For the solutions to exercises 9 and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 9: Up and Running with TensorFlow

1. Main benefits and drawbacks of creating a computation graph rather than directly executing the computations:
   - Main benefits:
     - TensorFlow can automatically compute the gradients for you (using reverse-mode autodiff).
     - TensorFlow can take care of running the operations in parallel in different threads.
     - It makes it easier to run the same model across different devices.
     - It simplifies introspection—for example, to view the model in TensorBoard.
   - Main drawbacks:
     - It makes the learning curve steeper.
     - It makes step-by-step debugging harder.
2. Yes, the statement `a_val = a.eval(session=sess)` is indeed equivalent to `a_val = sess.run(a)`.
3. No, the statement `a_val, b_val = a.eval(session=sess), b.eval(session=sess)` is not equivalent to `a_val, b_val = sess.run([a, b])`. Indeed, the first statement runs the graph twice (once to compute `a`, once to compute `b`), while the second statement runs the graph only once. If any of these operations (or the ops they depend on) have side effects (e.g., a variable is modified, an item is inserted in a queue, or a reader reads a file), then the effects will be different. If they don't have side effects, both statements will return the same result, but the second statement will be faster than the first.
4. No, you cannot run two graphs in the same session. You would have to merge the graphs into a single graph first.
5. In local TensorFlow, sessions manage variable values, so if you create a graph `g` containing a variable `w`, then start two threads and open a local session in each thread, both using the same graph `g`, then each session will have its own copy of the variable `w`. However, in distributed TensorFlow, variable values are stored in containers managed by the cluster, so if both sessions connect to the same cluster and use the same container, then they will share the same variable value for `w`.
6. A variable is initialized when you call its initializer, and it is destroyed when the session ends. In distributed TensorFlow, variables live in containers on the clus-

ter, so closing a session will not destroy the variable. To destroy a variable, you need to clear its container.

7. Variables and placeholders are extremely different, but beginners often confuse them:

   - A variable is an operation that holds a value. If you run the variable, it returns that value. Before you can run it, you need to initialize it. You can change the variable's value (for example, by using an assignment operation). It is stateful: the variable keeps the same value upon successive runs of the graph. It is typically used to hold model parameters but also for other purposes (e.g., to count the global training step).
   - Placeholders technically don't do much: they just hold information about the type and shape of the tensor they represent, but they have no value. In fact, if you try to evaluate an operation that depends on a placeholder, you must feed TensorFlow the value of the placeholder (using the `feed_dict` argument) or else you will get an exception. Placeholders are typically used to feed training or test data to TensorFlow during the execution phase. They are also useful to pass a value to an assignment node, to change the value of a variable (e.g., model weights).

8. If you run the graph to evaluate an operation that depends on a placeholder but you don't feed its value, you get an exception. If the operation does not depend on the placeholder, then no exception is raised.
9. When you run a graph, you can feed the output value of any operation, not just the value of placeholders. In practice, however, this is rather rare (it can be useful, for example, when you are caching the output of frozen layers; see Chapter 11).
10. You can specify a variable's initial value when constructing the graph, and it will be initialized later when you run the variable's initializer during the execution phase. If you want to change that variable's value to anything you want during the execution phase, then the simplest option is to create an assignment node (during the graph construction phase) using the `tf.assign()` function, passing the variable and a placeholder as parameters. During the execution phase, you can run the assignment operation and feed the variable's new value using the placeholder.

