

Introducción al Aprendizaje por Refuerzos

Métodos de Solución Aproximada

Métodos de Solución Aproximada

- Extensión de los métodos tabulares para aplicarlos a problemas con espacios de estados arbitrariamente grandes (enormes?).
- Ej. El número de imágenes posibles de una cámara.
- En tales casos, no buscamos una política óptima o una función de valor óptima, sino que generamos **buenas soluciones aproximadas** empleando recursos computacionales limitados.

Generalización en RL

- El problema de los espacios de estados de gran tamaño no es solo la memoria necesaria para almacenar las tablas de valor, sino el tiempo y los datos necesarios para llenarlos de manera correcta.
- En muchos problemas reales casi todos los estados visitados no han sido vistos antes por el agente, por lo que para tomar decisiones en tales estados es necesario llevar a cabo un proceso de **generalización** respecto de la información presente en estados considerados “similares”.
- **Pregunta:** *Cómo se puede generalizar de manera útil la experiencia en base a un número limitado de ejemplos del espacio de estados para producir buenas aproximaciones sobre un set de estados mucho mayor?*
- Integrar reinforcement learning con métodos de generalización preexistentes.
- Generalización = *function approximation*
= tomar ejemplos de una función y generalizar a partir de ellos para obtener una representación de la función completa. (e.g., función de valor V o Q) .
- La **aproximación funcional** es una instancia del **aprendizaje supervisado** (artificial neural networks, pattern recognition, and statistical curve fitting). En teoría, cualquiera de los métodos estudiados en dichas áreas podrían emplearse con algoritmos reinforcement learning, aunque en la práctica algunos de ellos son mejores que otros.

Aproximación Funcional en RL

- Uso en la estimación de V , Q o π .
- Las funciones mencionadas no se representan como tablas sino en forma de funciones parametrizadas con un vector de pesos $\mathbf{w} \in \mathbb{R}^d$

$$\hat{v}(\mathbf{s}, \mathbf{w}) \approx v_{\pi}(\mathbf{s}) \longrightarrow \text{Valor aproximado del estado } \mathbf{s} \text{ dado el vector de pesos } \mathbf{w}$$

- Por ejemplo, \hat{v} podría ser una función lineal en los *features* del estado, con \mathbf{w} como vector de pesos de los *features*.
- De manera más general, \hat{v} podría ser la función computada por una red neuronal multicapa, con \mathbf{w} representando el vector de pesos de conexiones entre neuronas en todas las capas. Por medio del ajuste de los pesos, un amplio rango de funciones puede ser implementado por la red.
- \hat{v} podría ser la función computada por un árbol de decisión, donde \mathbf{w} consiste en los valores que definen las divisiones en las ramas y los valores de las hojas. Normalmente, como la cantidad de pesos (dimensionalidad de \mathbf{w}) es menor que la cantidad de estados, cambiar un peso cambia la estimación de valor de muchos estados.
- Como consecuencia, cuando un estado se actualiza, el cambio se **generaliza** desde ese estado para afectar los valores de muchos otros estados.

Stochastic gradient-descent (SGD)

- El vector de pesos es un vector columna con un número fijo de components reales. $\mathbf{w} = (w_1, w_2, \dots, w_d)$ y la función de valor aproximada $\hat{v}(s, \mathbf{w})$ es una función diferenciable de \mathbf{w} para todo $s \in \mathcal{S}$.
- SGD actualiza \mathbf{w} en cada uno de los pasos $t = 0, 1, 2, 3, \dots, n$ de interacción intentando minimizar el error de predicción respecto de los ejemplos provenientes de la experimentación con el entorno.
- SGD realiza este proceso ajustando el vector de pesos después de la generación de cada ejemplo correcto en una pequeña cantidad en la dirección que más reduciría el error en dicho ejemplo.

Step-size

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2} \alpha \nabla \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right]^2 \\ &= \mathbf{w}_t + \alpha \left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t) \right] \nabla \hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

El vector de derivativas es el *gradiente* de f respecto de \mathbf{w} $\nabla f(\mathbf{w}) \doteq \left(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \dots, \frac{\partial f(\mathbf{w})}{\partial w_d} \right)^\top$

SGD

- Los métodos basados en SGD emplean el “gradiente descendente” porque el paso de actualización de \mathbf{w}_t es *proporcional al gradiente negativo del error*.

```
def eval_numerical_gradient(f, x):
    """
    a naive implementation of numerical gradient of f at x
    - f should be a function that takes a single argument
    - x is the point (numpy array) to evaluate the gradient at
    """

    fx = f(x) # evaluate function value at original point
    grad = np.zeros(x.shape)
    h = 0.00001

    # iterate over all indexes in x
    it = np.nditer(x, flags=['multi_index'], op_flags=['readwrite'])
    while not it.finished:

        # evaluate function at x+h
        ix = it.multi_index
        old_value = x[ix]
        x[ix] = old_value + h # increment by h
        fxh = f(x) # evaluate f(x + h)
        x[ix] = old_value # restore to previous value (very important!)

        # compute the partial derivative
        grad[ix] = (fxh - fx) / h # the slope
        it.iternext() # step to next dimension

    return grad
```

```
loss_original = CIFAR10_loss_fun(W) # the original loss
print 'original loss: %f' % (loss_original, )

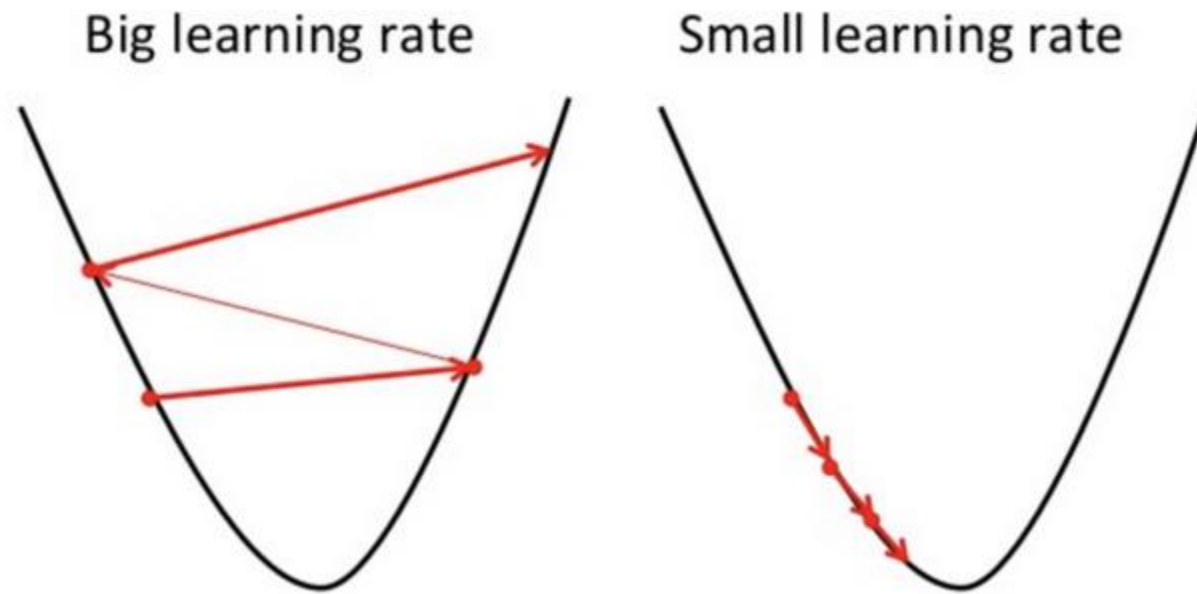
# lets see the effect of multiple step sizes
for step_size_log in [-10, -9, -8, -7, -6, -5, -4, -3, -2, -1]:
    step_size = 10 ** step_size_log
    W_new = W - step_size * df # new position in the weight space
    loss_new = CIFAR10_loss_fun(W_new)
    print 'for step size %f new loss: %f' % (step_size, loss_new)
```

Son estocásticos porque la actualización se lleva a cabo sobre un solo ejemplo a la vez, que puede ser seleccionado estocásticamente.

