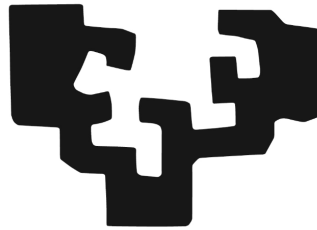


Experience replay

Informe de trabajo

eman ta zabal zazu



Universidad
del País Vasco

Euskal Herriko
Unibertsitatea

Técnicas Avanzadas de la Inteligencia Artificial

25/12/2022

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

Índice

Índice	2
1. Explicación	3
2. Cambios en el código.	4
• Cambios en push():	4
• Añadida función actualizar_pesos():	4
• Cambios en sample():	5
• Cambios en compute_td_loss_our()	6
3. Resultados	7

1.Explicación

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

Una de las mejoras de DQN es el *experience replay*. Éste método consiste en generar una serie de experiencias $\langle S, A, R, S' \rangle$ y almacenarlas en un buffer. En cada iteración seleccionaremos algunas de estas experiencias de forma aleatoria para entrenar la red.

Por otra parte, el entrenamiento generará nuevas experiencias, que serán almacenadas en el buffer, sustituyendo a las más antiguas.

Hay experiencias de las que el agente aprende más, así que es preferible seleccionar esas más frecuentemente. El objetivo de este apartado del proyecto es modificar la implementación del *experience replay* de tal manera que consigamos un *prioritized experience replay*, es decir, trabajar con las experiencias más útiles, en vez de con un subconjunto aleatorio

Para ello guardaremos una lista, las prioridades de cada experiencia. Dicha prioridad nos dirá la “importancia” que tiene la experiencia para que el agente aprenda. Utilizaremos el TD error para asignar probabilidades a cada experiencia.

$$p_j = |\text{TD_error}_j| = |\hat{q}(S_t, A_t, \mathbf{w}) - (r_{t+1} + \gamma \max_{a'} \hat{q}(S_{t+1}, a', \mathbf{w}))|$$

Donde p_j es la prioridad que tiene la experiencia j . Las prioridades nos servirán para calcular la probabilidad de elegir la experiencia para el mini-batch $P(j)$

$$P(j) = \frac{p_j^\kappa}{\sum_k p_k^\kappa}$$

$P(j)$ es la probabilidad de elegir la muestra j , donde p_j^κ es la prioridad de j (TD-error) y kappa (κ) un parámetro que define uniformidad en la probabilidad. Para nosotros no tiene mucha importancia, definiremos $\kappa = 0.6$

Al pasar de una distribución normal a una distribución definida por $P(j)$, añadimos un sesgo. Para solucionarlo, usamos importance sampling.

$$w_j = \left(\frac{1}{N} \cdot \frac{1}{P(j)} \right)^\beta / \max_i w_i$$

Aplicaremos esta fórmula para actualizar los pesos. N es el número total de elementos almacenados en el buffer. Ese valor obtenido lo normalizamos, dividiéndolo entre el peso máximo.

Beta es un nuevo parámetro, que tendrá el valor $\beta = 0.4$

Tras cada iteración debemos volver a calcular el TD-error de las experiencias utilizadas y actualizar su prioridad. Además actualizaremos los pesos añadiendo importance sampling, como mencionamos antes.

2. Cambios en el código.

Primero, a la hora de rellenar el buffer, añadimos también las prioridades, en el array `self.prioridades`. Cuando el buffer no se encuentra lleno, añadimos una prioridad de 1, mientras que en el caso contrario se modifican los valores más viejos y se reemplazan con el valor máximo de prioridad.

