CHAPTER 16
Reinforcement Learning

Reinforcement Learning (RL) is one of the most exciting fields of Machine Learning today, and also one of the oldest. It has been around since the 1950s, producing many interesting applications over the years,[1] in particular in games (e.g., *TD-Gammon*, a *Backgammon* playing program) and in machine control, but seldom making the headline news. But a revolution took place in 2013 when researchers from an English startup called DeepMind demonstrated a system that could learn to play just about any Atari game from scratch,[2] eventually outperforming humans[3] in most of them, using only raw pixels as inputs and without any prior knowledge of the rules of the games.[4] This was the first of a series of amazing feats, culminating in March 2016 with the victory of their system AlphaGo against Lee Sedol, the world champion of the game of *Go*. No program had ever come close to beating a master of this game, let alone the world champion. Today the whole field of RL is boiling with new ideas, with a wide range of applications. DeepMind was bought by Google for over 500 million dollars in 2014.

So how did they do it? With hindsight it seems rather simple: they applied the power of Deep Learning to the field of Reinforcement Learning, and it worked beyond their wildest dreams. In this chapter we will first explain what Reinforcement Learning is and what it is good at, and then we will present two of the most important techniques in deep Reinforcement Learning: *policy gradients* and *deep Q-networks* (DQN),

1 For more details, be sure to check out Richard Sutton and Andrew Barto's book on RL, *Reinforcement Learning: An Introduction* (MIT Press), or David Silver's free online RL course at University College London.

2 "Playing Atari with Deep Reinforcement Learning," V. Mnih et al. (2013).

3 "Human-level control through deep reinforcement learning," V. Mnih et al. (2015).

4 Check out the videos of DeepMind's system learning to play *Space Invaders*, *Breakout*, and more at *https://goo.gl/yTsH6X*.

including a discussion of *Markov decision processes* (MDP). We will use these techniques to train a model to balance a pole on a moving cart, and another to play Atari games. The same techniques can be used for a wide variety of tasks, from walking robots to self-driving cars.

Learning to Optimize Rewards

In Reinforcement Learning, a software *agent* makes *observations* and takes *actions* within an *environment*, and in return it receives *rewards*. Its objective is to learn to act in a way that will maximize its expected long-term rewards. If you don't mind a bit of anthropomorphism, you can think of positive rewards as pleasure, and negative rewards as pain (the term "reward" is a bit misleading in this case). In short, the agent acts in the environment and learns by trial and error to maximize its pleasure and minimize its pain.

This is quite a broad setting, which can apply to a wide variety of tasks. Here are a few examples (see Figure 16-1):

a. The agent can be the program controlling a walking robot. In this case, the environment is the real world, the agent observes the environment through a set of *sensors* such as cameras and touch sensors, and its actions consist of sending signals to activate motors. It may be programmed to get positive rewards whenever it approaches the target destination, and negative rewards whenever it wastes time, goes in the wrong direction, or falls down.
b. The agent can be the program controlling Ms. Pac-Man. In this case, the environment is a simulation of the Atari game, the actions are the nine possible joystick positions (upper left, down, center, and so on), the observations are screenshots, and the rewards are just the game points.
c. Similarly, the agent can be the program playing a board game such as the game of *Go*.
d. The agent does not have to control a physically (or virtually) moving thing. For example, it can be a smart thermostat, getting rewards whenever it is close to the target temperature and saves energy, and negative rewards when humans need to tweak the temperature, so the agent must learn to anticipate human needs.
e. The agent can observe stock market prices and decide how much to buy or sell every second. Rewards are obviously the monetary gains and losses.

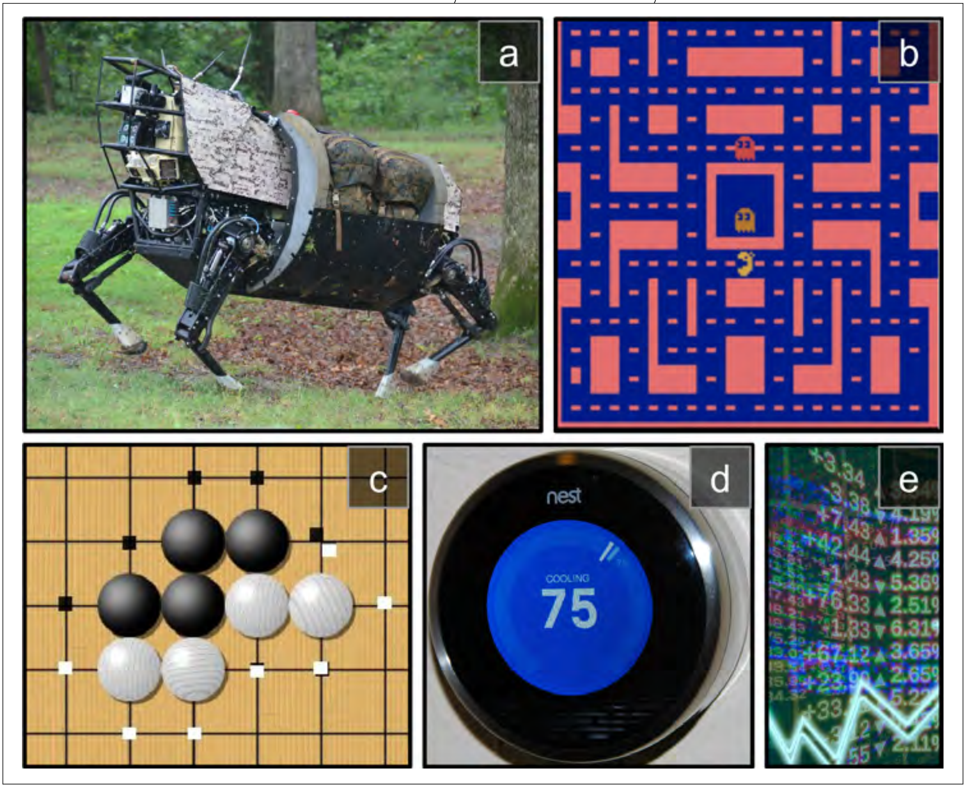


Figure 16-1. Reinforcement Learning examples: (a) walking robot, (b) Ms. Pac-Man, (c) Go player, (d) thermostat, (e) automatic trader[5]

Note that there may not be any positive rewards at all; for example, the agent may move around in a maze, getting a negative reward at every time step, so it better find the exit as quickly as possible! There are many other examples of tasks where Reinforcement Learning is well suited, such as self-driving cars, placing ads on a web page, or controlling where an image classification system should focus its attention.

5 Images (a), (c), and (d) are reproduced from Wikipedia. (a) and (d) are in the public domain. (c) was created by user Stevertigo and released under Creative Commons BY-SA 2.0. (b) is a screenshot from the Ms. Pac-Man game, copyright Atari (the author believes it to be fair use in this chapter). (e) was reproduced from Pixabay, released under Creative Commons CC0.

Policy Search

The algorithm used by the software agent to determine its actions is called its *policy*. For example, the policy could be a neural network taking observations as inputs and outputting the action to take (see Figure 16-2).

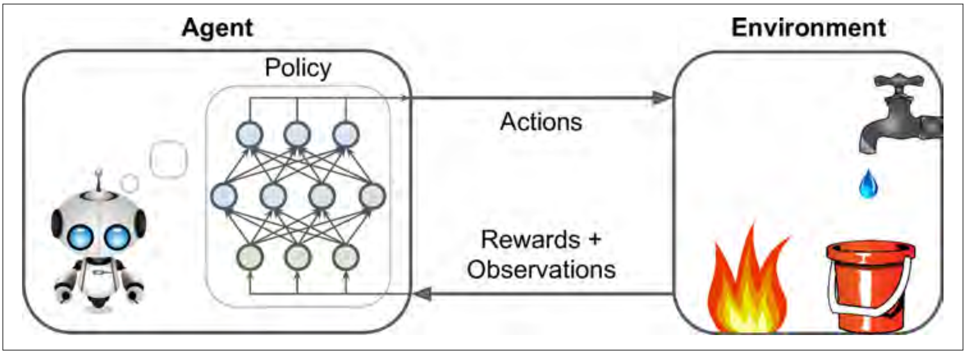


Figure 16-2. Reinforcement Learning using a neural network policy

The policy can be any algorithm you can think of, and it does not even have to be deterministic. For example, consider a robotic vacuum cleaner whose reward is the amount of dust it picks up in 30 minutes. Its policy could be to move forward with some probability *p* every second, or randomly rotate left or right with probability 1 – *p*. The rotation angle would be a random angle between –r and +r. Since this policy involves some randomness, it is called a *stochastic policy*. The robot will have an erratic trajectory, which guarantees that it will eventually get to any place it can reach and pick up all the dust. The question is: how much dust will it pick up in 30 minutes?

How would you train such a robot? There are just two *policy parameters* you can tweak: the probability *p* and the angle range *r*. One possible learning algorithm could be to try out many different values for these parameters, and pick the combination that performs best (see Figure 16-3). This is an example of *policy search*, in this case using a brute force approach. However, when the *policy space* is too large (which is generally the case), finding a good set of parameters this way is like searching for a needle in a gigantic haystack.

Another way to explore the policy space is to use *genetic algorithms*. For example, you could randomly create a first generation of 100 policies and try them out, then "kill" the 80 worst policies[6] and make the 20 survivors produce 4 offspring each. An off-

6 It is often better to give the poor performers a slight chance of survival, to preserve some diversity in the "gene pool."

spring is just a copy of its parent[7] plus some random variation. The surviving policies plus their offspring together constitute the second generation. You can continue to iterate through generations this way, until you find a good policy.

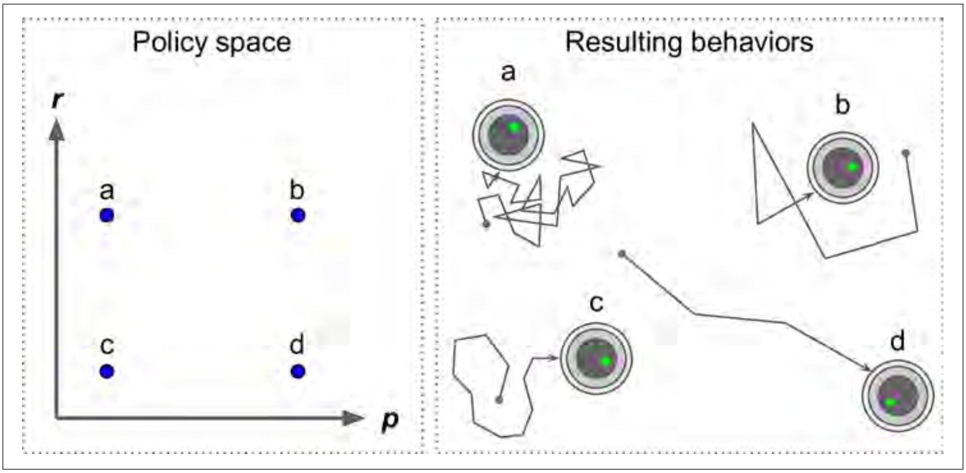


Figure 16-3. Four points in policy space and the agent's corresponding behavior

Yet another approach is to use optimization techniques, by evaluating the gradients of the rewards with regards to the policy parameters, then tweaking these parameters by following the gradient toward higher rewards (*gradient ascent*). This approach is called *policy gradients* (PG), which we will discuss in more detail later in this chapter. For example, going back to the vacuum cleaner robot, you could slightly increase *p* and evaluate whether this increases the amount of dust picked up by the robot in 30 minutes; if it does, then increase *p* some more, or else reduce *p*. We will implement a popular PG algorithm using TensorFlow, but before we do we need to create an environment for the agent to live in, so it's time to introduce OpenAI gym.

Introduction to OpenAI Gym

One of the challenges of Reinforcement Learning is that in order to train an agent, you first need to have a working environment. If you want to program an agent that will learn to play an Atari game, you will need an Atari game simulator. If you want to program a walking robot, then the environment is the real world and you can directly train your robot in that environment, but this has its limits: if the robot falls off a cliff, you can't just click "undo." You can't speed up time either; adding more computing

7 If there is a single parent, this is called *asexual reproduction*. With two (or more) parents, it is called *sexual reproduction*. An offspring's genome (in this case a set of policy parameters) is randomly composed of parts of its parents' genomes.

power won't make the robot move any faster. And it's generally too expensive to train 1,000 robots in parallel. In short, training is hard and slow in the real world, so you generally need a *simulated environment* at least to bootstrap training.

OpenAI gym[8] is a toolkit that provides a wide variety of simulated environments (Atari games, board games, 2D and 3D physical simulations, and so on), so you can train agents, compare them, or develop new RL algorithms.

Let's install OpenAI gym. For a minimal OpenAI gym installation, simply use pip:

```
$ pip3 install --upgrade gym
```

Next open up a Python shell or a Jupyter notebook and create your first environment:

```
>>> import gym
>>> env = gym.make("CartPole-v0")
[2016-10-14 16:03:23,199] Making new env: MsPacman-v0
>>> obs = env.reset()
>>> obs
array([-0.03799846, -0.03288115,  0.02337094,  0.00720711])
>>> env.render()
```

The `make()` function creates an environment, in this case a CartPole environment. This is a 2D simulation in which a cart can be accelerated left or right in order to balance a pole placed on top of it (see Figure 16-4). After the environment is created, we must initialize it using the `reset()` method. This returns the first observation. Observations depend on the type of environment. For the CartPole environment, each observation is a 1D NumPy array containing four floats: these floats represent the cart's horizontal position (`0.0` = center), its velocity, the angle of the pole (`0.0` = vertical), and its angular velocity. Finally, the `render()` method displays the environment as shown in Figure 16-4.

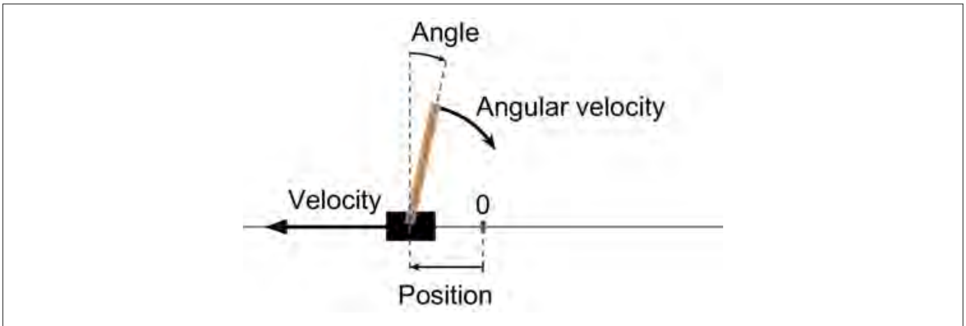


Figure 16-4. The CartPole environment

8 OpenAI is a nonprofit artificial intelligence research company, funded in part by Elon Musk. Its stated goal is to promote and develop friendly AIs that will benefit humanity (rather than exterminate it).

If you want `render()` to return the rendered image as a NumPy array, you can set the `mode` parameter to `rgb_array` (note that other environments may support different modes):

```
>>> img = env.render(mode="rgb_array")
>>> img.shape  # height, width, channels (3=RGB)
(400, 600, 3)
```

Unfortunately, the CartPole (and a few other environments) renders the image to the screen even if you set the mode to `"rgb_array"`. The only way to avoid this is to use a fake X server such as Xvfb or Xdummy. For example, you can install Xvfb and start Python using the following command: `xvfb-run -s "-screen 0 1400x900x24" python`. Or use the xvfbwrapper package.

Let's ask the environment what actions are possible:

```
>>> env.action_space
Discrete(2)
```

`Discrete(2)` means that the possible actions are integers 0 and 1, which represent accelerating left (0) or right (1). Other environments may have more discrete actions, or other kinds of actions (e.g., continuous). Since the pole is leaning toward the right, let's accelerate the cart toward the right:

```
>>> action = 1  # accelerate right
>>> obs, reward, done, info = env.step(action)
>>> obs
array([-0.03865608,  0.16189797,  0.02351508, -0.27801135])
>>> reward
1.0
>>> done
False
>>> info
{}
```

The `step()` method executes the given action and returns four values:

`obs`
: This is the new observation. The cart is now moving toward the right (`obs[1]>0`). The pole is still tilted toward the right (`obs[2]>0`), but its angular velocity is now negative (`obs[3]<0`), so it will likely be tilted toward the left after the next step.

`reward`
: In this environment, you get a reward of 1.0 at every step, no matter what you do, so the goal is to keep running as long as possible.

`done`
: This value will be `True` when the *episode* is over. This will happen when the pole tilts too much. After that, the environment must be reset before it can be used again.

`info`
: This dictionary may provide extra debug information in other environments. This data should not be used for training (it would be cheating).

Let's hardcode a simple policy that accelerates left when the pole is leaning toward the left and accelerates right when the pole is leaning toward the right. We will run this policy to see the average rewards it gets over 500 episodes:

```
def basic_policy(obs):
    angle = obs[2]
    return 0 if angle < 0 else 1

totals = []
for episode in range(500):
    episode_rewards = 0
    obs = env.reset()
    for step in range(1000): # 1000 steps max, we don't want to run forever
        action = basic_policy(obs)
        obs, reward, done, info = env.step(action)
        episode_rewards += reward
        if done:
            break
    totals.append(episode_rewards)
```

This code is hopefully self-explanatory. Let's look at the result:

```
>>> import numpy as np
>>> np.mean(totals), np.std(totals), np.min(totals), np.max(totals)
(42.125999999999998, 9.1237121830974033, 24.0, 68.0)
```

Even with 500 tries, this policy never managed to keep the pole upright for more than 68 consecutive steps. Not great. If you look at the simulation in the Jupyter notebooks, you will see that the cart oscillates left and right more and more strongly until the pole tilts too much. Let's see if a neural network can come up with a better policy.

Neural Network Policies

Let's create a neural network policy. Just like the policy we hardcoded earlier, this neural network will take an observation as input, and it will output the action to be executed. More precisely, it will estimate a probability for each action, and then we will select an action randomly according to the estimated probabilities (see Figure 16-5). In the case of the CartPole environment, there are just two possible actions (left or right), so we only need one output neuron. It will output the probability p of action 0 (left), and of course the probability of action 1 (right) will be $1 - p$.

