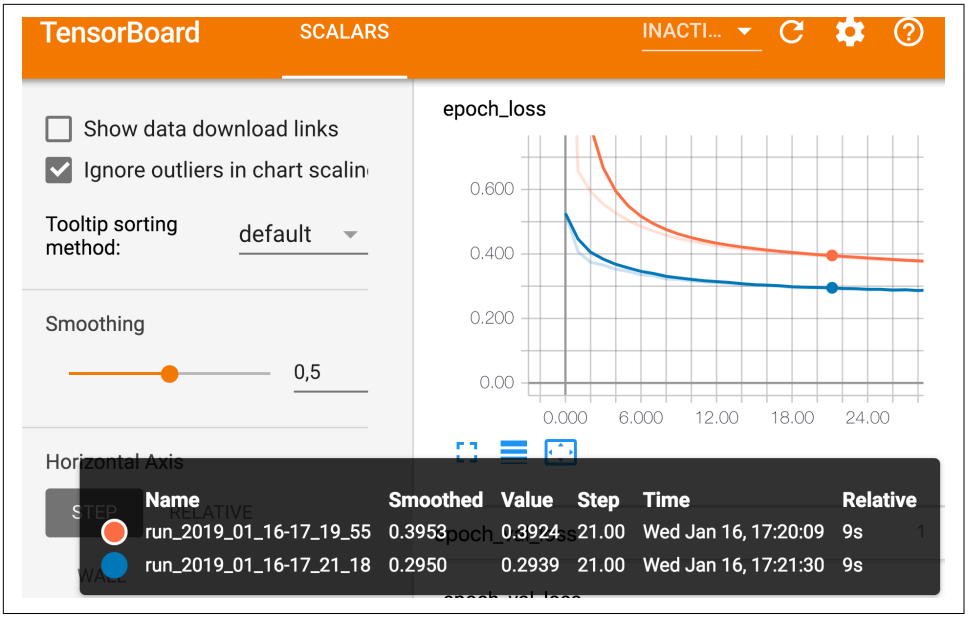

Figure 10-16. Visualizing Learning Curves with TensorBoard

Unfortunately, at the time of writing, no other data is exported by the `TensorBoard` callback, but this issue will probably be fixed by the time you read these lines. In TensorFlow 1, this callback exported the computation graph and many useful statistics: type `help(keras.callbacks.TensorBoard)` to see all the options.

Let's summarize what you learned so far in this chapter: we saw where neural nets came from, what an MLP is and how you can use it for classification and regression, how to build MLPs using tf.keras's Sequential API, or more complex architectures using the Functional API or `Model` Subclassing, you learned how to save and restore a model, use callbacks for checkpointing, early stopping, and more, and finally how to use TensorBoard for visualization. You can already go ahead and use neural networks to tackle many problems! However, you may wonder how to choose the number of hidden layers, the number of neurons in the network, and all the other hyperparameters. Let's look at this now.

Fine-Tuning Neural Network Hyperparameters

The flexibility of neural networks is also one of their main drawbacks: there are many hyperparameters to tweak. Not only can you use any imaginable network architecture, but even in a simple MLP you can change the number of layers, the number of neurons per layer, the type of activation function to use in each layer, the weight initi-

alization logic, and much more. How do you know what combination of hyperparameters is the best for your task?

One option is to simply try many combinations of hyperparameters and see which one works best on the validation set (or using K-fold cross-validation). For this, one approach is simply use `GridSearchCV` or `RandomizedSearchCV` to explore the hyperparameter space, as we did in Chapter 2. For this, we need to wrap our Keras models in objects that mimic regular Scikit-Learn regressors. The first step is to create a function that will build and compile a Keras model, given a set of hyperparameters:

```
def build_model(n_hidden=1, n_neurons=30, learning_rate=3e-3, input_shape=[8]):
    model = keras.models.Sequential()
    options = {"input_shape": input_shape}
    for layer in range(n_hidden):
        model.add(keras.layers.Dense(n_neurons, activation="relu", **options))
        options = {}
    model.add(keras.layers.Dense(1, **options))
    optimizer = keras.optimizers.SGD(learning_rate)
    model.compile(loss="mse", optimizer=optimizer)
    return model
```

This function creates a simple `Sequential` model for univariate regression (only one output neuron), with the given input shape and the given number of hidden layers and neurons, and it compiles it using an `SGD` optimizer configured with the given learning rate. The `options` dict is used to ensure that the first layer is properly given the input shape (note that if `n_hidden=0`, the first layer will be the output layer). It is good practice to provide reasonable defaults to as many hyperparameters as you can, as Scikit-Learn does.

