Construcción de Carro Autónomo mediante algoritmo DQN con Pytorch y Kivy.

Colorado, Cesar
Universidad Nacional de Ingeniería
Lima, Perú
ccoloradoc@uni.pe

Inocente, Patrichs
Universidad Nacional de Ingeniería
Lima, Perú
pinocentev@uni.pe

Guevara, Junior
Universidad Nacional de Ingeniería
Lima, Perú
jguevarat@uni.pe

Sulca, Leonardo
Universidad Nacional de Ingeniería
Lima, Perú
leonardo.sulca.d@uni.pe

***Resumen*—Deep Q-network (DQN) es un algoritmo propio del aprendizaje por refuerzo profundo, que tiene como herramienta principal una red neuronal usada para la estimación del valor Q (funcion Valor Acción) de las acciones disponibles para un estado en particular. En este trabajo se explora el uso de DQN para resolver la problemática de un carro autónomo que debe viajar desde el centro de una ciudad hacia un aeropuerto y viceversa; por lo que en cada momento se necesita obtener una ruta óptima para su recorrido, esta experiencia es simulada usando la plataforma Kivy y la libreria pytorch de python; de tal forma que Kivy actúe como visualizador y pytorch nos proporcione las herramientas para la implementación de nuestra DQN, dando como resultado varios agentes que logran viajar desde el origen hacia la meta evitando choques entre sí y sorteando obstáculos a medida que estos son instanciados en el mapa de forma interactiva.**

Índice de Términos— DQN, Kivy, pytorch, Carro aútonomo .

I. Introducción

Kivy es principalmente un framework de Python gratuito y de código abierto para desarrollar aplicaciones multitáctiles con una interfaz de usuario natural. El objetivo principal de usar Kivy en este proyecto es obtener un entorno de visualización y de interaccion dinámica para la puesta de obstaculos en el mapa. El mapa se compone de 3 zonas principales establecidas para el recorrido del carro autónomo (veáse Figura 1).

1. **Centro de la Ciudad**:Zona considerada como punto de origen o meta, dependiendo del viaje actual que debe realizar nuestro carro autónomo.
2. **Aeropuerto**: Zona considerada también como origen o meta.
3. **Ciudad**: Describe la zona intermedia del mapa donde encontraremos: - **Obstaculos:** Lineas dibujadas en tiempo de ejecución que no deberían ser transitados por el carro autónomo. - **Avenidas:** Zona libre de obstaculos.

II. Estado del Arte

Q-learning fue introducido por *Watkins* en 1989. Watkins presentó una prueba de convergencia a Dayan en 1992.

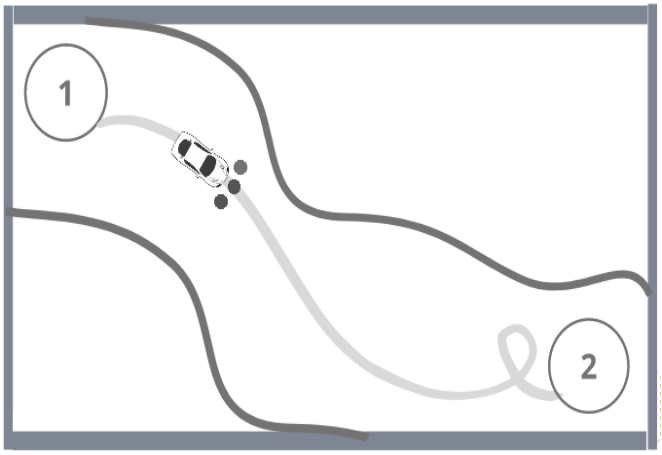

Figura 1. Representación del Entorno (Creación propia). Se muestra las partes del Mapa. 1. Centro de Ciudad. 2. Aeropuerto. La linea de color más suave es una ruta representativa que tomaría nuestro carro autónomo dado los obstaculos marcados con lineas de color más fuerte.

Una prueba matemática más detallada fue expuesta por **Tsitsiklis** en 1994, y por **Bertsekas** y **Tsitsiklis** en su libro de 1996 **Neuro-Dynamic Programming**.

Watkins lo señalaba ya el título de la tesis de su doctorado "Learning from delayed rewards" (Aprendiendo de las recompensas tardías). Ocho años antes en 1981 el mismo problema, bajo el nombre de "aprendizaje por refuerzo tardío", que fue solucionado por **Bozinovski's Crossbar Adaptive Array** (CAA).

