)

Iteration 700, loss = 1.3567
Checking accuracy on the val set
Got 468 / 1000 correct (46.80%)
```

Part III. PyTorch Module API

- `TwoLayerFC` 클래스
 - `__init__` 함수는 `input_size`, `hidden_size`, `num_classes` 를 받아서 두 개의 FC layer를 정의한다. `fc1` 은 입력에서 hidden layer로 연결되는 층이고, `fc2` 는 hidden layer에서 출력 클래스로 연결되는 층이다. 각 layer의 가중치는 Kaiming normalization로 초기화한다.
 - `forward` 함수는 입력 데이터를 flatten 하고 `fc1` 을 거쳐 ReLU 활성화를 적용한 후 `fc2` 를 통해 최종 score를 계산한다.
- `ThreeLayerConvNet` 클래스
 - `__init__` 함수는 `in_channel`, `channel_1`, `channel_2`, `num_classes` 를 받아서 두 개의 convolutional layer와 하나의 FC layer를 정의한다. `conv1` 은 입력에서 첫 번째 convolutional layer로 연결되는 층이고, `conv2` 는 첫 번째 convolutional layer에서 두 번째 convolutional layer로 연결되는 층이다. `fc` 는 두 번째 convolutional layer에서 출력 클래스로 연결되는 FC layer이다. 각 layer의 가중치는 Kaiming normalization로 초기화한다.

```
class ThreeLayerConvNet(nn.Module):
    def __init__(self, in_channel, channel_1, channel_2, num_classes):
        super().__init__()
        # Question 3
        # 첫 번째 convolutional layer
        self.conv1 = nn.Conv2d(in_channel, channel_1, kernel_size=5, padding=2)
        nn.init.kaiming_normal_(self.conv1.weight) # 가중치 초기화

        # 두 번째 convolutional layer
        self.conv2 = nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1)
        nn.init.kaiming_normal_(self.conv2.weight) # 가중치 초기화

        # FC layer
        self.fc = nn.Linear(channel_2 * 32 * 32, num_classes)
        nn.init.kaiming_normal_(self.fc.weight) # 가중치 초기화

        # ReLU
        self.relu = nn.ReLU()
```

- `forward` 함수는 입력 데이터를 `conv1` 을 통해 convolution 연산을 하고 ReLU 함수를 적용한 후, `conv2` 를 통해 다시 convolution 연산을 하고 ReLU 함수를 적용한다. 그 후 데이터를 flatten 하여 `fc` 층을 통해 최종 score를 계산한다.


```
def forward(self, x):
    scores = None
    # Question 4
    # 첫 번째 convolutional layer 통과 후 ReLU 활성화 함수 적용
    x = self.conv1(x)
    x = self.relu(x)

    # 두 번째 convolutional layer 통과 후 ReLU 활성화 함수 적용
    x = self.conv2(x)
    x = self.relu(x)

    x = flatten(x)

    # FC layer 통과 후 score 계산
    scores = self.fc(x)
```

`test_ThreeLayerConvNet` 함수를 실행한 결과는 아래와 같다.

 `torch.Size([64, 10])`

- `check_accuracy_part34` 함수는 loader를 사용하여 모델의 정확도를 평가한다. 평가 모드로 전환한 후, 각 batch에 대해 모델의 출력을 계산하고, 예측된 클래스와 실제 클래스를 비교하여 정확도를 계산한다.
- `train_part34` 함수에서는 모델을 주어진 dataset에 대해 학습시키고, 각 epoch마다 손실을 계산한 후 backwards pass를 통해 가중치를 업데이트한다.
- 앞서 정의한 `TwoLayerFC` 클래스의 인스턴스를 생성하고, SGD optimizer를 사용하여 모델을 학습시킨다.
- 앞서 정의한 `ThreeLayerConvNet` 클래스의 인스턴스를 생성하고, SGD optimizer를 사용하여 모델을 학습시킨다.