Next, let's create a `KerasRegressor` based on this `build_model()` function:

```
keras_reg = keras.wrappers.scikit_learn.KerasRegressor(build_model)
```

The `KerasRegressor` object is a thin wrapper around the Keras model built using `build_model()`. Since we did not specify any hyperparameter when creating it, it will just use the default hyperparameters we defined in `build_model()`. Now we can use this object like a regular Scikit-Learn regressor: we can train it using its `fit()` method, then evaluate it using its `score()` method, and use it to make predictions using its `predict()` method. Note that any extra parameter you pass to the `fit()` method will simply get passed to the underlying Keras model. Also note that the score will be the opposite of the MSE because Scikit-Learn wants scores, not losses (i.e., higher should be better).

```
keras_reg.fit(X_train, y_train, epochs=100,
              validation_data=(X_valid, y_valid),
              callbacks=[keras.callbacks.EarlyStopping(patience=10)])
mse_test = keras_reg.score(X_test, y_test)
y_pred = keras_reg.predict(X_new)
```

However, we do not actually want to train and evaluate a single model like this, we want to train hundreds of variants and see which one performs best on the validation set. Since there are many hyperparameters, it is preferable to use a randomized search rather than grid search (as we discussed in Chapter 2). Let's try to explore the number of hidden layers, the number of neurons and the learning rate:

```
from scipy.stats import reciprocal
from sklearn.model_selection import RandomizedSearchCV

param_distribs = {
    "n_hidden": [0, 1, 2, 3],
    "n_neurons": np.arange(1, 100),
    "learning_rate": reciprocal(3e-4, 3e-2),
}

rnd_search_cv = RandomizedSearchCV(keras_reg, param_distribs, n_iter=10, cv=3)
rnd_search_cv.fit(X_train, y_train, epochs=100,
                  validation_data=(X_valid, y_valid),
                  callbacks=[keras.callbacks.EarlyStopping(patience=10)])
```

As you can see, this is identical to what we did in Chapter 2, with the exception that we pass extra parameters to the `fit()` method: they simply get relayed to the underlying Keras models. Note that `RandomizedSearchCV` uses K-fold cross-validation, so it does not use `X_valid` and `y_valid`. These are just used for early stopping.

The exploration may last many hours depending on the hardware, the size of the dataset, the complexity of the model and the value of `n_iter` and `cv`. When it is over, you can access the best parameters found, the best score, and the trained Keras model like this:

```
>>> rnd_search_cv.best_params_
{'learning_rate': 0.0033625641252688094, 'n_hidden': 2, 'n_neurons': 42}
>>> rnd_search_cv.best_score_
-0.3189529188278931
>>> model = rnd_search_cv.best_estimator_.model
```

You can now save this model, evaluate it on the test set, and if you are satisfied with its performance, deploy it to production. Using randomized search is not too hard, and it works well for many fairly simple problems. However, when training is slow (e.g., for more complex problems with larger datasets), this approach will only explore a tiny portion of the hyperparameter space. You can partially alleviate this problem by assisting the search process manually: first run a quick random search using wide ranges of hyperparameter values, then run another search using smaller ranges of values centered on the best ones found during the first run, and so on. This will hopefully zoom in to a good set of hyperparameters. However, this is very time consuming, and probably not the best use of your time.

