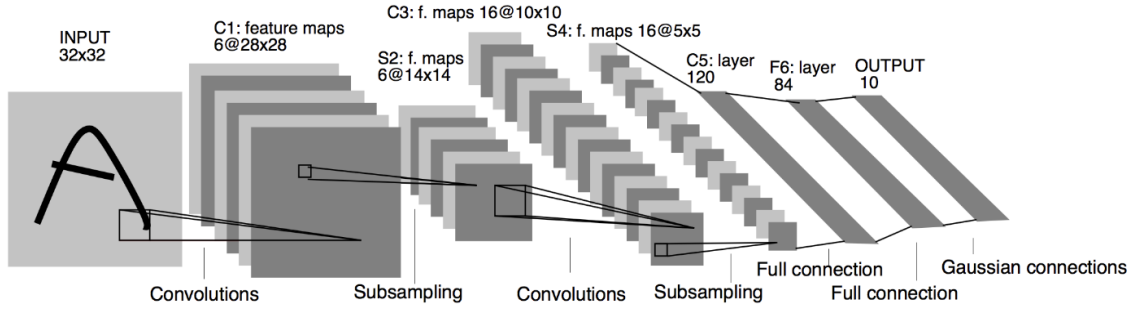
week 4

▼ LeNet-5

LeNet은 Yann LeCun과 그의 팀이 개발한 초기의 합성곱 신경망(CNN, Convolutional Neural Network) 모델로, 주로 이미지 인식 작업을 위해 설계되었습니다. LeNet-5라는 이름으로도 잘 알려져 있으며, 특히 손글씨 숫자 인식을 목적으로 개발되었습니다. LeNet 모델은 MNIST 데이터셋(손글씨 숫자 0~9)을 분류하는 데 큰 성과를 보였고, 이후 CNN의 발전에 중요한 기초를 마련했습니다.



입력 레이어 (Input Layer): 32x32 크기의 이미지를 입력으로 받습니다. MNIST 이미지의 원래 크기는 28x28이지만, LeNet은 32x32 이미지에 맞게 변형해서 사용합니다.

합성곱 레이어 (Convolutional Layer): 첫 번째 합성곱 레이어는 여러 필터(또는 커널)를 통해 이미지에서 중요한 특징을 추출합니다. 필터를 적용하여 이미지의 패턴을 감지하고, 패턴이 있는 영역에서 활성화를 시킵니다

풀링 레이어 (Pooling Layer): 최대 풀링(Max Pooling) 또는 평균 풀링을 사용하여 이미지의 크기를 줄이고, 중요한 특징만 남겨서 다음 레이어로 전달합니다.

완전 연결 레이어 (Fully Connected Layer): 마지막에 완전 연결 레이어를 추가하여 이전 단계에서 추출된 특징을 바탕으로 분류 작업을 수행합니다. LeNet에서는 주로 두 개의 완전 연결 레이어를 사용합니다.

출력 레이어 (Output Layer): 손글씨 숫자를 분류하는 작업에서는 10개의 클래스로 구분되기 때문에, 출력 레이어는 10개의 뉴런을 갖고 있으며 소프트맥스 활성화 함수를 통해 각 클래스의 확률을 계산합니다.

```
#필요한 라이브러리 가져오기
from tensorflow.keras import Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatt
en, Dense
from tensorflow.keras.datasets import mnist
import numpy as np
```

```
#데이터셋 로드 및 전처리
(x_train, y_train), (x_test, y_test) = mnist.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0

#하이퍼파라미터 설정
num_classes = 10
epochs = 100
batch_size = 32

#LeNet-5 모델 클래스 정의
class LeNet5(Model):
    def __init__(self, num_classes):
        super(LeNet5, self).__init__()
        self.conv1 = Conv2D(6, kernel_size=(5,5), padding='sam
e', activation='relu')
        self.conv2 = Conv2D(16, kernel_size=(5,5), activation
='relu')
        self.max_pool = MaxPooling2D(pool_size=(2,2))
        self.flatten = Flatten()
        self.dense1 = Dense(120, activation='relu')
        self.dense2 = Dense(84, activation='relu')
        self.dense3 = Dense(num_classes, activation='softmax')
    #모델의 순전파 정의
    def call(self, input_data):
        x = self.max_pool(self.conv1(input_data))
        x = self.max_pool(self.conv2(x))
        x = self.flatten(x)
        x = self.dense3(self.dense2(self.dense1(x)))

        return x
#모델 컴파일
model = LeNet5(num_classes)
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
#콜백 설정
callbacks = [tf.keras.callbacks.EarlyStopping(patience=3, monit
or='val_loss'),
             tf.keras.callbacks.TensorBoard(log_dir='./logs', h
istogram_freq=1)]
#모델학습
model.fit(x_train, y_train,
```

```
            batch_size = batch_size,
            epochs = epochs,
            validation_data = (x_test, y_test),
            callbacks=callbacks)
```

1. **필요한 라이브러리 가져오기**TensorFlow와 Keras에서 필요한 클래스 및 함수를 불러옵니다. Conv2D는 합성곱 레이어, MaxPooling2D는 풀링 레이어를 정의하는 데 사용됩니다. mnist 모듈은 MNIST 데이터셋을 제공합니다.

