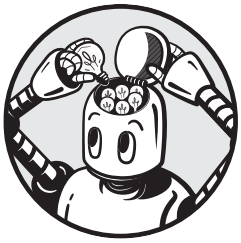
B3

KERAS PART 2



This chapter is a bonus chapter for my book *Deep Learning: A Visual Approach*. You can order the book from No Starch Press at *https://nostarch.com/deep-learning-visual-approach/*. The official version of this chapter can be found for free on my GitHub at *https://github.com/blueberrymusic/* (look for the repository "Deep-Learning-A-Visual-Approach"). All of the figures in this chapter, and all of the notebooks with complete, running implementations of the code discussed here, can also be found for free on the book's GitHub repository.

In Bonus Chapter 2 we introduced the Keras library and looked at how to build and train basic models.

Now we'll expand our horizons. We'll see how to improve our models, incorporate search routines from the scikit-learn library (discussed in Bonus Chapter 1), and build more complex models such as CNNs and RNNs.

Improving the Model

This section's notebook is Bonus03-Keras-1-Improving-the-Model.ipynb.

In Bonus Chapter 2, we explored a lot of the features Keras offers using a tiny, 2-layer model. As we saw, after only about 20 epochs of training this model was able to accurately classify about 98% of the images in the MNIST test set.

Let's see if we can improve that. How might we build a better model?

The answer is not obvious. It's made more difficult by the sheer number of choices that we can try out. Even in this extremely simple model with one hidden dense layer, we've made many choices, all of which influence how well and how fast our network learns.

Though sometimes a change to our model can bring about a big improvement in accuracy, much of the time improving a model's performance is a game of accumulating a sequence of tiny improvements.

Counting Up Hyperparameters

Before we start modifying the hyperparameters of our model, let's get a clearer picture of just how many choices we've made. Note that we've not counting up the weights, which aren't under our control. We're just looking at all the places where we could make a different decision in the design of our model.

Many of the routines we've used take multiple optional arguments that we've ignored. In a sense, we've chosen values for those arguments by letting them stay at their defaults. So let's count those, too.

Each of our two `Dense` layers took 2 arguments: the number of neurons, and the activation function. Consulting the Keras documentation, there are at least 7 more arguments that we could reasonably experiment with.

Then there's the choices we made when we compiled the model. We chose a loss function and an optimizer, giving us 2 more options to adjust.

Given that we chose the `adam` optimizer, there are 5 optional arguments that we can use to tune its behavior.

Finally, we supply a host of options when we call `fit()` to train the model. We have choices for the batch size and the number of epochs (we could argue that the number of epochs doesn't matter if we use early stopping, but then we'd have to set a value for that algorithm's patience). So we have at least 2 arguments here.

So in this casual tour of our choices, we've got 9 layer-level choices on each of 2 layers for a total of 18 choices, 2 choices when we compile, 5 more choices for our optimizer, and at least 2 choices when we fit. That's a total of 27 hyperparameters for this tiny model.

The number goes up fast as we add more layers.

Figuring out what changes to these choices will make the model better is a daunting task. Imagine sitting at a control panel with 27 sliders, switches, and knobs. This is just to control basic performance, and doesn't include controls for additional options like adjusting the learning rate schedule.

We might set the controls, push the big red button to train the network, wait for a while, and eventually look at the numbers that report how we did.

Then we could adjust one or more controls in the hopes of making things better, and repeat.

Complicating the problem is that many of the hyperparameters interact. So if we increase one value, we might only see an improvement if we simultaneously decrease two or three other values, and increase one or two others.

Things get even harder when we want to improve larger and deeper models with dozens of layers. The number of choices and their possible settings becomes enormous.

This is why we've gone through so many chapters of information to get here. The only chance we have of improving our model's performance is to draw from our knowledge about what the network is doing, and why, and what all of our choices do. When we understand what's happening inside, we have a fighting chance of learning from our experience and developing the intuition and hunches that are essential to building great deep-learning networks.

Though we almost always have to run experiments and see what happens, our knowledge and experience improve our chances of making things better.

Changing One Hyperparameter

A frequent rule in experimentation of all sorts is to change only one thing at a time and see what happens. This is a good plan if the values involved are largely *decoupled*, meaning that they don't affect one another. As an analogy, suppose we're adjusting the sound of our car radio, and boosting or cutting the highs and lows. The results of these choices combine, but they're independent: generally speaking, adding more treble doesn't change how much bass sound is delivered, and vice-versa.

Unfortunately, the hyperparameters of most real systems, and most deep-learning systems, are not decoupled. If we increase the amount of hyperparameter A and find things get better, and then increase the amount of hyperparameter B, we may find that we now have to *decrease* the value in A to make further progress. The connections are complex.

But still, changing one hyperparameter at a time is usually a good way to start. We can explore what that value does, find a good value for it, and then choose another hyperparameter to adjust, and so on, searching for a good combination by fine-tuning one hyperparameter at a time. If we have to go back, then our experience with each value can help guide us to select which one to adjust again, and by how much. We can also build up a sense of which values are related to which others, so we can anticipate their interactions.

Let's try that now, arbitrarily picking the batch size as our first hyperparameter to experiment with. We said above that when we're using a GPU we pick a batch size that best fits our particular hardware. But on a CPU we can pick almost any value we like. We've been using a batch size of 256, but like most of our initial choices for each of the 27 hyperparameters we just counted up, it was really just a shot in the dark. Let's try cranking that up and down and see what happens, if anything.

Figures B3-1 through B3-4 show the results of setting this hyperparameter to 2048, 512, 64, and finally 8. We used the same code for every run,

changing only the batch size. Note that the vertical scale on the graphs is not the same from one graph to the next. This allows us to show all the data, though it means we can't compare them equally at a glance.

Three things jump out from these figures.

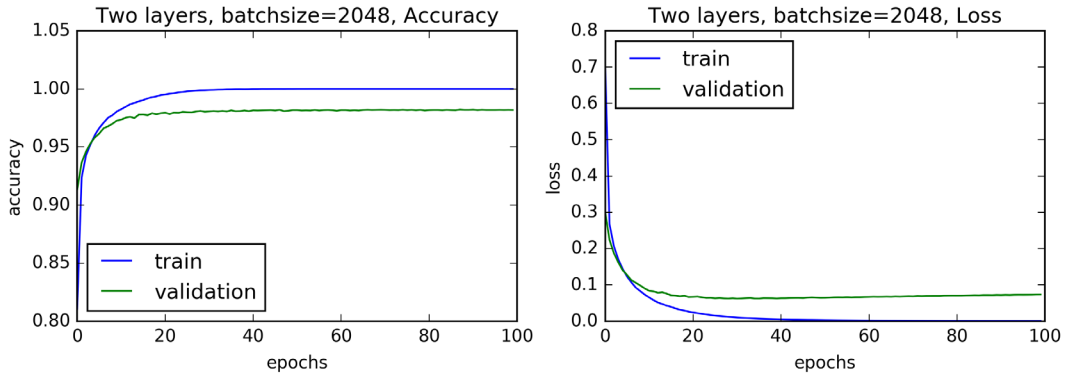


Figure B3-1: Training our two-layer model with a batch size of 2048

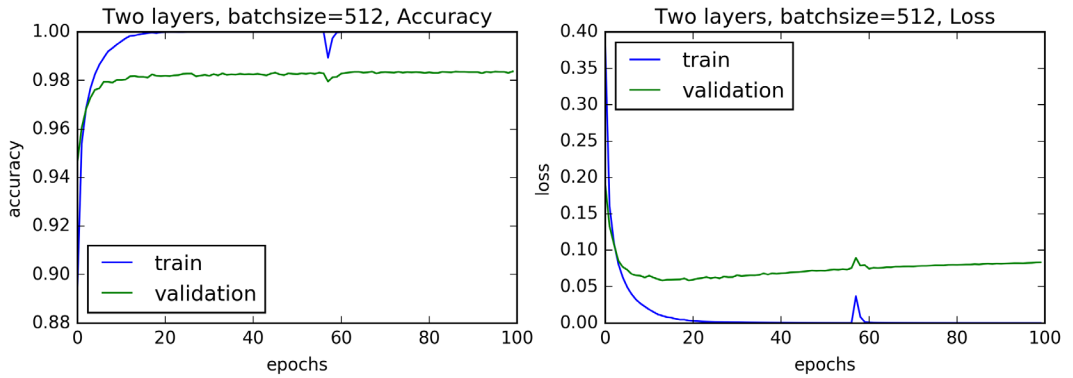


Figure B3-2: Training our two-layer model with a batch size of 512

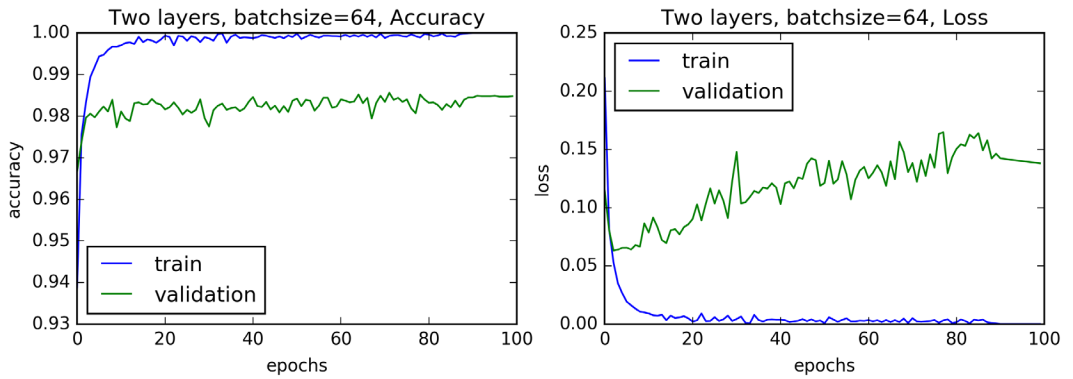


Figure B3-3: Training our two-layer model with a batch size of 64

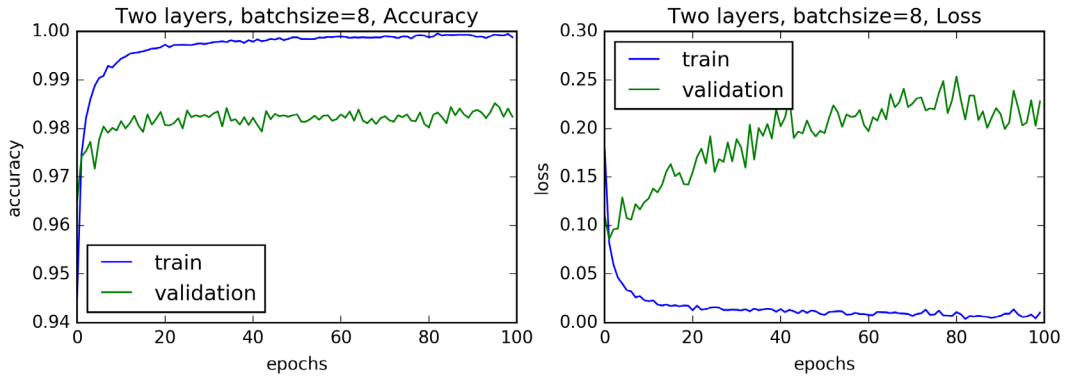


Figure B3-4: Training our two-layer model with a batch size of 8

First, as the batch size gets smaller, the results get more jittery, or noisy. This is because each new update is working with fewer samples, so it's responding to whatever happens to be in that batch. Larger batches tend to become more representative of the dataset as a whole, and give us smoother results. Smaller batches give us a lot of jumping around.

The second thing is that the training accuracy is about 98% on all the models, so the batch size didn't affect that accuracy very much.

The third thing is that although all of the models are overfitting, as demonstrated by the diverging training and validation losses, as the batch size gets smaller the divergence of the training and validation error increases. In other words, the amount of overfitting increases.

Smaller batches mean that epochs take longer, because we need to perform backprop and update the weights more frequently. Figure B3-5 shows the clock time, in seconds, for each of the above batch sizes, plus the other powers of 2 between them.

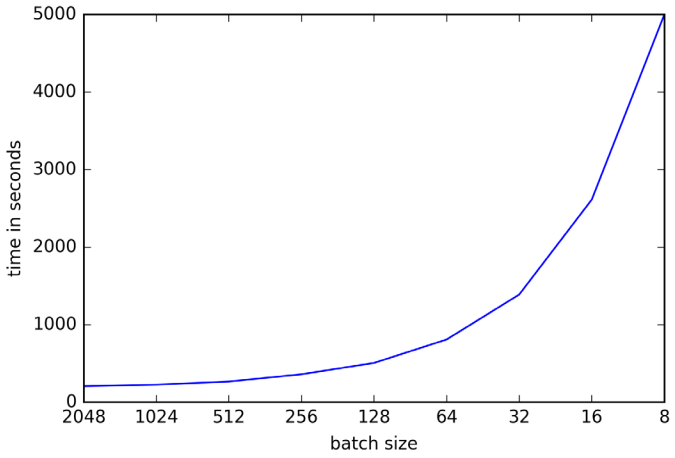


Figure B3-5: The timing results (running time in seconds, on a late 2014 iMac) taken by the experiments whose data are shown in Figures B3-1 through B3-4, as well as other intermediate batch sizes. Note that the vertical scale is linear while the horizontal scale is not.

The curve in Figure B3-5 confirms that on these CPU-only runs, as the batch size went down, we're running more backprop and update steps, so the total training time went up.

The above experiments tell us a lot about how the batch size affects training for this dataset on this model. They suggest that for this model and data, large batch sizes are more desirable than small ones.

Other Ways to Improve

When we seek to improve a model, it can help to keep in mind that we'll be unlikely to find *the very best* set of parameters for the training speed and accuracy that we're after. Instead, we look for parameters that come close enough.

It also helps to have one goal in mind at every step in the search. We might be looking to reduce overfitting, or drive down the test loss, or increase the test accuracy, or speed up training time, or fit best onto the GPU, or use the least computer memory, and so on. We're unlikely to be able to improve all of these at once.

So we typically pick out just one or two things to improve, and then modify some of our variables until they're as good as we can get them. Then we move on to another group of criteria, and look for the variables that will help with those, and so on.

For the MNIST problem, let's aim to improve accuracy while reducing overfitting.

Rather than continue to adjust hyperparameters, let's try something radically different: adding a second `Dense` layer.

In order to keep everything comparable with the models earlier in this chapter, we'll return to a batch size of 256, and leave out early stopping.

Adding Another Dense Layer

Let's add a second dense hidden layer, just as big as the first. After all, having more neurons means more ability to learn, right?

Not really. We've seen that our single-layer model is already too capable of learning the idiosyncrasies in the training data. That's why it's overfitting. If we throw in yet more neurons without making any other structural changes, then this overfitting should get worse, faster.

Let's try it and see.

Figure B3-6 shows the architecture for a model with two dense hidden layers, each arbitrarily given as many neurons as there are input elements (that is, each layer has 784 neurons), followed by a 10-neuron output layer with softmax on the output.

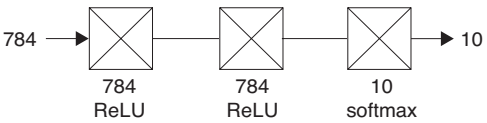


Figure B3-6: The architecture of our three-layer model. Each of the hidden layers is a dense layer with 784 neurons, one for each input. Though our convention is that dense layers have a ReLU activation function by default, for completeness we're listing it here explicitly.

We can make this model in Keras by adding just one line to our model-making routine, creating the second dense hidden layer. This line looks just like the one above it except that we don't include the `input_shape` argument, since that's only used by the first layer in the model. Listing B3-1 shows the code.

```
def make_two_hidden_layers_model():
    model = Sequential()
    model.add(Dense(number_of_pixels, input_shape=[number_of_pixels],
                    activation='relu'))
    model.add(Dense(number_of_pixels, activation='relu'))  # new layer
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model
```

Listing B3-1: Create and compile the network of Figure B3-6, with two identical `Dense` layers in a row.

Figure B3-7 shows the accuracy of the training and validation sets over 100 epochs of learning (we did not use early stopping).

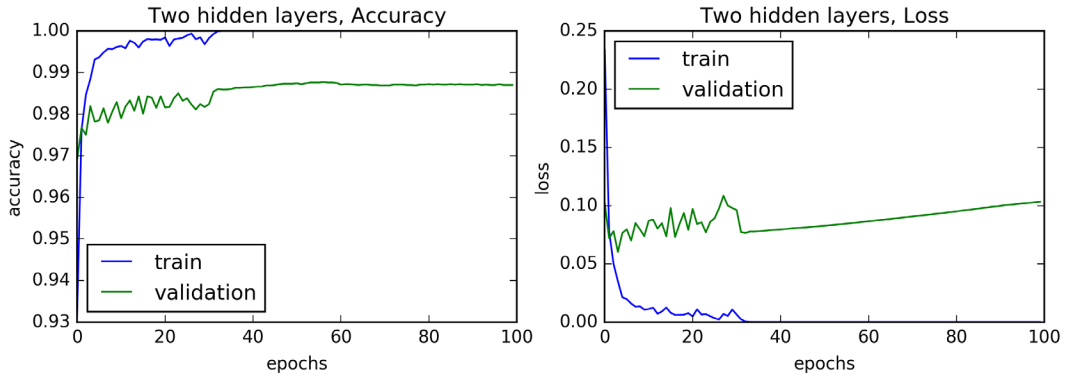


Figure B3-7: The accuracy and loss of our two-layer network plotted against the number of epochs

Compared to our previous results, the new curves are wigglier in the starting epochs, suggesting that the bigger network took more time to settle down. And just as we suspected, the system overfit the training data just like before, but it did so even more quickly, driving up the validation loss faster than before.

So just throwing more neurons at the problem did not make everything better. Validation accuracy improved a touch, but the loss is looking much worse, and we're still overfitting considerably.

Less Is More

Having too many neurons has made our network *too capable*. It had more than enough power for this task, so it used its extra abilities to extract more and more idiosyncratic detail from the training set, and thus overfit.

We generally want the smallest, simplest network that will get us the results we're after. A simpler network not only trains and predicts faster, but it's less prone to overfitting because there's less superfluous computational power to get distracted by irrelevant details in the training data.

Let's go back to our single dense layer, but make it far smaller, with only 64 neurons. This gives us roughly one neuron for every 12 input pixels. The new architecture is shown in Figure B3-8.

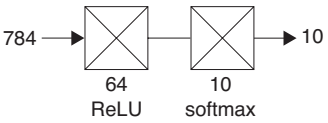


Figure B3-8: A new two-layer network where we'll only use 64 neurons in the first, fully-connected hidden layer

We'll just change the line that defines this layer to give it 64 neurons rather than 784. Listing B3-2 shows the change.

```
def make_smaller_one_hidden_layer_model():
    model = Sequential()
    model.add(Dense(64, input_shape=[number_of_pixels],
                    activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model
```

Listing B3-2: Building a model where the first (and only) hidden layer has just 64 neurons

The accuracy and loss results for 100 epochs are shown in Figure B3-9.

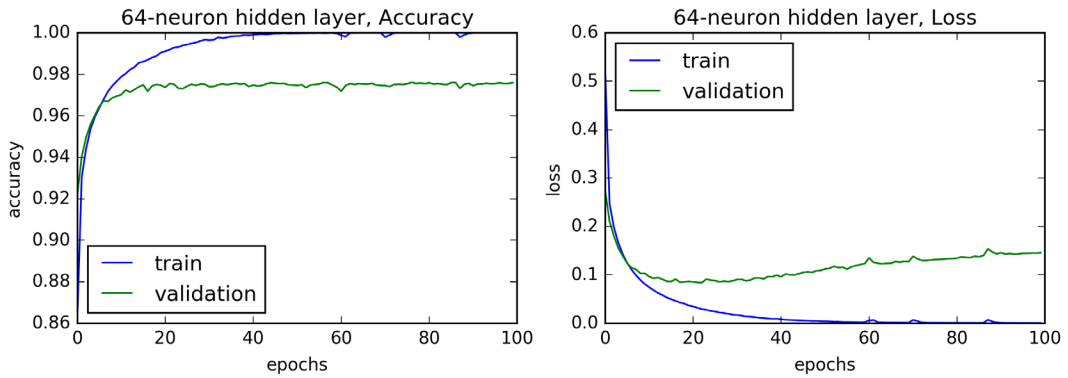


Figure B3-9: The accuracy of our model of Figure B3-8

Our network is giving us a bit less accuracy than the one with 784 neurons in the first layer, but even so, it's still overfitting. What to do? Let's try using our idea from the last section and use two hidden layers instead of one, but we'll keep the same number of neurons and split them evenly. In other words, we'll have two hidden layers of 32 neurons each, as shown in Figure B3-10.

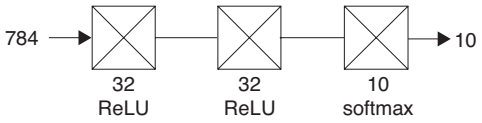


Figure B3-10: A deeper model with three layers. We're splitting up the 64-neuron hidden layer of our previous model into two separate, fully-connected 32-neuron hidden layers.

The accuracy and loss results for 100 epochs of training are in Figure B3-11.

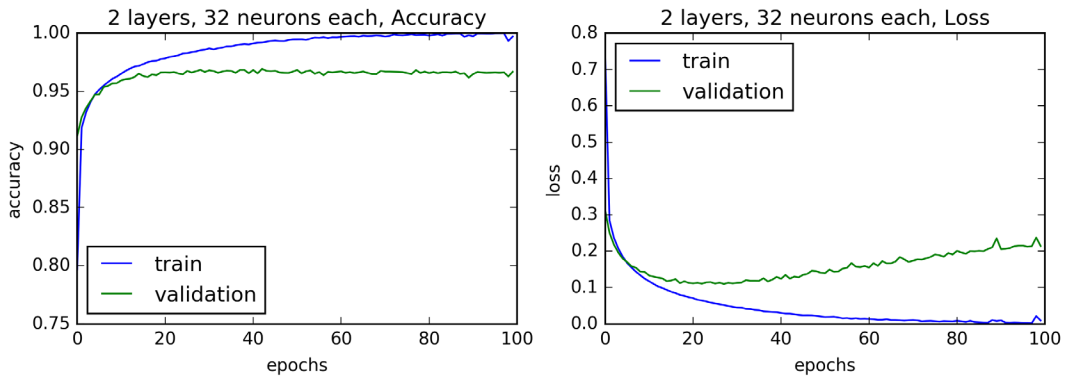


Figure B3-11: The accuracy of the model in Figure B3-10

The validation accuracy after 100 epochs has decreased a little bit from about 97.5% for our single 64-neuron layer to about 97% for our new, two-layer network. The loss has increased, too, and we seem to be overfitting even more rapidly than before.

We've taken a step backwards. A small step, granted, but the measurements are worse and we're still overfitting.

Though chopping up layers into multiple, smaller pieces can sometimes work, it didn't help us much in this example. This is often the way it goes: we try one thing and another, following up on ideas that work and setting aside those that don't make things better.

Before we give up on these two small layers, let's see if we can try another trick to get the overfitting under control.

Adding Dropout

Since overfitting is a problem for this model and data, let's try using *dropout*. As we discussed in Chapter 15, this is a regularization technique explicitly

designed to address overfitting. Dropout temporarily removes a random selection of neurons before each epoch, and puts them back in at the end. The intuition is that our neurons will be less likely to specialize (and potentially over-specialize), since they all need to be able to compensate for randomly missing neurons.

To apply dropout in Keras, we create a new *dropout layer* and add it to the growing stack, just after the layer we want to affect. When dropout is applied, randomly-chosen neurons are isolated from the network for one epoch, so they don't contribute to predictions, and they don't learn when the network's weights are updated. When the epoch is done, the neurons are restored, and before the next epoch, a new random collection gets disconnected. This process occurs only during training.

It might seem a little weird that dropout is included as a layer. It doesn't have any neurons, and it doesn't participate in backprop or computation, so how can it be a layer? Calling this a layer is really just a conceptual device. We'd like to apply dropout to not just `Dense` layers, but other types of layers, like the convolution and recurrent layers we'll cover later in this chapter. Rather than build dropout into each layer, Keras lets us specify this kind of "supplemental" layer that doesn't do any computing, but tells Keras about something we want it to do. Thinking of operations like dropout as implemented by their own layers lets us keep our conceptual view of our model simple and clean. We just have a big stack of layers. Some layers have neurons, and others perform operations on other layers or on data.

In this case, the dropout layer says to Keras, "apply dropout to the preceding layer." If there are, say, 3 `Dense` layers preceding this dropout layer, only the most recent one is affected. If we wanted to apply dropout to all three `Dense` layers, we'd have to follow each one individually with its own dropout layer.

The dropout layer in Keras takes only one parameter, and it's mandatory. It's a floating-point number between 0 and 1 that describes the percentage of neurons that will be temporarily removed after each batch. A value of 0 disables dropout, while a value of 1 would make the preceding layer effectively disappear. The authors of the original paper on dropout advise a value of 0.2, and that's generally a good place to start [Srivastava14].

The authors also advise constraining the magnitude of the weights on the dense layers that are affected by dropout. Speaking generally, the concern is that when some nodes are removed, the others might overcompensate by cranking their weights up very high. Without getting into the math, we can take their advice by setting an optional parameter on the `Dense` layer that's going to experience dropout. The parameter is called `kernel_constraint`, and the advice of the authors of the paper cited above is to set that to the value 3, so we'll do just that. We only need to add this option to `Dense` layers that will have dropout applied to them, as we'll see in a code listing just below.

The complete model specification for our two-layer model, with dropout, is shown in Figure B3-12. Here we're applying dropout to both of our two hidden layers.

In Figure B3-12 we show our schematic symbol for a dropout layer, which is a slanted line crossing the line carrying data, suggesting that some of the data is being struck out, or removed.

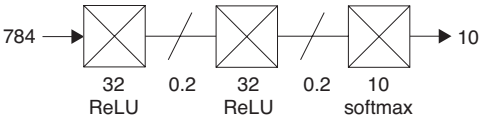


Figure B3-12: We'll change our model in Figure B3-10 to add dropout layers after each 32-neuron, fully-connected, hidden layer. Here we're using our symbol for dropout: a diagonal slash through the line connecting two layers. Each dropout layer applies to the layer preceding it.

The code for making this model is in Listing B3-3. There are a few new things happening in this code.

```
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm

def two_layers_with_dropout_model():
    model = Sequential()
    model.add(Dense(32, input_shape=[number_of_pixels],
              activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(32,
              activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Dense(number_of_classes, activation='softmax'))
    # compile the model to turn it from specification to code
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=['accuracy'])
    return model

model = two_layers_with_dropout_model()
```

Listing B3-3: Two dense layers of 32 neurons each are both followed by `Dropout` layers. We use a dropout percentage of 0.2. We also set `kernel_constraint` in the `Dense` layers.

First, of course, we're adding dropout layers. The argument 0.2 tells the layer to use the 20% dropout rate suggested by dropout's creators.

As we mentioned above, the original paper on dropout also suggested imposing a technical condition on the weights in the layer that's experiencing the dropout, and that advice is widely followed. In Listing B3-3 we do this by adding the optional argument `kernel_constraint` to the argument list for each layer that will be affected by dropout, and setting that parameter's value to `MaxNorm(3)` (note that we have to import `MaxNorm()` in order to use it). The thinking behind this step, which explains what this `MaxNorm()` thing is doing, is explained in the original paper [Srivastava14]. It's reasonable to just think of it as a mechanism to keep the weight values from getting too big.

Training this model for 100 epochs produces the results in Figure B3-13.

We've conquered over overfitting problem! The losses are no longer diverging. Dropout has done a great job for us.

The accuracy is a bit weird, since we're getting better accuracy on the validation data than the training data. The validation accuracy seems to

have taken a small hit, too, since it's not up to the 98.3% from before. We might be able to tweak our accuracy upwards by reducing the dropout rate a bit, or adding a few more neurons to our dense layers.

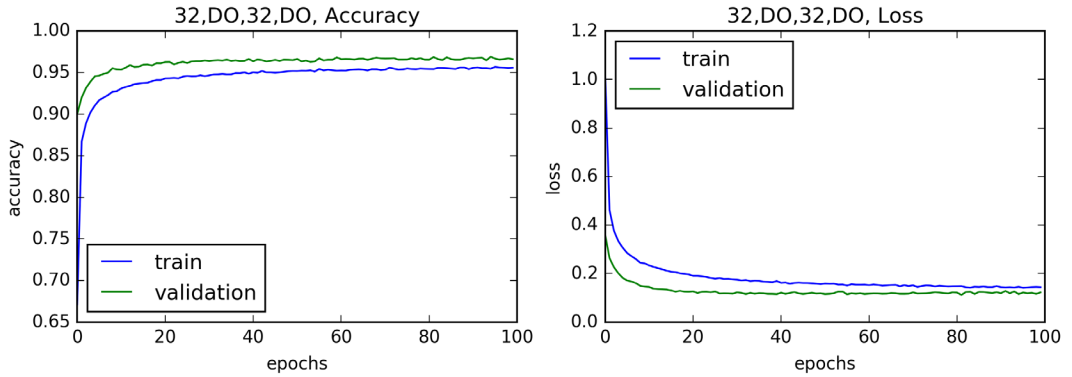


Figure B3-13: Accuracy and loss for the model of Figure B3-12

The dropout paper also recommends that that we configure our optimizer to use a learning rate 10-100 times larger than we normally would. We can tell Adam to start with any specific learning rate by setting its optional argument `lr` (that's a lower-case letter L followed by a lower-case letter R, standing for "learning rate"). This value defaults to 0.001.

To pass this argument to Adam, we have to make an `Adam` object as we did earlier, and pass it our new value to the learning rate parameter. Listing B3-4 shows how we'd set the initial learning rate to 0.1. This would replace the line previously calling `model.compile()`.

```
from tensorflow.keras.optimizers import Adam

# make our own Adam object
adam_optimizer = Adam(lr=0.1)

# optimizer gets our object, rather than a string
model.compile(loss='categorical_crossentropy',
              optimizer=adam_optimizer, metrics=['accuracy'])
```

Listing B3-4: We can provide our own optimizer object when we compile, rather than rely on a default. Here we make an `Adam` with our own choice of learning rate.

A shorter way to write this is shown in Listing B3-5, where we create the `Adam` object and assign it, without needing a temporary variable to hold it.

```
model.compile(loss='categorical_crossentropy',
              optimizer=Adam(lr=0.1),
              metrics=['accuracy'])
```

Listing B3-5: We don't need to store our new `Adam` object in its own variable. This shorter approach is more common.

The surprising results are shown in Figure B3-14.

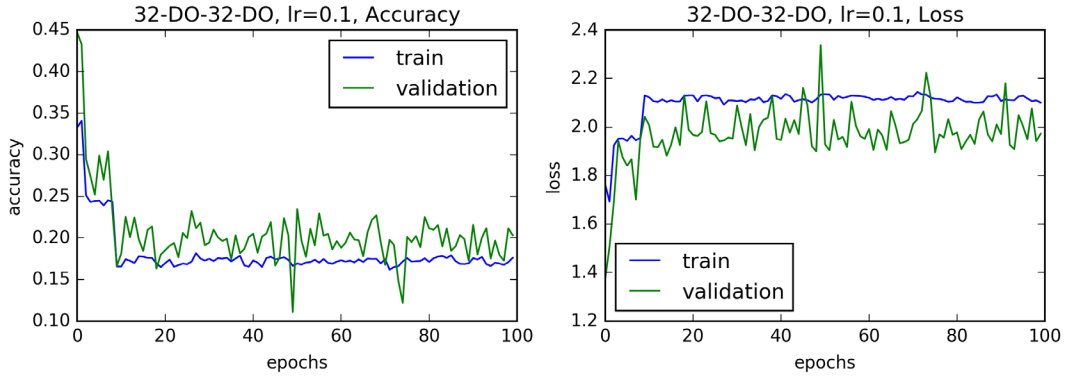


Figure B3-14: The accuracy and loss for our model with dropout when we set Adam's initial learning rate to 0.1

Wow. These graphs are as bad as they look.

For this data and architecture, starting Adam with a learning rate of 0.1 was much too aggressive. The training accuracy plummeted to about 0.18, which is terrible. The validating accuracy seems to be fluttering around 0.2, but it's got a lot of noise. The loss was also terrible, more than 10 times worse than before.

If we drop the learning rate down to 0.01, we get much more encouraging performance, as shown in Figure B3-15.

