Convolutional Neural Network

```
class SimpleCNN(nn.Module):
    def __init__(self):
        super().__init__()
        self.conv1 = nn.Conv2d(1, 32, kernel_size=3, padding=1)
        self.conv2 = nn.Conv2d(32, 64, kernel_size=3, padding=1)
        self.pool = nn.MaxPool2d(2, 2)
        self.fc1 = nn.Linear(64 * 7 * 7, 128)
        self.fc2 = nn.Linear(128, 10)  # 10개의 숫자 클래스

    def forward(self, x):
        x = self.pool(F.relu(self.conv1(x)))  # 28x28 -> 14x14
        x = self.pool(F.relu(self.conv2(x)))  # 14x14 -> 7x7
        x = x.view(-1, 64 * 7 * 7)
        x = F.relu(self.fc1(x))
        x = self.fc2(x)
        return x
```

당시로는 혁신적인 **2 GPU** 병렬 연산 사용
ImageNet 2012 대회 우승
CNN + ReLU + MaxPooling + FC Layer

```
class AlexNet(nn.Module):
    def __init__(self, num_classes=1000):
        super(AlexNet, self).__init__()

        self.features = nn.Sequential(
            nn.Conv2d(3, 96, kernel_size=11, stride=4, padding=2),  # (224 -> 55)
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.Conv2d(96, 256, kernel_size=5, padding=2),          # (27 -> 27)
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),                 # (27 -> 13)

            nn.Conv2d(256, 384, kernel_size=3, padding=1),         # (13 -> 13)
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 384, kernel_size=3, padding=1),         # (13 -> 13)
            nn.ReLU(inplace=True),
            nn.Conv2d(384, 256, kernel_size=3, padding=1),         # (13 -> 13)
            nn.ReLU(inplace=True),
            nn.MaxPool2d(kernel_size=3, stride=2),                 # (13 -> 6)
        )

        self.classifier = nn.Sequential(
```

```
            nn.Dropout(),
            nn.Linear(256 * 6 * 6, 4096),
            nn.ReLU(inplace=True),

            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(inplace=True),

            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)  # batch size 제외하고 평탄화
        x = self.classifier(x)
        return x
```

VGG의 기본 컨볼루션 블록 설정 (구조 리스트로 정의)
구조가 단순하고 통일감 있어서 전이학습**(Transfer Learning)** 용으로 가장 널리 쓰였음.

```
cfg = {
    'VGG16': [64, 64, 'M',
              128, 128, 'M',
              256, 256, 256, 'M',
              512, 512, 512, 'M',
              512, 512, 512, 'M']
}


def make_layers(cfg_list):
    layers = []
    in_channels = 3
    for v in cfg_list:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            layers += [
                nn.Conv2d(in_channels, v, kernel_size=3, padding=1),
                nn.ReLU(inplace=True)
            ]
            in_channels = v
    return nn.Sequential(*layers)


class VGG(nn.Module):
    def __init__(self, vgg_name='VGG16', num_classes=1000):
        super(VGG, self).__init__()
        self.features = make_layers(cfg[vgg_name])
```

```
        self.classifier = nn.Sequential(
            nn.Linear(512 * 7 * 7, 4096),  # 입력 이미지는 224x224 기준
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, 4096),
            nn.ReLU(True),
            nn.Dropout(),
            nn.Linear(4096, num_classes),
        )

    def forward(self, x):
        x = self.features(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Residual Network

CNN은 깊게 쌓을수록 성능이 떨어지는 문제 (기울기 소실, degradation)가 있었음.ResNet은 입력값을 다음 층에 더해주는 skip connection(잔차 연결)을 도입해서 수백 층의 깊은 네트워크도 안정적으로 학습할 수 있게 했습니다.

기울기 소실(Vanishing Gradient)

- 역전파(backpropagation) 시, 층이 깊어질수록 **기울기(gradient)**가 점점 작아져서 초기 층이 학습되지 않음.

딥러닝에서 층이 많아지면 이런 일이 생깁니다:

1. 각 층마다 미분된 값(gradient)이 곱해짐
2. 많은 층을 지나면서, 0보다 작은 수를 계속 곱하게 되면 → 값이 점점 작아짐
3. 결국 앞쪽(입력에 가까운) 층에 도달할 때쯤엔 gradient ≈ 0 이 되어버림
4. 그럼 앞쪽 층의 가중치가 거의 안 바뀜 → 학습이 안 됨

- 특히 ReLU 이전의 sigmoid/tanh에서 심각했음

SkipConnection

Skip Connection은 입력을 그대로 더해줌으로써, **"잔차(residual)"**만 학습하게 도와주는 연결 방식으로, ResNet이 수백 층짜리 네트워크도 안정적으로 학습할 수 있게 만든 핵심 기법입니다.

