Deep Reinforcement Learning: Deep Q Learning

Francesco Pugliese, PhD

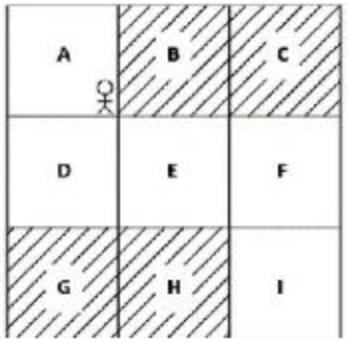
neural1977@gmail.com

We will discuss about…

- ✓ Markov Decision Process (MDP)
- ✓ Deep Q Learning (DQL)

Markov Decision Process

- ✓ Let's think about a **Grid World Environment** with the goal of the agent to reach state **I** from state **A, without visiting the states in the shadow.**

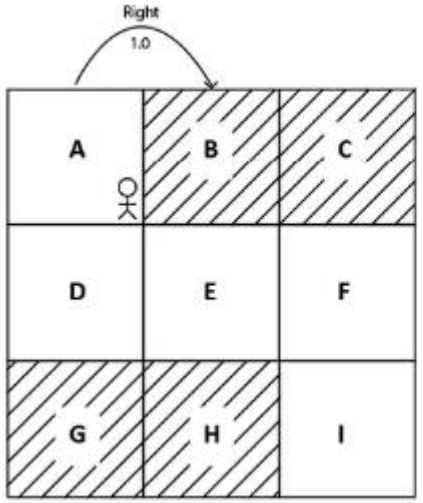


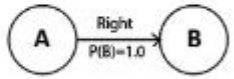
- ✓ For this Environmens:

1. **States:** In the grid world environment, we have states **A** to **I**.
2. **Actions:** In the grid world environment, the set of actions is **up, down, left, and right**.

Markov Decision Process

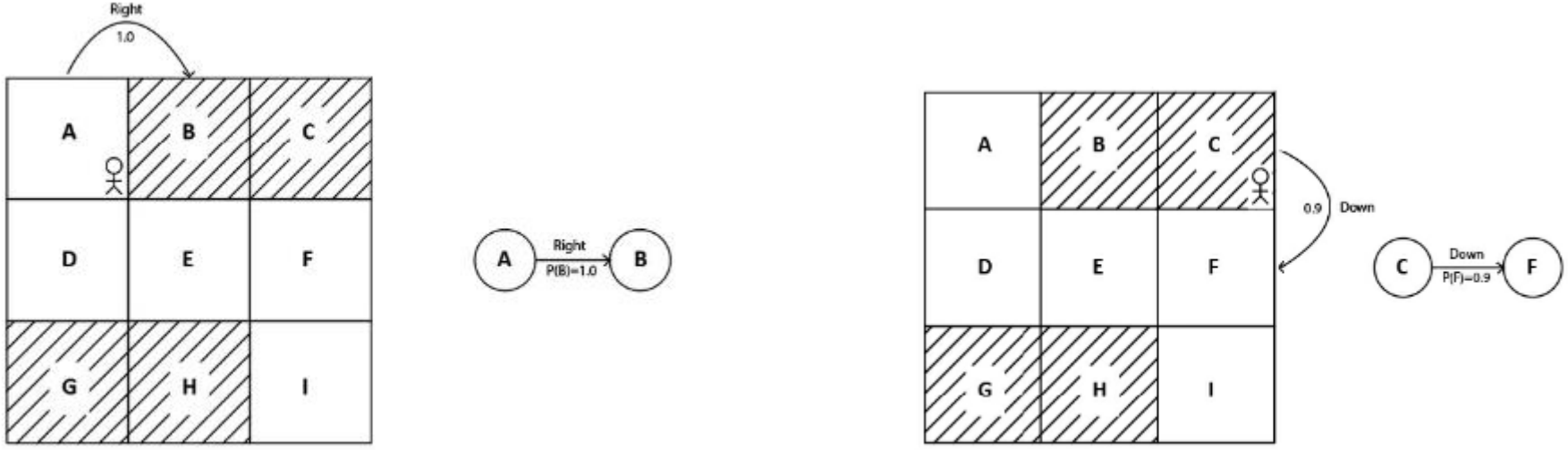
3. **Transition probability:** If you observe, in the MRP, the transition probability is just , that is, the probability of going from state *s* to state *s′*, and it doesn't include actions. But in the MDP, we include the actions, and thus the transition probability is denoted by $P(s'|s,a)$. For example, in our grid world environment, say the transition probability of moving from state A to state B while performing an action right is 100%. This can be expressed as **P(B|A, right) = 1.0**. We can also view this in the state diagram:





