

ECE 1508: Reinforcement Learning

Chapter 4: Function Approximation

Ali Bereyhi

ali.bereyhi@utoronto.ca

Department of Electrical and Computer Engineering
University of Toronto

Summer 2024

Review: Where Are We Now?

Model-Based RL

Bellman Equation

value iteration

policy iteration

Model-free RL

on-policy methods

temporal difference

Monte Carlo

SARSA

off-policy methods

Q-learning

We need to overcome one last challenge

we need to learn how to deal with large-scale problems

Tabular RL: What We Had Already

What we have studied up to now is usually called **tabular RL**

- + Why we call it **tabular**?
 - Because we think of **value** and **action-value** function as a **table**

Let's consider a mode-free control loop with **finite number of states** and **actions**

- We initiate all **action-values** with zero
 - ↳ We initiate a **table of zeros**
- But, we do know that these **action-values** are some specific values
 - ↳ We are dealing with a **table of unknowns**
- We try **estimating** them from samples

Tabular RL: Schematic

	a^1	a^2	...	a^M
s^1	$\hat{q}_\pi(s^1, a^1)$	$\hat{q}_\pi(s^1, a^2)$		
s^2				
\vdots				
s^N				$\hat{q}_\pi(s^N, a^M)$

Q-table

s^1	$\hat{v}_\pi(s^1)$
s^2	$\hat{v}_\pi(s^2)$
\vdots	\vdots
s^N	$\hat{v}_\pi(s^N)$

value-table

Computational Complexity of Tabular RL

A tabular approach needs estimation of NM values from samples

- Say we use Monte-Carlo

- ↳ We need to visit all the state-action pairs enough times
 - ↳ Say enough is C for us; then, we need

$$\# \text{ sample pairs in all episodes} \approx CNM$$

- Same thing with temporal difference

Moral of Story

In tabular RL methods, the number of required sample interactions with the environment scales with **number of states** and **number of actions**

Complexity Examples: Backgammon

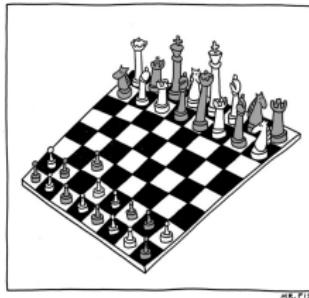


There are roughly 10^{20} possible states for backgammon

- Say we can get over with each state-action pair and update in only 1 Picosec = 10^{-12} sec; then, we need

$$\# \text{ time} \approx 10^8 CM \text{ sec} \approx 3.2 CM \text{ years}$$

Complexity Examples: Chess



In the fun part of Assignment 1, you saw that Shannon found out about roughly 10^{120} possible states for chess, and later on some people came out with some approximations for legal positions: let's take Tromp's number, i.e., $\approx 4 \times 10^{44}$

- Say again we need only 1 Picosec per step; then, we need

$$\# \text{ time} \approx 4 \times 10^{32} \text{ CM sec} \approx 12 \times 10^{24} \text{ CM years}$$

Milky Way is about 13.6 billion years old!

Complexity Examples: Game Go

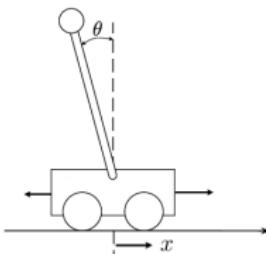


There are roughly 10^{170} possible states for board game Go

- *Say again we need only 1 picosec*

time \approx do we really need computing it? 😊

Computational Complexity: Continuous State Space



We saw the Cart-Pole problem: if we want to put it into a tabular form

- We need to put it into a **table**
 - ↳ We have to make a **discrete grid** of states
 - ↳ Say we have L state parameters and we grid each one with B bins
 - ↳ We have B^L grid points in total
- This grows **exponentially** large in number of state parameters!

Computational Complexity of Tabular RL

In most interesting problems, we are dealing with **scaling problem**

- *We have an exponentially large number of states*
 - *We cannot even visit a subset of them!*
 - *Our tabular RL algorithms will not give us any meaningful results!*
-
- + *What then?! Should we always play 4×4 Frozen Lake with RL?!*
 - **No!** We try to **approximate our table** from our limited observations

Approximating Value Function

Let's start with a simple case: say we want to evaluate a policy, i.e.,

We are given with a policy π and want to find some estimate $\hat{v}_\pi(\cdot)$

In the *tabular* RL, we assume that

$$v_\pi(s^1) = v^1 \quad \dots \quad v_\pi(s^N) = v^N$$

for some unknown v^1, \dots, v^N and try to estimate them

Approximating Value Function

Let us now approach the problem differently: we assume that

$$v_{\pi}(s) = f(s, w)$$

for some weights in w and a known function $f()$

- $f()$ is a **approximation model** that we assume for the value function
 - w contains a set of **learnable parameters**
-
- + How on earth we know such a thing?!
 - We don't really know it! We just assume it; however, **sometimes it really makes sense**

Approximating Value Function

If we assume an **approximation model**: we use our observations to find weight vector w^* that fits this **approximator** best to our observations. We then set

$$\hat{v}_\pi(s) = f(s, w^*)$$

- + How is it better than tabular then?!
 - Well! For lots of reasons
- ① We can use every single sample to update the estimate for all states
 - ↳ We use samples to fit w : this impacts on the whole value function
- ② We get some **updating** estimates for states that have not been visited
 - ↳ If we get the right w : we get the estimator for **all** states
- ③ We can capture the impact of one state on the others
 - ↳ If we update w after seeing S_t : we change estimated values of **all** states

Approximating Value Function

- + But how can we find such approximators?!
- There are various types of them
 - Linear function approximators
 - Deep neural networks (DNNs)
 - Eigen-transforms, e.g., Fourier or Wavelets
 - ...

In this course, we are focusing only on *parametric approximators* which include

- Linear function approximators
- DNNs

Because they are *differentiable*: we will see why this is important!

But before using them, let's look at them and understand how they *work*

Recap: Vectors and Matrices

We're going to use frequently linear algebra! So, let's recall some basics

$x \in \mathbb{R}^N$ is an N -dimensional *column*-vector with N *real* entries, i.e.,

$$\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \rightsquigarrow \mathbf{x}^\top = [x_1, \dots, x_N]$$

If we want to make a *a row*-vector, we *transpose* it, i.e., use \mathbf{x}^\top

Recap: Vectors and Matrices

We're going to use frequently linear algebra! So, let's recall some basics

$x \in \mathbb{R}^N$ is an N -dimensional *column*-vector with N *real* entries, i.e.,

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \rightsquigarrow \boldsymbol{x}^\top = [x_1, \dots, x_N]$$

If we want to make a *a row*-vector, we *transpose* it, i.e., use \boldsymbol{x}^\top

Notation

We show vectors with **bold-face** small letters and drop column/row

- A vector is **by default** a *column*-vector
- If we need a *row*-vector, we *transpose* its *column* version

Recap: Vectors and Matrices

Matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ can be seen as

- either as the *collection of M column-vectors of dimension N*

$$\mathbf{A} = [\mathbf{a}_1, \dots, \mathbf{a}_M]$$

with $\mathbf{a}_m \in \mathbb{R}^N$ for $m = 1, \dots, M$

Recap: Vectors and Matrices

Matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ can be seen as

- either as the *collection of M column-vectors of dimension N*

$$\mathbf{A} = [\mathbf{a}_1, \dots, \mathbf{a}_M]$$

with $\mathbf{a}_m \in \mathbb{R}^N$ for $m = 1, \dots, M$

- or as the *collection of N row-vectors of dimension M*

$$\mathbf{A} = \begin{bmatrix} \mathbf{b}_1^\top \\ \vdots \\ \mathbf{b}_N^\top \end{bmatrix}$$

with $\mathbf{b}_n \in \mathbb{R}^M$ for $n = 1, \dots, N$

Recap: Vectors and Matrices

Notation

We show matrices with **bold-face** capital letters

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ of same dimension N are inner-multiplied as

$$\mathbf{x}^\top \mathbf{y} = \mathbf{y}^\top \mathbf{x} = \sum_{n=1}^N x_n y_n$$

Recap: Vectors and Matrices

Notation

We show matrices with **bold-face** capital letters

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ of same dimension N are inner-multiplied as

$$\mathbf{x}^\top \mathbf{y} = \mathbf{y}^\top \mathbf{x} = \sum_{n=1}^N x_n y_n$$

They can further outer-multiplied as

$$\mathbf{x}\mathbf{y}^\top = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \begin{bmatrix} y_1, \dots, y_N \end{bmatrix} = \begin{bmatrix} x_1 y_1 & \dots & x_1 y_N \\ \vdots & \dots & \vdots \\ x_N y_1 & \dots & x_N y_N \end{bmatrix}$$

Recap: Vectors and Matrices

Notation

We show matrices with **bold-face** capital letters

Two vectors $\mathbf{x}, \mathbf{y} \in \mathbb{R}^N$ of same dimension N are inner-multiplied as

$$\mathbf{x}^\top \mathbf{y} = \mathbf{y}^\top \mathbf{x} = \sum_{n=1}^N x_n y_n$$

They can further outer-multiplied as

$$\mathbf{x}\mathbf{y}^\top = \begin{bmatrix} x_1 \\ \vdots \\ x_N \end{bmatrix} \begin{bmatrix} y_1, \dots, y_N \end{bmatrix} = \begin{bmatrix} x_1 y_1 & \dots & x_1 y_N \\ \vdots & \dots & \vdots \\ x_N y_1 & \dots & x_N y_N \end{bmatrix} = (\mathbf{y}^\top \mathbf{x})^\top$$

Recap: Vectors and Matrices

Multiplying matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ with $\mathbf{x} \in \mathbb{R}^M$ can be seen as

- either as the *linear combination of column-vectors in \mathbf{A}*

$$\mathbf{Ax} = [\mathbf{a}_1, \dots, \mathbf{a}_M] \begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix} = \sum_{m=1}^M x_m \mathbf{a}_m \in \mathbb{R}^N$$

Recap: Vectors and Matrices

Multiplying matrix $\mathbf{A} \in \mathbb{R}^{N \times M}$ with $\mathbf{x} \in \mathbb{R}^M$ can be seen as

- either as the *linear combination of column-vectors in \mathbf{A}*

$$\mathbf{Ax} = [\mathbf{a}_1, \dots, \mathbf{a}_M] \begin{bmatrix} x_1 \\ \vdots \\ x_M \end{bmatrix} = \sum_{m=1}^M x_m \mathbf{a}_m \in \mathbb{R}^N$$

- or as the *collection of inner-products of row-vectors with \mathbf{x}*

$$\mathbf{Ax} = \begin{bmatrix} \bar{\mathbf{a}}_1^\top \\ \vdots \\ \bar{\mathbf{a}}_N^\top \end{bmatrix} \mathbf{x} = \begin{bmatrix} \bar{\mathbf{a}}_1^\top \mathbf{x} \\ \vdots \\ \bar{\mathbf{a}}_N^\top \mathbf{x} \end{bmatrix} \in \mathbb{R}^N$$

Recap: Linear versus Affine Function

Linear Function

Scalar $y \in \mathbb{R}$ is a linear function of vector $x \in \mathbb{R}^N$ if

$$y = w^T x$$

for some constant vector $w \in \mathbb{R}^N$

Recap: Linear versus Affine Function

Linear Function

Scalar $y \in \mathbb{R}$ is a linear function of vector $x \in \mathbb{R}^N$ if

$$y = w^T x$$

for some constant vector $w \in \mathbb{R}^N$

Affine Function

Scalar $y \in \mathbb{R}$ is an affine function of vector $x \in \mathbb{R}^N$ if

$$y = w^T x + b$$

for some constant vector $w \in \mathbb{R}^N$ and scalar $b \neq 0$

Recap: Linear versus Affine Function

Key difference: If $\mathbf{x} = \mathbf{0}_N$ is the vector of all zeros

- Linear function returns zero: *it passes through the origin*
- Affine function returns non-zero: *it does not pass through the origin*

Recap: Linear versus Affine Function

Key difference: If $\mathbf{x} = \mathbf{0}_N$ is the vector of all zeros

- Linear function returns zero: it passes through the origin
- Affine function returns non-zero: it does not pass through the origin

We can simply extend the definition to a **vector-valued** functions

Let $\mathbf{A} \in \mathbb{R}^{M \times N}$ and $\mathbf{b} \in \mathbb{R}^M$; then,

$$\mathbf{y} = \mathbf{A}\mathbf{x}$$

is a linear **vector-valued** function of \mathbf{x} and

$$\mathbf{y} = \mathbf{A}\mathbf{x} + \mathbf{b}$$

is an affine **vector-valued** function

Function Approximation: *Formulation*

In function approximation, we have

- A set of sample **input-outputs** of a function that we *do not know*

$$\mathbb{D} = \{(\textcolor{blue}{x}_i, \textcolor{red}{y}_i) : i = 1, \dots, I\}$$

- ↳ \mathbf{x}_i is the **input** and y_i is the **output**
 - ↳ Let's denote the unknown function with $g(\cdot)$, i.e., $y_i = g(\mathbf{x}_i)$
 - We assume an **approximating model** for this **unknown** function

$$\hat{y} = f(\textcolor{blue}{x} | \mathbf{w})$$

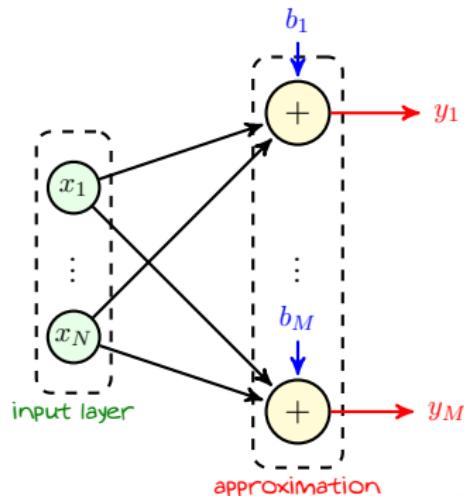
- ↳ In this **approximator** w is learnable, i.e., we are free to tune it as we wish
 - ↳ We treat output of this **approximator** as estimate of function outputs