For example, if it outputs 0.7, then we will pick action 0 with 70% probability, and action 1 with 30% probability.

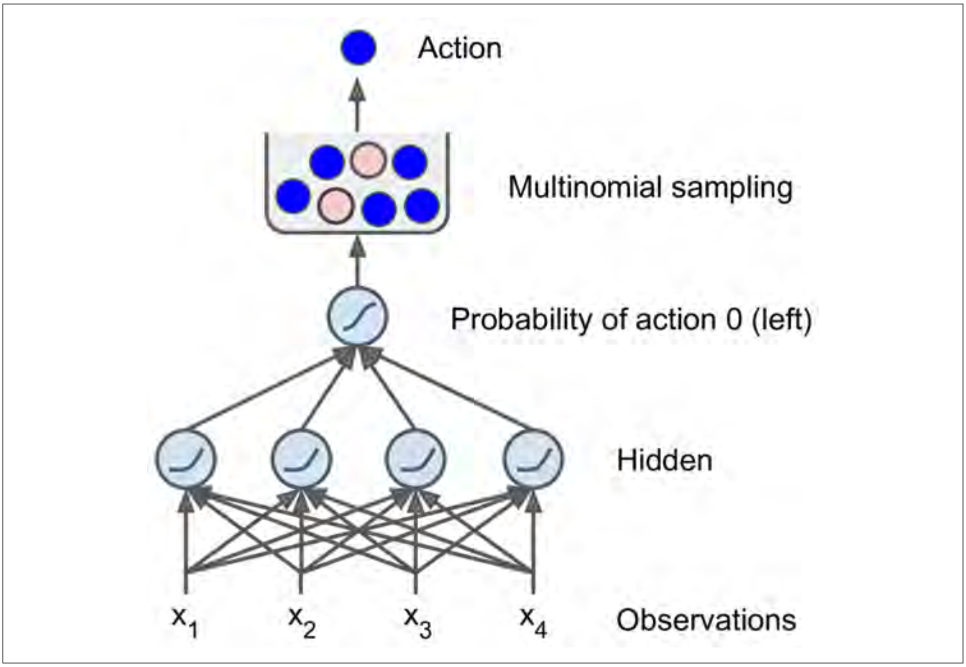


Figure 16-5. Neural network policy

You may wonder why we are picking a random action based on the probability given by the neural network, rather than just picking the action with the highest score. This approach lets the agent find the right balance between *exploring* new actions and *exploiting* the actions that are known to work well. Here's an analogy: suppose you go to a restaurant for the first time, and all the dishes look equally appealing so you randomly pick one. If it turns out to be good, you can increase the probability to order it next time, but you shouldn't increase that probability up to 100%, or else you will never try out the other dishes, some of which may be even better than the one you tried.

Also note that in this particular environment, the past actions and observations can safely be ignored, since each observation contains the environment's full state. If there were some hidden state, then you may need to consider past actions and observations as well. For example, if the environment only revealed the position of the cart but not its velocity, you would have to consider not only the current observation but also the previous observation in order to estimate the current velocity. Another example is when the observations are noisy; in that case, you generally want to use the past few observations to estimate the most likely current state. The CartPole problem is thus as simple as can be; the observations are noise-free and they contain the environment's full state.

Here is the code to build this neural network policy using TensorFlow:

```
import tensorflow as tf
from tensorflow.contrib.layers import fully_connected

# 1. Specify the neural network architecture
n_inputs = 4  # == env.observation_space.shape[0]
n_hidden = 4  # it's a simple task, we don't need more hidden neurons
n_outputs = 1 # only outputs the probability of accelerating left
initializer = tf.contrib.layers.variance_scaling_initializer()

# 2. Build the neural network
X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,
                         weights_initializer=initializer)
logits = fully_connected(hidden, n_outputs, activation_fn=None,
                         weights_initializer=initializer)
outputs = tf.nn.sigmoid(logits)

# 3. Select a random action based on the estimated probabilities
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

init = tf.global_variables_initializer()
```

Let's go through this code:

1. After the imports, we define the neural network architecture. The number of inputs is the size of the observation space (which in the case of the CartPole is four), we just have four hidden units and no need for more, and we have just one output probability (the probability of going left).
2. Next we build the neural network. In this example, it's a vanilla Multi-Layer Perceptron, with a single output. Note that the output layer uses the logistic (sigmoid) activation function in order to output a probability from 0.0 to 1.0. If there were more than two possible actions, there would be one output neuron per action, and you would use the softmax activation function instead.
3. Lastly, we call the `multinomial()` function to pick a random action. This function independently samples one (or more) integers, given the log probability of each integer. For example, if you call it with the array `[np.log(0.5), np.log(0.2), np.log(0.3)]` and with `num_samples=5`, then it will output five integers, each of which will have a 50% probability of being 0, 20% of being 1, and 30% of being 2. In our case we just need one integer representing the action to take. Since the `outputs` tensor only contains the probability of going left, we must first concatenate `1-outputs` to it to have a tensor containing the probability

of both left and right actions. Note that if there were more than two possible actions, the neural network would have to output one probability per action so you would not need the concatenation step.

Okay, we now have a neural network policy that will take observations and output actions. But how do we train it?

Evaluating Actions: The Credit Assignment Problem

If we knew what the best action was at each step, we could train the neural network as usual, by minimizing the cross entropy between the estimated probability and the target probability. It would just be regular supervised learning. However, in Reinforcement Learning the only guidance the agent gets is through rewards, and rewards are typically sparse and delayed. For example, if the agent manages to balance the pole for 100 steps, how can it know which of the 100 actions it took were good, and which of them were bad? All it knows is that the pole fell after the last action, but surely this last action is not entirely responsible. This is called the *credit assignment problem*: when the agent gets a reward, it is hard for it to know which actions should get credited (or blamed) for it. Think of a dog that gets rewarded hours after it behaved well; will it understand what it is rewarded for?

To tackle this problem, a common strategy is to evaluate an action based on the sum of all the rewards that come after it, usually applying a *discount rate* r at each step. For example (see Figure 16-6), if an agent decides to go right three times in a row and gets +10 reward after the first step, 0 after the second step, and finally –50 after the third step, then assuming we use a discount rate $r = 0.8$, the first action will have a total score of $10 + r \times 0 + r^2 \times (-50) = -22$. If the discount rate is close to 0, then future rewards won't count for much compared to immediate rewards. Conversely, if the discount rate is close to 1, then rewards far into the future will count almost as much as immediate rewards. Typical discount rates are 0.95 or 0.99. With a discount rate of 0.95, rewards 13 steps into the future count roughly for half as much as immediate rewards (since $0.95^{13} \approx 0.5$), while with a discount rate of 0.99, rewards 69 steps into the future count for half as much as immediate rewards. In the CartPole environment, actions have fairly short-term effects, so choosing a discount rate of 0.95 seems reasonable.

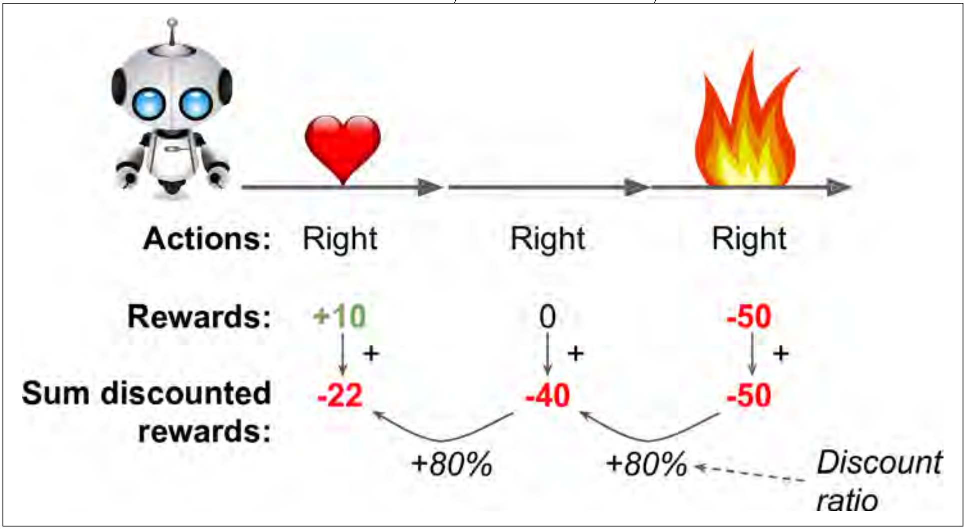


Figure 16-6. Discounted rewards

Of course, a good action may be followed by several bad actions that cause the pole to fall quickly, resulting in the good action getting a low score (similarly, a good actor may sometimes star in a terrible movie). However, if we play the game enough times, on average good actions will get a better score than bad ones. So, to get fairly reliable action scores, we must run many episodes and normalize all the action scores (by subtracting the mean and dividing by the standard deviation). After that, we can reasonably assume that actions with a negative score were bad while actions with a positive score were good. Perfect—now that we have a way to evaluate each action, we are ready to train our first agent using policy gradients. Let's see how.

Policy Gradients

As discussed earlier, PG algorithms optimize the parameters of a policy by following the gradients toward higher rewards. One popular class of PG algorithms, called *REINFORCE algorithms*, was introduced back in 1992[9] by Ronald Williams. Here is one common variant:

1. First, let the neural network policy play the game several times and at each step compute the gradients that would make the chosen action even more likely, but don't apply these gradients yet.

9 "Simple Statistical Gradient-Following Algorithms for Connectionist Reinforcement Learning," R. Williams (1992).

2. Once you have run several episodes, compute each action's score (using the method described in the previous paragraph).
3. If an action's score is positive, it means that the action was good and you want to apply the gradients computed earlier to make the action even more likely to be chosen in the future. However, if the score is negative, it means the action was bad and you want to apply the opposite gradients to make this action slightly *less* likely in the future. The solution is simply to multiply each gradient vector by the corresponding action's score.
4. Finally, compute the mean of all the resulting gradient vectors, and use it to perform a Gradient Descent step.

Let's implement this algorithm using TensorFlow. We will train the neural network policy we built earlier so that it learns to balance the pole on the cart. Let's start by completing the construction phase we coded earlier to add the target probability, the cost function, and the training operation. Since we are acting as though the chosen action is the best possible action, the target probability must be 1.0 if the chosen action is action 0 (left) and 0.0 if it is action 1 (right):

```
y = 1. - tf.to_float(action)
```

Now that we have a target probability, we can define the cost function (cross entropy) and compute the gradients:

```
learning_rate = 0.01

cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
                    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
```

Note that we are calling the optimizer's `compute_gradients()` method instead of the `minimize()` method. This is because we want to tweak the gradients before we apply them.[10] The `compute_gradients()` method returns a list of gradient vector/variable pairs (one pair per trainable variable). Let's put all the gradients in a list, to make it more convenient to obtain their values:

```
gradients = [grad for grad, variable in grads_and_vars]
```

Okay, now comes the tricky part. During the execution phase, the algorithm will run the policy and at each step it will evaluate these gradient tensors and store their values. After a number of episodes it will tweak these gradients as explained earlier (i.e., multiply them by the action scores and normalize them) and compute the mean of the tweaked gradients. Next, it will need to feed the resulting gradients back to the

10 We already did something similar in Chapter 11 when we discussed Gradient Clipping: we first computed the gradients, then we clipped them, and finally we applied the clipped gradients.

optimizer so that it can perform an optimization step. This means we need one placeholder per gradient vector. Moreover, we must create the operation that will apply the updated gradients. For this we will call the optimizer's `apply_gradients()` function, which takes a list of gradient vector/variable pairs. Instead of giving it the original gradient vectors, we will give it a list containing the updated gradients (i.e., the ones fed through the gradient placeholders):

```
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))

training_op = optimizer.apply_gradients(grads_and_vars_feed)
```

Let's step back and take a look at the full construction phase:

```
n_inputs = 4
n_hidden = 4
n_outputs = 1
initializer = tf.contrib.layers.variance_scaling_initializer()

learning_rate = 0.01

X = tf.placeholder(tf.float32, shape=[None, n_inputs])
hidden = fully_connected(X, n_hidden, activation_fn=tf.nn.elu,
                         weights_initializer=initializer)
logits = fully_connected(hidden, n_outputs, activation_fn=None,
                         weights_initializer=initializer)
outputs = tf.nn.sigmoid(logits)
p_left_and_right = tf.concat(axis=1, values=[outputs, 1 - outputs])
action = tf.multinomial(tf.log(p_left_and_right), num_samples=1)

y = 1. - tf.to_float(action)
cross_entropy = tf.nn.sigmoid_cross_entropy_with_logits(
                    labels=y, logits=logits)
optimizer = tf.train.AdamOptimizer(learning_rate)
grads_and_vars = optimizer.compute_gradients(cross_entropy)
gradients = [grad for grad, variable in grads_and_vars]
gradient_placeholders = []
grads_and_vars_feed = []
for grad, variable in grads_and_vars:
    gradient_placeholder = tf.placeholder(tf.float32, shape=grad.get_shape())
    gradient_placeholders.append(gradient_placeholder)
    grads_and_vars_feed.append((gradient_placeholder, variable))
training_op = optimizer.apply_gradients(grads_and_vars_feed)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

On to the execution phase! We will need a couple of functions to compute the total discounted rewards, given the raw rewards, and to normalize the results across multiple episodes:

```
def discount_rewards(rewards, discount_rate):
    discounted_rewards = np.empty(len(rewards))
    cumulative_rewards = 0
    for step in reversed(range(len(rewards))):
        cumulative_rewards = rewards[step] + cumulative_rewards * discount_rate
        discounted_rewards[step] = cumulative_rewards
    return discounted_rewards

def discount_and_normalize_rewards(all_rewards, discount_rate):
    all_discounted_rewards = [discount_rewards(rewards)
                              for rewards in all_rewards]
    flat_rewards = np.concatenate(all_discounted_rewards)
    reward_mean = flat_rewards.mean()
    reward_std = flat_rewards.std()
    return [(discounted_rewards - reward_mean)/reward_std
            for discounted_rewards in all_discounted_rewards]
```

Let's check that this works:

```
>>> discount_rewards([10, 0, -50], discount_rate=0.8)
array([-22., -40., -50.])
>>> discount_and_normalize_rewards([[10, 0, -50], [10, 20]], discount_rate=0.8)
[array([-0.28435071, -0.86597718, -1.18910299]),
 array([ 1.26665318,  1.0727777 ])]
```

The call to `discount_rewards()` returns exactly what we expect (see Figure 16-6). You can verify that the function `discount_and_normalize_rewards()` does indeed return the normalized scores for each action in both episodes. Notice that the first episode was much worse than the second, so its normalized scores are all negative; all actions from the first episode would be considered bad, and conversely all actions from the second episode would be considered good.

We now have all we need to train the policy:

```
n_iterations = 250      # number of training iterations
n_max_steps = 1000      # max steps per episode
n_games_per_update = 10 # train the policy every 10 episodes
save_iterations = 10    # save the model every 10 training iterations
discount_rate = 0.95

with tf.Session() as sess:
    init.run()
    for iteration in range(n_iterations):
        all_rewards = []    # all sequences of raw rewards for each episode
        all_gradients = []  # gradients saved at each step of each episode
        for game in range(n_games_per_update):
            current_rewards = []   # all raw rewards from the current episode
            current_gradients = [] # all gradients from the current episode
```

```
        obs = env.reset()
        for step in range(n_max_steps):
            action_val, gradients_val = sess.run(
                    [action, gradients],
                    feed_dict={X: obs.reshape(1, n_inputs)}) # one obs
            obs, reward, done, info = env.step(action_val[0][0])
            current_rewards.append(reward)
            current_gradients.append(gradients_val)
            if done:
                break
        all_rewards.append(current_rewards)
        all_gradients.append(current_gradients)

