

RNN 모델 기본 원리

■ 학습 목표

- RNN(순환 신경망)의 기본 원리를 이해하고, 데이터의 시퀀스 처리와 시간적 종속성을 모델링하는 방법을 설명할 수 있다.
- 다양한 RNN 구조 및 동작 방식을 비교 분석하고, 특히 LSTM(Long Short-Term Memory)과 GRU(Gated Recurrent Unit)의 개념과 구조적 특성을 명확히 파악할 수 있다.
- LSTM의 구성 요소(셀 상태, 게이트 메커니즘 등)에 대한 심층적인 이해를 통해, 장기 기억 문제를 해결할 수 있다.
- PyTorch를 활용하여 RNN 및 LSTM 모델을 구현하고, 기본적인 하이퍼파라미터 조정 및 성능 평가를 수행할 수 있다.

12.1 RNN(순환 신경망, Recurrent Neural Network) 원리

- RNN은 **순차적 데이터(Sequence Data)**를 처리하는 데 적합한 신경망입니다.
- 주로 시간적 연속성이 있는 데이터를 다루며, 자연어 처리(NLP), 음성 인식, 시계열 예측 등에서 널리 사용됩니다.
- RNN은 일반적인 신경망과 달리 순차적 데이터를 입력받아 이전의 출력을 다음 입력에 반영하는 구조를 가집니다.

1) RNN의 기본 원리:

- **순차적 처리** : RNN은 이전 타임스텝의 출력을 현재 타임스텝의 입력으로 사용하는 방식입니다. 이로 인해 순차적인 정보가 모델에 전달됩니다.
- **셀 상태(State)** : RNN은 각 타임스텝마다 은닉 상태(hidden state)를 업데이트하며, 이를 통해 모델은 과거의 정보를 기억합니다.
- **기울기 소실 문제(Vanishing Gradient Problem)** : RNN은 긴 시퀀스를 다룰 때 기울기 소실 문제로 인해 학습이 어려운 경우가 많습니다.

이를 해결하기 위한 개선된 모델로 LSTM(Long Short-Term Memory)과 GRU(Gated Recurrent Unit)가 있습니다.

- RNN은 현재 입력 x 와 그 이전의 상태 h 를 사용하여 현재 은닉 상태를 계산한다.

$$h_t = \sigma(W_h \cdot h_{t-1} + W_x \cdot x_t + b_h)$$

그림 12.1_RNN 은닉상태

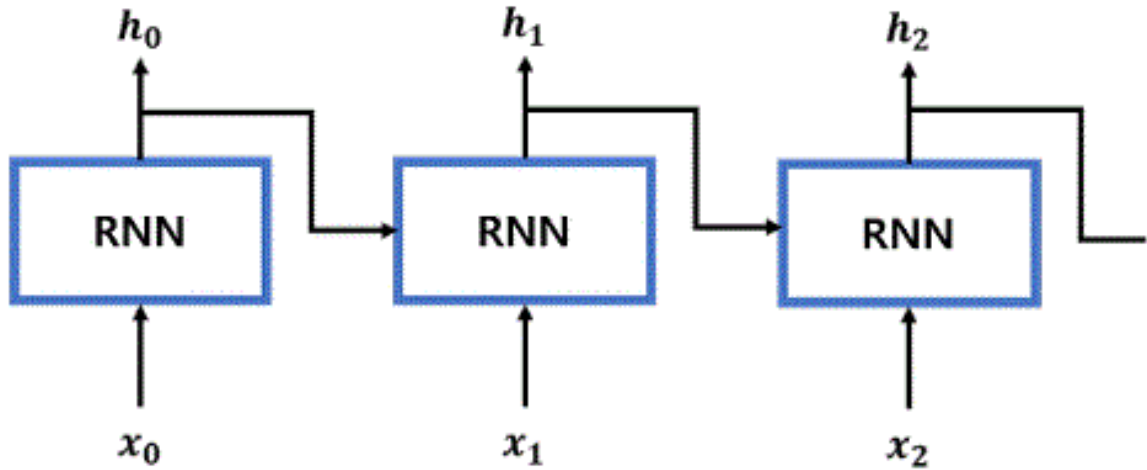


그림 12.2_RNN 순환구조

2) RNN의 한계

RNN은 시퀀스를 처리하는 데 매우 효과적이지만 다음과 같은 한계를 가지고 있습니다:

- **Vanishing Gradient Problem:** 장기 의존성을 학습하는 데 어려움을 겪는 현상으로, 기울기가 매우 작아져(사라져서) 학습이 진행되지 않거나 느려지는 문제가 발생합니다.

이는 긴 시퀀스 또는 시간적 간격의 데이터 처리 시 특히 문제됩니다.

- **Long-Term Dependencies:** 일반적인 RNN은 장기적인 의존성 정보를 잘 포착하지 못합니다.

3) LSTM(Long Short-Term Memory)

이러한 한계를 극복하기 위해 LSTM이 고안되었습니다. LSTM은 RNN의 특별한 형태로, 다음과 같은 구성 요소를 포함합니다:

- **셀 상태(Cell State):** 데이터를 장기적으로 기억할 수 있는 연속적 정보의 흐름을 제공합니다.
- **게이트(Gates):** LSTM은 입력 게이트, 삭제 게이트, 출력 게이트의 세 가지 게이트를 사용하여 정보를 조절합니다. 각각의 게이트는 다음과 같은 역할을 수행합니다:
 - **입력 게이트:** 새로운 정보가 셀 상태에 추가되어야 하는지를 결정합니다.
 - **삭제 게이트:** 셀 상태에서 어떤 정보를 삭제해야 할지를 결정합니다.
 - **출력 게이트:** 셀 상태에서 어떤 정보를 출력할지를 결정합니다.

- RNN과 LSTM은 시퀀스 데이터를 모델링하는 데 매우 유용한 도구입니다.

- RNN의 기본 원리를 이해하고, LSTM과 같은 발전된 모델의 구조와 작동 방식을 이해하면 시계열 데이터 또는 자연어 처리와 같은 복잡한 문제를 해결하는 데 큰 도움이 됩니다.
- PyTorch를 이용한 이러한 모델의 구현은 직관적이며, 특히 RNN과 LSTM의 이점을 활용하는 데 유용합니다.

12.2 RNN 모델 구현

```
In [1]: import torch
import torch.nn as nn
import torch.optim as optim
import torch.nn.functional as F
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
#torchvision: PyTorch에서 컴퓨터 비전용 데이터셋, 모델 등을 제공.
```

데이터셋 준비

- datasets.MNIST: train=True는 훈련 데이터셋, train=False는 테스트 데이터셋을 로드합니다.
- transforms.ToTensor(): 이미지를 텐서로 변환합니다.
- DataLoader: 데이터를 미니배치로 나누어 학습에 사용할 수 있게 합니다.

```
In [2]: # 데이터 전처리 설정
transform = transforms.Compose([
    transforms.ToTensor(),
    # 이미지를 텐서로 변환
    # transforms.ToTensor(): PIL 이미지나 NumPy 배열을 PyTorch 텐서로 변환.
    transforms.Normalize((0.5,),(0.5,)) # 평균과 표준편차로 정규화
])

# 데이터셋 로딩
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

# 데이터 로더 설정
train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)
```

RNN 모델 정의

```
In [3]: # 2. RNN 모델 정의
class RNN_Model(nn.Module):
    def __init__(self, input_size=28, hidden_size=128, output_size=10):
        super(RNN_Model, self).__init__()
        self.rnn = nn.RNN(input_size=input_size, hidden_size=hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # x: (batch_size, seq_len, input_size)
        rnn_out, _ = self.rnn(x) # (batch_size, seq_len, hidden_size)
        out = rnn_out[:, -1, :] # 마지막 타임스텝의 출력을 사용
        out = self.fc(out) # FC 층을 통해 최종 출력
        return out
```

```
# 3. 모델 초기화
model = RNN_Model()
```

- nn.RNN: 기본 RNN 레이어입니다. input_size=28 (각 픽셀), hidden_size=128 (RNN의 은닉 상태 차원), batch_first=True는 입력 텐서의 첫 번째 차원이 배치 크기임을 나타냅니다.
- forward: rnn_out[:, -1, :]는 시퀀스의 마지막 타임스텝에서 나온 출력을 사용합니다. 이는 MNIST의 숫자가 하나의 시퀀스로 간주되어 각 픽셀을 하나의 타임스텝으로 처리하기 때문입니다.
- fc: RNN의 출력을 10개의 클래스(0~9)로 변환하는 완전 연결층입니다.

```
In [4]: # 4. 손실 함수와 최적화 알고리즘
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- 학습 과정에서는 각 배치에 대해 RNN 모델을 사용해 예측을 하고, CrossEntropyLoss를 사용해 손실을 계산합니다.
- optimizer.step()을 통해 파라미터를 업데이트합니다.
- 매 에폭마다 훈련 데이터에서 손실 값과 정확도를 출력합니다.

```
In [5]: # 5. 모델 학습
num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for inputs, labels in train_loader:
        inputs, labels = inputs.squeeze(1), labels # (batch_size, 28, 28)
        inputs = inputs.view(-1, 28, 28) # (batch_size, seq_len=28, input_size=28)

        optimizer.zero_grad()
        outputs = model(inputs) # RNN 모델을 통해 출력값 계산
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader)}, Acc: {correct/total}")
```

```
Epoch [1/5], Loss: 0.6883858603391566, Accuracy: 77.27%
Epoch [2/5], Loss: 0.29793344690664997, Accuracy: 91.21%
Epoch [3/5], Loss: 0.22795759944288907, Accuracy: 93.30666666666667%
Epoch [4/5], Loss: 0.1945181574040988, Accuracy: 94.315%
Epoch [5/5], Loss: 0.16966787344817794, Accuracy: 95.115%
```

```
In [6]: # 6. 모델 평가
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for inputs, labels in test_loader:
```

```

inputs, labels = inputs.squeeze(1), labels
inputs = inputs.view(-1, 28, 28) # (batch_size, seq_len=28, input_size=28)
outputs = model(inputs)
_, predicted = torch.max(outputs, 1)
correct += (predicted == labels).sum().item()
total += labels.size(0)

print(f"Test Accuracy: {100 * correct / total}%")

# model.eval()은 평가 모드로 설정하여 dropout 같은 레이어를 비활성화합니다.
# 테스트 데이터에 대해 예측을 수행하고 정확도를 계산합니다.

```

Test Accuracy: 95.27%

예측 결과 시각화

생각해 보기

1. NumPy 배열을 PyTorch 텐서로 변환: img가 numpy.ndarray이므로, 이를 torch.tensor(img).float()로 변환하여 PyTorch 텐서로 만듭니다. float()는 텐서의 타입을 float32로 변환하는 것입니다. 만약 MNIST 이미지가 이미 torch.Tensor라면 이 변환은 필요 없습니다.
2. unsqueeze() 사용: img_tensor.unsqueeze(0)를 사용하여 배치 차원을 추가합니다. unsqueeze(0)은 텐서 앞에 하나의 차원을 추가하여 (1, 28, 28) 형태로 만듭니다. 이렇게 해야 RNN 모델에 입력으로 사용할 수 있습니다.
3. img 시각화: plt.imshow(img, cmap='gray')를 사용하여 이미지를 시각화합니다. img는 이미 (28, 28) 크기이므로 이 부분은 numpy 배열을 그대로 사용합니다.

```

In [7]: # 7. 예측 결과 시각화
import torch
import random
import matplotlib.pyplot as plt

# 테스트 데이터에서 랜덤 샘플을 선택
model.eval()
images, labels = next(iter(test_loader))
random_idx = random.randint(0, len(images) - 1)

# 랜덤 이미지와 레이블
img = images[random_idx]
label = labels[random_idx]

# 이미지를 원본 형태로 변환 (채널 순서 바꾸기, MNIST는 (1,28,28))
img = img.squeeze(0).numpy() # (28, 28)

# NumPy 배열을 PyTorch 텐서로 변환
# img는 numpy.ndarray이므로 이를 torch.tensor(img)로 변환하고 float()로 형 변환합니다.
img_tensor = torch.tensor(img).float()

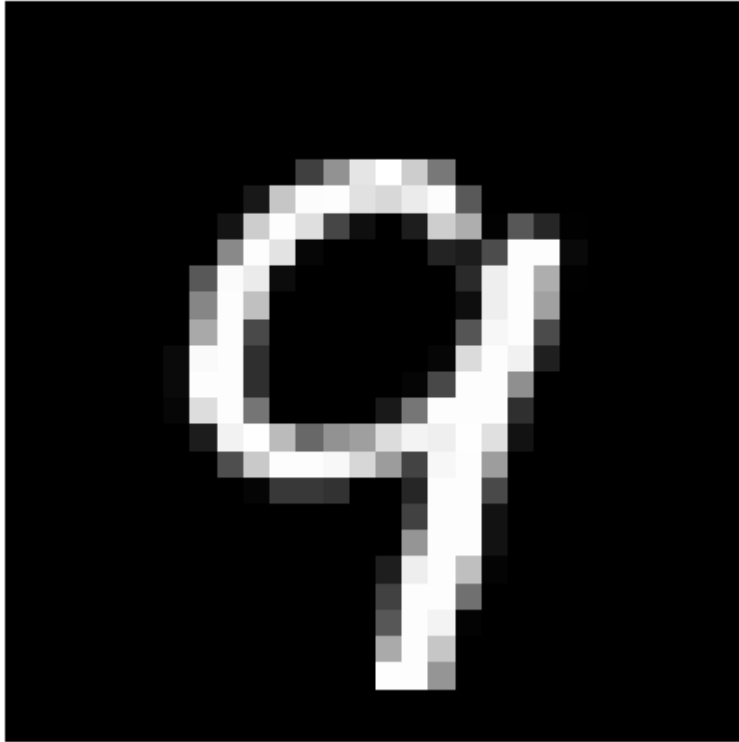
# 모델 예측
# unsqueeze() 메서드는 PyTorch 텐서에만 존재하므로, numpy.ndarray를 PyTorch 텐서로 변환
output = model(img_tensor.unsqueeze(0)) # unsqueeze로 배치 차원 추가, (1, 28, 28)
_, predicted_label = torch.max(output, 1)