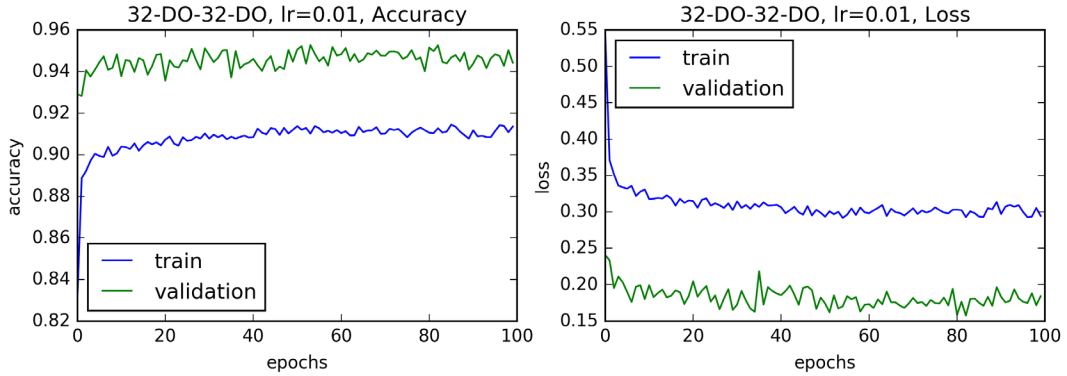


Figure B3-15: Accuracy and loss for our dropout model with Adam's initial learning rate set to 0.01

These results aren't nearly as good as what we got with the default learning rate of 0.001, but it was worth a shot. Things are much calmer, and our accuracies are both above 90%. And we're still not overfitting.

We could try out a variety of learning rates to try to see what value works the best for this model and data, but that would be a lot of typing and waiting.

It would be really nice if we could automate this search, so the computer could try out a variety of learning rates for us while we do other things. In an upcoming section, we'll see how to do just this using tools from scikit-learn.

Observations

We've only begun the process of tuning and refining our model. Tinkering with a practice model like this one in search of the best results is time well spent, since it hones our intuition and can help guide our choices with other, larger, databases and models in the future.

This is one of those times when saying "the exercise is left to the reader" is completely appropriate. There's no substitution for sitting down with a deep learning model and adjusting its structure and hyperparameters to get a feeling for how that model behaves on that data.

When we work with models that we will actually be deploying in the real world, we want the best-performing models we can develop. When models become so big that they can take days (or even weeks) to train, it's important to have a good sense of what's likely to work, since it lets us start much closer to the finish line than just assembling a model at random. Even if we automate the parameter search, we still want to focus our search where it's going to pay off the most.

We found that for this data a single giant fully-connected hidden layer was overkill. It started overfitting almost immediately, and the training accuracy went flat. Worse, the training loss was steadily climbing, which would eventually cause the training accuracy to drop. We were throwing a network with too many computing resources at the problem, and it used those resources to overfit, learning far too much about the idiosyncrasies of the training data even after it had perfect accuracy.

By significantly reducing the size of that layer we got away from overfitting, but our accuracy dropped.

Then by splitting that single layer into two pieces, and adding dropout, we got performance up, and stopped overfitting.

There are many other things left to try. Using more layers is always something to consider. Perhaps each layer could be smaller than the one before it, forcing the system to look for larger patterns. Or perhaps we could have one small "choke" layer between two larger ones. We might try applying dropout only on some of the layers, or applying it more aggressively (that is, raising the number of neurons we're suppressing).

Using Scikit-Learn

This section's notebook is Bonus03-Keras-2-scikit-learn.ipynb.

So far we've been searching our hyperparameters by hand. It's been illuminating, but it also required a lot of manual effort.

We saw in Bonus Chapter 1 that the scikit-learn library offers us routines to cross-validate our model (to estimate how good it is), and grid-search its hyperparameters (to find the best-performing combination).

Keras doesn't offer either of these tools directly, because it offers a way to use the ones already in scikit-learn.

Let's pause a moment to think about what we might be asking for from these tools. If a model takes 3 hours to train, then running cross-validation with 10 folds will take about 10 times longer, or 30 hours. If we're

grid-searching over, say, three hyperparameters with 5 values each (which is not a very large search), then it will take 125 times longer than that, or more than five months!

Is there any way to cut this down?

A popular approach is to extract a tiny piece of the data set, carefully selected to be representative of the whole, and search on that. Then each training run will be much faster.

By cross-validating and grid-searching one or more of these little proxy databases, we can get some guidance for what models and hyperparameters are worth exploring on a larger scale. Then we can take that knowledge and work with larger and larger pieces of the dataset, tuning the hyperparameters at each step. The hope is that by the time we reach the full database, we'll have a great set of hyperparameters to train on and we'll need only a little searching, or perhaps even none at all.

Keras Wrappers

It would be nice to use scikit-learn's cross-validation and grid-search tools directly on our Keras models. Happily, Keras "knows" about scikit-learn, and how to communicate with it.

In particular, Keras knows how scikit-learn expects its estimators to behave. With that, Keras can dress up one of its models to act like a scikit-learn estimator, bridging the gap.

This act of camouflage lets us place a Keras model into scikit-learn, and then do cross-validation, grid search, or any other operation we like. From scikit-learn's perspective, this object is just some custom estimator that we wrote and gave to it. It doesn't know that there's a deep network hiding inside.

We pull off this trick by embedding our Keras model in an object of type `KerasClassifier` or `KerasRegressor`, depending on the job it does. These objects are called *wrappers*, since they "wrap" our Keras model in a disguise that makes it look and act like a scikit-learn estimator. We don't have to modify our network in any way to wrap it. We just make a wrapper object, place our network inside, and we're done.

Since both wrappers work identically, we'll choose `KerasClassifier` as an example so we can stick with the MNIST classifiers we've been discussing so far.

We don't actually hand our model to the wrapper function. Instead, we hand it the name of a function that builds the model and returns it. This makes sense when we think about it. The searching process, for instance, may create many versions of our model with different hyperparameters. If we gave it a built and compiled model, there wouldn't be any way for it to make different versions. By giving it a function that builds the model, the searching program can call the function, and when doing so, to set various parameters as it desires.

The other arguments to the wrapper creator are arguments that get passed on. Some are given to the model-making function, and others get passed to scikit-learn.

Let's dig in, starting with the model-making function.

This argument is named `build_fn`, short for "build function." Its value is a function that we've written which will construct, compile, and return a Keras model, just like we've been seeing in listings like Listing B3-3, where the function `two_layers_with_dropout_model()` made a model, compiled it, and returned it.

There are some advanced options, but usually we assign this argument the name of a function in our code, such as `two_layers_with_dropout_model`. Note that we leave off the parentheses, since we're not calling the function, but only providing its name. Often we'd like to parameterize this function with arguments. For instance, we might be searching for the best number of neurons to use in the first two layers. So we make those numbers arguments that we use when we make the model.

As we said before, this model making-function will be called automatically by scikit-learn when the model is required. When we're grid searching, the model will usually be built over and over again at the start of each new step of the search.

So we have our model-making function that takes arguments, and a wrapper, and scikit-learn which is going to call our function. How do we get scikit-learn to include the arguments we want when it calls the model-making function?

Happily, the mechanism is easy. The trick is in the naming of our arguments. Recall from Bonus Chapter 1 that when we create a search using scikit-learn, we provide it with a dictionary that names each parameter we want it to search on as a key, with values to be tried as the value. Python could then match up those dictionary names with the names of parameters in the functions it called.

In the case of a wrapper, things are even easier. Thanks to Python's ability to "know" what the parameter names are in functions, we don't even need the dictionary. We can just name the parameters we want to assign values to, along with the values we want them to have.

For instance, if our model-making function takes a parameter to control the number of neurons it makes, perhaps called `number_of_neurons`, then we can place that into the wrapper's argument list with a value, just as if we were assigning a value to it when calling the function. Any arguments in our model-making function whose names match arguments in the wrapper-making step will be given the assigned values.

To see this in action, let's start with a model-making function that takes parameters. Listing B3-6 shows an example, building on our previous listings that imported and processed the MNIST data.

```
def make_model(number_of_layers=2, neurons_per_layer=32,
               dropout_ratio=0.2, optimizer='adam'):
    model = Sequential()

    # first layer is special, because it sets input_shape
    model.add(Dense(neurons_per_layer, input_shape=[number_of_pixels],
        activation='relu', kernel_constraint=MaxNorm(3)))
```

```
    model.add(Dropout(dropout_ratio))

    # now add in all the rest of the dense-dropout layers
    for i in range(number_of_layers-1):
        model.add(Dense(neurons_per_layer,
                        activation='relu', kernel_constraint=MaxNorm(3)))
        model.add(Dropout(dropout_ratio))

    # finish up with a softmax layer with 10 outputs
    model.add(Dense(number_of_classes,  activation='softmax'))

    # compile the model and return it
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer, metrics=['accuracy'])
    return model
```

Listing B3-6: Our model-making function `make_model()` takes four optional parameters. Two are integers, one is a float, and one is a string. It creates as many dense layers as we request, appending a dropout layer after each one.

Listing B3-7 shows how to pass parameters to our new `make_model()` function which takes arguments. There are ways to make this code smaller (such as by using Python's `*kwargs` technique), but we've chosen clarity over conciseness.

```
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier

kc_model = KerasClassifier(build_fn=make_model,
                           # parameters for the model-making function
                           number_of_layers=2, neurons_per_layer=32,
                           optimizer = 'adam',
                           # parameters for scikit-learn
                           epochs=100, batch_size=256, verbose=0)
```

Listing B3-7: We wrap up `make_model()` in a `KerasClassifier` along with defaults for all the variables we need to create the model and control scikit-learn's training process. When scikit-learn calls `make_model()`, it will assign the parameters of that function the values we provided when we made our `KerasClassifier`.

In effect, the wrapper just takes the values we provide to it and passes them to the model-making function arguments of the same name. The syntax is a little confusing because it looks like `KerasClassifier` is taking these arguments for itself, but it works because Python allows this kind of clever use of parameters.

It's a common convention to assign the wrapped-up object to a variable called `model`, but that can be confused with the more typical use of `model` to mean a normal, or unwrapped, neural network. To emphasize that this is not a straightforward Keras model, we're calling it `kc_model` for "Keras classifier model." This way we can speak of the wrapped object (`kc_model`) and the model that gets created using its `build_fn` function, which we'll continue to simply call a "model."

If we hand `kc_model` to scikit-learn for cross-validation, `make_model()` will be called and passed the value 2 for `number_of_layers`, the value 32 for

the argument `neurons_per_layer`, and the string `'adam'` for `optimizer`, just as though we'd assigned them ourselves Since we're not giving a value for `dropout_ratio` to `KerasClassifier`, it doesn't assign any value to that parameter, so `make_model()` will use its default value for that argument.

If we use this `kc_model` for grid searching, the searcher can assign its own values to any of these parameters when it calls the function to make the model. Any arguments we don't explicitly re-assign will use the default values specified when we make the wrapper.

The last set of three arguments to `KerasClassifier()` (named `epochs`, `batch_size`, and `verbose`) are not for our model, but are intended for scikit-learn. They get passed to the cross-validator's `fit()` routine to control the training process.

In addition to giving parameters for `fit()`, we can also name parameters that get passed to `predict()`, for use if and when that function gets called.

As long as we keep all of our parameter names distinct, Python will correctly pass the desired value to every function involved in the cross-validation and grid searching.

This is a flexible system. For example, it means that we could tune the value of `batch_size` when searching a grid, or the size of our network by searching trying different values of `number_of_neurons`, or different optimizers by trying different strings for `optimizer_choice`.

The key thing to remember is that the wrapper is basically remembering what values should be used for the arguments in the model-making function, and it will use those by default. It also remembers a few values that get passed on to scikit-learn. As long as the names we're assigning to in the wrapper match the names in the model-making routine, everything will be automatically matched up.

Cross-Validation

Let's use a Keras wrapper to do cross-validation on our recent three-layer deep learning system of Figure B3-12 and Listing 2-3, with two 32-neuron dense layers (each with dropout) and a 10-neuron dense output layer.

Applying cross-validation may seem pointless. After all, we already have an excellent, large testing set. What more are we going to learn from cross-validation that we haven't already seen by using our validation data?

In this case, not much. We should expect the results of cross-validation to be very close to what we saw above.

But having our own high-quality validation set coming along with the data is a luxury we can't always count on. Sometimes there is no validation set. Sometimes we have one but we're not sure it's very good. For instance, consider a new deck of cards. Ignoring any jokers or other cards, one typical arrangement for the new cards is to run from ace of hearts to king of hearts, then ace to king in the suit of clubs, then again for diamonds and then spades. This is called "New Deck Order" [Cain13]. Suppose someone opened up a new deck and took away the bottom 25%, calling it the validation data. This set is definitely not representative of the rest of the deck,

because it contains no red cards of any suit, and the remaining cards have no spades.

Another challenge of validation sets comes when we're working with a small version of an original dataset. If this dataset is small, as we saw in Chapter 8, cross-validation is a great way to evaluate it without making the training set even smaller by making a dedicated validation set.

So although the MNIST data makes a great validation set, we'll proceed as though that isn't the case, so we can see how to approach the problem of evaluating the quality of a trained model in the general case.

We'll first do something simple but incomplete, just to get a feeling for the process. Then we'll add in the missing step.

Cross-validation requires training and then validating our entire model over and over again with slightly different data. We'll be using 10 folds, so each session of the cross-validator will take 10 times longer than the training sessions earlier in this chapter.

Let's get going, since there's almost nothing to it. We'll just make our model and then run cross-validation with scikit-learn as in Bonus Chapter 1.

As we mentioned above, we'll repeat our model of Listing B3-7 and use 2 layers of 32 neurons, each with dropout. We'll stick with the default Adam optimizer, though we could make our own `Adam` object as before and use that here instead.

To make our model, we'll supply our generalized model-making routine `make_model()` in Listing B3-6 with the parameters that make our desired network, as shown in Listing B3-8.

```
kc_model = KerasClassifier(build_fn=make_model,
                           number_of_layers=2, neurons_per_layer=32,
                           optimizer='adam',
                           epochs=100, batch_size=256, verbose=0)
```

Listing B3-8: Placing our model inside of a Keras wrapper

In Listing B3-8 we built a Keras wrapper of type `KerasClassifier` with our new routine `make_model()`. We gave two arguments (`number_of_layers` and `neurons_per_layer`) that matched the arguments in `make_model()`, so they will get passed in when the model is built. We also set the parameters that we want to give to `fit()` when it gets called (`optimizer`, `epochs`, `batch_size`, and `verbose`). Now `kc_model` can be used inside of scikit-learn like any other estimator.

Before we actually do that, there are a couple of loose ends to clean up.

One issue is that scikit-learn's cross-validation function `cross_val_score()` doesn't want the one-hot encoded version of our label data. It wants the original versions that contain lists of integers. It so happens that we've been saving the original labels all this time in their own variables, so we have just what the routine needs. What a coincidence!

The other issue has to do with what data we pass to the cross-validation system. As we've done before, we'll simply pretend that we don't have a

validation set, and treat the training data as if it was our entire dataset. We'll let the cross-validator manage the train-validation split for us.

Now let's get this cross-validation going. There are two tasks to perform. First, we'll make the object that drives the cross-validation process. Let's use our old friend `StratifiedKFold()` from Bonus Chapter 1 with 10 splits. We'll shuffle the data, and we'll set the optional `random_state` variable to the value of `random_seed` that we already have around. That's useful for debugging.

We can use Listing B3-9 to make the `StratifiedKFold` object.

```
from sklearn.model_selection import StratifiedKFold

kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=random_seed)
```

Listing B3-9: Creating the `StratifiedKFold` object that will build the cross-validation training and test sets.

Now we're ready to go. Using the same techniques that we saw in Bonus Chapter 1, we just tell scikit-learn to run the cross-validator and track the scores by calling `cross_val_score()` with our model, our training data and original labels, and our folding object. Listing B3-10 shows the code.

```
from sklearn.model_selection import cross_val_score

results = cross_val_score(kc_model, X_train, original_y_train,
                          cv=kfold, verbose=0)
```

Listing B3-10: Running cross-validation from scikit-learn, using `kc_model`, our Keras model in a wrapper.]

Putting it all together, we get Listing B3-11.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense
from tensorflow.keras.layers import Dropout
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras import backend as tensorflow.keras_backend
from tensorflow.keras.utils import np_utils
from tensorflow.keras.models import load_model
from tensorflow.keras.wrappers.scikit_learn import KerasClassifier
from tensorflow.keras.utils import to_categorical

from sklearn.model_selection import StratifiedKFold
from sklearn.model_selection import cross_val_score
import numpy as np

random_seed = 42
np.random.seed(random_seed)

# load MNIST data and save sizes
(X_train, y_train), (X_test, y_test) = mnist.load_data()
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width
```

```
# convert to floating-point
X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)

# scale data to range [0, 1]
X_train /= 255.0
X_test /= 255.0

# save y_train and y_test for use when cross-validating
original_y_train = y_train
original_y_test = y_test

# replace label data with one-hot encoded versions
number_of_classes = 1 + max(np.append(y_train, y_test))
y_train = to_categorical(y_train, num_classes=number_of_classes)
y_test = to_categorical(y_test, num_classes=number_of_classes)

# reshape samples to 2D grid, one line per image
X_train = X_train.reshape(X_train.shape[0], number_of_pixels)
X_test = X_test.reshape(X_test.shape[0], number_of_pixels)

def make_model(number_of_layers=2, neurons_per_layer=32,
               dropout_ratio=0.2, optimizer='adam'):
    model = Sequential()

    # first layer is special, because it sets input_shape
    model.add(Dense(neurons_per_layer, input_shape=[number_of_pixels],
                    activation='relu', kernel_constraint=MaxNorm(3)))
    model.add(Dropout(dropout_ratio))
    # now add in all the rest of the dense-dropout layers
    for i in range(number_of_layers-1):
        model.add(Dense(neurons_per_layer, activation='relu',
                        kernel_constraint=MaxNorm(3)))
        model.add(Dropout(dropout_ratio))
    # finish up with a softmax layer with 10 outputs
    model.add(Dense(number_of_classes, kernel_initializer='normal',
          activation='softmax'))
    # compile the model and return it
    model.compile(loss='categorical_crossentropy',
                  optimizer=optimizer, metrics=['accuracy'])
    return model

kc_model = KerasClassifier(build_fn=make_model,
                           number_of_layers=2, neurons_per_layer=32,
                           optimizer='adam',
                           epochs=100, batch_size=256, verbose=0)

kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=random_seed)
results = cross_val_score(kc_model, X_train, original_y_train,
                          cv=kfold, verbose=0)
print("results = {}\nresults.mean = {}".format(results, results.mean()))
```

Listing B3-11: Doing cross-validation with our MNIST data

Running this code gives us the output shown in Listing B3-12.

```
results =[ 0.95221445  0.95019157  0.95617397  0.9525  0.95116667
           0.96166028  0.95999333  0.95415903  0.95797899  0.95346898]
results.mean=0.9549507265070032
```

Listing B3-12: Our first results running cross-validation on our simple Keras model

So the cross-validation run is telling us that on the original dataset of 60,000 images, we got a performance of a bit more than 95% accuracy. That's a just about the same as what we saw graphically for this model way back in Figure B3-13, where the validation accuracy was just a smidge better than 95%.

That's reassuring. It says that this whole wrapping and cross-validating scheme is producing the same results that we got when we trained and tested the model ourselves.

Although cross-validation didn't gain us anything in this example, since we already had a great validation set, now we know how to evaluate a model if we don't have such a test set handy.

Cross-Validation with Normalization

Earlier we said that something was missing from this process. The thing we left out was normalizing the data before each run of cross-validation.

We got away with it in this case because we already normalized the training data to the range [0,1] when we divided it by 255. So when cross-validation grabs a random 90% of these samples and trains on them, it's likely to get samples that run from 0 to 1.

But that's only because we've already normalized our data and things are very simple. In general, the data that's going to get chosen from our database and used for cross-validation won't be normalized to the range [0,1]. It's up to us to get that normalization in there, and then apply that same transform to the part of the data that was set aside for testing in that run.

Happily, that's easy. We just build a pipeline.

As we saw in Bonus Chapter 1, we can normalize the particular piece of training data that's built for each pass through cross-validation by building a `Pipeline` object composed of two steps: a normalizer followed by our model.

Let's do this by first making our objects, and then assembling them into a `Pipeline` object. For demonstration purposes our pipeline will contain a `MinMaxScaler` from scikit-learn, followed by our model. The `MinMaxScaler` is attractive because there are no parameters to set or options to pick. `MinMaxScaler` isn't a perfect choice for this case, since it adjusts each pixel independently, which could lead to bright or dark spots. But it should be okay for demonstration on this data. Listing B3-13 shows how to build the pipeline.

```
from sklearn.pipeline import Pipeline
from sklearn.preprocessing import MinMaxScaler

estimators = []
```

```
estimators.append(('normalize_step', MinMaxScaler()))
estimators.append(('model_step', kc_model))
pipeline = Pipeline(estimators)
```

Listing B3-13: Making a two-step pipeline with named components

Constructing a pipeline this way is useful when we want to later refer to the individual steps. We'll need to do that soon when we use grid searching.

But for this cross-validation step, we don't need that kind of access. We'll often see code that builds the pipeline in one line, using the shortcut `make_pipeline()` function. Listing B3-14 shows the code.

```
pipeline = make_pipeline(MinMaxScaler(), kc_model)
```

Listing B3-14: Making a pipeline using the shorthand notation, without names for the individual steps

These two `pipeline` objects are the same. The only difference is that we've given our own names to the steps in the first version.

To use our `pipeline` object, we just give it to `cross_val_score()` in place of a model (or wrapped model). Scikit-learn will recognize that it's a pipeline and take care of all the rest. So each time through the loop, `cross_val_score()` will select one of folds as a validation set. The rest of the data will be the training set for that run. It will give that training data to the `MinMaxScaler()` (using all the default arguments). Once the data has been analyzed, the transformation found by the `MinMaxScaler` will be applied to both the current training data and validation data. The model will then learn from the training data. When training is done, the system will run the transformed validation set through the model, predict its categories, compare those to the labels, and compute error scores.

That's a huge amount of work, all from one function call! That call is in Listing B3-15, where we simply replace `kc_model` in Listing B3-11 with `pipeline`.

```
results = cross_val_score(pipeline, X_train, original_y_train,
                          cv=kfold, verbose=0)
```

Listing B3-15: We cross-validate with our pipeline just as we do for a model.

Putting these new lines together, we get a new block of code that replaces the last few lines at the end of Listing B3-11. The new code, and the output we get from running the process with it, is shown in Listing B3-16.

```
pipeline = make_pipeline(MinMaxScaler(), kc_model)
kfold = StratifiedKFold(n_splits=10, shuffle=True, random_state=random_seed)
results = cross_val_score(pipeline, X_train, original_y_train,
                          cv=kfold, verbose=2)

print("results = {}\nresults.mean = {}".format(results, results.mean()))
```

```
results =[ 0.95454545  0.9508579   0.95534078  0. 0.95283333
    0.963994  0.95749292  0.95415903  0.95781224  0.95380253]
results.mean=0.9552838183370085
```

Listing B3-16: Setting up and calling the cross-validator with our pipeline

This value of about 0.954 matches our previous average accuracy.

In this case, the extra work of building the pipeline with the normalizer didn't pay off with any new benefits or accuracy. That's probably because randomly removing a batch of samples from the training set and then normalizing probably had little effect on the samples, since they were already normalized.

But that's definitely not going to be true for all datasets, and we should never take it for granted. Unless we are certain about the input data and its statistics, using a pipeline and processing our data is usually worth the extra effort on our part. It takes little extra computing time to compute and apply most common transformations, compared to training and testing.

We picked a `MinMaxScaler` here pretty much arbitrarily, but as we know, different data sets require different types of pre-processing. Using the pipeline mechanism, we can apply whatever steps we need.

Cross-validation is a great way to get a handle on the quality of our model. It's not so great when training times start to push our patience, since every fold is essentially a brand-new full-length training and testing process. Using 10 folds requires training and then testing our model 10 times in a row.

The time required can add up fast. But if we don't have a good validation set, then running cross-validation tests on small bits of our database, using different parameters, can teach us a lot about what's going on in our data. That knowledge can in turn help us design an efficient larger network to process the whole database.

Hyperparameter Searching

This section's notebook continues Bonus03-Keras-2-scikit-learn.ipynb.

We've been using some arbitrarily hand-picked numbers in this chapter. For instance, we've settled on an architecture with 2 layers of 32 neurons each, but not for any particular reason.

Going forward, we'll often refer to "parameters," rather than the more awkward "hyperparameters." As well as being more readable, this makes sense when we consider that many of these values are provided to the system as parameter, or arguments, of functions.

We can use the grid searching algorithms offered by scikit-learn to help us out. With those routines, we can automatically try out all the different combinations of multiple settings for multiple parameters. We could do this ourselves with some nested loops, but it's easier to relax and let scikit-learn do the driving.

The grid searching object `GridSearchCV` will try out every combination of the parameters we give it, and measure each model's performance using cross-validation. By default, it uses 3 folds to save time, but we can increase that with an optional argument.

We think of this as "searching" because we imagine that each combination of parameters is a point in some very high-dimensional space, called the *search space*. Each point in search space represents some combination of parameters, and the value of that combination (that is, the accuracy or loss that results from training a model with those parameters) is the value associated with that point. The intuition is that we're searching through this space, wandering from point to point and region to region, looking for the point that has the highest performance.

Figure B3-16 shows an example of a search space with two dimensions. The idea really comes into its own when we're dealing with many more dimensions. Though we can't visualize them, we can make the analogy to something like Figure B3-16 and talk about two sets of parameters being close or far apart, and even talk about regions of the space where it looks like we're finding good results.

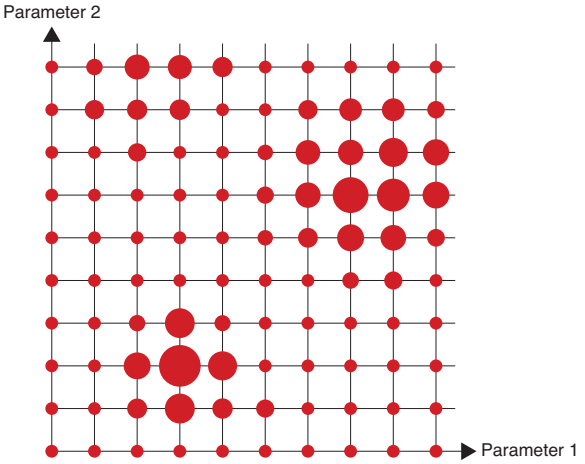


Figure B3-16: A two-dimensional search space

For each pairing of values of two parameters in Figure B3-16, the size of the circle shows the quality of the result, with larger circles better than smaller ones. It looks like there's a small but high-quality region in the lower left, and two other promising regions in the upper right and upper left. Now that we know where the good results can be found, we can investigate those areas with a finer resolution to find the best combination of parameters.

As we discussed earlier, often we'll use just a subset of our training data when searching, so that it runs more quickly. When we've found the best values for our model's parameters using this smaller database, we can use a larger version and work our way up to the full dataset.

To keep things simple, for this discussion we'll continue to use MNIST, and the whole `X_train` database while searching.

We'll want to use our normalizing pipeline again, since in general that will be necessary, or at least a very good idea.

As we saw in Bonus Chapter 1, when we prepare a pipeline for grid searching, we need to tell `GridSearchCV` where each parameter it's searching

through ought to be routed. This means we need to identify different steps in the pipeline. That's easy if we use the pipeline-building method of Listing B3-13, where we gave a name to each step.

As we saw in Bonus Chapter 1, referring to parameters inside pipelines is somewhat baroque. Let's recap briefly.

We build a dictionary, where each key is the name of a parameter to one of the steps in our pipeline, and its value is a list of all the values we'd like to explore. Each name is formed by combining the step name in our pipeline and the name of the parameter with *two* underscore characters, as in `step__parameter`. With some typefaces the two underscores look like just one big underscore, which is unfortunate, but that's how it is. For comparison, here is `one_underscore` and here are `two__underscores`.

Let's build a dictionary to search through three of our model's parameters: the number of dense layers (each with dropout), the number of neurons per dense layer, and, just for curiosity's sake, two different optimizers. Listing B3-17 shows this dictionary. Note that the keys are not strings.

```
param_grid = dict(model__number_of_layers=[ 2, 3, 4 ],
                  model__neurons_per_layer=[ 20, 30, 40 ],
                  model__optimizer=[ 'adam', 'adadelta' ])
```

Listing B3-17: A dictionary of parameters that we'd like to use for searching. Each key is a parameter named by gluing together the name of the pipeline step with the name of its parameter, with two underscores in between. Each value is a list of settings to be tried.

Now that we have our dictionary, we can use the pipeline we created above in Listing B3-14 and create our searching object, as in Listing B3-18.

```
grid_searcher = GridSearchCV(estimator=pipeline,
                             param_grid=param_grid, verbose=2)
```

Listing B3-18: Creating a `GridSearchCV` object that will go through our parameter grid, assemble a model for every combination of options, and cross-validate that model

Now we're ready to roll. We just call the searcher's `fit()` routine with our data, and let it run. Listing B3-19 puts together the searching code. We're going to suffix each variable with a `1` because we're going to run another grid search below, which will be version 2.

```
from sklearn.model_selection import GridSearchCV

param_grid1 = dict(model__number_of_layers=[ 2, 3, 4 ],
                   model__neurons_per_layer=[ 20, 30, 40 ],
                   model__optimizer=[ 'adam', 'adadelta' ])
grid_searcher1 = GridSearchCV(estimator=pipeline,
                              param_grid=param_grid1, verbose=2)

search_results1 = grid_searcher1.fit(X_train, original_y_train)
```

Listing B3-19: Combining the grid construction with `GridSearchCV` object construction, and then calling `fit()` to run the search.

Warning! Grid searches are *slow*.

The total number of full 3-fold cross-validation runs it will perform is reported by the searcher as soon as it starts up. Listing B3-20 shows what we'd see for the search we just defined.

```
Fitting 3 folds for each of 18 candidates, totaling 54 fits
```

Listing B3-20: As this output from the grid-search `fit()` shows, this exhaustive cross-validation will call `fit()` on our model 54 times.

The number 54 comes from multiplying the number of folds (3 by default) by the number of combinations of variables. In our case, we have three lists of variables, with lengths 3, 3, and 2. The total number of possibilities is found by multiplying these together: $3 \times 3 \times 2 = 18$. Since each possibility has to go through three steps of cross-validation, we will call `fit()` a total of $3 \times 18 = 54$ times.

If it takes a minute to train and evaluate the model, it will take about an hour to run this search.

The variable `search_results1` we get back contains a lot of information. One of the objects in `search_results1` is a dictionary called `cv_results_` (recall that all of scikit-learn's internal variables are suffixed with an underscore). The `cv_results_` dictionary contains detailed information on the cross-validation results.

Since we're interested in finding the best combination of parameters, two dictionary items are of particular interest. The `'params'` item tells us which set of parameters corresponds to each score. The `'mean_test_score'` item tells us the average value that came out of the cross-validation for each set of parameters.

Let's look first at the `'params'` entry, shown in Listing B3-21.

```
search_results1.cv_results_['params']
{'neurons_per_layer': 20, 'number_of_layers': 2, 'optimizer': 'adam'},
{'neurons_per_layer': 20, 'number_of_layers': 2, 'optimizer': 'adadelta'},
{'neurons_per_layer': 20, 'number_of_layers': 3, 'optimizer': 'adam'},
{'neurons_per_layer': 20, 'number_of_layers': 3, 'optimizer': 'adadelta'},
{'neurons_per_layer': 20, 'number_of_layers': 4, 'optimizer': 'adam'},
{'neurons_per_layer': 20, 'number_of_layers': 4, 'optimizer': 'adadelta'},
{'neurons_per_layer': 30, 'number_of_layers': 2, 'optimizer': 'adam'},
{'neurons_per_layer': 30, 'number_of_layers': 2, 'optimizer': 'adadelta'},
{'neurons_per_layer': 30, 'number_of_layers': 3, 'optimizer': 'adam'},
{'neurons_per_layer': 30, 'number_of_layers': 3, 'optimizer': 'adadelta'},
{'neurons_per_layer': 30, 'number_of_layers': 4, 'optimizer': 'adam'},
{'neurons_per_layer': 30, 'number_of_layers': 4, 'optimizer': 'adadelta'},
{'neurons_per_layer': 40, 'number_of_layers': 2, 'optimizer': 'adam'},
{'neurons_per_layer': 40, 'number_of_layers': 2, 'optimizer': 'adadelta'},
{'neurons_per_layer': 40, 'number_of_layers': 3, 'optimizer': 'adam'},
{'neurons_per_layer': 40, 'number_of_layers': 3, 'optimizer': 'adadelta'},
{'neurons_per_layer': 40, 'number_of_layers': 4, 'optimizer': 'adam'},
{'neurons_per_layer': 40, 'number_of_layers': 4, 'optimizer': 'adadelta'}
```

Listing B3-21: The contents of `search_results1.cv_results_['params']`. The prefix `model_step__` before each parameter has been removed to make the list fit and easier to read.

We can see that the searching algorithm checked every combination, but it proceeded in a different order than in our `param_grid1` dictionary. The outer loop ran through the 3 values of `neurons_per_layer`, the loop nested inside of that ran through the 3 values of `number_of_layers`, and finally the innermost loop ran through the 2 values of `optimizer`. Because Python dictionaries are not guaranteed to return their results in any particular order, we won't be able to predict in what order the search will proceed before we run it.

The numerical data that describes our cross-validation test performance is in `mean_test_score`, as shown in Listing B3-22.

