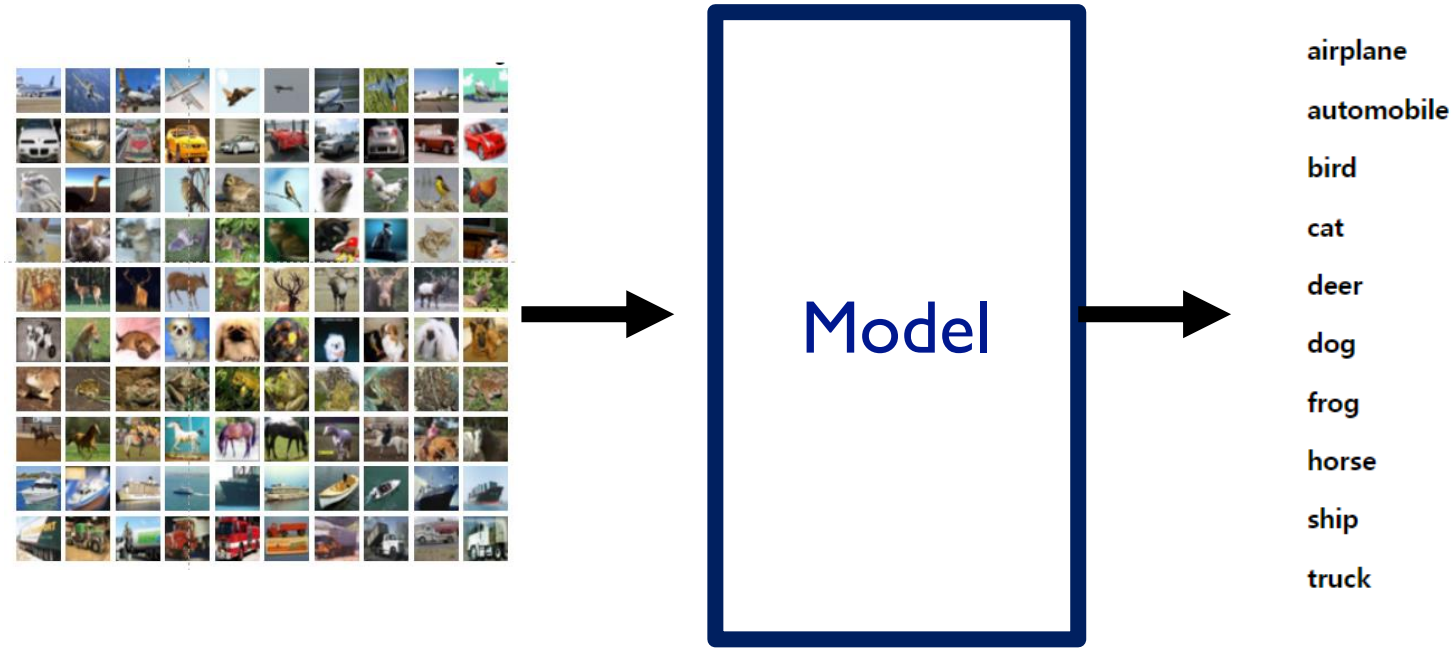
# Day 3

2023.10.23

# Model에게 원하는 것?

- Regression -> nn.MSELoss()
- Binary classification -> nn.BCELoss()
- **Multi class classification -> nn.CrossEntropyLoss()**



