Function Approximators

Scaling Reinforcement Learning Algorithms to High-Dimensional Problems

Matthia Sabatelli

December 10, 2021

Recap

Last week we have seen ...

Model-Free Reinforcement Learning

- How to construct algorithms when parts of the MDP $\mathcal{M}$ are unknown
- Monte-Carlo (MC) Learning
- Temporal-Difference (TD) Learning
- The concept of bootstrapping
- How to learn $Q^{\pi}(s, a)$ in an *on-policy* fashion or in an *off-policy* fashion

Today's Agenda

1. Function Approximation
2. Linear Functions
3. Neural Networks

Function Approximation

The algorithms that we have seen last week are typically implemented in a tabular fashion:

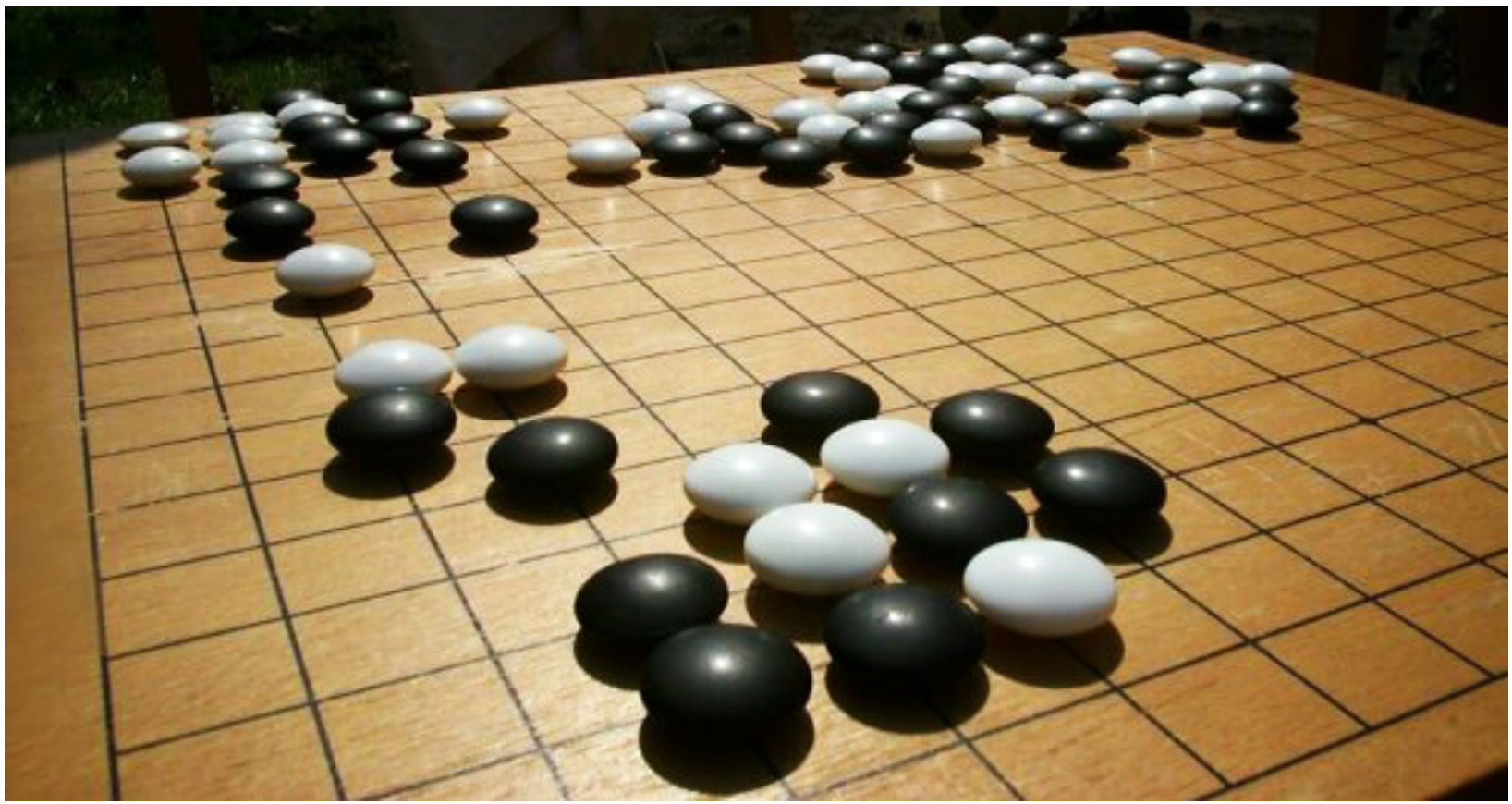
- If the goal is to learn $V^{\pi}(s)$ we use a table of size $|\mathcal{S}|$
- If the goal is to learn $Q^{\pi}(s, a)$ we use a table of size $|\mathcal{S} \times \mathcal{A}|$

