# CHAPTER 7: Convolutional Neural Networks

## Overview

In this chapter, you will learn how convolutional **neural networks** (**CNNs**) process image data. You will also learn how to correctly use a CNN on image data.

By the end of the chapter, you will be able to create your own CNN for classification and object identification on any image dataset using TensorFlow.

## Introduction

In This chapter covers CNNs. CNNs use convolutional layers that are well-suited to extracting features from images. They use learning filters that correlate with the task at hand. Simply put, they are very good at finding patterns in images.

In the previous chapter, you explored regularization and hyperparameter tuning. You used L1 and L2 regularization and added dropout to a classification model to prevent overfitting on the connect-4 dataset.

You will now be shifting gears quite a bit as you dive into deep learning with CNNs. In this chapter, you will learn the fundamentals of how CNNs process image data and how to apply those concepts to your own image classification problem. This is truly where TensorFlow shines.

## CNNs

CNNs share many common components with the ANNs you have built so far. The key difference is the inclusion of one or more convolutional layers within the network. Convolutional layers apply convolutions of input data with filters, also known as kernels. Think of a convolution as an image transformer. You have an input image, which goes through the CNN and gives you an output label. Each layer has a unique function or special ability to detect patterns such as curves or edges in an image. CNNs combine the power of deep neural networks and kernel convolutions to transform images and make these image edges or curves easy for the model to see. There are three key components in a CNN:

* **Input image:** The raw image data
* **Filter/kernel:** The image transformation mechanism
* **Output label:** The image classification

The following figure is an example of a CNN in which the image is input into the network on the left-hand side and the output is generated on the right-hand side. The image components are identified throughout the hidden layers with more basic components, such as edges, identified in earlier hidden layers. Image components combine in the hidden layers to form recognizable features from the dataset. For example, in a CNN to classify images into planes or cars, the recognizable features may be filters that resemble a wheel or propellor. Combinations of these features will be instrumental in determining whether the image is a plane or a car.

Finally, the output layer is a dense layer used to determine the specific output of the model. For a binary classification model, this may be a dense layer with one unit with a sigmoid activation function. For a more complex multi-class classification, it may be a dense layer with many units, determined by the number of classes, and a softmax activation function to determine one output label for each image presented to the model.

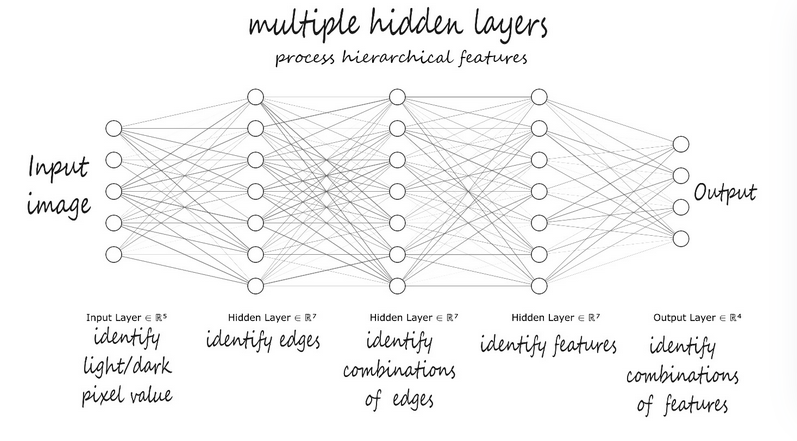


Figure 7.1: CNN

A common CNN configuration includes a convolutional layer followed by a pooling layer. These layers are often used together in this order, as pairs (convolution and pooling). We'll get into the reason for this later in the chapter, but for now, think of these pooling layers as decreasing the size of input images by summarizing the filter results.

Before you move deeper into convolutional layers, you first need to understand what the data looks like from the computer's perspective.

## Image Representation

First, consider how a computer processes an image. To a computer, images are numbers. To be able to work with images for classification or object identification, you need to understand how a model transforms an image input into data. A pixel in an image file is just a piece of data.

In the following figure, you can see an example of pixel values for a grayscale image of the number eight. For the 28x28-pixel image, there are a total of 784 pixels. Each pixel has a value between 0 and 255 identifying how light or dark the pixel is. On the right side, there is one large column vector with each pixel value listed. This is used by the model to identify the image.

