Report

Project: Image Classification Model의 이해

과 목 명:	딥러닝입문_04
담당교수님:	조성인 교수님
학 과:	컴퓨터공학과
학 번:	2018112068
제 출 일:	2024-12-02
이 름:	민강현

1. 프로젝트 환경 구성

1-1). 구성환경

Cuda 11.4

NVIDIA GeForce GTX 1660 Super

Visual Studio Code & Python

참고 자료: [AlexNet in PyTorch CIFAR10 Clas(83% Test Accuracy)]

Requirements:

```
absl-py==2.1.0
cachetools==5.5.0
certifi==2024.8.30
charset-normalizer==3.4.0
colorama==0.4.6
contourpy==1.1.1
cycler==0.12.1
filelock==3.16.1
fonttools==4.55.0
fsspec==2024.10.0
google-auth==2.36.0
google-auth-oauthlib==1.0.0
graphviz==0.20.3
grpcio==1.68.0
idna==3.10
importlib_metadata==8.5.0
importlib_resources==6.4.5
Jinja2==3.1.4
joblib==1.4.2
kiwisolver==1.4.7
Markdown==3.7
MarkupSafe==2.1.5
matplotlib==3.7.5
mpmath==1.3.0
networkx==3.1
numpy==1.24.4
oauthlib==3.2.2
packaging==24.2
pandas==2.0.3
pillow==10.4.0
pillow==10.4.0
protobuf==5.29.0
pyasn1==0.6.1
pyasn1_modules==0.4.1
pyparsing==3.1.4
python-dateutil==2.9.0.post0
pytz==2024.2
requests==2.32.3
requests-oauthlib==2.0.0
rsa==4.9
scikit-learn==1.3.2
scipy==1.10.1
seaborn==0.13.2
six==1.16.0
sympy==1.13.3
tensorboard==2.14.0
tensorboard-data-server==0.7.2
threadpoolctl==3.5.0
torch==1.12.1+cu116
torchaudio==0.12.1+cu116
torchsummary==1.5.1
torchvision==0.13.1+cu116
torchviz==0.0.2
tqdm==4.67.1
typing_extensions==4.12.2
tzdata==2024.2
urllib3==2.2.3
Werkzeug==3.0.6
zipp==3.20.2
```

2. Cifar10 Dataset에 대해서 AlexNet 훈련을 위한 설계

2-1). Skip Connection

Skip Connection은 신경망의 특정 계층 출력을 다음 계층으로 직접 연결하는 방식으로 Residual Connection이라고 부르기도 한다. 이는 네트워크가 깊어질 때 발생하는 Gradient Vanishing 문제를 완화하고 학습 성능을 향상시키는 기법이다. 다시 말해 초기에 업데이트 되는 weight들이 결과에 큰 영향을 끼치지 못하는 것을 방지하기 위해 특정 계층의 출력에 연산 없이 더하는 방식이나 별도 연산을 추가한다. ResNet에서 보통 F(X) + X의 형태로 입력 데이터를 출력에 더한다.

```python
# Skip Connection이 포함된 AlexNet Feature Extractor
class AlexNetWithSkip(nn.Module):
    def __init__(self, pretrained_model):
        super(AlexNetWithSkip, self).__init__()
        self.features = nn.Sequential(*list(pretrained_model.features.children())[:])
        self.skip = nn.Conv2d(256, 256, kernel_size=1)  # Skip Connection Conv layer
        self.pool = nn.AdaptiveAvgPool2d((6, 6))  # Pooling layer to match dimensions

    def forward(self, x):
        skip_output = None
        for i, layer in enumerate(self.features):
            x = layer(x)
            if i == 8:  # Add skip connection at Conv5 (Layer Index 8)
                skip_output = self.pool(x)
        x = x + skip_output  # Skip Connection Added
        return x
```

Figure 1: Skip Connection

Figure1은 AlexNet에 Feature Extractor에 Skip Connection을 추가하는 코드이며, Conv5 단계의 출력을 Conv Layer(1*1)와 Adaptive Average Pooling 방식으로 변환하여 구현하였다. 해당 방식은 입력 텐서의 크기가 고정되지 않으면 출력 텐서의 크기도 다르게 나오는데 이때 발생 가능한 문제로 linear layer는 입력 크기가 고정되어 입력 사이즈의 변화로 인해 에러가 발생 할 수 있다는 것이다. 이를 방지하기 위해 입력에 관계없이 출력을 고정시키도록 Adaptive Pooling을 사용한다. skip_output을 Conv5 이후 Feature Map(x)에 더하여 Skip Connection이 활성화 되어 기울기 소실 문제를 해결할 수 있다.