It is easy to see the limitations of this approach:

1. Unfeasible when the state-action space is large
2. Impossible to use in a continous setting
3. Requires a discretization of the environment
4. Lacks generalization

Function Approximation

A simple example: the game of Go



Function Approximation

A simple example: the game of Go

- The AlphaGo and AlphaZero programs are a typical example of the recent success of Reinforcement Learning
- Both programs heavily rely on a function approximator

Why?

- The size of the Go board is 19×19
- On each location there can, or can't, be a stone (white or black)
- State space $|\mathcal{S}| = 3^{19 \times 19} = 3^{361}$

Impossible to learn any value function!

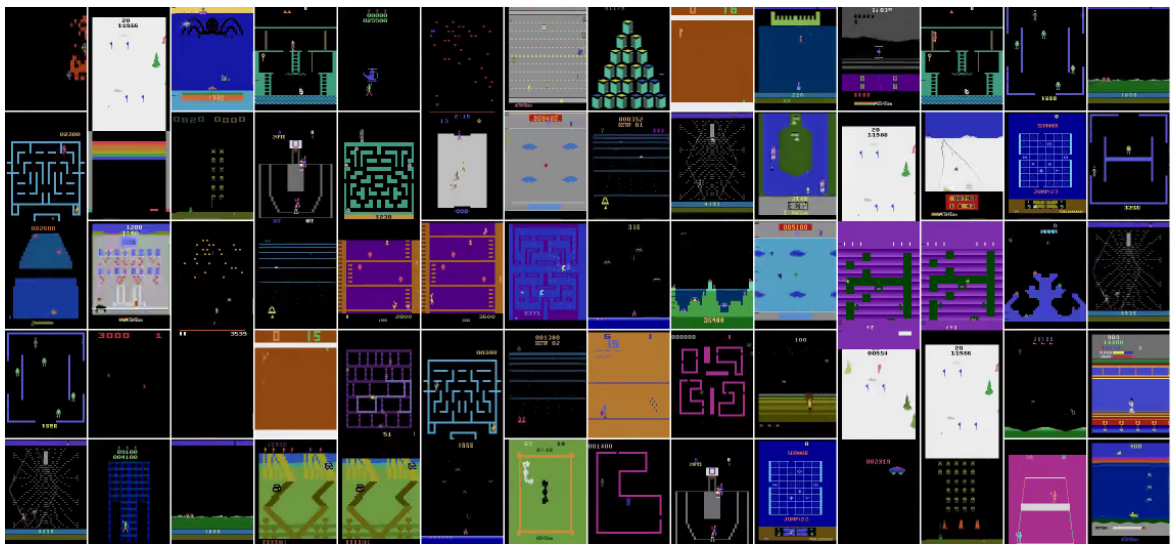
Function Approximation

Another example: the Atari-2600 benchmark

Function Approximation

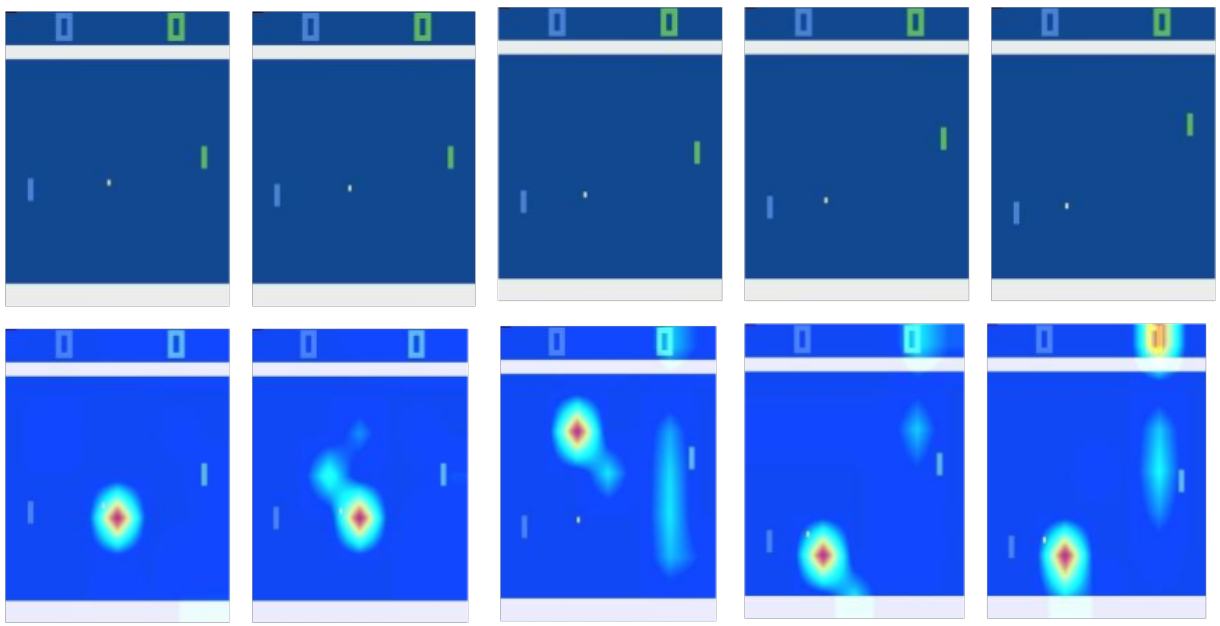
Another example: the Atari-2600 benchmark

- The first Deep Reinforcement Learning agent of mastering these games is based on a Convolutional Neural Network
- The network was trained on the raw input pixels of the game



Function Approximation

Let's take a look at some states of the `Pong` game:



- All states look very similar among each other
- Small portions of the state-space are actually informative
- Despite some pre-processing operations the state-space stays highly dimensional

Function Approximation

Fortunately we can overcome the aforementioned issues by including a function approximator in the reinforcement learning cooking recipe!

Function Approximators

We do not represent a value function as a table anymore, but rather with a parametrized functional form with weight vector $\mathbf{w} \in \mathbb{R}^d$.
Therefore we are now interested in learning:

$$V^{\pi}(s) \approx \hat{V}^{\pi}(s; \mathbf{w})$$

Function Approximation

We will see that $\hat{V}(s; \mathbf{w})$ can come in numerous forms but the key ideas behind using a function approximator are always the same:

1. We want to overcome the computational burdens that come from large state $|\mathcal{S}|$ and action $|\mathcal{A}|$ spaces
2. We wish to represent states through informative features
3. Ideally we would like to learn an approximation of a value function which generalizes well across states

$\Rightarrow$ Now that we have built some intuition around why function approximation is useful let us dive deeper into this family of techniques ...

