

CASO PRÁCTICO 1 APRENDIZAJE POR REFUERZO EN PYTHON

Aquí se hará una máquina que **aprenda a jugar Ping Pong sola**. El plan es simular el ambiente del juego y el comportamiento que tendrá en la Jupyter Notebook.

El agente aquí será el **player 1 y las acciones posibles son 2**:

- La primera mover hacia arriba.
- La segunda mover hacia abajo.

Donde las reglas de juego son las siguientes:

REGLAS

- El agente dispone de 3 vidas.
- Si el agente pierde será castigado y se restaran 10 puntos.
- Por cada vez que se le dé a la bola se obtendrá una recompensa que sumara 10 puntos.
- Se limitará el juego para que no quede jugando por siempre.
- Se acepta el máximo 3000 iteraciones o alcanzar 1000 y habremos ganado.

SOLUCIÓN PASO A PASO

Ahora, se agregará la **import** que se usará:

```
import numpy as np

import matplotlib.pyplot as plt

from random import randint

from time import sleep

from IPython.display import clear_output

from math import ceil, floor

%matplotlib inline
```

La clase de agente: En la parte interna de la clase de agente se encontrará la tabla donde se irá recopilando las políticas. En este caso la tabla consta de 3 coordenadas. a) la posición actual del jugador. b) la posición (y) de la pelota. c) la posición en el eje (x) de la pelota. aquí se definirá el factor descuento el learning rate y la ratio de la exploración.

Sus métodos más importantes son:

get_next_step: Este es el que decide la siguiente acción a tomar en base a la ratio de exploración si se toma el mejor paso que estuviera almacenado o tomar un paso a zar, dando la posibilidad de que pueda estudiar el ambiente.

Update: A través de este se actualizan las políticas mediante la ecuación de Bellman. Y su implementación en python queda de la siguiente manera.

```
def __init__(self, game, policy=None, discount_factor = 0.1, learning_rate = 0.1,
ratio_explotacion = 0.9):

    # Creamos la tabla de politicas

    if policy is not None:

        self._q_table = policy

    else:

        position = list(game.positions_space.shape)

        position.append(len(game.action_space))

        self._q_table = np.zeros(position)

        self.discount_factor = discount_factor

        self.learning_rate = learning_rate

        self.ratio_explotacion = ratio_explotacion

    def get_next_step(self, state, game):

        # Damos un paso aleatorio...

        next_step = np.random.choice(list(game.action_space))

        # o tomaremos el mejor paso...

        if

            np.random.uniform() <= self.ratio_explotacion:

                # tomar el maximo

                idx_action = np.random.choice(np.flatnonzero(

                    self._q_table[state[0],state[1],state[2]]

                    self._q_table[state[0],state[1],state[2]].max()
```

==

```

))

next_step = list(game.action_space)[idx_action]

return next_step

# actualizamos las politicas con las recompensas obtenidas

def update(self, game, old_state, action_taken, reward_action_taken, new_state,
reached_end):

    idx_action_taken = list(game.action_space).index(action_taken)

    actual_q_value_options = self._q_table[old_state[0], old_state[1], old_state[2]]

    actual_q_value = actual_q_value_options[idx_action_taken]

    future_q_value_options = self._q_table[new_state[0], new_state[1], new_state[2]]

    future_max_q_value = reward_action_taken +

    self.discount_factor*future_q_value_options.max()

    if reached_end:

        future_max_q_value = reward_action_taken #maximum reward

    self._q_table[old_state[0], old_state[1], old_state[2], idx_action_taken] =

    actual_q_value + \

    self.learning_rate*(future_max_q_value -actual_q_value

def print_policy(self):

    for row in np.round(self._q_table,1):

        for column in row:

            print('I', end='')

        for value in column:

            print(str(

            value).zfill(5), end=' ')

        print('I ', end='')

    print("")

def get_policy(self):

```

return

self._q_table

Clase Environment: En esta clase se encuentra implementada la lógica y el control de juego del pong. Se puede controlar el rebote de la pelota y que esta no se salga de la pantalla y se hallan los métodos para graficar y animar en matplotlib. Por lo que se define una pantalla de 40 pixeles x 50 px de alto y se utiliza la variable movimiento_px=5 la tabla de políticas quedará definida en 8 de alto y de ancho ($40/5=8$ y $50/5=10$). Valores que puedes modificar si lo deseas.

Lo mejor de todo, es que se tiene el control de cuando dar las recompensas y las penalizaciones, esto sucede al perder cada vida y detectar si el juego ha terminado.

```
def __init__(self, max_life=3, height_px = 40, width_px = 50, movimiento_px = 5):
```

```
    self.action_space = ['Arriba', 'Abajo']
```

```
    self._step_penalization = 0
```

```
    self.state = [0,0,0]
```

```
    self.total_reward = 0
```

```
    self.dx = movimiento_px
```

```
    self.dy = movimiento_px
```

```
    filas = ceil(height_px/movimiento_px)
```

```
    columnas = ceil(width_px/movimiento_px)
```

```
    self.positions_space = np.array([[[[0 for z in range(columnas)]
```

```
        for y in range(filas)]
```

```
        for x in range(filas)])]
```

```
self.lives = max_life
```

```
self.max_life=max_life
```

```
self.x = randint(int(width_px/2), width_px)
```

```
self.y = randint(0, height_px-10)
```

```
self.player_alto = int(height_px/4)
```

```
self.player1 = self.player_alto # posic. inicial del player
```

```
self.score = 0
```

```
self.width_px = width_px
```

```
self.height_px = height_px
```

```
self.radio = 2.5
```

```
def reset(self):
```

```
self.total_reward = 0
```

```
self.state = [0,0,0]
```

```
self.lives = self.max_life
```

```
self.score = 0
```

```
self.x = randint(int(self.width_px/2), self.width_px)
```

```
self.y = randint(0, self.height_px-10)
```

```
return self.state
```

```
def step(self, action, animate=False):
```

```
    self._apply_action(action, animate)
```

```
    done = self.lives <= 0 # final
```

```
    reward = self.score
```

```
    reward += self._step_penalization
```

```
    self.total_reward += reward
```

```
    return self.state, reward, done
```

```
def _apply_action(self, action, animate=False):
```

```
    if action == "Arriba":
```

```
        self.player1 += abs(self.dy)
```

```
    elif action == "Abajo":
```

```
        self.player1 -= abs(self.dy)
```

```
    self.avanza_player()
```

```
    self.avanza_frame()
```

```
    if animate:
```

```
        clear_output(wait=True);
```

```
        fig = self.dibujar_frame()
```

```
        plt.show()
```

```
    self.state = (floor(self.player1/abs(self.dy))-2, floor(self.y/abs(self.dy))-2,  
                 floor(self.x/abs(self.dx))-2)
```

```
def detectaColision(self, ball_y, player_y):
```

```
if (player_y+self.player_alto >= (ball_y-self.radio)) and (player_y <=
(ball_y+self.radio)):
```

```
    return True
```

```
else:
```

```
    return False
```

```
def avanza_player(self):if self.player1 + self.player_alto >=
```

```
self.height_px:
```

```
self.player1 = self.height_px - self.player_alto
```

```
elif self.player1 <= -abs(self.dy):
```

```
self.player1 = -abs(self.dy)
```

```
def avanza_frame(self):
```

```
self.x += self.dx
```

```
self.y += self.dy
```

```
if self.x <= 3 or self.x > self.width_px:
```

```
self.dx = -self.dx
```

```
if self.x <= 3:
```

```
ret = self.detectaColision(self.y, self.player1)
```

```
if ret:
```

```
self.score = 10
```

```
else:
```

```
self.score = -10
```

```
self.lives -= 1
```

```
if self.lives>0:
```

```
self.x = randint(int(self.width_px/2), self.width_px)
```

```

self.y = randint(0, self.height_px-10)

self.dx = abs(self.dx)

self.dy = abs(self.dy)

else:

self.score = 0

if self.y < 0 or self.y > self.height_px:

self.dy = -self.dy

def dibujar_frame(self):

fig = plt.figure(figsize=(5, 4))

a1 = plt.gca()

circle = plt.Circle((self.x, self.y), self.radio, fc='slategray', ec="black")

a1.set_ylim(-5, self.height_px+5)

a1.set_xlim(-5, self.width_px+5)

rectangle = plt.Rectangle((-5, self.player1), 5, self.player_alto, fc='gold',
ec="none")

a1.add_patch(circle);

a1.add_patch(rectangle)

#a1.set_yticklabels([]);a1.set_xticklabels([]);

plt.text(4, self.height_px, "SCORE:"+str(self.total_reward)+" LIFE:"+str(self.lives),
fontsize=12)

if self.lives <=0:

plt.text(10, self.height_px-14, "GAME OVER", fontsize=16)

elif self.total_reward >= 1000:

```



```
plt.text(10, self.height_px-14, "YOU WIN!", fontsize=16)
```

```
return fig
```

