this case all 1,000 dimensions are required to preserve 95% of the variance. So the answer is, it depends on the dataset, and it could be any number between 1 and 1,000. Plotting the explained variance as a function of the number of dimensions is one way to get a rough idea of the dataset's intrinsic dimensionality.

6. Regular PCA is the default, but it works only if the dataset fits in memory. Incremental PCA is useful for large datasets that don't fit in memory, but it is slower than regular PCA, so if the dataset fits in memory you should prefer regular PCA. Incremental PCA is also useful for online tasks, when you need to apply PCA on the fly, every time a new instance arrives. Randomized PCA is useful when you want to considerably reduce dimensionality and the dataset fits in memory; in this case, it is much faster than regular PCA. Finally, Kernel PCA is useful for nonlinear datasets.
7. Intuitively, a dimensionality reduction algorithm performs well if it eliminates a lot of dimensions from the dataset without losing too much information. One way to measure this is to apply the reverse transformation and measure the reconstruction error. However, not all dimensionality reduction algorithms provide a reverse transformation. Alternatively, if you are using dimensionality reduction as a preprocessing step before another Machine Learning algorithm (e.g., a Random Forest classifier), then you can simply measure the performance of that second algorithm; if dimensionality reduction did not lose too much information, then the algorithm should perform just as well as when using the original dataset.
8. It can absolutely make sense to chain two different dimensionality reduction algorithms. A common example is using PCA to quickly get rid of a large number of useless dimensions, then applying another much slower dimensionality reduction algorithm, such as LLE. This two-step approach will likely yield the same performance as using LLE only, but in a fraction of the time.

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 cluster, 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.

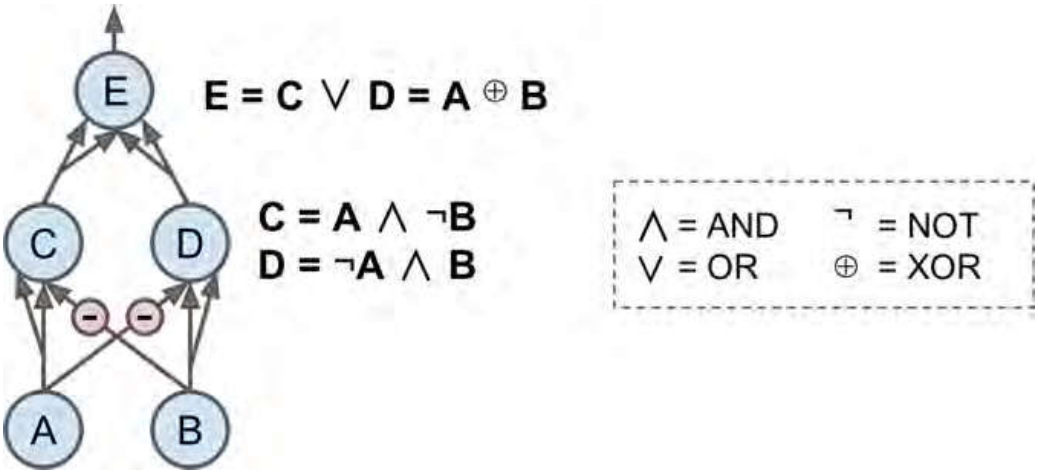
```
import tensorflow as tf

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.