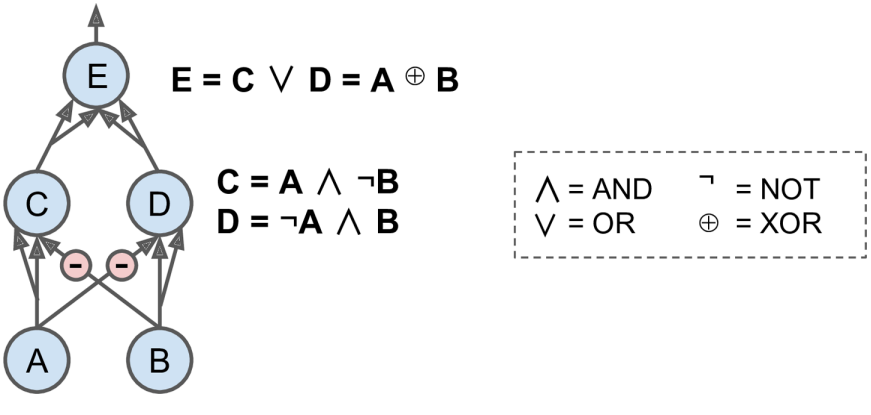
```
import tensorflow as tf

x = tf.Variable(tf.random_uniform(shape=(), minval=0.0, maxval=1.0))
x_new_val = tf.placeholder(shape=(), dtype=tf.float32)
x_assign = tf.assign(x, x_new_val)

with tf.Session():
    x.initializer.run() # random number is sampled *now*
    print(x.eval()) # 0.646157 (some random number)
```

```
x_assign.eval(feed_dict={x_new_val: 5.0})
print(x.eval()) # 5.0
```

11. Reverse-mode autodiff (implemented by TensorFlow) needs to traverse the graph only twice in order to compute the gradients of the cost function with regards to any number of variables. On the other hand, forward-mode autodiff would need to run once for each variable (so 10 times if we want the gradients with regards to 10 different variables). As for symbolic differentiation, it would build a different graph to compute the gradients, so it would not traverse the original graph at all (except when building the new gradients graph). A highly optimized symbolic differentiation system could potentially run the new gradients graph only once to compute the gradients with regards to all variables, but that new graph may be horribly complex and inefficient compared to the original graph.
12. See the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 10: Introduction to Artificial Neural Networks

1. Here is a neural network based on the original artificial neurons that computes $A \oplus B$ (where $\oplus$ represents the exclusive OR), using the fact that $A \oplus B = (A \wedge \neg B) \vee (\neg A \wedge B)$. There are other solutions—for example, using the fact that $A \oplus B = (A \vee B) \wedge \neg(A \wedge B)$, or the fact that $A \oplus B = (A \vee B) \wedge (\neg A \vee \wedge B)$, and so on.



2. A classical Perceptron will converge only if the dataset is linearly separable, and it won't be able to estimate class probabilities. In contrast, a Logistic Regression classifier will converge to a good solution even if the dataset is not linearly separable, and it will output class probabilities. If you change the Perceptron's activation function to the logistic activation function (or the softmax activation function if there are multiple neurons), and if you train it using Gradient Descent

(or some other optimization algorithm minimizing the cost function, typically cross entropy), then it becomes equivalent to a Logistic Regression classifier.

3. The logistic activation function was a key ingredient in training the first MLPs because its derivative is always nonzero, so Gradient Descent can always roll down the slope. When the activation function is a step function, Gradient Descent cannot move, as there is no slope at all.
4. The step function, the logistic function, the hyperbolic tangent, the rectified linear unit (see Figure 10-8). See Chapter 11 for other examples, such as ELU and variants of the ReLU.
5. Considering the MLP described in the question: 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.

   - The shape of the input matrix $\mathbf{X}$ is $m \times 10$, where m represents the training batch size.
   - The shape of the hidden layer's weight vector $\mathbf{W}_h$ is 10×50 and the length of its bias vector $\mathbf{b}_h$ is 50.
   - The shape of the output layer's weight vector $\mathbf{W}_o$ is 50×3, and the length of its bias vector $\mathbf{b}_o$ is 3.
   - The shape of the network's output matrix $\mathbf{Y}$ is $m \times 3$.
   - $\mathbf{Y} = \text{ReLU}(\text{ReLU}(\mathbf{X} \cdot \mathbf{W}_h + \mathbf{b}_h) \cdot \mathbf{W}_o + \mathbf{b}_o)$. Recall that the ReLU function just sets every negative number in the matrix to zero. Also note that when you are adding a bias vector to a matrix, it is added to every single row in the matrix, which is called *broadcasting*.

6. To classify email into spam or ham, you just need one neuron in the output layer of a neural network—for example, indicating the probability that the email is spam. You would typically use the logistic activation function in the output layer when estimating a probability. If instead you want to tackle MNIST, you need 10 neurons in the output layer, and you must replace the logistic function with the softmax activation function, which can handle multiple classes, outputting one probability per class. Now, if you want your neural network to predict housing prices like in Chapter 2, then you need one output neuron, using no activation function at all in the output layer.[4]

4 When the values to predict can vary by many orders of magnitude, then you may want to predict the logarithm of the target value rather than the target value directly. Simply computing the exponential of the neural network's output will give you the estimated value (since $\exp(\log v) = v$).

