Entrega Tarea 2:
Q-Learning y DQN

Fecha de entrega: Martes 26 de abril, 23:59 hrs.

Estudiante: Francisco Molina L.
Profesor: Javier Ruiz del Solar
Auxiliar: Francisco Leiva C.
Semestre: Otoño 2022

1. Parte I: Q-Learning

1. En este problema el MDP es parcialmente observable, dado que el agente no tiene acceso directo a la información del entorno, por lo que no se conoce la matriz de transición de estados. Con esto en mente, el **POMDP** $\langle \boldsymbol{S}, \boldsymbol{A}, \boldsymbol{R}, \boldsymbol{\gamma} \rangle$ del carrito unidimensional se define por:

   - El espacio de estados $\boldsymbol{S}$: Corresponde a valores continuos de la posición y velocidad del carrito, ambos con límites superiores e inferiores:

   $$\boldsymbol{S}(x, v), \quad x \in [-1.2, 0.6], \quad v \in [-0.07, 0.07] \tag{1}$$

     La posición x está delimitada por los límites espaciales de la simulación, en -1.2 se sale del mapa por la izquierda y en 0.6 se llega al objetivo. Mientras que la velocidad v está delimitada por la potencia simulada del carrito, no puede ir más rápido que 0.07 en ambas direcciones.
   - Las acciones $\boldsymbol{A}$: Corresponde a la aceleración que puede escoger el carrito, representada por números enteros del 0 al 2:
     0) Aceleración negativa, el carrito se mueve a la izquierda
     1) Aceleración neutra, el carrito se mueve por inercia
     2) Aceleración positiva, el carrito se mueve a la derecha
   - La función de recompensa $\boldsymbol{R}$: Corresponde a la recompensa que recibe el agente por ejecutar cada acción. Para que el agente aprenda a llegar a la meta de la manera más rápida posible se aplica como recompensa una penalización de -1.0 por cada transición, excepto cuando se alcanza el objetivo donde se obtiene una recompensa de 0.0.
   - El factor de descuento $\boldsymbol{\gamma}$: Representa la incertidumbre sobre el futuro para el agente, hace que se valoren más o menos las recompensas futuras dependiendo del valor escogido y se escoge de manera arbitraria como 0.95.

2. Para poder aplicar Q-Learning tabular al problema el espacio de estados debe ser discreto. Considerando que es un problema pequeño se puede discretizar la posición cada 0.1 unidades y la velocidad cada 0.01.

   Con esto se obtienen 19 valores posibles para la posición:

   $$x \in [-1.2,\ -1.1,\ -1.0,\ -0.9,\ -0.8\ \ldots,\ 0.2,\ 0.3,\ 0.4,\ 0.5,\ 0.6] \tag{2}$$

   Y 15 valores posibles para la velocidad:

   $$v \in [-0.07,\ -0.06,\ -0.05,\ -0.04,\ \ldots,\ 0.04,\ 0.05,\ 0.06,\ 0.07] \tag{3}$$

   Por lo que el espacio de estados discretizado $\boldsymbol{S}[x, v]$ corresponde a las 285 combinaciones posibles entre los x y v recién descritos.

3. La función `select_action` recibe una observación de la cual se extrae el estado s y devuelve:

$$\texttt{select\_action} \rightarrow \begin{cases} \text{La acción } a \text{ que maximiza } Q(s,a) & \text{con probabilidad } \epsilon \\ \text{Una acción } a \text{ aleatoria} & \text{con probabilidad } 1-\epsilon \end{cases}$$

La función `update` recibe una tupla (s_t, s_{t+1}, a, r) y si s_t no es un estado terminal, actualiza su *Q-Value* según la regla:

$$Q^\pi(s_t, a_t) \mathrel{+}= \alpha(r_t + \gamma \max_{a'} Q^\pi(s_{t+1}, a') - Q^\pi(s_t, a_t)) \tag{4}$$

4. Usando los parámetros por defecto se obtiene la recompensa promedio y la tasa de éxito presentadas en las **Figuras 1 y 2:**