Linear Functions

Any function approximator comes is the following form:

$$y = f(\mathbf{x}; \mathbf{w})$$

The easiest form of a function approximator that we can use is a linear function which is linear in the components of $\mathbf{x}$:

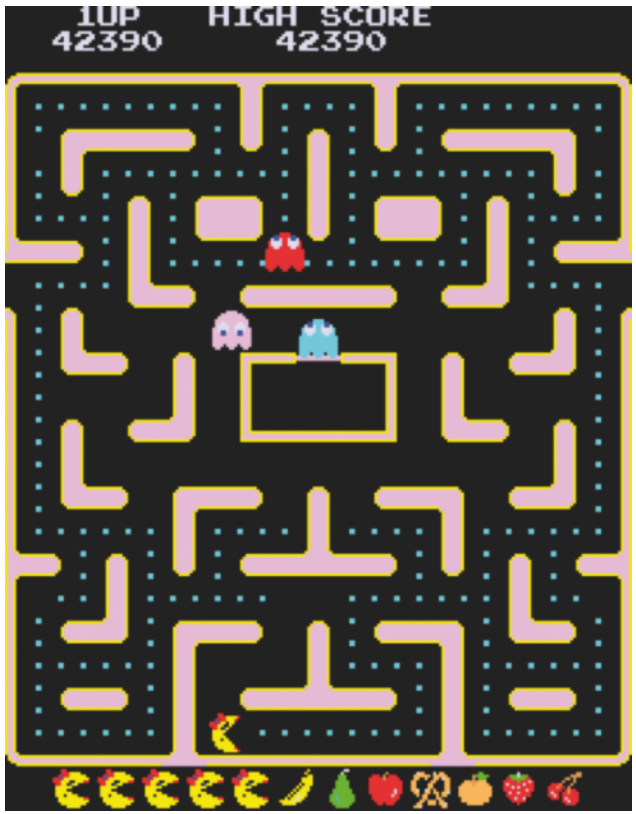
For example we can represent the value of a state as a two-dimensional feature vector:

$$\mathbf{x} = [f_1(s), f_2(s), ..., f_i(s)] \in \mathbb{R}^2$$

Linear Functions

Quiz Time!

Given the popular `Pacman` game, which features would you consider to be the most representative ones of the game?



Linear Functions

So far we have seen:

1. Why it is desirable to have a function approximator
2. One of the possible forms of such function
3. How crucial it is to carefully go through a process of feature engineering

However ...

How do we train these functions?

Linear Functions

We use Stochastic Gradient Descent (SGD):

- We have a weight vector $\mathbf{w} \doteq (w_1, w_2, ..., w_d)$ of real values parametrizing our function approximator
- The approximate value function $\approx \hat{V}^\pi(s; \mathbf{w})$ is a differentiable function of $\mathbf{w}$ for all $s \in \mathcal{S}$
- As SGD works in an iterative fashion we will make use of the underscript t denoting $\mathbf{w}_t$

Linear Functions

A simple example:

- We wish to evaluate the state-value function $V^\pi(s)$
- We assume that the correct values of a state are known
- We know the true value of a state under policy π

SGD in practice

We wish to minimize the estimates made by $\hat{V}(s;\mathbf{w})$ with respect to $V^\pi(s)$:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\Big[V^\pi(s_t) - \hat{V}(s;\mathbf{w}_t)\Big]^2 \\ &= \mathbf{w}_t + \alpha\Big[V^\pi(s_t) - \hat{V}(s_t;\mathbf{w}_t)\Big]\nabla\hat{V}(s;\mathbf{w}_t)\end{aligned}$$

where α is the step-size and $\nabla f(\mathbf{w})$ is the vector containing all partial derivatives wrt to the components of $\mathbf{w}$:

$$\nabla f(\mathbf{w}) \doteq \Big(\frac{\partial f(\mathbf{w})}{\partial w_1}, \frac{\partial f(\mathbf{w})}{\partial w_2}, \ldots, \frac{\partial f(\mathbf{w})}{\partial w_d}\Big).$$

Linear Functions

So far ...