Example: Linear Function Approximation

A simple example of a function approximator is the *linear approximator*

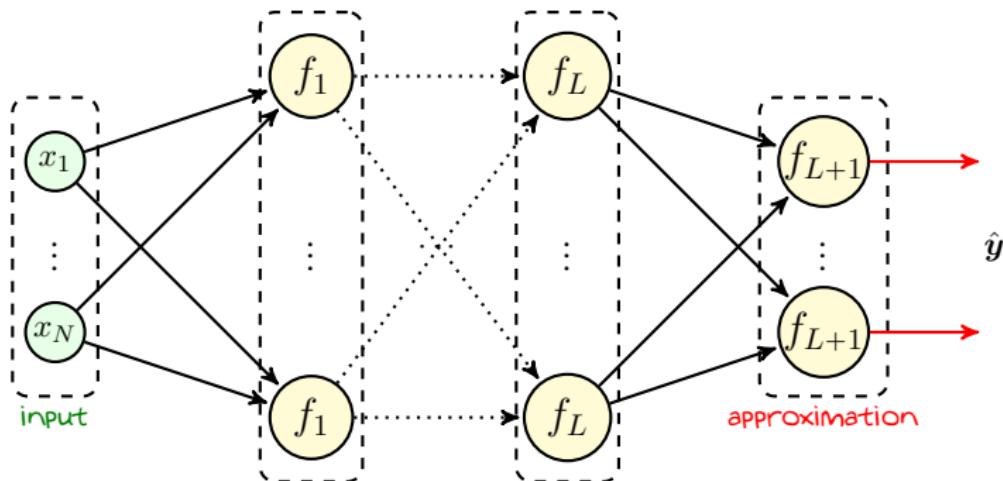
$$\hat{y} = \mathbf{W}\mathbf{x} + \mathbf{b}$$

- \mathbf{W} is the matrix of weights
- \mathbf{b} is the vector of biases



Example: Deep Neural Networks

A deep neural network is also a parameterized approximator



Example: Deep Neural Networks

- + Why should we rely on these approximators?
- They are known to be very powerful

Universal Approximation Theorem

Given a family for neural networks: for any function g from its function space, there exists a sequence of configurations that approximates g arbitrarily precise

Function Approximation: Formulation

With **parametric** approximators: we want to find weight w so that we have

- a **good approximator**: it fits best the **dataset**, i.e.,

$$\hat{y}_i = f(\mathbf{x}_i | \mathbf{w}) \approx y_i$$

↳ We need to define a notion for \approx

- an **approximator** that **generalizes**: if we get a new sample input \mathbf{x}_{new} , we somehow can make sure that

$$\hat{y}_{\text{new}} = f(\mathbf{x}_{\text{new}} | \mathbf{w}) \approx g(\mathbf{x}_{\text{new}})$$

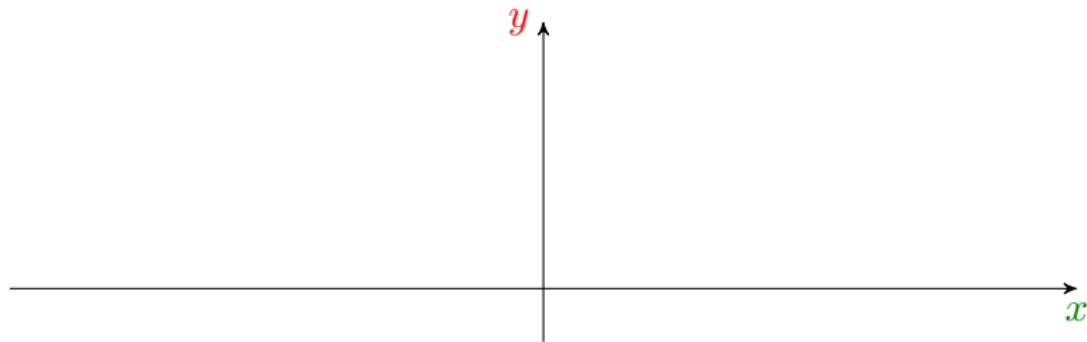
↳ We should find a way to check this

Visualizing Function Approximation

Let's assume one-dimensional inputs and outputs: *this assumption helps us visualize the function approximator*

$$y = f(\textcolor{green}{x} | \mathbf{w})$$

With scalar input, we can visualize the model as



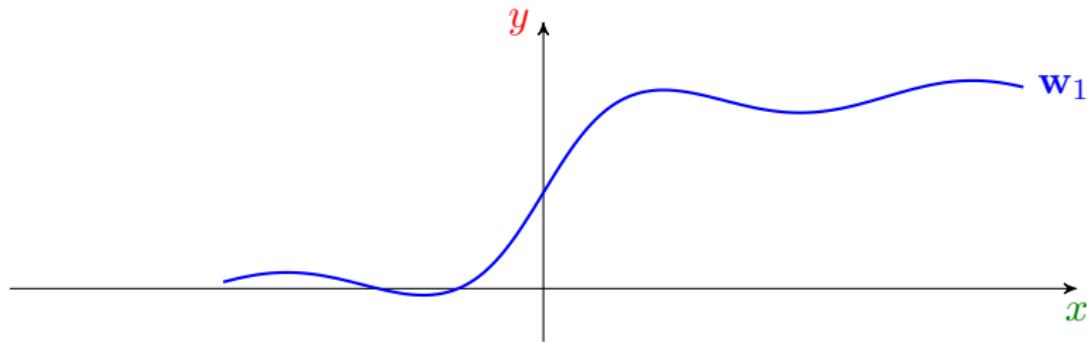
As learnable parameters change, approximator sketches different functions

Visualizing Function Approximation

Let's assume one-dimensional inputs and outputs: *this assumption helps us visualize the function approximator*

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

With scalar input, we can visualize the model as



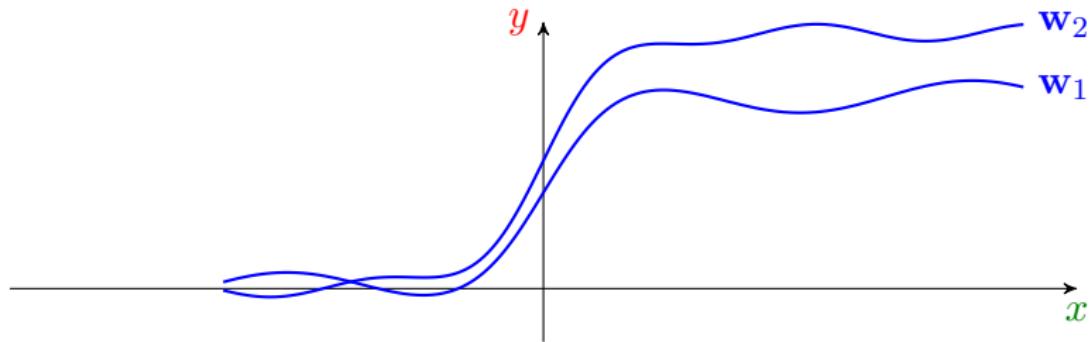
As learnable parameters change, approximator sketches different functions

Visualizing Function Approximation

Let's assume one-dimensional inputs and outputs: *this assumption helps us visualize the function approximator*

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

With scalar input, we can visualize the model as



As learnable parameters change, approximator sketches different functions

Function Approximation: Training

Training

The process of finding weight w such that the approximator fits dataset and generalize is called training

To formulate training mathematically: we should first define a loss

Loss

Loss function $\mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i)$ quantifies the difference between the output of the function approximator $\hat{\mathbf{y}}_i$ and the true output \mathbf{y}

Most classic example of loss is the squared error

$$\mathcal{L}(\hat{\mathbf{y}}_i, \mathbf{y}_i) = \|\hat{\mathbf{y}}_i - \mathbf{y}_i\|^2$$

Recap: Law of Large Numbers

Consider a random sequence X_1, \dots, X_I : we call it **independent and identically distributed (i.i.d.)** with $x \sim P(x)$ if X_i 's are generated independently all with the same distribution $P(x)$

Law of Large Numbers

Assume X_1, \dots, X_I is an i.i.d. sequence with $X \sim P(x)$; then, we have¹

$$\frac{1}{I} \sum X_i \rightarrow \mathbb{E}\{X\}$$

as $I \rightarrow \infty$

In simple words: if we **arithmetically average** too many samples of a random process, we get a value very close to its **expectation**

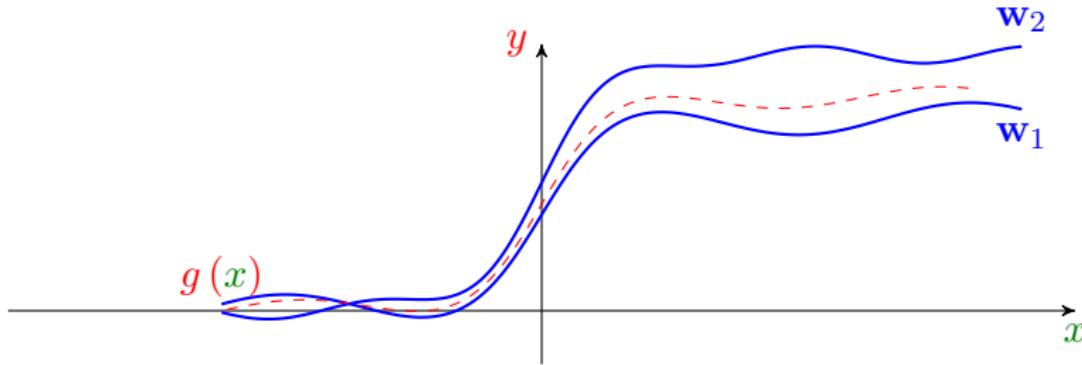
¹Of course under some conditions which we assume holding



Training

Let's now derive a meaningful approach for training: for simplicity, we stick to the one-dimensional case. We start with a thought experiment!

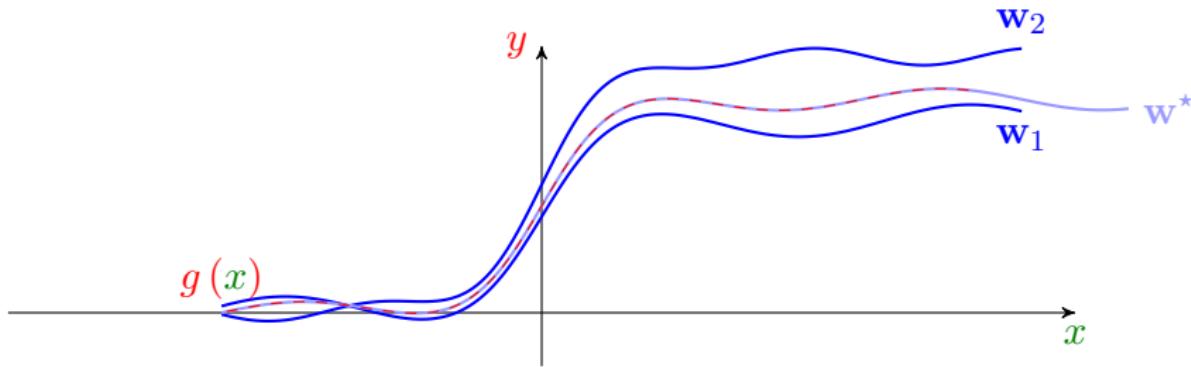
- + What if we knew function $g(\cdot)$? How would have we trained ?
- Well! We would have tuned w until the model matches $g(\cdot)$



Training

Let's now derive a meaningful approach for training: for simplicity, we stick to the one-dimensional case. We start with a thought experiment!

- + What if we knew function $g(\cdot)$? How would have we trained?
- Well! We would have tuned w until the model matches $g(\cdot)$



Training

We could represent \mathbf{w}^* using **loss**: let's consider an input x_0 to the function

- Approximator with **weight** \mathbf{w} returns $\hat{y}_0 = f(x_0|\mathbf{w})$
- The **loss** between this approximation and **true output** is

$$\ell(x_0|\mathbf{w}) = \mathcal{L}(\hat{y}_0, y_0) = \mathcal{L}(f(x_0|\mathbf{w}), g(x_0))$$

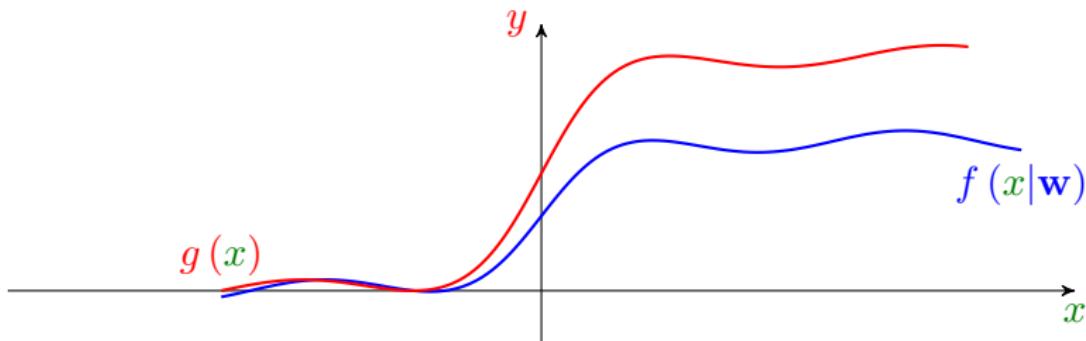
Training

We could represent \mathbf{w}^* using **loss**: let's consider an input x_0 to the function

- Approximator with weight \mathbf{w} returns $\hat{y}_0 = f(x_0|\mathbf{w})$
- The **loss** between this approximation and **true output** is

$$\ell(x_0|\mathbf{w}) = \mathcal{L}(\hat{y}_0, y_0) = \mathcal{L}(f(x_0|\mathbf{w}), g(x_0))$$