2. **데이터셋 로드 및 전처리**mnist.load_data() 함수를 통해 MNIST 데이터셋을 로드하고 (x_train, y_train), (x_test, y_test)로 구분합니다. x_train과 x_test의 픽셀 값을 255로 나눠 [0, 1] 범위로 정규화합니다.

3. **하이퍼파라미터 설정**• num_classes: 분류할 클래스 수(숫자 0~9)로 설정• epochs: 학습 반복 횟수로, 총 100번 반복하도록 설정.• batch_size: 배치 크기로 32를 설정하여, 한 번에 32개의 샘플을 사용해 학습합니다.

4. **LeNet-5 모델 클래스 정의**

- Conv2D 레이어를 사용해 두 개의 합성곱 레이어(conv1, conv2)를 정의합니다.
- MaxPooling2D 레이어를 사용해 풀링 레이어(max_pool)를 정의하여 특징 맵의 크기를 줄입니다.
- Flatten 레이어는 다차원 배열을 1차원으로 변환합니다.
- 세 개의 완전 연결(Dense) 레이어를 사용하여 이미지를 클래스별로 분류합니다.
- dense1: 120개의 노드와 ReLU 활성화 함수.
- dense2: 84개의 노드와 ReLU 활성화 함수.
- dense3: 10개의 클래스 노드와 softmax 활성화 함수를 사용하여 확률을 계산합니다.

5. **모델의 순전파(forward pass) 정의**call 메서드에서 레이어가 순서대로 호출되어 모델의 순전파가 수행됩니다.

6. **모델 컴파일**• optimizer='sgd': 확률적 경사 하강법(SGD)을 사용합니다.• loss='sparse_categorical_crossentropy': 라벨이 원-핫 인코딩 되지 않았으므로 sparse_categorical_crossentropy 손실 함수를 사용합니다.• metrics=['accuracy']: 학습 및 평가 시 정확도를 모니터링합니다.

7. **콜백 설정**• EarlyStopping: 검증 손실(val_loss)이 3번 이상 개선되지 않으면 학습을 중단.• TensorBoard: 로그 파일을 생성하여 TensorBoard에서 시각화할 수 있게 합니다.

8. **모델 학습**fit 메서드를 통해 모델을 학습합니다.

▼ LeNet의 장점

- **효율성**: 단순한 구조 덕분에 학습이 빠르고 효율적입니다.
- **시각적 특징 추출**: 이미지 데이터를 효과적으로 처리하여 특징을 추출하는 구조를 갖추고 있습니다.

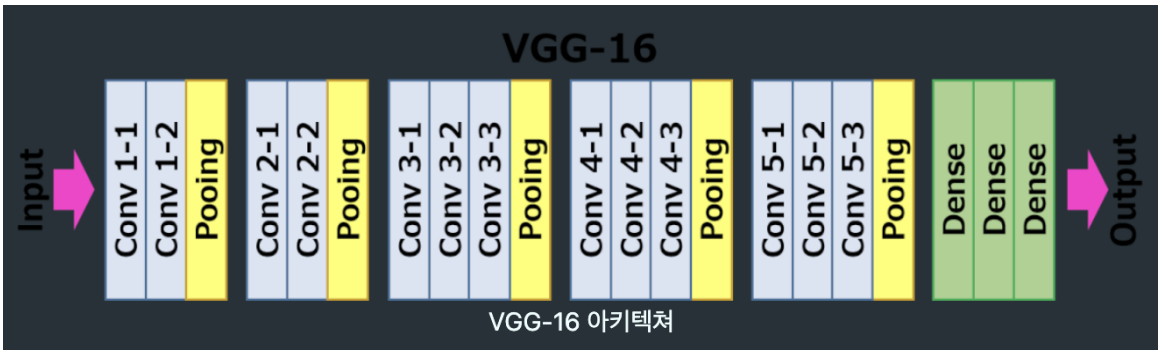
- **기초적인 CNN 구조**: LeNet의 구조는 이후 등장하는 다양한 CNN 아키텍처(VGG, ResNet 등)의 기초가 되었습니다.

