

# Types of Deep Learning Problems

## Supervised Learning

**Data:**  $(x, y)$   
 $x$  is data,  $y$  is label

**Goal:** Learn a *function* to map  $x \rightarrow y$

**Examples:** Classification, regression, object detection, semantic segmentation, image captioning, etc.



→ Cat

Classification

When we have data,  $x$ , and labels,  $y$ , supervised learning yields a function that is a *mapping* from  $x$  to  $y$ .

## Unsupervised Learning

**Data:**  $x$   
Just data, no labels!

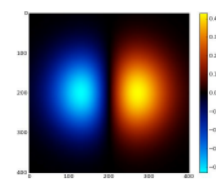
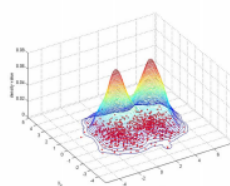
**Goal:** Learn some underlying hidden *structure* of the data

**Examples:** Clustering, dimensionality reduction, feature learning, density estimation, etc.



Figure copyright Ian Goodfellow, 2016. Reproduced with permission.

1-d density estimation



2-d density estimation

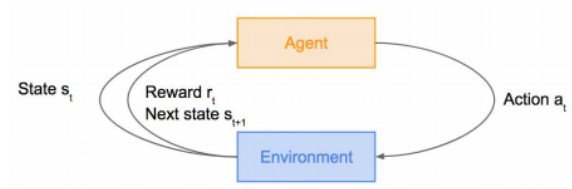
2-d density images [left](#) and [right](#) are CC0 public domain

When we have data and no labels, unsupervised learning allows us to discover the underlying, hidden structure of the data.

# Reinforcement Learning

Problems involving an **agent** interacting with an **environment**, which provides numeric **reward** signals

**Goal:** Learn how to take actions in order to maximize reward



In reinforcement learning we have an agent that can take actions within its environment, and the agent can receive rewards for its action. The goal is to learn which series of actions will maximize its overall reward.

## Reinforcement Learning

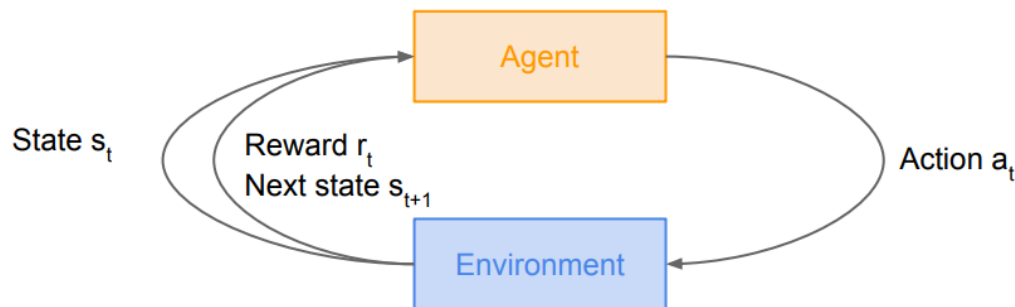
### Overview

- What is Reinforcement Learning?
- Markov Decision Processes
- Q-Learning
- Policy Gradients

First we will cover the reinforcement learning problem, from there we'll talk about Markov decision processes, which is a formalism of the reinforcement learning problem. Then we'll explain two major classes of RL algorithms, Q-learning and policy gradients.

# What is Reinforcement Learning?

## Basic Definition

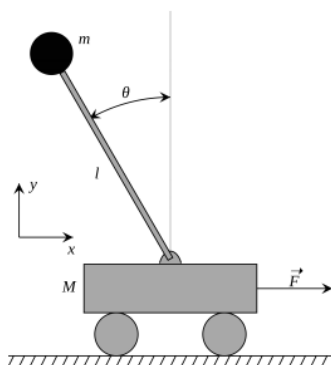


In the reinforcement learning setup, we have an **agent** and an **environment**. The environment gives the agent a **state**. In turn, the agent is going to take an **action**, and then the environment is going to give back a **reward**, as well as the next state. This is going to keep going on in this loop, until the environment returns a **terminal state**, which then ends the **episode**.