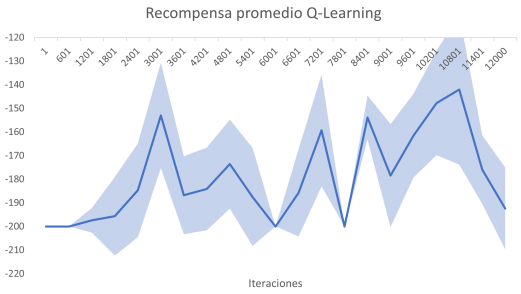

Figura 1: Recompensa promedio

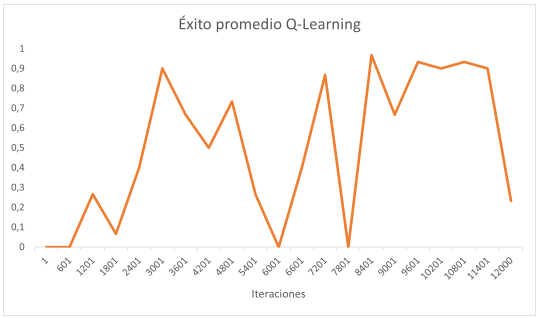

Figura 2: Tasa de éxito

5. Implementando ϵ-greedy se obtiene la recompensa promedio y la tasa de éxito presentadas en las **Figuras 3 y 4:**

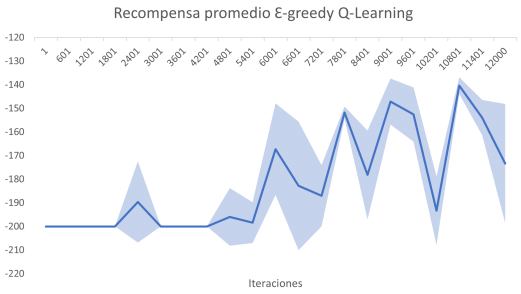

Figura 3: Recompensa promedio con ϵ-greedy

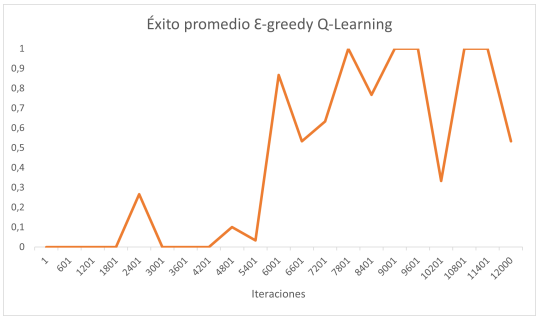

Figura 4: Tasa de éxito con ϵ-greedy

Se puede apreciar que al implementar ϵ-greedy se obtienen resultados mejores y más consistentes, especialmente en las últimas iteraciones. En la **Figura 4** se ve claramente que cerca de las 5400 iteraciones la tasa de éxito se dispara (ϵ está llegando a 0) y se mantiene alrededor de 0.8 de ahí en adelante.

2. Parte II: Deep Q-Network

1. Trabajando en el archivo `deep_qnetwork.py`:

1.1. Para completar la clase `DeepQNetwork` basta con definir una red neuronal con dos capas ocultas de 64 neuronas, usando funciones de activación ReLU:

```
    class DeepQNetwork(nn.Module):

    def __init__(self, dim_states, dim_actions):
        super(DeepQNetwork, self).__init__()
        self.device = torch.device("cuda" if torch.cuda.is_available()
                                   else "cpu")
        self.fc1 = nn.Linear(dim_states, 64)
        self.fc2 = nn.Linear(64, 64)
        self.fc3 = nn.Linear(64, dim_actions)
        self.float()

    def forward(self, input):
        out = input.to(self.device)
        out = F.relu(self.fc1(out))
        out = F.relu(self.fc2(out))
        out = self.fc3(out)
        return out

```

Listing 1: Clase DeepQNetwork

1.2. Para completar el constructor de la clase `DeepQNetworkAgent` se crea la *target network* copiando la red `deep_QNetwork` y se inicializan el optimizador y la pérdida

```
    # Complete
    self._deep_QNetwork = DeepQNetwork(self._dim_states,
                                       self._dim_actions)
    self._target_deepQ_network = copy.deepcopy(self._deep_QNetwork)

    # Adam optimizer and MSE Loss
    self._optimizer = torch.optim.Adam(self._deep_QNetwork.parameters(),
                                       lr=lr)
    self._criterion = nn.MSELoss()