Podemos simular el juego miles de veces para enseñar. Para eso, se define una función para jugar donde es indicada la cantidad de veces que se quiere iterar la simulación del juego y se irá almacenando algunas estadísticas sobre el comportamiento del agente, si hay mejoría del puntaje con las iteraciones y puntaje máximo alcanzado.

```
def play(rounds=5000, max_life=3, discount_factor = 0.1, learning_rate = 0.1,
        ratio_explotacion=0.9, learner=None, game=None, animate=False):

    if game is None:
        game = PongEnvironment(max_life=max_life, movimiento_px = 3)

    if learner is None:
        print("Begin new Train!")
        learner = PongAgent(game, discount_factor = discount_factor, learning_rate =
learning_rate, ratio_explotacion= ratio_explotacion)

    max_points= -9999
    first_max_reached = 0
    total_rw=0
    steps=&#91;]

    for played_games in range(0, rounds):
        state = game.reset()
        reward, done = None, None

        itera=0
        while (done != True) and (itera &lt; 3000 and game.total_reward&lt;=1000):
            old_state = np.array(state)
            next_action = learner.get_next_step(state, game)
            state, reward, done = game.step(next_action, animate=animate)
            if rounds > 1:
                learner.update(game, old_state, next_action, reward, state, done)
            itera+=1

        steps.append(itera)
```

```

total_rw+=game.total_reward
if game.total_reward > max_points:
    max_points=game.total_reward
    first_max_reached = played_games

if played_games %500==0 and played_games >1 and not animate:
    print("-- Partidas&#91;", played_games, "] Avg.Puntos&#91;",
int(total_rw/played_games),"] AVG Steps&#91;", int(np.array(steps).mean()), "] Max
Score&#91;", max_points,"]")

if played_games>1:
    print('Partidas&#91;',played_games,'] Avg.Puntos&#91;',int(total_rw/played_games),] Max
score&#91;', max_points,] en partida&#91;',first_max_reached,']')

#learner.print_policy()

return learner, game

```

Para poder **entrenar** se ejecutó la función con los siguientes parámetros.

- **Se podrán jugar 6000 partidas.**
- **El ratio de explotación:** Para el 85% de las veces será avaro, mientras que el 15% elige acciones aleatorias, dando lugar a la exploración.
- **Learning rate:** Aquí se suele dejar en el 10% como un valor razonable, proporcionando lugar a las recompensas y permitiendo actualizar la importancia de cada acción poco a poco. Después de más iteraciones, mayor importancia tendrá esa acción.
- **Discount_factor:** Se suele empezar con valor de 0.1 pero aquí no se utiliza un valor del 0.2 para intentar mostrar al algoritmo que nos interesa las recompensas de largo plazo.

learner, game = play(rounds=6000, discount_factor = 0.2, learning_rate = 0.1, ratio_explotacion=0.85)

En esta se ve la **salida del entreno después de 2 minutos:**

```

Begin new Train!
-- Partidas[ 500 ] Avg.Puntos[ -234 ] AVG Steps[ 116 ] Max Score[ 10 ]
-- Partidas[ 1000 ] Avg.Puntos[ -224 ] AVG Steps[ 133 ] Max Score[ 100 ]
-- Partidas[ 1500 ] Avg.Puntos[ -225 ] AVG Steps[ 134 ] Max Score[ 230 ]
-- Partidas[ 2000 ] Avg.Puntos[ -223 ] AVG Steps[ 138 ] Max Score[ 230 ]
-- Partidas[ 2500 ] Avg.Puntos[ -220 ] AVG Steps[ 143 ] Max Score[ 230 ]
-- Partidas[ 3000 ] Avg.Puntos[ -220 ] AVG Steps[ 145 ] Max Score[ 350 ]
-- Partidas[ 3500 ] Avg.Puntos[ -220 ] AVG Steps[ 144 ] Max Score[ 350 ]
-- Partidas[ 4000 ] Avg.Puntos[ -217 ] AVG Steps[ 150 ] Max Score[ 350 ]
-- Partidas[ 4500 ] Avg.Puntos[ -217 ] AVG Steps[ 151 ] Max Score[ 410 ]
-- Partidas[ 5000 ] Avg.Puntos[ -216 ] AVG Steps[ 153 ] Max Score[ 510 ]
-- Partidas[ 5500 ] Avg.Puntos[ -214 ] AVG Steps[ 156 ] Max Score[ 510 ]
Partidas[ 5999 ] Avg.Puntos[ -214 ] Max score[ 510 ] en partida[ 5050 ]

```

En las salidas se va viendo más que todo **como va mejorando en la cantidad de step que proporciona el agente** antes de perder la partida.

Podemos saber el resultado final, ya que contamos con un agente entrenado. Ahora, vamos a ver qué tal será su **comportamiento en una partida de pong**, y se puede ver jugando pasando el parámetro `animate=True`.

Antes de comenzar a jugar vamos a **instanciar un agente nuevo learner2**, el cual utilizará las políticas que fueron creadas anteriormente. A dicho agente le agregamos el valor de exploración en 1 para poder evitar que tome pasos aleatorios.

- `learner2 = PongAgent(game, policy=learner.get_policy())`
- `learner2.ratio_explotacion = 1.0` # con esto quitamos las elecciones aleatorias al jugar
- `player = play(rounds=1, learner=learner2, game=game, animate=True)`

En la **tabla de políticas resultantes** se mostrará un ejemplo de una tabla de 3 coordenadas. En la primera se tomaran valores de 0 a 7 (posición del jugador), en la segunda 0 valores (la altura de la bola de pong) y en tercer lugar va de 0 a 9 con el desplazamiento horizontal de la pelota.

Supongamos que el player está ubicado en la **posición "de debajo de todo" lo que es igual en la posición 0**, quedando la tabla conformada de la siguiente manera:

	x0		x1		x2		x3		x4		x5		x6		x7		x8		x9	
	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar	Subir	Bajar
y7	-2.4	-0.2	-0.3	-0.	-0.1	-0.	-0.	-0.	-0.	-0.	-0.1	-0.1	-0.3	-0.2	-39.6	-27.3	-16.1	-24.4	-21.2	-16.6
y6	-4.	-0.9	-0.6	-0.1	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.3	-9.7	-0.7	-15.7	-14.	-20.7	-45.9
y5	-5.4	-3.4	-0.4	-0.1	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.	-0.3	-0.	-0.7	-0.7	-17.4	-1.1	-53.2	-28.5
y4	-3.4	-1.4	-0.1	-0.4	-0.	0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.	-1.9	-0.6	-8.5	-19.5	-46.	-16.7
y3	-0.4	-3.4	-0.	-0.1	-0.	-0.	-0.	0.	-0.	-0.	-0.	-0.	-0.	-0.1	-23.3	-5.3	-12.4	-4.1	-2.4	-28.6
y2	0.4	-1.1	0.	-0.1	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.	-0.1	-0.4	-2.7	-14.4	1.9	-9.	3.8	-16.6
y1	-1.8	-2.2	-0.3	-0.1	-0.	-0.	-0.	0.	-0.	-0.	-0.	-0.1	-0.	-0.1	-17.6	-29.	-9.8	-4.1	-39.3	5.7
y0	-2.8	-0.7	-0.3	-0.1	-0.	0.	-0.	-0.	-0.	-0.	-0.1	-0.	-0.	-0.2	-5.7	-14.4	-16.5	0.1	-35.9	2.7

Aquí observamos la tabla con las acciones a tomar si el jugador está en la posición cero y según donde se encuentre la bola en los valores x e y. Recordando que, tenemos creadas 8 tablas como esta para cada posición del player.

Si vemos bien, en la **coordenada de la bola (x8, y1)** vemos los valores **1.9 para subir y -9 para bajar**. En ella se nota claramente que la recompensa mayor está en la acción de subir. Pero, si la pelotita estuviera en (x9,y4) la

mejor acción sería bajar, aunque tenga un puntaje negativo de -16,7 será mejor que restar 46.

Caso práctico 2 Aprendizaje por refuerzo en PYTHON

Para esta actividad los alumnos deberán recrear un código basado en el algoritmo de las redes neuronales autoconders, poder comprimir el tamaño de una imagen sin perder su calidad en pixel.

