딥러닝

- 인공신경망(Artificial Neural Network)을 기반으로 하는 머신러닝의 한 분야
- 여러 계층(Layer)을 쌓아 올린 '깊은(Deep)' 신경망을 통해 데이터 내의 복잡한 패턴을 학습하고 특징을 추출하는 기술
- 이미지, 음성, 텍스트와 같은 비정형 데이터 처리에서 뛰어난 성능을 보임
- 인간의 개입 없이 데이터로부터 직접 특징을 학습하는 능력이 강점

적용 가능한 상황

- **이미지 및 비디오 처리**: 이미지 분류, 객체 탐지, 얼굴 인식, 비디오 분석 등.
- **자연어 처리 (NLP)**: 기계 번역, 텍스트 분류, 감성 분석, 챗봇, 음성 인식 등.
- **음성 처리**: 음성 인식, 음성 합성, 화자 인식 등.
- **추천 시스템**: 사용자 행동 패턴 분석을 통한 맞춤형 추천.
- **시계열 데이터 분석**: 주가 예측, 센서 데이터 분석 등.
- **강화 학습**: 게임 플레이, 로봇 제어 등.

프레임워크: PyTorch, Keras/TensorFlow

- **Keras/TensorFlow**: 구글에서 개발한 딥러닝 프레임워크
  - Keras는 TensorFlow의 고수준 API로, 직관적이고 사용하기 쉬운 인터페이스를 제공하여 빠른 프로토타이핑에 강점
  - TensorFlow는 저수준의 유연한 API를 제공하여 복잡한 모델 구축 및 연구에 적합
  - **장점**: 쉬운 사용법, 풍부한 문서와 커뮤니티, 다양한 사전 학습 모델, 분산 학습 용이.
  - **단점**: 저수준 제어가 필요한 경우 복잡할 수 있음, 디버깅이 PyTorch보다 어려울 수 있음.
- **PyTorch**: Facebook(Meta)에서 개발한 딥러닝 프레임워크
  - 동적 계산 그래프(Dynamic Computation Graph)를 지원하여 유연한 모델 구축과 디버깅이 용이
  - 연구 분야에서 특히 인기가 많음
  - **장점**: 높은 유연성, 쉬운 디버깅, 파이썬 친화적, 연구 개발에 적합.
  - **단점**: Keras에 비해 학습 곡선이 다소 높을 수 있음, 배포(Deployment) 측면에서 TensorFlow보다 복잡할 수 있음.

1. DNN (Deep Neural Network)

- 다층 퍼셉트론(Multi-Layer Perceptron, MLP)의 확장된 형태
- 입력층(Input Layer)과 출력층(Output Layer) 사이에 여러 개의 은닉층(Hidden Layer)을 포함하는 신경망
- 각 층의 뉴런들은 이전 층의 모든 뉴런과 연결되어 있으며, 비선형 활성화 함수를 통해 복잡한 패턴을 학습

구축 방식

- **Keras (Sequential API)**: 가장 간단한 방식으로, 층을 순차적으로 쌓아 올립니다.
- **Keras (Functional API)**: 복잡한 모델 구조(다중 입력/출력, 공유 층 등)를 유연하게 구축할 수 있습니다.
- **PyTorch (`nn.Sequential` 및 클래스 기반)**: `nn.Sequential`은 간단한 순차적 모델에 사용되며, 복잡한 모델은 `nn.Module`을 상속받는 클래스 형태로 정의합니다.