Let's visualize this loss



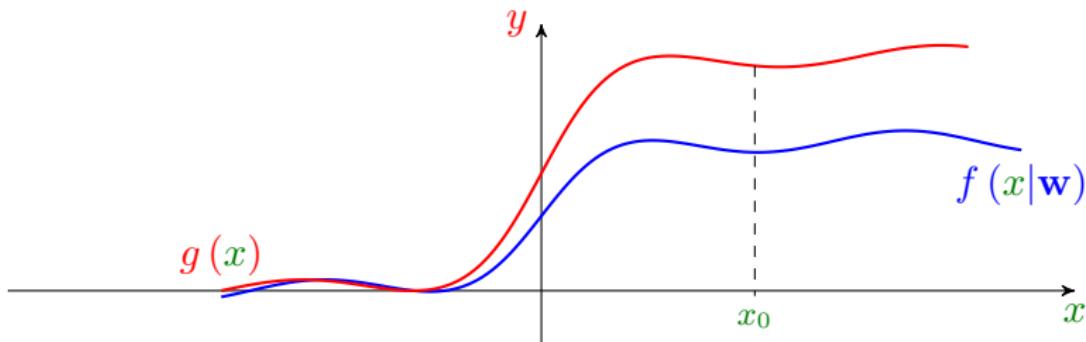
Training

We could represent w^* using **loss**: let's consider an input x_0 to the function

- Approximator with weight w returns $\hat{y}_0 = f(x_0|w)$
- The **loss** between this approximation and **true output** is

$$\ell(x_0|w) = \mathcal{L}(\hat{y}_0, y_0) = \mathcal{L}(f(x_0|w), g(x_0))$$

Let's visualize this loss



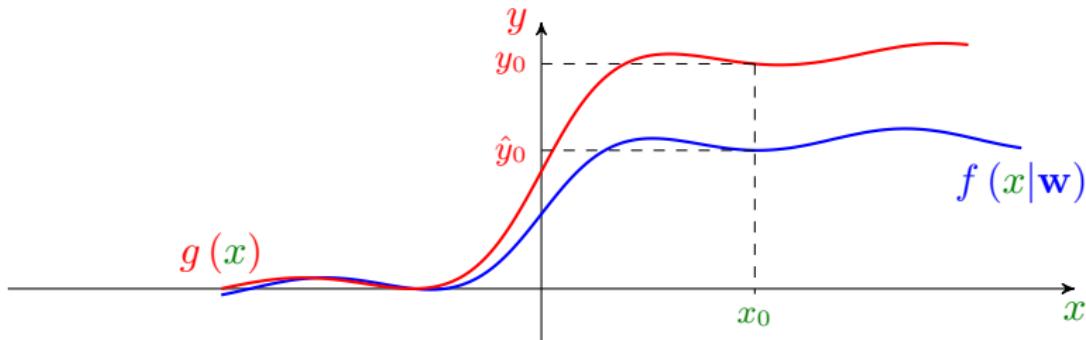
Training

We could represent w^* using **loss**: let's consider an input x_0 to the function

- Approximator with weight w returns $\hat{y}_0 = f(x_0|w)$
- The **loss** between this approximation and **true output** is

$$\ell(x_0|w) = \mathcal{L}(\hat{y}_0, y_0) = \mathcal{L}(f(x_0|w), g(x_0))$$

Let's visualize this loss



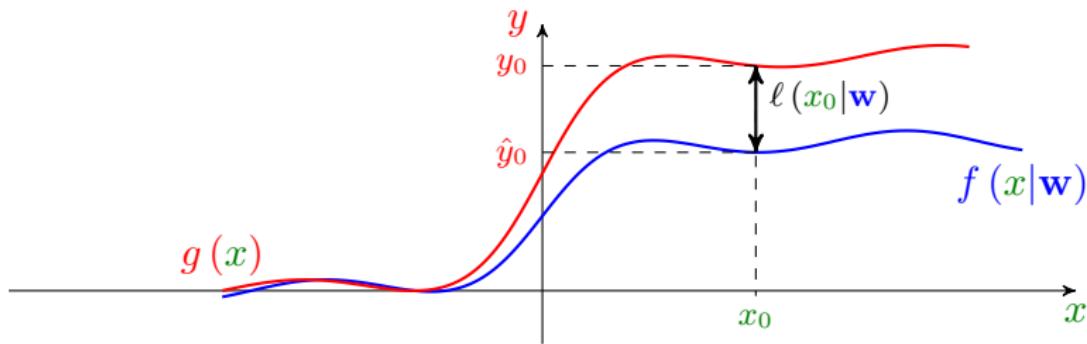
Training

We could represent w^* using **loss**: let's consider an input x_0 to the function

- Approximator with weight w returns $\hat{y}_0 = f(x_0|w)$
- The **loss** between this approximation and **true output** is

$$\ell(x_0|w) = \mathcal{L}(\hat{y}_0, y_0) = \mathcal{L}(f(x_0|w), g(x_0))$$

Let's visualize this loss



Training: Risk

We would like to recover the **true output** *as closely as possible*: so, the best option at the particular point x_0 is to find the choice of w that minimizes $\ell(x_0|w)$.

- + But x_0 is not fixed! We cannot find w for a single value of x_0 !
- Right! We should learn w for any x_0
- + How can we do it?
- We treat x_0 as a random variable with some distribution $P(x)$, and minimize the expected loss often called **risk**

Training: Risk

We would like to recover the **true output** *as closely as possible*: so, the best option at the particular point x_0 is to find the choice of w that minimizes $\ell(x_0|w)$.

- + But x_0 is not fixed! We cannot find w for a single value of x_0 !
- Right! We should learn w for any x_0
- + How can we do it?
- We treat x_0 as a random variable with some distribution $P(x)$, and minimize the expected loss often called **risk**

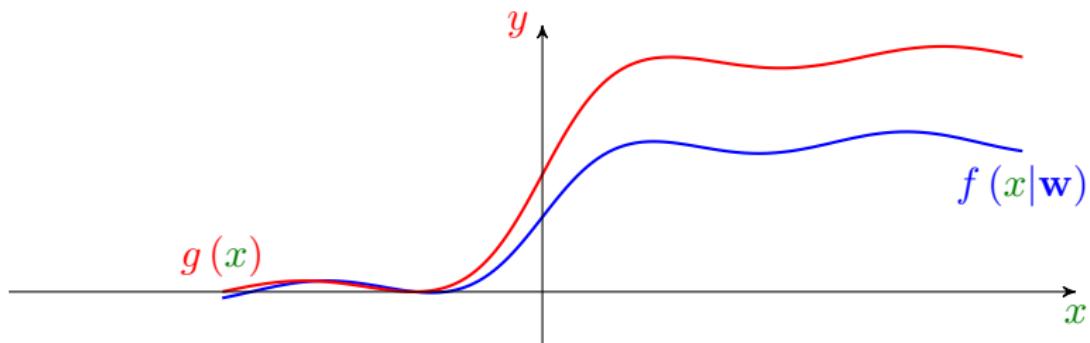
Risk

For given learnable parameter w , the **risk** is defined as

$$R(w) = \mathbb{E}\{\ell(X|w)\} = \int \ell(X|w) P(x) dx$$

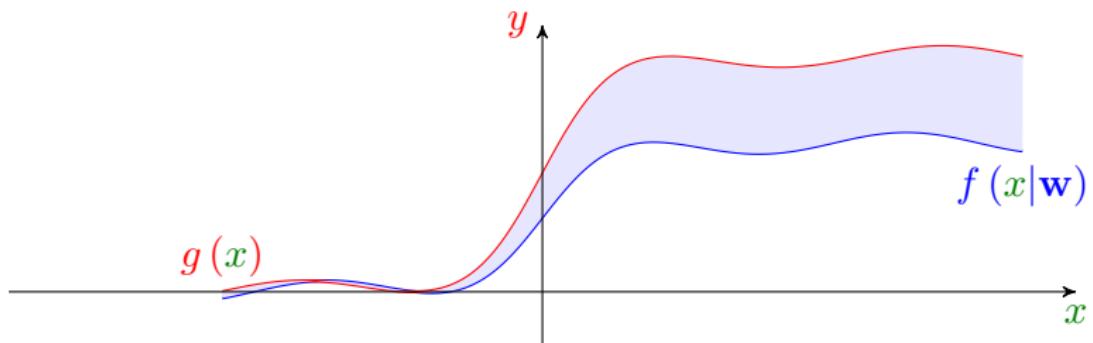
Training: Risk Minimization

Let's visualize the *risk*



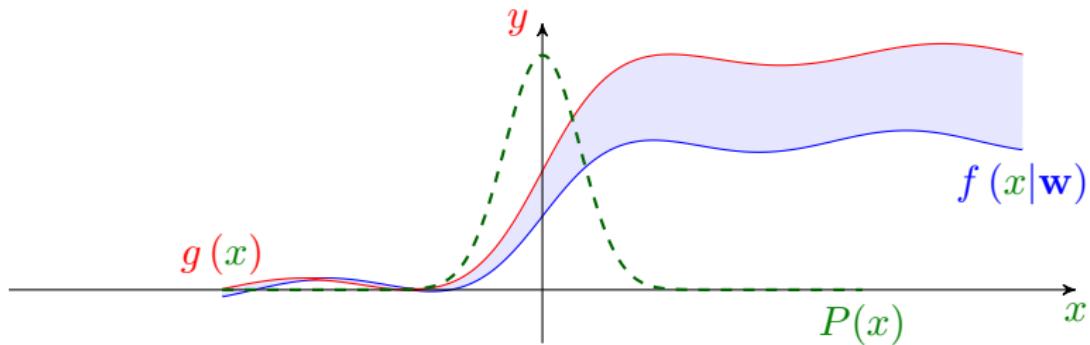
Training: Risk Minimization

Let's visualize the *risk*



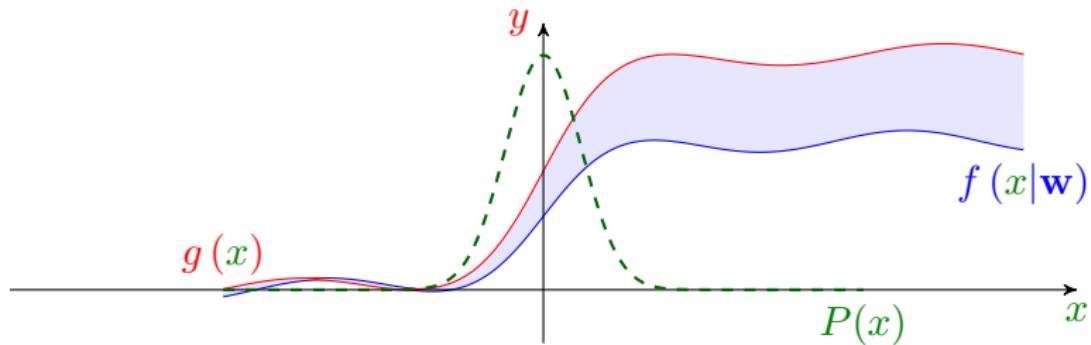
Training: Risk Minimization

Let's visualize the *risk*



Training: Risk Minimization

Let's visualize the **risk**



The training is then formulated as the minimization of **risk**

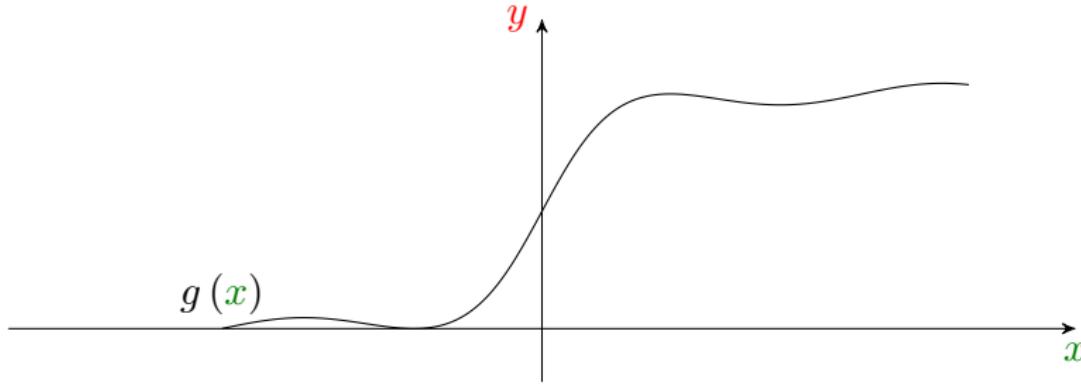
$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} R(\mathbf{w})$$

Training: Empirical Risk

- + Bravo! But, the *training* seems **infeasible**, since we have neither the **true function** $g(\cdot)$, nor the **distribution** $P(x)$!
- Right! But, we could handle this approximately using **the LLN**

Let's look at what we have: *the dataset* which contains **samples** of $g(\cdot)$

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \dots, I\}$$

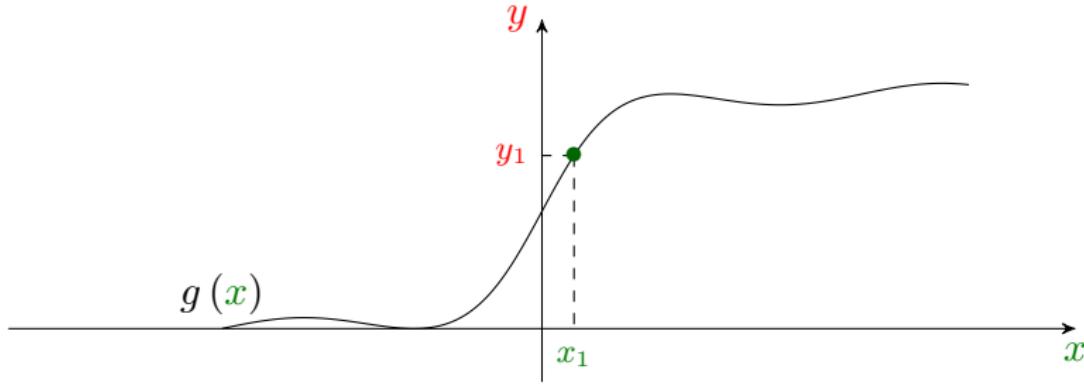


Training: Empirical Risk

- + Bravo! But, the *training* seems **infeasible**, since we have neither the **true function** $g(\cdot)$, nor the **distribution** $P(x)$!
- Right! But, we could handle this approximately using **the LLN**

Let's look at what we have: *the dataset* which contains **samples** of $g(\cdot)$