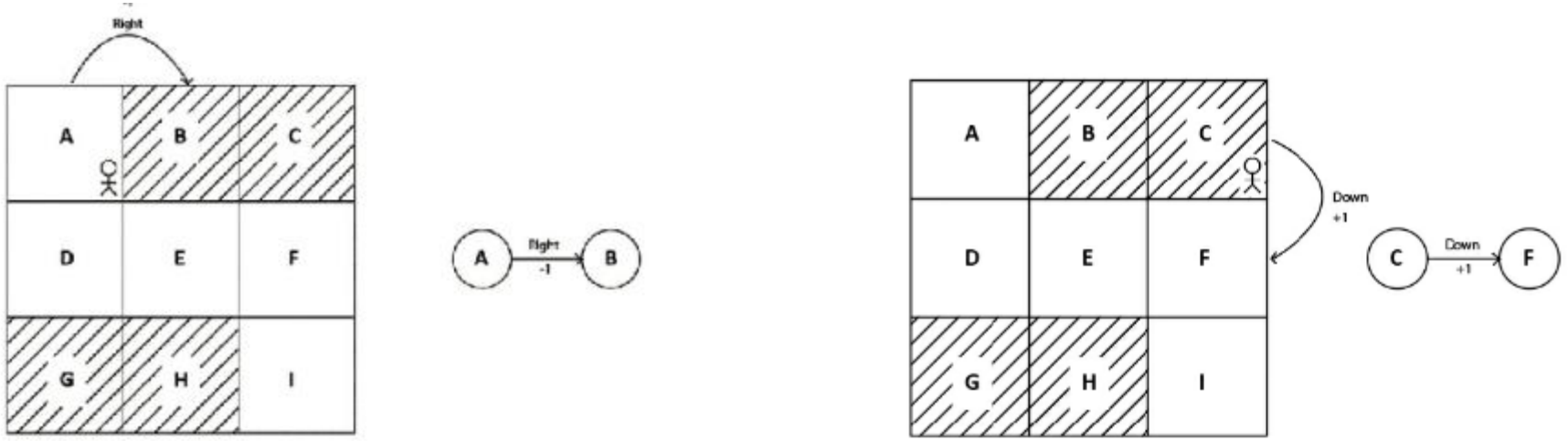
Markov Decision Process

3. **Transition probability:** If you observe, in the MRP, the transition probability is just , that is, the probability of going from state *s* to state *s′*, and it doesn't include actions. But in the MDP, we include the actions, and thus the transition probability is denoted by $P(s'|s,a)$. For example, in our grid world environment, say the transition probability of moving from state **A** to state B while performing an action right is 100%. This can be expressed as **P(B|A, right) = 1.0**. Suppose our agent is in state **C** and the transition probability of moving from state **C** to state **F** while performing the action down is **90%**, then it can be expressed as **P(F|C, down) = 0.9**.



Markov Decision Process

4. **Reward function:** The reward function is denoted by $R(s, a, s')$. It represents the reward our agent obtains while transitioning from state *s* to state *s´* while performing an action *a*. Say the reward we obtain while transitioning from state A to state B while performing the action right is -1, then it can be expressed as R(A, right, B) = -1. Suppose our agent is in state C and say the reward we obtain while transitioning from state C to state F while performing the action down is +1, then it can be expressed as R(C, down, F) = +1.



Deep Q Learning (DQL)

- ✓ The **Deep Q Learning** algorithm was first proposed by researchers at **Google's DeepMind** in 2013 in the paper Playing Atari with **Deep Reinforcement Learning**. They described the DQN architecture and explained why it was so effective at playing Atari games with human-level accuracy.

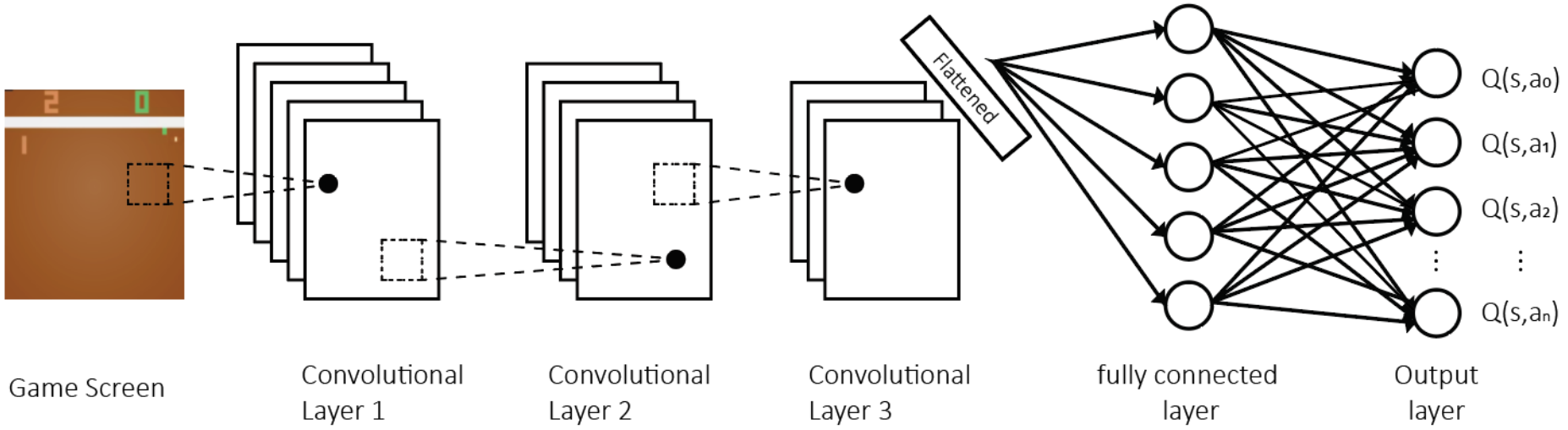
- ✓ **Atari 2600** is a popular **video game console** from a game company called **Atari**. The Atari game console provides several popular games such as pong, space invaders, Ms Pacman, break out, centipede and many more. In this section, we will learn how to build the deep Q network for playing the Atari games. First, let's understand the architecture of DQN for playing the Atari games.