문제에 따른 최종 레이어 구성

- **회귀 (Regression)**: 출력층에 활성화 함수 없이 단일 뉴런을 사용합니다. (예: `Dense(1)`) 또는 `activation='linear'`
- **이중 분류 (Binary Classification)**: 출력층에 시그모이드(Sigmoid) 활성화 함수를 가진 단일 뉴런을 사용합니다. (예: `Dense(1, activation='sigmoid')`)
- **다중 분류 (Multi-class Classification)**: 출력층에 소프트맥스(Softmax) 활성화 함수를 가진 클래스 개수 만큼의 뉴런을 사용합니다. (예: `Dense(num_classes, activation='softmax')`)

코드 예시 (Keras)

```
import tensorflow as tf
from tensorflow.keras import layers, models
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification, make_regression

# 1. 데이터 준비 (분류 예시)
X_clf, y_clf = make_classification(n_samples=1000, n_features=20,
n_informative=10, n_redundant=5, random_state=42)
X_clf_train, X_clf_test, y_clf_train, y_clf_test = train_test_split(X_clf, y_clf,
test_size=0.2, random_state=42)
scaler = StandardScaler()
X_clf_train_scaled = scaler.fit_transform(X_clf_train)
X_clf_test_scaled = scaler.transform(X_clf_test)

# 2. DNN 모델 구축 (이중 분류)
model_dnn_keras = models.Sequential([
    layers.Dense(64, activation='relu', input_shape=
(X_clf_train_scaled.shape[1],)),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='sigmoid') # 이중 분류
])

# 3. 모델 컴파일 및 학습
model_dnn_keras.compile(optimizer='adam', loss='binary_crossentropy', metrics=
['accuracy'])
model_dnn_keras.fit(X_clf_train_scaled, y_clf_train, epochs=10, batch_size=32,
verbose=0)
loss, accuracy = model_dnn_keras.evaluate(X_clf_test_scaled, y_clf_test,
verbose=0)
print(f"Keras DNN 이중 분류 정확도: {accuracy:.4f}") # 0.9300

# 4. 데이터 준비 (회귀 예시)
X_reg, y_reg = make_regression(n_samples=1000, n_features=20, n_informative=10,
random_state=42)
X_reg_train, X_reg_test, y_reg_train, y_reg_test = train_test_split(X_reg, y_reg,
test_size=0.2, random_state=42)
scaler_reg_X = StandardScaler()
X_reg_train_scaled = scaler_reg_X.fit_transform(X_reg_train)
X_reg_test_scaled = scaler_reg_X.transform(X_reg_test)
scaler_reg_y = StandardScaler()
y_reg_train_scaled = scaler_reg_y.fit_transform(y_reg_train.reshape(-1, 1))
```

```
y_reg_test_scaled = scaler_reg_y.transform(y_reg_test.reshape(-1, 1))

# 5. DNN 모델 구축 (회귀)
model_dnn_reg_keras = models.Sequential([
    layers.Dense(64, activation='relu', input_shape=
(X_reg_train_scaled.shape[1],)),
    layers.Dense(32, activation='relu'),
    layers.Dense(1, activation='linear') # 회귀
])

# 6. 모델 컴파일 및 학습
model_dnn_reg_keras.compile(optimizer='adam', loss='mse')
model_dnn_reg_keras.fit(X_reg_train_scaled, y_reg_train_scaled, epochs=10,
batch_size=32, verbose=0)
loss_reg = model_dnn_reg_keras.evaluate(X_reg_test_scaled, y_reg_test_scaled,
verbose=0)
print(f"Keras DNN 회귀 MSE: {loss_reg:.4f}") # 0.0315
```

