```
import torch
from d2l import torch as d2l
from torch import nn
from torch.nn import functional as F
```

2.0 Convolutional Layer

The earliest breakthrough in pattern recognition comes from LeCun's Digit Recognition model [1], which utilized Convolutional Layer to learn features. The model's implementation is based on Fukushima's Neocognitron architecture, which in turn is based on visual cortex of the human body. This mimicry shows that much can be learned from nature.

At present (writing this as of Sep. 1st, 2025), a lot of computer vision model still utilize Convolutional layer though they are being displaced by Transformers. They are still the basis for most computer vision tasks due to their ability to extract local features and patterns efficiently compared to the approaches in the previous notebook. When we start to stack layers together to add complexity and also train on bigger images with more channels, the time and space requirements grow in order of magnitude rendering our earlier model ill-suited for the task. A Convolutional layer is much more efficient, it utilizes **weight-sharing** or a **kernel** due to **translation invariance** and **locality**.

2.1. Convolutional Math

The reader may heard of this from ODE (or taught it, if you're reading this Professor Lin!), which has the following form

$$\int_{-\infty}^{\infty} f(x)g(t-x)dt$$

What convolution does is that it slides one function over the other, by adding the product of where they overlap, to produce a final function. Any who, integrals are just Riemann sum, we can get away with a finite sum rather than one that goes to infinity (NOT SUBSTANTIATED, TRUST ME AT YOUR OWN PERIL!!!).

$$\sum_{i=-\infty}^{\infty} f(x)g(t-x)dt$$

In the context of a Classification problem, we want our Convolution function to iterate over all part of an image and extract the important underlying features. We would like to think of these features as being like an image's edge (the outline of a character you want to identify), an ear, a mouth, etc. Now, these features can be anywhere (REALLY IMPORTANT TO REMEMBER THIS STATEMENT), which means we can treat the **feature map** or the **kernel** (let's call it **kernel**) to detect a feature all over an image. What convolution is saying is:

> "Hey, can you like try applying a square detector on every part of an image and see where's there's an ear? Just divide the image into sections to check and find sections where there may be an ear."

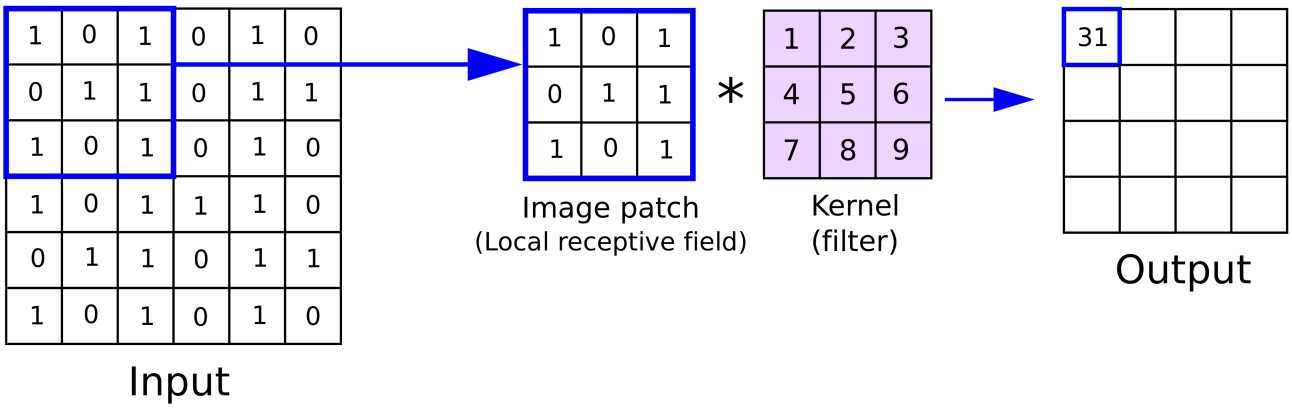

Image courtesy of Anh H. Reynolds

Pretty simple right? Okay, let $O(i,j)$ be the output of a Convolution, $I(i,j)$ be the function that grabs pixel value (we will clarify this later), and $K(i,j)$ be the function that grabs from a kernel. Let m and n be the set of things in a kernel.

$$O(i,j) = \sum\sum I(m,n)K(i-m,j-m)$$

Now according to the GoodFellow, you can flip it relatively to the kernel. It's commutative! (Yeah prove that Lin). [2].

$$O(i,j) = \sum_m \sum_n I(i-m, j-n)K(m,n)$$

But we can just follow the Cross-correlation version according to GoodFellow [2].

