15_Classifying_Images_With_Deep_Convolutional_Neural_Networks

June 29, 2018

1 Classifying Images with Deep Convolutional Neural Networks

In the previous chapter, we looked in depth at different aspects of the TensorFlow API, became familiar with tensors, naming variables, and operators, and learned how to work with variable scopes. In this chapter, we will now learn about **Convolutional Neural Networks (CNNs)**, and how we can implement CNNs in TensorFlow. We will also take an interesting journey in this chapter and we apply this type of deep neural network architecture to image classification.

So we will start by discussing the basic building blocks of CNNs, using a bottom-up approach. Then we will take a deeper dive into the CNN architecture and how to implement deep CNNs in TensorFlow. Along the way we will be covering the following topics:

- Understanding convolution operations in one or two dimensions
- Learning about the building blocks of CNN architectures
- Implementing deep convolutional neural networks in TensorFlow

2 Building blocks of convolutional neural networks

Convolutional neural networks, or CNNs, are a family of models that were inspired by how the visual cortex of human brain works when recognizing objects.

The development of CNNs goes back to the 1990's, when Yann LeCun and his colleagues proposed a novel neural network architecture for classifying handwritten digits from images.

Due to the outstanding performance of CNNs for image classification tasks, they have gained a lot of attention and this led to tremendous improvements in machine learning and computer vision applications.

In the following sections, we next see how CNNs are used as feature extraction engines, and then we will delve into the theoretical definition of convolution and computing convolution in one and two dimensions.

2.1 Understanding CNNs and learning feature hierarchies

Successfully extrating **salient (relevant) features** is key to the performance of any machine learning algorithm, of course, and traditional machine learning models rely on input features that may come from a domain expert, or are based on computational feature extraction techniques. Neural networks are able to automatically learn the features from raw data that are most useful for a

particular task. For this reason, it is common to consider a neural network as a feature extraction engine: the early layers (those right after the input layer) extract **low-level features**.

Multilayer neural networks, and in particular, deep convolutional neural networks, construct a so-called **feature hierarchy** by combining the low-level features in a layer-wise fashion to form high-level features. For example, if we are dealing with images, then low-level features, such as edges and blobs, are extracted from the earlier layers, which are combined together to form high-level features - as object shapes like a building, a car, or a dog.

As you can see in the following image, a CNN computes **feature maps** from an input image, where each element comes from a local patch of pixel in the input image:

This local patch of pixels is referred to as the **local receptive field**. CNNs will usually perform very well for image-related tasks, and that is largely due to two important ideas:

- **Sparse-connectivity**: A single element in the feature map is connected to only a small patch of pixels. (This is very different from connecting to the whole input image, in the case of perceptrons. You may find it useful to look back and compare how we implemented a fully connected network that connected to the whole image).
- **Parameter-sharing**: The same weights are used for different patches of the input image.

As a direct consequence of these two ideas, the number of weights (parameters) in the network decreases dramatically, and we see an improvement in the ability to capture **salient** features. Intuitively, it makes sense that nearby pixels are probably more relevant to each other than pixels that are far away from each other.

Typically, CNNs are composed for several **Convolutional (conv)** layers and subsampling (also known as **Pooling (P)**) layers that are followed by one or more **Fully Connected (FC)** layers at the end. The fully connected layers are essentially a multilayer perceptron, where every input unit i is connected to every output unit j with weight w_{ij}.

Please note that subsampling layers, commonly known as **pooling layers**, do not have any learnable parameters; for instance, there are no weights or bias units in pooling layers. However, both convolution and fully connected layers have such weights and biases.

In the following sections, we will study convolutional and pooling layers in more detail and see how they work. To understand how convolution operations work ,let's start with a convolution in one dimension before working through the typical two-dimensional cases as applications for two-dimensional images later.

2.2 Performing discrete convolutions

A **discrete convolution** (or simply **convolution**) is a fundamental operation in a CNN. Therefore, it is important to understand how this operation works. In this section, we will learn the mathematical definition and discuss some of the **naive** algorithms to compute convolutions of two one-dimensional vectors or two two-dimensional matrices.

