

Inteligencia Artificial aplicada a Negocios y Empresas

Traducido al castellano por Juan Gabriel Gomila

Hadelin de Ponteves y Kirill Ermenko

2020-04-09

Inteligencia Artificial aplicada a Negocios

Resuelve problemas de negocios reales con soluciones de IA

de Hadelin de Ponteves y Kirill Eremenko

Traducido por Juan Gabriel Gomila



Tabla de Contenidos

Introducción	5
1 Optimización de Procesos	7
1.1 Caso Práctico: Optimización de tareas en un almacén de comercio electrónico	7
1.2 Solución de Inteligencia Artificial	14
1.3 Implementación	19
2 Minimización de Costes	29
2.1 Caso Práctico: Minimización de Costes en el Consumo Energético de un Centro de Datos	29
2.2 Solución de IA	36
2.3 Implementation	39
2.4 Resumen: El Algoritmo General de IA	53
3 Maximización de Beneficios Revenues	55
3.1 Caso Práctico: Maximización de beneficios de un negocio de venta online en línea	55
3.2 Solución de IA	58
3.3 Implementación	59
Conclusión	67
4 Anexos adicionales	69
4.1 Anexo 1: Redes Neuronales Artificiales	69
4.2 Anexo 2: Tres modelos adicionales de IA	97
4.3 Anexo 3: Preguntas y Respuestas	107

Introducción

Este libro es el complemento de nuestro curso online *Inteligencia Artificial aplicada a Negocios y Empresas* disponible en Udemy. Cubre los tres casos prácticos de empresa reales que se resuelven, cada uno de ellos, con una técnica diferente del mundo de la inteligencia artificial:

Part	Case Study	AI Solution
1 - Optimización de Procesos	Optimizar el flujo en los almacenes de un ecommerce	Q-Learning
2 - Minimización de Costes	Minimizar los gastos de energía del servidor de un centro de datos	Deep Q-Learning
3 - Maximización de Beneficios	Mazimizar los beneficios de un negocio online de venta al por menor	Muestreo Thompson

En cada una de esas partes, seguiremos el mismo esquema de tres pasos:

- **Caso Práctico.** Comenzaremos explicando el problema que tenemos que resolver y construiremos desde cero el entorno en el que trabajaremos.
- **AI Solution.** Te daremos no solo la teoría general sino también todos los detalles matemáticos del modelo de IA que resolverán el caso práctico
- **Implementation.** Implementaremos toda la solución de inteligencia artificial en Python, lo pondremos en producción y mejoraremos la inteligencia artificial al completar tareas adicionales al final.

¡Muchas gracias a todos por unirte a nosotros en este curso, te deseo sin duda un emocionante viaje al mundo de la IA aplicada a Negocios y Empresas con nosotros!

Chapter 1

Optimización de Procesos

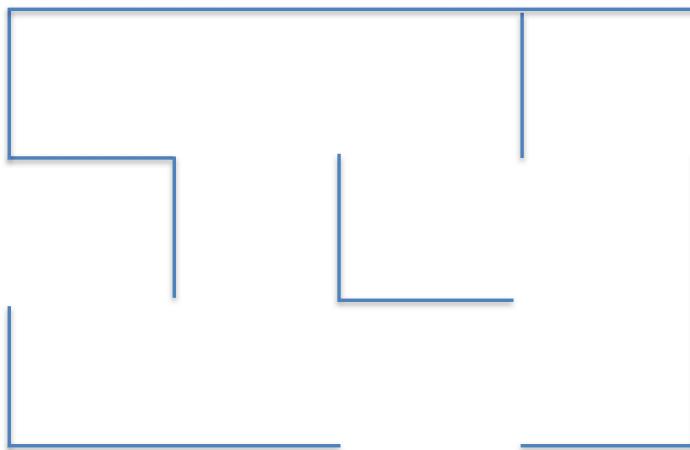
Aquí vamos con nuestro primer caso práctico y nuestro primer modelo de IA. ¡Esperamos que estés listo!

1.1 Caso Práctico: Optimización de tareas en un almacén de comercio electrónico

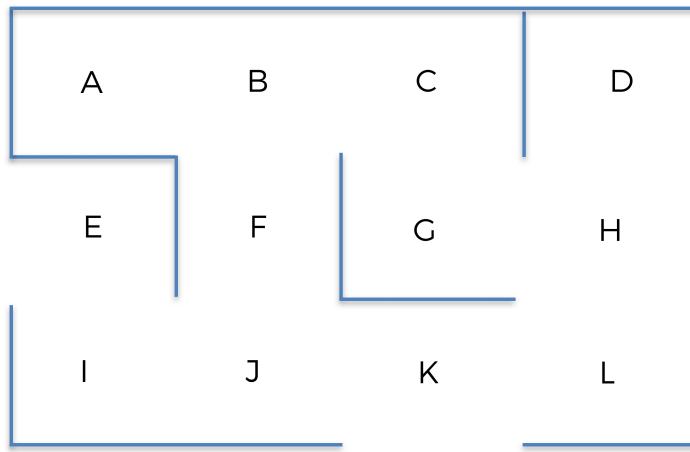
1.1.1 Problema a resolver

El problema a resolver será optimizar los flujos dentro del siguiente almacén:

Case Study #1 - Optimizing Warehouse Flows



Case Study #1 - Optimizing Warehouse Flows



Artificial Intelligence for Business

© SuperDataScience

A medida que los clientes hacen los pedidos online, un robot de almacén autónomo se mueve por el almacén para recoger los productos para futuras entregas. Así es como se ve:

Las 12 ubicaciones están conectadas a un sistema informático, que clasifica en tiempo real las prioridades de recolección de productos para estas 12 ubicaciones. Por ejemplo, en un momento específico t , devolverá la siguiente clasificación:

Rango de Prioridad	Ubicación
1	G
2	K
3	L
4	J
5	A
6	I
7	H
8	C
9	B
10	D
11	F
12	E

La ubicación G tiene prioridad 1, lo que significa que es la máxima prioridad, ya que contiene un producto que debe recogerse y entregarse de inmediato. Nuestro robot de almacén autónomo debe moverse a la ubicación G por la ruta más corta, dependiendo de dónde se encuentre. Nuestro objetivo es construir una IA que regrese esa ruta más corta, donde sea que esté el robot. Pero luego, como vemos, las ubicaciones K y L están en las 3 prioridades principales. Por lo tanto, querremos implementar una opción para que nuestro Robot de almacén autónomo pase por algunas ubicaciones intermedias antes de llegar a su ubicación final de máxima prioridad.



Figure 1.1: Robot de Almacen Autónomo

La forma en que el sistema calcula las prioridades de las ubicaciones está fuera del alcance de este caso práctico. La razón de esto es que puede haber muchas formas, desde reglas o algoritmos simples, hasta cálculos deterministas y aprendizaje automático. Pero la mayoría de estas formas no serían inteligencia artificial como la conocemos hoy. En lo que realmente queremos centrarnos es en la IA central, que abarca Q-Learning, Deep Q-Learning y otras ramas de Reinforcement Learning. Entonces, solo diremos, por ejemplo, que la ubicación G es la máxima prioridad porque uno de los clientes de platino más leales de la compañía hizo un pedido urgente de un producto almacenado en la ubicación G, que por lo tanto debe entregarse lo antes posible.

Por lo tanto, en conclusión, nuestra misión es construir una IA que siempre tome la ruta más corta a la ubicación de máxima prioridad, sea cual sea la ubicación desde la que comienza, y tener la opción de ir a una ubicación intermedia que se encuentre entre las 3 prioridades principales.

1.1.2 Entorno a definir

Al construir una IA, lo primero que siempre tenemos que hacer es definir el entorno. Y definir un entorno siempre requiere los tres elementos siguientes:

- Definir los estados
- Definir las acciones
- Definir las recompensas

Definamos estos tres elementos, uno por uno.

Definir los estados.

Comencemos con los estados. El estado de entrada es simplemente la ubicación donde está nuestro Robot de almacén autónomo en cada momento t . Sin embargo, dado que construiremos nuestra IA con ecuaciones matemáticas, codificaremos los nombres de las ubicaciones (A, B, C, ...) en números de índice, con respecto a la siguiente asignación:

Ubicación	Estado
A	0
B	1
C	2
D	3
E	4
F	5
G	6
H	7
I	8
J	9
K	10
L	11

Hay una razón específica por la que codificamos los estados con índices del 0 al 11, en lugar de otros enteros. La razón es que trabajaremos con matrices, una matriz de recompensas y una matriz de valores Q, y cada línea y columna de estas matrices corresponderá a una ubicación específica. Por ejemplo, la primera línea de cada matriz, que tiene el índice 0, corresponde a la ubicación A. La segunda línea / columna, que tiene el índice 1, corresponde a la ubicación B. Etc. Veremos el propósito de trabajar con matrices con más detalles.

un poco más tarde.

Definir las acciones.

Ahora definamos las posibles acciones a realizar. Las acciones son simplemente los siguientes movimientos que el robot puede hacer para ir de un lugar a otro. Entonces, por ejemplo, digamos que el robot está en la ubicación J, las posibles acciones que el robot puede llevar a cabo es ir a I, F o K. Y nuevamente, ya que trabajaremos con ecuaciones matemáticas, codificaremos estas acciones con los mismos índices que para los estados. Por lo tanto, siguiendo nuestro mismo ejemplo donde el robot está en la ubicación J en un momento específico, las posibles acciones que el robot puede jugar son, de acuerdo con nuestro mapeo anterior: 5, 8 y 10. De hecho, el índice 5 corresponde a F, el índice 8 corresponde a I y el índice 10 corresponde a K. Por lo tanto, eventualmente, la lista total de acciones que la IA puede llevar a cabo en general es la siguiente:

$$actions = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]$$

Obviamente, al estar en una ubicación específica, hay algunas acciones que el robot no puede llevar a cabo. Tomando el mismo ejemplo anterior, si el robot está en la ubicación J, puede ejecutar las acciones 5, 8 y 10, pero no puede ejecutar las otras acciones. Nos aseguraremos de especificar eso al atribuir una recompensa 0 a las acciones que no puede llevar a cabo, y una recompensa 1 a las acciones que si puede realizar. Y eso nos lleva a las recompensas.

Definir las recompensas.

Lo último que tenemos que hacer ahora para construir nuestro entorno es definir un sistema de recompensas. Más específicamente, tenemos que definir una función de recompensa R que toma como entradas un estado s y una acción a , y devuelve una recompensa numérica que la IA obtendrá al llevar a cabo la acción a en el estado s :

$$R : (\text{state}, \text{action}) \mapsto r \in \mathbb{R}$$

Entonces, ¿cómo vamos a construir esa función para nuestro caso práctico? Aquí esto es simple. Dado que hay un número discreto y finito de estados (los índices de 0 a 11), así como un número discreto y finito de acciones (mismos índices de 0 a 11), la mejor manera de construir nuestra función de recompensa R es simplemente hacer una matriz. Nuestra función de recompensa será exactamente una matriz de 12 filas y 12 columnas, donde las filas corresponden a los estados y las columnas corresponden a las acciones. De esa forma, en nuestra función $R: (s, a) \mapsto r$, s será el índice de la fila de la matriz, a será el índice de la columna de matriz, y r será la celda de los índices (s, a) en la matriz.

Por lo tanto, lo único que tenemos que hacer ahora para definir nuestra función de recompensa es simplemente llenar esta matriz con las recompensas numéricas. Y como acabamos de decir en el párrafo anterior, lo que tenemos que hacer primero es atribuir, para cada una de las 12 ubicaciones, una recompensa 0 por las acciones que el robot no puede ejecutar, y una recompensa 1 por las acciones que el robot puede llevar a cabo. Al hacer eso para cada una de las 12 ubicaciones, terminaremos con una matriz de recompensas. Vamos a construirlo paso a paso, comenzando con la primera ubicación:

Ubicación A.

Cuando se encuentra en la ubicación A, el robot solo puede ir a la ubicación B. Por lo tanto, dado que la ubicación A tiene el índice 0 (primera fila de la matriz) y la ubicación B tiene el índice 1 (segunda columna de la matriz), la primera fila de la matriz de las recompensas obtendrá un 1 en la segunda columna y un 0 en todas las otras columnas, así:

Case Study #1 - Optimizing Warehouse Flows

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B												
C												
D												
E												
F												
G												
H												
I												
J												
K												
L												

Defining the Rewards:

Artificial Intelligence for Business

© SuperDataScience

Ubicación B.

Al estar en la ubicación B, el robot solo puede ir a tres ubicaciones diferentes: A, C y F. Dado que B tiene el índice 1 (segunda fila), y A, C, F tienen los índices respectivos 0, 2, 5 (1ra, 3ra. , y sexta columna), entonces la segunda fila de la matriz de recompensas obtendrá un 1 en las columnas 1a, 3a y 6a, y 0 en todas las otras columnas. Por lo tanto obtenemos:

Case Study #1 - Optimizing Warehouse Flows

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C												
D												
E												
F												
G												
H												
I												
J												
K												
L												

Defining the Rewards:

Artificial Intelligence for Business

© SuperDataScience

Ubicación C.

Ocurre lo mismo, la ubicación C (de índice 2) solo está conectada a B y G (de índices 1 y 6), por lo que la tercera fila de la matriz de recompensas es:

Case Study #1 - Optimizing Warehouse Flows

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0
D												
E												
F												
G												
H												
I												
J												
K												
L												

Artificial Intelligence for Business

© SuperDataScience

En el resto de ubicaciones...

Al hacer lo mismo para todas las demás ubicaciones, finalmente obtenemos nuestra matriz final de recompensas:

Case Study #1 - Optimizing Warehouse Flows

Defining the Rewards:

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0
D	0	0	0	0	0	0	0	1	0	0	0	0
E	0	0	0	0	0	0	0	0	1	0	0	0
F	0	1	0	0	0	0	0	0	0	1	0	0
G	0	0	1	0	0	0	0	1	0	0	0	0
H	0	0	0	1	0	0	1	0	0	0	0	1
I	0	0	0	0	1	0	0	0	0	1	0	0
J	0	0	0	0	0	1	0	0	1	0	1	0
K	0	0	0	0	0	0	0	0	0	1	0	1
L	0	0	0	0	0	0	0	1	0	0	1	0

Artificial Intelligence for Business

© SuperDataScience

Felicidades, acabamos de definir las recompensas. Lo hicimos simplemente construyendo esta matriz de recompensas. Es importante entender que esta es la forma en que definimos el sistema de recompensas

cuando hacemos Q-Learning con un número finito de entradas y acciones. En el Caso Práctico 2, veremos que procederemos de manera muy diferente.

Ya casi hemos terminado, lo único que tenemos que hacer es atribuir grandes recompensas a las ubicaciones de mayor prioridad. Esto lo hará el sistema informático que devuelve las prioridades de recolección de productos para cada una de las 12 ubicaciones. Por lo tanto, dado que la ubicación G es la máxima prioridad, el sistema informático actualizará la matriz de recompensas atribuyendo una alta recompensa en la celda (G, G):

Case Study #1 - Optimizing Warehouse Flows

	A	B	C	D	E	F	G	H	I	J	K	L
A	0	1	0	0	0	0	0	0	0	0	0	0
B	1	0	1	0	0	1	0	0	0	0	0	0
C	0	1	0	0	0	0	1	0	0	0	0	0
D	0	0	0	0	0	0	0	1	0	0	0	0
E	0	0	0	0	0	0	0	0	1	0	0	0
F	0	1	0	0	0	0	0	0	0	1	0	0
G	0	0	1	0	0	0	1000	1	0	0	0	0
H	0	0	0	1	0	0	1	0	0	0	0	1
I	0	0	0	0	1	0	0	0	0	1	0	0
J	0	0	0	0	0	1	0	0	1	0	1	0
K	0	0	0	0	0	0	0	0	0	1	0	1
L	0	0	0	0	0	0	0	1	0	0	1	0

Defining the Rewards:

Artificial Intelligence for Business

© SuperDataScience

Y así es como el sistema de recompensas funcionará con Q-Learning. Atribuimos la recompensa más alta (aquí 1000) a la ubicación de máxima prioridad G. Luego puedes ver en las clases de vídeo del curso cómo podemos atribuir una recompensa más alta a la segunda ubicación de mayor prioridad (ubicación K), para hacer que nuestro robot pase por esto ubicación intermedia de máxima prioridad, optimizando así los flujos de movimiento por el almacén.

1.2 Solución de Inteligencia Artificial

La solución de IA que resolverá el problema descrito anteriormente es un modelo de Q-Learning. Dado que este último se basa en los procesos de decisión de Markov, o MDP, comenzaremos explicando cuáles son, y luego pasaremos a la intuición y los detalles matemáticos detrás del modelo Q-Learning.

1.2.1 Proceso de Decisión de Markov

Un Proceso de Decisión de Markov es una tupla (S, A, T, R) donde:

- S es el conjunto de los diferentes estados. Por lo tanto, en nuestro caso de estudio:

$$S = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

-
- A es el conjunto de las diferentes acciones que se pueden llevar a cabo en cada momento t . Por lo tanto, en nuestro caso de estudio:

$$A = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$$

- T es la llamada regla de transición:

$$T : (s_t \in S, s_{t+1} \in S, a_t \in A) \mapsto \mathbb{P}(s_{t+1}|s_t, a_t)$$

donde $\mathbb{P}(s_{t+1}|s_t, a_t)$ es la probabilidad de alcanzar el estado futuro s_{t+1} cuando se lleva a cabo la acción a_t en el estado s_t . Por lo tanto, T es la distribución de probabilidad de los estados futuros en el tiempo $t + 1$ dado el estado actual y la acción ejecutada en el tiempo t . En consecuencia, podemos predecir el estado futuro s_{t+1} tomando un valor aleatorio de esa distribución T :

$$s_{t+1} \sim T(s_t, ., a_t)$$

En nuestro estudio de caso, verás a través de nuestra implementación que esta distribución T de nuestra IA simplemente será la distribución uniforme, que es una opción clásica de distribución que funciona muy bien en el marco del Q-Learning.

- R es la función de recompensas:

$$R : (s_t \in S, a_t \in A) \mapsto r_t \in \mathbb{R}$$

donde r_t es la recompensa obtenida después de ejecutar la acción a_t en el estado s_t . En nuestro caso práctico, esta función de recompensa es exactamente la matriz que definimos previamente.

Después de definir el proceso de decisión de Markov, ahora es importante recordar que se basa en el siguiente supuesto: la probabilidad del estado futuro s_{t+1} solamente depende del estado actual s_t y la acción ejecutada a_t , y bajo ningún concepto no depende de ninguno de los estados y acciones anteriores. Es decir:

$$\mathbb{P}(s_{t+1}|s_0, a_0, s_1, a_1, \dots, s_t, a_t) = \mathbb{P}(s_{t+1}|s_t, a_t)$$

En otras palabras, un proceso de decisión de Markov no tiene memoria.

Ahora repasemos lo que va a ocurrir en términos de un proceso de decisión de Markov. En cada instante t :

- La IA observa el estado actual s_t .
- La IA ejecuta la acción a_t .
- La IA recibe la recompensa $r_t = R(s_t, a_t)$.
- La IA entra en el siguiente estado s_{t+1} .

Así que la pregunta clave es:

¿Cómo sabe la IA qué acción llevar a cabo en cada instante t ?

Para responder a esta pregunta, necesitamos introducir la función de política. La función de política π es exactamente la función que, dado un estado s_t , devuelve la acción a_t :

$$\pi : s_t \in S \mapsto a_t \in A$$

Denotemos por Π el conjunto de todas las funciones de política posibles. Entonces, la elección de las mejores acciones para jugar se convierte en un problema de optimización. De hecho, se trata de encontrar la política óptima π^* que maximice la recompensa acumulada:

$$\pi^* = \operatorname{argmax}_{\pi \in \Pi} \sum_{t \geq 0} R(s_t, \pi(s_t))$$

Por lo tanto, la pregunta anterior se convierte en:

¿Cómo encontrar esta política óptima π^* ?

Aquí es donde entra en juego el Q-Learning.

1.2.2 Q-Learning

Antes de comenzar a entrar en los detalles de Q-Learning, necesitamos explicar el concepto del valor Q.

El valor Q

Para cada par de estado y acción (s, a) , vamos a asociar un valor numérico llamado $Q(s, a)$:

$$Q : (s \in S, a \in A) \mapsto Q(s, a) \in \mathbb{R}$$

Diremos que $Q(s, a)$ es *el valor Q de la acción a llevada a cabo en el estado s*.

Para comprender el propósito de este *Valor Q*, necesitamos introducir la Diferencia Temporal.

La Diferencia Temporal

Al principio $t = 0$, todos los valores Q se inicializan a 0:

$$\forall s \in S, a \in A, Q(s, a) = 0$$

Ahora supongamos que estamos en el instante t , en cierto estado s_t . Llevamos a cabo una acción aleatoria a_t , que nos lleva al estado s_{t+1} y obtenemos la recompensa $R(s_t, a_t)$.

Ahora podemos presentar la diferencia temporal, que básicamente es el corazón de Q-Learning. La diferencia temporal en el tiempo t , denotada por $TD_t(s_t, a_t)$, es la diferencia entre:

-
- $R(s_t, a_t) + \gamma \max_a(Q(s_{t+1}, a))$, es decir la recompensa $R(s_t, a_t)$ obtenida al llevar a cabo la acción a_t en el estado s_t , más el valor Q de la mejor acción jugada en el estado futuro s_{t+1} , descontado por un factor $\gamma \in [0, 1]$, llamado *factor de descuento*. Y $Q(s_t, a_t)$, es decir el valor Q de la acción a_t llevada a cabo en el estado s_t ,

que nos lleva a:

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \max_a(Q(s_{t+1}, a)) - Q(s_t, a_t)$$

- **Bien, genial, pero ¿cuál es exactamente el propósito de esta diferencia temporal $TD_t(s_t, a_t)$?**

Respondamos esta pregunta para darnos una mejor idea de la IA. $TD_t(s_t, a_t)$ es como una recompensa intrínseca. La IA aprenderá los valores Q de tal manera que:

- Si $TD_t(s_t, a_t)$ es alta, la IA recibe una *buenas sorpresa*.
- Si $TD_t(s_t, a_t)$ es baja, la IA recibe *frustración*.

En ese sentido, la IA repetirá algunas actualizaciones de los valores Q (a través de una ecuación llamada la ecuación de Bellman) hacia diferencias temporales más altas.

En consecuencia, en el siguiente paso final del algoritmo Q-Learning, usamos la diferencia temporal para reforzar los pares (estado, acción) desde el tiempo $t - 1$ hasta el tiempo t , de acuerdo con la siguiente ecuación:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

donde $\alpha \in \mathbb{R}$ es la tasa de aprendizaje, que determina qué tan rápido va el aprendizaje de los valores Q o qué tan grandes son las actualizaciones de los mismos. Su valor suele ser un número real elegido entre 0 y 1, como por ejemplo 0.01, 0.05, 0.1 o 0.5. Cuanto menor sea su valor, más pequeñas serán las actualizaciones de los valores Q y más larga será la ejecución del algoritmo de Q-Learning. Cuanto mayor sea su valor, mayores serán las actualizaciones de los valores Q y más rápido será el algoritmo de Q-Learning.

Esta ecuación anterior es la ecuación de Bellman. Es el pilar fundamental del Q-Learning.

Con este punto de vista, los valores Q miden la acumulación de sorpresa o frustración asociada con el par de acciones y estados (s_t, a_t) . En el caso de recibir sorpresa, la IA se refuerza, y en el caso de recibir frustración, la IA se debilita. Por lo tanto, queremos aprender los valores Q que le darán a la IA la máxima *buenas sorpresa*.

En consecuencia, la decisión de qué acción ejecutar depende principalmente del valor Q $Q(s_t, a_t)$. Si la acción a_t ejecutada en el estado s_t está asociada con un valor Q alto $Q(s_t, a_t)$, la IA tendrá una mayor tendencia a elegir la acción a_t . Por otro lado, si la acción a_t que se ha llevado a cabo en el estado s_t está asociada con un valor Q pequeño $Q(s_t, a_t)$, la IA tendrá una tendencia menor a elegir la acción a_t .

Hay varias formas de elegir la mejor acción para ejecutar en cada estado. Primero, cuando estamos en cierto estado s_t , simplemente podríamos tomar la acción a_t que maximiza el valor Q $Q(s_t, a_t)$:

$$a_t = \operatorname{argmax}_a(Q(s_t, a))$$

Esta solución es el método **Argmax**.

Otra gran solución, que resulta ser una solución aún mejor para problemas complejos, es el método **Softmax**.

El método Softmax consiste en considerar para cada estado s la siguiente distribución:

$$W_s : a \in A \mapsto \frac{\exp(Q(s, a)^\tau)}{\sum_{a'} \exp(Q(s, a')^\tau)} \text{ with } \tau \geq 0$$

Luego, elegimos qué acción a llevar a cabo mediante una muestra de un valor aleatorio de esa distribución:

$$a \sim W_s(.)$$

Sin embargo, el problema que resolveremos en el Caso Práctico 1 será lo suficientemente simple como para usar el método Argmax, así que esto es lo que elegiremos.

1.2.3 El algoritmo de Q-Learning al completo

Resumamos los diferentes pasos de todo el proceso de Q-Learning:

Inicialización

Para todas las parejas de estados s y acciones a , los valores Q se inicializan a 0:

$$\forall s \in S, a \in A, Q_0(s, a) = 0$$

Comenzamos en el estado inicial s_0 . Llevamos a cabo una acción aleatoria posible y llegamos al primer estado s_1 .

Para cada instante $t \geq 1$, repetiremos un cierto número de veces (1000 veces en nuestro código) lo siguiente:

1. Seleccionamos un estado aleatorio s_t de nuestros 12 estados posibles:

$$s_t = \text{random}(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11)$$

2. Llevamos a cabo una acción aleatoria a_t que puede conducir al siguiente estado posible, es decir, de modo que $R(s_t, a_t) > 0$:

$$a_t = \text{random}(0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11) \text{ t.q. } R(s_t, a_t) > 0$$

3. Llegamos al siguiente estado s_{t+1} y obtenemos la recompensa $R(s_t, a_t)$
4. Calculamos la Diferencia Temporal $TD_t(s_t, a_t)$:

$$TD_t(s_t, a_t) = R(s_t, a_t) + \gamma \max_a (Q(s_{t+1}, a)) - Q(s_t, a_t)$$

5. Actualizamos el valor Q aplicando la ecuación de Bellman:

$$Q_t(s_t, a_t) = Q_{t-1}(s_t, a_t) + \alpha TD_t(s_t, a_t)$$

1.3 Implementación

Ahora proporcionemos y expliquemos la implementación completa de este modelo de Q-Learning, la solución de nuestro problema de optimización de flujos de almacén.

Primero, comenzamos importando las librerías que se usarán en esta implementación. Estos solo incluyen la biblioteca numpy, que ofrece una forma práctica de trabajar con matrices y operaciones matemáticas:

```
# Importar las librerías
import numpy as np
```

Luego establecemos los parámetros de nuestro modelo. Estos incluyen el factor de descuento γ y la tasa de aprendizaje α , que comovimos en la Sección 1.2, son los únicos parámetros del algoritmo Q – Learning :

```
# Configuración de los parámetros gamma y alfa para el Q-Learning
gamma = 0.75
alpha = 0.9
```

Las dos secciones de código anteriores eran simplemente las secciones introductorias, antes de comenzar realmente a construir nuestro modelo de IA. Ahora el siguiente paso es comenzar la primera parte de nuestra implementación: Parte 1 - Definición del entorno. Y para eso, por supuesto, comenzamos definiendo los estados, con un diccionario que asigna los nombres de las ubicaciones (en letras de la A a la L) en los estados (en índices del 0 al 11):

```
# PARTE 1 - DEFINICIÓN DEL ENTORNO
```

```
# Definición de los estados
location_to_state = {'A': 0,
                     'B': 1,
                     'C': 2,
                     'D': 3,
                     'E': 4,
                     'F': 5,
                     'G': 6,
                     'H': 7,
                     'I': 8,
                     'J': 9,
                     'K': 10,
                     'L': 11}
```

Luego definimos las acciones, con una simple lista de índices del 0 al 11. Recuerda que cada índice de acción corresponde al siguiente estado (siguiente ubicación) al que conduce dicha acción:

```
# Definición de las acciones
actions = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11]
```

Y eventualmente, definimos las recompensas, creando una matriz de recompensas, donde las filas corresponden a los estados actuales s_t , las columnas corresponden a las acciones a_t que conducen al siguiente estado s_{t+1} , y las celdas contienen las recompensas $R(s_t, a_t)$. Si una celda (s_t, a_t) tiene un 1, eso significa que podemos llevar a cabo la acción a_t del estado actual s_t para llegar al siguiente estado s_{t+1} . Si una celda (s_t, a_t) tiene un 0, eso significa que no podemos llevar a cabo la acción a_t del estado actual s_t para llegar a cualquier estado siguiente s_{t+1} . Y por ahora colocaremos manualmente una alta recompensa (1000) dentro de la celda correspondiente a la ubicación G, porque es la ubicación de máxima prioridad donde el almacén autónomo tiene que ir a

recoger los productos. Como la ubicación G ha codificado el estado como índice 6, colocamos una recompensa de 1000 en la celda de la fila 6 y la columna 6. Luego, mejoraremos nuestra solución al implementar una forma automática de ir a la ubicación de máxima prioridad, sin tener que actualizar manualmente la matriz de recompensas y dejándola inicializada con 0s y 1s como debería ser. Pero mientras tanto, aquí está debajo de nuestra matriz de recompensas, incluida la actualización manual:

```
# Definición de las recompensas
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0,0],
              [0,0,1,0,0,0,1000,1,0,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1,0],
              [0,0,0,0,1,0,0,0,0,1,0,0,0],
              [0,0,0,0,0,1,0,0,1,0,1,0,0],
              [0,0,0,0,0,0,0,0,0,1,0,1,0],
              [0,0,0,0,0,0,0,1,0,0,1,0,0]])
```

Eso finaliza esta primera parte. Ahora comenzemos la segunda parte de nuestra implementación: Parte 2 - Construcción la solución de IA con Q-Learning. En ese sentido, vamos a seguir el algoritmo de Q-Learning exactamente como lo vimos en la Sección 1.2. Por lo tanto, primero inicializamos todos los valores Q, creando nuestra matriz de valores Q llena de ceros (en los cuales, las filas corresponden a los estados actuales s_t , las columnas corresponden a las acciones a_t que conducen al siguiente estado s_{t+1} , y las celdas contienen los valores Q, $Q(s_t, a_t)$):

```
# PARTE 2 - CONSTRUCCIÓN DE LA SOLUCIÓN DE IA CON Q-LEARNING

# Inicialización de los valores Q
Q = np.array(np.zeros([12,12]))
```

Luego, por supuesto, implementamos el proceso de Q-Learning, con un bucle `for` que llevará a cabo un total de 1000 iteraciones, repitiendo 1000 veces los pasos del proceso de Q-Learning que analizamos a fondo al final de la Sección 1.2:

```
# Implementación del proceso de Q-Learning
for i in range(1000):
    current_state = np.random.randint(0,12)
    playable_actions = []
    for j in range(12):
        if R[current_state, j] > 0:
            playable_actions.append(j)
    next_state = np.random.choice(playable_actions)
    TD = R[current_state, next_state] + gamma*Q[next_state, np.argmax(Q[next_state, :])]
    - Q[current_state, next_state]
    Q[current_state, next_state] = Q[current_state, next_state] + alpha*TD
```

Opcional: en esta etapa del código, nuestra matriz de valores Q está lista. Podemos echarle un vistazo ejecutando todo el código que hemos implementado hasta ahora e ingresando las siguientes dos instrucciones en la consola:

```
print("Q-Values:")
print(Q.astype(int))
```

Y obtenemos la siguiente matriz de valores Q finales:

```
In [2]: print("Q-Values:")
...: print(Q.astype(int))
Q-Values:
[[ 0 1661 0 0 0 0 0 0 0 0 0 0 0]
 [1246 0 2213 0 0 1246 0 0 0 0 0 0 0]
 [ 0 1661 0 0 0 0 2970 0 0 0 0 0 0]
 [ 0 0 0 0 0 0 0 2225 0 0 0 0 0]
 [ 0 0 0 0 0 0 0 0 703 0 0 0 0]
 [ 0 1661 0 0 0 0 0 0 0 931 0 0 0]
 [ 0 0 2213 0 0 0 3968 2225 0 0 0 0 0]
 [ 0 0 0 1661 0 0 2968 0 0 0 0 0 1670]
 [ 0 0 0 0 528 0 0 0 0 936 0 0 0]
 [ 0 0 0 0 0 1246 0 0 703 0 1246 0]
 [ 0 0 0 0 0 0 0 0 0 936 0 1661]
 [ 0 0 0 0 0 0 0 2225 0 0 1246 0]]
```

Para una mayor claridad visual, incluso puede verificarse la matriz de valores Q directamente en el Explorador de variables, haciendo doble clic en la variable Q. Luego, para obtener los valores Q como enteros, es conveniente hacer clic en Formato e ingresar un formato de como flotante como %.0f. Se obtiene en este caso el siguiente resultado, que es un poco más claro ya que se pueden ver en la matriz Q los índices de las filas y columnas de la misma:

	0	1	2	3	4	5	6	7	8	9	10	11
0	0	1661	0	0	0	0	0	0	0	0	0	0
1	1247	0	2214	0	0	1247	0	0	0	0	0	0
2	0	1661	0	0	0	0	2970	0	0	0	0	0
3	0	0	0	0	0	0	0	2226	0	0	0	0
4	0	0	0	0	0	0	0	0	703	0	0	0
5	0	1661	0	0	0	0	0	0	0	931	0	0
6	0	0	2214	0	0	0	3968	2226	0	0	0	0
7	0	0	0	1661	0	0	2968	0	0	0	0	1670
8	0	0	0	0	528	0	0	0	0	936	0	0
9	0	0	0	0	0	1247	0	0	703	0	1246	0
10	0	0	0	0	0	0	0	0	0	936	0	1661
11	0	0	0	0	0	0	0	2226	0	0	1247	0

Bien, ahora que tenemos nuestra matriz de valores Q, ¡estamos listos para llevarlo a producción! Por lo tanto, podemos pasar a la tercera parte de la implementación, Parte 3: Poner el modelo en producción, dentro de la cual calcularemos la ruta óptima desde cualquier ubicación inicial a cualquier ubicación final de máxima prioridad. La idea aquí será implementar una función de *ruta*, que tomará como entradas la ubicación de inicio donde se encuentra nuestro robot de almacén autónomo en un momento específico y la ubicación de finalización donde tiene que ir con la máxima prioridad, y eso volverá como genera la ruta más corta dentro de una lista. Sin embargo, dado que queremos indicar las ubicaciones con sus nombres (en letras), a diferencia de sus estados (en índices), necesitaremos un diccionario que asigne los estados de ubicaciones (en índices) a los nombres de ubicaciones (en letras). Y eso es lo primero que haremos aquí en esta tercera parte, usando un truco para invertir nuestro diccionario anterior `location\to_state`, ya que de hecho simplemente queremos obtener el mapeo inverso exacto de este diccionario:

```
# PARTE 3 - PONER EL MODELO EN PRODUCCIÓN
```

```
# Hacer un mapeo de los estados a las ubicaciones
```

```
state_to_location = {state: location for location, state in location_to_state.items() }
```

Aquí es cuando entra en juego la sección de código más importante. Estamos a punto de implementar la función final `route()` que tomará como entradas las ubicaciones de inicio y finalización, y que devolverá la ruta óptima entre estas dos ubicaciones. Para explicar exactamente qué hará esta función de ruta, enumeraremos los diferentes pasos del proceso, al pasar de la ubicación E a la ubicación G:

1. Comenzamos en nuestra ubicación inicial E.
2. Obtenemos el estado de ubicación E, que según nuestro mapeo `location_to_state` es $s_0 = 4$.
3. En la fila del estado $s_0 = 4$ de nuestra matriz de valores Q, hallamos la columna con el mayor valor Q (703).
4. Esta columna tiene el índice 8, por lo que ejecutamos la acción del índice 8 que nos lleva al siguiente estado $s_{t+1} = 8$.
5. Obtenemos la ubicación del estado 8, que según nuestro mapeo `state_to_location` es la ubicación I. Por lo tanto, nuestra próxima ubicación es la ubicación I, que se adjunta a nuestra lista que contiene la ruta óptima global.
6. Repetimos los mismos 5 pasos anteriores desde nuestra nueva ubicación inicial I, hasta llegar a nuestro destino final, la ubicación G.