7. Backpropagation is a technique used to train artificial neural networks. It first computes the gradients of the cost function with regards to every model parameter (all the weights and biases), and then it performs a Gradient Descent step using these gradients. This backpropagation step is typically performed thousands or millions of times, using many training batches, until the model parameters converge to values that (hopefully) minimize the cost function. To compute the gradients, backpropagation uses reverse-mode autodiff (although it wasn't called that when backpropagation was invented, and it has been reinvented several times). Reverse-mode autodiff performs a forward pass through a computation graph, computing every node's value for the current training batch, and then it performs a reverse pass, computing all the gradients at once (see Appendix D for more details). So what's the difference? Well, backpropagation refers to the whole process of training an artificial neural network using multiple backpropagation steps, each of which computes gradients and uses them to perform a Gradient Descent step. In contrast, reverse-mode autodiff is a simply a technique to compute gradients efficiently, and it happens to be used by backpropagation.
8. Here is a list of all the hyperparameters you can tweak in a basic MLP: the number of hidden layers, the number of neurons in each hidden layer, and the activation function used in each hidden layer and in the output layer.[5] In general, the ReLU activation function (or one of its variants; see Chapter 11) is a good default for the hidden layers. For the output layer, in general you will want the logistic activation function for binary classification, the softmax activation function for multiclass classification, or no activation function for regression.

   If the MLP overfits the training data, you can try reducing the number of hidden layers and reducing the number of neurons per hidden layer.
9. See the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 11: Training Deep Neural Nets

1. No, all weights should be sampled independently; they should not all have the same initial value. One important goal of sampling weights randomly is to break symmetries: if all the weights have the same initial value, even if that value is not zero, then symmetry is not broken (i.e., all neurons in a given layer are equivalent), and backpropagation will be unable to break it. Concretely, this means that

5 In Chapter 11 we discuss many techniques that introduce additional hyperparameters: type of weight initialization, activation function hyperparameters (e.g., amount of leak in leaky ReLU), Gradient Clipping threshold, type of optimizer and its hyperparameters (e.g., the momentum hyperparameter when using a `MomentumOptimizer`), type of regularization for each layer, and the regularization hyperparameters (e.g., dropout rate when using dropout) and so on.

all the neurons in any given layer will always have the same weights. It's like having just one neuron per layer, and much slower. It is virtually impossible for such a configuration to converge to a good solution.

2. It is perfectly fine to initialize the bias terms to zero. Some people like to initialize them just like weights, and that's okay too; it does not make much difference.
3. A few advantages of the ELU function over the ReLU function are:

   - It can take on negative values, so the average output of the neurons in any given layer is typically closer to 0 than when using the ReLU activation function (which never outputs negative values). This helps alleviate the vanishing gradients problem.
   - It always has a nonzero derivative, which avoids the dying units issue that can affect ReLU units.
   - It is smooth everywhere, whereas the ReLU's slope abruptly jumps from 0 to 1 at $z = 0$. Such an abrupt change can slow down Gradient Descent because it will bounce around $z = 0$.

4. The ELU activation function is a good default. If you need the neural network to be as fast as possible, you can use one of the leaky ReLU variants instead (e.g., a simple leaky ReLU using the default hyperparameter value). The simplicity of the ReLU activation function makes it many people's preferred option, despite the fact that they are generally outperformed by the ELU and leaky ReLU. However, the ReLU activation function's capability of outputting precisely zero can be useful in some cases (e.g., see Chapter 15). The hyperbolic tangent (tanh) can be useful in the output layer if you need to output a number between –1 and 1, but nowadays it is not used much in hidden layers. The logistic activation function is also useful in the output layer when you need to estimate a probability (e.g., for binary classification), but it is also rarely used in hidden layers (there are exceptions—for example, for the coding layer of variational autoencoders; see Chapter 15). Finally, the softmax activation function is useful in the output layer to output probabilities for mutually exclusive classes, but other than that it is rarely (if ever) used in hidden layers.
5. If you set the `momentum` hyperparameter too close to 1 (e.g., 0.99999) when using a `MomentumOptimizer`, then the algorithm will likely pick up a lot of speed, hopefully roughly toward the global minimum, but then it will shoot right past the minimum, due to its momentum. Then it will slow down and come back, accelerate again, overshoot again, and so on. It may oscillate this way many times before converging, so overall it will take much longer to converge than with a smaller `momentum` value.
6. One way to produce a sparse model (i.e., with most weights equal to zero) is to train the model normally, then zero out tiny weights. For more sparsity, you can