    # At this point we have run the policy for 10 episodes, and we are
    # ready for a policy update using the algorithm described earlier.
    all_rewards = discount_and_normalize_rewards(all_rewards)
    feed_dict = {}
    for var_index, grad_placeholder in enumerate(gradient_placeholders):
        # multiply the gradients by the action scores, and compute the mean
        mean_gradients = np.mean(
            [reward * all_gradients[game_index][step][var_index]
                for game_index, rewards in enumerate(all_rewards)
                for step, reward in enumerate(rewards)],
            axis=0)
        feed_dict[grad_placeholder] = mean_gradients
    sess.run(training_op, feed_dict=feed_dict)
    if iteration % save_iterations == 0:
        saver.save(sess, "./my_policy_net_pg.ckpt")
```

Each training iteration starts by running the policy for 10 episodes (with maximum 1,000 steps per episode, to avoid running forever). At each step, we also compute the gradients, pretending that the chosen action was the best. After these 10 episodes have been run, we compute the action scores using the `discount_and_normalize_rewards()` function; we go through each trainable variable, across all episodes and all steps, to multiply each gradient vector by its corresponding action score; and we compute the mean of the resulting gradients. Finally, we run the training operation, feeding it these mean gradients (one per trainable variable). We also save the model every 10 training operations.

And we're done! This code will train the neural network policy, and it will successfully learn to balance the pole on the cart (you can try it out in the Jupyter notebooks). Note that there are actually two ways the agent can lose the game: either the pole can tilt too much, or the cart can go completely off the screen. With 250 training iterations, the policy learns to balance the pole quite well, but it is not yet good enough at avoiding going off the screen. A few hundred more training iterations will fix that.

Researchers try to find algorithms that work well even when the agent initially knows nothing about the environment. However, unless you are writing a paper, you should inject as much prior knowledge as possible into the agent, as it will speed up training dramatically. For example, you could add negative rewards proportional to the distance from the center of the screen, and to the pole's angle. Also, if you already have a reasonably good policy (e.g., hardcoded), you may want to train the neural network to imitate it before using policy gradients to improve it.

Despite its relative simplicity, this algorithm is quite powerful. You can use it to tackle much harder problems than balancing a pole on a cart. In fact, AlphaGo was based on a similar PG algorithm (plus *Monte Carlo Tree Search*, which is beyond the scope of this book).

We will now look at another popular family of algorithms. Whereas PG algorithms directly try to optimize the policy to increase rewards, the algorithms we will look at now are less direct: the agent learns to estimate the expected sum of discounted future rewards for each state, or the expected sum of discounted future rewards for each action in each state, then uses this knowledge to decide how to act. To understand these algorithms, we must first introduce *Markov decision processes* (MDP).

Markov Decision Processes

In the early 20th century, the mathematician Andrey Markov studied stochastic processes with no memory, called *Markov chains*. Such a process has a fixed number of states, and it randomly evolves from one state to another at each step. The probability for it to evolve from a state s to a state s' is fixed, and it depends only on the pair (s,s'), not on past states (the system has no memory).

Figure 16-7 shows an example of a Markov chain with four states. Suppose that the process starts in state s_0, and there is a 70% chance that it will remain in that state at the next step. Eventually it is bound to leave that state and never come back since no other state points back to s_0. If it goes to state s_1, it will then most likely go to state s_2 (90% probability), then immediately back to state s_1 (with 100% probability). It may alternate a number of times between these two states, but eventually it will fall into state s_3 and remain there forever (this is a *terminal state*). Markov chains can have very different dynamics, and they are heavily used in thermodynamics, chemistry, statistics, and much more.

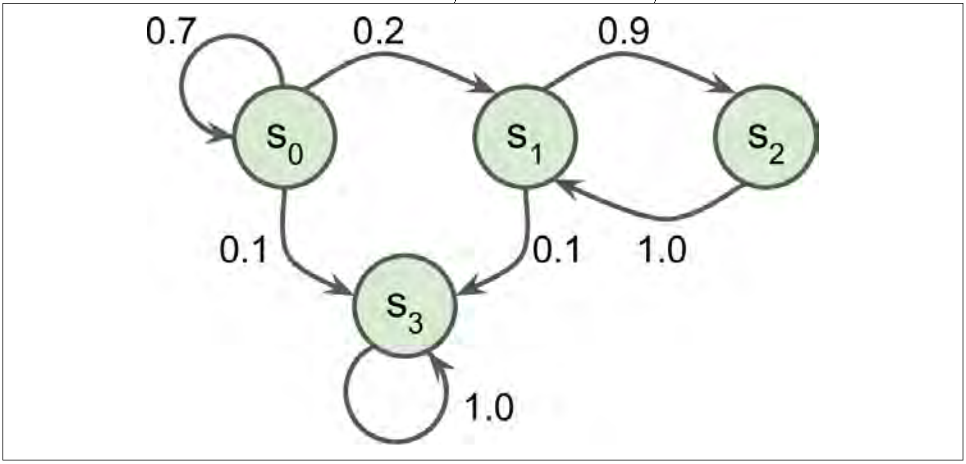


Figure 16-7. Example of a Markov chain

Markov decision processes were first described in the 1950s by Richard Bellman.[11] They resemble Markov chains but with a twist: at each step, an agent can choose one of several possible actions, and the transition probabilities depend on the chosen action. Moreover, some state transitions return some reward (positive or negative), and the agent's goal is to find a policy that will maximize rewards over time.

For example, the MDP represented in Figure 16-8 has three states and up to three possible discrete actions at each step. If it starts in state s_0, the agent can choose between actions a_0, a_1, or a_2. If it chooses action a_1, it just remains in state s_0 with certainty, and without any reward. It can thus decide to stay there forever if it wants. But if it chooses action a_0, it has a 70% probability of gaining a reward of +10, and remaining in state s_0. It can then try again and again to gain as much reward as possible. But at one point it is going to end up instead in state s_1. In state s_1 it has only two possible actions: a_0 or a_1. It can choose to stay put by repeatedly choosing action a_1, or it can choose to move on to state s_2 and get a negative reward of –50 (ouch). In state s_3 it has no other choice than to take action a_1, which will most likely lead it back to state s_0, gaining a reward of +40 on the way. You get the picture. By looking at this MDP, can you guess which strategy will gain the most reward over time? In state s_0 it is clear that action a_0 is the best option, and in state s_3 the agent has no choice but to take action a_1, but in state s_1 it is not obvious whether the agent should stay put (a_0) or go through the fire (a_2).

11 "A Markovian Decision Process," R. Bellman (1957).

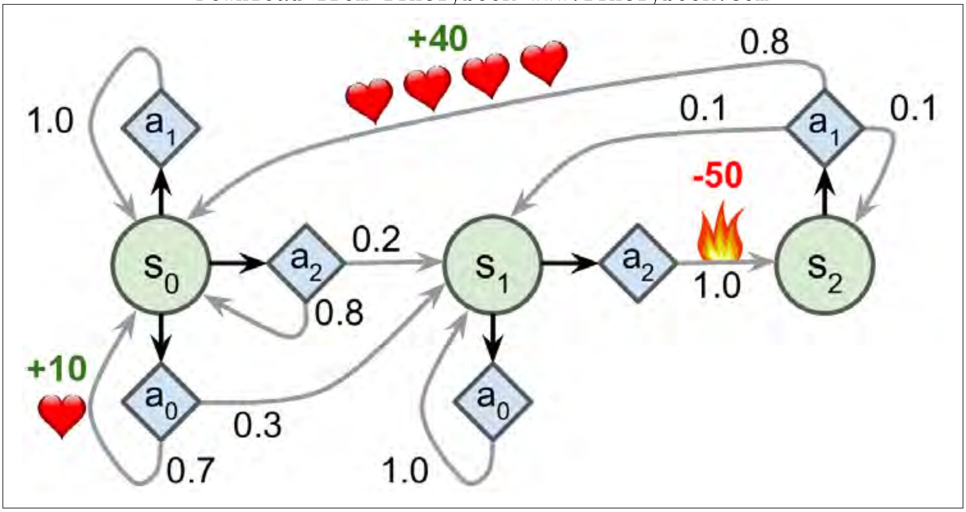


Figure 16-8. Example of a Markov decision process

Bellman found a way to estimate the *optimal state value* of any state *s*, noted $V^*(s)$, which is the sum of all discounted future rewards the agent can expect on average after it reaches a state *s*, assuming it acts optimally. He showed that if the agent acts optimally, then the *Bellman Optimality Equation* applies (see Equation 16-1). This recursive equation says that if the agent acts optimally, then the optimal value of the current state is equal to the reward it will get on average after taking one optimal action, plus the expected optimal value of all possible next states that this action can lead to.

Equation 16-1. Bellman Optimality Equation

$$V^*(s) = \max_a \sum_{s'} T(s, a, s')\left[R(s, a, s') + \gamma . V^*(s')\right] \quad \text{for all } s$$

- $T(s, a, s')$ is the transition probability from state s to state s', given that the agent chose action a.
- $R(s, a, s')$ is the reward that the agent gets when it goes from state s to state s', given that the agent chose action a.
- γ is the discount rate.

This equation leads directly to an algorithm that can precisely estimate the optimal state value of every possible state: you first initialize all the state value estimates to zero, and then you iteratively update them using the *Value Iteration* algorithm (see Equation 16-2). A remarkable result is that, given enough time, these estimates are guaranteed to converge to the optimal state values, corresponding to the optimal policy.

Equation 16-2. Value Iteration algorithm

$$V_{k+1}(s) \leftarrow \max_{a} \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \,.\, V_k(s')\right] \quad \text{for all } s$$

- $V_k(s)$ is the estimated value of state s at the k^{th} iteration of the algorithm.

This algorithm is an example of *Dynamic Programming*, which breaks down a complex problem (in this case estimating a potentially infinite sum of discounted future rewards) into tractable sub-problems that can be tackled iteratively (in this case finding the action that maximizes the average reward plus the discounted next state value).

Knowing the optimal state values can be useful, in particular to evaluate a policy, but it does not tell the agent explicitly what to do. Luckily, Bellman found a very similar algorithm to estimate the optimal *state-action values*, generally called *Q-Values*. The optimal Q-Value of the state-action pair (s,a), noted $Q^*(s,a)$, is the sum of discounted future rewards the agent can expect on average after it reaches the state s and chooses action a, but before it sees the outcome of this action, assuming it acts optimally after that action.

Here is how it works: once again, you start by initializing all the Q-Value estimates to zero, then you update them using the *Q-Value Iteration* algorithm (see Equation 16-3).

Equation 16-3. Q-Value Iteration algorithm

$$Q_{k+1}(s,a) \leftarrow \sum_{s'} T(s,a,s')\left[R(s,a,s') + \gamma \,.\, \max_{a'} Q_k(s',a')\right] \quad \text{for all } (s,a)$$

Once you have the optimal Q-Values, defining the optimal policy, noted $\pi^*(s)$, is trivial: when the agent is in state s, it should choose the action with the highest Q-Value for that state: $\pi^*(s) = \underset{a}{\operatorname{argmax}}\ Q^*(s,a)$.

Let's apply this algorithm to the MDP represented in Figure 16-8. First, we need to define the MDP:

```
nan=np.nan  # represents impossible actions
T = np.array([  # shape=[s, a, s']
        [[0.7, 0.3, 0.0], [1.0, 0.0, 0.0], [0.8, 0.2, 0.0]],
```

```
        [[0.0, 1.0, 0.0], [nan, nan, nan], [0.0, 0.0, 1.0]],
        [[nan, nan, nan], [0.8, 0.1, 0.1], [nan, nan, nan]],
    ])
R = np.array([  # shape=[s, a, s']
        [[10., 0.0, 0.0], [0.0, 0.0, 0.0], [0.0, 0.0, 0.0]],
        [[10., 0.0, 0.0], [nan, nan, nan], [0.0, 0.0, -50.]],
        [[nan, nan, nan], [40., 0.0, 0.0], [nan, nan, nan]],
    ])
possible_actions = [[0, 1, 2], [0, 2], [1]]
```

Now let's run the Q-Value Iteration algorithm:

```
Q = np.full((3, 3), -np.inf)  # -inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0  # Initial value = 0.0, for all possible actions

learning_rate = 0.01
discount_rate = 0.95
n_iterations = 100

for iteration in range(n_iterations):
    Q_prev = Q.copy()
    for s in range(3):
        for a in possible_actions[s]:
            Q[s, a] = np.sum([
                T[s, a, sp] * (R[s, a, sp] + discount_rate * np.max(Q_prev[sp]))
                for sp in range(3)
            ])
```

The resulting Q-Values look like this:

```
>>> Q
array([[ 21.89498982,  20.80024033,  16.86353093],
       [  1.11669335,         -inf,   1.17573546],
       [        -inf,  53.86946068,         -inf]])
>>> np.argmax(Q, axis=1)  # optimal action for each state
array([0, 2, 1])
```

This gives us the optimal policy for this MDP, when using a discount rate of 0.95: in state s_0 choose action a_0, in state s_1 choose action a_2 (go through the fire!), and in state s_2 choose action a_1 (the only possible action). Interestingly, if you reduce the discount rate to 0.9, the optimal policy changes: in state s_1 the best action becomes a_0 (stay put; don't go through the fire). It makes sense because if you value the present much more than the future, then the prospect of future rewards is not worth immediate pain.

Temporal Difference Learning and Q-Learning

Reinforcement Learning problems with discrete actions can often be modeled as Markov decision processes, but the agent initially has no idea what the transition probabilities are (it does not know $T(s, a, s')$), and it does not know what the rewards are going to be either (it does not know $R(s, a, s')$). It must experience each state and each transition at least once to know the rewards, and it must experience them multiple times if it is to have a reasonable estimate of the transition probabilities.

The *Temporal Difference Learning* (TD Learning) algorithm is very similar to the Value Iteration algorithm, but tweaked to take into account the fact that the agent has only partial knowledge of the MDP. In general we assume that the agent initially knows only the possible states and actions, and nothing more. The agent uses an *exploration policy*—for example, a purely random policy—to explore the MDP, and as it progresses the TD Learning algorithm updates the estimates of the state values based on the transitions and rewards that are actually observed (see Equation 16-4).

Equation 16-4. TD Learning algorithm

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha\left(r + \gamma \, . \, V_k(s')\right)$$

- α is the learning rate (e.g., 0.01).

TD Learning has many similarities with Stochastic Gradient Descent, in particular the fact that it handles one sample at a time. Just like SGD, it can only truly converge if you gradually reduce the learning rate (otherwise it will keep bouncing around the optimum).

For each state s, this algorithm simply keeps track of a running average of the immediate rewards the agent gets upon leaving that state, plus the rewards it expects to get later (assuming it acts optimally).

Similarly, the Q-Learning algorithm is an adaptation of the Q-Value Iteration algorithm to the situation where the transition probabilities and the rewards are initially unknown (see Equation 16-5).

Equation 16-5. Q-Learning algorithm

$$Q_{k+1}(s,a) \leftarrow (1-\alpha)Q_k(s,a) + \alpha\left(r + \gamma \, . \, \max_{a'} \; Q_k(s',a')\right)$$

For each state-action pair (s, a), this algorithm keeps track of a running average of the rewards r the agent gets upon leaving the state s with action a, plus the rewards it expects to get later. Since the target policy would act optimally, we take the maximum of the Q-Value estimates for the next state.

Here is how Q-Learning can be implemented:

```
import numpy.random as rnd

learning_rate0 = 0.05
learning_rate_decay = 0.1
n_iterations = 20000

s = 0 # start in state 0

Q = np.full((3, 3), -np.inf)  # -inf for impossible actions
for state, actions in enumerate(possible_actions):
    Q[state, actions] = 0.0  # Initial value = 0.0, for all possible actions

for iteration in range(n_iterations):
    a = rnd.choice(possible_actions[s])  # choose an action (randomly)
    sp = rnd.choice(range(3), p=T[s, a]) # pick next state using T[s, a]
    reward = R[s, a, sp]
    learning_rate = learning_rate0 / (1 + iteration * learning_rate_decay)
    Q[s, a] = learning_rate * Q[s, a] + (1 - learning_rate) * (
            reward + discount_rate * np.max(Q[sp])
        )
    s = sp # move to next state
```

Given enough iterations, this algorithm will converge to the optimal Q-Values. This is called an *off-policy* algorithm because the policy being trained is not the one being executed. It is somewhat surprising that this algorithm is capable of learning the optimal policy by just watching an agent act randomly (imagine learning to play golf when your teacher is a drunken monkey). Can we do better?

Exploration Policies

Of course Q-Learning can work only if the exploration policy explores the MDP thoroughly enough. Although a purely random policy is guaranteed to eventually visit every state and every transition many times, it may take an extremely long time to do so. Therefore, a better option is to use the *ε-greedy policy*: at each step it acts randomly with probability ε, or greedily (choosing the action with the highest Q-Value) with probability 1-ε. The advantage of the ε-greedy policy (compared to a completely random policy) is that it will spend more and more time exploring the interesting parts of the environment, as the Q-Value estimates get better and better, while still spending some time visiting unknown regions of the MDP. It is quite common to start with a high value for ε (e.g., 1.0) and then gradually reduce it (e.g., down to 0.05).

Alternatively, rather than relying on chance for exploration, another approach is to encourage the exploration policy to try actions that it has not tried much before. This can be implemented as a bonus added to the Q-Value estimates, as shown in Equation 16-6.

Equation 16-6. Q-Learning using an exploration function

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left(r + \gamma \cdot \max_{\alpha'} f\left(Q(s',a'), N(s',a')\right)\right)$$

- $N(s', a')$ counts the number of times the action a' was chosen in state s'.
- $f(q, n)$ is an *exploration function*, such as $f(q, n) = q + K/(1 + n)$, where K is a curiosity hyperparameter that measures how much the agent is attracted to to the unknown.

Approximate Q-Learning

The main problem with Q-Learning is that it does not scale well to large (or even medium) MDPs with many states and actions. Consider trying to use Q-Learning to train an agent to play Ms. Pac-Man. There are over 250 pellets that Ms. Pac-Man can eat, each of which can be present or absent (i.e., already eaten). So the number of possible states is greater than $2^{250} \approx 10^{75}$ (and that's considering the possible states only of the pellets). This is way more than atoms in the observable universe, so there's absolutely no way you can keep track of an estimate for every single Q-Value.

The solution is to find a function that approximates the Q-Values using a manageable number of parameters. This is called *Approximate Q-Learning*. For years it was recommended to use linear combinations of hand-crafted features extracted from the state (e.g., distance of the closest ghosts, their directions, and so on) to estimate Q-Values, but DeepMind showed that using deep neural networks can work much better, especially for complex problems, and it does not require any feature engineering. A DNN used to estimate Q-Values is called a *deep Q-network* (DQN), and using a DQN for Approximate Q-Learning is called *Deep Q-Learning*.

In the rest of this chapter, we will use Deep Q-Learning to train an agent to play Ms. Pac-Man, much like DeepMind did in 2013. The code can easily be tweaked to learn to play the majority of Atari games quite well. It can achieve superhuman skill at most action games, but it is not so good at games with long-running storylines.

Learning to Play Ms. Pac-Man Using Deep Q-Learning

Since we will be using an Atari environment, we must first install OpenAI gym's Atari dependencies. While we're at it, we will also install dependencies for other OpenAI gym environments that you may want to play with. On macOS, assuming you have installed Homebrew, you need to run:

```
$ brew install cmake boost boost-python sdl2 swig wget
```

On Ubuntu, type the following command (replacing `python3` with `python` if you are using Python 2):

```
$ apt-get install -y python3-numpy python3-dev cmake zlib1g-dev libjpeg-dev\
    xvfb libav-tools xorg-dev python3-opengl libboost-all-dev libsdl2-dev swig
```

Then install the extra Python modules:

```
$ pip3 install --upgrade 'gym[all]'
```

If everything went well, you should be able to create a Ms. Pac-Man environment:

```
>>> env = gym.make("MsPacman-v0")
>>> obs = env.reset()
>>> obs.shape  # [height, width, channels]
(210, 160, 3)
>>> env.action_space
Discrete(9)
```

As you can see, there are nine discrete actions available, which correspond to the nine possible positions of the joystick (left, right, up, down, center, upper left, and so on), and the observations are simply screenshots of the Atari screen (see Figure 16-9, left), represented as 3D NumPy arrays. These images are a bit large, so we will create a small preprocessing function that will crop the image and shrink it down to 88 × 80 pixels, convert it to grayscale, and improve the contrast of Ms. Pac-Man. This will reduce the amount of computations required by the DQN, and speed up training.