The goal is to learn which series of actions will maximize its overall reward.

## Example Problems

### Cart-Pole Problem



**Objective:** Balance a pole on top of a movable cart

**State:** angle, angular speed, position, horizontal velocity

**Action:** horizontal force applied on the cart

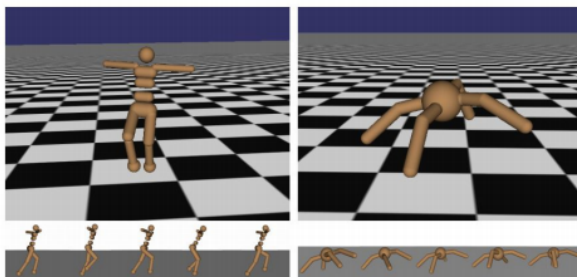
**Reward:** 1 at each time step if the pole is upright

This image is CC0 public domain

Let's start with some examples of this. First we have the cart-pole problem, a classic problem that some of you may have seen before. The objective here is that you want to

balance a pole on top of a movable cart. The state here is the current description of the system. For example, angle, angular speed of your pole, your position, and the horizontal velocity of your cart. The actions you can take are horizontal forces that you apply onto the cart. The goal is to move this cart around to try and balance this pole on top of it. And the reward that you get from the environment is 1 at each time step if your pole is upright. So you want to keep this pole balanced for as long as you can.

## Robot Locomotion



**Objective:** Make the robot move forward

**State:** Angle and position of the joints

**Action:** Torques applied on joints

**Reward:** 1 at each time step upright + forward movement

Figures copyright John Schulman et al., 2016. Reproduced with permission.

Another classic example of a RL problem is robot locomotion. One is the example of a humanoid robot, another is an ant robot. The objective is to make the robot move forward. So the *state* that describing the system is the *angle* and *positions* of all the joints of the robots. The *actions* that can be taken are the *torques* applied onto these joints. These are trying to make the robot move forward. The *reward* system will be setup to positively reinforce *forward movement* as well as, in the case of the humanoid, for each time step that the robot remains *upright*.

## Atari Games



**Objective:** Complete the game with the highest score

**State:** Raw pixel inputs of the game state

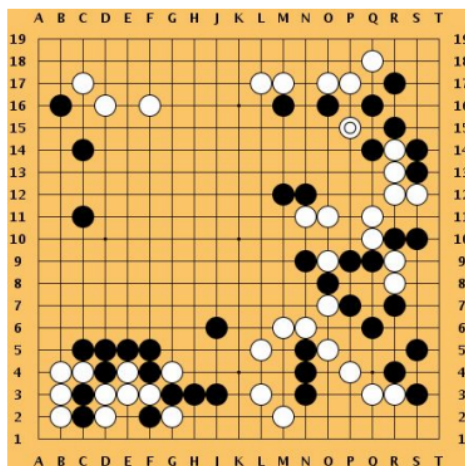
**Action:** Game controls e.g. Left, Right, Up, Down

**Reward:** Score increase/decrease at each time step

Figures copyright Volodymyr Mnih et al., 2013. Reproduced with permission.

Games are also a big class of problems that can be formulated with RL. Atari games are a classic success of deep reinforcement learning. The objective is to complete these games with the highest possible score. The agent is a player and the state is the raw pixels of the game state. These are just the pixels on the screen that you would see as you're playing the game and the actions that you have are your game controls, for example, moving left to right, up or down. The score that you have is your score increase or decrease at each time step, and your goal is going to be to maximize your total score over the course of the game.

## Go



**Objective:** Win the game!

**State:** Position of all pieces

**Action:** Where to put the next piece down

**Reward:** 1 if win at the end of the game, 0 otherwise

This image is CC0 public domain

Finally, here we have the example of the game Go. It was a huge achievement of deep reinforcement learning in 2016, when Deep Minds AlphaGo beat Lee Sedol - one of the best Go players of the last few years. The objective here is to win the game, our state is the position of all the pieces, the action is where to put the next piece down, and the reward is, 1 if you win at the end of the game, and 0 otherwise. We'll talk about this one in a little bit more detail, later.