Fortunately, there are many techniques to explore a search space much more efficiently than randomly. Their core idea is simple: when a region of the space turns out

to be good, it should be explored more. This takes care of the "zooming" process for you and leads to much better solutions in much less time. Here are a few Python libraries you can use to optimize hyperparameters:

- Hyperopt: a popular Python library for optimizing over all sorts of complex search spaces (including real values such as the learning rate, or discrete values such as the number of layers).
- Hyperas, kopt or Talos: optimizing hyperparameters for Keras model (the first two are based on Hyperopt).
- Scikit-Optimize (skopt): a general-purpose optimization library. The `BayesSearchCV` class performs Bayesian optimization using an interface similar to `GridSearchCV`.
- Spearmint: a Bayesian optimization library.
- Sklearn-Deap: a hyperparameter optimization library based on evolutionary algorithms, also with a `GridSearchCV`-like interface.
- And many more!

Moreover, many companies offer services for hyperparameter optimization. For example Google Cloud ML Engine has a hyperparameter tuning service. Other companies provide APIs for hyperparameter optimization, such as Arimo, SigOpt, Oscar and many more.

Hyperparameter tuning is still an active area of research. Evolutionary algorithms are making a comeback lately. For example, check out DeepMind's excellent 2017 paper[16], where they jointly optimize a population of models and their hyperparameters. Google also used an evolutionary approach, not just to search for hyperparameters, but also to look for the best neural network architecture for the problem. They call this *AutoML*, and it is already available as a cloud service. Perhaps the days of building neural networks manually will soon be over? Check out Google's post on this topic. In fact, evolutionary algorithms have also been used successfully to train individual neural networks, replacing the ubiquitous Gradient Descent! See this 2017 post by Uber where they introduce their *Deep Neuroevolution* technique.

Despite all this exciting progress, and all these tools and services, it still helps to have an idea of what values are reasonable for each hyperparameter, so you can build a quick prototype, and restrict the search space. Here are a few guidelines for choosing the number of hidden layers and neurons in an MLP, and selecting good values for some of the main hyperparameters.

16 "Population Based Training of Neural Networks," Max Jaderberg et al. (2017).

Number of Hidden Layers

For many problems, you can just begin with a single hidden layer and you will get reasonable results. It has actually been shown that an MLP with just one hidden layer can model even the most complex functions provided it has enough neurons. For a long time, these facts convinced researchers that there was no need to investigate any deeper neural networks. But they overlooked the fact that deep networks have a much higher *parameter efficiency* than shallow ones: they can model complex functions using exponentially fewer neurons than shallow nets, allowing them to reach much better performance with the same amount of training data.

To understand why, suppose you are asked to draw a forest using some drawing software, but you are forbidden to use copy/paste. You would have to draw each tree individually, branch per branch, leaf per leaf. If you could instead draw one leaf, copy/paste it to draw a branch, then copy/paste that branch to create a tree, and finally copy/paste this tree to make a forest, you would be finished in no time. Real-world data is often structured in such a hierarchical way and Deep Neural Networks automatically take advantage of this fact: lower hidden layers model low-level structures (e.g., line segments of various shapes and orientations), intermediate hidden layers combine these low-level structures to model intermediate-level structures (e.g., squares, circles), and the highest hidden layers and the output layer combine these intermediate structures to model high-level structures (e.g., faces).

Not only does this hierarchical architecture help DNNs converge faster to a good solution, it also improves their ability to generalize to new datasets. For example, if you have already trained a model to recognize faces in pictures, and you now want to train a new neural network to recognize hairstyles, then you can kickstart training by reusing the lower layers of the first network. Instead of randomly initializing the weights and biases of the first few layers of the new neural network, you can initialize them to the value of the weights and biases of the lower layers of the first network. This way the network will not have to learn from scratch all the low-level structures that occur in most pictures; it will only have to learn the higher-level structures (e.g., hairstyles). This is called *transfer learning*.

In summary, for many problems you can start with just one or two hidden layers and it will work just fine (e.g., you can easily reach above 97% accuracy on the MNIST dataset using just one hidden layer with a few hundred neurons, and above 98% accuracy using two hidden layers with the same total amount of neurons, in roughly the same amount of training time). For more complex problems, you can gradually ramp up the number of hidden layers, until you start overfitting the training set. Very complex tasks, such as large image classification or speech recognition, typically require networks with dozens of layers (or even hundreds, but not fully connected ones, as we will see in Chapter 14), and they need a huge amount of training data. However, you will rarely have to train such networks from scratch: it is much more common to

reuse parts of a pretrained state-of-the-art network that performs a similar task. Training will be a lot faster and require much less data (we will discuss this in Chapter 11).