▼ LeNet의 한계

- **복잡한 이미지 처리에 약함**: LeNet은 단순한 이미지에 적합하며, 복잡한 이미지나 고해상도 이미지 처리에는 적합하지 않습니다.
- **얕은 신경망 구조**: 깊이가 얕아, 복잡한 패턴을 학습하는 데 한계가 있습니다.

▼ VGGNet

VGGNet은 옥스퍼드 대학교와 구글 딥마인드 팀이 개발한 합성곱 신경망(CNN) 모델로, 이미지 분류 작업을 위해 설계되었습니다. VGGNet은 특히 심층 CNN의 중요성을 강조하며 단순히 레이어 수를 늘린 구조로 높은 성능을 달성한 모델입니다. VGGNet의 주요 특징은 여러 개의 작은 필터(3x3 필터)를 사용한 합성곱 레이어를 깊게 쌓아 복잡한 이미지의 특징을 추출하는 데 탁월하다는 점입니다.



VGG-16 아키텍쳐

VGGNet의 대표적인 모델은 VGG-16과 VGG-19로, 각각 16개와 19개의 레이어를 가지고 있습니다. 다음과 같은 기본적인 레이어 구조를 갖추고 있습니다

입력 레이어: 일반적으로 224x224 크기의 이미지를 받습니다.

합성곱 레이어: 3x3 크기의 필터와 ReLU 활성화 함수를 사용하여 여러 층을 쌓습니다

풀링 레이어: 2x2 크기의 최대 풀링(Max Pooling) 레이어를 사용하여 이미지의 크기를 줄입니다.

완전 연결 레이어: 마지막에 세 개의 완전 연결(Dense) 레이어를 추가하여 이미지의 특징을 바탕으로 분류합니다.

출력 레이어: 최종 출력 레이어에서는 소프트맥스 함수를 통해 각 클래스의 확률을 계산합니다.

```
from tensorflow.keras import Model
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Flatt
en, Dense, Input
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# 데이터셋 로드 및 전처리
```

```
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# VGG-16 모델 클래스 정의
class VGG16(Model):
    def __init__(self, num_classes):
        super(VGG16, self).__init__()
        # 1~2단계 합성곱과 풀링
        self.conv1_1 = Conv2D(64, (3, 3), padding='same', activ
ation='relu')
        self.conv1_2 = Conv2D(64, (3, 3), padding='same', activ
ation='relu')
        self.pool1 = MaxPooling2D((2, 2), strides=(2, 2))

        # 3~4단계 합성곱과 풀링
        self.conv2_1 = Conv2D(128, (3, 3), padding='same', acti
vation='relu')
        self.conv2_2 = Conv2D(128, (3, 3), padding='same', acti
vation='relu')
        self.pool2 = MaxPooling2D((2, 2), strides=(2, 2))

        # 5~7단계 합성곱과 풀링
        self.conv3_1 = Conv2D(256, (3, 3), padding='same', acti
vation='relu')
        self.conv3_2 = Conv2D(256, (3, 3), padding='same', acti
vation='relu')
        self.conv3_3 = Conv2D(256, (3, 3), padding='same', acti
vation='relu')
        self.pool3 = MaxPooling2D((2, 2), strides=(2, 2))

        # 8~10단계 합성곱과 풀링
        self.conv4_1 = Conv2D(512, (3, 3), padding='same', acti
vation='relu')
        self.conv4_2 = Conv2D(512, (3, 3), padding='same', acti
vation='relu')
        self.conv4_3 = Conv2D(512, (3, 3), padding='same', acti
vation='relu')
        self.pool4 = MaxPooling2D((2, 2), strides=(2, 2))

        # 11~13단계 합성곱과 풀링
        self.conv5_1 = Conv2D(512, (3, 3), padding='same', acti
```

```
vation='relu')
        self.conv5_2 = Conv2D(512, (3, 3), padding='same', acti
vation='relu')
        self.conv5_3 = Conv2D(512, (3, 3), padding='same', acti
vation='relu')
        self.pool5 = MaxPooling2D((2, 2), strides=(2, 2))

        # 완전 연결 레이어
        self.flatten = Flatten()
        self.fc1 = Dense(4096, activation='relu')
        self.fc2 = Dense(4096, activation='relu')
        self.output_layer = Dense(num_classes, activation='soft
max')

    def call(self, inputs):
        x = self.pool1(self.conv1_2(self.conv1_1(inputs)))
        x = self.pool2(self.conv2_2(self.conv2_1(x)))
        x = self.pool3(self.conv3_3(self.conv3_2(self.conv3_1
(x))))
        x = self.pool4(self.conv4_3(self.conv4_2(self.conv4_1
(x))))
        x = self.pool5(self.conv5_3(self.conv5_2(self.conv5_1
(x))))
        x = self.flatten(x)
        x = self.fc2(self.fc1(x))
        return self.output_layer(x)

# 하이퍼파라미터 설정 및 모델 컴파일
num_classes = 10
model = VGG16(num_classes)
model.compile(optimizer='adam',
              loss='categorical_crossentropy',
              metrics=['accuracy'])

# 모델 학습
model.fit(x_train, y_train,
          batch_size=64,
          epochs=20,
          validation_data=(x_test, y_test))
```

