Convolutional Neural Networks – ImageNet

- Deep models with many layers require large amounts of data in order to significantly outperform traditional methods (e.g., linear and kernel methods).

- Most research up until 2010 relied on tiny datasets.

- In 2009, the ImageNet dataset was released, challenging researchers to learn models from 1 million examples, 1,000 each from 1,000 distinct categories of objects.

    - This scale was unprecedented.
    - The associated competition, dubbed the ImageNet Challenge pushed computer vision and machine learning research forward, challenging researchers to identify which models performed best at a greater scale than academics had previously considered.

Convolutional Neural Networks – GPU Processing

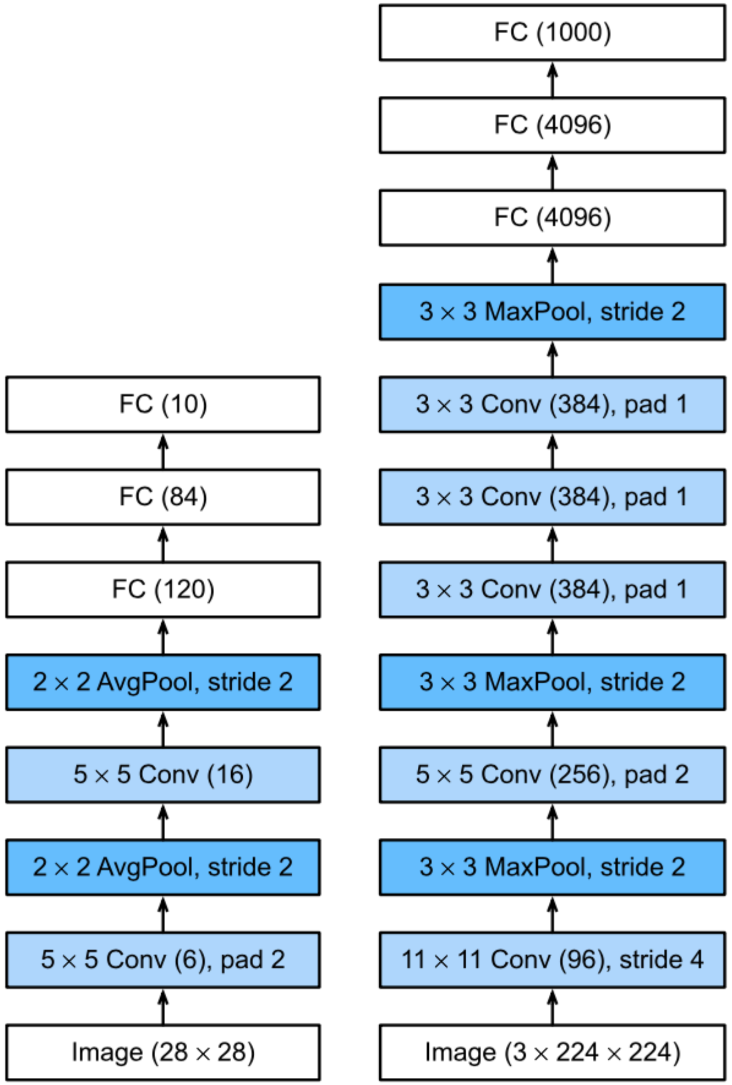
- Training Deep learning models can take hundreds of epochs, and each iteration requires passing data through many layers of computationally-expensive linear algebra operations.
- This is one of the main reasons why in the 90s and early 2000s, simple algorithms based on linear and kernel methods were preferred.
- Graphical processing units (GPUs) proved to be a game changer in make deep learning feasible.
    - Orginally optimized for high throughput 4x4 matrix-vector products for graphics tasks.
    - strikingly similar to what is required to calculate convolutional layers.

- Back to 2012: A major breakthrough came when Alex Krizhevsky and Ilya Sutskever implemented a deep convolutional neural network that could run on GPU hardware.
    - They realized that the computational bottlenecks in CNNs (convolutions and matrix multiplications) are all operations that could be parallelized in hardware.
- Using two NVIDIA GTX 580s with 3GB of memory, they implemented fast convolutions.

