15 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'll now learn about **Convolutional Neural Networks** (**CNNs**), and how we can implement CNNs in TensorFlow. We'll also take an interesting journey in this chapter as we apply this type of deep neural network architecture to image classification.

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

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

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 (*Handwritten Digit Recognition with a Back-Propagation Network, Y LeCun, and others*, *1989*, published at *Neural Information Processing Systems.(NIPS)* conference).

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'll delve into the theoretical definition of convolution and computing convolution in one and two dimensions.

Understanding CNNs and learning feature hierarchies

Successfully extracting **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's 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're 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.

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'll 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.

Performing discrete convolutions

A **discrete convolution** (or simply **convolution**) is a fundamental operation in a CNN. Therefore, it's important to understand how this operation works. In this section, we'll 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.

Mathematical notation

In this chapter, we will use subscripts to denote the size of a multidimensional array; for example, $\boldsymbol{A}_{n_1 \times n_2}$ is a two-dimensional array of size $n_1 \times n_2$. We use brackets $[\cdot]$ to denote the indexing of a multidimensional array. For example, $\boldsymbol{A}[i, j]$ means the element at index i, j of matrix $\boldsymbol{A}$. 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.

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 $\boldsymbol{y} = \boldsymbol{x} * \boldsymbol{w}$, 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:

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

You can see in the preceding example that the padding size is zero (*p = 0*). Notice that the rotated filter $\boldsymbol{w}^r$ 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'll talk more about padding and strides in the next section!

The effect of zero-padding in a convolution

So far here, we've 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 located 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).

Determining the size of the convolution output

The output size of a convolution is determined by the total number of times that 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 $\boldsymbol{x} * \boldsymbol{w}$ with padding *p* and stride *s* is determined as follows:

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

Here, $\lfloor \cdot \rfloor$ denotes the floor operation:

> The floor operation returns the largest integer that is equal or smaller to the input, for example:
>
> $$floor(1.77) = \lfloor 1.77 \rfloor = 1$$

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:

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

  (Note that in this case, the output size turns out to be the same 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?

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

If you are interested to learn more about the size of the convolution output, we recommend the manuscript *A guide to convolution arithmetic for deep learning, Vincent Dumoulin and Francesco Visin, 2016*, which is freely available at `https://arxiv.org/abs/1603.07285`.