- ✓ In the **Atari** environment, the image of the game screen is the state of the environment. So, we just feed the image of the game screen as an input to the **Deep Q Network (DQN)** and it returns the **Q value** of all the actions in the state. Since we are dealing with the images, instead of using the vanilla deep neural network for approximating the Q value, we can use the **Convolutional Neural Network (CNN)** since the convolutional neural network is very effective for handling images.

Deep Q Learning (DQL)

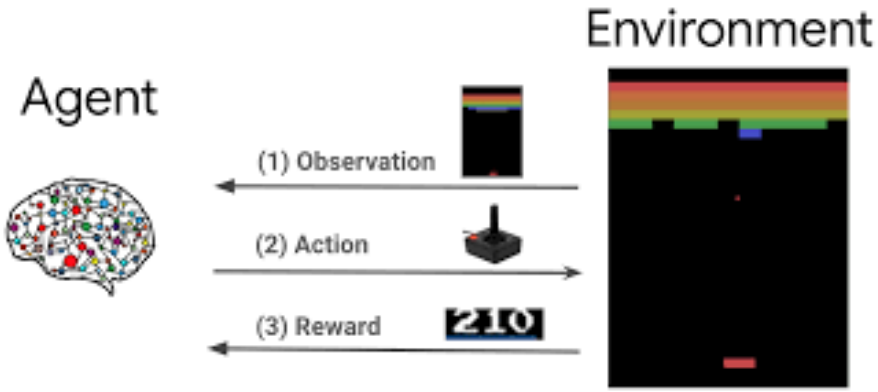
- ✓ Thus, now our **DQN** is the convolutional neural network. We feed the image of the game screen (game state) as an input to the convolutional neural network and it outputs the Q value of all the actions in the state.

- ✓ Given the image of the game screen (game state), the convolutional layers extract features from the image and produce a feature map. Next, we flatten the feature map and feed the flattened feature map as an input to the feedforward network. The feedforward network takes this flattened feature map as an input and returns the Q value of all the actions in the state:

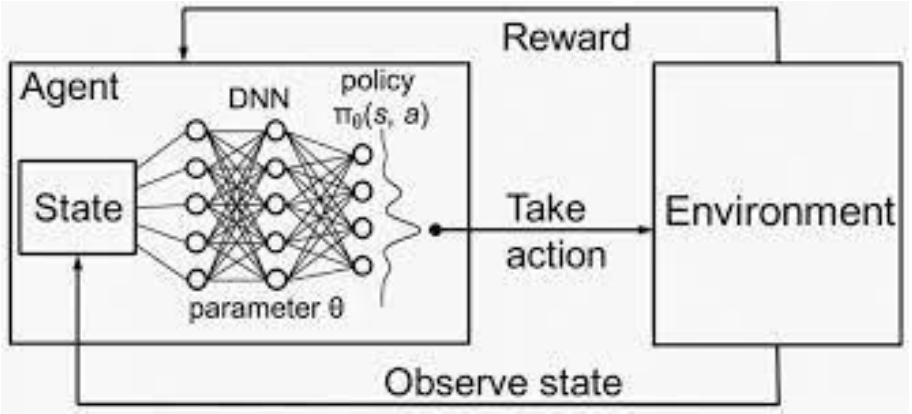


Deep Q Learning (DQL)

- ✓ We don't perform **pooling** operation. A pooling operation is useful when we perform tasks such as object detection, image classification and so on where we don't consider the position of the object in the image and we just want to know whether the desired object is present in the image. For example, if we want to identify whether there is a dog in an image, we only look for whether a dog is present in the image and we don't check the position of the dog in the image. Thus, in this case, pooling operation is used to identify whether there is a dog in the image irrespective of the position of the dog.



- ✓ But in our setting, pooling operation should not be performed. Because to understand the current **game screen** (state), the position is very important. For example, in a Pong game, we just don't want to classify if there is a ball in the game screen. We want to know the position of the ball so that we can take better action. Thus, we don't include the pooling operation in our **DQN architecture**.



Some Concepts of Deep Q Learning (DQL)

- ✓ **Policy**: A policy defines the agent's behavior in an environment. The policy tells the agent what action to perform in each state. For instance, in the grid world environment, we have states A to I and four possible actions. The policy may tell the agent to move down in state A, move right in state D, and so on. A **Deterministic Policy** tells the agent to perform one particular action in a state. Thus, the deterministic policy maps the state to one particular action. Unlike a deterministic policy, a **Stochastic Policy** does not map a state directly to one particular action; instead, it maps the state to a probability distribution over an action space. In a given a state, the deterministic policy will tell the agent to perform one particular action in the given state, so whenever the agent visits the state it always performs the same particular action. But with a stochastic policy, given a state, the stochastic policy will return a probability distribution over an action space. So instead of performing the same action every time the agent visits the state, the agent performs different actions each time based on a probability distribution returned by the stochastic policy. A stochastic policy is called a **Categorical Policy** when the action space is discrete. That is, the stochastic policy uses a categorical probability distribution over the action space to select actions when the action space is discrete. A stochastic policy is called a **Gaussian Policy** when our action space is continuous. That is, the stochastic policy uses a Gaussian probability distribution over the action space to select actions when the action space is continuous.