# 예측 결과 시각화
plt.imshow(img, cmap='gray')
plt.title(f"True Label: {label}, Predicted: {predicted_label.item()}")

```

```
plt.axis('off')
plt.show()
```

True Label: 9, Predicted: 9



12.3 LSTM(Long Short-Term Memory)

LSTM(Long Short-Term Memory) 는 순환 신경망(RNN : Recurrent Neural Network)의 일종으로, **장기적인 의존 관계(장기적인 종속성 : long-term dependencies)** 를 더 잘 처리할 수 있도록 설계된 아키텍처입니다. 기존의 RNN의 발전형으로 장기 기억 문제(long-term dependencies 문제)와 기울기 소실(vanishing gradient) 문제를 해결하기 위해 고안된 것입니다. LSTM은 1997년 Sepp Hochreiter와 Jürgen Schmidhuber에 의해 처음 제안되었습니다.

1) LSTM(Long Short-Term Memory) 구성 요소

- **셀 상태 (Cell State):**
 - 장기 메모리 역할을 하며, 시퀀스 데이터를 처리하는 동안 정보를 전송합니다.
 - 셀 상태는 시퀀스의 모든 타임 스텝에서 정보를 유지하고 업데이트할 수 있어, 장기 의존성을 잘 처리할 수 있도록 도와줍니다.
- **입력 게이트 (Input Gate):**
 - 현재 입력 정보를 셀 상태에 얼마나 추가할지를 결정합니다.
 - 이 게이트는 현재 입력 x_{t_txt} 와 이전 은닉 상태 h_{t-1} 에 기반하여 새로운 정보를 얼마나 반영할 것인지를 학습하며, 이를 통해 새로운 정보가 셀 상태에 추가됩니다.
- **망각 게이트 (Forget Gate):**
 - 이전 셀 상태에서 어떤 정보를 삭제할지를 결정합니다.
 - 이 게이트는 이전 은닉 상태와 현재 입력을 기반으로 작동하여, 불필요한 정보를 선택적으로 잊어버리는데 도움을 줍니다. 이를 통해 모델은 유용한 정보만 유지할 수 있습니다.

니다.

- **출력 게이트 (Output Gate):**

- 최종 출력을 계산할 때 셀 상태를 얼마나 반영할지를 결정합니다.
- LSTM의 출력은 셀 상태와 이 게이트의 조합으로 만들어지며, 출력을 결정할 때 현재 셀 상태의 정보를 얼마나 반영할지를 조절합니다. 이 과정에서 시퀀스의 의미 있는 정보를 다음 단계로 전달합니다.

2) LSTM의 동작 과정

LSTM의 각 타임 스텝에서는 위에서 정의한 게이트를 사용하여 다음과 같은 과정을 따릅니다:

- **입력 게이트**는 현재 입력 및 이전 은닉 상태에 기반하여 새로운 데이터를 셀 상태에 얼마나 반영할지 결정합니다.
- **삭제 게이트**는 이전 셀 상태에서 어떤 정보를 잃어버릴지를 결정합니다.
- **셀 상태**는 새로 입력된 정보와 이전 상태 정보를 결합하여 업데이트됩니다.
- **출력 게이트**는 업데이트된 셀 상태를 기반으로 출력할 정보를 결정합니다.

3) LSTM의 주요 장점

- **장기 의존성 문제 해결:** LSTM은 긴 시퀀스를 처리할 수 있는 능력이 뛰어나며, 정보를 오랜 시간 동안 유지할 수 있습니다.
- **기울기 소실 문제 완화:** LSTM의 구조는 기울기 소실 문제를 완화하여 좀 더 안정적인 학습을 가능하게 합니다.
- **유연성:** LSTM은 텍스트, 음성 인식, 시계열 데이터 분석 등 다양한 시퀀스 데이터에 널리 적용됩니다.

4) LSTM의 사용 예

RNN 또는 LSTM은 다음과 같은 다양한 분야에서 많이 사용됩니다:

- **자연어 처리 (NLP):** 문장의 문맥을 이해하고 다음 단어를 예측하는 데 사용됩니다.
- **음성 인식:** 주어진 음성 데이터를 텍스트로 변환하는 모델에서 사용됩니다.
- **시계열 예측:** 주식 시장 예측과 같은 연속적인 데이터를 분석할 때도 사용됩니다.

LSTM을 이용한 예제 : MNIST 숫자 분류

- 모델 정의 생각해 보기
- 입력 크기: MNIST 이미지의 각 행은 28개의 픽셀로 구성되어 있으므로, 각 입력 벡터의 크기는 28입니다.
- LSTM 층: LSTM은 input_size=28과 hidden_size=128로 설정됩니다. batch_first=True는 입력 텐서의 차원이 (batch_size, seq_len, input_size)임을 의미합니다.
- Fully Connected Layer: LSTM의 출력을 입력받아 최종 10개의 클래스 중 하나를 예측합니다.

1) 라이브러리 임포트 및 데이터 로드

```
In [8]: import torch
import torch.nn as nn
```

```

import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# 장치 설정
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 데이터 변환 설정
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# MNIST 데이터셋 로드
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)

```

2) LSTM 모델 정의

```

In [9]: # LSTM 모델 정의
class LSTMModel(nn.Module):
    def __init__(self, input_size=28, hidden_size=128, output_size=10):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), 28, 28) # (batch_size, seq_len, input_size)
        out, _ = self.lstm(x) # LSTM 처리
        out = self.fc(out[:, -1, :]) # 마지막 시퀀스의 출력
        return out

```

3) 손실 함수 및 옵티마이저 설정

```

In [10]: # 모델, 손실 함수 및 옵티마이저 설정
model = LSTMModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

```

4) 훈련 및 평가 함수 정의

```

In [11]: # 손실과 정확도를 기록할 리스트 초기화
train_losses = []
test_losses = []
accuracies = []

# 훈련 및 평가 함수 정의
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    total_loss = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)

```



```

optimizer.zero_grad() # 기울기 초기화
output = model(data) # 예측
loss = criterion(output, target) # 손실 계산
loss.backward() # 역전파
optimizer.step() # 가중치 업데이트

total_loss += loss.item()

if batch_idx % 100 == 0:
    print(f'Train Epoch: {epoch} [{batch_idx * len(data)} / {len(train_loader.dataset)}]
          ({100. * batch_idx / len(train_loader):.0f}%) \t Loss: {loss.item():.4f}')

return total_loss / len(train_loader) # 평균 손실 반환

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item() * data.size(0)
            # 배치 손실 합산

            pred = output.argmax(dim=1, keepdim=True)
            # 가장 높은 값을 가진 클래스 선택
            correct += pred.eq(target.view_as(pred)).sum().item()

    test_loss /= len(test_loader.dataset) # 평균 손실 계산
    accuracy = 100. * correct / len(test_loader.dataset) # 정확도 계산
    print(f'\nTest set: Average loss: {test_loss:.4f}, Accuracy: {correct}/{len(test_loader.dataset)}
          ({accuracy:.2f}%) \n')

    return test_loss, accuracy # 손실과 정확도 반환

```

5) 모델 훈련 실행

```

In [12]: # 모델 훈련 및 평가 실행
num_epochs = 5
for epoch in range(1, num_epochs + 1):
    train_loss = train(model, device, train_loader, optimizer, epoch)
    test_loss, accuracy = test(model, device, test_loader)

    # 손실과 정확도 기록
    train_losses.append(train_loss)
    test_losses.append(test_loss)
    accuracies.append(accuracy)

```

Train Epoch: 1 [0/60000 (0%)] Loss: 2.329484
Train Epoch: 1 [6400/60000 (11%)] Loss: 0.851197
Train Epoch: 1 [12800/60000 (21%)] Loss: 0.296403
Train Epoch: 1 [19200/60000 (32%)] Loss: 0.226675
Train Epoch: 1 [25600/60000 (43%)] Loss: 0.397131
Train Epoch: 1 [32000/60000 (53%)] Loss: 0.133969
Train Epoch: 1 [38400/60000 (64%)] Loss: 0.028770
Train Epoch: 1 [44800/60000 (75%)] Loss: 0.067647
Train Epoch: 1 [51200/60000 (85%)] Loss: 0.111540
Train Epoch: 1 [57600/60000 (96%)] Loss: 0.169929

Test set: Average loss: 0.1205, Accuracy: 9650/10000 (96.50%)

Train Epoch: 2 [0/60000 (0%)] Loss: 0.047304
Train Epoch: 2 [6400/60000 (11%)] Loss: 0.129161
Train Epoch: 2 [12800/60000 (21%)] Loss: 0.171918
Train Epoch: 2 [19200/60000 (32%)] Loss: 0.309996
Train Epoch: 2 [25600/60000 (43%)] Loss: 0.119627
Train Epoch: 2 [32000/60000 (53%)] Loss: 0.025934
Train Epoch: 2 [38400/60000 (64%)] Loss: 0.168841
Train Epoch: 2 [44800/60000 (75%)] Loss: 0.112262
Train Epoch: 2 [51200/60000 (85%)] Loss: 0.113027
Train Epoch: 2 [57600/60000 (96%)] Loss: 0.010328

Test set: Average loss: 0.1018, Accuracy: 9700/10000 (97.00%)

Train Epoch: 3 [0/60000 (0%)] Loss: 0.046774
Train Epoch: 3 [6400/60000 (11%)] Loss: 0.069691
Train Epoch: 3 [12800/60000 (21%)] Loss: 0.144629
Train Epoch: 3 [19200/60000 (32%)] Loss: 0.139346
Train Epoch: 3 [25600/60000 (43%)] Loss: 0.022870
Train Epoch: 3 [32000/60000 (53%)] Loss: 0.043247
Train Epoch: 3 [38400/60000 (64%)] Loss: 0.069910
Train Epoch: 3 [44800/60000 (75%)] Loss: 0.117827
Train Epoch: 3 [51200/60000 (85%)] Loss: 0.093206
Train Epoch: 3 [57600/60000 (96%)] Loss: 0.028465

Test set: Average loss: 0.0988, Accuracy: 9716/10000 (97.16%)

Train Epoch: 4 [0/60000 (0%)] Loss: 0.043301
Train Epoch: 4 [6400/60000 (11%)] Loss: 0.012200
Train Epoch: 4 [12800/60000 (21%)] Loss: 0.163648
Train Epoch: 4 [19200/60000 (32%)] Loss: 0.069388
Train Epoch: 4 [25600/60000 (43%)] Loss: 0.022740
Train Epoch: 4 [32000/60000 (53%)] Loss: 0.058587
Train Epoch: 4 [38400/60000 (64%)] Loss: 0.017100
Train Epoch: 4 [44800/60000 (75%)] Loss: 0.040936
Train Epoch: 4 [51200/60000 (85%)] Loss: 0.020974
Train Epoch: 4 [57600/60000 (96%)] Loss: 0.013834

Test set: Average loss: 0.0712, Accuracy: 9788/10000 (97.88%)

Train Epoch: 5 [0/60000 (0%)] Loss: 0.071904
Train Epoch: 5 [6400/60000 (11%)] Loss: 0.039439
Train Epoch: 5 [12800/60000 (21%)] Loss: 0.023377
Train Epoch: 5 [19200/60000 (32%)] Loss: 0.042095
Train Epoch: 5 [25600/60000 (43%)] Loss: 0.073016
Train Epoch: 5 [32000/60000 (53%)] Loss: 0.010706
Train Epoch: 5 [38400/60000 (64%)] Loss: 0.122148
Train Epoch: 5 [44800/60000 (75%)] Loss: 0.131820

Train Epoch: 5 [51200/60000 (85%)] Loss: 0.046071
Train Epoch: 5 [57600/60000 (96%)] Loss: 0.006616

Test set: Average loss: 0.0571, Accuracy: 9831/10000 (98.31%)

