TensorFlow Convolution Operations

TensorFlow also offers a few other kinds of convolutional layers:

- `conv1d()` creates a convolutional layer for 1D inputs. This is useful, for example, in natural language processing, where a sentence may be represented as a 1D array of words, and the receptive field covers a few neighboring words.
- `conv3d()` creates a convolutional layer for 3D inputs, such as 3D PET scan.
- `atrous_conv2d()` creates an *atrous convolutional layer* ("à trous" is French for "with holes"). This is equivalent to using a regular convolutional layer with a filter dilated by inserting rows and columns of zeros (i.e., holes). For example, a 1×3 filter equal to `[[1,2,3]]` may be dilated with a *dilation rate* of 4, resulting in a *dilated filter* `[[1, 0, 0, 0, 2, 0, 0, 0, 3]]`. This allows the convolutional layer to have a larger receptive field at no computational price and using no extra parameters.
- `conv2d_transpose()` creates a *transpose convolutional layer*, sometimes called a *deconvolutional layer*,[15] which *upsamples* an image. It does so by inserting zeros between the inputs, so you can think of this as a regular convolutional layer using a fractional stride. Upsampling is useful, for example, in image segmentation: in a typical CNN, feature maps get smaller and smaller as you progress through the network, so if you want to output an image of the same size as the input, you need an upsampling layer.
- `depthwise_conv2d()` creates a *depthwise convolutional layer* that applies every filter to every individual input channel independently. Thus, if there are f_n filters and $f_{n'}$ input channels, then this will output $f_n \times f_{n'}$ feature maps.
- `separable_conv2d()` creates a *separable convolutional layer* that first acts like a depthwise convolutional layer, then applies a 1×1 convolutional layer to the resulting feature maps. This makes it possible to apply filters to arbitrary sets of inputs channels.

Exercises

1. What are the advantages of a CNN over a fully connected DNN for image classification?
2. Consider a CNN composed of three convolutional layers, each with 3×3 kernels, a stride of 2, and SAME padding. The lowest layer outputs 100 feature maps, the

15 This name is quite misleading since this layer does *not* perform a deconvolution, which is a well-defined mathematical operation (the inverse of a convolution).

middle one outputs 200, and the top one outputs 400. The input images are RGB images of 200 × 300 pixels. What is the total number of parameters in the CNN? If we are using 32-bit floats, at least how much RAM will this network require when making a prediction for a single instance? What about when training on a mini-batch of 50 images?

3. If your GPU runs out of memory while training a CNN, what are five things you could try to solve the problem?
4. Why would you want to add a max pooling layer rather than a convolutional layer with the same stride?
5. When would you want to add a *local response normalization* layer?
6. Can you name the main innovations in AlexNet, compared to LeNet-5? What about the main innovations in GoogLeNet and ResNet?
7. Build your own CNN and try to achieve the highest possible accuracy on MNIST.
8. Classifying large images using Inception v3.
   a. Download some images of various animals. Load them in Python, for example using the `matplotlib.image.mpimg.imread()` function. Resize and/or crop them to 299 × 299 pixels, and ensure that they have just three channels (RGB), with no transparency channel.
   b. Download the latest pretrained Inception v3 model: the checkpoint is available at *https://goo.gl/nxSQvl*.
   c. Create the Inception v3 model by calling the `inception_v3()` function, as shown below. This must be done within an argument scope created by the `inception_v3_arg_scope()` function. Also, you must set `is_training=False` and `num_classes=1001` like so:

```
from tensorflow.contrib.slim.nets import inception
import tensorflow.contrib.slim as slim

X = tf.placeholder(tf.float32, shape=[None, 299, 299, 3])
with slim.arg_scope(inception.inception_v3_arg_scope()):
    logits, end_points = inception.inception_v3(
                             X, num_classes=1001, is_training=False)
predictions = end_points["Predictions"]
saver = tf.train.Saver()
```

   d. Open a session and use the `Saver` to restore the pretrained model checkpoint you downloaded earlier.
   e. Run the model to classify the images you prepared. Display the top five predictions for each image, along with the estimated probability (the list of class names is available at *https://goo.gl/brXRtZ*). How accurate is the model?
9. Transfer learning for large image classification.

a. Create a training set containing at least 100 images per class. For example, you could classify your own pictures based on the location (beach, mountain, city, etc.), or alternatively you can just use an existing dataset, such as the flowers dataset or MIT's places dataset (requires registration, and it is huge).

b. Write a preprocessing step that will resize and crop the image to 299 × 299, with some randomness for data augmentation.

c. Using the pretrained Inception v3 model from the previous exercise, freeze all layers up to the bottleneck layer (i.e., the last layer before the output layer), and replace the output layer with the appropriate number of outputs for your new classification task (e.g., the flowers dataset has five mutually exclusive classes so the output layer must have five neurons and use the softmax activation function).

d. Split your dataset into a training set and a test set. Train the model on the training set and evaluate it on the test set.

10. Go through TensorFlow's DeepDream tutorial. It is a fun way to familiarize yourself with various ways of visualizing the patterns learned by a CNN, and to generate art using Deep Learning.

Solutions to these exercises are available in Appendix A.

CHAPTER 14
Recurrent Neural Networks

The batter hits the ball. You immediately start running, anticipating the ball's trajectory. You track it and adapt your movements, and finally catch it (under a thunder of applause). Predicting the future is what you do all the time, whether you are finishing a friend's sentence or anticipating the smell of coffee at breakfast. In this chapter, we are going to discuss *recurrent neural networks* (RNN), a class of nets that can predict the future (well, up to a point, of course). They can analyze *time series* data such as stock prices, and tell you when to buy or sell. In autonomous driving systems, they can anticipate car trajectories and help avoid accidents. More generally, they can work on *sequences* of arbitrary lengths, rather than on fixed-sized inputs like all the nets we have discussed so far. For example, they can take sentences, documents, or audio samples as input, making them extremely useful for natural language processing (NLP) systems such as automatic translation, speech-to-text, or *sentiment analysis* (e.g., reading movie reviews and extracting the rater's feeling about the movie).

Moreover, RNNs' ability to anticipate also makes them capable of surprising creativity. You can ask them to predict which are the most likely next notes in a melody, then randomly pick one of these notes and play it. Then ask the net for the next most likely notes, play it, and repeat the process again and again. Before you know it, your net will compose a melody such as the one produced by Google's Magenta project. Similarly, RNNs can generate sentences, image captions, and much more. The result is not exactly Shakespeare or Mozart yet, but who knows what they will produce a few years from now?

In this chapter, we will look at the fundamental concepts underlying RNNs, the main problem they face (namely, vanishing/exploding gradients, discussed in Chapter 11), and the solutions widely used to fight it: LSTM and GRU cells. Along the way, as always, we will show how to implement RNNs using TensorFlow. Finally, we will take a look at the architecture of a machine translation system.