Some Concepts of Deep Q Learning (DQL)

- ✓ **Episode:** The agent interacts with the environment by performing some actions, starting from the initial state and reaches the final state. This agent-environment interaction starting from the initial state until the final state is called an episode. For instance, in a car racing video game, the agent plays the game by starting from the initial state (the starting point of the race) and reaches the final state (the endpoint of the race). This is considered an episode. An episode is also often called a trajectory (the path taken by the agent). An agent can play the game for any number of episodes, and each episode is independent of the others. What is the use of playing the game for multiple episodes? In order to learn the optimal policy, that is, the policy that tells the agent to perform the correct action in each state, the agent plays the game for many episodes.

- ✓ **Episodic task:** As the name suggests, an episodic task is one that has a terminal/final state. That is, episodic tasks are tasks made up of episodes and thus they have a terminal state. For example, in a car racing game, we start from the starting point (initial state) and reach the destination (terminal state).

- ✓ **Continuous task:** Unlike episodic tasks, continuous tasks do not contain any episodes and so they don't have any terminal state. For example, a personal assistance robot does not have a terminal state.

Some Concepts of Deep Q Learning (DQL)

- ✓ **Finite horizon:** If the agent-environment interaction stops at a particular time step, then the horizon is called a finite horizon.
- ✓ **Infinite horizon:** If the agent-environment interaction never stops, then it is called an infinite horizon.
- ✓ **Return:** A return can be defined as the sum of the rewards obtained by the agent in an episode.
- ✓ **Value Function:** also called the state value function, denotes the value of the state. The value of a state is the return an agent would obtain starting from that state following policy.
- ✓ **Q Function:** also called the state-action value function, denotes the value of a state-action pair. The value of a state-action pair is the return the agent would obtain starting from state s and performing action a following policy.
- ✓ **Q Value:** also called state-action value, it is the value of a state-action pair or Q function is usually denoted by Q(s,a).

Deep Q Learning (DQL)

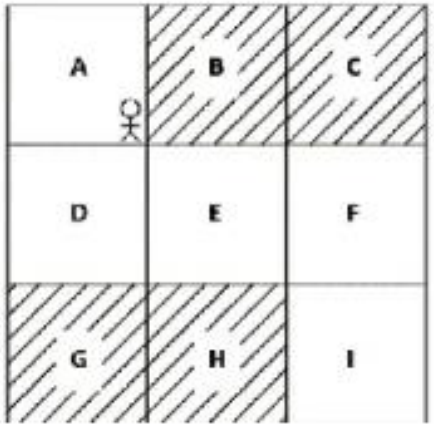
- ✓ The objective of deep reinforcement learning is to find the optimal policy, that is, the policy that gives us the maximum return **(the sum of rewards of the episode).** In order to compute the policy, first we compute the **Q function**. Once we have the Q function, then we extract the policy by selecting an action in each state that has the maximum **Q value**.

- ✓ Let's suppose we have two states A and B and our action space consists of two actions; let the actions be up and down. So, in order to find which action to perform in state A and B, first we compute the Q value of all state-action pairs:

State	Action	Value
A	up	17
A	down	10
B	up	11
B	down	20

Deep Q Learning (DQL)

- ✓ Let's consider the Grid Environment:



- ✓ We learned that in the grid world environment, the goal of our agent is to reach state I from state A without visiting the shaded states, and in each state, the agent has to perform one of the four actions—**up, down, left, right.**

- ✓ To compute the policy, first we compute the Q values of all state-action pairs. Here, the number of states is 9 (A to I) and we have 4 actions in our action space, so our Q table will consist of 9 x 4 = 36 rows containing the Q values of all possible state-action pairs.

Deep Q Learning (DQL)

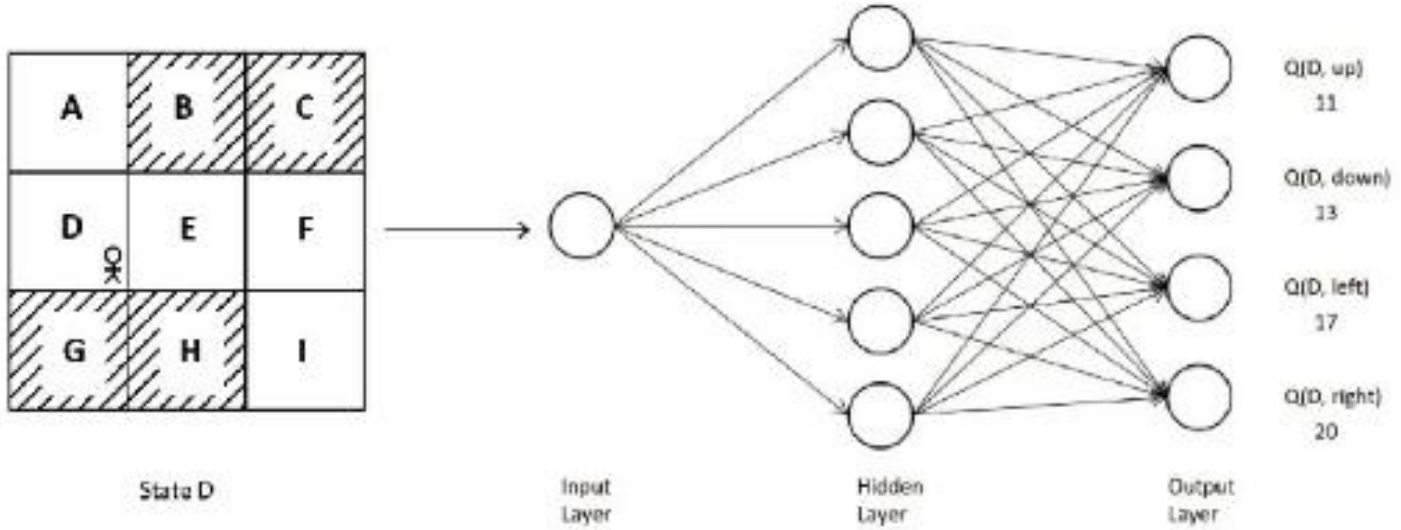
- ✓ Now we have the **Q Values**, then we extract the policy by selecting the action in each state that has the maximum Q value. But is it a good approach to compute the Q value exhaustively for all state-action pairs? Let's explore this in more detail.