```

Listing 2: Clase DeepQNetworkAgent

1.3. Para completar el método `select_action` se devuelve la acción que maximiza la salida de la red `deep_QNetwork` ($argmax_{a'} Q_\theta(s, a')$) con probabilidad ϵ o una acción aleatoria con probabilidad $1 - \epsilon$. Además, se reduce el valor de ϵ con cada iteración:

```
    def select_action(self, observation, greedy=False):
        if (np.random.random() > self._epsilon) or greedy:
            with torch.no_grad():
                Q_values = self._deep_QNetwork(torch.tensor(observation))
                action = Q_values.argmax()
        else:
            action = torch.tensor(np.random.randint(0, self._dim_actions))

        if not greedy and self._epsilon >= self._epsilon_min:
            self._epsilon -= self._epsilon_decay
        return action

```

Listing 3: select_action

2. Trabajando en el archivo `replay_buffer.py`:

2.1. Para completar el constructor de la clase `ReplayBuffer` se inicializan los *buffer* de s_t, a_t, r_t, s_{t+1} y *done* como *arrays* llenos de ceros con sus respectivas formas:

```
    def __init__(self, dim_states, dim_actions, max_size, sample_size):
        assert sample_size < max_size, "Sample size cannot be greater than
                                        buffer size"

        self._buffer_idx = 0
        self._exps_stored = 0
        self._buffer_size = max_size
        self._sample_size = sample_size

        self._s_t_array = np.zeros((self._buffer_size, dim_states))
        self._a_t_array = np.zeros((self._buffer_size, 1))
        self._r_t_array = np.zeros((self._buffer_size, 1))
        self._s_t1_array = np.zeros((self._buffer_size, dim_states))
        self._term_t_array = np.zeros((self._buffer_size, 1))

```

Listing 4: Clase ReplayBuffer

2.2. Para completar el método `store_transition` simplemente se añaden los elementos a los buffer según `_buffer_idx`, se aumenta en 1 la cantidad de elementos almacenados y `_buffer_idx` se aumenta de modo que se reinicie cuando alcance el tamaño máximo del buffer:

```
    def store_transition(self, s_t, a_t, r_t, s_t1, done_t):
        # Add transition to replay buffers
        self._s_t_array[self._buffer_idx] = s_t
        self._a_t_array[self._buffer_idx] = a_t
        self._r_t_array[self._buffer_idx] = r_t
        self._s_t1_array[self._buffer_idx] = s_t1
        self._term_t_array[self._buffer_idx] = done_t

        # Update replay buffer index
        self._buffer_idx = (self._buffer_idx + 1) % self._buffer_size
        self._exps_stored += 1

```

Listing 5: store_transition

2.3. Para completar el método `sample_transitions` se genera una lista de índices aleatorios, con valores desde 0 hasta `_buffer_idx` si el *buffer* no está lleno, o con valores hasta `_buffer_size` en caso contrario. Luego simplemente se devuelven los elementos de cada *buffer* correspondientes a los índices aleatorios:

```
    def sample_transitions(self):
        assert self._exps_stored + 1 > self._sample_size
        if self._exps_stored < self._sample_size:
            sample_ids = random.sample(range(0, self._buffer_idx),
                                       self._sample_size)

        else:
            sample_ids = random.sample(range(0, self._buffer_size),
                                       self._sample_size)

        return (self._s_t_array[sample_ids],
                self._a_t_array[sample_ids].astype(int),
                self._r_t_array[sample_ids],
                self._s_t1_array[sample_ids],
                self._term_t_array[sample_ids])

```

Listing 6: sample_transition

2.4. Para probar la efectividad del `ReplayBuffer` se usa el mismo ambiente 'CartPole-v1' con un *buffer* de tamaño 20 y muestras de tamaño 10. Usando la herramienta *debugger* de cualquier IDE, se pueden analizar las variables cuando se ejecuta el código. Con esto se puede apreciar que el `ReplayBuffer` efectivamente almacena la primera transición, en *arrays* de largo 20, como se ve en al **Figura 5**:

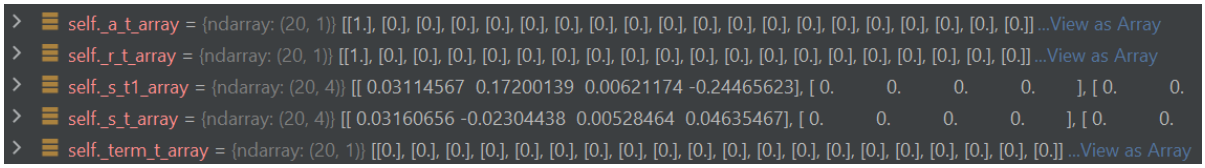

Figura 5: `ReplayBuffer` almacenando primera transición

Luego, cuando se ejecuta la línea:

```
    st, a, r, st1, end = self.replay_buffer.sample_transitions()
```

Se puede ver que efectivamente el `ReplayBuffer` devuelve muestras como *arrays* de largo 10, como se ve en la **Figura 6**:

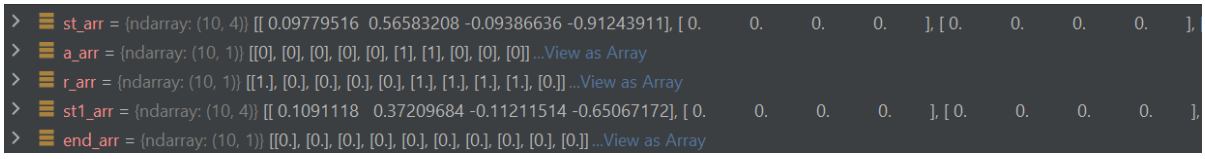

Figura 6: `ReplayBuffer` muestreando décima transición

En el caso en el que el *buffer* está vacío o tiene menos elementos que `_sample_size` se puede ver que sólo se pueden almacenar transiciones pero no muestrear, pues para obtener la **Figura 6** se tuvieron que correr 10 iteraciones primero. Por otra parte, si `_sample_size` es mayor o igual al tamaño máximo del *buffer*, el script no corre gracias al *assert* definido en `sample_transitions`.

3. Continuando con `deep_qnetwork.py`:

   3.1. Para completar el método replace_transition_network basta con simplemente cargar los pesos de la red `deep_QNetwork` hacia la *target network*:

```
    def replace_target_network(self):
        self._target_deepQ_network.load_state_dict(self._deep_QNetwork.
                                                   state_dict())

```

Listing 7: replace_target_network

   3.2. Por último, para completar el método `update` se sigue una secuencia de pasos. Primero se muestrean los *batches* de datos y se transforman a tensores con sus respectivos *dtypes*:

```
    def update(self):
        # get batches
        st_arr, a_arr, r_arr, st1_arr, end_arr = self.replay_buffer.
                                                  sample_transitions()

        # make tensors from batches
        st_arr = torch.from_numpy(st_arr).float()
        a_arr = torch.from_numpy(a_arr).type(torch.int64)
        r_arr = torch.from_numpy(r_arr).float()
        st1_arr = torch.from_numpy(st1_arr).float()
        end_arr = torch.from_numpy(end_arr).bool()

```

Luego, se calculan las salidas esperadas con el método auxiliar `compute_y`, el cual computa los *Q-values* máximos según la *target network* y luego llena el tensor y_j con r_j si end_j es True (estado terminal) y con r_j más los *Q-values* calculados anteriormente si end_j es False:

```
    y_t = self.compute_y(r_arr, st1_arr, end_arr)
                |
                v
    def compute_y(self, r, st1, end):
        with torch.no_grad():
            QValues = torch.amax(self._target_deepQ_network(st1), dim=1)
            y_j = torch.where(end.flatten(), r.flatten(),
                              torch.add(end.flatten(), QValues,
                              alpha=self._gamma))
        return y_j

```

Se llevan los gradientes a cero y se calculan los *Q-values* de la red `deep_QNetwork` con el método auxiliar `compute_QValues`. El método simplemente propaga los estados s_t por la red `deep_QNetwork` y por cada salida se escoge el $Q - value$ de la acción a_t perteneciente al mismo episodio que s_t:

```
    self._optimizer.zero_grad(set_to_none=True)
    Q_values = self.compute_QValues(st_arr, a_arr)
                    |
                    v
    def compute_QValues(self, st, a):
        QValues = self._deep_QNetwork(st)
        return torch.gather(QValues, dim=1, index=a).flatten()