2-2). AlexNet의 마지막 Classifier 부분 삭제 후 Linear Regressor 연결

사전에 학습된 AlexNet에서 마지막 Classifier 분류기 계층을 삭제하고 Linear Regressor를 연결하는 작업은 연결부분의 차원을 고려하여 설계한다. Feature Extractor는 새로운 데이터셋에 맞게 분류 계층만 훈련하여 전이 학습을 하여 Cifar10 분류 작업을 진행한다.

```python
# Linear Regressor 연결
class LinearRegressor(nn.Module):
    def __init__(self, feature_dim, output_dim=10):
        super(LinearRegressor, self).__init__()
        self.regressor = nn.Sequential(
            nn.Linear(feature_dim, 4096),
            nn.ReLU(),
            nn.Linear(4096, 1024),
            nn.ReLU(),
            nn.Linear(1024, output_dim)
        )

    def forward(self, x):
        x = x.view(x.size(0), -1)  # Flatten
        x = self.regressor(x)
        return x
```

Figure 2: Linear Regressor

기존 AlexNet의 분류기 대신 새로운 Regressor를 사용해 CIFAR-10 데이터셋에 맞는 분류기를 학습시키기 위한 코드이다. 기존 AlexNet은 1000개의 클래스로 사전 학습되었지만 CIFAR-10 데이터 셋은 10개의 클래스만 존재하므로 기존의 모델의 Classifier는 해당 데이터셋에 적합하지 않기 때문에 이와 같은 작업을 수행하는 것이다. AlexNet의 Feature Extractor 부분은 다양한 데이터셋에서 유용한 Feature들을 학습 시키기 용이하며 이를 사용하여 최종 분류단계만 1000 -> 10으로 알맞게 설계하면 바로 학습을 진행할 수 있다. 프로젝트를 진행하며 분류기와 회귀기에 대한 정의에 혼동이 와서 다시 한번 살펴보면 각각의 정의는 다음과 같다. 분류기는 입력 데이터를 특정 범주로 분류하며, 입력으로 이미지가 들어오면 출력으로 해당 이미지가 어떤 클래스인지(개, 고양이 등)를 보여주는 구조이다. 회귀기는 입력 데이터를 연속적인 실수 값으로 받고 출력 역시 연속적인 숫자의 값이 나온다. 주어진 조건으로 분류기가 아닌 회귀기를 사용하라고 하여 처음에는 조금 혼동이 있었지만 프로젝트를 진행하며 해당 회귀기가 결국 각 클래스에 해당하는 확률을 출력하는 것으로 정리할 수 있었고, 마지막 단계의 출력 크기가 CIFAR-10 클래스의 개수(10)임을 고려하며 진행하였다.

2-3). PCA Embedding Space 분석

PCA를 사용해 모델이 생성한 고차원 Feature Map을 2차원으로 축소하여 시각화한다. Feature Extractor의 출력 값(9216차원)을 2차원으로 축소하며 각 클래스에 대해 다른 색을 부여하여 클래스간의 분포를 확인할 수 있다.

```python
# PCA 구현 함수
def perform_pca(features, n_components=2):
    # Data Centering
    mean = np.mean(features, axis=0)
    centered_data = features - mean
    # Covariance Matrix 계산
    covariance_matrix = np.cov(centered_data, rowvar=False)
    # Eigenvalues, Eigenvectors 계산
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)
    # Principal Components 선택
    sorted_indices = np.argsort(eigenvalues)[::-1]
    top_eigenvectors = eigenvectors[:, sorted_indices[:n_components]]
    # Embedding Space로 변환
    reduced_data = np.dot(centered_data, top_eigenvectors)
    return reduced_data, top_eigenvectors
# Feature 추출 및 PCA 수행
features, labels = extract_features(feature_extractor, testloader, device)
reduced_data, _ = perform_pca(features, n_components=2)

# PCA 결과 시각화
plt.figure(figsize=(10, 8))
for i, class_name in enumerate(classes):
    idx = labels == i
    plt.scatter(reduced_data[idx, 0], reduced_data[idx, 1], label=class_name,
plt.title("PCA Embedding Space")
plt.xlabel("Principal Component 1")
plt.ylabel("Principal Component 2")
plt.legend()
plt.show()
```

Figure 3: PCA 구현

직접 PCA를 구현하기 위해 perform_pca 함수를 생성하여 입력 데이터의 중심화, 공분산 행렬 계산, 고유 값 및 고유벡터 계산에 차원 축소까지 진행하였다. Extract_Feature에서 추출한 feature들과 label을 사용하여 PCA를 수행하였다.