<https://youtu.be/aLcDJoGopec>

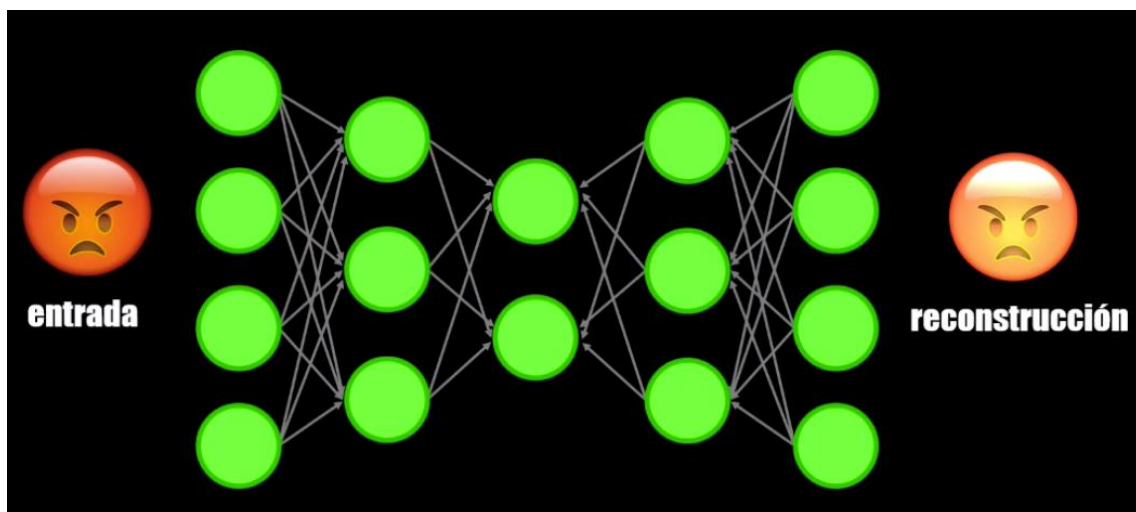
¿QUÉ ES UN AUTOENCODER?

UNA IDEA GENERAL

¿Qué pasa si en lugar de clasificar el dato de entrada intentamos reconstruirlo?

Es decir, ¿qué pasa si entrenamos la Red Neuronal para que aprenda a generar a la salida el mismo dato de entrada?

¡En este caso tendremos precisamente un Autoencoder!

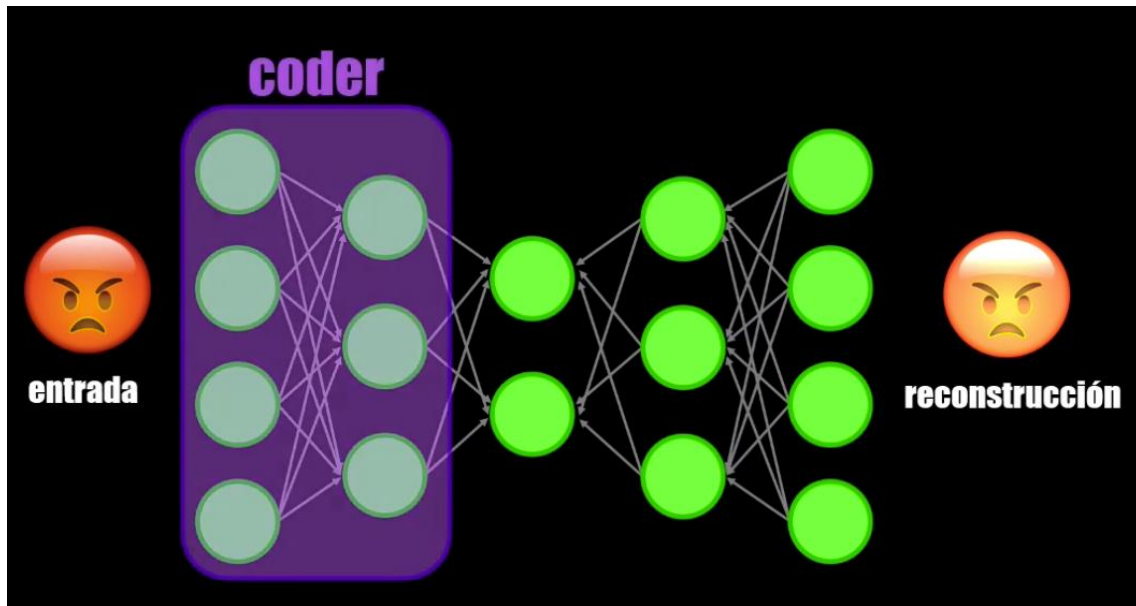


Si en lugar de entrenar la Red Neuronal para clasificar el dato, la entrenamos para que a la salida intente generar el mismo dato, tendremos precisamente un Autoencoder

ELEMENTOS DEL AUTOENCODER

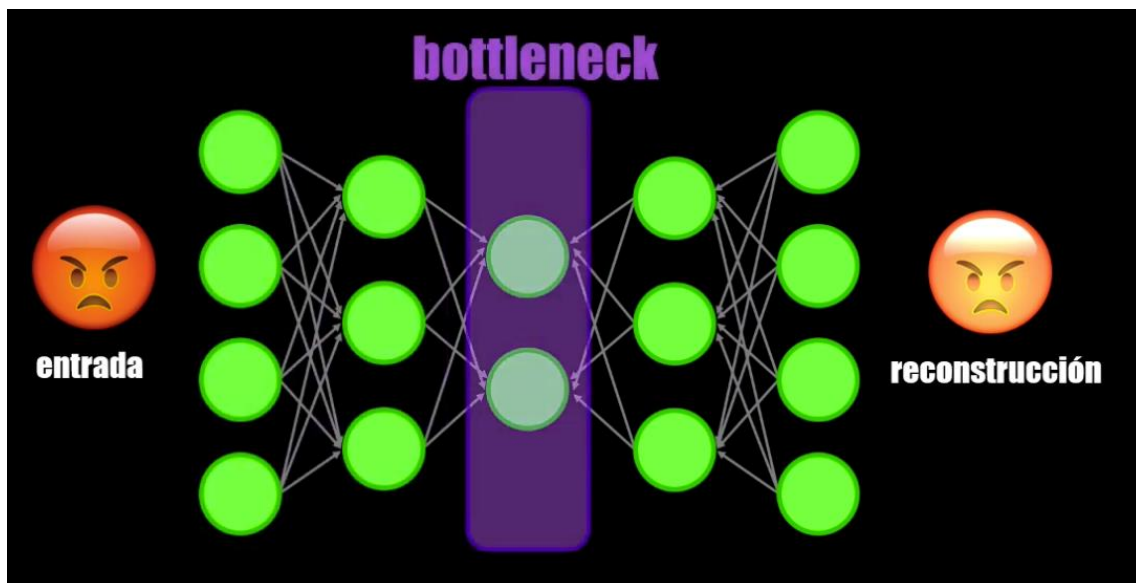
Un Autoencoder tiene tres elementos:

1. Un **encoder** que permite comprimir el dato de entrada:



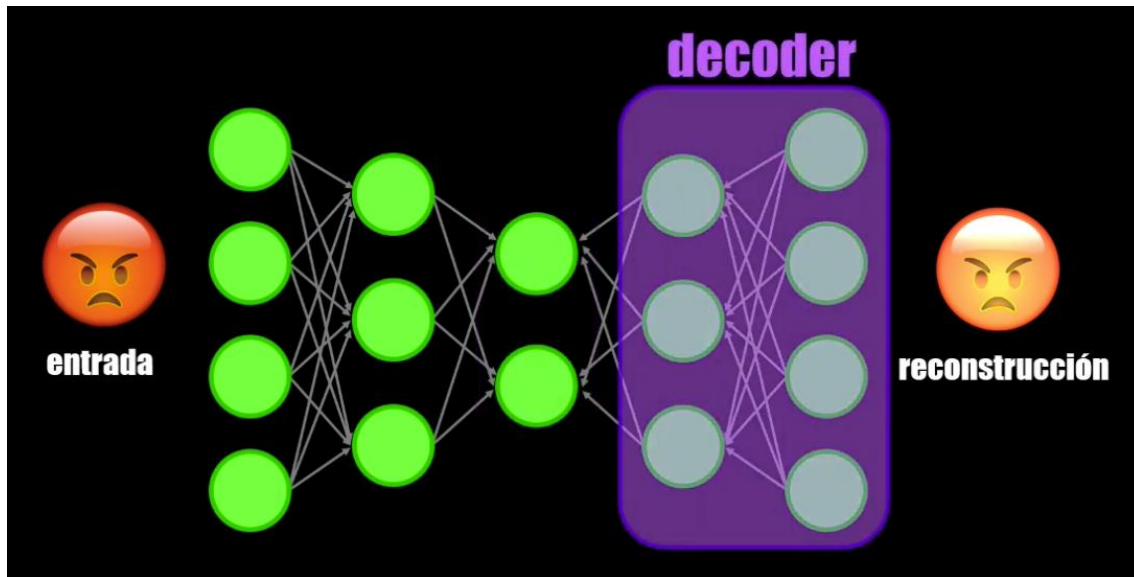
El encoder permite obtener una representación compacta del dato de entrada

2. El **bottleneck** (o cuello de botella, resultado de la compresión) que es como tal la representación compacta obtenida:



El bottleneck es la representación compacta, el dato comprimido, que se obtiene a la salida del encoder