apply ℓ_1 regularization during training, which pushes the optimizer toward sparsity. A third option is to combine ℓ_1 regularization with *dual averaging*, using TensorFlow's `FTRLOptimizer` class.

7. Yes, dropout does slow down training, in general roughly by a factor of two. However, it has no impact on inference since it is only turned on during training.

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 12: Distributing TensorFlow Across Devices and Servers

1. When a TensorFlow process starts, it grabs all the available memory on all GPU devices that are visible to it, so if you get a `CUDA_ERROR_OUT_OF_MEMORY` when starting your TensorFlow program, it probably means that other processes are running that have already grabbed all the memory on at least one visible GPU device (most likely it is another TensorFlow process). To fix this problem, a trivial solution is to stop the other processes and try again. However, if you need all processes to run simultaneously, a simple option is to dedicate different devices to each process, by setting the `CUDA_VISIBLE_DEVICES` environment variable appropriately for each device. Another option is to configure TensorFlow to grab only part of the GPU memory, instead of all of it, by creating a `ConfigProto`, setting its `gpu_options.per_process_gpu_memory_fraction` to the proportion of the total memory that it should grab (e.g., 0.4), and using this `ConfigProto` when opening a session. The last option is to tell TensorFlow to grab memory only when it needs it by setting the `gpu_options.allow_growth` to `True`. However, this last option is usually not recommended because any memory that TensorFlow grabs is never released, and it is harder to guarantee a repeatable behavior (there may be race conditions depending on which processes start first, how much memory they need during training, and so on).
2. By pinning an operation on a device, you are telling TensorFlow that this is where you would like this operation to be placed. However, some constraints may prevent TensorFlow from honoring your request. For example, the operation may have no implementation (called a *kernel*) for that particular type of device. In this case, TensorFlow will raise an exception by default, but you can configure it to fall back to the CPU instead (this is called *soft placement*). Another example is an operation that can modify a variable; this operation and the variable need to be collocated. So the difference between pinning an operation and placing an operation is that pinning is what you ask TensorFlow ("Please place this operation on GPU #1") while placement is what TensorFlow actually ends up doing ("Sorry, falling back to the CPU").

3. If you are running on a GPU-enabled TensorFlow installation, and you just use the default placement, then if all operations have a GPU kernel (i.e., a GPU implementation), yes, they will all be placed on the first GPU. However, if one or more operations do not have a GPU kernel, then by default TensorFlow will raise an exception. If you configure TensorFlow to fall back to the CPU instead (soft placement), then all operations will be placed on the first GPU except the ones without a GPU kernel and all the operations that must be collocated with them (see the answer to the previous exercise).
4. Yes, if you pin a variable to `/gpu:0`, it can be used by operations placed on `/gpu:1`. TensorFlow will automatically take care of adding the appropriate operations to transfer the variable's value across devices. The same goes for devices located on different servers (as long as they are part of the same cluster).
5. Yes, two operations placed on the same device can run in parallel: TensorFlow automatically takes care of running operations in parallel (on different CPU cores or different GPU threads), as long as no operation depends on another operation's output. Moreover, you can start multiple sessions in parallel threads (or processes), and evaluate operations in each thread. Since sessions are independent, TensorFlow will be able to evaluate any operation from one session in parallel with any operation from another session.
6. Control dependencies are used when you want to postpone the evaluation of an operation X until after some other operations are run, even though these operations are not required to compute X. This is useful in particular when X would occupy a lot of memory and you only need it later in the computation graph, or if X uses up a lot of I/O (for example, it requires a large variable value located on a different device or server) and you don't want it to run at the same time as other I/O-hungry operations, to avoid saturating the bandwidth.
7. You're in luck! In distributed TensorFlow, the variable values live in containers managed by the cluster, so even if you close the session and exit the client program, the model parameters are still alive and well on the cluster. You simply need to open a new session to the cluster and save the model (make sure you don't call the variable initializers or restore a previous model, as this would destroy your precious new model!).

