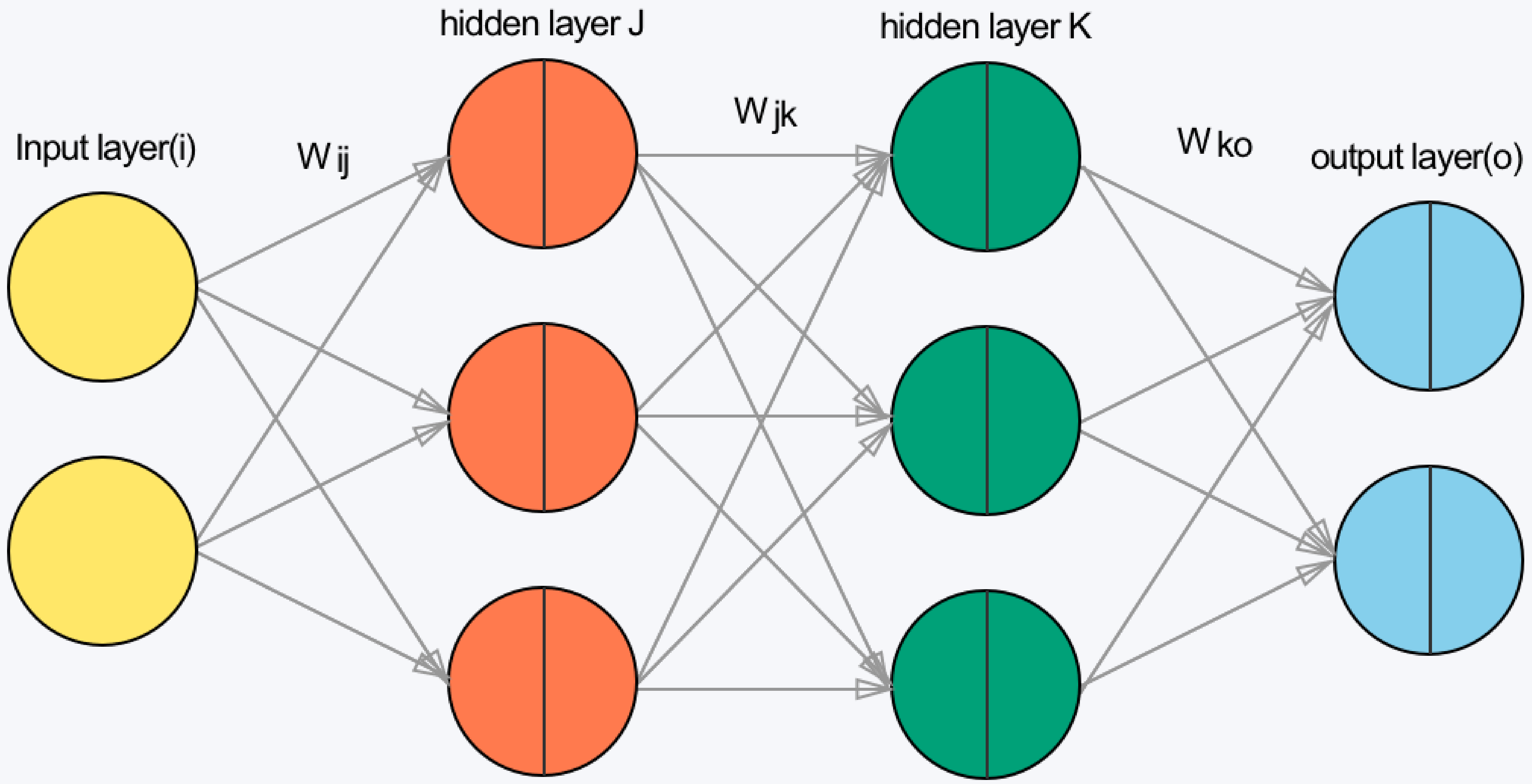
CNN

Convolutional Neural Network

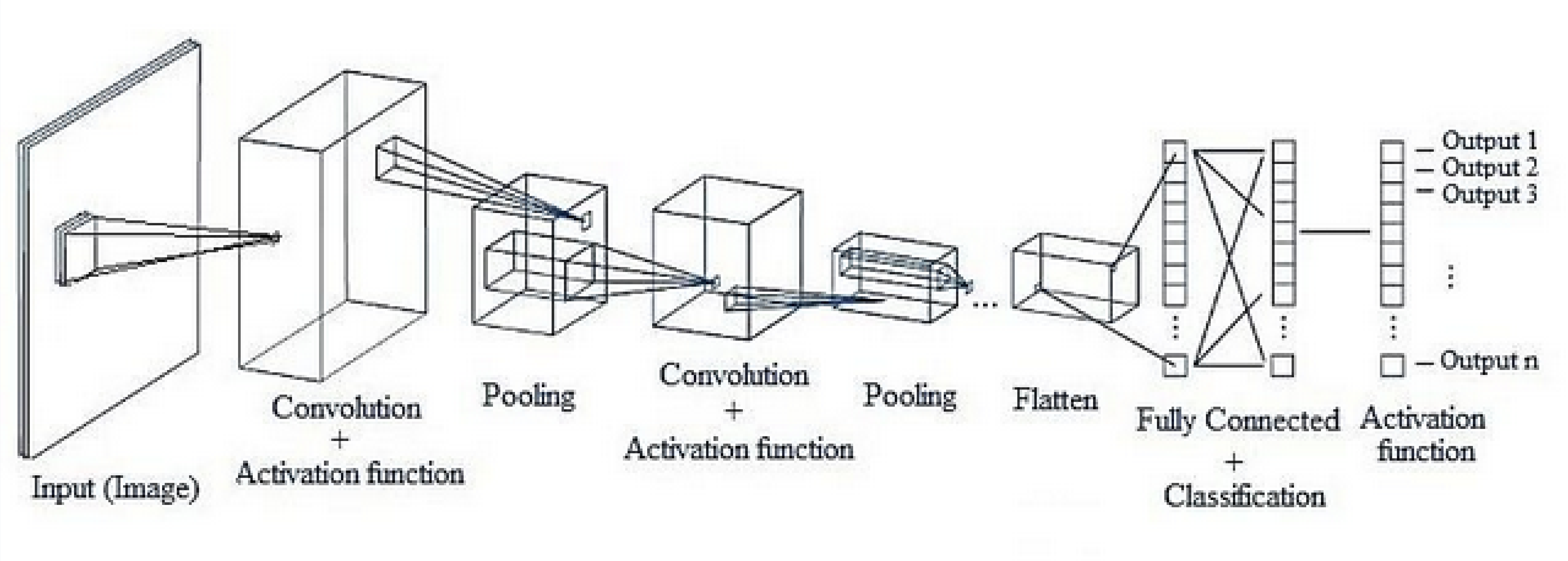
Artificial Neural Network



- **Forward propagation**
- **Backward propagation**

CNN

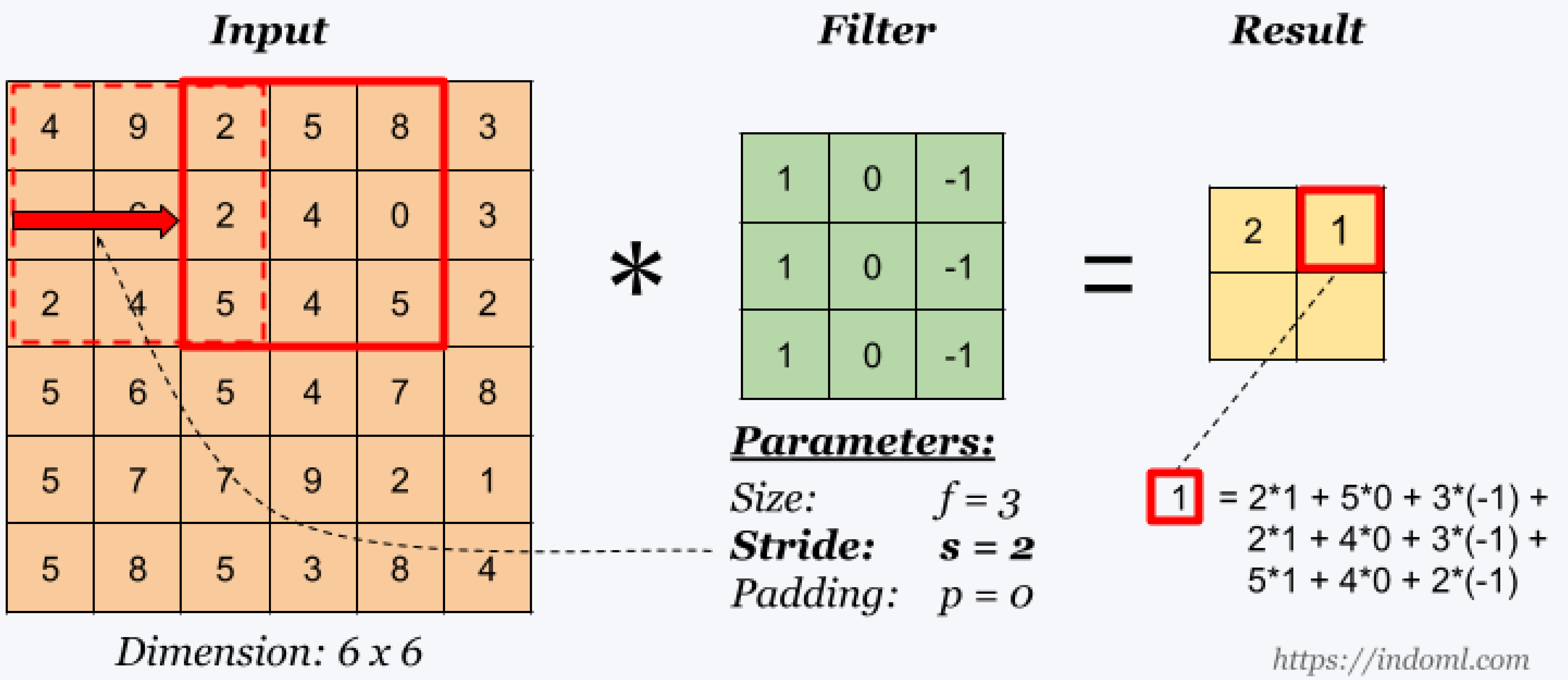