1. **데이터셋 로드 및 전처리**

CIFAR-10 데이터셋을 사용하여 모델을 훈련합니다. 이미지 픽셀 값을 [0, 1] 범위로 정규화하며, 라벨을 원-핫 인코딩하여 10개의 클래스로 구분합니다.

2. **VGG-16 모델 클래스 정의**

- **합성곱 레이어**: 3x3 필터와 ReLU 활성화 함수 사용. 첫 번째 단계에서는 64개의 필터를 적용하고, 이후 단계에서는 필터 수를 두 배로 늘려 더 많은 특징을 추출합니다.
- **풀링 레이어**: 각 단계마다 합성곱 레이어 2~3개 이후 풀링 레이어를 추가해 이미지의 크기를 줄입니다.
- **완전 연결 레이어**: 마지막에 두 개의 Dense 레이어(fc1, fc2)를 추가하여 특징을 종합합니다. 최종 출력 레이어는 10개의 클래스로 분류하도록 softmax 활성화 함수를 사용합니다.

3. **모델의 순전파 정의**call 메서드에서 순서대로 레이어가 호출되며 모델의 순전파(forward pass)가 수행됩니다.

4. **모델 컴파일**

- **최적화 함수**: adam을 사용하여 학습 속도를 조절합니다.
- **손실 함수**: 원-핫 인코딩된 레이블이므로 categorical_crossentropy 손실 함수를 사용합니다.
- **메트릭**: 정확도를 기준으로 성능을 모니터링합니다.

5. **모델 학습**

- fit 메서드로 모델을 훈련합니다. batch_size=64로 설정하여 64개의 샘플씩 처리하며, 20번의 에포크 동안 학습을 수행합니다. validation_data 인자를 통해 테스트 데이터의 성능도 함께 평가합니다.

▼ VGGNet의 장점

- **일관된 구조**: 3x3 필터를 반복적으로 사용하여 구조가 간단하고 재현이 쉽습니다.
- **깊은 네트워크로 높은 성능**: 깊은 구조 덕분에 이미지 분류 성능이 뛰어납니다.
- **전이 학습에 적합**: 사전 학습된 가중치가 있어 다른 데이터셋에서도 쉽게 활용할 수 있습니다.