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 13: Convolutional Neural Networks

1. These are the main advantages of a CNN over a fully connected DNN for image classification:

- Because consecutive layers are only partially connected and because it heavily reuses its weights, a CNN has many fewer parameters than a fully connected DNN, which makes it much faster to train, reduces the risk of overfitting, and requires much less training data.
- When a CNN has learned a kernel that can detect a particular feature, it can detect that feature anywhere on the image. In contrast, when a DNN learns a feature in one location, it can detect it only in that particular location. Since images typically have very repetitive features, CNNs are able to generalize much better than DNNs for image processing tasks such as classification, using fewer training examples.
- Finally, a DNN has no prior knowledge of how pixels are organized; it does not know that nearby pixels are close. A CNN's architecture embeds this prior knowledge. Lower layers typically identify features in small areas of the images, while higher layers combine the lower-level features into larger features. This works well with most natural images, giving CNNs a decisive head start compared to DNNs.

2. Let's compute how many parameters the CNN has. Since its first convolutional layer has 3 × 3 kernels, and the input has three channels (red, green, and blue), then each feature map has 3 × 3 × 3 weights, plus a bias term. That's 28 parameters per feature map. Since this first convolutional layer has 100 feature maps, it has a total of 2,800 parameters. The second convolutional layer has 3 × 3 kernels, and its input is the set of 100 feature maps of the previous layer, so each feature map has 3 × 3 × 100 = 900 weights, plus a bias term. Since it has 200 feature maps, this layer has 901 × 200 = 180,200 parameters. Finally, the third and last convolutional layer also has 3 × 3 kernels, and its input is the set of 200 feature maps of the previous layers, so each feature map has 3 × 3 × 200 = 1,800 weights, plus a bias term. Since it has 400 feature maps, this layer has a total of 1,801 × 400 = 720,400 parameters. All in all, the CNN has 2,800 + 180,200 + 720,400 = 903,400 parameters.

   Now let's compute how much RAM this neural network will require (at least) when making a prediction for a single instance. First let's compute the feature map size for each layer. Since we are using a stride of 2 and SAME padding, the horizontal and vertical size of the feature maps are divided by 2 at each layer (rounding up if necessary), so as the input channels are 200 × 300 pixels, the first layer's feature maps are 100 × 150, the second layer's feature maps are 50 × 75, and the third layer's feature maps are 25 × 38. Since 32 bits is 4 bytes and the first convolutional layer has 100 feature maps, this first layer takes up 4 x 100 × 150 × 100 = 6 million bytes (about 5.7 MB, considering that 1 MB = 1,024 KB and 1 KB = 1,024 bytes). The second layer takes up 4 × 50 × 75 × 200 = 3 million bytes (about 2.9 MB). Finally, the third layer takes up 4 × 25 × 38 × 400 = 1,520,000

bytes (about 1.4 MB). However, once a layer has been computed, the memory occupied by the previous layer can be released, so if everything is well optimized, only 6 + 3 = 9 million bytes (about 8.6 MB) of RAM will be required (when the second layer has just been computed, but the memory occupied by the first layer is not released yet). But wait, you also need to add the memory occupied by the CNN's parameters. We computed earlier that it has 903,400 parameters, each using up 4 bytes, so this adds 3,613,600 bytes (about 3.4 MB). The total RAM required is (at least) 12,613,600 bytes (about 12.0 MB).

Lastly, let's compute the minimum amount of RAM required when training the CNN on a mini-batch of 50 images. During training TensorFlow uses backpropagation, which requires keeping all values computed during the forward pass until the reverse pass begins. So we must compute the total RAM required by all layers for a single instance and multiply that by 50! At that point let's start counting in megabytes rather than bytes. We computed before that the three layers require respectively 5.7, 2.9, and 1.4 MB for each instance. That's a total of 10.0 MB per instance. So for 50 instances the total RAM is 500 MB. Add to that the RAM required by the input images, which is $50 \times 4 \times 200 \times 300 \times 3 = 36$ million bytes (about 34.3 MB), plus the RAM required for the model parameters, which is about 3.4 MB (computed earlier), plus some RAM for the gradients (we will neglect them since they can be released gradually as backpropagation goes down the layers during the reverse pass). We are up to a total of roughly 500.0 + 34.3 + 3.4 = 537.7 MB. And that's really an optimistic bare minimum.