Convolutional Neural Network



- Filter
- Activation function
- Padding
- Pooling

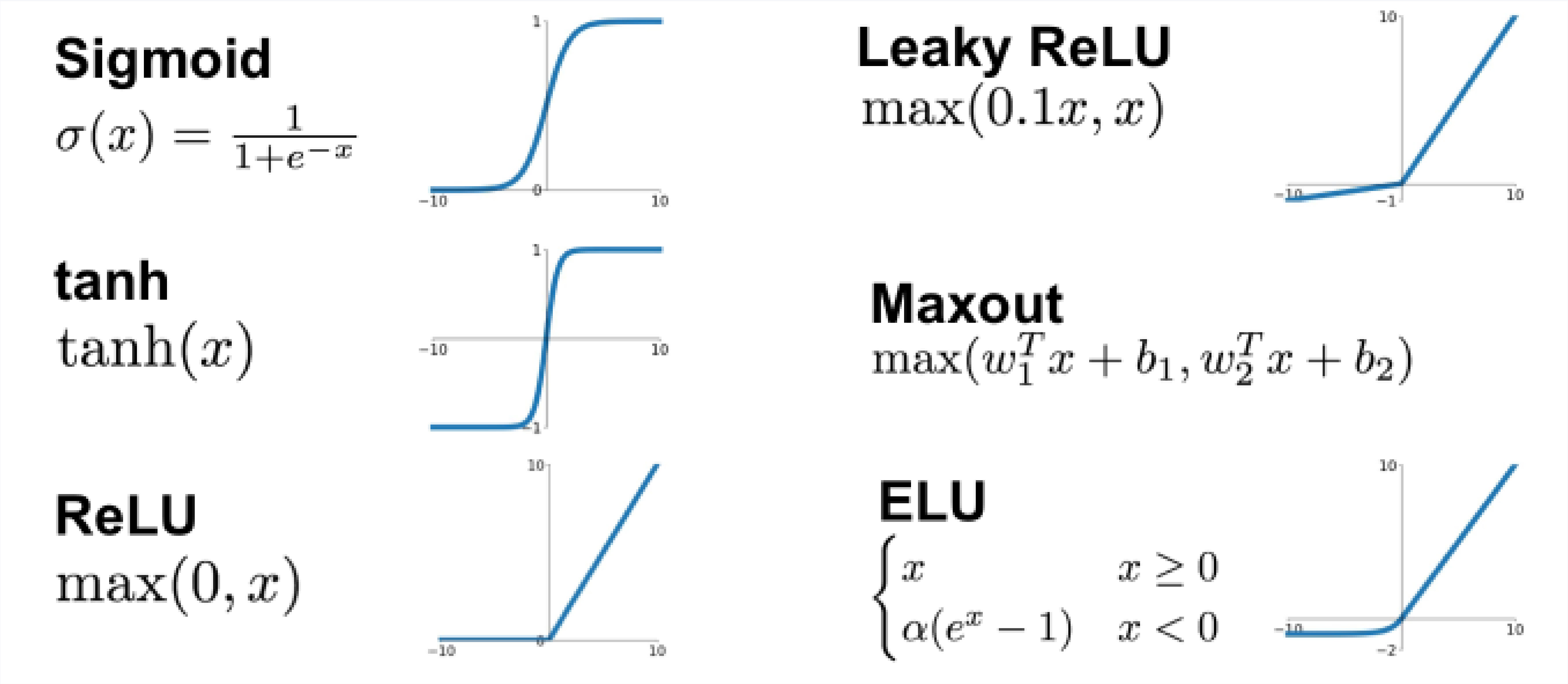
CNN (Convolutional Neural Network)

Filter (Kernel)



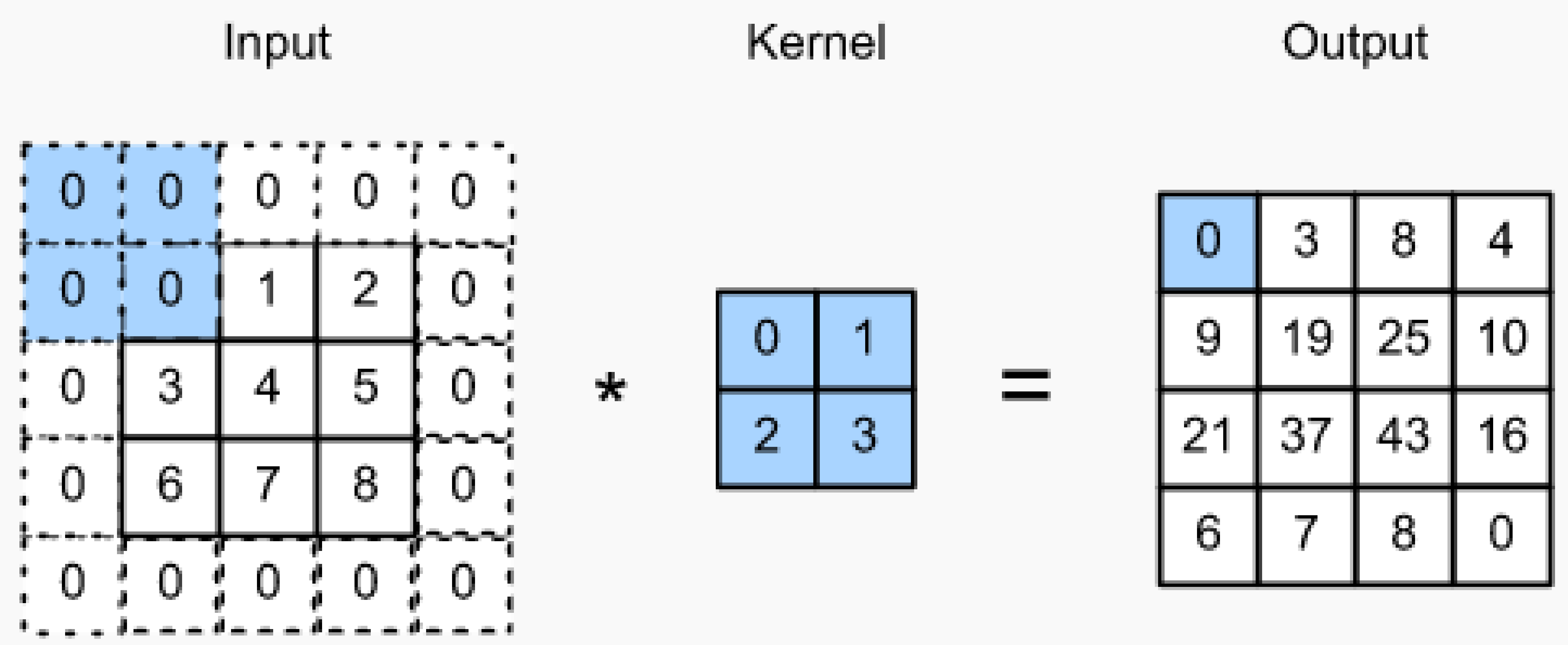
CNN (Convolutional Neural Network)