<http://ruder.io/optimizing-gradient-descent/>

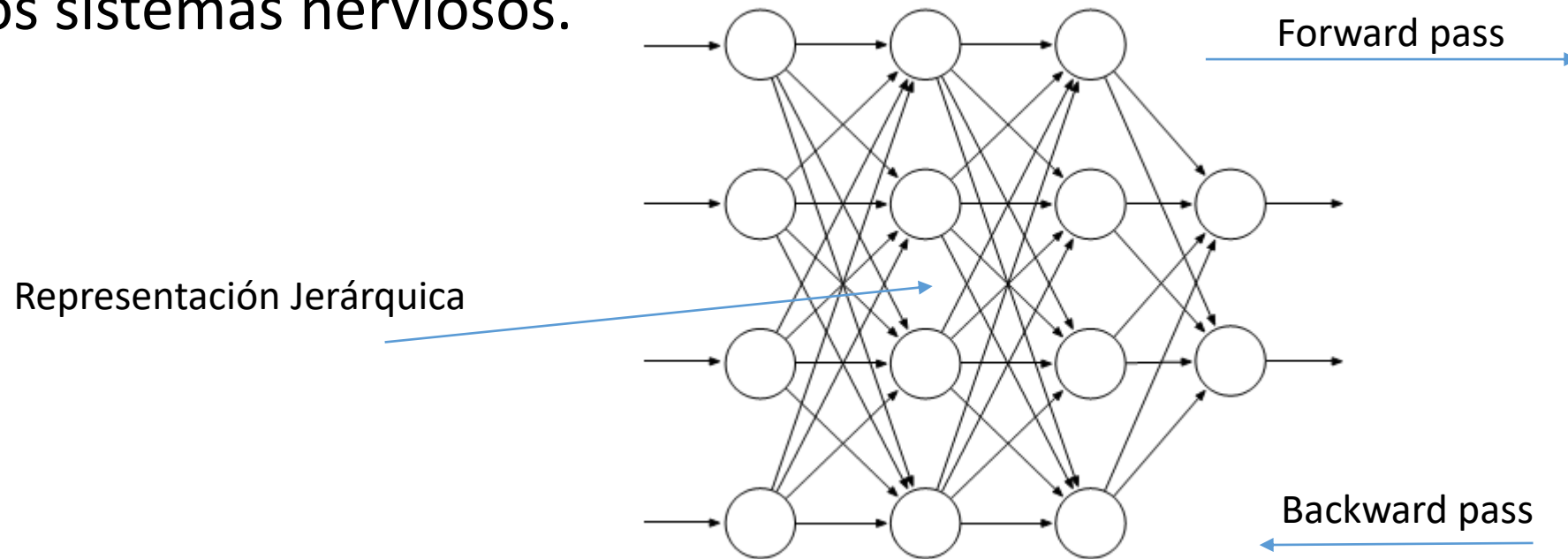
Variantes: Batch/Minibatch

Learning rate o Step Size



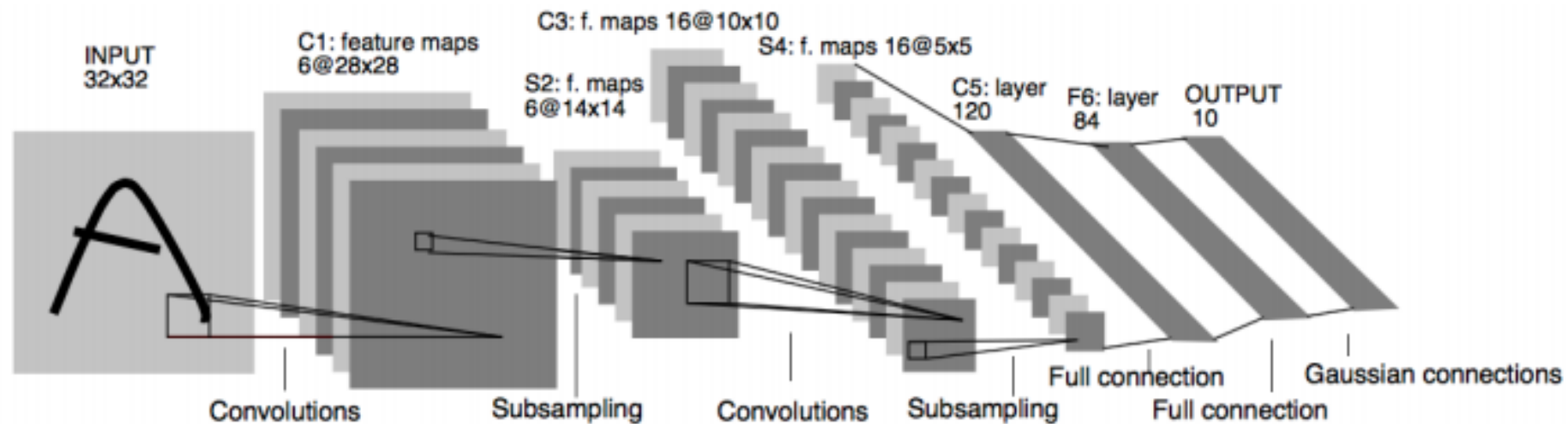
Nonlinear Function Approximation: Artificial Neural Networks

- Las ANNs son ampliamente utilizadas para aproximar funciones no lineales.
- Una ANN es una red de unidades interconectadas que tienen algunas de las “propiedades” de las neuronas, los principales components de los sistemas nerviosos.