Please note that this description is solely for understanding how a convolution works. Indeed, much more efficient implementations of convolutional operations already exist in packages such as TensorFlow, as we will see later in this chapter.

In this chapter, we will use subscripts to denote the size of a multidimensional array; for example, $A_{n_1 \times n_2}$ is a two-dimensional array of size $n_1 \times n_2$. We use brackets $[]$ to denote the indexing of a multidimensional array. For example, $A[i, j]$ means the element at index i, j of matrix A. Furthermore, note that we use a special symbol $*$ to denote the convolution operation between two vectors or matrices, which is not to be confused with the multiplication operator $*$ in Python.

2.2.1 Performing a discrete convolution in one dimension

Let's start with some basic definitions and notations we are going to use. A discrete convolution for two one-dimensional vectors x and w is denoted by $y = x * w$, in which vector x is our input (sometimes called **signal**) and w is called the **filter** or **kernel**. A discrete convolution is mathematically defined as follows:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i-k]w[k]$$

Here, the brackets [] are used to denote the indexing for vector elements. The index i runs through each element of the output vector y. There are two odds things in the preceding formula that we need to clarify: $-\infty$ to $+\infty$ indices and negative indexing for x.

Cross-correlation Cross-correlation (or simply correlation) between the input vector and a filter is denoted by $y = x * w$ and is very much like a sibling for a convolution with a small difference; the difference is that in cross-correlation, the multiplication is performed in the same direction. Therefore, it is not required to rotate the filter matrix w in each dimension. Mathematically, cross-correlation is defined as follows:

$$y = x * w \rightarrow y[i] = \sum_{k=-\infty}^{+\infty} x[i+k]w[k]$$

The same rules for padding and stride may be applied to cross-correlation as well.

The first issue where the sum runs through indices from $-\infty$ to $+\infty$ seems odd mainly because in machine learning applications, we always deal with finite feature vectors. For example, if x has 10 features with indices $0, 1, \ldots, 9$, then indices $-\infty : -1$ and $10 : +\infty$ are out of bounds for x. Therefore, to correctly compute the summation shown in the preceding formula, it is assumed that x and w are filled with zeros. This will result in an output vector y that also has infinite size with lots of zeros as well. Since this is not useful in practical situations, x is padded only with a finite number of zeros.

This process is called **zero-padding** or simply **padding**. Here, the number of zeros padded on each side is denoted by p. An example padding of a one-dimensional vector x is shown in the following figure:

Let's assume that the original input x and filter w have n and m elements, respectively, where $m \leq n$. Therefore, the padded vector x^p has size $n + 2p$. Then, the practical formula for computing a discrete convolution will change to the following:

$$y = x * w \rightarrow y[i] = \sum_{k=0}^{k=m-1} x^p[i+m-k]w[k]$$

Now that we have solved the infinite index issue, the second issue is indexing x with $i + m - k$. The important point to notice here is that x and w are indexed in different directions in this summation. For this reason, we can flip one of those vectors, x or w, after they are padded. Then, we can simply compute their dot product.

Let's assume we flip the filter w to get the rotated filter w^r. Then, the dot product $x[i : i+m].w^r$ is computed to get one element $y[i]$, where $x[i : i+m]$ is a patch of x with size m.

This operation is repeated like in a sliding window approach to get all the output elements. The following figure provides an example with $x = [3, 2, 1, 7, 1, 2, 5, 4]$ and $w = [\frac{1}{2}, \frac{3}{4}, 1, \frac{1}{4}]$ so that the first three output elements are computed:

You can see in the preceding example that the padding size is zero ($p = 0$). Notice that the rotated filter w^r is shifted by two cells each time we shift. This **shift** is another hyperparameter of a convolution, the **stride** s. In this example, the stride is two, $s = 2$. Note that the stride has to be a positive number smaller than the size of the input vector. We will talk more about padding and strides in the next section.

The effect of zero-padding in a convolution So far here, we have used zero-padding in convolutions to compute finite-sized output vectors. Technically, padding can be applied with any $p \geq 0$. Depending on the choice p, boundary cells may be treated differently than the cells localed in the middle of x.