```
learning_rate = 3e-3
channel_1 = 32
channel_2 = 16

model = None
optimizer = None
# Question 5
model = ThreeLayerConvNet(3, channel_1, channel_2, 10) # ThreeLayerConvNet 모델 인스턴스화
optimizer = optim.SGD(model.parameters(), lr=learning_rate) # SGD optimizer 설정

train_part34(model, optimizer)
```

`train_part34` 함수를 실행한 결과는 아래와 같다.

```

↔ Iteration 0, loss = 3.2911
Checking accuracy on validation set
Got 132 / 1000 correct (13.20)

Iteration 100, loss = 1.8676
Checking accuracy on validation set
Got 339 / 1000 correct (33.90)

Iteration 200, loss = 1.8600
Checking accuracy on validation set
Got 400 / 1000 correct (40.00)

Iteration 300, loss = 1.9497
Checking accuracy on validation set
Got 425 / 1000 correct (42.50)

Iteration 400, loss = 1.7837
Checking accuracy on validation set
Got 435 / 1000 correct (43.50)

Iteration 500, loss = 1.8155
Checking accuracy on validation set
Got 464 / 1000 correct (46.40)

Iteration 600, loss = 1.6098
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)

Iteration 700, loss = 1.4410
Checking accuracy on validation set
Got 457 / 1000 correct (45.70)

```

Part IV. PyTorch Sequential API

- Two-layer fully connected network
 - `Flatten` 클래스는 `nn.Module` 을 상속하며 `forward` 함수는 입력 `tensor`를 `flatten` 한다.
 - `nn.Sequential` 을 사용하여 two-layer fully connected network를 정의한다. 앞서 정의한 `Flatten()` 으로 입력 이미지를 `flatten` 한다. 첫 번째 FC layer의 입력 크기는 `3 * 32 * 32` 이고 출력 크기는 `hidden_layer_size` 이다. 활성화 함수를 적용한 후 두 번째 FC layer의 입력 크기는 `hidden_layer_size` 이고 출력 크기는 CIFAR-10 dataset의 클래스 수인 10이다.
 - optimizer로 `optim.SGD` 를 사용하며 Nesterov momentum을 적용한다. 마지막으로 `train_part34` 함수를 사용하여 모델을 학습시킨다.
- Three-layer ConvNet
 - `nn.Sequential` 을 사용하여 Three-layer ConvNet를 정의한다. 첫 번째 `nn.Conv2d` layer는 입력 채널 3, 출력 채널 32, 커널 크기 5x5, 패딩 2로 설정했다. 활성화 함수를 적용한 후, 두 번째 `nn.Conv2d` layer는 입력 채널 32, 출력 채널 16, 커널 크기 3x3, 패딩 1로 설정했다. 다시 활성화 함수를 적용한 후 마지막으로 `flatten` 된 출력에서 10개의 클래스에 대한 예측을 생성한다.
 - optimizer로 `optim.SGD` 를 사용하며 Nesterov momentum을 적용한다. 마지막으로 `train_part34` 함수를 사용하여 모델을 학습시킨다.

```

channel_1 = 32
channel_2 = 16
learning_rate = 1e-2

model = None
optimizer = None

# Question 6
model = nn.Sequential(
    nn.Conv2d(3, channel_1, kernel_size=5, padding=2), # 첫 번째 convolutional layer
    nn.ReLU(), # ReLU
    nn.Conv2d(channel_1, channel_2, kernel_size=3, padding=1), # 두 번째 convolutional layer
    nn.ReLU(), # ReLU
    Flatten(),
    nn.Linear(channel_2 * 32 * 32, 10) # FC layer
)

# optimizer 설정
optimizer = optim.SGD(model.parameters(), lr=learning_rate,
                       momentum=0.9, nesterov=True)

```

```
train_part34(model, optimizer)
```

`train_part34` 함수를 실행한 결과는 아래와 같다.