```
mspacman_color = np.array([210, 164, 74]).mean()

def preprocess_observation(obs):
    img = obs[1:176:2, ::2] # crop and downsize
    img = img.mean(axis=2) # to greyscale
    img[img==mspacman_color] = 0 # improve contrast
    img = (img - 128) / 128 - 1 # normalize from -1. to 1.
    return img.reshape(88, 80, 1)
```

The result of preprocessing is shown in Figure 16-9 (right).

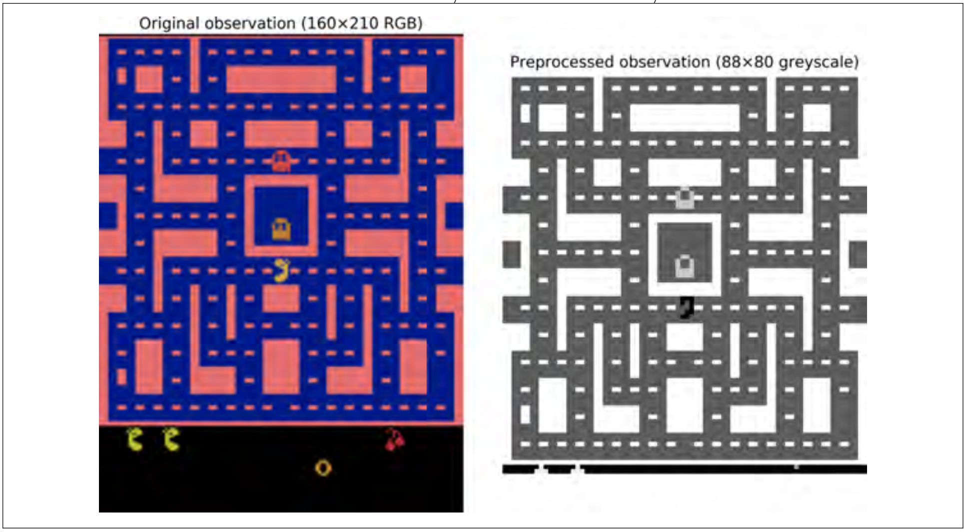


Figure 16-9. Ms. Pac-Man observation, original (left) and after preprocessing (right)

Next, let's create the DQN. It could just take a state-action pair (*s*,*a*) as input, and output an estimate of the corresponding Q-Value *Q*(*s*,*a*), but since the actions are discrete it is more convenient to use a neural network that takes only a state *s* as input and outputs one Q-Value estimate per action. The DQN will be composed of three convolutional layers, followed by two fully connected layers, including the output layer (see Figure 16-10).

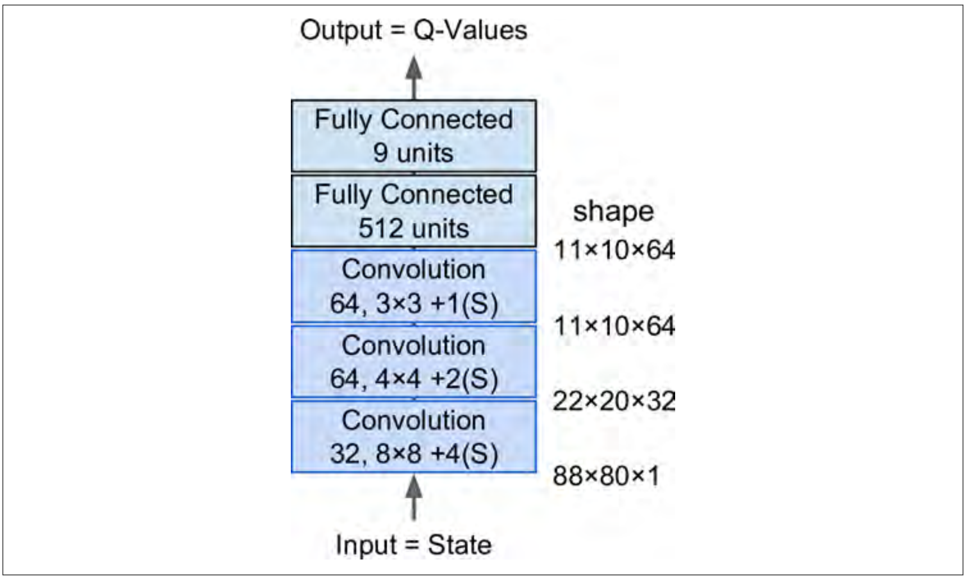


Figure 16-10. Deep Q-network to play Ms. Pac-Man

As we will see, the training algorithm we will use requires two DQNs with the same architecture (but different parameters): one will be used to drive Ms. Pac-Man during training (the *actor*), and the other will watch the actor and learn from its trials and errors (the *critic*). At regular intervals we will copy the critic to the actor. Since we need two identical DQNs, we will create a `q_network()` function to build them:

```
from tensorflow.contrib.layers import convolution2d, fully_connected

input_height = 88
input_width = 80
input_channels = 1
conv_n_maps = [32, 64, 64]
conv_kernel_sizes = [(8,8), (4,4), (3,3)]
conv_strides = [4, 2, 1]
conv_paddings = ["SAME"]*3
conv_activation = [tf.nn.relu]*3
n_hidden_in = 64 * 11 * 10  # conv3 has 64 maps of 11x10 each
n_hidden = 512
hidden_activation = tf.nn.relu
n_outputs = env.action_space.n  # 9 discrete actions are available
initializer = tf.contrib.layers.variance_scaling_initializer()

def q_network(X_state, scope):
    prev_layer = X_state
    conv_layers = []
    with tf.variable_scope(scope) as scope:
        for n_maps, kernel_size, stride, padding, activation in zip(
                conv_n_maps, conv_kernel_sizes, conv_strides,
                conv_paddings, conv_activation):
            prev_layer = convolution2d(
                prev_layer, num_outputs=n_maps, kernel_size=kernel_size,
                stride=stride, padding=padding, activation_fn=activation,
                weights_initializer=initializer)
            conv_layers.append(prev_layer)
        last_conv_layer_flat = tf.reshape(prev_layer, shape=[-1, n_hidden_in])
        hidden = fully_connected(
            last_conv_layer_flat, n_hidden, activation_fn=hidden_activation,
            weights_initializer=initializer)
        outputs = fully_connected(
            hidden, n_outputs, activation_fn=None,
            weights_initializer=initializer)
    trainable_vars = tf.get_collection(tf.GraphKeys.TRAINABLE_VARIABLES,
                                       scope=scope.name)
    trainable_vars_by_name = {var.name[len(scope.name):]: var
                              for var in trainable_vars}
    return outputs, trainable_vars_by_name
```

The first part of this code defines the hyperparameters of the DQN architecture. Then the `q_network()` function creates the DQN, taking the environment's state `X_state` as input, and the name of the variable scope. Note that we will just use one observation to represent the environment's state since there's almost no hidden state (except for blinking objects and the ghosts' directions).

The `trainable_vars_by_name` dictionary gathers all the trainable variables of this DQN. It will be useful in a minute when we create operations to copy the critic DQN to the actor DQN. The keys of the dictionary are the names of the variables, stripping the part of the prefix that just corresponds to the scope's name. It looks like this:

```
>>> trainable_vars_by_name
{'/Conv/biases:0': <tensorflow.python.ops.variables.Variable at 0x121cf7b50>,
 '/Conv/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_1/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_1/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_2/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/Conv_2/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected/weights:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected_1/biases:0': <tensorflow.python.ops.variables.Variable...>,
 '/fully_connected_1/weights:0': <tensorflow.python.ops.variables.Variable...>}
```

Now let's create the input placeholder, the two DQNs, and the operation to copy the critic DQN to the actor DQN:

```
X_state = tf.placeholder(tf.float32, shape=[None, input_height, input_width,
                                            input_channels])
actor_q_values, actor_vars = q_network(X_state, scope="q_networks/actor")
critic_q_values, critic_vars = q_network(X_state, scope="q_networks/critic")

copy_ops = [actor_var.assign(critic_vars[var_name])
            for var_name, actor_var in actor_vars.items()]
copy_critic_to_actor = tf.group(*copy_ops)
```

Let's step back for a second: we now have two DQNs that are both capable of taking an environment state (i.e., a preprocessed observation) as input and outputting an estimated Q-Value for each possible action in that state. Plus we have an operation called `copy_critic_to_actor` to copy all the trainable variables of the critic DQN to the actor DQN. We use TensorFlow's `tf.group()` function to group all the assignment operations into a single convenient operation.

The actor DQN can be used to play Ms. Pac-Man (initially very badly). As discussed earlier, you want it to explore the game thoroughly enough, so you generally want to combine it with an *ε-greedy policy* or another exploration strategy.

But what about the critic DQN? How will it learn to play the game? The short answer is that it will try to make its Q-Value predictions match the Q-Values estimated by the actor through its experience of the game. Specifically, we will let the actor play for a while, storing all its experiences in a *replay memory*. Each memory will be a 5-tuple (state, action, next state, reward, continue), where the "continue" item will be equal to 0.0 when the game is over, or 1.0 otherwise. Next, at regular intervals we will sample a batch of memories from the replay memory, and we will estimate the Q-Values from these memories. Finally, we will train the critic DQN to predict these Q-Values using regular supervised learning techniques. Once every few training iterations, we will copy the critic DQN to the actor DQN. And that's it! Equation 16-7 shows the cost function used to train the critic DQN:

Equation 16-7. Deep Q-Learning cost function

$$J\left(\theta_{\text{critic}}\right) = \frac{1}{m}\sum_{i=1}^{m}\left(y^{(i)} - Q\left(s^{(i)}, a^{(i)}, \theta_{\text{critic}}\right)\right)^2$$

$$\text{with } y^{(i)} = r^{(i)} + \gamma \cdot \max_{a'} Q\left(s'^{(i)}, a', \theta_{\text{actor}}\right)$$

- $s^{(i)}$, $a^{(i)}$, $r^{(i)}$ and $s'^{(i)}$ are respectively the state, action, reward, and next state of the i^{th} memory sampled from the replay memory.
- m is the size of the memory batch.
- θ_{critic} and θ_{actor} are the critic and the actor's parameters.
- $Q(s^{(i)},a^{(i)},\theta_{\text{critic}})$ is the critic DQN's prediction of the i^{th} memorized state-action's Q-Value.
- $Q(s'^{(i)}, a', \theta_{\text{actor}})$ is the actor DQN's prediction of the Q-Value it can expect from the next state $s'^{(i)}$ if it chooses action a'.
- $y^{(i)}$ is the target Q-Value for the i^{th} memory. Note that it is equal to the reward actually observed by the actor, plus the actor's *prediction* of what future rewards it should expect if it were to play optimally (as far as it knows).
- $J(\theta_{\text{critic}})$ is the cost function used to train the critic DQN. As you can see, it is just the Mean Squared Error between the target Q-Values $y^{(i)}$ as estimated by the actor DQN, and the critic DQN's predictions of these Q-Values.

> The replay memory is optional, but highly recommended. Without it, you would train the critic DQN using consecutive experiences that may be very correlated. This would introduce a lot of bias and slow down the training algorithm's convergence. By using a replay memory, we ensure that the memories fed to the training algorithm can be fairly uncorrelated.

Let's add the critic DQN's training operations. First, we need to be able to compute its predicted Q-Values for each state-action in the memory batch. Since the DQN outputs one Q-Value for every possible action, we need to keep only the Q-Value that corresponds to the action that was actually chosen in this memory. For this, we will convert the action to a one-hot vector (recall that this is a vector full of 0s except for a

1 at the i^{th} index), and multiply it by the Q-Values: this will zero out all Q-Values except for the one corresponding to the memorized action. Then just sum over the first axis to obtain only the desired Q-Value prediction for each memory.

```
X_action = tf.placeholder(tf.int32, shape=[None])
q_value = tf.reduce_sum(critic_q_values * tf.one_hot(X_action, n_outputs),
                        axis=1, keep_dims=True)
```

Next let's add the training operations, assuming the target Q-Values will be fed through a placeholder. We also create a nontrainable variable called `global_step`. The optimizer's `minimize()` operation will take care of incrementing it. Plus we create the usual `init` operation and a `Saver`.

```
y = tf.placeholder(tf.float32, shape=[None, 1])
cost = tf.reduce_mean(tf.square(y - q_value))
global_step = tf.Variable(0, trainable=False, name='global_step')
optimizer = tf.train.AdamOptimizer(learning_rate)
training_op = optimizer.minimize(cost, global_step=global_step)

init = tf.global_variables_initializer()
saver = tf.train.Saver()
```

That's it for the construction phase. Before we look at the execution phase, we will need a couple of tools. First, let's start by implementing the replay memory. We will use a `deque` list since it is very efficient at pushing items to the queue and popping them out from the end of the list when the maximum memory size is reached. We will also write a small function to randomly sample a batch of experiences from the replay memory:

```
from collections import deque

replay_memory_size = 10000
replay_memory = deque([], maxlen=replay_memory_size)

def sample_memories(batch_size):
    indices = rnd.permutation(len(replay_memory))[:batch_size]
    cols = [[], [], [], [], []] # state, action, reward, next_state, continue
    for idx in indices:
        memory = replay_memory[idx]
        for col, value in zip(cols, memory):
            col.append(value)
    cols = [np.array(col) for col in cols]
    return (cols[0], cols[1], cols[2].reshape(-1, 1), cols[3],
            cols[4].reshape(-1, 1))
```

Next, we will need the actor to explore the game. We will use the ε-greedy policy, and gradually decrease ε from 1.0 to 0.05, in 50,000 training steps:

```
eps_min = 0.05
eps_max = 1.0
eps_decay_steps = 50000
```

```
def epsilon_greedy(q_values, step):
    epsilon = max(eps_min, eps_max - (eps_max-eps_min) * step/eps_decay_steps)
    if rnd.rand() < epsilon:
        return rnd.randint(n_outputs) # random action
    else:
        return np.argmax(q_values) # optimal action
```

That's it! We have all we need to start training. The execution phase does not contain anything too complex, but it is a bit long, so take a deep breath. Ready? Let's go! First, let's initialize a few variables:

```
n_steps = 100000  # total number of training steps
training_start = 1000  # start training after 1,000 game iterations
training_interval = 3  # run a training step every 3 game iterations
save_steps = 50  # save the model every 50 training steps
copy_steps = 25  # copy the critic to the actor every 25 training steps
discount_rate = 0.95
skip_start = 90  # skip the start of every game (it's just waiting time)
batch_size = 50
iteration = 0  # game iterations
checkpoint_path = "./my_dqn.ckpt"
done = True # env needs to be reset
```

Next, let's open the session and run the main training loop:

```
with tf.Session() as sess:
    if os.path.isfile(checkpoint_path):
        saver.restore(sess, checkpoint_path)
    else:
        init.run()
    while True:
        step = global_step.eval()
        if step >= n_steps:
            break
        iteration += 1
        if done: # game over, start again
            obs = env.reset()
            for skip in range(skip_start): # skip the start of each game
                obs, reward, done, info = env.step(0)
            state = preprocess_observation(obs)

        # Actor evaluates what to do
        q_values = actor_q_values.eval(feed_dict={X_state: [state]})
        action = epsilon_greedy(q_values, step)

        # Actor plays
        obs, reward, done, info = env.step(action)
        next_state = preprocess_observation(obs)

        # Let's memorize what just happened
        replay_memory.append((state, action, reward, next_state, 1.0 - done))
        state = next_state
```

```
        if iteration < training_start or iteration % training_interval != 0:
            continue

        # Critic learns
        X_state_val, X_action_val, rewards, X_next_state_val, continues = (
            sample_memories(batch_size))
        next_q_values = actor_q_values.eval(
            feed_dict={X_state: X_next_state_val})
        max_next_q_values = np.max(next_q_values, axis=1, keepdims=True)
        y_val = rewards + continues * discount_rate * max_next_q_values
        training_op.run(feed_dict={X_state: X_state_val,
                                   X_action: X_action_val, y: y_val})