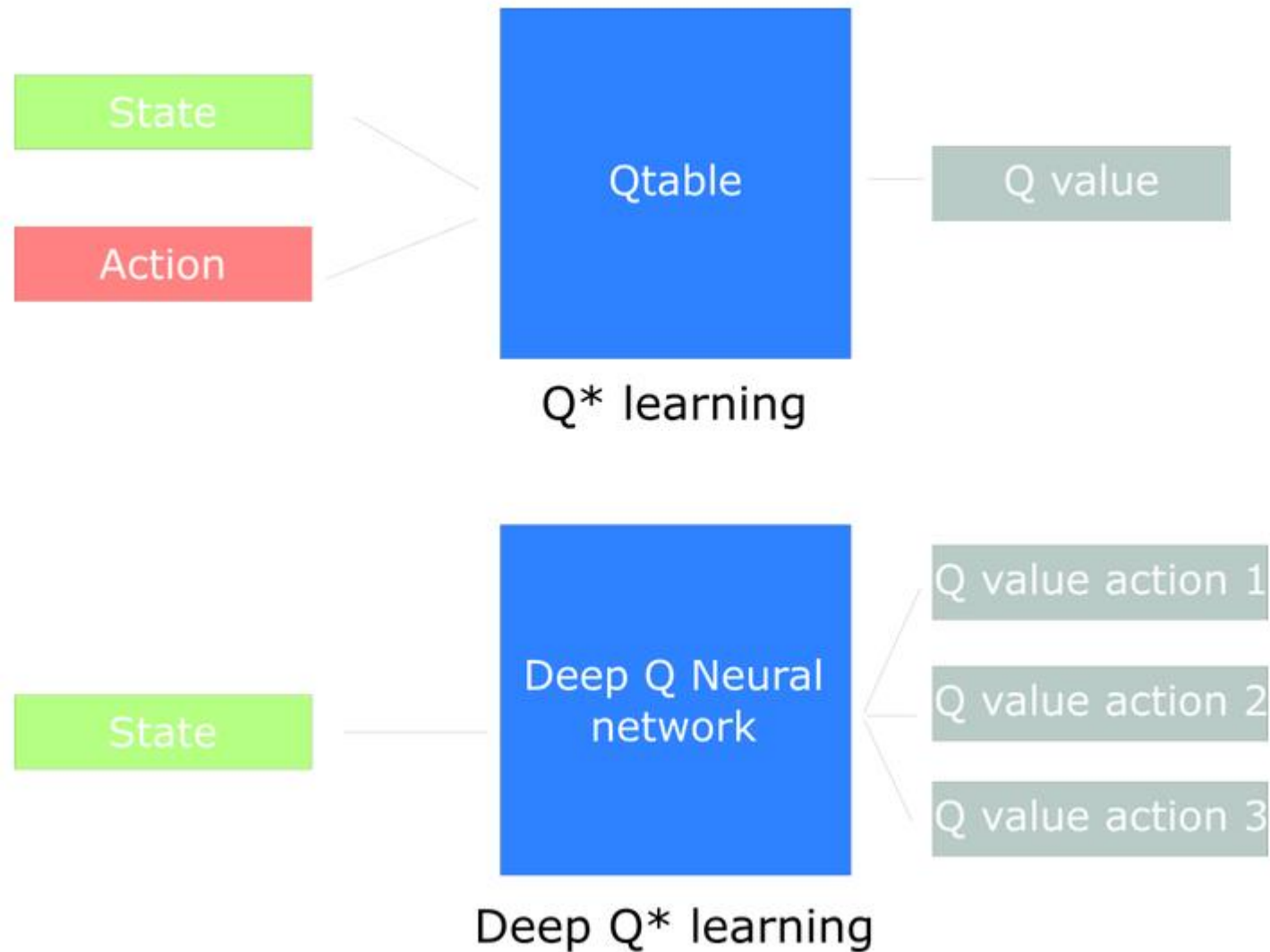
Convolutional NNs

- Un tipo de ANN empleado en muchas aplicaciones exitosas de RL.
- Específicas para el procesamiento de datos de alta dimensionalidad estructurados en arrays, tales como imágenes.



Deep Q-Network

Deep Q-networks



Deep Q-Learning

- Bellman Update:

$$NewQ(s, a) = Q(s, a) + \alpha [R(s, a) + \gamma \max Q'(s', a') - Q(s, a)]$$

New Q value for that state and that action

Current Q value

Learning Rate

Reward for taking that action at that state

Discount rate

Maximum expected future reward given the new s' and all possible actions at that new state

$$\Delta w = \alpha [(R + \gamma \max_a \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)] \nabla_w \hat{Q}(s, a, w)$$

Change in weights

learning rate

Maximum possible Qvalue for the next_state (= Q_target)

Current predicted Q-val

TD Error

Gradient of our current predicted Q-value

Deep Q-Learning

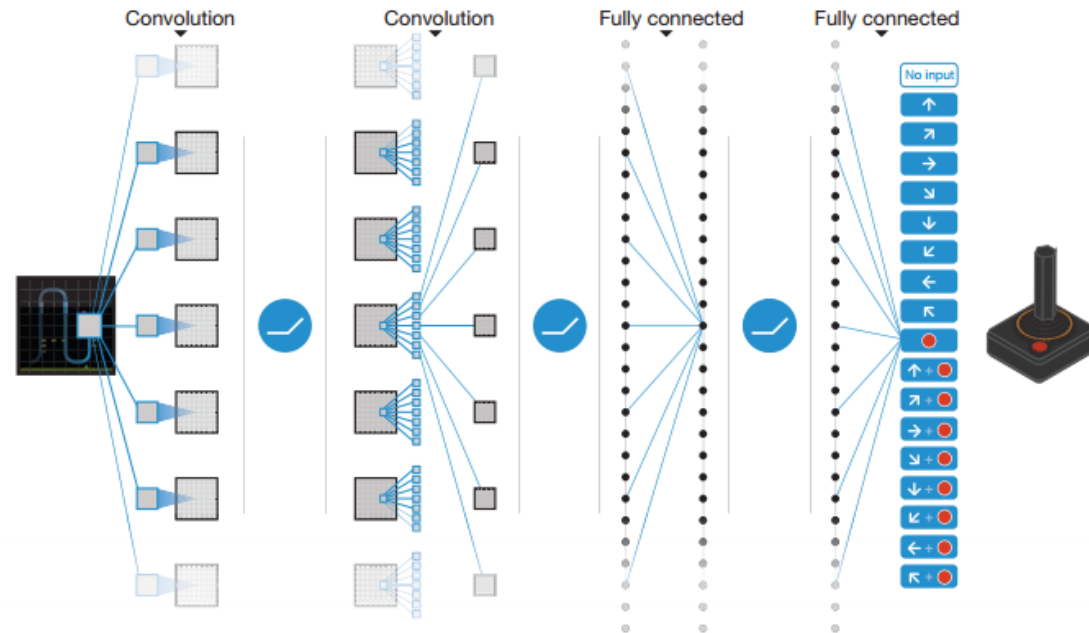
```
Initialize Environment E
Initialize replay Memory M with capacity N (= finite capacity)
Initialize the DQN weights w
for episode in max_episode:
    s = Environment state
    for steps in max_steps:
        Choose action a from state s using epsilon greedy.
        Take action a, get r (reward) and s' (next state)
        Store experience tuple <s, a, r, s'> in M
        s = s' (state = new_state)

    Get random minibatch of exp tuples from M
    Set Q_target = reward(s,a) +  $\gamma$ maxQ(s')
    Update w =  $\alpha(Q\_target - Q\_value) * \nabla_w Q\_value$ 
```

Dos procesos ocurren en este algoritmo:

- Sampleamos el entorno donde ejecutamos acciones y almacenamos las experiencias observadas en una “replay memory”.
- Seleccionamos aleatoriamente un pequeño batch de tuplas de experiencia y llevamos a cabo el aprendizaje (i.e. actualizamos la función de valor) empleando gradiente descendente.

Human-level control through Deep-RL (Mnih et al. 2015)



Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

Initialize target action-value function \hat{Q} with weights $\theta^- = \theta$

For episode = 1, M **do**

 Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$

For $t = 1, T$ **do**

 With probability ϵ select a random action a_t

 otherwise select $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

 Execute action a_t in emulator and observe reward r_t and image x_{t+1}

 Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$

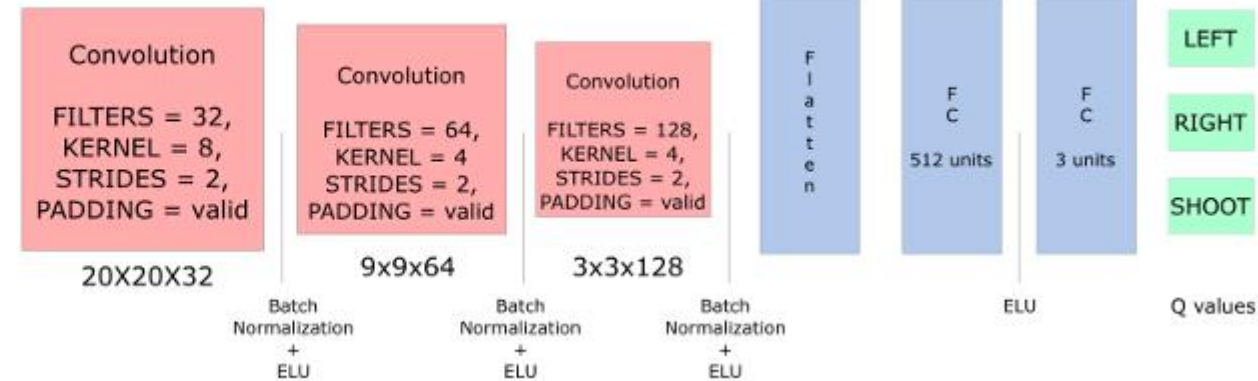
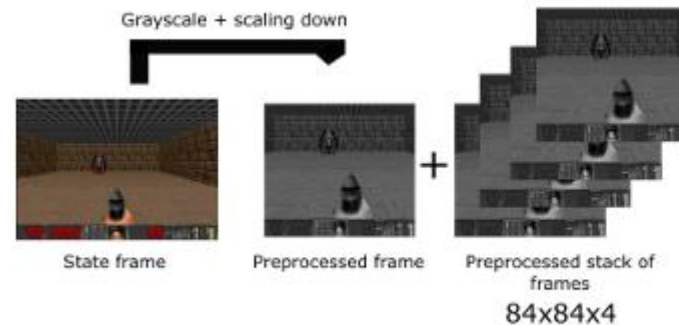
 Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in D

 Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from D

 Set $y_j = \begin{cases} r_j & \text{if episode terminates at step } j+1 \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$

 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

 Every C steps reset $\hat{Q} = Q$



Mejoras al Deep Q-Learning

$$\Delta w = \alpha [(\underbrace{R}_{\text{Reward of taking that action at that state}} + \underbrace{\gamma \max_a \hat{Q}(s', a, w)}_{\text{Maximum possible Qvalue for the next_state (= Q_target)}} - \underbrace{\hat{Q}(s, a, w)}_{\text{Current predicted Q-val}}] \nabla_w \hat{Q}(s, a, w)$$

Change in weights

learning rate

TD Error

Es una estimación!!!