- We have considered the case where we have access to the true value of a state $V^{\pi}(s)$
- But in practice this is never the case (see MC and TD-Learning)
- How to deal with the problem of control?

Let's see how to learn the state-action value function $\hat{Q}(s, a; \mathbf{w})$

Linear Functions

We follow the same steps that we have used beforehand for learning $\hat{V}(s; \mathbf{w})$:

1. We take a model-free Reinforcement Learning algorithm of our choice
2. We construct an objective function we wish to minimize
3. Perform gradient descent on the parameter vector $\mathbf{w}$

Q-Learning with function approximation

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \Big[\underbrace{r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t)}_{y_t} - Q(s_t, a_t) \Big].$$

Linear Functions

We follow the same steps that we have used beforehand for learning $\hat{V}(s; \mathbf{w})$:

1. We take a model-free Reinforcement Learning algorithm of our choice
2. We construct an objective function we wish to minimize
3. Perform gradient descent on the parameter vector $\mathbf{w}$

Q-Learning with function approximation

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \Big[\underbrace{r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t)}_{y_t} - Q(s_t, a_t) \Big].$$

$$\mathcal{L} = \frac{1}{2} \Big(y_t - Q(s_t, a_t) \Big)^2$$

Linear Functions

We follow the same steps that we have used beforehand for learning $\hat{V}(s; \mathbf{w})$:

1. We take a model-free Reinforcement Learning algorithm of our choice
2. We construct an objective function we wish to minimize
3. Perform gradient descent on the parameter vector $\mathbf{w}$

Q-Learning with function approximation

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \Big[\underbrace{r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t)}_{y_t} - Q(s_t, a_t) \Big].$$

$$\mathcal{L} = \frac{1}{2} \Big(y_t - Q(s_t, a_t) \Big)^2$$

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = - \Big(y_t - Q(s_t, a_t) \Big) \mathbf{x}(s_t)$$

Linear Functions

Linear functions allow us to overcome the curse of dimensionality issue, but:

1. Their representational power is still limited
2. Require a careful feature engineering stage

$\Rightarrow$ Therefore non-linear functions such as neural networks are preferred!

- They follow the same principles of linear functions
- Are powerful feature extractors
- Benefit from being universal function approximators

Are **extremely hard and slow** to train!

Neural Networks

Welcome to the magical world of Deep Reinforcement Learning (DRL)!

- In DRL we use deep neural networks as non-linear function approximators
- However, if a single hidden layer network is used we talk about ⇒ Connectionist Reinforcement Learning
- But typically Convolutional Neural Networks are preferred ⇒ Deep Reinforcement Learning

Neural Networks

Before AlphaGo and AlphaZero there was TD-Gammon:

1. The first computer program of mastering a boardgame
2. The first successful combination of Reinforcement Learning and neural networks
3. The first practical application of TD-Learning

$\Rightarrow$ Until $\approx$ 15 years ago TD-Gammon was arguably the most (and only) successful application combining neural networks and Reinforcement Learning

Neural Networks

Some Reinforcement Learning history ...

TD-Gammon, A Self-Teaching Backgammon Program, Achieves Master-Level Play

Gerald Tesauro
IBM Thomas J. Watson Research Center
P. O. Box 704
Yorktown Heights, NY 10598
(tesauro@watson.ibm.com)

Abstract. TD-Gammon is a neural network that is able to teach itself to play backgammon solely by playing against itself and learning from the results, based on the TD(λ) reinforcement learning algorithm (Sutton, 1988). Despite starting from random initial weights (and hence random initial strategy), TD-Gammon achieves a surprisingly strong level of play. With zero knowledge built in at the start of learning (i.e. given only a "raw" description of the board state), the network learns to play at a strong intermediate level. Furthermore, when a set of hand-crafted features is added to the network's input representation, the result is a truly staggering level of performance: the latest version of TD-Gammon is now estimated to play at a strong master level that is extremely close to the world's best human players.

