TP3: Deep-Q Learning - Zelda

Miguel Alfaro (97743), Jonathan Moguilevsky (95516)

Facultad de Ingeniería, Universidad de Buenos Aires, Argentina.
75.68 Sistemas de soporte para celdas de producción flexible.

Abstract. Implementación de Deep Q-Learning utilizando redes neuronales para el entrenamiento de un agente utilizando como entorno el videojuego Zelda.

Keywords: Deep Q Learning, Red neuronal, Zelda

1 Introducción

En el presente trabajo se introduce la técnica conocida como Deep Q-Learning con el objetivo práctico de entrenar un Agente para que aprenda a jugar a una versión simplificada del videojuego Zelda, junto a los frameworks y diferentes entrenamientos realizados.

2 Q-Learning

Q-Learning es un algoritmo de aprendizaje por refuerzo utilizado para aprender las recompensas que tienen las diferentes acciones que un agente puede tomar bajo ciertas circunstancias. Para esto no requiere un modelo, por lo que se dice que es un algoritmo 'model-free'.

Para saber que tanto le conviene al agente tomar una acción en un estado determinado se utiliza la función $q(s, a)$.

$$Q : S \times A \rightarrow R \tag{1}$$

Donde:
S: es el conjunto de todos los posibles estados donde puede estar el agente
A: es el conjunto de acciones que puede tomar el agente

Cuando no se conoce nada del entorno, Q se inicializa con valores arbitrarios, luego por cada paso de tiempo t se toma una acción a_t, se observa la recompensa r_t y el próximo estado s_{t+1}

Este algoritmo se basa fuertemente en la Ecuación de Bellman para encontrar los valores óptimos actualizando los valores pasados de Q con un promedio ponderado de los nuevos. Es decir, se define el valor de Q (o de utilidad) de

tomar una acción en un estado actual en función de la recompensa recibida y el valor del futuro.

Por lo que se define:

$$Q^{new}(s_t, a_t) \leftarrow \underbrace{Q(s_t, a_t)}_{oldvalue} + \underbrace{\alpha}_{learningrate} \cdot \overbrace{(\underbrace{r_t}_{reward} + \underbrace{\gamma}_{disc.fact.} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{est.optimalfutureval.}}^{newvalue(tdiff.)} \overbrace{- \underbrace{Q(s_t, a_t)}_{oldvalue})}^{tdiff} \tag{2}$$

Para guardar los valores intermedios se utiliza una tabla (llamada **Q-table**) y allí se van actualizando los valores a medida que el agente comienza a experimentar los diferentes estados, tomando diferentes acciones y observando las diferentes recompensas del entorno durante cierta cantidad de tiempo.

2.1 Deep Q-Learning

Unas de las grandes limitaciones de Q-Learning es el tamaño que toma la **Q-table** debido a que tiene que guardar el **q-value** por cada par (estado, acción) posible y esto, dependiendo del entorno, puede tener un gran tamaño.

Es por esto que otra opción es aproximar la **Q-function** óptima y acá es donde entran en juego las redes neuronales.

2.2 Algoritmo Deep Q-Learning

A continuación se presenta el algoritmo implementado (Double Deep Q-Learning), algunas de sus decisiones como las de usar 'experience replay memory', explorar vs. explotar, usar dos redes neuronales profundas en vez de una se pueden encontrar en las referencias del presente trabajo ya que no es nuestra intención entrar en detalles sobre este algoritmo en especifico ya que existen diversas variantes de Deep Q-Learning.