# Image Classification

- Cifar 10 dataset


airplane

automobile

bird

cat

deer

dog

frog

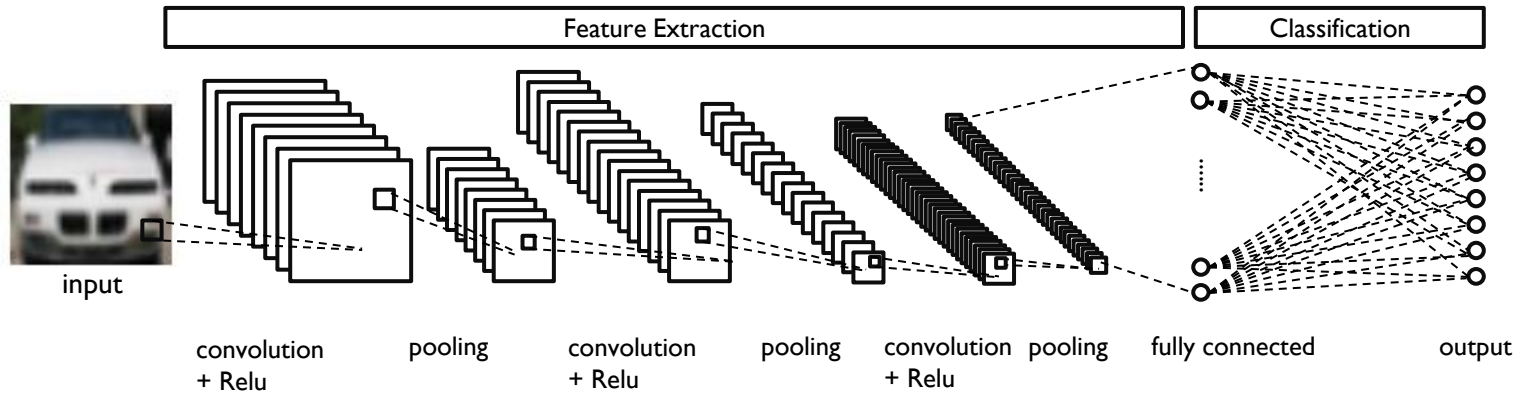
horse

ship

truck

# Coding 전에 생각해 볼 것

- 입력 데이터 : image( C , H , W )
- 출력 데이터 : class number
- Optimizer:  ?
- Loss function: cross entropy



# 1. Dataset

- 입, 출력 데이터

```
transform_CIFAR10 = transforms.Compose([
    transforms.ToTensor()
])
train_loader = torch.utils.data.DataLoader(
    datasets.CIFAR10(
        root      = './data_CIFAR10',
        train     = True,
        download  = True,
        transform = transform_CIFAR10),
    batch_size=BATCH_SIZE,
    shuffle=True
)
test_loader = torch.utils.data.DataLoader(
    datasets.CIFAR10(
        root      = './data_CIFAR10',
        train     = False,
        download  = True,
        transform = transform_CIFAR10),
    batch_size=BATCH_SIZE,
    shuffle=True
)
```

# 2. Model

## CNN + Linear

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3)
        self.conv3 = nn.Conv2d(16, 24, kernel_size=3)
        #self.conv3_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(24 * 2 * 2, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = F.relu(F.max_pool2d(F.dropout(self.conv3(x)), 2))

        x = x.view(-1, 24 * 2 * 2)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```

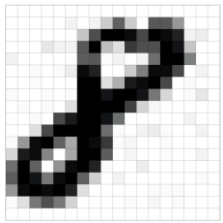
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 8, 30, 30]             224
            Conv2d-2         [-1, 16, 13, 13]           1,168
            Conv2d-3          [-1, 24, 4, 4]            3,480
         Dropout2d-4          [-1, 24, 4, 4]                0
            Linear-5               [-1, 128]           12,416
            Linear-6                [-1, 10]            1,290
================================================================
Total params: 18,578
Trainable params: 18,578
Non-trainable params: 0
```

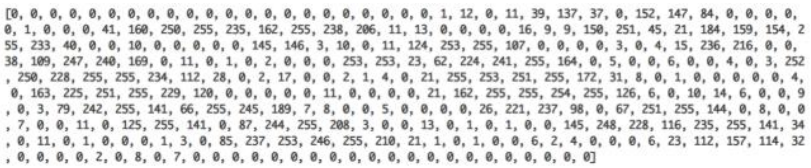
```
----------------------------------------------------------------
        Layer (type)               Output Shape         Param #
================================================================
            Conv2d-1          [-1, 8, 30, 30]             224
            Conv2d-2         [-1, 16, 13, 13]           1,168
            Conv2d-3          [-1, 24, 4, 4]            3,480
            Linear-4               [-1, 128]           12,416
            Linear-5                [-1, 10]            1,290
================================================================
Total params: 18,578
Trainable params: 18,578
Non-trainable params: 0
```

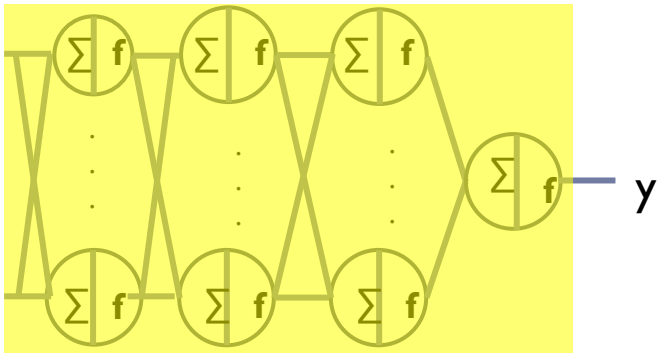
# nn.Linear

RECAP

