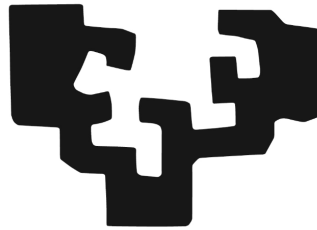


Double DQN y n-step DQN

Informe de trabajo

eman ta zabal zazu



Universidad
del País Vasco

Euskal Herriko
Unibertsitatea

Técnicas Avanzadas de la Inteligencia Artificial

21/12/2022

David Bernal Gómez
Sara Martín Aramburu
Ander Larrinaga Raya
Fernando Pérez Sanz

Índice

Índice	2
Double Deep Q-Network	3
Explicación	3
Cambios en el código	3
Resultados	4
N-Step	7
Explicación	7
Cambios en el código	7
Resultados	11

Double Deep Q-Network

En caso de que el código no sea visible, se puede acceder al collab mediante [este link](#).

Explicación

La diferencia principal entre *Deep Q-Network* y *Double Deep Q-Network* es que *Deep Q-Network* usa los mismos valores para seleccionar y evaluar una acción, esto hace que sea más probable seleccionar valores sobreestimados (con sesgo muy alto), lo que da como resultado estimaciones demasiado optimistas. Por esto, la idea principal es reducir esas sobreestimaciones que se realizan.

En Double Deep Q-Network usamos dos modelos en los que se asigna experiencia aleatoria para actualizar una de las dos funciones de valor. En cada actualización, se usa un conjunto de pesos para determinar la *greedy-policy* y el otro para determinar su valor.

En nuestro caso, disponemos de dos modelos:

- `current_model` (modelo actual, *self*): es el modelo que usaremos para elegir la acción. Actualizaremos los pesos de esta red en cada iteración.
- `target_model` (modelo objetivo): el modelo que usaremos para determinar el valor de la acción escogida por el `current_model`. Actualizaremos los pesos de esta red cada cierto número de iteraciones.

Cambios en el código

Para adaptar el código base al método Double Deep Q learning, hemos tenido que modificar la función de pérdida, `compute_td_loss_our()`.

La función de pérdida calcula el *loss* eligiendo la acción con el mejor q valor y aplicando la fórmula vista en clase:

$$q(s, a, w) \leftarrow q(s, a, w) + \alpha (r + \gamma * \max_a q(s', a', w) - q(s, a, w))$$

Que en el código se implementa así:

```
next_q_value = max(next_q_values_target_model[i])
expected_q_value[i] = reward[i] + self.gamma*next_q_value
```

Debemos modificar la función de pérdida, de tal manera que los pesos se actualicen de la siguiente manera:

$$\hat{q}(s, a, \mathbf{w}) \leftarrow \hat{q}(s, a, \mathbf{w}) + \alpha (r + \gamma \underbrace{\hat{q}(s', \arg \max_{a'} \hat{q}(s', a', \mathbf{w}'), \mathbf{w}')}_{\text{next_q_value}} - \hat{q}(s, a, \mathbf{w}))$$

Siendo $q(s, a, w)$ el modelo actual (elegirá la acción con mayor q-valor), y $q(s, a, w')$ el modelo objetivo, que calculará el q-valor de la acción elegida.