Por lo tanto, dado que no sabemos cuántas ubicaciones tendremos que atravesar entre las ubicaciones inicial y final, tenemos que hacer un bucle `while` que repetirá el proceso de 5 pasos descrito anteriormente, y que se detendrá tan pronto como lo hagamos llegar a la ubicación final de máxima prioridad:

```
# Hacer la función final que devolverá la ruta óptima
def route(starting_location, ending_location):
    route = [starting_location]
    next_location = starting_location
    while (next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state, :])
        next_location = state_to_location[next_state]
        route.append(next_location)
        starting_location = next_location
    return route
```

¡Felicitaciones, nuestra herramienta ya está lista! Cuando lo probamos para ir de E a G, obtenemos las dos rutas óptimas posibles después de imprimir la ruta final ejecutando el código completo varias veces:

```
# Imprimir la ruta final
```

```
print('Route:')
route('E', 'G')
```

```
Route:
```

```
Out[1]: ['E', 'I', 'J', 'F', 'B', 'C', 'G']
```

```
Out[2]: ['E', 'I', 'J', 'K', 'L', 'H', 'G']
```

Bien, tenemos una primera versión del modelo que funciona bien. Pero podemos mejorarlo de dos maneras. Primero, al automatizar la atribución de recompensas a la ubicación de máxima prioridad, para que no tengamos que hacerlo manualmente. Y segundo, al agregar una función que nos da la opción de ir a una

ubicación intermedia antes de ir a la ubicación de máxima prioridad. Esa ubicación intermedia debe estar, por supuesto, en las 3 ubicaciones prioritarias principales. Y, de hecho, en nuestra clasificación de ubicaciones de máxima prioridad, la segunda ubicación de máxima prioridad es la ubicación K. Por lo tanto, para optimizar aún más los flujos de almacén, nuestro robot de almacén autónomo debe ir por la ubicación K para recoger los productos en su camino a la ubicación de máxima prioridad G. Una forma de hacer esto es tener la opción de ir a cualquier ubicación intermedia en el proceso de nuestra función `route()`. Y esto es exactamente lo que implementaremos como segunda mejora. Pero primero, implementemos la primera mejora, que automatiza la atribución de recompensas.

La forma de hacerlo es en dos pasos: primero debemos hacer una copia (llamada `R_new`) de nuestra matriz de recompensa dentro de la cual la función `route()` actualizará automáticamente la recompensa en la celda de la ubicación final. De hecho, la ubicación final es una de las entradas de la función `route()`, por lo que al usar nuestro diccionario de `location_to_state` podemos encontrar fácilmente esa celda y actualizar su recompensa a 1000. Y segundo, debemos incluir toda la lógica del algoritmo de Q-learning (incluido el paso de inicialización) dentro de la función de ruta, justo después de hacer esa actualización de la recompensa en nuestra copia de la matriz de recompensas. De hecho, en nuestra implementación anterior anterior, el proceso de Q-Learning ocurre en la versión original de la matriz de recompensas, que ahora se supone que permanece como está, es decir, se inicializa solo a 1s y 0s. Por lo tanto, debemos incluir el proceso de Q-Learning dentro de la función de ruta y hacer que suceda en nuestra copia `R_new` de la matriz de recompensas, en lugar de la matriz de recompensas original `R`. Por lo tanto, nuestra implementación completa se convierte en lo siguiente:

```
# Inteligencia Artificial aplicada a Negocios y Empresas

# Optimización de Procesos en un almacén con Q-Learning

# # Importar las librerías
import numpy as np

# Configuración de los parámetros gamma y alfa para el Q-Learning
gamma = 0.75
alpha = 0.9

# PARTE 1 - DEFINICIÓN DEL ENTORNO

# Definición de los estados
location_to_state = {'A': 0,
                     'B': 1,
                     'C': 2,
                     'D': 3,
                     'E': 4,
                     'F': 5,
                     'G': 6,
                     'H': 7,
                     'I': 8,
                     'J': 9,
                     'K': 10,
                     'L': 11}

# Definición de las acciones
actions = [0,1,2,3,4,5,6,7,8,9,10,11]
```

```

# Definición de las recompensas
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0],
              [0,0,1,0,0,0,1,1,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0,0],
              [0,0,0,0,0,1,0,0,1,0,1,0],
              [0,0,0,0,0,0,0,0,1,0,1,0],
              [0,0,0,0,0,0,0,0,1,0,1,0]]))

# PARTE 2 - CONSTRUCCIÓN DE LA SOLUCIÓN DE IA CON Q-LEARNING

# Crear una función de mapping desde los estados a las ubicaciones
state_to_location = {state: location for location, state in location_to_state.items()}

# Crear una función que devuelva el camino más corto desde la ubicación inicial a la final
def route(starting_location, ending_location):
    R_new = np.copy(R)
    ending_state = location_to_state[ending_location]
    R_new[ending_state, ending_state] = 1000
    Q = np.array(np.zeros([12,12]))
    for i in range(1000):
        current_state = np.random.randint(0,12)
        playable_actions = []
        for j in range(12):
            if R_new[current_state, j] > 0:
                playable_actions.append(j)
        next_state = np.random.choice(playable_actions)
        TD = R_new[current_state, next_state]
        + gamma * Q[next_state, np.argmax(Q[next_state, :])]
        - Q[current_state, next_state]
        Q[current_state, next_state] = Q[current_state, next_state] + alpha * TD
    route = [starting_location]
    next_location = starting_location
    while (next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state,:])
        next_location = state_to_location[next_state]
        route.append(next_location)
        starting_location = next_location
    return route

# PARTE 3 - PONER EL MODELO EN PRODUCCIÓN

```

```
# Imprimir la ruta final
print('Route:')
route('E', 'G')
```

Al ejecutar este nuevo código varias veces, obtenemos, por supuesto, las mismas dos posibles rutas óptimas que antes.

Ahora abordemos la segunda mejora. Hay tres formas de agregar la opción de ir por la ubicación intermedia K, la segunda ubicación de máxima prioridad:

- Otorgamos una alta recompensa a la acción que lleva de la ubicación J a la ubicación K. Esta alta recompensa debe ser mayor que 1 y menor a 1000. De hecho, debe ser mayor que 1 para que el proceso de Q-Learning favorezca la acción que lleva de J a K, en oposición a la acción que lleva de J a F que tiene la recompensa 1. Y debe ser inferior a 1000, por lo que debemos mantener la recompensa más alta en la ubicación de mayor prioridad para asegurarnos de que terminemos allí. Por lo tanto, por ejemplo, en nuestra matriz de recompensas podemos dar una alta recompensa de 500 a la celda en la fila del índice 9 y la columna del índice 10, ya que de hecho esa celda corresponde a la acción que conduce desde la ubicación J (índice de estado 9) a ubicación K (índice de estado 10). De esa manera, nuestro robot de almacén autónomo siempre irá por la ubicación K en su camino hacia la ubicación G. Así es como sería la matriz de recompensas en ese caso:

```
# Definición de las recompensas
R = np.array([[0,1,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0],
              [0,1,0,0,0,0,0,0,0,1,0],
              [0,0,1,0,0,0,1,1,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0],
              [0,0,0,0,0,1,0,0,1,0,500],
              [0,0,0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,0,1,0,0,1,0]])
```

- Damos una mala recompensa a la acción que lleva de la ubicación J a la ubicación F. Esta mala recompensa solo tiene que ser inferior a 0. De hecho, al castigar esta acción con una mala recompensa, el proceso de Q-Learning nunca favorecerá esa acción que lleva de J a F. Por lo tanto, por ejemplo, en nuestra matriz de recompensas podemos dar una mala recompensa de -500 a la celda en la fila del índice 9 y la columna del índice 5, ya que esa celda corresponde a la acción que conduce desde la ubicación J (estado con índice 9) a la ubicación F (estado con índice 5). De esa manera, nuestro robot de almacén autónomo nunca pasará por la ubicación F en su camino hacia la ubicación G. Así es como sería la matriz de recompensas en ese caso:

```
# Definición de las recompensas
R = np.array([[0,1,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0],
              [0,1,0,0,0,0,0,0,0,1,0],
              [0,0,1,0,0,0,1,1,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0],
              [0,0,0,0,0,1,0,0,1,0,-500],
              [0,0,0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,0,1,0,0,1,0]])
```

```
[0,0,0,1,0,0,1,0,0,0,0,1],
[0,0,0,0,1,0,0,0,0,1,0,0],
[0,0,0,0,0,-500,0,0,1,0,1,0],
[0,0,0,0,0,0,0,0,0,1,0,1],
[0,0,0,0,0,0,1,0,0,1,0]])
```

3. Realizamos una función adicional `best_route()`, tomando como entradas las tres ubicaciones inicial, intermedia y final, que llamará a nuestra función `route()` anterior dos veces, una primera vez desde la ubicación inicial a la ubicación intermedia, y un segunda vez desde la ubicación intermedia hasta la ubicación final.

Las dos primeras ideas son fáciles de implementar manualmente, pero muy difíciles de implementar automáticamente. De hecho, es fácil encontrar automáticamente el índice de la ubicación intermedia donde queremos ir, pero es muy difícil obtener el índice de la ubicación que lleva a esa ubicación intermedia, ya que depende de la ubicación inicial y la ubicación final. Puedes intentar implementar la primera o la segunda idea, y verás lo que quiero decir. En consecuencia, implementaremos la tercera idea, que puede programarse en solo dos líneas adicionales de código:

```
# Hacer la función final que devuelve la ruta óptima
def best_route(starting_location, intermediary_location, ending_location):
    return route(starting_location, intermediary_location)
        + route(intermediary_location, ending_location) [1:]
```

Finalmente, el código final que incluye esa mejora importante para la optimización de los flujos de nuestro almacén se convierte en:

```
# Inteligencia Artificial aplicada a Negocios y Empresas
# Optimización de Procesos en un almacén con Q-Learning

# # Importar las librerías
import numpy as np

# Configuración de los parámetros gamma y alfa para el Q-Learning
gamma = 0.75
alpha = 0.9

# PARTE 1 - DEFINICIÓN DEL ENTORNO

# Definición de los estados
location_to_state = {'A': 0,
                     'B': 1,
                     'C': 2,
                     'D': 3,
                     'E': 4,
                     'F': 5,
                     'G': 6,
                     'H': 7,
                     'I': 8,
                     'J': 9,
                     'K': 10,
                     'L': 11}
```

```

# Definición de las acciones
actions = [0,1,2,3,4,5,6,7,8,9,10,11]

# Definición de las recompensas
R = np.array([[0,1,0,0,0,0,0,0,0,0,0,0],
              [1,0,1,0,0,1,0,0,0,0,0,0],
              [0,1,0,0,0,0,1,0,0,0,0,0],
              [0,0,0,0,0,0,0,1,0,0,0,0],
              [0,0,0,0,0,0,0,0,1,0,0,0],
              [0,1,0,0,0,0,0,0,0,1,0,0],
              [0,0,1,0,0,0,1,1,0,0,0,0],
              [0,0,0,1,0,0,1,0,0,0,0,1],
              [0,0,0,0,1,0,0,0,0,1,0,0],
              [0,0,0,0,0,1,0,0,1,0,1,0],
              [0,0,0,0,0,0,0,0,0,1,0,1],
              [0,0,0,0,0,0,1,0,0,1,0,0]]))

# PARTE 2 - CONSTRUCCIÓN DE LA SOLUCIÓN DE IA CON Q-LEARNING

# Crear una función de mapping desde los estados a las ubicaciones
state_to_location = {state: location for location, state in location_to_state.items()}

# Crear una función que devuelva el camino más corto desde la ubicación inicial a la final
def route(starting_location, ending_location):
    R_new = np.copy(R)
    ending_state = location_to_state[ending_location]
    R_new[ending_state, ending_state] = 1000
    Q = np.array(np.zeros([12,12]))
    for i in range(1000):
        current_state = np.random.randint(0,12)
        playable_actions = []
        for j in range(12):
            if R_new[current_state, j] > 0:
                playable_actions.append(j)
        next_state = np.random.choice(playable_actions)
        TD = R_new[current_state, next_state]
        + gamma * Q[next_state, np.argmax(Q[next_state,])]
        - Q[current_state, next_state]
        Q[current_state, next_state] = Q[current_state, next_state] + alpha * TD
    route = [starting_location]
    next_location = starting_location
    while (next_location != ending_location):
        starting_state = location_to_state[starting_location]
        next_state = np.argmax(Q[starting_state])
        next_location = state_to_location[next_state]
        route.append(next_location)
        starting_location = next_location
    return route

```

```
# PARTE 3 - PONER EL MODELO EN PRODUCCIÓN

# Crear la función final que devuelve la ruta óptima
def best_route(starting_location, intermediary_location, ending_location):
    return route(starting_location, intermediary_location)
    + route(intermediary_location, ending_location)[1:]

# Imprimir la ruta final
print('Route:')
best_route('E', 'K', 'G')
```

Al ejecutar este código completamente nuevo tantas veces como queramos, siempre obtendremos el mismo resultado esperado:

```
Best Route:
Out[1]: ['E', 'I', 'J', 'K', 'L', 'H', 'G']
```

Chapter 2

Minimización de Costes

¡Felicitaciones por darlo todo en el primer caso de estudio! Pasemos a una IA nueva y más avanzada.

2.1 Caso Práctico: Minimización de Costes en el Consumo Energético de un Centro de Datos

2.1.1 Problema a resolver

En 2016, la IA DeepMind minimizó una gran parte del costo de Google al reducir la factura de enfriamiento del centro de datos de Google en 40% utilizando su modelo de IA DQN (Deep Q-Learning). En este caso práctico, haremos algo muy similar. Configuraremos nuestro propio entorno de servidor y construiremos una IA que controlará el enfriamiento / calentamiento del servidor para que se mantenga en un rango óptimo de temperaturas mientras se ahorra la máxima energía, minimizando así los costes. Y tal como lo hizo la IA de DeepMind, nuestro objetivo será lograr al menos un 40% de ahorro de energía.

2.1.1.1 Entorno a definir

Antes de definir los estados, las acciones y las recompensas, debemos explicar cómo funciona el servidor. Lo haremos en varios pasos. Primero, enumeraremos todos los parámetros y variables del entorno por los cuales se controla el servidor. Después de eso, estableceremos la suposición esencial del problema, en la cual nuestra IA dependerá para proporcionar una solución. Luego especificaremos cómo simularemos todo el proceso. Y eventualmente explicaremos el funcionamiento general del servidor y cómo la IA desempeña su papel.

Parámetros

- la temperatura atmosférica promedio durante un mes
- el rango óptimo de temperaturas del servidor, que será $[18^{\circ}\text{C}, 24^{\circ}\text{C}]$
- la temperatura mínima del servidor por debajo de la cual no funciona, que será -20°C
- la temperatura máxima del servidor por encima de la cual no funciona, que será de 80°C
- el número mínimo de usuarios en el servidor, que será 10
- el número máximo de usuarios en el servidor, que será de 100
- el número máximo de usuarios en el servidor que puede subir o bajar por minuto, que será 5
- la tasa mínima de transmisión de datos en el servidor, que será 20
- la velocidad máxima de transmisión de datos en el servidor, que será de 300

-
- la velocidad máxima de transmisión de datos que puede subir o bajar por minuto, que será 10

Variables:

- la temperatura del servidor en cualquier momento
- la cantidad de usuarios en el servidor en cualquier momento
- la velocidad de transmisión de datos en cualquier minuto
- la energía gastada por la IA en el servidor (para enfriarlo o calentarlo) en cualquier momento
- la energía gastada por el sistema de enfriamiento integrado del servidor que automáticamente lleva la temperatura del servidor al rango óptimo cada vez que la temperatura del servidor sale de este rango óptimo

Todos estos parámetros y variables serán parte de nuestro entorno de servidor e influirán en las acciones de la IA en el servidor.

A continuación, expliquemos los dos supuestos básicos del entorno. Es importante comprender que estos supuestos no están relacionados con la inteligencia artificial, sino que se utilizan para simplificar el entorno para que podamos centrarnos al máximo en la solución de inteligencia artificial.

Suposiciones:

Nos basaremos en los siguientes dos supuestos esenciales:

Supuesto 1: la temperatura del servidor se puede aproximar mediante Regresión lineal múltiple, mediante una función lineal de la temperatura atmosférica, el número de usuarios y la velocidad de transmisión de datos:

$$\text{temp. del server} = b_0 + b_1 \times \text{temp. atmosf.} + b_2 \times \text{n. de usuarios} + b_3 \times \text{ratio de trans. de datos}$$

donde $b_0 \in \mathbb{R}$, $b_1 > 0$, $b_2 > 0$ y $b_3 > 0$.

La razón de ser de este supuesto y la razón por la cual $b_1 > 0$, $b_2 > 0$ y $b_3 > 0$ son fáciles de entender de entender. De hecho, tiene sentido que cuando la temperatura atmosférica aumenta, la temperatura del servidor aumenta. Además, cuanto más usuarios estén activos en el servidor, más gastará el servidor para manejarlos y, por lo tanto, mayor será la temperatura del servidor. Y finalmente, por supuesto, mientras más datos se transmitan dentro del servidor, más gastará el servidor para procesarlo y, por lo tanto, la temperatura más alta del servidor será. Y para fines de simplicidad, solo suponemos que estas correlaciones son lineales. Sin embargo, podría ejecutarse totalmente la misma simulación suponiendo que son cuadráticos o logarítmicos. Siéntete libre de retocar el modelo.

Finalmente, supongamos que después de realizar esta Regresión lineal múltiple, obtuvimos los siguientes valores de los coeficientes: $b_0 = 0$, $b_1 = 1$, $b_2 = 1.25$ y $b_3 = 1.25$. En consecuencia:

$$\text{temp. del server} = \text{temp. atmosf.} + 1.25 \times \text{n. de usuarios} + 1.25 \times \text{ratio de trans. de datos}$$

Supuesto 2: la energía gastada por un sistema (nuestra IA o el sistema de enfriamiento integrado del servidor) que cambia la temperatura del servidor de T_t a T_{t+1} en 1 unidad de tiempo (aquí 1 minuto), se puede aproximar nuevamente mediante regresión mediante una función lineal del cambio absoluto de temperatura del servidor:

$$E_t = \alpha |\Delta T_t| + \beta = \alpha |T_{t+1} - T_t| + \beta$$

donde:

-
- E_t es la energía gastada por el sistema en el servidor entre los tiempos t y $t + 1$,
 - ΔT_t es el cambio de temperatura del servidor causado por el sistema entre los tiempos t y $t + 1$,
 - T_t es la temperatura del servidor en el instante t ,
 - T_{t+1} es la temperatura del servidor en el instante $t + 1$,
 - $\alpha > 0$,
 - y $\beta \in \mathbb{R}$.

Nuevamente, expliquemos por qué tiene sentido intuitivamente hacer esta suposición con $\alpha > 0$. Eso es simplemente porque cuanto más se calienta la IA o se enfriá el servidor, más gasta energía para hacer esa transferencia de calor. De hecho, por ejemplo, imaginemos que el servidor de repente tiene problemas de sobrecalentamiento y acaba de alcanzar 80°C , luego, dentro de una unidad de tiempo (1 minuto), la IA necesitará mucha más energía para que la temperatura del servidor vuelva a su temperatura óptima de 24°C que devolverlo a 50°C por ejemplo. Y de nuevo por razones de simplicidad, solo suponemos que estas correlaciones son lineales. Además (en caso de que te lo estés preguntado), ¿por qué tomamos el valor absoluto? Eso es simplemente porque cuando la IA enfriá el servidor, $T_{t+1} < T_t$, entonces $\Delta T < 0$. Y, por supuesto, una energía siempre es positiva, por lo que tenemos que tomar el valor absoluto de ΔT .

Finalmente, para mayor simplicidad, también asumiremos que los resultados de la regresión son $\alpha = 1$ y $\beta = 0$, de modo que obtenemos la siguiente ecuación final basada en el supuesto 2:

$$E_t = |\Delta T_t| = |T_{t+1} - T_t| = \begin{cases} T_{t+1} - T_t & \text{si } T_{t+1} > T_t, \text{ es decir, si el servidor se calienta} \\ T_t - T_{t+1} & \text{si } T_{t+1} < T_t, \text{ es decir, si el servidor se enfria} \end{cases}$$

Ahora, expliquemos cómo simularemos el funcionamiento del servidor con los usuarios y los datos que entran y salen.

Simulación

El número de usuarios y la velocidad de transmisión de datos fluctuarán aleatoriamente para simular un servidor real. Esto lleva a una aleatoriedad en la temperatura y la IA tiene que entender cuánta potencia de enfriamiento o calefacción tiene que transferir al servidor para no deteriorar el rendimiento del servidor y, al mismo tiempo, gastar la menor energía optimizando su transferencia de calor.

Ahora que tenemos la imagen completa, expliquemos el funcionamiento general del servidor y la IA dentro de este entorno.

Funcionamiento general:

Dentro de un centro de datos, estamos tratando con un servidor específico que está controlado por los parámetros y variables enumerados anteriormente. Cada minuto, algunos usuarios nuevos inician sesión en el servidor y algunos usuarios actuales cierran sesión, por lo tanto, actualizan el número de usuarios activos en el servidor. Igualmente, cada minuto se transmiten algunos datos nuevos al servidor, y algunos datos existentes se transmiten fuera del servidor, por lo tanto, se actualiza la velocidad de transmisión de datos que ocurre dentro del servidor. Por lo tanto, según el supuesto 1 anterior, la temperatura del servidor se actualiza cada minuto. Ahora, concéntrate, porque aquí es donde entenderás el gran papel que la IA tiene que jugar en el servidor. Dos posibles sistemas pueden regular la temperatura del servidor: la IA o el sistema de enfriamiento integrado del servidor. El sistema de enfriamiento integrado del servidor es un sistema no inteligente que automáticamente devolverá la temperatura del servidor a su temperatura óptima. Expliquemos esto con más detalles: cuando la temperatura del servidor se actualiza cada minuto, puede mantenerse dentro del rango de temperaturas óptimas ($[18^\circ\text{C}, 24^\circ\text{C}]$), o salir de este rango. Si sale del rango óptimo, como por ejemplo 30°C , el sistema de enfriamiento integrado del servidor llevará automáticamente la temperatura al límite más cercano del rango óptimo, que es 24°C . Sin embargo, el sistema de enfriamiento integrado de este servidor lo hará solo cuando la IA no esté activada. Si la IA está activada, en ese caso el sistema

de enfriamiento integrado del servidor se desactiva y es la IA la que actualiza la temperatura del servidor para regularlo de la mejor manera. Pero la IA hace eso después de algunas predicciones previas, no de una manera determinista como con el sistema de enfriamiento integrado del servidor no inteligente. Antes de que haya una actualización de la cantidad de usuarios y la velocidad de transmisión de datos que hace que cambie la temperatura del servidor, la IA predice si debería enfriar el servidor, no hacer nada o calentar el servidor. Entonces ocurre el cambio de temperatura y la IA reitera. Y dado que estos dos sistemas son complementarios, los evaluaremos por separado para comparar su rendimiento.

Y eso nos lleva a la energía. De hecho, recordemos que un objetivo principal de la IA es ahorrar algo de energía gastada en este servidor. En consecuencia, nuestra IA tiene que gastar menos energía que la energía gastada por el sistema de enfriamiento no inteligente en el servidor. Y dado que, según el supuesto 2 anterior, la energía gastada en el servidor (por cualquier sistema) es proporcional al cambio de temperatura dentro de una unidad de tiempo:

$$E_t = |\Delta T_t| = \alpha |T_{t+1} - T_t| = \begin{cases} T_{t+1} - T_t & \text{si } T_{t+1} > T_t, \text{ es decir, si el servidor se calienta} \\ T_t - T_{t+1} & \text{if } T_{t+1} < T_t, \text{ es decir, si el servidor se enfria} \end{cases}$$

entonces eso significa que la energía ahorrada por la IA en cada instante t (cada minuto) es, de hecho, la diferencia en los cambios absolutos de temperatura causados en el servidor entre el sistema de enfriamiento integrado del servidor no inteligente y la IA de t y $t + 1$:

$$\begin{aligned} \text{Energía ahorrada por la IA entre } t \text{ y } t + 1 &= |\Delta T_t^{\text{Sistema de Enfriamiento Integrado del Servidor}}| - |\Delta T_t^{\text{IA}}| \\ &= |\Delta T_t^{\text{no IA}}| - |\Delta T_t^{\text{IA}}| \end{aligned}$$

donde:

- $\Delta T_t^{\text{no IA}}$ es el cambio de temperatura que causaría el sistema de enfriamiento integrado del servidor sin la IA en el servidor durante la iteración t , es decir, del instante t al instante $t + 1$,
- ΔT_t^{IA} es el cambio de temperatura causado por la IA en el servidor durante la iteración t , es decir, del instante t al instante $t + 1$.

Nuestro objetivo será ahorrar la energía máxima cada minuto, por lo tanto, ahorrar la energía total máxima durante 1 año completo de simulación y, finalmente, ahorrar los costos máximos en la factura de electricidad de refrigeración / calefacción.

¿Estamos preparados?

¡Excelente! Ahora que entendemos completamente cómo funciona nuestro entorno de servidor y cómo se simula, es hora de proceder con lo que debe hacerse absolutamente al definir un entorno de IA:

- Definir los estados
- Definir las acciones
- Definir las recompensas

Definir los estados

El estado de entrada s_t en el momento t se compone de los siguientes tres elementos:

- La temperatura del servidor en el instante t .
- El número de usuarios en el servidor en el instante t .
- La velocidad de transmisión de datos en el servidor en el instante t .

Por lo tanto, el estado de entrada será un vector de entrada de estos tres elementos. Nuestra futura IA tomará este vector como entrada y devolverá la acción para ejecutar en cada instante t .

Definir las acciones

Las acciones son simplemente los cambios de temperatura que la IA puede causar dentro del servidor, para calentarlo o enfriarlo. Para que nuestras acciones sean discretas, consideraremos 5 posibles cambios de temperatura de -3°C a $+3^{\circ}\text{C}$, para que terminemos con las 5 acciones posibles que la IA puede llevar a cabo para regular la temperatura del servidor:

Acción	¿Qué hace?
0	La IA enfriá el servidor 3°C
1	La IA enfriá el servidor 1.5°C
2	La IA no transfiere calor ni frio al servidor (sin cambio de temperatura)
3	La IA calienta el servidor 1.5°C
4	La IA calienta el servidor 3°C

Definir las recompensas.

Después de leer el párrafo “Funcionamiento general” anterior, puedes adivinar cuál será la recompensa. Por supuesto, la recompensa en la iteración t es la energía gastada en el servidor que la IA está ahorrando con respecto al sistema de enfriamiento integrado del servidor, es decir, la diferencia entre la energía que gastaría el sistema de enfriamiento no inteligente si la IA fuera desactivada y la energía que la IA gasta en el servidor:

$$\text{Reward}_t = E_t^{\text{no IA}} - E_t^{\text{IA}}$$

Y como (Supuesto 2), la energía gastada es igual al cambio de temperatura causado en el servidor (por cualquier sistema, incluido el AI o el sistema de enfriamiento no inteligente):

$$E_t = |\Delta T_t| = \alpha |T_{t+1} - T_t| = \begin{cases} T_{t+1} - T_t & \text{si } T_{t+1} > T_t, \text{ es decir, si el servidor se calienta} \\ T_t - T_{t+1} & \text{si } T_{t+1} < T_t, \text{ es decir, si el servidor se enfria} \end{cases}$$

entonces obtenemos que la recompensa recibida en el instante t es, de hecho, la diferencia en el cambio de temperatura causada en el servidor entre el sistema de enfriamiento no inteligente (es decir, cuando no hay IA) y la IA:

$$\begin{aligned} \text{Reward}_t &= \text{Energía ahorrada por la IA entre } t \text{ y } t + 1 \\ &= E_t^{\text{no IA}} - E_t^{\text{IA}} \\ &= |\Delta T_t^{\text{no IA}}| - |\Delta T_t^{\text{IA}}| \end{aligned}$$

donde:

- $\Delta T_t^{\text{no IA}}$ es el cambio de temperatura que causaría el sistema de enfriamiento integrado del servidor sin la IA en el servidor durante la iteración t , es decir, del instante t al instante $t + 1$,
- ΔT_t^{IA} es el cambio de temperatura causado por la IA en el servidor durante la iteración t , es decir, del instante t al instante $t + 1$.

Nota importante: es importante comprender que los sistemas (nuestra IA y el sistema de enfriamiento del servidor) se evaluarán por separado para calcular las recompensas. Y dado que cada vez que sus acciones conducen a temperaturas diferentes, tendremos que realizar un seguimiento por separado de las dos temperaturas T_t^{IA} and $T_t^{\text{no IA}}$.

Ahora, para terminar esta sección, vamos a hacer una pequeña simulación de 2 iteraciones (es decir, 2 minutos), como un ejemplo que hará que todo quede claro.

Ejemplo de simulación final.

Digamos que estamos en el instante de tiempo $t = 4 : 00 \text{ pm}$ y que la temperatura del servidor es $T_t = 28^\circ \text{ C}$, tanto con la IA como sin la IA. En este momento exacto, la IA predice la acción 0, 1, 2, 3 o 4. Desde ahora, la temperatura del servidor está fuera del rango de temperatura óptimo $[18^\circ \text{C}, 24^\circ \text{C}]$, la IA probablemente predecirá las acciones 0, 1 o 2. Digamos que predice 1, lo que corresponde a enfriar el servidor en 1.5° C . Por lo tanto, entre $t = 4 : 00 \text{ pm}$ y $t + 1 = 4 : 01 \text{ pm}$, la IA hace que la temperatura del servidor pase de $\$T_t^{\text{IA}} = 28^\circ \text{C}$ a $T_{t+1}^{\text{IA}} = 26.5^\circ \text{C}$:

$$\begin{aligned}\Delta T_t^{\text{IA}} &= T_{t+1}^{\text{IA}} - T_t^{\text{IA}} \\ &= 26.5 - 28 \\ &= -1.5^\circ \text{C}\end{aligned}$$

Por lo tanto, según el supuesto 2, la energía gastada por la IA en el servidor es:

$$\begin{aligned}E_t^{\text{IA}} &= |\Delta T_t^{\text{IA}}| \\ &= 1.5 \text{ Joules}\end{aligned}$$

Bien, ahora solo falta una información para calcular la recompensa: es la energía que el sistema de enfriamiento integrado del servidor habría gastado si la IA se hubiera desactivado entre las 4:00 p.m. y las 4:01 p.m. Recordemos que este sistema de enfriamiento no inteligente lleva automáticamente la temperatura del servidor de vuelta al límite más cercano del rango de temperatura óptimo $[18^\circ \text{C}, 24^\circ \text{C}]$. Entonces, dado que a $t = 4 : 00 \text{ pm}$ la temperatura era $T_t = 28^\circ \text{ C}$, entonces el límite más cercano del rango de temperatura óptimo en ese momento era 24° C . Por lo tanto, el sistema de enfriamiento integrado del servidor habría cambiado la temperatura de $T_t = 28^\circ \text{C}$ a $T_{t+1} = 24^\circ \text{C}$, y por lo tanto la temperatura del servidor cambia habría ocurrido si no hubiera IA es:

$$\begin{aligned}\Delta T_t^{\text{no IA}} &= T_{t+1}^{\text{no IA}} - T_t^{\text{no IA}} \\ &= 24 - 28 \\ &= -4^\circ \text{C}\end{aligned}$$

Por lo tanto, según el supuesto 2, la energía que el sistema de enfriamiento no inteligente habría gastado si no hubiera IA es:

$$\begin{aligned}E_t^{\text{no IA}} &= |\Delta T_t^{\text{no IA}}| \\ &= 4 \text{ Joules}\end{aligned}$$

En conclusión, la recompensa que obtenemos después de llevar a cabo esta acción en el momento $t = 4 : 00$ pm es:

$$\begin{aligned}\text{Reward} &= E_t^{\text{no IA}} - E_t^{\text{IA}} \\ &= 4 - 1.5 \\ &= 2.5\end{aligned}$$

Luego, entre $t = 4 : 00$ pm y $t + 1 = 4 : 01$ pm, suceden otras cosas: algunos usuarios nuevos inician sesión en el servidor, algunos usuarios existentes cierran sesión en el servidor, algunos datos nuevos son transmitiendo dentro del servidor, y algunos datos existentes se transmiten fuera del servidor. Según el supuesto 1, estos factores hacen que la temperatura del servidor cambie. Digamos que aumentan la temperatura del servidor en 5°C :

$$\Delta_t \text{ Temperatura Intrínseca} = 5^\circ \text{C}$$

Ahora recuerde que estamos evaluando dos sistemas por separado: nuestra IA y el sistema de enfriamiento integrado del servidor. Por lo tanto, debemos calcular por separado las dos temperaturas que obtendríamos con estos dos sistemas a $t + 1 = 4 : 01$ pm. Comencemos con la IA.