3. Y el **decoder**, que reconstruye la entrada a partir del BOTTLENECK:



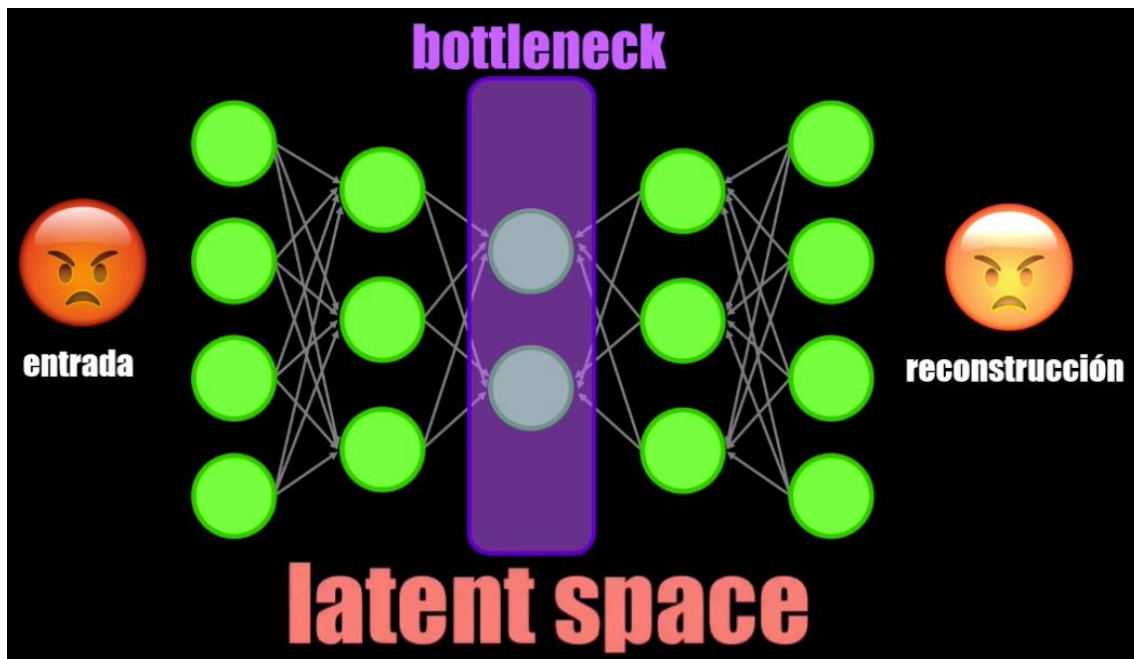
El decoder realiza el proceso inverso del encoder: toma el dato comprimido (con menores dimensiones) y lo lleva al espacio del dato original (con mayores dimensiones)

¿Pero para qué queremos entrenar una Red Neuronal que sea capaz de generar el mismo dato de entrada?

EL BOTTLENECK: UNA FORMA DE COMPRIMIR EL DATO

Si observamos el BOTTLENECK veremos que en este punto del Autoencoder se tiene una **representación compacta** de la entrada. Es decir, que el dato obtenido es una versión comprimida de la entrada, y que por tanto contiene una menor cantidad de datos.

La representación obtenida en el BOTTLENECK se conoce como **espacio latente** y es el resultado del entrenamiento, en donde la red aprende a extraer la información más relevante del dato de entrada:

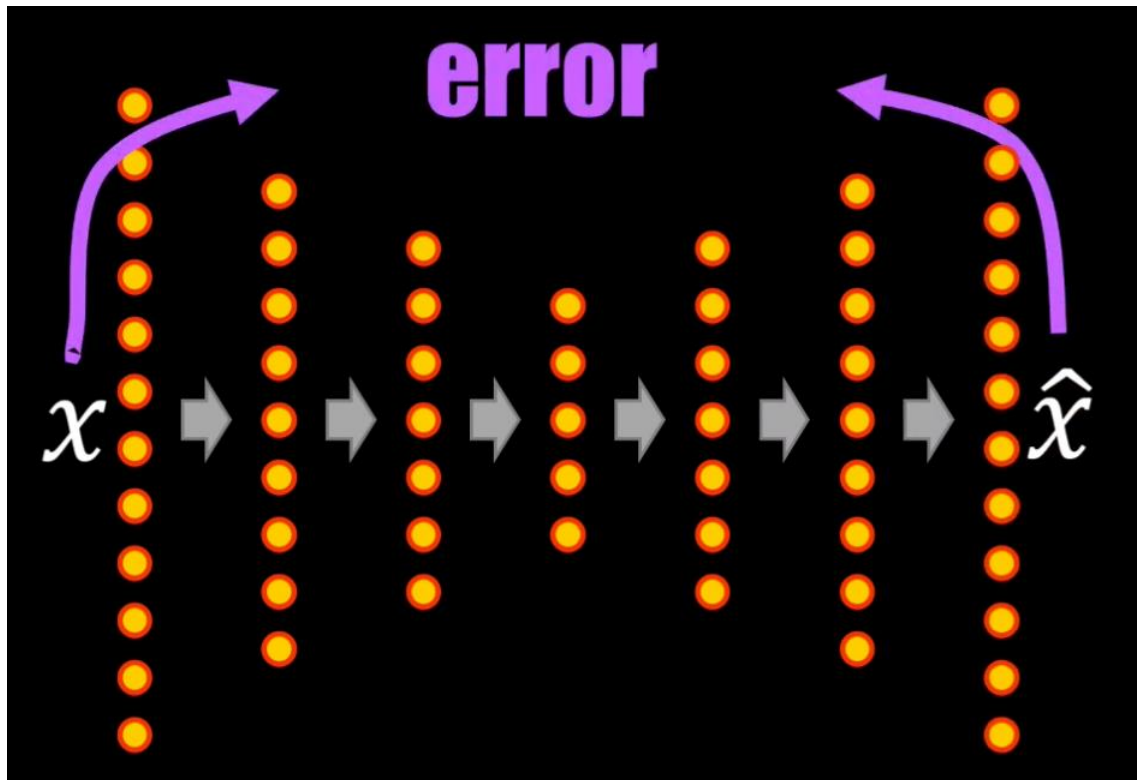


El espacio latente es simplemente la versión comprimida del dato de entrada. En el ejemplo de la figura la entrada tiene 4 datos (4 dimensiones) y el espacio latente tiene únicamente 2 dimensiones

ENTRENAMIENTO DEL AUTOENCODER

Para lograr encontrar esta representación compacta, el Autoencoder es entrenado de manera similar a una Red Neuronal.

Sin embargo, en este caso la función de error usada para actualizar los coeficientes del autoencoder es simplemente el resultado de comparar, punto a punto, el dato reconstruido con el dato original:

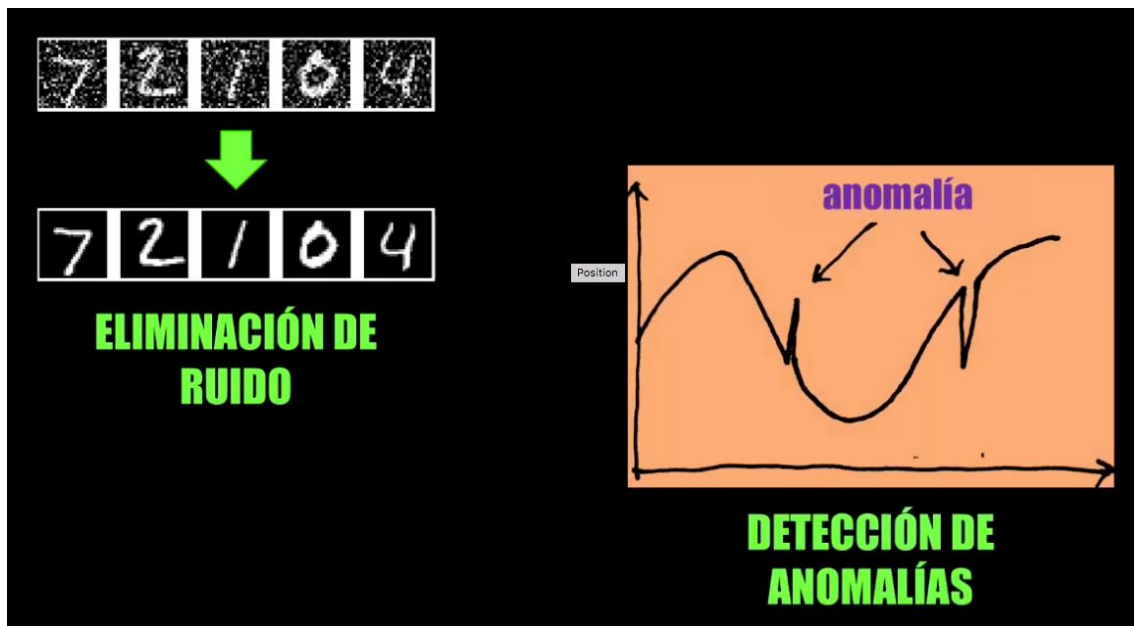


Durante el entrenamiento el error es calculado como la diferencia entre el dato original (izquierda) y el dato reconstruido (derecha)

Teniendo esto en cuenta podemos decir que el Autoencoder es un ejemplo de **aprendizaje no supervisado**, pues realmente durante el entrenamiento no definimos la categoría a la que pertenece cada entrada, ipues lo que queremos es que la salida sea precisamente el mismo dato de entrada!