El mismo conjunto de parámetros

Gradient of our current predicted Q-value

$$\underbrace{Q(s, a)}_{\text{Q target}} = \underbrace{r(s, a)}_{\text{Reward of taking that action at that state}} + \underbrace{\gamma \max_a Q(s', a)}_{\text{Discounted max q value among all. possibles actions from next state.}}$$

Mejoras al Deep Q-Learning

$$\underbrace{\Delta w}_{\text{Change in weights}} = \underbrace{\alpha}_{\text{learning rate}} \left[\underbrace{(R + \gamma \max_a \hat{Q}(s', a, \vec{w}))}_{\text{Maximum possible Qvalue for the next_state (= Q_target)}} - \underbrace{\hat{Q}(s, a, \vec{w})}_{\text{Current predicted Q-val}} \right] \underbrace{\nabla_w \hat{Q}(s, a, w)}_{\text{Gradient of our current predicted Q-value}}$$

Double DQNs

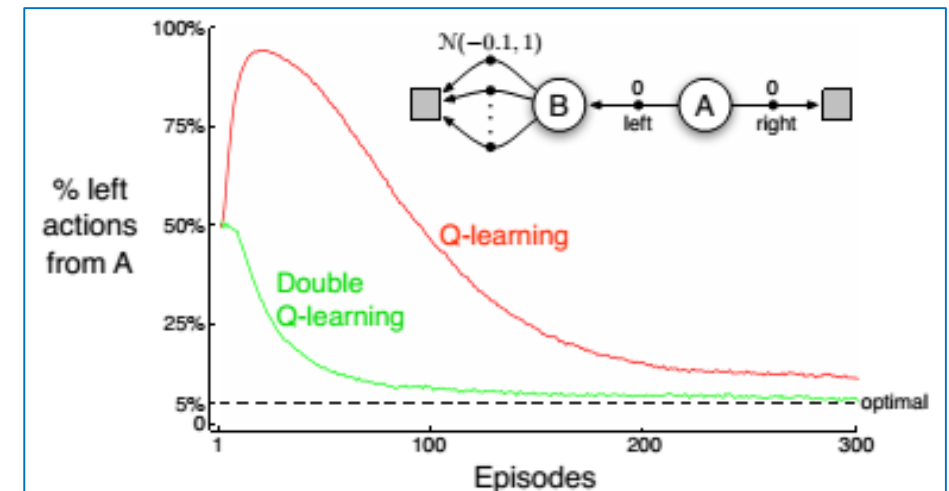
- Trata con el problema de **sobreestimar** algunos valores de Q. (Hado van Hasselt et al.)
- TD target: Cómo asegurar que la **mejor acción** para el estado siguiente es la acción con el **mayor valor Q**?
- La exactitud de los valores Q depende de que acciones se han ejecutado y que estados vecinos hemos explorado.
- Al inicio del entrenamiento **no tenemos suficiente información** respecto de qué acciones elegir, por lo que tomar los mejores valores de Q puede llevar a “falsos positivos” y a **demorar la convergencia a largo plazo**.

Algorithm 1 Double Q-learning

```

1: Initialize  $Q^A, Q^B, s$ 
2: repeat
3:   Choose  $a$ , based on  $Q^A(s, \cdot)$  and  $Q^B(s, \cdot)$ , observe  $r, s'$ 
4:   Choose (e.g. random) either UPDATE(A) or UPDATE(B)
5:   if UPDATE(A) then
6:     Define  $a^* = \arg \max_a Q^A(s', a)$ 
7:      $Q^A(s, a) \leftarrow Q^A(s, a) + \alpha(s, a) (r + \gamma Q^B(s', a^*) - Q^A(s, a))$ 
8:   else if UPDATE(B) then
9:     Define  $b^* = \arg \max_a Q^B(s', a)$ 
10:     $Q^B(s, a) \leftarrow Q^B(s, a) + \alpha(s, a) (r + \gamma Q^A(s', b^*) - Q^B(s, a))$ 
11:   end if
12:    $s \leftarrow s'$ 
13: until end

```



Double DQNs

- Solución: Cuando calculamos el Q target, empleamos dos redes para desacoplar la selección de acciones de la generación del Q target.
- La DQN Network selecciona cual es la major acción en el siguiente estado (acción con mayor Q).
- La Target Network calcula el Q target correspondiente a tomar esa acción en el estado siguiente.

$$\underbrace{Q(s, a)}_{\text{TD target}} = r(s, a) + \gamma \underbrace{Q(s', \argmax_a Q(s', a))}_{\substack{\text{DQN Network choose} \\ \text{action for next state}}}$$

Target network calculates the Q value of taking that action at that state

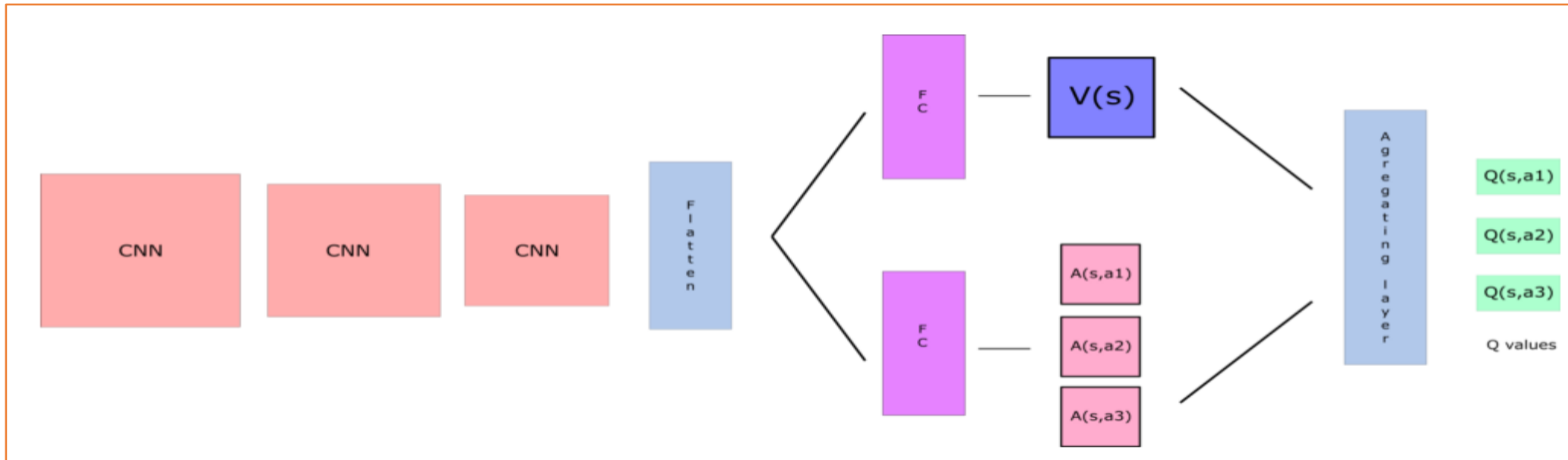
Dueling DQN

- **Q-values:** cuan bueno es estar en el estado s y ejecutar la acción a . $Q(s,a)$.
- $Q(s,a)$ puede descomponerse como la suma de:
- **$V(s)$:** el valor de estar en el estado s .
- **$A(s,a)$:** la ventaja de elegir la acción a en dicho estado, o cuan mejor es seleccionar esa acción respecto de todas las posibles.

