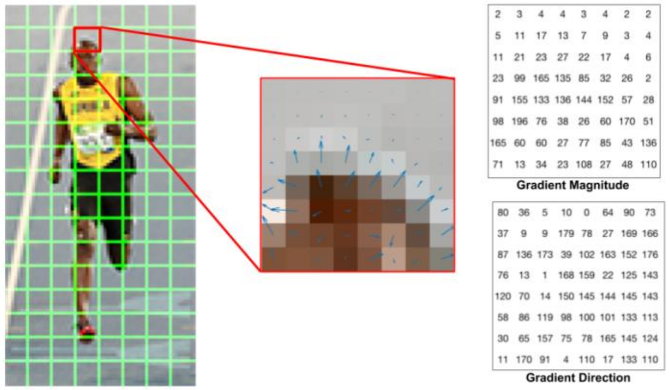
DSCI 565: MODERN CONVOLUTIONAL NEURAL NETWORKS

Ke-Thia Yao

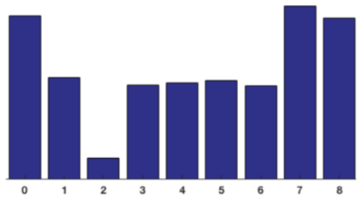
Lecture 9: 2025-09-24

Before Deep Convolutional Neural Networks

- Rather than *end-to-end* training, classical vision pipelines would
  - Manually preprocess the image dataset and extract features using extractors, like SIFT (scale-invariant feature transform), SURF (speeded up robust features) and HOG (histograms of oriented gradient)
  - Use these features to train a classifier (such as a linear model, or kernel method)
- Since everyone used the same classifier algorithms, improvements came from tweaking the preprocessing pipeline and feature extraction



Center : The RGB patch and gradients represented using arrows. Right : The gradients in the same patch represented as numbers

