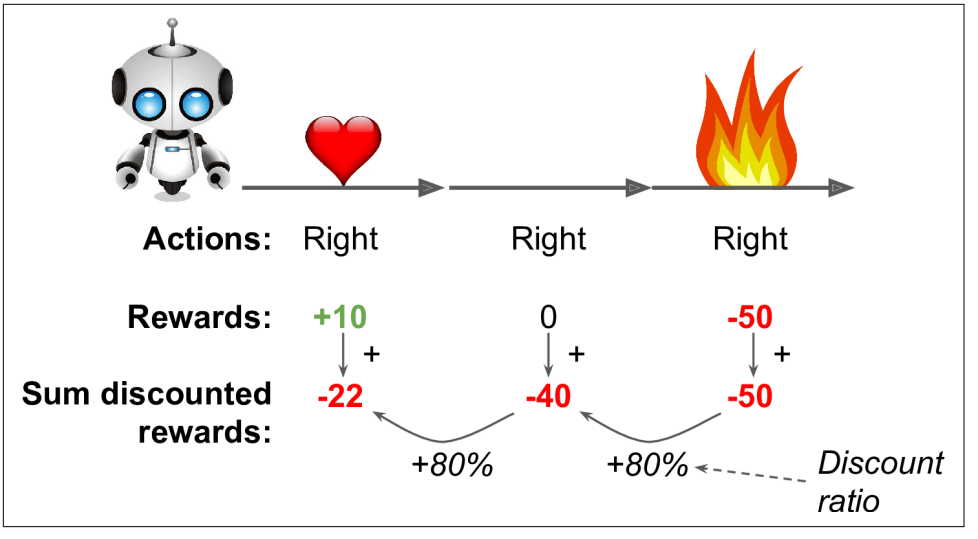


*Figura 16-6. Recompensas con descuento*

Por supuesto, una buena acción puede ir seguida de varias malas acciones que hacen que el poste caiga rápidamente, lo que hace que la buena acción obtenga una puntuación baja (de manera similar, un buen actor a veces puede protagonizar una película terrible). Sin embargo, si jugamos el juego suficientes veces, en promedio, las buenas acciones obtendrán una mejor puntuación que las malas. Por tanto, para obtener puntuaciones de acción bastante fiables, debemos ejecutar muchos episodios y normalizar todas las puntuaciones de acción (restando la media y dividiendo por la desviación estándar). Después de eso, podemos suponer razonablemente que las acciones con una puntuación negativa fueron malas mientras que las acciones con una puntuación positiva fueron buenas. Perfecto: ahora que tenemos una forma de evaluar cada acción, estamos listos para capacitar a nuestro primer agente utilizando gradientes de políticas. Veamos cómo.

## Gradientes de política

Como se discutió anteriormente, los algoritmos PG optimizan los parámetros de una política siguiendo los gradientes hacia recompensas más altas. Una clase popular de algoritmos PG, llamada *REFORZAR algoritmos,* estaba introducido en 1992[9] por Ronald Williams. Aquí hay una variante común:

1. Primero, deje que la política de red neuronal juegue varias veces y en cada paso calcule los gradientes que harían que la acción elegida sea aún más probable, pero no aplique estos gradientes todavía.

9 "Algoritmos estadísticos simples de seguimiento de gradientes para el aprendizaje por refuerzo conexionista", R. Williams (1992).

2. Una vez que haya ejecutado varios episodios, calcule la puntuación de cada acción (utilizando el método descrito en el párrafo anterior).
3. Si el puntaje de una acción es positivo, significa que la acción fue buena y desea aplicar los gradientes calculados anteriormente para que la acción sea aún más probable que se elija en el futuro. Sin embargo, si la puntuación es negativa, significa que la acción fue mala y desea aplicar los gradientes opuestos para hacer que esta acción sea leve. *Menos*
   probablemente en el futuro. La solución es simplemente multiplicar cada vector de gradiente por la puntuación de la acción correspondiente.
4. Finalmente, calcule la media de todos los vectores de gradiente resultantes y utilícela para realizar un paso de descenso de gradiente.

Implementemos este algoritmo usando TensorFlow. Entrenaremos la política de red neuronal que construimos anteriormente para que aprenda a equilibrar el poste en el carro. Comencemos por completar la fase de construcción que codificamos anteriormente para agregar la probabilidad objetivo, la función de costo y la operación de capacitación. Dado que estamos actuando como si la acción elegida fuera la mejor acción posible, la probabilidad objetivo debe ser 1.0 si la acción elegida es la acción 0 (izquierda) y 0.0 si es la acción 1 (derecha):

```
y = 1. - tf . flotar ( acción )
```

Ahora que tenemos una probabilidad objetivo, podemos definir la función de costo (entropía cruzada) y calcular los gradientes:

```
tasa de aprendizaje = 0,01

cross_entropy = tf . nn . sigmoid_cross_entropy_with_logits (
                                    etiquetas = y , logits = logits )
optimizador = tf . tren . AdamOptimizer ( tasa de aprendizaje )
grads_and_vars = optimizador . compute_gradients ( cross_entropy )
```

Tenga en cuenta que estamos llamando al optimizador compute_gradients () método en lugar del minimizar() método. Esto se debe a que queremos modificar los degradados antes de aplicarlos. [10] los compute_gradients () El método devuelve una lista de pares de variable / vector de gradiente (un par por variable entrenable). Pongamos todos los degradados en una lista, para que sea más conveniente obtener sus valores:

```
gradientes = [ graduado para graduado , variable en grads_and_vars ]
```

Bien, ahora viene la parte complicada. Durante la fase de ejecución, el algoritmo ejecutará la política y en cada paso evaluará estos tensores de gradiente y almacenará sus valores. Después de varios episodios, ajustará estos gradientes como se explicó anteriormente (es decir, los multiplicará por las puntuaciones de acción y los normalizará) y calculará la media de los gradientes ajustados. A continuación, deberá retroalimentar los degradados resultantes al

10 Ya hicimos algo similar en Capítulo 11 cuando hablamos del recorte de degradado: primero calculamos el degradados, luego los recortamos y finalmente aplicamos los degradados recortados.

optimizador para que pueda realizar un paso de optimización. Esto significa que necesitamos un marcador de posición por vector de gradiente. Además, debemos crear la operación que aplicará los gradientes actualizados. Para esto llamaremos al optimizador apply_gradients () función, que toma una lista de pares de vectores / variables de gradiente. En lugar de darle los vectores de degradado originales, le daremos una lista que contiene los degradados actualizados (es decir, los que se alimentan a través de los marcadores de posición de degradado):

```
marcadores de posición de gradiente = []
grads_and_vars_feed = []
para graduado , variable en grads_and_vars :
    gradient_placeholder = tf . marcador de posición ( tf . float32 , forma = graduado . get_shape ())
    marcadores de posición de gradiente . adjuntar ( gradient_placeholder )
    grads_and_vars_feed . adjuntar (( gradient_placeholder , variable ))

training_op = optimizador . Apply_gradients ( grads_and_vars_feed )
```

Retrocedamos y echemos un vistazo a la fase de construcción completa:

```
n_inputs = 4
n_hidden = 4
n_salidas = 1
inicializador = tf . contrib . capas . variance_scaling_initializer ()

tasa de aprendizaje = 0,01

X = tf . marcador de posición ( tf . float32 , forma = [ Ninguna , n_inputs ])
oculto = totalmente_conectado ( X , n_hidden , activación_fn = tf . nn . elu ,
                                weights_initializer = inicializador )
logits = totalmente_conectado ( oculto , n_salidas , activación_fn = Ninguna ,
                                weights_initializer = inicializador )
salidas = tf . nn . sigmoideo ( logits )
p_left_and_right = tf . concat ( eje = 1 , valores = [ salidas , 1 - salidas ])
acción = tf . multinomial ( tf . Iniciar sesión ( p_left_and_right ), num_samples = 1 )

y = 1. - tf . flotar ( acción )
cross_entropy = tf . nn . sigmoid_cross_entropy_with_logits (
                            etiquetas = y , logits = logits )
optimizador = tf . tren . AdamOptimizer ( tasa de aprendizaje )
grads_and_vars = optimizador . compute_gradients ( cross_entropy )
gradientes = [ graduado para graduado , variable en grads_and_vars ]
marcadores de posición de gradiente = []
grads_and_vars_feed = []
para graduado , variable en grads_and_vars :
    gradient_placeholder = tf . marcador de posición ( tf . float32 , forma = graduado . get_shape ())
    marcadores de posición de gradiente . adjuntar ( gradient_placeholder )
    grads_and_vars_feed . adjuntar (( gradient_placeholder , variable ))
training_op = optimizador . Apply_gradients ( grads_and_vars_feed )

en eso = tf . global_variables_initializer ()
ahorrador = tf . tren . Ahorrador ()
```

¡A la fase de ejecución! Necesitaremos un par de funciones para calcular el total de recompensas con descuento, dadas las recompensas en bruto, y para normalizar los resultados en múltiples episodios:

```
def Discount_rewards ( recompensas , tasa de descuento ):
    recompensasdescuento = notario público . vacío ( len ( recompensas ))
    recompensas_cumulativas = 0
    para paso en invertido ( rango ( len ( recompensas ))):
        recompensas_cumulativas = recompensas [ paso ] + recompensas_cumulativas * discount_rate descuento_rewards [ paso ] =
        recompensas_cumulativas
    regreso recompensasdescuento

def discount_and_normalize_rewards ( all_rewards , tasa de descuento ):
    all_discounted_rewards = [ Discount_rewards ( recompensas )
                                        para recompensas en all_rewards ]
    flat_rewards = notario público . concatenar ( all_discounted_rewards )
    recompensa_mean = flat_rewards . media ()
    recompensa_std = flat_rewards . std ()
    regreso [( recompensasdescuento - recompensa_mean ) / recompensa_std
                para recompensasdescuento en all_discounted_rewards ]
```

Comprobemos que esto funciona:

```
>>> Discount_rewards ([ 10 , 0 , - 50 ], tasa de descuento = 0,8 )
matriz ([- 22., -40., -50.])
>>> discount_and_normalize_rewards ([[ 10 , 0 , - 50 ], [ 10 , 20 ]], tasa de descuento = 0,8 )
[matriz ([- 0.28435071, -0.86597718, -1.18910299]),
 matriz ([1.26665318, 1.0727777])]
```

La llamada a discount_rewards () devuelve exactamente lo que esperamos (ver Figura 16-6. ). Puede verificar que la función discount_and_normalize_rewards () de hecho, devuelve las puntuaciones normalizadas para cada acción en ambos episodios. Observe que el primer episodio fue mucho peor que el segundo, por lo que sus puntuaciones normalizadas son todas negativas; todas las acciones del primer episodio se considerarían malas y, a la inversa, todas las acciones del segundo episodio se considerarían buenas.