La matriz de memoria W(a,s) es exactamente igual que la Q-table de Q-learning de ocho años más tarde. La arquitectura introdujo el término "evaluación de estado" en aprendizaje por refuerzo. El algoritmo de aprendizaje crossbar (escrito en pseudocódigo matemático en papel) realiza los siguientes cálculos en cada iteración:

- En el estado s efectúa la acción a
- En consecuencia recibe el estado s'

- Realiza la valuación del estado v(s')
- Actualiza el valor crossbar W'(a,s) = W(a,s) + v(s') − v(s')

El término "refuerzo secundario" viene de la teoría de aprendizaje animal, utilizada para modelar los valores de estado a través de propagación hacia atrás: el valor de estado v(s') de la situación consecuente es propagado hacia atrás hacia las situaciones encontradas previamente. CAA calcula los valores de estados verticalmente y las acciones horizontalmente. Grafos de demostración nos muestran los estados contenidos en el aprendizaje por refuerzo tardío (deseables, indeseables, y estados neutros), que han sido calculados por la función de evaluación de estados. Este sistema de aprendizaje fue el precursor del algoritmo Q-learning.

En 2014 Google DeepMind patentó una aplicación de Q-learning de cara al aprendizaje profundo, llamado .ªprendizaje de refuerzo profundo.º "Q-learning profundo"que puede jugar a los juegos Atari 2600 en niveles humanos expertos

III. Variantes:

Q-learning profundo:
El sistema DeepMind utilizó redes neuronales convolucionales profundas, con capas de filtros convolucionados enladrillados para asemejar los efectos de los campos receptivos. El aprendizaje por refuerzo es inestable o divergente cuando un aproximador de funciones no lineales se utiliza para representar Q. Esta inestabilidad proviene de la correlación presente en las secuencias de observación, el hecho de que actualizaciones pequeñas de Q puedan cambiar significativamente la política y la distribución de los datos, y las correlaciones entre Q y los valores objetivo.

La técnica utilizó repetición de experiencia, un mecanismo biológicamente inspirado que usa una muestra aleatoria de acciones previas en vez de la acción más reciente para continuar.Esto elimina las correlaciones en la secuencia de observación y suaviza los cambios en la distribución de los datos. La actualización iterativa ajusta Q de cara a los valores objetivo que solo se actualizan periódicamente, reduciendo más adelante las correlaciones con el objetivo.

Q-learning doble:
Puesto que el valor futuro máximo aproximado de una acción en Q-learning se evalúa utilizando la misma función Q que en la política de acción actualmente seleccionada, en entornos ruidosos Q-learning a veces puede sobrestimar los valores de la acción, retrasando el aprendizaje. Se propuso una variante llamada Q-learning doble para corregir esto. Q-learning doble es un algoritmo de aprendizaje por refuerzo sin política, donde una política diferente se utiliza para la evaluación del valor que se usa para seleccionar próxima acción.
En la práctica, dos funciones de valor diferentes se entrenan en Q^A una moda simétrica mutua usando experiencias separadas, Q^A y Q^B. Q^A Q^B El paso de actualización de Q-learning doble es, por tanto, así:

$$Q^A_{t+1}(s_t, a_t) = Q^A_t(s_t, a_t) + \alpha_t(s_t, a_t)\left(r_t + \gamma Q^B_t\left(s_{t+1}, \arg_a \max Q^A_t(s_{t+a}, a) - Q^A_t(s_t, a_t)\right)\right)$$

, y Ahora el valor estimado del futuro descontado se evalúa utilizando una política diferente, lo cual soluciona el problema de la sobrestimación.
Este algoritmo se combinó más tarde con aprendizaje profundo, como el algoritmo DQN, resultando en doble DQN, el cual supera en rendimiento el algoritmo DQN original.

IV. Articulos:

- Hester, T., Vecerik, M., Pietquin, O., Lanctot, M., Schaul, T., Piot, B., ... Gruslys, A. (2018, April). Deep q-learning from demonstrations. In Thirty-second AAAI conference on artificial intelligence.
- Van Hasselt, H., Guez, A., Silver, D. (2016, March). Deep reinforcement learning with double q-learning. In Proceedings of the AAAI conference on artificial intelligence (Vol. 30, No. 1).
- Ong, H. Y., Chavez, K., Hong, A. (2015). Distributed deep Q-learning. arXiv preprint arXiv:1508.04186.