```
search_results1.cv_results_['mean_test_score']

array([ 0.92901667,  0.91761667,  0.93081667,  0.91146667,  0.92288333,
        0.90051667,  0.9472    ,  0.93373333,  0.94561667,  0.93166667,
        0.94333333,  0.92621667,  0.95566667,  0.9424    ,  0.95376667,
        0.94185   ,  0.95413333,  0.9402    ])
```

Listing B3-22: The cross-validation scores for our search

Using NumPy's utility `argmax()` we can find the index of the largest value in this list, and then extract the corresponding element from the `'params'` item, so we can see which set of parameters gave us the best score. Listing B3-23 shows this step.

```
best_index1 = np.argmax(search_results1.cv_results_['mean_test_score'])
print('best set of parameters:\n  index {}\n  {}\n'.format(
    best_index1, search_results1.cv_results_['params'][best_index1]))

best set of parameters:
  index 12
  {'model_step__optimizer': 'adam',
   'model_step__neurons_per_layer': 40,
   'model_step__number_of_layers': 2}
```

Listing B3-23: A snippet of code that finds the best test score from our cross-validation results, and prints the parameters that gave us that score. The output was slightly reformatted to fit better.

So our best combination used 2 layers, with 40 neurons per layer, and the Adam optimizer. But how much better was this than the other combinations? Let's plot all the values of `mean_test_scores` so we can see how every combination performed, as in Figure B3-17.

Guided by the labels, we can interpret this graph as having three major sections, one each for 20, 30, and 40 neurons. Within each section we have three pairs, one pair each for 2, 3, and 4 layers. Finally, each pair of values shows the performance for Adam and then Adadelta.

Each of these innermost pairs slopes downwards, so we can say that Adam consistently outperformed Adadelta.

The general trend in each second-level group is also downwards, so adding more layers usually caused a loss of performance.

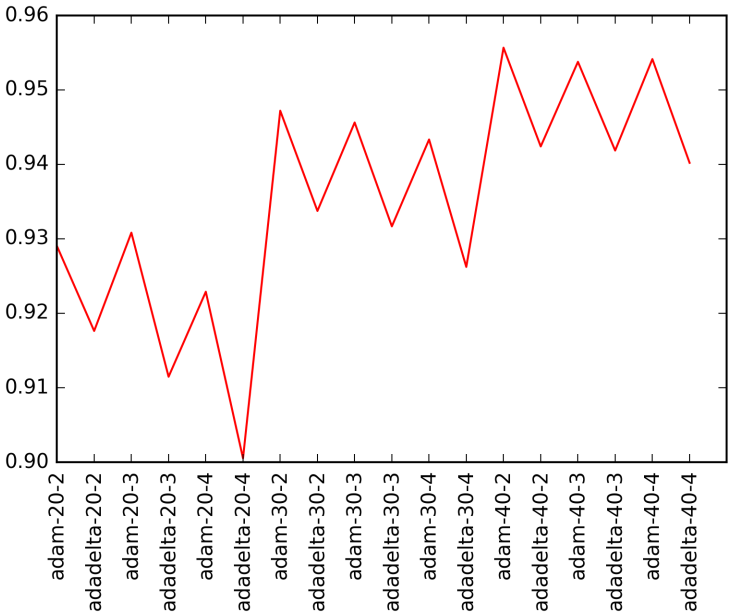


Figure B3-17: The values of `mean_test_scores` resulting from searching for the best cross-validation results for every combination of the parameters in our dictionary of Listing B3-19. The best accuracy came from 2 layers of 40 neurons each, using the Adam optimizer.

The general trend in the largest groups is going upwards, suggesting that more neurons are better than fewer.

As Listing B3-23 reports, the best set of parameters used 2 layers of 40 neurons each, optimized with Adam.

It's interesting to note that the worst performance by far was the result of 4 dense-dropout layers of 20 neurons each. So that's a structure to avoid for this data.

Keep in mind that we're always referring to "layers" as our combination of dense and dropout layers, using our default dropout rate of 0.2.

Let's search around this area of the parameter space. Since it seems fewer layers are working better than more, let's try searching for models with 1 or 2 layers. And since more neurons are performing better than fewer, we'll try going up from 40 and look at some larger values.

We could explore other optimizers, but this time around we'll stick with Adam.

There's no hard and fast rule for making these choices. We need to use our judgment based on our knowledge of our model and data, coupled with the results of our experiments, to guide our search strategy. If we search with too fine a grid we can waste a lot of time, but if we use too coarse a grid we could miss a big spike in performance. Generally speaking, searching for performance is a task that rewards both intuition and analysis.

Listing B3-24 shows the dictionary for our second search.

```
param_grid2 = dict(model_step__number_of_layers=[ 1, 2],
                   model_step__neurons_per_layer=[ 50, 80, 110, 140, 170 ])
```

Listing B3-24: The dictionary for our second parameter search.

The results are plotted in Figure B3-18.

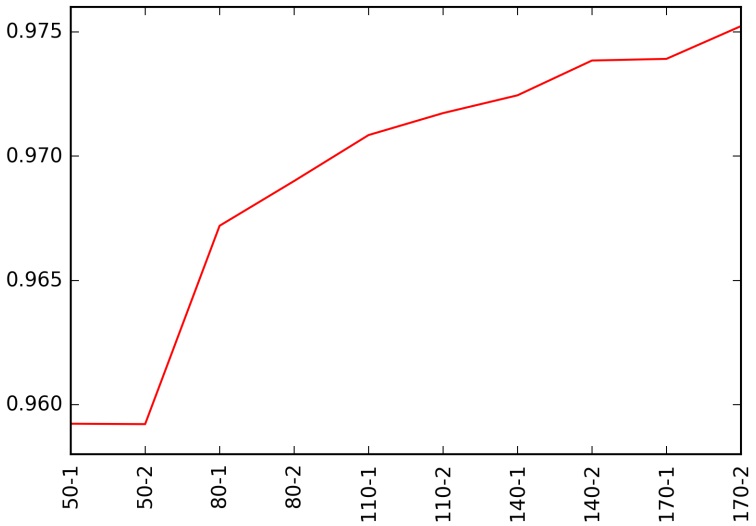


Figure B3-18: Searching 1 and 2 layer networks for layers with increasing numbers of neurons. The results alternate with the 1-layer version, then the 2-layer.

This tells us that more neurons keep working better, suggesting that our hunch was right. And 2-layer models are looking consistently better than 1-layer.

Let's crank up both the number of neurons and the search range quite a bit. Listing B3-25 shows the dictionary for our third search.

```
param_grid3 = dict(model_step__number_of_layers=[ 1, 2 ],
                   model_step__neurons_per_layer=[ 180, 280, 380, 480, 580 ])
```

Listing B3-25: The dictionary for our third parameter search.

The results are in Figure B3-19.

Figure B3-19 shows us that the best performance came from 2 layers of 280 neurons each. As we added more neurons, performance started to decline, though slowly. Perhaps this is due to overfitting, though we'd have to look more closely to be sure.

Notice the size of the vertical scale. Our first graph in Figure B3-17 showed an improvement of about 0.055 from the worst performer to the best, while in our most recent graph Figure B3-19 the difference is only about 0.0035, which is about 1/15 the size.

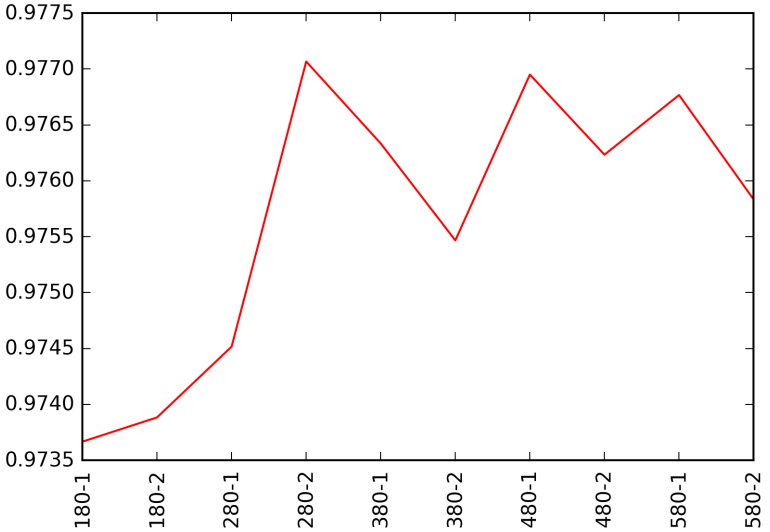


Figure B3-19: Searching 1 and 2 layer networks for layers with large numbers of neurons. The results alternate with the 1-layer version, then the 2-layer.

The overall feeling of the curve, though it's jumping around when we're zoomed in so much, seems to be flattening out. We might be pretty close to the best choice for this set of parameters.

We'll stop searching here, but we could continue to try different values for all of these parameters, or some we didn't even try (like any of the `Normalization` object's parameters, or `dropout_ratio` in our own model).

The time required to do a full grid search can quickly become impractical, because the searcher exhaustively tests every possible combination of the search parameters. So it's always a good idea to use the smallest number of combinations we can get away with, and to search with the smallest amount of data that will give us a reasonable prediction of the model's performance.

A good strategy is to start with searches that cover broad ranges with just a few values. When we see where the model is performing best, we can then run another, denser search to explore the area around that zone. This is called *multiresolution searching*, and it's just an algorithmic version of what we do when we look for something in the real world. Say we're looking for a book in the library. We have a call number, so we wander around to the right section of the library, then the right stack, the right shelf, and so on, using a series of ever-narrower searches until we find our book.

We do the same thing with ever-smaller searches until we find a combination of parameter values that work the best.

A useful alternative to the exhaustive search performed by the grid searcher is provided by scikit-learn's `RandomizedSearchCV` algorithm. As discussed in Bonus Chapter 1, this variation on the grid search picks an

unexplored random combination of the search parameters on each run. We could for instance just search a third of the total combinations. We'd get our answer back 3 times sooner than a complete grid search, but it would be incomplete. It would be incomplete in a nice way, though, giving us a roughly equal scattering of points throughout the parameter space. This might be enough to guide our choice for a smaller, more focused search.

Convolution Networks

This section's notebook is Bonus03-Keras-3-CNN.ipynb.

Let's build some *convolutional neural networks*, also called *convnets*, or more commonly, *CNNs*.

In Chapters 16, 17, and 23 we surveyed some of the remarkable things we can do with convnets. Recall that each convolution layer holds a collection of *filters*, or *kernels*, which are rectangles of numbers (often a small square that is 3, 5, or 7 elements on a side). When we use 2D convolution layers with images, each filter in the first layer is applied in turn to every pixel in the input. The output of the filter becomes the value of that element in a new tensor produced at the layer's output. If there are multiple filters, then the output tensor contains multiple *channels*, just like the red, green, and blue channels of a color image.

Although the original input to the first convolution layer of a model is often an image, the data that flows through the rest of the network is not. If a layer has 32 filters, for example, then the output will have 32 channels. It may have the same width and height as the input (though we'll see those measures often change as well), but it's not really an "image" any more. So while it's tempting to casually speak about convolution layers as processing "images" made of "pixels," it's a better idea to refer to them as tensors, made of elements.

Recall from Chapter 16 that we can characterize a convolution layer by the number of dimensions in which the filters are moved. If the filter is moved in just one dimension (for example, down), then we call it a 1D convolution layer. Typically, when we work with images, we slide our filters over the 2D width and height of the tensor, so we usually use 2D convolution layers for image processing. Keras also offers 3D convolution layers for working with volumetric data.

In this chapter, we'll focus on images, so we'll be using 2D convolution layers.

In the following sections we'll see how to build and train our own CNNs. As we'll discuss later, in practice we don't often build and train a new CNN from the ground up. Instead, we usually try to start with an existing network whenever possible, and specialize it for our task by perhaps modifying it, and then training it some more with our own data. Such *transfer learning* is appealing because we get to start with an existing architecture that is known to work well, and we save the time (sometimes days or weeks) that was invested in training the model we're building upon. We also get

the benefit of the data that network was trained on, which might not be available to us.

But it's important to know how to build our own from scratch. This lets us start fresh when we need to, and gives us the tools to modify an existing network when we want to. Whether we're working with our own model or one we've adopted, knowing what's going on inside will help us diagnose problems and get the best performance out of our model.

Let's begin by setting up a few basic ideas that will be helpful when we build our convnets.

Utility Layers

We'll briefly recap some of the utility layers that we saw throughout the book, focusing on those that are useful in convnets.

Like the dropout layer, these aren't fully-fledged computational layers. Instead, they're often "supplemental" layers that tell Keras how to process or manipulate data that flows through the layer, or how to affect a previous layer. Keras structures these as layers so that we can think of our model consistently as a stack of layers.

Most of the layers described below are available in 1, 2, and 3 dimensional forms. When working with images, we almost always use the 2D versions, so that's all we'll cover here.

A *flatten* layer takes a tensor of any number of dimensions and lines up all of its contents into a single one-dimensional list. It always does this in the same order, so we can predict where each element in the tensor will appear in that list (we usually don't care what in what order the elements are listed, as long as it's consistent from one sample to the next). Keras calls this layer `Flatten`.

A *pooling* layer looks at elements that make up a block in the input, calculates a single value from them, and saves that single value to the output in place of all the input elements. The most common use of pooling is to reduce the size of its input. For example, if the blocks are 2 by 2, and they don't overlap, the output will be half the width and height of the input. Keras offers two types of pooling layers. The `MaxPooling2D` layer finds the largest value in each block. We can tell it the *size* of the block to use, and the *stride*, or how many elements to move horizontally and vertically after each block. Commonly we use blocks that are 2 by 2 with a stride of 2 in each dimension. The `AveragePooling2D` layer works the same way, but instead computes the average value of each block.

As we discussed in Chapter 16, pooling layers after convolution layers are falling out of favor, replaced by *striding* within the convolution layer itself to achieve a similar result. The striding and pooling approaches produce similar but different results. Usually we don't care about that difference, but sometimes it matters, so pooling layers are still sometimes used.

A *cropping* layer removes a tensor's outermost elements, leaving just the inner rectangle. The Keras layer called `Cropping2D` takes arguments which let us describe how many elements to remove from each of the four sides.

An *upsampling* layer is designed to make the input tensor larger. Each element is just repeated horizontally and vertically by the given number of times. Keras calls this layer `UpSampling2D` (note the upper-case S in the middle of the name).

As we mentioned in Chapter 16, an alternative to an explicit upsampling layer after a convolution layer is to use transposed convolution (or fractional striding) in the convolution layer itself. Like normal striding and max pooling, transposed convolution produces a similar but different result compared to upsampling.

A *batchnorm* layer performs regularization on each batch of data flowing through it, giving it an average of 0 and a standard deviation of 1. This helps keep weights from growing too large.

A *noise* layer adds some random noise to every element in the tensor. This is rarely used, but can be helpful if some neurons seem to be overly-aggressive in matching specific features that are not ultimately important.

Finally, a *zero padding* layer places 0's around the perimeter of the input. This is usually so that convolution kernels will not "fall off the edge" and try to access non-existent data. Keras calls this the `ZeroPadding2D` layer (note the upper-case P). Because Keras now offers this feature in the convolution layers themselves, explicit zero-padding before convolution is rare in Keras 2 models.

A recap of our schematic symbols for the major types of layers is shown in Figure B3-20.

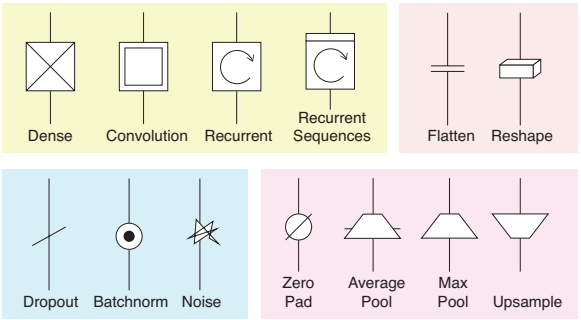


Figure B3-20: Schematic symbols for different layers

Preparing the Data for A CNN

We'll continue to use MNIST for our example data set.

We prepare our MNIST data for a convnet with *almost* the same process as we've been doing so far when the first layer was a `Dense`, or fully-connected, layer.

The difference is in the shape of the feature data. So far, we've been shaping our feature data in a 2D grid, with one row per image. Each row held all the pixels for that image.

Something important happened when we flattened out our image to make that grid: we lost the spatial information that tells us which pixels are near one another vertically (technically, it's still there, but definitely not in a structure that's easily useful). A great thing about CNNs is that they work with inputs as multidimensional tensors, not long 1D lists. For instance, the receptive field for a filter covers a group of spatially-related elements.

When working with CNNs there's no need to flatten out input 2D grids of pixels. We'll maintain them instead as three-dimensional volumes, where each input image has a height, width, and depth.

One important use of this third dimension is to bundle together the channels of data that represent color images. A typical digital color image has three channels, one each for red, green, and blue. So if we stack these up, we'll get a block of 3 layers. On the other hand, a picture that has been prepared for printing usually has four channels: cyan, magenta, yellow, and black. This requires a block of 4 layers. Figure B3-21 shows this idea.

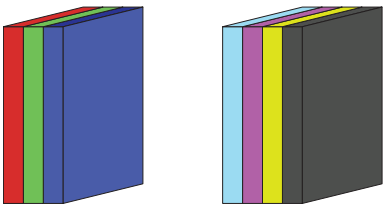


Figure B3-21: Multiple 2D grids of pixels are often used to represent richer types of images. (a) A typical digital image in color uses three channels, one each for red, green, and blue. (b) A file prepared for printing may have four channels, for cyan, magenta, yellow, and black.

The MNIST data is black and white, so we have just a single channel of pixel data. But we still have to explicitly tell Keras that we have just that one channel, by making it one of the dimensions of our input tensor.

The order in which we name our dimensions depends on whether we're using the `channels_first` or `channels_last` option, as we discussed at the start of Bonus Chapter 2.

We'll use `channels_last` in this section.

By adding a channel dimension, each MNIST image will become a 3D block with dimensions 28 by 28 by 1. Our input data structure will contain 60,000 of these 3D blocks. That means the complete tensor will have a first dimension of 60,000, followed by the shape of each image.

Using the `channels_last` convention, this tensor will have dimensions of 60,000 by 28 by 28 by 1.

We can't draw a 4D tensor, but we can show lots of 3D tensors in a list. Figure B3-22 uses that approach to picture our dataset.

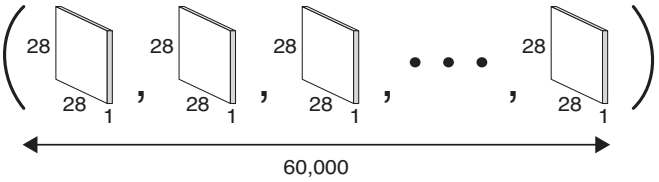


Figure B3-22: Each image in the input will be reshaped as a 3D block with dimensions 28 by 28 by 1, since we're using the `channels_last` convention. Then we stack all 60,000 of these 3D blocks together to make a 4D tensor of shape 60,000 by 28 by 28 by 1, which will serve as input to our CNN.

As we discussed earlier, it can be easier to think of this not as a 4D structure but instead as a sequence of nested lists: the outermost list contains 60,000 images, each image contains 28 rows, each row contains 28 elements, and each element has 1 channel.

This means that each pixel is named with *four* numbers: the image number, y position, x position, and channel number, in that order.

Convnets work best with input data scaled from –1 to 1 [Karpathy16b]. This means we can't just divide every pixel by 255. Instead, we'll use the NumPy function `interp()` to convert each input value in the range [0,255] to the range [–1,1], shown in Listing B3-26.

```
X_train = np.interp(X_train, [0, 255], [-1,1])
X_test = np.interp(X_test, [0, 255], [-1,1])
```

Listing B3-26: Using NumPy's `interp()` routine to convert all the input values from [0,255] to [–1,1]

Now we'll re-shape the data into the shape we just discussed. We just tell NumPy how to take our original version of `X_train`, which was 60,000 by 28 by 28, and reshape it into a 4D tensor that's 60,000 by 28 by 28 by 1. We're not changing the total number of elements, so this is a valid request. We just hand it the dimensions we want it to use. Listing B3-27 shows the code.

```
# reshape sample data to 4D tensor using channels_last convention
X_train = X_train.reshape(X_train.shape[0], image_height, image_width, 1)
X_test = X_test.reshape(X_test.shape[0], image_height, image_width, 1)
```

Listing B3-27: How to transform our input MNIST data into the 4D tensor that our CNN expects.

We'll place these re-shaping lines right after the scaling step. For completeness, Listing B3-28 shows all the pre-processing in one place. This includes all the `import` statements we'll be needing going forward. Aside from the `import` statements, and the final re-shaping, this pre-processing is identical to what we've been doing so far. A CNN is, after all, just another deep neural network, but jazzed up with some new types of layers.

Note that we're assuming that the `channels_last` option has been selected in the Keras configuration file. If that's not the case, either change the file, or import the backend and include a call to set the value of `image_data_format`.

```
from tensorflow.keras.datasets import mnist
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers.core import Dense, Dropout, Activation, Flatten
from tensorflow.keras.layers.convolutional import Conv2D, MaxPooling2D
from tensorflow.keras.constraints import MaxNorm
from tensorflow.keras.optimizers import Adam, SGD, RMSprop
from tensorflow.keras import backend as tensorflow.keras_backend
from tensorflow.keras.utils import np_utils
from tensorflow.keras.preprocessing.image import ImageDataGenerator
from tensorflow.keras.utils.np_utils import to_categorical
import numpy as np

random_seed = 42
np.random.seed(random_seed)

# load MNIST data and save sizes
(X_train, y_train), (X_test, y_test) = mnist.load_data()
image_height = X_train.shape[1]
image_width = X_train.shape[2]
number_of_pixels = image_height * image_width

# convert to floating-point
X_train = keras_backend.cast_to_floatx(X_train)
X_test = keras_backend.cast_to_floatx(X_test)

# scale data to range [-1, 1]
X_train = np.interp(X_train, [0, 255], [-1,1])
X_test = np.interp(X_test, [0, 255], [-1,1])

# save original y_train and y_test
original_y_train = y_train
original_y_test = y_test

# replace label data with one-hot encoded versions
number_of_classes = 1 + max(np.append(y_train, y_test))
y_train = to_categorical(y_train, num_classes=number_of_classes)
y_test = to_categorical(y_test, num_classes=number_of_classes)

# reshape sample data to 4D tensor using channels_last convention
X_train = X_train.reshape(X_train.shape[0], image_height, image_width, 1)
X_test = X_test.reshape(X_test.shape[0], image_height, image_width, 1)
```

Listing B3-28: The pre-processing step for our CNNs to categorize MNIST data

Shaping the feature data into these 4D tensors is a necessary pre-processing step. It puts the data into the structure that is expected by the convolution layer that will sit at the start of our convnet.

Convolution Layers

Let's look more closely at how we define a convolution layer. We saw that Keras offers convolution layers in 1, 2, and 3 dimensions. We'll pick the 2D version, since that's what we usually use for image data like our running example of MNIST digits.

The layer has the name `Conv2D`, and we access it by importing it from the module `tensorflow.keras.layers.convolutional`.

The `Conv2D` layer takes two unnamed, mandatory arguments at the start of its argument list, followed by a variety of optional arguments.

The first mandatory argument is an integer specifying the number of filters the layer should manage. Recall from Chapter 16 that each filter is applied to the input independently, and produces its own output. So if our input has one channel (as our input does), and we use 5 filters in a convolution layer, the output will have 5 channels, as shown in Figure B3-23.

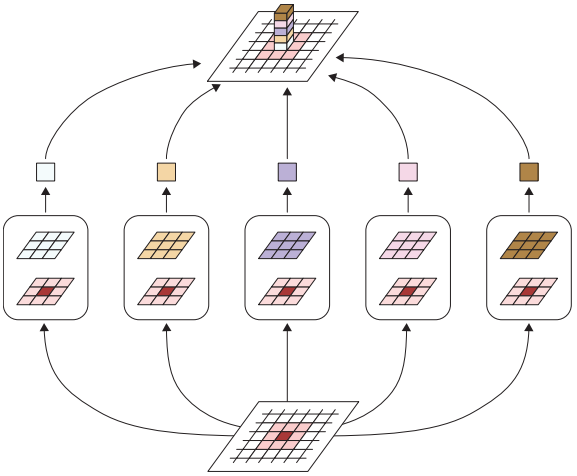


Figure B3-23: If we have a 1-channel input and 5 filters, the output tensor will have 5 channels, one for the output of each filter. Here we show the operation of the 5 filters on a single element of the input, producing an output with 5 channels.

The second argument to `Conv2D` is a list that gives the dimensions of the filters on this layer. In Keras, as in many libraries, all of the filters in a given layer are of the same size, so we only specify one filter size for the entire layer. Continuing our example from before, if we have 5 filters and each is 3 by 3, then these arguments would be `5,[3,3]`. This tells the layer to automatically allocate and initialize 5 volumes, each of shape 3 by 3 by 1 (the trailing 1 is the number of channels).

In practice, we almost always use square kernels, often of 3 or 5 elements on a side. Experience has shown that these sizes, coupled with reduction in the size of the input (either by pooling or striding), represents a good tradeoff of computation and results. Larger kernels are sometimes used, but they're not as common.

Keep in mind that these filter kernels are 3D volumes, since there's one channel in the kernel for each channel in the input. For example, suppose that our input image to the first layer is a color image, and that our layer is using 5 by 5 kernels. Since there are three channels, each filter is created as a block that is 5 by 5 by 3. This block is moved over the image in 2 dimensions (hence the name `Conv2D`), and at each element the 75 values from the input are multiplied with their corresponding 75 entries in the kernel, the results are all added together, and that's the output of this kernel at this element, as shown in Figure B3-24.

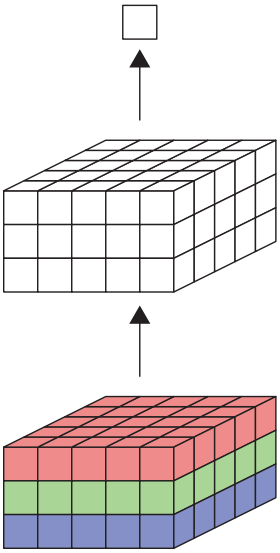


Figure B3-24: Each filter automatically holds as many channels as there are in the input.

In Figure B3-24, a 5 by 5 filter is being applied to a 3-channel input, so the system automatically gives the filter 3 channels as well. Each of the 75 values in the input (bottom) is multiplied by its corresponding value in the filter (middle), and all of those products are added together to produce a single number (top), which then passes through an activation function, producing the output of that filter for that location of the input.

If we have several of these filters in a given convolution layer, then we'll produce several outputs, just as in Figure B3-23, except now for a multi-channel input. Figure B3-25 shows the idea.

Keeping track of all of these shapes would be an administrative challenge, so Keras manages them for us. As a result, we can create sequences of convolution layers by doing nothing more than telling Keras how many filters we want to use on each layer, and what their footprint should be. Keras keeps track of the number of channels coming from the previous layer, and makes the filters of the necessary size, with no effort from us.

For example, let's suppose that we've made a convolution layer with 5 filters, each 5 by 5. Then every output it produces will have 5 channels. If the next layer is also a convolution layer, and we say that we want 2 filters

that are 3 by 3, Keras will automatically know to make each filter 5 channels deep, since that's what's coming out of the previous layer. In short, the number of channels in each filter is equal to the number of channels in the input, which in turn is the number of filters used in the previous convolution layer. Figure B3-26 shows this idea visually.

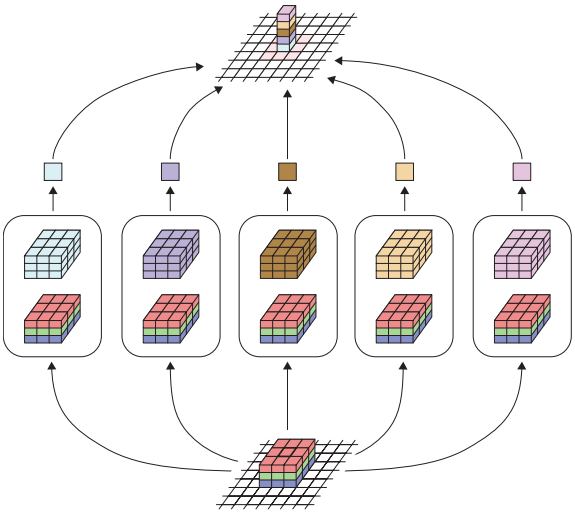


Figure B3-25: If we apply several filters to a multi-channel input, then each filter will also have multiple channels. The number of channels in the output is given by the number of filters that were used.

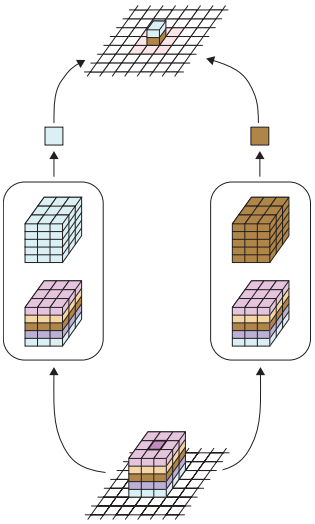


Figure B3-26: When one convolution filter follows another, the filters in the second layer are automatically configured to have as many channels as there are filters in the preceding layer.

This automatic sizing of filters is one of the great pleasures of using a library such as Keras. We don't have to manage any of this book-keeping ourselves, because the library already has everything it needs to know to always keep things straight.

Let's make a convolution layer with 15 filters, each 3 by 3. Listing B3-29 shows the code for making the layer and placing it into a model.

```
convolution_layer = Conv2D(15, (3, 3))
model.add(convolution_layer)
```

Listing B3-29: How to make a convolution layer with 15 filters that are each 3 by 3

In practice, we usually do this in one step, as we did with other layers earlier in the chapter. The single line that's usually used for this job is shown in Listing B3-30.

```
model.add(Conv2D(15, (3, 3)))
```

Listing B3-30: The more usual way to add a convolution layer to a model

The `Conv2D` layer accepts many optional arguments, all of which are described in the documentation. We'll discuss just the ones that we'll be needing.

We'll start with the arguments that are unique to convolution layers, and then touch on the ones we've already seen when using `Dense` layers. For convenience, we'll continue to refer to input tensor elements as "pixels," though as we discussed earlier that doesn't really make sense past the first layer.

A nice optional feature of the convolution layer is that we can include zero-padding inside the layer itself if we want it, rather than building an explicit zero-padding layer into the model. Even better, Keras can automatically compute how much zero padding we need if we want our output to have the same shape as our input. It uses the sizes of the filters, and our striding choices if necessary, and adds enough 0's around the outside to guarantee that the filters will never "fall off the edge" of the input.

To apply zero padding, we set the optional argument `padding` to the string `'same'`, meaning "make the output the same size as the input." The default value of padding is the string `'valid'`, which means "only place the filter where there is valid data available." This is a long-winded way of saying, "no padding."

We saw in Chapter 16 that striding allows us to move the filter by any distance on each step. We set the stride amounts with the optional parameter `strides`. This accepts a list of 2 numbers, giving the number of pixels to move horizontally and vertically. This list defaults to `(1,1)`. For example, if we set the stride values to `(2,2)`, then our output will be half the width and height as the input. Note that number of output channels is not affected by the stride, since that comes from the number of filters.

As a little convenience, we can set `strides` to just a single number, and it will use that for both directions. So instead of the list `(2,2)`, we can just give the single value `2`.

The last option we'll look at now is `activation`. Just like our previous discussion of `Dense` layers, every value produced by a convolution layer goes through an activation function. The default is a linear function, which in effect does nothing. We can set an activation function by providing its name as a string. All of the functions we discussed in Chapter 13 are available, plus several others (see the Keras documentation). Common choices for hidden convolution layers are `'relu'` and `'tanh'`.

Recall that a `Dense` layer required an argument to `input_shape` if and only if that layer was the first layer in the network. Convolution layers work the same way, and require a value to be assigned to `input_shape` if they're the very first layer.

The `input_shape` argument we applied to `Dense` layers was a list with a single value: the length of the list that represented an image. Just as with those layers, a CNN layer wants the `input_shape` to describe not the shape of the whole data set, but *just one sample*. We know that in our MNIST example we have 60,000 samples, each 28 by 28 by 1. Therefore, the value of `input_shape` is the list `(28,28,1)`, describing one image.

Now that we've covered all the background, let's construct a 2D convolution layer. This will be the first layer in our model, so we need the `input_shape` argument. Let's say that we want 16 filters, each 5 by 5. For the sake of demonstration, we'll pick the `relu` activation function, zero-padding so that the output is the same size as the input, and a stride of 2 in both X and Y. Listing B3-31 shows how we'd add this to a model named `model`.