Activation function



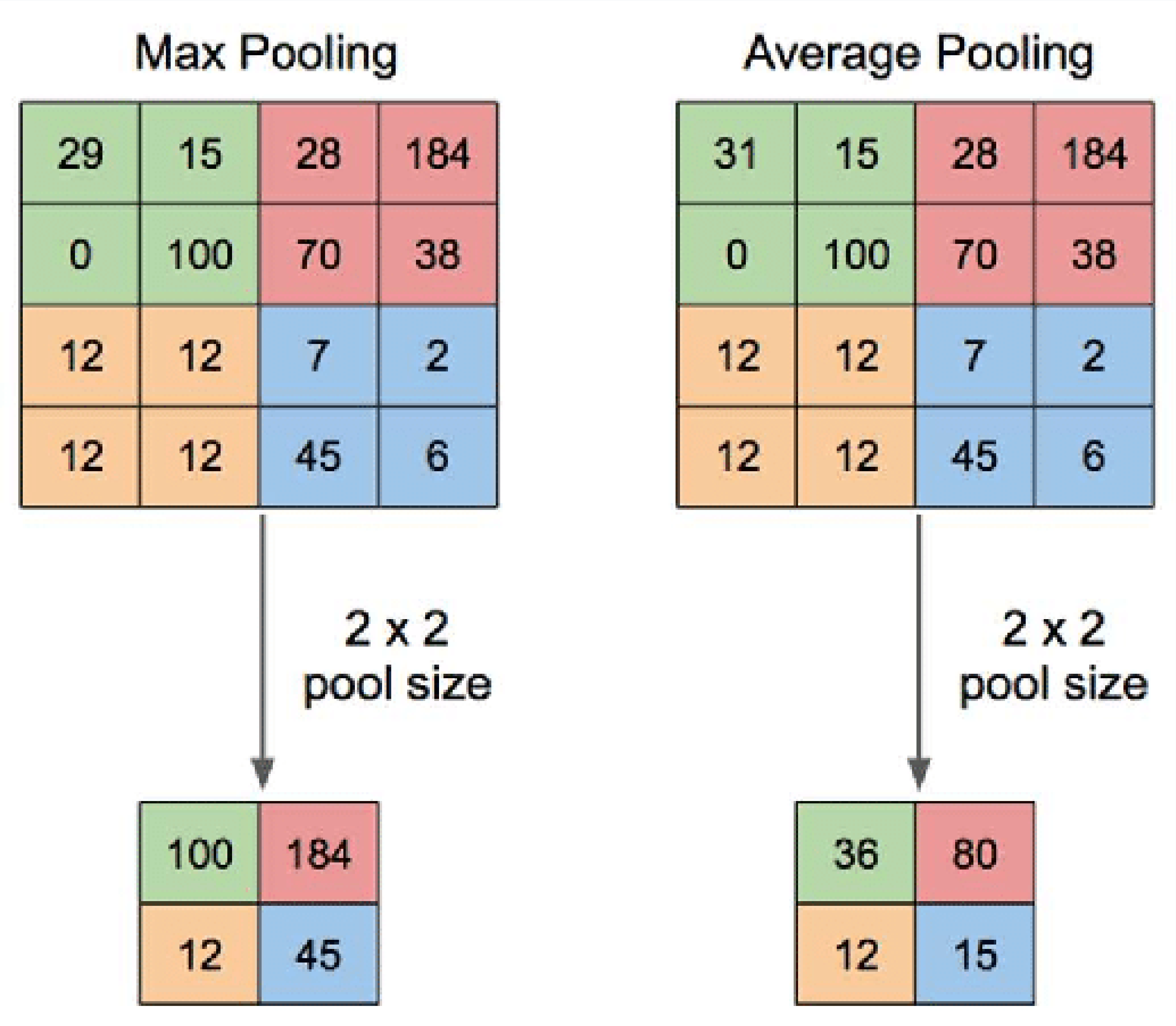
CNN (Convolutional Neural Network)

Padding

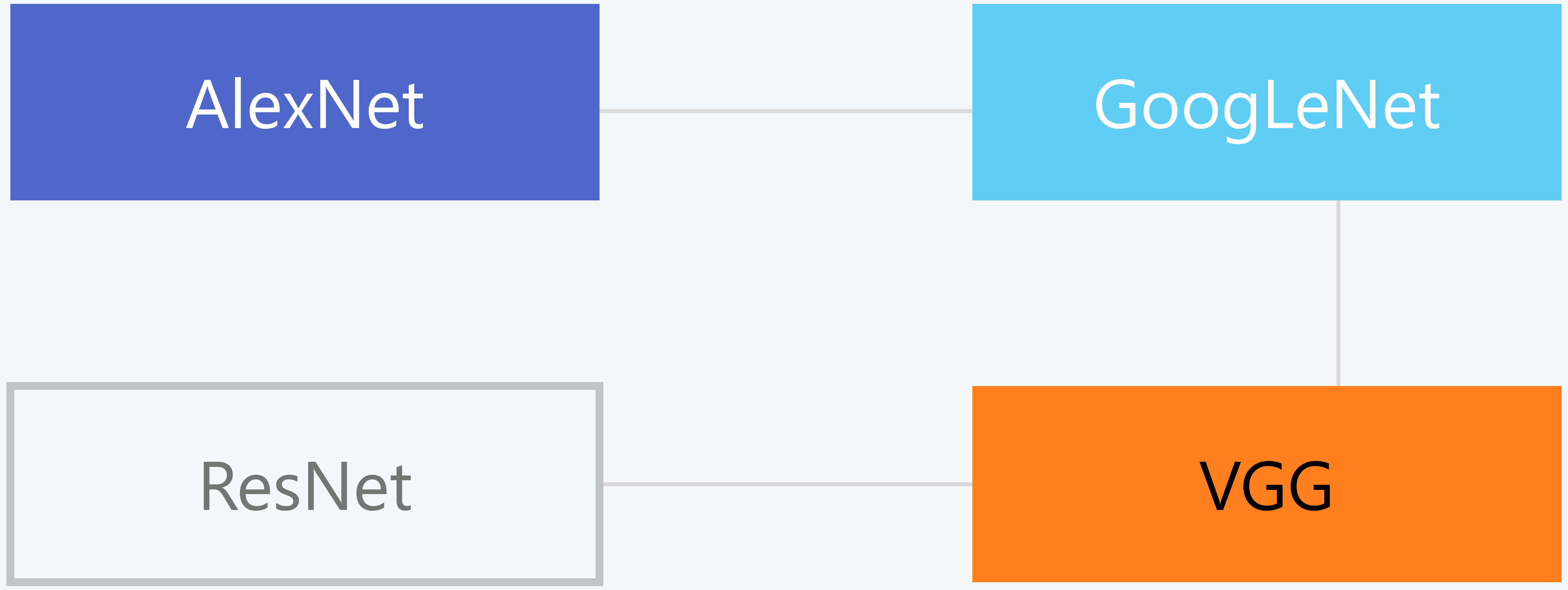


CNN (Convolutional Neural Network)