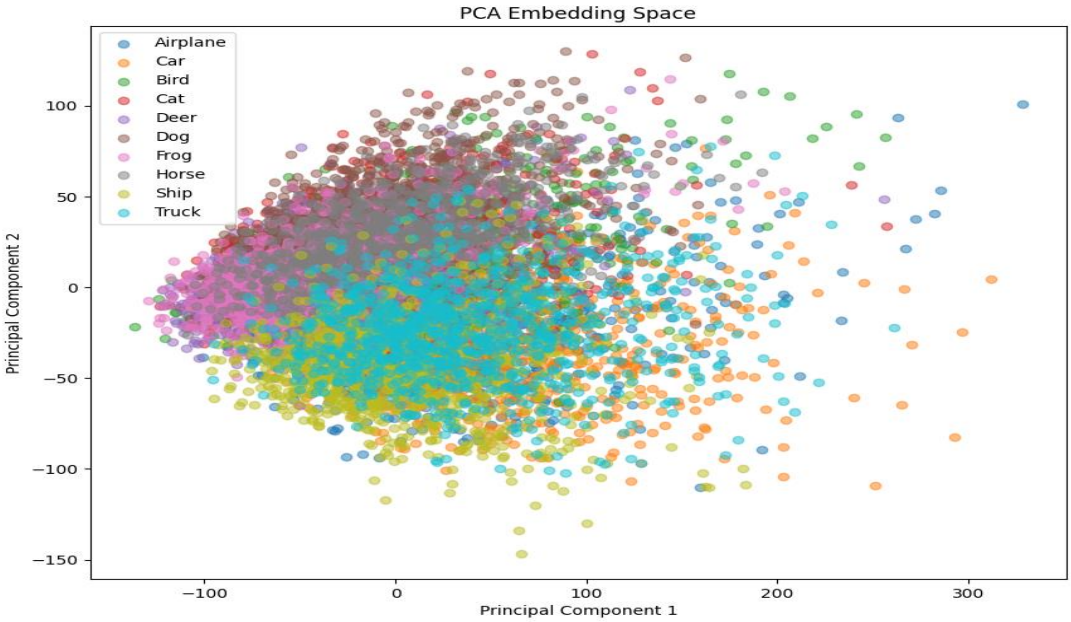

Figure 4: PCA 2D 결과

고차원 데이터를 2차원 데이터로 축소하여 표현하였지만 구분이 잘 되지 않는 것을 확인하였다. 데이터의 복잡성 문제와 분산 유지의 한계 등의 이유가 예상되며 3D 공간으로 확장한다면 Principal Component 3가 추가되어 클래스간의 구분이 더 명확해질 것으로 예상된다.