```
In [13]: # 훈련 손실, 테스트 손실, 정확도 리스트 출력
print("\nTraining Losses:", train_losses)
print("Test Losses:", test_losses)
print("Accuracies:", accuracies)
```

Training Losses: [0.40835502779464733, 0.11183679930226349, 0.07967761785103711, 0.06302302710707984, 0.05275903610256451]
Test Losses: [0.12046639200150967, 0.10176129299402237, 0.09880621223822236, 0.07115173072069883, 0.057064132483676075]
Accuracies: [96.5, 97.0, 97.16, 97.88, 98.31]

6) MNIST 데이터셋을 LSTM 모델로 학습한 후, 훈련 손실, 테스트 손실 및 정확도를 시각화

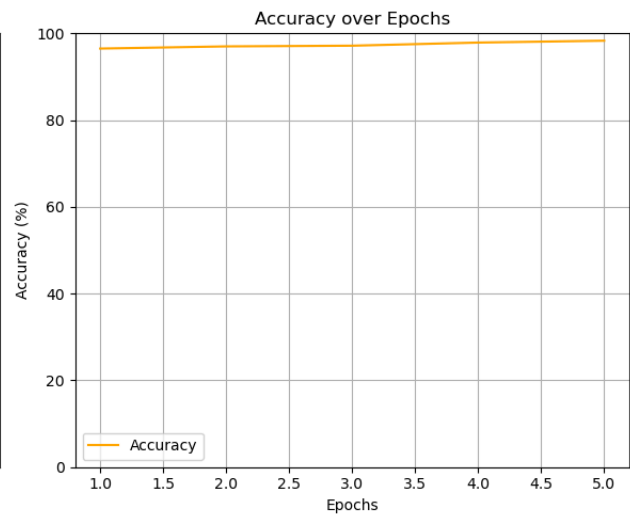
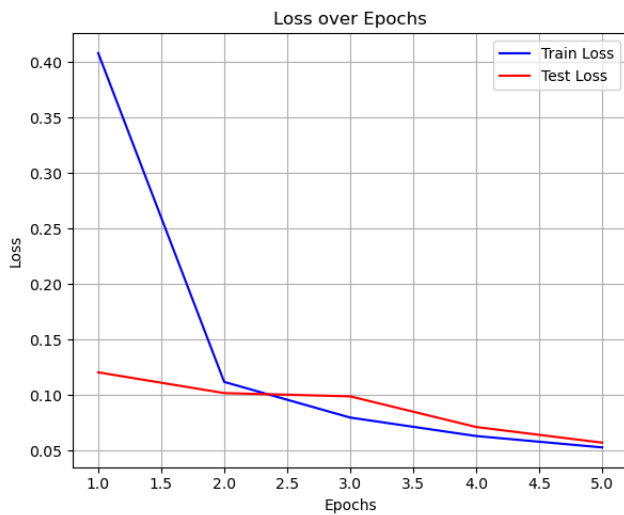
```
In [14]: # 시각화 코드
# 결과 시각화
plt.figure(figsize=(12, 5))

# 손실 그래프
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss', color='blue')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss', color='red')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()

# 정확도 그래프
plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), accuracies, label='Accuracy', color='orange')
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.ylim(0, 100) # Y축 범위 설정

plt.legend()
plt.grid()

plt.tight_layout()
plt.show() # 그래프 출력
```



```
In [15]: import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt

# 장치 설정
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 데이터 변환 설정
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# MNIST 데이터셋 로드
train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)

# LSTM 모델 정의
class LSTMModel(nn.Module):
    def __init__(self, input_size=28, hidden_size=128, output_size=10):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), 28, 28) # (batch_size, seq_len, input_size)
        out, _ = self.lstm(x) # LSTM 처리
        out = self.fc(out[:, -1, :]) # 마지막 시퀀스의 출력
        return out

# 모델, 손실 함수 및 옵티마이저 설정
model = LSTMModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# 손실과 정확도를 기록할 리스트 초기화
train_losses = []
test_losses = []
```

```

accuracies = []

# 훈련 및 평가 함수 정의
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    total_loss = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad() # 기울기 초기화
        output = model(data) # 예측
        loss = criterion(output, target) # 손실 계산
        loss.backward() # 역전파
        optimizer.step() # 가중치 업데이트

        total_loss += loss.item()

        if batch_idx % 100 == 0:
            print(f'Train Epoch: {epoch} [{batch_idx * len(data)} / {len(train_loader.dataset)}]
                  ({100. * batch_idx / len(train_loader):.0f}%) \t Loss: {loss.item():.4f}')

    average_loss = total_loss / len(train_loader) # 평균 손실 반환
    return average_loss

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item() * data.size(0)
            # 배치 손실 합산
            pred = output.argmax(dim=1, keepdim=True)
            # 가장 높은 값을 가진 클래스 선택
            correct += pred.eq(target.view_as(pred)).sum().item()

    average_loss = test_loss / len(test_loader.dataset) # 평균 손실 계산
    accuracy = 100. * correct / len(test_loader.dataset) # 정확도 계산
    print(f'Test set: Average loss: {average_loss:.4f}, Accuracy: {correct}/{len(test_loader.dataset)}
          ({accuracy:.2f}%) \n')

    return average_loss, accuracy # 손실과 정확도 반환

# 모델 훈련 및 평가 실행
num_epochs = 5
for epoch in range(1, num_epochs + 1):
    train_loss = train(model, device, train_loader, optimizer, epoch)
    test_loss, accuracy = test(model, device, test_loader)

    # 손실과 정확도 기록
    train_losses.append(train_loss)
    test_losses.append(test_loss)
    accuracies.append(accuracy)

# 훈련 손실, 테스트 손실, 정확도 리스트 출력
print("\nTraining Losses:", train_losses)
print("Test Losses:", test_losses)
print("Accuracies:", accuracies)

```

```
# 시각화 코드
# 결과 시각화
plt.figure(figsize=(12, 5))

# 손실 그래프
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss', color='blue')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss', color='red')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()

# 정확도 그래프
plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), accuracies, label='Accuracy', color='orange')
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.ylim(0, 100) # Y축 범위 설정

plt.legend()
plt.grid()

plt.tight_layout()
plt.show() # 그래프 출력
```

Train Epoch: 1 [0/60000 (0%)] Loss: 2.305958
Train Epoch: 1 [6400/60000 (11%)] Loss: 0.654900
Train Epoch: 1 [12800/60000 (21%)] Loss: 0.548571
Train Epoch: 1 [19200/60000 (32%)] Loss: 0.291806
Train Epoch: 1 [25600/60000 (43%)] Loss: 0.369903
Train Epoch: 1 [32000/60000 (53%)] Loss: 0.204993
Train Epoch: 1 [38400/60000 (64%)] Loss: 0.107359
Train Epoch: 1 [44800/60000 (75%)] Loss: 0.164750
Train Epoch: 1 [51200/60000 (85%)] Loss: 0.068103
Train Epoch: 1 [57600/60000 (96%)] Loss: 0.250978
Test set: Average loss: 0.1494, Accuracy: 9571/10000 (95.71%)

Train Epoch: 2 [0/60000 (0%)] Loss: 0.187250
Train Epoch: 2 [6400/60000 (11%)] Loss: 0.227676
Train Epoch: 2 [12800/60000 (21%)] Loss: 0.076466
Train Epoch: 2 [19200/60000 (32%)] Loss: 0.096363
Train Epoch: 2 [25600/60000 (43%)] Loss: 0.187040
Train Epoch: 2 [32000/60000 (53%)] Loss: 0.089352
Train Epoch: 2 [38400/60000 (64%)] Loss: 0.217397
Train Epoch: 2 [44800/60000 (75%)] Loss: 0.078168
Train Epoch: 2 [51200/60000 (85%)] Loss: 0.117527
Train Epoch: 2 [57600/60000 (96%)] Loss: 0.045106
Test set: Average loss: 0.0996, Accuracy: 9706/10000 (97.06%)

Train Epoch: 3 [0/60000 (0%)] Loss: 0.066594
Train Epoch: 3 [6400/60000 (11%)] Loss: 0.111761
Train Epoch: 3 [12800/60000 (21%)] Loss: 0.046905
Train Epoch: 3 [19200/60000 (32%)] Loss: 0.077822
Train Epoch: 3 [25600/60000 (43%)] Loss: 0.047518
Train Epoch: 3 [32000/60000 (53%)] Loss: 0.087108
Train Epoch: 3 [38400/60000 (64%)] Loss: 0.070632
Train Epoch: 3 [44800/60000 (75%)] Loss: 0.039836
Train Epoch: 3 [51200/60000 (85%)] Loss: 0.041987
Train Epoch: 3 [57600/60000 (96%)] Loss: 0.045100
Test set: Average loss: 0.0766, Accuracy: 9774/10000 (97.74%)

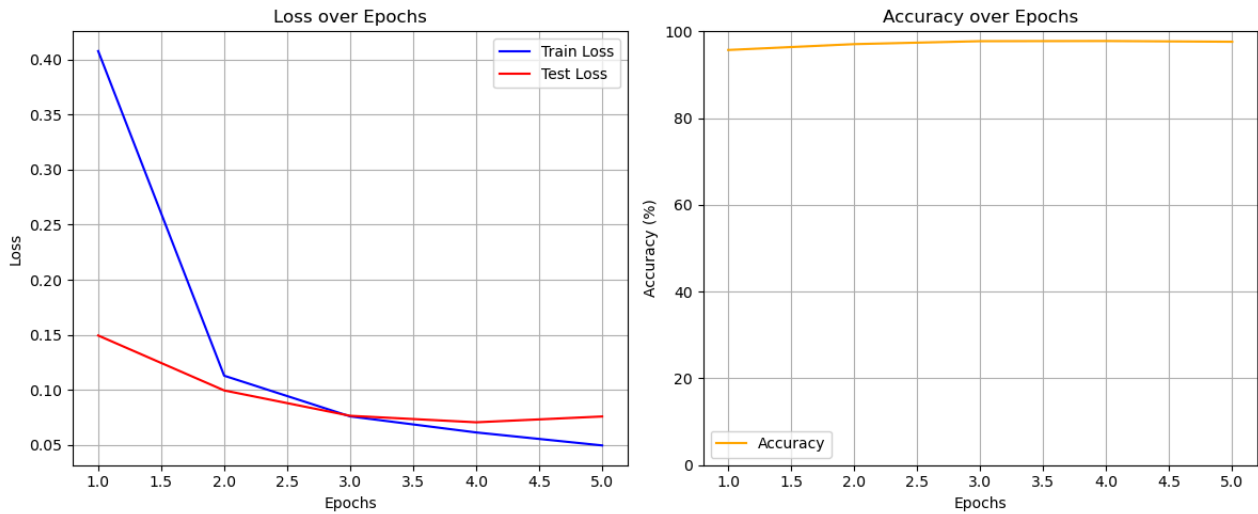
Train Epoch: 4 [0/60000 (0%)] Loss: 0.118796
Train Epoch: 4 [6400/60000 (11%)] Loss: 0.106063
Train Epoch: 4 [12800/60000 (21%)] Loss: 0.092581
Train Epoch: 4 [19200/60000 (32%)] Loss: 0.016419
Train Epoch: 4 [25600/60000 (43%)] Loss: 0.065028
Train Epoch: 4 [32000/60000 (53%)] Loss: 0.044546
Train Epoch: 4 [38400/60000 (64%)] Loss: 0.004954
Train Epoch: 4 [44800/60000 (75%)] Loss: 0.046814
Train Epoch: 4 [51200/60000 (85%)] Loss: 0.234569
Train Epoch: 4 [57600/60000 (96%)] Loss: 0.053477
Test set: Average loss: 0.0707, Accuracy: 9778/10000 (97.78%)

Train Epoch: 5 [0/60000 (0%)] Loss: 0.089244
Train Epoch: 5 [6400/60000 (11%)] Loss: 0.004302
Train Epoch: 5 [12800/60000 (21%)] Loss: 0.017235
Train Epoch: 5 [19200/60000 (32%)] Loss: 0.029997
Train Epoch: 5 [25600/60000 (43%)] Loss: 0.190982
Train Epoch: 5 [32000/60000 (53%)] Loss: 0.060044
Train Epoch: 5 [38400/60000 (64%)] Loss: 0.021332
Train Epoch: 5 [44800/60000 (75%)] Loss: 0.008066
Train Epoch: 5 [51200/60000 (85%)] Loss: 0.015250
Train Epoch: 5 [57600/60000 (96%)] Loss: 0.064715
Test set: Average loss: 0.0759, Accuracy: 9761/10000 (97.61%)

Training Losses: [0.4075658511418079, 0.11293299883298243, 0.0760167052818915, 0.06143976833551193, 0.04967013183685302]

Test Losses: [0.14942507848143577, 0.09956959730684757, 0.07655324637908488, 0.07066255191452801, 0.07592938072010874]

Accuracies: [95.71, 97.06, 97.74, 97.78, 97.61]



```
In [16]: # 1. 모델 구현
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# 데이터셋 로딩 및 전처리
# 데이터셋 로딩: datasets.MNIST를 사용하여 MNIST 데이터를 다운로드하고, transform을 통해
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# LSTM 모델 정의
class LSTM_Model(nn.Module):
    def __init__(self, input_size=28, hidden_size=128, output_size=10):
        super(LSTM_Model, self).__init__()
        self.hidden_size = hidden_size

        # LSTM 층 정의
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)

        # Fully Connected Layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # LSTM의 입력은 (batch_size, seq_len, input_size)
        lstm_out, (hn, cn) = self.lstm(x) # lstm_out: (batch_size, seq_len, hidden_size)

        # LSTM의 마지막 타임스텝 출력만 사용
        out = self.fc(lstm_out[:, -1, :]) # 마지막 타임스텝의 출력
        return out
```



```
# 모델, 손실 함수, 최적화 함수 정의
model = LSTM_Model(input_size=28, hidden_size=128, output_size=10)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