ALGUNAS APLICACIONES DE LOS AUTOENCODERS

La representación compacta obtenida con el Autoencoder es útil por ejemplo para eliminar ruido en imágenes o para detectar anomalías en una serie de datos:



Algunas aplicaciones del Autoencoder: la eliminación de ruido en imágenes (izquierda) y la detección de anomalías (derecha)

Es precisamente en la detección de anomalías en donde se centra el tutorial que veremos a continuación.

DETECCIÓN DE FRAUDES USANDO AUTOENCODERS

EL PROBLEMA PARA RESOLVER

A nivel mundial, y durante el año 2015, el fraude por uso de tarjetas (débito o crédito) fue de 22 billones de dólares, y se espera que para el año 2020 este monto ascienda a 32 billones.

Varias empresas del sector han venido usando técnicas de Machine Learning, como las Máquinas de Soporte Vectorial y técnicas de clustering, para detectar estos fraudes, pero sin resultados exitosos.

Recientemente, empresas como PayPal, han comenzado a hacer uso de técnicas de "Deep Learning" y autoencoders para la detección de fraudes.

Para implementar el autoencoder de este ejemplo usaremos un set de datos que contiene registros de transacciones realizadas en Europa durante septiembre de 2013. La idea es determinar cuáles de estos registros corresponden a transacciones fraudulentas, para lo cual diseñaremos un Autoencoder.

Este tutorial está implementado en Python, y se divide en cinco partes: primero realizaremos la lectura y análisis exploratorio del set de datos. En segundo lugar, realizaremos el pre-procesamiento de los datos. Luego implementaremos el autoencoder y haremos su entrenamiento con ayuda de

la librería Keras, y finalmente lo usaremos para detectar posibles transacciones fraudulentas en nuestro set de datos.

Comencemos entonces con la lectura y pre-procesamiento de los datos.

1. LECTURA Y ANÁLISIS EXPLORATORIO DE LOS DATOS

El set de datos contiene en total 284.315 transacciones "normales" y tan solo 492 "fraudulentas". Para realizar la lectura usaremos la librería Pandas de Python:

```
import pandas as pd

datos = pd.read_csv("creditcard.csv")
print(datos.head())

nr_clases = datos['Class'].value_counts(sort=True)
```

La variable `nr_clases` contiene precisamente las transacciones normales y las fraudulentas.

Cada uno de estos datos contiene 30 características. Por motivos de confidencialidad, el proveedor del set de datos ha transformado las características V1 a V28 usando PCA (PRINCIPAL COMPONENT ANALYSIS, una técnica usada para reducir la redundancia de información). Las características 29 y 30 ("Time" y "Amount") indican la cantidad de segundos transcurridos desde el primer registro del dataset así como el monto (en euros) de la transacción:

	Time	V1	V2	V3	V4	V5	V6	V7	\
0	0.0	-1.359807	-0.072781	2.536347	1.378155	-0.338321	0.462388	0.239599	
1	0.0	1.191857	0.266151	0.166480	0.448154	0.060018	-0.082361	-0.078803	
2	1.0	-1.358354	-1.340163	1.773209	0.379780	-0.503198	1.800499	0.791461	
3	1.0	-0.966272	-0.185226	1.792993	-0.863291	-0.010309	1.247203	0.237609	
4	2.0	-1.158233	0.877737	1.548718	0.403034	-0.407193	0.095921	0.592941	
	V8	V9	...	V21	V22	V23	V24	\	
0	0.098698	0.363787	...	-0.018307	0.277838	-0.110474	0.066928		
1	0.085102	-0.255425	...	-0.225775	-0.638672	0.101288	-0.339846		
2	0.247676	-1.514654	...	0.247998	0.771679	0.909412	-0.689281		
3	0.377436	-1.387024	...	-0.108300	0.005274	-0.190321	-1.175575		
4	-0.270533	0.817739	...	-0.009431	0.798278	-0.137458	0.141267		
	V25	V26	V27	V28	Amount	Class			
0	0.128539	-0.189115	0.133558	-0.021053	149.62	0			
1	0.167170	0.125895	-0.008983	0.014724	2.69	0			
2	-0.327642	-0.139097	-0.055353	-0.059752	378.66	0			
3	0.647376	-0.221929	0.062723	0.061458	123.50	0			
4	-0.206010	0.502292	0.219422	0.215153	69.99	0			

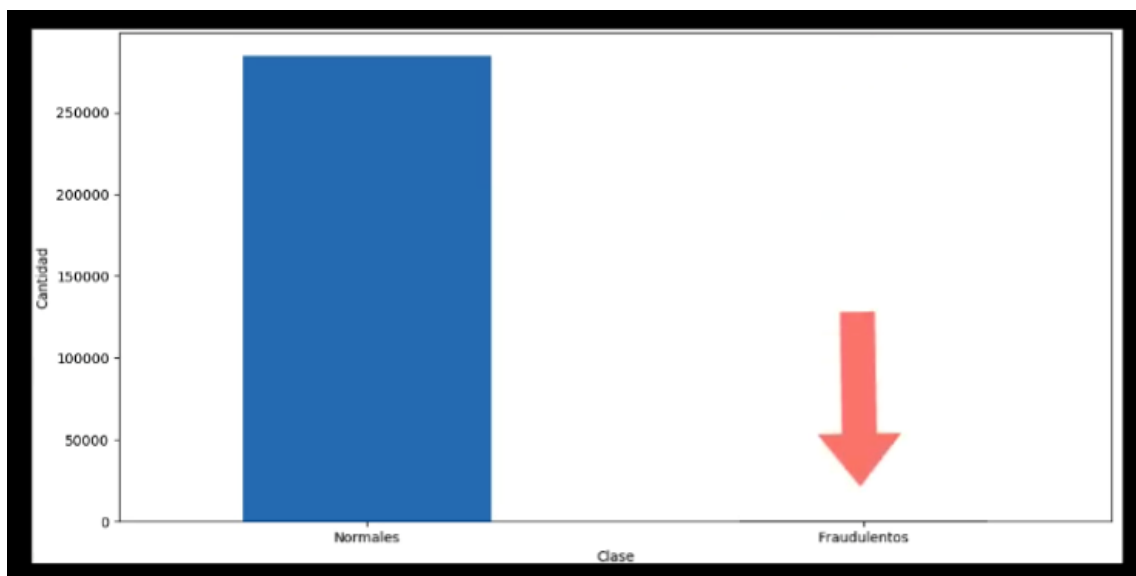
Algunos registros del set de datos a utilizar

Hagamos una exploración preliminar de los datos. Con ello podremos verificar que no hay una forma simple de detectar los fraudes, lo que justificará el uso de un Autoencoder.

En primer lugar, podemos obtener una gráfica que muestre la distribución de los registros "normales" y "fraudulentos":

```
nr_clases.plot(kind = 'bar', rot=0)
plt.xticks(range(2), ['Normales', 'Fraudulentos'])
plt.title("Distribución de los datos")
plt.xlabel("Clase")
plt.ylabel("Cantidad")
plt.show()
```

Podemos ver que los datos no están para nada balanceados, y la gran mayoría corresponde a registros "normales":

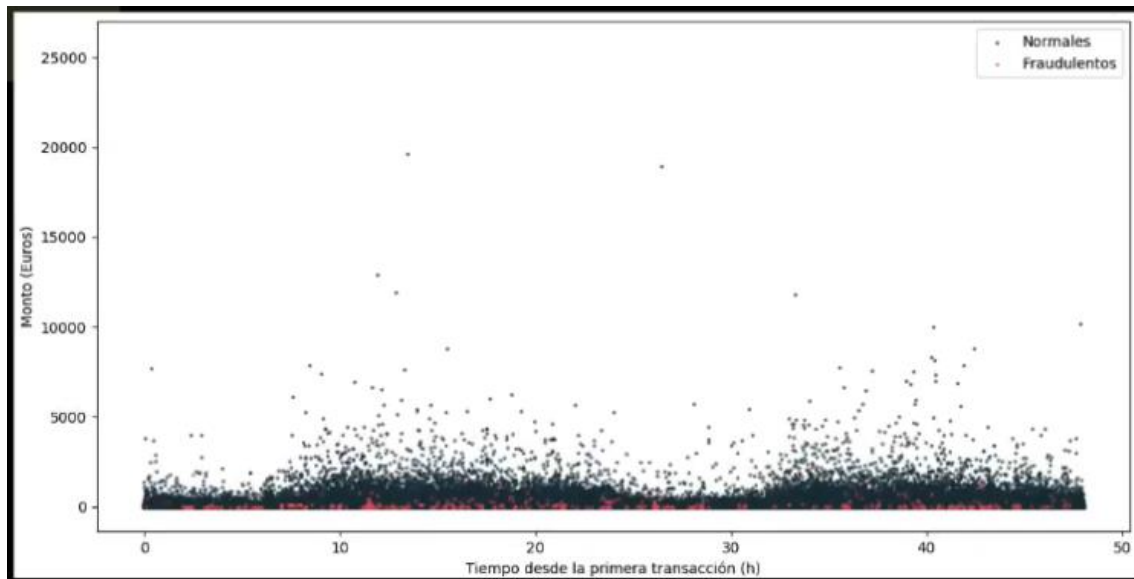


Distribución del set de datos: normales (a la izquierda) y fraudulentos (a la derecha)