Ahora tenemos todo lo que necesitamos para entrenar la política:

```
n_iteraciones = 250                 # número de iteraciones de entrenamiento
n_max_steps = 1000                  # pasos máximos por episodio
n_games_per_update = 10 # entrenar la política cada 10 episodios
save_iterations = 10                # guardar el modelo cada 10 iteraciones de entrenamiento
tasa de descuento = 0,95

con tf . Sesión () como sess :
    en eso . correr ()
    para iteración en rango ( n_iteraciones ):
        all_rewards = []                    # todas las secuencias de recompensas en bruto para cada episodio
        all_gradients = [] # gradientes guardados en cada paso de cada episodio
        para juego en rango ( n_games_per_update ):
            current_rewards = []                # todas las recompensas brutas del episodio actual
            gradientes_actual = [] # todos los gradientes del episodio actual
```

```
        obs = env . Reiniciar ()
        para paso en rango ( n_max_steps ):
            action_val , gradients_val = sess . correr (
                        [ acción , gradientes ],
                        feed_dict = { X : obs . remodelar ( 1 , n_inputs )}) # una obs
            obs , recompensa , hecho , info = env . paso ( action_val [ 0 ] [ 0 ])
            current_rewards . adjuntar ( recompensa )
            gradientes_actual . adjuntar ( gradients_val )
            Si hecho :
                descanso
        all_rewards . adjuntar ( current_rewards )
        all_gradients . adjuntar ( gradientes_actual )

    # En este punto, hemos ejecutado la política para 10 episodios y estamos
    # listo para una actualización de política utilizando el algoritmo descrito anteriormente.
    all_rewards = discount_and_normalize_rewards ( all_rewards )
    feed_dict = {}
    para var_index , grad_placeholder en enumerar ( marcadores de posición de gradiente ):
        # multiplique los gradientes por las puntuaciones de acción y calcule la media
        mean_gradients = notario público . media (
            [ recompensa * all_gradients [ game_index ] [ paso ] [ var_index ]
                para game_index , recompensas en enumerar ( all_rewards )
                para paso , recompensa en enumerar ( recompensas )],
            eje = 0 )
        feed_dict [ grad_placeholder ] = mean_gradients
    sess . correr ( training_op , feed_dict = feed_dict )
    Si iteración % save_iterations == 0 :
        ahorrador . salvar ( sess , "./my_policy_net_pg.ckpt" )
```

Cada iteración de entrenamiento comienza con la ejecución de la política durante 10 episodios (con un máximo 1,000 pasos por episodio, para evitar correr para siempre). En cada paso, también calculamos los gradientes, fingiendo que la acción elegida fue la mejor. Una vez ejecutados estos 10 episodios, calculamos las puntuaciones de acción mediante el discount_and_normal ize_rewards () función; revisamos cada variable entrenable, en todos los episodios y todos los pasos, para multiplicar cada vector de gradiente por su puntuación de acción correspondiente; y calculamos la media de los gradientes resultantes. Finalmente, ejecutamos la operación de entrenamiento, alimentándola con estos gradientes medios (uno por variable entrenable). También guardamos el modelo cada 10 operaciones de entrenamiento.

¡Y hemos terminado! Este código entrenará la política de la red neuronal y aprenderá con éxito a equilibrar el poste en el carrito (puede probarlo en los cuadernos de Jupyter). Tenga en cuenta que en realidad hay dos formas en que el agente puede perder el juego: o el poste puede inclinarse demasiado o el carro puede salirse completamente de la pantalla. Con 250 iteraciones de entrenamiento, la política aprende a equilibrar el poste bastante bien, pero aún no es lo suficientemente buena para evitar salirse de la pantalla. Unos cientos de iteraciones de entrenamiento más solucionarán eso.

Los investigadores intentan encontrar algoritmos que funcionen bien incluso cuando el agente inicialmente no sabe nada sobre el medio ambiente. Sin embargo, a menos que esté escribiendo un artículo, debe inyectar tantos conocimientos previos como sea posible al agente, ya que acelerará la capacitación de manera espectacular. Por ejemplo, puede agregar recompensas negativas proporcionales a la distancia desde el centro de la pantalla y al ángulo del poste. Además, si ya tiene una política razonablemente buena (por ejemplo, codificada), es posible que desee entrenar la red neuronal para que la imite antes de usar gradientes de política para mejorarla.

A pesar de su relativa simplicidad, este algoritmo es bastante poderoso. Puede usarlo para abordar problemas mucho más difíciles que equilibrar un poste en un carro. De hecho, AlphaGo se basó en un algoritmo PG similar (más *Búsqueda de árboles de Montecarlo,* que está más allá del alcance de este libro).

Ahora veremos otra familia popular de algoritmos. Mientras que los algoritmos de PG intentan directamente optimizar la política para aumentar las recompensas, los algoritmos que veremos ahora son menos directos: el agente aprende a estimar la suma esperada de recompensas futuras descontadas para cada estado, o la suma esperada de recompensas futuras descontadas para cada estado. acción en cada estado, luego usa este conocimiento para decidir cómo actuar. Para comprender estos algoritmos, primero debemos introducir *Procesos de decisión de Markov (* MDP).

## Procesos de decisión de Markov

A principios de los 20 th siglo, el matemático Andrey Markov estudió procesos estocásticos sin memoria, llamados *Cadenas de Markov.* Dicho proceso tiene un número fijo de estados y evoluciona aleatoriamente de un estado a otro en cada paso. La probabilidad de que evolucione de un estado $s$ a un estado $s'$ es fijo, y depende solo del par ( $s, s'$ ),
no en estados pasados (el sistema no tiene memoria).

Figura 16-7. muestra un ejemplo de una cadena de Markov con cuatro estados. Suponga que el
el proceso comienza en el estado $s_0$, y existe un 70% de probabilidad de que permanezca en ese estado en el siguiente paso. Eventualmente está obligado a dejar ese estado y nunca volver, ya que no
otro estado apunta a $s_0$. Si va al estado $s_1$, entonces lo más probable es que vaya al estado $s_2$
(90% de probabilidad), luego inmediatamente de vuelta al estado $s_1$ ( con 100% de probabilidad). Puede alternar varias veces entre estos dos estados, pero eventualmente caerá en
estado $s_3$ y permanecer ahí para siempre (esto es un *estado terminal).* Las cadenas de Markov pueden tener dinámicas muy diferentes y se utilizan mucho en termodinámica, química,
estadísticas y mucho más.

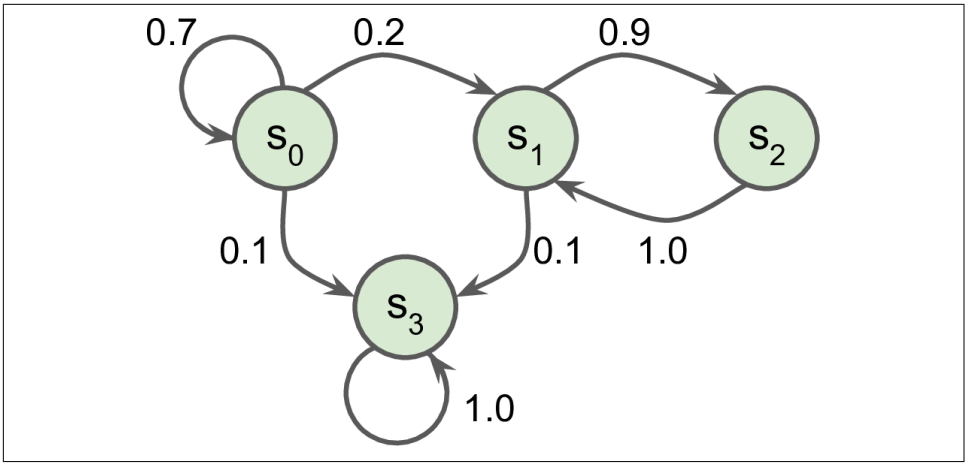


*Figura 16-7. Ejemplo de una cadena de Markov*

Los procesos de decisión de Markov fueron descrito por primera vez en la década de 1950 por Richard Bellman . [11]
Se parecen a las cadenas de Markov, pero con un giro: en cada paso, un agente puede elegir una de varias acciones posibles, y las probabilidades de transición dependen de la acción elegida. Además, algunas transiciones de estado devuelven alguna recompensa (positiva o negativa) y el objetivo del agente es encontrar una política que maximice las recompensas con el tiempo.

Por ejemplo, el MDP representado en Figura 16-8. tiene tres estados y hasta tres posibles acciones discretas en cada paso. Si comienza en estado $s_0$, el agente puede elegir entre acciones $un_0$, $un_1$, o $un_2$. Si elige acción $un_1$, solo permanece en estado $s_0$ con certeza y sin recompensa alguna. Por lo tanto, puede decidir quedarse allí para siempre si lo desea. Pero si elige acción $un_0$, tiene un 70% de probabilidad de obtener una recompensa de +10 y permanecer en el estado $s_0$. Luego, puede intentarlo una y otra vez para obtener la mayor recompensa posible. Pero en un momento va a terminar en el estado $s_1$. En el estado $s_1$ tiene solo dos acciones posibles: $un_0$ o $un_1$. Puede optar por quedarse quieto eligiendo repetidamente la acción $un_1$, o puede optar por pasar al estado $s_2$ y obtén una recompensa negativa de –50 (ay). En el estado $s_3$ no tiene más remedio que actuar $un_1$, lo que probablemente lo llevará de regreso al estado $s_0$, obteniendo una recompensa de +40 en el camino. Te dan la imagen. Al observar este MDP, ¿puede adivinar qué estrategia obtendrá la mayor recompensa con el tiempo? En el estado $s_0$ está claro que la acción $un_0$ es la mejor opción, y en estado $s_3$ el agente no tiene más remedio que actuar $un_1$, pero en estado $s_1$ no es obvio si el agente debe quedarse quieto ( $un_0$) o pasar por el fuego $un_2$).

11 “Un proceso de decisión arkoviano”, R. Bellman (1957).