- 학습 과정에서 각 배치마다 순전파와 역전파를 진행합니다.
- optimizer.zero_grad()는 이전 그래디언트를 초기화하고, optimizer.step()으로 파라미터를 업데이트합니다.

```
In [17]: # 2. 모델 학습
num_epochs = 5
for epoch in range(num_epochs):
    model.train()
    running_loss = 0.0
    correct = 0
    total = 0
    for images, labels in train_loader:
        images = images.squeeze(1) # (batch_size, 28, 28) 형태로 변경
        images = images.permute(0, 2, 1) # (batch_size, 28, 28) -> (batch_size, seq_L

        optimizer.zero_grad()
        outputs = model(images)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

        running_loss += loss.item()
        _, predicted = torch.max(outputs, 1)
        correct += (predicted == labels).sum().item()
        total += labels.size(0)

    # 에포크마다 손실과 정확도 출력
    print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}")

# 모델 평가
model.eval()
correct = 0
total = 0
with torch.no_grad(): # torch.no_grad()는 평가할 때 불필요한 그래디언트 계산을 방지함
    for images, labels in test_loader:
        images = images.squeeze(1)
        images = images.permute(0, 2, 1)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

print(f"Test Accuracy: {100 * correct / total:.2f}%")
```

```
Epoch [1/5], Loss: 0.4442, Accuracy: 85.60%
Epoch [2/5], Loss: 0.1200, Accuracy: 96.45%
Epoch [3/5], Loss: 0.0825, Accuracy: 97.52%
Epoch [4/5], Loss: 0.0646, Accuracy: 98.03%
Epoch [5/5], Loss: 0.0508, Accuracy: 98.44%
Test Accuracy: 98.36%
```

모델 훈련 과정 시각화

```
In [18]: import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from torchvision import datasets, transforms
import matplotlib.pyplot as plt

# 데이터셋 로딩 및 전처리
transform = transforms.Compose([transforms.ToTensor(), transforms.Normalize((0.5,), (0.5,))])

train_dataset = datasets.MNIST(root='./data', train=True, download=True, transform=transform)
test_dataset = datasets.MNIST(root='./data', train=False, download=True, transform=transform)

train_loader = DataLoader(train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(test_dataset, batch_size=64, shuffle=False)

# LSTM 모델 정의
class LSTM_Model(nn.Module):
    def __init__(self, input_size=28, hidden_size=128, output_size=10):
        super(LSTM_Model, self).__init__()
        self.hidden_size = hidden_size

        # LSTM 층 정의
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)

        # Fully Connected Layer
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        # LSTM의 입력은 (batch_size, seq_len, input_size)
        lstm_out, (hn, cn) = self.lstm(x) # lstm_out: (batch_size, seq_len, hidden_size)

        # LSTM의 마지막 타임스텝 출력만 사용
        out = self.fc(lstm_out[:, -1, :]) # 마지막 타임스텝의 출력
        return out

# 모델, 손실 함수, 최적화 함수 정의
model = LSTM_Model(input_size=28, hidden_size=128, output_size=10)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)
```

```
In [19]: # 학습 루프
num_epochs = 5 # 훈련할 에폭(epoch) 수
train_losses = [] # 각 에폭마다의 훈련 손실을 저장할 리스트
train_accuracies = [] # 각 에폭마다의 훈련 정확도를 저장할 리스트
test_accuracies = [] # 각 에폭마다의 테스트 정확도를 저장할 리스트

# 에폭 수만큼 반복
for epoch in range(num_epochs):
    model.train() # 모델을 훈련 모드로 설정 (드롭아웃, 배치 정규화 등 활성화)
    running_loss = 0.0 # 현재 에폭에서의 손실을 누적할 변수
    correct = 0 # 정확한 예측의 수를 세기 위한 변수
    total = 0 # 전체 예측의 수를 세기 위한 변수

    for images, labels in train_loader:
        images = images.squeeze(1) # (batch_size, 28, 28) 형태로 변경
```

```

images = images.permute(0, 2, 1)
# (batch_size, 28, 28) -> (batch_size, seq_len=28, input_size=28)

optimizer.zero_grad()
outputs = model(images)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()

running_loss += loss.item()
_, predicted = torch.max(outputs, 1)
correct += (predicted == labels).sum().item()
total += labels.size(0)

# Epoch마다 Loss와 accuracy 출력
train_losses.append(running_loss / len(train_loader))
train_accuracies.append(100 * correct / total)
print(f"Epoch [{epoch+1}/{num_epochs}], Loss: {running_loss/len(train_loader):.4f}")

# 모델 평가 (테스트 정확도 계산)
model.eval()
correct = 0
total = 0
with torch.no_grad():
    for images, labels in test_loader:
        images = images.squeeze(1)
        images = images.permute(0, 2, 1)

        outputs = model(images)
        _, predicted = torch.max(outputs, 1)

        total += labels.size(0)
        correct += (predicted == labels).sum().item()

# 전체 테스트 데이터에 대해 모델의 정확도를 계산 (백분율로 표현)
# 계산된 테스트 정확도를 test_accuracies 리스트에 추가
test_accuracy = 100 * correct / total
test_accuracies.append(test_accuracy)
print(f"Test Accuracy: {test_accuracy:.2f}%")

```

```

Epoch [1/5], Loss: 0.4343, Accuracy: 86.18%
Test Accuracy: 96.08%
Epoch [2/5], Loss: 0.1214, Accuracy: 96.46%
Test Accuracy: 97.24%
Epoch [3/5], Loss: 0.0837, Accuracy: 97.51%
Test Accuracy: 97.61%
Epoch [4/5], Loss: 0.0665, Accuracy: 98.02%
Test Accuracy: 97.92%
Epoch [5/5], Loss: 0.0538, Accuracy: 98.36%
Test Accuracy: 98.00%

```

In [20]:

```

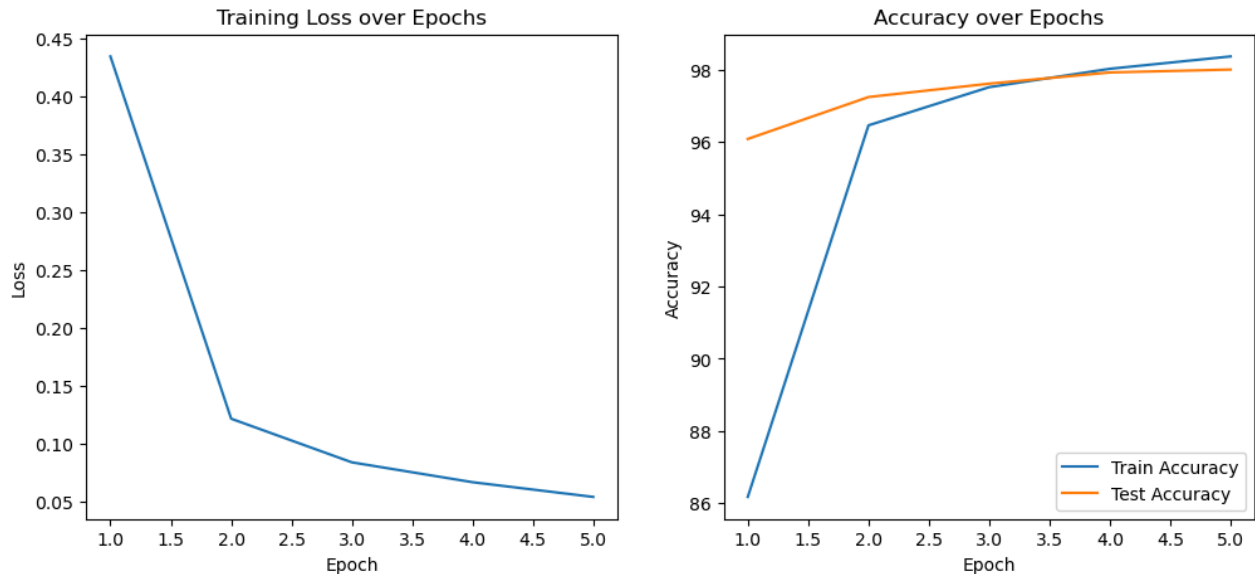
# 학습 과정 시각화
plt.figure(figsize=(12, 5))

# Loss 그래프
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label="Train Loss")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.title("Training Loss over Epochs")

```

```
# Accuracy 그래프
plt.subplot(1, 2, 2)
plt.plot(range(1, num_epochs + 1), train_accuracies, label="Train Accuracy")
plt.plot(range(1, num_epochs + 1), test_accuracies, label="Test Accuracy")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.title("Accuracy over Epochs")
plt.legend()

plt.show()
```



- 그래프를 보면 Loss는 점점 줄어듦, Test 정확도는 Train 정확도 보다 낮게 표시되었다.

Simple CNN Classifying Fashion MNIST

- Simple Convolutional Neural Network (CNN) with 3 convolutions and 4 fully connected layers working to classify clothing from Fashion MNIST;
- Dataset with 70,000 instances and 784 pixel columns. 60,000 for training, 10,000 for testing;
- Every image is in grayscale, 28x28 and belong to one of the 10 clothing classes:
 - Class 0 => T-shirt
 - Class 1 => Trouser
 - Class 2 => Pullover
 - Class 3 => Dress
 - Class 4 => Coat
 - Class 5 => Sandal
 - Class 6 => Shirt
 - Class 7 => Sneaker
 - Class 8 => Bag
 - Class 9 => Ankle boot
- CNN is trained from scratch. No transfer learning is performed;

- Dataset version on Kaggle: [Fashion MNIST](#).

간단한 CNN으로 패션 MNIST 분류하기

- 3개의 합성곱 층과 4개의 완전 연결 층으로 구성된 간단한 합성곱 신경망(CNN)으로 패션 MNIST에서 의류를 분류합니다.
- 데이터셋은 70,000개의 인스턴스와 784개의 픽셀 열로 구성되어 있습니다. 이 중 60,000개는 훈련용, 10,000개는 테스트용입니다.
- 모든 이미지는 그레이스케일이며 크기는 28x28이며 10개의 의류 클래스 중 하나에 속합니다:
 - 클래스 0 => 티셔츠
 - 클래스 1 => 바지
 - 클래스 2 => 풀오버
 - 클래스 3 => 드레스
 - 클래스 4 => 코트
 - 클래스 5 => 샌달
 - 클래스 6 => 셔츠
 - 클래스 7 => 스니커즈
 - 클래스 8 => 가방
 - 클래스 9 => 앵클 부츠
- CNN은 처음부터 훈련됩니다. 전이 학습은 수행되지 않습니다.
- 데이터셋 버전은 Kaggle에서 제공하는 [Fashion MNIST](#)입니다.

Data Exploration

Let us load the libraries we are going to use in our notebook and also the dataset to display it and get a grasp of the data.

The data is in the usual grayscale image numeric representation: one column per pixel and each value representing the gray color intensity. Color intensity ranges from 0 (completely black) to 255 (completely white).

데이터 탐색

우리가 노트북에서 사용할 라이브러리와 데이터를 로드하여 데이터를 표시하고 이해해 보겠습니다.

데이터는 일반적인 그레이스케일 이미지 수치 표현 방식으로, 각 픽셀당 하나의 열이 있으며 각 값은 회색의 색상 강도를 나타냅니다. 색상 강도 범위는 0(완전히 검은색)에서 255(완전히 흰색)입니다.

```
In [21]: import numpy as np
import pandas as pd
import seaborn as sns
```

```
from matplotlib import pyplot as plt
from tensorflow.keras import models, layers
from tensorflow.keras import callbacks
from sklearn.metrics import confusion_matrix, classification_report
```

```
In [22]: import zipfile
import os

# Extract training set from FashionMNIST dataset.
zip_object = zipfile.ZipFile('data/raw/fashion-mnist-train.zip')
zip_object.extractall('data/raw/')
zip_object.close()

# Extract testing set from FashionMNIST dataset.
zip_object = zipfile.ZipFile('data/raw/fashion-mnist-test.zip')
zip_object.extractall('data/raw/')
zip_object.close()
```

```
In [23]: fashion_train_df = pd.read_csv('data/raw/fashion-mnist-train.csv')
fashion_test_df = pd.read_csv('data/raw/fashion-mnist-test.csv')

display(fashion_train_df)
display(fashion_test_df)
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775
0	2	0	0	0	0	0	0	0	0	0	...	0
1	9	0	0	0	0	0	0	0	0	0	...	0
2	6	0	0	0	0	0	0	0	5	0	...	0
3	0	0	0	0	1	2	0	0	0	0	...	3
4	3	0	0	0	0	0	0	0	0	0	...	0
...
59995	9	0	0	0	0	0	0	0	0	0	...	0
59996	1	0	0	0	0	0	0	0	0	0	...	73
59997	8	0	0	0	0	0	0	0	0	0	...	160
59998	8	0	0	0	0	0	0	0	0	0	...	0
59999	7	0	0	0	0	0	0	0	0	0	...	0

60000 rows × 785 columns



	label	pixel1	pixel2	pixel3	pixel4	pixel5	pixel6	pixel7	pixel8	pixel9	...	pixel775	p
0	0	0	0	0	0	0	0	0	9	8	...	103	
1	1	0	0	0	0	0	0	0	0	0	...	34	
2	2	0	0	0	0	0	0	14	53	99	...	0	
3	2	0	0	0	0	0	0	0	0	0	...	137	
4	3	0	0	0	0	0	0	0	0	0	...	0	
...	
9995	0	0	0	0	0	0	0	0	0	0	...	32	
9996	6	0	0	0	0	0	0	0	0	0	...	0	
9997	8	0	0	0	0	0	0	0	0	0	...	175	
9998	8	0	1	3	0	0	0	0	0	0	...	0	
9999	1	0	0	0	0	0	0	0	140	119	...	111	

10000 rows × 785 columns



Pandas' `info` and `describe` will not tell us much on this dataset.

`info` basically only tells dataset dimensions and memory usage, which is important to evaluate on image datasets as they can be very large. Fashion MNIST only uses 359 MB; any decent machine can work on it fairly easy.

Meanwhile, `describe` presents descriptive statistics on the data. The only useful information here are the mean (or min and max) values for each pixel column, telling us that the object in the image is centered by default. We can say that because border pixels have mean values close to 0 and center pixels have mean values far from 0.

Pandas의 `info`와 `describe`는 이 데이터셋에 대해 많은 정보를 제공하지 않을 것입니다.

`info`는 기본적으로 데이터셋의 차원과 메모리 사용량만 알려주며, 이는 이미지 데이터셋에서 중요한 평가 요소입니다. Fashion MNIST의 경우 크기는 359 MB로, 어느 정도 성능이 좋은 머신이 라면 비교적 쉽게 작업할 수 있습니다.

한편, `describe`는 데이터에 대한 기술 통계를 제공합니다. 여기서 유용한 정보는 각 픽셀 열에 대한 평균(또는 최소 및 최대) 값인데, 이는 이미지의 객체가 기본적으로 중앙에 위치한다는 것을 나타냅니다. 이는 가장자리 픽셀의 평균 값이 0에 가까운 반면, 중앙 픽셀의 평균 값은 0에서 멀리 떨어져 있다는 사실로 확인할 수 있습니다.

In [24]: `fashion_train_df.info()`

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 60000 entries, 0 to 59999
Columns: 785 entries, label to pixel784
dtypes: int64(785)
memory usage: 359.3 MB
```

```
In [25]: fashion_train_df.describe()
```

	label	pixel1	pixel2	pixel3	pixel4	pixel5
count	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000	60000.000000
mean	4.500000	0.000900	0.006150	0.035333	0.101933	0.247967
std	2.872305	0.094689	0.271011	1.222324	2.452871	4.306912
min	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
25%	2.000000	0.000000	0.000000	0.000000	0.000000	0.000000
50%	4.500000	0.000000	0.000000	0.000000	0.000000	0.000000
75%	7.000000	0.000000	0.000000	0.000000	0.000000	0.000000
max	9.000000	16.000000	36.000000	226.000000	164.000000	227.000000