## Mathematically Formalize the RL Problem

First we need to mathematically formalize the RL problem - this loop that we talked about of environments giving agents states, and then agents taking actions. For this we will use a Markov Decision Process.

# Markov Decision Process

## Object Definitions

- Mathematical formulation of the RL problem
- **Markov property**: Current state completely characterises the state of the world

Defined by:  $(\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathbb{P}, \gamma)$

$\mathcal{S}$  : set of possible states

$\mathcal{A}$  : set of possible actions

$\mathcal{R}$  : distribution of reward given (state, action) pair

$\mathbb{P}$  : transition probability i.e. distribution over next state given (state, action) pair

$\gamma$  : discount factor

A Markov decision process (MDP) is the mathematical formulation of the RL problem, and an MDP satisfies the *Markov property*, which is that the *current state completely characterizes the state of the world*.

An MDP is defined by tuple of objects, consisting of:

- $\mathcal{S}$ , which is the *set of possible states*.
- $\mathcal{A}$ , our *set of possible actions*.
- $\mathcal{R}$ , our *distribution of our reward*, given a state, action pair. So it's a function mapping from state action to your reward.
- $\mathbb{P}$ , which is a *transition probability distribution* over your next state, that you're going to transition to given your state, action pair.
- $\gamma$ , a *discount factor*, which is basically saying *how much we value rewards coming up soon versus later on*.

## The Process

- At time step  $t=0$ , environment samples initial state  $s_0 \sim p(s_0)$
  - Then, for  $t=0$  until done:
    - Agent selects action  $a_t$
    - Environment samples reward  $r_t \sim R(\cdot | s_t, a_t)$
    - Environment samples next state  $s_{t+1} \sim P(\cdot | s_t, a_t)$
    - Agent receives reward  $r_t$  and next state  $s_{t+1}$
  - A policy  $\pi$  is a function from  $S$  to  $A$  that specifies what action to take in each state
  - **Objective:** find policy  $\pi^*$  that maximizes cumulative discounted reward:  $\sum_{t \geq 0} \gamma^t r_t$
- 
- At our initial time step  $t=0$ , the environment is going to sample some initial state as 0, from the *initial state distribution*,  $p(s_0)$ .
  - Once it has that, from time  $t=0$  until it's done, we're going to iterate through this loop
    - Agent *selects an action*,  $a_t$ .
    - Environment *samples a reward* given your state and the action that you just took.
    - *Samples the next state*, at time  $t+1$ , given your probability distribution.
    - Agent is going to *receives the reward*, as well as the *next state*.

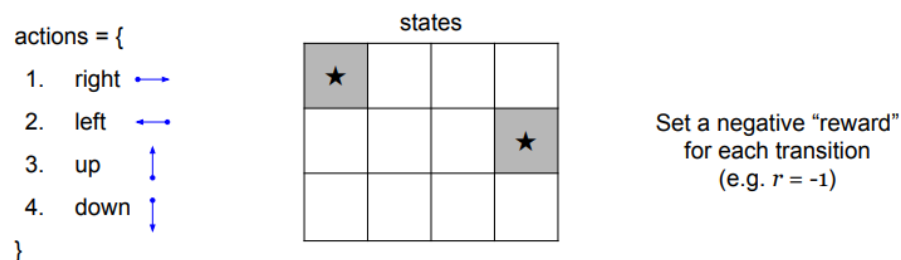
Continue looping through this process; agent will select the next action, and so on until the episode is over.

Based on this, we can define a *policy*  $\pi$ , which is a function that *specifies what action to take in each state*. This can be either deterministic or stochastic.

Our objective now is to going to be to find your *optimal policy*  $\pi^*$ , that *maximizes your cumulative discounted reward*.