```
model.add(Conv2D(16, (5, 5), activation='relu',
                 strides=(2, 2), padding='same',
                 input_shape=(image_height, image_width, 1)))
```

Listing B3-31: Adding a 2D convolution layer to our model. The first argument is the number of filters, followed by the width and height of each filter. The other, named arguments (except for `input_shape`) are optional. Here, we set the activation function to `relu`, choose zero-padding to make the output the same sizes as the input by setting `padding` to `same`, set the stride to (2,2), and we specify the shape of the input tensor using the `channels_last` convention.

After all that discussion, the mechanics end up to be pretty short, even with the optional arguments.

That covers the basics of using convolution layers. Everything else is just like before: we create our model, add in layers, compile it, and then train it. Later we can ask it for predictions.

Keras takes care of all the work of creating filter kernels of the right size, initializing them with good values, and improving them with backprop.

Now that we know how to create a convolution layer, let's build a fully-functional convnet.

Using Convolution for MNIST

Let's build a CNN for categorizing MNIST images. To start with, we'll make a simple convnet with just one convolution layer, a flattening layer, and a fully-connected output layer.

Our pre-processing step is unchanged from what we saw in Listing B3-28. It reads in our data, normalizes it, re-shapes the features to the 4D channels-last tensor, and makes one-hot encodings for the labels.

Our model will have the architecture of Figure B3-27.

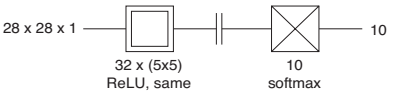


Figure B3-27: The architecture for a tiny convolutional neural network. We have one convolutional layer with 32 filters, each a square of 5 by 5 elements. Following that is a flatten layer, and then a 10-neuron fully-connected layer to present our categorization outputs.

To create our model, we start just as before, by creating a new object of type `Sequential`, then then adding layers one at a time.

We'll start with a 2D convolution layer with 32 kernels, each 5 by 5. We'll use a `relu` activation function, and set `padding='same'`, which will give the layer a temporary ring of 0-padding, so that the output will have the same horizontal and vertical sizes as the input. Since the Dense layer at the end takes a list as input, we'll use a Flatten layer to turn the 28 by 28 by 32 tensor into a list of 28*28*32 = 25,088 elements. We use `softmax` on that final layer, just as before.

As usual, we'll wrap all of this up in little function which creates the model, compiles it, and returns the final result.

Listing B3-32 shows the code.

```
def make_simple_cnn_model():
    model = Sequential()
    model.add(Conv2D(32, (5, 5),
                     activation='relu', padding='same',
                     input_shape=(1, image_height, image_width)))
    model.add(Flatten())
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-32: The code to make our first CNN for classifying MNIST digits

As far as Keras is concerned, this is just a `Sequential` object like any other. We can train this model just as we always have, by calling `fit()` with all the necessary parameters. Just for completeness, Listing B3-33 shows the code. Like our experiments in the last section, we'll run this model for 100 epochs, using a batch size of 256.

```
simple_cnn_model = make_simple_cnn_model()

simple_cnn_history = simple_cnn_model.fit(X_train, y_train,
                     validation_split=0.25,
                     epochs=100, batch_size=256)
```

Listing B3-33: To train our CNN, we only need to call its `fit()` method with the usual parameters.

The results of our this training session are shown in Figure B3-28.

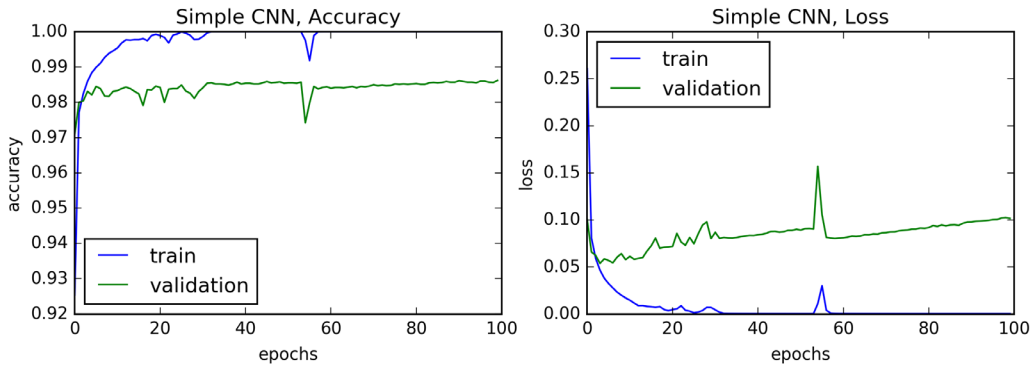


Figure B3-28: The accuracy and loss for our simple convnet

The numbers from the final epoch are shown in Listing B3-34.

```
Epoch 100/100
66s - loss: 2.7044e-04 - acc: 1.0000 - val_loss: 0.1016 - val_acc: 0.9862
```

Listing B3-34: The final values from Figure B3-28

The good news about these results is that everything seems to be working pretty well. Our system learns the training data and does a good job predicting the classes of the validation data, getting about 98.6% accuracy.

On the other hand, these curves aren't really looking great. The training accuracy gets up to about 1.0 within 35 epochs or so, and the validation accuracy seems to plateau about there as well. That's okay, but the loss curves tell a different story. The system starts overfitting before even 10 epochs are done, and it just gets worse as time goes on.

As usual, to cure problems like this we need to follow our hunches. Let's guess that maybe we would see better performance if we used a deeper model. We'll increase the number of convolution layers from 1 to 3, and to control overfitting we'll add dropout after each one.

Figure B3-29 shows our new, deeper architecture.

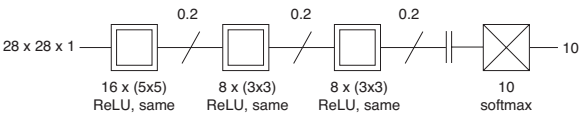


Figure B3-29: A bigger CNN using multiple convolution and dropout layers

The first convolution layer uses 16 filters of 5 by 5, and then we follow that with two layers that use 8 filters of 3 by 3. All of these numbers are more or less arbitrary, resulting from an initial guess and then some trial and error. We follow each convolution layer with a dropout layer, and at the end we flatten the result and feed it to a 10-neuron dense layer using softmax, as usual.

Because we're using the `border_mode='same'` option, and the default stride of 1 by 1, the output of each convolution will be the same width and height as the input.

Listing B3-35 shows a function to make this new model. Note that we're setting the optional argument `kernel_constraint` to the value `MaxNorm(3)`, just as we did with the `Dense` layers earlier. For convolution layers it prevents the values in the filters from getting too big, in the same way that it prevented the weights in the `Dense` layers from getting too big.

```
def make_bigger_cnn_model():
    model = Sequential()
    model.add(Conv2D(16, (5, 5), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3),
                     input_shape=(1, image_height, image_width)))
    model.add(Dropout(0.2))
    model.add(Conv2D(8, (3, 3), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Conv2D(8, (3, 3), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-35: The code to make the model of Figure B3-29

Listing B3-36 shows how we'd call this function to make a new model.

```
bigger_cnn_model = make_bigger_cnn_model()

bigger_cnn_history = bigger_cnn_model.fit(X_train, y_train,
                     validation_split=0.25,
                     epochs=100, batch_size=256)
```

Listing B3-36: Training the model of Listing B3-35

The results are shown in Figure B3-30.

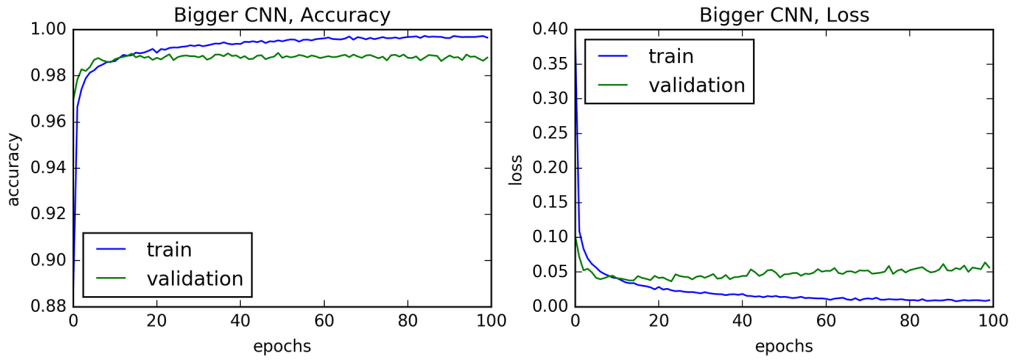


Figure B3-30: The accuracy and loss from training our model of Figure B3-29 for 100 epochs

The numbers from the end of this run are in Listing B3-37.

```
Epoch 100/100
127s - loss: 0.0094 - acc: 0.9965 - val_loss: 0.0565 - val_acc: 0.9879
```

Listing B3-37: The numbers from the final line of training our bigger CNN

We've pretty much eliminated overfitting, though the validation accuracy is just a bit lower than before. As far as validation accuracy goes, it seems we could have stopped after about 40 epochs.

There might be a small amount of overfitting going on, which we can try to reduce by adjusting hyperparameters, as we did before.

On a late-2014 iMac, without GPU support, using TensorFlow, this model took a little more than 2 minutes for each epoch. That's about double the time required by our first model with just one convolution layer.

Now that we have our feet wet, we can think about looking for better performance. But where do we start?

Progress on deep-learning architectures often comes from building on results other people have developed and published. Looking at the MNIST page [LeCun13], we can see architectures for convnets that have worked well on this data set. Most of these have some advanced or experimental features, but we can still emulate their basic structure.

Let's try making the image smaller and smaller as it works its way through the network. This way each layer gets to work with larger pieces of the original image.

We'll do this first using pooling layers. Though we've noted pooling layers are falling out of favor in convnets, we'll use them now because they let us explicitly show how the tensor gets reduced in size as it flows through the network.

Each pooling layer will look at 2 by 2 non-overlapping boxes, and return the largest value in that group of 4 input elements. As a result, the output of each of these layers will have half the width and height of its input. The input images are 28 by 28, so the output of the first max pooling layer will be 14 by 14, and the output of the second will be 7 by 7. As usual, the depth of each of these tensors is given by the number of filters in the preceding convolution layer.

We'll include three dense layers at the end, also of decreasing size. This is basically just working on a hunch that because we have fewer inputs arriving at the final dense layers (just $7 \times 7 = 49$ values in all), we could benefit from more processing of those values. And what the heck, let's include dropout after each convolution layer.

Figure B3-31 shows a diagram of this architecture.

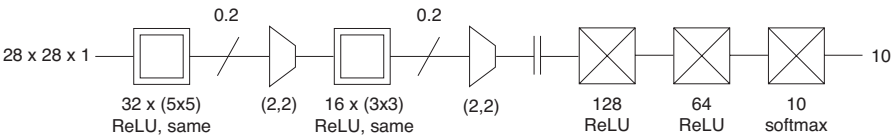


Figure B3-31: A CNN with dropout and pooling layers

Listing B3-38 shows a function to make this new model.

```
def make_pooling_cnn_model():
    model = Sequential()
    model.add(Conv2D(30, (5, 5), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3),
                     input_shape=(1, image_height, image_width)))
    model.add(Dropout(0.2))
    model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
    model.add(Conv2D(16, (3, 3), activation='relu', padding='same',
                     kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(MaxPooling2D(pool_size=(2, 2), padding='same'))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-38: Making the model in Figure B3-31

Listing B3-39 shows how we'd call this function to make a new model.

```
pooling_cnn_model = make_pooling_cnn_model()

pooling_cnn_history = pooling_cnn_model.fit(X_train, y_train,
                          validation_split=0.25,
                          epochs=100, batch_size=256)
```

Listing B3-39: Code to train the model of Figure B3-31

The results are shown in Figure B3-32.

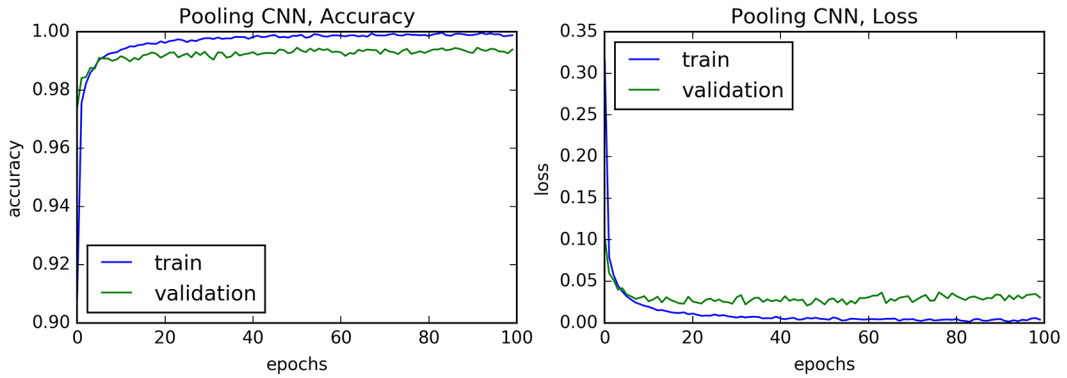


Figure B3-32: Results for training the model of Figure B3-31 for 100 epochs

The numbers from the end of this run are in Listing B3-40.

```
Epoch 100/100
147s - loss: 0.0038 - acc: 0.9988 - val_loss: 0.0304 - val_acc: 0.9939
```

Listing B3-40: The final line of data from training the model of Figure B3-31 for 100 epochs

We've picked up a small but meaningful increase in validation accuracy, from 0.9879 to 0.9939. As we mentioned earlier, progress at this point often proceeds in tiny steps. This is actually pretty large, since we've shaved off almost half of the distance to 1.

And we seem to have no overfitting. In fact, after about the 50th epoch everything looks pretty much settled.

We mentioned that pooling layers after convolution are being replaced these days with striding in the convolution layers themselves. So let's duplicate this most recent architecture, but instead of 2 by 2 max pooling layers, we'll use 2 by 2 striding in the convolution layers. The resulting sizes will the same as before. The results will be a little different, because the process of striding the convolution kernels is not identical to moving them by single steps and then pooling, but we'd expect a rough similarity at least.

Figure B3-33 shows a diagram of this architecture.

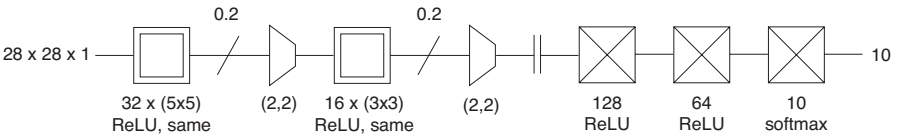


Figure B3-33: Our model of Figure B3-31, where we've replaced the pooling layers with striding in the convolution layers

Listing B3-41 shows a function to make this new model.

```
def make_striding_cnn_model():
    model = Sequential()
    model.add(Conv2D(30, (5, 5), activation='relu', padding='same',
                     strides=(2, 2), kernel_constraint=MaxNorm(3),
                     input_shape=(1, image_height, image_width)))
    model.add(Dropout(0.2))
    model.add(Conv2D(16, (3, 3), activation='relu', padding='same',
                     strides=(2, 2), kernel_constraint=MaxNorm(3)))
    model.add(Dropout(0.2))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-41: Code to make the striding CNN of Figure B3-33

Listing B3-42 shows how we'd call this function to make a new model.

```
striding_cnn_model = make_striding_cnn_model()

striding_cnn_history = striding_cnn_model.fit(X_train, y_train,
                            validation_split=0.25,
                            epochs=100, batch_size=256)
```

Listing B3-42: Code to build the striding CNN of Figure B3-33

The results are shown in Figure B3-34.

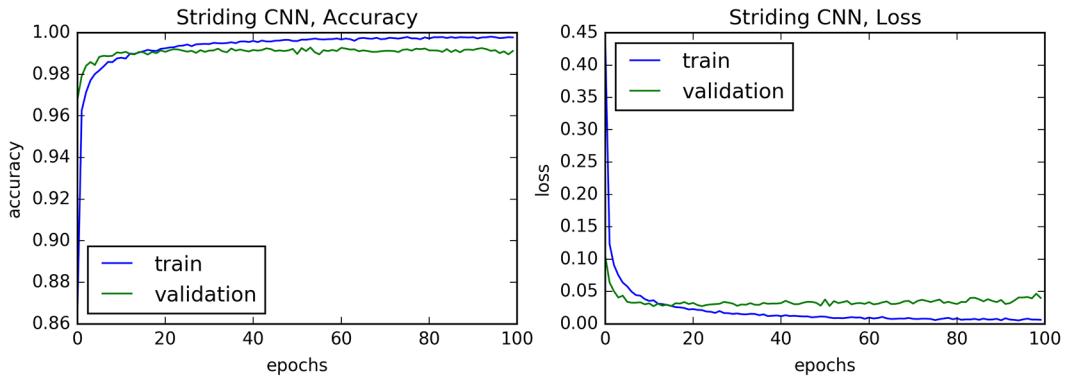


Figure B3-34: Accuracy and loss from training the striding CNN of Figure B3-33 for 100 epochs

The numbers from the end of this run are in Listing B3-43.

```
Epoch 100/100
36s - loss: 0.0062 - acc: 0.9978 - val_loss: 0.0400 - val_acc: 0.9912
```

Listing B3-43: The final lines from training the model of Figure B3-33

We've lost a very tiny bit of accuracy in both the training and validation sets, but otherwise these numbers and their graph look pretty much like what we saw from using explicit pooling layers.

But what has changed a lot is the timing. As we can see in Listing B3-40, the pooling model required about 147 seconds per epoch on a late-2014 iMac without GPU, while the striding version took only 36 seconds per epoch. The striding epochs took only about 25% of time required by the epochs that used explicit pooling. Clock times can be misleading, but it's hard not to like getting nearly the same performance with only 25% of the time and effort.

Can we increase performance even more?

We can play with any aspect of our network. We can add or remove filters at each layer, change their size, increase the dropout percentage, add more convolution layers, and so on. Just for variety, let's try replacing the dropout layers with batchnorm layers. Both are designed to reduce overfitting, so we can see which of the two techniques works best for this network and this data.

Figure B3-35 shows a diagram of this architecture.

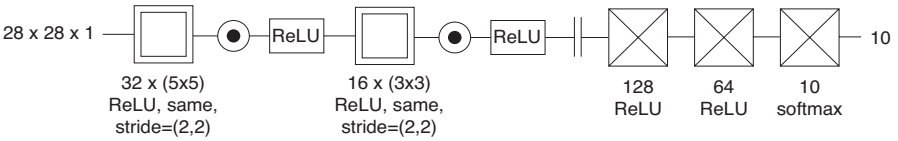


Figure B3-35: Our model of Figure B3-33, where we've replaced each dropout layer with a batchnorm layer.

Listing B3-44 shows a function to make this new model. We're setting the `activation` parameter in the convolution layers to `None`, because, as we saw in Chapter 15, batchnorm operates *between* a layer's output and its activation function. So we follow each convolution layer with a `BatchNormalization` layer, and then a layer to apply the `ReLU` activation function (recall that batch normalization was designed to take place after a layer's outputs, but before the activation function).

```
def make_striding_batchnorm_cnn_model():
    model = Sequential()
    model.add(Conv2D(30, (5, 5), activation=None, padding='same',
                     strides=(2, 2),
                     input_shape=(1, image_height, image_width)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Conv2D(16, (3, 3), activation=None, padding='same',
                     strides=(2, 2)))
    model.add(BatchNormalization())
    model.add(Activation('relu'))
    model.add(Flatten())
    model.add(Dense(128, activation='relu'))
    model.add(Dense(64, activation='relu'))
    model.add(Dense(number_of_classes, activation='softmax'))
    model.compile(loss='categorical_crossentropy', optimizer='adam',
                  metrics=["accuracy"])
    return model
```

Listing B3-44: Code to make the striding-batchnorm CNN of Figure B3-35. Note that we place the `BatchNormalization` layer between the convolution layer and its `relu` activation function, placed on its own layer.

Listing B3-45 shows how we'd call this function to make a new model.

```
striding_batchnorm_cnn_model = make_striding_batchnorm_cnn_model()

striding_batchnorm_cnn_history = striding_batchnorm_cnn_model.fit(
                          X_train, y_train,
                          validation_split=0.25),
                          epochs=100, batch_size=256)
```

Listing B3-45: Code to build the striding-batchnorm CNN of Figure B3-35

The results are shown in Figure B3-36.

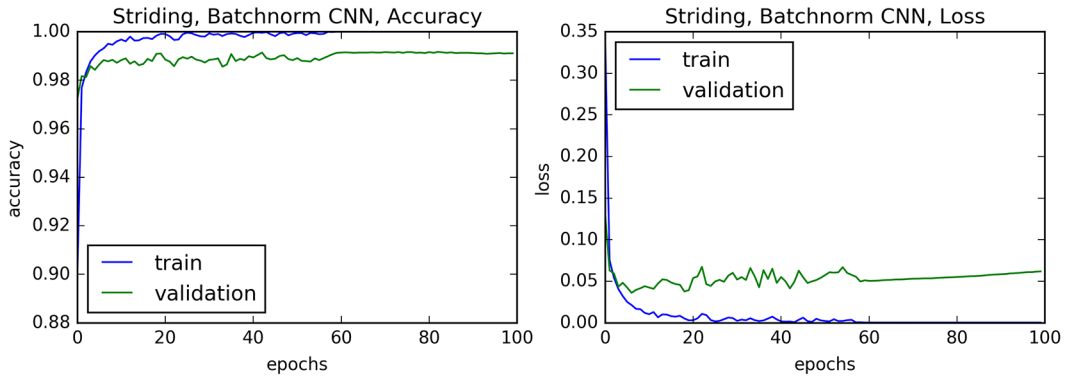


Figure B3-36: Accuracy and loss from training the striding-batchnorm CNN of Figure B3-35 for 100 epochs.

The numbers from the end of this run are in Listing B3-46.

```
Epoch 100/100
45s - loss: 2.6886e-04 - acc: 1.0000 - val_loss: 0.0618 - val_acc: 0.9911
```

Listing B3-46: The final lines from training the model of Figure B3-35

The validation accuracy is about the same as we got before, but we seem to have picked up a small amount of overfitting. The epochs also take about 25% longer to run.

The noise in the curves is a matter of some concern. We'd want to be careful when we stop training, to make sure we're not in one of those peaks in the validation loss (or the corresponding valleys in the validation accuracy). This is one of those times when it makes a lot of sense to keep multiple checkpoints, and then choose one based on looking at the performance graphs.

Overall, this variation doesn't seem to have given us anything better than before. This is the value of experimenting: until we try, we can't be sure how a network and a particular data set will behave.

Patterns

Historically, many CNNs were assembled by repeating a few types of recognizable clusters of layers [Karpathy16a]. Such a cluster is a set of convolution layers, followed by a pooling layer. This cluster is then repeated several times, perhaps with different parameters to the convolution layers. After that comes a series of fully-connected layers. Figure B3-37 shows an example of a such an architecture.

The pooling layers usually tile the input with 2 by 2 blocks. That is, the receptive field is 2 by 2, and we use a stride of (2,2) so that we produce a tiling with no overlaps or holes. This makes an output that has half the width and height of the input.

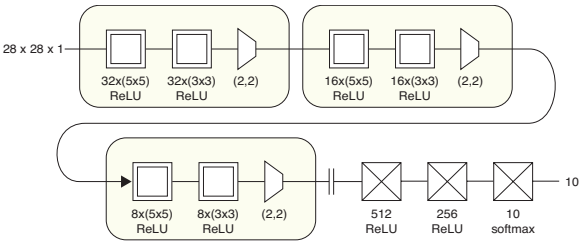


Figure B3-37: CNNs are often made up of repeating patterns. One popular pattern is a few convolution layers followed by a pooling layer, which we see here repeated three times. Typically, the parameters of the convolutions change from one repeat of the block to the next.

This network is a simplified version of the VGG16 network we've seen before, drawn here to demonstrate the idea of repeated units, but just for fun, let's run this on the MNIST data The results are shown in Figure B3-38.

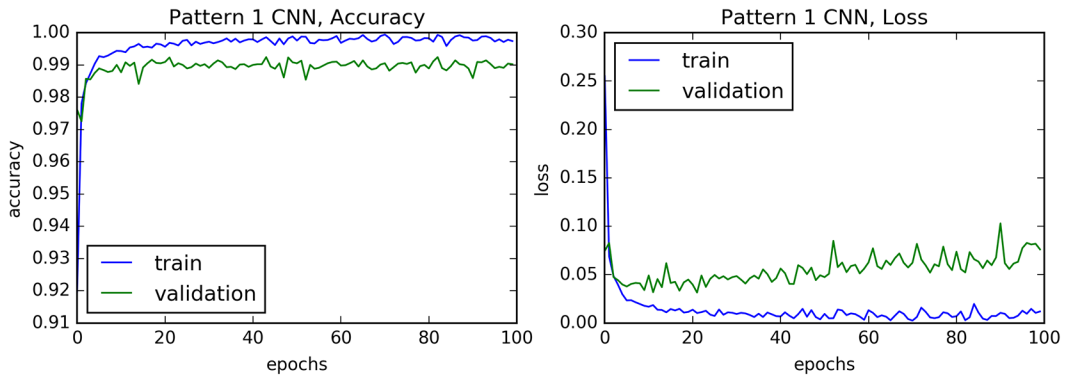


Figure B3-38: The architecture of Figure B3-37 evaluating MNIST data

The final lines from training are in Listing B3-47.

```
Epoch 100/100
339s - loss: 0.0119 - acc: 0.9974 - val_loss: 0.0760 - val_acc: 0.9902
```

Listing B3-47: The final lines from training the model of Figure B3-37

This isn't the best data we've seen, and there's some overfitting, but it's not bad for an essentially arbitrary network. It does take quite a while to run each epoch, as we might expect from all those layers.

As we did before, let's replace the pooling layers with striding in the final convolution layer of each set, as in Figure B3-39. We usually leave the stride of the other layers at 1. This is becoming a more attractive option as omitting the pooling layers gives us a smaller and faster network, and

when things are well-tuned there seems to be no loss of performance [Karpathy16a][Springenberg15]. The stride sizes are often (2,2), as they were for the pooling layers we're replacing.

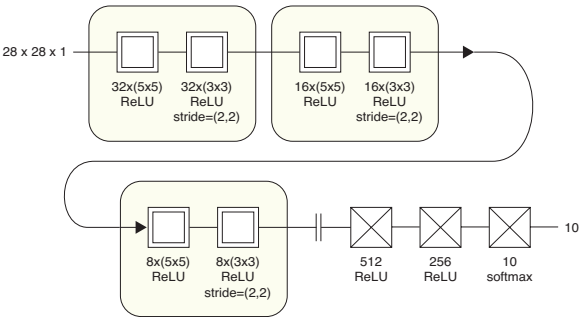


Figure B3-39: Replacing the pooling layers of Figure B3-37 with striding in the convolution layers

The results are shown in Figure B3-40.

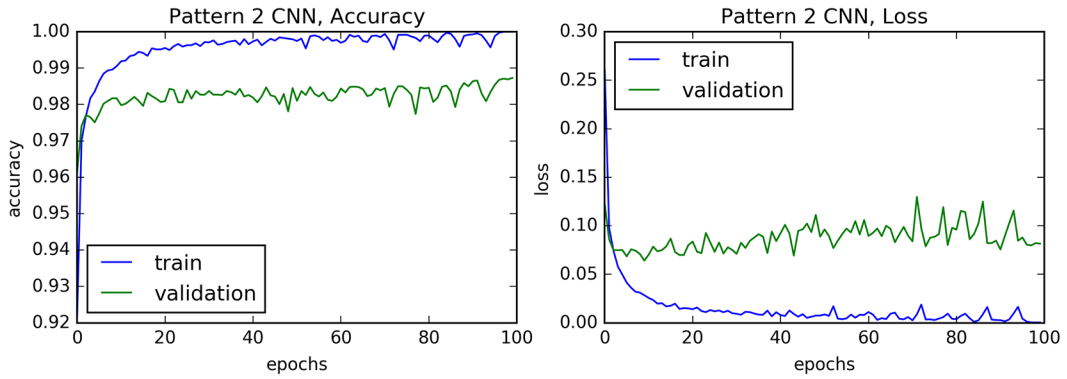


Figure B3-40: The architecture of Figure B3-39 evaluating MNIST data.

The final lines from training are in Listing B3-48.

```
Epoch 100/100
186s - loss: 2.8056e-04 - acc: 1.0000 - val_loss: 0.0815 - val_acc: 0.9873
```

Listing B3-48: The final lines from training the model of Figure B3-39

The results are just a little noisier than before, but they land in roughly the same place. We even cut the time per epoch roughly in half, suggesting that striding in the convolution layer is significantly faster than subsequent pooling. This makes sense, since convolution with striding means we apply the convolution filter less frequently, and we can skip the subsequent post processing step of pooling altogether.

Image Data Augmentation

One of the best ways to improve performance of any model is to give it as much training data as we can, while avoiding overfitting.

When we're working with images, we can easily create lots of new data by simply manipulating the images we already have, creating a wide variety of variations of the original. We could move each image left, right, up, or down, make it a little smaller or larger, rotate it clockwise or counter-clockwise by some amount, or perhaps flip it horizontally or vertically. Figure B3-41 shows some of these variations on an image of a Eurasian Eagle Owl.

Figure B3-41: Augmenting an image of a Eurasian Eagle Owl with rotations, flips, and scaling. The original is in the upper left. These transformations are deliberately extreme to show their effect.

The process of enlarging a dataset by creating variations is called *data amplification*, or *data augmentation*.

When we're working specifically with images, Keras provides a built-in object to perform data augmentation. It's called the `ImageDataGenerator`, and it performs all of the modifications we just mentioned, and a few others besides.

As its name suggests, this object is a "generator," which is a specific kind of object in the Python language [PythonWiki17]. In a nutshell, a generator can be thought of as a function that runs an internal loop, typically carrying out calculations and producing data. When that loop reaches a `yield` statement, the generator returns control to the routine that called it, with the argument to `yield` set to its value, just like a `return` statement. But if we call that function again, the loop picks up where it most recently stopped, and continues as though it had never been interrupted.

The `ImageDataGenerator` is set up this way because we can configure it to produce large numbers of variations on each of our input images. This can take a lot of time and a lot of computer memory. So rather than compute all the variations ahead of time and hold on to them until they're needed, the generator creates batches of images on demand. Each time we call the generator, it will produce and return another batch of images. The `fit()` method can use the generator as the source of training data, rather than

tensors that we pass in. The routine calls the generator over and over, each time it needs more data.

The transformations we applied to the images in Figure B3-41 were deliberately exaggerated to show their effects. In practice, we want to make new data that is close enough to the input to be plausible. After all, there's no reason to learn from distorted inputs that are not representative of the data we expect to see. In fact, that could hurt our ultimate performance, since some of the network's power would be uselessly directed to processing those inputs.

If we want to use our generated data again later, we can save time by telling the generator to read and write its images in a given directory. Then each time we ask for another batch of images, it reads and returns the saved, transformed files if they're available, or else generates them, saves them, and then returns them. This feature is also useful when we want to look at the generated files, to make sure we've selected the right options to get the sort of variations we're after. If disk space is precious and we're not pressed for time, we can just always skip the disk files altogether and make the transformed images fresh, on demand.

The `ImageDataGenerator` is a workhorse, capable of applying all sorts of transformations to our images. We only need to list the image-transforming operations we want when we build the object, and it will apply them all. We'll demonstrate the process with just a few transformations. The Keras documentation provides the complete list of all available options.

The overall process of setting up and using the generator takes only two steps. First, we create our `ImageDataGenerator` with the options we want. Second, we train our model. But instead of using `fit()` to start training, we use `fit_generator()`. These both take the same arguments with one exception: The first argument to `fit_generator()` is a function that returns batches of samples.

The usual function that we provide to `fit_generator()` is a function called `flow()`, which is automatically created for us as part of our `ImageDataGenerator` object. The metaphor is that the generator is producing a flow of data on demand, like water flowing out of a faucet when we turn the handle. Calling `flow()` provides a burst of training samples for our use.

Together, `fit_generator()` and `flow()` manage the production of batches of images, and presenting them to our model for training. Listing B3-49 shows a typical use of `ImageDataGenerator`.