코드 예시 (PyTorch)

```
import torch
import torch.nn as nn
import torch.optim as optim
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import StandardScaler
from sklearn.datasets import make_classification, make_regression
from torch.utils.data import DataLoader, TensorDataset

# 1. 데이터 준비 (분류 예시)
X_clf, y_clf = make_classification(n_samples=1000, n_features=20,
n_informative=10, n_redundant=5, random_state=42)
X_clf_train, X_clf_test, y_clf_train, y_clf_test = train_test_split(X_clf, y_clf,
test_size=0.2, random_state=42)
scaler = StandardScaler()
X_clf_train_scaled = scaler.fit_transform(X_clf_train)
X_clf_test_scaled = scaler.transform(X_clf_test)

X_clf_train_tensor = torch.tensor(X_clf_train_scaled, dtype=torch.float32)
y_clf_train_tensor = torch.tensor(y_clf_train, dtype=torch.float32).reshape(-1, 1)
X_clf_test_tensor = torch.tensor(X_clf_test_scaled, dtype=torch.float32)
y_clf_test_tensor = torch.tensor(y_clf_test, dtype=torch.float32).reshape(-1, 1)

train_dataset_clf = TensorDataset(X_clf_train_tensor, y_clf_train_tensor)
train_loader_clf = DataLoader(train_dataset_clf, batch_size=32, shuffle=True)

# 2. DNN 모델 구축 (이중 분류)
class SimpleDNN(nn.Module):
    def __init__(self, input_dim):
        super(SimpleDNN, self).__init__()
        self.layer1 = nn.Linear(input_dim, 64)
        self.relu = nn.ReLU()
```

```
        self.layer2 = nn.Linear(64, 32)
        self.layer3 = nn.Linear(32, 1)
        self.sigmoid = nn.Sigmoid() # 이중 분류

    def forward(self, x):
        x = self.relu(self.layer1(x))
        x = self.relu(self.layer2(x))
        x = self.sigmoid(self.layer3(x))
        return x

model_dnn_pytorch = SimpleDNN(X_clf_train_scaled.shape[1])

# 3. 손실 함수 및 옵티마이저 설정
criterion_clf = nn.BCELoss() # 이진 교차 엔트로피
optimizer_clf = optim.Adam(model_dnn_pytorch.parameters(), lr=0.001)

# 4. 모델 학습
for epoch in range(10):
    for inputs, labels in train_loader_clf:
        optimizer_clf.zero_grad()
        outputs = model_dnn_pytorch(inputs)
        loss = criterion_clf(outputs, labels)
        loss.backward()
        optimizer_clf.step()

# 5. 모델 평가
with torch.no_grad():
    y_pred_proba = model_dnn_pytorch(X_clf_test_tensor)
    y_pred = (y_pred_proba >= 0.5).float()
    accuracy = (y_pred == y_clf_test_tensor).float().mean()
    print(f"PyTorch DNN 이중 분류 정확도: {accuracy.item():.4f}") # 0.9250
```

2. CNN (Convolutional Neural Network)

- 이미지와 같은 그리드 형태의 데이터를 처리하는 데 특화된 신경망
- 컨볼루션 필터(커널)를 사용하여 이미지의 공간적 특징(엣지, 질감, 패턴 등)을 추출하고, 풀링 계층을 통해 특징 맵의 크기를 줄여 계산량을 감소시키고 중요한 특징을 강조
- 여러 컨볼루션 및 풀링 계층을 거쳐 추출된 특징들은 최종적으로 완전 연결 계층을 통해 분류 또는 회귀를 수행

구축 방식

- **Keras (Sequential/Functional API)**: `Conv2D`, `MaxPooling2D` 등의 층을 사용하여 구축합니다.
- **PyTorch (클래스 기반)**: `nn.Conv2d`, `nn.MaxPool2d` 등의 모듈을 사용하여 `nn.Module`을 상속받는 클래스 형태로 정의합니다.

문제에 따른 최종 레이어 구성

- **이미지 분류**: 출력층에 소프트맥스(Softmax) 활성화 함수를 가진 클래스 개수만큼의 뉴런을 사용합니다. (예: `Dense(num_classes, activation='softmax')`)

코드 예시 (Keras)

```
import tensorflow as tf
from tensorflow.keras import layers, models
from tensorflow.keras.datasets import mnist
from tensorflow.keras.utils import to_categorical

# 1. 데이터 로드 및 전처리
(train_images, train_labels), (test_images, test_labels) = mnist.load_data()

train_images = train_images.reshape((60000, 28, 28, 1)).astype('float32') / 255
test_images = test_images.reshape((10000, 28, 28, 1)).astype('float32') / 255

train_labels = to_categorical(train_labels)
test_labels = to_categorical(test_labels)

# 2. CNN 모델 구축
model_cnn_keras = models.Sequential([
    layers.Conv2D(32, (3, 3), activation='relu', input_shape=(28, 28, 1)),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.MaxPooling2D((2, 2)),
    layers.Conv2D(64, (3, 3), activation='relu'),
    layers.Flatten(),
    layers.Dense(64, activation='relu'),
    layers.Dense(10, activation='softmax') # 10개 클래스 분류
])

# 3. 모델 컴파일 및 학습
model_cnn_keras.compile(optimizer='adam', loss='categorical_crossentropy',
metrics=['accuracy'])
model_cnn_keras.fit(train_images, train_labels, epochs=5, batch_size=64,
verbose=0)
loss, accuracy = model_cnn_keras.evaluate(test_images, test_labels, verbose=0)
print(f"Keras CNN 분류 정확도: {accuracy:.4f}") # 0.9905
```