La temperatura que obtenemos en $t + 1 = 4 : 01$ pm cuando se activa la IA es:

$$\begin{aligned}T_{t+1}^{\text{IA}} &= T_t^{\text{IA}} + \Delta T_t^{\text{IA}} + \Delta_t \text{ Temperatura Intrínseca} \\ &= 28 + (-1.5) + 5 \\ &= 31.5^\circ \text{C}\end{aligned}$$

Y la temperatura que obtenemos en $t + 1 = 4 : 01$ pm cuando la IA no está activada es:

$$\begin{aligned}T_{t+1}^{\text{no IA}} &= T_t^{\text{no IA}} + \Delta T_t^{\text{no IA}} + \Delta_t \text{ Temperatura Intrínseca} \\ &= 28 + (-4) + 5 \\ &= 29^\circ \text{C}\end{aligned}$$

Perfecto, tenemos nuestras dos temperaturas separadas, que son $T_{t+1}^{\text{AI}} = 29.5^\circ \text{C}$ cuando la IA está activada, y $T_{t+1}^{\text{no AI}} = 27^\circ \text{C}$ cuando la IA no está activada.

Ahora simulemos lo que sucede entre los instantes $t + 1 = 4 : 01$ pm y $t + 2 = 4 : 02$ pm. Nuevamente, nuestra IA hará una predicción, y dado que el servidor se está calentando, digamos que predice la acción 0, que corresponde a enfriar el servidor en 3°C , reduciéndolo a $T_{t+2}^{\text{IA}} = 28.5^\circ \text{C}$. Por lo tanto, la energía gastada por la IA entre $t + 1 = 4 : 01$ pm y $t + 2 = 4 : 02$ pm, es:

$$\begin{aligned}E_{t+1}^{\text{IA}} &= |\Delta T_{t+1}^{\text{IA}}| \\ &= |28.5 - 31.5| \\ &= 3 \text{ Joules}\end{aligned}$$

Ahora con respecto al sistema de enfriamiento integrado del servidor (es decir, cuando no hay IA), ya que a $t + 1 = 4 : 01$ pm teníamos $T_{t+1}^{\text{no IA}} = 29^\circ\text{C}$, entonces el límite más cercano del rango óptimo de temperaturas sigue siendo 24°C , por lo que la energía que el sistema de enfriamiento no inteligente del servidor gastaría entre $t + 1 = 4 : 01$ pm y $t + 2 = 4 : 02$ pm, es:

$$\begin{aligned} E_{t+1}^{\text{no IA}} &= |\Delta T_{t+1}^{\text{no IA}}| \\ &= |24 - 29| \\ &= 5 \text{ Joules} \end{aligned}$$

De ahí la recompensa obtenida entre $t + 1 = 4 : 01$ pm y $t + 2 = 4 : 02$ pm, es:

$$\begin{aligned} \text{Reward} &= E_{t+1}^{\text{no IA}} - E_{t+1}^{\text{IA}} \\ &= 5 - 3 \\ &= 2 \end{aligned}$$

Y finalmente, la recompensa total obtenida entre $t = 4 : 00$ pm y $t + 2 = 4 : 02$ pm, es:

$$\begin{aligned} \text{Total Reward} &= (\text{Recompensa obtenida entre } t \text{ y } t + 1) + (\text{Recompensa obtenida entre } t + 1 \text{ y } t + 2) \\ &= 2.5 + 2 \\ &= 4.5 \end{aligned}$$

Ese fue un ejemplo de todo el proceso que sucedió en dos minutos. En nuestra implementación, ejecutaremos el mismo proceso durante 1000 épocas de 5 meses para el entrenamiento del algoritmo, y luego, una vez que nuestra IA esté entrenada, ejecutaremos el mismo proceso durante 1 año completo de simulación para la prueba. El entrenamiento se realizará con Deep Q-Learning, y aquí es donde entra en juego la siguiente sección.

2.2 Solución de IA

La solución de IA que resolverá el problema descrito anteriormente es un modelo Deep Q-Learning. Vamos a dar la teoría y las ecuaciones matemáticas detrás de esto.

2.2.1 Q-Learning en Deep Learning

El Deep Q-Learning consiste en combinar Q-Learning con una red neuronal artificial. Las entradas son vectores codificados, cada uno de los cuales define un estado del entorno. Estas entradas van a una red neuronal artificial, donde la salida es la acción a ejecutar. Más precisamente, digamos que el sistema tiene n acciones posibles, la capa de salida de la red neuronal está compuesta por n neuronas de salida, cada una correspondiente a los valores Q de cada acción que se juega en el estado actual. Entonces, la acción que se juega es la asociada con la neurona de salida que tiene el valor Q más alto (*argmax*), o la que devuelve el método *softmax*. En nuestro caso usaremos *argmax*. Y dado que los valores Q son números reales, eso hace que nuestra red neuronal sea un RNA para la regresión.

Así que, para cada estado s_t :

-
- la predicción es el valor Q, $Q(s_t, a_t)$ donde a_t es elegido por argmax o softmax,
 - el valor objetivo es $r_t + \gamma \max_a(Q(s_{t+1}, a))$,
 - el error de pérdida entre la predicción y el objetivo es el cuadrado de la diferencia temporal:

$$\text{Loss} = \frac{1}{2} \left(r_t + \gamma \max_a(Q(s_{t+1}, a)) - Q(s_t, a_t) \right)^2 = \frac{1}{2} TD_t(s_t, a_t)^2.$$

Luego, este error de pérdida se propaga hacia atrás en la red, y los pesos se actualizan de acuerdo con la cantidad que contribuyeron al error.

2.2.2 Experience Replay

Notemos que hasta ahora solo hemos considerado las transiciones de un estado s_t al siguiente estado s_{t+1} . El problema con esto es que s_t está casi siempre muy correlacionado con s_{t+1} . Por lo tanto, la red no está aprendiendo mucho. Esto podría mejorarse mucho si, en lugar de considerar solo esta transición anterior, consideramos las últimas m transiciones donde m es un gran número. Este paquete de las últimas m transiciones es lo que se llama Experience Replay o repetición de experiencia. Luego, a partir de esta repetición de experiencia, tomamos algunos bloques aleatorios de transiciones para realizar nuestras actualizaciones.

2.2.3 El cerebro

El cerebro, o más precisamente la red neuronal profunda de nuestra IA, será una red neuronal completamente conectada, compuesta de dos capas ocultas, la primera con 64 neuronas y la segunda con 32 neuronas. Y como recordatorio, esta red neuronal toma como entradas los estados del entorno y devuelve como salidas los valores Q para cada una de las 5 acciones. Este cerebro artificial se entrenará con una pérdida de “error cuadrático medio” y un optimizador Adam.

Así es como se ve este cerebro artificial:

Este cerebro artificial parece complejo de crear, pero lo construiremos muy fácilmente gracias a la increíble librería de Keras. Aquí hay una vista previa de la implementación completa que contiene la parte que construye este cerebro por sí mismo:

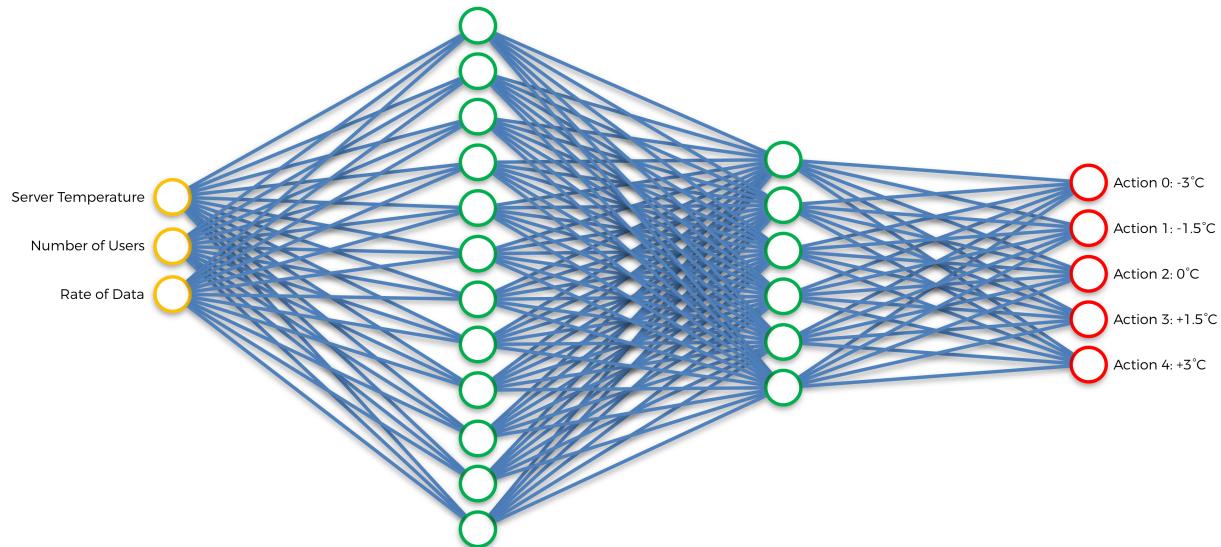
```
# Construcción del cerebro

class Brain(object):

    def __init__(self, learning_rate = 0.001, number_actions = 11):
        self.learning_rate = learning_rate
        states = Input(shape = (3,))
        x = Dense(units = 64, activation = 'sigmoid')(states)
        y = Dense(units = 32, activation = 'sigmoid')(x)
        q_values = Dense(units = number_actions, activation = 'softmax')(y)
        self.model = Model(inputs = states, outputs = q_values)
        self.model.compile(loss = 'mse', optimizer = Adam(lr = self.learning_rate))
```

Como podemos ver con gusto, solo son necesarias un par de líneas de código.

Case Study #2 - Minimizing the Energy Consumption in a Server



Artificial Intelligence for Business

© SuperDataScience

Figure 2.1: El cerebro artificial: una red neuronal completamente conectada

2.2.4 El algoritmo de Deep Q-Learning al completo

Resumamos los diferentes pasos de todo el proceso de Deep Q-Learning:

Inicialización

La memoria de Experience Replay se inicializa en una lista vacía M .

Elegimos un tamaño máximo de la memoria. En nuestro caso práctico elegimos un tamaño máximo de 100 transiciones.

Comenzamos en un primer estado, correspondiente a un momento específico dentro del año.

En cada instante t , repetimos el siguiente proceso, hasta el final de la época (5 meses en nuestra implementación)

1. Predecimos los valores Q del estado actual s_t .
2. Ejecutamos la acción que corresponde al máximo de estos valores Q predichos (método argmax):

$$a_t = \underset{a}{\operatorname{argmax}} Q(s_t, a)$$

3. Obtenemos la recompensa:

$$r_t = E_t^{\text{no IA}} - E_t^{\text{IA}}$$

4. Alcanzamos el siguiente estado s_{t+1} .
5. Añadimos la transición actual (s_t, a_t, r_t, s_{t+1}) a M .
6. Seleccionamos un bloque de transiciones al azar $B \subset M$. Para todas las transiciones $(s_{t_B}, a_{t_B}, r_{t_B}, s_{t_B+1})$ del bloque aleatorio B :

-
- Obtenemos las predicciones:

$$Q(s_{t_B}, a_{t_B})$$

- Obtenemos los objetivos:

$$r_{t_B} + \gamma \max_a(Q(s_{t_B+1}, a))$$

- Calculamos la pérdida entre las predicciones y los objetivos en todo el bloque B :

$$\text{Loss} = \frac{1}{2} \sum_B \left(r_{t_B} + \gamma \max_a(Q(s_{t_B+1}, a)) - Q(s_{t_B}, a_{t_B}) \right)^2 = \frac{1}{2} \sum_B TD_{t_B}(s_{t_B}, a_{t_B})^2$$

- Volvemos a propagar este error de pérdida en la red neuronal y, a través del descenso de gradiente estocástico, actualizamos los pesos según cuánto contribuyeron al error..

2.3 Implementation

Esta implementación se dividirá en 5 partes, cada parte con su propio archivo de Python. Estas 5 partes constituyen el algoritmo general de IA, o Blueprint de la AI, que debe seguirse cada vez que construimos un entorno para resolver cualquier problema comercial con Deep Reinforcement Learning.

Aquí están, del Paso 1 al Paso 5:

1. Construcción del entorno.
2. Construcción del cerebro.
3. Implementación del algoritmo de aprendizaje por refuerzo profundo (en nuestro caso será el modelo DQN).
4. Entrenar a la IA.
5. Probar de la IA.

Estos son los pasos principales (en ese mismo orden) de la sección de teoría general de IA anterior. Implementemos así nuestra IA para nuestro caso práctico específico, siguiendo este plan de IA, en las siguientes cinco secciones correspondientes a estos cinco pasos principales. Además en cada paso, distinguiremos los subpasos que todavía forman parte del algoritmo general de AI, de los subpasos que son específicos de nuestro caso práctico, escribiendo los títulos de las secciones de código en mayúsculas para todos los subpasos del algoritmo general de AI, y en letras mínimas para todos los subpasos específicos de nuestro caso práctico. Eso significa que cada vez que veamos una nueva sección de código cuyo título está escrito en letras mayúsculas, entonces es el siguiente subpaso del algoritmo general de IA, que también se debe seguir al crear una IA para cualquier otro problema comercial.

Así que ahora aquí vamos con el comienzo del viaje: Paso 1 - Construcción el entorno.

Este es el archivo de implementación de python más grande de este caso práctico, y del curso. Por lo tanto, asegúrate de descansar antes, recargar las baterías para obtener un buen nivel de energía y, tan pronto como estés listo, ¡abordemos esto juntos!

2.3.1 Paso 1: Construcción del Entorno

En este primer paso, vamos a construir el entorno dentro de una clase. ¿Por qué una clase? Porque nos gustaría tener nuestro entorno como un objeto que podamos crear fácilmente con cualquier valor de algunos parámetros que elijamos. Por ejemplo, podemos crear un objeto de entorno para un servidor que tenga un cierto número de usuarios conectados y una cierta velocidad de datos en un momento específico, y otro objeto de entorno para otro servidor que tenga un número diferente de usuarios conectados y un número

diferente tasa de datos en otro momento. Y gracias a esta estructura avanzada de la clase, podemos conectar y reproducir fácilmente los objetos del entorno que creamos en diferentes servidores que tienen sus propios parámetros, por lo tanto, regulamos sus temperaturas con varias IA diferentes, de modo que terminamos minimizando el consumo de energía. de un centro de datos completo, tal como lo hizo la DeepMind de Google para los centros de datos de Google con su algoritmo DQN.

Esta clase sigue los siguientes subpasos, que son parte del algoritmo general de IA dentro del Paso 1: construcción del entorno:

- **Paso 1-1:** Introducción e inicialización de todos los parámetros y variables del entorno.
- **Paso 1-2:** Hacer un método que actualice el entorno justo después de que la IA ejecute una acción.
- **Paso 1-3:** Hacer un método que restablezca el entorno.
- **Paso 1-4:** hacer un método que nos proporcione en cualquier momento el estado actual, la última recompensa obtenida y si el juego ha terminado.

Encontrarás toda la implementación de esta clase de creación de entorno en las próximas páginas. Recuerda lo más importante: todas las secciones de código que tienen sus títulos escritos en letras mayúsculas son los pasos del framework de IA o del Blueprint general, y todas las secciones de código que tienen sus títulos escritos en letras minúsculas son específicas de nuestro caso práctico.

A continuación se muestra la implementación completa de nuestro primer archivo de python. Los títulos de las secciones de código y los nombres de las variables elegidas son lo suficientemente claros como para comprender lo que se está codificando, pero si necesitas más explicaciones, te recomiendo que vea nuestros videos tutoriales en Udemy donde codificamos todo desde cero, paso a paso, mientras explicamos cada línea de código en términos de por qué, qué y cómo. Aquí vamos:

```
# Inteligencia Artificial aplicada a Negocios y Empresas - Caso Práctico 2
# Construcción del Entorno

# Importar las librerías
import numpy as np

# CONSTRUCCIÓN DEL ENTORNO EN UNA CLASE

class Environment(object):

    # INTRODUCCIÓN E INICIALIZACIÓN DE TODOS LOS PARÁMETROS Y VARIABLES DEL ENTORNO

    def __init__(self,
                  optimal_temperature = (18.0, 24.0),
                  initial_month = 0,
                  initial_number_users = 10,
                  initial_rate_data = 60):
        self.monthly_atmospheric_temperatures = [1.0, 5.0, 7.0, 10.0, 11.0, 20.0,
                                                23.0, 24.0, 22.0, 10.0, 5.0, 1.0]
        self.initial_month = initial_month
        self.atmospheric_temperature = \
            self.monthly_atmospheric_temperatures[initial_month]
        self.optimal_temperature = optimal_temperature
        self.min_temperature = -20
        self.max_temperature = 80
        self.min_number_users = 10
```

```

self.max_number_users = 100
self.max_update_users = 5
self.min_rate_data = 20
self.max_rate_data = 300
self.max_update_data = 10
self.initial_number_users = initial_number_users
self.current_number_users = initial_number_users
self.initial_rate_data = initial_rate_data
self.current_rate_data = initial_rate_data
self.intrinsic_temperature = self.atmospheric_temperature
                           + 1.25 * self.current_number_users
                           + 1.25 * self.current_rate_data
self.temperature_ai = self.intrinsic_temperature
self.temperature_noai = (self.optimal_temperature[0]
                         + self.optimal_temperature[1]) / 2.0
self.total_energy_ai = 0.0
self.total_energy_noai = 0.0
self.reward = 0.0
self.game_over = 0
self.train = 1

```

CREACIÓN DE UN MÉTODO QUE ACTUALIZA EL ENTORNO DESPUÉS DE QUE LA IA EJECUTE UNA ACCIÓN

```

def update_env(self, direction, energy_ai, month):

    # OBTENCIÓN DE LA RECOMPENSA

    # Calcular la energía gastada por el sistema de refrigeración del servidor cuando
    energy_noai = 0
    if (self.temperature_noai < self.optimal_temperature[0]):
        energy_noai = self.optimal_temperature[0] - self.temperature_noai
        self.temperature_noai = self.optimal_temperature[0]
    elif (self.temperature_noai > self.optimal_temperature[1]):
        energy_noai = self.temperature_noai - self.optimal_temperature[1]
        self.temperature_noai = self.optimal_temperature[1]
    # Cálculo de la recompensa
    self.reward = energy_noai - energy_ai
    # Escalado de la recompensa
    self.reward = 1e-3 * self.reward

    # OBTENCIÓN DEL SIGUIENTE ESTADO

    # Actualización de la temperatura atmosférica
    self.atmospheric_temperature = self.monthly_atmospheric_temperatures[month]
    # Actualización del número de usuarios conectados
    self.current_number_users += np.random.randint(-self.max_update_users,
                                                   self.max_update_users)
    if (self.current_number_users > self.max_number_users):
        self.current_number_users = self.max_number_users

```

```

    elif (self.current_number_users < self.min_number_users):
        self.current_number_users = self.min_number_users
    # Actualización del ratio de datos
    self.current_rate_data += np.random.randint(-self.max_update_data,
                                                self.max_update_data)
    if (self.current_rate_data > self.max_rate_data):
        self.current_rate_data = self.max_rate_data
    elif (self.current_rate_data < self.min_rate_data):
        self.current_rate_data = self.min_rate_data
    # Cálculo de la variación Temperatura Intrínseca
    past_intrinsic_temperature = self.intrinsic_temperature
    self.intrinsic_temperature = self.atmospheric_temperature
                + 1.25 * self.current_number_users
                + 1.25 * self.current_rate_data
    delta_intrinsic_temperature = self.intrinsic_temperature
                                - past_intrinsic_temperature
    # Cálculo de la variación de temperatura causada por la IA
    if (direction == -1):
        delta_temperature_ai = -energy_ai
    elif (direction == 1):
        delta_temperature_ai = energy_ai
    # Actualización de la temperatura del servidor cuando hay IA
    self.temperature_ai += delta_intrinsic_temperature + delta_temperature_ai
    # Actualización de la temperatura del servidor cuando no hay IA
    self.temperature_noai += delta_intrinsic_temperature

    # OBTENCIÓN DEL FIN DE LA PARTIDA

    if (self.temperature_ai < self.min_temperature):
        if (self.train == 1):
            self.game_over = 1
        else:
            self.total_energy_ai += self.optimal_temperature[0]
                            - self.temperature_ai
            self.temperature_ai = self.optimal_temperature[0]
    elif (self.temperature_ai > self.max_temperature):
        if (self.train == 1):
            self.game_over = 1
        else:
            self.total_energy_ai += self.temperature_ai
                            - self.optimal_temperature[1]
            self.temperature_ai = self.optimal_temperature[1]

    # ACTUALIZACIÓN DE LOS SCORES

    # Actualización del total de energía gastada cuando hay IA
    self.total_energy_ai += energy_ai
    # Actualización del total de energía gastada cuando no hay IA
    self.total_energy_noai += energy_noai

```

```

# ESCALADO DEL SIGUIENTE ESTADO

scaled_temperature_ai = (self.temperature_ai - self.min_temperature)
                      / (self.max_temperature - self.min_temperature)
scaled_number_users = (self.current_number_users - self.min_number_users)
                      / (self.max_number_users - self.min_number_users)
scaled_rate_data = (self.current_rate_data - self.min_rate_data)
                      / (self.max_rate_data - self.min_rate_data)
next_state = np.matrix([scaled_temperature_ai,
                       scaled_number_users,
                       scaled_rate_data])

# DEVOLVER EL SIGUIENTE ESTADO, LA RECOMPENSA Y EL ESTADO DE FIN DEL JUEGO

return next_state, self.reward, self.game_over

```

```

# CREACIÓN DE UN MÉTODO QUE REINICIA EL ENTORNO

def reset(self, new_month):
    self.atmospheric_temperature = self.monthly_atmospheric_temperatures[new_month]
    self.initial_month = new_month
    self.current_number_users = self.initial_number_users
    self.current_rate_data = self.initial_rate_data
    self.intrinsic_temperature = self.atmospheric_temperature
                           + 1.25 * self.current_number_users
                           + 1.25 * self.current_rate_data
    self.temperature_ai = self.intrinsic_temperature
    self.temperature_noai = (self.optimal_temperature[0]
                            + self.optimal_temperature[1]) / 2.0
    self.total_energy_ai = 0.0
    self.total_energy_noai = 0.0
    self.reward = 0.0
    self.game_over = 0
    self.train = 1

```

```

# CREACIÓN DE UN MÉTODO QUE NOS DA, A PARTIR DE CUALQUIER INSTANTE, EL ESTADO, LA RECOMPENSA Y EL ESTADO DE FIN DEL JUEGO

def observe(self):
    scaled_temperature_ai = (self.temperature_ai - self.min_temperature)
                           / (self.max_temperature - self.min_temperature)
    scaled_number_users = (self.current_number_users - self.min_number_users)
                           / (self.max_number_users - self.min_number_users)
    scaled_rate_data = (self.current_rate_data - self.min_rate_data)
                           / (self.max_rate_data - self.min_rate_data)
    current_state = np.matrix([scaled_temperature_ai,
                               scaled_number_users,
                               scaled_rate_data])
return current_state, self.reward, self.game_over

```

Felicidades por implementar el Paso 1: Construcción del entorno. Ahora pasemos al Paso 2: Construcción del cerebro.

2.3.2 Paso 2: Construcción del cerebro

En este Paso 2, vamos a construir el cerebro artificial de nuestra IA, que no es más que una red neuronal completamente conectada. Aquí está de nuevo:

Case Study #2 - Minimizing the Energy Consumption in a Server

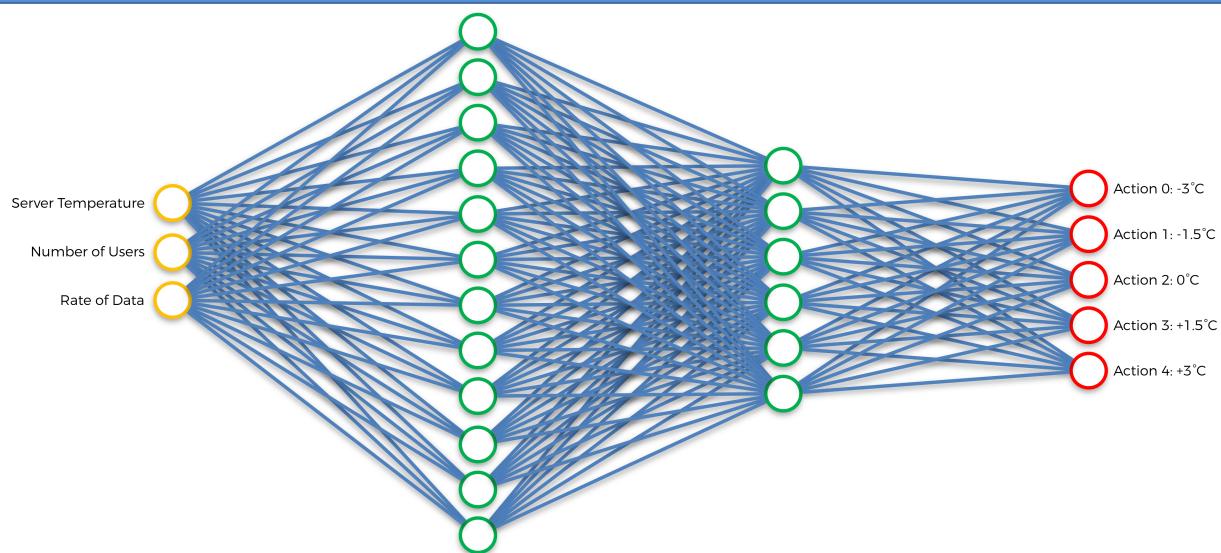


Figure 2.2: El cerebro artificial: una red neuronal completamente conectada

Nuevamente, construiremos este cerebro artificial dentro de una clase, por la misma razón que antes, que nos permite crear varios cerebros artificiales para diferentes servidores dentro de un centro de datos. De hecho, tal vez algunos servidores necesitarán cerebros artificiales diferentes con hiperparámetros diferentes que otros servidores. Es por eso que gracias a esta estructura avanzada de python de clase / objeto, podemos cambiar fácilmente de un cerebro a otro para regular la temperatura de un nuevo servidor que requiere una IA con diferentes parámetros de redes neuronales.

Construiremos este cerebro artificial gracias a la increíble biblioteca Keras. Desde esta librería utilizaremos la clase `Dense()` para crear nuestras dos capas ocultas completamente conectadas, la primera con 64 neuronas ocultas y la segunda con 32 neuronas. Y luego, utilizaremos la clase `Dense()` nuevamente para devolver los valores Q, que tienen en cuenta las salidas de las redes neuronales artificiales. Luego, más adelante en el entrenamiento y los archivos de prueba, utilizaremos el método `argmax` para seleccionar la acción que tenga el valor Q máximo. Luego, ensamblamos todos los componentes del cerebro, incluidas las entradas y las salidas, creándolo como un objeto de la clase `Model()` (muy útil para luego guardar y cargar un modelo en producción con pesos específicos). Finalmente, lo compilaremos con una función de pérdidas que medirá el error cuadrático medio y el optimizador de Adam. Así, aquí están los nuevos pasos del algoritmo general de IA:

- **Paso 2-1:** Construir la capa de entrada compuesta de los estados de entrada.

- **Paso 2-2:** Construir las capas ocultas con un número elegido de estas capas y neuronas dentro de cada una, completamente conectadas a la capa de entrada y entre ellas.
- **Paso 2-3:** Construir la capa de salida, completamente conectada a la última capa oculta.
- **Paso 2-4:** Ensamblar la arquitectura completa dentro de un modelo de Keras.
- **Paso 2-5:** Compilación del modelo con una función de pérdida de error cuadrático medio y el optimizador elegido.

Aquí vamos con la implementación:

```
# Inteligencia Artificial aplicada a Negocios y Empresas - Caso Práctico 2
# Construcción del cerebro

# Importar las librerías
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import Adam

# CONSTRUCCIÓN DEL CEREBRO

class Brain(object):

    # CONSTRUCCIÓN DE UNA RED NEURONAL TOTALMENTE CONECTADA EN EL MÉTODO DE INICIALIZACIÓN

    def __init__(self, learning_rate = 0.001, number_actions = 5):
        self.learning_rate = learning_rate

        # CONSTRUCCIÓN DE LA CAPA DE ENTRADA COMPUESTA DE LOS ESTADOS DE ETRADA
        states = Input(shape = (3,))

        # CONSTRUCCIÓN DE LAS DOS CAPAS OCULTAS TOTALMENTE CONECTADAS
        x = Dense(units = 64, activation = 'sigmoid')(states)
        y = Dense(units = 32, activation = 'sigmoid')(x)

        # CONSTRUCCIÓN DE LA CAPA DE SALIDA, TOTALMENTE CONECTADA A LA ÚLTIMA CAPA OCULTA
        q_values = Dense(units = number_actions, activation = 'softmax')(y)

        # ENSAMBLAR LA ARQUITECTURA COMPLETA EN UN MODELO DE KERAS
        self.model = Model(inputs = states, outputs = q_values)

        # COMPILAR EL MODELO CON LA FUNCIÓN DE PÉRDIDAS DE ERROR CUADRÁTICO MEDIO Y EL OPTIMIZADOR
        self.model.compile(loss = 'mse', optimizer = Adam(lr = learning_rate))
```

Dropout.

Hemos pensado que sería valioso para nosotros incluso agregar una técnica más poderosa en nuestro kit de herramientas de IA: el **Dropout**.

El **dropout** es una técnica de regularización que evita el sobreajuste. Simplemente consiste en desactivar una cierta proporción de neuronas aleatorias durante cada paso de propagación hacia adelante y hacia atrás. De esa manera, no todas las neuronas aprenden de la misma manera, evitando así que la red neuronal sobreajuste los datos de entrenamiento.

Así es como implementamos el Dropout:

1. Primero importamos la función Dropout:

```
from keras.layers:from keras.layers import Input, Dense, Dropout
```

2. Luego, activamos el Dropout en la primera capa oculta x , con una proporción de 0.1, lo que significa que el 10% de las neuronas se desactivarán aleatoriamente durante el entrenamiento a cada iteración:

```
x = Dense(units = 64, activation = 'sigmoid')(states)
x = Dropout(rate = 0.1)(x)
```

3. Y finalmente, activamos de nuevo e Dropout en la segunda capa oculta y , con una proporción de 0.1, lo que significa que en 10% de las neuronas se desactivarán aleatoriamente durante el entrenamiento a cada iteración:

```
y = Dense(units = 32, activation = 'sigmoid')(x)
y = Dropout(rate = 0.1)(y)
```

¡Felicitaciones! Has implementado el Dropout. Es realmente muy simple, una vez más gracias a Keras.

Debajo está la implementación mejorada del fichero new_brain.py con Dropout incluido:

```
# Inteligencia Artificial aplicada a Negocios y Empresas - Caso Práctico 2
# Construcción del cerebro

# Importar las librerías
from keras.layers import Input, Dense, Dropout
from keras.models import Model
from keras.optimizers import Adam

# CONSTRUCCIÓN DEL CEREBRO

class Brain(object):

    # CONSTRUCCIÓN DE UNA RED NEURONAL TOTALMENTE CONECTADA EN EL MÉTODO DE INICIALIZACIÓN

    def __init__(self, learning_rate = 0.001, number_actions = 5):
        self.learning_rate = learning_rate

        # CONSTRUCCIÓN DE LA CAPA DE ENTRADA COMPUESTA DE LOS ESTADOS DE ENTRADA
        states = Input(shape = (3,))

        # CONSTRUCCIÓN DE PRIMERA CAPA OCULTAS TOTALMENTE CONECTADA CON DROPOUT ACTIVADO
        x = Dense(units = 64, activation = 'sigmoid')(states)
        x = Dropout(rate = 0.1)(x)

        # CONSTRUCCIÓN DE SEGUNDA CAPA OCULTAS TOTALMENTE CONECTADA CON DROPOUT ACTIVADO
        y = Dense(units = 32, activation = 'sigmoid')(x)
        y = Dropout(rate = 0.1)(y)

        # CONSTRUCCIÓN DE LA CAPA DE SALIDA, TOTALMENTE CONECTADA A LA ÚLTIMA CAPA OCULTA
        q_values = Dense(units = number_actions, activation = 'softmax')(y)
```

```

# ENSAMBLAR LA ARQUITECTURA COMPLETA EN UN MODELO DE KERAS
self.model = Model(inputs = states, outputs = q_values)

# COMPIILAR EL MODELO CON LA FUNCIÓN DE PÉRDIDAS DE ERROR CUADRÁTICO MEDIO Y EL OPTIMIZADOR
self.model.compile(loss = 'mse', optimizer = Adam(lr = learning_rate))

```

Ahora pasemos al siguiente paso de nuestro algoritmo general de IA: Paso 3: Implementación del algoritmo DQN.