```
# create the image generator with rotations and flips
image_generator = ImageDataGenerator(
                        rotation_range=100, horizontal_flip=True)

# fit our model using images produced by the image generator
model.fit(image_generator.flow(X_train, Y_train, batch_size=256),
                    seed=42, epochs=100,
                    samples_per_epoch=len(X_train))
```

Listing B3-49: Using an `ImageDataGenerator` to produce transformed images. Each image might be randomly flipped horizontally, and/or rotated up to 100 degrees in either direction.

A few images produced by an `ImageDataGenerator` using just a single sample from the MNIST training set are shown in Figure B3-42. In practice, for this data, we'd probably use much less rotation, and we wouldn't allow flips, since a mirror-image 2 isn't really a 2 at all. Some noise is visible around the edges where the low-resolution original image has been resampled.

Figure B3-42: Using an `ImageDataGenerator` to produce variations on a single MNIST sample for the number 2. For demonstration purposes we've allowed horizontal flipping, and a large range for rotation.

Normally, each run of this code will produce different results. For testing and debugging, it's often useful to get back the same sequence every time. We can force this by setting the `seed` argument when we call `fit()`, as we did above. This has the same purpose as setting the seed for a random number generator, which sets it up to always produce the same sequence of pseudo-random values.

In Listing B3-49 we also told `fit()` how many samples we want per batch, how many samples make up an epoch, and how many epochs we want.

It might seem odd to have to specify the number of samples per epoch, since until now the library has been able to infer that from the size of the input tensor. But the generator will just keep cranking out variations as long as we keep calling it, so there's really no sense of having run through "all the data," which is what we normally call an epoch. Yet epochs are important. For example, it's at the end of an epoch when statistics get collected and our callbacks are invoked. So we tell `fit()` how many images to generate before it simply declares that an epoch has passed. The value of `samples_per_epoch` has to be a multiple of `batch_size` or we'll get an error.

Synthetic Data

This section's notebook is Bonus03-Keras-4-CNN-Synthetic-Data.ipynb.

We usually build networks because we want to deploy them for real-world use, so we train them on real-world data. But testing and practice datasets are useful for helping us experiment with architectures, pre-processing strategies, and hyperparameters.

A great way to create an environment where we control everything is to train on our own data, which we generate on demand. Then we can make the data we want, rather than search for something out there that comes close.

We use the phrase *synthetic data* to describe data that we create ourselves, usually on the fly with an algorithm. We saw synthetic data in Bonus Chapter 1, when we used scikit-learn's built-in algorithms for making half-moons and blobs.

The great thing about generating synthetic data is that we can make as much of it as we want or need, and then train with it as usual.

It's conceptually easy to do this. Rather than modify an `ImageDataGenerator` object, which can require a lot of work, we subclass Python's built-in `Sequence` object, and equip it with a few standard routines such as `__len__` and `__getitem__`.

To demonstrate the idea, we've written a little routine that draws an image in a 64 by 64 square. There are five types of images: a Z shape, a plus sign, three vertical lines, a squared U, and a circle. Each time we draw one of these shapes we wiggle the points around a little, so that no two shapes are the same. The function returns the image it drew and the label. The label is a number from 0 to 4 that identifies which type of shape is in the image.

Figure B3-43 shows a random collection of these images. Note that these variations are performed on the points that make up the image, and are inherently different than what we get by applying the types of deformations (scaling, rotating, etc.) with an `ImageDataGenerator` object.

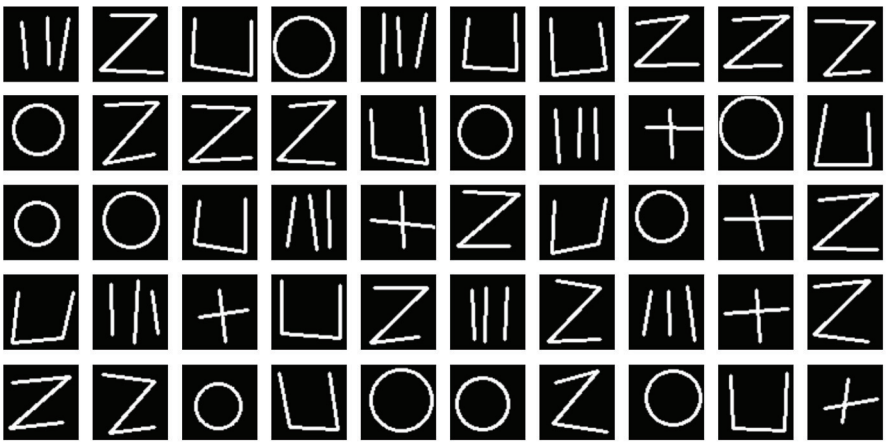


Figure B3-43: Synthetic images produced by a little routine. Each of the 5 types of images uses points (and in the case of the circle, a radius) that have been randomly perturbed from their starting positions.

We used this to train the simple convnet shown in Figure B3-44.

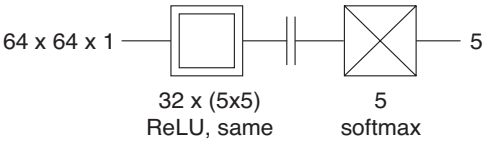


Figure B3-44: A simple convnet for classifying our synthetic data

The results are shown in Figure B3-45.

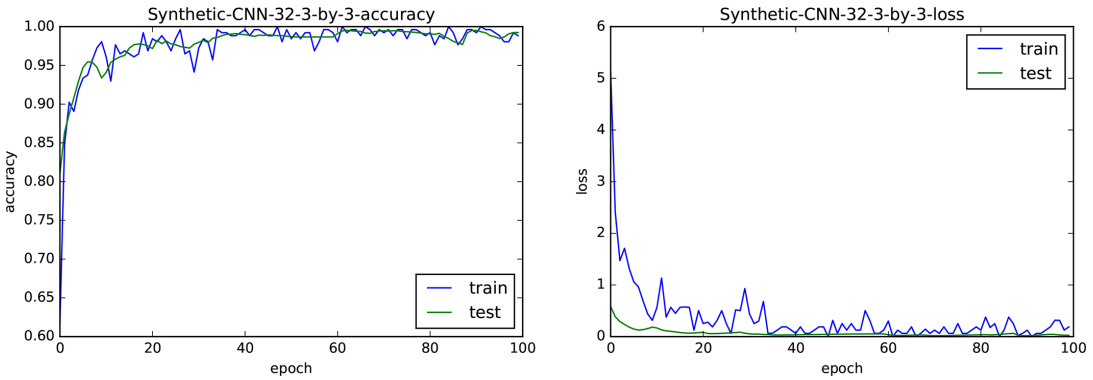


Figure B3-45: Accuracy and loss for our simple model on our synthetic data, generated on the fly

Despite being about as tiny as a classifier convnet can be, the network manages impressive accuracy on the test data, without obvious overfitting.

Parameter Searching for Convnets

Deep convnets can take a long time to train, particularly if we use big data sets. If we use the `GridSearchCV` or `RandomizedSearchCV` objects we used earlier to search for hyperparameters, we might not have enough compute power to produce results in a reasonable time.

There are faster alternatives. Unfortunately, they take some work to set up and use, so we won't go into them here. A good place to start for automatic parameter searching is the Spearmint project [Snoek16a][Snoek16b].

RNNs

This section's notebook is Bonus03-Keras-5-RNN.ipynb.

As we saw in Chapter 19, recurrent neural networks, or RNNs, are great for *sequential data*. The MNIST image data we've been using is not sequential, because there's no order to the images.

Sequential data, on the other hand, is inherently ordered. Classic examples are daily temperatures, the daily price of a stock, and the hourly height of a tide. There's also data that's ordered, but not in time, such as children lined up by height, shelved library books, and the colors of the rainbow.

In all of these phenomena, we want to use the information in the sequence of inputs to help us produce new output.

In RNN terminology, we still have a dataset made of samples, where each sample contains multiple features. But now each feature contains multiple values, called *time steps*. Recall that we can also think of "time steps" as "series of measurements for a given feature." Our example in Chapter 19 imagined a weather station on top of a mountain, taking multiple measurements every hour during 8 daytime hours. Each day's results make up a sample, and each type of measurement (such as temperature and wind speed) is a feature. Each feature contains 8 time steps, with one value for each hour.

Throughout this chapter we've been classifying the MNIST data. But the MNIST data has no sequential qualities, and our goal now is not classification, but prediction of the next value in a sequence. So in the next section we'll generate some new data of our own that we can use to show how to set up and run RNNs.

Generating Sequence Data

There are lots of sequential datasets available, but some of them are complicated or hard to draw. So let's make our own simple dataset that we can easily draw and interpret.

We'll just add up a bunch of sine waves, such as that at the top of Figure B3-46. Each sine wave has a frequency, an amplitude, and a phase (or offset). We'll write a routine that takes lists of these values, called `freqs`, `amps`, and `phases` respectively, and uses them to add up all the waves at many points. Figure B3-46 shows the idea.

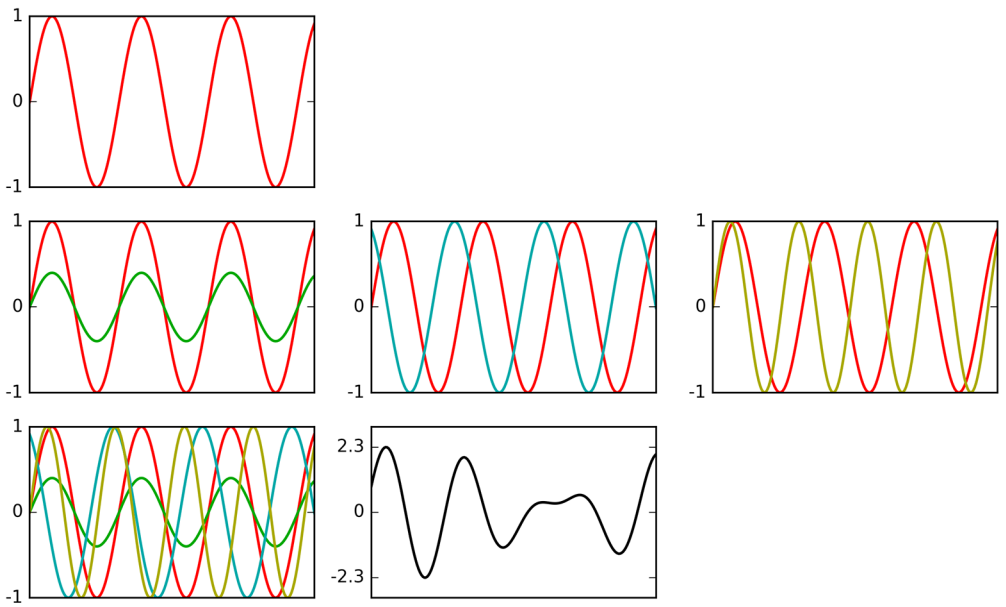


Figure B3-46: By adding up lots of sine waves we can make an interesting curve.

The top row of Figure B3-46 shows a single sine wave. On the left of the second row, we see our original wave in red, and another wave in green with smaller amplitude. In the center of the second row, we see our original wave in red, and another wave in blue with a different phase. And at the right end of the second row, we see our original wave in red, and another wave in yellow with a different frequency. The left image in the bottom row shows all four waves superimposed, and the one to its right shows all four waves added together. This is sine-wave-ish, but more interesting.

Our little curve-building routine takes three other arguments. The first is an integer called `number_of_steps` which tells it how many points to generate. The second is a float called `d_theta` that tells the routine the spacing of the samples (the name comes from thinking of the sine wave as based on an angle, which is often written with the lower-case Greek letter θ (theta). Finally, `skip_steps` is an integer that provides an offset to the starting point, so we don't always begin at 0. This is useful for creating the test data, which can start far to the right of the training data.

The routine `sum_of_sines()` is shown in Listing B3-50. We wrote it to emphasize clarity. Since this is plain Python programming, and nothing specific to machine learning, we won't go into the details.

```
def sum_of_sines(number_of_steps, d_theta, skip_steps, freqs, amps, phases):
    '''Add together multiple sine waves and return a list of values that is
    number_of_steps long. d_theta is the step (in radians) between samples.
    skip_steps determines the start of the sequence. The lists freqs, amps,
    and phases should all the same length (but we don't check!)'''
    values = []
    for step_num in range(number_of_steps):
        angle = d_theta * (step_num + skip_steps)
        sum = 0
        for wave in range(len(freqs)):
            y = amps[wave] * math.sin(freqs[wave]*(phases[wave] + angle))
            sum += y
        values.append(sum)
    return np.array(values)
```

Listing B3-50: A little routine to create a list that holds the sum of multiple, different sine waves

We'll use this routine to generate two different data sets, which we'll call data set 0 and data set 1. The values we used to construct them were found by trial and error until they produced one graph that felt "calm," so it wouldn't be too hard to predict, and one that felt "busy," for a harder challenge.

We'll consider the data we use for evaluation to be a test set, which is used just once, rather than a validation set, which can be used multiple times when evaluating different forms of the network.

Data set 0 is a gentle sum of two waves. We made one wave twice the speed of the other by setting `freqs` to `(1,2)`, the second wave twice as high as the first by setting `amps` to `(1,2)`, and started both waves at 0 by setting `phases` to `(0,0)`, Our training data came from using 200 steps (`number_of_steps` = 200), a step of about 0.057 radians (`d_theta` = 0.057), and no offset (`skip_steps` = 0). We chose the weird step size by eye so that 200 samples produced what we felt was a good amount of data for an easy test case.

The training set is 200 samples long, starting at 0. The test set is another 200 steps, starting far to the right of the training set. Listing B3-51 shows the calls to make this data set.

```
train_sequence_0 = sum_of_sines(200, 0.057, 0, [1, 2], [1, 2], [0, 0])

test_sequence_0 = sum_of_sines(200, 0.057, 400, [1, 2], [1, 2], [0, 0])
```

Listing B3-51: Creating data set 0

The resulting training and test data are shown in Figure B3-47.

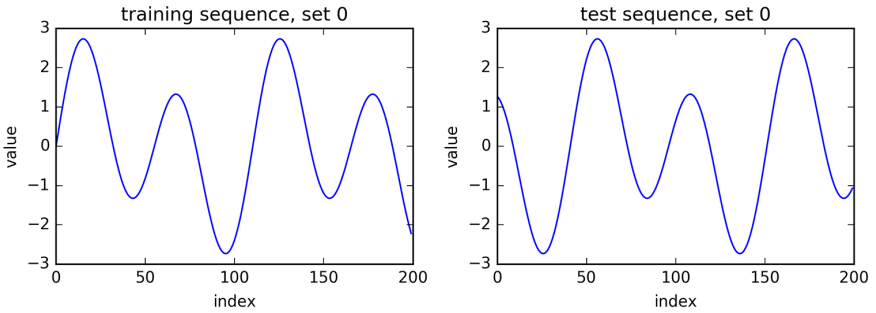


Figure B3-47: The training and test data for our first set of sine waves

Data set 1 is a harder challenge that uses 4 waves. For this set, we set `freqs` to `(1.1, 1.7, 3.1, 7)`, `amps` to `(1, 2, 2, 3)`, and we again left all the phases at 0, so `phases` is `(0,0,0,0)`. The weird frequencies are chosen so that the pattern won't repeat for tens of thousands of samples. The other variables are the same as for data set 0. Listing B3-52 shows the calls to make the data.

```
train_sequence_1 = sum_of_sines(200, 0.057 ,0, [1.1, 1.7, 3.1, 7],
                                [1, 2, 2, 3],  [0, 0, 0, 0])

test_sequence_1 = sum_of_sines(200, 0.057, 400, [1.1, 1.7, 3.1, 7],
                               [1, 2, 2, 3],  [0, 0, 0, 0])
```

Listing B3-52: Creating data set 1

The results are shown in Figure B3-48.

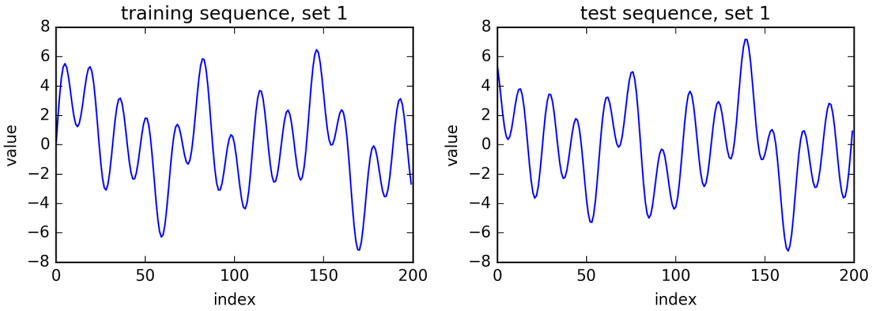


Figure B3-48: The training and test data for our second, busier set of sine waves

RNN Data Preparation

The mechanics for preparing data for RNNs in Keras are a little more complicated than what we've been working with so far, because we have to carry out a couple of reshaping steps in order to use all the library routines we want. We also need to extract our little windowed sublists, which we have to do ourselves since there aren't any library routines to do it for us.

Let's dig into the steps and knock them down one by one in order.

As before, we want to normalize our data to get it into the range [0,1]. The `MinMaxScaler` from scikit-learn is the perfect tool for the job. But recall from Bonus Chapter 1 that this routine expects our features to be arranged vertically, as in Figure B3-49.

	Temperature	Rainfall	Wind Speed	Humidity
June 3	60	0.2	4	0.1
June 6	75	0	8	0.05
June 9	70	0.1	12	0.2
	↓	↓	↓	↓
	[60, 75]	[0, 0.2]	[4, 12]	[0.05, 0.2]
	↓	↓	↓	↓
new June 3	0	1	0	0.33
new June 6	1	0	0.5	0
new June 9	0.66	0.5	1	1

Figure B3-49: The `MinMaxScaler`, like most feature-wise normalizers, reads all the values for each feature, finds the minimum and maximum, and re-scales the data to the range [0,1]. Each feature (a column in this example) is scaled independently (this is a variant of Figure 12-37).

Our sine wave data has only one feature, with many time steps, and it's a 1D list (that is, it's not a column as `MinMaxScaler` is expecting). So let's reshape our data into a column. In Python, that means making a 2D grid that is as tall as our collection of measurements, and just one element wide, as in Figure B3-50.

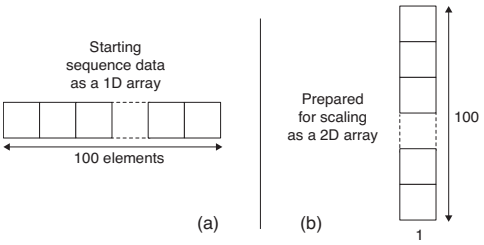


Figure B3-50: Reshaping our list of sine wave values as a 2D grid with one column

We can use `reshape()` to make this, as shown in in Listing B3-53, where `train_sequence` and `test_sequence` can be the corresponding variables from either data set 0 or 2.

```
train_sequence = np.reshape(train_sequence, (train_sequence.shape[0], 1))
test_sequence = np.reshape(test_sequence, (test_sequence.shape[0], 1))
```

Listing B3-53: Our input data contained in two 1D lists called `train_sequence` and `test_sequence`. To prepare them for `MinMaxScaler` we turn each list into a column of elements.

Now that we have our data in the right format to give to `MinMaxScaler`, we'll make an instance of that object and then call its `fit()` routine on the training data. It will find the minimum and maximum values, and remember them. Then, as usual, we apply the transformation to both the training and test data by calling the scaler's `transform()` method, as in Listing B3-54. To keep things clear, we'll give the results new names, prefixed with `scaled_`.

```
from sklearn.preprocessing import MinMaxScaler

min_max_scaler = MinMaxScaler(feature_range=(0, 1))
min_max_scaler.fit(train_sequence)
scaled_train_sequence = min_max_scaler.transform(train_sequence)
scaled_test_sequence = min_max_scaler.transform(test_sequence)
```

Listing B3-54: We make our transformation from the training data, and then apply it to both the training and test data, which we save in new variables.

Note that, as usual, we first fit the scaling object to the training data, and then applied that transformation to the test (or validation) data. Our object `min_max_scaler` remembers its transformation, so we'll later be able to apply its inverse to the output of our network, giving us a result in the same range as the input.

Now that our data is normalized, it's time to create the little windowed sublists that make up our training and test data. Figure B3-51 shows the idea.

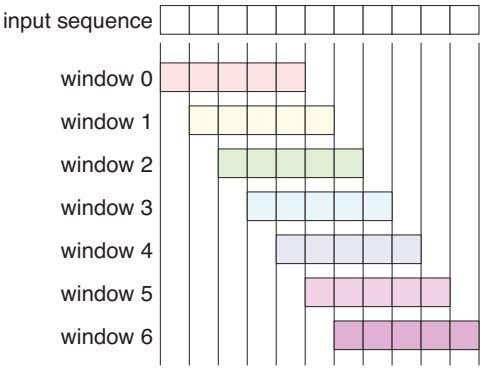


Figure B3-51: Chopping up an input sequence into a series of sub-lists. Each sub-list has the same length, called the window length. In this example, the windows are overlapping, with each one starting just one element to the right of the previous window.

Once we have the windows, we can split them into the sample, which will be everything but the final value, and the target, which is that final value, as in Figure B3-52.

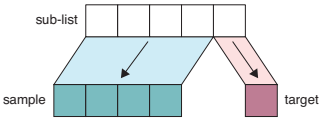


Figure B3-52: We break up each windowed list into all the elements but the last, which make up our sample, and the last element, which is the target.

Building these windows is a common operation when working with RNNs. There are probably a million ways to write a routine to do this job. We'll present a little version that emphasizes clarity. It moves the starting point of a window of the given size from the start of the sequence to a bit before the end, stopping at the last position where the whole window still fits inside the input sequence. Because this is straight Python programming, and doesn't have anything to do with machine learning, we won't go deeper into the details. Listing B3-55 shows our routine.

```
def samples_and_targets_from_sequence(sequence, window_size):
    '''Return lists of samples and targets built from overlapping
    windows of the given size. Windows start at the beginning of
    the input sequence and move right by 1 element.'''
    samples = []
    targets = []
    for i in range(sequence.shape[0]-window_size): # i is starting position
        sample = sequence[i:i+window_size]         # sub-list of elements
        target = sequence[i+window_size]           # next element
        samples.append(sample)                     # append sample to list
```

```
        targets.append(target[0])                  # append target to list
    return (np.array(samples), np.array(targets))  # return as Numpy arrays
```

Listing B3-55: A Python routine to convert a list of values and a window size into two new lists. The first contains multiple overlapping sub-sequences from the original list. Each sub-sequence is 1 less than the given value of `window_size`. The second list contains the next value in the original sequence, which will be our target when training and testing.

We can now create our training and test data just by handing our scaled sequences to this routine. Listing B3-56 shows the code. As before, we'll assign the windowed training data to `X_train` and `y_train` and the windowed testing data to `X_test` and `y_test`. We'll assume that the integer variable `window_size` has been set.

```
(X_train, y_train) = samples_and_targets_from_sequence(
                                  scaled_train_sequence, window_size)
(X_test, y_test) = samples_and_targets_from_sequence(
                                  scaled_test_sequence, window_size)
```

Listing B3-56: How to create our training and test data using the utility function in Listing B3-55

Now that we have our data, we have to make sure it has the necessary *shape*.

Getting the shape of the data into the right form for the network is as essential for RNNs as it is for all the other types of networks we've seen. If we organize the data in a way that doesn't match what the network is expecting, we'll typically either get an error, or our network will go haywire and produce crazy results.

The good news is that we've already accomplished that mission, because we wrote the routine `samples_and_targets_from_sequence` to return its data in the shapes that we need for RNN training in Keras.

Let's look at those shapes, so it's clear how the data is structured.

The easy part is the targets, which we're saving in `y_train` and `y_test`. These are just 1D lists.

The training and test data that we're saving in `X_train` and `X_test` are 3D blocks. The `X_train` block is as deep as the number of windows that we were able to make (that is, the number of samples), and as tall as the window size itself (that is, the number of time steps). The block is as wide as the number of features we're learning. Since we have only 1 feature in this dataset, the block is only 1 element wide.

This is the structure that we want for training RNNs in Keras. The depth of the block tells us the number of samples we'll train with. The time steps are arranged vertically, and the features horizontally. Figure B3-53 breaks down this shaping for an imaginary data set with 3 samples, each made of 2 features, each with 7 time steps.

In Figure B3-53, we have 3 samples, each containing 2 features, which in turn hold 7 time steps. In part (a) we see the `X_train` data set ready for learning by an RNN. Part (b) shows that the first sample from `X_train` is the slice of the block that is closest to us. Here it's the elements labeled A through G. This can be represented as a 2D grid. Part (c) shows that the

first feature in this sample is located in the leftmost column. In part (d), we see that the elements inside that column are the time steps corresponding to that feature.

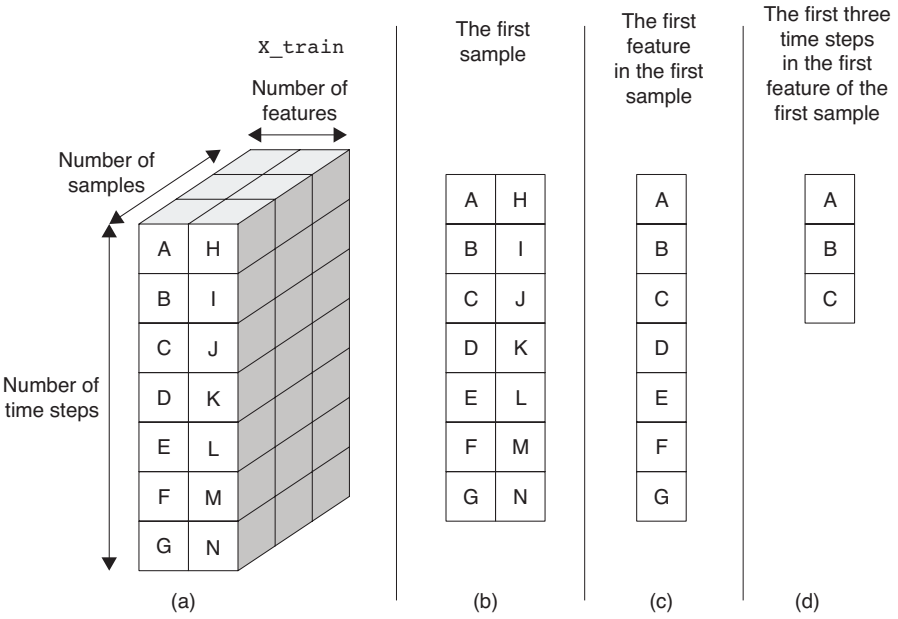


Figure B3-53: The structure of data prepared for RNN training

As we mentioned, we set up our pre-processing so that we now have our data in the proper shape, as shown in Figure B3-54.

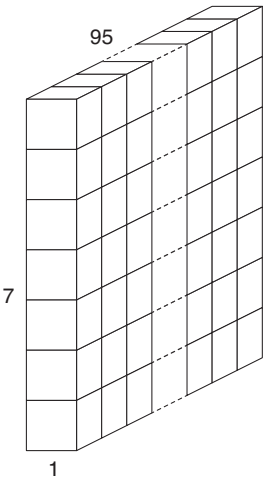


Figure B3-54: The shape of our training data, assuming we have 95 samples and a window size of 7. The test data has a similar shape, though fewer samples.

This wraps up the pre-processing of our data so it's ready for training.

Building, Compiling, and Running the RNN

Now that our data is properly normalized and structured, we can create the network and train it.

We'll create an extremely simple RNN that runs quickly, yet still demonstrates all the basic principles. We'll have one recurrent layer, followed by one dense layer. Figure B3-55 shows the architecture. Note that the dense layer has no activation function listed. That's because we don't want it to modify the value it computes, since that value is our prediction. We can either say that we've left off the activation function, or we've set it to the linear function.

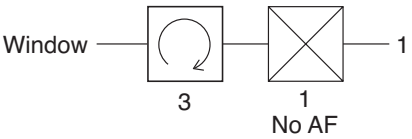


Figure B3-55: Our first RNN will have 1 recurrent layer with 3 elements of memory, followed by a fully-connected layer with one unit. The input contains as many time steps as are in our window. The dense layer has no activation function.

Recall from Chapter 19 that there are two standard types of building blocks used in RNNs: the LSTM (Long Short-Term Memory), and GRU (Gated Recurrent Unit). Keras supports both of these, and the layers are named for the unit they use.

Let's use an LSTM. An RNN icon, as in Figure B3-55, represents an LSTM unit unless explicitly marked otherwise.

To create an LSTM layer, we just create an `LSTM` object with the options we want, and then put it into our model with `add()` as usual.

How much memory should be in the state for this layer? Let's arbitrarily start with just 3 elements.

When we make an `LSTM` layer, we specify the number of cells we want. Because this will be our first layer of the network, we also have to supply the input dimensions. As usual, we set the argument `input_shape` to the shape of one sample. From our discussion above, we know that each sample is a 2D grid whose height is the number of time steps (that's the height of our window), and whose width is the number of features (we have just 1), as we saw in Figure B3-53.

Listing B3-57 shows how to create this layer.

```
lstm_layer = LSTM(3, input_shape=[window_size, 1])
```

Listing B3-57: How to create an `LSTM` layer object. The first argument is the number of LSTM cells in the layer. The second argument, `input_shape`, tells the size of a single sample.

We follow this up with a dense layer of a single neuron. If we don't specify an activation function in a `Dense` layer, Keras defaults to `None`. This is good for us in this situation, because we don't want the output to be squashed down to the range [0,1] or [–1,1] or any other range. Just to be explicit, we'll include a redundant assignment of `None` to the activation function.

The complete model is shown in Listing B3-58.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(3, input_shape=[window_size, 1]))
model.add(Dense(1, activation=None))
```

Listing B3-58: How to create our RNN model. We just create the model as a `Sequential` object as usual, add in our RNN layer (in this case, an `LSTM` object), and then we place a one-neuron `Dense` layer at the end.

That's it for building our model. Now we just compile it and run.

As usual, to compile the model we need to supply a loss function and an optimizer. We've been using the Adam optimizer in this chapter and it's been working great, so let's keep using it. For the loss function, we don't want to use the same categorization function we used earlier, because we're no longer doing categorization. What we want is something that will compare the single value that comes out of our network with the target value for that sample. Consulting the list of loss functions in the Keras documentation, we can see that the `mean_squared_error` loss function does the job, so let's use that.

The compilation step is shown in Listing B3-59.

```
model.compile(loss='mean_squared_error', optimizer='adam')
```

Listing B3-59: How to compile our RNN. We use the `'adam'` optimizer as before. We've chosen to use the `'mean_squared_error'` loss function which is appropriate for this RNN.

Now that our model is compiled, we train it just like every other model by calling `fit()`. The only change we'll make here is to use a batch size of 1, because some experimentation showed that for this tiny network and this small dataset, that produced the best results. Listing B3-60 shows our command to train the network.

```
history = model.fit(X_train, y_train, epochs=number_of_epochs,
                    batch_size=1, verbose=2)
```

Listing B3-60: Training our RNN with `fit()`. This is like all of our other training steps, except we've set `batch_size` to 1.

Bringing these steps together gives us the code in Listing B3-61. This builds our RNN, and then trains it for whatever number of epochs we choose to save in the variable `number_of_epochs`.

```
# create and fit the LSTM network
model = Sequential()
model.add(LSTM(3, input_shape=[window_size, 1]))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

history = model.fit(X_train, y_train, epochs=number_of_epochs,
                    batch_size=1, verbose=2)
```

Listing B3-61: Building and training our RNN

Now that we have our model trained, we can ask it for its predictions. In Listing B3-62 we compute predictions for both the training and test data.

```
y_train_predict = model.predict(X_train)
y_test_predict = model.predict(X_test)
```

Listing B3-62: We get predictions from our model by handing the sample data to the model's `predict()` method, as usual.