Los pasos que sigue el algoritmo de Deep Q-Learning son:

1. Inicializar la capacidad de la 'replay memory'
2. Inicializar una red neuronal con pesos aleatorios (policy network)
3. Inicializar otra red neuronal con los pesos de la anterior (target network)
4. Por cada episodio:
   (a) Ir al estado inicial
   (b) Por cada paso de tiempo **t**:

i. Elegir una acción (por exploración o explotación)
ii. Tomar la acción elegida y simularla en el entorno
iii. Observar el próximo estado y la recompensa
iv. Guardar la experiencia en la 'replay memory'
v. Generar una muestra de un tamaño fijo de experiencias
vi. Pasar la muestra de experiencias a la policy network
vii. Calcular el estimado de **q-value** óptimo utilizando la ecuación de Bellman (requiere pasar el siguiente estado a la target network)
viii. Minimizar la pérdida en la policy network con gradient descent
   A. Luego de un cierto tiempo actualizar los pesos de la target network con los de la policy network

2.3 Arquitectura de una 'Deep Q Network' (DQN)

En general la arquitectura de una red de Deep Q-Learning consiste en una serie de capas convolucionales que procesan las imágenes del entorno, en general se pasan varias imágenes en secuencia. A estas le siguen una serie de capas fully-connected (Dense en Keras) y por último el output layer tiene tantas neuronas como cantidad de acciones que puede tomar el agente en el entorno.

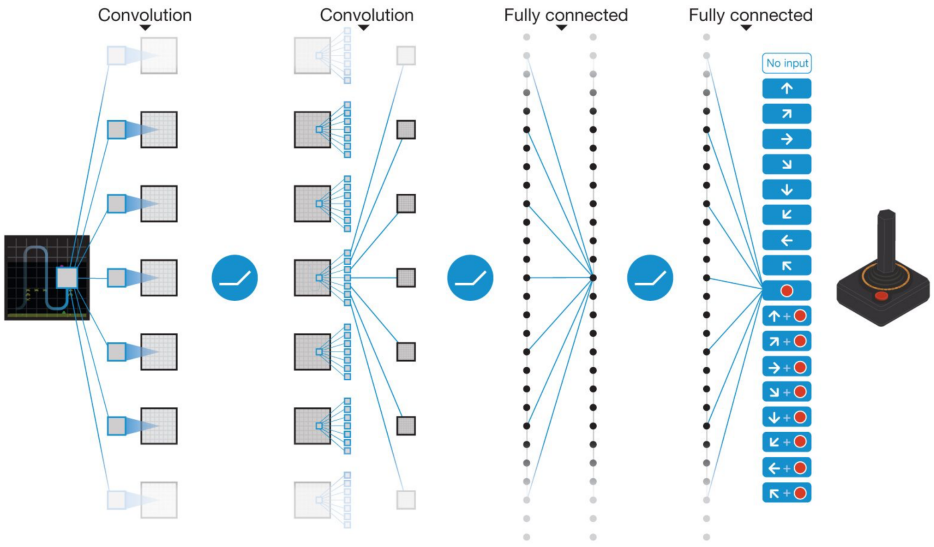

Fig. 1. Arquitectura de una DQN

En nuestra implementación no se utilizaron las capas convolucionales, si no que se realizó un preprocesamiento del estado (el cual estaba dado como una

matriz de píxeles) a una estructura de menor dimensión que consumía la red.

Queda para una futura iteración probar agregar unas capas convolucionales.

3 Implementación

A continuación se mencionan los detalles de implementación que se tomaron para la realización de este trabajo.

3.1 Frameworks

Los frameworks más principales que se utilizaron fueron:

GVGAI GYM Se utilizó este framework que permite codificar nuestro agente en Python y se comunica con un server en Java que corre diferentes juegos/entornos (en este caso Zelda).
Básicamente lo utilizamos para ver los diferentes estados del entorno y ejecutar diferentes acciones en él, viendo así las recompensas que se van obteniendo con el fin de entrenar nuestro agente.

Tensorflow (Keras) Keras es una librería construida sobre Tensorflow, se utilizó para crear y entrenar las diferentes redes neuronales que requiere la implementación de Deep Q-Learning.

Tensorboard Se utilizó Tensorboard para observar las métricas durante el entrenamiento que eran reportadas por callbacks en cada iteración.