2.3.3 Paso 3: Implementación del algoritmo de Deep Reinforcement Learning

En este nuevo archivo de python, simplemente tenemos que seguir el algoritmo Deep Q-Learning que hemos visto anteriormente. Por lo tanto, esta implementación sigue los siguientes subpasos, que forman parte del algoritmo general de IA:

- **Paso 3-1:** Introducción e inicialización de todos los parámetros y variables del modelo de DQN.
- **Paso 3-2:** Hacer un método que construya la memoria en Repetición de Experiencia.
- **Paso 3-3:** Hacer un método que construya y devuelva dos lotes de 10 entradas y 10 objetivos

A continuación se muestra el código que sigue a esta nueva parte del Blueprint de IA:

```

# Inteligencia Artificial aplicada a Negocios y Empresas - Caso Práctico 2
# Implementar el algoritmo de Deep Q-Learning con Repetición de Experiencia

# Importar las librerías
import numpy as np

# IMPLEMENTAR EL DEEP Q-LEARNING CON REPETICIÓN DE EXPERIENCIA

class DQN(object):

    # INTRODUCIR E INICIALIZAR TODOS LOS PARÁMETROS Y VARIABLES DEL DQN
    def __init__(self, max_memory = 100, discount = 0.9):
        self.memory = list()
        self.max_memory = max_memory
        self.discount = discount

    # CREACIÓN DE UN MÉTODO QUE CONSTRUYA LA MEMORIA DE LA REPETICIÓN DE EXPERIENCIA
    def remember(self, transition, game_over):
        self.memory.append([transition, game_over])
        if len(self.memory) > self.max_memory:
            del self.memory[0]

    # CREACIÓN DEL MÉTODO QUE COSTRUYE DOS LOTES DE ENTRADAS Y OBJETIVOS
    def get_batch(self, model, batch_size = 10):
        len_memory = len(self.memory)
        num_inputs = self.memory[0][0][0].shape[1]
        num_outputs = model.output_shape[-1]
        inputs = np.zeros((min(len_memory, batch_size), num_inputs))

        for i in range(batch_size):
            state = self.memory[i][0][0]
            reward = self.memory[i][0][1]
            done = self.memory[i][0][2]
            next_state = self.memory[i][1][0]
            target = reward
            if not done:
                target = reward + self.discount * np.max(next_state)
            inputs[i] = state
            targets[i] = target

```

```

targets = np.zeros((min(len_memory, batch_size), num_outputs))
for i, idx in enumerate(np.random.randint(0, len_memory,
                                         size = min(len_memory, batch_size))):
    current_state, action, reward, next_state = self.memory[idx][0]
    game_over = self.memory[idx][1]
    inputs[i] = current_state
    targets[i] = model.predict(current_state)[0]
    Q_sa = np.max(model.predict(next_state)[0])
    if game_over:
        targets[i, action] = reward
    else:
        targets[i, action] = reward + self.discount * Q_sa
return inputs, targets

```

2.3.4 Paso 4: Entrenar la IA

Ahora que nuestra IA tiene un cerebro completamente funcional, es hora de entrenarlo. Y esto es exactamente lo que hacemos en este cuarto archivo de python. El proceso es largo, pero muy fácil: comenzamos estableciendo todos los parámetros, luego construimos el entorno creando un objeto de la clase `Environment()`, luego construimos el cerebro de la IA creando un objeto de la clase `Brain()`, luego construimos el modelo de Deep Q-Learning creando un objeto de la clase `DQN()`, y finalmente lanzamos la fase de entrenamiento que conecta todos estos objetos, durante 1000 epochs de 5 meses cada uno. Notarás en la fase de entrenamiento de entrenamiento que también exploramos un poco cuando llevamos a cabo las acciones las acciones. Esto consiste en ejecutar algunas acciones aleatorias de vez en cuando. En nuestro Caso Práctico, esto se realizará el 30% de las veces, ya que usamos un parámetro de exploración $\epsilon = 0.3$, y luego lo forzamos a ejecutar una acción aleatoria al obtener un valor aleatorio entre 0 y 1 que está por debajo de $\epsilon = 0.3$. La razón por la que hacemos un poco de exploración es porque mejora el proceso de aprendizaje por refuerzo profundo. Este truco se llama: *Exploración vs. Explotación*. Luego, además, también veremos que utilizamos una técnica de detención temprana, que se asegurará de detener el entrenamiento si ya no hay una mejora palpable en el rendimiento.

Destacaremos estos nuevos pasos que aún pertenecen a nuestro algoritmo general de IA:

- **Paso 4-1:** Construcción del entorno creando un objeto de la clase `Environment`.
- **Paso 4-2:** Construyendo el cerebro artificial creando un objeto de la clase de `Brain`.
- **Paso 4-3:** Construyendo el modelo DQN creando un objeto de la clase `DQN`.
- **Paso 4-4:** Elección del modo de entrenamiento.
- **Paso 4-5:** Comenzar el entrenamiento con un bucle `for` durante más de 100 epochs de períodos de 5 meses.
- **Paso 4-6:** Durante cada epoch, repetimos todo el proceso de Deep Q-Learning, al tiempo que exploramos el 30% de las veces.

Y ahora implementemos esta nueva parte, Paso 4: Entrenamiento de la IA, de nuestro algoritmo general. A continuación se muestra la implementación completa de este cuarto archivo de python. Una vez más, los títulos de las secciones de código y los nombres de las variables elegidas son lo suficientemente claros como para comprender lo que se programa en cada caso Aquí vamos:

```
# Inteligencia Artificial aplicada a Negocios y Empresas - Caso Práctico 2
# Entrenamiento de la IA
```

```

# Instalación de Keras
# conda install -c conda-forge keras

# Importar las librerías y el resto de ficheros de python
import os
import numpy as np
import random as rn
import environment
import brain
import dqn

# Establecer semillas para la reproducibilidad del experimento
os.environ['PYTHONHASHSEED'] = '0'
np.random.seed(42)
rn.seed(12345)

# CONFIGURACIÓN DE LOS PARÁMETROS
epsilon = .3
number_actions = 5
direction_boundary = (number_actions - 1) / 2
number_epochs = 100
max_memory = 3000
batch_size = 512
temperature_step = 1.5

# CONSTRUCCIÓN DEL ENTORNO CREANDO UN OBJETO DE LA CLASE ENVIRONMENT CLASS
env = environment.Environment(optimal_temperature = (18.0, 24.0),
                             initial_month = 0,
                             initial_number_users = 20,
                             initial_rate_data = 30)

# CONSTRUCCIÓN DEL CEREBRO CREADO UN OBJETO DE LA CLASE BRAIN
brain = brain.Brain(learning_rate = 0.00001, number_actions = number_actions)

# CONSTRUCCIÓN DEL MODELO DE DQN CREANDO UN OBJETO DE LA CLASE DQN
dqn = dqn.DQN(max_memory = max_memory, discount = 0.9)

# ELECCIÓN DEL MODO DE ENTRENAMIENTO
train = True

# Entrenamiento de la IA
env.train = train
model = brain.model
early_stopping = True
patience = 10
best_total_reward = -np.inf
patience_count = 0
if (env.train):
    # ARRANCAR EL BUCLE SOBRE TODAS LAS EPOCHS (1 Epoch = 5 Meses)

```

```

for epoch in range(1, number_epochs):
    # INICIALIZACIÓN DE LAS VARIABLES TANTO DEL ENVIRONMENT COMO DEL BUCLE DE ENTRENAMIENTO
    total_reward = 0
    loss = 0.
    new_month = np.random.randint(0, 12)
    env.reset(new_month = new_month)
    game_over = False
    current_state, _, _ = env.observe()
    timestep = 0
    # EMPEZAR EL BUCLE SOBRE TODOS LOS TIMESTEPS (1 Timestep = 1 Minuto) EN UN EPOCH
    while ((not game_over) and timestep <= 5 * 30 * 24 * 60):
        # EJECUTAR LA SIGUIENTE ACCIÓN POR EXPLORACIÓN
        if np.random.rand() <= epsilon:
            action = np.random.randint(0, number_actions)
            if (action - direction_boundary < 0):
                direction = -1
            else:
                direction = 1
            energy_ai = abs(action - direction_boundary) * temperature_step
        # EJECUTAR LA SIGUIENTE ACCIÓN POR INFERENCIA
        else:
            q_values = model.predict(current_state)
            action = np.argmax(q_values[0])
            if (action - direction_boundary < 0):
                direction = -1
            else:
                direction = 1
            energy_ai = abs(action - direction_boundary) * temperature_step
        # ACTUALIZACIÓN DEL ENTORNO BUSCANDO EL SIGUIENTE ESTADO
        next_state, reward, game_over = env.update_env(direction,
                                                       energy_ai,
                                                       int(timestep / (30*24*60)))
        total_reward += reward
        # ALMACENAR LA NUEVA TRANSICIÓN EN LA MEMORIA
        dqn.remember([current_state, action, reward, next_state], game_over)
        # REUNIR EN DOS LOTES SEPARADOS LAS ENTRADAS Y LOS OBJETIVOS
        inputs, targets = dqn.get_batch(model, batch_size = batch_size)
        # CALCULAR LA PÉRDIDA EN LOS DOS LOTES DE ENTRADAS Y OBJETIVOS
        loss += model.train_on_batch(inputs, targets)
        timestep += 1
        current_state = next_state
    # IMPRIMIR EL RESULTADO DE ENTRENAMIENTO PARA CADA EPOCH
    print("\n")
    print("Epoch: {:03d}/{:03d}".format(epoch, number_epochs))
    print("Total Energy spent with an AI: {:.0f}".format(env.total_energy_ai))
    print("Total Energy spent with no AI: {:.0f}".format(env.total_energy_noai))
    # EARLY STOPPING
    if (early_stopping):

```

```

if (total_reward <= best_total_reward):
    patience_count += 1
elif (total_reward > best_total_reward):
    best_total_reward = total_reward
    patience_count = 0
if (patience_count >= patience):
    print("Early Stopping")
    break
# GUARDAR EL MODELO
model.save("model.h5")

```

Después de ejecutar el código, ya vemos un buen rendimiento de nuestra IA durante el entrenamiento, gastando la mayor parte del tiempo menos energía que el sistema alternativo, es decir, el sistema de enfriamiento integrado del servidor. Pero ese es solo el entrenamiento, ahora necesitamos ver si también obtenemos un buen rendimiento en una nueva simulación de 1 año. Ahí es donde entra en juego nuestro próximo y último archivo de python.

2.3.5 Paso 5: Probar nuestra IA

Ahora, de hecho, tenemos que probar el rendimiento de nuestra IA en una situación completamente nueva. Para hacerlo, ejecutaremos una simulación de 1 año, solo en modo de inferencia, lo que significa que no habrá entrenamiento en ningún momento. Nuestra IA solo devolverá predicciones durante un año completo de simulación. Luego, gracias a nuestro objeto Environment, obtendremos al final la energía total gastada por la IA durante este año completo, así como la energía total gastada por el sistema de enfriamiento integrado del servidor. Eventualmente compararemos estas dos energías totales gastadas, simplemente calculando su diferencia relativa (en %), lo que nos dará exactamente la energía total ahorrada por la IA. ¡Abróchate el cinturón para los ver los resultados finales, que revelaremos al final de esta Parte 2!

En términos de nuestro algoritmo de IA, aquí para la implementación de prueba casi tenemos lo mismo que antes, excepto que esta vez, no tenemos que crear un objeto Brain ni un objeto modelo DQN, y por supuesto no debemos ejecutar el proceso de Deep Q-Learning durante las épocas de entrenamiento. Sin embargo, tenemos que crear un nuevo objeto de Environment, y en lugar de crear un cerebro, cargaremos nuestro cerebro artificial con sus pesos pre-entrenados del entrenamiento anterior que ejecutamos en el Paso 4 - Entrenamiento de la IA. Por lo tanto, demos los subpasos finales de esta parte final del algoritmo de IA:

- **Paso 5-1:** Construcción de un nuevo entorno creando un objeto de la clase Environment.
- **Paso 5-2:** Carga del cerebro artificial con sus pesos pre-entrenados del entrenamiento anterior.
- **Paso 5-3:** Elección del modo de inferencia.
- **Paso 5-4:** Iniciación de la simulación de 1 año.
- **Paso 5-5:** En cada iteración (cada minuto), nuestra IA solo ejecuta la acción que resulta de su predicción, y no se lleva a cabo ninguna exploración o entrenamiento de Deep Q-Learning.

Y ahora implementemos esta quinta y última parte, Paso 5: Prueba de la IA. Una vez más, a continuación se muestra la implementación completa de nuestro último archivo de Python. Los títulos de las secciones de código y los nombres de las variables elegidas son lo suficientemente claros como para comprender lo que se está programando, pero si necesitas más explicaciones, te recomiendo que veas nuestros videos tutoriales en Udemy donde programamos todo desde cero, paso a paso, mientras explicamos cada línea de código en términos de por qué, qué y cómo. Aquí vamos:

```

# Inteligencia Artificial aplicada a Negocios y Empresas - Caso Práctico 2
# Prueba de la AI

# Instalación de Keras
# conda install -c conda-forge keras

# Importar las librerías y el resto de ficheros de python
import os
import numpy as np
import random as rn
from keras.models import load_model
import environment

# Establecer semillas para la reproducibilidad del experimento
os.environ['PYTHONHASHSEED'] = '0'
np.random.seed(42)
rn.seed(12345)

# CONFIGURACIÓN DE LOS PARÁMETROS
number_actions = 5
direction_boundary = (number_actions - 1) / 2
temperature_step = 1.5

# CONSTRUCCIÓN DEL ENTORNO CREANDO UN OBJETO DE LA CLASE ENVIRONMENT
env = environment.Environment(optimal_temperature = (18.0, 24.0),
                               initial_month = 0,
                               initial_number_users = 20,
                               initial_rate_data = 30)

# CARGA DEL MODELO PRE-ENTRENADO
model = load_model("model.h5")

# ELECCIÓN DEL MODO
train = False

# EJECUTAR UN AÑO DE SIMULACIÓN EN MODO INFERENCIA
env.train = train
current_state, _, _ = env.observe()
for timestep in range(0, 12 * 30 * 24 * 60):
    q_values = model.predict(current_state)
    action = np.argmax(q_values[0])
    if (action - direction_boundary < 0):
        direction = -1
    else:
        direction = 1
    energy_ai = abs(action - direction_boundary) * temperature_step
    next_state, reward, game_over = env.update_env(direction,
                                                   energy_ai,
                                                   int(timestep / (30*24*60)))

```

```

current_state = next_state

# IMPRIMIR LOS RESULTADOS DE ENTREAMIENTO PARA CADA EPOCH
print("\n")
print("Total Energy spent with an AI: {:.0f}".format(env.total_energy_ai))
print("Total Energy spent with no AI: {:.0f}".format(env.total_energy_noai))
print("ENERGY SAVED: {:.0f} %".format((env.total_energy_noai - env.total_energy_ai)
                                      / env.total_energy_noai * 100))

```

Y finalmente, obtenemos en los resultados impresos que el consumo total de energía ahorrado por la IA es ...:

Total Energy saved by the AI = 39 % !

¡Exactamente igual a lo que la DeepMind de Google logró en 2016! De hecho, si en Google escribes: *DeepMind reduce la factura de enfriamiento de Google*, verá que el resultado que lograron es del 40 %. Muy cerca de la nuestra!

Por lo tanto, lo que hemos construido es seguramente excelente para nuestro cliente comercial, ya que nuestra IA les ahorrará muchos costes. De hecho, recuerda que gracias a nuestra estructura orientada a objetos (trabajando con clases y objetos), podemos tomar fácilmente nuestros objetos creados en esta implementación que hicimos para un servidor, y luego conectarlos a otros servidores, para que al final podamos ¡terminar ahorrando en el consumo total de energía de un centro de datos al completo! Así es como Google ahorró miles de millones de dólares en costes relacionados con la energía, gracias a su modelo DQN creado por la IA DeepMind.

2.4 Resumen: El Algoritmo General de IA

Recapitulemos y proporcionemos el algoritmo completo de IA, para que puedas imprimirla y ponerla en tu pared.

Paso 1: Construcción del Entorno

- **Paso 1-1:** Introducción e inicialización de todos los parámetros y variables del entorno.
- **Paso 1-2:** Hacer un método que actualice el entorno justo después de que la IA ejecute una acción.
- **Paso 1-3:** Hacer un método que restablezca el entorno.
- **Paso 1-4:** hacer un método que nos proporcione en cualquier momento el estado actual, la última recompensa obtenida y si el juego ha terminado.

**Paso 2: Construcción del Cerebro

- **Paso 2-1:** Construir la capa de entrada compuesta de los estados de entrada.
- **Paso 2-2:** Construir las capas ocultas con un número elegido de estas capas y neuronas dentro de cada una, completamente conectadas a la capa de entrada y entre ellas.
- **Paso 2-3:** Construir la capa de salida, completamente conectada a la última capa oculta.
- **Paso 2-4:** Ensamblar la arquitectura completa dentro de un modelo de Keras.
- **Paso 2-5:** Compilación del modelo con una función de pérdida de error cuadrático medio y el optimizador elegido.

Paso 3: Implementación del algoritmo de Deep Reinforcement Learning

- **Paso 3-1:** Introducción e inicialización de todos los parámetros y variables del modelo de DQN.

-
- **Paso 3-2:** Hacer un método que construya la memoria en Repetición de Experiencia.
 - **Paso 3-3:** Hacer un método que construya y devuelva dos lotes de 10 entradas y 10 objetivos

Paso 4: Entrenamiento de la IA

- **Paso 4-1:** Construcción del entorno creando un objeto de la clase Environment.
- **Paso 4-2:** Construyendo el cerebro artificial creando un objeto de la clase de Brain
- **Paso 4-3:** Construyendo el modelo DQN creando un objeto de la clase DQN.
- **Paso 4-4:** Elección del modo de entrenamiento.
- **Paso 4-5:** Comenzar el entrenamiento con un bucle for durante más de 100 epochs de períodos de 5 meses.

Paso 5: Probar la IA

- **Paso 5-1:** Construcción de un nuevo entorno creando un objeto de la clase Environment.
- **Paso 5-2:** Carga del cerebro artificial con sus pesos pre-entrenados del entrenamiento anterior.
- **Paso 5-3:** Elección del modo de inferencia.
- **Paso 5-4:** Iniciación de la simulación de 1 año.
- **Paso 5-5:** En cada iteración (cada minuto), nuestra IA solo ejecuta la acción que resulta de su predicción, y no se lleva a cabo ninguna exploración o entrenamiento de Deep Q-Learning.

Chapter 3

Maximización de Beneficios Revenues

¡Felicitaciones por seguir el primer y segundo estudio de caso! Ahora pasemos a un tipo muy diferente de Inteligencia Artificial, que tiene una eficiencia tremenda para las empresas y negocios y que sin duda alguna debes conocer.

3.1 Caso Práctico: Maximización de beneficios de un negocio de venta online en línea

3.1.1 Problema a reesolver

Imagina un negocio minorista en línea que tiene millones de clientes. Estos clientes son solo personas que compran algunos productos en el sitio web de vez en cuando y se los entregan en casa (como Amazon). El negocio está funcionando bien, pero la junta directiva ha decidido tomar algún plan de acción para maximizar aún más los ingresos. Este plan consiste en ofrecer a los clientes la opción de suscribirse a un plan premium, que les dará algunos beneficios como precios reducidos, ofertas especiales, etc. Este plan premium se ofrece a un precio anual de 100 dólares y el objetivo de este negocio minorista en línea es, por supuesto, conseguir que el máximo de clientes se suscriba a este plan premium. Hagamos algunos cálculos rápidos para motivarnos a construir una IA para maximizar los ingresos de este negocio. Digamos que este negocio minorista en línea tiene 100 millones de clientes. Ahora consideremos dos estrategias de conversión que intentan convertir a los clientes al plan premium: una mala, con una tasa de conversión del 1%, y una buena, con una tasa de conversión del 11%. Si el negocio lleva a cabo la estrategia mala, obtendrá en un año un ingreso extra total proveniente de la suscripción al plan premium de: $100.000.000 \times 0.01 \times 100 = 100.000.000\$\text{}$. Por otro lado, si el negocio implementa la buena estrategia, obtendrá en un año un ingreso extra total proveniente de la suscripción al plan premium de $100.000.000 \times 0.11 \times 100 = 1.100.000.000\$\text{}$. Por lo tanto, al descubrir la buena estrategia para implementar, el negocio maximizará sus ingresos adicionales al ganar más de mil millones de dólares adicionales.

En este ejemplo utópico anterior, solo teníamos dos estrategias, y además sabíamos sus tasas de conversión. Sin embargo, en nuestro caso práctico enfrentaremos 9 estrategias diferentes, y nuestra IA no tendrá idea de cuál es la mejor, y absolutamente ninguna información previa sobre ninguna de sus tasas de conversión. Sin embargo, asumiremos que cada una de estas 9 estrategias tiene una tasa de conversión fija. Estas estrategias fueron elaboradas de forma cuidadosa e inteligente por el equipo de marketing, y cada una de ellas tiene el mismo objetivo: convertir a los clientes máximos en el plan premium. Sin embargo, estas 9 estrategias

son todas diferentes. Tienen diferentes formas, diferentes paquetes, diferentes anuncios y diferentes ofertas especiales para convencer y persuadir a los clientes a suscribirse al plan premium. Por supuesto, el equipo de marketing no tiene idea de cuál de estas 9 estrategias es la mejor. Pero quieren resolverlo lo antes posible y ahorrando los costes máximos, cuál tiene la tasa de conversión más alta, porque saben cómo encontrar e implementar esa mejor estrategia puede maximizar significativamente los ingresos. Además, los expertos en marketing optan por no enviar un correo electrónico a sus 100 millones de clientes, ya que sería costoso y correrían el riesgo de enviar spam a demasiados clientes. En su lugar, buscarán sutilmente esa mejor estrategia a través del aprendizaje en línea. ¿Qué es el aprendizaje en línea? Consistirá en implementar una estrategia cada vez que un cliente navegue por el sitio web de negocios minoristas en línea para pasar el rato o comprar algunos productos. Luego, mientras el cliente navega por el sitio web, de repente recibirá un anuncio emergente, sugiriéndole que se suscriba al plan premium. Y para cada cliente que navega por el sitio web, solo se implementará una de las 9 estrategias. Luego, el usuario elegirá, o no, tomar medidas y suscribirse al plan premium. Si el cliente se suscribe, es un éxito, de lo contrario, es un fracaso. Cuantos más clientes hagan esto, más comentarios recibiremos y mejor podremos tener una idea de cuál es la mejor estrategia. Pero, por supuesto, no lo resolveremos manualmente, visualmente o con algunas matemáticas simples. En cambio, queremos implementar el algoritmo más inteligente que descubra cuál es la mejor estrategia en el menor tiempo posible. Y eso es por las mismas dos razones: primero porque implementar cada estrategia tiene un coste (por ejemplo, proveniente del anuncio emergente en la web), y segundo porque la compañía quiere molestar a los clientes lo menos posible con su anuncio.

Resumamos las diferencias en las características de estas 9 estrategias simplemente de esta manera:

Case Study #3 - Maximizing the Revenues of an Online Retail Business

Strategy 1 <ul style="list-style-type: none">· Form 1· Package 1· Ad 1· Special Deal 1	Strategy 2 <ul style="list-style-type: none">· Form 2· Package 2· Ad 2· Special Deal 2	Strategy 3 <ul style="list-style-type: none">· Form 3· Package 3· Ad 3· Special Deal 3
Strategy 4 <ul style="list-style-type: none">· Form 4· Package 4· Ad 4· Special Deal 4	Strategy 5 <ul style="list-style-type: none">· Form 5· Package 5· Ad 5· Special Deal 5	Strategy 6 <ul style="list-style-type: none">· Form 6· Package 6· Ad 6· Special Deal 6
Strategy 7 <ul style="list-style-type: none">· Form 7· Package 7· Ad 7· Special Deal 7	Strategy 8 <ul style="list-style-type: none">· Form 8· Package 8· Ad 8· Special Deal 8	Strategy 9 <ul style="list-style-type: none">· Form 9· Package 9· Ad 9· Special Deal 9

Simulación

Para simular este Caso Práctico, asumiremos que estas estrategias tienen las siguientes tasas de conversión:

Case Study #3 - Maximizing the Revenues of an Online Retail Business

Simulation

Strategy	Conversion Rate
1	0.05
2	0.13
3	0.09
4	0.16
5	0.11
6	0.04
7	0.20
8	0.08
9	0.01

Sin embargo, asegúrate de comprender que en una situación de la vida real **no tendríamos idea** de cuáles serían estas tasas de conversión. Solo las conocemos aquí para fines de simulación, solo para que podamos verificar al final que nuestra IA logra descubrir la mejor estrategia, que según la tabla anterior, es la estrategia número 7 (la que tiene la tasa de conversión más alta).

3.1.2 Definición del Entorno

El aprendizaje en línea es una rama especial de la inteligencia artificial, donde no hay mucha necesidad de definir los estados y las acciones. Aquí, un estado sería simplemente un cliente específico en el que desplegaríamos una estrategia, y la acción sería simplemente la estrategia seleccionada. Luego lo verás más claro en el algoritmo de IA, donde no tenemos los estados como entradas y las acciones como salidas como en nuestros dos casos prácticos anteriores, porque esta vez no estamos haciendo Q-Learning o Deep Q-Learning. Aquí estamos haciendo aprendizaje en línea. Sin embargo, tenemos que definir las recompensas, ya que nuevamente tendremos que hacer una matriz de recompensas, donde cada fila corresponde a un usuario que está implementando una estrategia, y cada columna corresponde a una de las 9 estrategias. Por lo tanto, dado que realmente ejecutaremos este experimento de aprendizaje en línea en 10.000 clientes, esta matriz de recompensas tendrá 10.000 filas y 9 columnas. Luego, cada celda obtendrá un 0 si el cliente no se suscribe al plan premium después de ser abordado por la estrategia seleccionada, y un 1 si el cliente se suscribe después de ser abordado por la estrategia seleccionada. Y los valores en la celda son exactamente, las recompensas.

Ahora, una cosa muy importante para entender es que la matriz de recompensas solo está aquí para la simulación, y en la vida real no tendríamos una matriz de recompensas. Simplemente simularemos 10.000 clientes siendo abordados sucesivamente por una de las 9 estrategias, y gracias a la matriz de recompensas simularemos la decisión del cliente de suscribirse sí o no al plan premium. Si la celda correspondiente a un cliente específico y una estrategia seleccionada específica tiene un 1, eso simulará una conversión por parte del cliente al plan premium, y si la celda tiene un 0, simulará un rechazo. A continuación, como ejemplo, las primeras filas de una matriz de recompensas simulada:

	0	1	2	3	4	5	6	7	8
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	1	0	1	0	0
2	0	0	0	0	0	0	0	0	0
3	0	0	0	1	1	0	0	0	0
4	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	0	0	0	0
6	0	0	1	0	0	0	1	0	0
7	0	0	0	0	0	0	0	0	0
8	0	0	0	0	0	0	0	0	0
9	0	0	0	0	0	0	0	0	0
10	0	0	0	0	0	0	0	0	0

De acuerdo con esta simulación, todo dado en la matriz de recompensas anterior:

1. El primer cliente (fila con índice 0) no se suscribirá al plan premium después de haber sido abordado por cualquier estrategia.
2. El segundo cliente (fila con índice 1) se suscribiría al plan premium después de ser abordado únicamente por la estrategia 5 o la estrategia 7.
3. El tercer cliente (fila con índice 2) no se suscribiría al plan premium después de haber sido abordado por cualquier estrategia.

El muestreo de Thompson recopilará los comentarios de si cada uno de estos clientes se suscribe al plan premium uno tras otro y, gracias a su poderoso algoritmo, descubrirá rápidamente la estrategia con la tasa de conversión más alta, esa es la mejor. para ser implementado en los millones de clientes, maximizando así los ingresos de la compañía de esta nueva fuente de ingresos.

3.2 Solución de IA

La solución de IA que determinará la mejor estrategia se llama *muestreo de Thompson*. Es, con diferencia, el mejor modelo para ese tipo de problemas en esta rama de Aprendizaje en línea de Inteligencia Artificial. En resumen, cada vez que un nuevo cliente se conecta al sitio web de negocios minoristas en línea, esa es una nueva ronda n y seleccionamos una de nuestras 9 estrategias para intentar una conversión (suscripción al plan premium). El objetivo es seleccionar la mejor estrategia en cada ronda, y entrenar durante muchas rondas. Así es como el muestreo de Thompson lo hará:

Para cada ronda n , repetimos durante más de 1000 iteraciones, los siguientes tres pasos:

Paso 1 Para cada iteración i , se elige un valor aleatorio que siga la distribución:

$$\theta_i(n) \sim \beta(N_i^1(n) + 1, N_i^0(n) + 1)$$

donde:

- $N_i^1(n)$ es el número de veces que la estrategia i -ésima ha recibido una recompensa igual a 1 hasta la ronda n -ésima,
- $N_i^0(n)$ es el número de veces que la estrategia i -ésima ha recibido una recompensa igual a 0 hasta la ronda n -ésima.

Paso 2 Seleccionamos la estrategia $s(n)$ que nos da el mayor valor $\theta_i(n)$:

$$s(n) = \operatorname{argmax}_{i \in \{1, \dots, 9\}} (\theta_i(n))$$

Paso 3 Actualizamos $N_{s(n)}^1(n)$ y $N_{s(n)}^0(n)$ según las siguientes condiciones:

- Si la estrategia seleccionada $s(n)$ tiene una recompensa igual a 1:

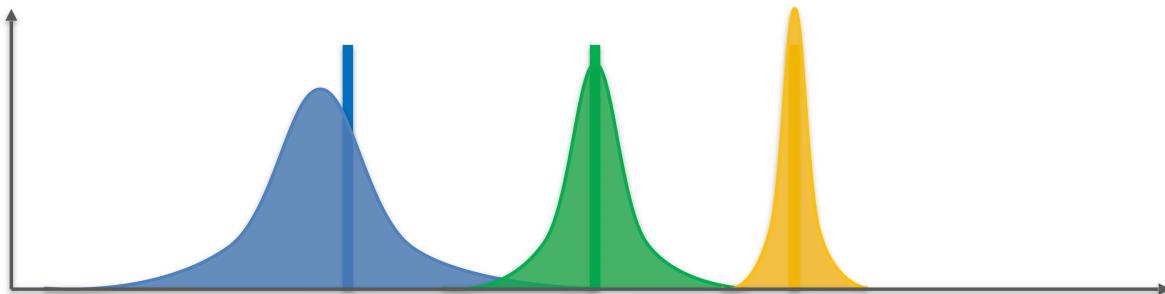
$$N_{s(n)}^1(n) := N_{s(n)}^1(n) + 1$$

- Si la estrategia seleccionada $s(n)$ tiene una recompensa igual a 0:

$$N_{s(n)}^0(n) := N_{s(n)}^0(n) + 1$$

Intuición. Cada estrategia tiene su propia distribución beta. A lo largo de las rondas, la distribución beta de la estrategia con la tasa de conversión más alta se desplazará progresivamente hacia la derecha, y las distribuciones beta de las estrategias con tasas de conversión más bajas se desplazarán progresivamente hacia la izquierda (Pasos 1 y 3). Por lo tanto, debido al Paso 2, la estrategia con la tasa de conversión más alta se seleccionará por probabilidad cada vez más. A continuación se muestra un gráfico que muestra tres distribuciones beta de tres estrategias, que te ayudarán a visualizar este hecho:

Case Study #3 - Maximizing the Revenues of an Online Retail Business



3.3 Implementación

Vamos a ver a continuación la implementación completa del muestreo de Thompson para este caso práctico específico, siguiendo la misma simulación vista anteriormente.

Al implementar el muestreo de Thompson, también implementaremos el algoritmo de selección aleatoria, que simplemente seleccionará una estrategia aleatoria en cada ronda. Este será nuestro punto de referencia para evaluar el rendimiento de nuestro modelo de muestreo de Thompson. Por supuesto, el muestreo de Thompson y el algoritmo de selección aleatoria competirán en la misma simulación, es decir, utilizando la misma matriz de recompensas. Y al final, una vez realizada la simulación completa, evaluaremos el rendimiento de Thompson Sampling calculando el rendimiento relativo, definido por la siguiente fórmula:

$$\text{Rendimiento Rel.} = \frac{(\text{Rec. del m. de Thompson}) - (\text{Rec. de la sel. Aleatoria})}{\text{Recompensa de la s. Aleatoria}} \times 100$$

También representaremos el histograma de los anuncios seleccionados, solo para verificar que la estrategia con la tasa de conversión más alta (Estrategia 7) ha sido en efecto la más seleccionada.

Pues si estás listo, aquí vamos:

Primero, importamos las librerías necesarias y establecemos los parámetros ($N = 10000$ clientes y $d = 9$ estrategias):

```
# Inteligencia Artificial aplicada a Negocios y Empresas
# Maximizando los ingresos de un negocio minorista en línea con el muestreo de Thompson

# Importar las librerías
import numpy as np
import matplotlib.pyplot as plt
import random

# Configuración de parámetros
N = 10000
d = 9
```

Luego, creamos la simulación, construyendo la matriz de recompensas de 10000 filas correspondientes a los clientes, y 9 columnas correspondientes a las estrategias. En cada ronda y para cada estrategia, seleccionamos un número aleatorio entre 0 y 1, y si este número aleatorio es menor que la tasa de conversión de dicha estrategia, la recompensa será 1; de lo contrario, será 0. De esa manera simulamos las tasas de conversión enumeradas anteriormente para nuestras 9 estrategias:

```
# Creación de la simulación
conversion_rates = [0.05, 0.13, 0.09, 0.16, 0.11, 0.04, 0.20, 0.08, 0.01]
X = np.array(np.zeros([N,d]))
for i in range(N):
    for j in range(d):
        if np.random.rand() <= conversion_rates[j]:
            X[i,j] = 1
```

Luego, recorreremos las 10000 filas (o rondas) de esta matriz de recompensas, y en cada ronda obtendremos dos selecciones de estrategia separadas: una del algoritmo de Selección aleatoria y otra del muestreo de Thompson. Llevamos un registro de las estrategias seleccionadas por cada uno de estos dos algoritmos, y calculamos la recompensa total acumulada durante las rondas por cada uno de ellos. El muestreo de Thompson se implementa siguiendo exactamente los pasos 1, 2 y 3 proporcionados anteriormente:

```
# Implementación de la selección aleatoria y del muestreo de Thompson
strategies_selected_rs = []
strategies_selected_ts = []
```

```

total_reward_rs = 0
total_reward_ts = 0
numbers_of_rewards_1 = [0] * d
numbers_of_rewards_0 = [0] * d
for n in range(0, N):
    # Selección aleatoria
    strategy_rs = random.randrange(d)
    strategies_selected_rs.append(strategy_rs)
    reward_rs = X[n, strategy_rs]
    total_reward_rs = total_reward_rs + reward_rs
    # Muestreo de Thompson
    strategy_ts = 0
    max_random = 0
    for i in range(0, d):
        random_beta = random.betavariate(numbers_of_rewards_1[i] + 1,
                                         numbers_of_rewards_0[i] + 1)
        if random_beta > max_random:
            max_random = random_beta
            strategy_ts = i
    reward_ts = X[n, strategy_ts]
    if reward_ts == 1:
        numbers_of_rewards_1[strategy_ts] = numbers_of_rewards_1[strategy_ts] + 1
    else:
        numbers_of_rewards_0[strategy_ts] = numbers_of_rewards_0[strategy_ts] + 1
    strategies_selected_ts.append(strategy_ts)
    total_reward_ts = total_reward_ts + reward_ts

```

Luego calculamos resultado final, que es el rendimiento relativo del muestreo de Thompson con respecto a nuestro punto de referencia que es la selección aleatoria:

```

# Cálculo del rendimiento relativo
relative_return = (total_reward_ts - total_reward_rs) / total_reward_rs * 100
print("Rendimiento Relativo: {:.0f} %".format(relative_return))

```

Y prepárate, porque al ejecutar este código obtenemos un retorno relativo final, de ...:

```
## Rendimiento Relativo: 78 %
```

En otras palabras, el muestreo de Thompson casi duplicó el rendimiento de nuestro punto de referencia de la selección aleatoria.