Before we look at the results, we should mention that when we use the network to make predictions, we rarely use the results directly.

The issue is that we've applied the `MinMaxScaler` to our input data (both training and test), transforming it from our original values to a range more suitable for training. That means that *the network's predictions are also transformed*. For example, if our original data was in the range [–6,6], then after the transformation it will be in the range [0,1]. This means that the predictions will be roughly in the range [0,1] as well.

Because of this, we cannot directly compare our predictions with our original data. In the case of a simple scaling, like we're doing here, that's not a major issue. But if we perform a more complicated processing step, then it could be very hard to mentally interpret the predicted data.

As we saw in Chapter 10, the general solution is to *inverse-transform* the predicted data. Like most of scikit-learn's transformation routines, `MinMaxScaler` come with a method called `inverse_transform()` that does just this.

In Listing B3-63 we use the `inverse_transform()` routine of `min_max_scaler` (our `MinMaxScaler` object) to un-transform our predictions. We'll also invert the transform on our training and test targets.

```
# inverse-transform original targets
inverse_y_train = min_max_scaler.inverse_transform([y_train])
inverse_y_test = min_max_scaler.inverse_transform([y_test])

# inverse-transform predictions
inverse_y_train_predict = min_max_scaler.inverse_transform(y_train_predict)
inverse_y_test_predict = min_max_scaler.inverse_transform(y_test_predict)
```

Listing B3-63: We invert both the original target data and the predicted targets for both the training and test sets. This undoes the scaling operation we performed using the `transform()` method of `min_max_scaler()`.

Inverting (that is, un-transforming) the original targets `y_train` and `y_test` may seem wasteful. Why not simply save the original targets in their un-transformed form, and use them here?

Let's look again at the last two lines of Listing B3-54, repeated here as Listing B3-64.

```
scaled_train_sequence = min_max_scaler.transform(train_sequence)
scaled_test_sequence = min_max_scaler.transform(test_sequence)
```

Listing B3-64: A repeat of the last two lines of Listing B3-54, where we applied transformations to our data

We can see that we transformed the entire windowed sequence before we split it into a sample and a label, so we never really had the labels `y_train` and `y_test` sitting around in non-transformed variables before.

We structured the code this way for clarity and simplicity. It's certainly reasonable to extract and save the labels before they're transformed. Then we wouldn't need to undo the transformation here. Either way works. We'll stick with the version we just presented.

Now that we have the predictions back in the original range of the data, we can plot them with the original data and see how good our predictions are. We can also use them to get a quick numerical summary of accuracy using a measure called the *root mean squared error*, or *RMS error*. This is a standard way to measure error that lets us compare apples to apples when we look at multiple networks. It's close to what the `'mean_squared_error'` loss function is computing, except that we include a square root. To compute this, we use the square-root routine `sqrt()` from the `math` module, and the `mean_squared_error()` routine from scikit-learn. Listing B3-65 shows the steps.

```
from sklearn.metrics import mean_squared_error

trainScore = math.sqrt(mean_squared_error(inverse_trainY[0],
                                          inverse_y_train_predict[:,0]))
print('Training RMS error: {:.2f}'.format(trainScore))

testScore = math.sqrt(mean_squared_error(inverse_testY[0],
                                         inverse_y_test_predict[:,0]))
print('Test RMS error: {:.2f}'.format(testScore))
```

Listing B3-65: Computing and reporting the root-mean-squared (RMS) error for our training and test predictions.

The funny indexing comes from the different shapes of the original targets and their predictions. In our code, `inverse_y_train` and `inverse_y_test` are 2D grids with one row (for example, if there are 30 targets in each set, their shapes would be 1 by 30), so we get a list containing the data stored in the first (and only) row by selecting `inverse_y_train[0]`. On the other hand, the predictions coming back from `model.predict()` are 2D grids that have one column (so they would be 30 by 1). We get a list of the data in the first (and only) column by selecting `inverse_y_train_predict[:,0]`.

Issues like getting these indices right are frequently hard to anticipate, so we often discover them only when we write some code that seems reasonable but then messes up. Using interpreted Python interactively lets us examine the shapes of our variables step by step and line by line, and develop the proper adjustments and selections to select the data we want at each step.

Analyzing RNN Performance

Let's try out our sine wave data on this tiny RNN. We'll start with the easier, first data set in Figure B3-46, shown here again as Figure B3-56.

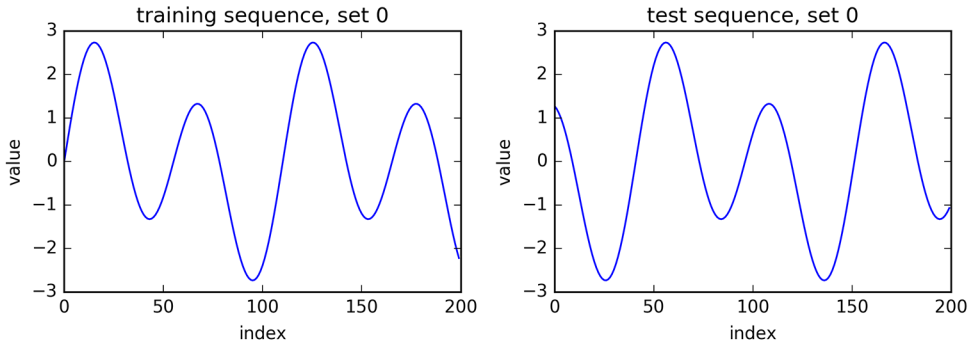


Figure B3-56: The "calm" data set. This figure is a repeat of Figure B3-46. There are 200 data points in this set.

Recall that our data-generating routine with the ungainly name `samples_and_targets_from_sequence()`, takes an argument called `window_size` that lets us specify how many time steps are to be used in each sample.

Let's arbitrarily start with a window size of 3 steps. This means each sample will have 3 values, and we'll ask the network to predict the one that comes after. We'll always provide that value as the target, so during training the system can learn to match that value, and during testing we can see how well we did.

As usual, we'll train for 100 epochs. After each epoch, we get back a single number that tells us our loss, measured by the difference between the value we predict and the target.

The loss is plotted in Figure B3-57.

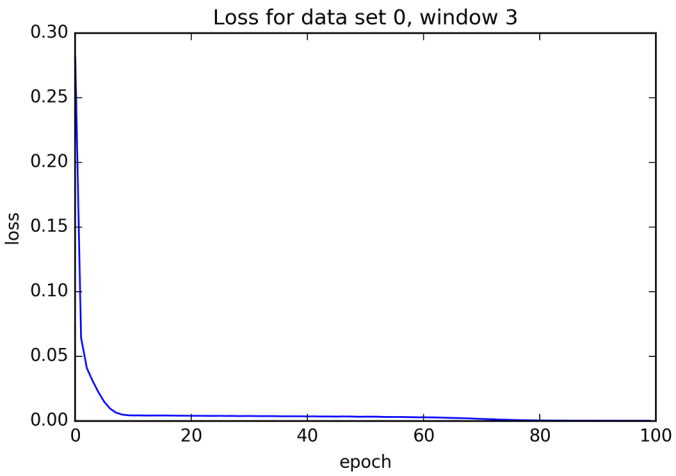


Figure B3-57: The loss from training 100 epochs of our RNN on data set 0 with a window of 3 time steps

The loss drops quickly to about 0.06, then more gently to something close to zero around epoch 8, and then over the next 60 epochs or so it continues to drop, finally hitting a value visually hard to distinguish from zero at around epoch 80.

Let's draw our predictions on top of our data so we can eyeball our performance.

Figure B3-58 shows our training data in black, and the predictions in red. Recall that we're using 200 samples in our training set.

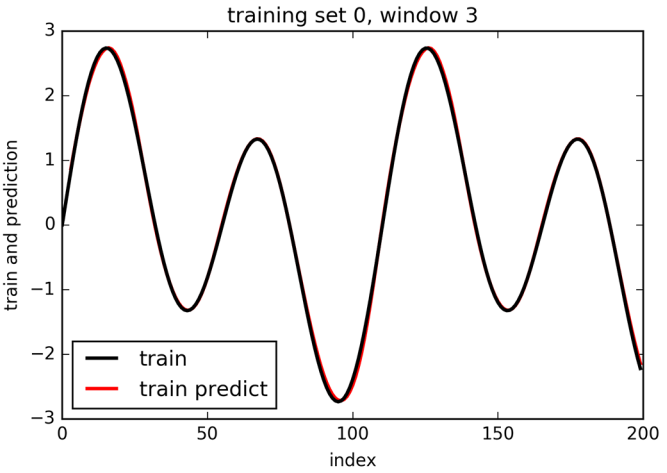


Figure B3-58: Our training data for set 0 is shown in black. Overlaid on that are the predictions from our RNN after 100 epochs of training with a window of 3 time steps.

This is pretty spectacular for a network with only 1 LSTM unit with 3 elements of state and 1 neuron after that. The match between the predictions and the real values isn't perfect, as we can see near the tops of the hills and bottoms of the valleys, but it's pretty great.

The predictions start slightly after the start of the training data, because the first prediction is the 4th element of the training data. This is hard to see in this figure, but will be easier to spot in later results (such as Figure B3-70).

Let's now looks at the test data. Figure B3-59 shows our test data in black, and the predicted data in red. Recall that we're also using 200 samples in our test set.

The test data looks similar to the training data because it's made from the same repeating sine waves, just located later in the sequence. The test predictions are close, again messing up near the extremes of the hills and valleys.

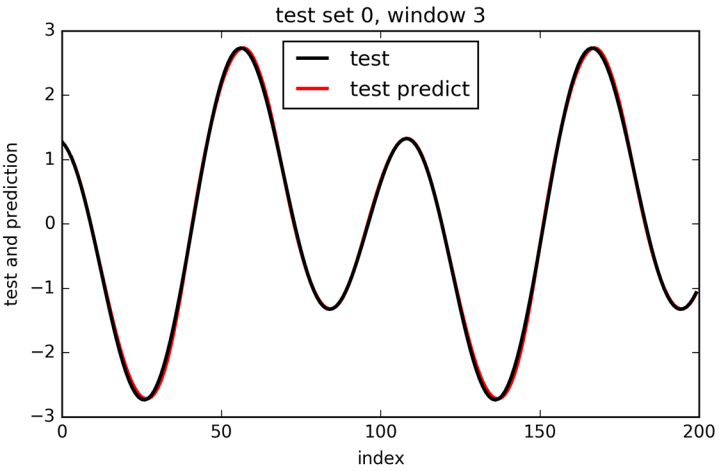


Figure B3-59: Our RNN's predictions for the test data of set 0, after 100 epochs of training with a window of 3 time steps

Maybe we don't even need 3 samples in our window. What if we try a window of just 1 sample? So we give the network a single value, and ask it to predict the next one. This only has even a hope of working because our dataset probably doesn't have any numbers that repeat exactly. So if it can learn the value that comes after each value in the training set, it should be able to reproduce those numbers. The loss is plotted in Figure B3-60.

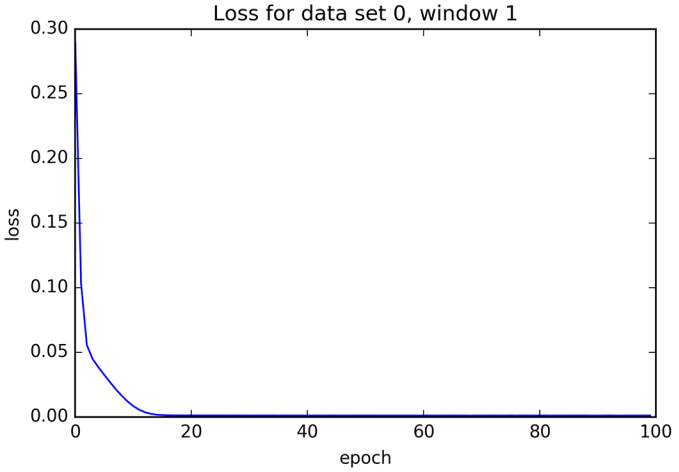


Figure B3-60: The loss from training 100 epochs of our RNN on data set 0 with a window of 1 time step

Figure B3-61 shows its predictions on our test data.

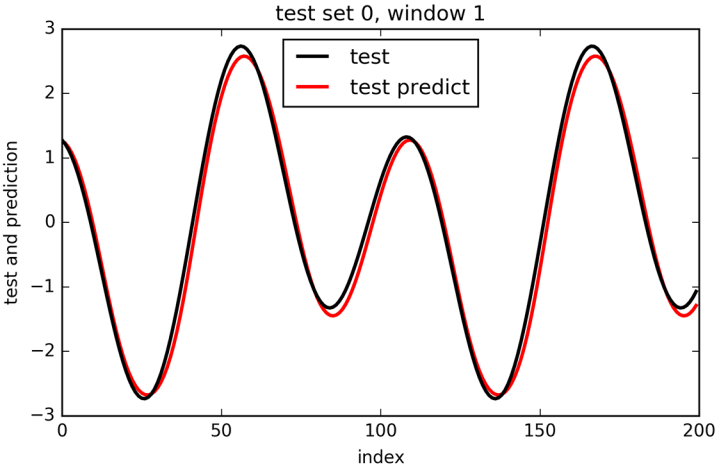


Figure B3-61: Our RNN's predictions for the test data from set 0, after 100 epochs of training with a window of 1 time step

We've clearly lost considerable performance, but the network is doing a great job at memorizing input/output pairs.

Even though 3 steps did a good job predicting our test data, let's go the other way and crank up our window size up to 5 time steps. Since we're just trying to get a feeling for things now, rather than carry out a detailed analysis, we'll skip the curve showing the training predictions, and go straight to the loss curve and test predictions. Figure B3-62 shows the loss for a window of 5 time steps.

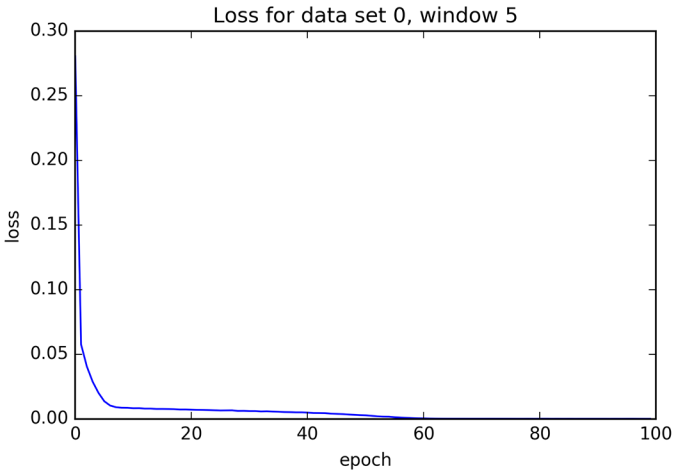


Figure B3-62: The test set loss from training 100 epochs of our RNN on data set 0 with a window of 5 time steps

Figure B3-63 shows our test predictions for a window of 5 steps.

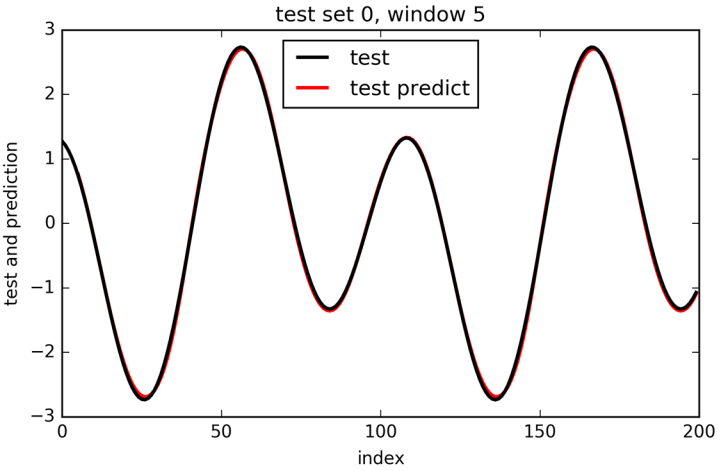


Figure B3-63: Our RNN's predictions for the test data from set 0, after 100 epochs of training with a window of 5 time steps

Visually, this looks a lot like our window of 3 steps in Figure B3-59. Perhaps a window of 3 pieces of data was enough for the network to do a really good job of predicting the 4th.

Let's try the more complicated data in our second test set, shown in Figure B3-48, repeated here in Figure B3-64.

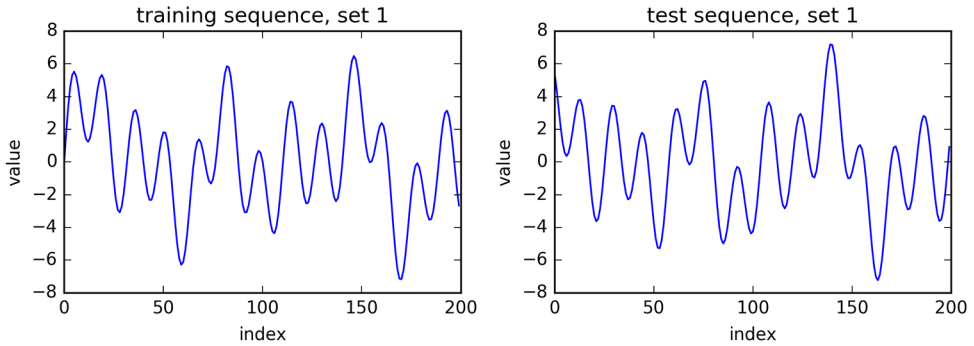


Figure B3-64: Data set 1. This figure is a repeat of Figure B3-48.

Since a window of 3 worked well for the simple data, let's try that again. The test set loss is plotted in Figure B3-65.

As with the first data set, the loss plunges at the start and then slows its descent. There's a knee around epoch 4 and a more gradual one around epoch 50, until the hits zero around epoch 80. This suggests that this test set is harder for our tiny network to learn than the last one, which makes sense.

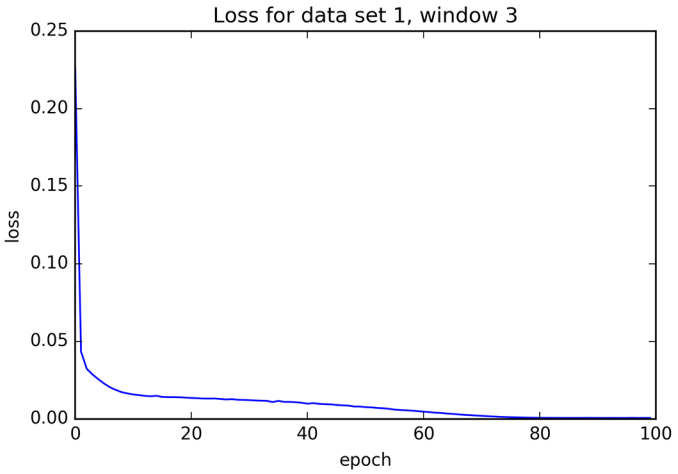


Figure B3-65: The test set loss from training 100 epochs of our RNN on data set 2 with a window of 3 times steps

Let's look at how well this window of 3 matches the test data. Figure B3-66 shows our results.

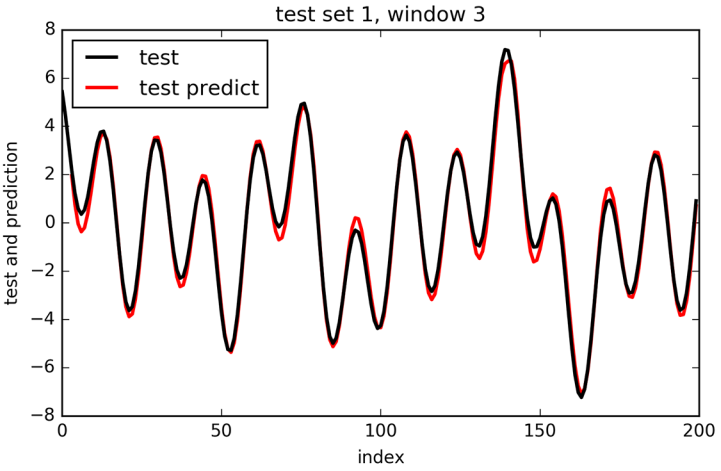


Figure B3-66: Our RNN's predictions for the test data of set 2 after 100 epochs of training with a window size of 3 time steps

This is a pretty great match for such a tiny network and such a wiggly set of data, particularly with such small windows. Let's try a window of 5 elements, shown in Figure B3-67.

Wow. Increasing the window from 3 to 5 seems to have mostly backfired, though in some places, like the peak around epoch 140, the match improved.

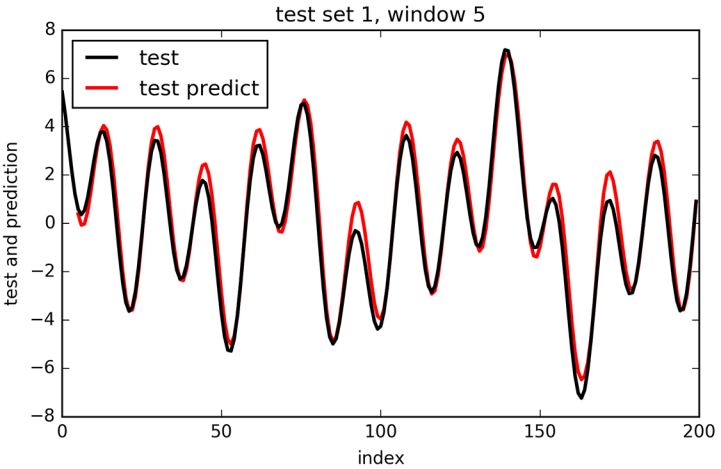


Figure B3-67: The predictions for the complex dataset with a window of 5 steps

Looking back over these results, our tiny network of just 1 LSTM unit with 3 steps of memory, and 1 final neuron, did a great job on both test sets. The window size made a difference. For these simple datasets, a window of 3 or 5 seemed to usually do a very good job. As we'll see later, more complicated datasets will often need larger windows.

A More Complex Dataset

We used only one recurrent layer so far because it worked so well. But we can build *deep recurrent networks* by simply adding in more recurrent layers. Depending on the data, it may be best to have just a few recurrent layers, each with lots of units of state memory, or it might be better to have many layers, each with just a small amount of memory.

We can make a small change to our sine-wave data to make it much more challenging for an RNN to learn: any time the curve is heading downwards, we'll flip it around the X axis so that it's heading upwards. Our curve will change from being smooth to choppy, with abrupt jumps. This is a completely arbitrary operation that we're applying to create a more challenging dataset for our networks.

Figure B3-68 shows this operation applied to the training data for our second set of waves, creating a third set of data, which we'll call data set 2.

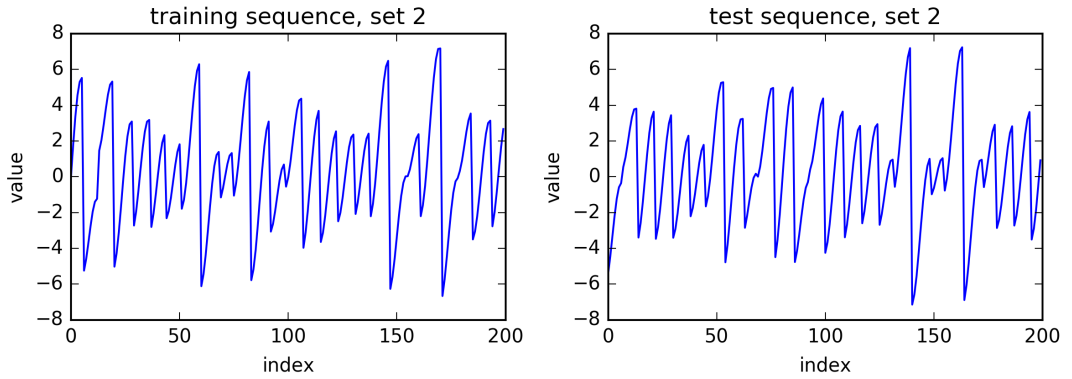


Figure B3-68: Creating a third set of data by starting with our second set of training data, only any time the curve starts to head downwards, we reflect it around the X axis. This is the data we'll use to train our deep RNNs.

Naturally enough, we call the modified data-making routine `sum_of_upsloping_sines()`, and present it in Listing B3-66.

```
def sum_of_upsloping_sines(number_of_steps, d_theta,
                           skip_steps, freqs, amps, phases):
    '''Like sum_of_sines(), but always sloping upwards'''
    values = []
    for step_num in range(number_of_steps):
        angle = d_theta * (step_num + skip_steps)
        sum = 0
        for wave in range(len(freqs)):
            y = amps[wave] * math.sin(freqs[wave]*(phases[wave] + angle))
            sum += y
        values.append(sum)
        if step_num > 0:                # are we past the first sample?
            sum_change = sum - prev_sum  # find direction we're headed in
            if sum_change < 0:          # are we going downward?
                values[-1] *= -1        # if so, flip the last value
                if step_num == 1:       # is this second sample in the set?
                    values[-2] *= -1    # if so, flip the first value, too
        prev_sum = sum      # remember the sum we found, without any flips
    return np.array(values)
```

Listing B3-66: A modification of `sum_of_sines()` where we flip the curve upside down any time it's heading downwards. The new lines are in the `if` statement and the assignment just after it.

Let's run our new dataset through the same tiny network as in our most recent experiment, keeping the window size of 5. The loss results are shown in Figure B3-69.

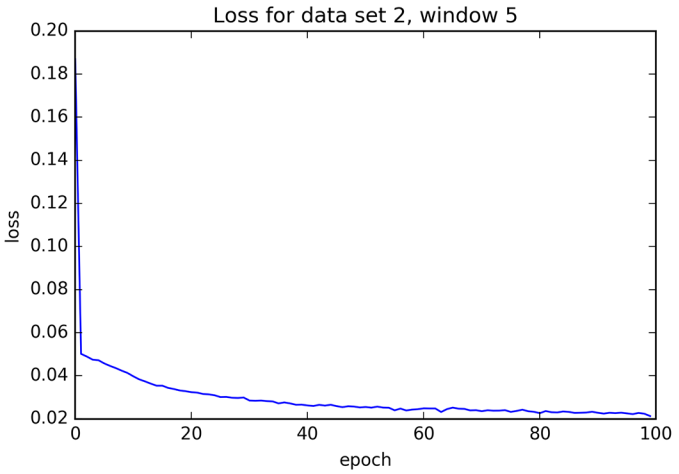


Figure B3-69: The test loss from running our upsloping test data through our 3-unit RNN with a window of 5 time steps

The loss drops fast at the very start, and then improvement slows down a lot. Around epoch 60 it seems to either have settled down, perhaps improving just a tiny bit from then on.

The predictions by this model to our modified test data are shown in Figure B3-70.

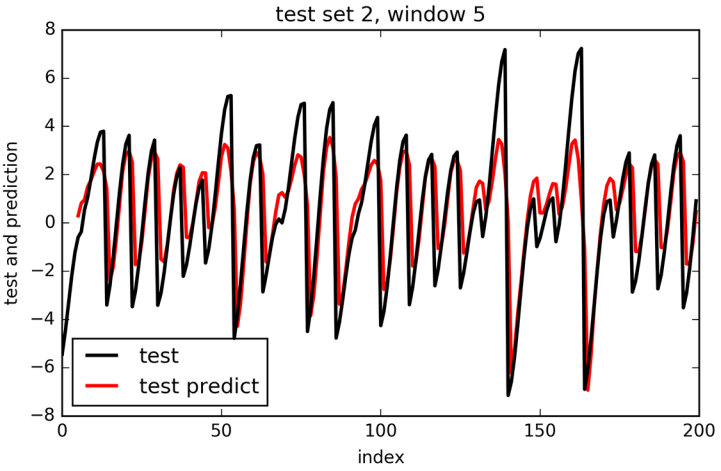


Figure B3-70: The quality of the predictions of our model to the modified test data

The late start (where the first 4 values have no prediction) is easier to see here at the far left. This result is pretty bad. The predicted values do tend to generally track the straighter sections of the test data, but the predictions frequently overshoot and undershoot the peaks and valleys.

Our tiny network worked surprisingly well up until now, but we've finally asked too much of it.

Deep RNNS

Let's add a second recurrent layer, also made of 3 LSTM units.

Figure B3-71 shows our new architecture.

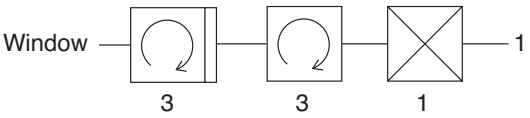


Figure B3-71: The diagram for our 2-layer RNN. The small box on the output end of the first LSTM indicates that it has `return_sequences` set to `True`.

Listing B3-67 shows how to build our 2-layer deep RNN model, using 2 layers of 3 LSTM units each.

The first LSTM needs us to include a new argument, `return_sequences`, which we need to set to `True`. We indicate this with a small box on the right of the LSTM icon (or, when the network is drawn bottom to top, on the top). We'll discuss what this `return_sequences` is about soon, but for now we can treat it as something that has to be included any time we create an LSTM that is followed by another LSTM.

Here's the code, with `return_sequences` set to `True` in the first LSTM.

```
model = Sequential()
model.add(LSTM(3, return_sequences=True, input_shape=[window_size, 1]))
model.add(LSTM(3))
model.add(Dense(1))
```

Listing B3-67: Building a deep RNN just means adding more recurrent layers. All recurrent layers that precede another must have their optional argument `return_sequences` set to `True`.

As far as Keras is concerned, this is just another `Sequential` model, so we can train this model and get predictions from it just as before.

Let's see how this performs with our new data.

Figure B3-72 shows the loss during training.

The loss gets down to a little more than 0.02, which is roughly what we saw before. So we shouldn't get too optimistic about the predictions.

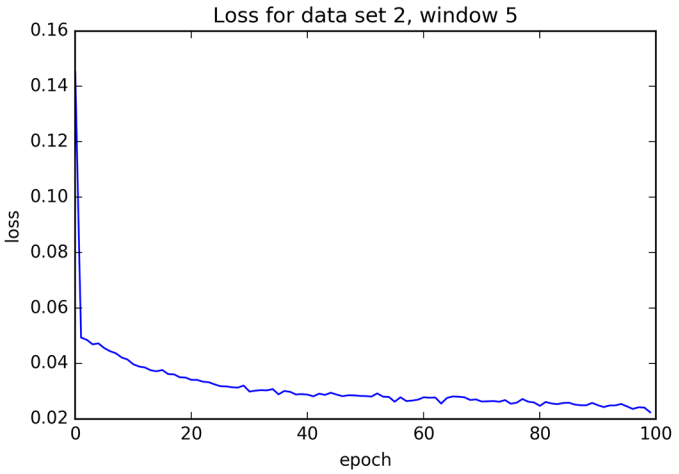


Figure B3-72: Training our model with 2 recurrent layers of 3 LSTM units each produces these loss results.

Figure B3-73 shows the predictions on the test data.

This is pretty bad. Around epochs 130 to 180 it seems to lose track of the value of the data, though it does roughly mimic its rising and falling. It seems that adding a second layer has made things a lot worse.

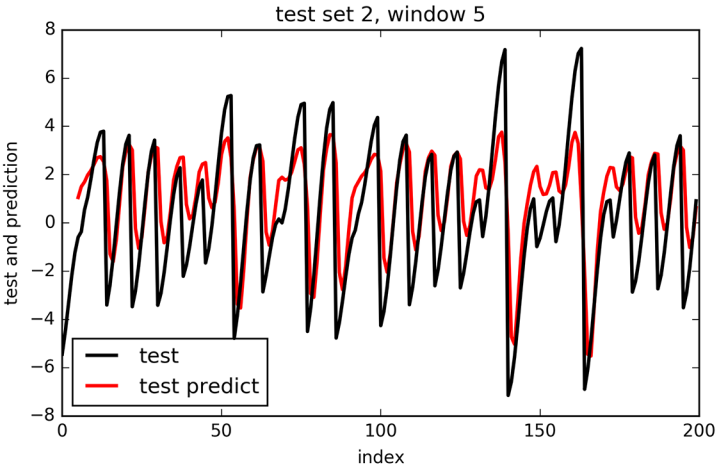


Figure B3-73: Training our model with 2 recurrent layers of 3 LSTM units each produces these predictions to the test data.

Let's see if we can get something better with an even deeper model. Let's make 3 LSTM layers of decreasing sizes, with 9, 6, and 3 units respectively. Figure B3-74 shows this architecture.