$$Q(s, a) = A(s, a) + V(s)$$

Dueling DQN

- Con DDQN, separamos el estimador de los mencionados elementos empleando dos streams:
- Uno que estima el valor del estado $V(s)$
- Uno que estima la ventaja para cada acción $A(s,a)$



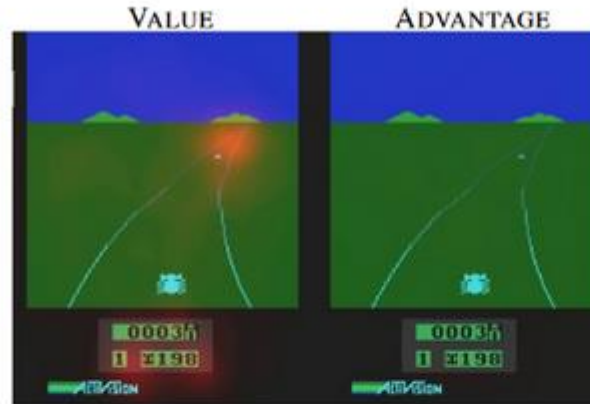
Dueling DQN

- Posteriormente se combinan los *streams* en una capa de agregación para obtener la estimación de $Q(s,a)$.
- Por qué calculamos los valores separadamente y luego los combinamos?
- Desacoplando la estimación la DDQN puede aprender cuáles estados son (o no) valorables sin tener que aprender el efecto de cada acción en cada estado (**ya que se calcula $V(s)$ por separado!**).
- Con una DQN normal necesitamos calcular el valor de cada acción en cada estado. Si el estado es de por sí “malo” calcular el valor de las acciones no aporta al aprendizaje.
- Al calcular $V(s)$, no es necesario calcular el valor de cada acción lo que es particularmente útil en aquellos estados en que no es relevante la acción que se toma.

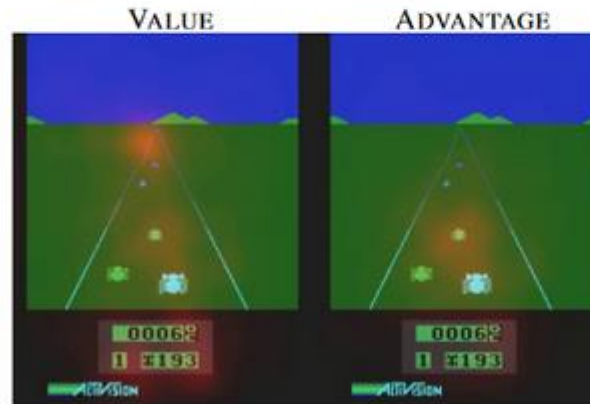
Dueling DQN

Focus on 2 things:

- The horizon where new cars appear
- On the score



No car in front,
does not pay
much attention
because action
choice making is
not relevant



Pays attention to
the front car, in this
case **choice**
making is crucial
to survive

Dueling DQN

$$Q(s, a; \theta, \alpha, \beta) = V(s; \theta, \beta) + (A(s, a; \theta, \alpha) - \underbrace{\frac{1}{\mathcal{A}} \sum_{a'} A(s, a'; \theta, \alpha)}_{\text{Average advantage}})$$

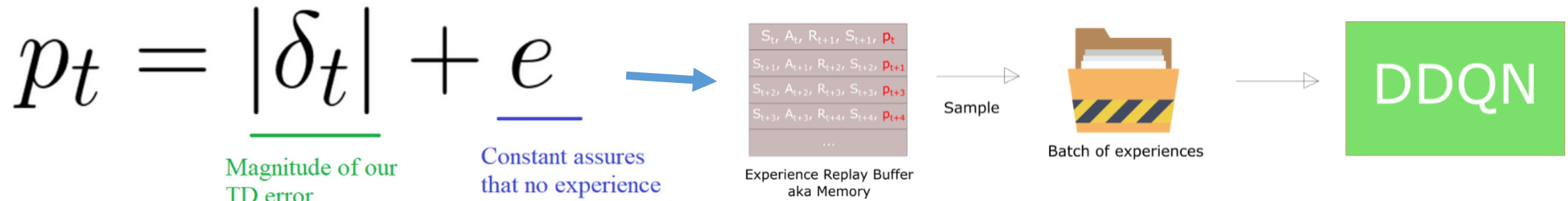
Diagram illustrating the Dueling DQN architecture components:

- Common network parameters:** Indicated by a purple line pointing to θ .
- Advantage stream parameters:** Indicated by a green line pointing to α .
- Value stream parameters:** Indicated by a blue line pointing to β .

Prioritized Experience Replay

- La idea central es que algunas experiencias pueden ser más importantes que otras para el entrenamiento, pero podrían ocurrir menos frecuentemente.
- Como el **batch** de experiencia es sampleado uniformemente las experiencias importantes que ocurren raramente casi no tienen chances de ser elegidas.
- Con PER, cambiamos la distribución del sampleo empleando un criterio para definir la prioridad de cada tupla de experiencia.
- Le asignamos más prioridad a aquellas experiencias en donde existe una gran diferencia entre la predicción y el **TD target, dado que esto implica que tenemos mucho para aprender acerca de dicha experiencia.**
- Para ello empleamos el valor absoluto del error TD

Prioritized Experience Replay



$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

Priority value

Hyperparameter used to reintroduce some randomness in the experience selection for the replay buffer

If $a = 0$ pure uniform randomness

If $a = 1$ only select the experiences with the highest priorities

Normalized by all priority values in Replay Buffer

Prioritized Experience Replay

- Problema: con el normal Experience Replay, usamos una regla de actualización estocástica.
- **Empleando un sampleo basado en prioridades, introducimos un sesgo en favor de los ejemplos con alta prioridad (más chances de ser elegidos), corriendo el riesgo de producir overfitting.**
- Para corregir el bias, empleamos Importance Sample Weights que ajustan la actualización reduciendo los pesos de los ejemplos vistos a menudo.

Prioritized Experience Replay

$$\left(\frac{1}{\underline{N}} \cdot \frac{1}{\boxed{P(i)}} \right)^{\boxed{b}}$$

Replay Buffer
Size

Sampling
probability

Controls how much the IS w affect learning. Close to 0 at the beginning of learning and annealed up to 1 over the duration of training because **these weights are more important in the end of learning when our q values begin to converge**

- Los pesos correspondientes a los ejemplos de alta prioridad influyen menos en el ajuste de la función de acción-valor porque serán vistos muchas veces.
- Por el contrario, los ejemplos con baja prioridad tendrán un efecto mayor en la actualización.
- El hiper-parámetro **b** controla cuanto los ISW afectan el aprendizaje.