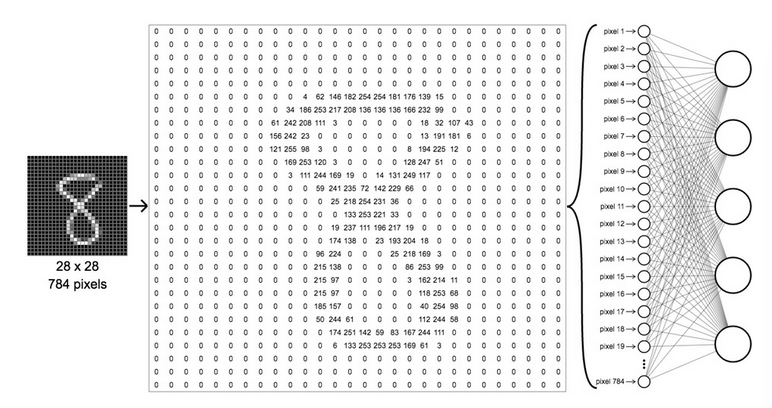


Figure 7.2: Pixel values

## The Convolutional Layer

Think of a convolution as nothing more than an image transformer with three key elements. First, there is an input image, then a filter, and finally, a feature map.

This section will cover each of these in turn to give you a solid idea of how images are filtered in a convolutional layer. The convolution is the process of passing a filter window over the input data, which will result in a map of activations known as a feature map. The input data may be the input image to the model or the output of a prior, intermediary layer of the model. The filter is generally a much smaller array, such as 3x3 for two-dimensional data, in which the specific values of the filter are learned during the training process. The filter passes across the input data with a window size equal to the size of the filter, then, the scalar product of the filter and section of the input data is applied, producing what's known as an activation. As this process continues across the entire input data using the same filter, the map of activations is produced, also known as the feature map.

This concept is illustrated in the following figure, which has two convolutional layers, producing two sets of feature maps. After the feature maps are produced from the first convolutional layer, they are passed into the second convolutional layer. The feature map of the second convolutional layer is passed into a classifier:

Diagram

Description automatically generated

Figure 7.3: Convolution for classification

The distance, or number of steps, the filter moves with each operation is known as the **stride**. If the filter goes off the edge, you can do what's called **padding with zeros**. This way, the output map size is the same as the input map size. This is called **same padding**. However, if the filter cannot take its required stride without leaning over the edge somewhat, it will count any value over the edge as 0. This is known as **valid padding**.

Let's recap some keywords. There's a **kernel**, which is a small matrix that is used to apply an effect, and what you saw in the example was a 2x2 kernel. There's **stride**, which is the number of pixels that you move the kernel by. Lastly, there's **padding with zeros** around the image, whether or not you add pixels. This ensures that the output is the same size as the input.

## Creating the Model

From the very first chapter, you encountered different types of dimensional tensors. One important thing to note is that you will only be working with Conv2D. The layer name Conv2D refers only to the movement of a **filter** or **kernel**. So, if you recall the description of what the convolutional process is doing, it's simply sliding a kernel across a 2D space. So, for a flat, square image, the kernel only slides in two dimensions.

When you implement Conv2D, you need to pass in certain parameters:

1. The first parameter is filter. The filters are the dimensionality of the output space.
2. Specify strides, which is how many pixels will move the kernel across.
3. Then, specify padding, which is usually valid or same depending on whether you want an output that is of the same dimension as the input.
4. Finally, you can also have activation. Here, you will specify what sort of activation you would like to apply to the outputs. If you don't specify an activation, it's simply a linear activation.

Before you continue, recall from Chapter 4, Regression and Classification Models, that a dense layer is one in which every neuron is connected to every neuron in the previous layer. As you can see in the following code, you can easily add a dense layer with model.add(Dense(32)). 32 is the number of neurons, followed by the input shape. AlexNet is an example of a CNN with multiple convolution kernels that extracts interesting information from an image.

Diagram

Description automatically generated

Figure 7.4: AlexNet consists of five convolution layers and three connected layers

Note

AlexNet is the name of a CNN designed by Alex Krizhevsky.