We can see here we have our some of our future rewards, which can be also discounted by your *discount factor* ( $\gamma$ ).

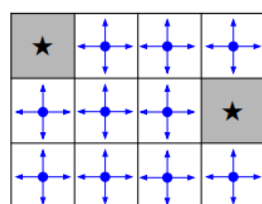
## Example MDP: Grid World



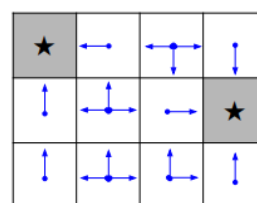
**Objective:** reach one of terminal states (greyed out) in least number of actions

Let's look at an example of a simple MDP. Here we have Grid World, a task where we have a grid of states. You can be in any of these cells of your grid, which are your *states*. And you can take actions from your states, these actions are going to be simple movements, moving to your right, to your left, up or down. You're going to get a *negative reward for each transition or each time step*. For each movement that you take your reward will be something like  $r = -1$ . So your objective is going to be to reach one of the terminal states, which are the gray states shown here, in the least number of actions.

The longer that you take to reach your terminal state, the more negative rewards you will accumulate.



Random Policy



Optimal Policy

Here, a random policy would consist of, taking any given state or cell that you're in just sampling randomly which direction that you're going to move in next. All of these have equal probability.

On the other hand, an optimal policy would take the action or direction that will move us closest to a terminal state. You can see here, if we're right next to one of the terminal states we should always move in the direction that gets us to this terminal state. And otherwise, if you're in one of these other states, you want to take the direction that will take you closest to one of these states.



## The Optimal Policy, $\pi^*$

We want to find optimal policy  $\pi^*$  that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?

Given this description of our MDP, what we want to do is we want to find our optimal policy,  $\pi^*$ . Our policy that maximizes the sum of the rewards. This optimal policy is going to tell us, given any state that we're in, what is the action that we should take in order to maximize the sum of the rewards that we'll get.

One question is how do we handle the randomness in the MDP. We have randomness in terms of our initial state that we're sampling, in terms of the *transition probability distribution* that will give us distribution of our next states, and so on.

Also we want to work on maximizing our expected sum of the rewards.

We want to find optimal policy  $\pi^*$  that maximizes the sum of rewards.

How do we handle the randomness (initial state, transition probability...)?

Maximize the **expected sum of rewards!**

$$\text{Formally: } \pi^* = \arg \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | \pi \right] \text{ with } s_0 \sim p(s_0), a_t \sim \pi(\cdot | s_t), s_{t+1} \sim p(\cdot | s_t, a_t)$$

We can write our optimal policy  $\pi^*$  as *maximizing the expected sum of future rewards* over policy's  $\pi$ , where

- initial state,  $s_0$ , is sampled from our state distribution
- actions,  $a_t$ , is sampled from our policy, given the state
- next states,  $s_{t+1}$ , is sampled from our transition probability distributions

## Value Function and Q-Value Function

Following a policy produces sample trajectories (or paths)  $s_0, a_0, r_0, s_1, a_1, r_1, \dots$

Before talking about how to find the optimal policy, let's first talk about a few definitions that will be helpful for us in doing so. Specifically, the **value function** and the **Q-value function**.

As we follow the policy, we're going to sample *trajectories* or paths, for every episode (path to terminal state). The initial state will be 0, and we have values  $a_0, r_0, s_0, a_1, r_1$ , and so on.

We'll have a trajectory of states, actions, and rewards for each policy. So we want to know, how good is a state that we're currently in?

#### How good is a state?

The **value function** at state  $s$ , is the expected cumulative reward from following the policy from state  $s$ :

$$V^\pi(s) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, \pi \right]$$

The *value function* at any state  $s$ , is the expected cumulative reward following the policy from state  $s$ , from here until termination.

It's the value of our expected cumulative reward, starting from our current state.

#### How good is a state-action pair?

The **Q-value function** at state  $s$  and action  $a$ , is the expected cumulative reward from taking action  $a$  in state  $s$  and then following the policy:

$$Q^\pi(s, a) = \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

Then how good is a state-action pair? As in, how good is taking action  $a$  in state  $s$ ?