En el código se implementa de la siguiente manera:

```
next_action = np.argmax(next_q_values_current_model[i])
next_q_value = next_q_values_target_model[i][next_action]

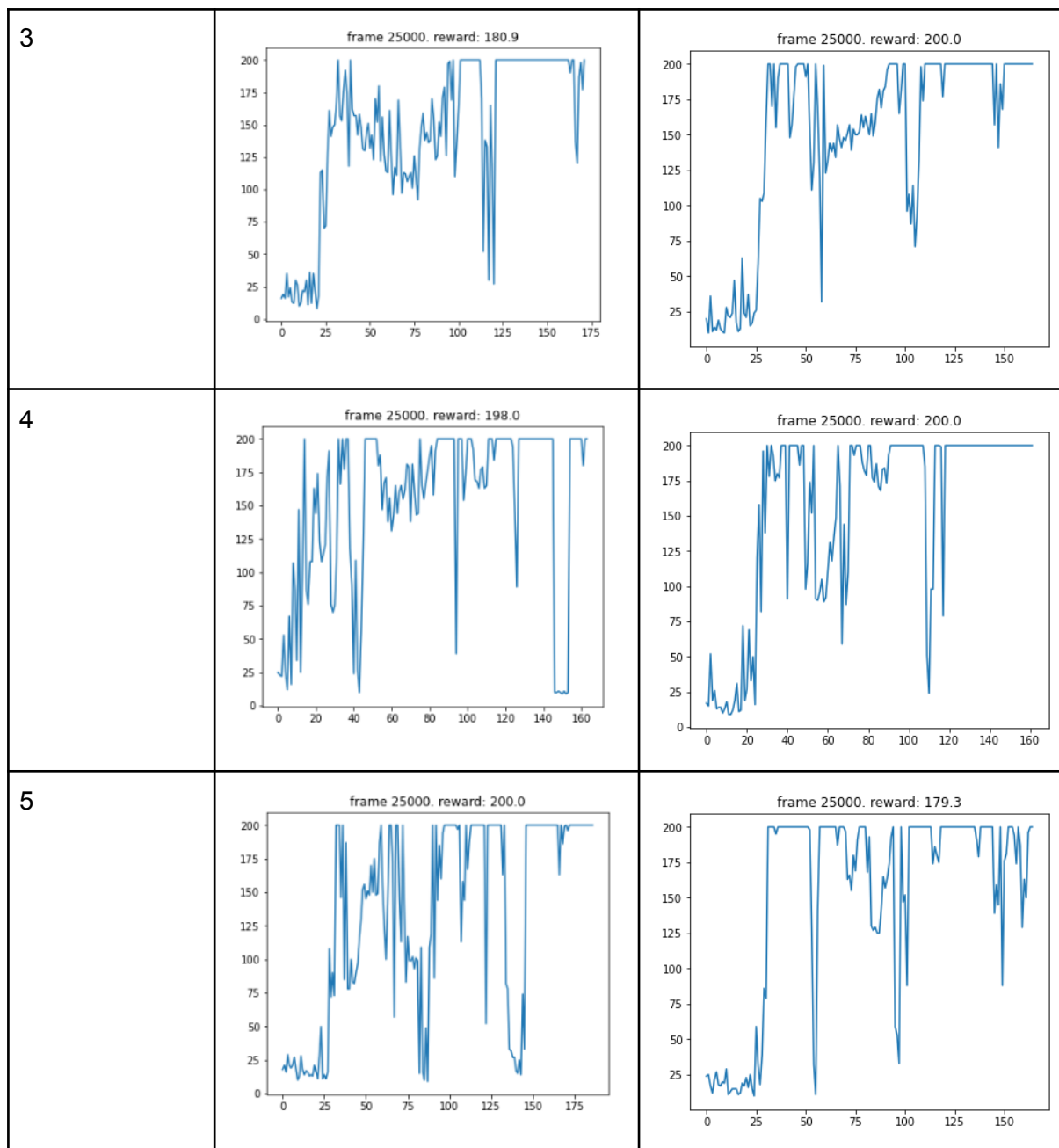
expected_q_value[i] = reward[i] + self.gamma*next_q_value
```

Resultados

La forma ideal de comprobar los resultados sería probar los algoritmos mediante la función `.test()`. Sin embargo obtenemos vídeos de corta duración como resultado, y no son suficientes para evaluar y comparar.