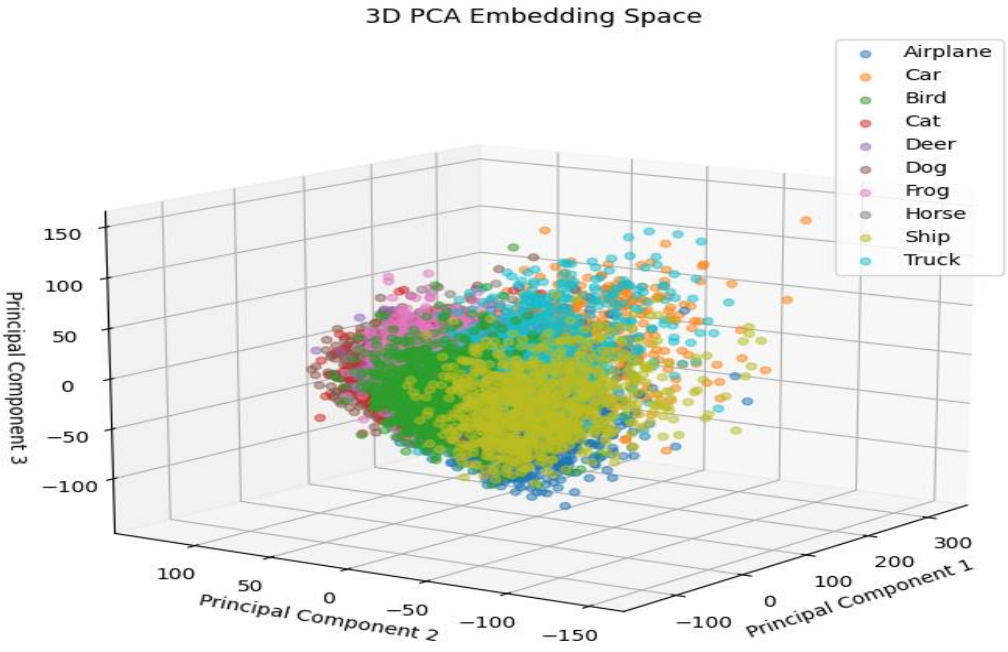

Figure 5: PCA 3D 결과

3 차원으로 Embedding Space 를 표현하였고 각도를 이리저리 돌려보며 확인한 결과 기존의 2 차원으로 확인한 것보다 조금 더 구분이 명확함을 확인할 수 있었지만 추가적인 개선이 더 필요해 보인다. 각 데이터간의 거리를 조금 더 늘리는 방향으로 구현한다면 각 클래스간의 구분이 더 잘 될 것으로 보인다.

2-4). 훈련 과정 및 결과

Adam을 사용하여 학습속도가 빠르고 동적 학습률 조정이 가능하게 하였고 초기의 학습률은 0.001로 부여하였다.

```
# 손실 함수 및 옵티마이저
criterion = nn.CrossEntropyLoss()
optimizer = torch.optim.Adam(regressor.parameters(), lr=0.001)
```

Figure 6: 하이퍼파라미터 1(Optimizer)

Batch Size는 너무 크거나 작으면 학습에 문제가 생길 수 있으므로 최적의 값이 나올 때까지 반복적으로 수정하며 작업하였다. 처음엔 32, 64, 128까지 해봤으며 기기의 성능으로 인해 너무 높은 값을 주면 모델이 중간에 멈추는 현상이 있을 수 있음을 고려하여 32로 진행하였다.

```
trainloader = torch.utils.data.DataLoader(train_data, batch_size=32, shuffle=True, num_workers=2)
testloader = torch.utils.data.DataLoader(test_data, batch_size=32, shuffle=False, num_workers=2)
```

Figure 7 하이퍼파라미터 2(Batch_size)

Epoch 역시 하드웨어의 성능에 따라 값이 너무 크면 모델 학습의 시간이 매우 오래 걸림을 감안하여 학습을 진행하였다. 보통 AlexNet을 학습시킬 때 사용하는 단위가 100~200임을 고려하여 100으로 설정하였고, 정확도면에서 충분히 만족할 만한 결과를 확인하였다.

```
# 학습 루프
def train_model(feature_extractor, regressor, trainloader, epochs=100):
    feature_extractor.eval()  # Feature Extractor 고정
    for epoch in range(epochs):
        regressor.train()
        running_loss = 0.0
        for inputs, labels in trainloader:
            inputs, labels = inputs.to(device), labels.to(device)
            optimizer.zero_grad()
            with torch.no_grad():
                features = feature_extractor(inputs)  # Feature Extractor 통과
            outputs = regressor(features)
            loss = criterion(outputs, labels)
            loss.backward()
            optimizer.step()
            running_loss += loss.item()
        print(f"Epoch {epoch + 1}, Loss: {running_loss / len(trainloader):.4f}")
```

Figure 8 하이퍼파라미터 3(Epoch)

아래는 테스트 결과를 Confusion Matrix와 Top1,3 Accuracy를 정리한 결과이다. Confusion Matrix에서 클래스별 예측 정확도를 시각적으로 보여주며 대부분의 경우 알맞게 예측한 것을 볼 수 있고 Accuracy Table에서 확인할 수 있듯이 80%정도라는 준수한 결과가 나오게 되었다.