Why did TD-Gammon Work?

Jordan B. Pollack & Alan D. Blair
Computer Science Department
Brandeis University
Waltham, MA 02254
{pollack,blair}@cs.brandeis.edu

Abstract

Although TD-Gammon is one of the major successes in machine learning, it has not led to similar impressive breakthroughs in temporal difference learning for other applications or even other games. We were able to replicate some of the success of TD-Gammon, developing a competitive evaluation function on a 4000 parameter feed-forward neural network, without using back-propagation, reinforcement or temporal difference learning methods. Instead we apply simple hill-climbing in a relative fitness environment. These results and further analysis suggest that the surprising success of Tesauro's program had more to do with the co-evolutionary structure of the learning task and the dynamics of the backgammon game itself.

Neural Networks

But in 2013 things started to change $\Rightarrow$ Deep-Q Networks (DQN) were introduced!

1. The community started to focus on using Convolutional Neural Networks
2. Powerful networks able of serving as function approximators as well as feature extractors
3. The first step was to adapt the Q-Learning algorithm

$\Rightarrow$ Let's see how DQN intuitively works!

Neural Networks

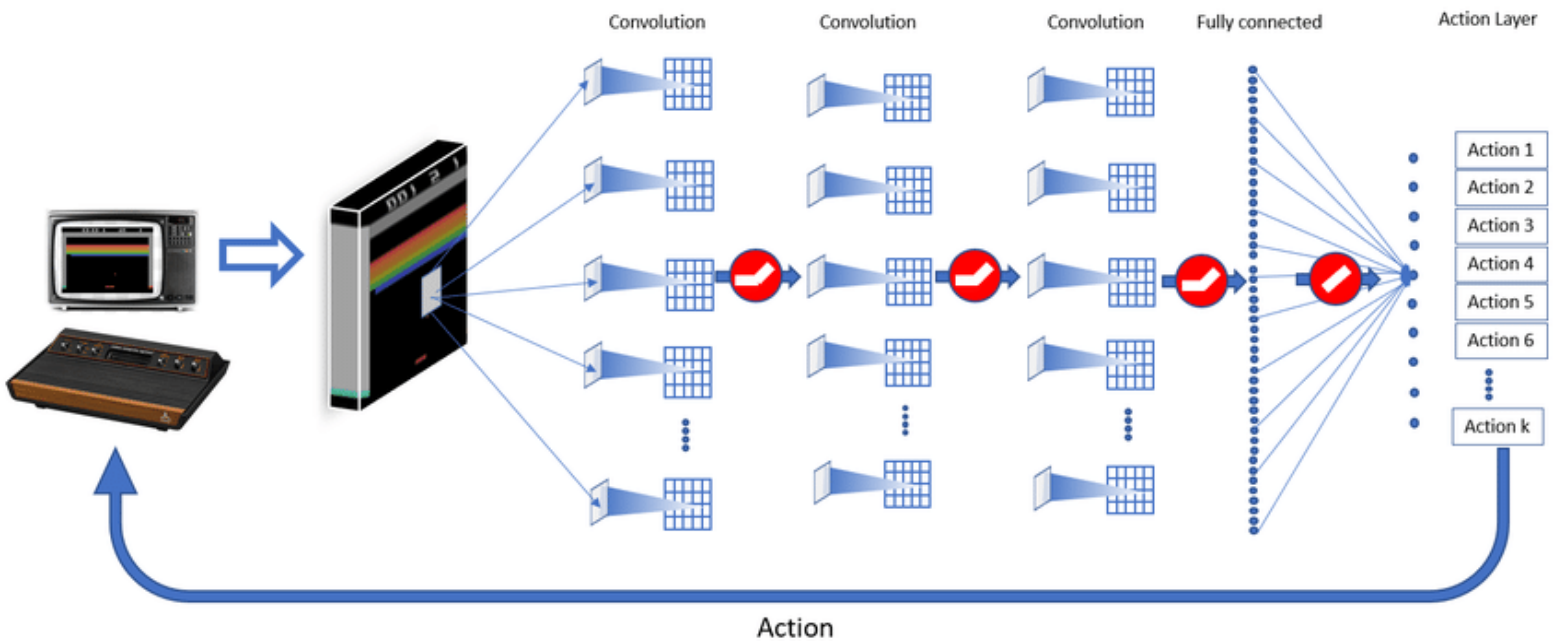


Figure: Image courtesy of Patel et al. (2019)

Neural Networks

How do we train such a system?

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

Neural Networks

How do we train such a system?

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

Neural Networks

How do we train such a system?

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

$\Rightarrow$ We reshape the Q-Learning algorithm into an objective function that can be used for learning θ

The DQN Algorithm

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t) - Q(s_t, a_t) \right].$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right].$$

Neural Networks

How do we train such a system?

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

$\Rightarrow$ We reshape the Q-Learning algorithm into an objective function that can be used for learning θ

The DQN Algorithm

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t) - Q(s_t, a_t) \right]$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

Neural Networks

How do we train such a system?

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

$\Rightarrow$ We reshape the Q-Learning algorithm into an objective function that can be used for learning θ

The DQN Algorithm