```
class Net(nn.Module):
    def __init__(self):
        super(Net, self).__init__()
        self.fc1 = nn.Linear(784, 256)
        self.fc2 = nn.Linear(256, 128)
        self.fc3 = nn.Linear(128, 10)

    def forward(self, x):
        x = x.view(-1, 784)
        x = torch.sigmoid(self.fc1(x))
        x = torch.sigmoid(self.fc2(x))
        x = self.fc3(x)
        return x
```

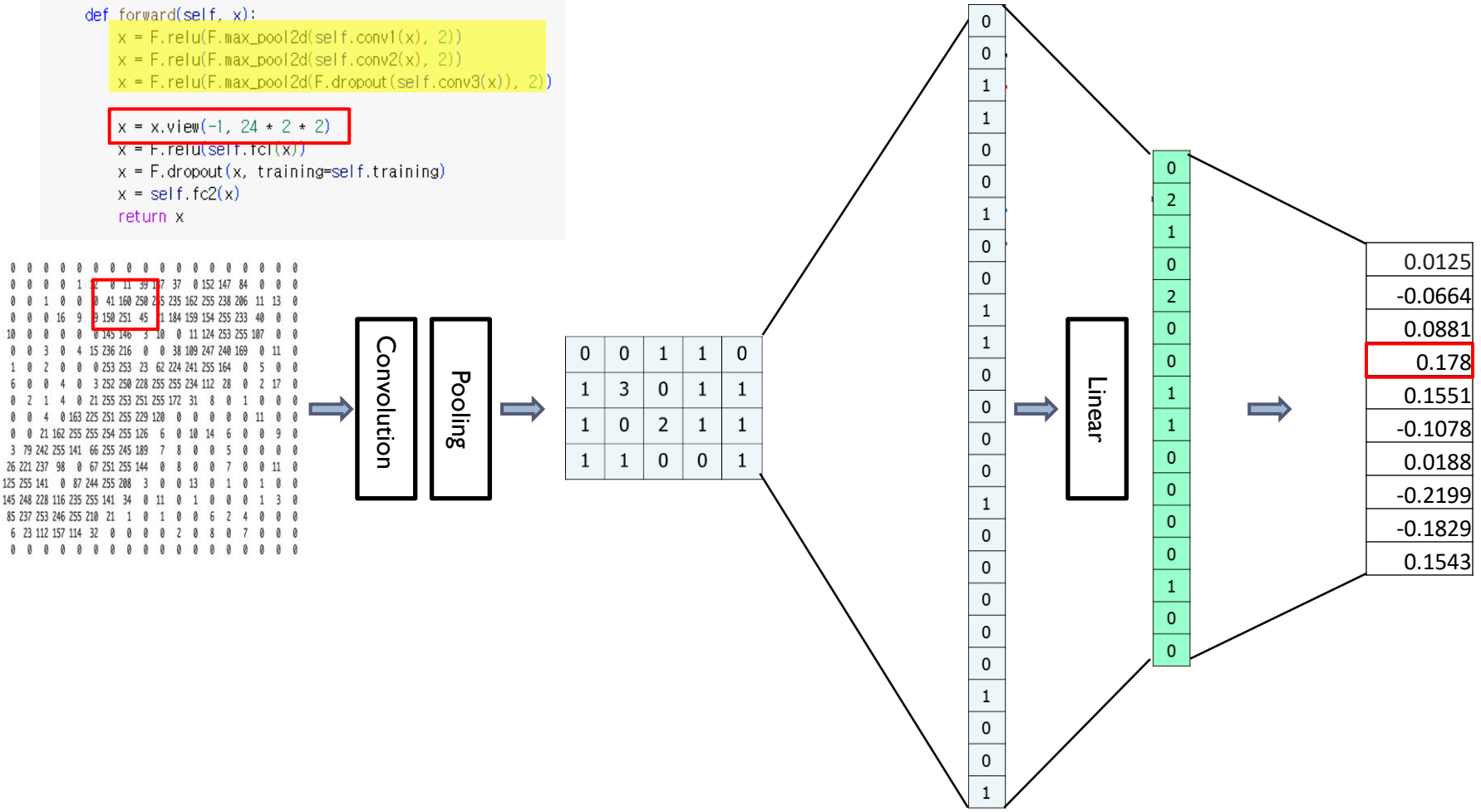






# nn.Conv2d

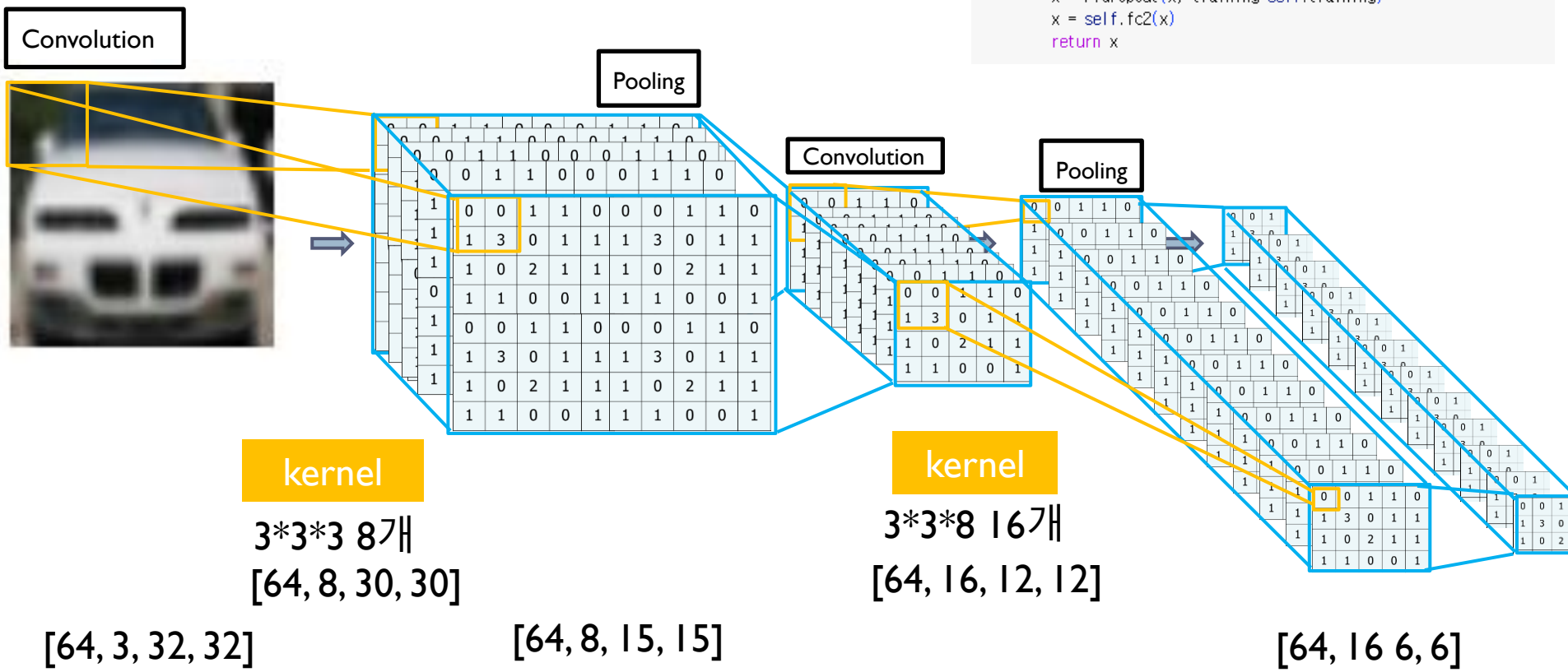
```
def forward(self, x):
    x = F.relu(F.max_pool2d(self.conv1(x), 2))
    x = F.relu(F.max_pool2d(self.conv2(x), 2))
    x = F.relu(F.max_pool2d(F.dropout(self.conv3(x)), 2))