3. If your GPU runs out of memory while training a CNN, here are five things you could try to solve the problem (other than purchasing a GPU with more RAM):

   - Reduce the mini-batch size.
   - Reduce dimensionality using a larger stride in one or more layers.
   - Remove one or more layers.
   - Use 16-bit floats instead of 32-bit floats.
   - Distribute the CNN across multiple devices.

4. A max pooling layer has no parameters at all, whereas a convolutional layer has quite a few (see the previous questions).
5. A *local response normalization* layer makes the neurons that most strongly activate inhibit neurons at the same location but in neighboring feature maps, which encourages different feature maps to specialize and pushes them apart, forcing them to explore a wider range of features. It is typically used in the lower layers to have a larger pool of low-level features that the upper layers can build upon.
6. The main innovations in AlexNet compared to LeNet-5 are (1) it is much larger and deeper, and (2) it stacks convolutional layers directly on top of each other,

instead of stacking a pooling layer on top of each convolutional layer. The main innovation in GoogLeNet is the introduction of *inception modules*, which make it possible to have a much deeper net than previous CNN architectures, with fewer parameters. Finally, ResNet's main innovation is the introduction of skip connections, which make it possible to go well beyond 100 layers. Arguably, its simplicity and consistency are also rather innovative.

For the solutions to exercises 7, 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 14: Recurrent Neural Networks

1. Here are a few RNN applications:

   - For a sequence-to-sequence RNN: predicting the weather (or any other time series), machine translation (using an encoder–decoder architecture), video captioning, speech to text, music generation (or other sequence generation), identifying the chords of a song.
   - For a sequence-to-vector RNN: classifying music samples by music genre, analyzing the sentiment of a book review, predicting what word an aphasic patient is thinking of based on readings from brain implants, predicting the probability that a user will want to watch a movie based on her watch history (this is one of many possible implementations of *collaborative filtering*).
   - For a vector-to-sequence RNN: image captioning, creating a music playlist based on an embedding of the current artist, generating a melody based on a set of parameters, locating pedestrians in a picture (e.g., a video frame from a self-driving car's camera).

2. In general, if you translate a sentence one word at a time, the result will be terrible. For example, the French sentence "Je vous en prie" means "You are welcome," but if you translate it one word at a time, you get "I you in pray." Huh? It is much better to read the whole sentence first and then translate it. A plain sequence-to-sequence RNN would start translating a sentence immediately after reading the first word, while an encoder–decoder RNN will first read the whole sentence and then translate it. That said, one could imagine a plain sequence-to-sequence RNN that would output silence whenever it is unsure about what to say next (just like human translators do when they must translate a live broadcast).
3. To classify videos based on the visual content, one possible architecture could be to take (say) one frame per second, then run each frame through a convolutional neural network, feed the output of the CNN to a sequence-to-vector RNN, and finally run its output through a softmax layer, giving you all the class probabilities. For training you would just use cross entropy as the cost function. If you

wanted to use the audio for classification as well, you could convert every second of audio to a spectrograph, feed this spectrograph to a CNN, and feed the output of this CNN to the RNN (along with the corresponding output of the other CNN).

4. Building an RNN using `dynamic_rnn()` rather than `static_rnn()` offers several advantages:

   - It is based on a `while_loop()` operation that is able to swap the GPU's memory to the CPU's memory during backpropagation, avoiding out-of-memory errors.
   - It is arguably easier to use, as it can directly take a single tensor as input and output (covering all time steps), rather than a list of tensors (one per time step). No need to stack, unstack, or transpose.
   - It generates a smaller graph, easier to visualize in TensorBoard.