Double DQN with Proportional Prioritization

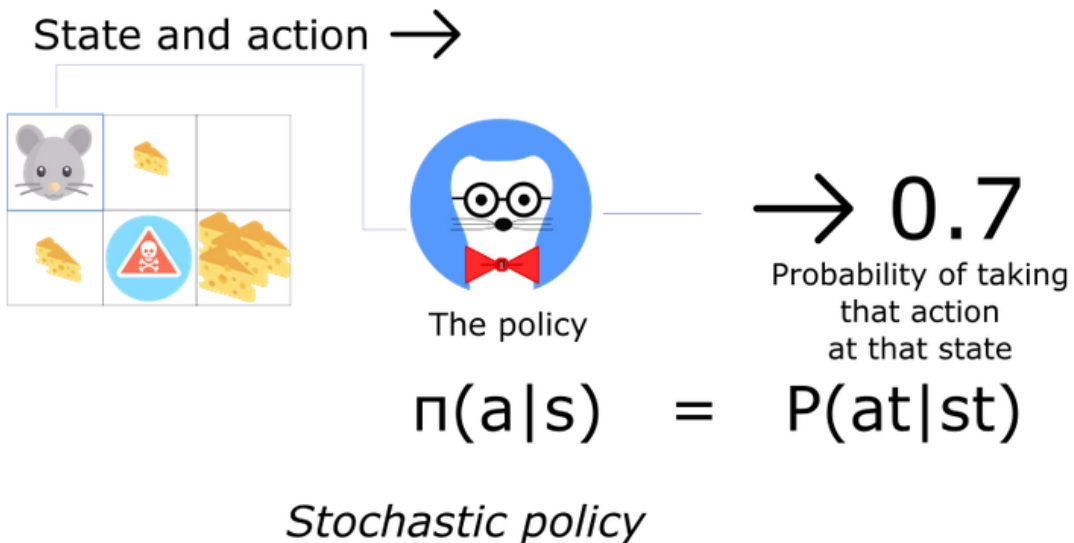
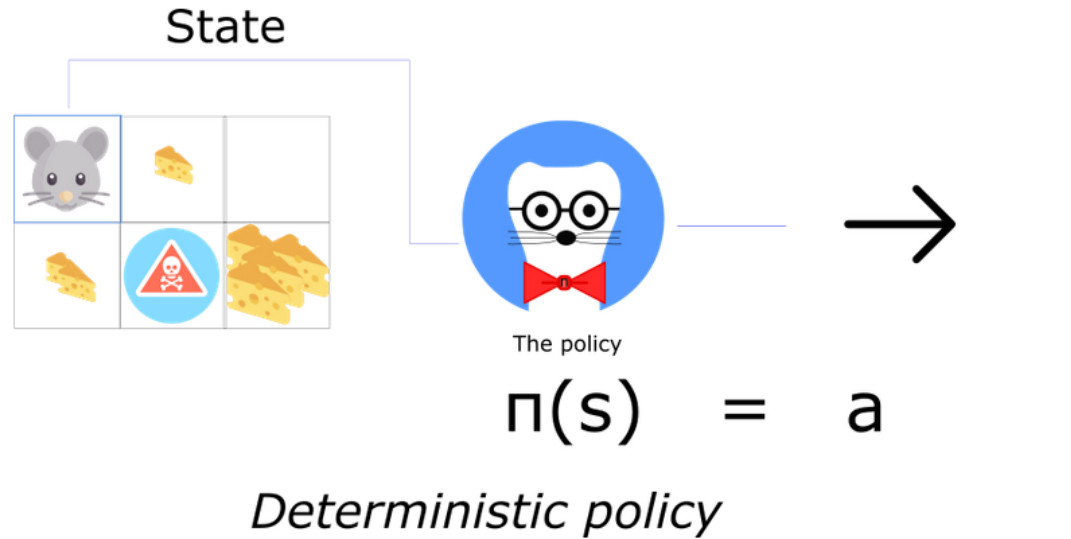
Algorithm 1 Double DQN with proportional prioritization

```

1: Input: minibatch  $k$ , step-size  $\eta$ , replay period  $K$  and size  $N$ , exponents  $\alpha$  and  $\beta$ , budget  $T$ .
2: Initialize replay memory  $\mathcal{H} = \emptyset$ ,  $\Delta = 0$ ,  $p_1 = 1$ 
3: Observe  $S_0$  and choose  $A_0 \sim \pi_\theta(S_0)$ 
4: for  $t = 1$  to  $T$  do
5:   Observe  $S_t, R_t, \gamma_t$ 
6:   Store transition  $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$  in  $\mathcal{H}$  with maximal priority  $p_t = \max_{i < t} p_i$ 
7:   if  $t \equiv 0 \pmod K$  then
8:     for  $j = 1$  to  $k$  do
9:       Sample transition  $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$ 
10:      Compute importance-sampling weight  $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$ 
11:      Compute TD-error  $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg \max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$ 
12:      Update transition priority  $p_j \leftarrow |\delta_j|$ 
13:      Accumulate weight-change  $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$ 
14:    end for
15:    Update weights  $\theta \leftarrow \theta + \eta \cdot \Delta$ , reset  $\Delta = 0$ 
16:    From time to time copy weights into target network  $\theta_{\text{target}} \leftarrow \theta$ 
17:  end if
18:  Choose action  $A_t \sim \pi_\theta(S_t)$ 
19: end for

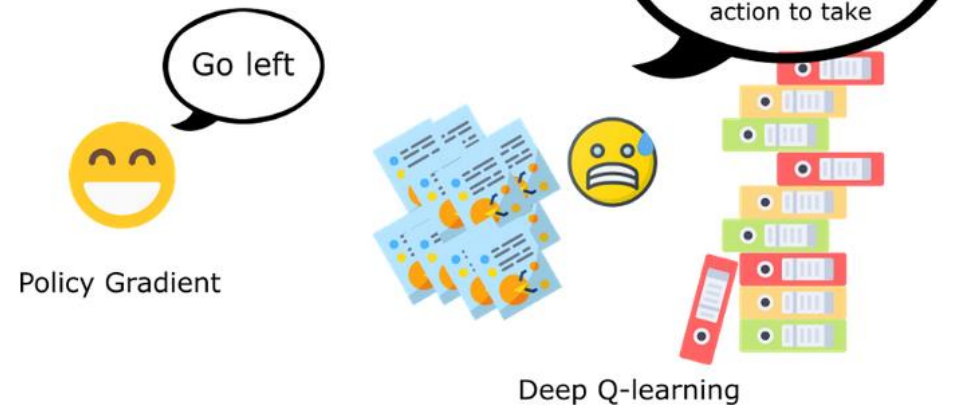
```

Policy Gradient Methods



Convergence

High dimensional action spaces



Policy gradients can learn stochastic policies

Idea Central

Función de Score de la Política

$$\pi_{\theta}(a|s) = P[a|s]$$

$$J(\theta) = E_{\pi_{\theta}}[\sum \gamma r]$$




Maximizar!

- Medir la calidad de π (policy) con una policy score function $J(\vartheta)$
- Utilizar ***policy gradient ascent*** para encontrar el θ que mejora π .

Policy Score Function

- En entornos episódicos, podemos utilizar el cálculo de la media del retorno desde el primer time step (G_1). Este valor sería la recompensa acumulada descontada para el episodio completo.

$$J_1(\theta) = E_{\pi}[G_1 = R_1 + \gamma R_2 + \gamma^2 R_3 + \dots] = E_{\pi}(V(s_1))$$



- Encontrar la política que maximiza G_1 (Óptima)

Policy Score Function

$$J_{avgv}(\theta) = E_{\pi}(V(s)) = \sum d(s)V(s)$$

where $d(s) = \frac{N(s)}{\sum_{s'} N(s')}$

Number of occurrences of the state

Total nb occurrences of all states

$$J_{avR}(\theta) = E_{\pi}(r) = \sum_s d(s) \sum_a \pi\theta(s, a) R_s^a$$

Probability that I'm in state s

Probability that I take this action a from that state under this policy

Immediate reward that I'll get

Policy gradient ascent

- Teniendo una Policy Score Function (cuan buena es la política), debemos encontrar un θ que maximiza dicha score function, porque eso implica encontrar la política óptima.
- Para maximizar $J(\vartheta)$, necesitamos aplicar el gradiente ascendente en los parámetros de la política.
- La idea es encontrar el gradient respecto de la política π que actualiza los parámetros en la dirección del mayor incremento de la función de score, y luego iterar.