Number of Neurons per Hidden Layer

Obviously the number of neurons in the input and output layers is determined by the type of input and output your task requires. For example, the MNIST task requires 28 x 28 = 784 input neurons and 10 output neurons.

As for the hidden layers, it used to be a common practice to size them to form a pyramid, with fewer and fewer neurons at each layer—the rationale being that many low-level features can coalesce into far fewer high-level features. For example, a typical neural network for MNIST may have three hidden layers, the first with 300 neurons, the second with 200, and the third with 100. However, this practice has been largely abandoned now, as it seems that simply using the same number of neurons in all hidden layers performs just as well in most cases, or even better, and there is just one hyperparameter to tune instead of one per layer—for example, all hidden layers could simply have 150 neurons. However, depending on the dataset, it can sometimes help to make the first hidden layer bigger than the others.

Just like for the number of layers, you can try increasing the number of neurons gradually until the network starts overfitting. In general you will get more bang for the buck by increasing the number of layers than the number of neurons per layer. Unfortunately, as you can see, finding the perfect amount of neurons is still somewhat of a dark art.

A simpler approach is to pick a model with more layers and neurons than you actually need, then use early stopping to prevent it from overfitting (and other regularization techniques, such as *dropout*, as we will see in Chapter 11). This has been dubbed the "stretch pants" approach:[17] instead of wasting time looking for pants that perfectly match your size, just use large stretch pants that will shrink down to the right size.

Learning Rate, Batch Size and Other Hyperparameters

The number of hidden layers and neurons are not the only hyperparameters you can tweak in an MLP. Here are some of the most important ones, and some tips on how to set them:

- The learning rate is arguably the most important hyperparameter. In general, the optimal learning rate is about half of the maximum learning rate (i.e., the learn-

17 By Vincent Vanhoucke in his Deep Learning class on Udacity.com.

ing rate above which the training algorithm diverges, as we saw in Chapter 4). So a simple approach for tuning the learning rate is to start with a large value that makes the training algorithm diverge, then divide this value by 3 and try again, and repeat until the training algorithm stops diverging. At that point, you generally won't be too far from the optimal learning rate. That said, it is sometimes useful to reduce the learning rate during training: we will discuss this in Chapter 11.

- Choosing a better optimizer than plain old Mini-batch Gradient Descent (and tuning its hyperparameters) is also quite important. We will discuss this in Chapter 11.
- The batch size can also have a significant impact on your model's performance and the training time. In general the optimal batch size will be lower than 32 (in April 2018, Yann Lecun even tweeted "*Friends don't let friends use mini-batches larger than 32*"). A small batch size ensures that each training iteration is very fast, and although a large batch size will give a more precise estimate of the gradients, in practice this does not matter much since the optimization landscape is quite complex and the direction of the true gradients do not point precisely in the direction of the optimum. However, having a batch size greater than 10 helps take advantage of hardware and software optimizations, in particular for matrix multiplications, so it will speed up training. Moreover, if you use *Batch Normalization* (see Chapter 11), the batch size should not be too small (in general no less than 20).
- We discussed the choice of the activation function earlier in this chapter: in general, the ReLU activation function will be a good default for all hidden layers. For the output layer, it really depends on your task.
- In most cases, the number of training iterations does not actually need to be tweaked: just use early stopping instead.

For more best practices, make sure to read Yoshua Bengio's great 2012 paper[18], which presents many practical recommendations for deep networks.

This concludes this introduction to artificial neural networks and their implementation with Keras. In the next few chapters, we will discuss techniques to train very deep nets, we will see how to customize your models using TensorFlow's lower-level API and how to load and preprocess data efficiently using the Data API, and we will dive into other popular neural network architectures: convolutional neural networks for image processing, recurrent neural networks for sequential data, autoencoders for

18 "Practical recommendations for gradient-based training of deep architectures," Yoshua Bengio (2012).

representation learning, and generative adversarial networks to model and generate data.[19]