$$Q(s_t, a_t) := Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a_t) - Q(s_t, a_t) \right].$$

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right].$$

Neural Networks

A closer look at DQN's objective function:

The DQN Algorithm

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

1. Uses Q-Learning's TD-target: $y_t^{DQN} = r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-)$.
2. Requires an additional component called Experience Replay: D

$\Rightarrow$ We are reducing the reinforcement learning problem to a supervised learning problem ...

Neural Networks

$\Rightarrow$ The idea of Experience Replay is to collect RL trajectories $\tau\langle s_t, a_t, r_t, s_{t+1}\rangle$ and then use them for minimizing $\mathcal{L}$

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

$\Rightarrow$ We are constructing a dataset of experiences ...

Neural Networks

At training time we uniformly sample $\sim U(D)$ from the buffer and construct a mini-batch of samples for learning

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Neural Networks

At training time we uniformly sample from the buffer and construct a mini-batch of trajectories for learning

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

Neural Networks

At training time we uniformly sample from the buffer and construct a mini-batch of trajectories for learning

$$D = \begin{pmatrix} s_t & a_t & r_t & s_{t+1} \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & r_t & \vdots \\ s_t & a_t & \vdots & \vdots \\ s_t & \vdots & \vdots & \vdots \\ \vdots & \vdots & \vdots & \vdots \end{pmatrix}$$

The DQN Algorithm

$$\mathcal{L}(\theta) = \mathbb{E}_{\langle s_t, a_t, r_t, s_{t+1} \rangle \sim U(D)} \left[\left(r_t + \gamma \max_{a \in \mathcal{A}} Q(s_{t+1}, a; \theta^-) - Q(s_t, a_t; \theta) \right)^2 \right]$$

Neural Networks

$\Rightarrow$ The Experience Replay memory buffer plays a crucial role in DRL: without it, it would be impossible to train an agent

1. It makes agents more sample efficient ✓
2. Helps generalization ✓
3. Goes against the principle of online learning ✗
4. Reinforcement learning $\rightarrow$ Supervised learning ✗

Is only one among the many "tricks" that are necessary if we want to successfully combine Reinforcement Learning algorithms with deep neural networks.

Final Slide!

Lecture Takeaway

- How to deal with MDPs where the the state space $|\mathcal{S}|$ and the action space $|\mathcal{A}|$ are large
- Why it is desirable to introduce function approximators into the Reinforcement Learning loop
- Linear vs non-linear functions
- Connectionist vs Deep Reinforcement Learning
- Deep-Q Networks