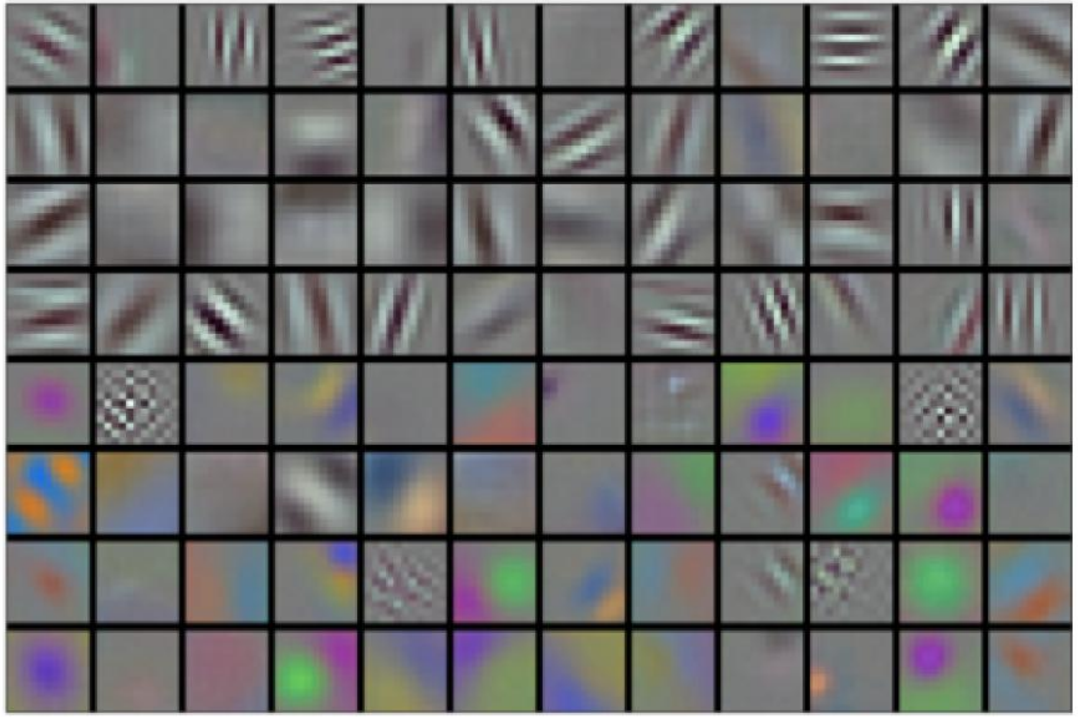


https://learnopencv.com/histogram-of-oriented-gradients/

AlexNet

- AlexNet (Krizhevksy et al. 2012) is the first deep CNN to surpass classical vision approaches
- AlexNet won the 2012 ImageNet competition by ~10 percentage points
- Representational Learning: AlexNet directly learns previously hand-crafted vision representations