We define this using a *Q-value function*, which is the expected cumulative reward from taking action  $a$  in state  $s$ , and then following the policy.

## Bellman Equation

The optimal Q-value function  $Q^*$  is the maximum expected cumulative reward achievable from a given (state, action) pair:

$$Q^*(s, a) = \max_{\pi} \mathbb{E} \left[ \sum_{t \geq 0} \gamma^t r_t | s_0 = s, a_0 = a, \pi \right]$$

The optimal Q-value function that we can get is going to be  $Q^*$ , which is the maximum expected cumulative reward that we can get from a given state-action pair, defined here.

$Q^*$  satisfies the following **Bellman equation**:

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

$Q^*$  is the optimal Q-value function which, by definition, will satisfy the Bellman equation defined above.

Verbally, this means is that given any state-action pair, we can determine the value of this pair as:

the reward that you will get by taking the action  $a$   
+  
the value of whatever state that you end up in

Let's say the state you end up in is  $s'$  ("s prime"). Since we know that we have the optimal policy, we know that we're going to play the best action that we can at our state,  $s'$ .

So like before with our previous state-action pair, the Q-value of the next state-action pair, is going to be the maximum over our actions,  $a'$ , given state  $s'$ .

This is how we get the identity above for the optimal Q-value. Again we have an *expectation* of the value in the definition, because we have randomness over what state that we're going to end up in.

**Intuition:** if the optimal state-action values for the next time-step  $Q^*(s', a')$  are known, then the optimal strategy is to take the action that maximizes the expected value of  $r + \gamma Q^*(s', a')$

We can infer from here that our optimal policy is going to consist of taking the best action in any state, as specified by  $Q^*$ .

$Q^*$  is going to tell us the maximum future reward that we can get from any of our actions, so we should take a policy that's following this and taking actions that are going to lead to best reward.

The optimal policy  $\pi^*$  corresponds to taking the best action in any state as specified by  $Q^*$

## Solving for the Optimal Policy, $\pi^*$

**Value iteration** algorithm: Use Bellman equation as an iterative update

$$Q_{i+1}(s, a) = \mathbb{E} \left[ r + \gamma \max_{a'} Q_i(s', a') | s, a \right]$$

$Q_i$  will converge to  $Q^*$  as  $i \rightarrow \text{infinity}$

So how can we solve for this optimal policy? One way is by a *value iteration algorithm*, where we use the Bellman equation as an iterative update.

At each step, we refine our approximation of  $Q^*$  by trying to enforce the Bellman equation. Under some mathematical conditions, we know that this sequence  $Q_i$  of our Q-function is going to converge to our optimal  $Q^*$  as  $i$  approaches infinity.

This works well, but there is a problem in that this is not scalable. We have to compute  $Q(s, a)$  for every state-action pair in order to make our iterative updates. This is a problem if the state space is too large, like in the example of an Atari game where the state space is the screen of pixels. This is a huge state space, and it's basically computationally infeasible to compute this for the entire state space.

So what's the solution to this? We can use a function approximator to estimate  $Q(s, a)$  for example, a neural network! If we have some really complex function that we want to estimate, a neural network is a good way to do this.


## Q-Learning

Q-learning: Use a function approximator to estimate the action-value function

$$Q(s, a; \theta) \approx Q^*(s, a)$$

Using a function approximator in order to estimate our action value function is known as Q-learning. If this function approximator is a deep neural network, which is what's been used recently, then this is going to be called **deep Q-learning**.

$$Q(s, a; \theta) \approx Q^*(s, a)$$


 function parameters (weights)

As we are using a neural network, we now include function parameters  $\theta$  in our definition here as our Q-value function is determined by these weights,  $\theta$ , of our neural network.

Now that we are using a neural network as a function approximation, how do we solve for our optimal policy?

Remember: want to find a Q-function that satisfies the Bellman Equation:

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

Remember that we want to find a Q-function that satisfies the Bellman equation.