Veamos ahora cómo se comportan los dos tipos de dato con respecto al tiempo:

```
normales = datos[datos.Class==0]
fraudulentos = datos[datos.Class==1]
plt.scatter(normales.Time/3600, normales.Amount,
            alpha = 0.5, c='#19323C', label='Normales', s=3)
plt.scatter(fraudulentos.Time/3600, fraudulentos.Amount,
            alpha = 0.5, c='#F2545B', label='Fraudulentos', s=3)
plt.xlabel('Tiempo desde la primera transacción (h)')
plt.ylabel('Monto (Euros)')
plt.legend(loc='upper right')
plt.show()
```

Observamos que no hay un patrón en particular, y que las transacciones "normales" y las "fraudulentas" no pueden ser discriminadas usando la variable tiempo:

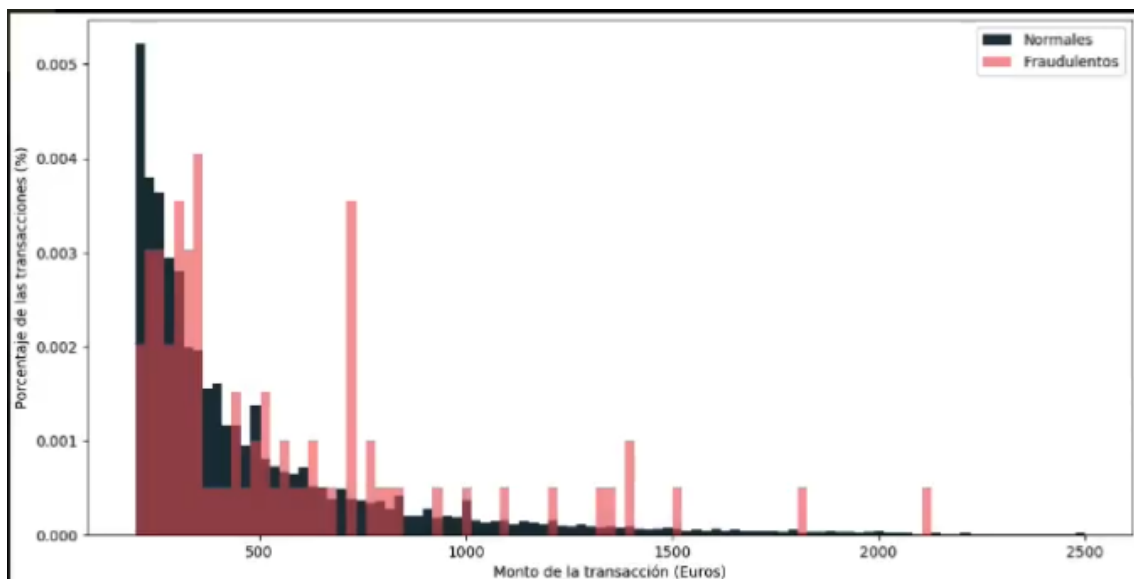


Distribución del set de datos con respecto al tiempo: normales (con tonalidad oscura) y fraudulentos (en rojo)

¿Es posible que el monto de cada transacción nos permita diferenciar un tipo de dato de otro?:

```
import numpy as np

bins = np.linspace(200, 2500, 100)
plt.hist(normales.Amount, bins, alpha=1, normed=True,
        label='Normales', color='#19323C')
plt.hist(fraudulentos.Amount, bins, alpha=0.6,
        normed=True, label='Fraudulentos', color='#F2545B')
plt.legend(loc='upper right')
plt.xlabel("Monto de la transacción (Euros)")
plt.ylabel("Porcentaje de las transacciones (%)")
plt.show()
```



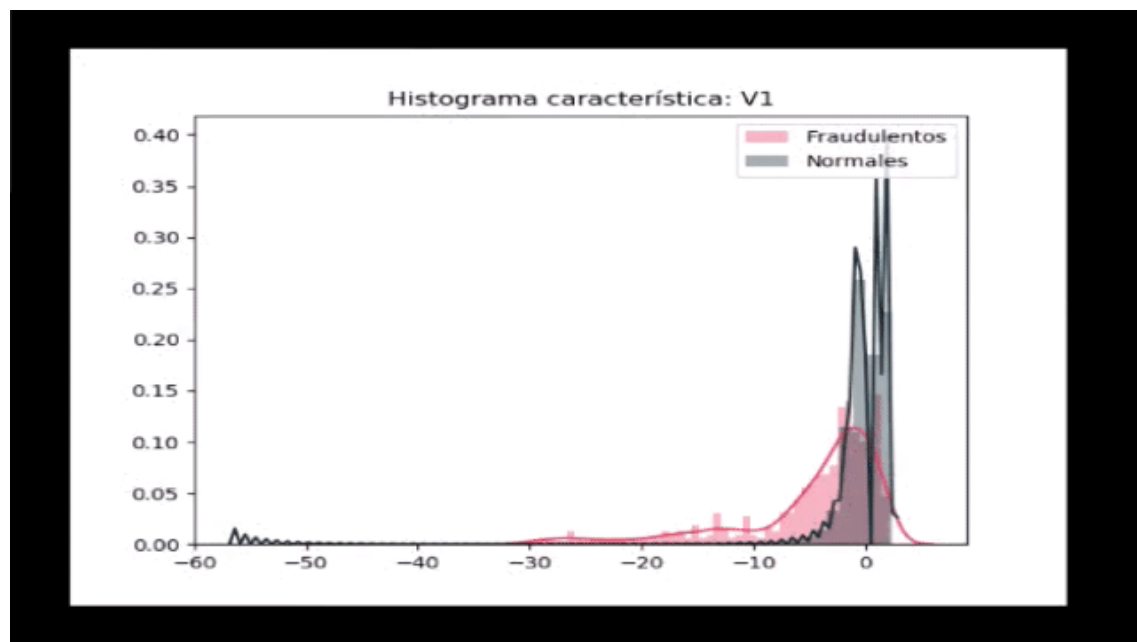
Monto de las transacciones para las categorías normal y fraudulento

Al observar la distribución de transacciones con respecto al monto, vemos que tampoco hay una clara diferencia, pues a pesar de que los registros "fraudulentos" tienden a tener montos más bajos, de todos modos estos se solapan con la distribución de los registros "normales".

Por último, veamos si las distribuciones de características V1 a V28 presentan alguna diferencia entre los dos tipos de registros:

```
import matplotlib.gridspec as gridspec
import seaborn as sns

v_1_28 = datos.iloc[:,1:29].columns
gs = gridspec.GridSpec(28, 1)
for i, cn in enumerate(datos[v_1_28]):
    sns.distplot(datos[cn][datos.Class == 1], bins=50,
                  label='Fraudulentos', color='#F2545B')
    sns.distplot(datos[cn][datos.Class == 0], bins=50,
                  label='Normales', color='#19323C')
    plt.xlabel("")
    plt.title('Histograma característica: ' + str(cn))
    plt.legend(loc='upper right')
    plt.show()
```



Distribución de las características V1 a V28

Observamos que ninguna de estas características permite una clara diferenciación de los datos, y con esto concluimos que un método simple (enfocado por ejemplo en definir un umbral que permita separar unas características de otras) no resulta útil.

Con este análisis preliminar podemos concluir que no existe una forma sencilla de separar una categoría de la otra, y que tampoco podemos entrenar una Red Neuronal para realizar esta clasificación pues el set de datos está desbalanceado: existen muchos más registros "normales" que "fraudulentos".

Lo anterior justifica el uso de un Autoencoder. Sin embargo, primero veamos cómo realizar el pre-procesamiento de los datos.