Por lo tanto, la forma en que lo haremos nosotros será, entrenaremos 5 veces cada modelo (double DQN y DQN) para posteriormente comparar las gráficas resultantes. Las gráficas muestran la recompensa obtenida (eje y) a lo largo de 25.000 iteraciones (eje x). Solo se representan los valores de recompensa cada 200 iteraciones, (por ejemplo, cuando $x=20$, nos encontramos en la iteración $20 \cdot 200 = 4000$)

Entrenamiento	Double DQN	DQN
1	<p>frame 25000. reward: 200.0</p>	<p>frame 25000. reward: 192.0</p>
2	<p>frame 25000. reward: 200.0</p>	<p>frame 25000. reward: 117.6</p>



Con estos resultados podemos comparar:

- Las veces que finaliza con la recompensa máxima:
 - DDQN: 3/5 pruebas
 - DQN: 2/5 pruebas
- El momento en el que la recompensa se dispara:
 - DDQN: entre las iteraciones 0 y 4000 ($20 \cdot 200$)
 - DQN: entre las iteraciones 0 y 6000 ($30 \cdot 200$)
- Cuánto tiempo se mantienen los modelos con una recompensa máxima:
 - Ambos modelos sufren altibajos en las iteraciones intermedias

Con estos resultados no podemos demostrar con claridad que uno sea mejor que otro. Primeramente porque no es la mejor forma de comprobar el desempeño de los modelos, y

por otra parte, porque el número de pruebas realizadas no es muy grande. Sin embargo, sí que podemos decir que DDQN empieza a tener una recompensa mucho más alta unas 2000 iteraciones antes que DDQN, y finaliza con una recompensa máxima en más ocasiones, lo que nos puede llevar a la conclusión de que el desempeño de DDQN es ligeramente superior.

N-Step

En caso de que el código no sea visible, se puede acceder al collab mediante [este link](#).

Explicación

El n-Step en DQN es una implementación del n-step TD en DQN. Para entenderlo mejor primero expliquemos en qué consiste n-step TD.

El n-step TD es un compromiso entre Monte Carlo y Temporal Difference, en la que cada valor de cada estado no se calcula únicamente con la recompensa de tomar una acción y el valor del próximo estado (como en TD[0]) o con el cálculo completo de una rama (como en Monte Carlo).

N-step actúa como Monte Carlo pero únicamente hasta un estado n, no sigue calculando los siguientes estados hasta llegar al terminal, siguiendo la siguiente fórmula.

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n}) - V(S_t))$$

Para adaptar esto al DQN, lo que se debe de hacer es cambiar la manera en la que se guardan las experiencias, específicamente las recompensas de cada experiencia y también indicando que el próximo estado no será S_{t+1} sino el S_{t+n} .

$$D = \{ \langle s_t, a_t, (r_{t+1} + \gamma r_{t+2} + \dots + \gamma^{n-1} r_{t+n}), s_{t+n} \rangle \}$$

Por último, según la teoría del n-step, la información de cada experiencia se debe de hacer el el estado n de cada respectivo estado excepto cuando se ha llegado a un estado terminal. En ese caso, se deben de subir todas las experiencias acumuladas.

Cambios en el código

Para adaptar el código base al método N-Step DQN, se han de modificar varias funciones del código. Empecemos por el `train(self,n)`, que es donde se define cómo se almacenan las experiencias. Para ello, vamos a dividir el código en partes para explicarlo mejor

```
list_states = []
list_actions = []
list_rewards = []

for frame_idx in range(1, num_frames + 1):
    epsilon = self.get_epsilon_value(frame_idx)
    action = self.act(state, epsilon)

    next_state, reward, done, _ = self.env.step(action)
```

```
list_states.append(state)
list_actions.append(action)
list_rewards.append(reward)
```

Al inicio creamos tres listas distintas. Una será para guardar los estados, otra las acciones y la última las recompensas de llegar a un estado.

Tras esto, guardamos en la lista de estados el estado del que partimos, en la lista de acciones guardamos la acción tomada (teniendo en cuenta el estado en el que estamos y el epsilon) y finalmente la recompensa de tomar esa acción en ese estado la guardamos en el array de recompensas.

```
if not done:
    state = next_state
    episode_reward += reward
```

A continuación, si no nos encontramos en un estado terminal, lo que el programa hará es pasar al siguiente estado (el estado que obtenemos al tomar la acción anteriormente almacenada) y guardar en las listas el mismo tipo de información que con el estado anterior (el nuevo estado, su acción y su recompensa), solo que esta vez teniendo en cuenta que estamos en este nuevo estado.