### Forward Pass

Loss function:  $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where  $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

So when we have this neural network approximating our Q-function we can train with a loss function that is going to try and minimize the error of our Bellman equation. Or how far  $Q(s, a)$  is from its target. The target,  $y_i$ , has therefore been set to the definition of the Bellman equation.

### Backward Pass

Gradient update (with respect to Q-function parameters  $\theta$ ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[ (r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i)) \nabla_{\theta_i} Q(s, a; \theta_i) \right]$$

So, we're going to take these forward passes of our loss function, try to minimize this error and then our backward pass, our gradient update, is going to take the gradient of this loss, with respect to our network parameters  $\theta$ .

To reiterate, our goal is to have the effect of taking gradient steps to iteratively make our Q-function closer to our target value,  $y_i$ . The Q-function that maps directly to this target value will, by our definition be our optimal Q-function,  $Q^*$ , and so will provide us with the optimal policy,  $\pi^*$ .

## Case Study: Atari Games

### Case Study: Playing Atari Games



**Objective:** Complete the game with the highest score

**State:** Raw pixel inputs of the game state

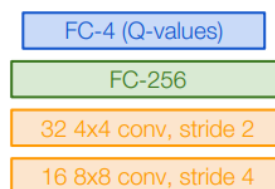
**Action:** Game controls e.g. Left, Right, Up, Down

**Reward:** Score increase/decrease at each time step

Let's look at a case study of an example where one of the classic examples of deep reinforcement learning where this approach was applied. Atari games have the objective of completing the game with the highest score. Their state is the raw pixels of the game state, and the actions are usually limited to the movements: left, right, up, and down. The reward at each time step can be a score increase or decrease, and the cumulative total reward is this total reward that is usually seen at the top of the screen.

## Q-Network Architecture

$Q(s, a; \theta)$ :  
neural network  
with weights  $\theta$



**Current state  $s_i$ : 84x84x4 stack of last 4 frames**  
(after RGB->grayscale conversion, downsampling, and cropping)

The network architecture will look something like the above, imagine a Q-network with weights,  $\theta$ . The input, or state  $s$ , will be the current game screen. In practice it's required to take a stack of the last four frames or so to gather information about movement on the screen. From these raw pixel values, apply some RGB to gray-scale conversions,

down-sampling, cropping, etc. whatever preprocessing is required. Out of this an 84 by 84 by four stack of the last four frames is produced.

On top of this input to the network, there will be a couple of convolutional layers, one eight-by-eight, then one four-by-four, and a fully-connected layer. This is just a standard kind of network that you may have seen before. Finally, the last fully-connected layer has a vector of outputs, which corresponds to the Q-value for each action given the state that you've input.

In this example, there are four possible actions, so there is a four-dimensional output corresponding to  $Q(s, a_1)$ ,  $Q(s, a_2)$ ,  $Q(s, a_3)$ ,  $Q(s, a_4)$ . This is going to be one scalar value for each of our actions.

A nice thing about using this network structure it's really efficient. A single feedforward pass is able to compute the Q-values for all actions from the current state.

## Training the Q-Network: Loss Function

Remember: want to find a Q-function that satisfies the Bellman Equation:

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

**Forward Pass**

Loss function:  $L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot)} [(y_i - Q(s, a; \theta_i))^2]$

where  $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a \right]$

Iteratively try to make the Q-value close to the target value ( $y_i$ ) it should have, if Q-function corresponds to optimal  $Q^*$  (and optimal policy  $\pi^*$ )

**Backward Pass**

Gradient update (with respect to Q-function parameters  $\theta$ ):

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s, a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[ r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s, a; \theta_i) \right] \nabla_{\theta_i} Q(s, a; \theta_i)$$

Remember that the goal is to to enforce the Bellman equation this is used as the loss function, trying to iteratively make the Q-value as close to this target value as possible.

The backward pass calculates the gradient of this loss function and then takes a gradient step based on that.