First layer convolutional filters of AlexNet



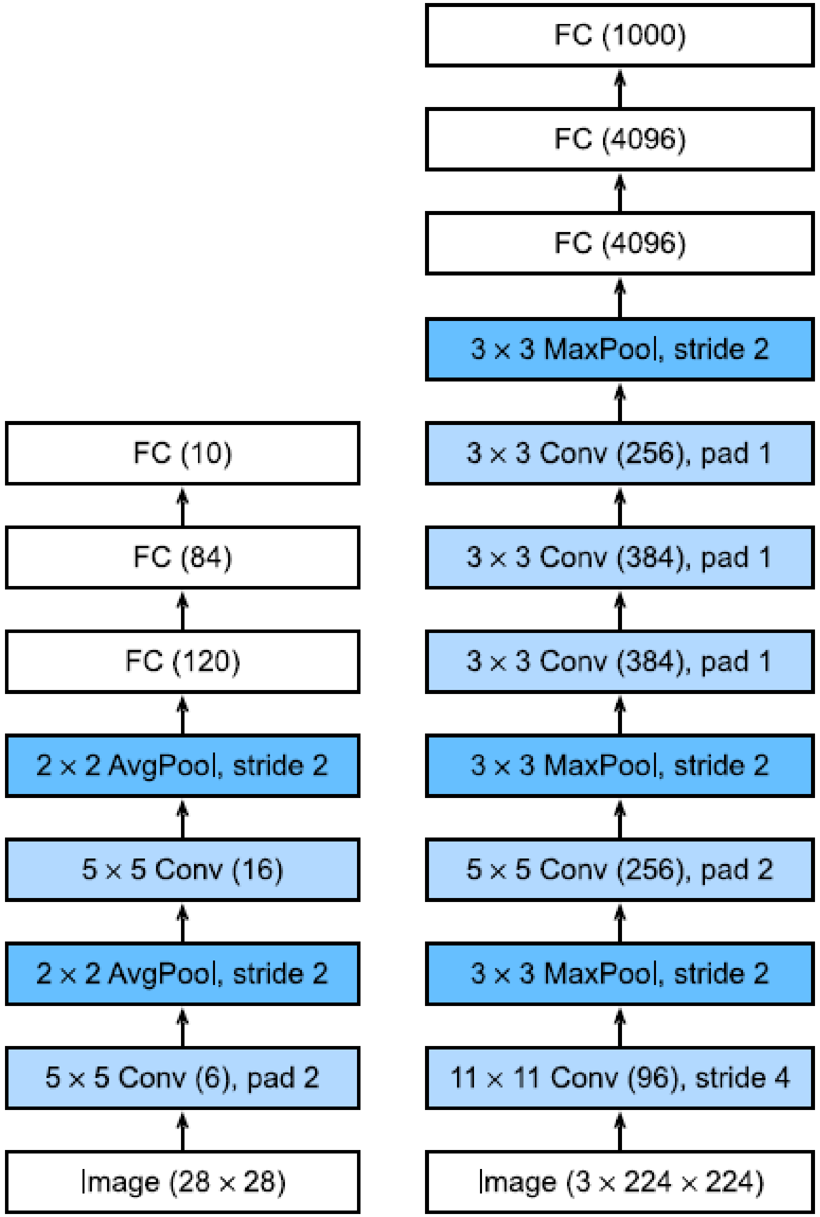
8x12= 96

Data and Hardware Needed

- Architecturally AlexNext (2012) is similar to LeNet (1995)
- Why did it take so long? What was missing?
- Data
  - Deep networks require large training datasets
  - ImageNet has 1 million images of 228x288 resolution with 1000 classes
  - Compared to CIFAR-10 60,000 images of 32x32 with 10 classes
- Hardware
  - Deep networks require compute power
  - AlexNet used two GPUs (NVIDIA GTX 580 with 1.5 TFLOPs each)
  - For 16-bit precision NVIDIA A100 GPU offers 300 TFLOPS, and H100 offers >1,500 TFLOPS

AlexNet Archtecture

- AlexNet has 8 CNN layers and 3 fully connected layers
- Compared with LeNet
  - First conv size is much larger (11x11 versus 5x5), because the image size is larger
  - More than 10X filter channels
  - Uses max pool
  - Uses ReLu activation
  - Uses dropout for normalization



LeNet AlexNet

AlexNet


```
class AlexNet(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nn.LazyConv2d(96, kernel_size=11, stride=4, padding=1),
            nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(256, kernel_size=5, padding=2), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(384, kernel_size=3, padding=1), nn.ReLU(),
            nn.LazyConv2d(256, kernel_size=3, padding=1), nn.ReLU(),
            nn.MaxPool2d(kernel_size=3, stride=2), nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(p=0.5),
            nn.LazyLinear(4096), nn.ReLU(),nn.Dropout(p=0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

AlexNet