8 rows × 785 columns



Nonetheless, numeric information will not help much on visual datasets, so we build a grid with random images from the dataset. Image labels are printed above the subfigures.

We normalize our images to have their pixel values in the [0, 1] range. In this manner, any classifier can update its gradient faster and, therefore, converge faster.

그럼에도 불구하고 숫자 정보는 시각적 데이터셋에서는 큰 도움이 되지 않으므로, 데이터셋에서 무작위 이미지를 사용하여 그리드를 만듭니다. 이미지의 레이블은 서브피겨 위에 표시됩니다.

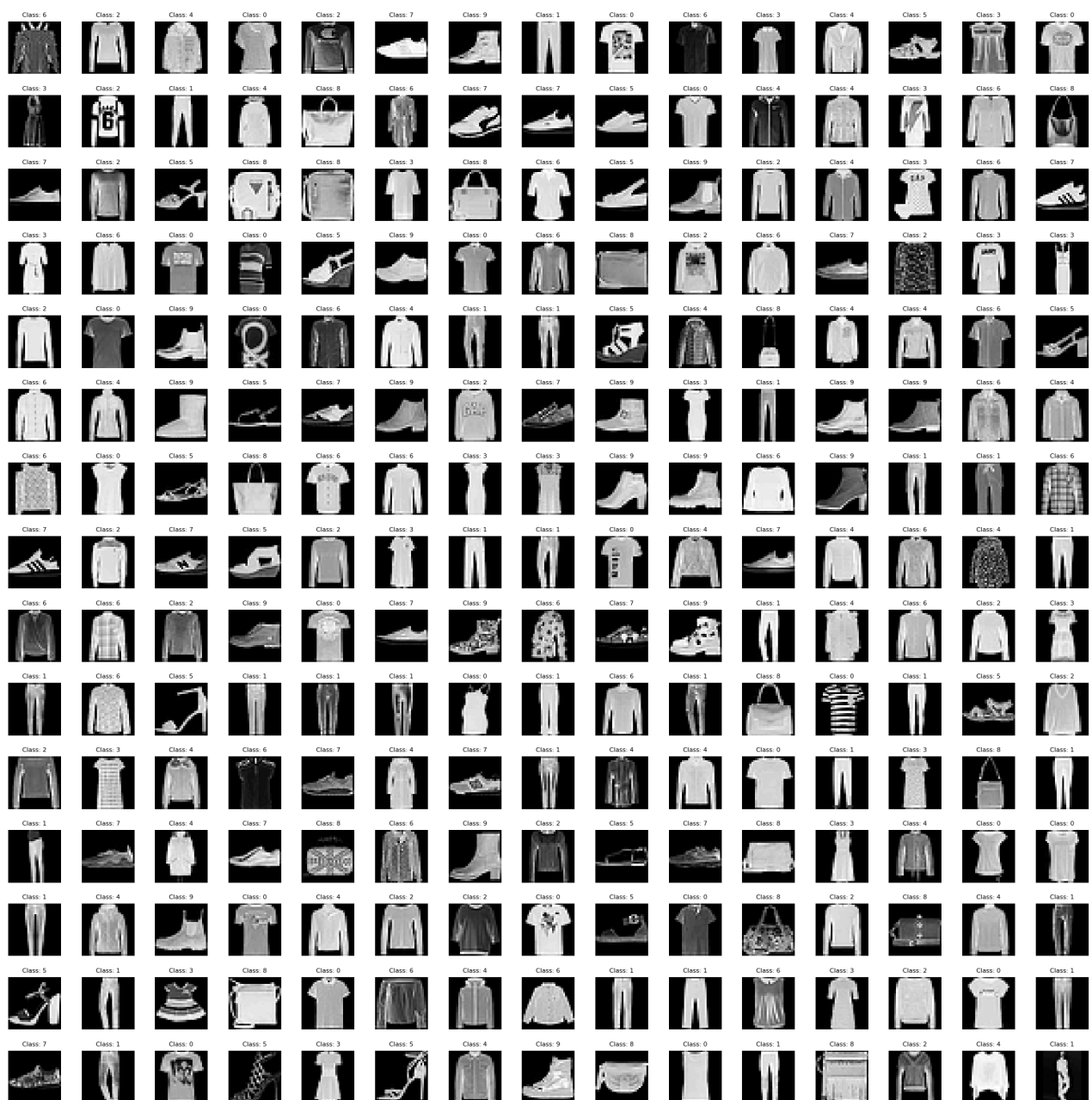
이미지를 [0, 1] 범위로 정규화하여 픽셀 값이 이 범위 안에 있도록 합니다. 이러한 방식으로 모든 분류기는 기울기를 더 빨리 업데이트할 수 있으며, 따라서 더 빠르게 수렴할 수 있습니다.

```
In [26]: # Get values from Pandas dataframe to plot with matplotlib.
fashion_train = fashion_train_df.values
fashion_test = fashion_test_df.values

# Dimensions of the image grid.
grid_width = 15
grid_height = 15

fig, axes = plt.subplots(grid_height, grid_width, figsize=(25, 25))
axes = axes.ravel()

for i in np.arange(0, grid_height * grid_width):
    # Draft random index to get random image.
    index = np.random.randint(0, fashion_train.shape[0])
    axes[i].imshow(fashion_train[index, 1:].reshape(28, 28), cmap='gray')
    # Show image class label above the drafted image.
    axes[i].set_title("Class: " + str(fashion_train[index, 0]), fontsize=8)
    axes[i].axis('off')
plt.subplots_adjust(hspace=0.4)
```

```
In [27]: # Normalize each pixel intensity to be in the [0, 1] range.
X_train = fashion_train[:, 1:] / 255

# Reshape the 784 pixel columns to 28x28, in a single channel.
X_train = X_train.reshape(X_train.shape[0], 28, 28, 1)
y_train = fashion_train[:, 0]

X_test = fashion_test[:, 1:] / 255
X_test = X_test.reshape(X_test.shape[0], 28, 28, 1)
y_test = fashion_test[:, 0]

print(X_train.shape)
print(X_test.shape)
```

```
(60000, 28, 28, 1)
(10000, 28, 28, 1)
```

Model Building and Training

Our model is a simple one with only a single stream of layers. We can use the `Sequential` framework from the TensorFlow API to build such a model.

We use 3 convolutional layers and 4 fully connected ones. The first convolutional has 32 feature maps and the remaining have 64. Every convolutional layer has a 3x3 kernel.

We define each fully connected layer to have its number of neurons approximately equal to half the number of input features. Therefore, since the flatten layer returns 576 features, the first fully connected uses 250, the second uses 125 and the penultimate uses 60. The last fully connected layer uses 10 neurons because we have 10 classes.

We add max pooling layers between each convolutional, except for the last one. Max pooling is used instead of average pooling because the edges of the objects, which max pooling identifies better, have more useful information than the contrast of the objects.

All layers use ReLU as activation function, except for the last layer that uses softmax.

모델 구축 및 교육

우리의 모델은 단일 스트림 레이어로 구성된 간단한 모델입니다. TensorFlow API의 `Sequential` 프레임워크를 사용하여 이러한 모델을 구축할 수 있습니다.

3개의 합성곱(convolutional) 레이어와 4개의 완전 연결(fully connected) 레이어를 사용합니다. 첫 번째 합성곱 레이어는 32개의 특징 맵(feature maps)을 가지고 있으며, 나머지 레이어는 64개를 사용합니다. 각 합성곱 레이어는 3x3 커널을 갖습니다.

각 완전 연결 레이어는 입력 특징의 수의 약 절반에 해당하는 뉴런 수를 갖도록 정의합니다. 따라서 풀 Flatten 레이어가 576개의 특징을 반환하므로, 첫 번째 완전 연결 레이어는 250개, 두 번째 레이어는 125개, 그 이전 레이어는 60개 뉴런을 사용합니다. 마지막 완전 연결 레이어는 10개의 클래스를 가지고 있기 때문에 10개의 뉴런을 사용합니다.

모든 합성곱 레이어 사이에 최대 풀링(max pooling) 레이어를 추가하며, 마지막 레이어는 제외합니다. 최대 풀링은 평균 풀링보다 객체의 가장자리를 더 잘 식별하므로, 객체의 컨트라스트보다 더 유용한 정보를 제공합니다.

모든 레이어는 ReLU를 활성화 함수로 사용하며, 마지막 레이어는 softmax를 사용합니다.

```
In [28]: cnn_model = models.Sequential()

cnn_model.add(layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)))
cnn_model.add(layers.MaxPool2D(pool_size=(2, 2)))

cnn_model.add(layers.Conv2D(64, (3, 3), activation='relu'))
cnn_model.add(layers.MaxPool2D(pool_size=(2, 2)))

cnn_model.add(layers.Conv2D(64, (3, 3), activation='relu'))

cnn_model.add(layers.Flatten())

# Each fully connected has half its input as its number of neurons.
cnn_model.add(layers.Dense(250, activation='relu'))
cnn_model.add(layers.Dense(125, activation='relu'))
cnn_model.add(layers.Dense(60, activation='relu'))
```

```
# 10 neurons in the last layer as we have 10 classes.
cnn_model.add(layers.Dense(10, activation='softmax'))

cnn_model.summary()
```

C:\Users\ASUS\AppData\Roaming\Python\Python312\site-packages\keras\src\layers\convolutional\base_conv.py:107: UserWarning: Do not pass an `input_shape`/`input_dim` argument to a layer. When using Sequential models, prefer using an `Input(shape)` object as the first layer in the model instead.

```
super().__init__(activity_regularizer=activity_regularizer, **kwargs)
```

Model: "sequential"

Layer (type)	Output Shape	Param #
conv2d (Conv2D)	(None, 26, 26, 32)	320
max_pooling2d (MaxPooling2D)	(None, 13, 13, 32)	0
conv2d_1 (Conv2D)	(None, 11, 11, 64)	18,496
max_pooling2d_1 (MaxPooling2D)	(None, 5, 5, 64)	0
conv2d_2 (Conv2D)	(None, 3, 3, 64)	36,928
flatten (Flatten)	(None, 576)	0
dense (Dense)	(None, 250)	144,250
dense_1 (Dense)	(None, 125)	31,375
dense_2 (Dense)	(None, 60)	7,560
dense_3 (Dense)	(None, 10)	610

Total params: 239,539 (935.70 KB)

Trainable params: 239,539 (935.70 KB)

Non-trainable params: 0 (0.00 B)

For training, we empirically defined 20 batches with 3000 images each. We also set an early stopping callback that waits for 10 iterations without improvement before being triggered.

500 epochs are more than enough to let our early stopping callback perform its work.

Our validation set has 15% the size of the training set. Close to the testing set that has 10,000 images (nearly 15% of the 60,000 training images).

훈련을 위해 20개의 배치(batch)를 정의하였고, 각 배치에는 3000개의 이미지가 포함되어 있습니다. 또한 개선이 없는 10회의 반복을 기다린 후에 트리거되는 조기 종료(early stopping) 콜백을 설정했습니다.

500 에폭(epoch)은 우리의 조기 종료 콜백이 작동하기에 충분한 양입니다.