AlexNet

- AlexNet was introduced in 2012, named after Alex Krizhevsky, the first author of the breakthrough ImageNet classification paper
    - It won the ImageNet Large Scale Visual Recognition Challenge 2012 by a phenomenally large margin.
- The architectures of AlexNet and LeNet are *very similar*
- There are also significant differences.
    - First, AlexNet is much deeper than the comparatively small LeNet5.
    - Second, AlexNet used the ReLU instead of the sigmoid

AlexNet



- LeNet (left) and AlexNet (right)

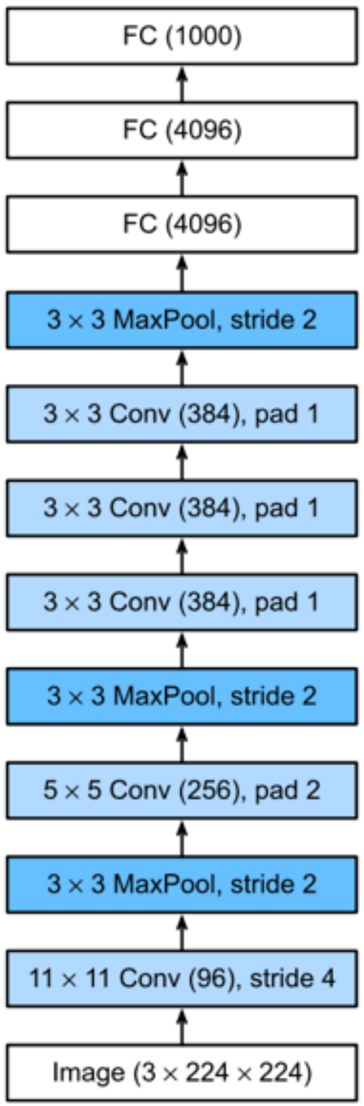
Concise Implementation of LeNet

- Goal: use high-level APIs of PyTorch for implementing LeNet for classification.

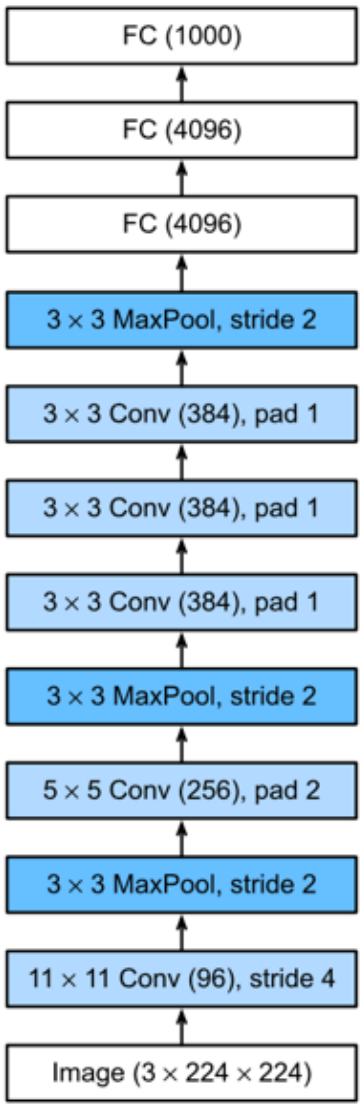
In [3]:
```
# Read training and test data
batch_size = 256
train_iter, test_iter = mu.load_data_fashion_mnist(batch_size, resize=224)
# type(train_iter)
```