    x = x.view(-1, 24 * 2 * 2)
    x = F.relu(self.fc1(x))
    x = F.dropout(x, training=self.training)
    x = self.fc2(x)
    return x
```



# feature map size

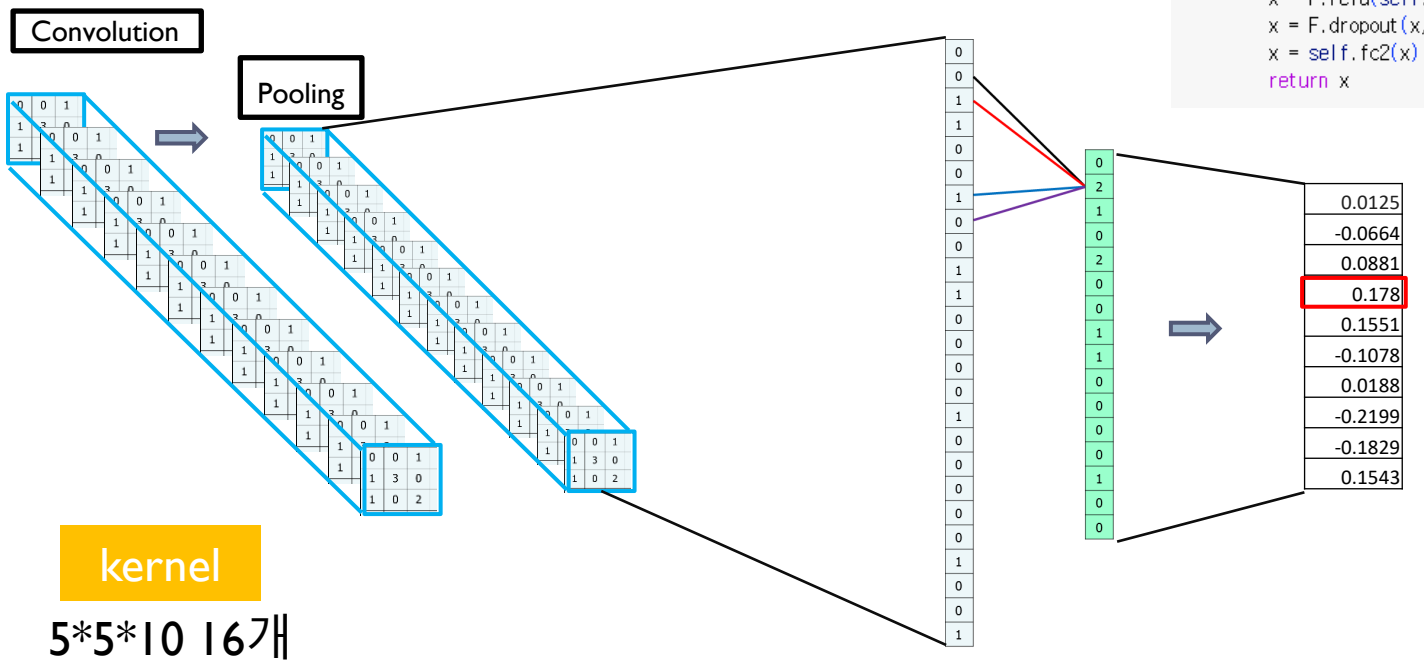
$$out = \frac{in + 2p - f}{s} + 1$$