우리의 검증 세트는 훈련 세트 크기의 15%를 차지합니다. 테스트 세트는 10,000개의 이미지를 포함하고 있으며(60,000개의 훈련 이미지의 거의 15%에 해당), 검증 세트와 비슷한 규모입니다.

```
In [29]: early_stopping_callback = callbacks.EarlyStopping(patience=10)
batch_size = X_train.shape[0] // 20


# 모델을 컴파일할 때, metrics를 리스트로 변경
cnn_model.compile(optimizer='Adam',
                  loss='sparse_categorical_crossentropy',
                  metrics=['accuracy']) # metrics을 리스트로 수정

epochs_info = cnn_model.fit(X_train,
                            y_train,
                            batch_size=batch_size,
                            epochs=500,
                            callbacks=[early_stopping_callback],
                            validation_split=0.15)
```


Epoch 1/500

17/17  11s 488ms/step - accuracy: 0.2684 - loss: 2.0779 - val_accuracy: 0.5919 - val_loss: 1.0278


Epoch 2/500

17/17  8s 490ms/step - accuracy: 0.6466 - loss: 0.9375 - val_accuracy: 0.7017 - val_loss: 0.7952


Epoch 3/500

17/17  8s 471ms/step - accuracy: 0.7172 - loss: 0.7569 - val_accuracy: 0.7411 - val_loss: 0.6905


Epoch 4/500

17/17  8s 464ms/step - accuracy: 0.7526 - loss: 0.6626 - val_accuracy: 0.7628 - val_loss: 0.6231


Epoch 5/500

17/17  8s 463ms/step - accuracy: 0.7800 - loss: 0.5891 - val_accuracy: 0.7789 - val_loss: 0.5840


Epoch 6/500

17/17  7s 437ms/step - accuracy: 0.7894 - loss: 0.5578 - val_accuracy: 0.7959 - val_loss: 0.5545


Epoch 7/500

17/17  7s 427ms/step - accuracy: 0.8059 - loss: 0.5242 - val_accuracy: 0.8077 - val_loss: 0.5188


Epoch 8/500

17/17  7s 421ms/step - accuracy: 0.8192 - loss: 0.4924 - val_accuracy: 0.8223 - val_loss: 0.4863


Epoch 9/500

17/17  7s 428ms/step - accuracy: 0.8260 - loss: 0.4764 - val_accuracy: 0.8340 - val_loss: 0.4600


Epoch 10/500

17/17  7s 426ms/step - accuracy: 0.8399 - loss: 0.4443 - val_accuracy: 0.8440 - val_loss: 0.4416


Epoch 11/500

17/17  8s 440ms/step - accuracy: 0.8486 - loss: 0.4192 - val_accuracy: 0.8437 - val_loss: 0.4369


Epoch 12/500

17/17  7s 440ms/step - accuracy: 0.8425 - loss: 0.4312 - val_accuracy: 0.8440 - val_loss: 0.4272


Epoch 13/500

17/17  7s 438ms/step - accuracy: 0.8539 - loss: 0.4083 - val_accuracy: 0.8579 - val_loss: 0.4066


Epoch 14/500

17/17  7s 429ms/step - accuracy: 0.8594 - loss: 0.3923 - val_accuracy: 0.8650 - val_loss: 0.3892


Epoch 15/500

17/17  7s 434ms/step - accuracy: 0.8675 - loss: 0.3740 - val_accuracy: 0.8654 - val_loss: 0.3895


Epoch 16/500

17/17  7s 431ms/step - accuracy: 0.8650 - loss: 0.3696 - val_accuracy: 0.8632 - val_loss: 0.3799


Epoch 17/500

17/17  7s 422ms/step - accuracy: 0.8733 - loss: 0.3567 - val_accuracy: 0.8759 - val_loss: 0.3568


Epoch 18/500

17/17  7s 429ms/step - accuracy: 0.8752 - loss: 0.3488 - val_accuracy: 0.8703 - val_loss: 0.3651


Epoch 19/500

17/17  8s 445ms/step - accuracy: 0.8783 - loss: 0.3428 - val_accuracy: 0.8822 - val_loss: 0.3416


Epoch 20/500

17/17  7s 429ms/step - accuracy: 0.8830 - loss: 0.3269 - val_accuracy: 0.8766 - val_loss: 0.3438


Epoch 21/500

17/17  7s 422ms/step - accuracy: 0.8799 - loss: 0.3281 - val_accuracy: 0.8839 - val_loss: 0.3341


Epoch 22/500

17/17  8s 459ms/step - accuracy: 0.8865 - loss: 0.3190 - val_accuracy: 0.8810 - val_loss: 0.3437


Epoch 23/500

17/17  8s 466ms/step - accuracy: 0.8854 - loss: 0.3174 - val_accuracy: 0.8829 - val_loss: 0.3262


Epoch 24/500

17/17  8s 462ms/step - accuracy: 0.8850 - loss: 0.3110 - val_accuracy: 0.8837 - val_loss: 0.3219


Epoch 25/500

17/17  8s 459ms/step - accuracy: 0.8913 - loss: 0.3003 - val_accuracy: 0.8888 - val_loss: 0.3172


Epoch 26/500

17/17  8s 476ms/step - accuracy: 0.8947 - loss: 0.2922 - val_accuracy: 0.8879 - val_loss: 0.3226


Epoch 27/500

17/17  8s 474ms/step - accuracy: 0.8949 - loss: 0.2913 - val_accuracy: 0.8924 - val_loss: 0.3050


Epoch 28/500

17/17  8s 442ms/step - accuracy: 0.8929 - loss: 0.2934 - val_accuracy: 0.8867 - val_loss: 0.3137


Epoch 29/500

17/17  7s 421ms/step - accuracy: 0.8972 - loss: 0.2845 - val_accuracy: 0.8951 - val_loss: 0.3017


Epoch 30/500

17/17  7s 414ms/step - accuracy: 0.8994 - loss: 0.2775 - val_accuracy: 0.8980 - val_loss: 0.2950


Epoch 31/500

17/17  7s 436ms/step - accuracy: 0.8994 - loss: 0.2792 - val_accuracy: 0.8971 - val_loss: 0.2953


Epoch 32/500

17/17  7s 416ms/step - accuracy: 0.9004 - loss: 0.2720 - val_accuracy: 0.8956 - val_loss: 0.3029


Epoch 33/500

17/17  7s 431ms/step - accuracy: 0.9012 - loss: 0.2732 - val_accuracy: 0.8956 - val_loss: 0.2957


Epoch 34/500

17/17  7s 434ms/step - accuracy: 0.9046 - loss: 0.2672 - val_accuracy: 0.8966 - val_loss: 0.2905


Epoch 35/500

17/17  8s 469ms/step - accuracy: 0.9045 - loss: 0.2634 - val_accuracy: 0.8970 - val_loss: 0.2932


Epoch 36/500

17/17  8s 491ms/step - accuracy: 0.9053 - loss: 0.2613 - val_accuracy: 0.8916 - val_loss: 0.2990


Epoch 37/500

17/17  8s 489ms/step - accuracy: 0.9045 - loss: 0.2620 - val_accuracy: 0.9046 - val_loss: 0.2782


Epoch 38/500

17/17  8s 490ms/step - accuracy: 0.9068 - loss: 0.2569 - val_accuracy: 0.9029 - val_loss: 0.2795


Epoch 39/500

17/17  8s 481ms/step - accuracy: 0.9119 - loss: 0.2449 - val_accuracy: 0.9008 - val_loss: 0.2775


Epoch 40/500

17/17  8s 490ms/step - accuracy: 0.9156 - loss: 0.2385 - val_accuracy: 0.9022 - val_loss: 0.2777


Epoch 41/500

17/17  8s 486ms/step - accuracy: 0.9108 - loss: 0.2461 - val_accuracy: 0.9028 - val_loss: 0.2722


Epoch 42/500

17/17  8s 486ms/step - accuracy: 0.9127 - loss: 0.2388 - val_accuracy: 0.9037 - val_loss: 0.2720


Epoch 43/500

17/17  8s 474ms/step - accuracy: 0.9157 - loss: 0.2354 - val_accuracy: 0.8940 - val_loss: 0.2928


Epoch 44/500

17/17  8s 482ms/step - accuracy: 0.9117 - loss: 0.2424 - val_accuracy: 0.8963 - val_loss: 0.2875


Epoch 45/500

17/17  8s 457ms/step - accuracy: 0.9153 - loss: 0.2321 - val_accuracy: 0.9028 - val_loss: 0.2745


Epoch 46/500

17/17  8s 438ms/step - accuracy: 0.9181 - loss: 0.2310 - val_accuracy: 0.9051 - val_loss: 0.2667


Epoch 47/500

17/17  8s 477ms/step - accuracy: 0.9163 - loss: 0.2288 - val_accuracy: 0.8996 - val_loss: 0.2751


Epoch 48/500

17/17  9s 507ms/step - accuracy: 0.9142 - loss: 0.2344 - val_accuracy: 0.9053 - val_loss: 0.2653


Epoch 49/500

17/17  8s 457ms/step - accuracy: 0.9219 - loss: 0.2187 - val_accuracy: 0.9067 - val_loss: 0.2608


Epoch 50/500

17/17  8s 441ms/step - accuracy: 0.9225 - loss: 0.2144 - val_accuracy: 0.9066 - val_loss: 0.2663


Epoch 51/500

17/17  8s 498ms/step - accuracy: 0.9234 - loss: 0.2142 - val_accuracy: 0.9010 - val_loss: 0.2798


Epoch 52/500

17/17  8s 476ms/step - accuracy: 0.9232 - loss: 0.2121 - val_accuracy: 0.9030 - val_loss: 0.2665


Epoch 53/500

17/17  10s 576ms/step - accuracy: 0.9253 - loss: 0.2109 - val_accuracy: 0.9043 - val_loss: 0.2651


Epoch 54/500

17/17  10s 555ms/step - accuracy: 0.9272 - loss: 0.2022 - val_accuracy: 0.9086 - val_loss: 0.2560


Epoch 55/500

17/17  9s 520ms/step - accuracy: 0.9226 - loss: 0.2105 - val_accuracy: 0.9048 - val_loss: 0.2690


Epoch 56/500

17/17  9s 511ms/step - accuracy: 0.9243 - loss: 0.2089 - val_accuracy: 0.9053 - val_loss: 0.2618


Epoch 57/500

17/17  9s 520ms/step - accuracy: 0.9266 - loss: 0.2042 - val_accuracy: 0.9079 - val_loss: 0.2604


Epoch 58/500

17/17  9s 507ms/step - accuracy: 0.9264 - loss: 0.2046 - val_accuracy: 0.9061 - val_loss: 0.2606
















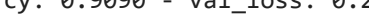

Epoch 59/500

17/17  9s 524ms/step - accuracy: 0.9282 - loss: 0.1945 - val_accuracy: 0.9104 - val_loss: 0.2510

Epoch 60/500

17/17  10s 576ms/step - accuracy: 0.9317 - loss: 0.1891 - val_accuracy: 0.9089 - val_loss: 0.2577

```

Epoch 61/500
17/17  9s 532ms/step - accuracy: 0.9313 - loss: 0.1921 - val_accuracy: 0.9070 - val_loss: 0.2635
Epoch 62/500
17/17  9s 525ms/step - accuracy: 0.9292 - loss: 0.1957 - val_accuracy: 0.9073 - val_loss: 0.2607
Epoch 63/500
17/17  9s 542ms/step - accuracy: 0.9311 - loss: 0.1910 - val_accuracy: 0.9066 - val_loss: 0.2642
Epoch 64/500
17/17  10s 555ms/step - accuracy: 0.9326 - loss: 0.1879 - val_accuracy: 0.9067 - val_loss: 0.2617
Epoch 65/500
17/17  9s 520ms/step - accuracy: 0.9309 - loss: 0.1904 - val_accuracy: 0.9091 - val_loss: 0.2578
Epoch 66/500
17/17  9s 524ms/step - accuracy: 0.9362 - loss: 0.1762 - val_accuracy: 0.9087 - val_loss: 0.2603
Epoch 67/500
17/17  9s 532ms/step - accuracy: 0.9364 - loss: 0.1732 - val_accuracy: 0.9121 - val_loss: 0.2495
Epoch 68/500
17/17  9s 535ms/step - accuracy: 0.9381 - loss: 0.1704 - val_accuracy: 0.9067 - val_loss: 0.2653
Epoch 69/500
17/17  9s 534ms/step - accuracy: 0.9352 - loss: 0.1791 - val_accuracy: 0.9116 - val_loss: 0.2557
Epoch 70/500
17/17  10s 573ms/step - accuracy: 0.9365 - loss: 0.1726 - val_accuracy: 0.9099 - val_loss: 0.2606
Epoch 71/500
17/17  9s 531ms/step - accuracy: 0.9385 - loss: 0.1676 - val_accuracy: 0.9098 - val_loss: 0.2540
Epoch 72/500
17/17  9s 529ms/step - accuracy: 0.9373 - loss: 0.1699 - val_accuracy: 0.9077 - val_loss: 0.2569
Epoch 73/500
17/17  8s 496ms/step - accuracy: 0.9395 - loss: 0.1671 - val_accuracy: 0.9109 - val_loss: 0.2524
Epoch 74/500
17/17  9s 505ms/step - accuracy: 0.9406 - loss: 0.1621 - val_accuracy: 0.9130 - val_loss: 0.2497
Epoch 75/500
17/17  9s 513ms/step - accuracy: 0.9454 - loss: 0.1541 - val_accuracy: 0.9102 - val_loss: 0.2535
Epoch 76/500
17/17  9s 546ms/step - accuracy: 0.9384 - loss: 0.1662 - val_accuracy: 0.9090 - val_loss: 0.2608
Epoch 77/500
17/17  9s 499ms/step - accuracy: 0.9434 - loss: 0.1583 - val_accuracy: 0.9111 - val_loss: 0.2520