Y finalmente, representemos el histograma de las estrategias seleccionadas, para verificar que efectivamente la Estrategia 7 (la de índice 6) fue la más seleccionada por el algoritmo, ya que es la que tiene la tasa de conversión más alta:;

```
# Representación del histograma de selecciones
```

```
plt.hist(strategies_selected_ts)
```

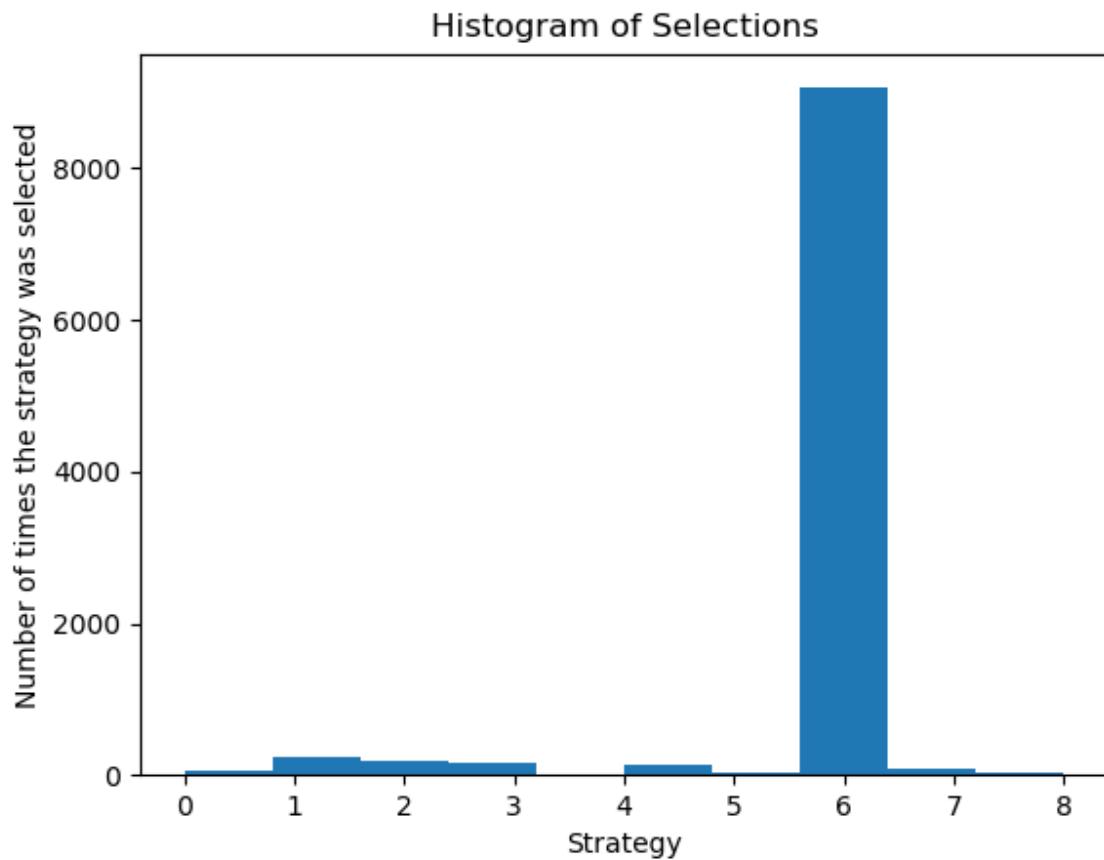
```

## (array([ 46.,   441.,    46.,   836.,     0.,    86.,    36.,   8328.,
##        147.,    34.]), array([ 0. ,  0.8,  1.6,  2.4,  3.2,  4. ,  4.8,  5.6,  6.4,
plt.title('Histograma de Selecciones')
plt.xlabel('Estrategia')
```

```
plt.ylabel('Número de veces que se ha seleccionado la estrategia')
plt.show()
```

```
/Users/juangabriel/Developer/Projects/Matematicas/bookdown-iabusiness/curso-ia-business-ud
```

Al ejecutar este código final, obtenemos el siguiente histograma:



Y de hecho, es la estrategia del índice 6, es decir, la Estrategia 7, la que fue, con diferencia, la más seleccionada. El muestreo de Thompson ha sido capaz de identificarlo rápidamente. Y, de hecho, si volvemos a ejecutar el mismo código pero con solo 1000 clientes, nos damos cuenta de que el muestreo de Thompson todavía puede identificar la Estrategia 7 como la mejor, con muchas menos pruebas.

En consecuencia, el muestreo de Thompson seguramente ha hecho un trabajo increíble para este negocio minorista en línea. Porque no solo ha sido capaz de identificar la mejor estrategia rápidamente en un número reducido de rondas, es decir, con solamente algunos clientes, lo que nos ha ahorrado mucho en publicidad y costes operativos. Pero también, por supuesto, ha sido capaz de descubrir claramente la estrategia con la tasa de conversión más alta. Y, de hecho, si este negocio minorista en línea tiene 100 millones de clientes, y si el plan premium tiene un precio de 100 dólares al año, la implementación de esta mejor estrategia que tiene una tasa de conversión del 20% conduciría a generar un ingreso adicional de ...:

$$\text{Ingresos extra generados} = 100000000 \times 0.2 \times 100 = 2 \text{ mil millones de \$!!}$$

En otras palabras, muestreo de Thompson maximizó clara y rápidamente los ingresos de este negocio minorista en línea, al mismo tiempo que ahorró mucho en los costos, maximizando así la rentabilidad del negocio.

Curva del arrepentimiento.

La curva de arrepentimiento de un modelo (con estrategia aleatoria o con el muestreo de Sampling) es la representación gráfica de la diferencia entre la mejor estrategia y el modelo desplegado, con respecto a las rondas.

La mejor estrategia se calcula simplemente obteniendo, en cada ronda, el máximo de las recompensas acumuladas sobre todas las diferentes estrategias. Por lo tanto, en nuestra implementación, obtendremos la mejor estrategia de la siguiente manera:

```
rewards_strategies = [0] * d
for n in range(0, N):
    # La mejor estrategia
    for i in range(0, d):
        rewards_strategies[i] = rewards_strategies[i] + X[n, i]
    total_reward_bs = max(rewards_strategies)
```

Entonces, el arrepentimiento del muestreo de Thompson se calcula simplemente como la diferencia entre la mejor estrategia y el modelo del muestreo de Thompson:

```
# Arrepentimiento del muestreo Thompson
strategies_selected_ts = []
total_reward_ts = 0
total_reward_bs = 0
numbers_of_rewards_1 = [0] * d
numbers_of_rewards_0 = [0] * d
rewards_strategies = [0] * d
regret = []
for n in range(0, N):
    # Muestreo de Thompson
    strategy_ts = 0
    max_random = 0
    for i in range(0, d):
        random_beta = random.betavariate(numbers_of_rewards_1[i] + 1,
                                         numbers_of_rewards_0[i] + 1)
        if random_beta > max_random:
            max_random = random_beta
            strategy_ts = i
    reward_ts = X[n, strategy_ts]
    if reward_ts == 1:
        numbers_of_rewards_1[strategy_ts] = numbers_of_rewards_1[strategy_ts] + 1
    else:
        numbers_of_rewards_0[strategy_ts] = numbers_of_rewards_0[strategy_ts] + 1
    strategies_selected_ts.append(strategy_ts)
    total_reward_ts = total_reward_ts + reward_ts
    # La mejor estrategia
```

```

for i in range(0, d):
    rewards_strategies[i] = rewards_strategies[i] + X[n, i]
total_reward_bs = max(rewards_strategies)
# Arrepentimiento
regret.append(total_reward_bs - total_reward_ts)

```

Y lo mismo, el arrepentimiento de la estrategia aleatoria simplemente se calcula como la diferencia entre la mejor estrategia y el algoritmo de selección aleatoria:

```

# Arrepentimiento de la estrategia aleatoria
strategies_selected_rs = []
total_reward_rs = 0
total_reward_bs = 0
numbers_of_rewards_1 = [0] * d
numbers_of_rewards_0 = [0] * d
rewards_strategies = [0] * d
regret = []
for n in range(0, N):
    # Estrategia aleatoria
    strategy_rs = random.randrange(d)
    strategies_selected_rs.append(strategy_rs)
    reward_rs = X[n, strategy_rs]
    total_reward_rs = total_reward_rs + reward_rs
    # La mejor estrategia
    for i in range(0, d):
        rewards_strategies[i] = rewards_strategies[i] + X[n, i]
    total_reward_bs = max(rewards_strategies)
    # Arrepentimiento
    regret.append(total_reward_bs - total_reward_rs)

```

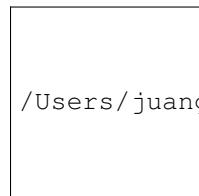
Y finalmente, por supuesto, representamos el arrepentimiento sobre las rondas con este simple código (no tenemos que especificar las coordenadas x en la función plt.plot() porque las rondas ya son índices desde 0 hasta N):

```

# Representación de la curva de arrepentimiento
plt.plot(regret)
plt.title('Curva de Arrepentimiento')
plt.xlabel('Ronda')
plt.ylabel('Arrepentimiento')
plt.show()

```

Si representamos la curva de arrepentimiento de la estrategia aleatoria, obtenemos lo siguiente:



/Users/juangabriel/Developer/Projects/Matematicas/bookdown-iabusiness/curso-ia-business-ud

Y, por supuesto, no observamos absolutamente ninguna convergencia de la estrategia aleatoria hacia la mejor estrategia.

Sin embargo, si ahora representamos la curva de arrepentimiento del modelo de muestreo de Thompson, obtenemos la siguiente curva hermosa:

```
/Users/juangabriel/Developer/Projects/Matematicas/bookdown-iabusiness/curso-ia-business-ud
```

Y obviamente, el muestreo de Thompson está convergiendo muy bien hacia la mejor estrategia.

Finalmente, aquí está el código final que incluye esa Curva de arrepentimiento del muestreo de Thompson:

```
# Muestreo de Thompson

# Importar las librerías
import numpy as np
import matplotlib.pyplot as plt
import random

# Configuración de los parámetros
N = 10000
d = 9

# Creación de la simulación
conversion_rates = [0.05, 0.13, 0.09, 0.16, 0.11, 0.04, 0.20, 0.08, 0.01]
X = np.array(np.zeros([N, d]))
for i in range(N):
    for j in range(d):
        if np.random.rand() <= conversion_rates[j]:
            X[i, j] = 1

# Implementación de la estrategia aleatoria y del muestreo de Thompson con la curva de arre
strategies_selected_rs = []
strategies_selected_ts = []
total_reward_rs = 0
total_reward_ts = 0
total_reward_bs = 0
numbers_of_rewards_1 = [0] * d
numbers_of_rewards_0 = [0] * d
rewards_strategies = [0] * d
regret = []
for n in range(0, N):
    # Estrategia aleatoria
    strategy_rs = random.randrange(d)
    strategies_selected_rs.append(strategy_rs)
    reward_rs = X[n, strategy_rs]
    total_reward_rs = total_reward_rs + reward_rs
    # Muestreo de Thompson
    strategy_ts = 0
    max_random = 0
```

```

for i in range(0, d):
    random_beta = random.betavariate(numbers_of_rewards_1[i] + 1,
                                      numbers_of_rewards_0[i] + 1)
    if random_beta > max_random:
        max_random = random_beta
        strategy_ts = i
    reward_ts = X[n, strategy_ts]
    if reward_ts == 1:
        numbers_of_rewards_1[strategy_ts] = numbers_of_rewards_1[strategy_ts] + 1
    else:
        numbers_of_rewards_0[strategy_ts] = numbers_of_rewards_0[strategy_ts] + 1
    strategies_selected_ts.append(strategy_ts)
    total_reward_ts = total_reward_ts + reward_ts
# La mejor estrategia
for i in range(0, d):
    rewards_strategies[i] = rewards_strategies[i] + X[n, i]
total_reward_bs = max(rewards_strategies)
# Arrepentimiento
regret.append(total_reward_bs - total_reward_ts)

# Calcular el rendimiento absoluto y relativo
absolute_return = total_reward_ts - total_reward_rs
relative_return = (total_reward_ts - total_reward_rs) / total_reward_rs * 100
print("Rendimiento Absoluto: {:.0f} ${}".format(absolute_return))
print("Rendimiento Relativo: {:.0f} %".format(relative_return))

# Representación de los histogramas de selecciones
plt.hist(strategies_selected_ts)
plt.title('Histograma de Selecciones')
plt.xlabel('Estrategia')
plt.ylabel('Número de veces que la estrategia ha sido seleccionada')
plt.show()
plt.close()

# Representación de la curva de arrepentimiento
plt.plot(regret)
plt.title('Curva de Arrepentimiento')
plt.xlabel('Ronda')
plt.ylabel('Arrepentimiento')
plt.show()

```

Conclusión

¡Muchas gracias de nuevo por unirte a este curso en Udemy y felicidades por completarlo! Te recomendamos encarecidamente que mantengas este libro cerca de ti cada vez que construyas una IA para resolver un problema comercial. O al menos, trata de mantener el algoritmo general de IA que hemos construido en cada parte del curso. Para nosotros, ha sido todo un placer hacer este curso y escribir este libro. No dudes en dejar un comentario en el curso si lo deseas. Hasta entonces, ¡disfruta de IA aplicada a Negocios y Empresas!

Chapter 4

Anexos adicionales

4.1 Anexo 1: Redes Neuronales Artificiales

En este apartado Anexo, encontrarás toda la teoría necesaria de Redes Neuronales Artificiales, que son el apartado principal del modelo de Deep Q-Learning model que hemos creado en la Parte 2 - Minimización de Costes. Este es el plan de ataque que seguiremos para estudiar acerca de las Redes Neuronales Artificiales:

- La Neurona
- La Función de Activación
- ¿Cómo funcionan las Redes Neuronales?
- ¿Cómo aprenden las Redes Neuronales?
- Propagación hacia adelante and Propagación hacia atrás
- Gradiente Descendente
- Gradiente Descendente por bloques y Gradiente Descendente Estocástico

4.1.1 La Neurona

La neurona es el bloque de construcción básico de las redes neuronales artificiales. En las siguientes imágenes se muestran neuronas reales de la vida real que se han colocado sobre un vidrio, coloreadas un poco con tinte y observadas con un microscopio:

The Neuron

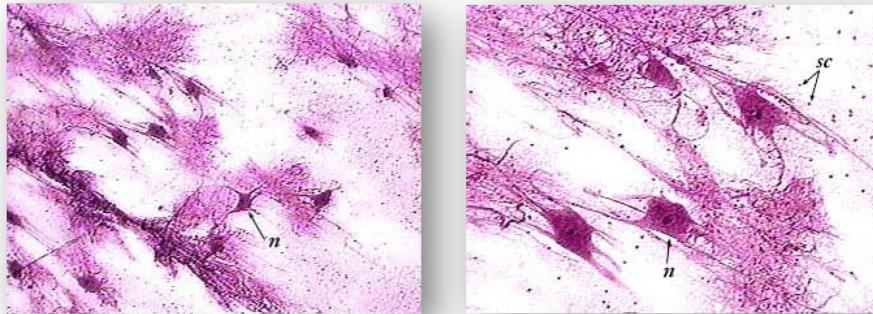


Image Source: www.austincc.edu

Artificial Intelligence for Business

© SuperDataScience

Como podemos ver, tienen la estructura de un cuerpo con muchas ramas diferentes que salen de ellos. Pero la pregunta es: ¿cómo podemos recrear esta estructura en una máquina? De hecho, realmente necesitamos recrearlo en un ordenador, ya que el objetivo de Deep Learning es imitar cómo funciona el cerebro humano, con la esperanza de que al hacerlo creemos algo increíble: una infraestructura poderosa para que las máquinas puedan aprender.

¿Por qué esperamos eso? Porque el cerebro humano resulta ser una de las herramientas de aprendizaje más poderosas del planeta. Así que solo esperamos que si lo recreamos, tendremos algo tan increíble como eso. Entonces, este será nuestro desafío en este momento, este es nuestro primer paso para crear Redes Neuronales Artificiales: ser capaces de recrear una neurona.

Entonces, ¿cómo lo hacemos? Bueno, antes que nada echemos un vistazo más de cerca a lo que realmente es una neurona. La imagen a continuación fue creada por primera vez por un neurocientífico español Santiago Ramón y Cajal en 1899:

The Neuron

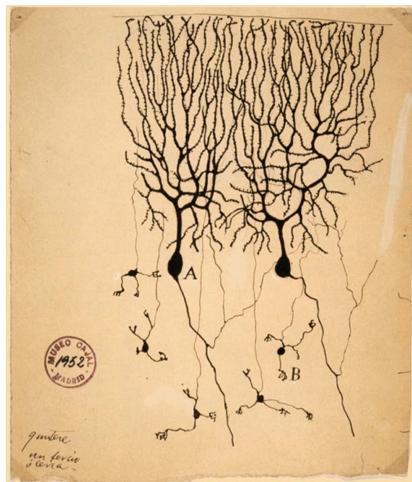


Image Source: Wikipedia

Artificial Intelligence for Business

© SuperDataScience

Este neurocientífico tiñó neuronas en el tejido cerebral real y las observó con un microscopio. Mientras los miraba, dibujó lo que vio, que es exactamente lo que vemos en la imagen de arriba. Hoy en día, la tecnología ha avanzado bastante permitiéndonos ver las neuronas mucho más cerca con más detalle para que podamos dibujar lo que parece en forma de diagrama.

The Neuron

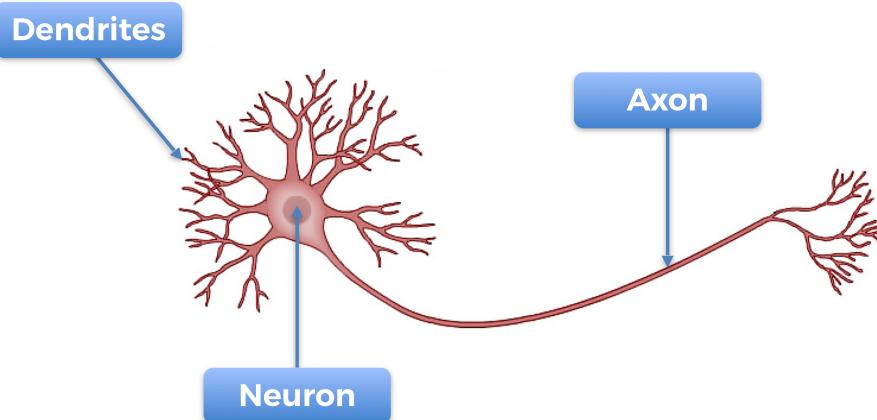


Image Source: Wikipedia

Artificial Intelligence for Business

© SuperDataScience

Esto es una neurona. Esta neurona intercambia señales con sus neuronas vecinas. Las dendritas son los receptores de la señal y el axón es el transmisor de la señal. Aquí hay una imagen de cómo funciona todo conceptualmente:

The Neuron

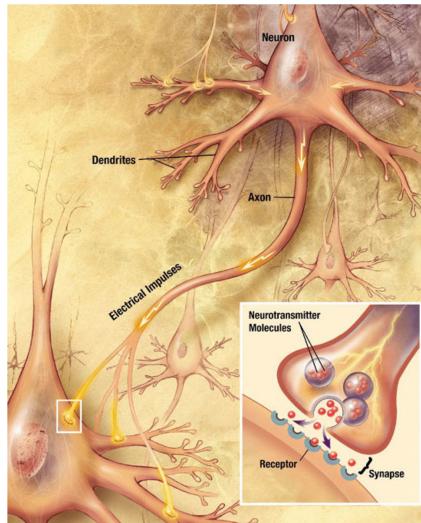


Image Source: Wikipedia

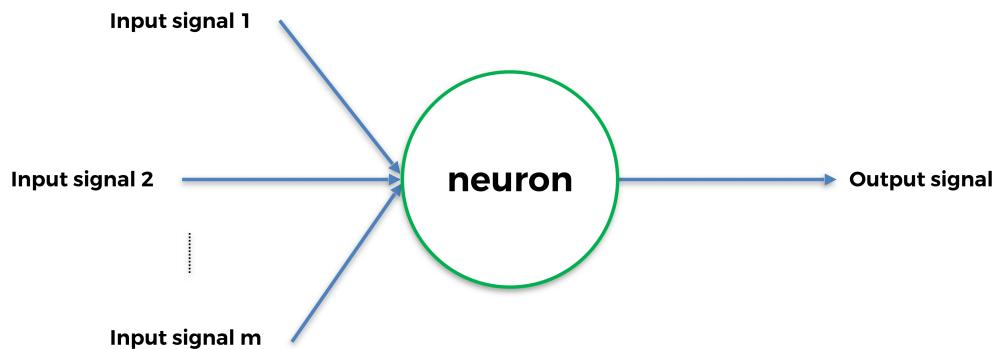
Artificial Intelligence for Business

© SuperDataScience

Podemos ver que las dendritas de la neurona están conectadas a los axones de otras neuronas por encima. Luego, la señal viaja por su axón y pasa a las dendritas de la siguiente neurona. Así es como están conectados y cómo funciona una neurona. Por lo tanto, ahora es el momento de pasar de la neurociencia a la tecnología.

Así es como se representa una neurona dentro de una red neuronal artificial:

The Neuron



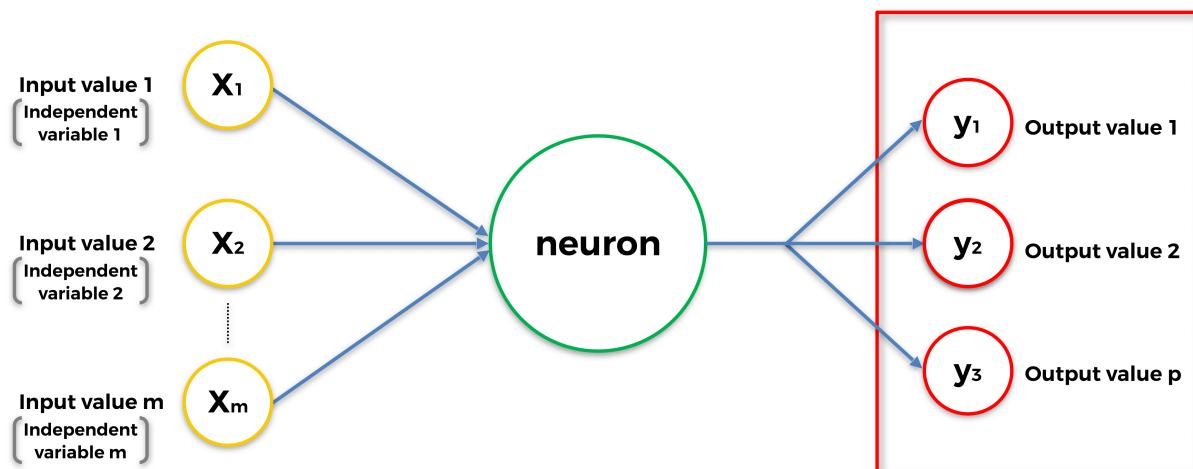
Artificial Intelligence for Business

© SuperDataScience

Al igual que una neurona humana, recibe algunas señales de entrada y tiene una señal de salida. La flecha azul que conecta las señales de entrada a la neurona, y la neurona a la señal de salida, son como las sinapsis en la neurona humana. Pero aquí, en la neurona de la máquina, ¿cuáles serán exactamente estas señales de

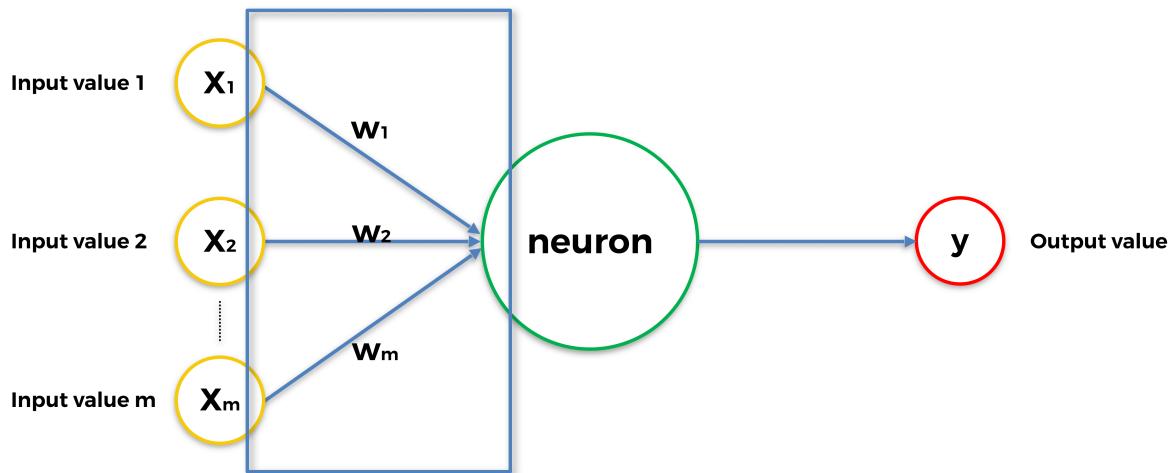
entrada y salida? Bueno, las señales de entrada serán las variables independientes escaladas que componen los estados del entorno, que recordemos en el caso práctico que nos ocupa son la temperatura del servidor, el número de usuarios y la velocidad de transmisión de datos, y la señal de salida será los valores de salida, que en el modelo de Deep Q-Learning son siempre los Q-Values. Por lo tanto, obtenemos la representación general de una neurona para ordenadores:

The Neuron



Y ahora para terminar con la neurona, agreguemos los últimos elementos que faltan en esta representación, pero también los más importantes: los pesos. A cada sinapsis (flecha azul) se le atribuirá un peso. Cuanto mayor sea el peso, más fuerte será la señal a través de la sinapsis. Y lo que es fundamental entender es que, estos pesos, serán lo que la máquina actualizará y actualizará con el tiempo para mejorar las predicciones. Veámoslo en el gráfico anterior, para asegurarnos de que los visualizamos correctamente antes de seguir:

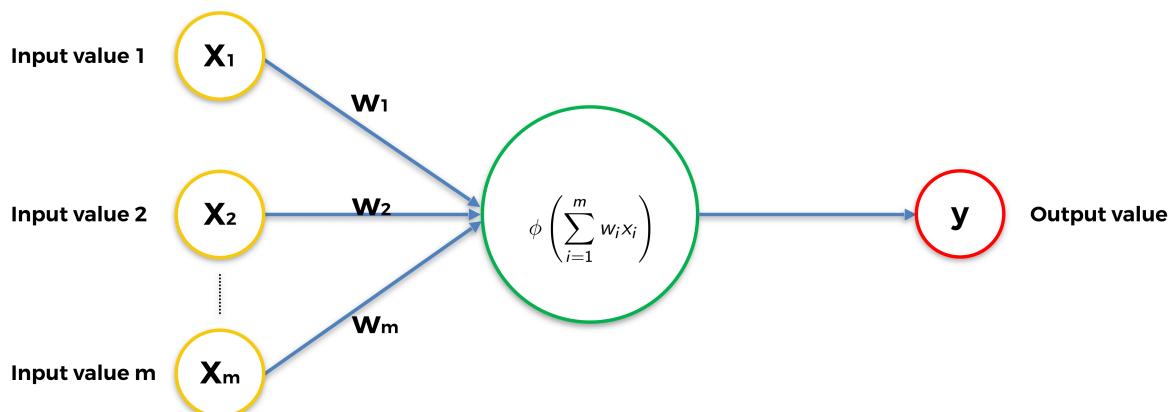
The Neuron



4.1.2 La Función de Activación

La función de activación es la función ϕ que opera dentro de la neurona, que tomará como entradas la combinación lineal de los valores de entrada multiplicados por sus pesos asociados, y sumados entre si y que devolverá el valor de salida:

The Neuron



calculado como

$$y = \phi \left(\sum_{i=1}^m w_i x_i \right)$$

¿Cuál será exactamente la función ϕ ?

Puede haber muchas de ellos, diferentes pero vamos a ver aquí los cuatro más usados, incluido, por supuesto, el que usamos en la Parte 2: Minimización de costos:

- La función de activación de umbral
- La función de activación sigmoidea
- La función de activación del rectificador
- La función de activación de la tangente hiperbólica

Vamos a verlas todas una por una:

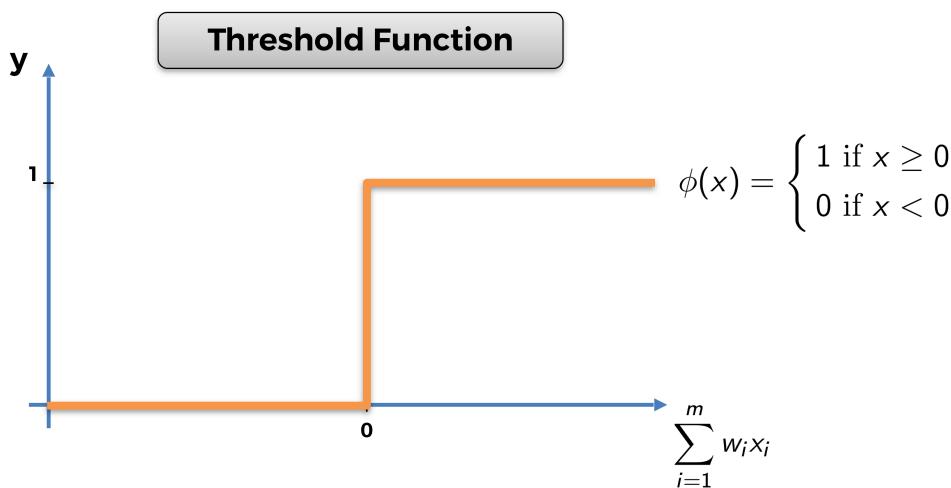
4.1.2.1 La función de activación de umbral

La función de activación de umbral, también llamada función escalón, se define simplemente como:

$$\phi(x) = \begin{cases} 1 & \text{si } x \geq 0 \\ 0 & \text{si } x < 0 \end{cases}$$

de modo que su representación gráfica es:

The Activation Function



Esto significa que la señal que pasa por la neurona será discontinua, y solo se activará si:

$$\sum_{i=1}^m w_i x_i \geq 0$$

4.1.2.2 La función de activación sigmoidea

Ahora echemos un vistazo a la siguiente función de activación: la función de activación sigmoidea.

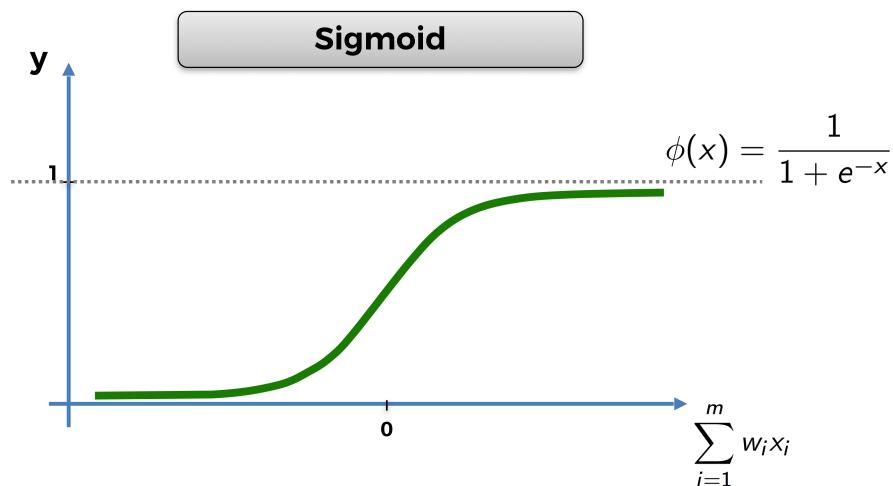
La función de activación sigmoidea es la más efectiva y ampliamente utilizada en Redes Neuronales Artificiales, pero principalmente dentro de la última capa oculta (si se trata de una red neuronal profunda compuesta de varias capas ocultas) que pasa la señal hacia la capa de salida.

La función de activación sigmoidea se define como:

$$\phi(x) = \frac{1}{1 + e^{-x}}$$

y da como resultado la siguiente representación gráfica:

The Activation Function



Esto significa que la señal que pasa por la neurona será continua y siempre se activará. Y cuanto mayor sea $\sum_{i=1}^m w_i x_i$, más fuerte será esa señal.

4.1.2.3 La función de activación rectificadora

Ahora echemos un vistazo a otra función de activación ampliamente utilizada: la función de activación rectificadora.

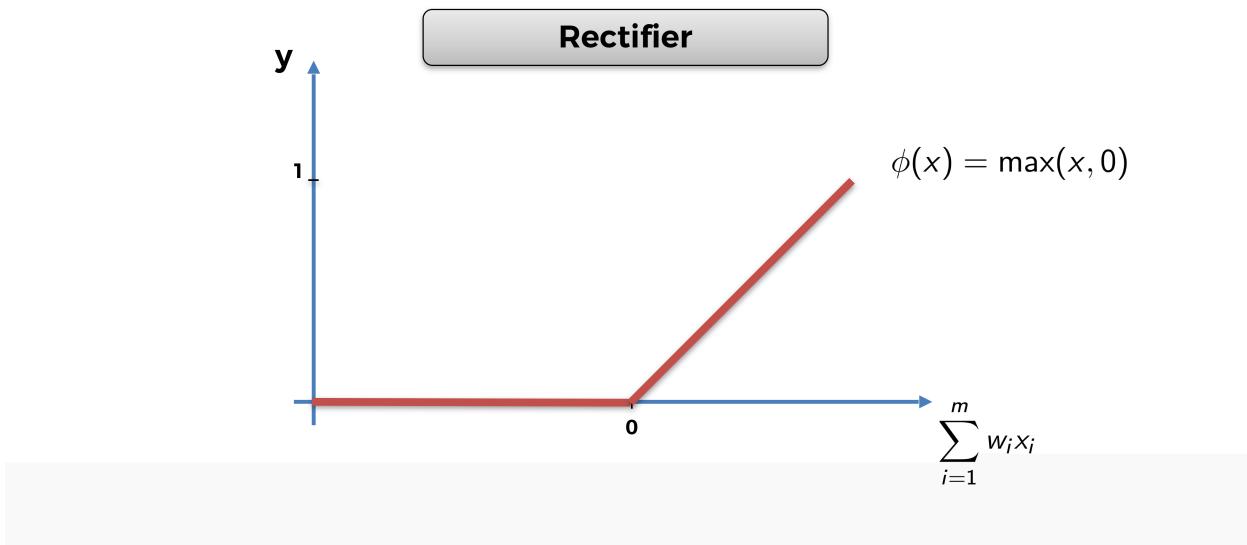
La solemos encontrar en la mayoría de las redes neuronales **profundas**, pero sobretodo dentro de las capas ocultas, a diferencia de la función sigmoidea que se usa más bien para la capa de salida.