2. PRE-PROCESAMIENTO DE LOS DATOS

Comencemos eliminando la característica "tiempo" del set de datos (pues, como vimos anteriormente, no brinda información relevante):

```
from sklearn.preprocessing import StandardScaler
datos.drop(['Time'], axis=1, inplace=True)
```

Además, normalicemos la característica "amount" (el monto de las transacciones), para que tenga valor medio igual a cero y desviación estándar igual a 1 (comparables con las características V1 a V28). Esto garantizará la convergencia del algoritmo del Gradiente Descendente durante el entrenamiento:

```
datos['Amount'] = StandardScaler().fit_transform(datos['Amount'].values.reshape(-1,1))
```

Ahora creemos el set de entrenamiento (que corresponde al 80% de los registros) y el de validación (que tendrá el 20% restante). Esto se logra fácilmente con la función `train_test_split` de la librería [Scikit-Learn](https://scikit-learn.org/).

El autoencoder será entrenado únicamente con registros "normales", para que de esta forma aprenda a obtener una representación compacta de esos datos "normales". Una vez entrenado, y cuando se introduzca un registro "fraudulento", se esperaría que la reconstrucción del dato no sea tan precisa y por tanto la diferencia (o error) entre el dato reconstruido y el original será más grande que aquella obtenida cuando el dato ingresado es normal.

Así que el set de entrenamiento contendrá únicamente registros normales, mientras que el de validación tendrá los dos tipos de registros (normales y fraudulentos):

```
from sklearn.model_selection import train_test_split

X_train, X_test = train_test_split(datos, test_size=0.2, random_state=42)
X_train = X_train[X_train.Class == 0] ## Usaremos únicamente la clase 0 (transacciones normales)
X_train = X_train.drop(['Class'], axis=1)
X_train = X_train.values

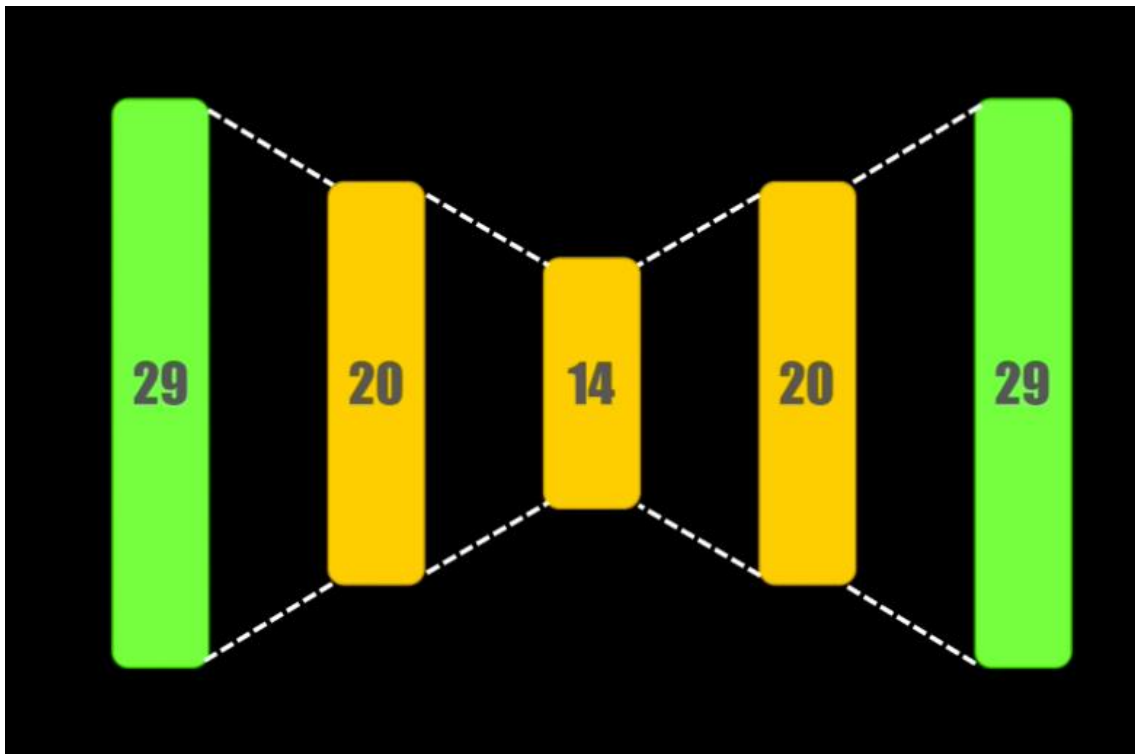
Y_test = X_test['Class']
X_test = X_test.drop(['Class'], axis=1)
X_test = X_test.values
```

Perfecto. Tenemos listos nuestros sets de entrenamiento y validación. Veamos ahora cómo implementar el Autoencoder en Keras.

3. CREACIÓN DEL AUTOENCODER EN KERAS

Diseñaremos un autoencoder con las siguientes características:

- 29 entradas, correspondientes al número de características de cada dato de entrenamiento,
- El encoder, encargado de obtener una representación compacta de la entrada, tendrá una primera capa con 20 neuronas, con [función de activación](#) `TANH` seguida de otra capa con 14 neuronas y activación `RELU`. Esto quiere decir que el dato comprimido tendrá un tamaño igual a 14.
- El decoder, encargado de reconstruir el dato original a partir de su versión comprimida, tendrá una capa con 20 neuronas y activación `TANH` seguida por la capa de salida con 29 neuronas y activación `RELU`, cuya salida corresponde a la versión reconstruida.



La estructura del Autoencoder a implementar

Para crear este Autoencoder primero definimos la capa de entrada, que tendrá el mismo número de elementos que cada ejemplo en el set de entrenamiento (es decir 29 datos):

```
import numpy as np
np.random.seed(5)
from keras.layers import Input
```

```
dim_entrada = X_train.shape[1]
capa_entrada = Input(shape=(dim_entrada,))
```

En donde en las líneas de código anteriores hemos usado `np.random.seed(5)` para fijar la semilla del generador aleatorio y garantizar de esta forma la reproducibilidad del algoritmo.

A continuación, creamos el encoder con dos capas, de 20 y 14 neuronas, y funciones de activación `tanh` y `relu` respectivamente:

```
from keras.layers import Dense

encoder = Dense(20, activation='tanh')(capa_entrada)
encoder = Dense(14, activation='relu')(encoder)
```

El encoder que acabamos de crear permite pasar de un dato de entrada con 29 características a una representación compacta con 14 características.

Ahora implementaremos el decoder, que realiza el proceso de reconstrucción del dato, pasando de una representación compacta con 14 características a la versión reconstruida con 29 características.

El decoder tendrá entonces dos capas: la primera con 20 y la segunda con 29 neuronas. Acá usamos nuevamente las funciones de activación `tanh` y `relu`:

```
decoder = Dense(20, activation='tanh')(encoder)
decoder = Dense(29, activation='relu')(decoder)
```

Finalmente, combinamos la capa de entrada, el encoder y el decoder en un modelo de Keras, que será precisamente el autoencoder. Esto lo hacemos con la función `Model`, donde basta únicamente con especificar la capa de entrada y el decoder, y Keras automáticamente se encarga de interconectar todos los objetos:

```
from keras.models import Model

autoencoder = Model(inputs=capa_entrada, outputs=decoder)
```

Ahora debemos compilar el modelo. Es decir que debemos definir el algoritmo para minimizar la función de error, que será el Gradiente Descendente (`SGD` en Keras). Este método usará una tasa de aprendizaje de 0.01.

También debemos definir la función de error, que en este caso será el error cuadrático medio, que permitirá comparar punto a punto el dato de salida reconstruido con el dato original. En keras esta función se especifica con el parámetro `loss='mse'`.

Finalmente, todos estos parámetros se definen a través de la función `compile`:

```
from keras.optimizers import SGD

sgd = SGD(lr=0.01)
autoencoder.compile(optimizer='sgd', loss='mse')
```

4. ENTRENAMIENTO DEL AUTOENCODER

Recordemos que el entrenamiento se hará únicamente con las transacciones normales (set `X_train`).



Entrenaremos el modelo usando un total de 100 iteraciones, y usando tamaños de lote de 32 (es decir, presentaremos al modelo bloques de 32 ejemplos durante el entrenamiento).

Es importante observar que para este entrenamiento definimos `X_train` como el dato tanto de entrada como de salida del modelo:

```
nits = 100
tam_lote = 32
autoencoder.fit(X_train, X_train, epochs=nits, batch_size=tam_lote, shuffle=True,
validation_data=(X_test,X_test), verbose=1)
```

5. VALIDACIÓN DEL MODELO Y DETECCIÓN DE FRAUDES

Como el modelo fue entrenado sólo con transacciones normales, es de esperar que al introducir un registro fraudulento la reconstrucción del dato no sea tan precisa y por tanto la diferencia entre el dato reconstruido y el original será más grande que la obtenida cuando el dato ingresado es normal:

	error de validación
normales	
fraudulentos	

Al reconstruir un dato fraudulento con el Autoencoder entrenado el error en la reconstrucción debería ser mayor que para un dato normal

Así que para realizar la detección de fraudes seguiremos este procedimiento:

1. En primer lugar tomaremos el Autoencoder entrenado y lo usaremos para generar una predicción, usando el set de validación (`X_test`)
2. Luego calcularemos el error cuadrático medio entre el dato original (a la entrada del Autoencoder) y el dato reconstruido (generado a través de la predicción)
3. Posteriormente estableceremos un umbral: si el error calculado en el punto anterior supera el umbral, tendremos un registro "fraudulento", y en caso contrario tendremos un registro "normal".