```

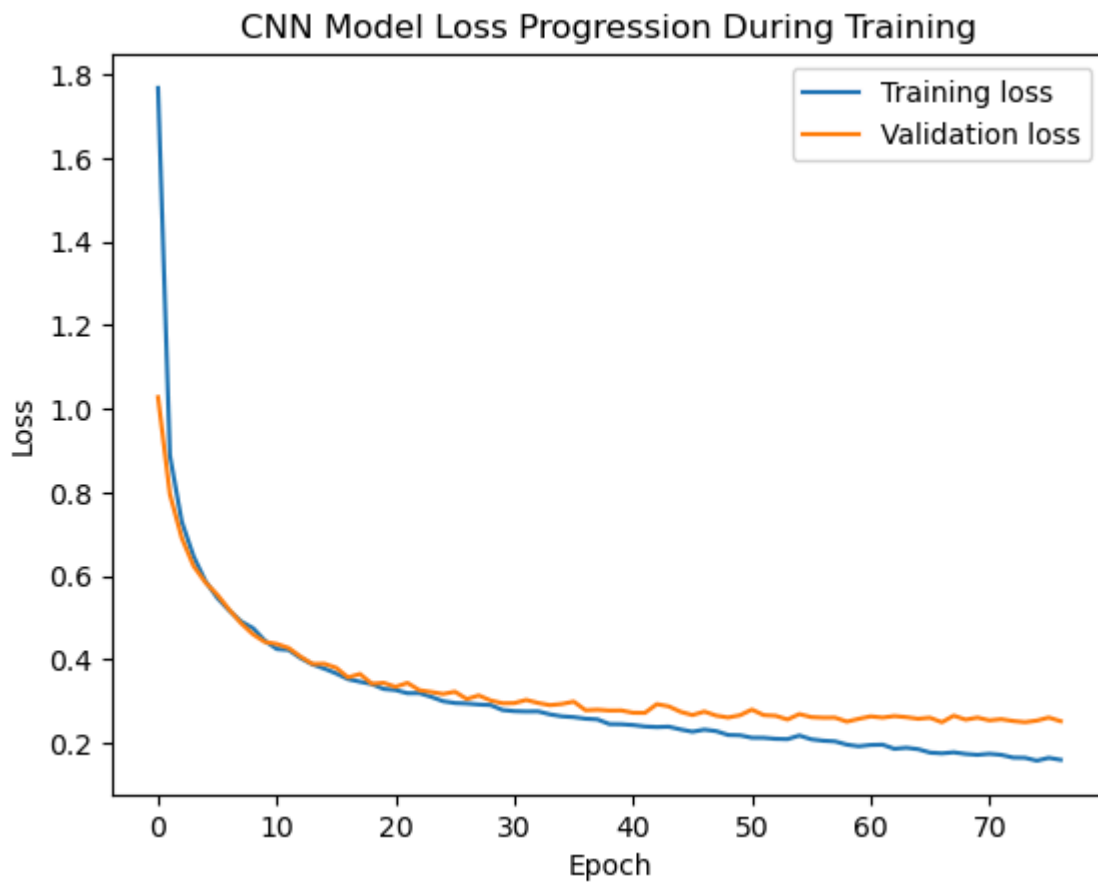
We can see in the loss curves below that our model overfits a bit before the validation loss stabilizes. Just so our model stays simple, and as the overfitting was not severe, we have opted for not adding dropout layers or any other countermeasures for overfitting.

Other than that, we have also chosen to let the validation loss converge instead of stopping the training once the training loss had left the validation loss behind.

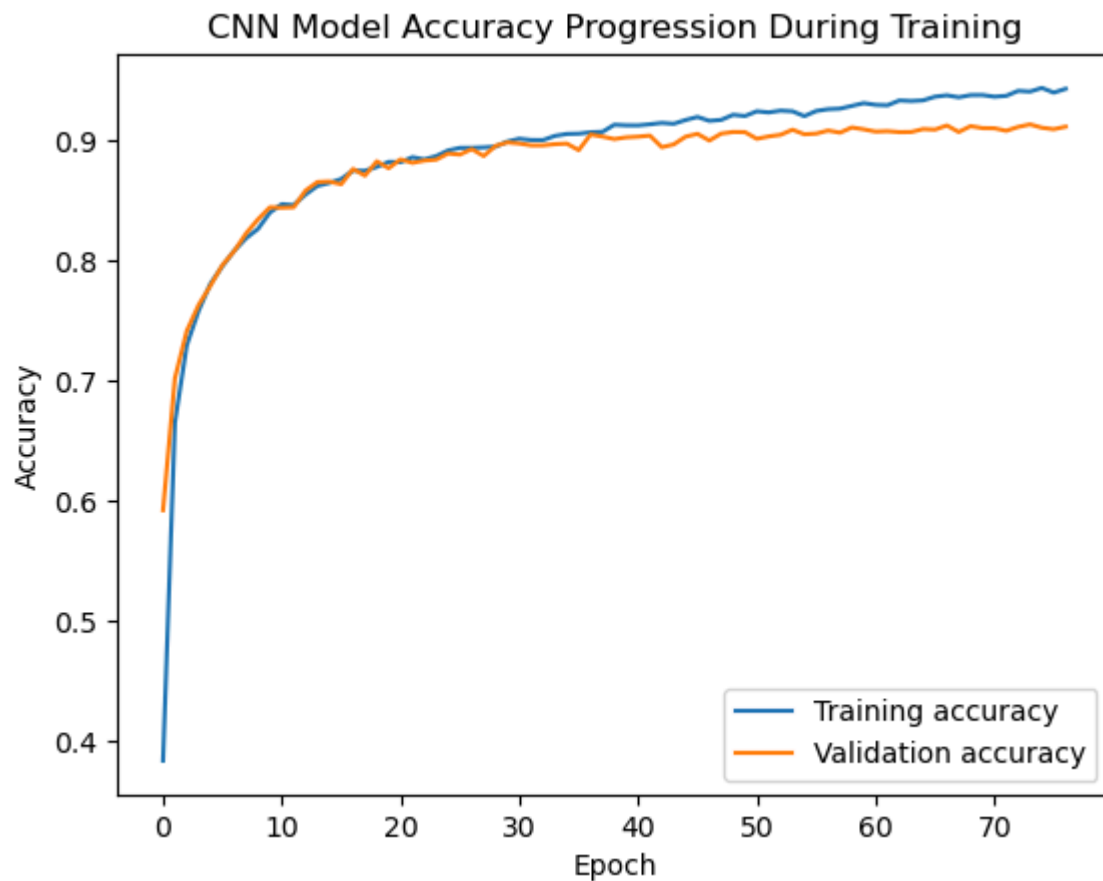
아래 손실 곡선에서 볼 수 있듯이, 우리의 모델은 검증 손실이 안정화되기 전에 약간의 오버피팅(overfitting)을 보입니다. 모델을 간단하게 유지하기 위해 오버피팅이 심각하지 않았으므로 드롭아웃(dropout) 레이어나 기타 오버피팅 방지 조치를 추가하지 않기로 결정했습니다.

그 외에도, 훈련 손실이 검증 손실을 초과했을 때 훈련을 중단하는 대신, 검증 손실이 수렴할 때까지 훈련을 계속 진행하기로 선택했습니다.

```
In [30]: plt.plot(epochs_info.history['loss'])
plt.plot(epochs_info.history['val_loss'])
plt.title("CNN Model Loss Progression During Training")
plt.xlabel("Epoch")
plt.ylabel("Loss")
plt.legend(["Training loss", "Validation loss"])
plt.show()
```



```
In [31]: plt.plot(epochs_info.history['accuracy'])
plt.plot(epochs_info.history['val_accuracy'])
plt.title("CNN Model Accuracy Progression During Training")
plt.xlabel("Epoch")
plt.ylabel("Accuracy")
plt.legend(["Training accuracy", "Validation accuracy"])
plt.show()
```



Model Evaluation

Using TensorFlow's `evaluate` function we get an accuracy of nearly 91% for our model. This is a neat performance, considering that our model is rather simple, being shallow and ignoring a bit of overfitting.

The confusion matrix for our model's predictions also has the majority of values in the main diagonal, indicating a good performance. Note that the mistakes are concentrated in some specific classes, like class 0 (t-shirt) that gets misclassified as class 6 (shirt) a lot of times. This is understandable, since both objects are quite similar and the images do not have much details to help differentiate them.

모델 평가

TensorFlow의 `evaluate` 함수를 사용한 결과, 우리 모델의 정확도는 거의 91%에 달합니다. 이는 모델이 상대적으로 간단하고 얕으며 약간의 오버피팅을 무시하고 있다는 점을 고려할 때 괜찮은 성능입니다.

모델의 예측에 대한 혼동 행렬(confusion matrix)에서도 대부분의 값이 주 대각선(main diagonal)에 집중되어 있어 좋은 성능을 나타냅니다. 잘못 분류된 경우는 특정 클래스에 집중되어 있으며, 예를 들어 클래스 0(티셔츠)이 클래스 6(셔츠)로 많이 잘못 분류됩니다. 이는 두 객체가 상당히 유사하고 이미지에 그들을 구별하는 데 도움이 되는 세부 정보가 많지 않기 때문에 이해할 수 있습니다.

```
In [32]: cnn_model.evaluate(X_test, y_test)
```

313/313 ————— 1s 4ms/step - accuracy: 0.9126 - loss: 0.2578

Out[32]: [0.2404756098985672, 0.9165999889373779]

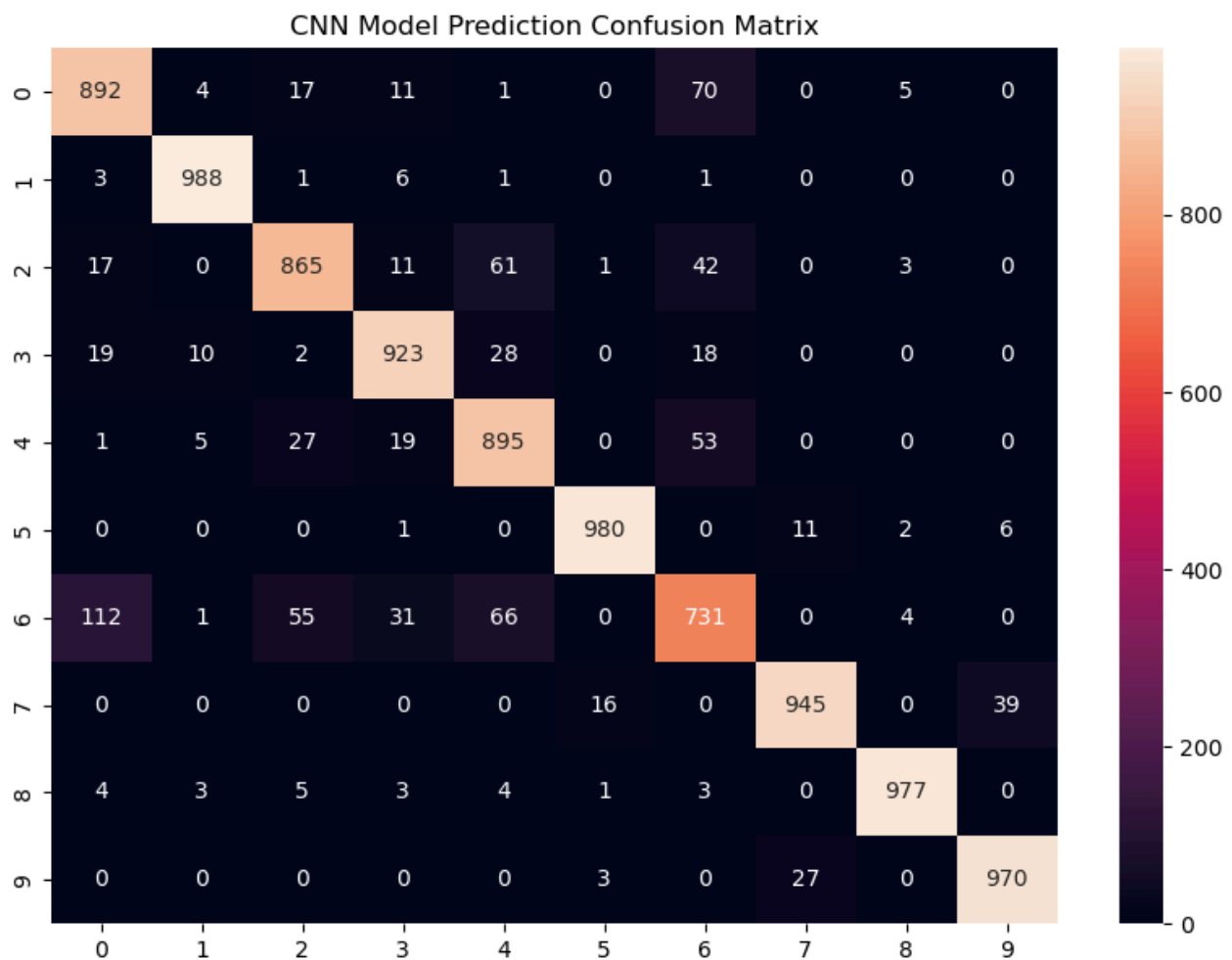
```
In [33]: y_pred = cnn_model.predict(X_test)

# Get the class with highest predicted probability and assume it is the
# model prediction.
y_pred_classes = np.argmax(y_pred, axis=1)

cm = confusion_matrix(y_test, y_pred_classes)

plt.figure(figsize=(10, 7))
sns.heatmap(cm, annot=True, fmt='d')
plt.title("CNN Model Prediction Confusion Matrix")
plt.show()
```

313/313 ————— 1s 4ms/step



Sklearn's `classification_report` reassures our previous conclusions. Overall accuracy is at 91%. Class 6 has a low F1-score, for being a false positive for class 0 too much.