## Training the Q-Network: Experience Replay

In the above described architecture learning from batches of consecutive samples is problematic because:

1. all of the samples are correlated so it leads to inefficient learning
2. The current Q-network parameters determine the next training samples and if, for example, the current maximizing value action is to move left, this will bias all of the upcoming training examples to be dominated by samples from the left-hand side, potentially leading to bad feedback loops

Experience replay addresses this problem by keeping a **replay memory** table that is continuously updated with transitions of state ( $s_t, a_t, r_t, s_{t+1}$ ) as game episodes are played and experience of play increases.

In this way the Q-network can be trained on random mini-batches of transitions from the replay memory instead of using consecutive samples. This way the system can generate random samples from the table of the transitions that have been accumulated. In this way it addresses the problems caused by correlated samples, and means that each of these transitions can also contribute to potentially multiple weight updates leading to greater data efficiency.

Putting it all together: Deep Q-Learning with Experience Replay

---

**Algorithm 1** Deep Q-learning with Experience Replay

---

```

Initialize replay memory  $\mathcal{D}$  to capacity  $N$ 
Initialize action-value function  $Q$  with random weights
for episode = 1,  $M$  do
  Initialise sequence  $s_1 = \{x_1\}$  and preprocessed sequenced  $\phi_1 = \phi(s_1)$ 
  for  $t = 1, T$  do
    With probability  $\epsilon$  select a random action  $a_t$ 
    otherwise select  $a_t = \max_a Q^*(\phi(s_t), a; \theta)$ 
    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  $\mathcal{D}$ 
    Sample random minibatch of transitions  $(\phi_j, a_j, r_j, \phi_{j+1})$  from  $\mathcal{D}$ 
    Set  $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$ 
    Perform a gradient descent step on  $(y_j - Q(\phi_j, a_j; \theta))^2$  according to equation 3
  end for
end for

```

---

The algorithm starts by initializing the replay memory to some capacity that we choose, and the Q-network is initialized with random weights.

From there  $M$  episodes, or full games, will be played. These are the training episodes.

The state will be initialized using the starting game screen pixels at the beginning of each episode. (There will be a pre-processing step to get to the actual input state).

For each time step of a game that is currently being playing, there is a small probability that a random action will be selected. Otherwise a selection will be made from the greedy actions of the current policy. Greedy actions are those that are predicted to give the highest reward. In most cases a greedy action will be chosen, but the chance of a random action being selected is important as it is necessary to make sure that different parts of the state space are being sampled.



The selected action is taken,  $a_t$ , and the reward  $r_t$  and the next state  $s_{t+1}$  is received.

This transition will be stored in the replay memory that is being built up.

Next the network will be trained using experience replay. A sample of a random mini-batch of transitions from the replay memory will be generated, and gradient descent step will be performed on this.

This is the full training loop.

The loop will continue with the game repeatedly played and samples of mini-batches generated from the replay memory, used to update the weights of the Q-network.

## Example: Google DeepMind Atari Breakout

Google DeepMind trained an Atari game of Breakout in this way. Checkout the YouTube video at <https://www.youtube.com/watch?v=V1eYniJ0Rnk> to see how it performs at different levels of training. After 240 minutes of training it's found the pro strategy, to tunnel through the blocks and get all the way to the top, then have it go by itself.

## The Problems with Q-Learning

Q-learning is very complicated due to the requirement of having to learn the value of every state-action pair. For example, if you have the problem of a robot trying to grasp an object, this requires dealing with a really high dimensional state. There will be all of the joints, joint positions, and angles etc. Due to its high dimensional environment, learning the exact value of every state-action pair that you have can be really hard to do.

However, your optimal policy for deciding actions within this problem or environment might be much simpler, perhaps considering only the simple motion of the robot just opening and closing its hand, or just moving its fingers in a particular direction and whether to keep going.

This leads to the question of whether a policy can be learnt directly. Is it possible to just find the best policy from a collection of policies? Instead of trying to go through this process of estimating a Q-value and then using *that* to infer the policy.