```
AlexNet().layer_summary((1, 1, 224, 224))
```

```
Conv2d output shape:        torch.Size([1, 96, 54, 54])
ReLU output shape:   torch.Size([1, 96, 54, 54])
MaxPool2d output shape:     torch.Size([1, 96, 26, 26])
Conv2d output shape:        torch.Size([1, 256, 26, 26])
ReLU output shape:   torch.Size([1, 256, 26, 26])
MaxPool2d output shape:     torch.Size([1, 256, 12, 12])
Conv2d output shape:        torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:        torch.Size([1, 384, 12, 12])
ReLU output shape:   torch.Size([1, 384, 12, 12])
Conv2d output shape:        torch.Size([1, 256, 12, 12])
ReLU output shape:   torch.Size([1, 256, 12, 12])
MaxPool2d output shape:     torch.Size([1, 256, 5, 5])
Flatten output shape:       torch.Size([1, 6400])
Linear output shape:        torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:       torch.Size([1, 4096])
Linear output shape:        torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:       torch.Size([1, 4096])
Linear output shape:        torch.Size([1, 10])
```

(384x3x3 x256)

6400*4096+4096 = 26M Parameters

4096*4096+4096 = 16M Parameters

AlexNet Implementation

extend data size

- In addition to dropout, they used image augmentation, e.g., flipping, clipping, color transformations
- At the time there were no off-the-shelf deep learning packages, they had to write their own cuda-convnet code
- AlexNet was too large for one GPU
- They must partition the neural network into two "groups," and train on two GPUs

Network using Blocks: VGG

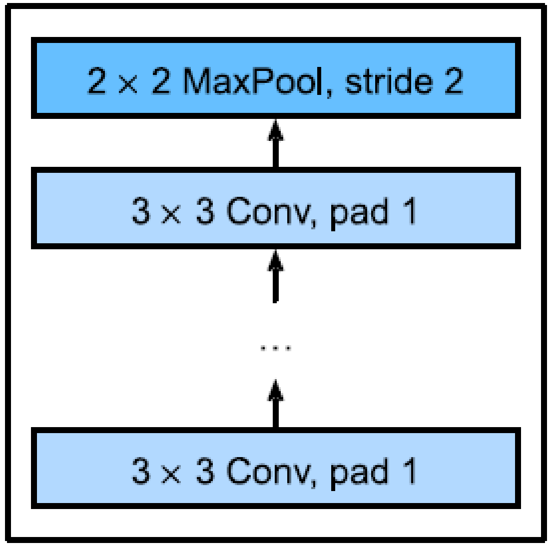
- VGG (Simonyan and Zisserman, 2014) from Visual Geometry Group at Oxford University
- Better to have shallow wide networks, or deep narrow networks?
- Assume the input and output channel dimensions are both c,
  - Number of parameters for 5x5 convolution is $25 * c^2$
  - Number of parameters for 3 layers of 3x3 convolution is $3 * 9 * c^2$
- Both have about the same number of parameters
- But 3 layers of 3x3 convolution performs better

VGG Block

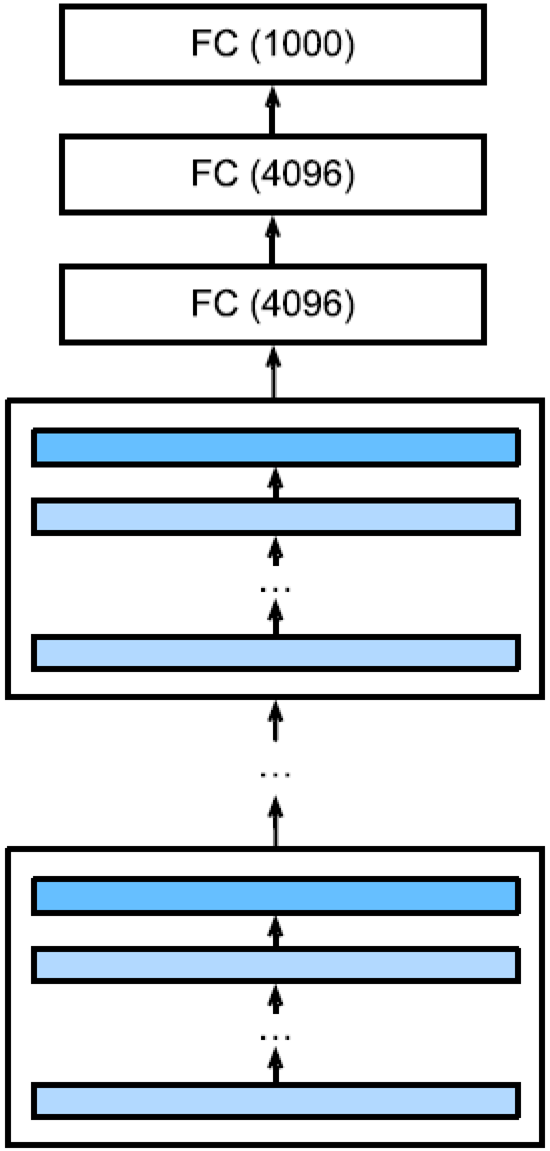
```
def vgg_block(num_convs, out_channels):
    layers = []
    for _ in range(num_convs):
        layers.append(nn.LazyConv2d(out_channels, kernel_size=3, padding=1))
        layers.append(nn.ReLU())
    layers.append(nn.MaxPool2d(kernel_size=2,stride=2))
    return nn.Sequential(*layers)
```

- Multiple convolution layers before down sampling with max pooling

VGG block



VGG



VGG Architecture

- VGG architecture specified by a list of (num_convs, out_channel) pairs, i.e., input to `vgg_block(num_convs, out_channel)`

```
class VGG(d2l.Classifier):
    def __init__(self, arch, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        conv_blks = []
        for (num_convs, out_channels) in arch:
            conv_blks.append(vgg_block(num_convs, out_channels))
        self.net = nn.Sequential(
            *conv_blks, nn.Flatten(),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(4096), nn.ReLU(), nn.Dropout(0.5),
            nn.LazyLinear(num_classes))
        self.net.apply(d2l.init_cnn)
```

VGG-11

- VGG-11 has 8 conv layers and 3 fully connected layers

```
VGG(arch=((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))).layer_summary(
    (1, 1, 224, 224))
```