A sequential model can be used to build a CNN. Different methods can be used to add a layer; here, we will use the framework of sequentially adding layers to the model using the model's add method or passing in a list of all layers when the model is instantiated:

*model = models.Sequential()*

*model.add(Dense(32, input\_shape=(250,)))*

The following is a code block showing the code that you'll be using later in the chapter:

*our\_cnn\_model = models.Sequential([layers.Conv2D\*

*(filters = 32, \*

*kernel\_size = (3,3),*

*input\_shape=(28, 28, 1)), \*

*layers.Activation('relu'), \*

*layers.MaxPool2D\*

*(pool\_size = (2, 2)), \*

*layers.Conv2D\*

*(filters = 64, \*

*kernel\_size = (3,3)), \*

*layers.Activation('relu'), \*

*layers.MaxPool2D\*

*(pool\_size = (2,2)), \*

*layers.Conv2D\*

*(filters = 64, \*

*kernel\_size = (3,3)), \*

*layers.Activation('relu')])*

Use the Conv2D layer when working with data that you want to convolve in two dimensions, such as images. For parameters, set the number of filters to 32, followed by the kernel size of 3x3 pixels ((3, 3) in the example). In the first layer, you will always need to specify the input\_shape dimensions, the height, width, and depth. input\_shape is the size of the images you will be using. You can also select the activation function to be applied at the end of the layer.

Now that you have learned how to build a CNN layer in your model, you will practice doing so in your first exercise. In this exercise, you will build the first constructs of a CNN, initialize the model, and add a single convolutional layer to the model.

## Exercise 7.01: Creating the First Layer to Build a CNN

As a TensorFlow freelancer, you've been asked to show your potential employer a few lines of code that demonstrate how you might build the first layer in a CNN. They ask that you keep it simple but provide the first few steps to create a CNN layer. In this exercise, you will complete the first step in creating a CNN—that is, adding the first convolutional layer.

Follow these steps to complete this exercise:

1. Open a new Jupyter notebook.
2. Import the TensorFlow library and the models and layers classes from tensorflow.keras:

*import tensorflow as tf*

*from tensorflow.keras import models, layers*

1. Check the TensorFlow version:

*print(tf.\_\_version\_\_)*

You should get the following output:



1. Now, use models.Sequential to create your model. The first layer (Conv2D) will require the number of nodes (filters), the filter size (3,3), and the shape of the input. input\_shape for your first layer will determine the shape of your input images. Add a ReLU activation layer:

*image\_shape = (300, 300, 3)*

*our\_first\_layer = models.Sequential([layers.Conv2D\*

*(filters = 16, \*

*kernel\_size = (3,3), \*

*input\_shape = image\_shape), \*

*layers.Activation('relu')])*

Simple enough. You have just taken the first steps in creating your first CNN

You will now move on to the type of layer that usually follows a convolutional layer-the pooling layer.

## Pooling Layer

Pooling is an operation that is commonly added to a CNN to reduce the dimensionality of an image by reducing the number of pixels in the output from the convolutional layer it follows. **Pooling layers** shrink the input image to increase computational efficiency and reduce the number of parameters to limit the risk of **overfitting**.

A **pooling layer** immediately follows a convolution layer and is considered another important part of the CNN structure. This section will focus on two types of pooling:

* Max pooling
* Average pooling

## Max Pooling

With max pooling, a filter or kernel only retains the largest pixel value from an input matrix. To get a clearer idea of what is happening, consider the following example. Say you have a 4x4 input. This first step in max pooling would be to divide the 4x4 matrix into four quadrants. Each quadrant will be of the size 2x2. Apply a filter of size 2. This means that your filter will look exactly like a 2x2 matrix.

Begin by placing the filter on top of your input. For max pooling, this filter will look at all values within the 2x2 area that it covers. It will find the largest value, send that value to your output, and store it there in the upper-left corner of the feature map.

Diagram

Description automatically generated

Figure 7.5: Max Pooling

Then, the filter will move over to the right and repeat the same process, storing the value in the upper-right corner of the 2x2 matrix. Once this operation is complete, the filter will slide down and start at the far left, again repeating the same process, looking for the largest (or maximum) value, and then storing it in the correct place on the 2x2 matrix.

Recall that the sliding movement is referred to as stride. So, the filter was moving over two places. This would mean it has a stride value of 2. This process is repeated until the maximum values in each of the four quadrants are 8, 5, 7, and 5, respectively. Again, to get these numbers, you used a filter of 2x2 and filtered for the largest number within that 2x2 matrix.

So, in this case, you had a stride of two because you moved two pixels. These are the **hyperparameters** for max pooling. The values of filter and stride are 2. Figure 7.6 shows what an implementation of max pooling might look like with a filter size of 3 x 3 and a stride of 1.

There are two steps shown in Figure 7.6. Start at the upper left of the feature map. With the 3x3 filter, you would look at the following numbers, 2, 8, 2, 5, 4, 9, 8, 4, and 6, and choose the largest value, 9. The 9 would be placed in the upper-left box of our pooled feature map. With a stride of 1, you would slide the filter one place to the right, as shown in gray.