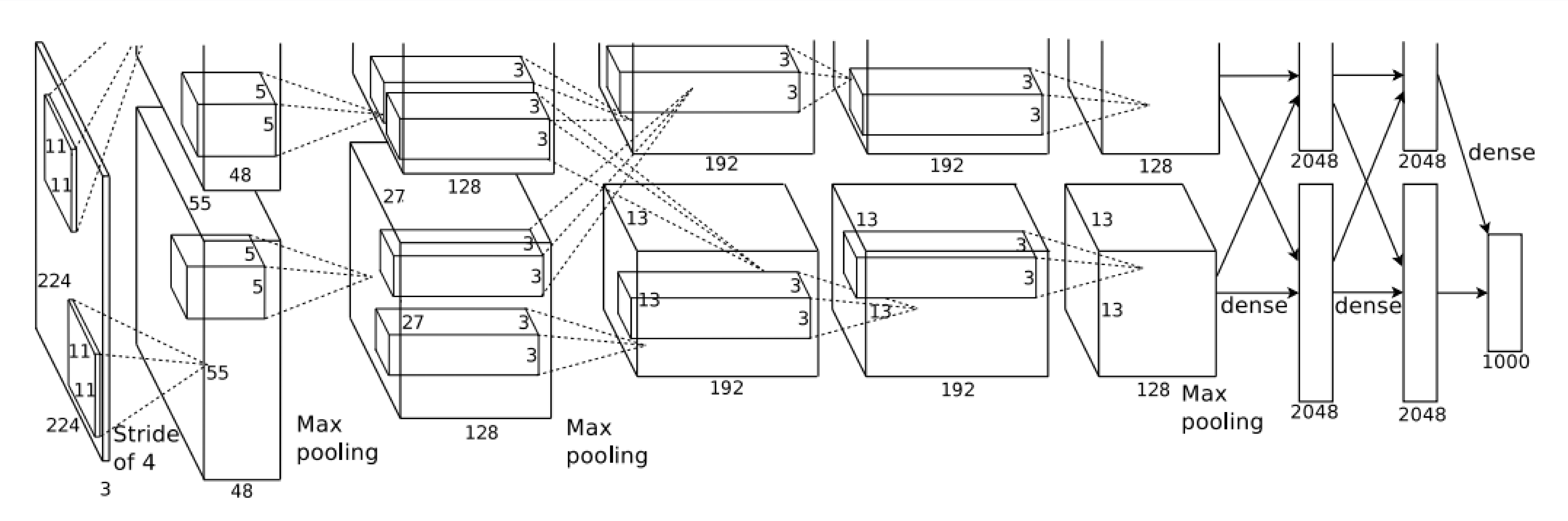
Pooling



Modern Architecture in Deep Learning

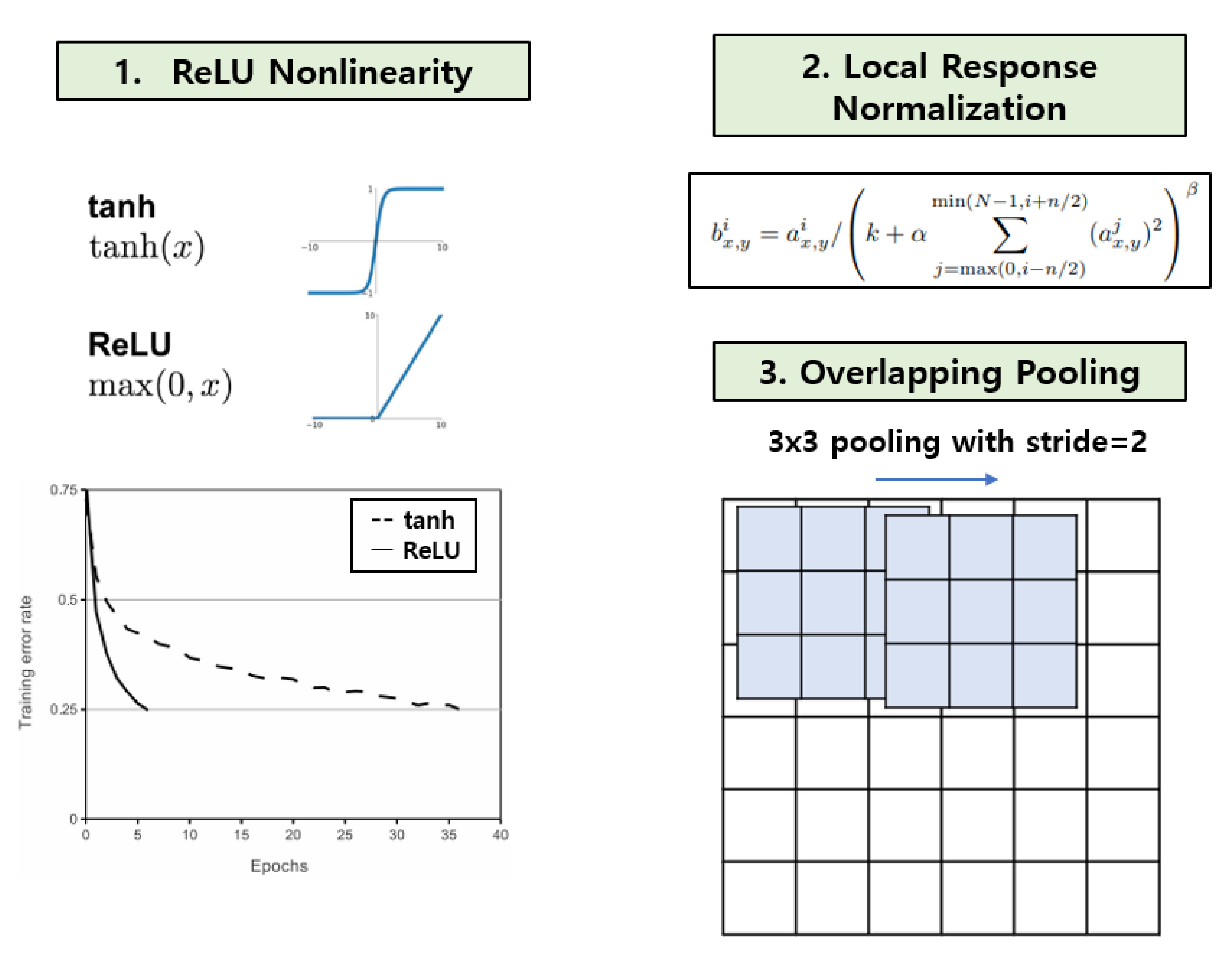


AlexNet



224
224
3
11
11
11
11
Stride
of 4
55
55
48
48
5
5
5
5
Max
pooling
27
27
128
128
3
3
3
3
3
3
3
3
Max
pooling
13
13
192
192
3
3
3
3
13
13
192
192
3
3
3
3
13
13
128
128
Max
pooling
dense
dense
dense
2048
2048
2048
2048
1000

AlexNet

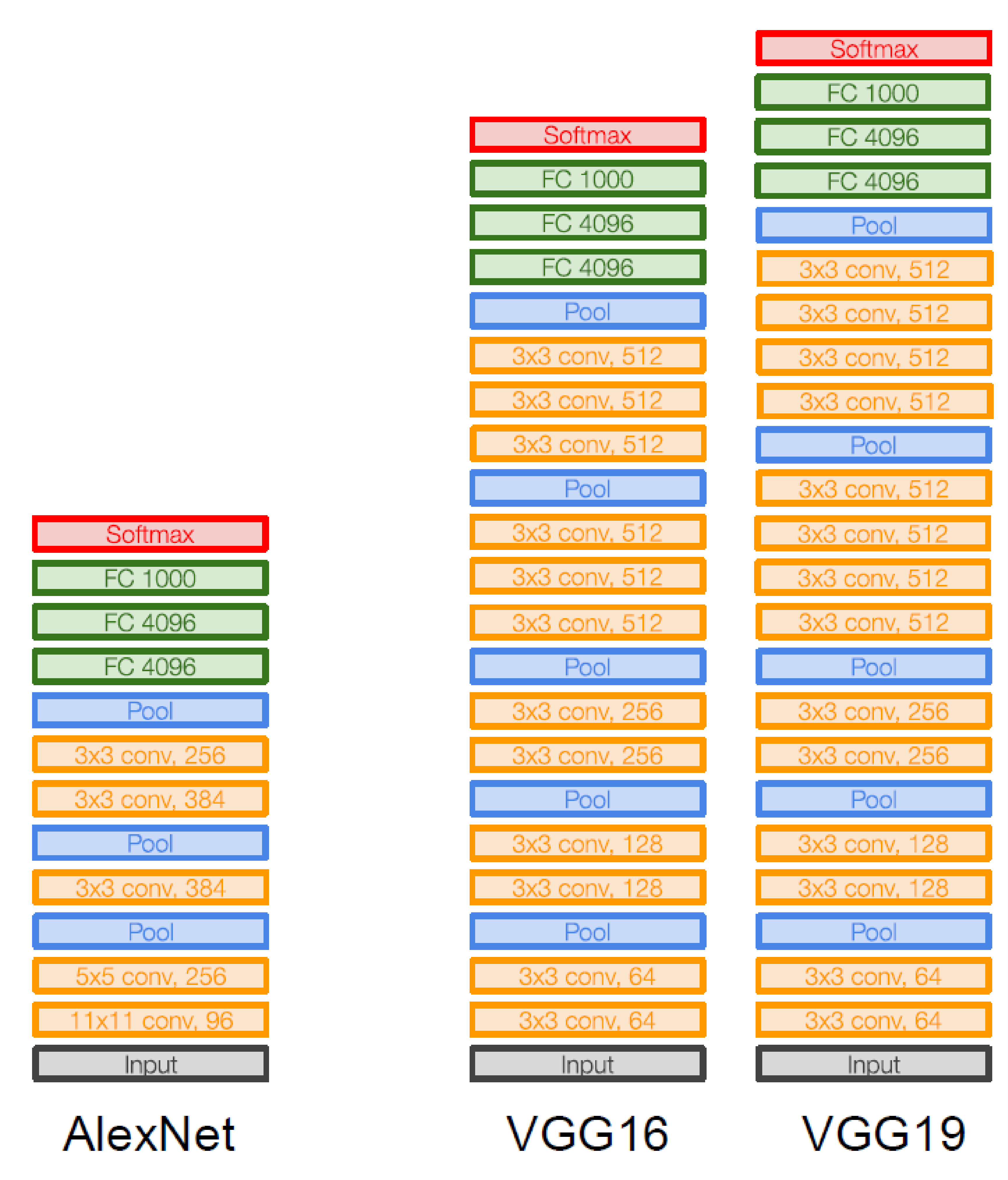


- Activation function -ReLU
- LRN
- Overlapping Pooling
- Data augmentation
- Dropout