- ✓ Let's suppose we have an environment where we have 1,000 states and 50 possible actions in each state. In this case, **our Q table will consist of 1,000 x 50 = 50,000 rows** containing the Q values of all possible state-action pairs. In cases like this, where our environment consists of a large number of states and actions, it will be very expensive to compute the **Q values** of all possible state-action pairs in an exhaustive fashion.

- ✓ Instead of computing **Q values** in this way, can we approximate them using any function approximator, such as a neural network? Yes! We can parameterize our Q function by a parameter and compute the Q value where the parameter is just the parameter of our neural network. So, we just feed the state of the environment to a neural network and it will return the Q value of all possible actions in that state. Once we obtain the Q values, then we can select the **best action** as the one that has the **maximum Q value**.

Deep Q Learning (DQL)

- ✓ For example, let's consider our **grid world environment**. We just feed state D as an input to the network and it returns the Q value of all actions in state D, which are **up, down, left, and right**, as output. Then, we select the action that has the maximum Q value. Since action right has a maximum **Q value**, we select action right in the state D.



Deep Q Learning (DQL)

- ✓ Since we are using a neural network to approximate the **Q value**, the neural network is called the **Q network**, and if we use a deep neural network to approximate the Q value, then the deep neural network is called a **Deep Q Network (DQN)** and the process of learning is called **Deep Q Learning (DQL)**.

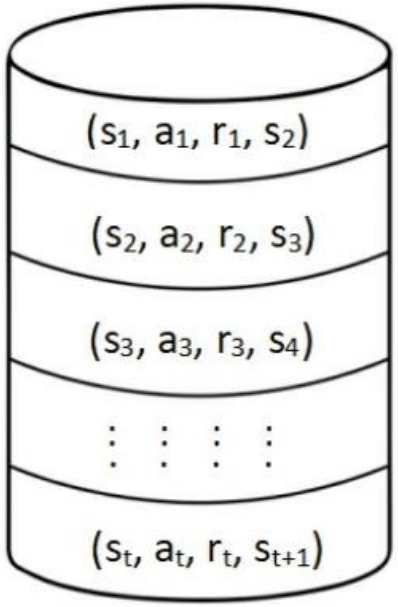
- ✓ We can denote our **Q function** by $Q_\theta(s, a)$, where the parameters **"theta"** indicates that our Q function is parameterized the parameters of our neural network. We initialize the network parameters with random values and approximate the **Q function** (Q values), but since we initialized with random values, the approximated Q function will not be optimal, initially. So, we train the network for several iterations by finding the optimal parameters. Once we find the optimal parameters, we will have the optimal Q function. Then we can extract the optimal policy from the optimal Q function (the maximum).

- ✓ Okay, but how can we train our network? What about the training data and the loss function? Is it a classification or regression task? Now that we have a basic understanding of how DQN works, we must get into the details and address all these questions.

Deep Q Learning (DQL)

- ✓ The Q value is just a continuous number, so we are essentially using our DQN to perform a regression task, so this is a Regression problem.

- ✓ What about the training data? We use a buffer called a **Replay Buffer** to collect the agent's experience and, based on this experience, we train our network.



Deep Q Learning (DQL)

- ✓ We know that the agent makes a transition from a state *s* to the next state *s´* by performing some action *a*, and then receives a reward *r*. We can save this transition information *(s, a, r, s')* into a buffer called a **Replay Buffer** (or experience replay). The replay buffer is usually denoted by *D*. This transition information is basically the agent's experience. We store the agent's experience over several episodes in the replay buffer. The key idea of using the replay buffer to store the agent's experience is that we can train our DQN with experience (transition) sampled from the buffer.

- ✓ The following steps help us to understand how we store the transition information in the replay buffer :
    1. Initialize the replay buffer.
    2. For each episode perform step 3.
    3. For each step in the episode:
        1. Make a transition, that is, perform an action a in the state s, move to the next state s', and receive the reward r.

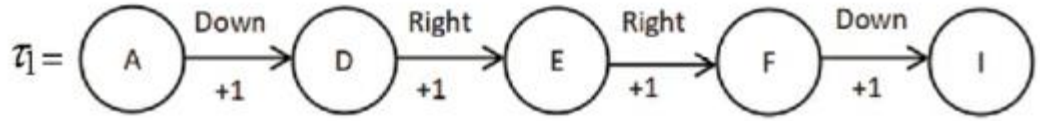
        2. Store the transition information *(s, a, r, s')* in the replay buffer *D*.