3.2 Entorno

El entorno que se utilizó para entrenar el agente se llama 'gvgai-zelda-lvl0-v0' y consta de un rectángulo de 13x9 cuadrados, donde se encuentran el jugador en sí cuyo objetivo es agarrar la llave y salir por la puerta esquivando/matando los enemigos en el camino.

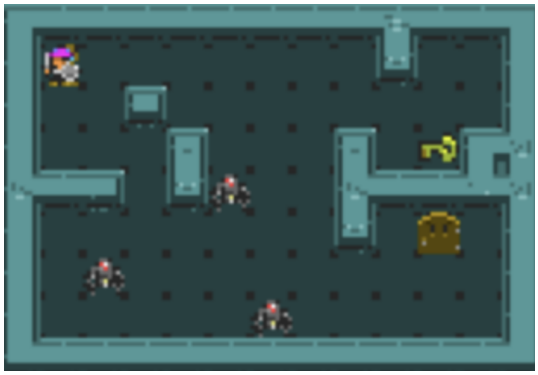

Fig. 2. Entorno - Zelda

La entrada de la red fue cambiando mientras probamos diferentes implementaciones. Comenzamos teniendo como representación del estado (y como entrada de la red) todos los píxeles de la imagen del juego y rapidamente nos dimos cuenta que no fue una decisión acertada, al tener una entrada con tantas neuronas la red no aprendía ya que no podía reconocer los diferentes elementos del entorno por lo que tendríamos que haber agregado una capa convolucional si hubiesemos querido seguir con esta representación.

Finalmente optamos por realizar una transformación de la matriz de píxeles a una matriz de 9x13 teniendo como entrada una capa de 117 neuronas donde cada una tiene como valor el elemento que se encuentra en el juego (pared, lugar vacío, jugador, enemigo, llave o puerta).

3.3 Acciones y recompensas

Las acciones que puede tomar el agente son:

1. NIL: No hacer nada
2. USE: Mata un enemigo si lo tiene enfrente o agarra la llave
3. LEFT: Moverse a la izquierda
4. RIGHT: Moverse a la derecha
5. DOWN: Moverse abajo
6. UP: Moverse arriba

En cuanto a las recompensas fuimos cambiado según las implementaciones, pero generalmente (en el caso de cambiar las recompensas que trae el framework) siempre castigamos (recompensa negativa) la acción NIL y el USE (en caso de que no haya un enemigo o no esté la llave) y en algunos casos castigamos el moverse y el quedarse en el lugar.

Los resultados se discutiran en las secciones siguientes.

3.4 Estructura de la red

A continuación se presenta la estructura de las redes utilizada para las diferentes implementaciónes, los parámetros indicados no se cambiaron entre los diferentes entrenamientos.