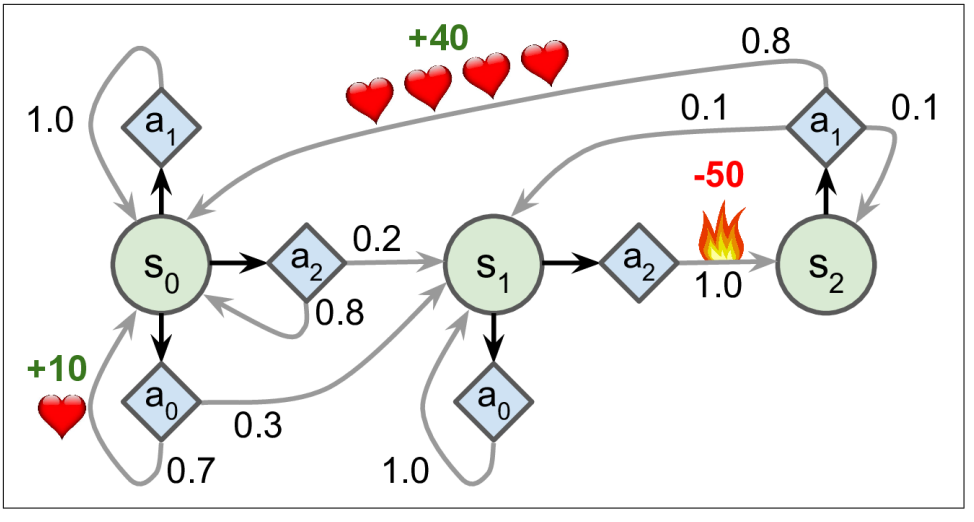


*Figura 16-8. Ejemplo de un proceso de decisión de Markov*

Bellman encontró una manera de estimar el *valor de estado óptimo* de cualquier estado *s,* célebre *V * (s),* que es la suma de todas las recompensas futuras con descuento que el agente puede esperar en promedio después de alcanzar un estado *s,* asumiendo que actúa de manera óptima. Demostró que si el agente actúa de manera óptima, entonces el *Ecuación de Optimidad de Bellman* se aplica (ver Ecuación 16-1 ). Esta ecuación recursiva dice que si el agente actúa de manera óptima, entonces el valor óptimo del estado actual es igual a la recompensa que obtendrá en promedio después de tomar una acción óptima, más el valor óptimo esperado de todos los estados siguientes posibles que esta acción puede llevar a.

*Ecuación 16-1. Ecuación de Optimidad de Bellman*

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

- $T(s, a, s')$ es la probabilidad de transición del estado $s$ a estado $s'$, dado que el agente eligió la acción $a$.
- $R(s, a, s')$ es la recompensa que obtiene el agente cuando pasa de estado $s$ a estado $s'$, dado que el agente eligió la acción $a$.
- $\gamma$ es la tasa de descuento.

Esta ecuación conduce directamente a un algoritmo que puede estimar con precisión el valor de estado óptimo de cada estado posible: primero inicializa todas las estimaciones de valor de estado a cero, y luego las actualiza iterativamente usando el *Iteración de valor* algoritmo (ver Ecuación 16-2 ). Un resultado notable es que, dado el tiempo suficiente, estas estimaciones son

garantizado para converger a los valores de estado óptimos, correspondientes a la política óptima.

*Ecuación 16-2. Algoritmo de iteración de valor*

$$V_{k+1}(s) \leftarrow \max_{un} \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \cdot V_k(s') \right] \quad \text{para todos } s$$

- $V_k(s)$ es el valor estimado del estado $s$ en el $k^{th}$ iteración del algoritmo.

Este algoritmo es un ejemplo de *Programación dinámica,* que descompone un problema complejo (en este caso, estimando una suma potencialmente infinita de recompensas futuras descontadas) en subproblemas manejables que se pueden abordar de manera iterativa (en este caso, encontrar la acción que maximiza la recompensa promedio más el siguiente valor descontado del estado ).

Conocer los valores de estado óptimos puede ser útil, en particular para evaluar una política, pero no le dice al agente explícitamente qué hacer. Afortunadamente, Bellman encontró un algoritmo muy similar para estimar el óptimo *valores de acción estatal,* generalmente llamado *Valores Q.* El valor Q óptimo del par estado-acción ( *s, a),* célebre *Q * (s, a),* es la suma de las recompensas futuras con descuento que el agente puede esperar en promedio después de llegar al estado *s* y elige la acción *un,* pero antes de ver el resultado de esta acción, asumiendo que actúa de manera óptima después de esa acción.

Así es como funciona: una vez más, comience por inicializar todas las estimaciones de Q-Value a cero, luego las actualice usando el *Iteración de valor Q* algoritmo (ver Ecuación 16-3 ).

*Ecuación 16-3. Algoritmo de iteración de valor Q*

$$Q_{k+1}(s, a) \leftarrow \sum_{s'} T(s, a, s') \left[ R(s, a, s') + \gamma \cdot \max_{un'} Q_k(s', un') \right] \quad \text{para todos } (s, a)$$

Una vez que tenga los Q-Values óptimos, definiendo la política óptima, anotó $\pi^*(s)$, es trivial: cuando el agente está en estado *s,* debe elegir la acción con el valor Q más alto para ese estado: $\pi^*(s) = \text{argmax}_{un} Q^*(s, un)$.

Apliquemos este algoritmo al MDP representado en Figura 16-8. . Primero, necesitamos definir el MDP:

```
yaya = notario público . yaya # representa acciones imposibles
T = notario público . formación ([ # forma = [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 ], [ yaya , yaya , yaya ], [ 0.0 , 0.0 , 1.0 ]], [[ yaya , yaya , yaya ], [ 0,8 , 0,1 ,
        0,1 ], [ yaya , yaya , yaya ]],
    ])
R = notario público . formación ([ # forma = [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 ], [ yaya , yaya , yaya ], [ 0.0
        , 0.0 , - 50. ]], [[ yaya , yaya , yaya ], [ 40. , 0.0 , 0.0 ], [ yaya , yaya , yaya ]],

    ])
posibles_acciones = [[ 0 , 1 , 2 ], [ 0 , 2 ], [ 1 ]]
```

Ahora ejecutemos el algoritmo de iteración de valor Q:

```
Q = notario público . completo (( 3 , 3 ), - notario público . inf ) # - inf para acciones imposibles
para estado , comportamiento en enumerar ( posibles_acciones ):
    Q [ estado , comportamiento ] = 0.0 # Valor inicial = 0.0, para todas las acciones posibles

tasa de aprendizaje = 0,01
tasa de descuento = 0,95
n_iteraciones = 100

para iteración en rango ( n_iteraciones ):
    Q_prev = Q . Copiar ()
    para s en rango ( 3 ):
        para un en posibles_acciones [ s ]:
            Q [ s , un ] = notario público . suma ([
                T [ s , un , sp ] * ( R [ s , un , sp ] + tasa de descuento * notario público . max ( Q_prev [ sp ]))
                para sp en rango ( 3 )
            ])
```

Los Q-Values resultantes se ven así:

```
>>> Q
matriz ([[21.89498982,  20.80024033,  16.86353093],
         [1.11669335,       - inf,   1.17573546],
         [      - inf,  53.86946068,       - inf]])
>>> notario público . argmax ( Q , eje = 1 ) # óptimo  acción para cada estado
matriz ([0, 2, 1])
```

Esto nos da la política óptima para este MDP, cuando se utiliza una tasa de descuento de 0,95: en estado $s_0$ Elige Acción $un_0$, en estado $s_1$ Elige Acción $un_2$ ( pasar por el fuego!), y en estado $s_2$ Elige Acción $un_1$ ( la única acción posible). Curiosamente, si reduce el descuento tasa a 0.9, la política óptima cambia: en el estado $s_1$ la mejor acción se convierte en $un_0$ ( quedarse quieto; no pases por el fuego). Tiene sentido porque si valoras mucho más el presente que el futuro, entonces la perspectiva de recompensas futuras no merece un dolor inmediato.

## Aprendizaje de diferencia temporal y Q-Learning

Los problemas de aprendizaje por refuerzo con acciones discretas a menudo se pueden modelar como procesos de decisión de Markov, pero el agente inicialmente no tiene idea de cuáles son las probabilidades de transición (no sabe *T (s, a, s ')*), y tampoco sabe cuáles van a ser las recompensas (no sabe *R (s, a, s ')*). Debe experimentar cada estado y

cada transición al menos una vez para conocer las recompensas, y debe experimentarlas varias veces para tener una estimación razonable de las probabilidades de transición.

los *Aprendizaje de la diferencia temporal (* El algoritmo TD Learning) es muy similar al algoritmo Value Iteration, pero modificado para tener en cuenta el hecho de que el agente solo tiene un conocimiento parcial del MDP. En general asumimos que el agente inicialmente conoce solo los posibles estados y acciones, y nada más. El agente usa un
*política de exploración* —Por ejemplo, una política puramente aleatoria— para explorar el MDP y, a medida que avanza, el algoritmo de TD Learning actualiza las estimaciones de los valores estatales en función de las transiciones y recompensas que se observan realmente (consulte Ecuación 16-4 ).

*Ecuación 16-4. Algoritmo TD Learning*

$$V_{k+1}(s) \leftarrow (1-\alpha)V_k(s) + \alpha(r + \gamma \cdot V_k(s'))$$

- $\alpha$ es la tasa de aprendizaje (por ejemplo, 0.01).

TD Learning tiene muchas similitudes con Stochastic Gradient Descent, en particular el hecho de que maneja una muestra a la vez. Al igual que SGD, solo puede converger realmente si reduce gradualmente la tasa de aprendizaje (de lo contrario, seguirá rebotando alrededor del óptimo).

Para cada estado *s,* este algoritmo simplemente realiza un seguimiento de un promedio móvil de las recompensas inmediatas que el agente obtiene al salir de ese estado, más las recompensas que espera obtener más adelante (asumiendo que actúa de manera óptima).

De manera similar, el algoritmo Q-Learning es una adaptación del algoritmo de iteración de Q-Value a la situación en la que las probabilidades de transición y las recompensas son inicialmente desconocidas (ver Ecuación 16-5 ).

*Ecuación 16-5. Algoritmo de Q-Learning*

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

Para cada par estado-acción ( *s, a),* este algoritmo realiza un seguimiento de un promedio móvil de las recompensas *r* el agente consigue al salir del estado *s* con accion *un,* además de las recompensas que espera obtener más adelante. Dado que la política objetivo actuaría de manera óptima, tomamos el máximo de las estimaciones de Q-Value para el siguiente estado.

Así es como se puede implementar Q-Learning:

```
importar numpy.random como rnd

tasa_de_aprendizaje0 = 0,05
learning_rate_decay = 0,1
n_iteraciones = 20000

s = 0 # comenzar en el estado 0

Q = notario público . completo (( 3 , 3 ), - notario público . inf ) # - inf para acciones imposibles
para estado , comportamiento en enumerar ( posibles_acciones ):
    Q [ estado , comportamiento ] = 0.0 # Valor inicial = 0.0, para todas las acciones posibles

para iteración en rango ( n_iteraciones ):
    un = rnd . elección ( posibles_acciones [ s ])            # elige una acción (aleatoriamente)
    sp = rnd . elección ( rango ( 3 ), pags = T [ s , un ])   # elige el siguiente estado usando T [s, a]
    recompensa = R [ s , un , sp ]
    tasa de aprendizaje = tasa_de_aprendizaje0 / ( 1          + iteración * learning_rate_decay )
    Q [ s , un ] = tasa de aprendizaje * Q [ s , un ] +       ( 1 - tasa de aprendizaje ) * (
                recompensa + tasa de descuento * notario público .  max ( Q [ sp ])
        )
    s = sp # pasar al siguiente estado
```

Dadas suficientes iteraciones, este algoritmo convergerá a los Q-Values óptimos. Esto se llama *fuera de la política* algoritmo porque la política que se está entrenando no es la que se está ejecutando. Es algo sorprendente que este algoritmo sea capaz de aprender la política óptima con solo ver a un agente actuar al azar (imagínese aprender a jugar al golf cuando su maestro es un mono borracho). ¿Podemos hacerlo mejor?