Policy : π_θ

Objective function : $J(\theta)$

Gradient : $\nabla_\theta J(\theta)$

Update : $\theta \leftarrow \theta + \alpha \nabla_\theta J(\theta)$

$$\theta^* = \underset{\theta}{\operatorname{argmax}} \underbrace{E_{\pi_\theta} \left[\sum_t R(s_t, a_t) \right]}_{J(\theta)}$$

Policy Gradient Ascent

$$J(\theta) = \underbrace{E_{\pi}}_{\substack{\text{Expected} \\ \text{given} \\ \text{policy}}} \left[\underbrace{R(\tau)}_{\substack{\text{Expected future} \\ \text{reward}}} \right] \quad \begin{array}{l} s_0, a_0, r_0, \\ s_1, a_1, r_1 \dots \end{array}$$

$$J_1(\theta) = V_{\pi\theta}(s_1) = E_{\pi\theta}[v_1] = \underbrace{\sum_{s \in S} d(s)}_{\text{State distribution}} \underbrace{\sum_{a \in A} \pi_{\theta}(s, a) R_s^a}_{\text{Action distribution}}$$

Policy Gradient Ascent

- Problema: Como estimamos el ∇ (gradiente) respecto de θ , cuando el mismo depende del efecto desconocido proveniente del cambio de política?
- **Policy Gradient Theorem.**

Policy Gradient Ascent

$$J(\theta) = \underbrace{E_{\pi}}_{\text{Expected given policy}} [\underbrace{R(\tau)}_{\text{Expected future reward}}]$$

s0, a0, r0,
s1, a1, r1...

Expected
given
policy

Expected future
reward

$$\nabla_{\theta} J(\theta) = \nabla_{\theta} \sum_{\tau} \pi(\tau; \theta) R(\tau)$$

$$= \sum_{\tau} \nabla_{\theta} \pi(\tau; \theta) R(\tau)$$

$$= \sum_{\tau} \pi(\tau; \theta) \nabla_{\theta} (\log \pi(\tau; \theta)) R(\tau)$$

$$\pi(\tau; \theta) \frac{\nabla_{\theta} \pi(\tau; \theta)}{\pi(\tau; \theta)}$$

$$\nabla \log x = \frac{\nabla x}{x}$$

Likelihood ratio
trick

$$\nabla_{\theta} J(\theta) = E_{\pi} [\underbrace{\nabla_{\theta} (\log \pi(\tau | \theta))}_{\text{Policy function}} \underbrace{R(\tau)}_{\text{Score function}}]$$

$$\text{Policy gradient} : E_{\pi} [\underbrace{\nabla_{\theta} (\log \pi(s, a, \theta))}_{\text{Policy function}} \underbrace{R(\tau)}_{\text{Score function}}]$$

$$\text{Update rule} : \underbrace{\Delta \theta}_{\text{Change in parameters}} = \underbrace{\alpha}_{\text{Learning rate}} * \nabla_{\theta} (\log \pi(s, a, \theta)) R(\tau)$$

REINFORCE

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \theta)$

Algorithm parameter: step size $\alpha > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

 Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$

 Loop for each step of the episode $t = 0, 1, \dots, T-1$:

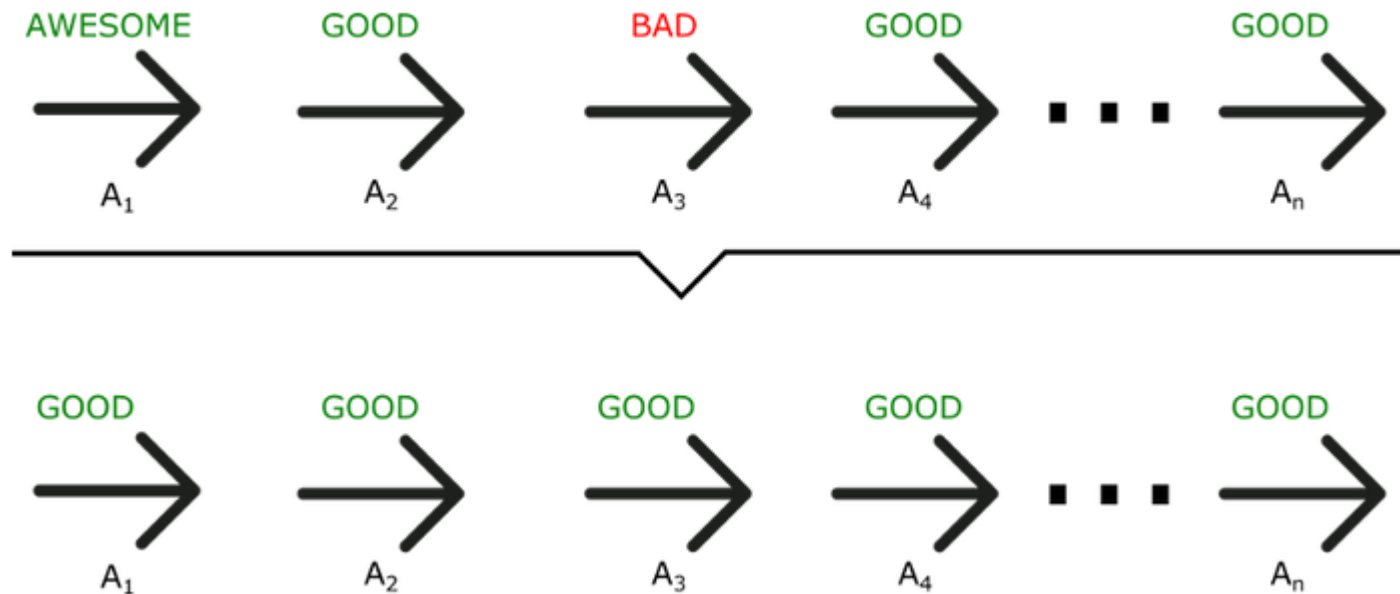
$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad (G_t)$$

$$\theta \leftarrow \theta + \alpha \gamma^t G \nabla \ln \pi(A_t|S_t, \theta)$$

**Podemos medir G a partir de trayectorias
sampleadas del entorno y utilizarlas para actualizar
el gradiente de política.**

“hybrid method”: Actor Critic

- Empleamos dos redes neuronales:
- Critic: mide cuan Buena es la acción tomada (value-based)
- Actor: controla como se comporta el agente (policy-based)



Actor Critic

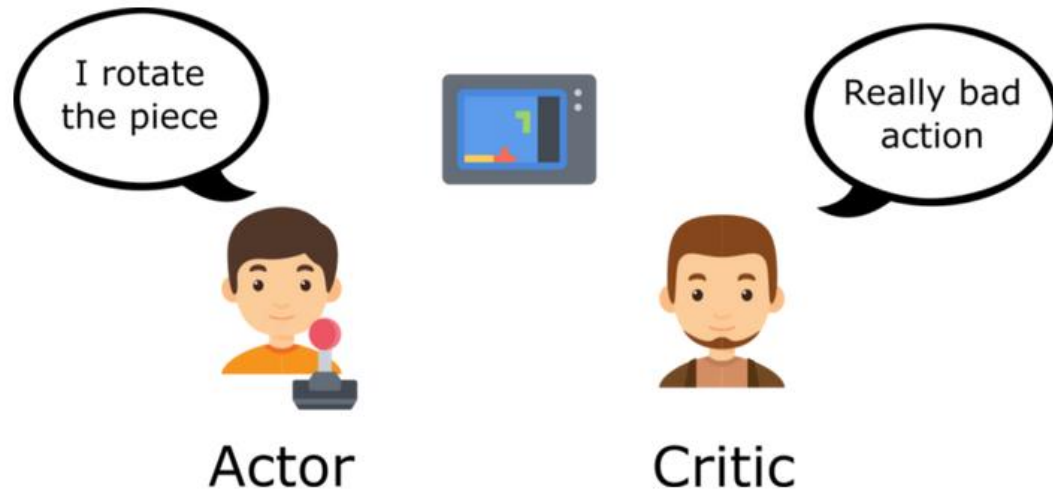
- En lugar de esperar hasta el final del episodio, (Monte Carlo REINFORCE), **realizamos una actualización en cada estado (TD Learning)**.