Deep Q Learning (DQL)

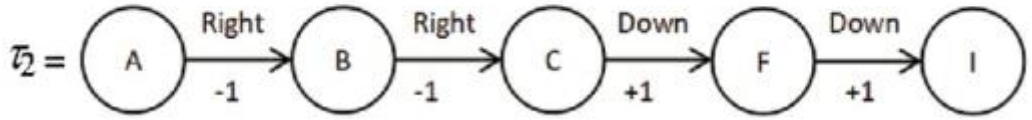
- ✓ As explained in the preceding steps, we collect the agent's transition information over many episodes and save it in the replay buffer. To understand this clearly, let's consider our favorite grid world environment.

- ✓ Let's suppose we have the following two episodes/trajectories:

Episode 1:

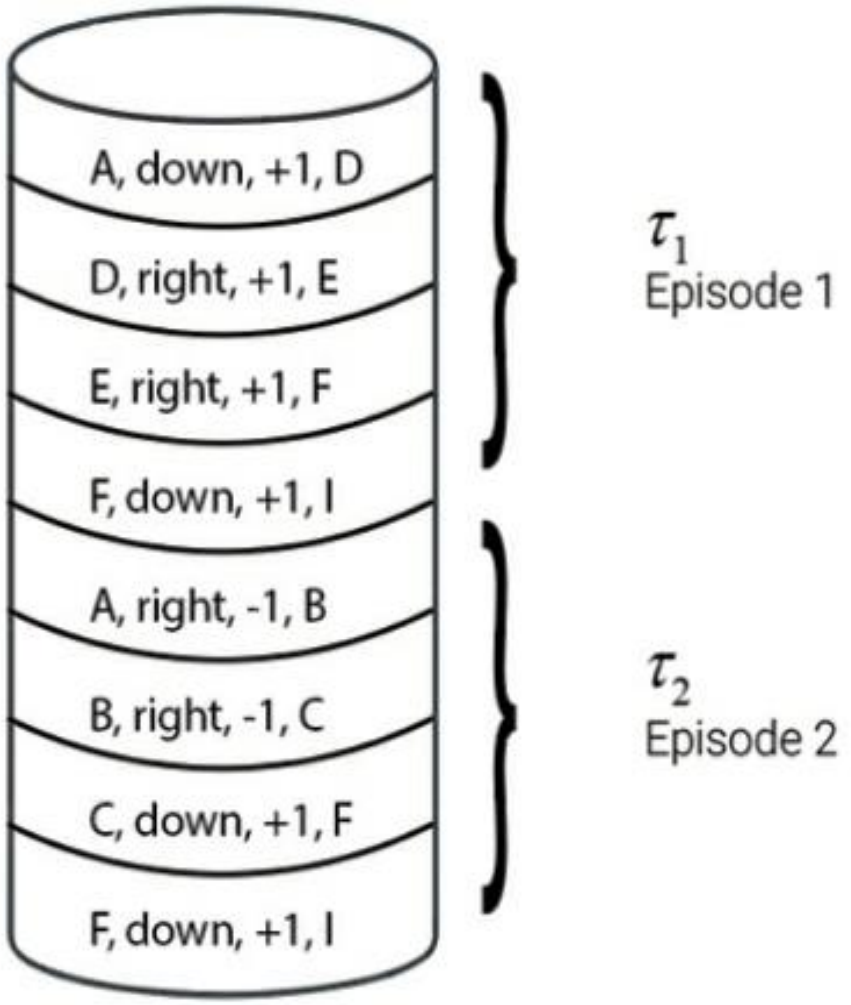


Episode 2:



Deep Q Learning (DQL)

- ✓ Now, this information will be stored in the replay buffer.
- ✓ We store the transition information by stacking it sequentially one after another. We train the network by sampling a minibatch of transitions from the replay buffer.
- ✓ But since we are stacking up the agent's experience (transition) one after another sequentially, the agent's experience will be highly correlated. For example, as shown in the preceding figure, transitions will be correlated with the rows above and below. If we train our network with this correlated experience then our neural network will easily overfit. So, to combat this, we sample a random minibatch of transitions from the replay buffer and train the network.



Deep Q Learning (DQL)

- ✓ The replay buffer is of limited size, that is, a replay buffer will store only a fixed amount of the agent's experience. So, when the buffer is full we replace the old experience with new experience.

- ✓ Here is the reason we a Reply Buffer is usually implemented as a **Queue** structure **(first in first out)** rather than a list. So, if the buffer is full when new experience comes in, we remove the old experience and add the new experience into the buffer.

- ✓ We have learned that we train our network by randomly sampling a minibatch of experience from the buffer. But how exactly does the training happen? How does our network learn to approximate the optimal Q function using this minibatch of samples?

- ✓ **Loss Function:** We learned that in **DQN**, our goal is to predict the Q value, which is just a continuous value. Thus, in **DQN** we basically perform a regression task. We generally use the **mean squared error (MSE)** as the loss function for the regression task. MSE can be defined as the average squared difference between the target value and the predicted valu:

$$\mathrm{MSE} = \frac{1}{K}\sum_{i=1}^{K}(y_i - \hat{y}_i)^2$$

Deep Q Learning (DQL)

- ✓ Now, let's learn how to use **MSE** in the **DQN** and train the network. We can train our network by minimizing the MSE between the target Q value and predicted Q value. First, how can we obtain the target Q value? Our target Q value should be the optimal Q value so that we can train our network by minimizing the error between the optimal Q value and predicted Q value.