```
class BasicBlock(nn.Module):
    def __init__(self, in_channels):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, in_channels, kernel_size=3, padding=1)
        self.bn1 = nn.BatchNorm2d(in_channels)
        self.conv2 = nn.Conv2d(in_channels, in_channels, kernel_size=3, padding=1)
        self.bn2 = nn.BatchNorm2d(in_channels)

    def forward(self, x):
        identity = x  # Skip Connection 저장
```

```
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.bn2(self.conv2(out))

        out += identity  # Skip Connection 더하기
        return F.relu(out)
```

```
import torch
import torch.nn as nn
import torch.nn.functional as F

class BasicBlock(nn.Module):
    expansion = 1

    def __init__(self, in_channels, out_channels, stride=1, downsample=None):
        super().__init__()
        self.conv1 = nn.Conv2d(in_channels, out_channels, kernel_size=3,stride=stride, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(out_channels)
        self.conv2 = nn.Conv2d(out_channels, out_channels, kernel_size=3, stride=1, padding=1, bias=False)
        self.bn2 = nn.BatchNorm2d(out_channels)
        self.downsample = downsample

    def forward(self, x):
        identity = x
        out = F.relu(self.bn1(self.conv1(x)))
```

```
        out = self.bn2(self.conv2(out))

        if self.downsample is not None:
            identity = self.downsample(x)

        out += identity
        return F.relu(out)

class ResNet(nn.Module):
    def __init__(self, block, layers, num_classes=10):  # CIFAR10 기준
        super().__init__()
        self.in_channels = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3, stride=1, padding=1, bias=False)  # 32x32 -> 32x32
        self.bn1 = nn.BatchNorm2d(64)
        self.relu = nn.ReLU(inplace=True)

        self.layer1 = self._make_layer(block, 64,  layers[0], stride=1)
        self.layer2 = self._make_layer(block, 128, layers[1], stride=2)
        self.layer3 = self._make_layer(block, 256, layers[2], stride=2)
        self.layer4 = self._make_layer(block, 512, layers[3], stride=2)

        self.avgpool = nn.AdaptiveAvgPool2d((1, 1))  # 1x1로 축소
        self.fc = nn.Linear(512 * block.expansion, num_classes)

    def _make_layer(self, block, out_channels, blocks, stride):
        downsample = None
        if stride != 1 or self.in_channels != out_channels * block.expansion:
            downsample = nn.Sequential(
                nn.Conv2d(self.in_channels, out_channels * block.expansion,
```

```
                    kernel_size=1, stride=stride, bias=False),
                nn.BatchNorm2d(out_channels * block.expansion)
            )

        layers = [block(self.in_channels, out_channels, stride, downsample)]
        self.in_channels = out_channels * block.expansion

        for _ in range(1, blocks):
            layers.append(block(self.in_channels, out_channels))

        return nn.Sequential(*layers)

    def forward(self, x):
        x = self.relu(self.bn1(self.conv1(x)))

        x = self.layer1(x)
        x = self.layer2(x)
        x = self.layer3(x)
        x = self.layer4(x)

        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.fc(x)

        return x

def resnet18(num_classes=10):
    return ResNet(BasicBlock, [2, 2, 2, 2], num_classes=num_classes)

# ✅ 사용 예시
```

```
# model = resnet18(num_classes=10)

# print(model)
```