5. To handle variable length input sequences, the simplest option is to set the `sequence_length` parameter when calling the `static_rnn()` or `dynamic_rnn()` functions. Another option is to pad the smaller inputs (e.g., with zeros) to make them the same size as the largest input (this may be faster than the first option if the input sequences all have very similar lengths). To handle variable-length output sequences, if you know in advance the length of each output sequence, you can use the `sequence_length` parameter (for example, consider a sequence-to-sequence RNN that labels every frame in a video with a violence score: the output sequence will be exactly the same length as the input sequence). If you don't know in advance the length of the output sequence, you can use the padding trick: always output the same size sequence, but ignore any outputs that come after the end-of-sequence token (by ignoring them when computing the cost function).
6. To distribute training and execution of a deep RNN across multiple GPUs, a common technique is simply to place each layer on a different GPU (see Chapter 12).

For the solutions to exercises 7, 8, and 9, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 15: Autoencoders

1. Here are some of the main tasks that autoencoders are used for:

   - Feature extraction
   - Unsupervised pretraining

- Dimensionality reduction
- Generative models
- Anomaly detection (an autoencoder is generally bad at reconstructing outliers)

2. If you want to train a classifier and you have plenty of unlabeled training data, but only a few thousand labeled instances, then you could first train a deep autoencoder on the full dataset (labeled + unlabeled), then reuse its lower half for the classifier (i.e., reuse the layers up to the codings layer, included) and train the classifier using the labeled data. If you have little labeled data, you probably want to freeze the reused layers when training the classifier.
3. The fact that an autoencoder perfectly reconstructs its inputs does not necessarily mean that it is a good autoencoder; perhaps it is simply an overcomplete autoencoder that learned to copy its inputs to the codings layer and then to the outputs. In fact, even if the codings layer contained a single neuron, it would be possible for a very deep autoencoder to learn to map each training instance to a different coding (e.g., the first instance could be mapped to 0.001, the second to 0.002, the third to 0.003, and so on), and it could learn "by heart" to reconstruct the right training instance for each coding. It would perfectly reconstruct its inputs without really learning any useful pattern in the data. In practice such a mapping is unlikely to happen, but it illustrates the fact that perfect reconstructions are not a guarantee that the autoencoder learned anything useful. However, if it produces very bad reconstructions, then it is almost guaranteed to be a bad autoencoder. To evaluate the performance of an autoencoder, one option is to measure the reconstruction loss (e.g., compute the MSE, the mean square of the outputs minus the inputs). Again, a high reconstruction loss is a good sign that the autoencoder is bad, but a low reconstruction loss is not a guarantee that it is good. You should also evaluate the autoencoder according to what it will be used for. For example, if you are using it for unsupervised pretraining of a classifier, then you should also evaluate the classifier's performance.
4. An undercomplete autoencoder is one whose codings layer is smaller than the input and output layers. If it is larger, then it is an overcomplete autoencoder. The main risk of an excessively undercomplete autoencoder is that it may fail to reconstruct the inputs. The main risk of an overcomplete autoencoder is that it may just copy the inputs to the outputs, without learning any useful feature.
5. To tie the weights of an encoder layer and its corresponding decoder layer, you simply make the decoder weights equal to the transpose of the encoder weights. This reduces the number of parameters in the model by half, often making training converge faster with less training data, and reducing the risk of overfitting the training set.
6. To visualize the features learned by the lower layer of a stacked autoencoder, a common technique is simply to plot the weights of each neuron, by reshaping

each weight vector to the size of an input image (e.g., for MNIST, reshaping a weight vector of shape `[784]` to `[28, 28]`). To visualize the features learned by higher layers, one technique is to display the training instances that most activate each neuron.

7. A generative model is a model capable of randomly generating outputs that resemble the training instances. For example, once trained successfully on the MNIST dataset, a generative model can be used to randomly generate realistic images of digits. The output distribution is typically similar to the training data. For example, since MNIST contains many images of each digit, the generative model would output roughly the same number of images of each digit. Some generative models can be parametrized—for example, to generate only some kinds of outputs. An example of a generative autoencoder is the variational autoencoder.