VGG

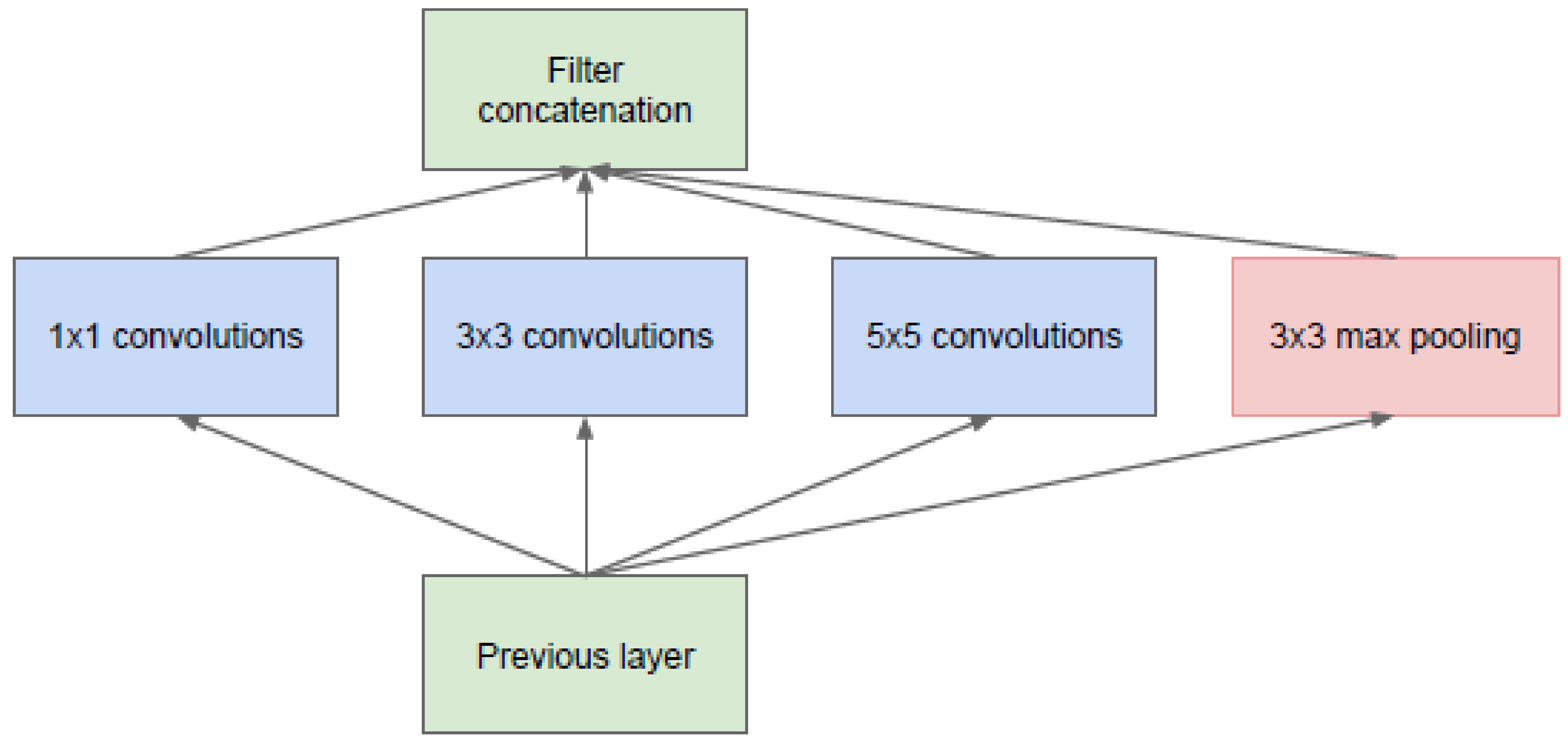
Very Deep Convolutional Networks

ConvNet Configuration					
A	A-LRN	B	C	D	E
11 weight layers	11 weight layers	13 weight layers	16 weight layers	16 weight layers	19 weight layers
input (224 × 224 RGB image)					
conv3-64	conv3-64 **LRN**	conv3-64 **conv3-64**	conv3-64 conv3-64	conv3-64 conv3-64	conv3-64 conv3-64
maxpool					
conv3-128	conv3-128	conv3-128 **conv3-128**	conv3-128 conv3-128	conv3-128 conv3-128	conv3-128 conv3-128
maxpool					
conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256	conv3-256 conv3-256 **conv1-256**	conv3-256 conv3-256 **conv3-256**	conv3-256 conv3-256 conv3-256 **conv3-256**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512	conv3-512 conv3-512 **conv1-512**	conv3-512 conv3-512 **conv3-512**	conv3-512 conv3-512 conv3-512 **conv3-512**
maxpool					
FC-4096					
FC-4096					
FC-1000					
soft-max					