## Políticas de exploración

Por supuesto, Q-Learning solo puede funcionar si la política de exploración explora el MDP lo suficientemente a fondo. Aunque se garantiza que una póliza puramente aleatoria eventualmente visitará cada estado y cada transición muchas veces, puede llevar mucho tiempo hacerlo. Por tanto, una mejor opción es utilizar el *ε-política codiciosa:* en cada paso actúa aleatoriamente con probabilidad ε, o con avidez (eligiendo la acción con el valor Q más alto) con probabilidad 1-ε. La ventaja de la política ε-codiciosa (en comparación con una política completamente aleatoria) es que dedicará cada vez más tiempo a explorar las partes interesantes del entorno, a medida que las estimaciones de Q-Value mejoran cada vez más, sin dejar de pasar algún tiempo visitando regiones desconocidas del MDP. Es bastante común comenzar con un valor alto para ε (por ejemplo, 1.0) y luego reducirlo gradualmente (por ejemplo, hasta 0.05).

Alternativamente, en lugar de depender del azar para la exploración, otro enfoque es alentar a la política de exploración a probar acciones que no ha probado mucho antes. Esto se puede implementar como un bono agregado a las estimaciones de Q-Value, como se muestra en Ecuación 16-6 .

*Ecuación 16-6. Q-Learning usando una función de exploración Q s, a*

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

- $N(s', a')$ cuenta el número de veces que la acción $a'$ fue elegido en el estado $s'$.
- $f(q, n)$ es un *función de exploración,* como $f(q, n) = q + K / (1 + n)$, dónde $K$ es un hiperparámetro de curiosidad que mide cuánto le atrae al agente lo desconocido.

### Q-Learning aproximado

El principal problema con Q-Learning es que no escala bien a MDP grandes (o incluso medianos) con muchos estados y acciones. Considere intentar usar Q-Learning para entrenar a un agente para que interprete a la Sra. Pac-Man. Hay más de 250 gránulos que la Sra. Pac-Man puede comer, cada uno de los cuales puede estar presente o ausente (es decir, ya comido). Entonces el número de estados posibles es mayor que $2^{250} \approx 10^{75}$ ( y eso es considerando los posibles estados solo de los pellets). Esto es mucho más que los átomos en el universo observable, por lo que es absolutamente imposible realizar un seguimiento de una estimación de cada valor Q individual.

La solución es encontrar una función que se aproxime a los Q-Values usando un número manejable de parámetros. Se llama *Q-Learning aproximado.* Durante años se recomendó usar combinaciones lineales de características hechas a mano extraídas del estado (por ejemplo, la distancia de los fantasmas más cercanos, sus direcciones, etc.) para estimar los valores Q, pero DeepMind demostró que el uso de redes neuronales profundas puede funcionar mucho mejor, especialmente para problemas complejos, y no requiere ninguna ingeniería de funciones. Un DNN utilizado para estimar Q-Values se llama *red Q profunda (* DQN), y el uso de un DQN para Aproximado Q-Learning se llama *Q-Learning profundo.*

En el resto de este capítulo, usaremos Deep Q-Learning para entrenar a un agente para que juegue a Ms. Pac-Man, al igual que DeepMind hizo en 2013. El código se puede modificar fácilmente para aprender a jugar la mayoría de los juegos de Atari bastante bien. . Puede lograr habilidades sobrehumanas en la mayoría de los juegos de acción, pero no es tan bueno en juegos con historias de larga duración.

## Aprendiendo a jugar a Ms. Pac-Man usando Deep Q-Learning

Dado que usaremos un entorno Atari, primero debemos instalar las dependencias de Atari del gimnasio OpenAI. Mientras estamos en eso, también instalaremos dependencias para otros entornos de gimnasio OpenAI con los que quizás quieras jugar. En macOS, suponiendo que haya instalado Homebrew , necesitas ejecutar:

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

En Ubuntu, escriba el siguiente comando (reemplazando python3 con pitón si está utilizando 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
```

Luego instale los módulos de Python adicionales:

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

Si todo salió bien, debería poder crear un entorno de Ms.Pac-Man:

```
>>> env = Gimnasio . hacer ( "MsPacman-v0" )
>>> obs = env . Reiniciar ()
>>> obs . forma # [ alto, ancho, canales]
(210, 160, 3)
>>> env . action_space
Discreto (9)
```

Como puede ver, hay nueve acciones discretas disponibles, que corresponden a las nueve posiciones posibles del joystick (izquierda, derecha, arriba, abajo, centro, arriba a la izquierda, etc.), y las observaciones son simplemente capturas de pantalla del Atari. pantalla (ver Figura 16-9. , izquierda), representados como matrices 3D NumPy. Estas imágenes son un poco grandes, por lo que crearemos una pequeña función de preprocesamiento que recortará la imagen y la reducirá a 88 × 80 píxeles, la convertirá a escala de grises y mejorará el contraste de Ms. Pac-Man. Esto reducirá la cantidad de cálculos requeridos por el DQN y acelerará el entrenamiento.

```
mspacman_color = notario público . formación ([ 210 , 164 , 74 ]) . media ()

def preprocess_observation ( obs ):
    img = obs [ 1 : 176 : 2 , :: 2 ] # recortar y reducir
    img = img . media ( eje = 2 ) # a escala de grises
    img [ img == mspacman_color ] = 0 # mejorar el contraste
    img = ( img - 128 ) / 128 - 1 # normalizar desde -1. a 1.
    regreso img . remodelar ( 88 , 80 , 1 )
```

El resultado del preprocesamiento se muestra en Figura 16-9. (derecho).

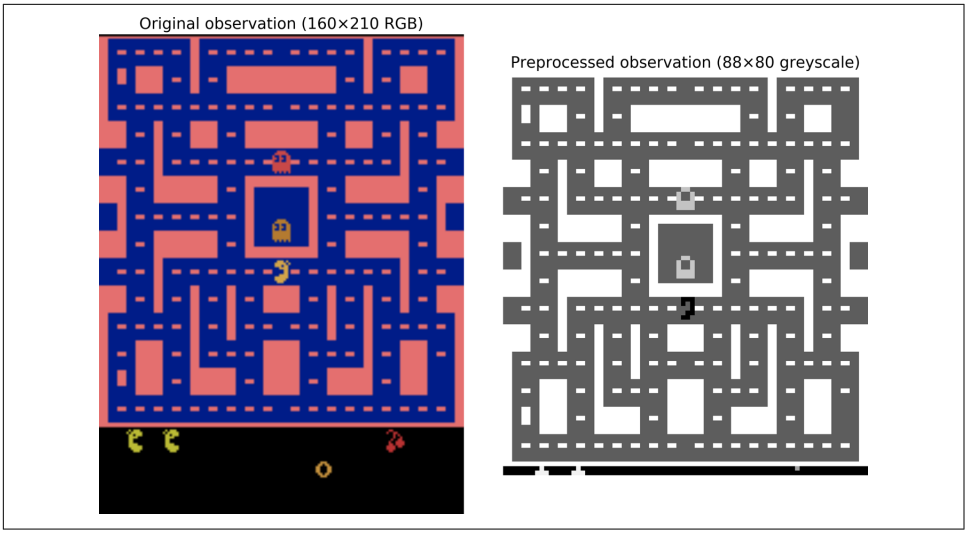


*Figura 16-9. Observación de la Sra. Pac-Man, original (izquierda) y después del preprocesamiento (derecha)*

A continuación, creemos el DQN. Solo podría tomar un par de acción de estado ( *s, a)* como entrada y salida una estimación del valor Q correspondiente *Q (s, a),* pero dado que las acciones son discretas, es más conveniente utilizar una red neuronal que solo toma un estado *s* como entrada y salida una estimación de Q-Value por acción. El DQN estará compuesto por tres capas convolucionales, seguidas de dos capas completamente conectadas, incluida la capa de salida (ver Figura 16-10. ).

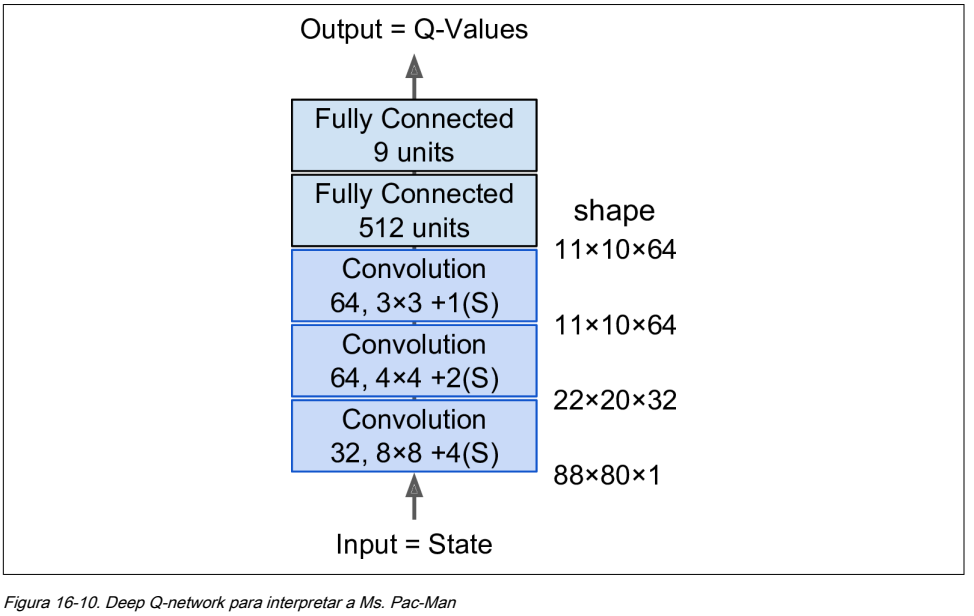


*Figura 16-10. Deep Q-network para interpretar a Ms. Pac-Man*

Como veremos, el algoritmo de entrenamiento que usaremos requiere dos DQN con la misma arquitectura (pero diferentes parámetros): uno se usará para conducir a la Sra. Pac-Man durante el entrenamiento (el *actor),* y el otro observará al actor y aprenderá de sus pruebas y errores (el *crítico).* A intervalos regulares copiaremos la crítica al actor. Dado que necesitamos dos DQN idénticos, crearemos un q_network () función para construirlos:

```
de tensorflow.contrib.layers importar convolution2d , totalmente_conectado

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 = [ "MISMO" ] * 3
conv_activation = [ tf . nn . relu ] * 3
n_hidden_in = 64 * 11 * 10 # conv3 tiene 64 mapas de 11x10 cada uno
n_hidden = 512
hidden_activation = tf . nn . r elu
n_salidas = env . action_spac mi . norte # 9 acciones discretas están disponibles
inicializador = tf . contrib . capas . variance_scaling_initializer ()

def q_network ( X_state , alcance ):
    prev_layer = X_state
    conv_layers = []
    con tf . Alcance variable ( alcance ) como alcance :
        para n_maps , kernel_size , paso , relleno , activación en Código Postal (
                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 ,
                paso = paso , relleno = relleno , activación_fn = activación ,
                weights_initializer = inicializador )
            conv_layers . adjuntar ( prev_layer )
        last_conv_layer_flat = tf . remodelar ( prev_layer , forma = [ - 1 , n_hidden_in ])
        oculto = totalmente_conectado (
            last_conv_layer_flat , n_hidden , activación_fn = hidden_activation ,
            weights_initializer = inicializador )
        salidas = totalmente_conectado (
            oculto , n_salidas , activación_fn = Ninguna ,
            weights_initializer = inicializador )
    entrenables_vars = tf . get_collection ( tf . GraphKeys . TRAINABLE_VARIABLES ,
                                             alcance = alcance . nombre )
    entrenables_vars_by_name = { var . nombre [ len ( alcance . nombre ):]: var
                                 para var en entrenables_vars }
    regreso salidas , entrenables_vars_by_name
```

La primera parte de este código define los hiperparámetros de la arquitectura DQN. Entonces el q_network () La función crea el DQN, tomando el estado del entorno.

X_state como entrada y el nombre del ámbito de la variable. Tenga en cuenta que solo usaremos uno

observación para representar el estado del entorno, ya que casi no hay un estado oculto (a excepción de los objetos parpadeantes y las direcciones de los fantasmas).

los entrenables_vars_by_name El diccionario reúne todas las variables entrenables de este DQN. Será útil en un minuto cuando creemos operaciones para copiar el crítico DQN al actor DQN. Las claves del diccionario son los nombres de las variables, eliminando la parte del prefijo que solo corresponde al nombre del alcance. Se parece a esto:

```
>>> entrenables_vars_by_name
{'/ Conv / bias: 0': <tensorflow.python.ops.variables.Variable at 0x121cf7b50>, '/ Conv / weights: 0':
  <tensorflow.python.ops.variables.Variable ...>, '/ Conv_1 / bias: 0 ': <tensorflow.python.ops.variables.Variable ...>,' /
  Conv_1 / weights: 0 ': <tensorflow.python.ops.variables.Variable ...>,' / Conv_2 / sesgos: 0 ':
  <tensorflow.python.ops.variables.Variable ...>,' / Conv_2 / weights: 0 ': <tensorflow.python.ops.variables.Variable ...>,'
  / totalmente_conectado / sesgos: 0 ': <tensorflow.python.ops.variables.Variable ...>,' / totalmente_conectado / pesos:
  0 ': <tensorflow.python.ops.variables.Variable ...>,' / totalmente_conectado_1 / sesgos: 0 ' :
  <tensorflow.python.ops.variables.Variable ...>, '/ full_connected_1 / weights: 0':
  <tensorflow.python.ops.variables.Variable ...>}
```

Ahora creemos el marcador de posición de entrada, los dos DQN y la operación para copiar el DQN crítico al actor DQN:

```
X_state = tf . marcador de posición ( tf . float32 , forma = [ Ninguna , input_height , input_width ,
                                                               input_channels ])
actor_q_values , actor_vars = q_network ( X_state , alcance = "q_networks / actor" )
critic_q_values , critico_vars = q_network ( X_state , alcance = "q_networks / critico" )

copy_ops = [ actor_var . asignar ( critico_vars [ var_name ])
             para var_name , actor_var en actor_vars . artículos ()]
copy_critic_to_actor = tf . grupo ( * copy_ops )
```

Retrocedamos un segundo: ahora tenemos dos DQN que son capaces de tomar un estado del entorno (es decir, una observación preprocesada) como entrada y generar un valor Q estimado para cada posible acción en ese estado. Además, tenemos una operación llamada copy_critic_to_actor copiar todas las variables entrenables del crítico DQN al actor DQN. Usamos TensorFlow's tf.group () función para agrupar todas las operaciones de asignación en una única operación conveniente.

El actor DQN se puede utilizar para interpretar a la Sra. Pac-Man (inicialmente muy mal). Como se mencionó anteriormente, desea que explore el juego lo suficientemente a fondo, por lo que generalmente desea combinarlo con un *ε-política codiciosa* u otra estrategia de exploración.

Pero ¿y el crítico DQN? ¿Cómo aprenderá a jugar? La respuesta corta es que intentará hacer que sus predicciones de valor Q coincidan con los valores Q estimados por el actor a través de su experiencia en el juego. En concreto, dejaremos que el actor juegue un rato, almacenando todas sus experiencias en un *reproducir la memoria*. Cada recuerdo será una tupla de 5 (estado, acción, siguiente estado, recompensa, continuar), donde el elemento "continuar" será igual a
0.0 cuando el juego termina, o 1.0 en caso contrario. A continuación, a intervalos regulares, probaremos un

lote de memorias de la memoria de repetición, y estimaremos los valores Q de estas memorias. Finalmente, entrenaremos al crítico DQN para predecir estos Q-Values usando técnicas regulares de aprendizaje supervisado. Una vez cada pocas iteraciones de entrenamiento, copiaremos el crítico DQN al actor DQN. ¡Y eso es! Ecuación 16-7 muestra la función de costo utilizada para capacitar al crítico DQN:

*Ecuación 16-7. Función de costos de Deep Q-Learning*

$$J(\theta_{\text{crítico}}) = \frac{1}{metro}\sum_{yo=1}^{metro}\left(y^{(yo)} - Q\left(s^{(yo)}, un^{(yo)}, \theta_{\text{crítico}}\right)\right)^2$$

$$\text{con } y^{(yo)} = r^{(yo)} + \gamma \cdot \max_{un'} Q\left(s'^{(yo)}, un', \theta_{\text{actor}}\right)$$

- $s^{(yo)}$, $un^{(yo)}$, $r^{(yo)}$ y $s'^{(yo)}$ son respectivamente el estado, la acción, la recompensa y el siguiente estado de la $i^{th}$ memoria muestreada de la memoria de repetición.
- *metro* es el tamaño del lote de memoria.
- $\theta_{\text{crítico}}$ y $\theta_{\text{actor}}$ son los parámetros del crítico y del actor.
- $Q\left(s^{(yo)}, un^{(yo)}, \theta_{\text{crítico}}\right)$ es la predicción del crítico DQN de la $i^{th}$ Valor Q de estado-acción memorizado.
- $Q\left(s'^{(yo)}, un', \theta_{\text{actor}}\right)$ es la predicción del actor DQN del valor Q que puede esperar del siguiente estado $s'^{(yo)}$ si elige acción $un'$.
- $y^{(yo)}$ es el valor Q objetivo para $i^{th}$ memoria. Tenga en cuenta que es igual a la recompensa realmente observada por el actor, más la recompensa del actor. *predicción* de las recompensas futuras que debería esperar si jugara de manera óptima (hasta donde sabe).
- $J(\theta_{\text{crítico}})$ es la función de coste utilizada para formar al crítico DQN. Como puede ver, es solo el error cuadrático medio entre los valores Q objetivo $y^{(yo)}$ según lo estimado por el actor DQN, y las predicciones del crítico DQN de estos Q-Values.

La memoria de reproducción es opcional, pero muy recomendable. Sin él, entrenarías al crítico DQN utilizando experiencias consecutivas que pueden estar muy correlacionadas. Esto introduciría mucho sesgo y ralentizaría la convergencia del algoritmo de entrenamiento. Al usar una memoria de repetición, nos aseguramos de que las memorias alimentadas al algoritmo de entrenamiento puedan no estar correlacionadas.

Agreguemos las operaciones de entrenamiento del crítico DQN. Primero, necesitamos poder calcular sus Q-Values predichos para cada acción de estado en el lote de memoria. Dado que el DQN emite un valor Q para cada acción posible, necesitamos mantener solo el valor Q que corresponde a la acción que realmente se eligió en esta memoria. Para esto, convertiremos la acción en un vector one-hot (recuerde que este es un vector lleno de ceros excepto por un

1 en la $i^{th}$ índice) y multiplicarlo por los Q-Values: esto pondrá a cero todos los Q-Values excepto el correspondiente a la acción memorizada. Luego, simplemente sume sobre el primer eje para obtener solo la predicción del valor Q deseado para cada memoria.

```
X_acción = tf . marcador de posición ( tf . int32 , forma = [ Ninguna ])
q_value = tf . reduce_sum ( critic_q_values * tf . one_hot ( X_acción , n_salidas ),
                                        eje = 1 , keep_dims = Cierto )
```

A continuación, agreguemos las operaciones de entrenamiento, asumiendo que los Q-Values objetivo se alimentarán a través de un marcador de posición. También creamos una variable no entrenable llamada paso_global.

El optimizador minimizar() la operación se encargará de incrementarlo. Además creamos lo habitual en eso operación y un Ahorrador.