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \dots, I\}$$

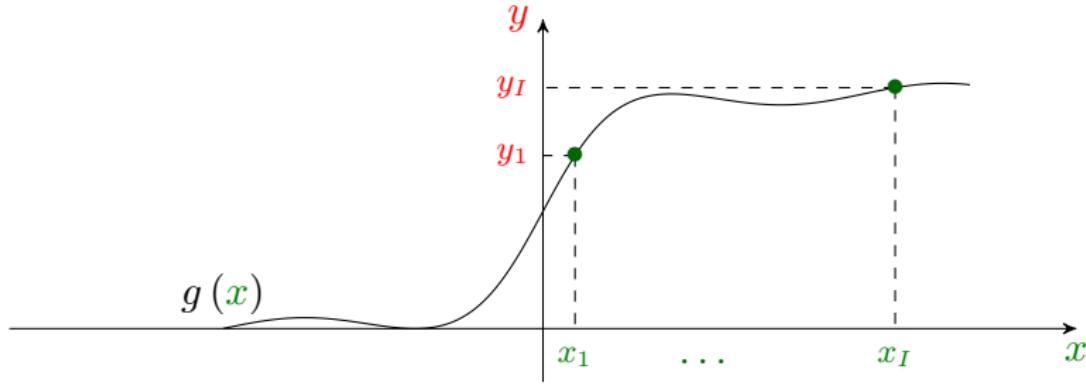


Training: Empirical Risk

- + Bravo! But, the *training* seems **infeasible**, since we have neither the **true function** $g(\cdot)$, nor the **distribution** $P(x)$!
- Right! But, we could handle this approximately using **the LLN**

Let's look at what we have: *the dataset* which contains **samples** of $g(\cdot)$

$$\mathbb{D} = \{(x_i, y_i) : i = 1, \dots, I\}$$



Training: Empirical Risk

At these data-points, we can determine the loss

$$\ell(x_i|\mathbf{w}) = \mathcal{L}(f(x_i|\mathbf{w}), g(x_i)) = \mathcal{L}(f(x_i|\mathbf{w}), y_i)$$

Training: Empirical Risk

At these data-points, we can determine the loss

$$\ell(\mathbf{x}_i|\mathbf{w}) = \mathcal{L}(f(\mathbf{x}_i|\mathbf{w}), g(\mathbf{x}_i)) = \mathcal{L}(f(\mathbf{x}_i|\mathbf{w}), \mathbf{y}_i)$$

Now, we assume that the data-points are independently drawn from the distribution $P(x)$: the LLN then suggests that the arithmetic average of losses converges to the risk when we have a large enough number of data-points, i.e., as we increase I

$$\frac{1}{I} \sum_{i=1}^I \ell(\mathbf{x}_i|\mathbf{w}) \rightarrow \mathbb{E}\{\ell(\mathbf{X}|\mathbf{w})\} = R(\mathbf{w})$$

We call this arithmetic average the empirical risk

Empirical risk is the best estimate of risk that we get from our dataset

Training: Empirical Risk Minimization

Empirical Risk

Let \mathbf{w} includes all learnable parameters, and the dataset be

$$\mathbb{D} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, \dots, I\}$$

for loss function \mathcal{L} , the empirical risk is defined as

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^I \mathcal{L}(f(\mathbf{x}_i | \mathbf{w}), \mathbf{y}_i)$$

Training: Empirical Risk Minimization

Empirical Risk

Let \mathbf{w} includes all learnable parameters, and the dataset be

$$\mathbb{D} = \{(\mathbf{x}_i, \mathbf{y}_i) : i = 1, \dots, I\}$$

for loss function \mathcal{L} , the empirical risk is defined as

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^I \mathcal{L}(f(\mathbf{x}_i | \mathbf{w}), \mathbf{y}_i)$$

The training is performed by minimizing the empirical risk

$$\mathbf{w}^* = \underset{\mathbf{w}}{\operatorname{argmin}} \hat{R}(\mathbf{w}) = \underset{\mathbf{w}}{\operatorname{argmin}} \frac{1}{I} \sum_{i=1}^I \mathcal{L}(f(\mathbf{x}_i | \mathbf{w}), \mathbf{y}_i) \quad (\text{Training})$$

Function Optimization

The *training* in general is of the form of an optimization problem

$$\mathbf{w}^* = \operatorname{argmin}_{\mathbf{w}} \hat{R}(\mathbf{w}) = \operatorname{argmin}_{\mathbf{w}} \frac{1}{I} \sum_{i=1}^I \mathcal{L}(f_{\mathbf{h}}(\mathbf{x}_i | \mathbf{w}), \mathbf{y}_i) \quad (\text{Training})$$

In this optimization problem

- $f_{\mathbf{h}}(\cdot | \mathbf{w})$ model with **hyperparameters** \mathbf{h} and **learnable parameters** \mathbf{w}
 - ↳ in DL, it is the input-output relation of a neural network whose **architecture** is specified by \mathbf{h} and whose **weights and biases** are collected in \mathbf{w}
- \mathbf{x}_i is a data-point with **label** \mathbf{y}_i , and I is size of **dataset**
- \mathcal{L} is the loss function

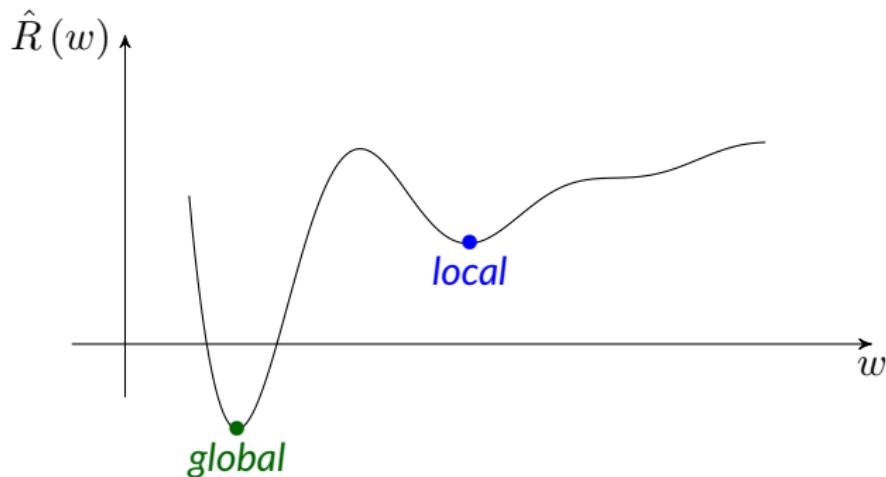
No matter what we choose, at the end of the day we need to solve

$$\min_{\mathbf{w}} \hat{R}(\mathbf{w})$$

Function Optimization

In general, the *empirical risk* $\hat{R}(w)$ can have *local* and *global* minima

Let's take a look at a simple visual case with only one parameter, i.e., $w \in \mathbb{R}$



We are happy if we get the *global*; but, many times getting to a *local* is enough!

Function Optimization

- + Why is it a big problem? We could grid w and search for the grid with smallest **empirical risk**. We then find it with a **good accuracy!**
- For only one parameter yes! But, in **deep neural networks** we have **too many parameters!**

Say for an **accurate approximation** with **only one parameter**, we need G grids.

If we have D parameters, i.e., $\mathbf{w} \in \mathbb{R}^D$, we need

G^D grids

to get an approximation with the same accuracy!

For practical neural networks with $D = 10^5$, this is **impossible!**

we need to have an **optimization algorithm** with **feasible complexity**

Optimization Algorithms

We look for an **optimization algorithm**, or as ML people call it “**an optimizer**”: it starts from an **initial point** and moves us towards the point where **objective** is minimized (at least a **local** one) in a **feasible number of steps**

Let’s clear things up: we are looking for an iterative approach as below

- 1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
- 2: Choose some small ϵ , and set $t = 1$
- 3: **while** $\Delta > \epsilon$ **do**
- 4: Determine a vector $\boldsymbol{\mu}^{(t)} \in \mathbb{R}^D$ based on $\hat{R}(\mathbf{w}) \leftarrow$ we need to figure out
- 5: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} + \boldsymbol{\mu}^{(t)}$
- 6: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
- 7: **end while**

where we would like to have following properties

- ↳ most of the time the **empirical risk reduces** in each iteration
- ↳ the algorithm stops after a **feasible number of iterations**

Optimization Algorithms

- 1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
- 2: Choose some small ϵ , and set $t = 1$
- 3: **while** $\Delta > \epsilon$ **do**
- 4: Determine a vector $\boldsymbol{\mu}^{(t)} \in \mathbb{R}^D$ based on $\hat{R}(\mathbf{w}) \leftarrow$ we need to figure out
- 5: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} + \boldsymbol{\mu}^{(t)}$
- 6: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
- 7: **end while**

We are going to get what we want, if we set

$\boldsymbol{\mu}^{(t)}$ to be *proportional* to the *negative of gradient* at $\mathbf{w}^{(t-1)}$

This is what we call the *gradient descent algorithm*. But, before we start with this algorithm, let's recap some basic notions of calculus!

Review: Derivative of a Function

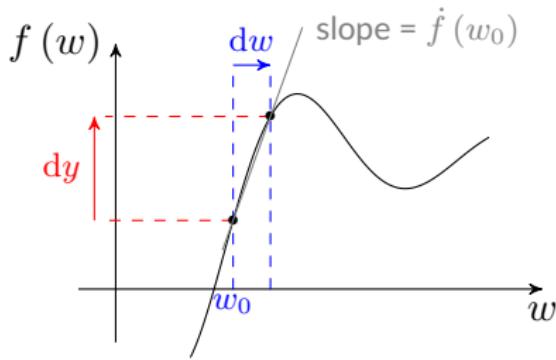
Derivative of one-dimensional function $f(w)$ at point $w = w_0$ is defined as

$$\dot{f}(w_0) = \frac{d}{dw} f(w_0) = f'(w_0) = \lim_{\delta \rightarrow 0} \frac{f(w_0 + \delta) - f(w_0)}{\delta}$$

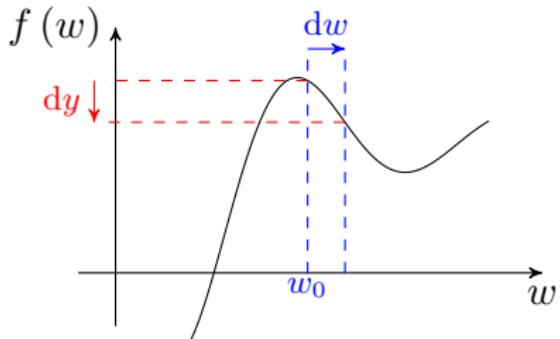
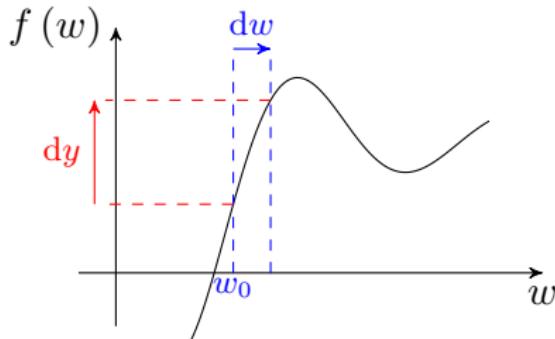
This definition is intuitively interpreted as follows:

Let $y = f(w)$. If we vary w around w_0 with a tiny step dw ; then,

$$\text{Variation of } y = dy = \dot{f}(w_0) dw$$



Review: Derivative of a Function



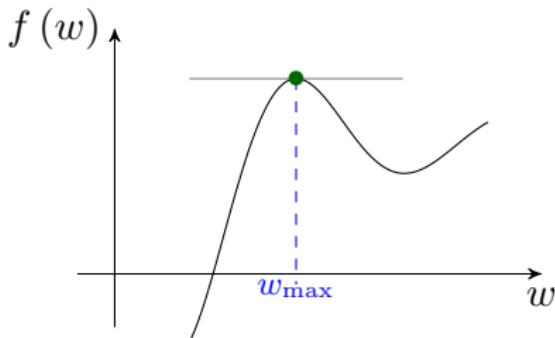
The derivative represents the slope of function

- $\dot{f}(w_0) > 0$ means increasing w will increase $y = f(w)$
- $\dot{f}(w_0) < 0$ means increasing w will decrease $y = f(w)$

So, we could also say: the derivative shows the **moving direction** on w -axis towards which the function **increases**; or alternatively, **its negative** is the **direction** that function **decreases**

Review: Derivative of a Function

When do we have the derivative equal to zero? Either we are at a *maximum*



Starting before the maximum,

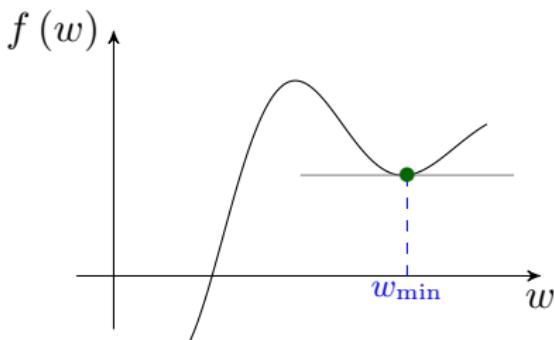
- The derivative is *first positive* and gradually *reduces to zero*
- As we pass the maximum the derivative gets *more and more negative*

So around the maximum as we *increase w*, the derivative *reduces*

$$\ddot{f}(w_{\max}) = \frac{d^2}{dw^2} f(w_{\max}) = f''(w_{\max}) < 0$$

Review: Derivative of a Function

When do we have the derivative equal to zero? Either we are at a *minimum*



Starting before the minimum,

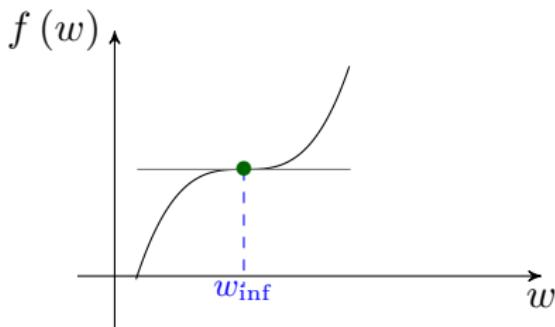
- The derivative is **first negative** and gradually **increases to zero**
- As we pass the minimum the derivative gets **more and more positive**