Defining the Model

In [4]:
```python
class AlexNet(torch.nn.Module):
    def __init__(self, num_outputs):
        super(AlexNet, self).__init__()
        self.conv1 = nn.Conv2d(1, 96, 11, stride=4)
        self.rl1 = nn.ReLU()
        self.max1 = nn.MaxPool2d(3, stride=2)
        self.conv2 = nn.Conv2d(96, 256, 5, stride=1, padding=2)
        self.rl2 = nn.ReLU()
        self.max2 = nn.MaxPool2d(3, stride=2)

        self.conv3 = nn.Conv2d(256, 384, 3, 1, padding=1)
        self.rl3 = nn.ReLU()
        self.conv4 = nn.Conv2d(384, 384, 3, 1, padding=1)
        self.rl4 = nn.ReLU()
        self.conv5 = nn.Conv2d(384, 256, 3, 1, padding=1)
        self.rl5 = nn.ReLU()
        self.max3 = nn.MaxPool2d(3, stride=2)

        self.fl = nn.Flatten()
        self.linear1 = nn.Linear(6400, 4096)
        self.rl6 = nn.ReLU()
        self.linear2 = nn.Linear(4096, 4096)
        self.rl7 = nn.ReLU()
        self.linear3 = nn.Linear(4096, num_outputs)
```



```python
def forward(self, x):
    out = self.conv1(x)
    out = self.rl1(out)
    out = self.max1(out)
    out = self.conv2(out)
    out = self.rl2(out)
    out = self.max2(out)
    out = self.conv3(out)
    out = self.rl3(out)
    out = self.conv4(out)
    out = self.rl4(out)
    out = self.conv5(out)
    out = self.rl5(out)
    out = self.max3(out)

    out = self.fl(out)
    out = self.linear1(out)
    out = self.rl6(out)
    out = self.linear2(out)
    out = self.rl7(out)
    out = self.linear3(out)

    return out
```



In [5]:

```
def init_weights(m):
    if type(m) == nn.Linear or type(m) == nn.Conv2d: # by checking type we can i
nit different layers in different ways
        torch.nn.init.xavier_uniform_(m.weight)

num_outputs = 1000
net = AlexNet(num_outputs)

net.apply(init_weights);
print(net)
```

Loss and Optimization Algorithm

- As in Softmax Regression

In [6]:

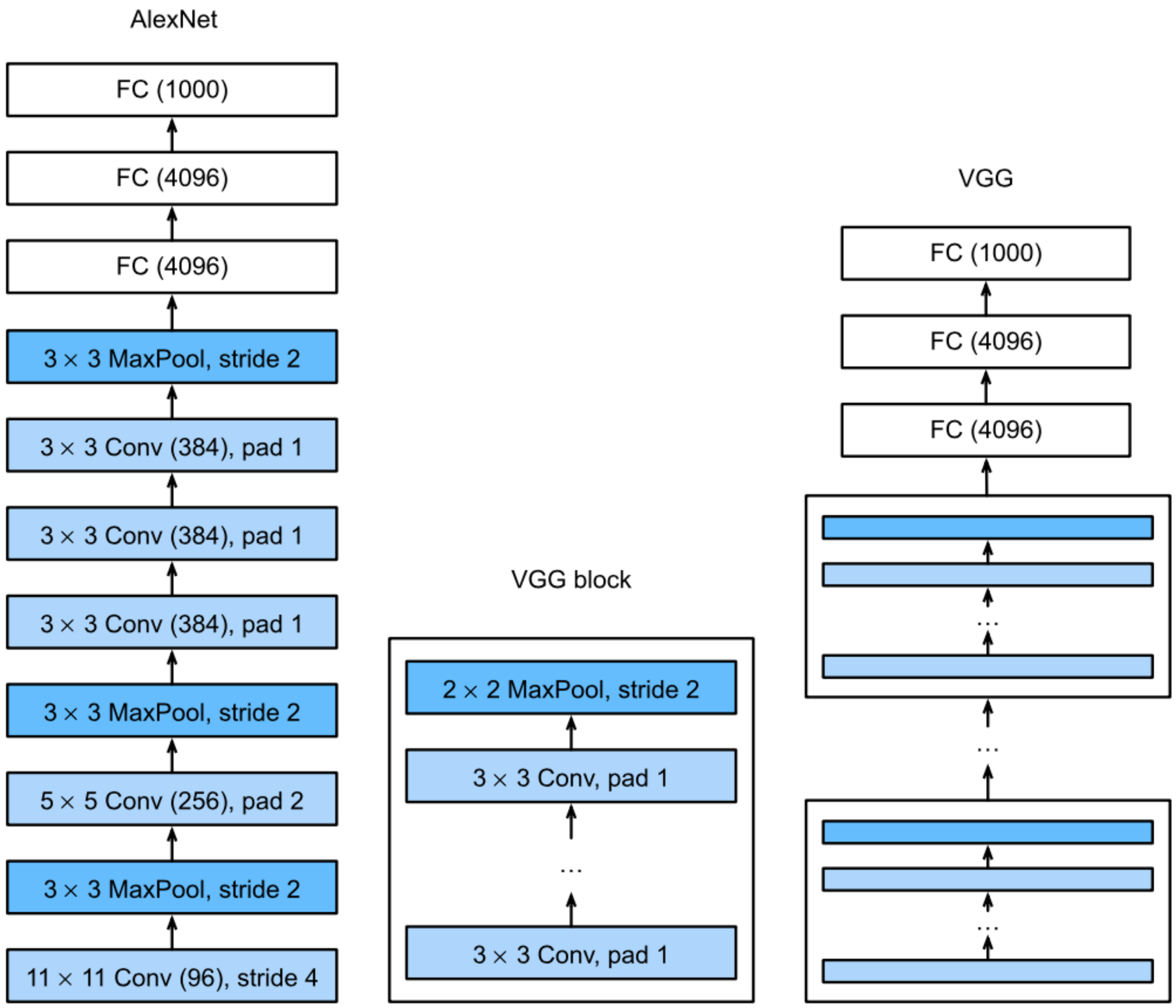
```
loss = nn.CrossEntropyLoss()
lr = 0.01
optimizer = torch.optim.SGD(model.parameters(), lr=lr)
```