V. Diseño de Experimento

Viendo los casos en que algunos equipos de nuestro grupo hacían los movimientos de cada carrito en diferente FPS, decidimos establecer un FPS estable para que detecte el obstáculo y aprenda correctamente.

```
Config.set('input', 'mouse', 'mouse, multitouch_on_demand')
Config.set('graphics', 'maxfps', '20')
```

Aquí haremos la inicialización del Modelo con los parámetros "input_size", "nb_action", "gamma"

```
class Dqn(object):

    def __init__(self, input_size, nb_action, gamma):
        self.start = time.time()
        self.gamma = gamma
        self.model = Network(input_size, nb_action)
        self.memory = ReplayMemory(capacity=100000)
        self.optimizer = optim.Adam(params=self.model.parameters())
        self.last_state = torch.Tensor(input_size).unsqueeze(
            0)
        self.last_action = 0
```

```
        self.last_reward = 0
        self.reward_window = []
```

Aquí definimos el uso de una función para la toma de acciones, donde tendremos en cuenta la probabilidad de cada estado actual

```
def select_action(self, state):
        probs = F.softmax(self.model(Variable(state)
    )*100)
        action = probs.multinomial(len(probs))
        return action.data[0, 0]
```

Aquí definimos el uso de una función el aprendizaje:

```
def learn(self, batch_states, batch_actions,
    batch_rewards, batch_next_states):
        # Conseguimos el output del Modelo.
        # Gather hace la eleccion, con unsqueeze
    agregamos una dimension para seguir
        # trabajando con la eleccion, y despues con
    squeeze le bajamos a la dimension inicial
        # para obtener las salidas del Modelo
        batch_outputs = self.model(batch_states).
    gather(
            1, batch_actions.unsqueeze(1)).squeeze
    (1)

        # Recibe una tupla de dimension 3 y devuelve
     el maximo valor
        # de cada una de las tuplas evaluadas, y se
    almacena en batch_next_outputs
        # print(batch_next_outputs)
        batch_next_outputs = self.model(
    batch_next_states).detach().max(1)[0]

        # Q(s,a) = R(s,a) + gamma*max(Q(s',a))
        # Q(varios) para el lote examinado
        batch_targets = batch_rewards + self.gamma *
     batch_next_outputs

        # Calculamos la perdida y tratamos de
    minimizar la perdida con el Optimizador
        # Adam
        td_loss = F.smooth_l1_loss(batch_outputs,
    batch_targets)
        self.optimizer.zero_grad()  # -> Poner los
    gradientes a Cero
        td_loss.backward()  # -> Backpropagation,
    obtenemos las gradientes
        self.optimizer.step()  # -> actualizando los
     pesos con Adam
```

Aquí definimos el uso de una función para la actualización de un estado:

```
def update(self, new_state, new_reward):
        # Convierte el estado a un tensor float, y
    le aumenta 1 dimension
        new_state = torch.Tensor(new_state).float().
    unsqueeze(0)

        # Agregando a la memoria un evento, el
    evento es:
        # El estado actual, la accion, la recompensa
    , y el estado siguiente
        # last_state es el estado actual
        # last_reward -> la recompensa inmediata
        # new_state -> el estado siguiente
        self.memory.push((self.last_state, torch.
    LongTensor(
            [int(self.last_action)]), torch.Tensor([
    self.last_reward]), new_state))

        # Le pasamos un estado, y con select_action
    elije la accion
        # a tomar, de las probabilidades con la
    multinomial
        # tomando una accion [0,1,2]
        new_action = self.select_action(new_state)

        # Si la memoria de eventos se llena
    significa que
        # Empezamos a aprender tomando 100 muestras
        # Y lo mandamos a aprender, siempre y cuando
     tengamos
        # + de 100 muestras

        if len(self.memory.memory) > 100:
            batch_states, batch_actions,
    batch_rewards, batch_next_states = self.memory.
    sample(
                100)
            self.learn(batch_states, batch_actions,
                       batch_rewards,
    batch_next_states)
        self.last_state = new_state
        self.last_action = new_action
        self.last_reward = new_reward
        self.reward_window.append(new_reward)
        if len(self.reward_window) > 1000:
            del self.reward_window[0]
        return new_action
```