Exercises

1. Visit the TensorFlow Playground at *https://playground.tensorflow.org/*

   - Layers and patterns: try training the default neural network by clicking the run button (top left). Notice how it quickly finds a good solution for the classification task. Notice that the neurons in the first hidden layer have learned simple patterns, while the neurons in the second hidden layer have learned to combine the simple patterns of the first hidden layer into more complex patterns. In general, the more layers, the more complex the patterns can be.
   - Activation function: try replacing the Tanh activation function with the ReLU activation function, and train the network again. Notice that it finds a solution even faster, but this time the boundaries are linear. This is due to the shape of the ReLU function.
   - Local minima: modify the network architecture to have just one hidden layer with three neurons. Train it multiple times (to reset the network weights, click the reset button next to the play button). Notice that the training time varies a lot, and sometimes it even gets stuck in a local minimum.
   - Too small: now remove one neuron to keep just 2. Notice that the neural network is now incapable of finding a good solution, even if you try multiple times. The model has too few parameters and it systematically underfits the training set.
   - Large enough: next, set the number of neurons to 8 and train the network several times. Notice that it is now consistently fast and never gets stuck. This highlights an important finding in neural network theory: large neural networks almost never get stuck in local minima, and even when they do these local optima are almost as good as the global optimum. However, they can still get stuck on long plateaus for a long time.
   - Deep net and vanishing gradients: now change the dataset to be the spiral (bottom right dataset under "DATA"). Change the network architecture to have 4 hidden layers with 8 neurons each. Notice that training takes much longer, and often gets stuck on plateaus for long periods of time. Also notice that the neurons in the highest layers (i.e. on the right) tend to evolve faster than the neurons in the lowest layers (i.e. on the left). This problem, called the "vanishing gradients" problem, can be alleviated using better weight initialization and

19 A few extra ANN architectures are presented in ???.

other techniques, better optimizers (such as AdaGrad or Adam), or using Batch Normalization.

- More: go ahead and play with the other parameters to get a feel of what they do. In fact, you should definitely play with this UI for at least one hour, it will grow your intuitions about neural networks significantly.

2. Draw an ANN using the original artificial neurons (like the ones in Figure 10-3) that computes $A \oplus B$ (where $\oplus$ represents the XOR operation). Hint: $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$.
3. Why is it generally preferable to use a Logistic Regression classifier rather than a classical Perceptron (i.e., a single layer of threshold logic units trained using the Perceptron training algorithm)? How can you tweak a Perceptron to make it equivalent to a Logistic Regression classifier?
4. Why was the logistic activation function a key ingredient in training the first MLPs?
5. Name three popular activation functions. Can you draw them?
6. Suppose you have an MLP composed of one input layer with 10 passthrough neurons, followed by one hidden layer with 50 artificial neurons, and finally one output layer with 3 artificial neurons. All artificial neurons use the ReLU activation function.

   - What is the shape of the input matrix $\mathbf{X}$?
   - What about the shape of the hidden layer's weight vector $\mathbf{W}_h$, and the shape of its bias vector $\mathbf{b}_h$?
   - What is the shape of the output layer's weight vector $\mathbf{W}_o$, and its bias vector $\mathbf{b}_o$?
   - What is the shape of the network's output matrix $\mathbf{Y}$?
   - Write the equation that computes the network's output matrix $\mathbf{Y}$ as a function of $\mathbf{X}$, $\mathbf{W}_h$, $\mathbf{b}_h$, $\mathbf{W}_o$ and $\mathbf{b}_o$.

7. How many neurons do you need in the output layer if you want to classify email into spam or ham? What activation function should you use in the output layer? If instead you want to tackle MNIST, how many neurons do you need in the output layer, using what activation function? Answer the same questions for getting your network to predict housing prices as in Chapter 2.
8. What is backpropagation and how does it work? What is the difference between backpropagation and reverse-mode autodiff?
9. Can you list all the hyperparameters you can tweak in an MLP? If the MLP overfits the training data, how could you tweak these hyperparameters to try to solve the problem?

10. Train a deep MLP on the MNIST dataset and see if you can get over 98% precision. Try adding all the bells and whistles (i.e., save checkpoints, use early stopping, plot learning curves using TensorBoard, and so on).

Solutions to these exercises are available in ???.