Training

- Use `my_utils.train_ch3` as in Softmax Regression

In [ ]:
```
num_epochs = 10
mu.train_ch3(model, train_iter, test_iter, loss, num_epochs, optimizer)
```

VGG



AlexNet
FC (1000)
FC (4096)
FC (4096)
3 × 3 MaxPool, stride 2
3 × 3 Conv (384), pad 1
3 × 3 Conv (384), pad 1
3 × 3 Conv (384), pad 1
3 × 3 MaxPool, stride 2
5 × 5 Conv (256), pad 2
3 × 3 MaxPool, stride 2
11 × 11 Conv (96), stride 4
VGG block
2 × 2 MaxPool, stride 2
3 × 3 Conv, pad 1
...
3 × 3 Conv, pad 1
VGG
FC (1000)
FC (4096)
FC (4096)
...

VGG

- Invented by the Visual Geometry Group in Oxford University
- The original VGG network had 5 convolutional blocks (VGG blocks)
    - The VGG block is the main building of the VGG network.
    - The first two have one convolutional layer each.
    - The latter three contain two convolutional layers each.
    - The first block has 64 output channels and each subsequent block doubles the number of output channels, until that number reaches 512.
- Since this network uses 8 convolutional layers and 3 fully-connected layers, it is called VGG-11.
    - The deepest network trained has 19 layers (called VGG-19).

VGG block

In [63]:

```
class VGG_block(nn.Module):
    def __init__(self, num_convs, input_channels, output_channels):
        super(VGG_block, self).__init__()
        self.num_convs = num_convs
        for i in range(num_convs):
            self.add_module('conv{0}'.format(i), nn.Conv2d(input_channels,
                                                           output_channels, kern
el_size=3, padding=1))
            input_channels = output_channels
            self.add_module('relu{0}'.format(i), nn.ReLU())

        self.max_pool = nn.MaxPool2d(kernel_size=2,stride=2)

    def forward(self, x):
        out = x
        for i in range(self.num_convs):
            out = self._modules['conv{0}'.format(i)](out)
            out = self._modules['relu{0}'.format(i)](out)

        out = self.max_pool(out)
        return out
```