De nuestro archivo mapa.py

Función de actualización:

```
    def update(self, dt):
        global brain
        global reward
        global conteo
        global last_distance
        global goal_x
        global goal_y
        global longueur
        global largeur

        longueur = self.width
        largeur = self.height
        if first_update:
            init()

        update_sand = True
        if update_sand:
            sand = np.zeros((int(self.width), int(
    self.height)))
            update_sand = False

        # Diferencia de x-coordenadas entre la meta
    y el carrito.
        xx = goal_x - self.car.x
        # Diferencia de y-coordenadas entre la meta
    y el carrito.
        yy = goal_y - self.car.y
        orientation = Vector(*self.car.velocity).
    angle((xx, yy))/180.
```

```
        state = [orientation, self.car.signal1,
                 self.car.signal2, self.car.signal3]
        action = brain.update(state, reward)

        rotation = action2rotation[action]

        self.car.move(rotation)

        distance = np.sqrt((self.car.x - goal_x)**2
    + (self.car.y - goal_y)**2)

        # Actualizamos las posiciones de los
    sensores en el mapa.
        self.ball1.pos = self.car.sensor1
        self.ball2.pos = self.car.sensor2
        self.ball3.pos = self.car.sensor3

        self.scores.append(brain.score())

#************** ZONA DE RECOMPENSAS *************

        # Cuando el carrito 1 se encuentra en
    obstaculo disminuye su velocidad a 1.
        if sand[int(self.car.x), int(self.car.y)] >
    0 or self.car.collide_widget(self.car2) or self.
    car.collide_widget(self.car3) or self.car.
    collide_widget(self.car4):
            self.car.velocity = Vector(1, 0).rotate(
    self.car.angle)
            reward = -1
        # Caso contrario el carrito1 mantiene una
    velocidad de 6.
        else:
            self.car.velocity = Vector(6, 0).rotate(
    self.car.angle)
            reward = -0.2
            if distance < last_distance:
                reward = 0.1

        # Actualizamos lectura de sensores del carro
     1 para que detecte cuando choca con otro carro.
        if self.ball1.collide_widget(self.car2) or
    self.ball1.collide_widget(self.car3) or self.
    ball1.collide_widget(self.car4):
            self.car.signal1 = 1.0


        if self.ball2.collide_widget(self.car2) or
    self.ball2.collide_widget(self.car3) or self.
    ball2.collide_widget(self.car4):
             self.car.signal2 = 1.0

        if self.ball3.collide_widget(self.car2) or
    self.ball3.collide_widget(self.car3) or self.
    ball3.collide_widget(self.car4):
             self.car.signal3 = 1.0

    # Si el carrito logra salirse de los bordes del
    mapa, se asigna una penalidad de -1.
        if self.car.x < 10:
            self.car.x = 10
            reward = -1
        if self.car.x > self.width - 10:
            self.car.x = self.width - 10
            reward = -1
        if self.car.y < 10:
            self.car.y = 10
            reward = -1
        if self.car.y > self.height - 10:
            self.car.y = self.height - 10
            reward = -1

        if distance < 100:
            goal_x = self.width-goal_x
            goal_y = self.height-goal_y

        last_distance = distance
```

class Network(nn.Module):

Construcción de Red Neuronal Predictora de Q-values

Nuestra DQN tiene como eje principal una red neuronal que recibe un estado dado por la Orientación y las lecturas de los sensores y obtiene como salida los Q-values de las acciones disponibles para ese estado (Veáse figura 2), a continuación se muestra la implementación de nuestra red.

```
class Network(nn.Module):

    def __init__(self, input_size, nb_action):
        super(Network, self).__init__()
        self.input_size = input_size
        self.nb_action = nb_action
        # Combinacion Lineal de toda la data
        self.fc1 = nn.Linear(input_size, 30)
        self.fc2 = nn.Linear(30, nb_action)

    def forward(self, state):
        x = F.tanh(self.fc1(state))
        q_values = self.fc2(x)
        return q_values
```

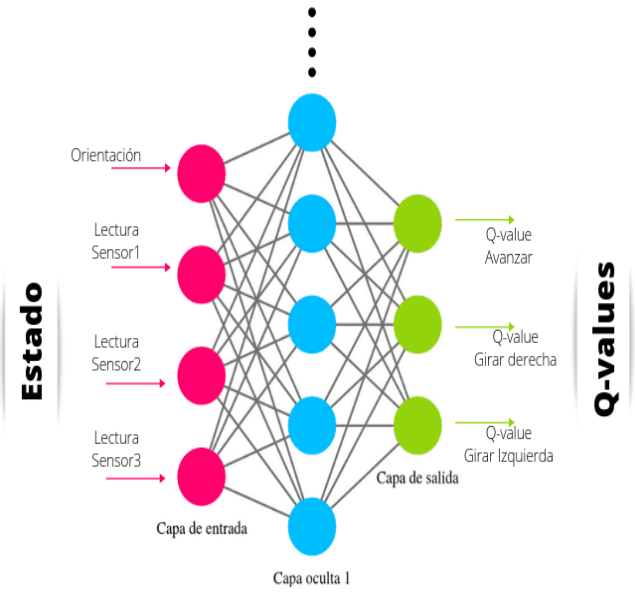

Figura 2. Representación de la Red Neuronal predictora de Q-values (Creación propia). Se muestra como un estado (Orientación, Sensor1, Sensor2, Sentor3) es ingresado a la red y se obtiene los Q-values de cada opción permitida.