- Cambios en `push()`:

```
def push(self, state, action, reward, next_state, done):
    state      = np.expand_dims(state, 0)
    next_state = np.expand_dims(next_state, 0)

    if len(self.buffer) < self.capacity:
        self.buffer.append((state, action, reward, next_state, done))
        #####Añadido#####
        self.prioridades.append(1)
    else:
        self.buffer[self.pos] = (state, action, reward, next_state, done)
        self.prioridades[self.pos] = max(self.prioridades)
```

- Añadida función `actualizar_pesos()`:

También en el buffer, añadimos una función que nos permite actualizar los pesos.

```
#####Añadido#####
#dado el peso delta y el indice, lo actualiza en el array de prioridades
def actualizar_peso(self, delta, indice):
    # elevamos delta^2 (recomendacion de las transparencias)
    # self.prioridades[indice] = pow(delta,2)
    # otra solucion: utilizar el valor absoluto en vez de ^2, como pone en las transparencias
    self.prioridades[indice] = abs(delta)
#####
```

- Cambios en `sample()`:

En la función `sample`, calculamos las probabilidades de las experiencias almacenadas aplicando la fórmula mencionada anteriormente.

```
def sample(self, batch_size):
    #*****Añadido*****

    #calcular las probabilidades de las experiencias
    probabilities[j] = prioridad[j]^kappa / sum(prioridad ^ kappa)
    prioridadesNumpy = np.array(self.prioridades)
    probabilities = np.zeros(len(self.buffer)-1)
    probabilities = pow(prioridadesNumpy, self.kappa)
    probabilities = probabilities / np.sum(probabilities)
```

Obtenemos los índices de las muestras seleccionadas siguiendo la distribución de probabilidad que nos indica el array `probabilities`

```
#obtener batch_size elementos siguiendo las probabilidades calculadas
#obtenemos una lista con los indices de las experiencias elegidas
indices = random.choices(range(0,len(self.buffer)), weights=probabilities, k=batch_size )
```

Obtenemos las muestras seleccionadas, así como sus probabilidades

```
#obtenemos las experiencias elegidas a partir de los indices obtenidos
exp_elegidas = []
prob_elegidas = []
for i in indices:
    exp_elegidas.append(self.buffer[i])
    prob_elegidas.append(probabilities[i])
```

Actualizamos pesos.

```
#la pasamos a numpy array para que sea más facil operar
prob_elegidas = np.array(prob_elegidas)

#importance sampling w[i] = ( (1/N) * (1 / probabilities[i]) ) ^beta
N = len(self.buffer)
w = np.zeros(batch_size)
w = pow( ((1/N) * (1/prob_elegidas)) , self.beta)
w = w / w.max() #para normalizar
```

Además, `sample` devolverá además del conjunto de experiencias `< S, A, R, S' >` los pesos de dichas experiencias, para utilizarlos posteriormente en la función `compute_td_loss_our()`

- Cambios en `compute_td_loss_our()`

Cuando actualizamos los q values, actualizamos también los pesos según las experiencias obtenidas, con la función auxiliar `actualizar_peso`

```
#####Añadido#####
#actualizar el peso en el buffer
self.replay_buffer.actualizar_peso(expected_q_value[i], i)
#####
```

También cambiamos la forma en la que calculamos el loss, añadiendo a la ecuación los pesos calculados en `sample()`:

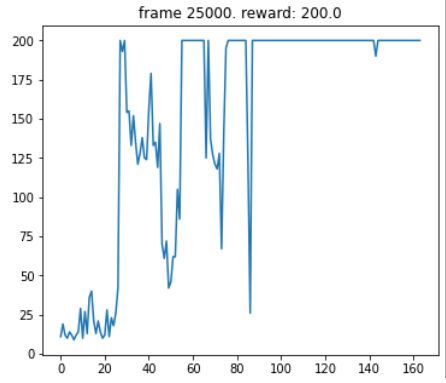
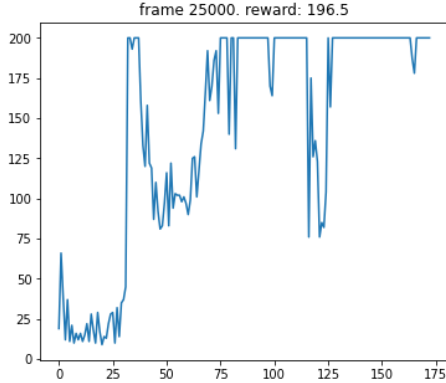
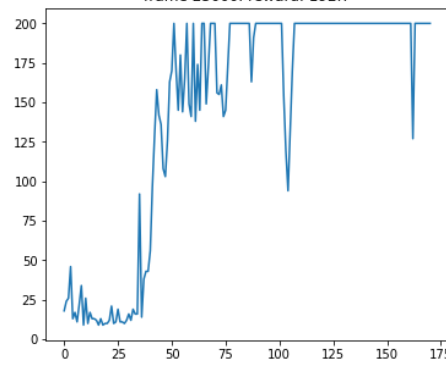
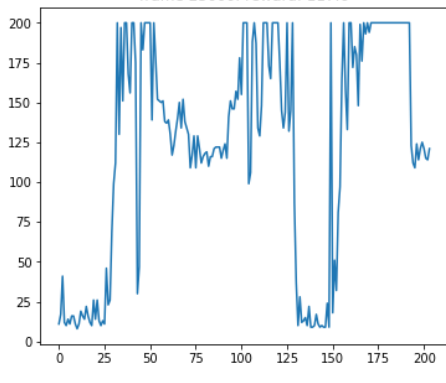
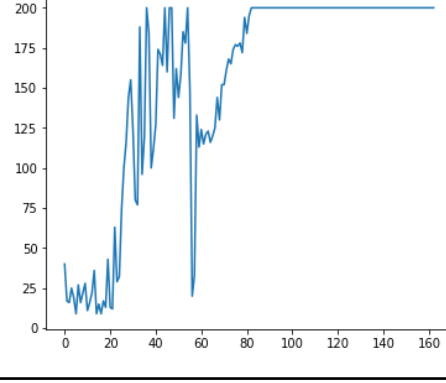
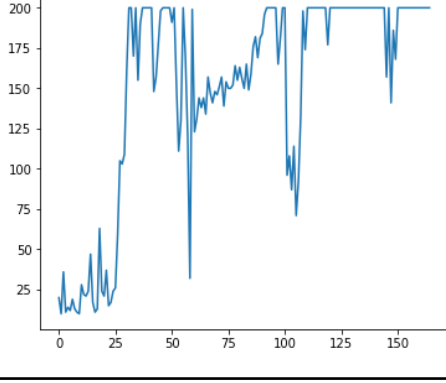
```
#####Añadido#####
loss = (q_value - expected_q_values.detach()).pow(2) * torch.tensor(weights)
loss = loss.mean()
#####
```