        # Regularly copy critic to actor
        if step % copy_steps == 0:
            copy_critic_to_actor.run()

        # And save regularly
        if step % save_steps == 0:
            saver.save(sess, checkpoint_path)
```

We start by restoring the models if a checkpoint file exists, or else we just initialize the variables normally. Then the main loop starts, where `iteration` counts the total number of game steps we have gone through since the program started, and `step` counts the total number of training steps since training started (if a checkpoint is restored, the global step is restored as well). Then the code resets the game (and skips the first boring game steps, where nothing happens). Next, the actor evaluates what to do, and plays the game, and its experience is memorized in replay memory. Then, at regular intervals (after a warmup period), the critic goes through a training step. It samples a batch of memories and asks the actor to estimate the Q-Values of all actions for the next state, and it applies Equation 16-7 to compute the target Q-Value `y_val`. The only tricky part here is that we must multiply the next state's Q-Values by the `continues` vector to zero out the Q-Values corresponding to memories where the game was over. Next we run a training operation to improve the critic's ability to predict Q-Values. Finally, at regular intervals we copy the critic to the actor, and we save the model.

Unfortunately, training is very slow: if you use your laptop for training, it will take days before Ms. Pac-Man gets any good, and if you look at the learning curve, measuring the average rewards per episode, you will notice that it is extremely noisy. At some points there may be no apparent progress for a very long time until suddenly the agent learns to survive a reasonable amount of time. As mentioned earlier, one solution is to inject as much prior knowledge as possible into the model (e.g., through preprocessing, rewards, and so on), and you can also try to bootstrap the model by first training it to imitate a basic strategy. In any case, RL still requires quite a lot of patience and tweaking, but the end result is very exciting.

Exercises

1. How would you define Reinforcement Learning? How is it different from regular supervised or unsupervised learning?
2. Can you think of three possible applications of RL that were not mentioned in this chapter? For each of them, what is the environment? What is the agent? What are possible actions? What are the rewards?
3. What is the discount rate? Can the optimal policy change if you modify the discount rate?
4. How do you measure the performance of a Reinforcement Learning agent?
5. What is the credit assignment problem? When does it occur? How can you alleviate it?
6. What is the point of using a replay memory?
7. What is an off-policy RL algorithm?
8. Use Deep Q-Learning to tackle OpenAI gym's "BypedalWalker-v2." The Q-networks do not need to be very deep for this task.
9. Use policy gradients to train an agent to play *Pong*, the famous Atari game (`Pong-v0` in the OpenAI gym). Beware: an individual observation is insufficient to tell the direction and speed of the ball. One solution is to pass two observations at a time to the neural network policy. To reduce dimensionality and speed up training, you should definitely preprocess these images (crop, resize, and convert them to black and white), and possibly merge them into a single image (e.g., by overlaying them).
10. If you have about $100 to spare, you can purchase a Raspberry Pi 3 plus some cheap robotics components, install TensorFlow on the Pi, and go wild! For an example, check out this fun post by Lukas Biewald, or take a look at GoPiGo or BrickPi. Why not try to build a real-life cartpole by training the robot using pol-

icy gradients? Or build a robotic spider that learns to walk; give it rewards any time it gets closer to some objective (you will need sensors to measure the distance to the objective). The only limit is your imagination.

Solutions to these exercises are available in Appendix A.

Thank You!

Before we close the last chapter of this book, I would like to thank you for reading it up to the last paragraph. I truly hope that you had as much pleasure reading this book as I had writing it, and that it will be useful for your projects, big or small.

If you find errors, please send feedback. More generally, I would love to know what you think, so please don't hesitate to contact me via O'Reilly, or through the *ageron/handson-ml* GitHub project.

Going forward, my best advice to you is to practice and practice: try going through all the exercises if you have not done so already, play with the Jupyter notebooks, join Kaggle.com or some other ML community, watch ML courses, read papers, attend conferences, meet experts. You may also want to study some topics that we did not cover in this book, including recommender systems, clustering algorithms, anomaly detection algorithms, and genetic algorithms.

My greatest hope is that this book will inspire you to build a wonderful ML application that will benefit all of us! What will it be?

Aurélien Géron, November 26th, 2016

APPENDIX A
Exercise Solutions

Solutions to the coding exercises are available in the online Jupyter notebooks at *https://github.com/ageron/handson-ml*.

Chapter 1: The Machine Learning Landscape

1. Machine Learning is about building systems that can learn from data. Learning means getting better at some task, given some performance measure.
2. Machine Learning is great for complex problems for which we have no algorithmic solution, to replace long lists of hand-tuned rules, to build systems that adapt to fluctuating environments, and finally to help humans learn (e.g., data mining).
3. A labeled training set is a training set that contains the desired solution (a.k.a. a label) for each instance.
4. The two most common supervised tasks are regression and classification.
5. Common unsupervised tasks include clustering, visualization, dimensionality reduction, and association rule learning.
6. Reinforcement Learning is likely to perform best if we want a robot to learn to walk in various unknown terrains since this is typically the type of problem that Reinforcement Learning tackles. It might be possible to express the problem as a supervised or semisupervised learning problem, but it would be less natural.
7. If you don't know how to define the groups, then you can use a clustering algorithm (unsupervised learning) to segment your customers into clusters of similar customers. However, if you know what groups you would like to have, then you

can feed many examples of each group to a classification algorithm (supervised learning), and it will classify all your customers into these groups.

8. Spam detection is a typical supervised learning problem: the algorithm is fed many emails along with their label (spam or not spam).

9. An online learning system can learn incrementally, as opposed to a batch learning system. This makes it capable of adapting rapidly to both changing data and autonomous systems, and of training on very large quantities of data.

10. Out-of-core algorithms can handle vast quantities of data that cannot fit in a computer's main memory. An out-of-core learning algorithm chops the data into mini-batches and uses online learning techniques to learn from these mini-batches.

11. An instance-based learning system learns the training data by heart; then, when given a new instance, it uses a similarity measure to find the most similar learned instances and uses them to make predictions.

12. A model has one or more model parameters that determine what it will predict given a new instance (e.g., the slope of a linear model). A learning algorithm tries to find optimal values for these parameters such that the model generalizes well to new instances. A hyperparameter is a parameter of the learning algorithm itself, not of the model (e.g., the amount of regularization to apply).

13. Model-based learning algorithms search for an optimal value for the model parameters such that the model will generalize well to new instances. We usually train such systems by minimizing a cost function that measures how bad the system is at making predictions on the training data, plus a penalty for model complexity if the model is regularized. To make predictions, we feed the new instance's features into the model's prediction function, using the parameter values found by the learning algorithm.

14. Some of the main challenges in Machine Learning are the lack of data, poor data quality, nonrepresentative data, uninformative features, excessively simple models that underfit the training data, and excessively complex models that overfit the data.

15. If a model performs great on the training data but generalizes poorly to new instances, the model is likely overfitting the training data (or we got extremely lucky on the training data). Possible solutions to overfitting are getting more data, simplifying the model (selecting a simpler algorithm, reducing the number of parameters or features used, or regularizing the model), or reducing the noise in the training data.

16. A test set is used to estimate the generalization error that a model will make on new instances, before the model is launched in production.

17. A validation set is used to compare models. It makes it possible to select the best model and tune the hyperparameters.
18. If you tune hyperparameters using the test set, you risk overfitting the test set, and the generalization error you measure will be optimistic (you may launch a model that performs worse than you expect).
19. Cross-validation is a technique that makes it possible to compare models (for model selection and hyperparameter tuning) without the need for a separate validation set. This saves precious training data.

Chapter 2: End-to-End Machine Learning Project

See the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 3: Classification

See the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 4: Training Linear Models

1. If you have a training set with millions of features you can use Stochastic Gradient Descent or Mini-batch Gradient Descent, and perhaps Batch Gradient Descent if the training set fits in memory. But you cannot use the Normal Equation because the computational complexity grows quickly (more than quadratically) with the number of features.
2. If the features in your training set have very different scales, the cost function will have the shape of an elongated bowl, so the Gradient Descent algorithms will take a long time to converge. To solve this you should scale the data before training the model. Note that the Normal Equation will work just fine without scaling.
3. Gradient Descent cannot get stuck in a local minimum when training a Logistic Regression model because the cost function is convex.[1]
4. If the optimization problem is convex (such as Linear Regression or Logistic Regression), and assuming the learning rate is not too high, then all Gradient Descent algorithms will approach the global optimum and end up producing fairly similar models. However, unless you gradually reduce the learning rate, Stochastic GD and Mini-batch GD will never truly converge; instead, they will keep jumping back and forth around the global optimum. This means that even

1 If you draw a straight line between any two points on the curve, the line never crosses the curve.

if you let them run for a very long time, these Gradient Descent algorithms will produce slightly different models.

5. If the validation error consistently goes up after every epoch, then one possibility is that the learning rate is too high and the algorithm is diverging. If the training error also goes up, then this is clearly the problem and you should reduce the learning rate. However, if the training error is not going up, then your model is overfitting the training set and you should stop training.
6. Due to their random nature, neither Stochastic Gradient Descent nor Mini-batch Gradient Descent is guaranteed to make progress at every single training iteration. So if you immediately stop training when the validation error goes up, you may stop much too early, before the optimum is reached. A better option is to save the model at regular intervals, and when it has not improved for a long time (meaning it will probably never beat the record), you can revert to the best saved model.
7. Stochastic Gradient Descent has the fastest training iteration since it considers only one training instance at a time, so it is generally the first to reach the vicinity of the global optimum (or Mini-batch GD with a very small mini-batch size). However, only Batch Gradient Descent will actually converge, given enough training time. As mentioned, Stochastic GD and Mini-batch GD will bounce around the optimum, unless you gradually reduce the learning rate.
8. If the validation error is much higher than the training error, this is likely because your model is overfitting the training set. One way to try to fix this is to reduce the polynomial degree: a model with fewer degrees of freedom is less likely to overfit. Another thing you can try is to regularize the model—for example, by adding an ℓ_2 penalty (Ridge) or an ℓ_1 penalty (Lasso) to the cost function. This will also reduce the degrees of freedom of the model. Lastly, you can try to increase the size of the training set.
9. If both the training error and the validation error are almost equal and fairly high, the model is likely underfitting the training set, which means it has a high bias. You should try reducing the regularization hyperparameter α.
10. Let's see:

    - A model with some regularization typically performs better than a model without any regularization, so you should generally prefer Ridge Regression over plain Linear Regression.[2]
    - Lasso Regression uses an ℓ_1 penalty, which tends to push the weights down to exactly zero. This leads to sparse models, where all weights are zero except for

2 Moreover, the Normal Equation requires computing the inverse of a matrix, but that matrix is not always invertible. In contrast, the matrix for Ridge Regression is always invertible.

the most important weights. This is a way to perform feature selection automatically, which is good if you suspect that only a few features actually matter. When you are not sure, you should prefer Ridge Regression.

- Elastic Net is generally preferred over Lasso since Lasso may behave erratically in some cases (when several features are strongly correlated or when there are more features than training instances). However, it does add an extra hyperparameter to tune. If you just want Lasso without the erratic behavior, you can just use Elastic Net with an `l1_ratio` close to 1.

11. If you want to classify pictures as outdoor/indoor and daytime/nighttime, since these are not exclusive classes (i.e., all four combinations are possible) you should train two Logistic Regression classifiers.
12. See the Jupyter notebooks available at *https://github.com/ageron/handson-ml*.

Chapter 5: Support Vector Machines

1. The fundamental idea behind Support Vector Machines is to fit the widest possible "street" between the classes. In other words, the goal is to have the largest possible margin between the decision boundary that separates the two classes and the training instances. When performing soft margin classification, the SVM searches for a compromise between perfectly separating the two classes and having the widest possible street (i.e., a few instances may end up on the street). Another key idea is to use kernels when training on nonlinear datasets.
2. After training an SVM, a *support vector* is any instance located on the "street" (see the previous answer), including its border. The decision boundary is entirely determined by the support vectors. Any instance that is *not* a support vector (i.e., off the street) has no influence whatsoever; you could remove them, add more instances, or move them around, and as long as they stay off the street they won't affect the decision boundary. Computing the predictions only involves the support vectors, not the whole training set.
3. SVMs try to fit the largest possible "street" between the classes (see the first answer), so if the training set is not scaled, the SVM will tend to neglect small features (see Figure 5-2).
4. An SVM classifier can output the distance between the test instance and the decision boundary, and you can use this as a confidence score. However, this score cannot be directly converted into an estimation of the class probability. If you set `probability=True` when creating an SVM in Scikit-Learn, then after training it will calibrate the probabilities using Logistic Regression on the SVM's scores (trained by an additional five-fold cross-validation on the training data). This will add the `predict_proba()` and `predict_log_proba()` methods to the SVM.

5. This question applies only to linear SVMs since kernelized can only use the dual form. The computational complexity of the primal form of the SVM problem is proportional to the number of training instances m, while the computational complexity of the dual form is proportional to a number between m^2 and m^3. So if there are millions of instances, you should definitely use the primal form, because the dual form will be much too slow.

6. If an SVM classifier trained with an RBF kernel underfits the training set, there might be too much regularization. To decrease it, you need to increase `gamma` or `C` (or both).

7. Let's call the QP parameters for the hard-margin problem $\mathbf{H}'$, $\mathbf{f}'$, $\mathbf{A}'$ and $\mathbf{b}'$ (see "Quadratic Programming" on page 159). The QP parameters for the soft-margin problem have m additional parameters ($n_p = n + 1 + m$) and m additional constraints ($n_c = 2m$). They can be defined like so:

   - $\mathbf{H}$ is equal to $\mathbf{H}'$, plus m columns of 0s on the right and m rows of 0s at the bottom: $\mathbf{H} = \begin{pmatrix} \mathbf{H}' & 0 & \cdots \\ 0 & 0 & \\ \vdots & & \ddots \end{pmatrix}$
   - $\mathbf{f}$ is equal to $\mathbf{f}'$ with m additional elements, all equal to the value of the hyperparameter C.
   - $\mathbf{b}$ is equal to $\mathbf{b}'$ with m additional elements, all equal to 0.
   - $\mathbf{A}$ is equal to $\mathbf{A}'$, with an extra $m \times m$ identity matrix $\mathbf{I}_m$ appended to the right, $-\mathbf{I}_m$ just below it, and the rest filled with zeros: $\mathbf{A} = \begin{pmatrix} \mathbf{A}' & \mathbf{I}_m \\ \mathbf{0} & -\mathbf{I}_m \end{pmatrix}$

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

Chapter 6: Decision Trees

1. The depth of a well-balanced binary tree containing m leaves is equal to $\log_2(m)$[3], rounded up. A binary Decision Tree (one that makes only binary decisions, as is the case of all trees in Scikit-Learn) will end up more or less well balanced at the end of training, with one leaf per training instance if it is trained without restrictions. Thus, if the training set contains one million instances, the Decision Tree will have a depth of $\log_2(10^6) \approx 20$ (actually a bit more since the tree will generally not be perfectly well balanced).

3 $\log_2$ is the binary log, $\log_2(m) = \log(m) / \log(2)$.

2. A node's Gini impurity is generally lower than its parent's. This is ensured by the CART training algorithm's cost function, which splits each node in a way that minimizes the weighted sum of its children's Gini impurities. However, if one child is smaller than the other, it is possible for it to have a higher Gini impurity than its parent, as long as this increase is more than compensated for by a decrease of the other child's impurity. For example, consider a node containing four instances of class A and 1 of class B. Its Gini impurity is $1 - \frac{1}{5}^2 - \frac{4}{5}^2 = 0.32$. Now suppose the dataset is one-dimensional and the instances are lined up in the following order: A, B, A, A, A. You can verify that the algorithm will split this node after the second instance, producing one child node with instances A, B, and the other child node with instances A, A, A. The first child node's Gini impurity is $1 - \frac{1}{2}^2 - \frac{1}{2}^2 = 0.5$, which is higher than its parent. This is compensated for by the fact that the other node is pure, so the overall weighted Gini impurity is $\frac{2}{5} \times 0.5 + \frac{3}{5} \times 0 = 0.2$, which is lower than the parent's Gini impurity.
3. If a Decision Tree is overfitting the training set, it may be a good idea to decrease `max_depth`, since this will constrain the model, regularizing it.
4. Decision Trees don't care whether or not the training data is scaled or centered; that's one of the nice things about them. So if a Decision Tree underfits the training set, scaling the input features will just be a waste of time.
5. The computational complexity of training a Decision Tree is $O(n \times m \log(m))$. So if you multiply the training set size by 10, the training time will be multiplied by $K = (n \times 10m \times \log(10m)) / (n \times m \times \log(m)) = 10 \times \log(10m) / \log(m)$. If $m = 10^6$, then $K \approx 11.7$, so you can expect the training time to be roughly 11.7 hours.
6. Presorting the training set speeds up training only if the dataset is smaller than a few thousand instances. If it contains 100,000 instances, setting `presort=True` will considerably slow down training.

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

Chapter 7: Ensemble Learning and Random Forests

1. If you have trained five different models and they all achieve 95% precision, you can try combining them into a voting ensemble, which will often give you even better results. It works better if the models are very different (e.g., an SVM classifier, a Decision Tree classifier, a Logistic Regression classifier, and so on). It is even better if they are trained on different training instances (that's the whole point of bagging and pasting ensembles), but if not it will still work as long as the models are very different.

2. A hard voting classifier just counts the votes of each classifier in the ensemble and picks the class that gets the most votes. A soft voting classifier computes the average estimated class probability for each class and picks the class with the highest probability. This gives high-confidence votes more weight and often performs better, but it works only if every classifier is able to estimate class probabilities (e.g., for the SVM classifiers in Scikit-Learn you must set `probability=True`).

3. It is quite possible to speed up training of a bagging ensemble by distributing it across multiple servers, since each predictor in the ensemble is independent of the others. The same goes for pasting ensembles and Random Forests, for the same reason. However, each predictor in a boosting ensemble is built based on the previous predictor, so training is necessarily sequential, and you will not gain anything by distributing training across multiple servers. Regarding stacking ensembles, all the predictors in a given layer are independent of each other, so they can be trained in parallel on multiple servers. However, the predictors in one layer can only be trained after the predictors in the previous layer have all been trained.

4. With out-of-bag evaluation, each predictor in a bagging ensemble is evaluated using instances that it was not trained on (they were held out). This makes it possible to have a fairly unbiased evaluation of the ensemble without the need for an additional validation set. Thus, you have more instances available for training, and your ensemble can perform slightly better.

5. When you are growing a tree in a Random Forest, only a random subset of the features is considered for splitting at each node. This is true as well for Extra-Trees, but they go one step further: rather than searching for the best possible thresholds, like regular Decision Trees do, they use random thresholds for each feature. This extra randomness acts like a form of regularization: if a Random Forest overfits the training data, Extra-Trees might perform better. Moreover, since Extra-Trees don't search for the best possible thresholds, they are much faster to train than Random Forests. However, they are neither faster nor slower than Random Forests when making predictions.

6. If your AdaBoost ensemble underfits the training data, you can try increasing the number of estimators or reducing the regularization hyperparameters of the base estimator. You may also try slightly increasing the learning rate.

7. If your Gradient Boosting ensemble overfits the training set, you should try decreasing the learning rate. You could also use early stopping to find the right number of predictors (you probably have too many).

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

Chapter 8: Dimensionality Reduction

1. Motivations and drawbacks:
   - The main motivations for dimensionality reduction are:
     - — To speed up a subsequent training algorithm (in some cases it may even remove noise and redundant features, making the training algorithm perform better).
     - — To visualize the data and gain insights on the most important features.
     - — Simply to save space (compression).
   - The main drawbacks are:
     - — Some information is lost, possibly degrading the performance of subsequent training algorithms.
     - — It can be computationally intensive.
     - — It adds some complexity to your Machine Learning pipelines.
     - — Transformed features are often hard to interpret.
2. The curse of dimensionality refers to the fact that many problems that do not exist in low-dimensional space arise in high-dimensional space. In Machine Learning, one common manifestation is the fact that randomly sampled high-dimensional vectors are generally very sparse, increasing the risk of overfitting and making it very difficult to identify patterns in the data without having plenty of training data.
3. Once a dataset's dimensionality has been reduced using one of the algorithms we discussed, it is almost always impossible to perfectly reverse the operation, because some information gets lost during dimensionality reduction. Moreover, while some algorithms (such as PCA) have a simple reverse transformation procedure that can reconstruct a dataset relatively similar to the original, other algorithms (such as T-SNE) do not.
4. PCA can be used to significantly reduce the dimensionality of most datasets, even if they are highly nonlinear, because it can at least get rid of useless dimensions. However, if there are no useless dimensions—for example, the Swiss roll—then reducing dimensionality with PCA will lose too much information. You want to unroll the Swiss roll, not squash it.
5. That's a trick question: it depends on the dataset. Let's look at two extreme examples. First, suppose the dataset is composed of points that are almost perfectly aligned. In this case, PCA can reduce the dataset down to just one dimension while still preserving 95% of the variance. Now imagine that the dataset is composed of perfectly random points, scattered all around the 1,000 dimensions. In

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.

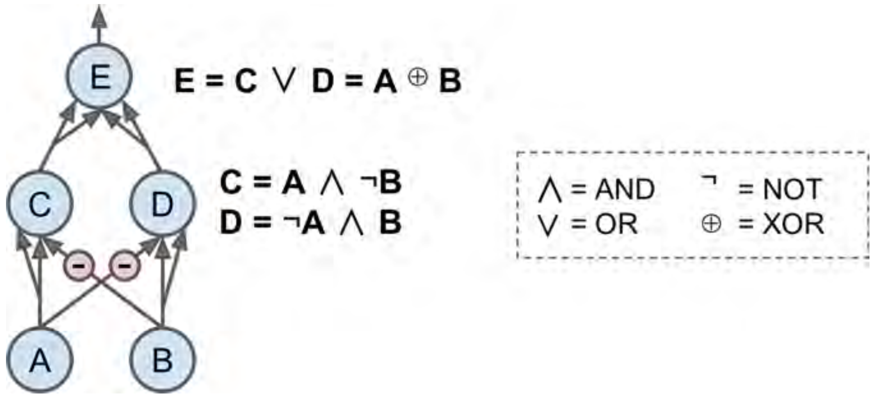
```
import tensorflow as tf

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.