De esta manera, las listas se irán llenando poco a poco.

```
if len(list_states) == n+1: #Si no es terminal pero ya tenemos los n estado siguientes de un estado, se sube su experiencia

    experience_reward = 0
    for i in range(len(list_rewards)):
        experience_reward += pow(self.gamma,i)*list_rewards[i]
    list_rewards.pop(0)

self.replay_buffer.push(list_states.pop(0),list_actions.pop(0),experience_reward,state,
done) #Cada vez que se sube una experiencia, se borra su información de los arrays de guardado
```

Llegará un momento en el que las listas tendrán un tamaño de $n+1$, es decir, tendrán almacenada la información del nodo inicial más de los siguientes n nodos.

Con esto, podemos construir la experiencia de ese primer nodo. Subiendo al buffer la información del primer nodo (que será el primero de la lista), con su acción (una vez más, el primer dato) y su recompensa, que será la suma de la recompensa de ese nodo más las recompensas de los siguientes n nodos almacenados cada uno con su respectiva potencia de gamma.

Por último, el parámetro de próximo estado que le debemos de pasar a la experiencia es el propio estado en el que está en ese momento el entorno. Lo mismo ocurre con el done.

Se ha de aclarar, que cada vez que se sube una experiencia al buffer, se borra de cada lista la información del nodo sobre el que se sube la experiencia.

```
if done:#Si estamos en un estado terminal, debemos subir todas las experiencias guardadas

    for i in range (len(list_states)):
        experience_reward = 0
        for i in range(len(list_rewards)):
            experience_reward += pow(self.gamma,i)*list_rewards[i]
        list_rewards.pop(0)

    self.replay_buffer.push(list_states.pop(0),list_actions.pop(0),
        experience_reward,state, done)
    #Cada vez que se sube una experiencia, se borra su información de los arrays de guardado

    state = self.env.reset()
    all_rewards.append(episode_reward)
    episode_reward = 0
```

Por último, si hemos llegado a un estado terminal, debemos de subir todas las experiencias guardadas (el conjunto de las tres listas) y vaciar las listas. Además, tras esto se debe reiniciar el estado.

Con todo esto, el código completo es el siguiente:

```
list_states = []
list_actions = []
list_rewards = []

for frame_idx in range(1, num_frames + 1):
    epsilon = self.get_epsilon_value(frame_idx)
    action = self.act(state, epsilon)

    next_state, reward, done, _ = self.env.step(action)
    list_states.append(state)
    list_actions.append(action)
    list_rewards.append(reward)

    if done:#Si estamos en un estado terminal, debemos subir todas las experiencias guardadas

        for i in range (len(list_states)):
            experience_reward = 0
```

```

        for i in range(len(list_rewards)):
            experience_reward += pow(self.gamma,i)*list_rewards[i]
        list_rewards.pop(0)

self.replay_buffer.push(list_states.pop(0),list_actions.pop(0),experience_reward,state,
done) #Cada vez que se sube una experiencia, se borra su información de los arrays de
guardado

state = self.env.reset()
all_rewards.append(episode_reward)
episode_reward = 0

if len(list_states) == n+1:#Si no es terminal pero ya tenemos los n estado siguientes
de un estado, se sube su experiencia

    experience_reward = 0
    for i in range(len(list_rewards)):
        experience_reward += pow(self.gamma,i)*list_rewards[i]
    list_rewards.pop(0)
    self.replay_buffer.push(list_states.pop(0),list_actions.pop(0),
experience_reward,state, done)
#Cada vez que se sube una experiencia, se borra su información de los arrays de
guardado

if not done:
    state = next_state
    episode_reward += reward