```

Por último, se calcula la *loss*, se propaaga con *backpropagation*, se limitan los gradientes a (-1, 1) y se realiza un paso de optimización:

```
    loss = self._criterion(Q_values, y_t)
    loss.backward()
    torch.nn.utils.clip_grad_norm_(self._deep_QNetwork.parameters(), 1)
    self._optimizer.step()

```

4. Se fijan los parámetros en `nb_training_steps:20000`, `gamma:0.99`, `epsilon:0.8` y `lr:0.01`. Cabe destacar que en estos experimentos de manera adicional se almacenan los pesos de la red según si superan la máxima recompensa promedio obtenida y dichos pesos se cargan en la última evaluación del agente.

   4.1. A continuación se presentan los resultados del **Experimento 1**, donde se analiza el efecto de variar el tamaño máximo del *buffer*:

   exp_11 `(1000, 16, 100)`
   Al ejecutar el exp_11 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 7 y 8**:

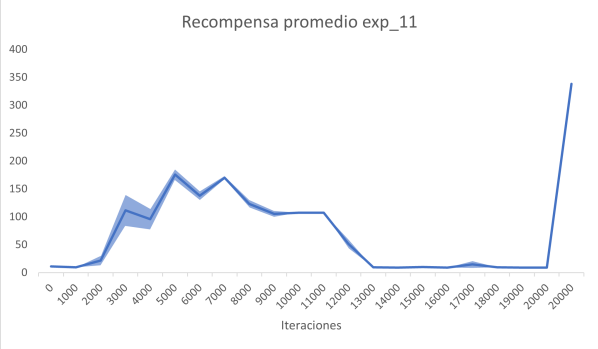

Figura 7: Recompensa promedio exp_11

Figura 8: Tasa de éxito exp_11

   exp_12 `(2500, 16, 100)`
   Al ejecutar el exp_12 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 9 y 10**:

Figura 9: Recompensa promedio exp_12

Figura 10: Tasa de éxito exp_12

exp_13 `(5000, 16, 100)`

Al ejecutar el exp_13 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 11 y 12**:

Figura 11: Recompensa promedio exp_13

Figura 12: Tasa de éxito exp_13

A continuación se presentan los resultados del **Experimento 2**, donde se analiza el efecto de variar el tamaño de las muestras:

exp_21 `(5000, 32, 100)`

Al ejecutar el exp_21 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 13 y 14**:

Figura 13: Recompensa promedio exp_21

Figura 14: Tasa de éxito exp_21

exp_22 `(5000, 64, 100)`

Al ejecutar el exp_22 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 15 y 16**:

Figura 15: Recompensa promedio exp_22

Figura 16: Tasa de éxito exp_22

exp_23 `(5000, 128, 100)`

Al ejecutar el exp_23 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 17 y 18**:

Figura 17: Recompensa promedio exp_23

Figura 18: Tasa de éxito exp_23

A continuación se presentan los resultados del **Experimento 3**, donde se analiza el efecto de variar la cantidad de iteraciones que se espera para actualizar la *target network*:

exp_31 `(5000, 128, 1)`

Al ejecutar el exp_31 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 19 y 20**:

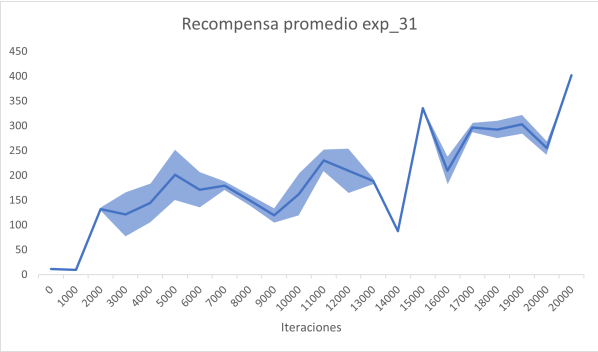

Figura 19: Recompensa promedio exp_31

Figura 20: Tasa de éxito exp_31

exp_32 `(5000, 128, 100)`

Dados los parámetros escogidos el exp_32 es el mismo experimento que el exp_23, por lo que sus resultados y por ende su análisis son iguales. Es decir, para este experimento se deben considerar la recompensa promedio y la tasa de éxito presentados en las **Figuras 17 y 18**.