Now, look for the largest values from 8, 2, 1, 4, 9, 6, 4, 6, and 4. Again, 9 is the largest value, so add a 9 to the middle place in the top row of the pooled feature map (shown in gray).

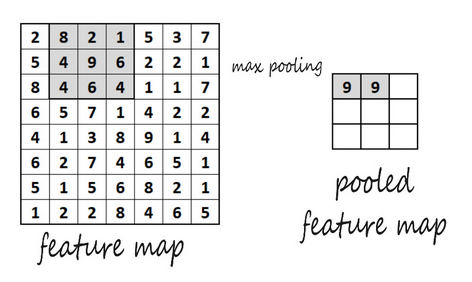


Figure 7.6: Pooled feature map

The preceding pool size is (2, 2). It specifies factors that you will downscale with. Here's a more detailed look at what you could do to implement MaxPool2D:

*layers.MaxPool2D(pool\_size=(2, 2), strides=None, \*

*padding='valid')*

**MaxPool2D**: The preceding code snippet introduces a MaxPool2D instance. The code snippet initializes a max pooling layer with a pool size of 2x2 and the stride value is not specified, so it will default to the pool size value. The padding parameter is set to valid, meaning there is no padding added. The following code snippet demonstrates its use within a CNN:

*image\_shape = (300, 300, 3)*

*our\_first\_model = models.Sequential([*

*layers.Conv2D(filters = 16, kernel\_size = (3,3), \*

*input\_shape = image\_shape), \*

*layers.Activation('relu'), \*

*layers.MaxPool2D(pool\_size = (2, 2)), \*

*layers.Conv2D(filters = 32, kernel\_size = (3,3)), \*

*layers.Activation('relu')])*

In the preceding example, a sequential model is created with two convolutional layers, after each layer is a ReLU activation function, and after the activation function of the first convolutional layer is a max pooling layer.

Now that you have explored max pooling, let's look at the other type of pooling: average pooling.

## Average Pooling

**Average pooling** operates in a similar way to max pooling, but instead of extracting the largest weight value within the filter, it calculates the average. It then passes along that value to the feature map. Figure 7.7 highlights the difference between max pooling and average pooling.

In Figure 7.7, consider the 4x4 matrix on the left. The average of the numbers in the upper-left quadrant is 13. This would be the average pooling value. The same upper-left quadrant would output 20 to its feature map if it were max pooled because 20 is the largest value within the filter frame. This is a comparison between max pooling and average pooling with hyperparameters, with the filter and stride parameters both set to 2:

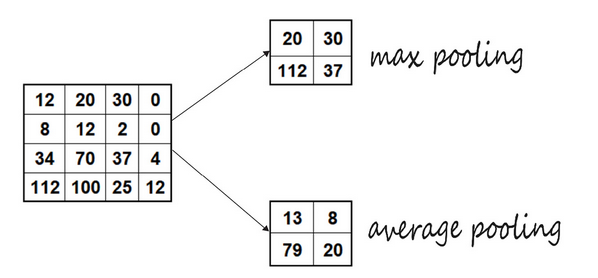


Figure 7.7: Max versus average pooling

For average pooling, you would use AveragePooling2D in place of MaxPool2D.

To implement the average pooling code, you could use the following:

*layers.AveragePooling2D(pool\_size=(2, 2), strides=None, \*

*padding='valid')*

**AveragePooling2D**: The preceding code snippet demonstrates how to invoke an AveragePooling2D layer. In a similar manner to max pooling, the pool\_size, strides, and padding parameters can be modified. The following code snippet demonstrates its use within a CNN:

*image\_shape = (300, 300, 3)*

*our\_first\_model = models.Sequential([*

*layers.Conv2D(filters = 16, kernel\_size = (3,3), \*

*input\_shape = image\_shape), \*

*layers.Activation('relu'), \*

*layers.AveragePooling2D(pool\_size = (2, 2)), \*

*layers.Conv2D(filters = 32, kernel\_size = (3,3)), \*

*layers.Activation('relu')])*

It's a good idea to keep in mind the benefits of using pooling layers. One of these benefits is that if you down-sample the image, the image shrinks. This means that you have less data to process and fewer multiplications to do, which, of course, speeds things up.

Up to this point, you've created your first CNN layer and learned how to use pooling layers. Now you'll use what you've learned so far to build a pooling layer for the CNN in the following exercise.