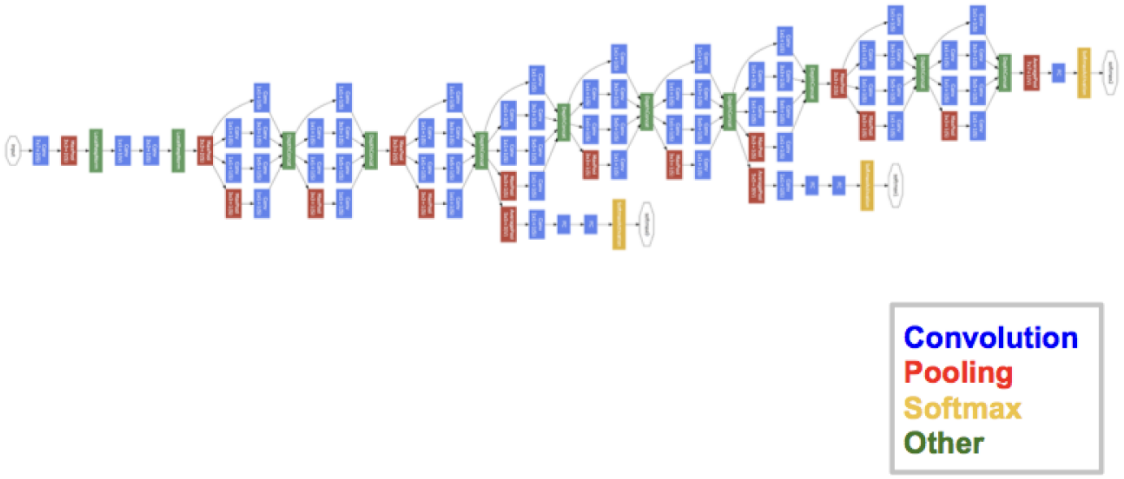
▼ VGGNet의 한계

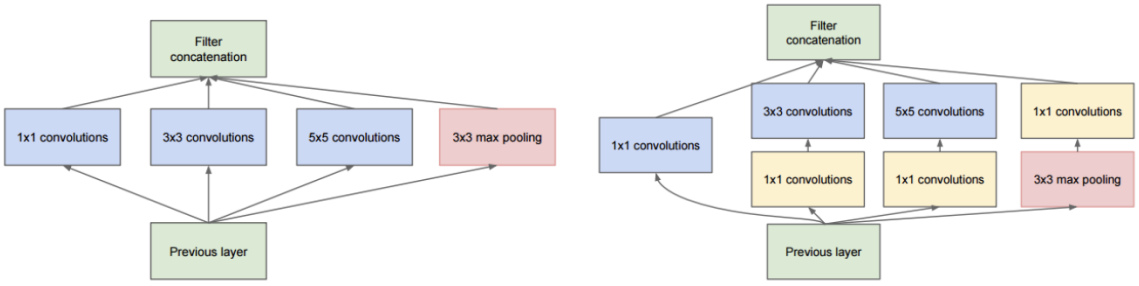
- **높은 계산 비용**: 레이어가 깊어 메모리 사용량과 계산량이 큽니다.
- **과적합 위험**: 파라미터가 많아 데이터가 적을 때 과적합될 수 있습니다.
- **경사 소실 문제**: 더 깊은 모델로 가면 학습이 어려워지는 문제가 있습니다.

▼ GoogLeNet

GoogLeNet(또는 Inception v1)은 Google이 2014년 이미지 분류 대회인 ILSVRC에서 우승하기 위해 개발한 합성곱 신경망(CNN)입니다. GoogLeNet은 VGGNet이나 AlexNet처럼 단순히 레이어를 깊게 쌓는 방식이 아니라, 네트워크의 효율성을 높이기 위해 병렬 합성곱과 풀링 레이어

를 사용한 **Inception 모듈**을 도입했습니다. 이로 인해 연산량은 줄이고 정확도는 높일 수 있었습니다.





(a) Inception module, naïve version (b) Inception module with dimension reductions

```
from tensorflow.keras import Model, Input
from tensorflow.keras.layers import Conv2D, MaxPooling2D, Avera
gePooling2D, Dense, Flatten, concatenate
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

# Inception 모듈 정의
def inception_module(x, filters_1x1, filters_3x3_reduce, filter
s_3x3, filters_5x5_reduce, filters_5x5, filters_pool_proj):
    conv_1x1 = Conv2D(filters_1x1, (1, 1), padding='same', acti
vation='relu')(x)

    conv_3x3 = Conv2D(filters_3x3_reduce, (1, 1), padding='sam
e', activation='relu')(x)
    conv_3x3 = Conv2D(filters_3x3, (3, 3), padding='same', acti
vation='relu')(conv_3x3)
```

```
    conv_5x5 = Conv2D(filters_5x5_reduce, (1, 1), padding='sam
e', activation='relu')(x)
    conv_5x5 = Conv2D(filters_5x5, (5, 5), padding='same', acti
vation='relu')(conv_5x5)

    pool_proj = MaxPooling2D((3, 3), strides=(1, 1), padding='s
ame')(x)
    pool_proj = Conv2D(filters_pool_proj, (1, 1), padding='sam
e', activation='relu')(pool_proj)

    output = concatenate([conv_1x1, conv_3x3, conv_5x5, pool_pr
oj], axis=-1)
    return output

# GoogLeNet 모델 정의
def googlenet(input_shape=(32, 32, 3), num_classes=10):
    input_layer = Input(shape=input_shape)

    # 초기 합성곱 및 풀링 레이어
    x = Conv2D(64, (7, 7), strides=(2, 2), padding='same', acti
vation='relu')(input_layer)
    x = MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

    x = Conv2D(64, (1, 1), padding='same', activation='relu')
(x)
    x = Conv2D(192, (3, 3), padding='same', activation='relu')
(x)
    x = MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

    # Inception 모듈 쌓기
    x = inception_module(x, 64, 96, 128, 16, 32, 32)
    x = inception_module(x, 128, 128, 192, 32, 96, 64)
    x = MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

    x = inception_module(x, 192, 96, 208, 16, 48, 64)
    x = inception_module(x, 160, 112, 224, 24, 64, 64)
    x = inception_module(x, 128, 128, 256, 24, 64, 64)
    x = inception_module(x, 112, 144, 288, 32, 64, 64)
    x = inception_module(x, 256, 160, 320, 32, 128, 128)
    x = MaxPooling2D((3, 3), strides=(2, 2), padding='same')(x)

    # 마지막 Inception 모듈들
    x = inception_module(x, 256, 160, 320, 32, 128, 128)
```

```
    x = inception_module(x, 384, 192, 384, 48, 128, 128)

    # 완전 연결 레이어
    x = AveragePooling2D((4, 4), padding='same')(x)
    x = Flatten()(x)
    x = Dense(1024, activation='relu')(x)
    x = Dense(num_classes, activation='softmax')(x)

    model = Model(input_layer, x, name='GoogLeNet')
    return model

# 데이터셋 로드 및 전처리
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# 모델 컴파일 및 학습
model = googlenet(input_shape=(32, 32, 3), num_classes=10)
model.compile(optimizer='adam', loss='categorical_crossentrop
y', metrics=['accuracy'])
model.fit(x_train, y_train, batch_size=128, epochs=10, validati
on_data=(x_test, y_test))
```