$$O(i,j) = \sum_m \sum_n I(i+m, j+n)K(m,n)$$

Every input and output could have multiple channels, the input could be an RGB image and thus have three channels, or it could be an output of another layer that has like 64 channels. We would also want to output a 4D tensors that would have multiple channels for further analysis for the model. We denote l to be the kernel index for the corresponding input channels and k be the corresponding index for the output channel.

$$O_{k,i,j} = \sum_l \sum_m \sum_n I_{l,i+m,j+n} K_{k,l,m,n}$$

2.2 Striding and Padding It seems a bit costly to go over all the pixels of an image to find an Ear, after all, the ear is usually in one or two spots of our image. We could get by through applying a kernel every *sth* pixel through striding.

$$O_{s,k,i,j} = \sum_l \sum_m \sum_n I_{l,i\times s+m,j\times s+n} K_{k,l,m,n}$$

The Convolution function **down samples** a matrix, meaning it would have its dimension reduced. We would want to avoid that if we are to make our model complex, by adding more layers on top. If we don't have a way to mitigate aggressive down sampling, we would end up with a pretty small matrix, whose content cannot be used for further analysis by the model. To prevent down sampling, we would implement **padding**, or specifically **zero-padding** by adding zeroes all around the matrix so that we retain some or all of the dimensions.

2.3 Implementation

With all the prerequisites for a suitable Convolution Layer explained, we will now implement the **Module** for it. Our naive implementation of the function above would not be feasible for our architecture, since we would have through work through five layers of loops. This implementation could make both our time and space complexity grow exponentially.

Introducing **im2col**! Through some clever manipulation, we could resize our inputs into patches and perform a matrix-multiplication of each patches with the kernel.

```
class Conv2D(d2l.Module):
    def __init__(self, in_channels, out_channels, kernel_size,
                 padding=0, stride=1, lr=0.01, bias=True):
        super().__init__()
        self.in_channels = in_channels
        self.out_channels = out_channels
        self.kernel_size = kernel_size
        self.stride = stride
        self.padding = padding
        self.bias = bias
        self.w = nn.Parameter(torch.normal(0, 0.01, (out_channels, in_channels, kernel_size,
kernel_size)))
        self.b = nn.Parameter(torch.zeros(out_channels)) if bias else None

    def forward(self, X):
        # create dimension
        p = self.padding
        s = self.stride
        k = self.kernel_size

        # # if padding gt 1
        # if p > 0:
        #     batch_size, channels, height, width = X.shape
        #     padded_tensor = torch.zeros((batch_size, channels, height + 2 * p, width + 2 * p),
```

```
        #                                 dtype=X.dtype, device=X.device)
        #     padded_tensor[:, :, p:height + p, p:width + p] = X
        #     X = padded_tensor

        ## NAIVE IMPLEMENTATION, SUPER SLOW!!
        # # Output size calculation
        # batch_size, channels, input_height, input_width = X.shape
        # output_height = (input_height - k) // s + 1
        # output_width = (input_width - k) // s + 1

        # # Create the result tensor
        # result = torch.zeros((batch_size, self.out_channels, output_height, output_width),
        #                      dtype=X.dtype, device=X.device)

        # # Manual loop-based convolution
        # for b in range(batch_size):
        #     for i in range(output_height):
        #         for j in range(output_width):
        #             for out_c in range(self.out_channels):
        #                 # Extract the slice of the input tensor
        #                 input_slice = X[b, :, i * s : i * s + k, j * s : j * s + k]
        #                 # Grab the corresponding kernel
        #                 kernel = self.w[out_c]
        #                 # Perform element-wise multiplication and sum
        #                 result[b, out_c, i, j] = (input_slice * kernel).sum()
        #                 if self.b is not None:
        #                     result[b, out_c, i, j] += self.b[out_c]

        # return result

        # unfold X into flattened_kernel_size * patches
        unfolded_X = F.unfold(X, kernel_size=k, stride=s, padding=p)

        # unfold weight into out_channel * flattened_kernel_size
        unfolded_weight = self.w.view(self.out_channels, -1)

        # Perform matrix multiplication
        output_matrix = unfolded_weight @ unfolded_X

        # Calculate output dimensions
        batch_size, _, input_height, input_width = X.shape
        output_height = (input_height - k + 2 * p) // s + 1
        output_width = (input_width - k + 2 * p) // s + 1

        # Reshape the output matrix to the correct tensor shape
        output_tensor = output_matrix.view(batch_size, self.out_channels, output_height, output_width)

        return output_tensor + self.b[None, :, None, None] if self.bias else 0
```

However, we tend to stack Convolution layers and blocks of these stacked Convolutions together to make our architecture more deep and expressive. It would not be fruitful to discuss our convolution function in just the context of images but **feature maps**. As an image progresses into deeper and deeper layer, it transforms into these feature maps due to our convolutional filter or **kernel*. Each feature map represents a learned characteristic of our original images, such as edges, textures, patterns, etc.