```
Iteration 0, loss = 2.2909
Checking accuracy on validation set
Got 115 / 1000 correct (11.50)

Iteration 100, loss = 1.3000
Checking accuracy on validation set
Got 449 / 1000 correct (44.90)

Iteration 200, loss = 1.5308
Checking accuracy on validation set
Got 458 / 1000 correct (45.80)

Iteration 300, loss = 1.4014
Checking accuracy on validation set
Got 522 / 1000 correct (52.20)

Iteration 400, loss = 1.2672
Checking accuracy on validation set
Got 493 / 1000 correct (49.30)

Iteration 500, loss = 1.7245
Checking accuracy on validation set
Got 545 / 1000 correct (54.50)

Iteration 600, loss = 1.3820
Checking accuracy on validation set
Got 556 / 1000 correct (55.60)

Iteration 700, loss = 1.1973
Checking accuracy on validation set
Got 560 / 1000 correct (56.00)

Iteration 0, loss = 1.1678
Checking accuracy on validation set
Got 551 / 1000 correct (55.10)

Iteration 100, loss = 1.1410
Checking accuracy on validation set
Got 571 / 1000 correct (57.10)

Iteration 200, loss = 1.2906
Checking accuracy on validation set
Got 560 / 1000 correct (56.00)

Iteration 300, loss = 1.1523
Checking accuracy on validation set
Got 585 / 1000 correct (58.50)

Iteration 400, loss = 1.0358
Checking accuracy on validation set
Got 591 / 1000 correct (59.10)

Iteration 500, loss = 1.0571
Checking accuracy on validation set
Got 600 / 1000 correct (60.00)

Iteration 600, loss = 0.9551
Checking accuracy on validation set
Got 587 / 1000 correct (58.70)

Iteration 700, loss = 1.2113
Checking accuracy on validation set
Got 591 / 1000 correct (59.10)
```

Part V. CIFAR-10 open-ended challenge

- CIFAR-10 validation set에서 70% 이상의 정확도를 달성하는 모델을 생성하기 위해 아래 내용을 구현했다.
 - 모델의 필터 크기를 5x5에서 3x3으로 변경하여 더 작은 필터를 사용했다.
 - max pooling을 사용하여 spatial dimension을 줄였다.
 - 두 번째 convolutional layer에서 더 많은 feature를 학습하기 위해 출력 channel 수를 64로 증가시켰다.
 - Dropout을 추가하여 과적합을 방지했다.
 - 마지막에는 두 개의 FC layer를 추가했다.


```

# Question 7
model = None
optimizer = None

model = nn.Sequential(
    # 커널 크기 3*3으로 변경
    nn.Conv2d(3, 32, 3, padding=1),
    nn.ReLU(),

    # max pooling
    nn.MaxPool2d(kernel_size=2, stride=2),

    # 두 번째 convolutional layer에서 출력 채널 수 64로 변경
    nn.Conv2d(32, 64, 3, padding=1),
    nn.ReLU(),

    # max pooling
    nn.MaxPool2d(kernel_size=2, stride=2),

    # Dropout
    nn.Dropout2d(0.2),
    nn.Flatten(),

    # FC layer
    nn.Linear(64 * 8 * 8, 512),
    nn.ReLU(),

    # Dropout
    nn.Dropout(0.5),

    # FC layer
    nn.Linear(512, 10)
)

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

train_part34(model, optimizer, epochs=10)


```

실행 결과는 아래와 같다.

```

[ ] best_model = model
    check_accuracy_part34(loader_test, best_model)

```

 Checking accuracy on test set
 Got 7562 / 10000 correct (75.62)

결론

- Barebones PyTorch는 가장 낮은 수준의 추상화로, flexibility는 높으나 코드 작성이 복잡하고 많은 시간이 소요된다.
- PyTorch Module API는 모듈화된 아키텍처를 통해 재사용성과 가독성을 높이고 복잡한 모델을 보다 쉽게 관리할 수 있게 해준다.
- PyTorch Sequential API는 가장 높은 수준의 추상화로, 매우 간단하고 빠르게 모델을 구성할 수 있으나 flexibility는 낮은 편이다.
- architectures, hyperparameters, loss functions, optimizers 등을 변경하여 모델의 정확도를 향상시킬 수 있다.