## Exercise 7.02: Creating a Pooling Layer for a CNN

In this exercise, You receive an email from your potential employer for the TensorFlow freelancing job that you applied for in Exercise 7.01, Creating the First Layer to Build a CNN. The email asks whether you can show how you would code a pooling layer for a CNN. In this exercise, you will build your base model by adding a pooling layer, as requested by your potential employer:

1. Open a new Jupyter notebook.
2. Import the TensorFlow library:

*import tensorflow as tf*

*from tensorflow.keras import models, layers*

1. Create your model using models.Sequential. The first layer, Conv2D, will require the number of nodes, the filter size, and the shape of the tensor, as in the previous exercise. It will be followed by an activation layer, a node at the end of the neural network:

*image\_shape = (300, 300, 3)*

*our\_first\_model = models.Sequential([*

*layers.Conv2D(filters = 16, kernel\_size = (3,3), \*

*input\_shape = image\_shape), \*

*layers.Activation('relu')])*

1. Now, add a MaxPool2D layer by using the model’s add method.

*our\_first\_model.add(layers.MaxPool2D(pool\_size = (2, 2)))*

In this model, you have created a CNN with a convolutional layer followed by a ReLU activation function then a max pooling layer. The models take images of size 300x300 with three color channels.

Now that you have successfully added a MaxPool2D layer to your CNN, the next step is to add a **flattening layer** so that your model can use all the data.

## Flattening Layer

Adding a flattening layer is an important step as you will need to provide the neural network with data in a form that it can process. Remember that after you perform the convolution operation, it will still be multi-dimensional. So, to change your data back into one-dimensional form, you will use a flattening layer. To achieve this, you take the pooled feature map and flatten it into a column, as shown in the following figure. In Figure 7.8, you can see that you start with the input matrix on the left side of the diagram, use a final pooled feature map, and stretch it out into a single column vector:

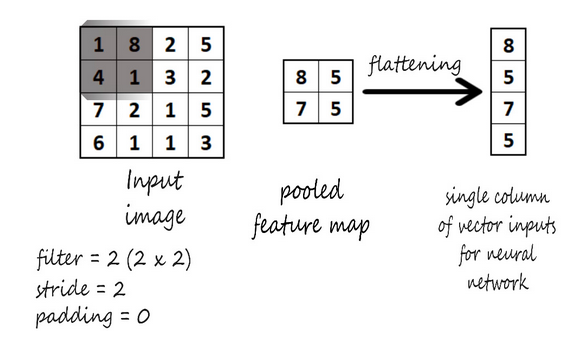


Figure 7.8: Flattening layer

The following is an implemented flattening layer:

*image\_shape = (300, 300, 3)*

*our\_first\_model = models.Sequential([*

*layers.Conv2D(filters = 16, kernel\_size = (3,3), \*

*input\_shape = image\_shape), \*

*layers.Activation('relu'), \*

*layers.MaxPool2D(pool\_size = (2, 2)), \*

*layers.Conv2D(filters = 32, kernel\_size = (3,3)), \*

*layers.Activation('relu'), \*

*layers.MaxPool2D(pool\_size = (2, 2)), \*

*layers.Flatten()])*

Here, a flatten layer is added as the final layer to this model. Now that you've created your first CNN and pooling layers, you will put all the pieces together and build a CNN in the upcoming exercise.

## Image Augmentation

Augmentation is defined as making something better by making it greater in size or amount. This is exactly what data or image augmentation does. You use augmentation to provide the model with more versions of your image training data. Remember that the more data you have, the better the model's performance will be. By augmenting your data, you can transform your images in a way that makes the model generalize better on real data. To do this, you transform the images that you have at your disposal so that you can use your augmented images alongside your original image dataset to train with a greater variation and variety than you would have otherwise. This improves results and prevents overfitting. Take a look at the following three images:

A group of lions

Description automatically generated with medium confidence

Figure 7.9: Augmented lion images

It's clear that this is the same leopard in all three images. They're just in different positions. Neural networks can still make sense of this due to convolution. However, with the use of image augmentation, you can improve the model's ability to learn translational invariance.