코드 예시 (PyTorch)

```
import torch
import torch.nn as nn
import torch.optim as optim
from torchvision import datasets, transforms
from torch.utils.data import DataLoader

# 1. 데이터 로드 및 전처리
transform = transforms.Compose([
    transforms.ToTensor(),
    transforms.Normalize((0.1307,), (0.3081,))
])
```

```
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)

# 2. CNN 모델 구축
class SimpleCNN_PyTorch(nn.Module):
    def __init__(self):
        super(SimpleCNN_PyTorch, self).__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(kernel_size=2, stride=2)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.fc1 = nn.Linear(64 * 7 * 7, 10) # MNIST 이미지 크기 28x28 -> 풀링 2번
후 7x7

    def forward(self, x):
        x = self.pool(torch.relu(self.conv1(x)))
        x = self.pool(torch.relu(self.conv2(x)))
        x = x.view(-1, 64 * 7 * 7) # Flatten
        x = self.fc1(x)
        return x

model_cnn_pytorch = SimpleCNN_PyTorch()

# 3. 손실 함수 및 옵티마이저 설정
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model_cnn_pytorch.parameters(), lr=0.001)

# 4. 모델 학습
for epoch in range(5):
    for inputs, labels in train_loader:
        optimizer.zero_grad()
        outputs = model_cnn_pytorch(inputs)
        loss = criterion(outputs, labels)
        loss.backward()
        optimizer.step()

# 5. 모델 평가
with torch.no_grad():
    correct = 0
    total = 0
    for images, labels in test_loader:
        outputs = model_cnn_pytorch(images)
        _, predicted = torch.max(outputs.data, 1)
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
    print(f'PyTorch CNN 분류 정확도: {100 * correct / total:.2f}%')
```

3. RNN (Recurrent Neural Network)

- 시퀀스(Sequence) 데이터를 처리하는 데 특화된 신경망
- 내부적으로 '순환(Recurrent)' 연결을 가지고 있어 이전 시점의 정보를 현재 시점의 예측에 활용 가능
- 시계열 데이터, 자연어, 음성 등 시간적 또는 순서적 의존성을 가지는 데이터의 패턴을 학습하는데 적합

구축 방식

- **Keras (Sequential/Functional API)**: `SimpleRNN`, `LSTM`, `GRU` 등의 층을 사용합니다.
- **PyTorch (클래스 기반)**: `nn.RNN`, `nn.LSTM`, `nn.GRU` 등의 모듈을 사용합니다.

문제에 따른 최종 레이어 구성

- **시퀀스 예측 (Regression)**: 출력층에 활성화 함수 없이 단일 뉴런을 사용합니다. (예: `Dense(1)`)
- **시퀀스 분류 (Classification)**: 출력층에 시그모이드(이중 분류) 또는 소프트맥스(다중 분류) 활성화 함수를 가진 뉴런을 사용합니다.