- ✓ But how can we compute the optimal Q value? This is where the Bellman equation helps us. In **Dynamic Programming**, the **Bellman Equation** helps to find the optimal **Q value:**

$$Q^*(s,a) = \underset{s' \sim P}{E}\left[r + \gamma \max_{a'} Q^*(s',a')\right]$$

- ✓ Where *r* represents the immediate reward that we obtain while performing an action *a* in state *s* and moving to the next state *s′*

- ✓ In the equation, we can remove the expectation. We will approximate the expectation by sampling K number of transitions from the replay buffer and taking the average value.

Deep Q Learning (DQL)

- ✓ Thus, according to the **Bellman** optimality equation, the optimal Q value is just the sum of the reward and the discounted maximum Q value of the next state-action pair:

$$Q^*(s,a) = r + \gamma \max_{a'} Q^*(s',a')$$

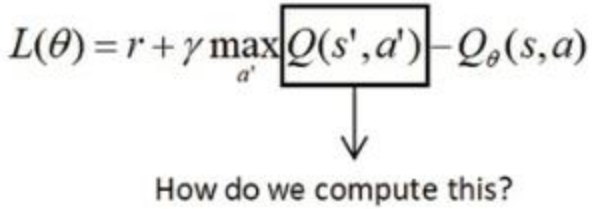
- ✓ So, we can define our loss as the difference between the target value (the optimal Q value) and the predicted value (the Q value predicted by the DQN) and express the loss function L as:

$$L(\theta) = Q^*(s,a) - Q_\theta(s,a)$$

$$L(\theta) = r + \gamma \max_{a'} Q(s',a') - Q_\theta(s,a)$$

Deep Q Learning (DQL)

- ✓ We know that we compute the predicted Q value using the parameterized network. How can we compute the target value? That is, we learned that the target value is the sum of the reward and the discounted maximum Q value of the next state-action pair. How do we compute the Q value of the next state-action pair?

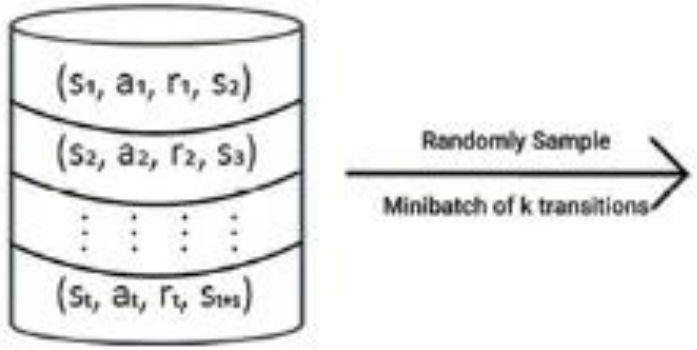


- ✓ Similar to the predicted Q value, we can compute the Q value of the next state-action pair in the target using the same DQN parameterized. So, both the target value and the predicted Q value are parameterized.

$$L(\theta) = r + \gamma \max_{a'} Q_\theta(s', a') - Q_\theta(s, a)$$

Deep Q Learning (DQL)

- ✓ Instead of computing the loss as just the difference between the target Q value and the predicted Q value, we use MSE as our loss function. We learned that we store the agent's experience in a buffer called a replay buffer. So, we randomly sample a minibatch of K number of transitions from the replay buffer and train the network by minimizing the MSE:



$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} Q_\theta(s_i', a') - Q_\theta(s_i, a_i))^2$$

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} Q_\theta(s_i', a') - Q_\theta(s_i, a_i))^2$$

Deep Q Learning (DQL)

- ✓ We have learned that the target value is just the sum of the reward and the discounted maximum Q value of the next state-action pair. But what if the next state is a terminal state? If the next state is terminal then we cannot compute the Q value as we don't take any action in the terminal state, so in that case, the target value will be just the reward, as shown here:

$$y_i = \begin{cases} r_i & \text{if } s' \text{ is terminal} \\ r_i + \gamma \max_{a'} Q_\theta(s_i', a') & \text{if } s' \text{ is not terminal} \end{cases}$$

$$L(\theta) = \frac{1}{K} \sum_{i=1}^{K} (y_i - Q_\theta(s_i, a_i))^2$$

Deep Q Learning (DQL)

- ✓ We train our network by minimizing the loss function. We can minimize the loss function by finding the optimal parameters. So, we use gradient descent to find the optimal parameters. We compute the gradient of our loss function and update our network parameters:

$$\theta = \theta - \alpha \nabla_{\theta} L(\theta)$$

- ✓ However, there is a small issue with our loss function. We have learned that the target value is just the sum of the reward and the discounted maximum Q value of the next state-action pair. We compute this Q value of the next state-action pair in the target and predicted Q values using the same network parameterized by theta:

$$L(\theta) = \frac{1}{K} \sum_{i=1}^{K} (r_i + \gamma \max_{a'} \underbrace{Q_{\theta}(s_i', a')}_{\text{Compute using } \theta} - \underbrace{Q_{\theta}(s_i, a_i)}_{\text{Compute using } \theta})^2$$

Deep Q Learning (DQL)

- ✓ **The problem is:** since the target and predicted value depend on the same parameter , this will cause instability in the MSE and the network learns poorly. It also causes a lot of divergence during training. Let's understand this with a simple example. We will take arbitrary numbers to make it easier to understand. We know that we try to minimize the difference between the target value and the predicted value. So, on every iteration, we compute the gradient of loss and update our network parameter so that we can make our predicted value the same as the target value.