Unlike most other types of data with images, you can shift, rotate, and move the images around to make variations of the original image. This creates more data, and with CNNs, more data and data variation will create a better-performing model. To be able to create these image augmentations, take a look at how you would do this in TensorFlow with the loaded tf.data.Dataset object. You will use the dataset.map() function to map preprocessing image augmentation functions to your dataset, that is, our\_train\_dataset:

*from tensorflow import image as tfimage*

*from tensorflow.keras.preprocessing import image as kimage*

You will use the tensorflow.image and tensorflow.keras.preprocessing.image packages for this purpose. These packages have a lot of image manipulation functions that can be used for image data augmentation:

*augment\_dataset(image, label):*

*image = kimage.random\_shift(image, wrg = 0.1, hrg = 0.1)*

*image = tfimage.random\_flip\_left\_right(image)*

*return image, label*

Additional functions include the following:

* kimage.random\_rotation: This function allows you to rotate an image randomly between specified degrees.
* kimage.random\_brightness: This function randomly adjusts the brightness level.
* kimage.random\_shear: This function applies shear transformations.
* kimage.random\_zoom: This function randomly zooms images.
* tfimage.random\_flip\_left\_right: This function randomly flips images horizontally.
* tfimage.random\_flip\_up\_down: This function randomly flips images vertically.

In the next step, you will pass in the data that you want to augment with the tf.data.Dataset.map() function:

*augment\_dataset(image, label):*

*image = kimage.random\_shift(image, wrg = 0.1, hrg = 0.1)*

*image = tfimage.random\_flip\_left\_right(image)*

*return image, label*

*our\_train\_dataset = our\_train\_dataset.map(augment\_dataset)*

*model.fit(our\_train\_dataset,\*

*epochs=50,\*

*validation\_data=our\_test\_dataset)*

In the preceding code block, with fit(), you just need to pass the generator that you have already created. You need to pass in the epochs value. If you don't do this, the generator will never stop. The fit() function returns the history (plots loss per iteration and so on).

You need some more functions to add to our\_train\_dataset before you can train the model on it. With batch() function, you specify how many images per batch you will train. With cache() function, you fit your dataset in memory to improve performance. With shuffle() function, you set the shuffle buffer of your dataset to the entire length of the dataset, for true randomness. prefetch() function is also used for good performance:

*our\_train\_dataset = our\_train\_dataset.cache()*

*our\_train\_dataset = our\_train\_dataset.map(augment\_dataset)*

*our\_train\_dataset = our\_train\_dataset.shuffle\*

*(len(our\_train\_dataset))*

*our\_train\_dataset = our\_train\_dataset.batch(128)*

*our\_train\_dataset = our\_train\_dataset.prefetch\*

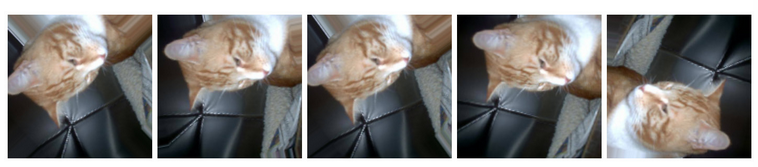
*(tf.data.experimental.AUTOTUNE)*

Now that you've seen how you would implement augmentation in your training model, take a closer look at what some of those transformations are doing.

Here's an example of random\_rotation, random\_shift, and random\_brightnes implementation. Use the following code to randomly rotate an image up to an assigned value:

*image = kimage.random\_rotation(image, rg = 135)*

In Figure 7.10, you can see the outcome of random\_rotation



The images were randomly rotated up to 135 degrees.

random\_shift is used to randomly shift the pixels width-wise. Notice the .15 in the following code, which means the image can be randomly shifted up to 15 pixels:

*image = kimage.random\_shift(image, wrg = 0.15, hrg = 0)*

The following figure shows the random adjustment of an image's width by up to 15 pixels:

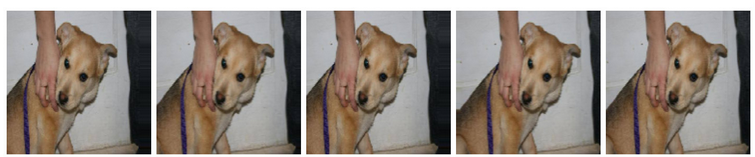


Figure 7.11: Width shift range

Again, random\_shift is used here, which randomly adjusts the height by 15 pixels:

*image = kimage.random\_shift(image, wrg = 0, hrg = 0.15)*

Figure 7.12 shows the random adjustment of an image’s height by up to 15 pixels:

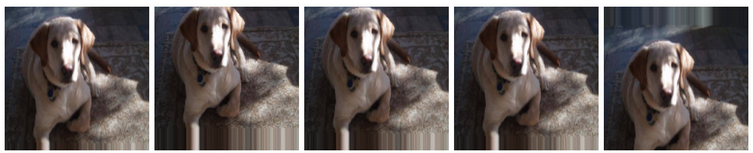


Figure 7.13: Height shift range

For random brightness levels using random\_brightness, you will use a float value range to lighten or darken the image by percentage. Anything below 1.0 will darken the image. So, in this example, the images are being darkened randomly between 10% and 90%:

*image = kimage.random\_brightness(image, brightness\_range=(0.1,0.9))*

In the following figure, you've adjusted the brightness with random\_brightness:

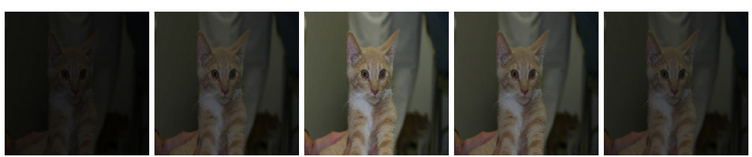


Figure 7.14 Brightness range

Now that you've been exposed to some of the image augmentation options, take a look at how you can use batch normalization to drive performance improvement in models.

## Batch Normalization

In 2015, batch normalization, also called batch norm, was introduced by Christian Szegedy and Sergey Ioffe. Batch norm is a technique that reduces the number of training epochs to improve performance. Batch norm standardizes the inputs for a mini-batch and "normalizes" the input layer. It is most commonly used following a convolutional layer, as shown in the following figure:

Text

Description automatically generated

Figure 7.14: Batch norm

The following figure shows one common way that batch normalization is implemented. In the following example, you can see that you have a batch norm layer following a convolutional layer three times. Then you have a flattening layer, followed by two dense layers:

Text

Description automatically generated

Figure 7.15: Layer sequences

Batch norm helps the model generalize better. With each batch that batch norm trains, the model has a different mean and standard deviation. Because the batch means and standard deviations each vary slightly from the true overall mean and standard deviation, these changes act as noise that you are training with, making the model perform better overall.

The following is an example of BatchNormalization implementation. You can simply add a batch norm layer, followed by an activation layer:

*model.add(layers.Conv2D(filters = 64, kernel\_size = (3, 3), use\_bias=False))*

*model.add(layers.BatchNormalization())*

*model.add(layers.Activation("relu"))*

So far, you've created a CNN model and learned how to utilize image augmentation.

## Binary Image Classification

Binary classification is the simplest approach for classification models as it classifies images into just two categories. In this chapter, we started with the convolutional operation and discussed how you use it as an image transformer. Then, you learned what a pooling layer does and the differences between max and average pooling. Next, we also looked at how a flattening layer converts a pooled feature map into a single column. Then, you learned how and why to use image augmentation, and how to use batch normalization. These are the key components that differentiate CNNs from other ANNs.

After convolutional base layers, pooling, and normalization layers, CNNs are often structured like many ANNs you've built thus far, with a series of one or more dense layers. Much like other binary classifiers, binary image classifiers terminate with a dense layer with one unit and a sigmoid activation function. To provide more utility, image classifiers can be outfitted to classify more than two objects. Such classifiers are known generally as object classifiers, which you will learn about in the next section.

## Object Classification

In this section, you will learn about object detection and classification. The next step involves image classification for a dataset with more than two classes. The three different types of models for object classification we will cover are **image classification**, **classification with localization**, and **detection**:

* **Image classification**: This involves training with a set number of classes and then trying to determine which of those classes is shown in the image. Think of the MNIST handwriting dataset. For these problems, you'll use a traditional CNN.
* **Classification with localization**: With this type, the model tries to predict where the object is in the image space. For these models, you use a simplified You Only Look Once (YOLO) or R-CNN.
* **Detection**: The last type is detection. This is where your model can detect several different objects and where they are located. For this, you use YOLO or an R-CNN:



Figure 7.16: Object Classification types

## Exercise 7.03: Building a CNN

Now, you'll take a brief look at image classification with the MNIST dataset. The dataset consists of 10 classes with a training set of 60,000 28x28 grayscale images and 10,000 test images.