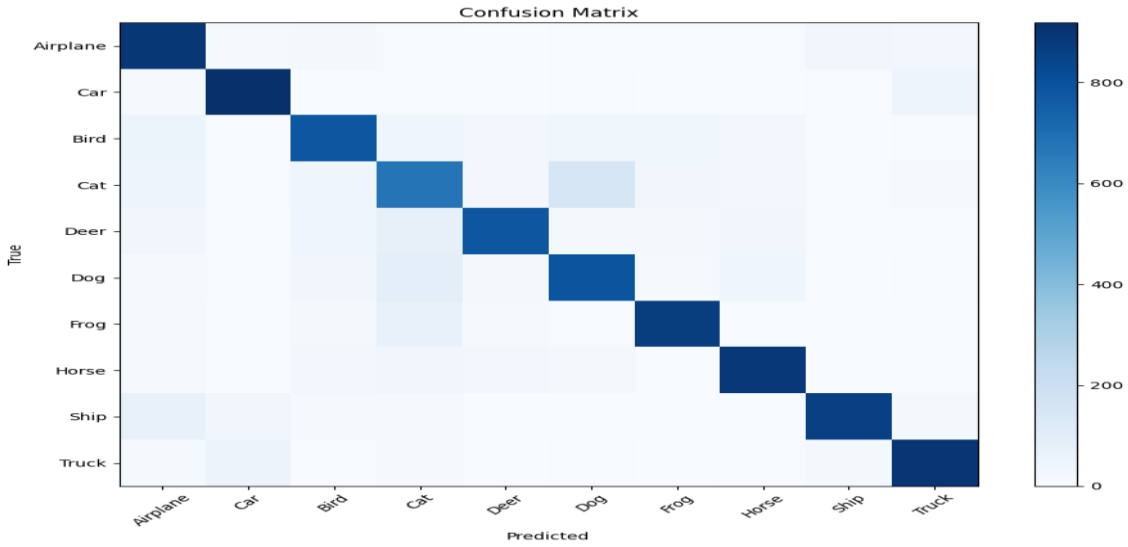

Figure 9 Confusion Matrix

	Without Skip Connection	With Skip Connection
Low Epoch	Top-1 Accuracy: 0.7865 Top-3 Accuracy: 0.9483 Confusion Matrix: [[902 1 8 8 24 1 2 14 37 3] [103 640 7 8 6 1 8 4 92 131] [84 1 655 33 133 16 50 22 6 0] [21 1 36 632 107 88 46 46 22 1] [9 0 16 16 893 8 18 35 5 0] [5 1 23 151 61 673 15 63 4 4] [9 0 21 26 49 6 877 4 8 0] [6 0 6 15 86 19 2 864 1 1] [77 0 3 3 8 0 3 2 901 3] [53 13 3 16 13 0 4 11 59 828]]	Top-1 Accuracy: 0.7865 Top-3 Accuracy: 0.9483 Confusion Matrix: [[902 1 8 8 24 1 2 14 37 3] [103 640 7 8 6 1 8 4 92 131] [84 1 655 33 133 16 50 22 6 0] [21 1 36 632 107 88 46 46 22 1] [9 0 16 16 893 8 18 35 5 0] [5 1 23 151 61 673 15 63 4 4] [9 0 21 26 49 6 877 4 8 0] [6 0 6 15 86 19 2 864 1 1] [77 0 3 3 8 0 3 2 901 3] [53 13 3 16 13 0 4 11 59 828]]
High Epoch	Top-1 Accuracy: 0.8345 Top-3 Accuracy: 0.9683 Confusion Matrix: [[890 12 18 7 3 4 4 5 31 26] [13 918 1 6 2 5 3 2 5 45] [50 1 780 43 27 33 35 22 3 6] [44 4 37 676 27 144 29 26 5 8] [29 1 41 74 781 17 20 32 3 2] [14 4 29 93 17 788 11 40 0 4] [13 2 20 70 9 7 871 5 0 3] [12 1 23 32 24 18 0 888 1 1] [67 29 9 9 1 1 2 6 858 18] [13 53 2 10 0 4 1 7 15 895]]	Top-1 Accuracy: 0.8357 Top-3 Accuracy: 0.9672 Confusion Matrix: [[876 8 13 29 2 3 1 10 30 28] [4 904 2 12 0 0 0 1 5 72] [43 2 778 81 29 28 10 15 10 4] [16 4 15 829 14 80 12 13 6 11] [20 1 17 96 773 39 5 44 2 3] [6 1 26 173 14 750 1 21 2 6] [5 4 28 131 12 15 793 4 5 3] [7 1 6 56 29 23 0 867 1 10] [55 31 6 14 1 1 0 0 872 20] [13 30 3 13 0 3 1 3 19 915]]

Accuracy Table