코드 예시 (Keras)

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

# 1. 데이터 준비 (간단한 시퀀스 예측 예시)
# 시퀀스 데이터: [0, 1, 2, 3] -> 4 예측
def create_sequences(data, seq_length):
    xs, ys = [], []
    for i in range(len(data) - seq_length):
        xs.append(data[i:(i + seq_length)])
        ys.append(data[i + seq_length])
    return np.array(xs), np.array(ys)

data = np.arange(0, 100)
seq_length = 10
X_seq, y_seq = create_sequences(data, seq_length)

X_seq = X_seq.reshape(-1, seq_length, 1) # (samples, timesteps, features)

# 2. RNN 모델 구축
model_rnn_keras = models.Sequential([
    layers.SimpleRNN(32, activation='relu', input_shape=(seq_length, 1)),
    layers.Dense(1) # 회귀 예측
])

# 3. 모델 컴파일 및 학습
model_rnn_keras.compile(optimizer='adam', loss='mse')
model_rnn_keras.fit(X_seq, y_seq, epochs=10, verbose=0)

# 4. 예측
new_sequence = np.array([90, 91, 92, 93, 94, 95, 96, 97, 98, 99]).reshape(1,
seq_length, 1)
predicted_value = model_rnn_keras.predict(new_sequence)[0][0]
print(f"Keras RNN 예측: {predicted_value:.2f}")
```

코드 예시 (PyTorch)

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# 1. 데이터 준비 (간단한 시퀀스 예측 예시)
def create_sequences_torch(data, seq_length):
    xs, ys = [], []
    for i in range(len(data) - seq_length):
        xs.append(data[i:(i + seq_length)])
        ys.append(data[i + seq_length])
    return torch.tensor(np.array(xs), dtype=torch.float32).unsqueeze(-1), \
           torch.tensor(np.array(ys), dtype=torch.float32).unsqueeze(-1)

data = np.arange(0, 100)
seq_length = 10
X_seq_torch, y_seq_torch = create_sequences_torch(data, seq_length)

# 2. RNN 모델 구축
class SimpleRNN_PyTorch(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleRNN_PyTorch, self).__init__()
        self.hidden_size = hidden_size
        self.rnn = nn.RNN(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device) #
(num_layers * num_directions, batch, hidden_size)
        out, _ = self.rnn(x, h0)
        out = self.fc(out[:, -1, :]) # 마지막 시점의 은닉 상태 사용
        return out

model_rnn_pytorch = SimpleRNN_PyTorch(input_size=1, hidden_size=32, output_size=1)

# 3. 손실 함수 및 옵티마이저 설정
criterion = nn.MSELoss()
optimizer = optim.Adam(model_rnn_pytorch.parameters(), lr=0.01)

# 4. 모델 학습
for epoch in range(10):
    optimizer.zero_grad()
    outputs = model_rnn_pytorch(X_seq_torch)
    loss = criterion(outputs, y_seq_torch)
    loss.backward()
    optimizer.step()

# 5. 예측
new_sequence_torch = torch.tensor(np.array([90, 91, 92, 93, 94, 95, 96, 97, 98,
```

```
99]), dtype=torch.float32).reshape(1, seq_length, 1)
with torch.no_grad():
    predicted_value = model_rnn_pytorch(new_sequence_torch).item()
print(f"PyTorch RNN 예측: {predicted_value:.2f}")
```

4. LSTM (Long Short-Term Memory)

- RNN의 한 종류로, 장기 의존성(Long-Term Dependency) 문제를 해결하기 위해 고안
- '셀 상태(Cell State)'와 '게이트(Gate)' 메커니즘(입력 게이트, 망각 게이트, 출력 게이트)을 통해 정보를 선택적으로 기억하거나 잊어버리면서 먼 과거의 정보도 효과적으로 학습 가능
- 긴 시퀀스 데이터에서 RNN보다 훨씬 뛰어난 성능

구축 방식

- **Keras (Sequential/Functional API)**: `LSTM` 층을 사용합니다.
- **PyTorch (클래스 기반)**: `nn.LSTM` 모듈을 사용합니다.

문제에 따른 최종 레이어 구성

- RNN과 동일하게 시퀀스 예측 또는 시퀀스 분류에 따라 최종 레이어를 구성합니다.