```
Sequential output shape:     torch.Size([1, 64, 112, 112])
Sequential output shape:     torch.Size([1, 128, 56, 56])
Sequential output shape:     torch.Size([1, 256, 28, 28])
Sequential output shape:     torch.Size([1, 512, 14, 14])
Sequential output shape:     torch.Size([1, 512, 7, 7])
Flatten output shape:        torch.Size([1, 25088])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 4096])
ReLU output shape:   torch.Size([1, 4096])
Dropout output shape:        torch.Size([1, 4096])
Linear output shape:         torch.Size([1, 10])
```

25088*4096=103M
Parameters

VGG Innovations

- Preference for deep and narrow networks
- Blocks of multiple convolutions
- Families of networks

Network in Network (NiN)

- Problems with AlexNet and VGG
  - Huge fully connect layers at the end of the architecture, e.g., VGG-11 has 103M matrix, which takes ~412MB to store in single precision float
  - Not possible to add fully connected layer earlier in the network to increase nonlinearity, because this would destroy spatial structure and use more memory
- NiN solution (Lin et al., 2013)
  - Global average pooling to avoid huge fully connect layers
  - 1x1 convolution to add local non-linearity

NiN

1 x 1 Convolution

- Given input $c * h * w$, 1x1 convolution layer of filter size 1 outputs:
$$1 * h * w$$
- Each output pixel is fully connected to the c channels of the corresponding pixel
- Adds nonlinearity if nonlinear activation function is used

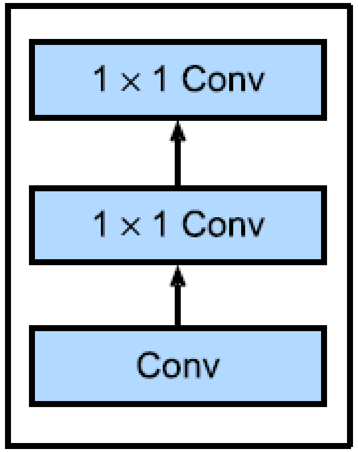
Global Average Pool

- Given input of $c * h * w$, global average pooling outputs:
$$c * 1 * 1$$
- Outputs average over $h * w$ pixels
- c is the number of category classes
- Removes the need for fully connected layers

NiN Block


```
def nin_block(out_channels, kernel_size, strides, padding):
    return nn.Sequential(
        nn.LazyConv2d(out_channels, kernel_size, strides, padding), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU(),
        nn.LazyConv2d(out_channels, kernel_size=1), nn.ReLU())
```

NiN block

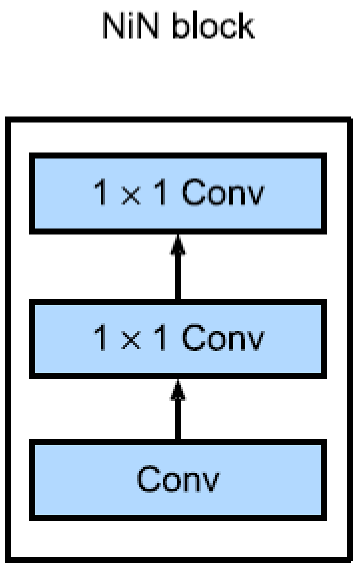


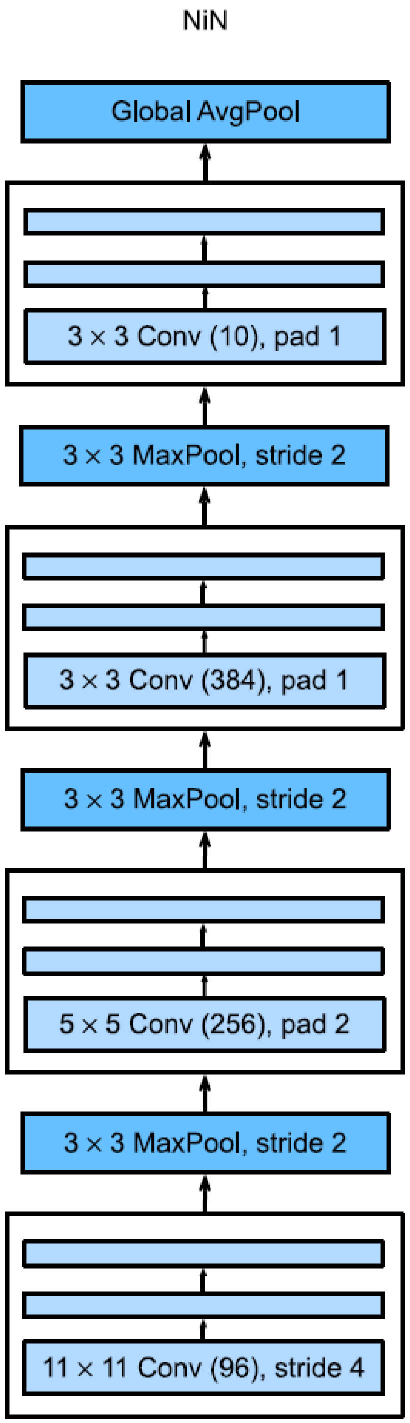
Global Average Pooling: AdaptiveAvePool2d

- PyTorch nn.AdaptiveAvePool2d((1, 1))
- Given input shape $c * h * w$, find corresponding parameters such that the output shape is $c * 1 * 1$

NiN Architecture