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

```
>>> 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))
Conv1d Implementation: [  5. 14. 16. 26. 24. 34. 19. 22.]

>>> print('Numpy Results:',
... np.convolve(x, w, mode='same'))
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.

Performing a discrete convolution in 2D

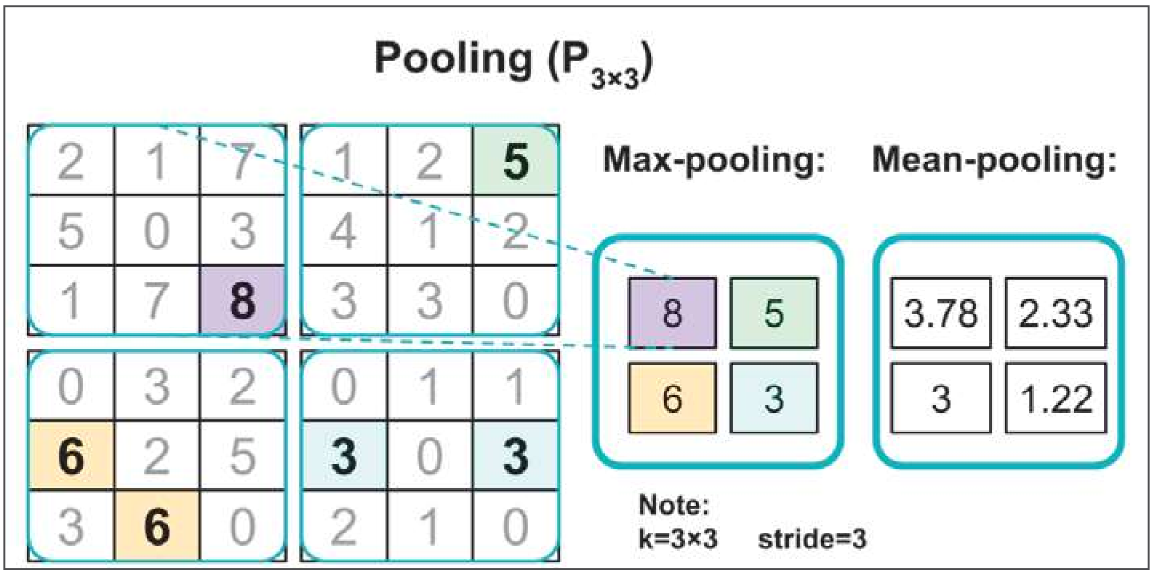
The concepts you learned in the previous sections are easily extendible to two dimensions. When we deal with two-dimensional input, such as a matrix $X_{n_1 \times n_2}$ and the filter matrix $W_{m_1 \times m_2}$, where $m_1 \leq n_1$ and $m_2 \leq n_2$, then the matrix $Y = X * W$ is the result of 2D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

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

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 $\boldsymbol{P}_{n_1 \times n_2}$. Here, the subscript determines the size of the neighborhood (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. See the following example that shows max-pooling of two different input matrices $\boldsymbol{X}_1$ and $\boldsymbol{X}_2$ results in the same output:

Putting everything together to build a CNN

So far, we've 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 the matrix-vector multiplication.

For instance, we use matrix-vector multiplications to pre-activations (or net input) as in $\boldsymbol{a} = \boldsymbol{W}\boldsymbol{x} + \boldsymbol{b}$. Here, **x** is a column vector representing pixels, and **W** is the weight matrix connecting the pixel inputs to each hidden unit. In a convolutional neural network, this operation is replaced by a convolution operation, as in $\boldsymbol{A} = \boldsymbol{W} * \boldsymbol{X} + b$, where **X** is a matrix representing the pixels in a height x width arrangement. In both cases, the pre-activations are passed to an activation function to obtain the activation of a hidden unit $\boldsymbol{H} = \phi(\boldsymbol{A})$, 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.

Working with multiple input or 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: $\boldsymbol{X}_{N_1 \times N_2 \times C_{in}}$, 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 color 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.

Regularizing a neural network with dropout

Choosing the size of a network, whether we are dealing with a traditional (fully connected) neural network or a CNN, has always been a challenging problem. For instance, the size of a weight matrix and the number of layers need to be tuned to achieve a reasonably good performance.

The **capacity** of a network refers to the level of complexity of the function that it can learn. Small networks, networks with a relatively small number of parameters, have a low capacity and are therefore likely to be **under fit**, resulting in poor performance since they cannot learn the underlying structure of complex datasets.

Yet, very large networks may more easily result in **overfitting**, where the network will memorize the training data and do extremely well on the training set while achieving poor performance on the held-out test set. When we deal with real-world machine learning problems, we do not know how large the network should be **a priori**.

One way to address this problem is to build a network with a relatively large capacity (in practice, we want to choose a capacity that is slightly larger than necessary) to do well on the training set. Then, to prevent overfitting, we can apply one or multiple regularization schemes to achieve good generalization performance on new data, such as the held-out test set. A popular choice for regularization is L2 regularization, which we discussed previously in this book.

In recent years, another popular regularization technique called **dropout** has emerged that works amazingly well for regularizing (deep) neural networks (*Dropout: a simple way to prevent neural networks from overfitting*, *Nitish Srivastava and. others, Journal of Machine Learning Research 15.1*, pages 1929-1958, *2014*, `http://www.jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf`).

Intuitively, dropout can be considered as the consensus (averaging) of an ensemble of models. In ensemble learning, we train several models independently. During prediction, we then use the consensus of all the trained models. However, both training several models and collecting and averaging the output of multiple models is computationally expensive. Here, dropout offers a workaround with an efficient way to train many models at once and compute their average predictions at test or prediction time.

Dropout is usually applied to the hidden units of higher layers. During the training phase of a neural network, a fraction of the hidden units is **randomly** dropped at every iteration with probability p_{drop} (or the keep probability $p_{keep} = 1 - p_{drop}$).

So, what is the relationship between dropout and ensemble learning? Since we drop different hidden neurons at each iteration, effectively we are training different models. When all these models are finally trained, we set the keep probability to 1 and use all the hidden units. This means we are taking the average activation from all the hidden units.

Implementing a deep convolutional neural network using TensorFlow

In *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow*, you may recall that we implemented a multilayer neural network for handwritten digit recognition problems, using different API levels of TensorFlow. You may also recall that we achieved about 97 percent accuracy.

So now, we want to implement a CNN to solve this same problem and see its predictive power in classifying handwritten digits. Note that the fully connected layers that we saw in the *Chapter 13*, *Parallelizing Neural Network Training with TensorFlow* were able to perform well on this problem. However, in some applications, such as reading bank account numbers from handwritten digits, even tiny mistakes can be very costly. Therefore, it is crucial to reduce this error as much as possible.

The multilayer CNN architecture

The architecture of the network that we are going to implement is shown in the following figure. The input is 28 x 28 grayscale images. Considering the number of channels (which is 1 for grayscale images) and a batch of input images, the input tensor's dimensions will be batchsize x 28 x 28 x 1.

The input data goes through two convolutional layers that have a kernel size of 5 x 5. The first convolution has 32 output feature maps, and the second one has 64 output feature maps. Each convolution layer is followed by a subsampling layer in the form of a max-pooling operation.

Loading and preprocessing the data

If you'll recall again from *Chapter 13, Parallelizing Neural Network Training with TensorFlow*, we used a function called `load_mnist` to read the MNIST handwritten digit dataset. Now we need to repeat the same procedure here as well, as follows:

```
>>> #### Loading the data
>>> X_data, y_data = load_mnist('./mnist/', kind='train')
>>> print('Rows: {},  Columns: {}'.format(
...               X_data.shape[0], X_data.shape[1]))
>>> X_test, y_test = load_mnist('./mnist/', kind='t10k')
>>> print('Rows: {},  Columns: {}'.format(
...               X_test.shape[0], X_test.shape[1]))