La función de activación rectificadora se define simplemente como:

$$\phi(x) = \max(x, 0)$$

de modo que nos da lugar a la siguiente representación gráfica:

The Activation Function



Artificial Intelligence for Business

© SuperDataScience

Esto significa que la señal que pasa por la neurona será continua y solo se activará si:

$$\sum_{i=1}^m w_i x_i \geq 0$$

Y como mayor sea $\sum_{i=1}^m w_i x_i$ por encima de 0, más fuerte será la señal transmitida.

4.1.2.4 La función de activación de la tangente hiperbólica

Ahora echemos un vistazo a la siguiente función de activación: la función de activación de la tangente hiperbólica.

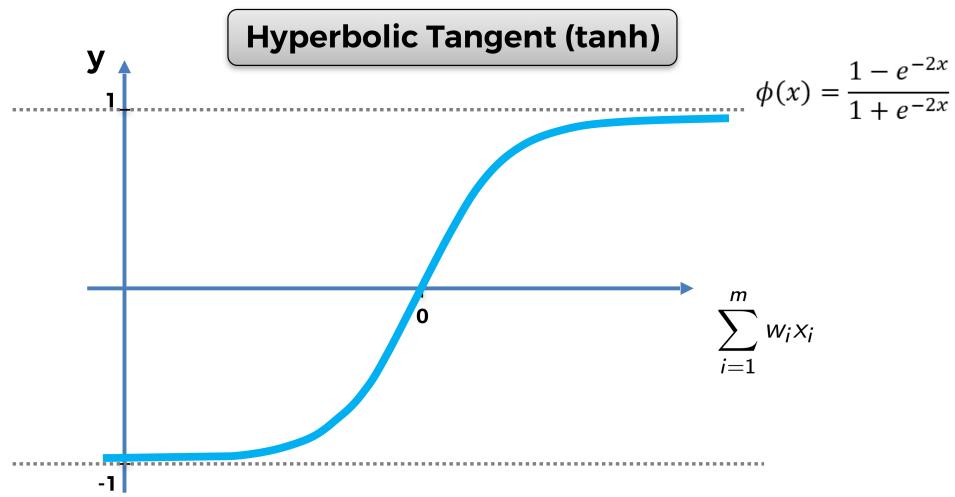
La función de activación de la tangente hiperbólica es la menos utilizada, aunque a veces puede ser una opción más relevante en algunas Redes Neuronales Artificiales, especialmente cuando las entradas están estandarizadas.

La función de activación de la tangente hiperbólica se define por lo siguiente:

$$\phi(x) = \frac{1 - e^{-2x}}{1 + e^{-2x}}$$

de modo que da lugar a la siguiente:

The Activation Function



Artificial Intelligence for Business

© SuperDataScience

Esto significa que la señal que pasa por la neurona será continua y siempre se activará. Cuanto más el valor de $\sum_{i=1}^m w_i x_i$ está por encima de 0, más fuerte será esa señal. Cuanto menos el valor de $\sum_{i=1}^m w_i x_i$ está por debajo de 0, más débil será esa señal.

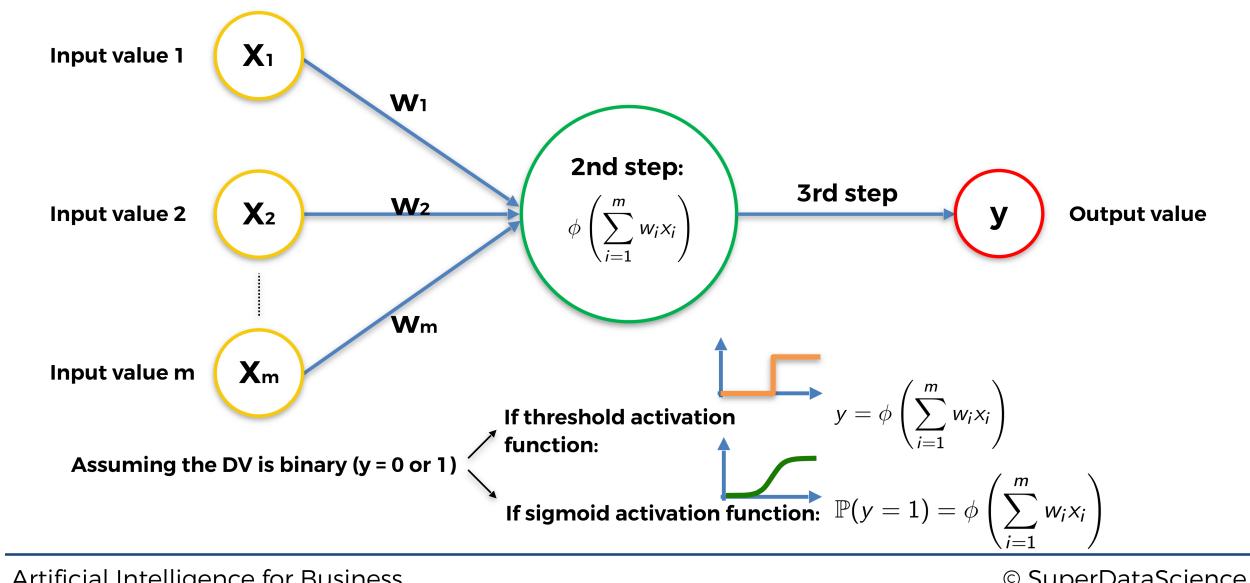
Eso plantea la pregunta: ¿qué función de activación deberíamos elegir? O también la pregunta que más me suelen hacer, ¿cómo sabemos cuál elegir?

Buenas noticias, la respuesta es simple, y vamos a verlo dentro de un pequeño algoritmo.

Eso realmente depende de lo que se devuelve como la variable dependiente. Si es un resultado binario 0 o 1, entonces una mejor opción sería la función de activación del umbral. Si lo que desea que se devuelva es la probabilidad de que la variable dependiente sea 1, entonces una opción excelente es la función de activación sigmoidea, ya que su curva sigmoidea se ajusta perfectamente a las probabilidades del modelo.

Aquí está este pequeño detalle resaltado en esta imagen:

The Activation Function



Artificial Intelligence for Business

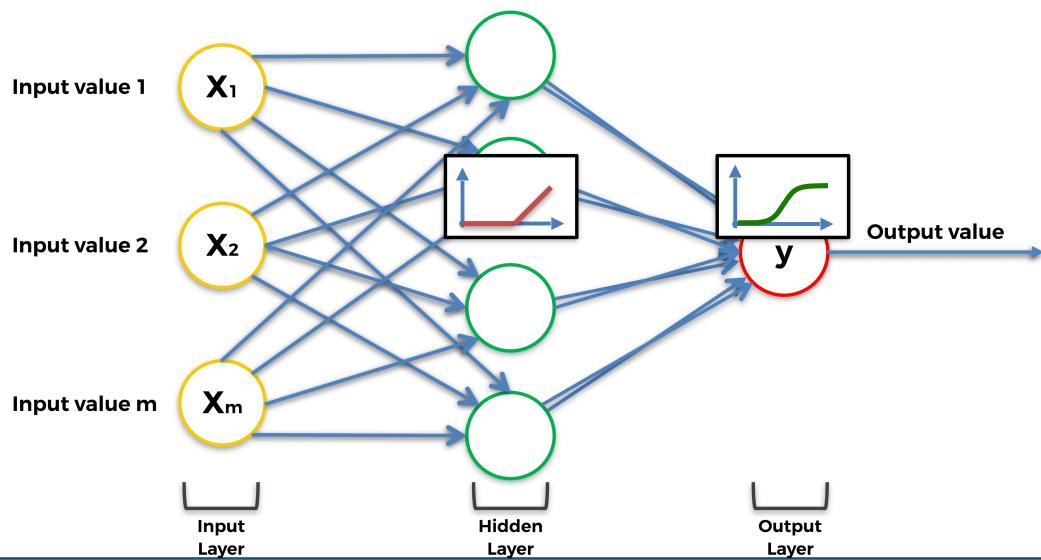
© SuperDataScience

Pero entonces, ¿cuándo debo usar las otras dos funciones de activación, es decir, la función de activación del rectificador y la función de activación de la tangente hiperbólica?

Nuevamente, las funciones de activación rectificador y tangente hiperbólica deben usarse dentro de las capas ocultas de una Red neuronal profunda (con más de una capa oculta), excepto en la última capa oculta que conduce a la capa de salida para la cual se recomienda usar un función de activación sigmoidea.

Recapitulemos esto nuevamente dentro de la siguiente imagen:

The Activation Function



Artificial Intelligence for Business

© SuperDataScience

Y, por último, ¿cómo elegir entre la función de activación del rectificador y la función de activación de la tangente hiperbólica en las capas ocultas? Pues es recomendable considerar usar la función de activación del rectificador cuando las entradas están normalizadas (escaladas entre 0 y 1), y la función de activación de Tangente hiperbólica cuando las entradas están estandarizadas (escaladas entre -1 y +1):

Standardisation	Normalisation
$x_{\text{stand}} = \frac{x - \text{mean}(x)}{\text{standard deviation } (x)}$	$x_{\text{norm}} = \frac{x - \min(x)}{\max(x) - \min(x)}$

Ahora pasemos a la siguiente sección para explicar cómo funcionan las redes neuronales.

4.1.3 ¿Cómo funcionan las Redes Neuronales?

Para explicar esto, consideremos el problema de predecir los precios inmobiliarios. Tenemos algunas variables independientes que vamos a utilizar para predecir el precio de casas y apartamentos. Para simplificar, y para poder representar todo en un gráfico, digamos que nuestras variables independientes (nuestros predictores) son las siguientes:

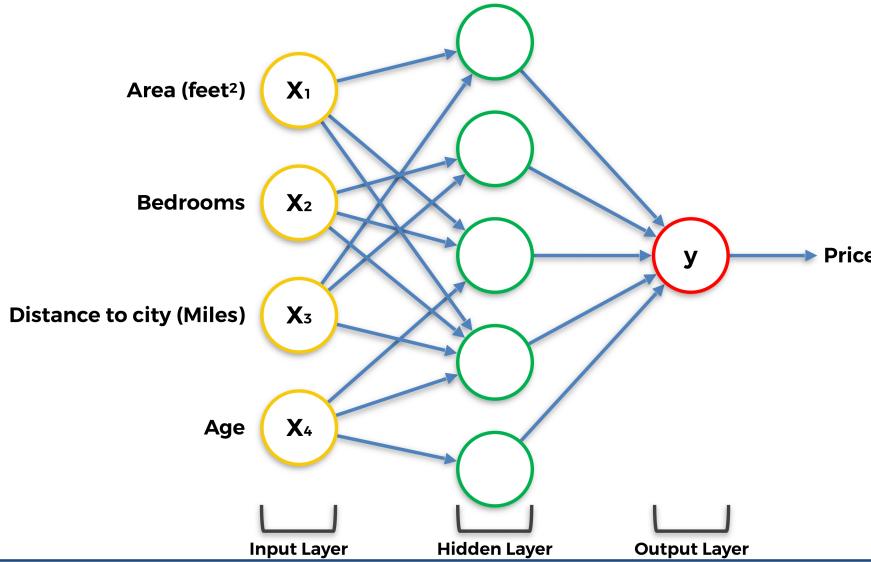
- Área (pies cuadrados)
- Número de habitaciones
- Distancia a la ciudad (Millas)
- Edad

Entonces, nuestra variable dependiente es, por supuesto, el precio del apartamento para predecir.

A cada una de las variables independientes se le atribuye un peso, de tal manera que cuanto mayor sea el peso, mayor será el efecto que tendrá la variable independiente en la variable dependiente, es decir, el predictor más fuerte será de la variable dependiente. Por lo tanto, tan pronto como nuevas entradas ingresen a la red neuronal, las señales se propagarán hacia adelante desde cada una de las entradas, llegando a las neuronas de la capa oculta. Luego, dentro de cada neurona de la capa oculta, se aplicará la función de activación, de modo que cuanto menor sea el peso de la entrada, la función de activación más bloqueará la señal procedente de esa entrada y cuanto mayor será el peso de esa entrada, la función de activación más permitirá que pase la señal a través de ella. Y finalmente, todas las señales procedentes de las neuronas ocultas, más o menos bloqueadas por las funciones de activación respectivas, se propagan hacia la capa de salida, para devolver el resultado final, es decir, la predicción del precio.

Representemos esto en el siguiente gráfico:

How Do Neural Networks Work?



4.1.4 ¿Cómo aprenden las Redes Neuronales?

En pocas palabras, las redes neuronales aprenden actualizando, durante muchas iteraciones llamadas epochs, los pesos de todas las entradas y neuronas ocultas (cuando tienen varias capas ocultas), hacia siempre el mismo objetivo: el de reducir el error de pérdida entre las predicciones y los valores reales.

De hecho, para que las Redes Neurales aprendan, necesitamos los valores reales, que también se denominan objetivos. En nuestro ejemplo anterior sobre la fijación de precios inmobiliarios, los valores reales son los precios reales de las casas y apartamentos en ventas. Estos precios reales dependen de las variables independientes enumeradas anteriormente (área, número de habitaciones, distancia a la ciudad y edad), y la red neuronal aprenderá a hacer mejores predicciones de estos precios, ejecutando el siguiente proceso:

- La red neuronal primero propaga las señales procedentes de las variables independientes de entrada x_1 , x_2 , x_3 y x_4 .
- Luego obtiene el precio predicho \hat{y} en la capa de salida.
- Luego calcula el error de pérdida C entre el precio predicho \hat{y} (predicción) y el precio real y (objetivo):

$$C = \frac{1}{2}(\hat{y} - y)^2$$

- Luego, este error de pérdida se propaga hacia atrás dentro de la red neuronal, de derecha a izquierda en nuestra representación.
- Luego, en cada una de las neuronas, la red neuronal ejecuta una técnica llamada Gradiente Descendente (que discutiremos en la siguiente sección), para actualizar los pesos en la dirección de reducción de pérdidas, es decir, en nuevos pesos que reducen el error de pérdida C .

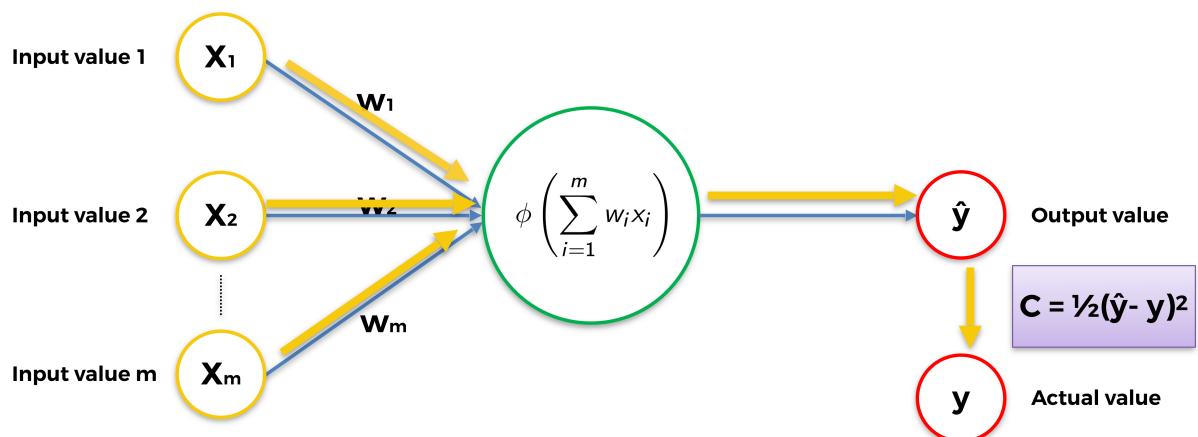
-
- Luego, todo este proceso se repite muchas veces, con cada vez nuevas entradas y nuevos objetivos, hasta que obtenemos el rendimiento deseado (detención temprana o *early stopping*) o la última iteración (número de iteraciones elegidas en la implementación).

Representemos las dos fases principales, Propagación hacia adelante y Propagación hacia atrás, de todo este proceso en los dos gráficos siguientes:

4.1.5 Propagación hacia adelante and propagación hacia atrás

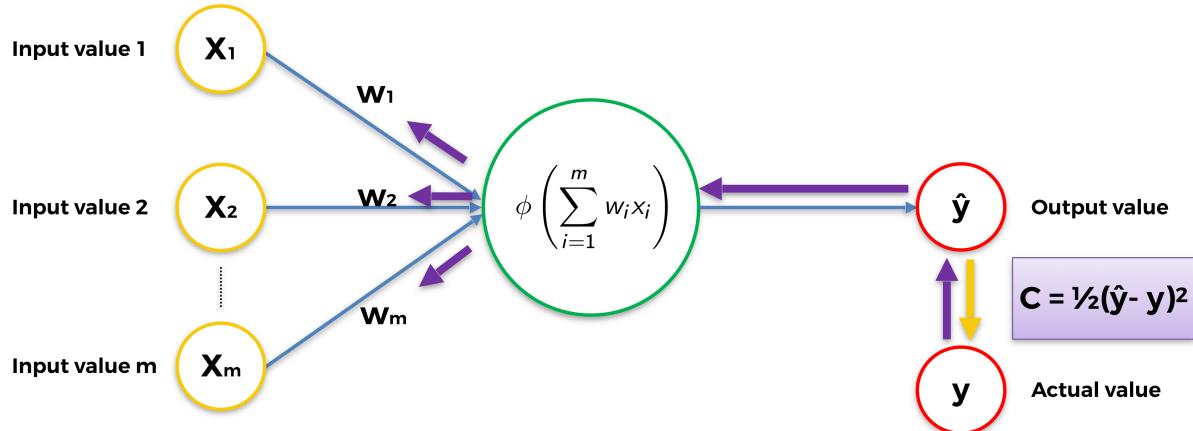
Fase 1: Propagación hacia adelante:

How do Neural Networks learn?



Fase 2: Propagación hacia atrás:

How do Neural Networks learn?



4.1.6 Gradiente Descendente

4.1.6.1 Introducción al Gradiente Descendente

Cuando las personas hablan de Machine Learning o Deep Learning, hablan principalmente de los algoritmos que se utilizan. Pero las preguntas reales son, ¿por qué se considera que esos algoritmos son algoritmos de Aprendizaje automático o Aprendizaje profundo y otros no? ¿Cuál es la técnica subyacente que los diferencia?

La respuesta a la primera pregunta es bastante intuitiva: se considera que esos algoritmos aprenden sus parámetros por sí mismos. Esta propiedad no era muy común antes y la mayoría de los algoritmos fueron ajustados a mano por ingenieros para lograr una especificación / meta requerida.

Pero entonces apareció la idea de incorporar el Gradiente Descendente a los mismos y la mayoría de los algoritmos que no funcionaban antes, de repente tuvieron sentido y comenzaron a optimizarse.

Entonces, ¿es mágica la técnica del Gradiente Descendente? Bueno, para alguien lo podría ser, pero para nosotros es un algoritmo matemático que se utiliza para optimizar un modelo que tiene sus parámetros internos (pesos). O, para ser más técnicos, veamos qué dice Wikipedia al respecto:

“El Gradiente Descendente es un algoritmo de optimización iterativa de primer orden para encontrar el mínimo de una función”.

Esa es una definición correcta pero con poco contenido, y para alguien que recién está comenzando, ¡encima es aterradora! Vamos a desglosarlo:

- **Algoritmo:** en pocas palabras, es un plan sobre cómo resolver un problema. El ejemplo cotidiano de un algoritmo sería una receta de cocina.
- **Iterativo:** esto significa que utiliza algún tipo de bucle (para programadores, bucles `for` o `while`) para realizar pasos. Cada paso usa valores calculados previamente como entrada para el paso actual. Ahora, surge una pregunta: “¿Cuál es nuestro valor inicial?”. Responderemos sobre esto un poco más adelante a través de ejemplos.

-
- **Optimización:** intenta encontrar las mejores soluciones de acuerdo con algunos criterios que conducen a varias soluciones alternativas, pero solo una se considera la mejor.
 - **De primer orden** El Gradiente Descendente está utilizando la primera derivada de una función de criterio elegida (coste, pérdida) para encontrar cuál es una mejor solución para el problema dado.

Por lo tanto, cuando lo ponemos todo junto en palabras inteligibles, obtenemos la siguiente definición:

El Gradiente Descendente es un plan sobre cómo encontrar la mejor solución para un problema donde más de una solución es aceptable. Utiliza un objetivo para determinar qué tan lejos estamos de encontrar la mejor solución.

Hasta este punto, tenemos todo aclarado, excepto la función de costes.

El coste es el indicador que seguimos durante el proceso de optimización. En base a ese indicador, podemos decir qué tan lejos estamos del óptimo de una función. Un buen ejemplo del costo es el error cuadrático medio, que hemos visto anteriormente en este libro:

$$\text{MSE} = \frac{1}{n} \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

donde:

$$\begin{cases} \hat{y}_i \text{ es la predicción del modelo} \\ y_i \text{ es el objetivo (el valor actual)} \\ n \text{ es el número de muestras del data set} \end{cases}$$

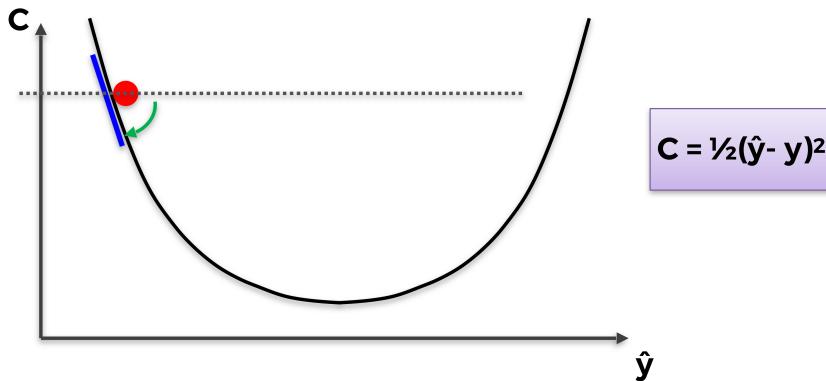
Cada algoritmo que utiliza el Gradiente Descendente como técnica de optimización tiene parámetros (pesos) que cambian durante el proceso de optimización. Cuando decimos que estamos buscando el mínimo de la función de pérdida, en realidad queremos decir que estamos buscando los valores de los pesos para los cuales la pérdida tiene el valor más bajo posible.

En consecuencia, para responder a nuestra segunda pregunta desde el principio, el punto en común que tienen todos los algoritmos de Aprendizaje automático desde la regresión lineal a las redes neuronales más complicadas es, de hecho, el Gradiente Descendente.

4.1.6.2 Idea del Gradiente Descendente

Como hemos visto, el Gradiente Descendente es una técnica de optimización que nos ayuda a encontrar el mínimo de una función de costo. Ahora visualicémoslo de la manera más intuitiva, como la siguiente bola en un bol (junto con un poco de spray matemático encima):

Gradient Descent



Imagina que esta es una sección transversal de un bol, dentro del cual dejamos caer una pequeña bola roja y dejamos que llegue al fondo del mismo. Después de un tiempo dejará de rodar, ya que ha encontrado el punto ideal para ello, en el fondo del tazón.

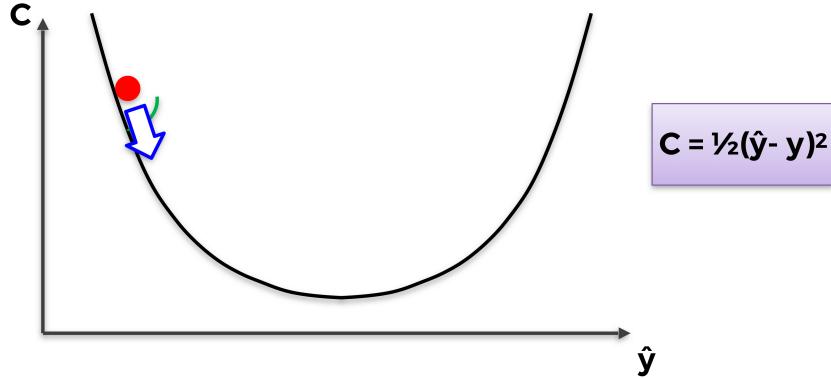
Puedes pensar en el Gradiente Descendente de la misma manera. Comienza en algún lugar del bol (valores iniciales de los parámetros) e intenta encontrar el fondo del mismo, o en otras palabras, el mínimo de una función de coste.

Veamos el ejemplo que se muestra en la imagen de arriba. Los valores iniciales de los parámetros han establecido nuestra bola en la posición que se muestra. En base a eso obtenemos algunas predicciones, que comparamos con nuestros valores objetivo. La diferencia entre estos dos conjuntos será nuestra pérdida para el conjunto actual de parámetros.

Luego calculamos la primera derivada de la función de coste, con respecto a los parámetros. De aquí proviene el nombre **Gradient**. Aquí, esta primera derivada nos da la pendiente de la tangente a la curva donde está la bola. Si la pendiente es negativa, como en la imagen de arriba, damos el siguiente paso hacia el lado derecho. Si la pendiente es positiva, damos el siguiente paso hacia el lado izquierdo.

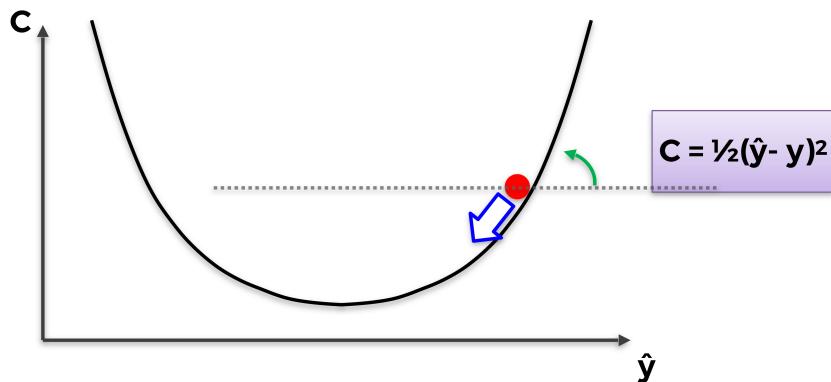
El nombre **Descendente** proviene del hecho de que siempre damos el siguiente paso que apunta hacia abajo, como se representa en el siguiente gráfico:

Gradient Descent



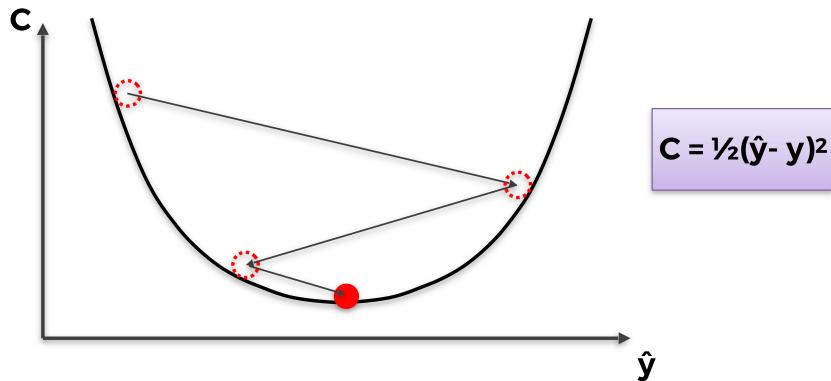
Ahora, en esta posición, nuestra pelota tiene una pendiente positiva, por lo que debemos dar el siguiente paso hacia la izquierda.

Gradient Descent



Eventualmente, al repetir los mismos pasos, la pelota terminará en el fondo del bol:

Gradient Descent

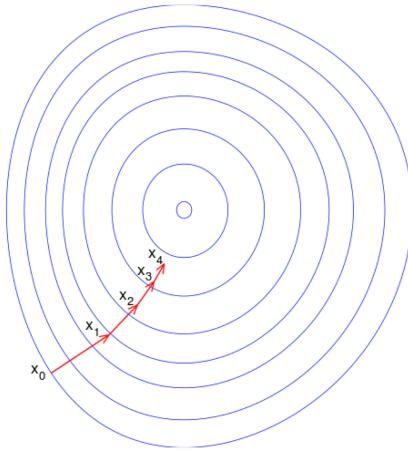


¡Y eso es! Así es como el Gradiente Descendente opera en una dimensión (un parámetro). Ahora puedes preguntarte:

Genial, pero ¿cómo funciona esto a gran escala? Hemos visto un ejemplo de optimización unidimensional, ¿qué pasa con dos o incluso 3 dimensiones?

Buena pregunta. El Gradiente Descendente garantiza que este enfoque se escala en tantas dimensiones como sea necesario, siempre que la función de costo sea convexa. De hecho, si la función de costo es convexa, el Gradiente Descendente encontrará siempre el mínimo **absoluto** de la función de costo. A continuación se muestra un ejemplo en 2 dimensiones:

Gradient Descent

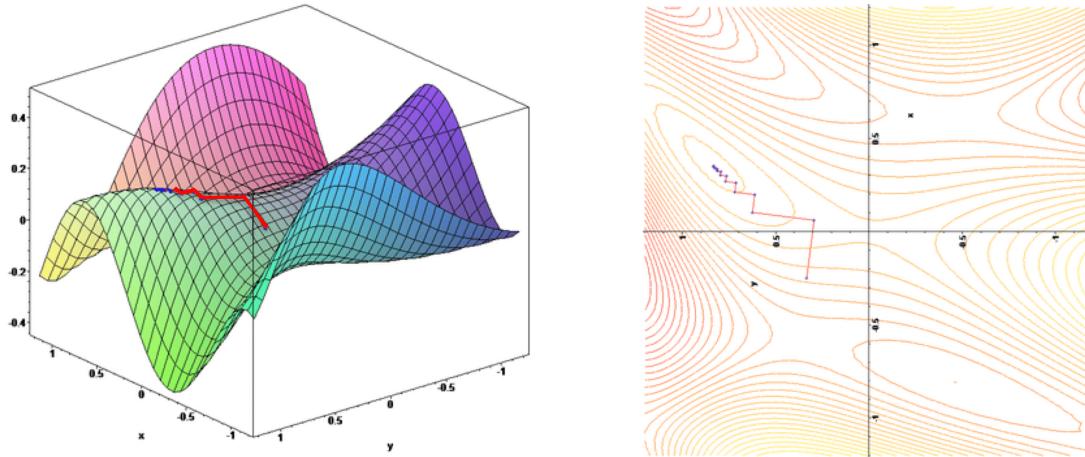


Artificial Intelligence for Business

© SuperDataScience

Sin embargo, si la función de coste no es convexa, solo encontrará un mínimo **local**. A continuación se muestra un ejemplo en 3 dimensiones:

Gradient Descent



Artificial Intelligence for Business

© SuperDataScience

Ahora que entendemos de qué se trata el gradiente descendente, es hora de estudiar las versiones más avanzadas y efectivas....

- Gradiente Descendente en bloques
- Mini-Batch Gradiente Descendente
- Stochastic Gradiente Descendente

4.1.6.3 Gradiente Descendente en bloques

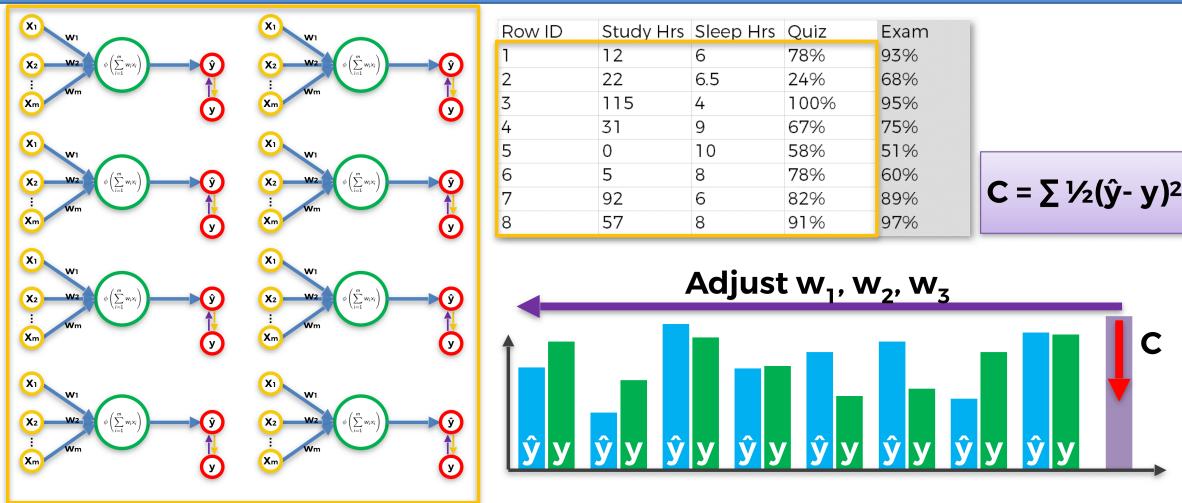
“Gradiente Descendente”, “Gradiente Descendente en Bloques”, “Gradiente Descendente Mini Batch”, “Gradiente Descendente Estocástico”.. Hay tantos términos y alguien que justo comienza a meterse en este mundillo puede encontrarlo muy confuso

La principal diferencia en todas estas versiones de Gradiente Descendente es la forma en que subministramos nuestros datos a un modelo y la frecuencia con la que actualizamos nuestros parámetros (pesos) para mover nuestra pequeña bola roja. Comencemos explicando el Gradiente Descendente en bloques.

Gradiente descendente en bloques es exactamente lo que hicimos en la Parte 2: Minimización de costes, donde recordemos que teníamos un lote de entradas que alimentaban la red neuronal, propagándolas hacia adelante para obtener al final un lote de predicciones, que a su vez se comparan con un lote de objetivos. El error de pérdida global entre las predicciones y los objetivos de los dos lotes se calcula como la suma de los errores de pérdida entre cada predicción y su objetivo asociado. Esa pérdida global se propaga nuevamente a la red neuronal, donde se realiza Gradiente Descendente o Gradiente Descendente estocástico para actualizar todos los pesos, de acuerdo con cómo fueron de responsables de ese error de pérdida global.