Now consider an example where $n = 5, m = 3$. Then, $p = 0, x[0]$ is only used in computing one output element (for instance, $y[0]$), while $x[1]$ is used in the computation of two output elements (for instance, $y[0]$ and $y[1]$). So, you can see that this different treatment of elements of x can artificially put more emphasis on the middle element, $x[2]$, since it has appeared in most computations. We can avoid this issue if we choose $p = 2$, in which case, each element of x will be involved in computing three elements of y.

Furthermore, the size of the output y also depends on the choice of the padding strategy we use. There are three modes of padding that are commonly used in practice: **full**, **same**, and **valid**:

- In the **full** mode, the padding parameter p is set to $p = m - 1$. Full padding increases the dimensions of the output; thus, it is rarely used in convolutional neural network architectures.
- **Same** padding is usually used if you want to have the size of the output the same as the input vector x. In this case, the padding parameter p is computed according to the filter size, along with the requirement that the input size and output size are the same.
- Finally, computing a convolution in the **valid** mode refers to the case where $p = 0$ (no padding).

The following figure illustrates the three different padding modes for a simple 5×5 pixel input with a kernel size of 3×3 and a stride of 1:

The most commonly used padding mode in convolutional neural networks is **same** padding. One of its advantage over the other padding modes is that same padding preserves the height and width of the input images or tensors, which makes designing a network architecture more convenient.

One big disadvantage of the **valid** padding versus **full** and **same** padding, for example, is that the volume of the tensors would decrease substantially in neural networks with many layers, which can be detrimental to the network performance.

In practice, it is recommended that you preserve the spatial size using same padding for the convolutional layers and decrease the spatial size via pooling layers instead. As for the full padding, its size results in an output larger than the input size. Full padding is usually used in signal processing applications where it is important to minimize boundary effects. However, in deep learning context, boundary effect is not usually an issue, so we rarely see full padding.

Determining the size of the convolution output The output size of a convolution is determined by the total number of times hat we shift the filter w along the input vector. Let's assume that the input vector has size n and the filter is of size m. Then, the size of the output resulting from $x * m$ with padding p and stride s is determined as follows:

$$o = \left\lfloor \frac{n + 2p - m}{s} \right\rfloor + 1$$

Here, $\lfloor \rfloor$ denotes the floor operation.
Consider the following two cases:

- Compute the output size for an input vector of size 10 with a convolution kernel of size 5, padding 2, and stride 1:

$$n = 10, m = 5, p = 2, s = 1 \rightarrow o = \left\lfloor \frac{10 + 2.2 - 5}{1} \right\rfloor + 1 = 10$$

  (Note that in this case, the output size turns out to be the smae as the input; therefore, we conclude this as **mode='same'**).
- How can the output size change for the same input vector, but have a kernel of size 3, and stride 2?

$$n = 10, m = 3, p = 2, s = 2 \rightarrow o = \left\lfloor \frac{10 + 2.2 - 3}{2} \right\rfloor + 1 = 6$$

Finally, in order to learn how to compute convolutions in one dimension, a naive implementation is shown in the following code block, and the results are compared with the *numpy.convolve* function. The code is as follows:

```
In [1]: import numpy as np

        def conv1d(x, w, p=0, s=1):
            w_rot = np.array(w[::-1])
            x_padded = np.array(x)
            if p > 0:
                zero_pad = np.zeros(shape=p)
                x_padded = np.concatenate([zero_pad, x_padded, zero_pad])
            res = []
            for i in range(0, int(len(x)/s), s):
                res.append(np.sum(x_padded[i:i+w_rot.shape[0]] * w_rot))
            return np.array(res)

        ## testing
        x = [1, 3, 2, 4, 5, 6, 1, 3]
        w = [1, 0, 3, 1, 2]

        print('Conv1d implementation:', conv1d(x, w, p=2, s=1))
        print('Numpy results:', np.convolve(x, w, mode='same'))

Conv1d implementation: [ 5. 14. 16. 26. 24. 34. 19. 22.]
Numpy results: [ 5 14 16 26 24 34 19 22]
```

So far, here, we have explored the convolution in 1D. We started with 1D case to make the concepts easier to understand. In the next section, we will extend this to two dimensions,