So around the maximum as we **increase w**, the derivative **reduces**

$$\ddot{f}(w_{\min}) = \frac{d^2}{dw^2} f(w_{\min}) = f''(w_{\min}) > 0$$

Review: Derivative of a Function

When do we have the derivative equal to zero? Either we are at an *inflection*



Starting before the inflection point,

- The derivative is *first positive* and gradually *decreases to zero*
- As we pass the inflection point the derivative gets *again positive*

So around the inflection point, the second derivative *changes sign*

$$\ddot{f}(w_{\text{inf}}) = \frac{d^2}{dw^2} f(w_{\text{inf}}) = f''(w_{\text{inf}}) = 0$$

Review: Gradient of a Function

- + What about *multi-variable functions*, e.g., $f(\mathbf{w})$ for $\mathbf{w} = [w_1, \dots, w_N]$?
- We can take derivative with respect to each variable, i.e.,

$$\dot{f}_n(\mathbf{w}_0) = \frac{\partial}{\partial w_n} f(\mathbf{w}_0)$$

This is what we call *partial derivative*

Partial derivative n represents the same thing: *slope in direction of w_n*

Let $y = f(\mathbf{w})$. If we vary \mathbf{w} around \mathbf{w}_0 in N -dimensional space with

$$d\mathbf{w} = [dw_1, \dots, dw_N]$$

whose entries are very tiny; then, the variation of y is

$$dy = \dot{f}_1(\mathbf{w}_0) dw_1 + \dots + \dot{f}_N(\mathbf{w}_0) dw_N = \sum_{n=1}^N \dot{f}_n(\mathbf{w}_0) dw_n$$

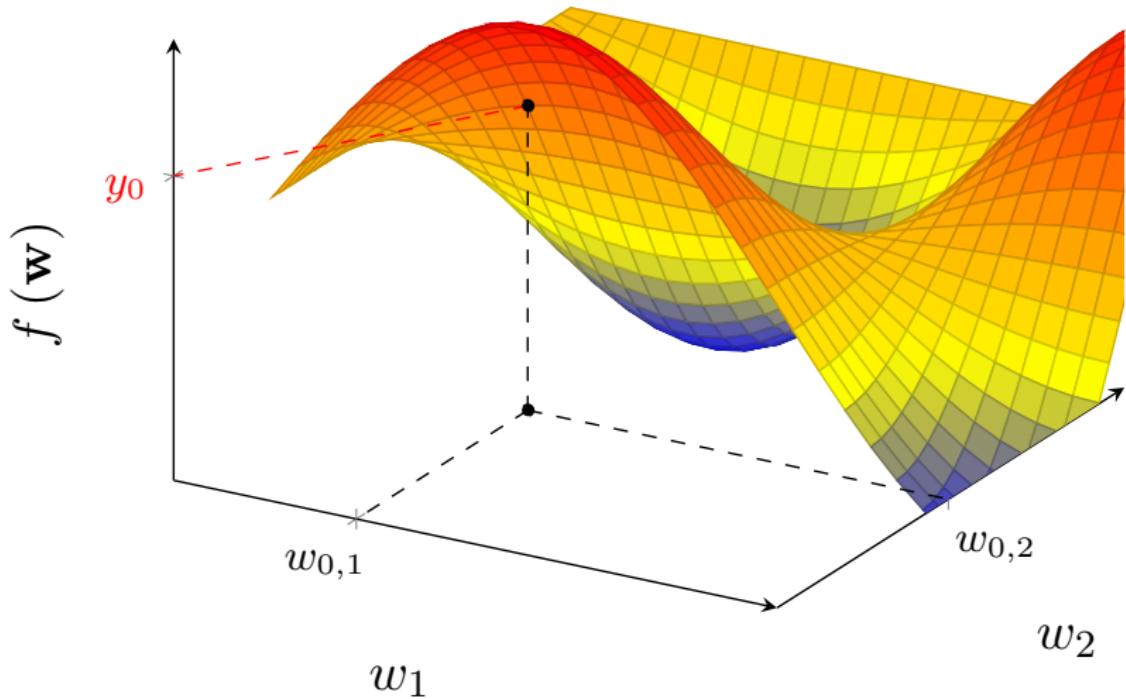
Review: Gradient of a Function

We can use inner-product to represent dy compactly

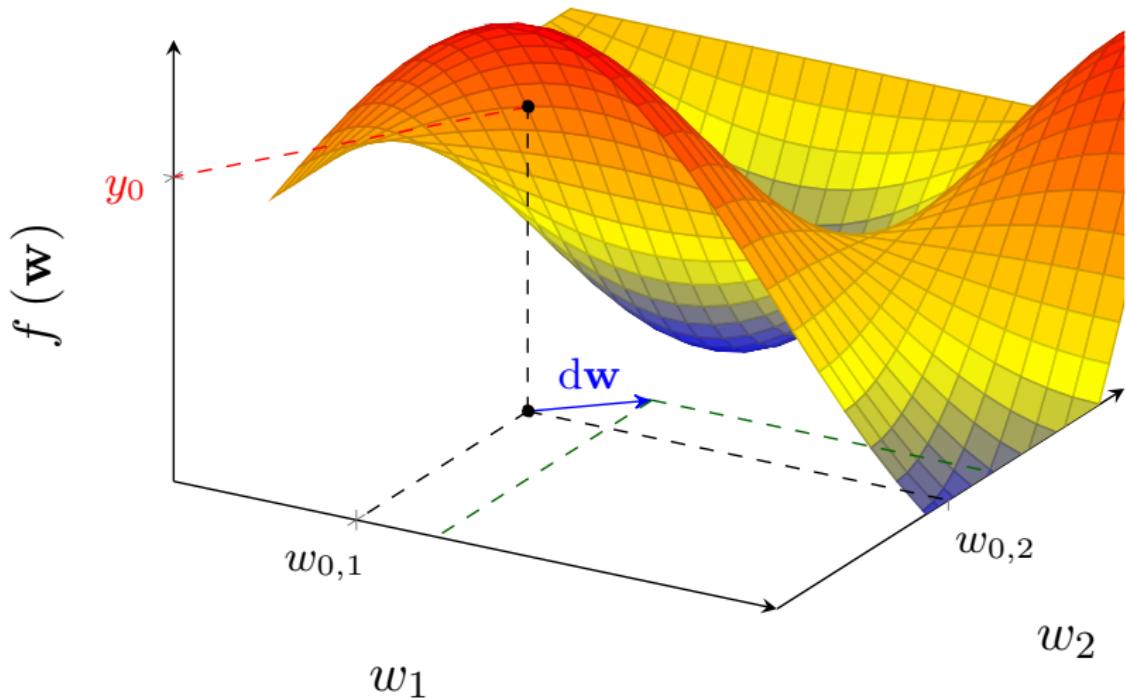
$$\begin{aligned} dy &= \sum_{n=1}^N \dot{f}_n(\mathbf{w}_0) dw_n = \underbrace{\begin{bmatrix} \dot{f}_1(\mathbf{w}_0) & \dots & \dot{f}_N(\mathbf{w}_0) \end{bmatrix}}_{\nabla f(\mathbf{w}_0)^T} \begin{bmatrix} dw_1 \\ \vdots \\ dw_N \end{bmatrix} \\ &= \nabla f(\mathbf{w}_0)^T d\mathbf{w} \end{aligned}$$

We call $\nabla f(\mathbf{w}_0)$ the gradient of $f(\cdot)$ at $\mathbf{w} = \mathbf{w}_0$

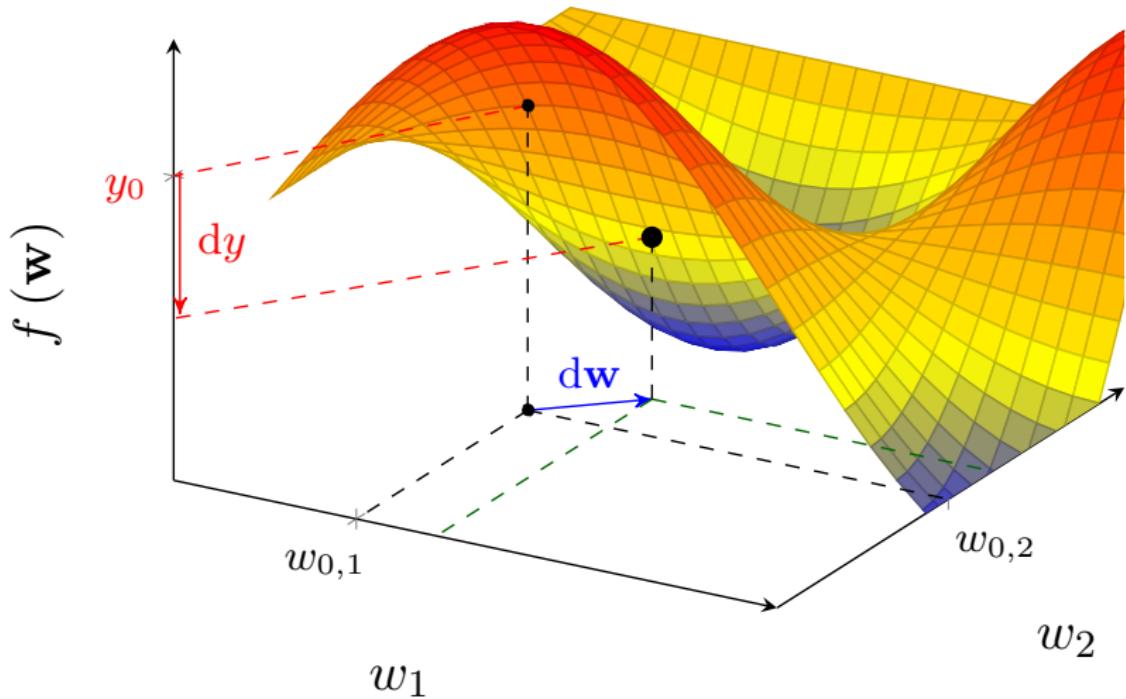
Review: Gradient of a Function



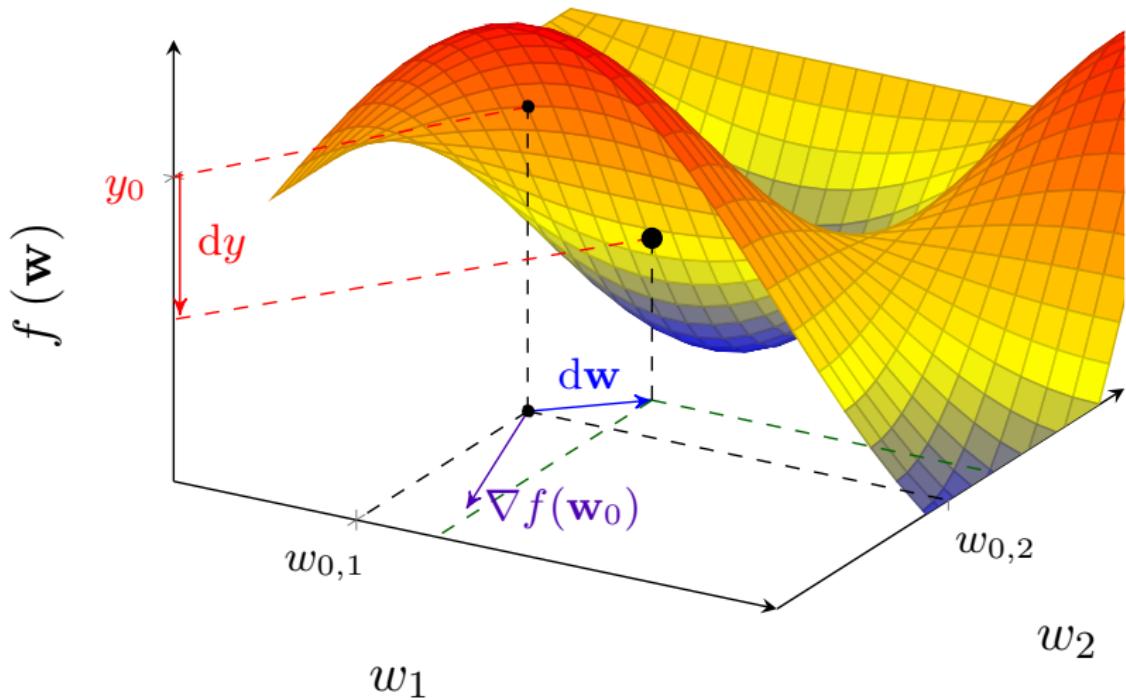
Review: Gradient of a Function



Review: Gradient of a Function

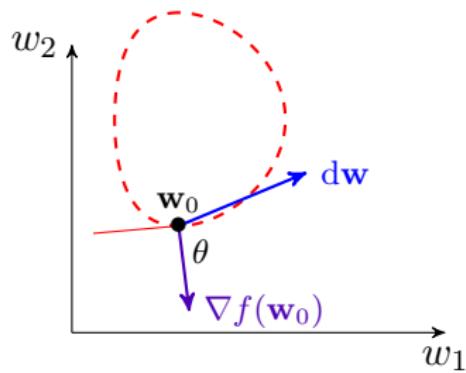


Review: Gradient of a Function



Review: Gradient of a Function

Let's get to the w -plane: the gradient is perpendicular to the contour level



The variation of y is the inner product of these two vectors

$$dy = \nabla f(\mathbf{w}_0)^T d\mathbf{w} = \|\nabla f(\mathbf{w}_0)\| \|d\mathbf{w}\| \cos(\theta)$$

where $\|\cdot\|$ is the Euclidean norm, i.e., $\|\mathbf{w}\| = \sqrt{w_1^2 + w_2^2}$

Review: Gradient of a Function

We move with a tiny step of fixed size: so we have

$$\|d\mathbf{w}\| = \epsilon$$

for some small ϵ

How can we move, such that y maximally increases? Well, we need $\theta = 0$ meaning that

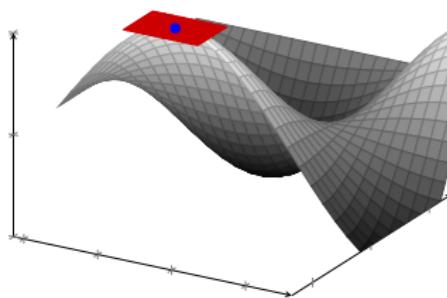
we should move in the direction of gradient