VI. Experimentos y resultados

Inicialmente solo contamos con un carro, y diseñamos un mapa que se puede ver en figura 3, para el entrenamiento del carro. Y en la figura 4 se puede observar las recompensas a la lo largo del entrenamiento.

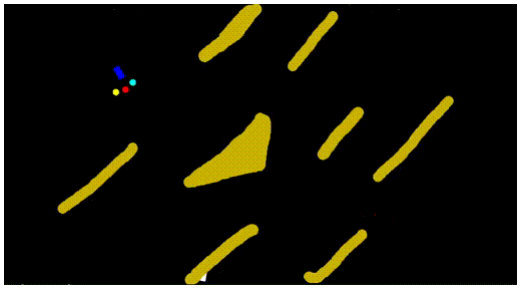

Figura 3. Mapa usado para el entrenamiento

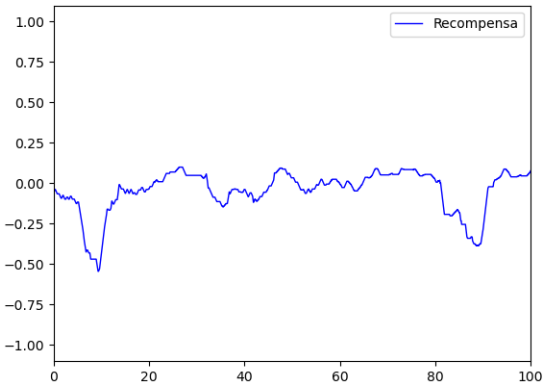

Figura 4. Recompensa a lo largo del entrenamiento

En nuestra implementación hemos agregado 3 carros más en el mismo mapa de entrenamiento como se puede ver en la figura 5, de la misma manera se midió las recompensas a lo largo del entrenamiento como se muestra en la figura 6.

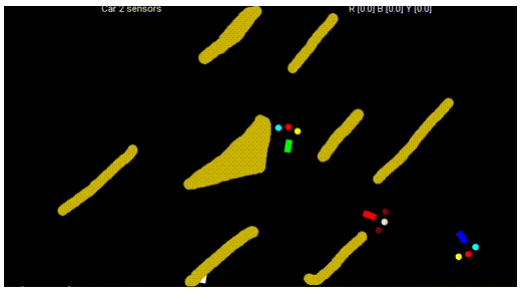

Figura 5. Mapa usado para el entrenamiento

VII. Conclusiones

- Deep Q-Learning no es perfecto para la solución de este problema, más tiene un increíble acercamiento al objetivo de un "Vehículo Autónomo"
- Las funciones de activación influyen fuertemente a la eficiencia del vehículo, mostrando mejores resultados con una función tanh que con una función RElu durante las pruebas.

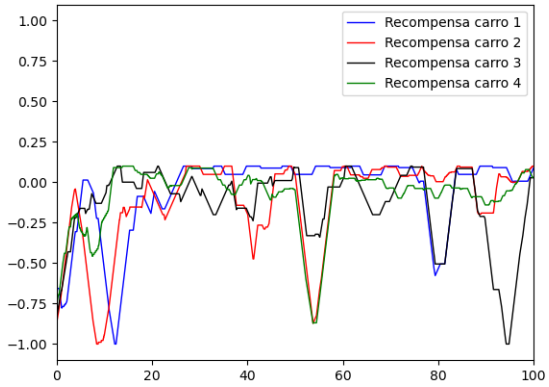

Figura 6. Recompensa a lo largo del entrenamiento

Referencias

[1] Deep Q-Learning — An Introduction To Deep Reinforcement Learning. (2021). Retrieved 23 July 2021, from https://www.analyticsvidhya.com/blog/2019/04/introduction-deep-q-learning-python/

[2] Reinforcement Learning (DQN) Tutorial — PyTorch Tutorials 1.9.0+cu102 documentation. (2021). Retrieved 23 July 2021, from *https://pytorch.org/tutorials/intermediate/reinforcement*$_{q}$*learning.html*

[3] Welcome to Kivy — Kivy 2.0.0 documentation. (2021). Retrieved 25 July 2021, from https://kivy.org/doc/stable/