```

Con el `train(self,n)` aclarado, pasemos a la otra función que ha sido alterada: `compute_td_loss_our(self, target_model,n)`:
 En esta función es donde se calcula la pérdida para luego calcular los gradientes y actualizar los pesos.

```

state, action, reward, next_state, done = self.replay_buffer.sample(self.batch_size)
...
#código intermedio que no es relevante para la explicación
...

for i in range(self.batch_size):
    next_q_value = 0
    if not(done[i]):
        next_q_value = max(next_q_values_target_model[i])

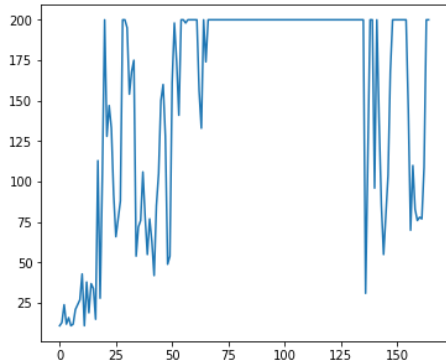
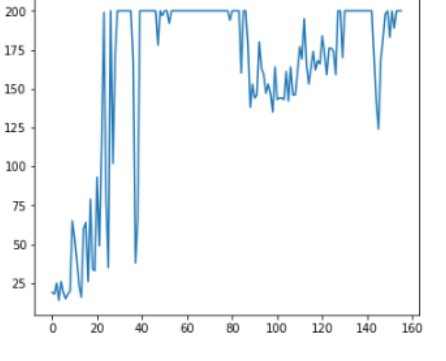
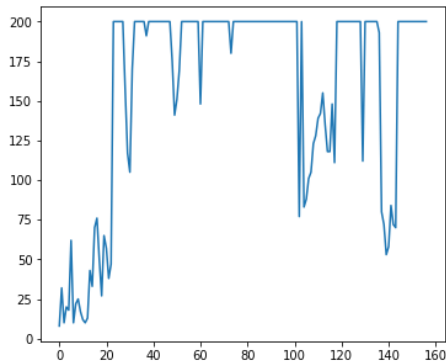
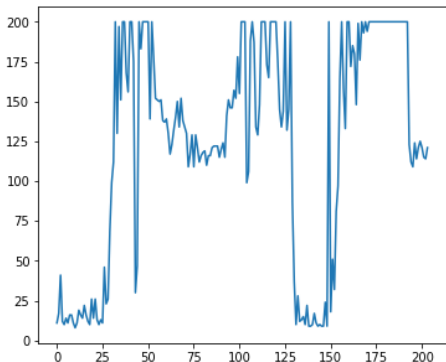
    expected_q_value[i] = reward[i] + pow(self.gamma,n)*next_q_value

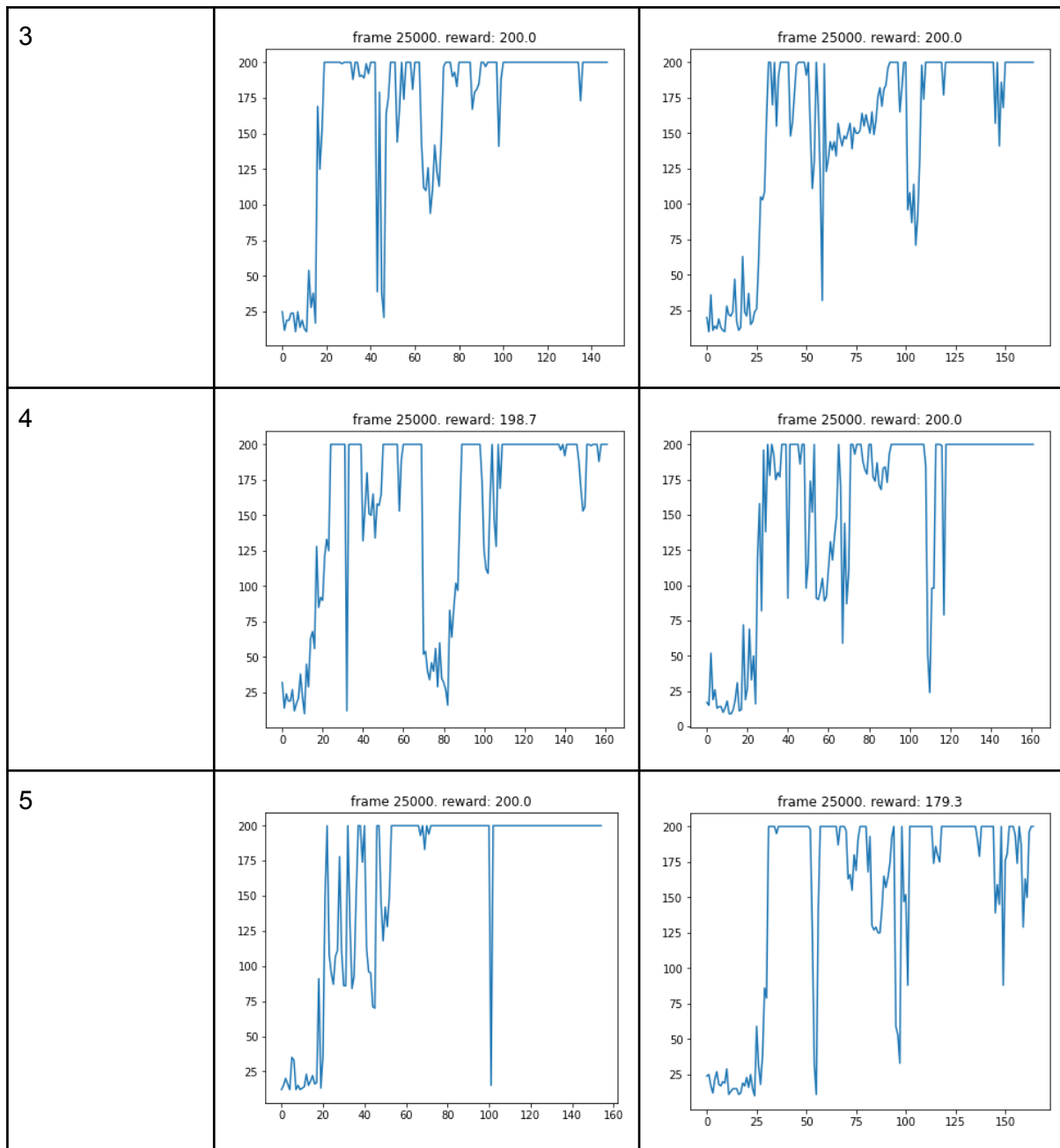
```