- ✓ Let's suppose in iteration 1, the target value is 13 and the predicted value is 11. So, we update our parameter to match the predicted value to the target value, which is 13. But in the next iteration, the target value changes to 15 and the predicted value becomes 13 since we updated our network parameter . So, again we update our parameter to match the predicted value to the target value, which is now 15. But in the next iteration, the target value changes to 17 and the predicted value becomes 15 since we updated our network parameter.

Target value	Predicted value
13	11
15	13
17	15

Deep Q Learning (DQL)

- ✓ This is because the predicted and target values both depend on the same parameter . If we update, then both the target and predicted values change. Thus, the predicted value keeps on trying to be the same as the target value, but the target value keeps on changing due to the update on the network parameter . How can we avoid this? Can we freeze the target value for a while and compute only the predicted value so that our predicted value matches the target value? Yes! To do this, we introduce another neural network called a **Target Network** for computing the Q value of the next state-action pair in the target. The parameter of the target network is represented by theta' . So, our main deep Q network, which is used for predicting Q values, learns the optimal parameter theta using gradient descent.

- ✓ The target network is frozen for a while and then the target network parameter theta' is updated by just copying the main deep Q network parameter theta. Freezing the target network for a while and then updating its parameter theha' with the main network parameter theta stabilizes the training. The new Loss:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a') - Q_{\theta}(s_i, a_i))^2$$

Deep Q Learning (DQL)

- ✓ Thus, the Q value of the next state-action pair in the target is computed by the target network parameterized by theta', and the predicted Q value is computed by our main network parameterized by theta:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(r_i + \gamma \max_{a'} \underbrace{Q_{\theta'}(s_i', a')}_{\text{Compute using } \theta'} - \underbrace{Q_{\theta}(s_i, a_i)}_{\text{Compute using } \theta})^2$$

Deep Q Learning (DQL)

- ✓ **All the algorithm together**: First, we initialize the main network parameter with random values. We learned that the target network parameter is just a copy of the main network. So, we initialize the target network parameter theta' by just copying the main network parameter theta. We also initialize the replay buffer *D*. Now, for each step in the episode, we feed the state of the environment to our network and it outputs the Q values of all possible actions in that state. Then, we select the action that has the maximum Q value:

$$a = \arg\max_{a} Q_{\theta}(s, a)$$

- ✓ If we only select the action that has the highest Q value, then we will not explore any new actions. So, to avoid this, we select actions using the epsilon-greedy policy. With the epsilon-greedy policy, we select a random action with probability epsilon and with probability 1-epsilon, we select the best action that has the maximum Q value. Note that, since we initialized our network parameter theta with random values, the action we select by taking the maximum Q value will not be the optimal action. But that's okay, we simply perform the selected action, move to the next state, and obtain the reward. If the action is good then we will receive a positive reward, and if it is bad then the reward will be negative. We store all this transition information *(s,a,r,s')* in the replay buffer *D*.

Deep Q Learning (DQL)

- ✓ Next, we randomly sample a minibatch of K transitions from the replay buffer and compute the loss. We have learned that our loss function is computed as:

$$L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a_i))^2$$

Where y_i is the target value, that is,

$$y_i = r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a')$$

- ✓ In the initial iterations, the loss will be very high since our network parameter theta is just random values. To minimize the loss, we compute the gradients of the loss and update our network parameter theta as:

$$\theta = \theta - \alpha \nabla_\theta L(\theta)$$

- ✓ We don't update the target network parameter theta' in every time step. We freeze the target network parameter theta' for several time steps and then we copy the main network parameter theta to the target network parameter theta'. We keep repeating the preceding steps for several episodes to approximate the optimal Q value. Once we have the optimal Q value, we extract the optimal policy from them.

Deep Q Learning Algorithm

The DQN algorithm is given in the following steps:

1. Initialize the main network parameter θ with random values
2. Initialize the target network parameter θ' by copying the main network parameter θ
3. Initialize the replay buffer $\mathcal{D}$
4. For N number of episodes, perform *step 5*
5. For each step in the episode, that is, for t = 0, . . ., T-1:
    1. Observe the state s and select an action using the epsilon-greedy policy, that is, with probability epsilon, select random action a and with probability 1-epsilon, select the action $a = \arg\max_a Q_\theta(s, a)$

Deep Q Learning (DQL)

2. Perform the selected action and move to the next state s' and obtain the reward r
3. Store the transition information in the replay buffer $\mathcal{D}$
4. Randomly sample a minibatch of K transitions from the replay buffer $\mathcal{D}$
5. Compute the target value, that is, $y_i = r_i + \gamma \max_{a'} Q_{\theta'}(s_i', a')$
6. Compute the loss, $L(\theta) = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q_\theta(s_i, a_i))^2$
7. Compute the gradients of the loss and update the main network parameter θ using gradient descent: $\theta = \theta - \alpha \nabla_\theta L(\theta)$
8. Freeze the target network parameter θ' for several time steps and then update it by just copying the main network parameter θ

References

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Francesco **Pugliese**