```python
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3)
        self.conv3 = nn.Conv2d(16, 24, kernel_size=3)
        self.conv3_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(24 * 2 * 2, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = F.relu(F.max_pool2d(self.conv3_drop(self.conv3(x)), 2))
        print( x.shape )
        x = x.view(-1, 24 * 2 * 2)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```



# feature map size

```
class CNN(nn.Module):
    def __init__(self):
        super(CNN, self).__init__()
        self.conv1 = nn.Conv2d(3, 8, kernel_size=3)
        self.conv2 = nn.Conv2d(8, 16, kernel_size=3)
        self.conv3 = nn.Conv2d(16, 24, kernel_size=3)
        self.conv3_drop = nn.Dropout2d()
        self.fc1 = nn.Linear(24 * 2 * 2, 128)
        self.fc2 = nn.Linear(128, 10)

    def forward(self, x):
        x = F.relu(F.max_pool2d(self.conv1(x), 2))
        x = F.relu(F.max_pool2d(self.conv2(x), 2))
        x = F.relu(F.max_pool2d(self.conv3_drop(self.conv3(x)), 2))
        print( x.shape )
        x = x.view(-1, 24 * 2 * 2)
        x = F.relu(self.fc1(x))
        x = F.dropout(x, training=self.training)
        x = self.fc2(x)
        return x
```




kernel

5*5*10 16개

[64, 24 5, 5]

[64, 24, 2, 2] [64, 24*2*2] [64, 128] [64, 10]

# MaxPool2d

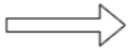
For each block, or "pool", the operation simply involves computing the [latex]max[/latex] value, like this:

| | | | |
|---|---|---|---|
| 4 | 3 | 1 | 5 |
| 1 | 3 | 4 | 8 |
| 4 | 5 | 4 | 3 |
| 6 | 5 | 9 | 4 |

**Max([4, 3, 1, 3]) = 4**

Doing so for each pool, we get a nicely downsampled outcome, greatly benefiting the spatial hierarchy we need:

| | | | |
|---|---|---|---|
| 4 | 3 | 1 | 5 |
| 1 | 3 | 4 | 8 |
| 4 | 5 | 4 | 3 |
| 6 | 5 | 9 | 4 |



| | |
|---|---|
| 4 | 8 |
| 6 | 9 |

Docs > torch.nn > MaxPool2d

## MAXPOOL2D

CLASS `torch.nn.MaxPool2d(kernel_size, stride=None, padding=0, dilation=1, return_indices=False, ceil_mode=False)` [SOURCE]

Applies a 2D max pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size $(N, C, H, W)$, output $(N, C, H_{out}, W_{out})$ and `kernel_size` $(kH, kW)$ can be precisely described as:

$$out(N_i, C_j, h, w) = \max_{m=0,\ldots,kH-1} \max_{n=0,\ldots,kW-1} \text{input}(N_i, C_j, \text{stride}[0] \times h + m, \text{stride}[1] \times w + n)$$

If `padding` is non-zero, then the input is implicitly padded with negative infinity on both sides for `padding` number of points. `dilation` controls the spacing between the kernel points. It is harder to describe, but this link has a nice visualization of what `dilation` does.

https://github.com/christianversloot/machine-learning-articles/blob/main/what-are-max-pooling-average-pooling-global-max-pooling-and-global-average-pooling.md

# AvgPool2d

## Average Pooling

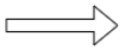
Another type of pooling layers is the Average Pooling layer. Here, rather than a `max` value, the `avg` for each block is computed:

| | | | |
|---|---|---|---|
| 4 | 3 | 1 | 5 |
| 1 | 3 | 4 | 8 |
| 4 | 5 | 4 | 3 |
| 6 | 5 | 9 | 4 |

**Avg([4, 3, 1, 3]) = 2.75**

As you can see, the output is also different - and less extreme compared to Max Pooling:

| | | | |
|---|---|---|---|
| 4 | 3 | 1 | 5 |
| 1 | 3 | 4 | 8 |
| 4 | 5 | 4 | 3 |
| 6 | 5 | 9 | 4 |



| | |
|---|---|
| 2.8 | 4.5 |
| 5.3 | 5.0 |

Docs > torch.nn > AvgPool2d

### AVGPOOL2D

CLASS `torch.nn.AvgPool2d(kernel_size, stride=None, padding=0, ceil_mode=False, count_include_pad=True, divisor_override=None)` [SOURCE]

Applies a 2D average pooling over an input signal composed of several input planes.