exp_33 `(5000, 128, 1000)`

Al ejecutar el exp_33 cinco veces y promediando los resultados se obtienen la recompensa promedio y la tasa de éxito promedio presentados en las **Figuras 21 y 22**:

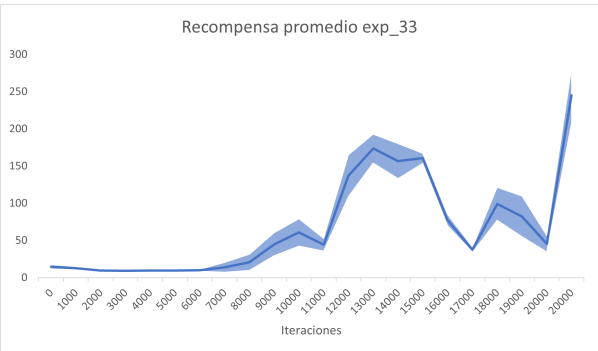

Figura 21: Recompensa promedio exp_33

Figura 22: Tasa de éxito exp_33

4.2. A continuación se presenta el análisis de los experimentos recién mostrados.

El **Experimento 1** varía el tamaño máximo del *buffer* desde 1000 hasta 5000. Una idea central detrás de usar un *replay buffer* es evitar la correlación entre las muestras que le llegan a la red, para así mejorar su desempeño. Por ende, *buffers* más grandes promueven las chances de escoger muestras no correlacionadas, por lo que el aprendizaje de la red debiese ser más estable.

Si bien los resultados presentados en las **Figuras 7, 9 y 11** no logran los 500 puntos de recompensa, sí se condicen con lo recién descrito en el párrafo anterior. En la **Figura 7** el agente aprende un poco pero tiene una caída a las 12 mil iteraciones de la cual no se recupera. En la **Figura 9** el agente tiene una pérdida catastrófica también cerca de las 13 mil iteraciones. En cambio, en la **Figura 11** el agente obtiene recompensas, si bien no muy grandes, más estables. Estas bajas recompensas probablemente se deben a que el tamaño de las muestras es muy pequeño.

El **Experimento 2** varía el tamaño de las muestras desde 32 hasta 128. El tamaño de las muestras va de la mano con la idea del *replay buffer*. Muestras más grandes permiten que el agente se entrene con más episodios en menos tiempo, lo cual debiese acelerar su rendimiento. Sin embargo, tomar muchas muestras (casi tantas como el máximo) aumenta las chances de que estas por azar se encuentren correlacionadas.

Los resultados presentados en las **Figuras 13, 15 y 17** también se condicen con lo discutido anteriormente. Gracias al tamaño de su *buffer* no tienen muchas pérdidas catastróficas y se puede apreciar que a medida que aumenta el tamaño de las muestras los resultados también mejoran, obteniendo los mejores en el exp_23 (**Figura 17**).

Por último, el **Experimento 3** varía la cantidad de iteraciones que se espera para actualizar los pesos de la *target network*. El propósito de incluir la *target network* es estabilizar el entrenamiento del agente. Si se actualiza la *target network* muy seguido, el objetivo hacia el cual se entrena la red se mueve mucho y por ende el entrenamiento es inestable. En el otro extremo, si la *target network* se actualiza muy poco la red nunca va a aprender y se quedará estancada con malos resultados. Por ende, hay que encontrar un "punto dulce" para la actualización de la *target network*, no muy bajo pero tampoco no muy grande.

Los resultados persentados en las **Figuras 19, 17 y 21** también se condicen con lo explicado en el párrafo anterior. En la **Figura 19** la red se actualiza en cada iteración, sorprendentemente obtiene buenos resultados, pero se puede apreciar que tiene bastantes cambios bruscos y que hay mucha varianza de un experimento a otro (mostrado por las sombras de error). En la **Figura 17** la red se actualiza cada 100 iteraciones, los resultados no tienen mucha varianza y aumentan consistentemente. En la **Figura 21** el agente se estanca y no logra aprender, llegando a una máxima tasa de éxito de 0.25 (**Figura 22**). Por ende, el "punto dulce" se encuentra con reemplazar la *target network* cada 100 iteraciones, siendo ese el mejor modelo para este problema.