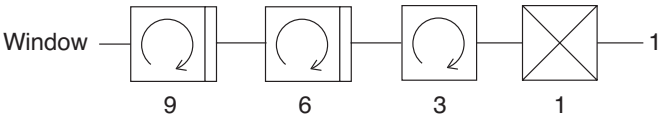


Figure B3-74: The diagram for our 3-layer RNN

Listing B3-68 shows the code to build the model.

```
model = Sequential()
model.add(LSTM(9, return_sequences=True, input_shape=[window_size, 1]))
model.add(LSTM(6, return_sequences=True))
model.add(LSTM(3))
model.add(Dense(1))
```

Listing B3-68: Making an even deeper RNN with 3 recurrent layers of decreasing numbers of LSTM units

Once again, each LSTM that feeds another LSTM has to have `return_sequences` set to `True`, indicated by the small box at the top of the icon.

Figure B3-75 shows the loss of our 3-layer model during training.

This is encouraging, because while the loss at epoch 100 is still only about 0.025, like in previous runs, the loss is still dropping, while the previous loss graphs were flat. If we kept learning, we ought to expect further improvements. It's not a dramatic improvement, though.

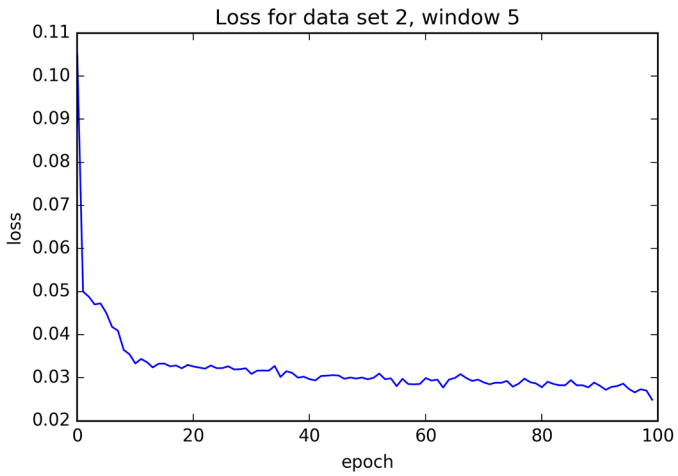


Figure B3-75: The loss from training our model with 3 recurrent layers of 9, 6, and 3 LSTM units

Figure B3-76 shows the prediction results for this deeper RNN.

The improved numerical loss is encouraging, but the network is still not doing a good job visually. This might even be worse than before.

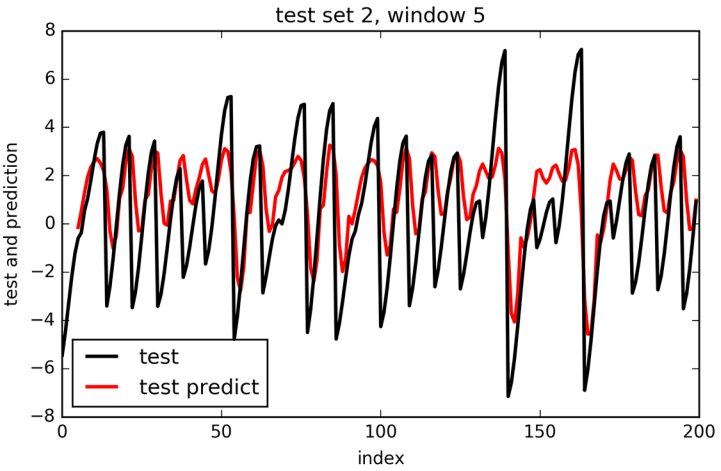


Figure B3-76: The test data predictions made by our deep RNN with 9, 6, and 3 LSTM units on successive layers

The Value of More Data

Remember our general principle that more data is usually better than fancier algorithms. So rather than continue to tweak the network, let's get more data.

One of the pleasures of working with synthetic data is that we can make as much of it as we want. The values of the frequencies in the test 2 dataset don't repeat for a long time, so we can crank out a lot of data (over 40,000 samples) without repeating. Let's increase the size of our training set from 200 samples to 2000. To match the 10-fold increase in the number of training samples, let's increase the window from 5 to 13, leaving all the other parameters the same.

The loss during training is shown in Figure B3-77.

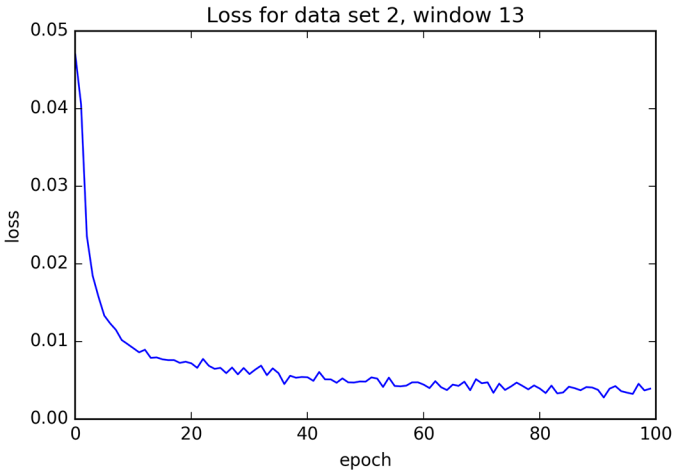


Figure B3-77: Loss from training our 3-layer network with 2000 samples and a window size of 13 time steps

This is a huge drop in loss. In Figure B3-75 the loss got down to about 0.025 after 100 epochs. Here, the loss seems to be about 0.004.

If we plot all 2000 samples they'll jam up and give us a black rectangle, so let's look at the just the first 200 samples. This has the added benefit that we're already familiar with them. Figure B3-78 shows the predictions on our test data.

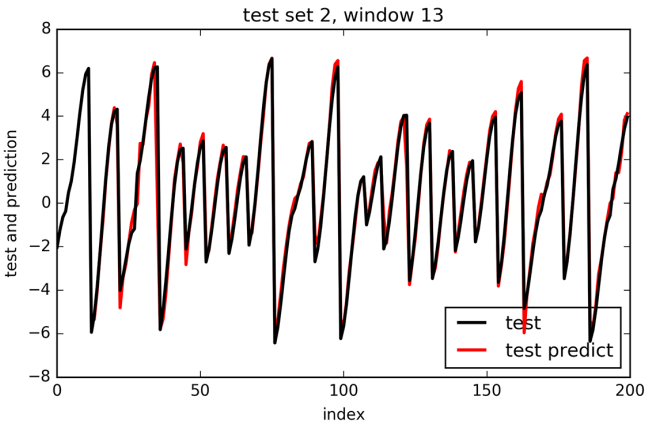


Figure B3-78: Test data predictions by our 3-layer deep RNN after training on 2000 samples chunked into windows of 13 time steps

This is a big improvement. There's still some over- and under-shooting going on, but generally we have a much better match than before.

More data really does help!

Thanks to our procedural data generator, can crank up the size of our training set by another factor of 10 to see what happens. We'll leave everything else the same, but increase the training set from 2,000 to 20,000 samples.

The loss results are in Figure B3-79.

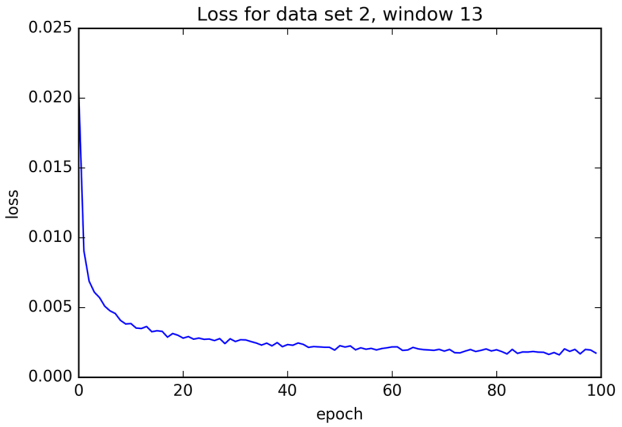


Figure B3-79: Loss from training our 3-layer network with 20,000 samples and a window size of 13 time steps

The loss has dropped to about 0.0015, which is less than half of the roughly 0.004 we had before.

Figure B3-80 shows the predictions on our test data.

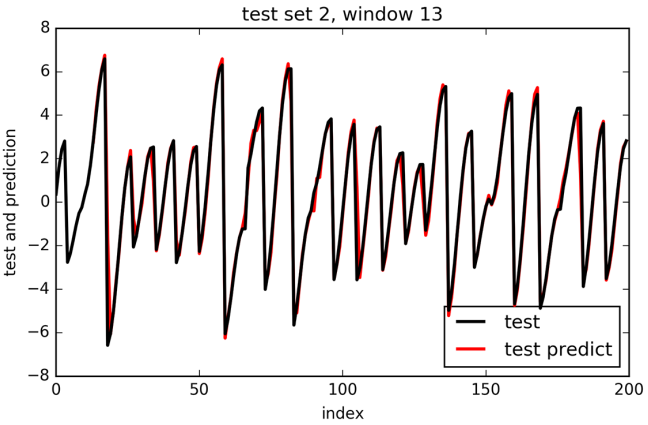


Figure B3-80: Test data predictions by our 3-layer deep RNN after training on 20,000 samples chunked into windows of 13 time steps

This is the best set of predictions of this difficult dataset that we've seen yet. There are still a few obvious misses, but compared to previous results this is pretty close.

Even more data helps even more!

It's worth noting that the time taken to learn these increasingly large training sets is roughly proportional to the number of samples. Each epoch of the 20,000 sample training set took a little over 5 seconds using only the CPU on a late 2014 iMac (that is, there was no GPU acceleration). So Figure B3-80 took a bit over 28 hours to compute. Every one of these epochs took about 10 times longer than each epoch of the 2,000 sample set, which in turn were about 10 times longer than epochs when training the 200 sample set. More data is great, but processing it comes at a price.

The good news is that we pay that price only once, during training. We've been steadily improving our 3-layer RNN without changing the model, so all of these models take the same amount of time to predict new values after training is over. Our up-front training cost is amortized over every use of our model, forever.

As we've seen, RNNs are sensitive to how they're trained. Deep RNNs are even more sensitive. Because we didn't tune these architectures at all as we added new layers, we're probably leaving a lot of performance on the table. By adjusting the window size, the learning rate, the parameters to our Adam optimizer, and our choice of loss function, we might be able to improve our best results in Figure B3-80. In practice, it is usually worth exploring the effects of different modeling and training parameters to see what works best for a given network and data.

Returning Sequences

This section's notebook is Bonus03-Keras-6-RNN-Sequence-Shapes.ipynb.

In our deep networks above, we used the output of one RNN as the input of another RNN. We saw that the earlier RNN layer (the one providing the input to the next) needed a new argument. Its name was `return_sequences` and we set it to `True` (the default is `False`). Now it's time to make good on our promise to discuss what that argument is about.

Let's return to our first, simple network of an RNN that followed by a dense layer, as we saw back in Figure B3-55. Our goal was to hand the network a sequence of time steps, and then have it predict the next value after the sequence.

Our samples contained just one feature, which held a series of values from a 1D curve. These made up the time steps.

When we gave the RNN our sample, it read the first time step and produced an output. This output was the contents of the internal state, after passing through the RNN's internal selection gate. The output could be thought of as the RNN's prediction of the next value of the curve.

But we didn't care about that prediction, because we already knew the second time step. Keras knew we had more time steps to come, so it automatically ignored that output, and didn't even send it to the dense layer. Instead, it gave the RNN the second time step. Again, the RNN produced an output, and again, Keras ignored it. Figure B3-81 shows the idea, for an RNN with 4 elements of internal data. Here we've handed the RNN the third time step, and it's produced the third output, which we're ignoring.

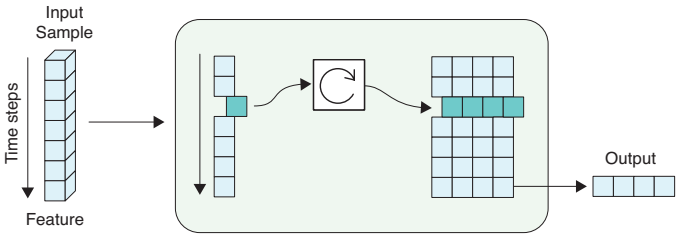


Figure B3-81: Using an RNN with 4 elements of state. The input sample has a single feature, containing a list of time steps. After each time step is evaluated, the 4-cell RNN produces an output 4 elements long, as shown here for the 3rd time step. We've been ignoring all outputs except the last one.

We did this over and over, handing the RNN sequential time steps and ignoring the outputs, until we gave it the last time step in the sample. The output of *that* time step was the prediction for the value of the sequence after the end of our inputs, so that output was the value we were after all along. We fed that to our dense layer, and the output was the prediction.

Suppose our inputs had more than one feature. If our data held weather measurements at the top of a mountain, maybe each sample held temperature, wind speed, and humidity. Let's say what we want from our RNN is a prediction of how good the radio reception would have been on the mountain at that time. So at each time step we give the RNN the values

for all three features at that time step. The output is again the RNN's internal state after the selection gate, so it has as many elements as there are elements in the internal state. As before, we get back one such output for every time step input we provide, and we only pay attention to the last one. Figure B3-82 shows the idea.

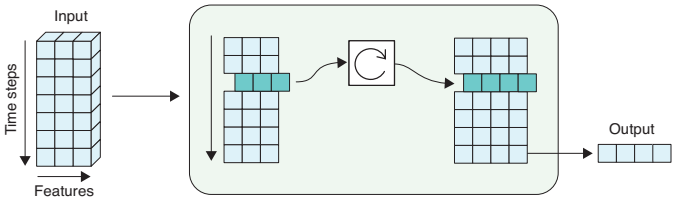


Figure B3-82: When we have multiple features in our sample, we provide all the features for a given time step to the RNN.

There are a couple of things worth noting in Figure B3-82.

First, our input is a 3D tensor. In this example, it has 1 sample, 7 time steps, and 3 features, so it has dimensions 1 by 7 by 3. The number of time steps, and the number of features, don't appear in the output, which is a 2D tensor of shape 1 by 4. The 1 is because we only care about one output (the last one), and the 4 comes from the internal state of the RNN, which we've assuming has 4 elements.

We "lost" the number of features because they are used internally by the RNN to control the forgetting, remembering, and selecting of the internal state. We "lost" the number of time steps because we chose to ignore all but the last one.

Our input could have had 19 features and 37 time steps, and the output would still be 1 by 4.

Let's expand the picture a little to include the unrolled RNN diagram. In Figure B3-83 we have a sample with 5 time steps and 3 features. Once again, the RNN's internal state has 4 elements. We can see in the figure that at each time step, an entire row of features is fed to the RNN, which produces an output. Then the RNN's state changes, which sets up the RNN for the next input, as shown by the open downward-pointing arrow. We only pay attention to the last output. The output is a 2D grid of shape 1 by 4.

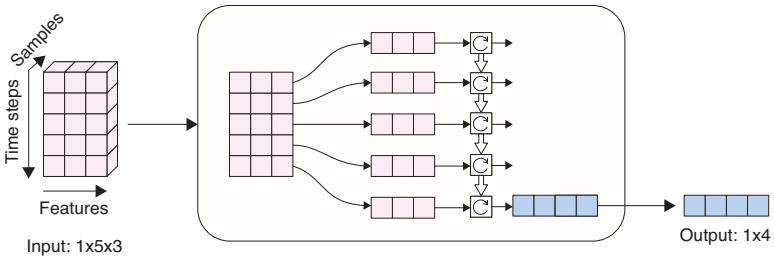


Figure B3-83: Passing a sample to an RNN. The unrolled RNN diagram is shown vertically. Read the contents of the rounded box from top to bottom over time. As before, the outputs are ignored except for the final time step.

If we want to feed this output to a dense layer, we don't have to do a thing.

But suppose we want to take our sequence of outputs and present them as a sequence of inputs to another RNN layer, as we did in some of our deep RNN models above. We know that an RNN needs a 3D input, and the output here is 2D.

We could just give it a depth of 1, producing a shape that's 1 by 1 by 4. While this is now legal for an RNN, it doesn't make any sense. A tensor with this shape would be interpreted as a single sample (the first 1) with 1 time step (the second 1), containing 4 features (the 4 at the end). That's nothing like our single sample of 5 time steps and 3 features.

Losing the time step information is a big problem, because that's the idea at the heart of an RNN. We're giving our first layer 5 time steps, and it's producing 5 outputs. We then want to hand those 5 outputs to the next layer. Each output will have 4 elements (since we're supposing that our RNN has 4 elements of internal state), so those 4 values will be interpreted by the next RNN as 4 features. But we need the 5 time steps.

That's actually easy to do. We just tell Keras to *not* ignore the output after each step. We tell it to take the outputs and stack them up to make a grid. It will be as tall as there are time steps, and as wide as there are elements in the internal state. Now we can give that grid a depth of 1, and it makes sense as an input to an RNN.

Figure B3-84 shows the idea.

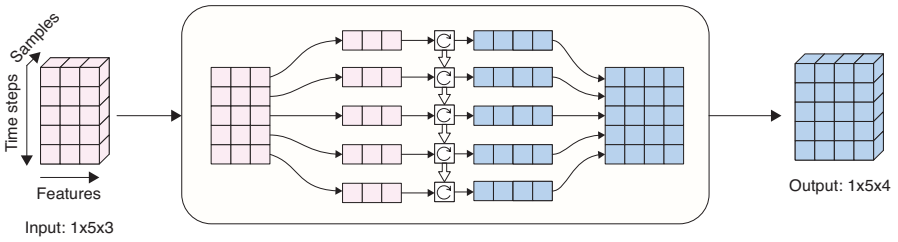


Figure B3-84: To send an RNN's output to another RNN, we just remember the output at every time step. These outputs are stacked together to form a 2D grid, and then we give it a depth of 1 to mean it's all one sample, and we're ready to give this to another RNN.

To tell Keras to remember the output after each time step and build up this grid, we tell it that we want the RNN to return not just a single output, but the whole sequence of outputs corresponding to the sequence of inputs.

By setting the optional argument `return_sequences` to `True`, we're telling Keras to do exactly what Figure B3-84 shows.

Now that we know what `return_sequences` is all about, we can usually invoke it without even thinking about all of this. If our RNN's output is going into another RNN, just set `return_sequences` to `True`. If we want only the output after the last time step, we can set `return_sequences` to `False`, or just leave it off, since that's the default value.

A few input shapes and their outputs with `return_sequences` set to both `False` and `True` are shown in Figure B3-85.

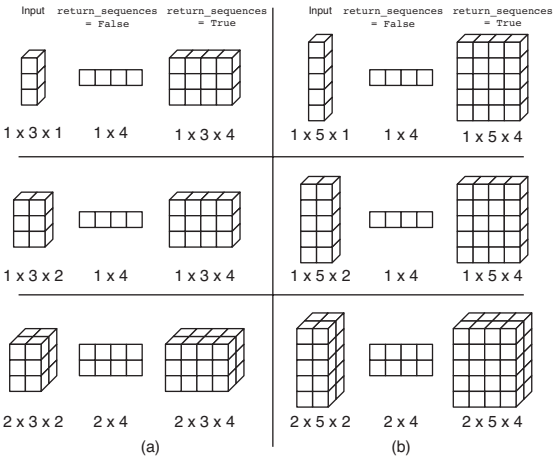


Figure B3-85: The output of a 4-cell RNN for different shapes of input. In each box, the input shape is on the left, the output with `return_sequences=False` is in the middle, and the output with `return_sequences=True` is on the right.

It's useful to see at a glance whether an RNN returns just the final output or the full sequence. We mark the icon for an RNN that returns a sequence with a small box on the output side, suggesting multiple outputs, as in Figure B3-86.

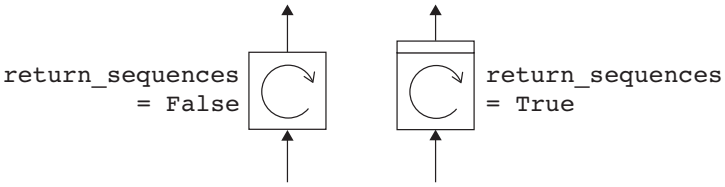


Figure B3-86: Icons for RNN units. Left: When `return_sequences` is `False`, the RNN returns only the final value in the sequence. Right: When `return_sequences` is `True`, the RNN returns its output for each time step. We mark this with a small box on the output side of the icon.

Stateful RNNs

We've been focusing on one sample at a time, but in practice we usually train in mini-batches. This poses an interesting question for RNNs, since their internal memory is always influenced by previous inputs. When should we clear that memory and let the RNN start over?

The usual approach is to clear the internal memory at the start of a new batch, or mini-batch. We don't clear or reset the weights belonging to the neurons inside the RNN, since those tell it how to do its job. We only clear its changing memory that holds the inputs it's recently seen. The thinking is

that when a new batch begins we'll possibly be getting data that isn't a continuation of the most recent samples, so we don't want to remember stuff from back then.

Usually, we shuffle our samples between epochs, so they arrive in an unpredictable order each time. But we can keep their order consistent from epoch to epoch, if we want to. We do this when we call `fit()` to train our model, setting the optional argument `shuffle` to `False` (the default is `True`).

When the data is always arriving in sequence, there's no reason to reset the memory at the start of each batch, because those samples follow the samples in the previous batch. In other words, the batching just breaks up the grouping of the samples, and not their sequence. In that situation, we can tell Keras to *not* reset the memory at the start of each batch. This sometimes can help us train a little faster.

In Keras, when we take over the responsibility for clearing the memory we say that the RNN is in the `stateful` mode. In this mode, Keras only resets the internal state when we tell it to. Usually this is at the start (or end) of each epoch.

Stateful mode can make training go a little faster, but it comes with limitations. The batch size must be determined in advance, and it becomes a part of the model. The dataset must be a multiple of this batch size. For instance, if the batch size is 100, the dataset must be 100, 200, 300, and so on samples long. If it's 130 or 271 samples, we'll get an error.

When we later give new data to the model for it to evaluate, that data also has to come in batches of the same size we used when we trained. If we want only one prediction, but our batch size is 100, then we can either pad out our one request with 99 more copies of itself, or just load up all the unused entries in the batch with 0's. We'll still end up waiting for the network to evaluate all those samples, though.

To make a stateful network, we need to do four things.

First, we need to include the optional argument `stateful` to each RNN (such as an LSTM or GRU) and set it to `True`. This tells Keras that we're taking care of when to reset the cell's state.

Second, we need to include the argument `batch_size` to the first RNN we make, and set it to the batch size that we're going to use during training.

Third, when we call `fit()` we need to set `shuffle` to `False`.

Finally, when we want to reset the state, we need to explicitly call `reset_states()` on our model.

Listing B3-69 shows an example of the first two points while building a small RNN with two `LSTM` layers and a one-neuron `Dense` layer at the end. This is adapted from the *stateful_lstm.py* example in the Keras documentation [Chollet17b]. We assume that the variable `time_steps` holds the number of time steps in our inputs, and `batch_size` is set to the batch size we plan to use.

```
model = Sequential()
model.add(LSTM(50,
               input_shape=(time_steps, 1),
```

```
                return_sequences=True,
                batch_size=batch_size, stateful=True))
model.add(LSTM(50, stateful=True))
model.add(Dense(1))
```

Listing B3-69: Building a stateful RNN. We need to give the first LSTM a value for `batch_size`, and set `stateful=True` in each LSTM. Adapted from [Chollet17b].

To train our model, we need to remember to tell `fit()` not to shuffle our data. Because we want to reset the state after each epoch, we don't want to do the usual thing of telling `fit()` how many epochs to train for, and then walking away, because then the RNN will never be reset.

Instead, we'll tell `fit()` to train for only 1 epoch, and we'll put that call in a loop. The loop will repeat for the number of epochs we want to train for. Doing it this way lets us put in a call to `reset_states()` at the end of each epoch of training.

Listing B3-70 shows this step.

```
for i in range(number_of_epochs):
    model.fit(X_train, y_train, batch_size=batch_size,
              epochs=1, verbose=1, shuffle=False)
    model.reset_states()
```

Listing B3-70: Training a stateful RNN. We need to tell `fit()` not to shuffle the data, and then we need to call `reset_states()` after each epoch. Adapted from [Chollet17b].

Time-Distributed Layers

As we've seen, when we set an RNN unit's `return_sequences` argument to `True`, Keras saves its output after each time step.

We saw in Figure B3-84 that this results in one output for each time step in the sample. The outputs for a sample get gathered together into a grid, and the grids for many samples get gathered together into a volume.

Let's suppose we want to process this output volume in a `Dense`, or fully-connected layer. We'd need to flatten it first into a 1D list, and then feed that list to the layer. Sometimes that's fine.

But other times, we'd like the dense layer to process the individual outputs one by one. So we want to run the dense layer over each of the separate lists coming out of the RNN in Figure B3-84 rather than over the 2D grid they get assembled into.

Once these lists have been assembled into a grid, it would be hard to pull them apart. We could write our own custom layer, or make a custom contraption using the Functional API (discussed below), but we'd like an easier approach that lets us treat these individual outputs one by one.

Keras provides a special-purpose layer for exactly this job. It's called a `TimeDistributed` layer. It's not really a layer, though. Keras calls it a *wrapper* layer. The idea is that it's a container that we put one or more layers into, and then those layers get treated in a special way.

To get a feeling for what the `TimeDistributed` wrapper does for us, let's build a tiny network without one. Figure B3-87 shows an RNN with 4 elements of state, followed by a `Dense` layer with 5 neurons. Since there's no box on top of the RNN icon, we know that it doesn't return a sequence.

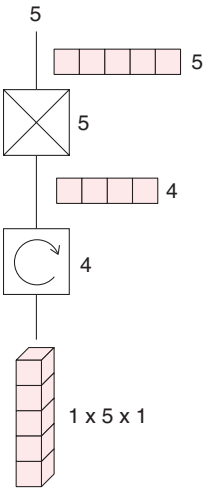


Figure B3-87: A small network to set up our discussion of the `TimeDistributed` layer

As we can see, the input is 1 sample, made of 1 feature, with 5 time steps. After passing through the RNN, we get a single 4-element list. We can then feed that directly into a `Dense` layer of 5 neurons, getting back a list of 5 values at the output.

Let's have the RNN return the individual sequence outputs by setting `return_sequences` to `True`. Now we have Figure B3-88 (a), where we had to insert a `Flatten` layer between the RNN and the dense layer.

In part (a) of Figure B3-88, we see that the RNN's 3D output does not fit the `Dense` layer's need for a 1D list, so we can flatten it first. But then the `Dense` layer is processing all 20 outputs at once. In part (b), we wrap the `Dense` layer in a `TimeDistributed` layer. Now Keras will hand it each sequence of the output in turn, and then combine the results again.

Although flattening the RNN output as in Figure B3-88(a) works, and everything will run, it's not what we want. The problem is that the `Dense` layer will process all 20 values coming out of the RNN at once. What we want is for it to process each time step individually.

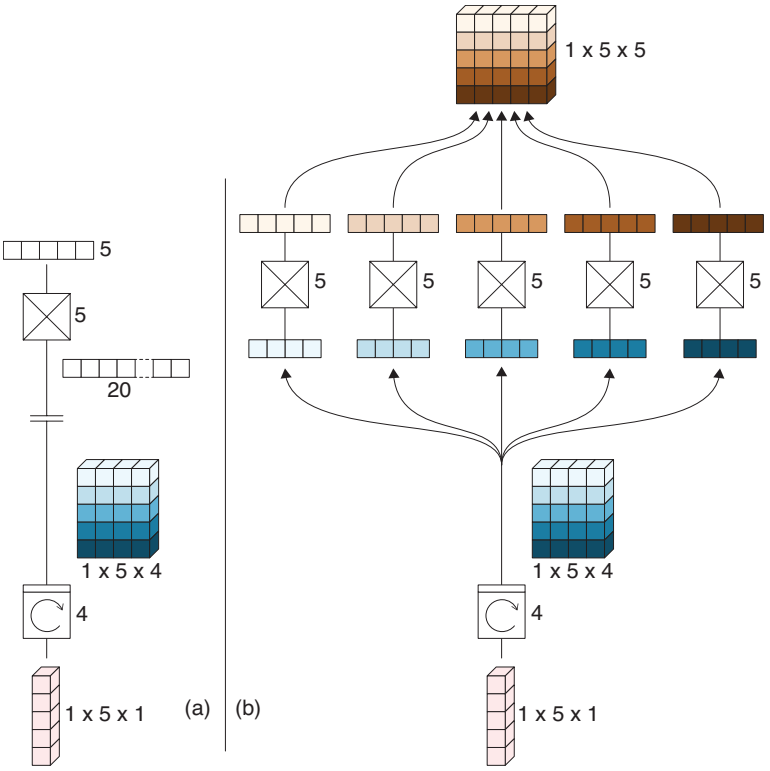


Figure B3-88: When an RNN returns a sequence, we can apply some other layer to each step of the sequence by wrapping it in a `TimeDistributed` layer.

If we wrap the `Dense` layer in a `TimeDistributed` wrapper, we invoke a bunch of machinery inside of Keras that gives us an operation shown in Figure B3-88(b). Each time step is individually handed to the `Dense` layer and then the results are combined. Note that in this figure there is only one `Dense` layer which gets applied to all 5 time steps.

Another version of Figure B3-88(b) is shown in Figure B3-89. On the left we show how we'd draw our network schematically. The five-sided shape around the `Dense` layer is our icon for the `TimeDistributed` layer. The V at the bottom is meant to suggest the branching of the lines in Figure B3-88(b), telling us that the one input is being broadened. On the right is an expanded view of what's happening inside of the `TimeDistributed` layer. Again, there's just one `Dense` layer that is being applied to each time step in this sample.

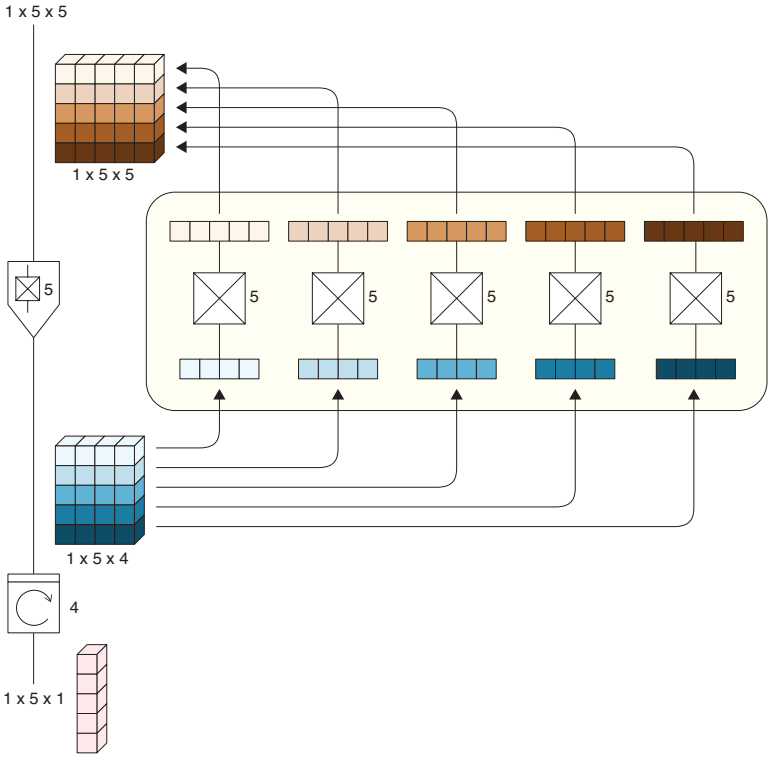


Figure B3-89: By wrapping our `Dense` layer in a `TimeDistributed` layer, the `Dense` layer will individually process each sequential output from the RNN, and then those results will be assembled into a new tensor.

To create a layer wrapped in a `TimeDistributed` layer, we just nest the calls, as in Listing B3-71.

```
model = Sequential()
model.add(LSTM(4, return_sequences=True, input_shape=[window_size, 1]))
model.add(TimeDistributed(Dense(5))
```

Listing B3-71: Feeding the output of an `LSTM` layer to a `Dense` layer, wrapped up in a `TimeDistributed` wrapper, as in Figure B3-89

Figure B3-90 shows our icon for a `TimeDistributed` layer with different contents. With the Functional API (discussed below) we can wrap many layers with just one `TimeDistributed` wrapper. Alternatively, we can wrap each one individually.