Alternatively, the function decreases maximally if $\theta = \pi$ or

we move in the direction of negative gradient

Review: Gradient of a Function

When do we have zero gradient? Either when we are at a **maximum**

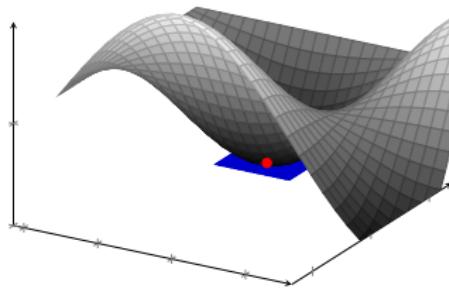


We can again relate it to the **second order derivatives** of the function

at maximum Hessian matrix is negative definite

Review: Gradient of a Function

When do we have zero gradient? or when we are at a **minimum**

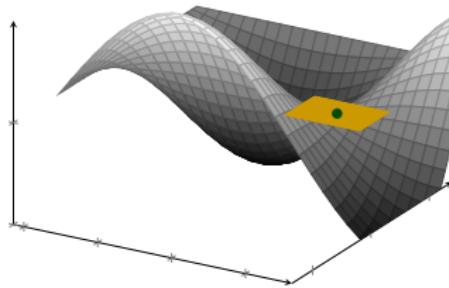


We can again relate it to the **second order derivatives** of the function

at **minimum Hessian matrix is positive** definite

Review: Gradient of a Function

When do we have zero gradient? or when we are at a *saddle point*



We can again relate it to the **second order derivatives** of the function

at saddle point **Hessian matrix** is neither **negative** nor **positive definite**

Review: Gradient of a Function

Just as a reminder: Hessian is the matrix of all *second order derivatives*

$$\nabla^2 f(\mathbf{w}_0) = \begin{bmatrix} \frac{\partial^2}{\partial w_1^2} f(\mathbf{w}_0) & \frac{\partial^2}{\partial w_1 \partial w_2} f(\mathbf{w}_0) & \dots & \frac{\partial^2}{\partial w_1 \partial w_N} f(\mathbf{w}_0) \\ \vdots & & & \vdots \\ \frac{\partial^2}{\partial w_N \partial w_1} f(\mathbf{w}_0) & \frac{\partial^2}{\partial w_N \partial w_2} f(\mathbf{w}_0) & \dots & \frac{\partial^2}{\partial w_N^2} f(\mathbf{w}_0) \end{bmatrix}$$

We **never** use the Hessian matrix in this course

Moral of Story: Gradient Decent

- + What is the whole motive of this discussions?
- Simple: at any point \mathbf{w}_0 , if we want to move in a direction that the function reduces, the best direction is negative of gradient at \mathbf{w}_0

So, we can complete our optimization algorithm as follows:

- 1: Initiate at some $\mathbf{w}^{(0)} \in \mathbb{R}^D$ and deviation $\Delta = +\infty$
- 2: Choose some small ϵ and η , and set $t = 1$
- 3: **while** $\Delta > \epsilon$ **do**
- 4: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
- 5: Update the deviation $\Delta = |\hat{R}(\mathbf{w}^{(t)}) - \hat{R}(\mathbf{w}^{(t-1)})|$
- 6: **end while**

The scalar η is the step-size we take in each iteration:

we usually call it **learning rate**

Stochastic Gradient Descent

Let's use **gradient descent** for function approximation: we want to minimize

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^I \mathcal{L}(f(\mathbf{x}_i | \mathbf{w}), \mathbf{y}_i)$$

GradientDescent() :

- 1: Initiate with some initial $\mathbf{w}^{(0)}$ and set a learning rate η
- 2: **while** weights not converged **do**
- 3: **for** $i = 1, \dots, I$ **do**
- 4: Compute gradient for $\nabla_i = \mathcal{L}\left(f\left(\mathbf{x}_i | \mathbf{w}^{(t-1)}\right), \mathbf{y}_i\right)$
- 5: Update gradient as $\nabla \hat{R}(\mathbf{w}^{(t-1)}) \leftarrow \nabla \hat{R}(\mathbf{w}^{(t-1)}) + \nabla_i / I$
- 6: **end for**
- 7: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$
- 8: **end while**

We call this form of gradient descent **full-batch** which can be too complex

Stochastic Gradient Descent

Let's use **gradient descent** for function approximation: we want to minimize

$$\hat{R}(\mathbf{w}) = \frac{1}{I} \sum_{i=1}^I \mathcal{L}(f(\mathbf{x}_i | \mathbf{w}), \mathbf{y}_i)$$

SGD():

- 1: Initiate with some initial $\mathbf{w}^{(0)}$ and set a learning rate η
- 2: **while** weights not converged **do**
- 3: **for** a random subset of batch $b = 1, \dots, B$ **do**
- 4: Compute gradient for $\nabla_b = \mathcal{L}\left(f\left(\mathbf{x}_b | \mathbf{w}^{(t-1)}\right), \mathbf{y}_b\right)$
- 5: Update gradient as $\nabla \hat{R}(\mathbf{w}^{(t-1)}) \leftarrow \nabla \hat{R}(\mathbf{w}^{(t-1)}) + \nabla_b / I$
- 6: **end for**
- 7: Update weights as $\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \eta \nabla \hat{R}(\mathbf{w}^{(t-1)})$ \leftarrow unbiased estimator
- 8: **end while**

This is what we call **stochastic mini-batch** gradient descent

Back to Model-free RL

We now get back to model-free RL: *this time however*

we use a parameterized estimator

Back to Model-free RL

We now get back to model-free RL: *this time however*

we use a parameterized estimator

This means that

- to estimate the **value** function, we use

$$\hat{v}_{\mathbf{w}}(s)$$

- to estimate the **action-value** function, we use

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

Back to Model-free RL

We now get back to model-free RL: *this time however*

we use a parameterized estimator

This means that

- to estimate the **value** function, we use

$$\hat{v}_{\mathbf{w}}(s)$$

- to estimate the **action-value** function, we use

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

↳ These functions are **parameterized** by some \mathbf{w}

↳ They could be as **simple** as linear approximators or **advanced** like DNNs

Back to Model-free RL

- + But, we have in general ***prediction*** and ***control*** problems. In which one are we going to use function ***approximation***?
- Well, we can use in both

Back to Model-free RL

- + But, we have in general ***prediction*** and ***control*** problems. In which one are we going to use function ***approximation***?
- Well, we can use in both

Recall

We have two major problems in ***model-free*** RL

- ***Prediction*** in which for a given policy π we evaluate values by sampling the environment
- ***Control*** in which after each interaction, we improve our policy aiming to converge to the ***optimal policy***

Back to Model-free RL

- + But, we have in general ***prediction*** and ***control*** problems. In which one are we going to use function ***approximation***?
- Well, we can use in both

Recall

We have two major problems in ***model-free*** RL

- ***Prediction*** in which for a given policy π we evaluate values by sampling the environment
- ***Control*** in which after each interaction, we improve our policy aiming to converge to the ***optimal policy***

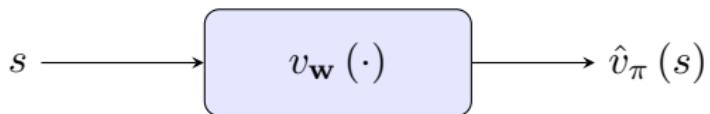
Let's start with the ***prediction***

Basic Prediction via Approximator

Say we want to **evaluate** the value function of a policy π :

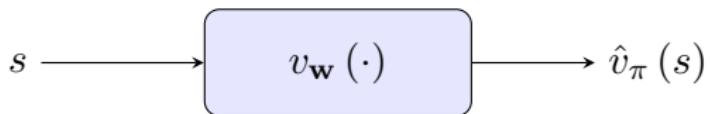
Basic Prediction via Approximator

Say we want to **evaluate** the value function of a policy π : with **function approximation** we assume that the **value approximator** is



Basic Prediction via Approximator

Say we want to **evaluate** the value function of a policy π : with **function approximation** we assume that the **value approximator** is

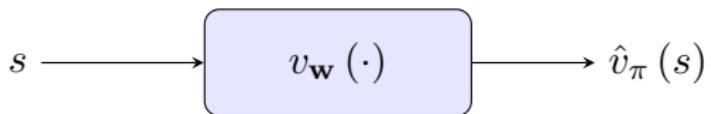


- We sample the environment by policy π
 - ↳ We draw sample trajectories using policy π

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

Basic Prediction via Approximator

Say we want to **evaluate** the value function of a policy π : with **function approximation** we assume that the **value approximator** is



- We sample the environment by policy π
 - ↳ We draw sample trajectories using policy π

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

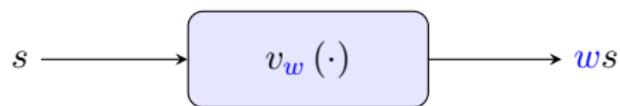
- We want to use these samples to train our **approximation model**
 - ↳ We want to find w that approximates the best $v_\pi(\cdot)$

Building Simple Approximator

- + Well you have talked a lot about **approximators**, but can you give us a concrete example?!
- Sure! Let's look at **linear approximator**: say our approximator is a linear function of the input

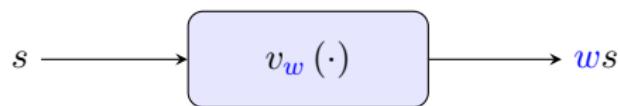
Building Simple Approximator

- + Well you have talked a lot about **approximators**, but can you give us a concrete example?!
- Sure! Let's look at **linear approximator**: say our approximator is a linear function of the input
- + Does it mean the following diagram?!



Building Simple Approximator

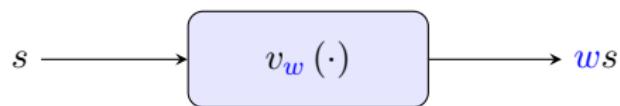
- + Well you have talked a lot about **approximators**, but can you give us a concrete example?!
- Sure! Let's look at **linear approximator**: say our approximator is a linear function of the input
- + Does it mean the following diagram?!



- + But it doesn't seem to work with only **one parameter**!

Building Simple Approximator

- + Well you have talked a lot about **approximators**, but can you give us a concrete example?!
- Sure! Let's look at **linear approximator**: say our approximator is a linear function of the input
- + Does it mean the following diagram?!



- + But it doesn't seem to work with only **one parameter**!
- Oops! We haven't yet defined the **feature representation of states**!

Feature Representation

In general, we could represent a particular state in *various forms*

Feature Representation

In general, we could represent a particular state in *various forms*

Say we have N states s^1, \dots, s^N : we can represent them by a *scalar*, e.g.

$$s^n \mapsto x(s^n) = n$$

Feature Representation

In general, we could represent a particular state in *various forms*

Say we have N states s^1, \dots, s^N : we can represent them by a *scalar*, e.g.

$$s^n \mapsto \mathbf{x}(s^n) = n$$

Or a *one-sparse* vector, i.e.,

$$s^n \mapsto \mathbf{x}(s^n) = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{entry } n$$

Feature Representation

Feature Representation of States

Feature representation maps each *state* into a vector of *features* that corresponds to that *state*, i.e.,

$$\mathbf{x}(\cdot) : \mathbb{S} \mapsto \mathbb{R}^J$$

for some integer J that is the *feature dimension*

Feature Representation

Feature Representation of States

Feature representation maps each **state** into a vector of **features** that corresponds to that **state**, i.e.,

$$\mathbf{x}(\cdot) : \mathbb{S} \mapsto \mathbb{R}^J$$

for some integer J that is the **feature dimension**

Example: We can represent the feature by **tokenization**

$$s^n \mapsto \mathbf{x}(s^n) = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{entry } n$$

Feature Representation

Feature Representation of States

Feature representation maps each *state* into a vector of features that corresponds to that *state*, i.e.,

$$\mathbf{x}(\cdot) : \mathcal{S} \mapsto \mathbb{R}^J$$

for some integer J that is the *feature dimension*

In practice use more *advanced problem-specific* features

Feature Representation

Feature Representation of States

Feature representation maps each **state** into a vector of features that corresponds to that **state**, i.e.,

$$\mathbf{x}(\cdot) : \mathcal{S} \mapsto \mathbb{R}^J$$

for some integer J that is the **feature dimension**

In practice use more **advanced problem-specific** features

- The state of a robot can be specified by its **geometric features**
 - ↳ its distance to the edges of the room, its direction, etc

Feature Representation

Feature Representation of States

Feature representation maps each **state** into a vector of features that corresponds to that **state**, i.e.,

$$\mathbf{x}(\cdot) : \mathcal{S} \mapsto \mathbb{R}^J$$

for some integer J that is the **feature dimension**

In practice use more **advanced problem-specific** features

- The state of a robot can be specified by its **geometric features**
 - ↳ its distance to the edges of the room, its direction, etc
- The state of a compute game is completely explained by its **frames**
 - ↳ we collect all frames from a state to another

Example: Mountain Car



Let's consider the famous *mountain car example*

- A car is stuck in a valley
 - ↳ it can *accelerate to left, accelerate to right, or do nothing*
- It observes its *velocity v and location x* on the horizontal axis
 - ↳ they are *continuous* variables

The goal is to train the car to get out of this valley as quick as possible

- The car is *rewarded by -1* after each unsuccessful trial

Example: Mountain Car



This car can be in **infinite** number of **states**; however, we can represent its state by its **velocity and location**

$$\mathbf{x}(s) = \begin{bmatrix} v \\ x \end{bmatrix}$$

This is much **more feasible** to work with!