```
y = tf . marcador de posición ( tf . float32 , forma = [ Ninguna , 1 ])
costo = tf . reduce_mean ( tf . cuadrado ( y - q_value ))
paso_global = tf . Variable ( 0 , entrenable = Falso , nombre = 'paso_global' )
optimizador = tf . tren . AdamOptimizer ( tasa de aprendizaje )
training_op = optimizador . minimizar ( costo , paso_global = paso_global )

en eso = tf . global_variables_initializer ()
ahorrador = tf . tren . Ahorrador ()
```

Eso es todo para la fase de construcción. Antes de analizar la fase de ejecución, necesitaremos un par de herramientas. Primero, comencemos implementando la memoria de reproducción. Usaremos un deque list, ya que es muy eficaz para enviar elementos a la cola y sacarlos del final de la lista cuando se alcanza el tamaño máximo de memoria. También escribiremos una pequeña función para muestrear aleatoriamente un lote de experiencias de la memoria de reproducción:

```
de colecciones importar deque

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

def sample_memories ( tamaño del lote ):
    índices = rnd . permutación ( len ( replay_memory )) [: tamaño del lote ]
    cols = [[], [], [], [], []] # estado, acción, recompensa, next_state, continuar
    para idx en índices :
        memoria = replay_memory [ idx ]
        para columna , valor en Código Postal ( cols , memoria ):
            columna . adjuntar ( valor )
    cols = [ notario público . formación ( columna ) para columna en cols ]
    regreso ( cols [ 0 ], cols [ 1 ], cols [ 2 ] . remodelar ( - 1 , 1 ), cols [ 3 ],
            cols [ 4 ] . remodelar ( - 1 , 1 ))
```

A continuación, necesitaremos que el actor explore el juego. Usaremos la política ε-codiciosa y disminuiremos gradualmente ε de 1.0 a 0.05, en 50,000 pasos de entrenamiento:

```
eps_min = 0,05
eps_max = 1.0
eps_decay_steps = 50000
```

```
def épsilon_greedy ( q_values , paso ):
    épsilon = max ( eps_min , eps_max - ( eps_max - eps_min ) * paso / eps_decay_steps )
    Si rnd . rand () < épsilon :
        regreso rnd . randint ( n_salidas ) # acción aleatoria
    más :
        regreso notario público . argmax ( q_values ) # acción óptima
```

¡Eso es todo! Tenemos todo lo que necesitamos para empezar a entrenar. La fase de ejecución no contiene nada demasiado complejo, pero es un poco larga, así que respira hondo. Listo? ¡Vamonos! Primero, inicialicemos algunas variables:

```
n_pasos = 100000 # número total de pasos de entrenamiento
training_start = 1000 # comenzar a entrenar después de 1,000 iteraciones del juego
entrenamiento_intervalo = 3 # ejecutar un paso de entrenamiento cada 3 iteraciones del juego
save_steps = 50 # guarda el modelo cada 50 pasos de entrenamiento
copy_steps = 25 # Copiar la crítica al actor cada 25 pasos de entrenamiento.
tasa de descuento = 0,95
skip_start = 90 # omitir el inicio de cada juego (es solo tiempo de espera)
tamaño del lote = 50
iteración = 0 # iteraciones del juego
checkpoint_path = "./my_dqn.ckpt"
hecho = Cierto # env necesita ser reiniciado
```

A continuación, abramos la sesión y ejecutemos el ciclo de entrenamiento principal:

```
con tf . Sesión () como sess :
    Si os . camino . isfile ( checkpoint_path ):
        ahorrador . restaurar ( sess , checkpoint_path )
    más :
        en eso . correr ()
    mientras Cierto :
        paso = paso_global . eval ()
        Si paso > = n_pasos :
            descanso
        iteración + = 1
        Si hecho : # juego terminado, empezar de nuevo
            obs = env . Reiniciar ()
            para omitir en rango ( skip_start ): # omitir el inicio de cada juego
                obs , recompensa , hecho , info = env . paso ( 0 )
            estado = preprocess_observation ( obs )

        # El actor evalúa qué hacer
        q_values = actor_q_values . eval ( feed_dict = { X_state : [ estado ]})
        acción = épsilon_greedy ( q_values , paso )

        # Actor juega
        obs , recompensa , hecho , info = env . paso ( acción )
        next_state = preprocess_observation ( obs )

        # Memoricemos lo que acaba de pasar
        replay_memory . adjuntar (( estado , acción , recompensa , next_state , 1.0 - hecho ))
        estado = next_state
```

```
        Si iteración < training_start o iteración % entrenamiento_intervalo ! = 0 :
            Seguir

        # El crítico aprende
        X_state_val , X_action_val , recompensas , X_next_state_val , continúa = (
            sample_memories ( tamaño del lote ))
        next_q_values = actor_q_values . eval (
            feed_dict = { X_state : X_next_state_val })
        max_next_q_values = notario público . max ( next_q_values , eje = 1 , Keepdims = Cierto )
        y_val = recompensas + continúa * tasa de descuento * max_next_q_values   training_op . correr ( feed_dict
        = { X_state : X_state_val ,
                                                X_acción : X_action_val , y : y_val })

        # Copiar regularmente el crítico al actor
        Si paso % copy_steps == 0 :
            copy_critic_to_actor . correr ()

