With millions of parameters you can fit the whole zoo. In this section we will present some of the most popular regularization techniques for neural networks, and how to implement them with TensorFlow: early stopping, ℓ_1 and ℓ_2 regularization, dropout, max-norm regularization, and data augmentation.

Early Stopping

To avoid overfitting the training set, a great solution is early stopping (introduced in Chapter 4): just interrupt training when its performance on the validation set starts dropping.

One way to implement this with TensorFlow is to evaluate the model on a validation set at regular intervals (e.g., every 50 steps), and save a "winner" snapshot if it outperforms previous "winner" snapshots. Count the number of steps since the last "winner" snapshot was saved, and interrupt training when this number reaches some limit (e.g., 2,000 steps). Then restore the last "winner" snapshot.

Although early stopping works very well in practice, you can usually get much higher performance out of your network by combining it with other regularization techniques.

ℓ_1 and ℓ_2 Regularization

Just like you did in Chapter 4 for simple linear models, you can use ℓ_1 and ℓ_2 regularization to constrain a neural network's connection weights (but typically not its biases).

One way to do this using TensorFlow is to simply add the appropriate regularization terms to your cost function. For example, assuming you have just one hidden layer with weights `weights1` and one output layer with weights `weights2`, then you can apply ℓ_1 regularization like this:

```
[...] # construct the neural network
base_loss = tf.reduce_mean(xentropy, name="avg_xentropy")
reg_losses = tf.reduce_sum(tf.abs(weights1)) + tf.reduce_sum(tf.abs(weights2))
loss = tf.add(base_loss, scale * reg_losses, name="loss")
```

However, if there are many layers, this approach is not very convenient. Fortunately, TensorFlow provides a better option. Many functions that create variables (such as `get_variable()` or `fully_connected()`) accept a `*_regularizer` argument for each created variable (e.g., `weights_regularizer`). You can pass any function that takes weights as an argument and returns the corresponding regularization loss. The `l1_regularizer()`, `l2_regularizer()`, and `l1_l2_regularizer()` functions return such functions. The following code puts all this together:

```
with arg_scope(
        [fully_connected],
```

```
    weights_regularizer=tf.contrib.layers.l1_regularizer(scale=0.01)):
hidden1 = fully_connected(X, n_hidden1, scope="hidden1")
hidden2 = fully_connected(hidden1, n_hidden2, scope="hidden2")
logits = fully_connected(hidden2, n_outputs, activation_fn=None,scope="out")
```

This code creates a neural network with two hidden layers and one output layer, and it also creates nodes in the graph to compute the ℓ_1 regularization loss corresponding to each layer's weights. TensorFlow automatically adds these nodes to a special collection containing all the regularization losses. You just need to add these regularization losses to your overall loss, like this:

```
reg_losses = tf.get_collection(tf.GraphKeys.REGULARIZATION_LOSSES)
loss = tf.add_n([base_loss] + reg_losses, name="loss")
```

Don't forget to add the regularization losses to your overall loss, or else they will simply be ignored.

Dropout

The most popular regularization technique for deep neural networks is arguably *dropout*. It was proposed[20] by G. E. Hinton in 2012 and further detailed in a paper[21] by Nitish Srivastava et al., and it has proven to be highly successful: even the state-of-the-art neural networks got a 1–2% accuracy boost simply by adding dropout. This may not sound like a lot, but when a model already has 95% accuracy, getting a 2% accuracy boost means dropping the error rate by almost 40% (going from 5% error to roughly 3%).

It is a fairly simple algorithm: at every training step, every neuron (including the input neurons but excluding the output neurons) has a probability p of being temporarily "dropped out," meaning it will be entirely ignored during this training step, but it may be active during the next step (see Figure 11-9). The hyperparameter p is called the *dropout rate*, and it is typically set to 50%. After training, neurons don't get dropped anymore. And that's all (except for a technical detail we will discuss momentarily).

20 "Improving neural networks by preventing co-adaptation of feature detectors," G. Hinton et al. (2012).

21 "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," N. Srivastava et al. (2014).