```python
class NiN(d2l.Classifier):
    def __init__(self, lr=0.1, num_classes=10):
        super().__init__()
        self.save_hyperparameters()
        self.net = nn.Sequential(
            nin_block(96, kernel_size=11, strides=4, padding=0),
            nn.MaxPool2d(3, stride=2),
            nin_block(256, kernel_size=5, strides=1, padding=2),
            nn.MaxPool2d(3, stride=2),
            nin_block(384, kernel_size=3, strides=1, padding=1),
            nn.MaxPool2d(3, stride=2),
            nn.Dropout(0.5),
            nin_block(num_classes, kernel_size=3, strides=1, padding=1),
            nn.AdaptiveAvgPool2d((1, 1)),
            nn.Flatten())
        self.net.apply(d2l.init_cnn)
```



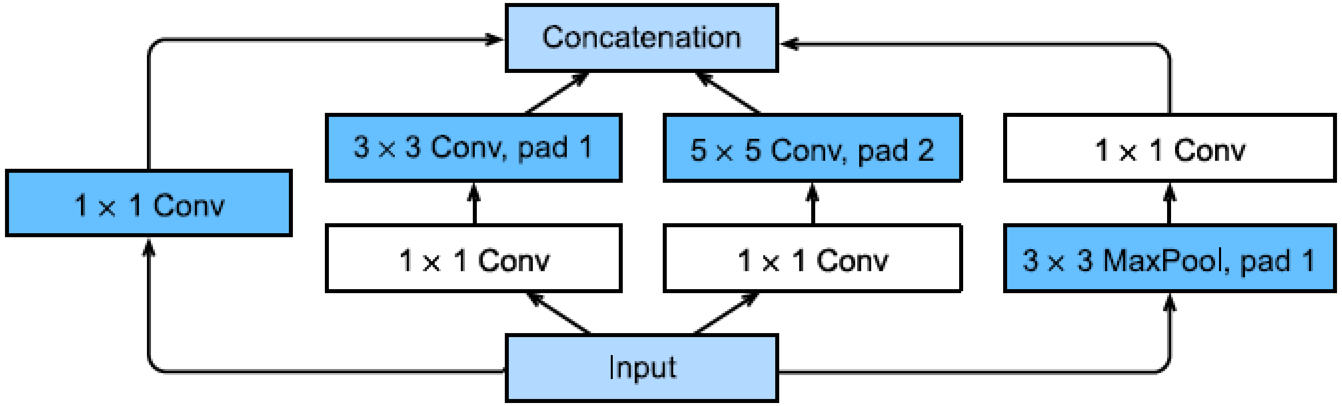


Multi-Branch Networks (GoogLeNet)

- GoogLeNet (Szegedy et al., 2015) won the 2014 ImageNet Competition
- Key contribution: Inception block
  - Instead of trying to figure out the appropriate convolution size, just generate multiple convolution sizes and concatenate the results
- Also, the network is partitioned into three distinct parts:
  - Stem for ingesting the image and extract low level features
  - Body for generating features
  - Head for using the features for specific tasks, such as classification

Inception Block

- Concatenate results of four branches into one
- Each branch outputs the same image size $h * w$
- Convolution size ranges from 1x1 to 3x3 to 5x5. This enables it to extract feature of different sizes.

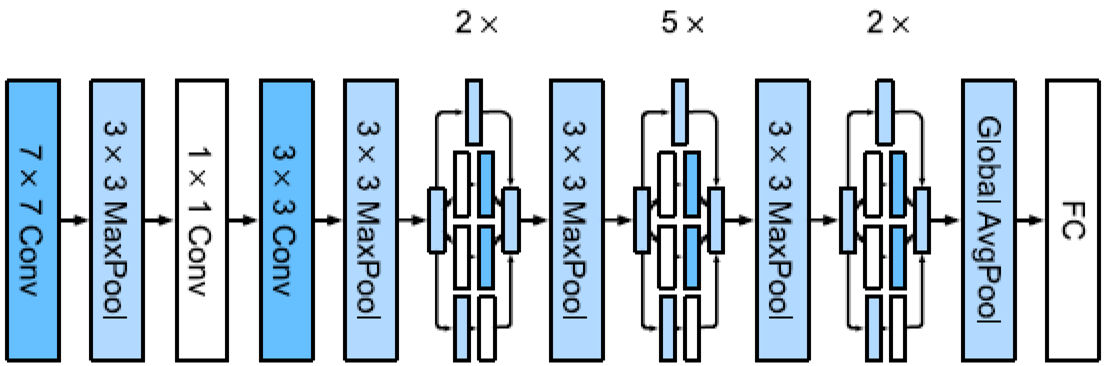


Inception Module