Policy Update: $\Delta\theta = \alpha * \nabla_{\theta} * (\log \pi(S_t, A_t, \theta)) * \cancel{R(t)}$

New update: $\Delta\theta = \alpha * \nabla_{\theta} * (\log \pi(S_t, A_t, \theta)) * Q(S_t, A_t)$

- Como se realiza una actualización en cada estado no se puede utilizar el reward total $R(t)$. En su lugar, entrenamos un crítico que aproxima la **función de acción-valor**. Dicha función reemplaza a $R(t)$ en **policy gradient**.

Actor Critic



- Al principio, el actor no sabe cómo jugar, por lo cuál prueba acciones al azar. El crítico observa la acción y provee feedback.
- Aprendiendo a partir del feedback, se actualiza la política.
- Al mismo tiempo, el crítico actualiza su manera de proveer feedback de manera tal de mejorar su performance.
- Dos redes neuronales:

$$\pi(s, a, \theta)$$

ACTOR : A policy function, controls how our agent acts.

$$\hat{q}(s, a, w)$$

CRITIC : A value function, measures how good these actions are.

Actor Critic

Policy Update: $\Delta\theta = \alpha \nabla_{\theta}(\log \pi_{\theta}(s, a)) \hat{q}_w(s, a)$

$\hat{q}_w(s, a)$
q learning function approximation
(estimate action value)

Value update: $\Delta w = \beta (R(s, a) + \gamma \hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w(s_t, a_t)) \nabla_w \hat{q}_w(s_t, a_t)$

Policy and value have
different learning rates

$R(s, a) + \gamma \hat{q}_w(s_{t+1}, a_{t+1}) - \hat{q}_w(s_t, a_t)$
TD error

$\nabla_w \hat{q}_w(s_t, a_t)$
Gradient of our value
function

Actor Critic

Actor-critic methods consist of two models, which may optionally share parameters:

- **Critic** updates the value function parameters w and depending on the algorithm it could be action-value $Q_w(a|s)$ or state-value $V_w(s)$.
- **Actor** updates the policy parameters θ for $\pi_\theta(a|s)$, in the direction suggested by the critic.

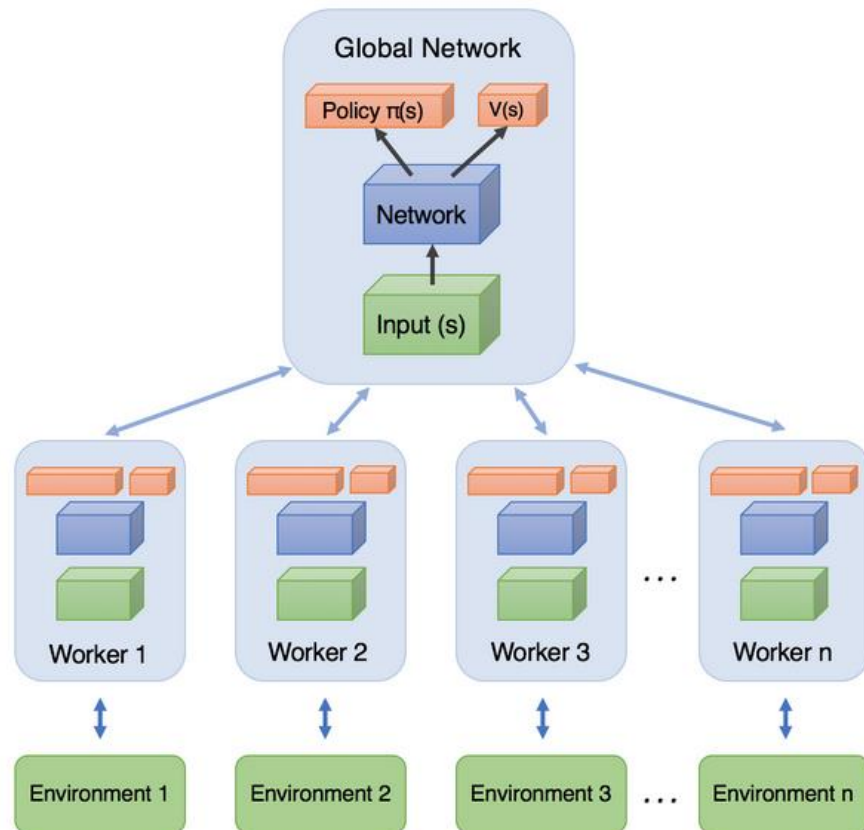
1. Initialize s , θ , w at random; sample $a \sim \pi_\theta(a|s)$.
2. For $t = 1 \dots T$:
 1. Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;
 2. Then sample the next action $a' \sim \pi_\theta(a'|s')$;
 3. Update the policy parameters: $\theta \leftarrow \theta + \alpha_\theta Q_w(s, a) \nabla_\theta \ln \pi_\theta(a|s)$;
 4. Compute the correction (TD error) for action-value at time t :

$$\delta_t = r_t + \gamma Q_w(s', a') - Q_w(s, a)$$
 and use it to update the parameters of action-value function:

$$w \leftarrow w + \alpha_w \delta_t \nabla_w Q_w(s, a)$$
 5. Update $a \leftarrow a'$ and $s \leftarrow s'$.

Two learning rates, α_θ and α_w , are predefined for policy and value function parameter updates respectively.

A2C/A3C



1. We have global parameters, θ and w ; similar thread-specific parameters, θ' and w' .
2. Initialize the time step $t = 1$
3. While $T \leq T_{\text{MAX}}$:
 1. Reset gradient: $d\theta = 0$ and $dw = 0$.
 2. Synchronize thread-specific parameters with global ones: $\theta' = \theta$ and $w' = w$.
 3. $t_{\text{start}} = t$ and sample a starting state s_t .
 4. While $(s_t \neq \text{TERMINAL})$ and $t - t_{\text{start}} \leq t_{\text{max}}$:
 1. Pick the action $A_t \sim \pi_{\theta'}(A_t|S_t)$ and receive a new reward R_t and a new state s_{t+1} .
 2. Update $t = t + 1$ and $T = T + 1$
 5. Initialize the variable that holds the return estimation

$$R = \begin{cases} 0 & \text{if } s_t \text{ is TERMINAL} \\ V_{w'}(s_t) & \text{otherwise} \end{cases}$$
 6. For $i = t - 1, \dots, t_{\text{start}}$:
 1. $R \leftarrow \gamma R + R_i$; here R is a MC measure of G_i .
 2. Accumulate gradients w.r.t. θ' : $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi_{\theta'}(a_i|s_i)(R - V_{w'}(s_i))$;
 Accumulate gradients w.r.t. w' : $dw \leftarrow dw + 2(R - V_{w'}(s_i))\nabla_{w'}(R - V_{w'}(s_i))$.
 7. Update synchronously θ using $d\theta$, and w using dw .

Baseline

A2C/A3C

- **Asynchronous:** a diferencia de las [DQNs](#), donde un único agente representado por una NN interactúa con un entorno A3C utiliza múltiples instancias de manera tal de aprender más eficientemente. En A3C existe una red global y múltiples agentes que actualizan su propio set de parámetros. Por lo tanto, la experiencia de cada agente es independiente del resto.
- **Actor-Critic:** En el caso de A3C, la red estima $V(s)$ y $\pi(s)$. El agente utiliza la estimación de valor (el crítico) para actualizar la política (the actor) de manera más eficiente que los métodos tradicionales.
- **Advantage:** $A = Q(s,a) - V(s)$ como baseline.

Créditos!!!



- Thomas Simonini (Parte de las Figuras)
- Hado van Hasselt (Double Q-Learning)
- Lilian Weng (Actor Critic Algorithms)
- Ziyu Weng (Figura Dueling DQN)
- Andrej Karpathy (Figura Gradient Descent Algorithm)
- Richard Sutton
- Tom Schaul (PER)