VGG-11

In [68]:
```
class VGG(nn.Module):
    def __init__(self, conv_arch):
        super(VGG, self).__init__()
        in_channels = 1
        self.conv_arch = conv_arch
        for i, (num_convs, out_channels) in enumerate(conv_arch):
            self.add_module('vgg_block{0}'.format(i), VGG_block(num_convs, in_ch
annels, out_channels))
            in_channels = out_channels

        self.last = nn.Sequential(nn.Flatten(), nn.Linear(out_channels*7*7, 4096
),
                                  nn.ReLU(), nn.Dropout(0.5), nn.Linear(4096, 40
96),
                                  nn.ReLU(), nn.Dropout(0.5), nn.Linear(4096, 10
))

    def forward(self, x):
        out = x
        for i in range(len(self.conv_arch)):
            out = self._modules['vgg_block{0}'.format(i)](out)
        out = self.last(out)
        return out
```

In [11]:

```
conv_arch = ((1, 64), (1, 128), (2, 256), (2, 512), (2, 512))
net = VGG(conv_arch)

a = torch.rand(16, 1, 224, 224)
print(a.size())
out = net(a)
print(out.size())
```

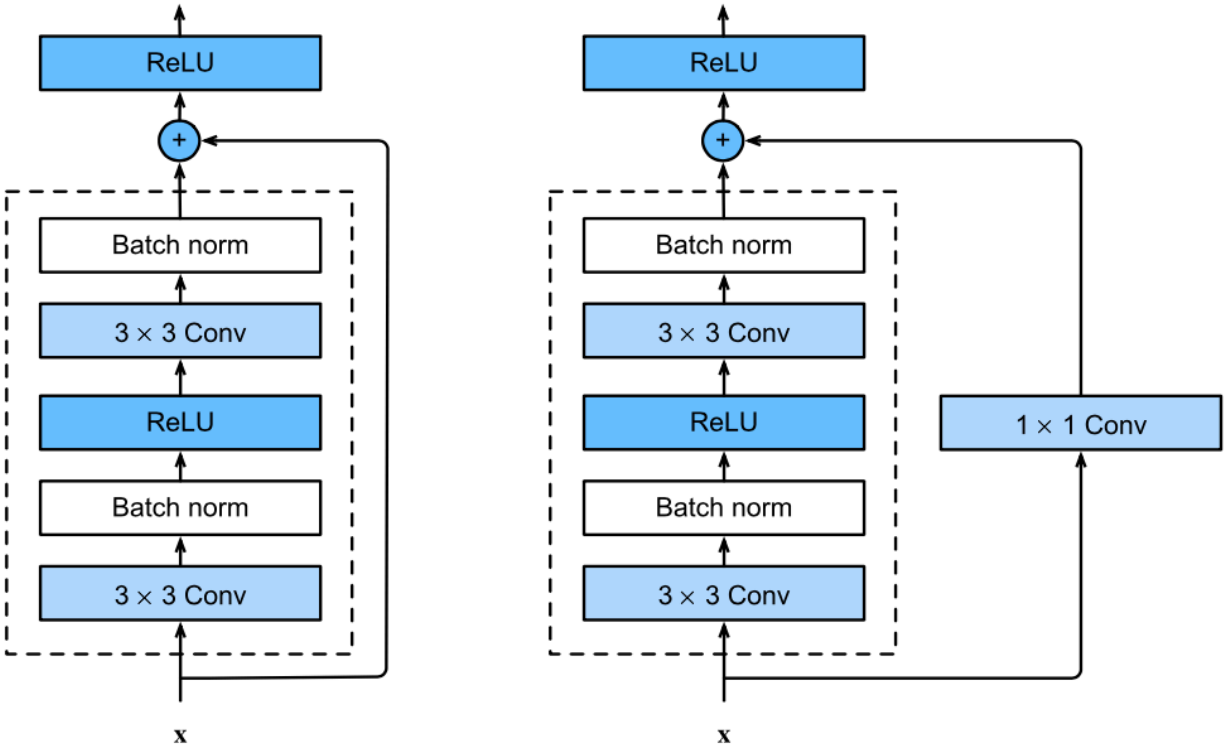
```
torch.Size([16, 1, 224, 224])
torch.Size([16, 10])
torch.Size([16, 10])
```