```python
class Inception(nn.Module):
    # c1--c4 are the number of output channels for each branch
    def __init__(self, c1, c2, c3, c4, **kwargs):
        super(Inception, self).__init__(**kwargs)
        # Branch 1
        self.b1_1 = nn.LazyConv2d(c1, kernel_size=1)
        # Branch 2
        self.b2_1 = nn.LazyConv2d(c2[0], kernel_size=1)
        self.b2_2 = nn.LazyConv2d(c2[1], kernel_size=3, padding=1)
        # Branch 3
        self.b3_1 = nn.LazyConv2d(c3[0], kernel_size=1)
        self.b3_2 = nn.LazyConv2d(c3[1], kernel_size=5, padding=2)
        # Branch 4
        self.b4_1 = nn.MaxPool2d(kernel_size=3, stride=1, padding=1)
        self.b4_2 = nn.LazyConv2d(c4, kernel_size=1)

    def forward(self, x):
        b1 = F.relu(self.b1_1(x))
        b2 = F.relu(self.b2_2(F.relu(self.b2_1(x))))
        b3 = F.relu(self.b3_2(F.relu(self.b3_1(x))))
        b4 = F.relu(self.b4_2(self.b4_1(x)))
        return torch.cat((b1, b2, b3, b4), dim=1)
```

GoogLeNet Model



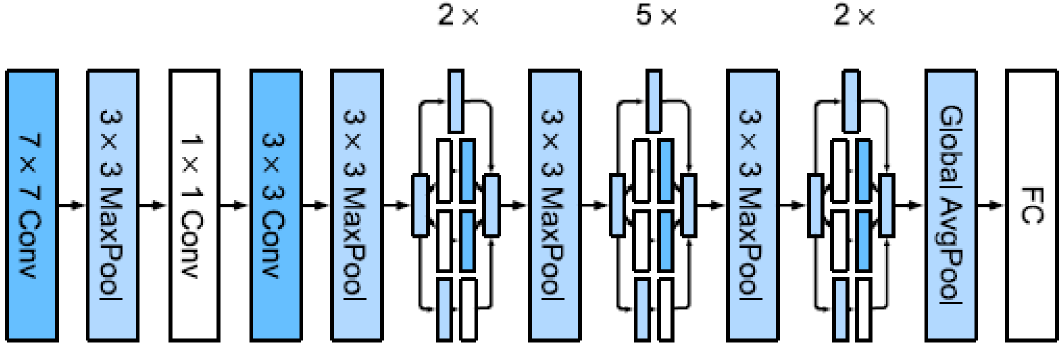
- Stem, outputs 64 features