These features map are what help our network model to decide if it's actually a cat or a dog (or neither! Remember!). Further operations such as combining them, or taking the max/average of (for translation invariance) may occur. These aggregate operations will be exemplified in our next notebook where we will construct a whole ResNet architecture.

2.4 Pooling

In order to stay true to the idea of **translation invariance**, where small changes to the image doesn't affect the overall outcome, a pooling function is needed [2]. We will use **max pooling** that will project a rectangular window throughout the input to find the maximum value. That maximum value can inform the model if a representation appears within a certain part of the input, thus aiding our model in identification and classification.

We will now implement a module for this Pooling function. It will be attached after a **Convolution layer** and a **non-linearity layer** (to be discussed).

```
class MaxPool2d(d2l.Module):
    def __init__(self, kernel_size, stride=None, padding=0):
        super().__init__()
        self.kernel_size = kernel_size
        self.stride = stride if stride is not None else kernel_size
        self.padding = padding

    def forward(self, X):
        # Unfold the tensor into patches
        unfolded_X = F.unfold(X, kernel_size=self.kernel_size, padding=self.padding, stride=self.stride)

        batch_size, _, L = unfolded_X.shape
        channels = X.shape[1]

        # unfold x furthers
        unfolded_X = unfolded_X.view(
            batch_size, channels, self.kernel_size * self.kernel_size, L
        )

        # Take max over kernel dimension (not channels)
        pooled_X = unfolded_X.max(dim=2)[0]  # shape: (batch_size, channels, L)

        # Get the original dimensions
        height, width = X.shape[2], X.shape[3]

        # Correctly calculate output height and width using original dimensions.
        output_height = (height + 2 * self.padding - self.kernel_size) // self.stride + 1
        output_width = (width + 2 * self.padding - self.kernel_size) // self.stride + 1

        # Reshape the pooled output back to the correct 4D tensor shape.
        output = pooled_X.view(batch_size, channels, output_height, output_width)

        return output
```

The final step of our architecture will require a global average pooling function that needs to reduce the whole feature map into a single scalar. By average across all of our feature map, it helps make our model adhere to translation invariance.

```
class GlobalAvgPool2d(d2l.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return X.mean(dim=[2, 3], keepdim=True)
```

2.5 Noise and Non-linearity

The introduction of non-linearity makes our model more expressive, allowing it to learn non-linear relationships in our data. Without them, they would be able to learn linear data. A widely-used and modern non-linear function is **ReLU** which is defined as follows

$$f(x) = max(0, x)$$

By this definition, it means that any input that is negative would be set to 0, otherwise, it would be positive. Krizhevsky showed that it's faster than other saturating methods like tanh [3] and that a faster converging method like ReLU translates into better Model performance.

```
class ReLU(d2l.Module):
    def __init__(self):
        super().__init__()

    def forward(self, X):
        return max(torch.tensor(0.0), X)
```

With great power, comes great responsibility. The responsibility to reduce overfitting through the injection of noise and randomness, which is what **Dropout** does. We inject randomness by randomly shutting off neuron to see if other neuron plays a greater role in representation, and thus a potential to reduce overfitting and actually generalize to features that matters. Such an activation method helps reduce "complex co-adaptations of neurons", which means it could prevent a neuron dependency on other neuron [3].

Dropout works by shutting off a neuron with a probability of 0.5, then rescaling the activated neurons to ensure consistent output. It is implemented as below

```
class Dropout(d2l.Module):
    def __init__(self, p=0.5):
        super().__init__()
        self.p = p

    def forward(self, X):
        # During inference, just return X unchanged
        if not self.training:
            return X

        # Create a mask with the same shape as X
        mask = (torch.rand(X.shape, device=X.device) > self.p).float()

        # Apply the mask and scale to maintain the expected value
        return X * mask / (1 - self.p)
```

We have to differentiate between training mode and validation mode since we do not want our neuron to turn off randomly while using it.

2.6 Conclusion

There are other things that are intentionally omitted from this notebook since they are not required for our ResNet architecture. Although we have finished constructing all the building blocks needed for our model, we did not discuss the rationale for them. I suggest to the readers to consult `D2L.ai` or other resources to dive further into these design choices.

References

- [1] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," Proc. IEEE, vol. 86, no. 11, pp. 2278–2324, Nov. 1998, doi: 10.1109/5.726791.
- [2] I. Goodfellow, Y. Bengio, and A. Courville, "Convolutional Networks," in Deep Learning. Cambridge, MA, USA: MIT Press, 2016, pp. 321-360.
- [3] Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet Classification with Deep Convolutional Neural Networks. In Advances in Neural Information Processing Systems 25 (NIPS 2012).