코드 예시 (Keras)

```
import tensorflow as tf
from tensorflow.keras import layers, models
import numpy as np

# 1. 데이터 준비 (RNN 예시와 동일)
def create_sequences(data, seq_length):
    xs, ys = [], []
    for i in range(len(data) - seq_length):
        xs.append(data[i:(i + seq_length)])
        ys.append(data[i + seq_length])
    return np.array(xs), np.array(ys)

data = np.arange(0, 100)
seq_length = 10
X_seq, y_seq = create_sequences(data, seq_length)

X_seq = X_seq.reshape(-1, seq_length, 1) # (samples, timesteps, features)

# 2. LSTM 모델 구축
model_lstm_keras = models.Sequential([
    layers.LSTM(32, activation='relu', input_shape=(seq_length, 1)),
    layers.Dense(1) # 회귀 예측
])

# 3. 모델 컴파일 및 학습
model_lstm_keras.compile(optimizer='adam', loss='mse')
model_lstm_keras.fit(X_seq, y_seq, epochs=10, verbose=0)
```

```
# 4. 예측
new_sequence = np.array([90, 91, 92, 93, 94, 95, 96, 97, 98, 99]).reshape(1,
seq_length, 1)
predicted_value = model_lstm_keras.predict(new_sequence)[0][0]
print(f"Keras LSTM 예측: {predicted_value:.2f}")
```

코드 예시 (PyTorch)

```
import torch
import torch.nn as nn
import torch.optim as optim
import numpy as np

# 1. 데이터 준비 (RNN 예시와 동일)
def create_sequences_torch(data, seq_length):
    xs, ys = [], []
    for i in range(len(data) - seq_length):
        xs.append(data[i:(i + seq_length)])
        ys.append(data[i + seq_length])
    return torch.tensor(np.array(xs), dtype=torch.float32).unsqueeze(-1), \
           torch.tensor(np.array(ys), dtype=torch.float32).unsqueeze(-1)

data = np.arange(0, 100)
seq_length = 10
X_seq_torch, y_seq_torch = create_sequences_torch(data, seq_length)

# 2. LSTM 모델 구축
class SimpleLSTM_PyTorch(nn.Module):
    def __init__(self, input_size, hidden_size, output_size):
        super(SimpleLSTM_PyTorch, self).__init__()
        self.hidden_size = hidden_size
        self.lstm = nn.LSTM(input_size, hidden_size, batch_first=True)
        self.fc = nn.Linear(hidden_size, output_size)

    def forward(self, x):
        h0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)
        c0 = torch.zeros(1, x.size(0), self.hidden_size).to(x.device)
        out, _ = self.lstm(x, (h0, c0))
        out = self.fc(out[:, -1, :]) # 마지막 시점의 은닉 상태 사용
        return out

model_lstm_pytorch = SimpleLSTM_PyTorch(input_size=1, hidden_size=32,
output_size=1)

# 3. 손실 함수 및 옵티마이저 설정
criterion = nn.MSELoss()
optimizer = optim.Adam(model_lstm_pytorch.parameters(), lr=0.01)

# 4. 모델 학습
for epoch in range(10):
```

```
    optimizer.zero_grad()
    outputs = model_lstm_pytorch(X_seq_torch)
    loss = criterion(outputs, y_seq_torch)
    loss.backward()
    optimizer.step()

# 5. 예측
new_sequence_torch = torch.tensor(np.array([90, 91, 92, 93, 94, 95, 96, 97, 98,
99]), dtype=torch.float32).reshape(1, seq_length, 1)
with torch.no_grad():
    predicted_value = model_lstm_pytorch(new_sequence_torch).item()
print(f"PyTorch LSTM 예측: {predicted_value:.2f}")
```

구축한 모델 시각화하는 방법

Keras

- **모델 구조 시각화**: `tf.keras.utils.plot_model` 함수를 사용하여 모델의 층 구조, 입력/출력 형태 등을 그래프로 시각화할 수 있습니다.
- **학습 과정 시각화**: `model.fit()` 메서드가 반환하는 `history` 객체에는 에포크(epoch)별 손실(loss)과 정확도(accuracy) 등의 학습 지표가 저장되어 있습니다. 이를 `matplotlib`을 이용하여 그래프로 그릴 수 있습니다.