1. **Inception 모듈 정의**

inception_module 함수는 GoogLeNet의 핵심 요소로, 다양한 크기의 필터와 풀링 레이어를 병렬로 구성합니다. 이 모듈을 통해 다양한 크기의 특징을 추출할 수 있습니다.

- **1×1, 3×3, 5×5 합성곱**: 각각 다른 크기의 특징을 추출하기 위해 병렬로 사용됩니다.
- **채널 축소**: filters_3×3_reduce와 filters_5×5_reduce 필터가 1x1 합성곱을 통해 입력 채널을 줄여 연산량을 줄입니다.
- **풀링**: MaxPooling 레이어는 공간적으로 중요한 특징만 추출하고, 1x1 필터를 통해 채널 수를 조절합니다.
- **출력 연결**: 이 모듈의 병렬 연산 결과는 concatenate로 연결하여 Inception 모듈의 최종 출력을 생성합니다.

2. **GoogLeNet 모델 정의**

googlenet 함수는 GoogLeNet의 전체 구조를 정의합니다. 초기에는 큰 필터로 특징을 추출하고 이후에는 Inception 모듈을 여러 번 쌓아 특징을 학습합니다.

- **초기 레이어**: 초기 합성곱 및 풀링 레이어는 고수준의 특징을 추출하며, 이미지 크기를 줄이는 데 사용됩니다.

- **Inception 모듈 블록**: 다양한 크기의 필터가 있는 Inception 모듈을 여러 층에 걸쳐 쌓아, 다양한 해상도의 특징을 추출합니다. 이 과정에서 두 번의 MaxPooling을 통해 네트워크 깊이를 조절하고, 공간 크기를 줄입니다.
- **평균 풀링과 완전 연결 레이어**: AveragePooling2D는 Inception 모듈에서 추출된 다양한 특징을 평탄화하여 완전 연결 레이어로 보냅니다. 마지막 Dense 레이어는 각 클래스에 대한 확률을 출력합니다.

3. **데이터셋 로드 및 전처리**

CIFAR-10 데이터셋을 사용해 이미지를 정규화하고 레이블을 원-핫 인코딩하여 모델에 적합한 형태로 준비합니다.

- **정규화**: 픽셀 값을 0에서 1 사이로 조정하여 학습 안정성을 높입니다.
- **원-핫 인코딩**: 클래스 레이블을 원-핫 벡터로 변환하여 모델의 다중 클래스 분류에 맞게 준비합니다.

4. **모델 학습 및 평가**

model.fit을 통해 모델을 학습하고, CIFAR-10 테스트 데이터로 성능을 평가합니다.

- **모델 컴파일**: Adam 최적화 알고리즘과 categorical_crossentropy 손실 함수를 사용하여 컴파일합니다.
- **학습**: 배치 크기 128과 10개의 에포크로 학습하며, 테스트 데이터에 대해 검증을 수행합니다.

▼ GoogLeNet의 장점

- **효율적인 연산**: Inception 모듈의 1x1 합성곱을 통해 차원을 축소하고, 연산량을 줄여 더 깊은 네트워크를 효과적으로 쌓을 수 있습니다.
- **다양한 특징 추출**: 1x1, 3×3, 5×5 필터와 풀링 레이어를 병렬로 사용하여 다양한 크기의 특징을 동시에 추출할 수 있어 네트워크의 표현력을 높입니다.
- **경량 모델**: 다른 깊은 신경망에 비해 파라미터 수가 적어 메모리와 연산 자원을 적게 사용합니다.