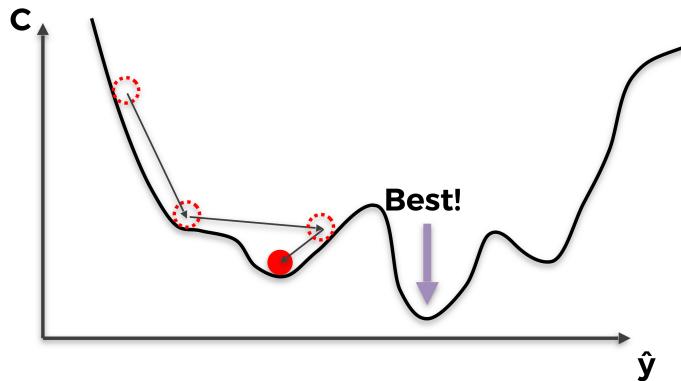
En la siguiente página a continuación hay un ejemplo de gradiente descendente en bloques. El problema a resolver es predecir la nota (de 0 a 100%) que los estudiantes obtienen en un examen, en función del tiempo dedicado a estudiar y el tiempo dedicado a dormir:

Stochastic Gradient Descent



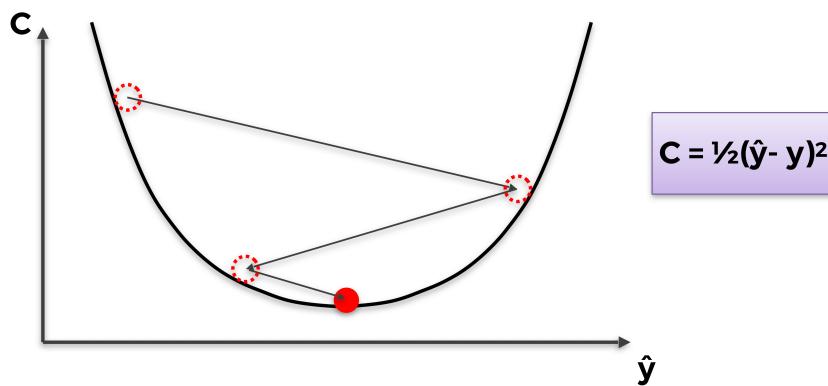
Una cosa importante a tener en cuenta en este gráfico anterior es que estas no son redes neuronales múltiples, sino una sola representada por actualizaciones de peso separadas. Y nuevamente, como podemos notar que en este ejemplo de Gradiente Descendente en bloques, alimentamos todos nuestros datos al modelo a la vez. Esto producirá actualizaciones colectivas de los pesos y una rápida optimización de la red. Sin embargo, también está el lado malo de esto. Existe una vez más la posibilidad de quedarse atascado en un mínimo local, como podemos ver en el siguiente gráfico a continuación:

Stochastic Gradient Descent



La razón por la que esto sucede se ha explicado un poco antes: es porque la función de coste en el gráfico anterior no es convexa. Y este tipo de optimización (Gradiente Descendente simple) requiere que la función de costo sea convexa. Si ese no es el caso, podemos encontrarnos atrapados en un mínimo local y nunca encontrar el mínimo global que tenga los parámetros óptimos. Por otro lado, a continuación se muestra un ejemplo de una función de coste convexa, la misma que vimos anteriormente:

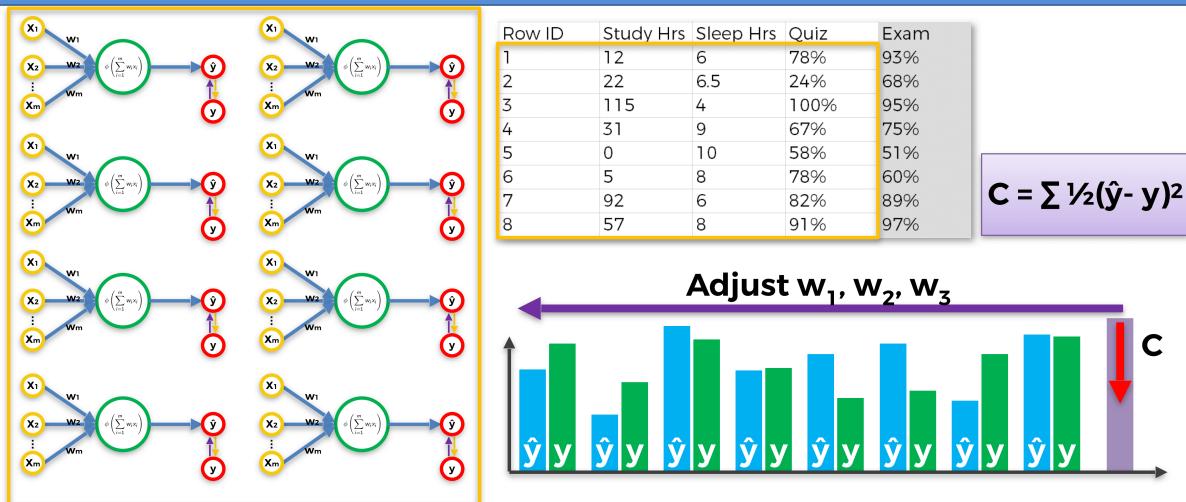
Gradient Descent



En pocas palabras, una función es convexa si solo tiene un mínimo global. Y el gráfico de una función convexa tiene la forma del bol.

Sin embargo, en la mayoría de los problemas, incluidos los problemas comerciales, la función de coste no será convexa (como en este mismo ejemplo gráfico en 3D a continuación), por lo que no permite que Gradiente Descendente funcione correctamente. Aquí es donde entra en juego el Gradiente Descendente Estocástico.

Stochastic Gradient Descent



4.1.6.4 Gradiente Descendente Estocástico

Gradiente Descendente Estocástico (SGD) viene a salvarnos de la catástrofe anterior. De hecho, proporciona mejores resultados en general, evitando que el algoritmo se atasque en un mínimo local. Sin embargo, como su nombre lo indica, es estocástico, o en otras palabras, aleatorio. Debido a esta propiedad, no importa cuántas veces ejecutemos el algoritmo, el proceso siempre será ligeramente diferente. Y eso, independientemente de la inicialización.

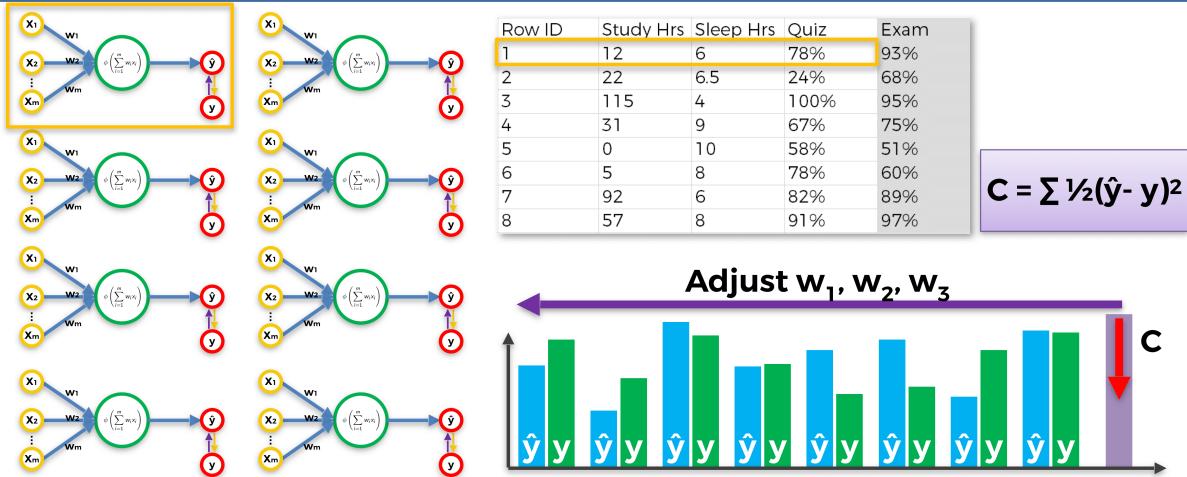
El Gradiente Descendente Estocástico no se ejecuta sobre todo el conjunto de datos a la vez, sino que se introducen las entradas una a una. Por lo tanto, el proceso es así:

1. Se introduce una sola observación
2. Se obtiene una única predicción
3. Se calcula el error de pérdida entre la predicción y el objetivo
4. Se vuelve a propagar el error de pérdida en la red neuronal
5. Se actualizan los pesos con Gradiente Descendente
6. Se repiten los pasos 1. a 5. a través de todo el conjunto de datos

Representemos las tres primeras iteraciones en las tres primeras entradas individuales para este mismo ejemplo dado anteriormente sobre la predicción de las puntuaciones en un examen:

Primera fila de entrada de observación:

Stochastic Gradient Descent

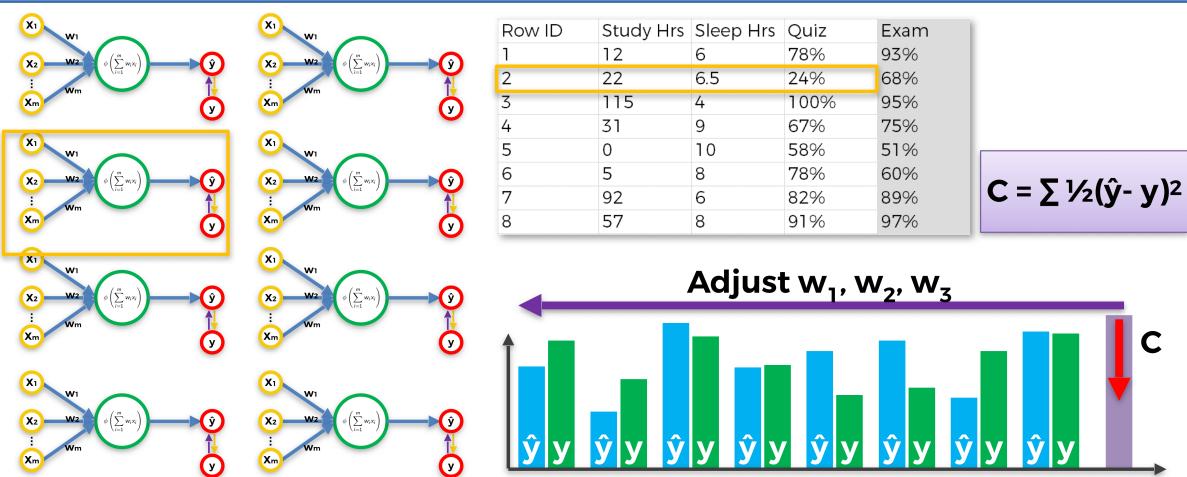


Artificial Intelligence for Business

© SuperDataScience

Segunda fila de entrada de observación:

Stochastic Gradient Descent

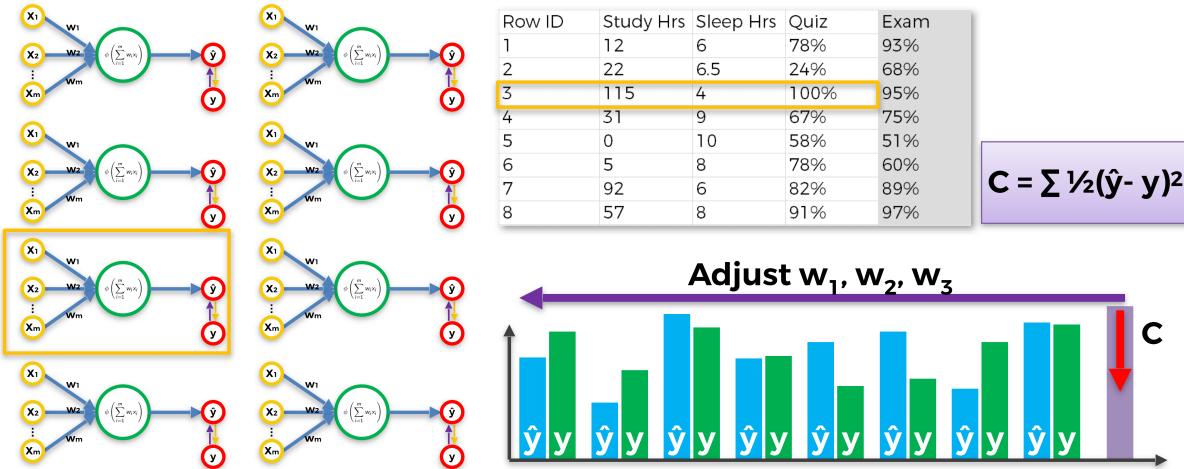


Artificial Intelligence for Business

© SuperDataScience

Tercera fila de entrada de observación:

Stochastic Gradient Descent



Artificial Intelligence for Business

© SuperDataScience

Cada uno de los tres gráficos anteriores es un ejemplo de una actualización de pesos ejecutada por el Gradiente Descendente Estocástico. Como podemos ver, cada vez introducimos una sola fila de observación desde nuestro conjunto de datos a la red neuronal, luego actualizamos los pesos en consecuencia y procedemos a la siguiente fila de entrada de observación.

A primera vista, el gradiente descendente estocástico parece más lento, porque introducimos cada fila por separado. Pero en realidad, es mucho más rápido debido al hecho de que no tenemos que cargar todo el conjunto de datos en la memoria, ni esperar a que todo el conjunto de datos pase por el modelo actualizando los pesos.

Para finalizar esta sección, repasemos la diferencia entre gradiente descendente en bloque y gradiente descendente estocástico, con el siguiente gráfico:

Stochastic Gradient Descent

Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

Upd w's ↙

Row ID	Study Hrs	Sleep Hrs	Quiz	Exam
1	12	6	78%	93%
2	22	6.5	24%	68%
3	115	4	100%	95%
4	31	9	67%	75%
5	0	10	58%	51%
6	5	8	78%	60%
7	92	6	82%	89%
8	57	8	91%	97%

Upd w's ↙

Batch Gradient Descent

Stochastic Gradient Descent

Artificial Intelligence for Business

© SuperDataScience

4.1.6.5 Gradiente Descendente Mini-Batch

El Gradiente Descendente Mini-Batch utiliza lo mejor de ambos mundos para combinar gradiente descendente en bloques con el gradiente descendente estocástico. Esto se hace alimentando la red neuronal con lotes de datos en lugar de alimentar filas de observaciones individuales una por una o todo el conjunto de datos a la vez.

Este enfoque es más rápido que el Gradiente Descendente Estocástico clásico y evita que se atasque en un mínimo local. Esto también ayuda cuando las personas no tienen suficientes recursos informáticos para cargar todo el conjunto de datos en la memoria, o suficiente potencia de procesamiento para obtener el beneficio completo del Gradiente Descendente Estocástico.

4.1.7 Optimizadores

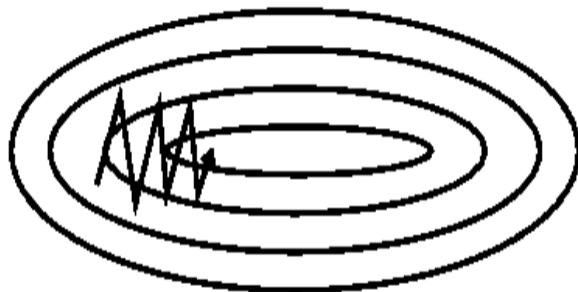
El optimizador es exactamente la herramienta que actualizará los pesos de la red neuronal a través del gradiente estocástico descendente. Hasta este punto, solo hemos mencionado el optimizador Adam (consulta la Parte 2: Minimización de costos), que es el optimizador más común utilizado para los modelos de Aprendizaje profundo y Aprendizaje por refuerzo profundo. Sin embargo, hay muchos más optimizadores que tienen sus propios beneficios y aplicaciones.

Veamos los optimizadores de gradiente descendente más famosos y ampliamente utilizados.

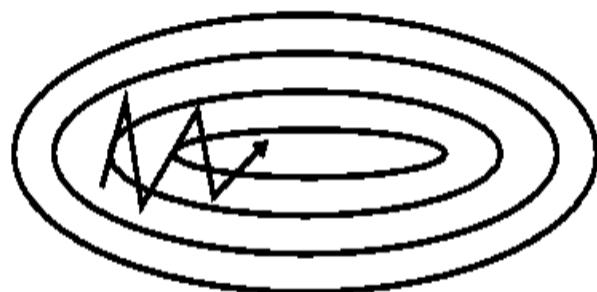
4.1.7.1 Optimizador Momentum

El Gradiente Descendente Estocástico clásico tiene oscilaciones muy grandes, lo que deja margen de mejora. El optimizador Momentum (también llamado optimizador de impulso) maneja estas grandes oscilaciones agregando fracciones de las direcciones calculadas en el paso anterior al paso actual. Esto amplifica la

velocidad de la actualización de dirección actual. En el gráfico a continuación podemos ver y comparar el SGD clásico y el SGD Momentum en acción:



Classical SGD optimizer



Momentum SGD optimizer

Los beneficios de optimizador Momentum son los siguientes:

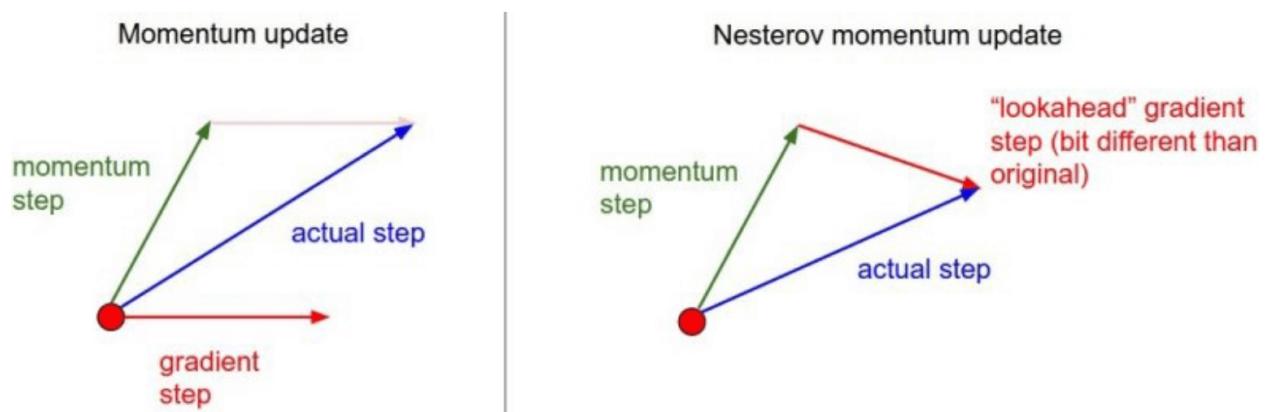
- Convergencia más rápida
- Menos oscilaciones

Pero el optimizador Momentum también tiene inconvenientes, que son los siguientes:

- Tendencia a sobrepasar el mínimo global de la función de coste debido al impulso.
- Menos frecuente en las librerías de Deep Learning, lo que requiere el conocimiento adicional de programación para su implementación.

4.1.7.2 El optimizador de gradiente acelerado de Nesterov

Yuri Nesterov resolvió el problema de sobrepasar el mínimo global del optimizador Momentum invirtiendo el orden de cálculo en la fórmula de actualización:



4.1.7.3 El optimizador AdaGrad (gradientes adaptativos)

La idea de adaptar nuestras actualizaciones de acuerdo con la pendiente de la función de error, proveniente del optimizador de Nesterov, se toma y aplica en el optimizador AdaGrad al tiempo que se optimiza también la tasa de aprendizaje.

Por lo tanto, en el optimizador AdaGrad tenemos el mismo principio, no solo aplicado en gradientes sino también en la tasa de aprendizaje.

Estos son los beneficios de este optimizador:

- El optimizador AdaGrad permite realizar grandes actualizaciones para parámetros poco frecuentes.
- Y permite realizar pequeñas actualizaciones para parámetros frecuentes.

Y aquí están los inconvenientes:

- La tasa de aprendizaje siempre disminuye, y esto podría llevar a actualizaciones muy pequeñas, si es que las hay.
- Es menos frecuente en las librerías de Deep Learning, lo que requiere el conocimiento necesario para su implementación.

4.1.7.4 El optimizador AdaDelta

El AdaDelta Optimizer se inventó para corregir ese problema de disminución de la tasa de aprendizaje del optimizador AdaGrad (el primer inconveniente anterior). No es necesario dar más detalles, solo necesitamos presentar el optimizador AdaDelta y su particularidad para comprender la fortaleza del optimizador más utilizado y más eficaz: el optimizador Adam

4.1.7.5 El optimizador Adam (Adaptive Moment Estimation)

El optimizador Adam es una mejora enorme con respecto al AdaDelta. La idea detrás de esto es almacenar en una memoria los cambios de impulso, ya que calculamos la tasa de aprendizaje para cada parámetro por separado.

Conviene recordar bien los beneficios del optimizador Adam, que deben tenerse en cuenta al construir una red neuronal.

- Es uno de los optimizadores más potentes.
- Viene implementado de serie en la mayoría de librerías de Deep Learning (Keras, TensorFlow, PyTorch). No lo echarás en falta.

Por supuesto, este es el que usamos al construir el Cerebro Artificial de nuestra IA en la Parte 2 - Minimización de los costes. Veamos nuevamente el código que construye este cerebro artificial y notemos, en la última línea de código, la simplicidad de seleccionar el Adam Optimizer al crearlo:

```
# Inteligencia Artificial aplicada a Negocios y Empresas – Caso Práctico 2
# Construcción del cerebro

# Importar las librerías
from keras.layers import Input, Dense
from keras.models import Model
from keras.optimizers import Adam
```

```

# CONSTRUCCIÓN DEL CEREBRO

class Brain(object):

    # CONSTRUCCIÓN DE UNA RED NEURONAL TOTALMENTE CONECTADA EN EL MÉTODO DE INICIALIZACIÓN

    def __init__(self, learning_rate = 0.001, number_actions = 5):
        self.learning_rate = learning_rate

        # CONSTRUCCIÓN DE LA CAPA DE ENTRADA COMPUESTA DE LOS ESTADOS DE ETRADA
        states = Input(shape = (3,))

        # CONSTRUCCIÓN DE LAS DOS CAPAS OCULTAS TOTALMENTE CONECTADAS
        x = Dense(units = 64, activation = 'sigmoid')(states)
        y = Dense(units = 32, activation = 'sigmoid')(x)

        # CONSTRUCCIÓN DE LA CAPA DE SALIDA, TOTALMENTE CONECTADA A LA ÚLTIMA CAPA OCULTA
        q_values = Dense(units = number_actions, activation = 'softmax')(y)

        # ENSAMBLAR LA ARQUITECTURA COMPLETA EN UN MODELO DE KERAS
        self.model = Model(inputs = states, outputs = q_values)

        # COMPILAR EL MODELO CON LA FUNCIÓN DE PÉRDIDAS DE ERROR CUADRÁTICO MEDIO Y EL OPTIMIZADOR
        self.model.compile(loss = 'mse', optimizer = Adam(lr = learning_rate))

```

4.2 Anexo 2: Tres modelos adicionales de IA

Como bonificación, en esta sección proporcionamos tres modelos adicionales de IA, más cercanos al estado del arte. Sin embargo, estos modelos de IA no están necesariamente adaptados para resolver problemas comerciales, sino más bien para resolver tareas específicas como entrenar a un robot virtual para caminar. Vamos a estudiar tres modelos poderosos, incluidos dos en la rama de Aprendizaje por refuerzo profundo de IA, y uno en la rama de políticas de gradiente (Policy Gradient):

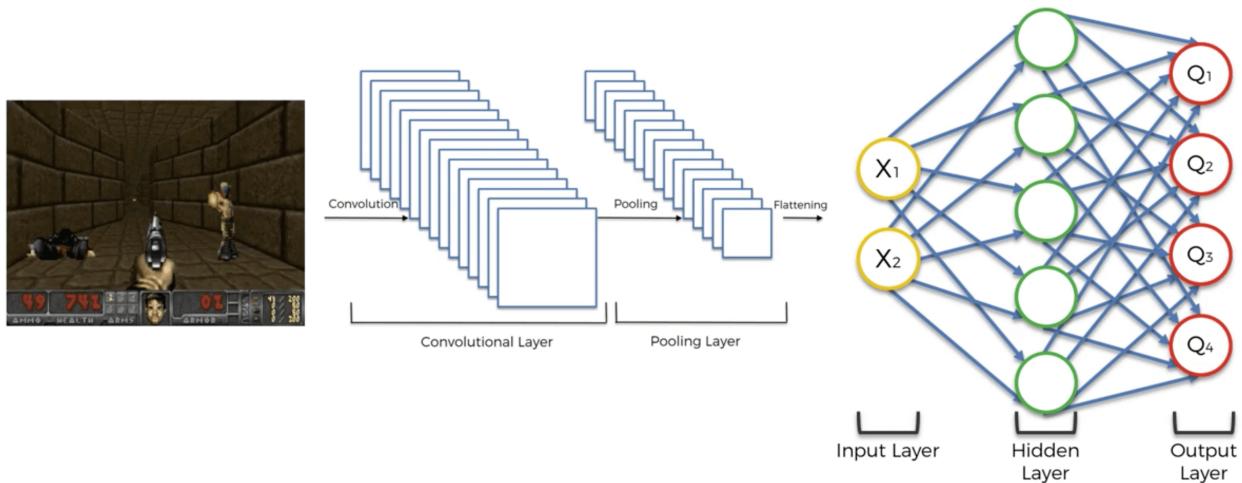
- Aprendizaje convolucional Q-profundo (Deep Reinforcement Learning)
- A3C (Deep Reinforcement Learning)
- Búsqueda aleatoria aumentada (Policy Gradient)

4.2.1 Aprendizaje convolucional Q-profundo

En la sección anterior, nuestras entradas eran valores codificados por vectores que definían los estados del entorno. Pero dado que un vector codificado no conserva la estructura espacial de una imagen, esta no es la mejor forma de describir un estado. La estructura espacial es realmente importante porque nos da más información para predecir el próximo estado, y predecir el siguiente estado es, por supuesto, esencial para que nuestra IA sepa cuál es el próximo movimiento correcto. Por lo tanto, debemos preservar la estructura espacial y, para hacerlo, nuestras entradas deben ser imágenes en 3D (2D para la matriz de píxeles más una dimensión adicional para los colores). En ese caso, las entradas son simplemente las imágenes de la pantalla en sí, exactamente como lo que ve un humano cuando ve. Siguiendo esta analogía, la IA actúa como un

humano: observa las imágenes de entrada de la pantalla al jugar, las imágenes de entrada entran en una red neuronal convolucional (el cerebro para un humano) que detectará el estado en cada imagen. Sin embargo, esta red neuronal convolucional no contiene capas de agrupación, porque perderían la ubicación de los objetos dentro de la imagen y, por supuesto, la IA debe realizar un seguimiento de los objetos. Por lo tanto, solo conservamos las capas convolucionales, y luego al aplanarlas en un vector unidimensional, obtenemos la entrada de nuestra red de Deep Q-Learning anterior. Entonces se está ejecutando el mismo proceso.

Por lo tanto, en resumen, Deep Convolutional Q-Learning es lo mismo que Deep Q-Learning, con la única diferencia de que las entradas ahora son imágenes, y se agrega una red neuronal convolucional al comienzo de la red Deep Q-Learning totalmente conectada a detectar los estados (o simplemente los objetos) de las imágenes.



4.2.2 Asynchronous Actor-Critic Agents (A3C)

4.2.2.1 Idea del A3C

Hasta ahora, la acción que se ejecuta en cada momento ha sido la salida de una red neuronal, como si solo un agente estuviera decidiendo la estrategia para jugar el juego. Este ya no será el caso con A3C. Esta vez, vamos a tener varios agentes, cada uno interactuando con su propia copia del entorno. Digamos que hay n agentes A_1, A_2, \dots, A_n .

Cada agente comparte dos redes: el actor y el crítico. El crítico evalúa los estados actuales, mientras que el actor evalúa los posibles valores en el estado actual. El actor está acostumbrado a tomar decisiones. En cada época del entrenamiento para el agente, selecciona la última versión de las redes compartidas y usa al actor durante n pasos para tomar una decisión. Sobre los n pasos, recopila todos los nuevos estados observados, los valores de estos nuevos estados, las recompensas, etc. Despues de los n pasos, el agente utiliza las observaciones recopiladas para actualizar los modelos compartidos. Los tiempos de época y, por lo tanto, los tiempos de actualizaciones de la red compartida por el agente no son sincrónos, de ahí el nombre.

De esa manera, si un agente desafortunado comienza a quedar atrapado en una política subóptima pero atractiva, alcanzará ese estado, porque otros agentes también actualizaron la política compartida antes de que el agente se atasque, y continuará una exploración efectiva.

Para explicar las reglas de actualización del actor y el crítico, veamos las redes como funciones que dependen de los vectores de los parámetros θ (para el actor) y θ_v (para el crítico).

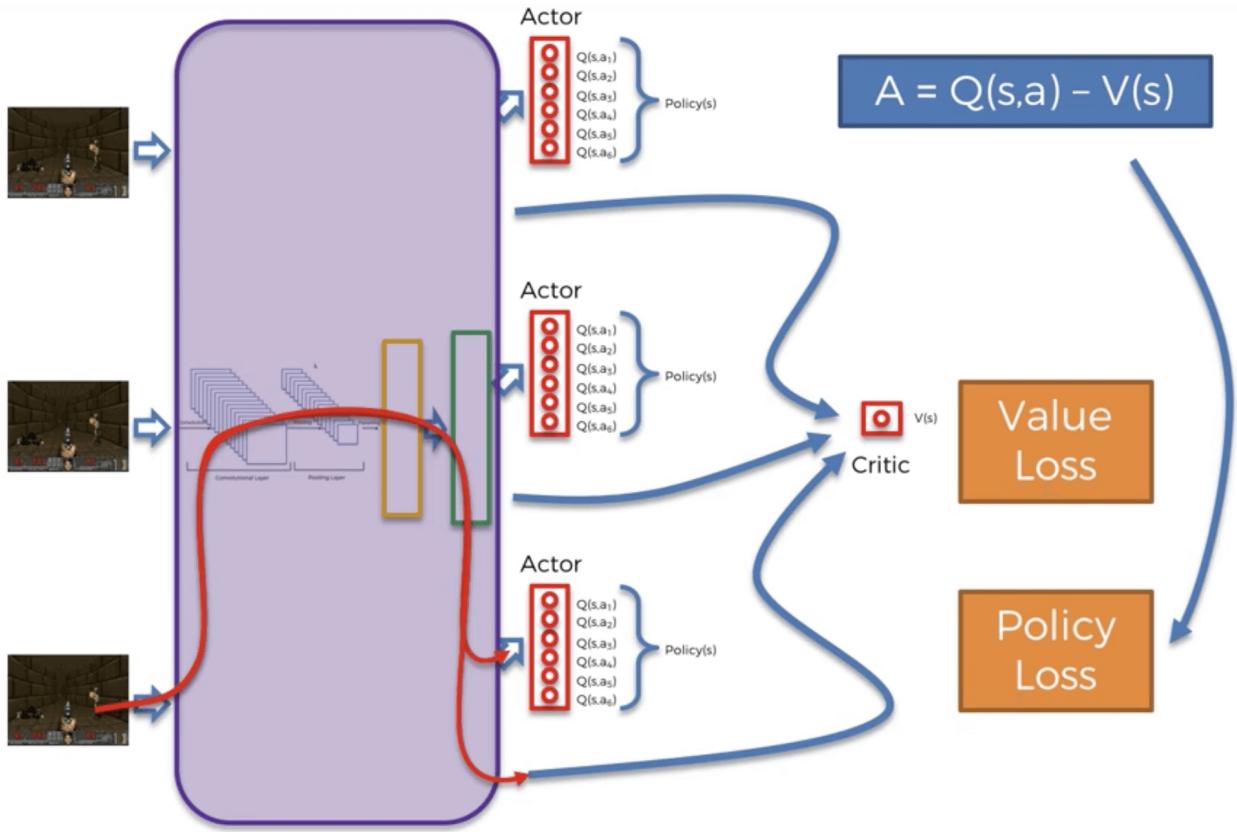
4.2.2.2 El proceso al completo del A3C

El algoritmo oficial A3C es el del paper de Google DeepMind, Métodos asíncronos para el aprendizaje por refuerzo profundo. En este artículo se encuentra precisamente dicho algoritmo S3:

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

```
// Assume global shared parameter vectors  $\theta$  and  $\theta_v$  and global shared counter  $T = 0$ 
// Assume thread-specific parameter vectors  $\theta'$  and  $\theta'_v$ 
Initialize thread step counter  $t \leftarrow 1$ 
repeat
    Reset gradients:  $d\theta \leftarrow 0$  and  $d\theta_v \leftarrow 0$ .
    Synchronize thread-specific parameters  $\theta' = \theta$  and  $\theta'_v = \theta_v$ 
     $t_{start} = t$ 
    Get state  $s_t$ 
    repeat
        Perform  $a_t$  according to policy  $\pi(a_t|s_t; \theta')$ 
        Receive reward  $r_t$  and new state  $s_{t+1}$ 
         $t \leftarrow t + 1$ 
         $T \leftarrow T + 1$ 
    until terminal  $s_t$  or  $t - t_{start} == t_{max}$ 
     $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \end{cases}$  // Bootstrap from last state
    for  $i \in \{t - 1, \dots, t_{start}\}$  do
         $R \leftarrow r_i + \gamma R$ 
        Accumulate gradients wrt  $\theta'$ :  $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i; \theta')(R - V(s_i; \theta'_v))$ 
        Accumulate gradients wrt  $\theta'_v$ :  $d\theta_v \leftarrow d\theta_v + \partial (R - V(s_i; \theta'_v))^2 / \partial \theta'_v$ 
    end for
    Perform asynchronous update of  $\theta$  using  $d\theta$  and of  $\theta_v$  using  $d\theta_v$ .
until  $T > T_{max}$ 
```

Figure 4.1: A3C algorithm (<https://arxiv.org/pdf/1602.01783.pdf>)



En la figura anterior, podemos ver claramente los tres A del A3C:

- **Asynchronous:** Hay varios agentes, cada uno con su propia copia del entorno, y todos asíncronos (jugando al juego en diferentes momentos).
- **Advantage:** La ventaja es la diferencia entre la predicción del actor, $Q(s, a)$, y la predicción del crítico, $V(s)$:

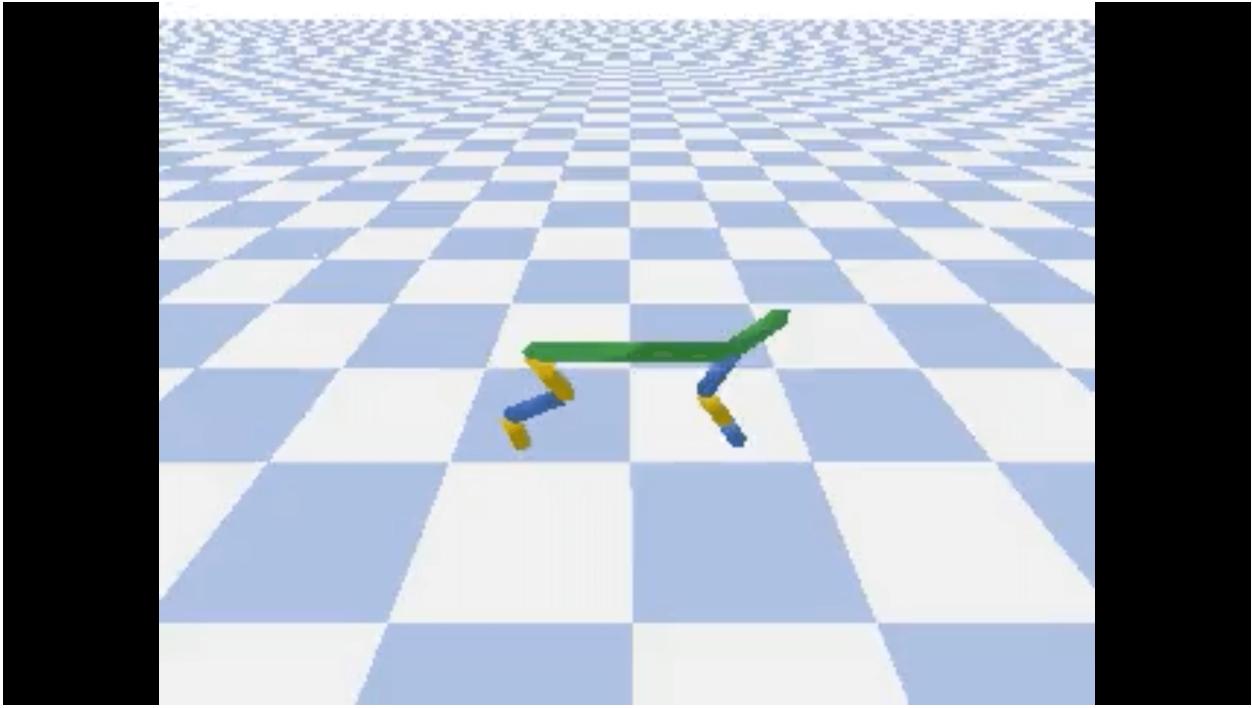
$$A = Q(s, a) - V(s)$$

- **Actor-Critic:** Por supuesto, podemos ver al actor y el crítico, que por lo tanto generan dos pérdidas diferentes: la pérdida de la política y la pérdida de valor. La pérdida de la política es la pérdida relacionada con las predicciones del actor. La pérdida de valor es la pérdida relacionada con las predicciones del crítico. Durante muchas épocas del entrenamiento, estas dos pérdidas se propagarán nuevamente a la Red Neural, y luego se reducirán con un optimizador a través del gradiente descendente estocástico.