- 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 $m \times 3$.
- $\mathbf{Y} = (\mathbf{X} \cdot \mathbf{W}_h + \mathbf{b}_h) \cdot \mathbf{W}_o + \mathbf{b}_o$. 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]
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.

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 $\exp(\log v) = v$).

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 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.

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.

- 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 + 9 = 15 million bytes (about 14.3 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) 18,613,600 bytes (about 17.8 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 × 4 × 200 × 300 × 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. 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 consecutive 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 inter-

ested 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), independently of how the agent actually acts. 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*.

APPENDIX B

Machine Learning Project Checklist

This checklist can guide you through your Machine Learning projects. There are eight main steps:

1. Frame the problem and look at the big picture.
2. Get the data.
3. Explore the data to gain insights.
4. Prepare the data to better expose the underlying data patterns to Machine Learning algorithms.
5. Explore many different models and short-list the best ones.
6. Fine-tune your models and combine them into a great solution.
7. Present your solution.
8. Launch, monitor, and maintain your system.

Obviously, you should feel free to adapt this checklist to your needs.

Frame the Problem and Look at the Big Picture

1. Define the objective in business terms.
2. How will your solution be used?
3. What are the current solutions/workarounds (if any)?
4. How should you frame this problem (supervised/unsupervised, online/offline, etc.)?
5. How should performance be measured?
6. Is the performance measure aligned with the business objective?

7. What would be the minimum performance needed to reach the business objective?
8. What are comparable problems? Can you reuse experience or tools?
9. Is human expertise available?
10. How would you solve the problem manually?
11. List the assumptions you (or others) have made so far.
12. Verify assumptions if possible.

Get the Data

Note: automate as much as possible so you can easily get fresh data.

1. List the data you need and how much you need.
2. Find and document where you can get that data.
3. Check how much space it will take.
4. Check legal obligations, and get authorization if necessary.
5. Get access authorizations.
6. Create a workspace (with enough storage space).
7. Get the data.
8. Convert the data to a format you can easily manipulate (without changing the data itself).
9. Ensure sensitive information is deleted or protected (e.g., anonymized).
10. Check the size and type of data (time series, sample, geographical, etc.).
11. Sample a test set, put it aside, and never look at it (no data snooping!).

Explore the Data

Note: try to get insights from a field expert for these steps.

1. Create a copy of the data for exploration (sampling it down to a manageable size if necessary).
2. Create a Jupyter notebook to keep a record of your data exploration.
3. Study each attribute and its characteristics:
    - Name
    - Type (categorical, int/float, bounded/unbounded, text, structured, etc.)

- % of missing values
- Noisiness and type of noise (stochastic, outliers, rounding errors, etc.)
- Possibly useful for the task?
- Type of distribution (Gaussian, uniform, logarithmic, etc.)

4. For supervised learning tasks, identify the target attribute(s).
5. Visualize the data.
6. Study the correlations between attributes.
7. Study how you would solve the problem manually.
8. Identify the promising transformations you may want to apply.
9. Identify extra data that would be useful (go back to "Get the Data" on page 498).
10. Document what you have learned.

Prepare the Data

Notes:

- Work on copies of the data (keep the original dataset intact).
- Write functions for all data transformations you apply, for five reasons:
  — So you can easily prepare the data the next time you get a fresh dataset
  — So you can apply these transformations in future projects
  — To clean and prepare the test set
  — To clean and prepare new data instances once your solution is live
  — To make it easy to treat your preparation choices as hyperparameters

1. Data cleaning:
   - Fix or remove outliers (optional).
   - Fill in missing values (e.g., with zero, mean, median…) or drop their rows (or columns).
2. Feature selection (optional):
   - Drop the attributes that provide no useful information for the task.
3. Feature engineering, where appropriate:
   - Discretize continuous features.

- Decompose features (e.g., categorical, date/time, etc.).
- Add promising transformations of features (e.g., log(x), sqrt(x), x^2, etc.).
- Aggregate features into promising new features.

4. Feature scaling: standardize or normalize features.

Short-List Promising Models

Notes:

- If the data is huge, you may want to sample smaller training sets so you can train many different models in a reasonable time (be aware that this penalizes complex models such as large neural nets or Random Forests).
- Once again, try to automate these steps as much as possible.

1. Train many quick and dirty models from different categories (e.g., linear, naive Bayes, SVM, Random Forests, neural net, etc.) using standard parameters.
2. Measure and compare their performance.
   - For each model, use *N*-fold cross-validation and compute the mean and standard deviation of the performance measure on the *N* folds.
3. Analyze the most significant variables for each algorithm.
4. Analyze the types of errors the models make.
   - What data would a human have used to avoid these errors?
5. Have a quick round of feature selection and engineering.
6. Have one or two more quick iterations of the five previous steps.
7. Short-list the top three to five most promising models, preferring models that make different types of errors.

Fine-Tune the System

Notes:

- You will want to use as much data as possible for this step, especially as you move toward the end of fine-tuning.
- As always automate what you can.

1. Fine-tune the hyperparameters using cross-validation.
   - Treat your data transformation choices as hyperparameters, especially when you are not sure about them (e.g., should I replace missing values with zero or with the median value? Or just drop the rows?).
   - Unless there are very few hyperparameter values to explore, prefer random search over grid search. If training is very long, you may prefer a Bayesian optimization approach (e.g., using Gaussian process priors, as described by Jasper Snoek, Hugo Larochelle, and Ryan Adams).[1]
2. Try Ensemble methods. Combining your best models will often perform better than running them individually.
3. Once you are confident about your final model, measure its performance on the test set to estimate the generalization error.

> Don't tweak your model after measuring the generalization error: you would just start overfitting the test set.

Present Your Solution

1. Document what you have done.
2. Create a nice presentation.
   - Make sure you highlight the big picture first.
3. Explain why your solution achieves the business objective.
4. Don't forget to present interesting points you noticed along the way.
   - Describe what worked and what did not.
   - List your assumptions and your system's limitations.
5. Ensure your key findings are communicated through beautiful visualizations or easy-to-remember statements (e.g., "the median income is the number-one predictor of housing prices").

1 "Practical Bayesian Optimization of Machine Learning Algorithms," J. Snoek, H. Larochelle, R. Adams (2012).

Launch!

1. Get your solution ready for production (plug into production data inputs, write unit tests, etc.).
2. Write monitoring code to check your system's live performance at regular intervals and trigger alerts when it drops.
   - Beware of slow degradation too: models tend to "rot" as data evolves.
   - Measuring performance may require a human pipeline (e.g., via a crowdsourcing service).
   - Also monitor your inputs' quality (e.g., a malfunctioning sensor sending random values, or another team's output becoming stale). This is particularly important for online learning systems.
3. Retrain your models on a regular basis on fresh data (automate as much as possible).

APPENDIX C
SVM Dual Problem

To understand *duality*, you first need to understand the *Lagrange multipliers* method. The general idea is to transform a constrained optimization objective into an unconstrained one, by moving the constraints into the objective function. Let's look at a simple example. Suppose you want to find the values of x and y that minimize the function $f(x,y) = x^2 + 2y$, subject to an *equality constraint*: $3x + 2y + 1 = 0$. Using the Lagrange multipliers method, we start by defining a new function called the *Lagrangian* (or *Lagrange function*): $g(x, y, \alpha) = f(x, y) - \alpha(3x + 2y + 1)$. Each constraint (in this case just one) is subtracted from the original objective, multiplied by a new variable called a Lagrange multiplier.

Joseph-Louis Lagrange showed that if $(\hat{x}, \hat{y})$ is a solution to the constrained optimization problem, then there must exist an $\hat{\alpha}$ such that $(\hat{x}, \hat{y}, \hat{\alpha})$ is a *stationary point* of the Lagrangian (a stationary point is a point where all partial derivatives are equal to zero). In other words, we can compute the partial derivatives of $g(x, y, \alpha)$ with regards to x, y, and α; we can find the points where these derivatives are all equal to zero; and the solutions to the constrained optimization problem (if they exist) must be among these stationary points.

In this example the partial derivatives are: $\begin{cases} \frac{\partial}{\partial x} g(x, y, \alpha) = 2x - 3\alpha \\ \frac{\partial}{\partial y} g(x, y, \alpha) = 2 - 2\alpha \\ \frac{\partial}{\partial \alpha} g(x, y, \alpha) = -3x - 2y - 1 \end{cases}$

When all these partial derivatives are equal to 0, we find that $2\hat{x} - 3\hat{\alpha} = 2 - 2\hat{\alpha} = -3\hat{x} - 2\hat{y} - 1 = 0$, from which we can easily find that $\hat{x} = \frac{3}{2}$, $\hat{y} = -\frac{11}{4}$, and $\hat{\alpha} = 1$. This is the only stationary point, and as it respects the constraint, it must be the solution to the constrained optimization problem.

However, this method applies only to equality constraints. Fortunately, under some regularity conditions (which are respected by the SVM objectives), this method can be generalized to *inequality constraints* as well (e.g., $3x + 2y + 1 \geq 0$). The *generalized Lagrangian* for the hard margin problem is given by Equation C-1, where the $\alpha^{(i)}$ variables are called the *Karush–Kuhn–Tucker* (KKT) multipliers, and they must be greater or equal to zero.

Equation C-1. Generalized Lagrangian for the hard margin problem

$$\mathscr{L}(\mathbf{w}, b, \alpha) = \frac{1}{2}\mathbf{w}^T \cdot \mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)}\left(t^{(i)}\left(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b\right) - 1\right)$$
$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

Just like with the Lagrange multipliers method, you can compute the partial derivatives and locate the stationary points. If there is a solution, it will necessarily be among the stationary points $\left(\hat{\mathbf{w}}, \hat{b}, \hat{\alpha}\right)$ that respect the *KKT conditions*:

- Respect the problem's constraints: $t^{(i)}\left((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}\right) \geq 1 \quad \text{for } i = 1, 2, \cdots, m,$
- Verify $\hat{\alpha}^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m,$
- Either $\hat{\alpha}^{(i)} = 0$ or the i^th^ constraint must be an *active constraint*, meaning it must hold by equality: $t^{(i)}\left((\hat{\mathbf{w}})^T \cdot \mathbf{x}^{(i)} + \hat{b}\right) = 1$. This condition is called the *complementary slackness* condition. It implies that either $\hat{\alpha}^{(i)} = 0$ or the i^th^ instance lies on the boundary (it is a support vector).

Note that the KKT conditions are necessary conditions for a stationary point to be a solution of the constrained optimization problem. Under some conditions, they are also sufficient conditions. Luckily, the SVM optimization problem happens to meet these conditions, so any stationary point that meets the KKT conditions is guaranteed to be a solution to the constrained optimization problem.

We can compute the partial derivatives of the generalized Lagrangian with regards to **w** and b with Equation C-2.

Equation C-2. Partial derivatives of the generalized Lagrangian

$$\nabla_{\mathbf{w}}\mathscr{L}(\mathbf{w}, b, \alpha) = \mathbf{w} - \sum_{i=1}^{m} \alpha^{(i)} t^{(i)} \mathbf{x}^{(i)}$$
$$\frac{\partial}{\partial b}\mathscr{L}(\mathbf{w}, b, \alpha) = -\sum_{i=1}^{m} \alpha^{(i)} t^{(i)}$$

When these partial derivatives are equal to 0, we have Equation C-3.

Equation C-3. Properties of the stationary points

$$\widehat{\mathbf{w}} = \sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} \mathbf{x}^{(i)}$$

$$\sum_{i=1}^{m} \hat{\alpha}^{(i)} t^{(i)} = 0$$

If we plug these results into the definition of the generalized Lagrangian, some terms disappear and we find Equation C-4.

Equation C-4. Dual form of the SVM problem

$$\mathscr{L}\left(\widehat{\mathbf{w}}, \hat{b}, \alpha\right) = \frac{1}{2} \sum_{i=1}^{m} \sum_{j=1}^{m} \alpha^{(i)} \alpha^{(j)} t^{(i)} t^{(j)} \mathbf{x}^{(i)T} \cdot \mathbf{x}^{(j)} \quad - \quad \sum_{i=1}^{m} \alpha^{(i)}$$

$$\text{with} \quad \alpha^{(i)} \geq 0 \quad \text{for } i = 1, 2, \cdots, m$$

The goal is now to find the vector $\hat{\alpha}$ that minimizes this function, with $\hat{\alpha}^{(i)} \geq 0$ for all instances. This constrained optimization problem is the dual problem we were looking for.

Once you find the optimal $\hat{\alpha}$, you can compute $\widehat{\mathbf{w}}$ using the first line of Equation C-3. To compute $\hat{b}$, you can use the fact that a support vector verifies $t^{(i)}(\mathbf{w}^T \cdot \mathbf{x}^{(i)} + b) = 1$, so if the k^th instance is a support vector (i.e., $\alpha_k > 0$), you can use it to compute $\hat{b} = 1 - t^{(k)}\left(\widehat{\mathbf{w}}^T \cdot \mathbf{x}^{(k)}\right)$. However, it is often prefered to compute the average over all support vectors to get a more stable and precise value, as in Equation C-5.

Equation C-5. Bias term estimation using the dual form

$$\hat{b} = \frac{1}{n_s} \sum_{\substack{i=1 \\ \hat{\alpha}^{(i)} > 0}}^{m} \left[1 - t^{(i)}\left(\widehat{\mathbf{w}}^T \cdot \mathbf{x}^{(i)}\right)\right]$$

APPENDIX D
Autodiff

This appendix explains how TensorFlow's autodiff feature works, and how it compares to other solutions.

Suppose you define a function $f(x,y) = x^2y + y + 2$, and you need its partial derivatives $\frac{\partial f}{\partial x}$ and $\frac{\partial f}{\partial y}$, typically to perform Gradient Descent (or some other optimization algorithm). Your main options are manual differentiation, symbolic differentiation, numerical differentiation, forward-mode autodiff, and finally reverse-mode autodiff. TensorFlow implements this last option. Let's go through each of these options.

Manual Differentiation

The first approach is to pick up a pencil and a piece of paper and use your calculus knowledge to derive the partial derivatives manually. For the function $f(x,y)$ just defined, it is not too hard; you just need to use five rules:

- The derivative of a constant is 0.
- The derivative of λx is λ (where λ is a constant).
- The derivative of x^λ is $\lambda x^{\lambda-1}$, so the derivative of x^2 is $2x$.
- The derivative of a sum of functions is the sum of these functions' derivatives.
- The derivative of λ times a function is λ times its derivative.

From these rules, you can derive Equation D-1:

Equation D-1. Partial derivatives of f(x,y)

$$\frac{\partial f}{\partial x} = \frac{\partial(x^2 y)}{\partial x} + \frac{\partial y}{\partial x} + \frac{\partial 2}{\partial x} = y\frac{\partial(x^2)}{\partial x} + 0 + 0 = 2xy$$

$$\frac{\partial f}{\partial y} = \frac{\partial(x^2 y)}{\partial y} + \frac{\partial y}{\partial y} + \frac{\partial 2}{\partial y} = x^2 + 1 + 0 = x^2 + 1$$

This approach can become very tedious for more complex functions, and you run the risk of making mistakes. The good news is that deriving the mathematical equations for the partial derivatives like we just did can be automated, through a process called *symbolic differentiation*.

Symbolic Differentiation

Figure D-1 shows how symbolic differentiation works on an even simpler function, $g(x,y) = 5 + xy$. The graph for that function is represented on the left. After symbolic differentiation, we get the graph on the right, which represents the partial derivative $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1) = y$ (we could similarly obtain the partial derivative with regards to y).

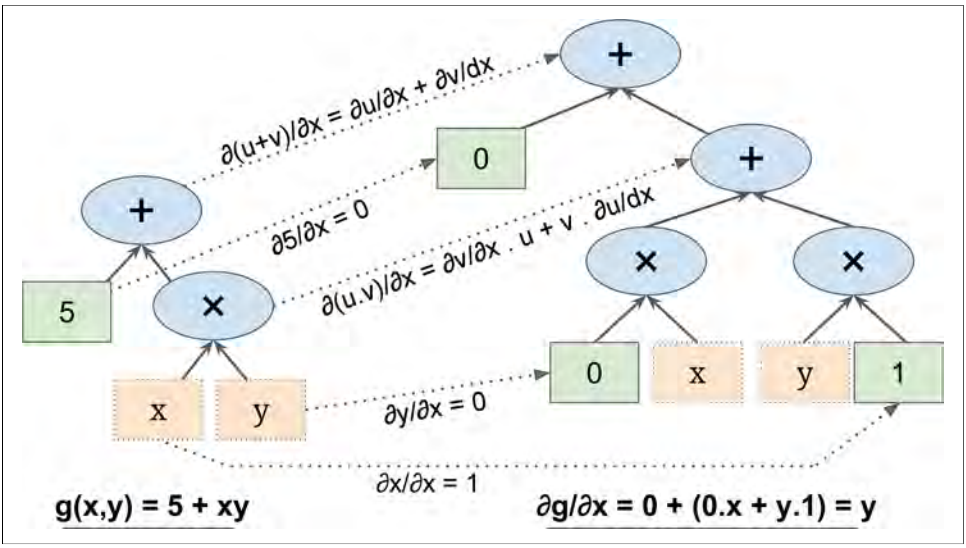


Figure D-1. Symbolic differentiation

The algorithm starts by getting the partial derivative of the leaf nodes. The constant node (5) returns the constant 0, since the derivative of a constant is always 0. The variable x returns the constant 1 since $\frac{\partial x}{\partial x} = 1$, and the variable y returns the constant 0 since $\frac{\partial y}{\partial x} = 0$ (if we were looking for the partial derivative with regards to y, it would be the reverse).