        # Y ahorra regularmente
        Si paso % save_steps == 0 :
            ahorrador . salvar ( sess , checkpoint_path )
```

Comenzamos restaurando los modelos si existe un archivo de punto de control, o simplemente inicializamos las variables normalmente. Entonces comienza el bucle principal, donde iteración cuenta el número total de pasos del juego que hemos seguido desde que comenzó el programa, y paso
cuenta el número total de pasos de entrenamiento desde que comenzó el entrenamiento (si se restaura un punto de control, también se restaura el paso global). Luego, el código reinicia el juego (y omite los primeros pasos aburridos del juego, donde no sucede nada). A continuación, el actor evalúa qué hacer, juega el juego y su experiencia se memoriza en la memoria de repetición. Luego, a intervalos regulares (después de un período de calentamiento), el crítico pasa por un paso de entrenamiento. Muestra un lote de memorias y le pide al actor que estime los Q-Values de todas las acciones para el siguiente estado, y aplica Ecuación 16-7 para calcular el valor Q objetivo y_val.

La única parte complicada aquí es que debemos multiplicar los valores Q del siguiente estado por el continúa vector para poner a cero los Q-Values correspondientes a las memorias donde el juego terminó. A continuación, realizamos una operación de formación para mejorar la capacidad del crítico de predecir los Q-Values. Finalmente, a intervalos regulares copiamos la crítica al actor y guardamos el modelo.

Desafortunadamente, el entrenamiento es muy lento: si usa su computadora portátil para entrenar, pasarán días antes de que la Sra. Pac-Man mejore, y si observa la curva de aprendizaje, midiendo las recompensas promedio por episodio, notará que es extremadamente ruidoso. En algunos puntos, es posible que no haya un progreso aparente durante mucho tiempo hasta que, de repente, el agente aprenda a sobrevivir un período de tiempo razonable. Como se mencionó anteriormente, una solución es inyectar tanto conocimiento previo como sea posible en el modelo (por ejemplo, a través del preprocesamiento, recompensas, etc.), y también puede intentar arrancar el modelo entrenándolo primero para imitar un modelo básico. estrategia. En cualquier caso, RL todavía requiere mucha paciencia y ajustes, pero el resultado final es muy emocionante.

## Ejercicios

1. ¿Cómo definiría el aprendizaje por refuerzo? ¿En qué se diferencia del aprendizaje regular supervisado o no supervisado?
2. ¿Puede pensar en tres posibles aplicaciones de RL que no se mencionaron en este capítulo? Para cada uno de ellos, ¿qué es el medio ambiente? ¿Qué es el agente? ¿Cuáles son las posibles acciones? ¿Cuáles son las recompensas?
3. ¿Qué es la tasa de descuento? ¿Puede cambiar la política óptima si modifica la tasa de descuento?
4. ¿Cómo se mide el desempeño de un agente de aprendizaje por refuerzo?
5. ¿Cuál es el problema de la asignación de créditos? ¿Cuándo ocurre? ¿Cómo se puede aliviar?
6. ¿Cuál es el punto de usar una memoria de reproducción?
7. ¿Qué es un algoritmo RL fuera de la política?
8. Utilice Deep Q-Learning para abordar el "BypedalWalker-v2" del gimnasio OpenAI. Las redes Q no necesitan ser muy profundas para esta tarea.
9. Utilice gradientes de políticas para capacitar a un agente para que juegue *Apestar,* el famoso juego de Atari Apestar-v0 en el gimnasio OpenAI). Atención: una observación individual es insuficiente para saber la dirección y la velocidad de la pelota. Una solución es pasar dos observaciones a la vez a la política de red neuronal. Para reducir la dimensionalidad y acelerar el entrenamiento, definitivamente debería preprocesar estas imágenes (recortarlas, redimensionarlas y convertirlas a blanco y negro), y posiblemente fusionarlas en una sola imagen (por ejemplo, superponiéndolas).
10. Si tiene alrededor de $ 100 de sobra, puede comprar una Raspberry Pi 3 más algunos componentes robóticos baratos, instalar TensorFlow en la Pi y ¡enloquecer! Por ejemplo, mira este publicación divertida de Lukas Biewald, o echa un vistazo a GoPiGo o BrickPi. ¿Por qué no intentar construir un mástil de la vida real entrenando al robot con pol-

gradientes helados? O construye una araña robótica que aprenda a caminar; dele recompensas cada vez que se acerque a algún objetivo (necesitará sensores para medir la distancia al objetivo). El único límite es tu imaginación.

Las soluciones a estos ejercicios están disponibles en Apéndice A .

# ¡Gracias!

Antes de cerrar el último capítulo de este libro, me gustaría agradecerle por leerlo hasta el último párrafo. Realmente espero que haya disfrutado tanto leyendo este libro como yo escribiéndolo, y que sea útil para sus proyectos, grandes o pequeños.

Si encuentra errores, envíe sus comentarios. De manera más general, me encantaría saber lo que piensa, así que no dude en ponerse en contacto conmigo a través de O'Reilly o mediante el *ageron / handson-ml* Proyecto GitHub.

De cara al futuro, mi mejor consejo es que practique y practique: intente realizar todos los ejercicios si aún no lo ha hecho, juegue con los cuadernos de Jupyter, únase a Kaggle.com o alguna otra comunidad de aprendizaje automático, vea cursos de aprendizaje automático, lea artículos , asistir a conferencias, conocer expertos. Es posible que también desee estudiar algunos temas que no cubrimos en este libro, incluidos los sistemas de recomendación, los algoritmos de agrupamiento, los algoritmos de detección de anomalías y los algoritmos genéticos.

Mi mayor esperanza es que este libro lo inspire a crear una aplicación de aprendizaje automático maravillosa que nos beneficiará a todos. ¿Qué será?

*Aurélien Géron, 26 de noviembre de 2016*

APÉNDICE A

# Soluciones de ejercicio

Las soluciones para los ejercicios de codificación están disponibles en los cuadernos de Jupyter en línea en *https://github.com/ageron/handson-ml* .

## Capítulo 1 : El panorama del aprendizaje automático

1. El aprendizaje automático se trata de crear sistemas que puedan aprender de los datos. Aprender significa mejorar en alguna tarea, dada alguna medida de desempeño.

2. El aprendizaje automático es excelente para problemas complejos para los que no tenemos una solución algorítmica, para reemplazar largas listas de reglas ajustadas a mano, para construir sistemas que se adapten a entornos fluctuantes y, finalmente, para ayudar a los humanos a aprender (por ejemplo, minería de datos) .

3. Un conjunto de entrenamiento etiquetado es un conjunto de entrenamiento que contiene la solución deseada (también conocida como una etiqueta) para cada instancia.

4. Las dos tareas supervisadas más comunes son la regresión y la clasificación.

5. Las tareas comunes no supervisadas incluyen agrupamiento, visualización, reducción de dimensionalidad y aprendizaje de reglas de asociación.

6. Es probable que el aprendizaje por refuerzo funcione mejor si queremos que un robot aprenda a caminar en varios terrenos desconocidos, ya que este es típicamente el tipo de problema que aborda el aprendizaje por refuerzo. Podría ser posible expresar el problema como un problema de aprendizaje supervisado o semisupervisado, pero sería menos natural.

7. Si no sabe cómo definir los grupos, puede utilizar un algoritmo de agrupamiento (aprendizaje no supervisado) para segmentar a sus clientes en grupos de clientes similares. Sin embargo, si sabe qué grupos le gustaría tener, entonces

puede alimentar muchos ejemplos de cada grupo a un algoritmo de clasificación (aprendizaje supervisado), y clasificará a todos sus clientes en estos grupos.

8. La detección de spam es un problema típico de aprendizaje supervisado: el algoritmo recibe muchos correos electrónicos junto con su etiqueta (spam o no spam).

9. Un sistema de aprendizaje en línea puede aprender de forma incremental, a diferencia de un sistema de aprendizaje por lotes. Esto lo hace capaz de adaptarse rápidamente tanto a los datos cambiantes como a los sistemas autónomos, y de entrenar en cantidades muy grandes de datos.

10. Los algoritmos fuera del núcleo pueden manejar grandes cantidades de datos que no caben en la memoria principal de una computadora. Un algoritmo de aprendizaje fuera del núcleo divide los datos en mini lotes y utiliza técnicas de aprendizaje en línea para aprender de estos mini lotes.

11. Un sistema de aprendizaje basado en instancias aprende los datos de entrenamiento de memoria; luego, cuando se le da una nueva instancia, usa una medida de similitud para encontrar las instancias aprendidas más similares y las usa para hacer predicciones.

12. Un modelo tiene uno o más parámetros de modelo que determinan lo que predecirá dada una nueva instancia (por ejemplo, la pendiente de un modelo lineal). Un algoritmo de aprendizaje intenta encontrar valores óptimos para estos parámetros de manera que el modelo se generalice bien a nuevas instancias. Un hiperparámetro es un parámetro del algoritmo de aprendizaje en sí, no del modelo (por ejemplo, la cantidad de regularización a aplicar).

13. Los algoritmos de aprendizaje basados en modelos buscan un valor óptimo para los parámetros del modelo de modo que el modelo se generalice bien a nuevas instancias. Por lo general, entrenamos dichos sistemas minimizando una función de costo que mide qué tan malo es el sistema para hacer predicciones sobre los datos de entrenamiento, más una penalización por la complejidad del modelo si el modelo está regularizado. Para hacer predicciones, alimentamos las características de la nueva instancia en la función de predicción del modelo, utilizando los valores de los parámetros encontrados por el algoritmo de aprendizaje.

14. Algunos de los principales desafíos en el aprendizaje automático son la falta de datos, la mala calidad de los datos, los datos no representativos, las características poco informativas, los modelos excesivamente simples que se ajustan a los datos de entrenamiento y los modelos excesivamente complejos que se ajustan a los datos.

15. Si un modelo se desempeña muy bien en los datos de entrenamiento, pero se generaliza mal a las nuevas instancias, es probable que el modelo sobreajuste los datos de entrenamiento (o tuvimos mucha suerte con los datos de entrenamiento). Las posibles soluciones para el sobreajuste son obtener más datos, simplificar el modelo (seleccionar un algoritmo más simple, reducir la cantidad de parámetros o funciones utilizadas o regularizar el modelo) o reducir el ruido en los datos de entrenamiento.

16. Un conjunto de prueba se utiliza para estimar el error de generalización que cometerá un modelo en nuevas instancias, antes de que el modelo se lance en producción.

17. Se utiliza un conjunto de validación para comparar modelos. Permite seleccionar el mejor modelo y ajustar los hiperparámetros.

18. Si ajusta los hiperparámetros con el conjunto de prueba, corre el riesgo de sobreajustar el conjunto de prueba y el error de generalización que mida será optimista (puede lanzar un modelo que funcione peor de lo esperado).

19. La validación cruzada es una técnica que permite comparar modelos (para la selección de modelos y el ajuste de hiperparámetros) sin la necesidad de un conjunto de validación separado. Esto ahorra valiosos datos de entrenamiento.

## Capitulo 2 : Proyecto de aprendizaje automático de extremo a extremo

Vea los cuadernos de Jupyter disponibles en *https://github.com/ageron/handson-ml* .

## Capítulo 3 : Clasificación

Vea los cuadernos de Jupyter disponibles en *https://github.com/ageron/handson-ml* .

## Capítulo 4 : Entrenamiento de modelos lineales

1. Si tiene un conjunto de entrenamiento con millones de funciones, puede usar Descenso por gradiente estocástico o Descenso por gradiente por mini lotes, y quizás Descenso por gradiente por lotes si el conjunto de entrenamiento cabe en la memoria. Pero no puede usar la ecuación normal porque la complejidad computacional crece rápidamente (más que cuadráticamente) con el número de características.

2. Si las características de su conjunto de entrenamiento tienen escalas muy diferentes, la función de costo tendrá la forma de un cuenco alargado, por lo que los algoritmos de Gradient Descent tardarán mucho en converger. Para resolver esto, debe escalar los datos antes de entrenar el modelo. Tenga en cuenta que la ecuación normal funcionará bien sin escalar.

3. Gradient Descent no puede quedarse atascado en un mínimo local cuando se entrena un modelo de regresión logística porque la función de costo es convexa. [1]

4. Si el problema de optimización es convexo (como Regresión lineal o Regresión logística) y asumiendo que la tasa de aprendizaje no es demasiado alta, todos los algoritmos de Gradient Descent se acercarán al óptimo global y terminarán produciendo modelos bastante similares. Sin embargo, a menos que reduzca gradualmente la tasa de aprendizaje, la GD estocástica y la GD por mini lotes nunca convergerán realmente; en cambio, seguirán saltando de un lado a otro alrededor del óptimo global. Esto significa que incluso

1 Si dibuja una línea recta entre dos puntos cualesquiera de la curva, la línea nunca cruza la curva.

si los dejas funcionar durante mucho tiempo, estos algoritmos de Gradient Descent producirán modelos ligeramente diferentes.

5. Si el error de validación aumenta constantemente después de cada época, entonces una posibilidad es que la tasa de aprendizaje sea demasiado alta y el algoritmo diverja. Si el error de entrenamiento también aumenta, entonces este es claramente el problema y debe reducir la tasa de aprendizaje. Sin embargo, si el error de entrenamiento no aumenta, entonces su modelo está sobreajustando el conjunto de entrenamiento y debe dejar de entrenar.

6. Debido a su naturaleza aleatoria, ni el Descenso de gradiente estocástico ni el Descenso de gradiente por mini lotes están garantizados para progresar en cada iteración de entrenamiento. Por lo tanto, si deja de entrenar inmediatamente cuando aumenta el error de validación, es posible que se detenga demasiado pronto, antes de alcanzar el nivel óptimo. Una mejor opción es guardar el modelo a intervalos regulares, y cuando no ha mejorado durante mucho tiempo (lo que significa que probablemente nunca batirá el récord), puede volver al modelo mejor guardado.

