Applied Deep Learning with Tensorflow and PyTorch, Chapter 6

Conv layer vs Dense layer

- Conv Layer vs Dense layer Conv layers require many **fewer parameters** than fully-connected or dense layers.
- CNNs apply a **local search** for features while a dense layer conducts a global search for features.
- CNNs are **equivariance to translation** means translation invariance in images implies that all patches of an image will be treated in the same manner.

Cross-correlation

```
import torch
from torch import nn

def corr2d(X, K):
    h, w = K.shape
    Y = torch.zeros((X.shape[0] - h + 1, X.shape[1] - w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            Y[i, j] = (X[i:i + h, j:j + w] * K).sum()
    return Y
```

Convolutional Layer in PyTorch

```
class Conv2D(nn.Module):
    def __init__(self, kernel_size):
        super().__init__()
        self.weight = nn.Parameter(torch.rand(kernel_size))
        self.bias = nn.Parameter(torch.zeros(1))

    def forward(self, x):
        return corr2d(x, self.weight) + self.bias
```

Convolutional Layer in TF

```
class Conv2D(tf.keras.layers.Layer):
    def __init__(self):
        super().__init__()

    def build(self, kernel_size):
        initializer = tf.random_normal_initializer()
        self.weight = self.add_weight(name='w', shape=kernel_size,
                                      initializer=initializer)
        self.bias = self.add_weight(name='b', shape=(1,),
                                    initializer=initializer)

    def call(self, inputs):
        return corr2d(inputs, self.weight) + self.bias
```

The core computation of a two-dimensional convolutional layer is a two-dimensional cross-correlation operation. In its simplest form, this performs a cross-correlation operation on the two-dimensional input data and the kernel, and then adds a bias.

Padding

Padding can increase the height and width of the output. This is often used to give the output the same height and width as the input.

Padding in PyTorch

```
import torch
from torch import nn

def comp_conv2d(conv2d, X):
    X = X.reshape((1, 1) + X.shape)
    Y = conv2d(X)
    return Y.reshape(Y.shape[2:])

conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1)
X = torch.rand(size=(8, 8))
comp_conv2d(conv2d, X).shape
```

Padding in TF

```
import tensorflow as tf

def comp_conv2d(conv2d, X):
    X = tf.reshape(X, (1,) + X.shape + (1,))
    Y = conv2d(X)
    return tf.reshape(Y, Y.shape[1:3])

conv2d = tf.keras.layers.Conv2D(1, kernel_size=3, padding='same')
X = tf.random.uniform(shape=(8, 8))
comp_conv2d(conv2d, X).shape
```

Stride

The stride can reduce the resolution of the output, for example reducing the height and width of the output to only $1/n$ of the height and width of the input (n is an integer greater than 1).

Stide in PyTorch

```
conv2d = nn.Conv2d(1, 1, kernel_size=3, padding=1, stride=2)
comp_conv2d(conv2d, X).shape
```

Stide in TF

```
conv2d = tf.keras.layers.Conv2D(1, kernel_size=3, padding='same', strides=2)
comp_conv2d(conv2d, X).shape
```

Padding and stride can be used to adjust the dimensionality of the data effectively.

Pooling

Pooling in PyTorch

```
import torch
from torch import nn

def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = torch.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j] = X[i:i + p_h, j:j + p_w].max()
            elif mode == 'avg':
                Y[i, j] = X[i:i + p_h, j:j + p_w].mean()
    return Y
```

Pooling in TF

```
import tensorflow as tf

def pool2d(X, pool_size, mode='max'):
    p_h, p_w = pool_size
    Y = tf.Variable(tf.zeros((X.shape[0] - p_h + 1, X.shape[1] - p_w + 1)))
    for i in range(Y.shape[0]):
        for j in range(Y.shape[1]):
            if mode == 'max':
                Y[i, j].assign(tf.reduce_max(X[i:i + p_h, j:j + p_w]))
            elif mode == 'avg':
                Y[i, j].assign(tf.reduce_mean(X[i:i + p_h, j:j + p_w]))
    return Y
```

We can specify the padding and stride for the pooling layer.

```
pool2d = nn.MaxPool2d(3, padding=1, stride=2)
pool2d(X)
```

The pooling layer's number of output channels is the same as the number of input channels.

Multiple Input/Output Channel

```
import torch

def corr2d_multi_in(X, K):
    return sum(corr2d(x, k) for x, k in zip(X, K))

def corr2d_multi_in_out(X, K):
    return torch.stack([corr2d_multi_in(X, k) for k in K], 0)
```