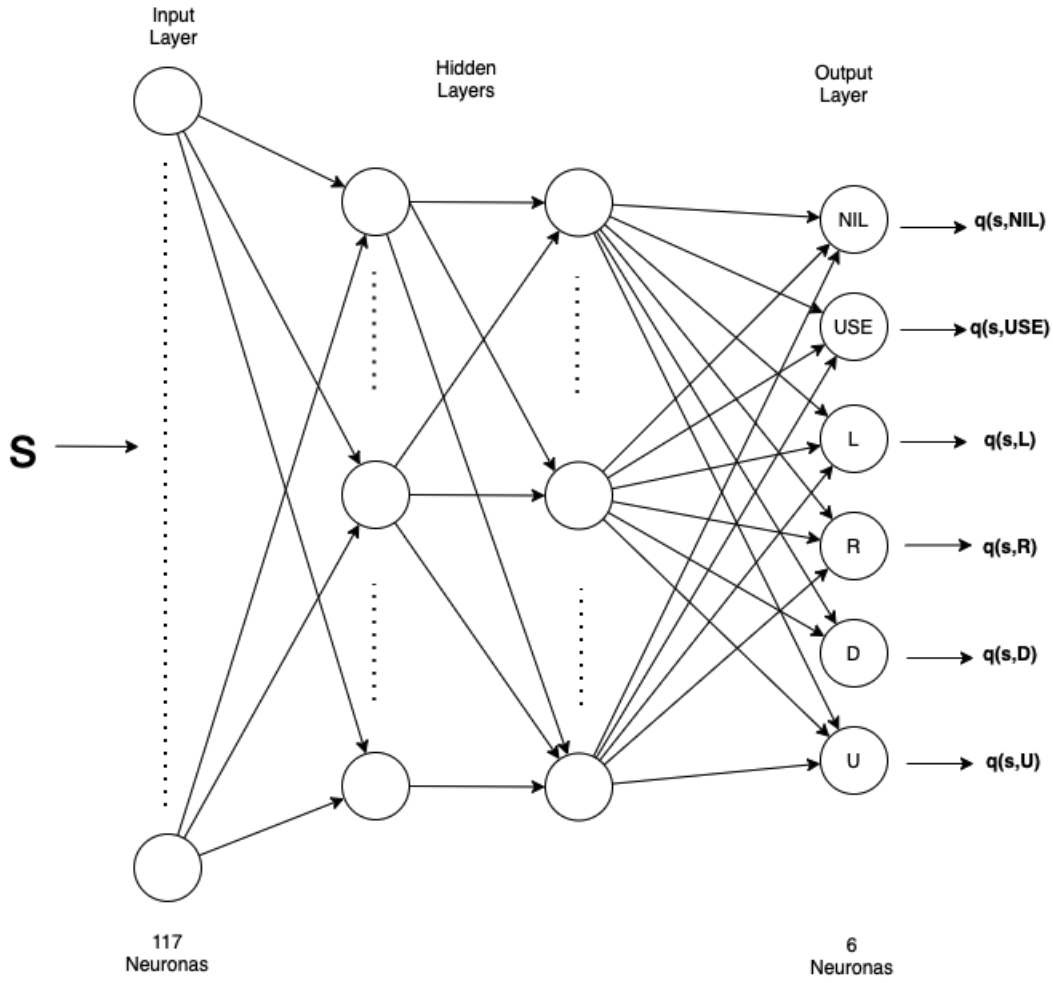

Fig. 3. DQN - Estructura implementada

4 Entrenamiento

En esta sección presentaremos los distintos intentos junto a los parámetros utilizados para entrenar el agente y que aprenda a jugar al Zelda.

4.1 Primer intento

En un primer intento se probó directamente tener como representación del estado la matriz de píxeles, pero esto fue descartado rápidamente ya que no se observó aprendizaje alguno, así mismo se observó que la 'loss' function no llegaba a un mínimo y tenia forma de serrucho.

4.2 Segundo intento (Zelda-01)

En este segundo intento se realizó la transformación de la matriz de píxeles que nos devuelve el framework como representación del estado a una matriz de 9x13 donde cada valor representa un elemento en el mapa y se utilizó esto como input de la red.

En cuanto a las **recompensas** se utilizaron las que trae el framework GYM AI por default, donde:

- Matar un enemigo, agarrar la llave y salir por la puerta tienen recompensa positiva.
- Morir tiene recompensa negativa.
- Moverse en cualquier dirección, no hacer nada o usar (cuando no está la llave o no hay enemigos) tiene recompensa 0.

Los **parámetros** utilizados fueron:

- Input layer: 117 neuronas (una por cada casillero del tablero)
- Hidden layers: 2 capas de 24 neuronales fully-connected (activation function: ReLU)
- Output layer: 6 neuronas (activation function: Linear)
- Loss function: MSE (mean squared error)
- Learning rate: 0.1
- Gamma: 0.6
- Epsilon inicial: 1.0
- Epsilon mínimo: 0.1
- Epsilon descuento por timestamp: 0.0001
- Replay memory size: 2000
- Batch size: 32
- Episodios: 100
- Timestamps por episodio: 1000

En cuanto a la función 'loss' podemos observar que disminuye con el paso de los episodios pero luego se vio un crecimiento.