7. El descenso de gradiente estocástico tiene la iteración de entrenamiento más rápida, ya que considera solo una instancia de entrenamiento a la vez, por lo que generalmente es el primero en alcanzar la vecindad del óptimo global (o GD de mini lotes con un tamaño de mini lotes muy pequeño) . Sin embargo, solo el Descenso de gradiente por lotes realmente convergerá, con suficiente tiempo de entrenamiento. Como se mencionó, Stochastic GD y Mini-batch GD rebotarán alrededor del óptimo, a menos que reduzca gradualmente la tasa de aprendizaje.

8. Si el error de validación es mucho mayor que el error de entrenamiento, es probable que su modelo esté sobreajustando el conjunto de entrenamiento. Una forma de intentar solucionar este problema es reducir el grado del polinomio: es menos probable que un modelo con menos grados de libertad se sobreajuste. Otra cosa que puede intentar es regularizar el modelo, por ejemplo,
agregando un $\ell_2$ penalización (Ridge) o un $\ell_1$ penalización (lazo) a la función de coste. Esto también reducirá los grados de libertad del modelo. Por último, puedes intentar
aumentar el tamaño del conjunto de entrenamiento.

9. Si tanto el error de entrenamiento como el error de validación son casi iguales y bastante altos, es probable que el modelo no se ajuste al conjunto de entrenamiento, lo que significa que tiene un alto sesgo. Debería intentar reducir el hiperparámetro de regularización $\alpha$.

10. Veamos:

    • Un modelo con algo de regularización generalmente funciona mejor que un modelo sin regularización, por lo que generalmente debe preferir la regresión de crestas sobre la regresión lineal simple. [2]

    • La regresión de lazo usa un $\ell_1$ penalización, que tiende a reducir los pesos a exactamente cero. Esto conduce a modelos dispersos, donde todos los pesos son cero excepto para

2 Además, la ecuación normal requiere calcular la inversa de una matriz, pero esa matriz no siempre es invertible. Por el contrario, la matriz para la regresión de crestas siempre es invertible.

los pesos más importantes. Esta es una forma de realizar la selección de características automáticamente, lo cual es bueno si sospecha que solo algunas características realmente importan. Cuando no esté seguro, debería preferir Ridge Regression.

- Generalmente, se prefiere Elastic Net a Lasso, ya que Lasso puede comportarse de manera errática en algunos casos (cuando varias características están fuertemente correlacionadas o cuando hay más características que instancias de entrenamiento). Sin embargo, agrega un hiperparámetro extra para ajustar. Si solo desea Lasso sin el comportamiento errático, puede usar Elastic Net con un l1_ratio cerca de 1.

11. Si desea clasificar imágenes como exteriores / interiores y diurnas / nocturnas, ya que estas no son clases exclusivas (es decir, las cuatro combinaciones son posibles), debe entrenar dos clasificadores de regresión logística.

12. Consulte los cuadernos de Jupyter disponibles en *https://github.com/ageron/handson-ml* .

## Capítulo 5 : Máquinas vectoriales de soporte

1. La idea fundamental detrás de Support Vector Machines es encajar la “calle” más amplia posible entre las clases. En otras palabras, el objetivo es tener el mayor margen posible entre el límite de decisión que separa las dos clases y las instancias de capacitación. Al realizar una clasificación de margen suave, la SVM busca un compromiso entre separar perfectamente las dos clases y tener la calle más ancha posible (es decir, algunos casos pueden terminar en la calle). Otra idea clave es usar kernels al entrenar en conjuntos de datos no lineales.

2. Después de entrenar un SVM, un *vector de apoyo* es cualquier instancia ubicada en la "calle" (ver la respuesta anterior), incluido su borde. El límite de decisión está completamente determinado por los vectores de soporte. Cualquier instancia que sea *no* un vector de apoyo (es decir, fuera de la calle) no tiene influencia alguna; puede eliminarlos, agregar más instancias o moverlos, y mientras permanezcan fuera de la calle no afectarán el límite de decisión. Calcular las predicciones solo involucra los vectores de apoyo, no todo el conjunto de entrenamiento.

3. Las SVM intentan encajar la "calle" más grande posible entre las clases (consulte la primera respuesta), por lo que si el conjunto de entrenamiento no se escala, la SVM tenderá a descuidar las características pequeñas (consulte Figura 5-2 ).

4. Un clasificador de SVM puede generar la distancia entre la instancia de prueba y el límite de decisión, y puede usar esto como una puntuación de confianza. Sin embargo, esta puntuación no se puede convertir directamente en una estimación de la probabilidad de la clase. Si pones probabilidad = Verdadero al crear una SVM en Scikit-Learn, luego del entrenamiento, calibrará las probabilidades usando Regresión logística en las puntuaciones de SVM (entrenadas por una validación cruzada adicional de cinco veces en los datos de entrenamiento). Esta agregará el predict_proba () y predecir_log_proba () métodos a la SVM.

5. Esta pregunta se aplica solo a las SVM lineales, ya que las kernelizadas solo pueden usar la forma dual. La complejidad computacional de la forma primaria del problema SVM es proporcional al número de instancias de entrenamiento *metro,* mientras que la complejidad computacional de la forma dual es proporcional a un número entre $metro^2$ y $metro^3$. Entonces, si hay millones de instancias, definitivamente debería usar la forma primaria, porque la forma dual será demasiado lenta.

6. Si un clasificador SVM entrenado con un kernel RBF no se adapta al conjunto de entrenamiento, puede haber demasiada regularización. Para disminuirlo, necesita aumentar gama o C (o ambos).

7. Llamemos a los parámetros QP para el problema del margen rígido **H** ′, **F** ′, **UN** ′ y **segundo** ′ ( ver "Programación cuadrática" en la página 159 ). Los parámetros QP para el margen blando problema tiene *metro* parámetros adicionales ( $norte_p = n + 1 + metro$) y *metro* restricciones adicionales ( $norte_c = 2\ metro$). Se pueden definir así:

   - **H** es igual a **H** ′, más *metro* columnas de ceros a la derecha y *metro* filas de ceros en la parte inferior: $\begin{pmatrix} \text{H}' & 0 & \cdots \\ \vdots & & \ddots \end{pmatrix}$
   - **F** es igual a **F** ′ con *metro* elementos adicionales, todos iguales al valor del hiper- parámetro *C.*
   - **segundo** es igual a **segundo** ′ con *metro* elementos adicionales, todos iguales a 0.
   - **UN** es igual a **UN** ′, con un extra *metro* × *metro* matriz de identidad **yo** $_{metro}$ anexado a la derecha,
     - **yo** $_{metro}$ justo debajo de él, y el resto lleno de ceros: $\begin{pmatrix} \text{A}' & I_{metro} \\ \mathbf{0} & -I_{metro} \end{pmatrix}$

Para conocer las soluciones a los ejercicios 8, 9 y 10, consulte los cuadernos de Jupyter disponibles en *https://github.com/ageron/handson-ml* .

# Capítulo 6 : Árboles de decisión

1. La profundidad de un árbol binario bien equilibrado que contiene *metro* hojas es igual a $registro_2(metro)$[3], redondeado. Un árbol de decisión binario (uno que solo toma decisiones binarias, como es el caso de todos los árboles en Scikit-Learn) terminará más o menos bien equilibrado al final del entrenamiento, con una hoja por instancia de entrenamiento si se entrena sin restricciones. Por tanto, si el conjunto de entrenamiento contiene un millón de instancias, el árbol de decisiones tendrá una profundidad de $registro_2(10^6) \approx 20$ (en realidad un poco más, ya que el árbol generalmente no estará perfectamente equilibrado).

3 $registro_2$ es el registro binario, $registro_2(m) = \text{Iniciar sesión}(m) / \text{registro}(2)$.

2. La impureza de Gini de un nodo es generalmente menor que la de su padre. Esto está asegurado por la función de costo del algoritmo de entrenamiento CART, que divide cada nodo de una manera que minimiza la suma ponderada de las impurezas de Gini de sus hijos. Sin embargo, si un niño es más pequeño que el otro, es posible que tenga una impureza de Gini más alta que su padre, siempre que este aumento sea más que compensado por una disminución de la impureza del otro niño. Por ejemplo, considere un nodo que contiene cuatro instancias de clase A y 1 de clase B. Su impureza de Gini es 1 - 1 2 - 4 2 $\frac{}{5}$ $\frac{}{5}$ = 0,32. Ahora suponga que el conjunto de datos es unidimensional y las instancias están alineadas en el siguiente orden: A, B, A, A, A. Puede verificar que el algoritmo dividirá este nodo después de la segunda instancia, produciendo un nodo hijo con instancias A, B y el otro nodo hijo con instancias A, A, A. El Gini del primer nodo hijo la impureza es 1 - 1 2 - 1 2 $\frac{}{2}$ $\frac{}{2}$ = 0.5, que es mayor que su padre. Esto es compensado porque por el hecho de que el otro nodo es puro, la impureza de Gini ponderada general es 2 $\frac{}{5}$ × 0,5 + 3 $\frac{}{5}$ × 0 = 0,2, que es menor que la impureza de Gini del padre.

3. Si un árbol de decisión sobreajusta el conjunto de entrenamiento, puede ser una buena idea reducir máxima profundidad, ya que esto limitará el modelo, regularizándolo.

4. A los árboles de decisión no les importa si los datos de entrenamiento están escalados o centrados; esa es una de las cosas buenas de ellos. Por tanto, si un árbol de decisiones no se adapta al conjunto de formación, escalar las características de entrada será una pérdida de tiempo.

5. La complejidad computacional de entrenar un árbol de decisión es *En* × *metro* Iniciar sesión( *metro)).* Entonces Si multiplica el tamaño del conjunto de entrenamiento por 10, el tiempo de entrenamiento se multiplicará por *K = (n* × 10 *metro* × registro (10 *m)) / (n* × *metro* × registro ( *m)) =* 10 × registro (10 *m) /* Iniciar sesión( *metro).* Si *m =* 10 6, entonces *K* ≈ 11,7, por lo que puede esperar que el tiempo de entrenamiento sea de aproximadamente 11,7 horas.

6. La clasificación previa del conjunto de entrenamiento acelera el entrenamiento solo si el conjunto de datos es menor que unos pocos miles de instancias. Si contiene 100,000 instancias, configurando presort = Verdadero ralentizará considerablemente el entrenamiento.

Para conocer las soluciones a los ejercicios 7 y 8, consulte los cuadernos de Jupyter disponibles en *https://github.com/ageron/handson-ml* .

## Capítulo 7 : Aprendizaje conjunto y bosques aleatorios

1. Si ha entrenado cinco modelos diferentes y todos alcanzan un 95% de precisión, puede intentar combinarlos en un conjunto de votación, lo que a menudo le dará mejores resultados. Funciona mejor si los modelos son muy diferentes (por ejemplo, un clasificador SVM, un clasificador de árbol de decisión, un clasificador de regresión logística, etc.). Es incluso mejor si están entrenados en diferentes instancias de entrenamiento (ese es el objetivo de empaquetar y pegar conjuntos), pero si no, seguirá funcionando siempre que los modelos sean muy diferentes.

2. Un clasificador de votación estricta solo cuenta los votos de cada clasificador en el conjunto y elige la clase que obtiene la mayor cantidad de votos. Un clasificador de votación suave calcula la probabilidad de clase estimada promedio para cada clase y elige la clase con la probabilidad más alta. Esto le da más peso a los votos de alta confianza y, a menudo, se desempeña mejor, pero solo funciona si cada clasificador es capaz de estimar las probabilidades de clase (por ejemplo, para los clasificadores SVM en Scikit-Learn, debe establecer

   probabilidad = Verdadero).

3. Es muy posible acelerar el entrenamiento de un conjunto de ensacado distribuyéndolo en varios servidores, ya que cada predictor del conjunto es independiente de los demás. Lo mismo ocurre con el pegado de conjuntos y bosques aleatorios, por la misma razón. Sin embargo, cada predictor en un conjunto de impulso se crea en función del predictor anterior, por lo que el entrenamiento es necesariamente secuencial y no obtendrá nada al distribuir el entrenamiento en varios servidores. En cuanto a los conjuntos de apilamiento, todos los predictores de una capa determinada son independientes entre sí, por lo que se pueden entrenar en paralelo en varios servidores. Sin embargo, los predictores de una capa solo se pueden entrenar después de que se hayan entrenado todos los predictores de la capa anterior.

4. Con la evaluación fuera de la bolsa, cada predictor en un conjunto de ensacado se evalúa utilizando instancias en las que no se entrenó (se mantuvieron). Esto hace posible tener una evaluación bastante imparcial del conjunto sin la necesidad de un conjunto de validación adicional. Por lo tanto, tiene más instancias disponibles para entrenamiento y su conjunto puede funcionar un poco mejor.

5. Cuando está cultivando un árbol en un bosque aleatorio, solo se considera un subconjunto aleatorio de las características para dividir en cada nodo. Esto también es cierto para los árboles extra, pero van un paso más allá: en lugar de buscar los mejores umbrales posibles, como hacen los árboles de decisión normales, utilizan umbrales aleatorios para cada característica. Esta aleatoriedad adicional actúa como una forma de regularización: si un bosque aleatorio sobreajusta los datos de entrenamiento, Extra-Trees podría funcionar mejor. Además, dado que los árboles extra no buscan los mejores umbrales posibles, son mucho más rápidos de entrenar que los bosques aleatorios. Sin embargo, no son ni más rápidos ni más lentos que los bosques aleatorios al hacer predicciones.

6. Si su conjunto AdaBoost no se ajusta a los datos de entrenamiento, puede intentar aumentar el número de estimadores o reducir los hiperparámetros de regularización del estimador base. También puede intentar aumentar ligeramente la tasa de aprendizaje.
7. Si su conjunto Gradient Boosting se adapta al conjunto de entrenamiento, debería intentar disminuir la tasa de aprendizaje. También puede utilizar la parada temprana para encontrar el número correcto de predictores (probablemente tenga demasiados).

Para conocer las soluciones a los ejercicios 8 y 9, consulte los cuadernos de Jupyter disponibles en *https://github.com/ageron/handson-ml* .