ResNet

- Prior to ResNet, it was hard to train very deep networks
  - Depth facilitates learning very powerful networks
- Kaiming He solved this in his paper: *Deep Residual Learning for Image Recognition*, CVPR 2016
  - He showed how how to train networks with 152 convolutional layers!
  - The most popular computer vision paper ever written!
- Main idea: add the input feature x to the output of a conv. layer F
  - $y = F(x) + x$
  - This is called **skip connection**
- The advantage is that there's a direct path for propagating gradients during back-prop.

ResNet block

- ResNet is similar to VGG but uses a ResNet block which has a skip connection.
  - The right block is used when the number of channels in the input is not the same as the number of channels in the output
    - An 1×1 conv. layer is used to make them equal.



- Ignore BatchNorm for now.

ResNet-18 – Overview of architecture

- We will consider here the implementation of the smallest ResNet, called ResNet-18.
- ResNet-18 (and all other ResNets) has 1 stem and 4 macro-modules followed by 1 linear (FC) layer.
- Stem = simple processing unit with 1 standard conv. layer (with stride 2) and 1 max pool (with stride 2).
    - Reduces input resolution from 224 to 112 and then to 56.
        - Nothing special so far.
- Each macro-module processes features in a different resolution.
    - Resolutions used are: 56, 28, 14, 7
- The macro-modules are composed of the so-called **ResNet blocks**
    - In ResNet-18, there are 2 ResNet blocks per macro-module
- Each ResNet (Residual) block consists of 2 convolutions with skip connections
    - Actually, the ResNet block is what He proposed in his paper
- In total $1 + 4 \times 2 \times 2 + 1 = 18$ layers
    - That's why it's called ResNet-18
    - Different blocks per macro-module results in different models like the 152-layer ResNet-152

Batch Normalization

- Batch Normalization (BN) is a popular and effective technique that consistently accelerates the convergence of deep nets.
- Together with residual blocks, BN made it possible to routinely train networks with over 100 layers.

Batch Normalization

- Batch normalization is applied to individual layers and works as follows:
    - In each training iteration (i.e. for each mini-batch), we normalize each channel of the input feature tensor seperately by subtracting their mean and dividing by their standard deviation (std).
        - Mean and std are estimated based on the statistics of the current minibatch.
    - Next, we apply a scaling coefficient and a scaling offset.

Batch Normalization

- Denoting a particular minibatch by $\mathcal{B}$, we firstly calculate $\hat{\mu}_\mathcal{B}$ and $\hat{\sigma}_\mathcal{B}$ as follows:

$$\hat{\mu}_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} \mathbf{x} \text{ and } \hat{\sigma}^2_\mathcal{B} \leftarrow \frac{1}{|\mathcal{B}|} \sum_{\mathbf{x} \in \mathcal{B}} (\mathbf{x} - \mu_\mathcal{B})^2 + \epsilon$$

  - There's a different mean and std per channel.
  - A small constant $\epsilon > 0$ is added to ensure that we never attempt division by zero.

- BN transforms the activations at a given feature tensor $\mathbf{x}$ according to the following expression:

$$\mathrm{BN}(\mathbf{x}) = \gamma \odot \frac{\mathbf{x} - \hat{\mu}_\mathcal{B}}{\hat{\sigma}_\mathcal{B}} + \beta$$

  - The above formula is applied *channelwise* i.e. there's a different mean and std per channel.

- After normalization, the resulting minibatch of activations has zero mean and unit variance. Because this is an arbitrary choice, we commonly include channel-wise scaling coefficients γ and offsets β.

  - These are learnable parameters learnt via back-propagation!