4.2.3 Búsqueda aleatoria aumentada

4.2.3.1 Problema a resolver

Queremos construir y entrenar una IA que camina o corre por un campo. El campo es un terreno plano que se ve así:



En este mismo campo puedes ver un *Half-Cheetah*. Este será uno de los agentes que entrenaremos para caminar en este campo. Tanto el campo como el agente forman lo que llamamos un entorno, que pertenece a PyBullet, la interfaz oficial de Python para Bullet Physics SDK especializada en Simulación de Robótica y Aprendizaje ‘pr refuerzo, construida y desarrollada por Erwin Coumans. Para obtener más información, puedes leer en este documento o también puede consultar su página de GitHub.

4.2.3.2 Solución de IA

La solución a nuestro problema es un modelo de IA muy reciente llamado **ARS**, o **Búsqueda aleatoria aumentada**. El documento de investigación relacionado fue publicado por Horia Mania, Aurelia Guy y Benjamin Recht el 20 de marzo de 2018. Puede leer aquí el documento de investigación completo.

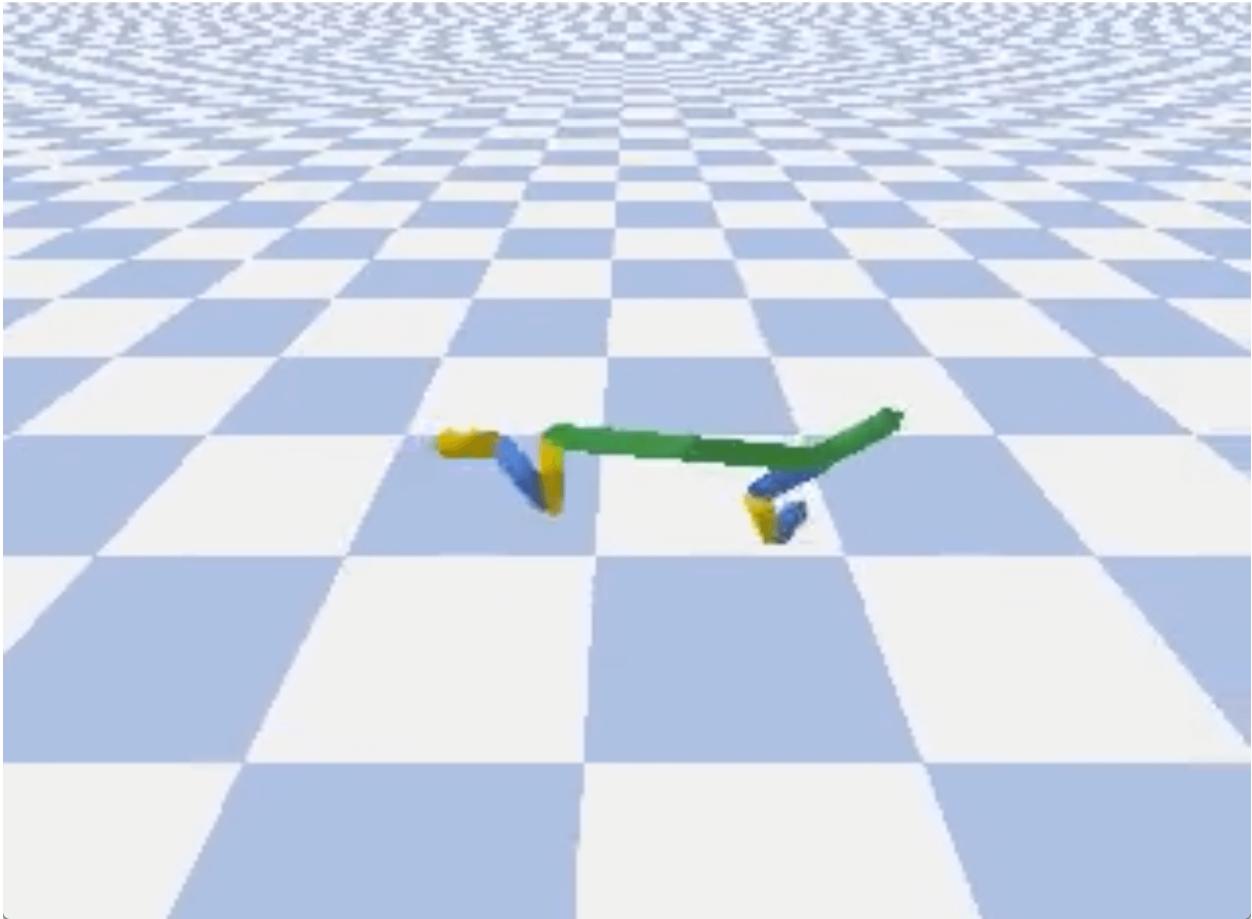
ARS se basa en una rama específica de Inteligencia Artificial llamada Estrategias Evolutivas. La diferencia es que en estrategias evolutivas se usan redes neuronales profundas paralelas de varias capas, mientras que ARS usa una política lineal simple, que es un Perceptron (una red neuronal superficial de una capa compuesta de varias neuronas). ARS también es ligeramente similar a **PPO - Optimización de política proximal**, en el sentido de que ARS tiene como objetivo optimizar una política (una función de los estados que devuelven las acciones para ejecutar) que realiza las mejores acciones que permiten la IA caminar. Sin embargo, la técnica es diferente. Si tiene curiosidad sobre **PPO**, puedes consultar el trabajo de investigación.

Ahora profundicemos en el ARS.

La idea es en realidad bastante simple. Tenemos una política que toma como entradas los estados del entorno y devuelve como salidas las acciones a ejecutar para caminar y correr por un campo. Ahora, antes de comenzar a explicar el algoritmo, describamos con más detalle las entradas, las salidas y la política.

Entradas

La entrada es un vector que codifica los estados del entorno. ¿Qué significa eso? Primero, expliquemos qué es exactamente un estado del entorno en este contexto. Un estado es la situación exacta que ocurre en un momento específico t , por ejemplo:



Podemos ver al guepardo en el aire, patas traseras hacia arriba, patas delanteras dobladas, a punto de aterrizar en el suelo. Todo esto está codificado en un vector. ¿Cómo? Simplemente reuniendo suficientes valores que puedan describir lo que está sucediendo aquí. Entonces, el vector codificado contendrá las coordenadas de los puntos angulares del guepardo, así como los ángulos de rotación alrededor de los rotores, y más valores como la velocidad. Por lo tanto, en cada momento t , un vector del mismo formato codifica lo que sucede exactamente en el entorno. Este vector codificado es lo que llamamos el estado de entrada del entorno, y será la entrada de nuestra política que intentaremos optimizar.

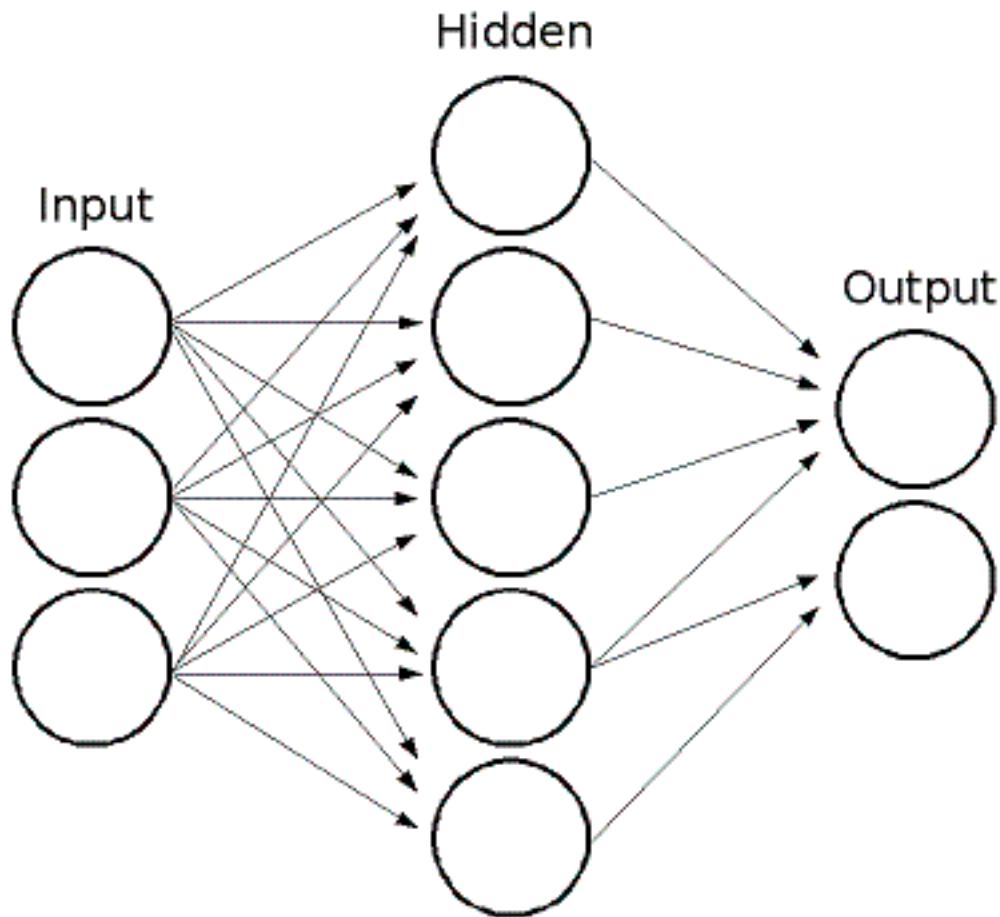
Salidas

El resultado, devuelto por nuestra política, es el grupo de acciones desempeñadas por el agente. Más precisamente, estas acciones son los diferentes impulsos musculares del agente. Por ejemplo, una de las acciones será la intensidad del músculo que empuja la pierna trasera al nivel del pie. Lo que es importante entender aquí es más el hecho de que la política está devolviendo un **grupo de acciones** en lugar de una sola acción. De hecho, una práctica común en el aprendizaje por refuerzo es devolver una acción discreta cada instante t . Aquí, no solo devolvemos un grupo de acciones, sino que cada una de estas acciones es continua. De hecho, para que un agente camine en un campo, tiene que mover todas las partes de su cuerpo en cada momento t , en lugar de solo una pierna, por ejemplo. Y las acciones son continuas porque los impulsos de los músculos se miden mediante métricas continuas. Por lo tanto, la salida también es un vector de varios valores continuos, al igual que el estado de entrada.

Política

Entre las entradas y las salidas tenemos una política, que no es más que una función, tomando como entradas

los estados de entrada y devolviendo como salidas las acciones a jugar, es decir, los impulsos musculares. Esta política será lineal, ya que de hecho será un perceptrón, que es una red neuronal simple de una capa y varias neuronas:



La capa oculta en el medio contiene las diferentes neuronas del perceptrón. A cada par de (valor de entrada, valor de salida) se le atribuye un peso. Por lo tanto, en total tenemos $\text{number_of_inputs} \times \text{number_of_outputs}$ pesos. Todos estos pesos están reunidos en una matriz, que no es más que la matriz de nuestra política lineal. En esta matriz, las filas corresponden a los valores de salida (las acciones) y las columnas corresponden a los valores de entrada (de los estados). Por lo tanto, esta matriz de pesos, llamada Θ , se compone de $n = \text{number_of_outputs}$ filas y $m = \text{number_of_inputs}$ columnas:

$$\Theta = \begin{pmatrix} (\text{input 1}, \text{output 1}) & (\text{input 2}, \text{output 1}) & \cdots & (\text{input } m, \text{output 1}) \\ (\text{input 1}, \text{output 2}) & (\text{input 2}, \text{output 2}) & \cdots & (\text{input } m, \text{output 2}) \\ \vdots & \vdots & \ddots & \vdots \\ (\text{input 1}, \text{output } n) & (\text{input 2}, \text{output } n) & \cdots & (\text{input } m, \text{output } n) \end{pmatrix} = \begin{pmatrix} \theta_{1,1} & \theta_{2,1} & \cdots & \theta_{m,1} \\ \theta_{1,2} & \theta_{2,2} & \cdots & \theta_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{1,n} & \theta_{2,n} & \cdots & \theta_{m,n} \end{pmatrix}$$

El algoritmo ARS

Inicialización

Al principio, todos los pesos $\theta_{i,j}$ de nuestra política lineal se inicializan a cero:

$$\forall i, j \in \{1, n\} \times \{1, m\}, \theta_{i,j} = 0$$

Aplicación de perturbaciones a los pesos.

Luego, aplicaremos algunas perturbaciones muy pequeñas a cada uno de estos pesos, agregando algunos valores muy pequeños $\delta_{i,j}$ a cada uno de los $\theta_{i,j}$ en nuestra matriz de pesos :

$$\begin{pmatrix} \theta_{1,1} & \theta_{2,1} & \cdots & \theta_{m,1} \\ \theta_{1,2} & \theta_{2,2} & \cdots & \theta_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{1,n} & \theta_{2,n} & \cdots & \theta_{m,n} \end{pmatrix} \longrightarrow \begin{pmatrix} \theta_{1,1} + \delta_{1,1} & \theta_{2,1} + \delta_{2,1} & \cdots & \theta_{m,1} + \delta_{m,1} \\ \theta_{1,2} + \delta_{1,2} & \theta_{2,2} + \delta_{2,2} & \cdots & \theta_{m,2} + \delta_{m,1} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{1,n} + \delta_{1,n} & \theta_{2,n} + \delta_{2,n} & \cdots & \theta_{m,n} + \delta_{m,n} \end{pmatrix}$$

Llamaremos a esto: “aplicar algunas perturbaciones en una **dirección positiva**” $+ \Delta_k$, donde Δ_k es la siguiente matriz de perturbaciones:

$$\Delta_k = \begin{pmatrix} \delta_{1,1} & \delta_{2,1} & \cdots & \delta_{m,1} \\ \delta_{1,2} & \delta_{2,2} & \cdots & \delta_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ \delta_{1,n} & \delta_{2,n} & \cdots & \delta_{m,n} \end{pmatrix}$$

“Positivo” viene del hecho de que estamos *agregando* los valores pequeños $\delta_{i,j}$ a nuestros pesos $\theta_{i,j}$. Estas pequeñas perturbaciones $\delta_{i,j}$ se muestran a partir de una distribución gaussiana $N(0, \sigma)$ (\$la desviación estándar \$\sigma\$ es lo que llamamos “ruido” en el modelo ARS .

Y cada vez que hacemos esto, también aplicamos exactamente las mismas perturbaciones $\delta_{i,j}$ a nuestros pesos $\theta_{i,j}$, pero en la dirección opuesta $- \Delta_k$, simplemente esta vez restando exactamente el mismo $\delta_{i,j}$:

$$\begin{pmatrix} \theta_{1,1} & \theta_{2,1} & \cdots & \theta_{m,1} \\ \theta_{1,2} & \theta_{2,2} & \cdots & \theta_{m,2} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{1,n} & \theta_{2,n} & \cdots & \theta_{m,n} \end{pmatrix} \longrightarrow \begin{pmatrix} \theta_{1,1} - \delta_{1,1} & \theta_{2,1} - \delta_{2,1} & \cdots & \theta_{m,1} - \delta_{m,1} \\ \theta_{1,2} - \delta_{1,2} & \theta_{2,2} - \delta_{2,2} & \cdots & \theta_{m,2} - \delta_{m,1} \\ \vdots & \vdots & \ddots & \vdots \\ \theta_{1,n} - \delta_{1,n} & \theta_{2,n} - \delta_{2,n} & \cdots & \theta_{m,n} - \delta_{m,n} \end{pmatrix}$$

Llamaremos a esto: “aplicación de algunas perturbaciones en la **dirección negativa**” $- \Delta_k$.

Por lo tanto, en conclusión, muestreamos una matriz específica de perturbaciones Δ_k con algunos valores $\delta_{i,j}$ cercanos a cero y actualizamos los pesos de nuestra matriz Θ en la dirección positiva $+ \Delta_k$ y la dirección negativa $- \Delta_k$:

Dirección Positiva: $\Theta \rightarrow \Theta + \Delta_k$

Dirección Negativa: $\Theta \rightarrow \Theta - \Delta_k$

Y de hecho, sobre cada episodio completo, aplicaremos estas perturbaciones positivas y negativas para muchas direcciones diferentes $\Delta_1, \Delta_2, \Delta_3$, etc. Haremos esto para 16 direcciones diferentes:

Direcciones Positivas: $\Theta \rightarrow \Theta + \Delta_1, \Theta \rightarrow \Theta + \Delta_2, \dots, \Theta \rightarrow \Theta + \Delta_{16}$

Direcciones Negativas: $\Theta \rightarrow \Theta - \Delta_1, \Theta \rightarrow \Theta - \Delta_2, \dots, \Theta \rightarrow \Theta - \Delta_{16}$

Ahora es el momento de preguntar: ¿por qué estamos haciendo esto?

La razón es realmente simple e intuitiva de entender. Queremos actualizar los pesos en estas direcciones diferentes para encontrar las que aumentarán más la recompensa total durante los episodios. Queremos averiguar qué actualizaciones de los pesos conducirán a las recompensas más altas. De hecho, aumentar la recompensa total acumulada durante el episodio es nuestro objetivo final, ya que cuanto mayor sea la recompensa, mejor será la capacidad del agente para caminar.

Ahora otra pregunta, menos obvia: ¿Por qué, para cada dirección, queremos tomar lo positivo y lo negativo?

Esto es porque, una vez que descubrimos las direcciones que aumentan más las recompensas (simplemente obteniendo la recompensa acumulada durante el episodio completo para cada dirección y luego clasificándolas por la más alta obtenida), haremos un paso de gradiente descendente para actualizar los pesos en estas mejores direcciones. Sin embargo, no tenemos ninguna función de recompensa de los pesos, por lo que no podremos aplicar el gradiente descendente directamente. De hecho, para aplicar el gradiente descendente necesitaríamos tener una función de recompensa de los pesos, $r(\Theta)$, y derivarla con respecto a los pesos:

$$\frac{\partial r(\Theta)}{\partial \Theta}$$

y luego hacer el paso de gradiente descendente para actualizar los pesos:

$$\Theta(\text{nuevo}) := \Theta(\text{antiguo}) + \frac{\partial r(\Theta)}{\partial \Theta} d\Theta$$

Pero no podemos hacer eso porque no tenemos una expresión explícita de la recompensa con respecto a los pesos. Entonces, en lugar de calcular directamente este gradiente, lo aproximaremos. Y ahí es donde entra en juego el combo de direcciones positivas y negativas, con el método de las diferencias finitas.

Gradiente Descendente aproximado con el método de diferencias finitas.

Entonces, ahora entendemos que tenemos que hacer un paso de gradiente descendente para actualizar los pesos en las direcciones que aumentan más la recompensa, y que para hacer este paso no tenemos más remedio que aproximar el gradiente de las recompensas con respecto a los pesos. Más específicamente, tenemos que aproximar:

$$\frac{\partial r(\Theta)}{\partial \Theta} d\Theta$$

Bueno, con lo que hemos hecho antes de aplicar las perturbaciones en las direcciones positiva y negativa, podremos aproximar esto fácilmente. Dado que el valor de cada perturbación δ es un número muy pequeño cercano a cero, entonces la diferencia entre la recompensa r_+ que obtenemos al aplicar la perturbación en la dirección positiva ($\Theta \rightarrow \Theta + \Delta$) y la recompensa r_- que obtenemos al aplicar la perturbación en la dirección negativa (u opuesta) ($\Theta \rightarrow \Theta - \Delta$) es aproximadamente igual a ese gradiente:

$$r_+ - r_- \approx \frac{\partial r(\Theta)}{\partial \Theta}$$

de modo que obtenemos la siguiente aproximación:

$$(r_+ - r_-)\Delta \approx \frac{\partial r(\Theta)}{\partial \Theta} d\Theta$$

Esta aproximación es el resultado del método de diferencias finitas y nos permite hacer este paso de gradiente descendente aproximado.

Luego elegimos una serie de mejores direcciones que queremos mantener como las que conducen a las recompensas más altas y hacemos este paso de Gradiente Descendente aproximado en todas estas mejores direcciones. ¿Cómo sabemos las principales direcciones que aumentan más las recompensas? Bueno, digamos que queremos mantener las 16 mejores direcciones, simplemente aplicamos las perturbaciones positivas y negativas para cada una de nuestras direcciones en un episodio completo, almacenamos el par de recompensas (r_+, r_-) que obtenemos para cada una de estas direcciones, y eventualmente mantenemos los 16 máximos más altos de r_+ y r_- . Estas 16 recompensas más altas corresponden a nuestras 16 mejores direcciones.

Luego, finalmente hacemos el promedio de nuestros gradientes aproximados sobre esas 16 mejores direcciones para actualizar toda la matriz de pesos Θ :

$$\Theta(\text{nuevo}) = \Theta(\text{antiguo}) + \frac{1}{16} \sum_{k=1}^{16} [r_+(k^{\text{esima}} \text{ mejor dirección}) - r_-(k^{\text{esima}} \text{ mejor dirección})] \Delta_{k^{\text{esima}} \text{ mejor dirección}}$$

Justo después de esta actualización, el paso de gradiente sescendente se aplica a toda la matriz de pesos Θ , de modo que los pesos de nuestra política se actualizan en las direcciones principales que aumentan más la recompensa acumulada.

Bucle de entrenamiento

Finalmente, repetimos todo este proceso (salvo el paso de la inicialización de los pesos a cero) para un cierto número de pasos (por ejemplo, 1000 pasos).

Podemos mejorar el rendimiento del ARS con los dos siguientes elementos de acción:

- Normalización de los estados
- Escalado de la desviación estándar de la recompensa
- Ajuste de la tasa de aprendizaje

Echemos un vistazo a cada una de estas soluciones:

Normalización de los estados

En el siguiente trabajo de investigación, tenemos las opciones entre $V1$ y $V2$ (ver la página 6). $V1$ es el algoritmo anterior sin normalizar los estados de entrada, y $V2$ es el ARS con estados de entrada normalizados.

La normalización de los estados mejora claramente el rendimiento.

Escalado de la desviación estándar de la recompensa

Podemos escalar dividiendo la suma anterior en la ecuación (1) por la desviación estándar σ_r de la recompensa, para obtener:

$$\Theta(\text{new}) = \Theta(\text{old}) + \frac{1}{16\sigma_r} \sum_{k=1}^{16} [r_+(k^{\text{th}} \text{ best direction}) - r_-(k^{\text{th}} \text{ best direction})] \Delta_{k^{\text{th}} \text{ best direction}}$$

Ajustar la tasa de aprendizaje

Para ajustar los parámetros del algoritmo, podemos agregar un factor de tasa de aprendizaje en la ecuación (2) (denotado por α en el documento):

$$\Theta(\text{new}) = \Theta(\text{old}) + \frac{\alpha}{16\sigma_r} \sum_{k=1}^{16} [r_+(k^{\text{th}} \text{ best direction}) - r_-(k^{\text{th}} \text{ best direction})] \Delta_{k^{\text{th}} \text{ best direction}}$$

4.3 Anexo 3: Preguntas y Respuestas

4.3.1 P&R de la Parte 1 - Optimización de Procesos

¿Qué son el Plan y la Política?

En pocas palabras, el plan es el proceso de crear el entorno de los estados de entrada, y la política es la función que toma los estados de entrada definidos por el plan como entradas y devuelve las acciones a ejecutar como salidas. Así, en el estudio de caso, todo el proceso que hacemos de definir el entorno de almacenamiento es nuestro plan y la política es nuestra IA. Echa un vistazo a los siguientes enlaces, ya que pueden ayudarte a obtener un contexto adicional:

- Link 1
- Link 2

¿Quién y cómo determina el factor de descuento en la ecuación de Bellman?

Está determinado por el desarrollador de IA a través de la experimentación. Primero se intenta con 1 (sin descuento), luego se disminuye un poco y se observa si obtienen mejores resultados. Y al repetir esto se encuentra un valor óptimo.

Aquí se agregarán más preguntas y sus respuestas, tan pronto como se hagan preguntas relevantes dentro del curso.

4.3.2 P&R de la Parte 2 - Minimización de Costes

En Deep Reinforcement Learning, ¿cuándo usar Argmax vs Softmax?

Utilizamos Argmax para problemas no demasiado complejos (como problemas de negocios) y Softmax para problemas complejos como la IA de un videojuego o hacer que un robot camine. De hecho, para los problemas complejos se suele necesitar hacer un poco de exploración vs explotación, y eso es exactamente lo que Softmax nos permite hacer. Sin embargo, los problemas empresariales no son demasiado complejos, por lo que no es necesario explorar mucho y, por lo tanto, un método Argmax suele ser suficiente.

¿Hay alguna razón específica para elegir dos capas con 64 y 32 neuronas para la arquitectura del cerebro? ¿Deberíamos prestar atención al sobreajuste?

Lo que debemos hacer es comenzar con algunas arquitecturas clásicas que encontramos en los documentos (ImageNet, ResNet, Inception, MobileNets, etc.). Luego intentamos, vemos si obtenemos buenos resultados, y si ese es el caso, podemos detenernos allí. Para nuestras redes neuronales profundas, simplemente tomamos una arquitectura clásica, con 2 capas completamente conectadas de 64 y 32 neuronas, que resulta funcionar muy bien para nuestro caso de estudio. Luego, hemos evitado el sobreajuste en el curso aplicando dos técnicas diferentes. Estas nos permiten mejorar el modelo y, por lo tanto, mejorar la puntuación. Estas dos técnicas son:

- Parada temprana (Early Stopping)

-
- Capa de olvido (Dropout)

Aquí se agregarán más preguntas y sus respuestas, tan pronto como se hagan preguntas relevantes dentro del curso.

4.3.3 P&R de la Parte 3 - Maximización de Beneficios

¿Podríamos ver con mayor detalle qué es una distribución? ¿Qué hay en el eje x y el eje y?

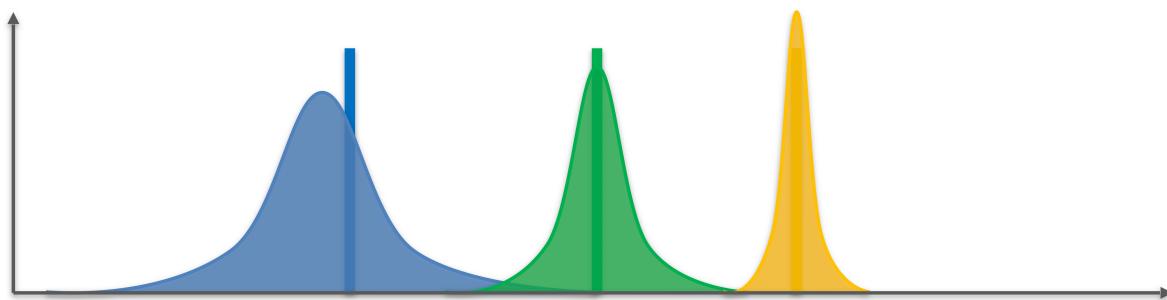
Supongamos que tenemos un experimento para seleccionar la estrategia que se mostrará a los clientes un total de 100 veces (es decir, con 100 clientes separados, uno tras otro), y calculamos la frecuencia de la selección de la estrategia. Luego lo repetimos nuevamente por otras 100 veces. Y nuevamente por otras 100 veces. Por lo tanto, obtenemos muchas frecuencias. Si repetimos tales cálculos de frecuencia muchas veces, por ejemplo 500 veces, podemos trazar un histograma de estas frecuencias. Según el Teorema central del límite, tendrá forma de campana y su media será la media de todas las frecuencias que obtuvimos durante el experimento. Por lo tanto, en conclusión, en el eje x tendremos los diferentes valores posibles de estas frecuencias, y en el eje y tendremos el número de veces que obtuvimos cada frecuencia durante el experimento.

En la primera clase de teoría del curso, ¿por qué D5 (en naranja) es la mejor distribución? ¿Por qué no es D3 (en rosa)?

En esta situación, 0 es perder y 1 es ganar. El D5 es el mejor porque está sesgado, por lo que tendremos resultados promedio cercanos a 1, lo que significa que allí tenemos más ganancias. Y en realidad todas las máquinas de casino hoy en día están cuidadosamente programadas para tener distribución como D1 o D3. Pero es un buen ejemplo concreto.

En el siguiente gráfico, ¿por qué la marca amarilla es la mejor opción y no la marca verde?

Case Study #3 - Maximizing the Revenues of an Online Retail Business



Al realizar el muestreo de Thompson, aún podemos realizar actualizaciones en nuestro algoritmo (como hacer nuevas conjeturas para las distribuciones con datos existentes, tomar muestras de la distribución adivinada, etc.) mientras esperamos los resultados de un experimento en el mundo real. Esto no impedirá que nuestro algoritmo funcione. Es por eso que puede aceptar feedback a posteriori.

¿Cuáles son otros ejemplos de aplicaciones de muestreo de Thompson?

El ejemplo más clásico es la optimización de la tasa de conversión. Tenemos varios anuncios de un mismo producto y deseamos saber cuál tiene el CTR más alto. Así que haríamos lo mismo que hicimos con las estrategias, excepto que esta vez los brazos serán los anuncios.

Otra aplicación potencial de bandidos con múltiples brazos (MAB) puede ser la prueba en línea de algoritmos. Por ejemplo, supongamos que está ejecutando un sitio web de comercio electrónico y tiene a su disposición varios algoritmos de Machine Learning para proporcionar recomendaciones a los usuarios (de lo que sea que el sitio web esté vendiendo), pero no sabemos qué algoritmo conduce a las mejores recomendaciones. Podríamos considerar este problema como un problema MAB y definir cada algoritmo de Machine Learning como un “brazo”: en cada ronda cuando un usuario solicita una recomendación, se seleccionará un brazo (es decir, uno de los algoritmos) para hacer las recomendaciones, y recibir una recompensa. En este caso, se puede definir la recompensa de varias maneras, un ejemplo simple es “1” si el usuario hace clic / compra un artículo y “0” en caso contrario. Finalmente, el algoritmo de bandido multi brazo convergerá y terminará eligiendo siempre el algoritmo más eficiente para proporcionar recomendaciones. Esta es una buena manera de encontrar el modelo más adecuado para resolver un problema en línea.

Otro ejemplo que se me viene a la mente es encontrar el mejor tratamiento clínico para los pacientes: cada tratamiento posible podría considerarse como un “brazo”, y una forma simple de definir la recompensa sería un número entre 0 (el tratamiento no tiene ningún efecto) y 1 (el paciente se cura perfectamente). En este caso, el objetivo es encontrar lo más rápido posible el mejor tratamiento y, al mismo tiempo, minimizar el arrepentimiento acumulativo (lo que equivale a decir que desea evitar lo más posible la selección de tratamientos “malos” o incluso subóptimos durante el proceso).

¿Dónde puedo encontrar algún recurso excelente en la distribución Beta?

El mejor que conozco es el siguiente

Tengo curiosidad por saber cómo se aplicaría Thompson Sampling de manera proactiva al ejecutar esta campaña de estrategia teórica. ¿Repetiríamos el programa en cada ronda?

Primero, un ingeniero de datos crearía un flujo completo para leer datos del sitio web y reaccionar a ellos en tiempo real. Luego, una visita al sitio web, activaría una respuesta para recalcular los parámetros y elegir una estrategia para la próxima vez.

Aquí se agregarán más preguntas y sus respuestas, tan pronto como se hagan preguntas relevantes dentro del curso.