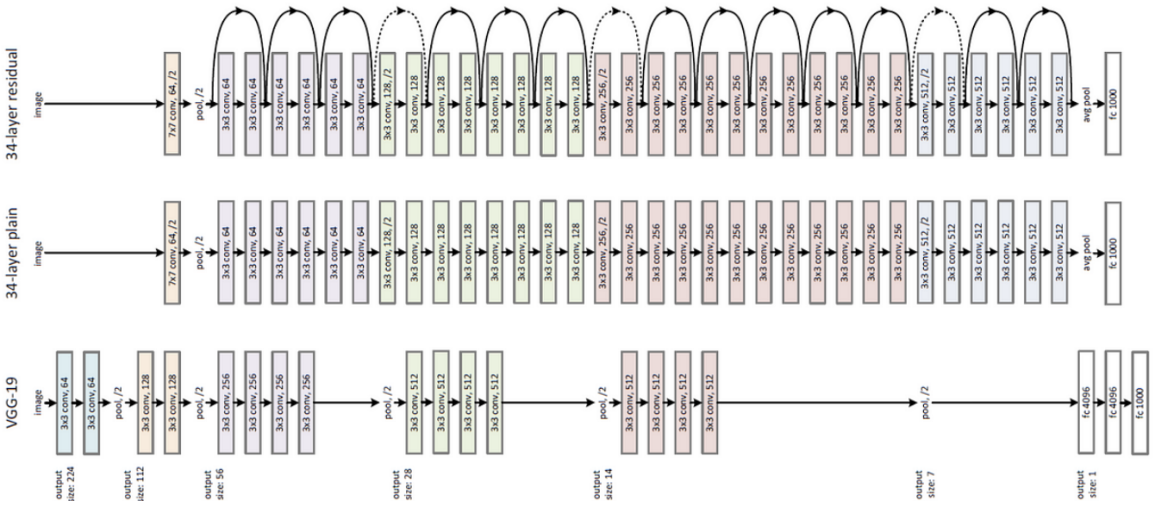
▼ GoogLeNet의 한계

- **복잡한 구조**: Inception 모듈의 복잡한 구조로 인해 구현이 어렵고, 최적의 설정을 찾는 데 시간이 걸립니다.
- **하이퍼파라미터 튜닝**: 각 필터와 레이어의 수를 조절해야 해 하이퍼파라미터 튜닝이 번거롭고 시간이 많이 소요됩니다.
- **모델 확장성**: Inception 모듈을 쌓는 방식이 고정적이어서 다른 모델에 비해 유연성이 떨어질 수 있습니다.

▼ ResNet

ResNet(Residual Network)은 심층 신경망의 학습이 어려운 문제를 해결하기 위해 **잔차 연결 (residual connections)**을 도입한 모델입니다. 이 잔차 연결은 네트워크가 더 깊어져도 그래디언트 소실 문제 없이 학습할 수 있도록 돕습니다. ResNet의 주요 아이디어는 입력이 다음 레이어

로 직접 전달되는 스킵 연결(skip connection)을 추가하여 층이 더 깊어질수록 발생하는 성능 저하 문제를 줄이는 것입니다.




```
import tensorflow as tf
from tensorflow.keras import layers, models, Input
from tensorflow.keras.datasets import cifar10
from tensorflow.keras.utils import to_categorical

def residual_block(x, filters, kernel_size=3, stride=1):
    # 잔차 연결에 사용될 입력
    shortcut = x

    # 첫 번째 합성곱 레이어
    x = layers.Conv2D(filters, kernel_size=kernel_size, strides
=stride, padding='same', activation='relu')(x)
    x = layers.BatchNormalization()(x)

    # 두 번째 합성곱 레이어
    x = layers.Conv2D(filters, kernel_size=kernel_size, padding
='same', activation=None)(x)
    x = layers.BatchNormalization()(x)

    # 스킵 연결이 필요할 경우 (stride가 1이 아닐 때)
if stride != 1 or x.shape[-1] != shortcut.shape[-1]:
        shortcut = layers.Conv2D(filters, kernel_size=1, stride
s=stride, padding='same')(shortcut)
        shortcut = layers.BatchNormalization()(shortcut)

    # 스킵 연결 수행
```

```
    x = layers.add([x, shortcut])
    x = layers.Activation('relu')(x)
    return x

def ResNet(input_shape=(32, 32, 3), num_classes=10):
    input_layer = Input(shape=input_shape)

    # 초기 합성곱 레이어
    x = layers.Conv2D(64, (3, 3), padding='same', activation='r
elu')(input_layer)
    x = layers.BatchNormalization()(x)

    # Residual Block 스택 생성
    x = residual_block(x, filters=64, stride=1)
    x = residual_block(x, filters=64, stride=1)

    x = residual_block(x, filters=128, stride=2)  # 공간 크기 감소
    x = residual_block(x, filters=128, stride=1)

    x = residual_block(x, filters=256, stride=2)
    x = residual_block(x, filters=256, stride=1)

    x = residual_block(x, filters=512, stride=2)
    x = residual_block(x, filters=512, stride=1)

    # 출력 레이어
    x = layers.GlobalAveragePooling2D()(x)
    x = layers.Dense(num_classes, activation='softmax')(x)

    model = models.Model(input_layer, x, name="ResNet")
    return model

# CIFAR-10 데이터셋 로드 및 전처리
(x_train, y_train), (x_test, y_test) = cifar10.load_data()
x_train, x_test = x_train / 255.0, x_test / 255.0
y_train = to_categorical(y_train, 10)
y_test = to_categorical(y_test, 10)

# 모델 생성 및 컴파일
model = ResNet(input_shape=(32, 32, 3), num_classes=10)
model.compile(optimizer='adam', loss='categorical_crossentrop
y', metrics=['accuracy'])
```

```
# 모델 학습
model.fit(x_train, y_train, batch_size=128, epochs=10, validati
on_data=(x_test, y_test))
```