Now we have all we need to move up the graph to the multiplication node in function g. Calculus tells us that the derivative of the product of two functions u and v is $\frac{\partial(u \times v)}{\partial x} = \frac{\partial v}{\partial x} \times u + \frac{\partial u}{\partial x} \times u$. We can therefore construct a large part of the graph on the right, representing $0 \times x + y \times 1$.

Finally, we can go up to the addition node in function g. As mentioned, the derivative of a sum of functions is the sum of these functions' derivatives. So we just need to create an addition node and connect it to the parts of the graph we have already computed. We get the correct partial derivative: $\frac{\partial g}{\partial x} = 0 + (0 \times x + y \times 1)$.

However, it can be simplified (a lot). A few trivial pruning steps can be applied to this graph to get rid of all unnecessary operations, and we get a much smaller graph with just one node: $\frac{\partial g}{\partial x} = y$.

In this case, simplification is fairly easy, but for a more complex function, symbolic differentiation can produce a huge graph that may be tough to simplify and lead to suboptimal performance. Most importantly, symbolic differentiation cannot deal with functions defined with arbitrary code—for example, the following function discussed in Chapter 9:

```python
def my_func(a, b):
    z = 0
    for i in range(100):
        z = a * np.cos(z + i) + z * np.sin(b - i)
    return z
```

Numerical Differentiation

The simplest solution is to compute an approximation of the derivatives, numerically. Recall that the derivative $h'(x_0)$ of a function $h(x)$ at a point x_0 is the slope of the function at that point, or more precisely Equation D-2.

Equation D-2. Derivative of a function $h(x)$ at point x_0

$$\begin{aligned} h'(x) &= \lim_{x \to x_0} \frac{h(x) - h\left(x_0\right)}{x - x_0} \\ &= \lim_{\epsilon \to 0} \frac{h\left(x_0 + \epsilon\right) - h\left(x_0\right)}{\epsilon} \end{aligned}$$

So if we want to calculate the partial derivative of $f(x,y)$ with regards to x, at $x = 3$ and $y = 4$, we can simply compute $f(3 + \epsilon, 4) - f(3, 4)$ and divide the result by ϵ, using a very small value for ϵ. That's exactly what the following code does:

```
def f(x, y):
    return x**2*y + y + 2

def derivative(f, x, y, x_eps, y_eps):
    return (f(x + x_eps, y + y_eps) - f(x, y)) / (x_eps + y_eps)

df_dx = derivative(f, 3, 4, 0.00001, 0)
df_dy = derivative(f, 3, 4, 0, 0.00001)
```

Unfortunately, the result is imprecise (and it gets worse for more complex functions). The correct results are respectively 24 and 10, but instead we get:

```
>>> print(df_dx)
24.000039999805264
>>> print(df_dy)
10.000000000331966
```

Notice that to compute both partial derivatives, we have to call `f()` at least three times (we called it four times in the preceding code, but it could be optimized). If there were 1,000 parameters, we would need to call `f()` at least 1,001 times. When you are dealing with large neural networks, this makes numerical differentiation way too inefficient.

However, numerical differentiation is so simple to implement that it is a great tool to check that the other methods are implemented correctly. For example, if it disagrees with your manually derived function, then your function probably contains a mistake.

Forward-Mode Autodiff

Forward-mode autodiff is neither numerical differentiation nor symbolic differentiation, but in some ways it is their love child. It relies on *dual numbers*, which are (weird but fascinating) numbers of the form $a + b\epsilon$ where a and b are real numbers and ϵ is an infinitesimal number such that $\epsilon^2 = 0$ (but $\epsilon \neq 0$). You can think of the dual number $42 + 24\epsilon$ as something akin to $42.0000\cdots000024$ with an infinite number of 0s (but of course this is simplified just to give you some idea of what dual numbers are). A dual number is represented in memory as a pair of floats. For example, $42 + 24\epsilon$ is represented by the pair (42.0, 24.0).

Dual numbers can be added, multiplied, and so on, as shown in Equation D-3.

Equation D-3. A few operations with dual numbers

$$\lambda(a + b\epsilon) = \lambda a + \lambda b\epsilon$$

$$(a + b\epsilon) + (c + d\epsilon) = (a + c) + (b + d)\epsilon$$

$$(a + b\epsilon) \times (c + d\epsilon) = ac + (ad + bc)\epsilon + (bd)\epsilon^2 = ac + (ad + bc)\epsilon$$

Most importantly, it can be shown that $h(a + b\epsilon) = h(a) + b \times h'(a)\epsilon$, so computing $h(a + \epsilon)$ gives you both $h(a)$ and the derivative $h'(a)$ in just one shot. Figure D-2 shows how forward-mode autodiff computes the partial derivative of $f(x,y)$ with regards to x at $x = 3$ and $y = 4$. All we need to do is compute $f(3 + \epsilon, 4)$; this will output a dual number whose first component is equal to $f(3, 4)$ and whose second component is equal to $\frac{\partial f}{\partial x}(3, 4)$.

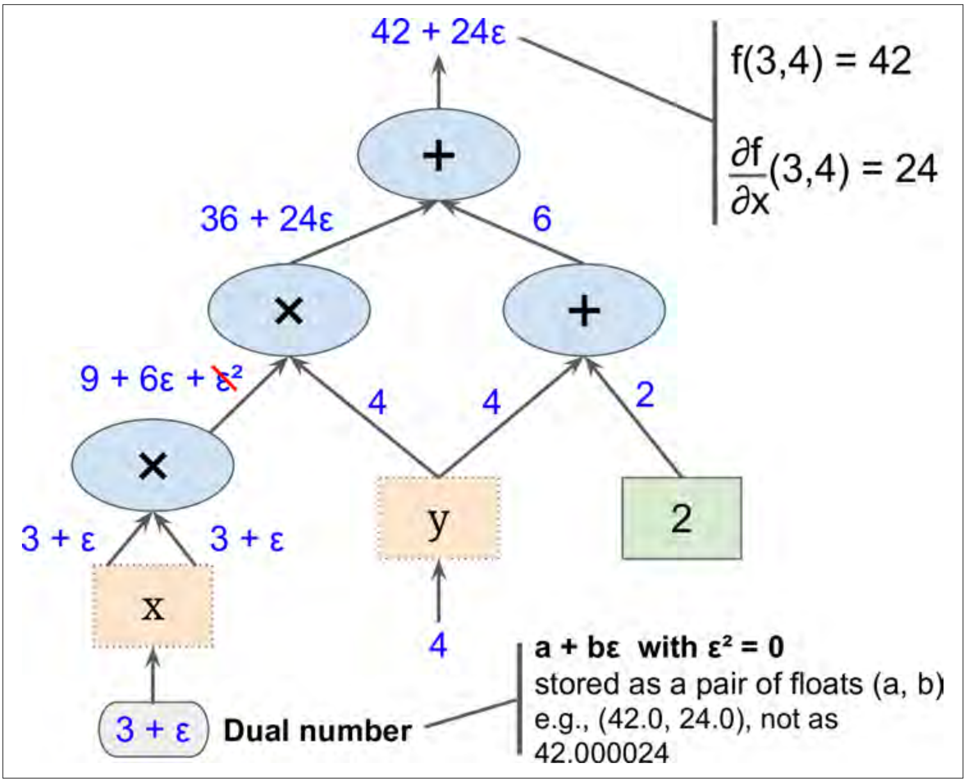


Figure D-2. Forward-mode autodiff

To compute $\frac{\partial f}{\partial y}(3, 4)$ we would have to go through the graph again, but this time with $x = 3$ and $y = 4 + \epsilon$.

So forward-mode autodiff is much more accurate than numerical differentiation, but it suffers from the same major flaw: if there were 1,000 parameters, it would require 1,000 passes through the graph to compute all the partial derivatives. This is where reverse-mode autodiff shines: it can compute all of them in just two passes through the graph.

Reverse-Mode Autodiff

Reverse-mode autodiff is the solution implemented by TensorFlow. It first goes through the graph in the forward direction (i.e., from the inputs to the output) to compute the value of each node. Then it does a second pass, this time in the reverse direction (i.e., from the output to the inputs) to compute all the partial derivatives. Figure D-3 represents the second pass. During the first pass, all the node values were computed, starting from $x = 3$ and $y = 4$. You can see those values at the bottom right of each node (e.g., $x \times x = 9$). The nodes are labeled n_1 to n_7 for clarity. The output node is n_7: $f(3,4) = n_7 = 42$.

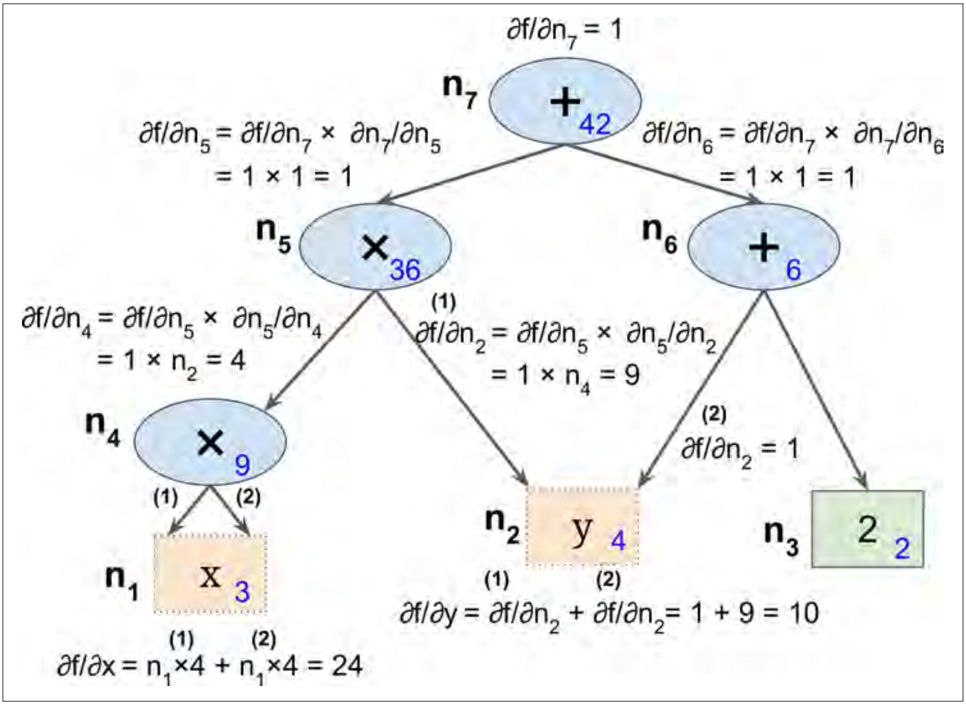


Figure D-3. Reverse-mode autodiff

The idea is to gradually go down the graph, computing the partial derivative of $f(x,y)$ with regards to each consecutive node, until we reach the variable nodes. For this, reverse-mode autodiff relies heavily on the *chain rule*, shown in Equation D-4.

Equation D-4. Chain rule

$$\frac{\partial f}{\partial x} = \frac{\partial f}{\partial n_i} \times \frac{\partial n_i}{\partial x}$$

Since n_7 is the output node, $f = n_7$ so trivially $\frac{\partial f}{\partial n_7} = 1$.

Let's continue down the graph to n_5: how much does f vary when n_5 varies? The answer is $\frac{\partial f}{\partial n_5} = \frac{\partial f}{\partial n_7} \times \frac{\partial n_7}{\partial n_5}$. We already know that $\frac{\partial f}{\partial n_7} = 1$, so all we need is $\frac{\partial n_7}{\partial n_5}$. Since n_7 simply performs the sum $n_5 + n_6$, we find that $\frac{\partial n_7}{\partial n_5} = 1$, so $\frac{\partial f}{\partial n_5} = 1 \times 1 = 1$.

Now we can proceed to node n_4: how much does f vary when n_4 varies? The answer is $\frac{\partial f}{\partial n_4} = \frac{\partial f}{\partial n_5} \times \frac{\partial n_5}{\partial n_4}$. Since $n_5 = n_4 \times n_2$, we find that $\frac{\partial n_5}{\partial n_4} = n_2$, so $\frac{\partial f}{\partial n_4} = 1 \times n_2 = 4$.

The process continues until we reach the bottom of the graph. At that point we will have calculated all the partial derivatives of $f(x,y)$ at the point $x = 3$ and $y = 4$. In this example, we find $\frac{\partial f}{\partial x} = 24$ and $\frac{\partial f}{\partial y} = 10$. Sounds about right!

Reverse-mode autodiff is a very powerful and accurate technique, especially when there are many inputs and few outputs, since it requires only one forward pass plus one reverse pass per output to compute all the partial derivatives for all outputs with regards to all the inputs. Most importantly, it can deal with functions defined by arbitrary code. It can also handle functions that are not entirely differentiable, as long as you ask it to compute the partial derivatives at points that are differentiable.

> If you implement a new type of operation in TensorFlow and you want to make it compatible with autodiff, then you need to provide a function that builds a subgraph to compute its partial derivatives with regards to its inputs. For example, suppose you implement a function that computes the square of its input $f(x) = x^2$. In that case you would need to provide the corresponding derivative function $f'(x) = 2x$. Note that this function does not compute a numerical result, but instead builds a subgraph that will (later) compute the result. This is very useful because it means that you can compute gradients of gradients (to compute second-order derivatives, or even higher-order derivatives).

APPENDIX E

Other Popular ANN Architectures

In this appendix we will give a quick overview of a few historically important neural network architectures that are much less used today than deep Multi-Layer Perceptrons (Chapter 10), convolutional neural networks (Chapter 13), recurrent neural networks (Chapter 14), or autoencoders (Chapter 15). They are often mentioned in the literature, and some are still used in many applications, so it is worth knowing about them. Moreover, we will discuss *deep belief nets* (DBNs), which were the state of the art in Deep Learning until the early 2010s. They are still the subject of very active research, so they may well come back with a vengeance in the near future.

Hopfield Networks

Hopfield networks were first introduced by W. A. Little in 1974, then popularized by J. Hopfield in 1982. They are *associative memory* networks: you first teach them some patterns, and then when they see a new pattern they (hopefully) output the closest learned pattern. This has made them useful in particular for character recognition before they were outperformed by other approaches. You first train the network by showing it examples of character images (each binary pixel maps to one neuron), and then when you show it a new character image, after a few iterations it outputs the closest learned character.

They are fully connected graphs (see Figure E-1); that is, every neuron is connected to every other neuron. Note that on the diagram the images are 6 × 6 pixels, so the neural network on the left should contain 36 neurons (and 648 connections), but for visual clarity a much smaller network is represented.

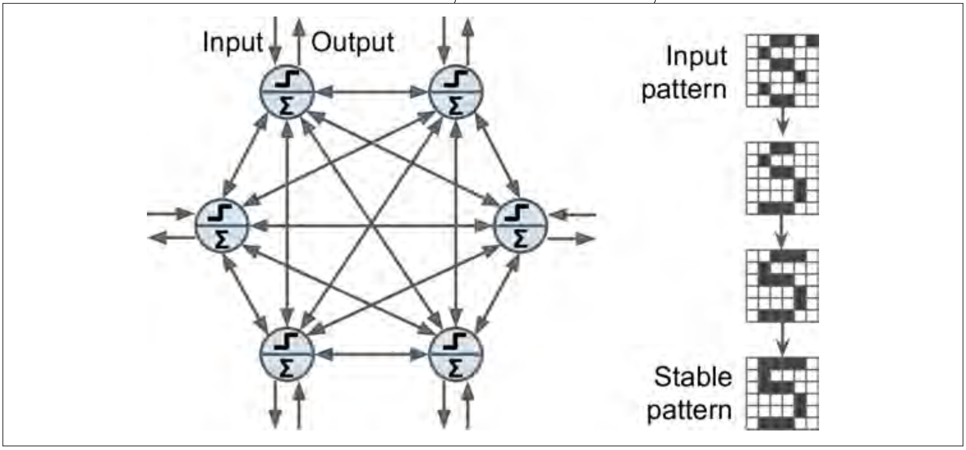


Figure E-1. Hopfield network

The training algorithm works by using Hebb's rule: for each training image, the weight between two neurons is increased if the corresponding pixels are both on or both off, but decreased if one pixel is on and the other is off.

To show a new image to the network, you just activate the neurons that correspond to active pixels. The network then computes the output of every neuron, and this gives you a new image. You can then take this new image and repeat the whole process. After a while, the network reaches a stable state. Generally, this corresponds to the training image that most resembles the input image.

A so-called *energy function* is associated with Hopfield nets. At each iteration, the energy decreases, so the network is guaranteed to eventually stabilize to a low-energy state. The training algorithm tweaks the weights in a way that decreases the energy level of the training patterns, so the network is likely to stabilize in one of these low-energy configurations. Unfortunately, some patterns that were not in the training set also end up with low energy, so the network sometimes stabilizes in a configuration that was not learned. These are called *spurious patterns*.

Another major flaw with Hopfield nets is that they don't scale very well—their memory capacity is roughly equal to 14% of the number of neurons. For example, to classify 28 × 28 images, you would need a Hopfield net with 784 fully connected neurons and 306,936 weights. Such a network would only be able to learn about 110 different characters (14% of 784). That's a lot of parameters for such a small memory.