In the simplest case, the output value of the layer with input size $(N, C, H, W)$, output $(N, C, H_{out}, W_{out})$ and `kernel_size` $(kH, kW)$ can be precisely described as:

$$out(N_i, C_j, h, w) = \frac{1}{kH * kW} \sum_{m=0}^{kH-1} \sum_{n=0}^{kW-1} input(N_i, C_j, stride[0] \times h + m, stride[1] \times w + n$$

If `padding` is non-zero, then the input is implicitly zero-padded on both sides for `padding` number of points.

# Global Max Pooling

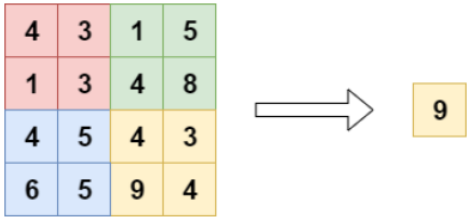
## Global Max Pooling

Another type of pooling layer is the Global Max Pooling layer. Here, we set the pool size equal to the input size, so that the `max` of the entire input is computed as the output value (Dernoncourt, 2017):

| | | | |
|---|---|---|---|
| 4 | 3 | 1 | 5 |
| 1 | 3 | 4 | 8 |
| 4 | 5 | 4 | 3 |
| 6 | 5 | 9 | 4 |

Max( [4, 3, 1, 5]
[1, 3, 4, 8]
[4, 5, 4, 3]
[6, 5, 9, 4] ) = 9

Or, visualizing it differently:




## ADAPTIVEMAXPOOL2D

CLASS `torch.nn.AdaptiveMaxPool2d(output_size, return_indices=False)` [SOURCE]

Applies a 2D adaptive max pooling over an input signal composed of several input planes.

The output is of size $H_{out} \times W_{out}$, for any input size. The number of output features is equal to the number of input planes.

Parameters

- **output_size** – the target output size of the image of the form $H_{out} \times W_{out}$. Can be a tuple $(H_{out}, W_{out})$ or a single $H_{out}$ for a square image $H_{out} \times H_{out}$. $H_{out}$ and $W_{out}$ can be either a `int`, or `None` which means the size will be the same as that of the input.
- **return_indices** – if `True`, will return the indices along with the outputs. Useful to pass to nn.MaxUnpool2d. Default: `False`

Shape:

- Input: $(N, C, H_{in}, W_{in})$ or $(C, H_{in}, W_{in})$.
- Output: $(N, C, H_{out}, W_{out})$ or $(C, H_{out}, W_{out})$, where $(H_{out}, W_{out}) = \text{output\_size}$.

Examples

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveMaxPool2d((5,7))
>>> input = torch.randn(1, 64, 8, 9)
>>> output = m(input)
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveMaxPool2d(7)
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
>>> # target output size of 10x7
>>> m = nn.AdaptiveMaxPool2d((None, 7))
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
```

# Global Avg Pooling

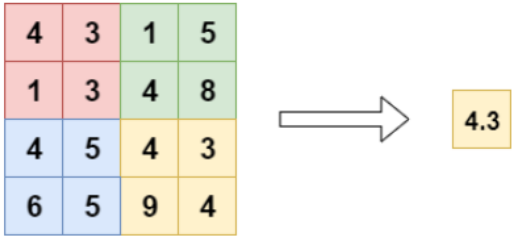
## Global Average Pooling

When applying Global Average Pooling, the pool size is still set to the size of the layer input, but rather than the maximum, the average of the pool is taken:

| | | | |
|---|---|---|---|
| 4 | 3 | 1 | 5 |
| 1 | 3 | 4 | 8 |
| 4 | 5 | 4 | 3 |
| 6 | 5 | 9 | 4 |

Avg( [4, 3, 1, 5] [1, 3, 4, 8] [4, 5, 4, 3] [6, 5, 9, 4] ) = 4.3125

Or, once again when visualized differently:




CLASS torch.nn.AdaptiveAvgPool2d(*output_size*) [SOURCE]

Applies a 2D adaptive average pooling over an input signal composed of several input planes.

The output is of size H x W, for any input size. The number of output features is equal to the number of input planes.

Parameters

**output_size** – the target output size of the image of the form H x W. Can be a tuple (H, W) or a single H for a square image H x H. H and W can be either a `int`, or `None` which means the size will be the same as that of the input.

Shape:

- Input: $(N, C, H_{in}, W_{in})$ or $(C, H_{in}, W_{in})$.
- Output: $(N, C, S_0, S_1)$ or $(C, S_0, S_1)$, where $S = \text{output\_size}$.

Examples