1. **Residual Block 정의**

- **합성곱 레이어**: 두 개의 합성곱 레이어를 쌓고 각 레이어 뒤에 Batch Normalization을 적용하여 학습 안정성을 높입니다.
- **스킵 연결**: 입력과 출력이 더해지며, 활성화 함수 'ReLU'가 적용됩니다. 이 스킵 연결을 통해 잔차(Residual)를 학습하여 깊은 네트워크에서도 성능 저하를 줄입니다.
- **Shortcut 조정**: stride가 1이 아니거나, 필터 수가 변경된 경우(차원이 다를 경우) 1x1 합성곱을 사용하여 shortcut의 크기를 맞춥니다.

2. **ResNet 모델 정의**

- **초기 합성곱 레이어**: 입력 이미지를 처음으로 받아들여 64개의 필터로 특징을 추출합니다.
- **Residual Block**: 4개의 블록으로 구성되며, 각 블록은 필터 수가 증가하면서 입력 크기를 줄입니다 (stride=2).
- **Global Average Pooling**: 마지막으로, 특징맵을 평균 풀링하여 1차원 벡터로 변환합니다.
- **Dense 레이어**: 다중 클래스 분류를 위한 출력 레이어로 softmax 활성화 함수를 사용하여 각 클래스에 대한 확률을 반환합니다.

3. **데이터셋 로드 및 전처리**

CIFAR-10 데이터셋을 사용해 이미지를 정규화하고 레이블을 원-핫 인코딩하여 모델에 적합한 형태로 준비합니다.

- **정규화**: 픽셀 값을 0에서 1 사이로 조정하여 학습 안정성을 높입니다.
- **원-핫 인코딩**: 클래스 레이블을 원-핫 벡터로 변환하여 모델의 다중 클래스 분류에 맞게 준비합니다.

4. **모델 학습 및 평가**

model.fit을 통해 모델을 학습하고, CIFAR-10 테스트 데이터로 성능을 평가합니다.

- **모델 컴파일**: Adam 최적화 알고리즘과 categorical_crossentropy 손실 함수를 사용하여 컴파일합니다.
- **학습**: 배치 크기 128과 10개의 에포크로 학습하며, 테스트 데이터에 대해 검증을 수행합니다.

▼ ResNet의 장점

- **깊은 네트워크 가능** : ResNet은 잔차 연결(Residual Connection)을 사용하여 매우 깊은 네트워크를 구성할 수 있게 합니다. 이는 네트워크가 더 깊어져도 성능이 저하되지 않도록 도와줍니다.
- **효율적인 학습**: 잔차 연결은 그래디언트 소실 문제를 완화하여 더 효과적으로 학습할 수 있도록 하여, 더 많은 층을 쌓을 수 있습니다.

- **높은 성능**: 여러 이미지 분류 대회에서 좋은 성과를 거두었으며, 다양한 데이터셋에서 높은 정확도를 보입니다.
- **유연한 구조**: 다양한 변형이 가능하여 다른 문제에도 쉽게 적용할 수 있습니다. 예를 들어, ImageNet, CIFAR-10 등 다양한 데이터셋에 맞춰 조정할 수 있습니다.

▼ ResNet의 한계

- **계산 자원 요구**: 깊은 네트워크는 많은 계산 자원을 요구하며, 이는 훈련 및 추론 시간을 늘릴 수 있습니다. 따라서 고성능의 GPU가 필요합니다.
- **모델 복잡성**: 네트워크의 깊이가 증가하면 모델이 복잡해져 과적합(overfitting)의 위험이 커질 수 있습니다. 이를 방지하기 위해 정규화 기법이나 드롭아웃 등의 추가적인 방법이 필요할 수 있습니다.
- **잔차 연결 이해의 필요성**: 잔차 연결 구조는 일반적인 CNN과는 다른 개념이므로, 이를 이해하고 설계하는 데 추가적인 학습이 필요할 수 있습니다.