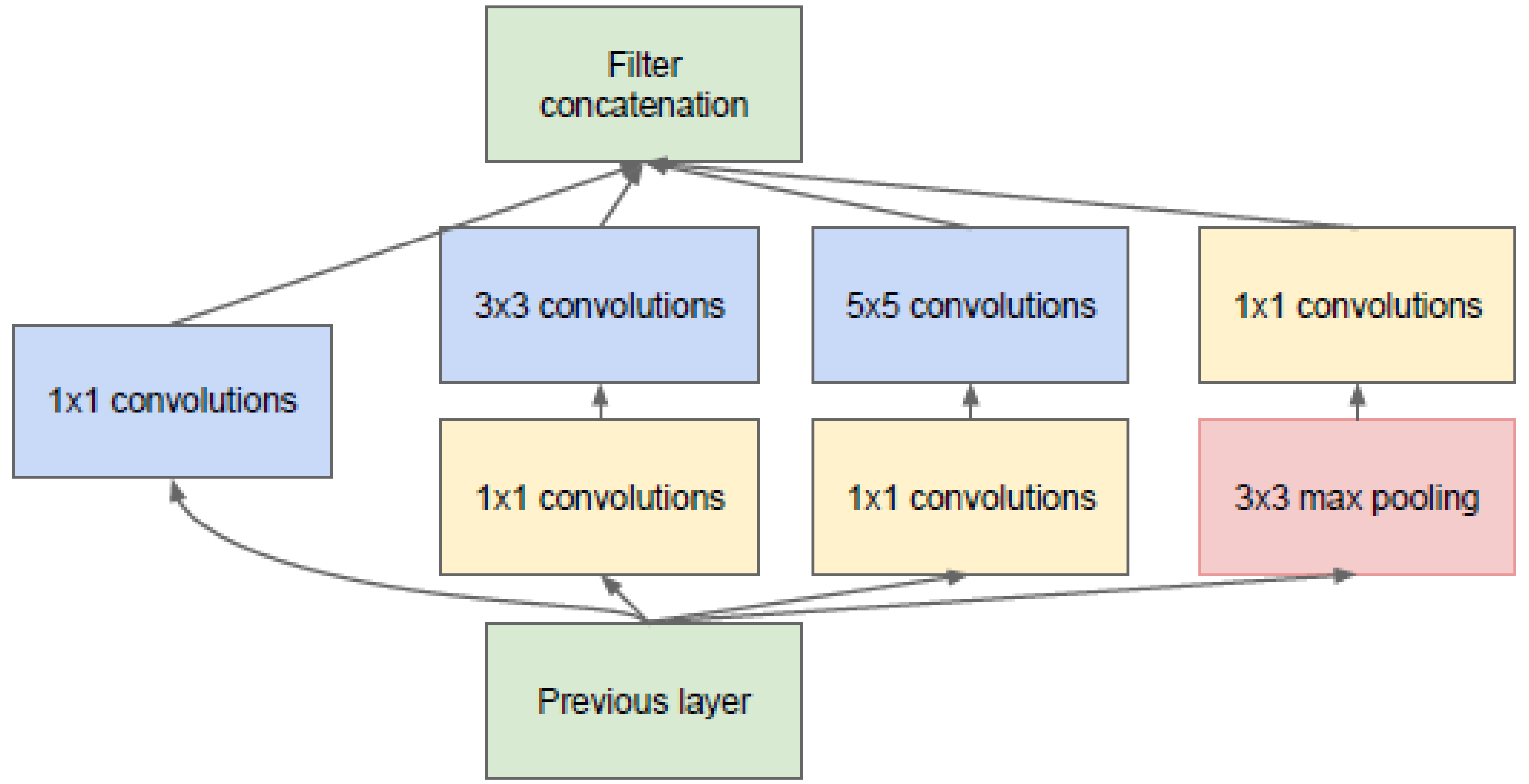


GoogLeNet

Inception module



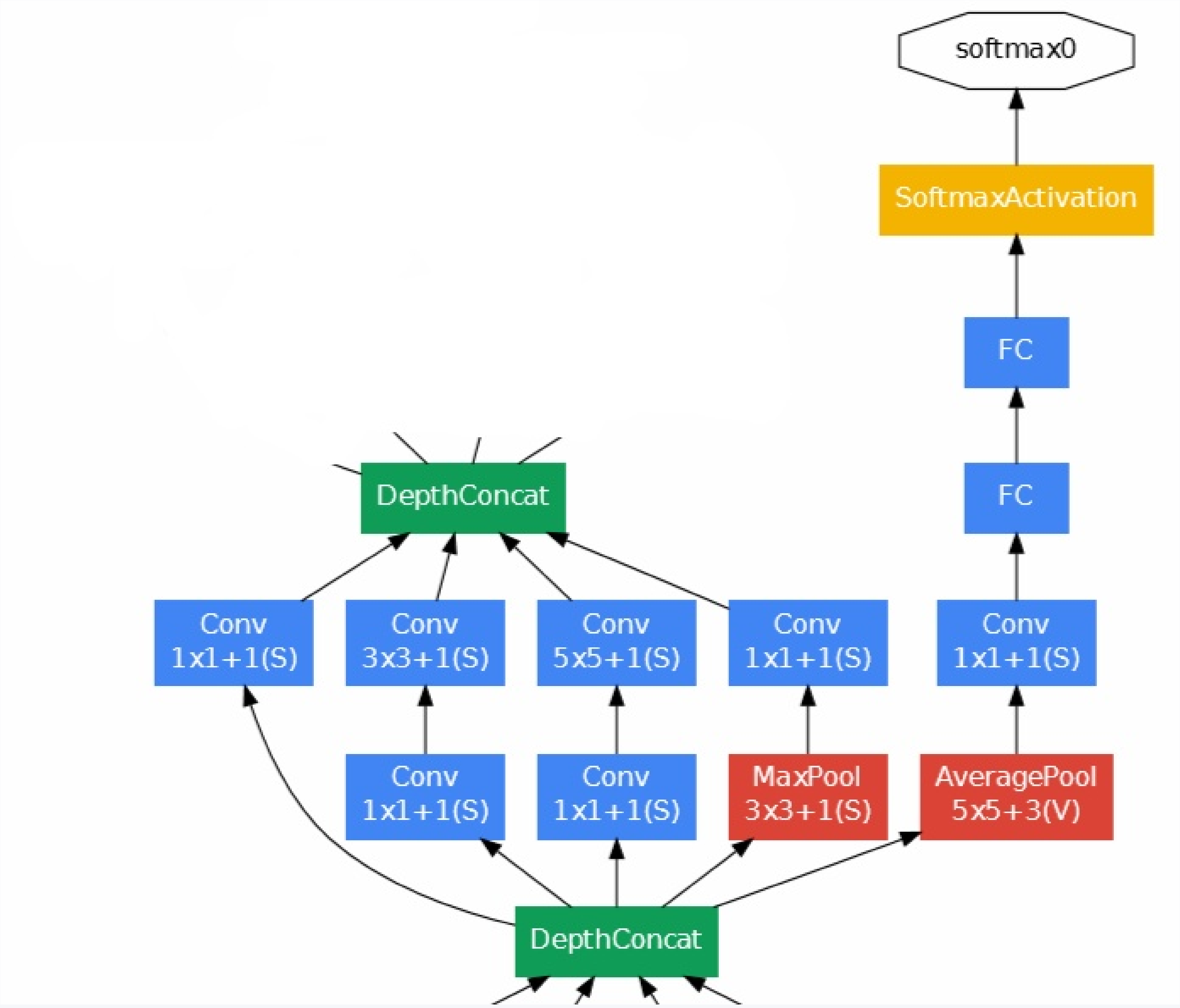
(a) Inception module, naïve version

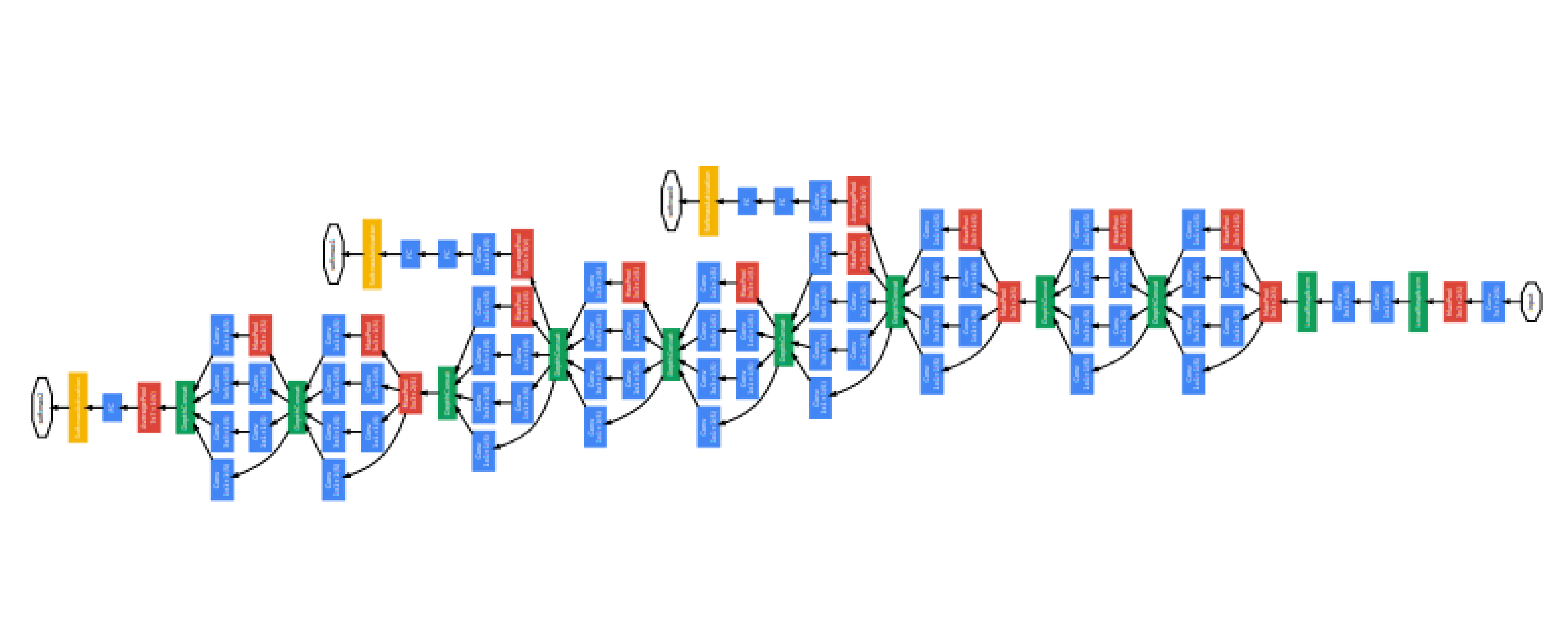


(b) Inception module with dimensionality reduction

GoogLeNet

Architecture

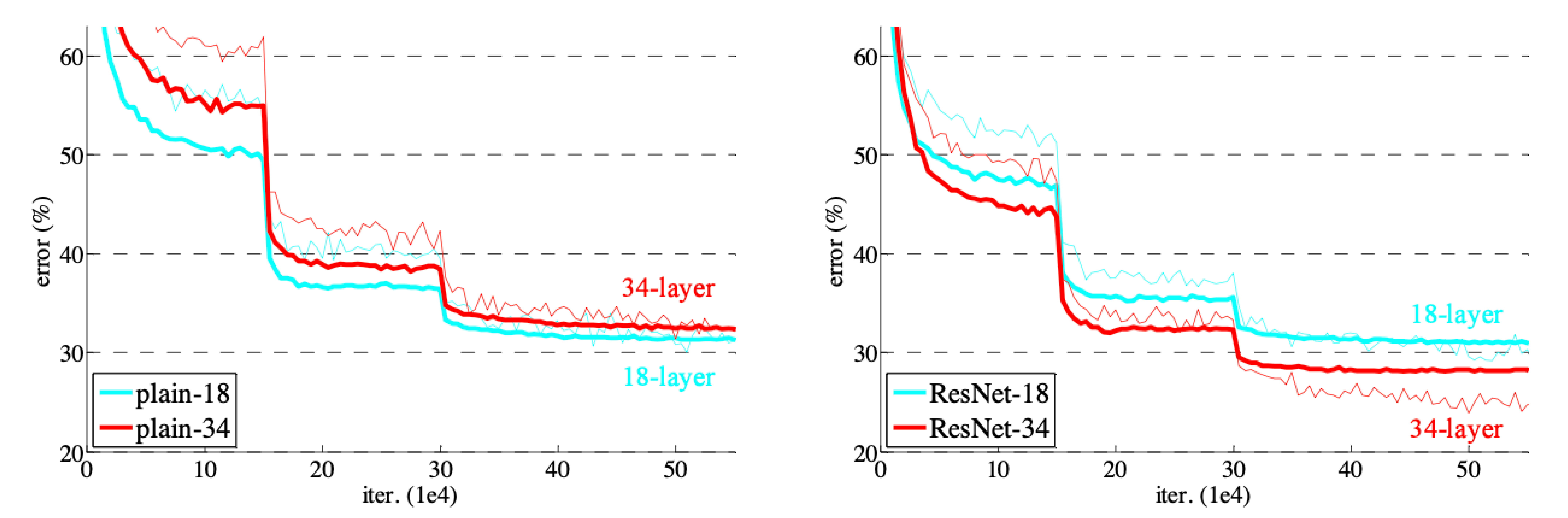




- **Auxiliary Classifier**
- **FC layer -> Global Average Pooling**

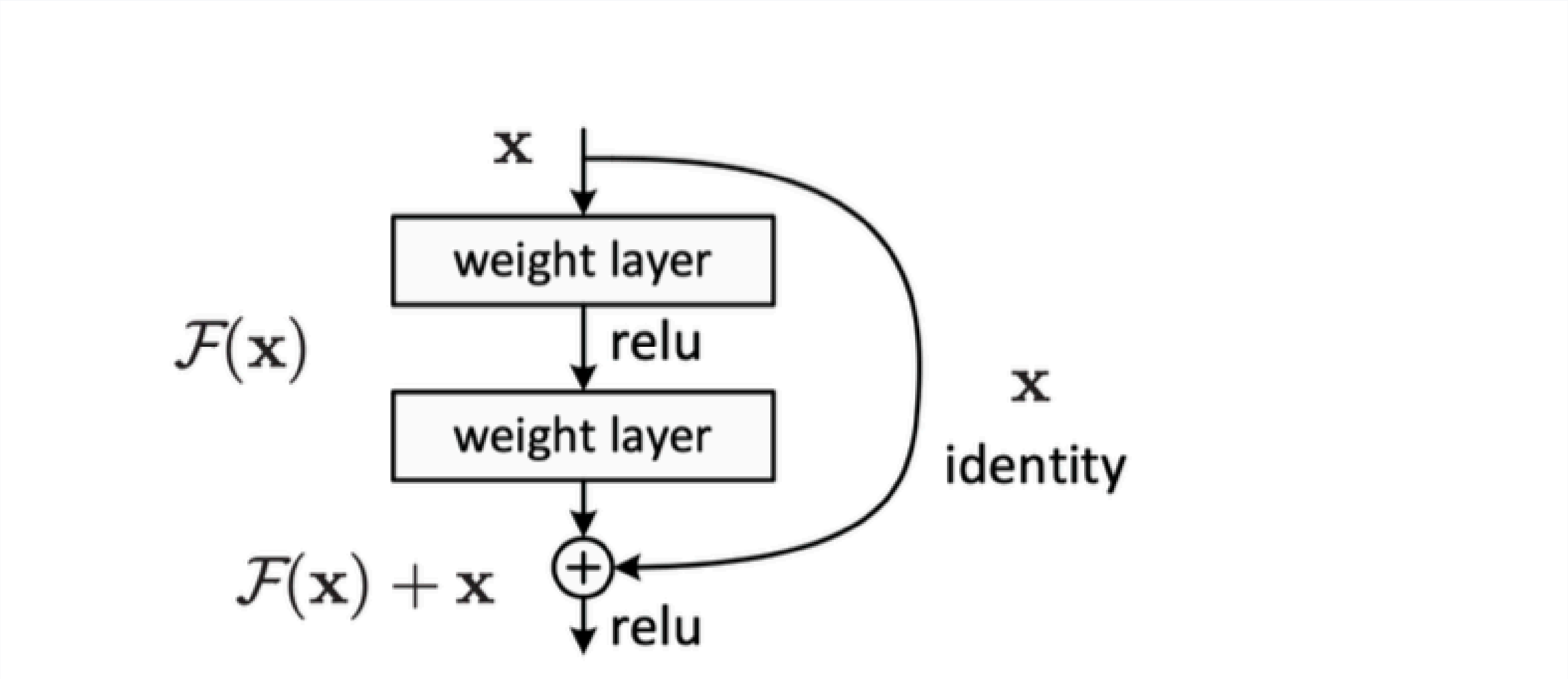
ResNet

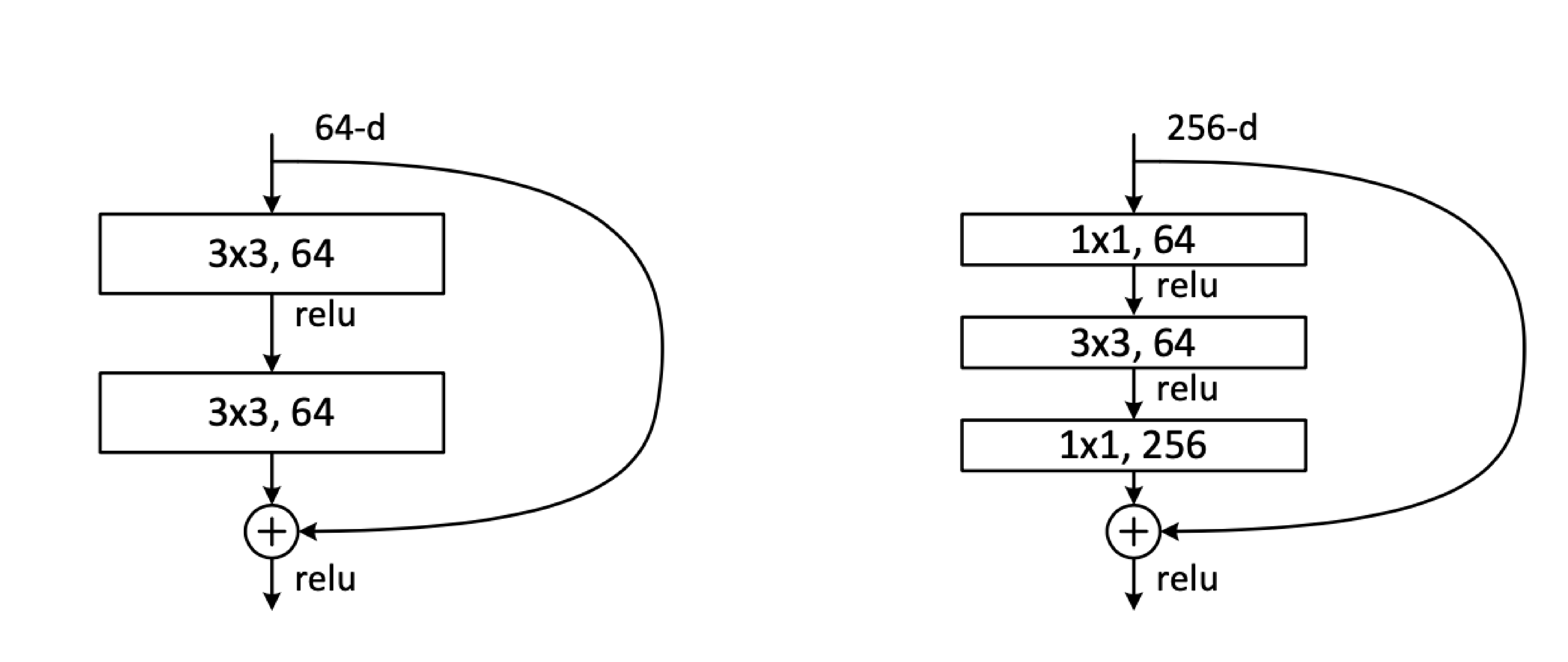
Degradataion



ResNet

Residual learning

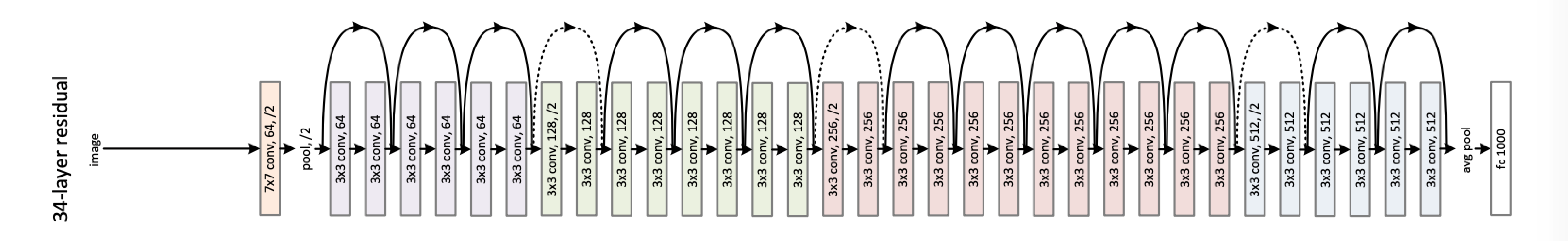


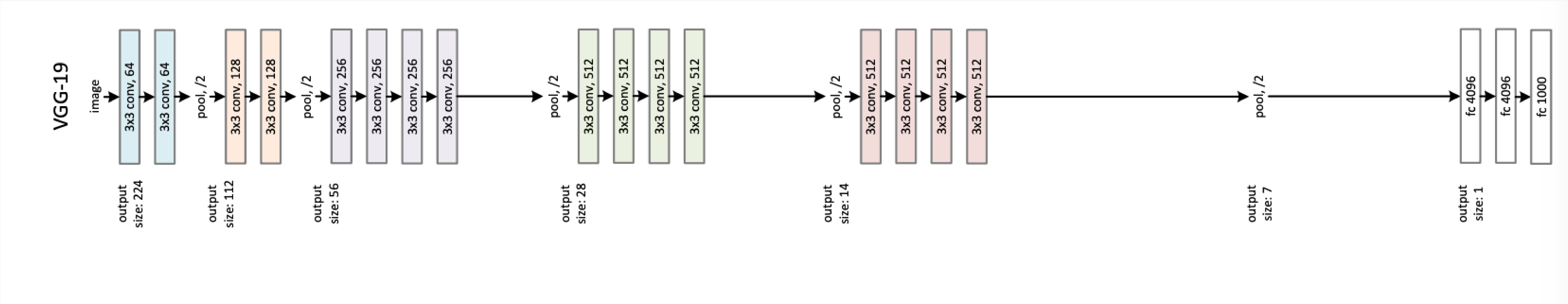


- **Identity shortcut**
- **Bottleneck residual block**

ResNet

Architecture





Dogs-vs-Cats Classification Practice

Training
- RandomResizedCrop(224)
- RandomRotation(90)
- Normalize

Validation
- Resize(224)
- Normalize

Dogs-vs-Cats Classification Practice

AlexNet

```
class AlexNet(nn.Module):
    def __init__(self, num_classes):
        self.num_classes = num_classes
        super(AlexNet, self).__init__()
        self.features = nn.Sequential(nn.Conv2d(3, 64, kernel_size=11, stride=4, padding=2),
                                      nn.ReLU(inplace=True),
                                      nn.MaxPool2d(kernel_size=3, stride=2),
                                      nn.Conv2d(64, 192, kernel_size=5, padding=2),
                                      nn.ReLU(inplace=True),
                                      nn.MaxPool2d(kernel_size=3,stride=2),
                                      nn.Conv2d(192, 384, kernel_size=3, padding=1),
                                      nn.ReLU(inplace=True),
                                      nn.Conv2d(384, 256, kernel_size=3, padding=1),
                                      nn.ReLU(inplace=True),
                                      nn.Conv2d(256, 256, kernel_size=3, padding=1),
                                      nn.ReLU(inplace=True),
                                      nn.MaxPool2d(kernel_size=3, stride=2),)

        self.avgpool = nn.AdaptiveAvgPool2d((6, 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:torch.Tensor) -> torch.Tensor:
        x = self.features(x)
        x = self.avgpool(x)
        x = torch.flatten(x, 1)
        x = self.classifier(x)
        return x
```