Esta idea del umbral debería funcionar, pues se espera que los registros "normales" tengan un error en la reconstrucción menor que los registros "fraudulentos".

Veamos en detalle cada uno de estos pasos:

1. **Predicción:** para generar la predicción usaremos la función `predict`, que permitirá obtener la reconstrucción de cada dato de entrada:

```
X_pred = autoencoder.predict(X_test)
```

Cada uno de los datos predichos por el modelo tiene un total de 29 características (el mismo tamaño del dato de entrada).

2. **Cálculo del error cuadrático medio**

```
ecm = np.mean(np.power(X_test-X_pred,2), axis=1)
```

3. **Definición del umbral:**

Para definir el umbral, usaremos dos métricas que resultan útiles en casos como este (donde los datos no están balanceados): "precisión" y "recall"

Para esto tengamos en cuenta que el modelo puede generar cuatro tipos de clasificación:

- La situación ideal: un fraude es clasificado como un fraude. Esto se conoce como un "true positive" (ó TP, un positivo verdadero)
- Un fraude es clasificado como un registro normal. Esto se conoce como un "false negative" (FN ó un falso negativo)
- Un registro normal es clasificado como normal. Esto se conoce como un "true negative" (TN ó un verdadero negativo)
- Finalmente, un registro normal puede ser clasificado como fraudulento. Esto se conoce como "false positive" (FP ó un falso positivo).

Un clasificador ideal no debería generar ni falsos positivos ni falsos negativos. Sin embargo, ningún clasificador es ideal y por tanto siempre tendremos, en mayor o menor grado, falsos positivos y negativos.

El parámetro "precision" es una medida de la cantidad de falsos positivos generados por el clasificador. Si la cantidad de falsos positivos se reduce a cero, se alcanzaría una "precisión" máxima igual a 1.

Por su parte, el "recall" es una medida de la cantidad de falsos negativos generados por el clasificador. Si la cantidad de falsos negativos (FN) se reduce a cero, se alcanzaría un "recall" ideal igual a 1.

PRECISION

$$TP$$
$$\frac{TP}{TP + FP}$$

RECALL

$$TP$$
$$\frac{TP}{TP + FN}$$

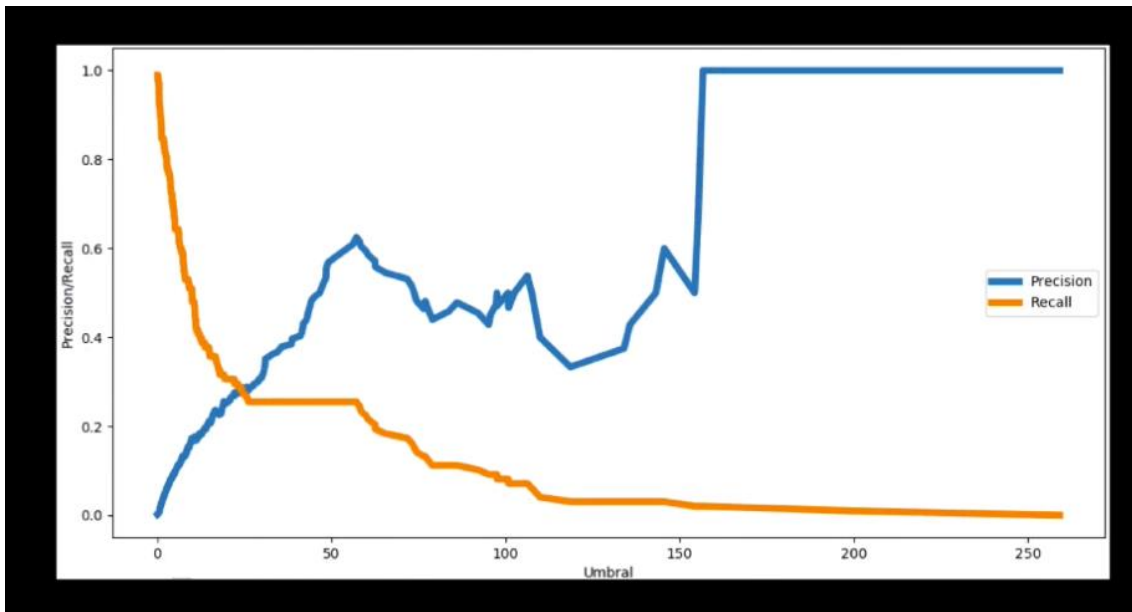
Definición del 'precision' y el 'recall'

En la práctica no se pueden lograr simultáneamente los dos valores ideales: una mayor "precisión" implica sacrificar el "recall" y viceversa.

En nuestro Autoencoder, el precisión y el recall cambiarán dependiendo del umbral escogido. Para observar este comportamiento podemos hacer una gráfica del precisión y el recall vs el umbral, que en Scikit-Learn se puede obtener fácilmente usando la función `precision_recall_curve`:

```
from sklearn.metrics import precision_recall_curve
precision, recall, umbral = precision_recall_curve(Y_test, ecm)

plt.plot(umbral, precision[1:], label="Precision",linewidth=5)
plt.plot(umbral, recall[1:], label="Recall",linewidth=5)
plt.title('Precision y Recall para diferentes umbrales')
plt.xlabel('Umbral')
plt.ylabel('Precision/Recall')
plt.legend()
plt.show()
```



A medida que el umbral aumenta, la precisión aumenta pero el recall disminuye

Como se mencionó anteriormente, en efecto el precisión y el recall tienen comportamientos opuestos: un aumento del umbral genera un aumento de la precisión pero una reducción en el recall.

Lo que queda ahora es definir el umbral, y esto depende de qué es lo que queremos favorecer. ¿Nos interesa detectar la mayor parte de los fraudes, es decir tener pocos falsos negativos, a costa de un alto número de falsos positivos? O por el contrario, ¿nos interesa que los registros normales sean clasificados correctamente, a costa de un elevado número de registros fraudulentos clasificados incorrectamente?

Para este ejemplo vamos a escoger la primera opción, es decir que ajustaremos el umbral para obtener un recall alto.

Fijaremos entonces un umbral de 0.75. Si el error de la predicción sobrepasa este umbral, asignaremos la categoría "0" (registro fraudulento), mientras que si el error es menor o igual al umbral asignaremos la categoría "1" (registro normal):

```
umbral_fijo = 0.75
Y_pred = [1 if e > umbral_fijo else 0 for e in ecm]
```

Finalmente, podemos imprimir la matriz de confusión de este clasificador tomando como referencia la categoría a la que corresponde realmente el dato (almacenada en la variable `Y_test`):

```
from sklearn.metrics import confusion_matrix

conf_matrix = confusion_matrix(Y_test, Y_pred)
print(conf_matrix)
```

Tengamos en cuenta que la matriz de confusión muestra en su diagonal principal el número de verdaderos negativos (normales clasificados como normales) y verdaderos positivos (fraudulentos clasificados como

fraudulentos), mientras que en la diagonal secundaria encontraremos los falsos positivos (normales clasificados como fraudulentos) y los falsos negativos (fraudulentos clasificados como normales):

categoría real	predicción	
	normales	fraudulentos
	normales	fraudulentos
normales	TN	FP
fraudulentos	FN	TP

Definición de la matriz de confusión para el caso de las transacciones normales y fraudulentas

Al ejecutar el código anterior, obtenemos la siguiente matriz de confusión:

categoría real	predicción	
	normales	fraudulentos
	normales	fraudulentos
normales	49585	7279
fraudulentos	8	90

La matriz de confusión obtenida para el autoencoder diseñado y con un umbral de 0.75

Con los datos de la matriz de confusión podemos obtener el precisión y recall logrados por el autoencoder con el umbral escogido.

Al momento de definir el umbral decidimos favorecer el recall, sacrificando el "precisión. De hecho, al observar la matriz de confusión vemos que 49580 datos normales fueron clasificados correctamente, mientras que 7284 fueron clasificados como fraudulentos. Es decir, tenemos una cantidad relativamente alta de falsos positivos, y por tanto un "precision" de tan solo 0.01!

Sin embargo, al analizar el recall, vemos que un total de 90 registros fraudulentos fueron detectados correctamente, mientras que 8 de ellos fueron detectados como registros normales. ¡No está nada mal!, pues tenemos un recall de 0.92.

Con esto podemos concluir que para el Autoencoder diseñado, y con el umbral seleccionado, se tiene un buen desempeño al momento de detectar registros fraudulentos.