2.2.2 Performing a discrete convolution in 2D

The concepts you learned in the previous sections are easily extendible to two dimensions. When we deal with two-dimensional input, such as matrix $X_{n_1 \times n_2}$ and the filter matrix $W_{m_1 \times m_2}$, where $m_1 \leq n1$ and $m_2 \leq n_2$, then the matrix $Y = X * W$ is the result of 2D convolution of X and W. This is mathematically defined as follows:

$$Y = X * W \rightarrow Y[i,j] = \sum_{k1=-\infty}^{+\infty} \sum_{k2=-\infty}^{+\infty} X[i-k_1, j-k_2]W[k_1, k_2]$$

Notice that if you omit one of the dimensions, the remaining formula is exactly the same as the one we used previously to compute the convolution in 1D. In fact, all the previously mentioned techniques, such as zero-padding, rotating the filter matrix, and the use of strides, are also applicable to 2D convolutions, provided that they are extended to both the dimensions independently. The following example illustrates the computation of a 2D convolution between an input matrix $X_{3\times3}$, a kernel matrix $W_{3\times3}$, padding $p = (1,1)$, and stride $s = (2,2)$. According to the specified padding, one layer of zeros are padded on each side of the input matrix, which results in the padded matrix $X^{padded}_{5\times5}$, as follows:

With the preceding filter, the rotate filter will be:

$$W^r = \begin{bmatrix} 0.5 & 1.0 & 0.5 \\ 0.1 & 0.4 & 0.3 \\ 0.4 & 0.7 & 0.5 \end{bmatrix}$$

Note that this rotation is not the same as the transpose matrix. To the the rotated filter in NumPy, we can write *W_rot = W[::-1, ::-1]*. Next, we can shift the rotated filter matrix along the padded input matrix X^{padded} like a sliding window and compute the sum of the element-wise product, which is denoted by the $\odot$ operator in the following picture:

The result will be the 2×2 matrix Y.

Let's also implement the 2D convolution according to the **naive** algorithm described. The *scipy.signal* package provides a way to compute 2D convolution via the *scipy.signal.convolve2d* function:

```
In [2]: import numpy as np
        import scipy.signal

        def conv2d(X, W, p=(0, 0), s=(1, 1)):
            W_rot = np.array(W)[::-1, ::-1]
            X_orig = np.array(X)
            n1 = X_orig.shape[0] + 2*p[0]
            n2 = X_orig.shape[1] + 2*p[1]
            X_padded = np.zeros(shape=(n1, n2))
            X_padded[p[0]:p[0]+X_orig.shape[0],
                     p[1]:p[1]+X_orig.shape[1]] = X_orig
            res = []
            for i in range (0, int((X_padded.shape[0] - W_rot.shape[0])/s[0])+1, s[0]):
                res.append([])
                for j in range(0, int((X_padded.shape[1] - W_rot.shape[1])/s[1])+1, s[1]):
                    X_sub = X_padded[i:i+W_rot.shape[0], j:j+W_rot.shape[1]]
```

```
                res[-1].append(np.sum(X_sub * W_rot))
        return np.array(res)

X = [[1, 3, 2, 4], [5, 6, 1, 3], [1, 2, 0, 2], [3, 4, 3, 2]]
W = [[1, 0, 3], [1, 2, 1], [0, 1, 1]]

print('Conv2d implementation:\n', conv2d(X, W, p=(1, 1), s=(1, 1)))
print('SciPy results:\n', scipy.signal.convolve2d(X, W, mode='same'))
```

```
Conv2d implementation:
 [[11. 25. 32. 13.]
 [19. 25. 24. 13.]
 [13. 28. 25. 17.]
 [11. 17. 14.  9.]]
SciPy results:
 [[11 25 32 13]
 [19 25 24 13]
 [13 28 25 17]
 [11 17 14  9]]
```

We provided a naive implementation to compute a 2D convolution for the purpose of understand the concepts. However, this implementation is very inefficient in terms of memory requirements and computational complexity. Therefore, it should not be used in real-world neural network applications.