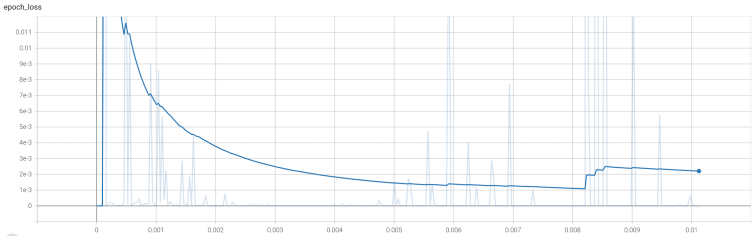

Fig. 4. Segundo intento - Evolución de la 'loss function' en el entrenamiento

Los **resultados** obtenidos no fueron muy alentadores, observamos al agente quedarse en su lugar y utilizar la función 'USE' constantemente. Nuestra teoría es que al no tener recompensa negativa el quedarse en el lugar el agente aprovecha y ve esto como la acción más beneficiosa.

4.3 Tercer intento (Zelda-02)

En este intento probamos darle una recompensa negativa al agente por no hacer nada esperando así que no se quede en un mismo lugar.

Es por esto que sobreescribimos las **recompensas** por default del framework y utilizamos:

- Si gana el juego: 100
- Si pierde el juego (lo matan): -100
- Si la recompensa es mayor que cero: Se multiplica por un factor de 10
- Si no hace nada (NIL) o quiere ejecutar (USE) sin enemigos o sin llave: -100
- Si la recompensa es 0: -10

Los **parámetros** utilizados fueron:

- Input layer: 117 neuronas (una por cada casillero del tablero)
- Hidden layers: 2 capas de 24 neuronales fully-connected (activation function: ReLU)
- Output layer: 6 neuronas (activation function: Linear)
- Loss function: MSE (mean squared error)
- Learning rate: 0.1
- Gamma: 0.6
- Epsilon inicial: 1.0
- Epsilon mínimo: 0.1
- Epsilon descuento por timestamp: 0.0001
- Replay memory size: 2000
- Batch size: 32
- Episodios: 100
- Timestamps por episodio: 1000

En cuanto a la función 'loss' podemos observar que disminuye con el paso de los episodios hasta llegar a una meceta.

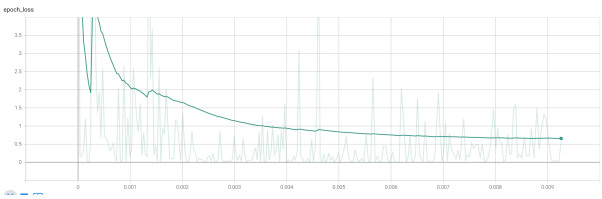

Fig. 5. Tercer intento - Evolución de la 'loss function' en el entrenamiento

En este caso los **resultados** fueron que el agente selecciona (casi) siempre la misma acción para ejecutar, moviendose pero siempre para el mismo sentido. Se obtuvieron algunos escenarios alentadores (que nombraremos en la conclusión) pero aún así no logramos que el agente resuelva el juego.

4.4 Cuarto intento (Zelda-03)

En un último intento de lograr que el agente no se quede en su lugar tratamos dar una recompensa negativa al agente por no moverse de lugar, calculando su posición a partir del estado.

Las **recompensas** en este intento fueron:

- Si gana el juego: 100
- Si pierde el juego (lo matan): -100
- Si la recompensa es mayor que cero: Se multiplica por un factor de 10
- Si no hace nada (NIL): -200
- Si quiere ejecutar (USE) sin enemigos o sin llave: -100
- Si no se movió de su posición: -5
- Otro caso (cómo moverse): -1

También probamos cambiar un poco los **parámetros** utilizados:

- Input layer: 117 neuronas (una por cada casillero del tablero)
- Hidden layers: 4 capas fully-connected de (96,48,24,12) neuronas respectivamente (activation function: ReLU)
- Output layer: 6 neuronas (activation function: Linear)
- Loss function: MSE (mean squared error)
- Learning rate: 0.01
- Gamma: 0.6
- Epsilon inicial: 1.0
- Epsilon mínimo: 0.1
- Epsilon descuento por timestamp: 0.0001
- Replay memory size: 2000
- Batch size: 256
- Episodios: 100
- Timestamps por episodio: 1000

No encontramos anomalías en la evolución de la 'loss function'.