Dogs-vs-Cats Classification Practice

VGG

```python
class VGG(nn.Module):
    def __init__(self, features, num_classes=1000, init_weights=True):
        super(VGG, self).__init__()
        self.features = features
        self.avgpool = nn.AdaptiveAvgPool2d((7, 7))
        self.classifier = nn.Sequential(nn.Linear(512*7*7, 4096),
                                        nn.ReLU(True),
                                        nn.Linear(4096, 4096),
                                        nn.ReLU(True),
                                        nn.Linear(4096, num_classes),)
        if init_weights:
            self._initialize_weights()

    def forward(self, x):
        x = self.features(x)
        x = self.avgpool(x)
        x = x.view(x.size(0), -1)
        x = self.classifier(x)
        return x

    def _initialize_weights(self):
        for m in self.modules():
            if isinstance(m, nn.Conv2d):
                nn.init.kaiming_normal_(m.weight, mode='fan_out', nonlinearity='relu')
                if m.bias is not None:
                    nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.BatchNorm2d):
                nn.init.constant_(m.weight, 1)
                nn.init.constant_(m.bias, 0)
            elif isinstance(m, nn.Linear):
                nn.init.normal_(m.weight, 0, 0.01)
                nn.init.constant_(m.bias, 0)
```

```python
def make_layers(cfg, batch_norm=False):
    layers = []
    in_channels = 3
    for v in cfg:
        if v == 'M':
            layers += [nn.MaxPool2d(kernel_size=2, stride=2)]
        else:
            conv2d = nn.Conv2d(in_channels, v, kernel_size=3, padding=1)
            if batch_norm:
                layers += [conv2d, nn.BatchNorm2d(v), nn.ReLU(inplace=True)]
            else:
                layers += [conv2d, nn.ReLU(inplace=True)]
            in_channels = v
    return nn.Sequential(*layers)
```

Dogs-vs-Cats Classification Practice

GoogLeNet

```python
def conv_1(in_dim,out_dim):
    model = nn.Sequential(
        nn.Conv2d(in_dim,out_dim,1,1),
        nn.ReLU(),
    )
    return model

def conv_1_3(in_dim,mid_dim,out_dim):
    model = nn.Sequential(
        nn.Conv2d(in_dim,mid_dim,1,1),
        nn.ReLU(),
        nn.Conv2d(mid_dim,out_dim,3,1,1),
        nn.ReLU()
    )
    return model

def conv_1_5(in_dim,mid_dim,out_dim):
    model = nn.Sequential(
        nn.Conv2d(in_dim,mid_dim,1,1),
        nn.ReLU(),
        nn.Conv2d(mid_dim,out_dim,5,1,2),
        nn.ReLU()
    )
    return model


def max_3_1(in_dim,out_dim):
    model = nn.Sequential(
        nn.MaxPool2d(3,1,1),
        nn.Conv2d(in_dim,out_dim,1,1),
        nn.ReLU(),
    )
    return model
```

```python
class inception_module(nn.Module):
    def __init__(self,in_dim,out_dim_1,mid_dim_3,out_dim_3,mid_dim_5,out_dim_5,pool):
        super(inception_module,self).__init__()
        self.conv_1 = conv_1(in_dim,out_dim_1)
        self.conv_1_3 = conv_1_3(in_dim,mid_dim_3,out_dim_3)
        self.conv_1_5 = conv_1_5(in_dim,mid_dim_5,out_dim_5)
        self.max_3_1 = max_3_1(in_dim,pool)

    def forward(self,x):
        out_1 = self.conv_1(x)
        out_2 = self.conv_1_3(x)
        out_3 = self.conv_1_5(x)
        out_4 = self.max_3_1(x)
        output = torch.cat([out_1,out_2,out_3,out_4],1)
        return output
```

```python
class GoogLeNet(nn.Module):
    def __init__(self, base_dim, num_classes=2):
        super(GoogLeNet, self).__init__()
        self.num_classes=num_classes
        self.layer_1 = nn.Sequential(
            nn.Conv2d(3,base_dim,7,2,3),
            nn.MaxPool2d(3,2,1),
            nn.Conv2d(base_dim,base_dim*3,3,1,1),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_2 = nn.Sequential(
            inception_module(base_dim*3,64,96,128,16,32,32),
            inception_module(base_dim*4,128,128,192,32,96,64),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_3 = nn.Sequential(
            inception_module(480,192,96,208,16,48,64),
            inception_module(512,160,112,224,24,64,64),
            inception_module(512,128,128,256,24,64,64),
            inception_module(512,112,144,288,32,64,64),
            inception_module(528,256,160,320,32,128,128),
            nn.MaxPool2d(3,2,1),
        )
        self.layer_4 = nn.Sequential(
            inception_module(832,256,160,320,32,128,128),
            inception_module(832,384,192,384,48,128,128),
            nn.AvgPool2d(7,1),
        )
        self.layer_5 = nn.Dropout2d(0.4)
        self.fc_layer = nn.Linear(1024,self.num_classes)


    def forward(self, x):
        out = self.layer_1(x)
        out = self.layer_2(out)
        out = self.layer_3(out)
        out = self.layer_4(out)
        out = self.layer_5(out)
        out = out.view(batch_size,-1)
        out = self.fc_layer(out)
        return out
```

Dogs-vs-Cats Classification Practice

ResNet

```python
def conv_block_1(in_dim, out_dim, act_fn, stride=1):
    model = nn.Sequential(nn.Conv2d(in_dim, out_dim, kernel_size=1, stride=stride),
                          act_fn)
    return model

def conv_block_3(in_dim, out_dim, act_fn):
    model = nn.Sequential(nn.Conv2d(in_dim, out_dim, kernel_size=3, stride=1, padding=1),
                          act_fn)
    return model
```

```python
class BottleNeck(nn.Module):
    def __init__(self, in_dim, mid_dim, out_dim, act_fn, down=False):
        super(BottleNeck, self).__init__()
        self.act_fn = act_fn
        self.down = down

        if self.down:
            self.layer = nn.Sequential(conv_block_1(in_dim, mid_dim, act_fn, 2),
                                       conv_block_3(mid_dim, mid_dim, act_fn),
                                       conv_block_1(mid_dim, out_dim, act_fn))
            self.downsample = nn.Conv2d(in_dim, out_dim, 1, 2)

        else:
            self.layer = nn.Sequential(conv_block_1(in_dim, mid_dim, act_fn),
                                       conv_block_3(mid_dim, mid_dim, act_fn),
                                       conv_block_1(mid_dim, out_dim, act_fn))

        self.dim_equalizer = nn.Conv2d(in_dim, out_dim, kernel_size=1)

    def forward(self, x):
        if self.down:
            downsample = self.downsample(x)
            out = self.layer(x)
            out = out + downsample
        else:
            out = self.layer(x)
            if x.size() is not out.size():
                x = self.dim_equalizer(x)
            out = out + x
        return out
```

```python
class ResNet(nn.Module):
    def __init__(self, base_dim, num_classes=2):
        super(Resnet, self).__init__()

        self.act_fn = nn.ReLU()

        self.layer_1 = nn.Sequential(nn.Conv2d(3, base_dim, 7, 2, 3),
                                     nn.ReLU(),
                                     nn.MaxPool2d(3, 2, 1))

        self.layer_2 = nn.Sequential(BottelNeck(base_dim, base_dim, base_dim*4, self.act_fn),
                                     BottelNeck(base_dim*4, base_dim, base_dim*4, self.act_fn),
                                     BottelNeck(base_dim*4, base_dim, base_dim*4, self.act_fn, down=True),)

        self.layer_3 = nn.Sequential(BottleNeck(base_dim*4, base_dim*2, base_dim*8, self.act_fn),
                                     BottleNeck(base_dim*8, base_dim*2, base_dim*8, self.act_fn),
                                     BottleNeck(base_dim*8, base_dim*2, base_dim*8, self.act_fn),
                                     BottleNeck(base_dim*8, base_dim*2, base_dim*8, self.act_fn, down=True),)

        self.layer_4 = nn.Sequential(BottleNeck(base_dim*8, base_dim*4, base_dim*16, self.act_fn),
                                     BottleNeck(base_dim*16, base_dim*4, base_dim*16, self.act_fn),
                                     BottleNeck(base_dim*16, base_dim*4, base_dim*16, self.act_fn),
                                     BottleNeck(base_dim*16, base_dim*4, base_dim*16, self.act_fn),
                                     BottleNeck(base_dim*16, base_dim*4, base_dim*16, self.act_fn),
                                     BottleNeck(base_dim*16, base_dim*4, base_dim*16, self.act_fn, down=True),)

        self.layer_5 = nn.Sequential(BottleNeck(base_dim*16, base_dim*8, base_dim*32, self.act_fn),
                                     BottleNeck(base_dim*32, base_dim*8, base_dim*32, self.act_fn),
                                      BottleNeck(base_dim*32, base_dim*8, base_dim*32, self.act_fn),)

        self.avgpool = nn.AvgPool2d(7, 1)

        self.fc_layer = nn.Linear(base_dim*32, num_classes)

    def forward(self, x):
        out = self.layer_1(x)
        out = self.layer_2(out)
        out = self.layer_3(out)
        out = self.layer_4(out)
        out = self.layer_5(out)
        out = self.avgpool(out)
        out = out.view(batch_size, -1)
        out = self.fc_layer(out)
        return out
```

Dogs-vs-Cats Classification Practice

Accuracy

- **AlexNet : 0.9515**
- **VGG16: 0.9750**
- **GoogLeNet: 0.9850**
- **Resnet34: 0.9885**