For last, we have another image grid with random images from the dataset. This time, we print the actual label and the predicted label for each image. We can see the mistakes described by the confusion matrix in line 14, column 5 and line 14, column 8.

We can also see how much class 0 (t-shirt) and class 6 (shirt) look alike. Another class with low recall is class 2 (pullover) and it is also very similar to classes 0 and 6. Low recall means many false negatives, i.e., class 2 is commonly said to be class 0 or 6.

Our CNN model was not able to capture features well enough to distinguish the similar classes.

Sklearn의 `classification_report`는 우리가 이전에 도출한 결론을 입증해줍니다. 전체 정확도는 91%이며, 클래스 6은 클래스 0에 대한 잘못된 긍정(false positive) 때문에 낮은 F1 점수를 기록하고 있습니다.

마지막으로, 데이터셋의 무작위 이미지로 구성된 또 다른 이미지 그리드를 보여드립니다. 이번에는 각 이미지에 대해 실제 레이블과 예측된 레이블을 출력합니다. 혼동 행렬에서 설명한 오류는 14행 5열과 14행 8열에서 확인할 수 있습니다.

클래스 0(티셔츠)과 클래스 6(셔츠)은 얼마나 유사하게 보이는지도 확인할 수 있습니다. 낮은 리콜(recall)을 보이는 또 다른 클래스는 클래스 2(풀오버)이며, 이 클래스도 클래스 0과 6과 매우 유사합니다. 낮은 리콜은 많은 거짓 부정(false negatives)을 의미하며, 즉 클래스 2가 일반적으로 클래스 0 또는 6으로 잘못 분류된다는 것을 나타냅니다.

우리의 CNN 모델은 유사한 클래스들을 구별하기에 충분히 특징을 잘 포착하지 못했습니다.

```
In [34]: class_names = ["Class 0 => T-shirt",
                        "Class 1 => Trouser",
                        "Class 2 => Pullover",
                        "Class 3 => Dress",
                        "Class 4 => Coat",
                        "Class 5 => Sandal",
                        "Class 6 => Shirt",
                        "Class 7 => Sneaker",
                        "Class 8 => Bag",
                        "Class 9 => Ankle boot"]
print(classification_report(y_test, y_pred_classes, target_names=class_names))
```

	precision	recall	f1-score	support
Class 0 => T-shirt	0.85	0.89	0.87	1000
Class 1 => Trouser	0.98	0.99	0.98	1000
Class 2 => Pullover	0.89	0.86	0.88	1000
Class 3 => Dress	0.92	0.92	0.92	1000
Class 4 => Coat	0.85	0.90	0.87	1000
Class 5 => Sandal	0.98	0.98	0.98	1000
Class 6 => Shirt	0.80	0.73	0.76	1000
Class 7 => Sneaker	0.96	0.94	0.95	1000
Class 8 => Bag	0.99	0.98	0.98	1000
Class 9 => Ankle boot	0.96	0.97	0.96	1000
accuracy			0.92	10000
macro avg	0.92	0.92	0.92	10000
weighted avg	0.92	0.92	0.92	10000

```
In [35]: # Grid dimensions.
grid_height = 15
grid_width = 15

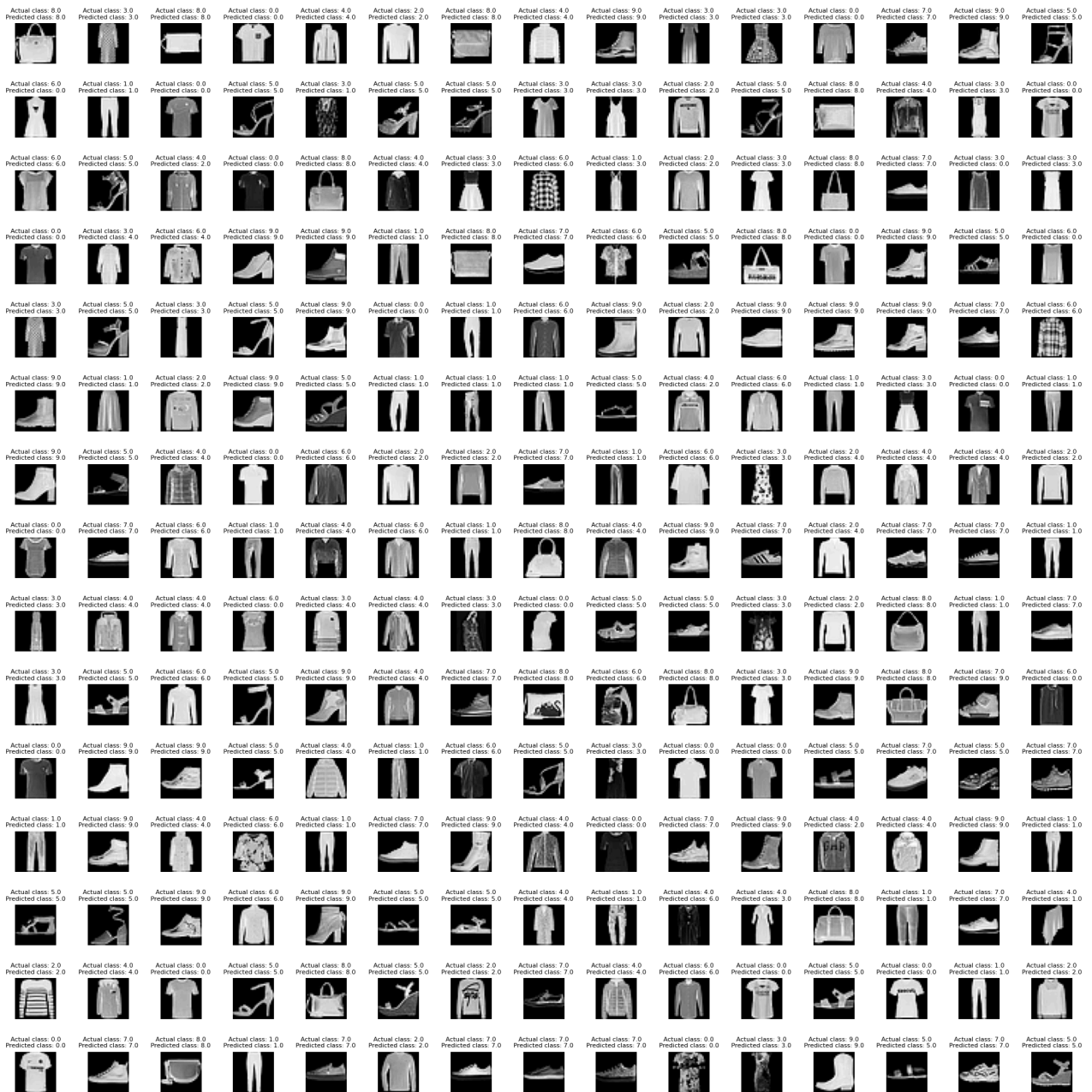
fig, axes = plt.subplots(grid_height, grid_width, figsize=(25, 25))
axes = axes.ravel()

for i in range(grid_height * grid_width):
    # Draft random index to plot random image.
```

```

drafted_image = np.random.randint(0, X_test.shape[0])
axes[i].imshow(X_test[drafted_image].reshape(28, 28), cmap='gray')
# Print actual and predicted labels for image.
axes[i].set_title("Actual class: {:.1f}\nPredicted class: {:.1f}".format(
    y_test[drafted_image], y_pred_classes[drafted_image]), fontsize=8)
axes[i].axis('off')
plt.subplots_adjust(hspace=0.8)

```



```

In [36]: import zipfile
import os
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader
import matplotlib.pyplot as plt
from keras import callbacks # Keras에서 필요한 콜백

```

```

# Fashion MNIST 데이터셋 압축 해제
# Extract training set from FashionMNIST dataset.
zip_object = zipfile.ZipFile('data/raw/fashion-mnist-train.zip')
zip_object.extractall('data/raw/')

```

```

zip_object.close()

# Extract testing set from FashionMNIST dataset.
zip_object = zipfile.ZipFile('data/raw/fashion-mnist-test.zip')
zip_object.extractall('data/raw/')
zip_object.close()

# 장치 설정
device = torch.device("cuda" if torch.cuda.is_available() else "cpu")

# 데이터 변환 설정
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.5,), (0.5,))
])

# Fashion MNIST 데이터셋 로드
train_dataset = datasets.FashionMNIST(root='data/', train=True, download=True, transfo
test_dataset = datasets.FashionMNIST(root='data/', train=False, download=True, transfo

train_loader = DataLoader(dataset=train_dataset, batch_size=64, shuffle=True)
test_loader = DataLoader(dataset=test_dataset, batch_size=64, shuffle=False)

# LSTM 모델 정의
class LSTMModel(nn.Module):
    def __init__(self, input_size=28, hidden_size=128, output_size=10):
        super(LSTMModel, self).__init__()
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        x = x.view(x.size(0), 28, 28) # (batch_size, seq_len, input_size)
        out, _ = self.lstm(x) # LSTM 처리
        out = self.fc(out[:, -1, :]) # 마지막 시퀀스의 출력
        return out

# 모델, 손실 함수 및 옵티마이저 설정
cnn_model = LSTMModel().to(device)
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(cnn_model.parameters(), lr=0.001)

# EarlyStopping 콜백 설정
early_stopping_callback = callbacks.EarlyStopping(patience=10)

# 손실과 정확도를 기록할 리스트 초기화
train_losses = []
test_losses = []
accuracies = []

# 훈련 및 평가 함수 정의
def train(model, device, train_loader, optimizer, epoch):
    model.train()
    total_loss = 0
    for batch_idx, (data, target) in enumerate(train_loader):
        data, target = data.to(device), target.to(device)
        optimizer.zero_grad() # 기울기 초기화
        output = model(data) # 예측
        loss = criterion(output, target) # 손실 계산
        loss.backward() # 역전파
        optimizer.step() # 가중치 업데이트

```

```

        total_loss += loss.item()

    average_loss = total_loss / len(train_loader) # 평균 손실 반환
    return average_loss

def test(model, device, test_loader):
    model.eval()
    test_loss = 0
    correct = 0
    with torch.no_grad():
        for data, target in test_loader:
            data, target = data.to(device), target.to(device)
            output = model(data)
            test_loss += criterion(output, target).item() * data.size(0)
            # 배치 손실 합산
            pred = output.argmax(dim=1, keepdim=True)
            # 가장 높은 값을 가진 클래스 선택
            correct += pred.eq(target.view_as(pred)).sum().item()

    average_loss = test_loss / len(test_loader.dataset) # 평균 손실 계산
    accuracy = 100. * correct / len(test_loader.dataset) # 정확도 계산
    return average_loss, accuracy # 손실과 정확도 반환

# 모델 훈련 및 평가 실행
num_epochs = 5
for epoch in range(1, num_epochs + 1):
    train_loss = train(cnn_model, device, train_loader, optimizer, epoch)
    test_loss, accuracy = test(cnn_model, device, test_loader)

    # 손실과 정확도 기록
    train_losses.append(train_loss)
    test_losses.append(test_loss)
    accuracies.append(accuracy)

    # 훈련 손실, 테스트 손실 및 정확도 출력
    print(f'Epoch [{epoch}/{num_epochs}] - '
          f'Train Loss: {train_loss:.4f}, Test Loss: {test_loss:.4f}, Test Accuracy: {ac

# 훈련 손실, 테스트 손실, 정확도 리스트 출력
print("\nTraining Losses:", train_losses)
print("Test Losses:", test_losses)
print("Accuracies:", accuracies)

# 시각화 코드
# 결과 시각화
plt.figure(figsize=(12, 5))

# 손실 그래프
plt.subplot(1, 2, 1)
plt.plot(range(1, num_epochs + 1), train_losses, label='Train Loss', color='blue')
plt.plot(range(1, num_epochs + 1), test_losses, label='Test Loss', color='red')
plt.title('Loss over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Loss')
plt.legend()
plt.grid()

# 정확도 그래프
plt.subplot(1, 2, 2)

```

```
plt.plot(range(1, num_epochs + 1), accuracies, label='Accuracy', color='orange')
plt.title('Accuracy over Epochs')
plt.xlabel('Epochs')
plt.ylabel('Accuracy (%)')
plt.ylim(0, 100) # Y축 범위 설정

plt.legend()
plt.grid()

plt.tight_layout()
plt.show() # 그래프 출력
```

Epoch [1/5] - Train Loss: 0.6710, Test Loss: 0.4507, Test Accuracy: 83.70%
 Epoch [2/5] - Train Loss: 0.4106, Test Loss: 0.3925, Test Accuracy: 85.71%
 Epoch [3/5] - Train Loss: 0.3640, Test Loss: 0.3836, Test Accuracy: 86.29%
 Epoch [4/5] - Train Loss: 0.3344, Test Loss: 0.3525, Test Accuracy: 87.49%
 Epoch [5/5] - Train Loss: 0.3132, Test Loss: 0.3419, Test Accuracy: 87.27%

Training Losses: [0.6709728504834908, 0.4105805147653704, 0.3640453794649415, 0.33441797256279093, 0.3132075867148986]

Test Losses: [0.45072387919425966, 0.39253795812129977, 0.38364503514766696, 0.35248711514472963, 0.3418782120466232]

Accuracies: [83.7, 85.71, 86.29, 87.49, 87.27]