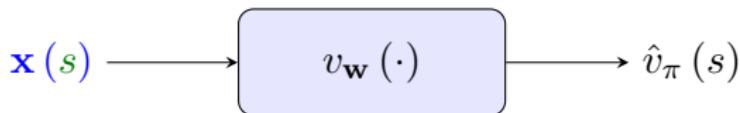
Back to Simple Approximator

- + How is this **feature** related to our discussion on **function approximation**?!
- Well! Approximation model maps the **features** to **value**

Back to Simple Approximator

- + How is this **feature** related to our discussion on **function approximation**?!
 - Well! Approximation model maps the **features** to **value**

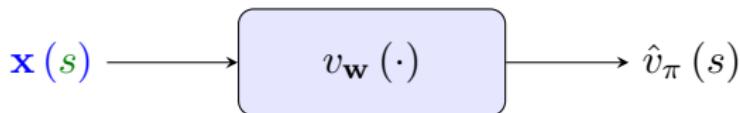
So, in our problem, we in fact assume that



Back to Simple Approximator

- + How is this **feature** related to our discussion on **function approximation**?!
 - Well! Approximation model maps the **features** to **value**

So, in our problem, we in fact assume that



Example: Linear Approximator

Linear approximation model maps the state features to its value as

$$v_{\mathbf{w}}(s) = \mathbf{w}^T \mathbf{x}(s)$$

Example: Mountain Car



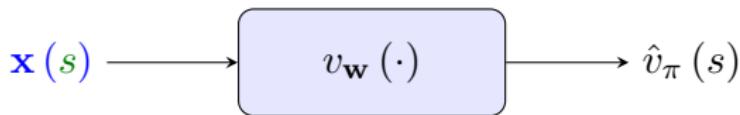
In mountain car example, we could have a two-dimensional linear approximator

$$\begin{aligned}v_{\mathbf{w}}(s) &= \mathbf{w}^T \mathbf{x}(s) = [w_1 \quad w_2] \begin{bmatrix} v \\ x \end{bmatrix} \\&= w_1 v + w_2 x\end{aligned}$$

Back to Simple Approximator

- + How is this feature related to our discussion on function approximation?!
- Well! Approximation model maps the features to value

So, in our problem, we in fact assume that



Example: Deep Approximator

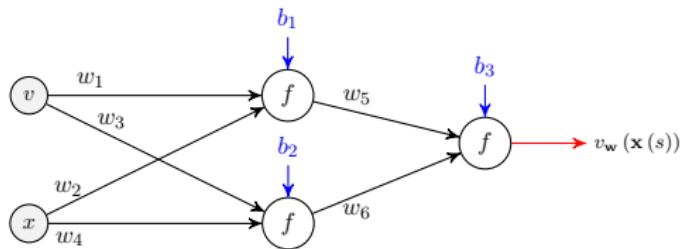
Deep approximation model maps the state features to its value via a DNN

$$v_{\mathbf{w}}(s) = \text{DNN}(\mathbf{x}(s) | \mathbf{w})$$

Example: Mountain Car



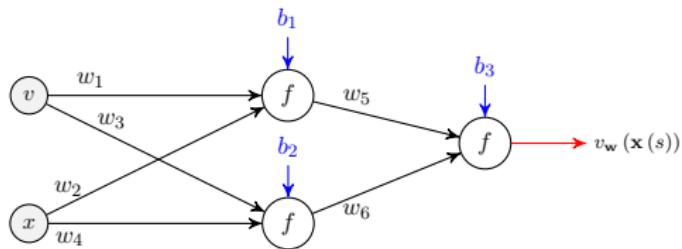
We may also use a NN to map the **feature** to its value



Example: Mountain Car



We may also use a NN to map the **feature** to its value



Here, the weights are $\mathbf{w} = [w_1, \dots, w_6, b_1, b_2, b_3]^T$

Training Approximation Model

- + *How can we train a given approximation model?*
- Let's again get help from a **genie**

Training Approximation Model

- + How can we train a given approximation model?
- Let's again get help from a **genie**

Assume that a **genie** could tell us the exact value of all states: in this case we want to find w such that for each s^n

$$v_w(s^n) \stackrel{!}{=} v_\pi(s^n)$$

Training Approximation Model

- + How can we train a given approximation model?
- Let's again get help from a genie

Assume that a **genie** could tell us the exact value of all states: in this case we want to find w such that for each s^n

$$v_w(s^n) \stackrel{!}{=} v_\pi(s^n) \rightsquigarrow |v_w(s^n) - v_\pi(s^n)| \stackrel{!}{=} 0$$

Training Approximation Model

- + How can we train a given approximation model?
- Let's again get help from a **genie**

Assume that a **genie** could tell us the exact value of all states: in this case we want to find w such that for each s^n

$$v_w(s^n) \stackrel{!}{=} v_\pi(s^n) \rightsquigarrow |v_w(s^n) - v_\pi(s^n)| \stackrel{!}{=} 0$$

This is **equivalent** to say that

$$\frac{1}{N} \sum_{n=1}^N |v_w(s^n) - v_\pi(s^n)|^2 \stackrel{!}{=} 0$$

Training Approximation Model

But, we cannot necessarily find such w : we could instead find

$$\mathbf{w}^* = \min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N |v_{\mathbf{w}}(s^n) - v_{\pi}(s^n)|^2$$

Training Approximation Model

But, we cannot necessarily find such \mathbf{w} : we could instead find

$$\mathbf{w}^* = \min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N |v_{\mathbf{w}}(s^n) - v_{\pi}(s^n)|^2$$

One may also think about a more general weighted average: we can assume that under policy π each state s^n happens with a probability $p_{\pi}(s^n)$ and write

$$\mathbf{w}^* = \min_{\mathbf{w}} \sum_{n=1}^N p_{\pi}(s^n) |v_{\mathbf{w}}(s^n) - v_{\pi}(s^n)|^2$$

Training Approximation Model

But, we cannot necessarily find such \mathbf{w} : we could instead find

$$\mathbf{w}^* = \min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N |v_{\mathbf{w}}(s^n) - v_{\pi}(s^n)|^2$$

One may also think about a more general weighted average: we can assume that under policy π each state s^n happens with a probability $p_{\pi}(s^n)$ and write

$$\begin{aligned}\mathbf{w}^* &= \min_{\mathbf{w}} \sum_{n=1}^N p_{\pi}(s^n) |v_{\mathbf{w}}(s^n) - v_{\pi}(s^n)|^2 \\ &= \min_{\mathbf{w}} \mathbb{E}_{\pi} \{ |v_{\mathbf{w}}(S) - v_{\pi}(S)|^2 \}\end{aligned}$$

Training Approximation Model

But, we cannot necessarily find such \mathbf{w} : we could instead find

$$\mathbf{w}^* = \min_{\mathbf{w}} \frac{1}{N} \sum_{n=1}^N |v_{\mathbf{w}}(s^n) - v_{\pi}(s^n)|^2$$

One may also think about a more general weighted average: we can assume that under policy π each state s^n happens with a probability $p_{\pi}(s^n)$ and write

$$\begin{aligned}\mathbf{w}^* &= \min_{\mathbf{w}} \sum_{n=1}^N p_{\pi}(s^n) |v_{\mathbf{w}}(s^n) - v_{\pi}(s^n)|^2 \\ &= \min_{\mathbf{w}} \mathbb{E}_{\pi} \{ |v_{\mathbf{w}}(S) - v_{\pi}(S)|^2 \}\end{aligned}$$

We train by minimizing residual sum of squares \equiv least-squares (LS) method

LS Training: Gradient Descent

We use gradient descent to solve this problem: *we are minimizing the risk*

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_{\pi} \left\{ |v_{\mathbf{w}}(S) - v_{\pi}(S)|^2 \right\}$$

LS Training: Gradient Descent

We use gradient descent to solve this problem: *we are minimizing the risk*

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_{\pi} \left\{ |v_{\mathbf{w}}(S) - v_{\pi}(S)|^2 \right\}$$

GradientDescent():

- 1: *Initiate with some initial \mathbf{w} and learning rate η*
- 2: **while** weights not converged **do**
- 3: Compute gradient $\nabla \mathcal{L}(\mathbf{w})$
- 4: Update weights as $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \mathcal{L}(\mathbf{w}^{t-1})$
- 5: **end while**

LS Training: Gradient Descent

We use gradient descent to solve this problem: *we are minimizing the risk*

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_{\pi} \left\{ |v_{\mathbf{w}}(S) - v_{\pi}(S)|^2 \right\}$$

`GradientDescent()`:

- 1: *Initiate with some initial \mathbf{w} and learning rate η*
- 2: **while** weights not converged **do**
- 3: Compute gradient $\nabla \mathcal{L}(\mathbf{w})$
- 4: Update weights as $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \mathcal{L}(\mathbf{w}^{t-1})$
- 5: **end while**

Let's compute the gradient: we can use the chain rule

$$\nabla \mathcal{L}(\mathbf{w}) = \frac{\partial \mathcal{L}}{\partial v_{\mathbf{w}}(S)} \nabla v_{\mathbf{w}}(S)$$

LS Training: Gradient Descent

We use gradient descent to solve this problem: *we are minimizing the risk*

$$\mathcal{L}(\mathbf{w}) = \mathbb{E}_{\pi} \left\{ |v_{\mathbf{w}}(S) - v_{\pi}(S)|^2 \right\}$$

`GradientDescent()`:

- 1: Initiate with some initial \mathbf{w} and learning rate η
- 2: **while** weights not converged **do**
- 3: Compute gradient $\nabla \mathcal{L}(\mathbf{w})$
- 4: Update weights as $\mathbf{w} \leftarrow \mathbf{w} - \eta \nabla \mathcal{L}(\mathbf{w}^{t-1})$
- 5: **end while**

Let's compute the gradient: we can use the chain rule

$$\begin{aligned}\nabla \mathcal{L}(\mathbf{w}) &= \frac{\partial \mathcal{L}}{\partial v_{\mathbf{w}}(S)} \nabla v_{\mathbf{w}}(S) \\ &= 2 \mathbb{E}_{\pi} \{ (v_{\mathbf{w}}(S) - v_{\pi}(S)) \nabla v_{\mathbf{w}}(S) \}\end{aligned}$$

LS Training: Gradient Descent

Now we set the learning rate to $\eta = 0.5\alpha$ for some α ; then, we have

$$\mathbf{w}^{(t)} \leftarrow \mathbf{w}^{(t-1)} - \alpha \mathbb{E}_\pi \{(v_{\mathbf{w}}(S) - v_\pi(S)) \nabla v_{\mathbf{w}}(S)\}$$

LS Training: Gradient Descent

Now we set the learning rate to $\eta = 0.5\alpha$ for some α ; then, we have

$$\begin{aligned}\mathbf{w}^{(t)} &\leftarrow \mathbf{w}^{(t-1)} - \alpha \mathbb{E}_\pi \{(v_{\mathbf{w}}(S) - v_\pi(S)) \nabla v_{\mathbf{w}}(S)\} \\ &\leftarrow \mathbf{w}^{(t-1)} + \alpha \mathbb{E}_\pi \{(v_\pi(S) - v_{\mathbf{w}}(S)) \nabla v_{\mathbf{w}}(S)\}\end{aligned}$$

LS Training: Gradient Descent

Now we set the learning rate to $\eta = 0.5\alpha$ for some α ; then, we have

$$\begin{aligned}\mathbf{w}^{(t)} &\leftarrow \mathbf{w}^{(t-1)} - \alpha \mathbb{E}_\pi \{(v_\mathbf{w}(S) - v_\pi(S)) \nabla v_\mathbf{w}(S)\} \\ &\leftarrow \mathbf{w}^{(t-1)} + \alpha \mathbb{E}_\pi \{(v_\pi(S) - v_\mathbf{w}(S)) \nabla v_\mathbf{w}(S)\}\end{aligned}$$

So, we the gradient descent based evaluation reduces to

GD_Eval():

- 1: Initiate with some initial $\mathbf{w}^{(0)}$ and learning rate η
- 2: **while** weights not converged **do**
- 3: Update weights as $\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbb{E}_\pi \{(v_\pi(S) - v_\mathbf{w}(S)) \nabla v_\mathbf{w}(S)\}$
- 4: **end while**

LS Training: Gradient Descent

Now we set the learning rate to $\eta = 0.5\alpha$ for some α ; then, we have

$$\begin{aligned}\mathbf{w}^{(t)} &\leftarrow \mathbf{w}^{(t-1)} - \alpha \mathbb{E}_\pi \{(v_\mathbf{w}(S) - v_\pi(S)) \nabla v_\mathbf{w}(S)\} \\ &\leftarrow \mathbf{w}^{(t-1)} + \alpha \mathbb{E}_\pi \{(v_\pi(S) - v_\mathbf{w}(S)) \nabla v_\mathbf{w}(S)\}\end{aligned}$$

So, we the gradient descent based evaluation reduces to

GD_Eval():

- 1: Initiate with some initial $\mathbf{w}^{(0)}$ and learning rate η
- 2: **while** weights not converged **do**
- 3: Update weights as $\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbb{E}_\pi \{(v_\pi(S) - v_\mathbf{w}(S)) \nabla v_\mathbf{w}(S)\}$
- 4: **end while**

- + That's nice! But, how in earth we know $v_\pi(S)$!?
- Well! We may use what we learned in model-free RL again!