Para calcular el q value esperado para luego realizar la perdida, lo que hacemos es sumarle a las recompensas de un estado (previamente calculadas y almacenadas en el buffer) el valor del próximo estado (que será el next_stateN que almacenamos en el buffer) multiplicado por gamma elevado a la n.

Resultados

Al igual que con Double DQN, probaremos n-step DQN de la misma manera: 5 entrenamientos de n-step y 5 entrenamientos de DQN, para posteriormente comparar las gráficas.

Entrenamiento	n-step DQN	DQN
1	<p>frame 25000. reward: 114.7</p> 	<p>frame 25000. reward: 192.0</p> 
2	<p>frame 25000. reward: 200.0</p> 	<p>frame 25000. reward: 117.6</p> 



Con estos resultados podemos comparar:

- Las veces que finaliza con la recompensa máxima:
 - 3-Step DQN: 3/5 pruebas
 - DQN: 2/5 pruebas
- El momento en el que la recompensa se dispara:
 - 3-Step DQN: Entre las iteraciones 0 y 4000 (20×200)
 - DQN: Entre las iteraciones 0 y 6000 (30×200)
- Cuánto tiempo se mantienen los modelos con una recompensa máxima:
 - 3-Step DQN: Tarda más iteraciones de las realizadas para la prueba y no llega a converger del todo. Siempre observamos subidas y bajadas en la gráfica.
 - DQN: Sufre altibajos en las iteraciones intermedias

Tanto DQN como 3-Step DQN tienen gráficas parecidas, por lo que no es sencillo determinar que algoritmo es mejor con este método. Si que es verdad que 3-Step DQN tiene mejor resultado cuando se realizan más iteraciones y se mantiene un poco más estable que DQN, pero tampoco es una mejoría notable.