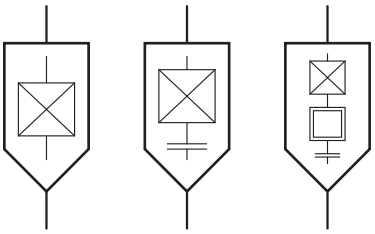


Figure B3-90: Three collections of layers, each in a `TimeDistributed` wrapper

Generating Text

The letter by letter notebook is Bonus03-Keras-7-Generate-Text-By-Letter.ipynb.

The word by word notebook is Bonus03-Keras-8-Generate-Text-By-Word.ipynb.

In Chapter 19 we experimented with generating new text based on the stories of Sherlock Holmes.

This isn't hard to do, but it requires more than just a couple of lines of Python programming. The notebooks for this section contain all the code for making new text, either letter by letter or word by word. Rather than go through all the details, we'll just walk through the big pieces and mention some highlights. Our code is influenced by a popular presentation available online [Karpathy15].

In previous notebooks in this chapter we've presented the code as essentially a single big list of lines to be executed in order. We broke up the lines into conceptual chunks and placed them in cells, but that didn't change how we wrote or used the code. For variation, this time we packaged up each of the steps into its own procedure. Then when we're ready to make text, we just call some of those procedures and let them do their work.

Our first step is to read in the source text. We replaced multiple spaces with single spaces, and removed newline characters since they don't have any semantic meaning.

This code is shown in Listing B3-72, where we've wrapped up the job of reading and processing the file into a routine called `get_text()`.

```
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Activation
from tensorflow.keras.layers import LSTM
from tensorflow.keras.optimizers import RMSprop
import numpy as np
import random
import sys

def get_text(input_file):
    # open the input file and do minor processing
    file = open(input_file, 'r')
    text = file.read()
    file.close()
    text = text.lower()
    # replace newlines with blanks, and double blanks with singles
    text = text.replace('\n',' ')
```

```
    text = text.replace('  ', ' ')
    print('corpus length:', len(text))
    return text
```

Listing B3-72: To generate new text, we start by reading in and processing the text file with our source text.

Now we have to chop up the input into overlapping windows. We need to pick the window size and how much they overlap. The routine `build_fragments()` creates these little fragments for us, as in Listing B3-73.

```
def build_fragments(text, window_length):
    # make overlapping fragments of window_length characters
    fragments = []
    targets = []
    for i in range(0, len(text)-window_length, window_step):
        fragments.append(text[i: i + window_length])
        targets.append(text[i + window_length])
    print('number of fragments of length window_length=',
                window_length,':', len(fragments))
    return (fragments, targets)
```

Listing B3-73: We build text fragments by breaking up the input text into overlapping pieces, each with `window_length` characters.

Since our network wants numbers, not letters, we'll assign a unique number to each letter. To make it easy to go back and forth, we'll make two dictionaries. One is keyed on characters and returns their number, and the other is keyed on number and returns their character. We'll call the number an "index." We can get the total number of unique characters by using Python's `set()` operation. Just for general tidiness we'll sort that list before using it. Listing B3-74 shows the routine `build_libraries()` that does the job.

```
def build_dictionaries(text):
    unique_chars = sorted(list(set(text)))
    print('total unique chars:', len(unique_chars))
    char_to_index = dict((ch, index) \
        for index, ch in enumerate(unique_chars))
    index_to_char = dict((index, ch) \
        for index, ch in enumerate(unique_chars))
    return (unique_chars, char_to_index, index_to_char)
```

Listing B3-74: We build a pair of dictionaries to let us turn each letter into a unique number, and vice-versa.

Now we want to turn our samples and targets into one-hot vectors. We're already familiar with using one-hot targets. We'll use one-hot encoding for the samples here as well because we want each letter to be a feature in our data. That feature will have as many time steps as there are unique characters in our data. They'll all be 0 except for a 1 corresponding to the character being represented.

Listing B3-75 shows one way to build the one-hot versions. We'll make a couple of grids full of zeroes, and then set the ones where needed.

```
def encode_training_data(fragments, window_length, targets,
                         char_to_index, index_to_char):
    # Turn inputs and targets into one-hot versions
    X = np.zeros((len(fragments), window_length, len(char_to_index)),
                         dtype=np.bool)
    y = np.zeros((len(fragments), len(char_to_index)), dtype=np.bool)
    for i, fragment in enumerate(fragments):
        for t, char in enumerate(fragment):
            X[i, t, char_to_index[char]] = 1
        y[i, char_to_index[targets[i]]] = 1
    return (X, y)
```

Listing B3-75: Turning our fragments and targets into one-hot versions called `X` and `y`.

Now let's build the model. After a little playing around, we chose the simple deep model of Figure B3-91. It's just two `LSTM` layers and a single `Dense` layer. The first `LSTM` has `return_sequences=True`, because it feed another `LSTM`. The second one produces a single output, which will lead us to the letter the network is predicting. To get that letter, we use a `Dense` layer with one neuron per letter, and a softmax output. This will give us the probability of each character being the next one.

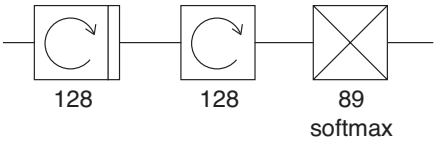


Figure B3-91: Our simple deep network for generating text one letter at a time

Listing B3-76 gives the source for building this model.

```
def build_model(window_length, num_unique_chars):
    # build the model. Two layers of a single LSTM cell with 128
    # elements of memory,then a dense layer with as many outputs
    # as there are characters.
    # We'll train with the RMSprop optimizer. Some experiments suggest that
    # a learning rate of 0.01 is a good place to start.
    model = Sequential()
    model.add(LSTM(128, return_sequences=True,
              input_shape=(window_length, num_unique_chars)))
    model.add(LSTM(128))
    model.add(Dense(num_unique_chars, activation='softmax'))
    optimizer = RMSprop(lr=0.01)
    model.compile(loss='categorical_crossentropy', optimizer=optimizer)
    return model
```

Listing B3-76: Build our little RNN architecture

Now we're ready to generate text. We'll call a new routine called `generate_text()` that will train the model for a single epoch, and then print out some text that it generates. This way we can see how the quality of the text improves over time.

After each call to `fit()` to train the model, we'll pick a random starting point in the original document and extract characters from there. We'll pick as many characters as in the window size we trained on. We'll one-hot encode that sequence of characters and give the result to `predict()`. This will give us back one probability for each unique character in the original text, telling us how likely it is that that character is the one that comes next after the input text.

We could just use the most probable character, but in practice that tends to give us a lot of repeated words. A nice alternative is to juggle around the probabilities a little so that less-likely letters also have a chance of being chosen. A nice algorithm for that adds metaphorical "heat" to the probabilities to change their values [Chollet17c]. We've wrapped that up in a routine called `choose_probability()` that's in the notebook.

Once we've got the prediction for the next character, we append that prediction to a growing output string. Then we append the new character to the end of our input to the model, while also dropping the first character from that string, so the input is always the length of the training windows. Then we train the system for another epoch and do it all again.

The code for `generated_text()` is shown in Listing B3-77. Rather than simply printing strings to the output, we hand them to a routine named `print_string()` that both prints them, and saves them in a file that we've opened.

```
def generate_text(model, X, y, number_of_epochs, temperatures,
                  index_to_char, char_to_index, file_writer):
    # train the model, output generated text after each iteration
    for iteration in range(number_of_epochs):
        print_string('--------------------------------------------------\n',
                     file_writer)
        print_string('Iteration '+str(iteration)+'\n', file_writer)
        history = model.fit(X, y, batch_size=batch_size, epochs=1)
        start_index = random.randint(0, len(text) - window_length - 1)

        for temperature in temperatures:
            print_string('\n----- temperature: '+str(temperature)+'\n',
                         file_writer)
            seed = text[start_index: start_index + window_length]
            generated = seed
            print_string('----- Generating with seed: <'+seed+'>\n',
                         file_writer)

            for i in range(generated_text_length):
                x = np.zeros((1, window_length, len(index_to_char)))
                for t, char in enumerate(seed):
                    x[0, t, char_to_index[char]] = 1.

                preds = model.predict(x, verbose=0)[0]
                next_index = choose_probability(preds, temperature)
                next_char = index_to_char[next_index]

                generated += next_char
                seed = seed[1:] + next_char
```

```
        print_string(generated+'\n\n', file_writer)
        file_writer.flush()
```

Listing B3-77: Generate new text using our trained model.

The majority of the work in this program involves messing about with the data, making the dictionaries and windows and doing the one-hot encoding and so on. The actual neural network code was just a few lines to make the network, and one line each to train it and get predictions.

To train the system, we picked a window length of 40 characters, and a step of 3, so each training string overlapped 37 characters with the one before. We used a batch size of 100, and generated 1000 new characters after each training step, using "temperatures" of 0.5, 1.0, and 1.5. The text with temperature 0.5 tended to produce the same words frequently, and temperature 1.5 produced mostly words, but also lots of strings that weren't words. It's fun to play with the temperature to find the sweet spot where the output is interesting, with the occasional weird almost-word.

As we mentioned in Chapter 19, this can take a long time to run. On a late 2014 iMac, without GPU support, each iteration takes about 1400 seconds, or a little more than 23 minutes. Networks like this often take 800 epochs or so to start producing text that is close to the source. That would be about 13 days of 24/7 crunching. So we ran this network for 100 epochs on Amazon Web Services, watching the loss drop from about 2.6 to 1.1. Here's the start of what it generated after that much training, starting with the seed last time in my life. Certainly a gray m, and with a temperature of 1.0.

> last time in my life. Certainly a gray myself under the great tautoh;
> harm I should be a busy because cameful allo done." "Why dud
> that you dedy hour any one of these chimnes of this pricaption
> is to his, If the tall. Up appeared to very set over with Mr. Trem,
> there, if we confeeliin, I fawny of days if so far"

Clearly we have a long way to go. But remember that this is letter by letter, from a system that has no idea of English or language or any such structure. Given that it started from nothing and had only such a small amount of training, this is pretty great.

An alternative way to generate text that we discussed in Chapter 19 is to focus on sequences of words, rather than letters. This is appealing in many ways, but it's also slower to train. If we have 7000 or 8000 unique words, that's a lot more work to manage than 89 unique characters.

We experimented with this a bit and chose the architecture in Figure B3-92 to generate new text word by word. The full code is provided in the accompanying notebook.

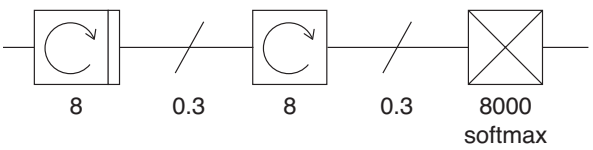


Figure B3-92: A network for word-by-word text generation

We trained with the 8000 most frequently-used words in the text, replacing all others with the marker GLORP. Here's the output after the first epoch, starting with the seed, tell her future husband the whole story and to trust to his generosity . Milverton chuckled . You evidently do not know the Earl , said he . From the baffled look upon Holmes's face , I could. Note that the punctuation marks have been isolated as their own words.

> tell her future husband the whole story and to trust to his GLORP . Milverton chuckled . You evidently do not know the Earl said he . From the baffled look upon Holmes's face , I could each clear at screen At there by put got His you openly is do that were once Your plans from my He greatest life to did mantle it first India , drive as come really It black build my is put hearty Stanley sprang , afraid once quite whom had comes sole snuff Francisco"

Training on an Amazon Web Services GPU-enabled p2.xlarge instance took 15 minutes per epoch. Over the first 10 epochs, the training loss dropped from about 6.8 to about 5.3. But with so many thousands of words to choose from, things didn't get much better.

Here's the output from epoch 10, starting with the seed, it would be a grief to me to be forced to take any extreme measure . You smile , sir , but I assure you that it really would. GLORP is part of my trade , ,

> it would be a grief to me to be forced to take any extreme measure . You smile , sir , but I assure you that it really would . GLORP is part of my trade , ," I door small little who very lamps into dropped imagine the the GLORP , . his that the would nose , tell . Smith said was the The and is . a know to would are none very had there was are It a Mother upon away my for - and the about are not the for to I open one , it far ?

We'd have to spend a lot of time training this model before the results got interesting.

These examples used tiny RNNs. Using larger RNNs, or even better, transformers (as discussed in Chapter 20) could give us better results in less time.

The Functional API

So far in this chapter we've built our models by placing one layer after another. The *Sequential API* that we've been using was designed for just this sort of architecture. Keras offers a second way to build our models, called the *Functional API*.

The reason for a second API is that sometimes we want to build models that are not strictly sequential. For example, in Chapter 18 we build a model called a *variational autoencoder*. It starts with a sequence of layers, but then it splits into two, with two different layers getting input from the same predecessor, as in Figure B3-93. Then we combine those two layers back into one and continue with a sequential model.

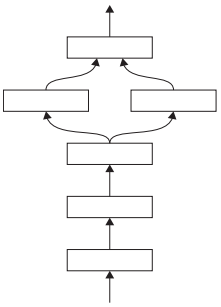


Figure B3-93: This model cannot be made with the Sequential API. We need to use the Functional API to build this.

We can't make a network like Figure B3-93 with the Sequential API, because it assumes that each layer gets input and sends output to no more than one layer.

Using the Functional API, creating layers and connecting them are two separate operations. We can first make whatever layers we need, and then connect them together however we like.

The functional API is powerful, but it can also be complex and subtle. Here we'll stick just the basics that will let us make a model like Figure B3-93. If we just want to make a straight chain of layers, like one the sequential API builds, we can make that with the functional API as well.

The key thing to know about this approach is that each layer is its own object. That is, we create it and assign it to a variable. Once we create a layer, it contains its own weights, parameters, and internal processing. We will then connect that layer to other layers to build the model.

But because each layer is an object, we can use it more than once. Let's look at an imaginary model that we might use for image classification. We've decided that the first two layers will be fully-connected, or `Dense`, layers of 100 and 200 neurons, as in Figure B3-94(a).

We can build both models in Figure B3-94 and use one and then the other. The layers in red are not copied, but shared, so any learning from either network is also used in the other.

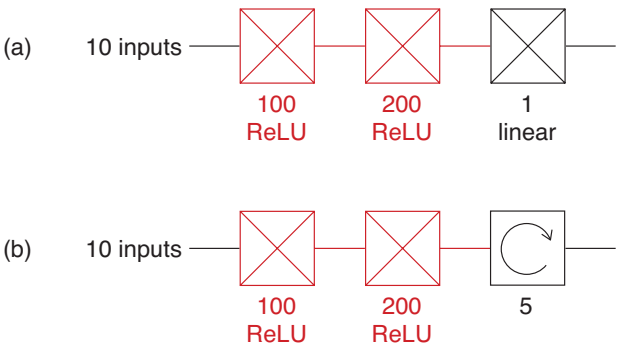


Figure B3-94: Re-using layers in Keras. (a) A small network using two hidden `Dense` layers, shown in red. (b) A different network that re-uses the two red layers from part (a) as the first stages.

Let's suppose that later on we want to build a different model for roughly the same task. But this time we want to put a recurrent layer at the end, as in Figure B3-94(b). We can re-use the first two layers from our first model. These aren't copies, but the same layers, just taking place in this new network. They retain all the weights that they've learned when they were part of the other network. So as we train either model, the other model is trained as well.

It may help to think of the layers, the connections, and the models as three different ideas. We start with a "soup" of layers, all floating around and not connected to anything, as in Figure B3-95(a). Then we decide to build some connections from one layer to the next, as in Figure B3-95(b). That set of connections is a model. Later we might build a second set of connections, making a second model, as in Figure B3-95(c). We're re-using the same layers in each model, just changing where their inputs come from and where the outputs go. When any layer learns, that change will be incorporated into any other model that layer is used in. Any training that happens in any model stays with the layers that learned it, so the other model will benefit from those improved weights.

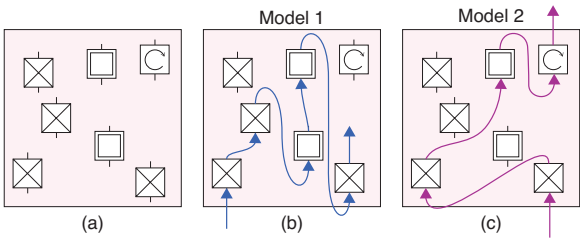


Figure B3-95: In the Functional API, we distinguish the layers from how they're connected. Left: A "soup" of layers. Middle: Connections between layers makes a model. Right: A different set of connections between the layers at the far left that compose another model.

This flexibility in connections and re-use of layers allows us to perform a useful operation called *pre-training*. In this technique, we teach a piece of a network in isolation before we teach the whole thing. The idea is to build a small network with just the layers we want to train, and teach that for a while. This is a useful technique when we anticipate that one piece of the network is going to be take much more time to train than the rest of it. We can teach the difficult part first, in isolation, which will often be much faster than training the whole thing. When those layers have become good at their jobs, we then connect them to the larger model and train up the whole thing.

This idea generalizes even to the level of the model. We can create a model, and then make a second model that includes the first one. We don't even need to explicitly include all the layers. Keras allows us to place one model into another by name, just as if it was a layer.

Input Layers

The Functional API supports all the layers offered by the Sequential API, including those we've seen above. But it requires one additional layer to act as the input layer.

Recall that the input layer is usually implicit, since it's just a place to "park" the incoming data. When we make a sequential model, we tell Keras the size of the input layer with the `input_shape` argument on the first layer, and Keras makes an input layer for us that's the right size to hold one sample of that shape.

In the functional API, it's our job to create that layer explicitly and add it into the model.

The new layer is called an `Input` layer, and it takes one argument called `shape` that tells it the structure of the input. This is identical to how we use `input_shape`, so it's unfortunate that they have different names.

Let's think back to the MNIST data set. To create an input for a piece of flattened MNIST data containing a list of 784 elements, we could write the code in Listing B3-78.

```
input_layer = Input(shape=[784])
```

Listing B3-78: Creating an input layer for 784 elements

A common alternative way to write a one-element list in Python is `(784,)`, where the comma tells the system that this isn't just the number 784 in parentheses, but a list with one element.

Let's suppose our first layer following the `Input` layer was a convolution layer, expecting an input tensor of shape 28 by 28 by 1. Then we could write Listing B3-79.

```
input_layer = Input(shape=[28,28,1])
```

Listing B3-79: Creating an input layer for a tensor of 28 by 28 by 1 elements

Now that we have our input layer, we can move on to making a model.

Making A Functional Model

To build a model in the Functional API, we create each layer as its own object, and then add it to the model by identifying where it gets its input from.

Our first job is to make a layer, which we save in a variable. To use it in a model, we need to specify its inputs. We don't need to specify where the output goes, because the system can figure that out from the other layers. Let's say our current layer is named `layer_1`, and later we create a layer named `layer_2` that says it gets input from `layer_1`. Then it's easy to work out that `layer_2` is one of the outputs of `layer_1`. If we like, multiple layers can take their input from `layer_1`.

Let's imagine a simple model that takes a flattened list of 784 values as input, and returns a single number giving the probability that the image is an MNIST-style digit. We're not categorizing the inputs here, but rather just presenting a single value at the output that tells us if the input is or is not an MNIST digit. Let's propose trying the network of Figure B3-96 for the job. We could easily make this model with the Sequential API, but let's use the Functional API to see how it's done.

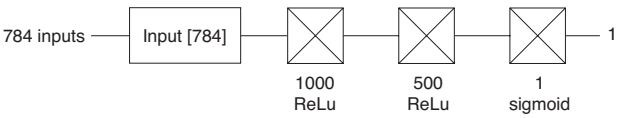


Figure B3-96: A simple network to tell us if an input is an MNIST-style digit image

To create this model, we need an `Input` layer and three `Dense` layers. Listing B3-80 makes the layers and saves each in a variable. We'll call these "unconnected layers," since they're not connected to anything.

```
input = Input(shape=(784,))
dense_1 = Dense(1000, activation='relu')
dense_2 = Dense(500, activation='relu')
output = Dense(1, activation='sigmoid')
```

Listing B3-80: Creating the layers for our network of Figure B3-96

Now we need to connect these layers. We'll create a new object called a "connection layer." Like some of the layers we've seen before, this is really just a wrapper or container. A connection layer points to two objects: an unconnected layer, and another connection layer. This lets us build up a chain of connection layers that define a whole network.

Let's see how this works. On the right of Figure B3-97 we see the four unconnected layers we just made in Listing B3-80. Let's start building connection layers at the bottom, with the input layer. The input layer is a special case that doesn't take input from any other layer. As we said, the connection layer points to two objects. First there's the layer it's referring to, which in this case is the input layer. The other object is the connection layer that provides the input to this layer. Since this layer doesn't take input, we leave that pointer empty. We just built our first connection layer. We'll call this `C1_input`, where `C` refers to this being a connection layer, and the `1` distinguishes it from other such layers we'll make later.

Let's move up now to the first hidden layer, which we called `dense_1`. To build its connection layer, we point its first value at the unconnected layer `dense_1`. Then we point it at the connection layer that provides `dense_1` with input. This is `C1_input`, which we just made. That's it for this connection layer, which we'll call `C1_dense_1`.

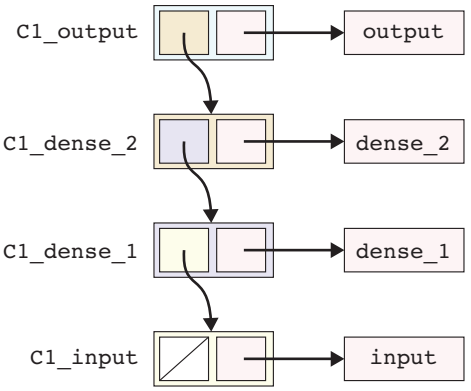


Figure B3-97: Building connection layers. On the right are the original, unconnected layers. On the left are the connection layers. Each connection layer points to an unconnected layer, and to the preceding connection layer. The connection layer for the input layer is a special case.

Moving upwards, we repeat the process for the next two layers.

This chain of layers is everything we need to build a model. When Keras writes the code to route the output of each layer to some other layer, it just follows the chain of connection layers.

The key thing is that the connection layers aren't duplicating the unconnected layers. If we make another connection layer that points to, say, `dense_2`, then we're not modifying `dense_2` in any way.

Listing B3-81 shows the code for Figure B3-97. We make our connection layers implicitly by treating each unconnected layer like a function, and giving it the argument of the connection layer it points to. The syntax can seem a little weird. Keras makes it easy to make the connection layer `C1_input`, which only needs to point to the unconnected `input` layer.

```
# The input layer has no previous connections
C1_input = input
C1_dense_1 = dense_1(C1_input)
C1_dense_2 = dense_2(C1_dense_1)
C1_output = output(C1_dense_2)
```

Listing B3-81: Building our connected layers, combining a layer with its connection to a previous layer. Only `input_layer` does not have an input.

Now we have four new variables, each prefixed with `C1_`. Each one tells us about a layer and where it gets its input from. We can make a model out of these connection layers by calling `Model()` with the input and output connection layers as arguments, as in Listing B3-82.

```
network_1 = Model(C1_input, C1_output)
```

Listing B3-82: Making our model based on the input and output connection layers. The other layers are included implicitly.

Notice that we don't have to specify all the connection layers between the input and the output. Keras can figure out that they're needed because it can follow the chain of connected layers backwards from the output layer to the input layer.

Now that we have our model, we can treat it just like the sequential models we saw above. So we'll call `compile()` to actually build the model and then `fit()` to train it.

Let's say that later we want to play around with this architecture a little. Maybe we'd improve performance by replacing the next-to-final stage of processing in the `dense_2` layer with a convolution layer followed by a flatten layer. We don't want to train from scratch, since our first dense layer and our output layer (also a dense layer) are the same. We'd like to make a second model out of the pieces of the first, but we don't want to disassemble the first model.

This is easy with the Functional API. We start by making a couple of new unconnected layers (our convolution and flatten layers). Now we build up a new set of connection layers, as shown in Figure B3-98. The first two layers, and the last, will re-use the layers from our first model. All of their learned weights come along for the ride.

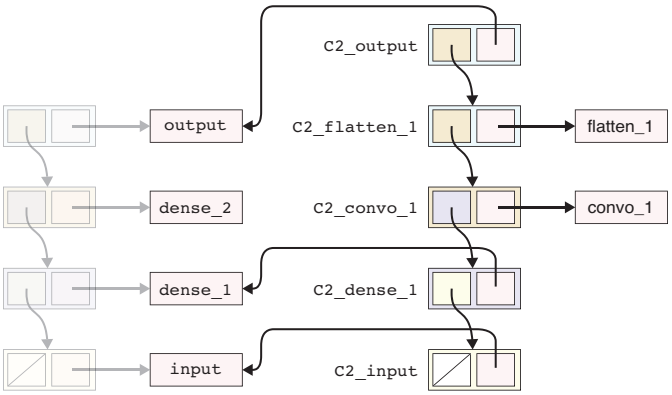


Figure B3-98: Building connections

We can even flip back and forth, training the first model for a little while, then the second, then back to the first.

The code for this new model is shown in Listing B3-83.

```
# define the new layers
convo_1 = Conv2D(32, (5,5))
flotten_1 = Flatten()

# Build the new connection layers
C2_input = input
C2_dense_1 = dense_1(C2_input)
C2_convo_1 = convo_1(C2_dense_1)
C2_flatten_1 = flatten_1(C2_convo_1)
```

```
C2_output = output(C2_flatten_1)

# build the model
model2 = Model(C2_input, C2_output)
```

Listing B3-83: Building our new model using some layers from the first model

Sometimes we don't want to change the shared layers. For example, we might find that `dense_1` changes considerably depending on whether we're training the first or second model. Perhaps we want to keep all the layers in the first model where they are, but just train the two new layers in our second model. In that case we can use the freezing mechanism to prevent any layer from changing. We just set any layer's optional parameter `trainable` to `False` to freeze it, and then set it to `True` if we want to make it trainable again later.

We started this section by considering the difficulty of using the Sequential API to build a branching architecture such as in Figure B3-93. Figure B3-99(a) shows a version of such a model.

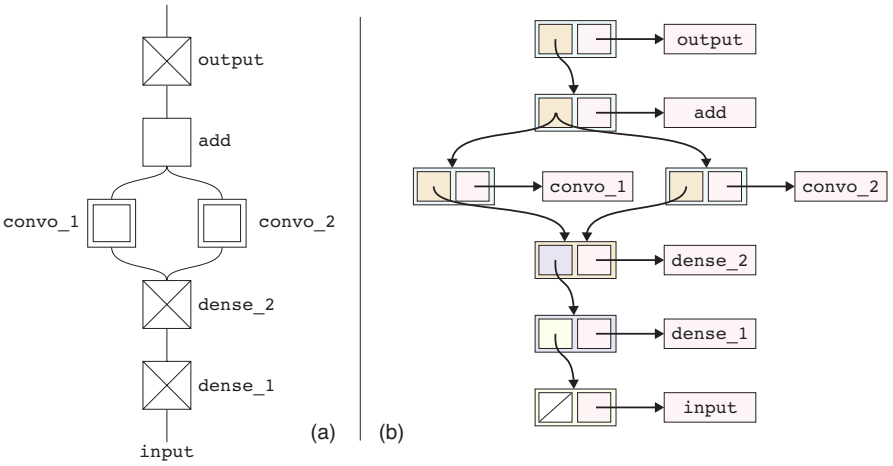


Figure B3-99: A branching architecture using the Functional API. (a) The architecture we'd like to make. (b) A set of connection layers to represent it.

Figure B3-99(b) shows the structure of a set of connection layers that can do the job. Note that the connection layers for both convolution layers get their input from the same connection layer, associated with `dense_2`. This kind of branching doesn't require any special effort when using the Functional API, since we just point our connection layers where we want them to go.

Figure B3-99(b) has a layer called `add` that we haven't discussed. Keras offers us a variety of layers that can combine multiple layers. They're called "Merge Layers," and the Keras documentation lists a half-dozen possibilities for us [Chollet17a]. Each of these layers takes a list of other layers in the model, and combines their outputs. We can also write our own custom

layers if we need something that's not already available. In this case, the layer adds together the output tensors of the two convolution layers. Of course, they must be the same size for this to make sense.

We've seen that the Functional API lets us re-use layers in multiple models, and build models whose connections are not simply a single stack of layers.

Summary

This wraps up our discussion of Keras. There's much more to be found! Guidance for building more complex structures can be found online in various blogs and GitHub repos, as well as the Keras documentation itself.

One of the best ways to learn how to write deep learning code in any language, and for any library, is to look at how other people have done it. The web is filled with blog posts and articles with advice on writing code in Keras, PyTorch, and TensorFlow (as well as other libraries), and there are many, many GitHub repos that contain the complete source code for a tremendous diversity of project applications and complexities.

Any time you're starting a new project, it's often a great idea to look for something that someone has already implemented. You can see how they approached the program, and how they used the library functions to prepare the data, and build and train their learners. It's a great way to learn about useful new routines, and perhaps surprising new ways to use routines you thought you already knew.

In the best case, you can reuse some of their code and save yourself some time and debugging. But even if you can't, you can usually find valuable insight into how to structure your approach to the problem (or, equally valuable, how to *not* structure your approach!).

I often start my projects small, with tiny databases and toy learners. When I've confirmed that the pieces are all working, I can gradually grow a larger system from there, checking my results after each change. Working in a Jupyter notebook makes this really easy, as I can change just one cell at a time and reevaluate the whole notebook, checking that everything still works. If it doesn't, I know exactly where I've introduced the change that broke things. Of course, working in a full development environment such as PyCharm, with breakpoints and debuggers, is also a great way to grow a program. Choose the style (or a different one) that suits you best.

The deep learning software environment is changing and growing rapidly. As you read this, there may be new libraries or frameworks out there that offer tools or conveniences beyond what's provided by the packages I've mentioned here. Don't be afraid of jumping in; you may discover a new favorite way to build your systems!

Deep learning is challenging, rewarding, and fun. It's also dangerous and can deliberately or accidentally cause harm to real people and other living things. There is immense power here. Use it responsibly to make the world a better place.

References

[Cain13]: Cain, Answer to "What Is New Deck Order?" The Magic Cafe, 2013. *https://www.themagiccafe.com/forums/viewtopic.php?topic=501138*

[Chollet17a]: François Chollet, "Keras Documentation," 2017. *https://keras.io/ and https://github.com/fchollet/keras*

[Chollet17b]: François Chollet, "stateful_lstm.py," Keras documentation, 2017. *https://github.com/fchollet/keras/blob/master/examples/stateful_lstm.py*

[Chollet17c]: François Chollet, *Deep Learning with Python*, Manning Publications, 2017.

[Jagtap20]: Rohan Jagtap, "Implementing Custom Data Generators in Keras," 2020. *https://towardsdatascience.com/implementing-custom-data-generators-in-keras-de56f013581c*

[Karpathy15]: Andrej Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks," Andrej Karpathy Blog, 2015. *https://karpathy.github.io/2015/05/21/rnn-effectiveness/*

[Karpathy16a]: Andrej Karpathy, "Convolutional Neural Networks (CNNs/ConvNets)," Stanford CS 231n Course Notes, 2016. *http://cs231n.github.io/convolutional-networks/*

[Karpathy16b]: Andrej Karpathy, "Convolutional Neural Networks for Visual Recognition," Stanford CS 231 Course Notes, 2016. *http://cs231n.github.io/neural-networks-2/*

[LeCun13]: Yann LeCun, Corinna Cortes, Christopher J. C. Burges, "The MNIST Database of Handwritten Digits," 2013. *http://yann.lecun.com/exdb/mnist/*

[PythonWiki17]: Python Wiki contributors, "Generators," The Python Wiki, 2017. *https://wiki.python.org/moin/Generators*

[Snoek16a]: Jasper Snoek, "Spearmint," 2016. *https://github.com/JasperSnoek/spearmint*

[Snoek16b]: Jasper Snoek, "Spearmint" (updated), 2016. *https://github.com/HIPS/Spearmint*

[Springenberg15]: Jost Tobias Springenberg, Alexey Dosovitskiy, Thomas Brox, Martin Riedmiller, "Striving for Simplicity: The All Convolutional Net," ICLR 2015, 2015. *https://arxiv.org/abs/1412.6806*

[Srivastava14]: Nitish Srivastava, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, Ruslan Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting," *Journal of Machine Learning Research*, Number 15, pp. 1929–1958, 2014. *http://jmlr.org/papers/volume15/srivastava14a/srivastava14a.pdf*

Image Credits

Figure B3-41, Eurasian eagle owl, *https://pixabay.com/en/eurasian-eagle-owl-bird-wildlife-1627541*