```
class GoogleNet(d2l.Classifier):
    def b1(self):
        return nn.Sequential(
            nn.LazyConv2d(64, kernel_size=7, stride=2, padding=3),
            nn.ReLU(), nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

- Body module 2, outputs 192 features

```
@d2l.add_to_class(GoogleNet)
def b2(self):
    return nn.Sequential(
        nn.LazyConv2d(64, kernel_size=1), nn.ReLU(),
        nn.LazyConv2d(192, kernel_size=3, padding=1), nn.ReLU(),
        nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet Model



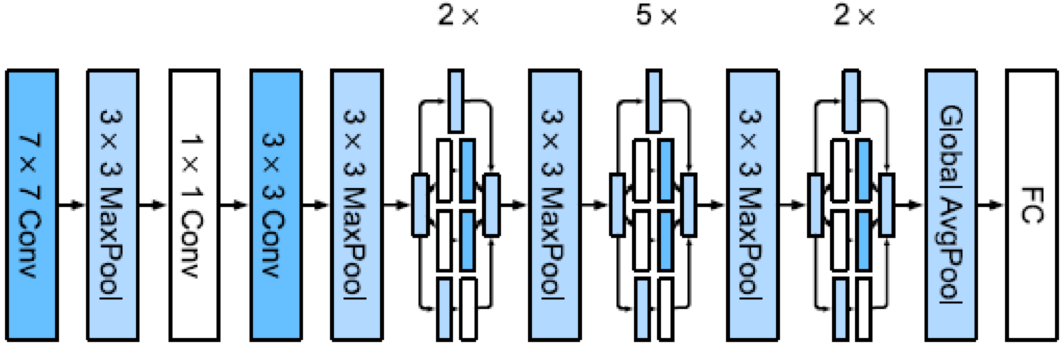
- Body module 3, outputs 480=128+192+96+64 features

```
@d2l.add_to_class(GoogleNet)
def b3(self):
    return nn.Sequential(Inception(64, (96, 128), (16, 32), 32),
                         Inception(128, (128, 192), (32, 96), 64),
                         nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

- Body module 4, outputs 832=256+320+128+128 features

```
@d2l.add_to_class(GoogleNet)
def b4(self):
    return nn.Sequential(Inception(192, (96, 208), (16, 48), 64),
                         Inception(160, (112, 224), (24, 64), 64),
                         Inception(128, (128, 256), (24, 64), 64),
                         Inception(112, (144, 288), (32, 64), 64),
                         Inception(256, (160, 320), (32, 128), 128),
                         nn.MaxPool2d(kernel_size=3, stride=2, padding=1))
```

GoogLeNet Model



- Body module 5, outputs flattened 1024=384+384+128+128

```
@d2l.add_to_class(GoogleNet)
def b5(self):
    return nn.Sequential(Inception(256, (160, 320), (32, 128), 128),
                         Inception(384, (192, 384), (48, 128), 128),
                         nn.AdaptiveAvgPool2d((1,1)), nn.Flatten())
```

- Combine modules and add head with fully connected layer

```
@d2l.add_to_class(GoogleNet)
def __init__(self, lr=0.1, num_classes=10):
    super(GoogleNet, self).__init__()
    self.save_hyperparameters()
    self.net = nn.Sequential(self.b1(), self.b2(), self.b3(), self.b4(),
                             self.b5(), nn.LazyLinear(num_classes))
    self.net.apply(d2l.init_cnn)
```

GoogLeNet

- Cheaper to compute than its predecessors
- Provides higher accuracy
- Considered to be the first truly modern CNN