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 16: Reinforcement Learning

1. Reinforcement Learning is an area of Machine Learning aimed at creating agents capable of taking actions in an environment in a way that maximizes rewards over time. There are many differences between RL and regular supervised and unsupervised learning. Here are a few:

   - In supervised and unsupervised learning, the goal is generally to find patterns in the data and use them to make predictions. In Reinforcement Learning, the goal is to find a good policy.
   - Unlike in supervised learning, the agent is not explicitly given the "right" answer. It must learn by trial and error.
   - Unlike in unsupervised learning, there is a form of supervision, through rewards. We do not tell the agent how to perform the task, but we do tell it when it is making progress or when it is failing.
   - A Reinforcement Learning agent needs to find the right balance between exploring the environment, looking for new ways of getting rewards, and exploiting sources of rewards that it already knows. In contrast, supervised and unsupervised learning systems generally don't need to worry about exploration; they just feed on the training data they are given.
   - In supervised and unsupervised learning, training instances are typically independent (in fact, they are generally shuffled). In Reinforcement Learning, consecutive observations are generally *not* independent. An agent may remain in the same region of the environment for a while before it moves on, so consecu-

tive observations will be very correlated. In some cases a replay memory is used to ensure that the training algorithm gets fairly independent observations.

2. Here are a few possible applications of Reinforcement Learning, other than those mentioned in Chapter 16:

   Music personalization
   : The environment is a user's personalized web radio. The agent is the software deciding what song to play next for that user. Its possible actions are to play any song in the catalog (it must try to choose a song the user will enjoy) or to play an advertisement (it must try to choose an ad that the user will be interested in). It gets a small reward every time the user listens to a song, a larger reward every time the user listens to an ad, a negative reward when the user skips a song or an ad, and a very negative reward if the user leaves.

   Marketing
   : The environment is your company's marketing department. The agent is the software that defines which customers a mailing campaign should be sent to, given their profile and purchase history (for each customer it has two possible actions: send or don't send). It gets a negative reward for the cost of the mailing campaign, and a positive reward for estimated revenue generated from this campaign.

   Product delivery
   : Let the agent control a fleet of delivery trucks, deciding what they should pick up at the depots, where they should go, what they should drop off, and so on. They would get positive rewards for each product delivered on time, and negative rewards for late deliveries.

3. When estimating the value of an action, Reinforcement Learning algorithms typically sum all the rewards that this action led to, giving more weight to immediate rewards, and less weight to later rewards (considering that an action has more influence on the near future than on the distant future). To model this, a discount rate is typically applied at each time step. For example, with a discount rate of 0.9, a reward of 100 that is received two time steps later is counted as only $0.9^2 \times 100 = 81$ when you are estimating the value of the action. You can think of the discount rate as a measure of how much the future is valued relative to the present: if it is very close to 1, then the future is valued almost as much as the present. If it is close to 0, then only immediate rewards matter. Of course, this impacts the optimal policy tremendously: if you value the future, you may be willing to put up with a lot of immediate pain for the prospect of eventual rewards, while if you don't value the future, you will just grab any immediate reward you can find, never investing in the future.

4. To measure the performance of a Reinforcement Learning agent, you can simply sum up the rewards it gets. In a simulated environment, you can run many episodes and look at the total rewards it gets on average (and possibly look at the min, max, standard deviation, and so on).
5. The credit assignment problem is the fact that when a Reinforcement Learning agent receives a reward, it has no direct way of knowing which of its previous actions contributed to this reward. It typically occurs when there is a large delay between an action and the resulting rewards (e.g., during a game of Atari's *Pong*, there may be a few dozen time steps between the moment the agent hits the ball and the moment it wins the point). One way to alleviate it is to provide the agent with shorter-term rewards, when possible. This usually requires prior knowledge about the task. For example, if we want to build an agent that will learn to play chess, instead of giving it a reward only when it wins the game, we could give it a reward every time it captures one of the opponent's pieces.
6. An agent can often remain in the same region of its environment for a while, so all of its experiences will be very similar for that period of time. This can introduce some bias in the learning algorithm. It may tune its policy for this region of the environment, but it will not perform well as soon as it moves out of this region. To solve this problem, you can use a replay memory; instead of using only the most immediate experiences for learning, the agent will learn based on a buffer of its past experiences, recent and not so recent (perhaps this is why we dream at night: to replay our experiences of the day and better learn from them?).
7. An off-policy RL algorithm learns the value of the optimal policy (i.e., the sum of discounted rewards that can be expected for each state if the agent acts optimally) while the agent follows a different policy. Q-Learning is a good example of such an algorithm. In contrast, an on-policy algorithm learns the value of the policy that the agent actually executes, including both exploration and exploitation.

For the solutions to exercises 8, 9, and 10, please see the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.