Boltzmann Machines

Boltzmann machines were invented in 1985 by Geoffrey Hinton and Terrence Sejnowski. Just like Hopfield nets, they are fully connected ANNs, but they are based on *sto-*

chastic neurons: instead of using a deterministic step function to decide what value to output, these neurons output 1 with some probability, and 0 otherwise. The probability function that these ANNs use is based on the Boltzmann distribution (used in statistical mechanics) hence their name. Equation E-1 gives the probability that a particular neuron will output a 1.

Equation E-1. Probability that the i^th^ neuron will output 1

$$p\left(s_i^{(\text{next step})} = 1\right) = \sigma\left(\frac{\sum_{j=1}^{N} w_{i,j} s_j + b_i}{T}\right)$$

- s_j is the j$^{\text{th}}$ neuron's state (0 or 1).
- $w_{i,j}$ is the connection weight between the i$^{\text{th}}$ and j$^{\text{th}}$ neurons. Note that $w_{i,i} = 0$.
- b_i is the i$^{\text{th}}$ neuron's bias term. We can implement this term by adding a bias neuron to the network.
- N is the number of neurons in the network.
- T is a number called the network's *temperature*; the higher the temperature, the more random the output is (i.e., the more the probability approaches 50%).
- σ is the logistic function.

Neurons in Boltzmann machines are separated into two groups: *visible units* and *hidden units* (see Figure E-2). All neurons work in the same stochastic way, but the visible units are the ones that receive the inputs and from which outputs are read.

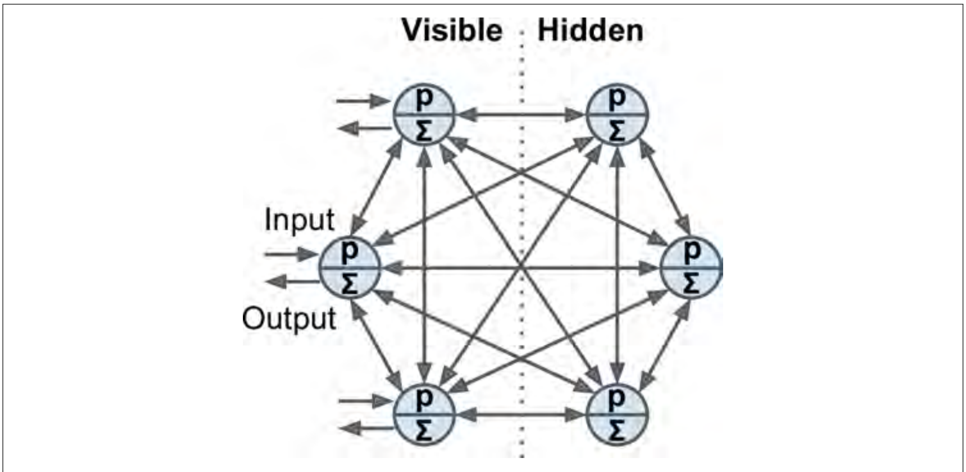


Figure E-2. Boltzmann machine

Because of its stochastic nature, a Boltzmann machine will never stabilize into a fixed configuration, but instead it will keep switching between many configurations. If it is left running for a sufficiently long time, the probability of observing a particular configuration will only be a function of the connection weights and bias terms, not of the original configuration (similarly, after you shuffle a deck of cards for long enough, the configuration of the deck does not depend on the initial state). When the network reaches this state where the original configuration is "forgotten," it is said to be in *thermal equilibrium* (although its configuration keeps changing all the time). By setting the network parameters appropriately, letting the network reach thermal equilibrium, and then observing its state, we can simulate a wide range of probability distributions. This is called a *generative model*.

Training a Boltzmann machine means finding the parameters that will make the network approximate the training set's probability distribution. For example, if there are three visible neurons and the training set contains 75% (0, 1, 1) triplets, 10% (0, 0, 1) triplets, and 15% (1, 1, 1) triplets, then after training a Boltzmann machine, you could use it to generate random binary triplets with about the same probability distribution. For example, about 75% of the time it would output the (0, 1, 1) triplet.

Such a generative model can be used in a variety of ways. For example, if it is trained on images, and you provide an incomplete or noisy image to the network, it will automatically "repair" the image in a reasonable way. You can also use a generative model for classification. Just add a few visible neurons to encode the training image's class (e.g., add 10 visible neurons and turn on only the fifth neuron when the training image represents a 5). Then, when given a new image, the network will automatically turn on the appropriate visible neurons, indicating the image's class (e.g., it will turn on the fifth visible neuron if the image represents a 5).

Unfortunately, there is no efficient technique to train Boltzmann machines. However, fairly efficient algorithms have been developed to train *restricted Boltzmann machines* (RBM).

Restricted Boltzmann Machines

An RBM is simply a Boltzmann machine in which there are no connections between visible units or between hidden units, only between visible and hidden units. For example, Figure E-3 represents an RBM with three visible units and four hidden units.

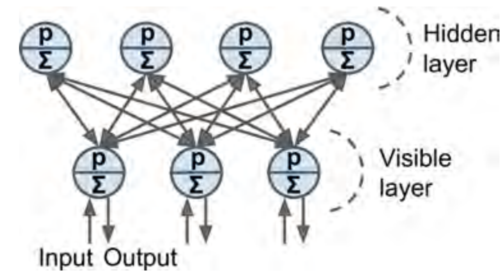


Figure E-3. Restricted Boltzmann machine

A very efficient training algorithm, called *Contrastive Divergence*, was introduced in 2005 by Miguel Á. Carreira-Perpiñán and Geoffrey Hinton.[1] Here is how it works: for each training instance **x**, the algorithm starts by feeding it to the network by setting the state of the visible units to $x_1, x_2, \cdots, x_n$. Then you compute the state of the hidden units by applying the stochastic equation described before (Equation E-1). This gives you a hidden vector **h** (where h_i is equal to the state of the i[th] unit). Next you compute the state of the visible units, by applying the same stochastic equation. This gives you a vector $\dot{\mathbf{x}}$. Then once again you compute the state of the hidden units, which gives you a vector $\dot{\mathbf{h}}$. Now you can update each connection weight by applying the rule in Equation E-2.

Equation E-2. Contrastive divergence weight update

$$w_{i,j}^{(\text{next step})} = w_{i,j} + \eta\left(\mathbf{x}\mathbf{h}^T - \dot{\mathbf{x}}\dot{\mathbf{h}}^T\right)$$

The great benefit of this algorithm it that it does not require waiting for the network to reach thermal equilibrium: it just goes forward, backward, and forward again, and that's it. This makes it incomparably more efficient than previous algorithms, and it was a key ingredient to the first success of Deep Learning based on multiple stacked RBMs.

Deep Belief Nets

Several layers of RBMs can be stacked; the hidden units of the first-level RBM serves as the visible units for the second-layer RBM, and so on. Such an RBM stack is called a *deep belief net* (DBN).

Yee-Whye Teh, one of Geoffrey Hinton's students, observed that it was possible to train DBNs one layer at a time using Contrastive Divergence, starting with the lower

1 "On Contrastive Divergence Learning," M. Á. Carreira-Perpiñán and G. Hinton (2005).

layers and then gradually moving up to the top layers. This led to the groundbreaking article that kickstarted the Deep Learning tsunami in 2006.[2]

Just like RBMs, DBNs learn to reproduce the probability distribution of their inputs, without any supervision. However, they are much better at it, for the same reason that deep neural networks are more powerful than shallow ones: real-world data is often organized in hierarchical patterns, and DBNs take advantage of that. Their lower layers learn low-level features in the input data, while higher layers learn high-level features.

Just like RBMs, DBNs are fundamentally unsupervised, but you can also train them in a supervised manner by adding some visible units to represent the labels. Moreover, one great feature of DBNs is that they can be trained in a semisupervised fashion. Figure E-4 represents such a DBN configured for semisupervised learning.

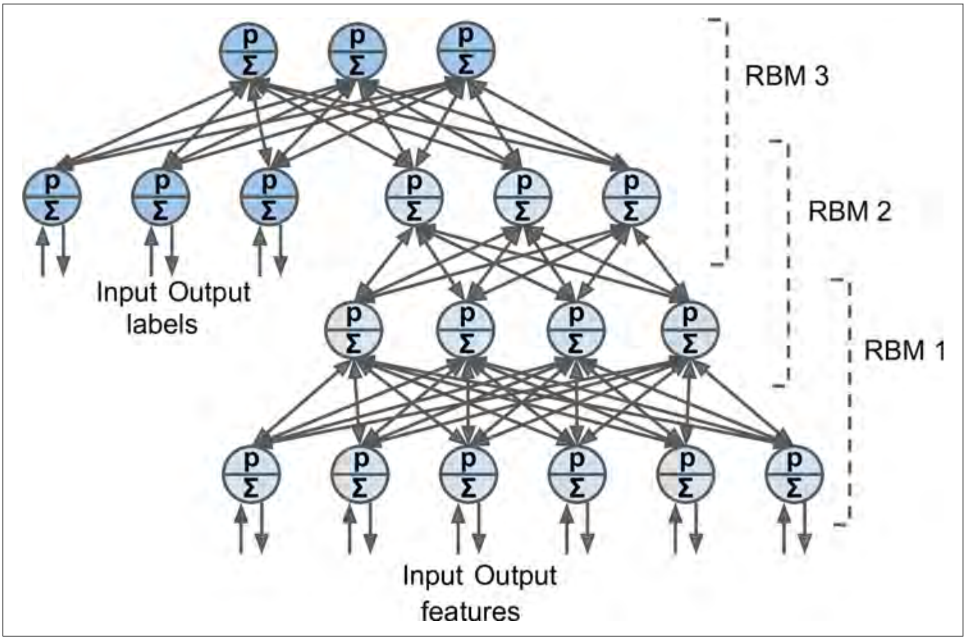


Figure E-4. A deep belief network configured for semisupervised learning

First, the RBM 1 is trained without supervision. It learns low-level features in the training data. Then RBM 2 is trained with RBM 1's hidden units as inputs, again without supervision: it learns higher-level features (note that RBM 2's hidden units include only the three rightmost units, not the label units). Several more RBMs could be stacked this way, but you get the idea. So far, training was 100% unsupervised.

2 "A Fast Learning Algorithm for Deep Belief Nets," G. Hinton, S. Osindero, Y. Teh (2006).

Lastly, RBM 3 is trained using both RBM 2's hidden units as inputs, as well as extra visible units used to represent the target labels (e.g., a one-hot vector representing the instance class). It learns to associate high-level features with training labels. This is the supervised step.

At the end of training, if you feed RBM 1 a new instance, the signal will propagate up to RBM 2, then up to the top of RBM 3, and then back down to the label units; hopefully, the appropriate label will light up. This is how a DBN can be used for classification.

One great benefit of this semisupervised approach is that you don't need much labeled training data. If the unsupervised RBMs do a good enough job, then only a small amount of labeled training instances per class will be necessary. Similarly, a baby learns to recognize objects without supervision, so when you point to a chair and say "chair," the baby can associate the word "chair" with the class of objects it has already learned to recognize on its own. You don't need to point to every single chair and say "chair"; only a few examples will suffice (just enough so the baby can be sure that you are indeed referring to the chair, not to its color or one of the chair's parts).

Quite amazingly, DBNs can also work in reverse. If you activate one of the label units, the signal will propagate up to the hidden units of RBM 3, then down to RBM 2, and then RBM 1, and a new instance will be output by the visible units of RBM 1. This new instance will usually look like a regular instance of the class whose label unit you activated. This generative capability of DBNs is quite powerful. For example, it has been used to automatically generate captions for images, and vice versa: first a DBN is trained (without supervision) to learn features in images, and another DBN is trained (again without supervision) to learn features in sets of captions (e.g., "car" often comes with "automobile"). Then an RBM is stacked on top of both DBNs and trained with a set of images along with their captions; it learns to associate high-level features in images with high-level features in captions. Next, if you feed the image DBN an image of a car, the signal will propagate through the network, up to the top-level RBM, and back down to the bottom of the caption DBN, producing a caption. Due to the stochastic nature of RBMs and DBNs, the caption will keep changing randomly, but it will generally be appropriate for the image. If you generate a few hundred captions, the most frequently generated ones will likely be a good description of the image.[3]

Self-Organizing Maps

Self-organizing maps (SOM) are quite different from all the other types of neural networks we have discussed so far. They are used to produce a low-dimensional repre-

3 See this video by Geoffrey Hinton for more details and a demo: *http://goo.gl/7Z5QiS*.

sentation of a high-dimensional dataset, generally for visualization, clustering, or classification. The neurons are spread across a map (typically 2D for visualization, but it can be any number of dimensions you want), as shown in Figure E-5, and each neuron has a weighted connection to every input (note that the diagram shows just two inputs, but there are typically a very large number, since the whole point of SOMs is to reduce dimensionality).

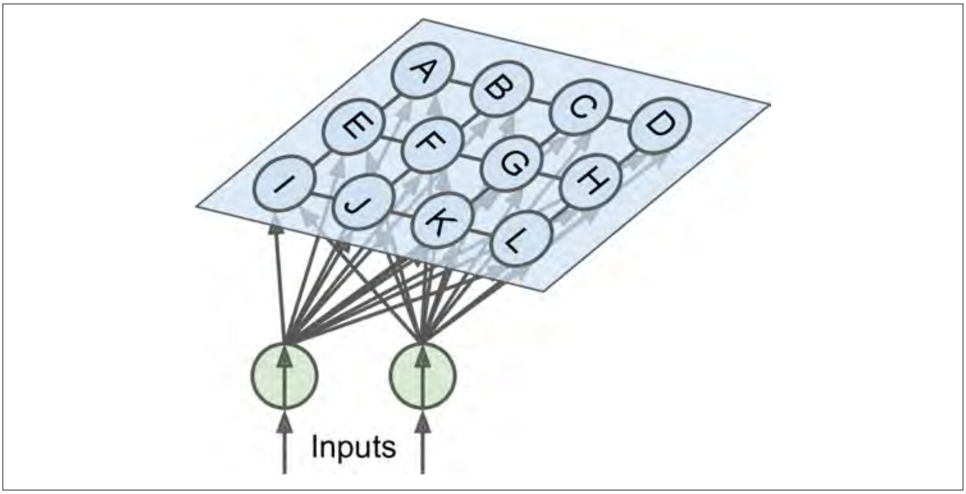


Figure E-5. Self-organizing maps

Once the network is trained, you can feed it a new instance and this will activate only one neuron (i.e., hence one point on the map): the neuron whose weight vector is closest to the input vector. In general, instances that are nearby in the original input space will activate neurons that are nearby on the map. This makes SOMs useful for visualization (in particular, you can easily identify clusters on the map), but also for applications like speech recognition. For example, if each instance represents the audio recording of a person pronouncing a vowel, then different pronunciations of the vowel "a" will activate neurons in the same area of the map, while instances of the vowel "e" will activate neurons in another area, and intermediate sounds will generally activate intermediate neurons on the map.

> One important difference with the other dimensionality reduction techniques discussed in Chapter 8 is that all instances get mapped to a discrete number of points in the low-dimensional space (one point per neuron). When there are very few neurons, this technique is better described as clustering rather than dimensionality reduction.

The training algorithm is unsupervised. It works by having all the neurons compete against each other. First, all the weights are initialized randomly. Then a training instance is picked randomly and fed to the network. All neurons compute the distance between their weight vector and the input vector (this is very different from the artificial neurons we have seen so far). The neuron that measures the smallest distance wins and tweaks its weight vector to be even slightly closer to the input vector, making it more likely to win future competitions for other inputs similar to this one. It also recruits its neighboring neurons, and they too update their weight vector to be slightly closer to the input vector (but they don't update their weights as much as the winner neuron). Then the algorithm picks another training instance and repeats the process, again and again. This algorithm tends to make nearby neurons gradually specialize in similar inputs.[4]

4 You can imagine a class of young children with roughly similar skills. One child happens to be slightly better at basketball. This motivates her to practice more, especially with her friends. After a while, this group of friends gets so good at basketball that other kids cannot compete. But that's okay, because the other kids specialize in other topics. After a while, the class is full of little specialized groups.

Index

Symbols

A

D

E

F

G

H

I

J

K

L

M

N

Q

R

S

T

U

V

W

X

Y

Z

About the Author

Aurélien Géron is a Machine Learning consultant. A former Googler, he led the YouTube video classification team from 2013 to 2016. He was also a founder and CTO of Wifirst from 2002 to 2012, a leading Wireless ISP in France; and a founder and CTO of Polyconseil in 2001, the firm that now manages the electric car sharing service Autolib'.

Before this he worked as an engineer in a variety of domains: finance (JP Morgan and Société Générale), defense (Canada's DOD), and healthcare (blood transfusion). He published a few technical books (on C++, WiFi, and internet architectures), and was a Computer Science lecturer in a French engineering school.

A few fun facts: he taught his three children to count in binary with their fingers (up to 1023), he studied microbiology and evolutionary genetics before going into software engineering, and his parachute didn't open on the second jump.

Colophon

The animal on the cover of *Hands-On Machine Learning with Scikit-Learn and TensorFlow* is the far eastern fire salamander (*Salamandra infraimmaculata*), an amphibian found in the Middle East. They have black skin featuring large yellow spots on their back and head. These spots are a warning coloration meant to keep predators at bay. Full-grown salamanders can be over a foot in length.

Far eastern fire salamanders live in subtropical shrubland and forests near rivers or other freshwater bodies. They spend most of their life on land, but lay their eggs in the water. They subsist mostly on a diet of insects, worms, and small crustaceans, but occasionally eat other salamanders. Males of the species have been known to live up to 23 years, while females can live up to 21 years.

Although not yet endangered, the far eastern fire salamander population is in decline. Primary threats include damming of rivers (which disrupts the salamander's breeding) and pollution. They are also threatened by the recent introduction of predatory fish, such as the mosquitofish. These fish were intended to control the mosquito population, but they also feed on young salamanders.

Many of the animals on O'Reilly covers are endangered; all of them are important to the world. To learn more about how you can help, go to *animals.oreilly.com*.

The cover image is from *Wood's Illustrated Natural History*. The cover fonts are URW Typewriter and Guardian Sans. The text font is Adobe Minion Pro; the heading font is Adobe Myriad Condensed; and the code font is Dalton Maag's Ubuntu Mono.