1. Import necessary packages:

Text

Description automatically generated

1. Load the MNIST dataset using tdfs in any one of the datasets that they have decided to include. Others include CIFAR-10 and CIFAR-100, just to name to couple:

Graphical user interface, text

Description automatically generated

1. Check the data for its properties:

Text

Description automatically generated

This will give you the following output:

Chart

Description automatically generated with low confidence

Figure 7.17: Details of properties for data

1. Now, print the total examples of the train and test data:

Text

Description automatically generated

This will give you the following output:

A picture containing text, orange, dark

Description automatically generated

1. Create a function to normalize the images and set some parameters

Text

Description automatically generated

Shape

Description automatically generated with medium confidence

1. Setup for the train dataset

Text

Description automatically generated

1. Setup for the test dataset

Text

Description automatically generated with medium confidence

1. Build your model with the functional API:

Text

Description automatically generated

1. Compile and fit your model. With compile( ) method, use adam as your optimizer, set the loss to spare\_categorical\_crossentropy, and set the accuracy metric. Then, call model.fit( ) on your training and validation sets:

Graphical user interface, text, application

Description automatically generated



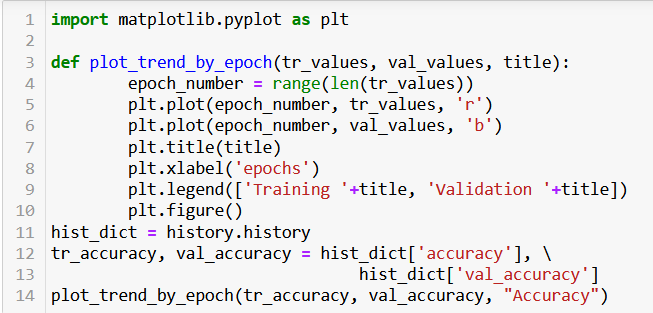
This will give the following as output:

Table

Description automatically generated

Figure 7.18: Function returning history

1. Use Matplotlib.pyplot to plot the loss and accuracy:



This will give the following plot as output:

Chart, line chart

Description automatically generated

Figure 7.19: Training and Validation Accuracy

1. Plot the validation loss and training loss. Use the following code:

A picture containing text

Description automatically generated

This will give the following plot as output:

Chart

Description automatically generated

Figure 7.20: Training and Validation loss

As you can see from the accuracy and loss curves as a function of epochs, the accuracy increases, and loss decreases. On the validation set, both to plateau, which is good signal to stop training to prevent overfitting to the training datast.

Now, that you have completed this chapter, it’s time to put everything that you’ve learned to the test with Activity 7.01, Building a CNN with More ANN Layers, where you’ll be building a CNN with additional ANN layers.

## Activity 7.01: Building a CNN with More ANN Layers

The start-up that you've been working for has loved your work so far. They have tasked you with creating a new model classifying images from 10 different classes.

In this activity, you'll be putting everything that you've learned to use as you build your own classifier with dataset, with 10 classes, and is commonly used for benchmarking performance in machine learning research.

* Start a new Jupyter notebook.
* Import the TensorFlow library.
* Import the additional libraries that you will need, including NumPy, Matplotlib, Input, Conv2D, Dense, Flatten, Dropout, GlobalMaxPooling2D, Activation, Model, confusion\_matrix, and itertools.
* Load the CIFAR-10 dataset directly from tensorflow\_datasets and view its properties from the metadata, and build a train and test data pipeline:

A picture containing text

Description automatically generated

A screenshot of a computer

Description automatically generated with low confidence

* Create a function to rescale images. Then, build a test and train data pipeline by rescaling, caching, shuffling, batching, and prefetching the images.
* Build the model using the functional API using Conv2D and Flatten, among others.
* Compile and fit the model using model.compile and model.fit:

Table

Description automatically generated

* Plot the loss with plt.plot. Remember to use the history collected during the model.fit() procedure:

Chart, line chart

Description automatically generated

* Plot the accuracy with plt.plot:

Chart, line chart

Description automatically generated

* Specify the labels for the different classes in your dataset.
* Display a misclassified example with plt.imshow:

Graphical user interface

Description automatically generated with low confidence

Note

The solution to this activity can be found via this link

## SUMMARY

This chapter covered CNNs. We reviewed core concepts such as neurons, layers, model architecture, and tensors to understand how to create effective CNNs.

You learned about the convolution operation and explored kernels and feature maps. We analyzed how to assemble a CNN, and then explored the different types of pooling layers and when to apply them.

You then learned about the stride operation and how padding is used to create extra space around images if needed. Then, we delved into the flattening layer and how it is able to convert data into a 1D array for the next layer. You put everything that you learned to the test in the final activity, as you were presented with several classification problems, including MNIST and even CIFAR-10.

In completing this chapter, you are now well on your way to being able to implement CNNs to confront image classification problems head-on and with confidence.

In the next chapter, you'll learn about pre-trained models and how to utilize them for your own applications by adding ANN layers on top of the pre-trained model and fine-tuning the weights given your own training data.