PyTorch

- **모델 구조 시각화**: `torchsummary` 라이브러리(`pip install torchsummary`)의 `summary()` 함수를 사용하여 모델의 층별 출력 형태, 파라미터 수 등을 확인할 수 있습니다.
- **학습 과정 시각화**: 학습 루프에서 에포크별 손실과 정확도를 리스트에 저장한 후, `matplotlib`을 이용하여 그래프로 그릴 수 있습니다.

모델 학습 및 평가 예제 코드

1. 가중치 업데이트 (PyTorch)

PyTorch에서는 옵티마이저를 통해 가중치를 업데이트합니다. 각 학습 스텝마다 다음 과정을 거칩니다.

1. `optimizer.zero_grad()`: 이전 스텝에서 계산된 기울기를 0으로 초기화합니다.
2. `loss.backward()`: 역전파(Backpropagation)를 통해 손실 함수에 대한 각 파라미터의 기울기를 계산합니다.
3. `optimizer.step()`: 계산된 기울기를 사용하여 파라미터(가중치, 편향)를 업데이트합니다.

```
# PyTorch 학습 루프 내에서
optimizer.zero_grad()
outputs = model(inputs)
loss = criterion(outputs, labels)
loss.backward()
optimizer.step()
```

2. 성능 평가 코드

Keras

```
model.evaluate(test_data, test_labels, verbose=2)
# 반환값: [loss, metric1, metric2, ...]
```

PyTorch

```
# 모델을 평가 모드로 전환
model.eval()
correct = 0
total = 0
with torch.no_grad(): # 기울기 계산 비활성화
    for images, labels in test_loader:
        outputs = model(images)
        _, predicted = torch.max(outputs.data, 1) # 가장 높은 확률을 가진 클래스 선
택
        total += labels.size(0)
        correct += (predicted == labels).sum().item()
accuracy = 100 * correct / total
print(f'테스트 정확도: {accuracy:.2f}%')
```

3. 추론 코드

Keras

```
predictions = model.predict(new_data)
# 이진 분류: (predictions > 0.5).astype(int)
# 다중 분류: np.argmax(predictions, axis=1)
```

PyTorch

```
# 모델을 평가 모드로 전환
model.eval()
with torch.no_grad():
    outputs = model(new_data_tensor)
    # 이진 분류: predicted_class = (torch.sigmoid(outputs) >= 0.5).float()
    # 다중 분류: _, predicted_class = torch.max(outputs.data, 1)
```

4. 모델 저장 방법

Keras

```
# 모델 구조와 가중치를 HDF5 파일로 저장
model.save('my_model.h5')

# 모델 로드
loaded_model = tf.keras.models.load_model('my_model.h5')
```

PyTorch

```
# 모델의 상태 사전(state_dict)만 저장 (권장)
torch.save(model.state_dict(), 'my_model_weights.pth')

# 모델 로드 (모델 클래스 정의 후 가중치 로드)
loaded_model = MyModelClass(*args, **kwargs)
loaded_model.load_state_dict(torch.load('my_model_weights.pth'))
loaded_model.eval() # 추론 시 평가 모드로 전환

# 전체 모델 저장 (모델 구조와 가중치 모두 저장)
torch.save(model, 'my_model.pth')
loaded_model = torch.load('my_model.pth')
```

5. 학습 과정 시각화

`matplotlib`을 사용하여 학습 중 기록된 손실(loss)과 정확도(accuracy) 등의 지표를 그래프로 시각화하여 모델의 학습 진행 상황과 과적합 여부를 판단할 수 있습니다.

```
import matplotlib.pyplot as plt

# Keras의 history 객체 사용 예시
plt.plot(history.history['accuracy'], label='accuracy')
plt.plot(history.history['val_accuracy'], label='val_accuracy')
plt.xlabel('Epoch')
plt.ylabel('Accuracy')
plt.legend()
plt.show()

# PyTorch에서 리스트에 저장된 값 사용 예시
plt.plot(train_losses, label='Train Loss')
plt.plot(val_losses, label='Validation Loss')
plt.xlabel('Epoch')
plt.ylabel('Loss')
plt.legend()
plt.show()
```