In recent years, much more efficient algorithms have been developed that use the Fourier transformation for computing convolutions. It is also important to note that in the context of neural networks, the size of a convolutional kernel is usually much smaller than the size of the input image. For example, modern CNNs usually use kernel sizes such as 1×1, 3×3 or 5×5, for which efficient algorithms have been designed that can carry out the convolutional operations much more efficiently, such as the **Winograd's Minimal Filtering** algorithm.

In the next section, we will discuss subsampling, which is another important operation often used in CNNs.

2.3 Subsampling

Subsampling is typically applied in two forms of pooling operations in convolutional neural networks: **max pooling** and **mean-pooling** (also known as **average-pooling**).

The pooling layer is usually denoted by $P_{n_1 \times n_2}$. Here, the subscript determines the size of the neighborhodd (the number of adjacent pixels in each dimension), where the max or mean operation is performed. We refer to such a neighborhood as the **pooling size**.

The operation is described in the following figure. Here, max-pooling takes the maximum value from a neighborhood of pixels, and mean-pooling computes their average:

The advantage of pooling is twofold:

- Pooling (max-pooling) introduces some sort of local invariance. This means that small changes in a local neighborhood do not change the result of max-pooling. Therefore, it helps generate features that are more robust to noise in the input data.

- Pooling decreases the size of features, which results in higher computational efficiency. Furthermore, reducing the number of features may reduce the degree of overfitting as well.

Traditionally, pooling is assumed to be nonoverlapping. Pooling is typically performed on nonoverlapping neighborhoods, which can be done by setting the stride parameter equal to the pooling size. For example, a nonoverlapping pooling layer $P_{n_1 \times n_2}$ requires a stride parameter $s = (n_1, n_2)$. On the other hand, overlapping pooling occurs if the stride is smaller than pooling size.

3 Putting everything together to build a CNN

So far, we have learned about the basic building blocks of convolutional neural networks. The concepts illustrated in this chapter are not really more difficult than traditional multilayer neural networks. Intuitively, we can say that the most important operation in a traditional neural network is matrix-vector multiplication.

For instance, we use matrix-vector multiplications to pre-activations (or net input) as in $a = Wx + b$. Here, x is a column vector representing pixels, and W is the weight matrix connecting the pixel inputs to each hidden unit. In convolutional neural network, this operation is replaced by a convolution operation, as in $A = W * X + b$, where X is a matrix representing the pixels in a height $\times$ width arrangement. In both cases, the pre-activations are passed to an activation function to obtain the activation of a hidden unit $H = \phi(A)$, where ϕ is the activation function. Furthermore, recall that subsampling is another building block of a convolutional neural network, which may appear in the form of pooling, as we described in the previous section.

3.1 Working with multiple input of color channels

An input sample to a convolutional layer may contain one or more 2D arrays or matrices with dimensions $N_1 \times N_2$ (for example, the image height and width in pixels). These $N_1 \times N_2$ matrices are called **channels**. Therefore, using multiple channels as input to a convolutional layer requires us to use a rank-3 tensor or a three-dimensional array: $X_{N_1 \times N_2 \times C_{in}}$, where C_{in} is the number of input channels. For example, let's consider images as input to the first layer of a CNN. If the image is colored and uses the RGB color mode, then $C_{in} = 3$ (for the red, green, and blue colors channels in RGB). However, if the image is in grayscale, then we have $C_{in} = 1$ because there is only one channel with the grayscale pixel intensity values.

When we work with images, we can read images into NumPy arrays using the *'uint8'* (unsigned 8-bit integer) data type to reduce memory usage compared to 16-bit, 32-bit, or 64-bit integer types, for example. Unsigned 8-bit integers take values in the range [0, 255], which are sufficient to store the pixel information in RGB images, which also takes values in the same range.

Next, let's look at an example of how we can read in an image into our Python session using SciPy. However, please note that reading images with SciPy requires that you have the **Python Imaging Library (PIL)** package installed. We can install Pillow, a more user-friendly for of PIL, to satisfy those requirements, by executing the command **pip install pillow**.

Once Pillow is installed, we can use the *imread* function from the *scipy.misc* module to read an RGB image:

```
In [3]: import scipy.misc

        img = scipy.misc.imread('./example-image.png', mode='RGB')
```