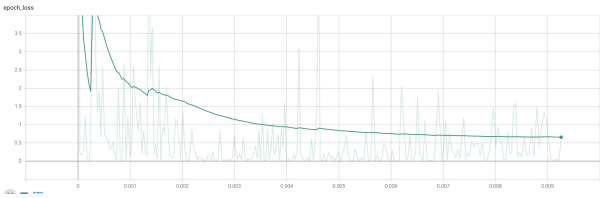

Fig. 6. Tercer intento - Evolución de la 'loss function' en el entrenamiento

Se observó un progreso en los **resultados** pero no logramos que el agente resuelva el nivel, incluso su performance fue peor que el intento anterior.

En este caso al tener varias neuronas el entrenamiento tardó demasiado e ibamos guardando modelos intermedio entre los episodios. Una serie de 10 episodios de 1000 timesteps nos llevó alrededor de 1 día completo en correr en nuestras maquinas.

5 Conclusión

5.1 Tiempo de entrenamiento

Una de las grandes desventajas de no poder haber acelerado el entrenamiento por GPU son los tiempos que se demoró en entrenar los diferentes agentes, en promedio los intentos llevaron más de un dia (24 horas) hasta poder correr una cantidad considerable de escenarios.

5.2 Mejores resultados

Se obtuvieron resultados interesantes en algunas pruebas del agente luego del tercer intento donde se logró que el agente mate a todos los enemigos. Aún así no fueron reproducibles consistentemente y otras veces se quedaba en un mismo lugar.

Fig. 7. Tercer intento - El agente logró matar a los enemigos

Y también se logró que pueda agarrar la llave.

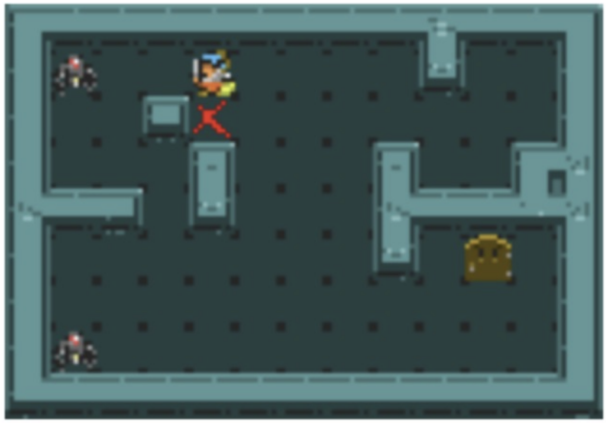

Fig. 8. Tercer intento - El agente logró agarrar la llave

5.3 Comentarios finales

Por más que no se logró obtener una Q-function cercana a la óptima, si pudimos implementar el algoritmo de Deep Q-Learning y aprender los fundamentos teóricos que subyacen sobre este. También se logró trabajar con frameworks que no habiamos utilizado (o muy poco) hasta esta altura de la carrera.

6 Referencias

1. Standford - Reinforcement Learning: An Introduction
   https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf
2. Deeplizard - Deep Learning playlist overview Machine Learning
   https://youtu.be/gZmobeGL0Yg
3. Deeplizard - Reinforcement Learning - Goal Oriented Intelligence
   https://deeplizard.com/learn/playlist/PLZbbT5o$_s$2$xoWNVdDudn51XM8lOuZ_NjvDeepQ-LearningwithPythonandTensorFlow$2.0
   $https://rubikscode.net/2019/07/08/deep-q-learning-with-python-and-tensorflow-2-0/$