>>> X_train, y_train = X_data[:50000,:], y_data[:50000]
>>> X_valid, y_valid = X_data[50000:,:], y_data[50000:]

>>> print('Training:   ', X_train.shape, y_train.shape)
>>> print('Validation: ', X_valid.shape, y_valid.shape)
>>> print('Test Set:   ', X_test.shape, y_test.shape)
```

We are splitting the data into a training, a validation, and a test sets. The following result shows the shape of each set:

```
Rows: 60000,  Columns: 784
Rows: 10000,  Columns: 784
Training:    (50000, 784) (50000,)
Validation:  (10000, 784) (10000,)
Test Set:    (10000, 784) (10000,)
```

After we've loaded the data, we need a function for iterating through mini-batches of data, as follows:

```
>>> def batch_generator(X, y, batch_size=64,
...                     shuffle=False, random_seed=None):
...
...     idx = np.arange(y.shape[0])
...
...     if shuffle:
...         rng = np.random.RandomState(random_seed)
...         rng.shuffle(idx)
...         X = X[idx]
...         y = y[idx]

...     for i in range(0, X.shape[0], batch_size):
...         yield (X[i:i+batch_size, :], y[i:i+batch_size])
```

This function returns a generator with a tuple for a match of samples, for instance, data *X* and labels *y*. We then need to normalize the data (mean centering and division by the standard deviation) for better training performance and convergence.

We compute the mean of each feature using the training data (`X_train`) and calculate the standard deviation across all features. The reason why we don't compute the standard deviation for each feature individually is because some features (pixel positions) in image datasets such as MNIST have a constant value of 255 across all images corresponding to white pixels in a grayscale image.

A constant value across all samples indicates no variation, and therefore, the standard deviation of those features will be zero, and a result would yield the division-by-zero error, which is why we compute the standard deviation from the `X_train` array using `np.std` without specifying an `axis` argument:

```
>>> mean_vals = np.mean(X_train, axis=0)
>>> std_val = np.std(X_train)

>>> X_train_centered = (X_train - mean_vals)/std_val
>>> X_valid_centered = (X_valid - mean_vals)/std_val
>>> X_test_centered = (X_test - mean_vals)/std_val
```

Now we are ready to implement the CNN we just described. We will proceed by implementing the CNN model in TensorFlow.

Implementing a CNN in the TensorFlow low-level API

For implementing a CNN in TensorFlow, first we define two wrapper functions to make the process of building the network simpler: a wrapper function for a convolutional layer and a function for building a fully connected layer.

The first function for a convolution layer is as follows:

```
import tensorflow as tf
import numpy as np

def conv_layer(input_tensor, name,
               kernel_size, n_output_channels,
               padding_mode='SAME', strides=(1, 1, 1, 1)):
    with tf.variable_scope(name):
        ## get n_input_channels:
        ##   input tensor shape:
        ##   [batch x width x height x channels_in]
```

Implementing a CNN in the TensorFlow Layers API

For the implementation in the TensorFlow Layers API, we need to repeat the same process of loading the data and preprocessing steps to get `X_train_centered`, `X_valid_centered`, and `X_test_centered`. Then, we can implement the model in a new class, as follows:

```
import tensorflow as tf
import numpy as np


class ConvNN(object):
    def __init__(self, batchsize=64,
                 epochs=20, learning_rate=1e-4,
                 dropout_rate=0.5,
                 shuffle=True, random_seed=None):
        np.random.seed(random_seed)
        self.batchsize = batchsize
        self.epochs = epochs
        self.learning_rate = learning_rate
        self.dropout_rate = dropout_rate
        self.shuffle = shuffle

        g = tf.Graph()
        with g.as_default():
            ## set random-seed:
            tf.set_random_seed(random_seed)

            ## build the network:
            self.build()

            ## initializer
            self.init_op = \
                tf.global_variables_initializer()

            ## saver
            self.saver = tf.train.Saver()

        ## create a session
```

The obtained prediction accuracy is 99.32 percent, which means there are only 68 misclassified test samples!

This concludes our discussion on implementing convolutional neural networks using the TensorFlow low-level API and TensorFlow Layers API. We defined some wrapper functions for the first implementation using the low-level API. The second implementation was more straightforward since we could use the `tf.layers.conv2d` and `tf.layers.dense` functions to build the convolutional and the fully connected layers.

Summary

In this chapter, we learned about CNNs, or convolutional neural networks, and explored the building blocks that form different CNN architectures. We started by defining the convolution operation, then we learned about its fundamentals by discussing 1D as well as 2D implementations.

We also covered subsampling by discussing two forms of pooling operations: max-pooling and average-pooling. Then, putting all these blocks together, we built a deep convolutional neural network and implemented it using the TensorFlow core API as well as the TensorFlow **Layers** API to apply CNNs for image classification.

In the next chapter, we'll move on to **Recurrent Neural Networks** (**RNN**). RNNs are used for learning the structure of sequence data, and they have some fascinating applications, including language translation and image captioning!