```
>>> # target output size of 5x7
>>> m = nn.AdaptiveAvgPool2d((5,7))
>>> input = torch.randn(1, 64, 8, 9)
>>> output = m(input)
>>> # target output size of 7x7 (square)
>>> m = nn.AdaptiveAvgPool2d(7)
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
>>> # target output size of 10x7
>>> m = nn.AdaptiveAvgPool2d((None, 7))
>>> input = torch.randn(1, 64, 10, 9)
>>> output = m(input)
```

# ResNet

| layer name | output size | 18-layer | 34-layer | 50-layer |
|---|---|---|---|---|
| conv1 | 112×112 | 7×7, 64, stride | | |
| | | 3×3 max pool, stri | | |
| conv2_x | 56×56 | [3×3, 64; 3×3, 64] ×2 | [3×3, 64; 3×3, 64] ×3 | [1×1, 64; 3×3, 64; 1×1, 256] ×3 |
| conv3_x | 28×28 | [3×3, 128; 3×3, 128] ×2 | [3×3, 128; 3×3, 128] ×4 | [1×1, 128; 3×3, 128; 1×1, 512] ×4 |
| conv4_x | 14×14 | [3×3, 256; 3×3, 256] ×2 | [3×3, 256; 3×3, 256] ×6 | [1×1, 256; 3×3, 256; 1×1, 1024] ×6 |
| conv5_x | 7×7 | [3×3, 512; 3×3, 512] ×2 | [3×3, 512; 3×3, 512] ×3 | [1×1, 512; 3×3, 512; 1×1, 2048] ×3 |
| | 1×1 | average pool, 1000-d fc | | |
| FLOPs | | $1.8\times10^9$ | $3.6\times10^9$ | $3.8\times10^9$ |

① `self.avgpool = nn.AdaptiveAvgPool2d((1, 1))`

```
class ResNet(nn.Module):
    def __init__(self, block, num_blocks, num_classes=10):
        super(ResNet, self).__init__()
        self.in_planes = 64

        self.conv1 = nn.Conv2d(3, 64, kernel_size=3,
                               stride=1, padding=1, bias=False)
        self.bn1 = nn.BatchNorm2d(64)
        self.layer1 = self._make_layer(block, 64, num_blocks[0], stride=1)
        self.layer2 = self._make_layer(block, 128, num_blocks[1], stride=2)
        self.layer3 = self._make_layer(block, 256, num_blocks[2], stride=2)
        self.layer4 = self._make_layer(block, 512, num_blocks[3], stride=2)
        self.linear = nn.Linear(512*block.expansion, num_classes)

    def _make_layer(self, block, planes, num_blocks, stride):
        strides = [stride] + [1]*(num_blocks-1)
        layers = []
        for stride in strides:
            layers.append(block(self.in_planes, planes, stride))
            self.in_planes = planes * block.expansion
        return nn.Sequential(*layers)

    def forward(self, x):
        out = F.relu(self.bn1(self.conv1(x)))
        out = self.layer1(out)
        out = self.layer2(out)
        out = self.layer3(out)
        out = self.layer4(out)
        out = F.avg_pool2d(out, 4)
        out = out.view(out.size(0), -1)
        out = self.linear(out)
        return out
```

② `out = F.avg_pool2d(out, 4)`

# Question and Answer