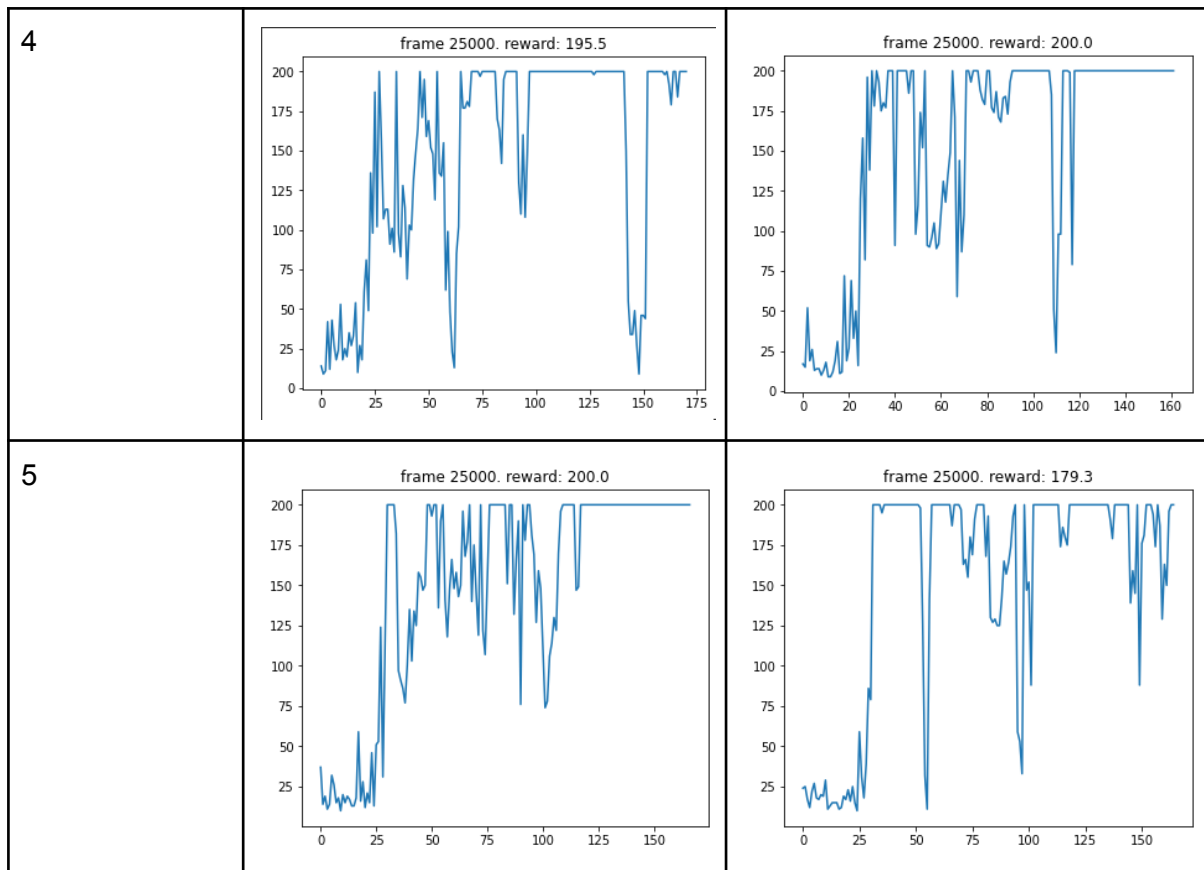
3.Resultados

La forma en la que vamos a evaluar y comparar es la siguiente:

Entrenaremos 5 veces cada modelo 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	Prioritized experience replay	“Basic” experience replay
1	 <p>frame 25000. reward: 200.0</p>	 <p>frame 25000. reward: 196.5</p>
2	 <p>frame 25000. reward: 192.7</p>	 <p>frame 25000. reward: 117.6</p>
3	 <p>frame 25000. reward: 200.0</p>	 <p>frame 25000. reward: 200.0</p>



Además, comparamos también las recompensas medias en cada prueba, así como la recompensa media de cada algoritmo:

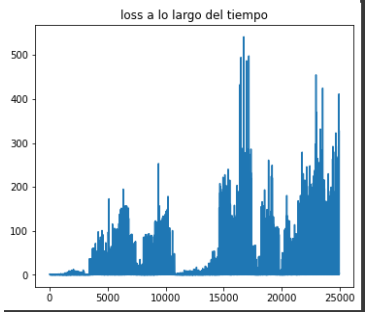
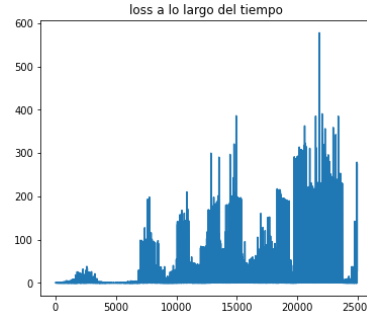
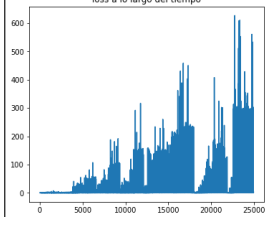
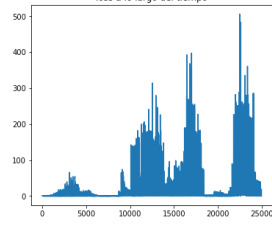
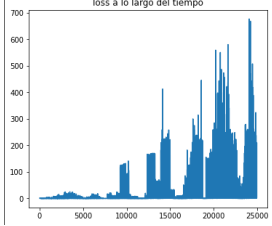
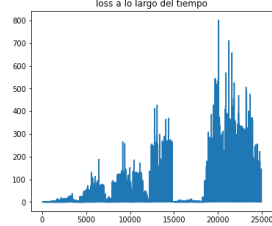
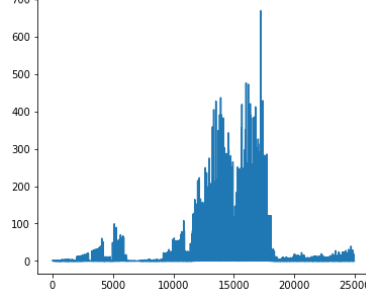
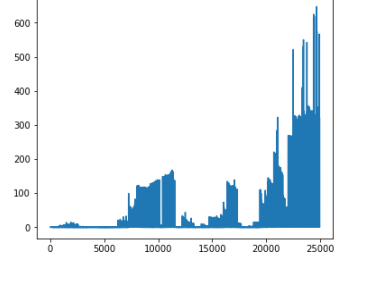
Entrenamiento	Prioritized experience replay	“Basic” experience replay
1	152	144
2	145	135
3	151	142
4	153	134
5	149	146
Media total	150	140

Con estos resultados podemos comparar:

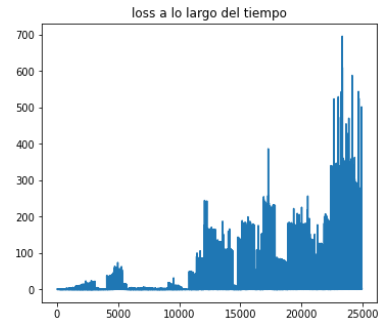
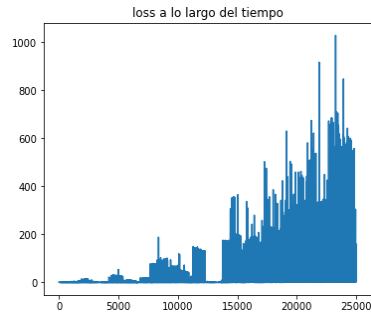
- Las veces que finaliza con la recompensa máxima:
 - Prioritized Experience replay: 3/5 pruebas
 - Basic Experience replay: 2/5 pruebas
- El momento en el que la recompensa se dispara:
 - Prioritized Experience replay: entre las iteraciones 0 y 5000 ($25 \cdot 200$)
 - Basic Experience replay: entre 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, aunque el experience replay menos.
- Recompensa media a lo largo de los entrenamientos:
 - Prioritized Experience replay: bastante superior, obtenemos una media de 150 puntos
 - Basic Experience replay: media de 140 puntos, ligeramente inferior

Además, obtenemos las siguientes gráficas de loss:

Entrenamiento	“Basic” experience replay	Prioritized experience replay
1		
2		
3		
4		

5



Como podemos observar gracias a la tabla, usando PER obtenemos una recompensa media mayor que usando BER (basic experience replay).

Por lo tanto, teniendo en cuenta ambas tablas con todas las pruebas vemos que obtenemos resultados un poco mejores usando PER. Usando esta técnica, los rewards se disparan un poco antes, aunque obtenemos la máxima recompensa solo una vez más.

En cuanto al loss, usando la primera técnica, nos mantenemos en valores más bajos más tiempo.