LS Training via Monte Carlo

The key challenge is to find out *an estimator for*

$$\mathbb{E}_{\pi} \{(v_{\pi}(S) - v_w(S)) \nabla v_w(S)\}$$

We can do it by Monte Carlo:

LS Training via Monte Carlo

The key challenge is to find out *an estimator for*

$$\mathbb{E}_\pi \{(v_\pi(S) - v_w(S)) \nabla v_w(S)\}$$

We can do it by Monte Carlo: *say we have an episodic environment with terminal state and sampled an episode*

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

LS Training via Monte Carlo

The key challenge is to find out *an estimator for*

$$\mathbb{E}_{\pi} \{ (v_{\pi}(S) - v_w(S)) \nabla v_w(S) \}$$

We can do it by Monte Carlo: *say we have an episodic environment with terminal state and sampled an episode*

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We can convert this trajectory into

$$(S_0, G_0) \rightarrow (S_1, G_1) \rightarrow \dots \rightarrow (S_{T-1}, G_{T-1})$$

with G_t being the sample return, i.e.,

$$G_t = \sum_{i=0}^{T-t-1} \gamma^i R_{t+1+i}$$

LS Training via Monte Carlo

The key challenge is to find out *an estimator for*

$$\mathbb{E}_\pi \{(v_\pi(S) - v_w(S)) \nabla v_w(S)\}$$

We do know that G_t is an estimator of $v_\pi(S_t)$:

LS Training via Monte Carlo

The key challenge is to find out *an estimator for*

$$\mathbb{E}_\pi \{(v_\pi(S) - v_w(S)) \nabla v_w(S)\}$$

We do know that G_t is an estimator of $v_\pi(S_t)$: so we could say

$$\mathbb{E}_\pi \{(v_\pi(S) - v_w(S)) \nabla v_w(S)\} \approx \frac{1}{T} \sum_{t=0}^{T-1} (G_t - v_w(S_t)) \nabla v_w(S_t)$$

We can compute this one from our **observations!**

LS Training via Monte Carlo

We can then write this approximate gradient descent approach as

GD_MC_Eval():

- 1: Initiate with some initial w and learning rate α
- 2: **while** weights not converged **do**
- 3: Sample a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

- 4: Compute the sequence

$$(S_0, G_0) \rightarrow (S_1, G_1) \rightarrow \dots \rightarrow (S_{T-1}, G_{T-1})$$

- 5: Update weights as

$$w \leftarrow w +$$

- 6: **end while**

LS Training via Monte Carlo

We can then write this approximate gradient descent approach as

`GD_MC_Eval()`:

- 1: Initiate with some initial \mathbf{w} and learning rate α
- 2: **while** weights not converged **do**
- 3: Sample a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

- 4: Compute the sequence

$$(S_0, G_0) \rightarrow (S_1, G_1) \rightarrow \dots \rightarrow (S_{T-1}, G_{T-1})$$

- 5: Update weights as

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha}{T} \sum_{t=0}^{T-1} (G_t - v_{\mathbf{w}}(S_t)) \nabla v_{\mathbf{w}}(S_t)$$

- 6: **end while**

LS Training via Monte Carlo

This is not practical to wait for exact convergence: we try couple of episodes

GD_MC_Eval():

- 1: Initiate with some initial \mathbf{w} and learning rate α
- 2: **for** episode $k = 1 : K$ **do**
- 3: Sample a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

- 4: Compute the sequence

$$(S_0, G_0) \rightarrow (S_1, G_1) \rightarrow \dots \rightarrow (S_{T-1}, G_{T-1})$$

- 5: Update weights as

$$\mathbf{w} \leftarrow \mathbf{w} + \frac{\alpha}{T} \sum_{t=0}^{T-1} (G_t - v_{\mathbf{w}}(S_t)) \nabla v_{\mathbf{w}}(S_t)$$

- 6: **end for**

LS Training via Monte Carlo

- + *But, should we really update once an episode?!*
- Not really!

LS Training via Monte Carlo

- + But, should we really update once an episode?!
- Not really!

We can look at this algorithm as batch training with the batch being

$$(S_0, G_0), (S_1, G_1), \dots, (S_{T-1}, G_{T-1})$$

LS Training via Monte Carlo

- + But, should we really update *once an episode*?!
 - Not really!

We can look at this algorithm as batch training with the batch being

$$(S_0, G_0), (S_1, G_1), \dots, (S_{T-1}, G_{T-1})$$

We can also use mini-batches, and we can reduce the *mini-batch size* to 1: in this case, the estimator of the gradient will be a single sample, i.e.,

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha (G_t - v_{\mathbf{w}}(S_t)) \nabla v_{\mathbf{w}}(S_t)$$

LS Training via Monte Carlo

We can train the approximator via SGD and Monte Carlo

`SGD_MC_Eval()`:

- 1: Initiate with some initial \mathbf{w} and learning rate α
- 2: **for** episode $k = 1 : K$ **do**
- 3: Sample a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

- 4: Compute the sequence

$$(S_0, G_0) \rightarrow (S_1, G_1) \rightarrow \dots \rightarrow (S_{T-1}, G_{T-1})$$

- 5: **for** $t = 0 : T - 1$ **do**
- 6: Update weights as

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha (G_t - v_{\mathbf{w}}(S_t)) \nabla v_{\mathbf{w}}(S_t)$$

- 7: **end for**
- 8: **end for**

LS Training via Temporal Difference

- + So can't we use also TD to acquire an estimate?
- Sure!

LS Training via Temporal Difference

- + So can't we use also TD to acquire an estimate?
- Sure!

We can evaluate an estimator by TD: say we sample

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

LS Training via Temporal Difference

- + So can't we use also TD to acquire an estimate?
- Sure!

We can evaluate an estimator by TD: say we sample

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We can then compute an estimator of $v_\pi(S_t)$ via TD as

$$G_t^0 = R_{t+1} + \gamma v_w(S_{t+1})$$

LS Training via Temporal Difference

We can alternatively train the approximator via temporal difference

SGD_TD_Eval():

- 1: Initiate with some initial w and learning rate α
- 2: **for** episode $k = 1 : K$ **do**
- 3: Sample a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

- 4: **for** $t = 0 : T - 1$ **do**
- 5: Update weights as

$$w \leftarrow w + \alpha (R_{t+1} + \gamma v_w(S_{t+1}) - v_w(S_t)) \nabla v_w(S_t)$$

- 6: **end for**
- 7: **end for**

LS Training via Temporal Difference

We can alternatively train the approximator via temporal difference

`SGD_TD_Eval()`:

- 1: Initiate with some initial w and learning rate α
- 2: **for** episode $k = 1 : K$ **do**
- 3: Sample a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

- 4: **for** $t = 0 : T - 1$ **do**
- 5: Update weights as

$$w \leftarrow w + \alpha (R_{t+1} + \gamma v_w(S_{t+1}) - v_w(S_t)) \nabla v_w(S_t)$$

- 6: **end for**
- 7: **end for**

You can extend it to TD- n or TD $_{\lambda}$ as well

Back to Tabular RL

- + *These algorithm look like what we had before!*
- Well! This is actually an **extended version** of it!

Back to Tabular RL

- + These algorithm look like what we had before!
- Well! This is actually an extended version of it!

Let's consider a special case in which

- ① We use **tokenization** for feature representation

Back to Tabular RL

- + These algorithm look like what we had before!
- Well! This is actually an **extended version** of it!

Let's consider a special case in which

- ① We use **tokenization** for feature representation

Tokenization

We represent the feature vector as

$$\mathbf{x}(s^n) = \begin{bmatrix} 0 \\ \vdots \\ 1 \\ \vdots \\ 0 \end{bmatrix} \leftarrow \text{entry } n = \mathbf{1}\{s = s^n\}$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Linear Model

We approximate the value function via a linear transform

$$v_{\mathbf{w}}(s) = \mathbf{w}^T \mathbf{x}(s)$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Linear Model

We approximate the value function via a linear transform

$$v_{\mathbf{w}}(s) = \mathbf{w}^T \mathbf{x}(s)$$

With linear model and tokenization, our estimate of value at state s^n is

$$\hat{v}_{\pi}(s^n) = v_{\mathbf{w}}(s^n) = \mathbf{w}^T \mathbf{1}\{s = s^n\} = w_n$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Linear Model

We approximate the value function via a linear transform

$$v_{\mathbf{w}}(s) = \mathbf{w}^T \mathbf{x}(s)$$

With linear model and tokenization, our estimate of value at state s^n is

$$\hat{v}_{\pi}(s^n) = v_{\mathbf{w}}(s^n) = \mathbf{w}^T \mathbf{1}\{s = s^n\} = w_n$$

Also we have

$$\nabla v_{\mathbf{w}}(s^n) = \mathbf{x}(s^n) = \mathbf{1}\{s = s^n\}$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Let's now look at the SGD update with TD

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha (R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)) \nabla v_{\mathbf{w}}(S_t)$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Let's now look at the SGD update with TD

$$\begin{aligned}\mathbf{w} &\leftarrow \mathbf{w} + \alpha (R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)) \nabla v_{\mathbf{w}}(S_t) \\ &\leftarrow \mathbf{w} + \alpha \left(R_{t+1} + \gamma \underbrace{v_{\mathbf{w}}(S_{t+1})}_{\hat{v}_{\pi}(S_{t+1})} - \underbrace{v_{\mathbf{w}}(S_t)}_{\hat{v}_{\pi}(S_t)} \right) \underbrace{\mathbf{x}(S_t)}_{\mathbf{1}\{s=S_t\}}\end{aligned}$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Let's now look at the SGD update with TD

$$\begin{aligned} \mathbf{w} &\leftarrow \mathbf{w} + \alpha (R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)) \nabla v_{\mathbf{w}}(S_t) \\ &\leftarrow \mathbf{w} + \alpha \left(R_{t+1} + \gamma \underbrace{v_{\mathbf{w}}(S_{t+1})}_{\hat{v}_{\pi}(S_{t+1})} - \underbrace{v_{\mathbf{w}}(S_t)}_{\hat{v}_{\pi}(S_t)} \right) \underbrace{\mathbf{x}(S_t)}_{\mathbf{1}\{s=S_t\}} \end{aligned}$$

Say $S_t = s^n$: then, it is only entry n which gets update

$$w_n \leftarrow w_n + \alpha (R_{t+1} + \gamma \hat{v}_{\pi}(S_{t+1}) - \hat{v}_{\pi}(S_t))$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Say $S_t = s^n$: then, it is only entry n which gets updated

$$w_n \leftarrow w_n + \alpha (R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t))$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Say $S_t = s^n$: then, it is only entry n which gets updated

$$w_n \leftarrow w_n + \alpha (R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t))$$

and we know that $w_n = \hat{v}_\pi(s^n) = \hat{v}_\pi(S_t)$,

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Say $S_t = s^n$: then, it is only entry n which gets updated

$$w_n \leftarrow w_n + \alpha (R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t))$$

and we know that $w_n = \hat{v}_\pi(s^n) = \hat{v}_\pi(S_t)$, so we can write

$$\hat{v}_\pi(S_t) \leftarrow \hat{v}_\pi(S_t) + \alpha (R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t))$$

Back to Tabular RL

Let's consider a special case in which

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

Say $S_t = s^n$: then, it is only entry n which gets updated

$$w_n \leftarrow w_n + \alpha (R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t))$$

and we know that $w_n = \hat{v}_\pi(s^n) = \hat{v}_\pi(S_t)$, so we can write

$$\hat{v}_\pi(S_t) \leftarrow \hat{v}_\pi(S_t) + \alpha (R_{t+1} + \gamma \hat{v}_\pi(S_{t+1}) - \hat{v}_\pi(S_t))$$

Bingo! This is the **tabular TD update!**

Back to Tabular RL

Moral of Story

Tabular RL is a special case of RL with function approximation when

- ① We use **tokenization** for feature representation
- ② We use a **linear** model for approximation

*So, we expect learning **better** when we go for other **feature representation and approximation models***

Training Action-Value Approximator

- + In practice however we always need **action-values!** Right?
- Well! We can simply extend everything to state-action pairs

Training Action-Value Approximator

- + In practice however we always need **action-values!** Right?
- Well! We can simply extend everything to state-action pairs

Feature Representation of State-Actions

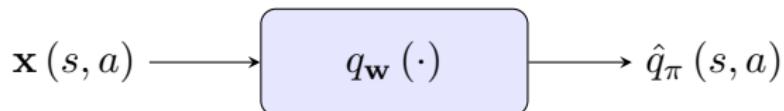
Feature representation maps each state-action pair into a vector of features that correspond to that state and action, i.e.,

$$\mathbf{x}(\cdot) : \mathcal{S} \times \mathcal{A} \mapsto \mathbb{R}^J$$

for some integer J that is the feature dimension

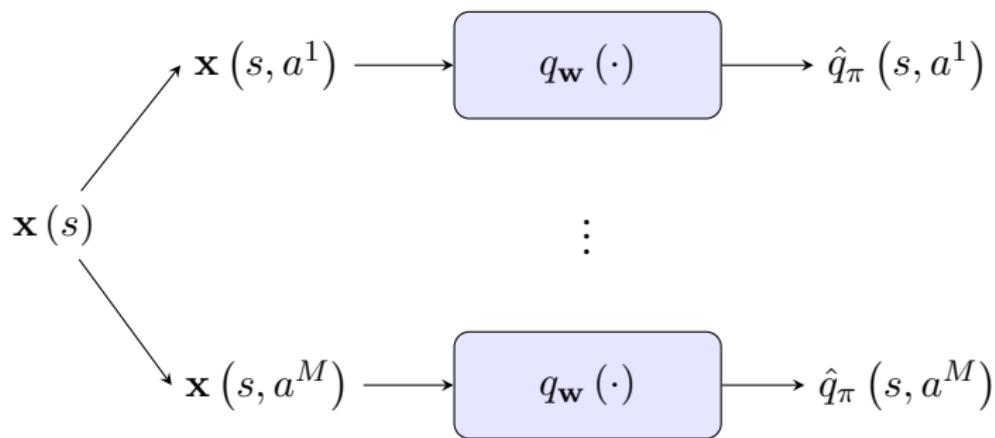
Action-Value Approximator: Form I

We can further consider an approximation model: *it maps the feature vector of each state-action pair (s, a) into its action-value*



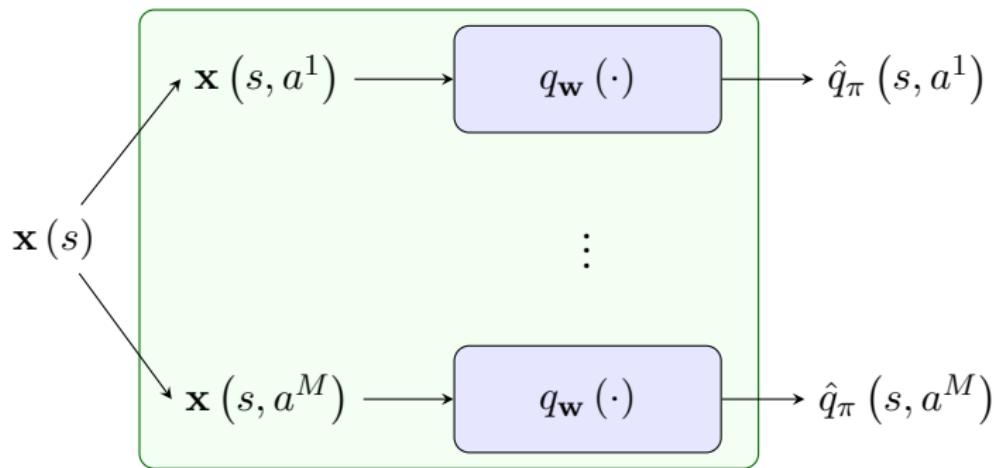
Action-Value Approximator: Form II

We may stack this approximator for various actions: *we can look at the end-to-end setting as a general approximator*



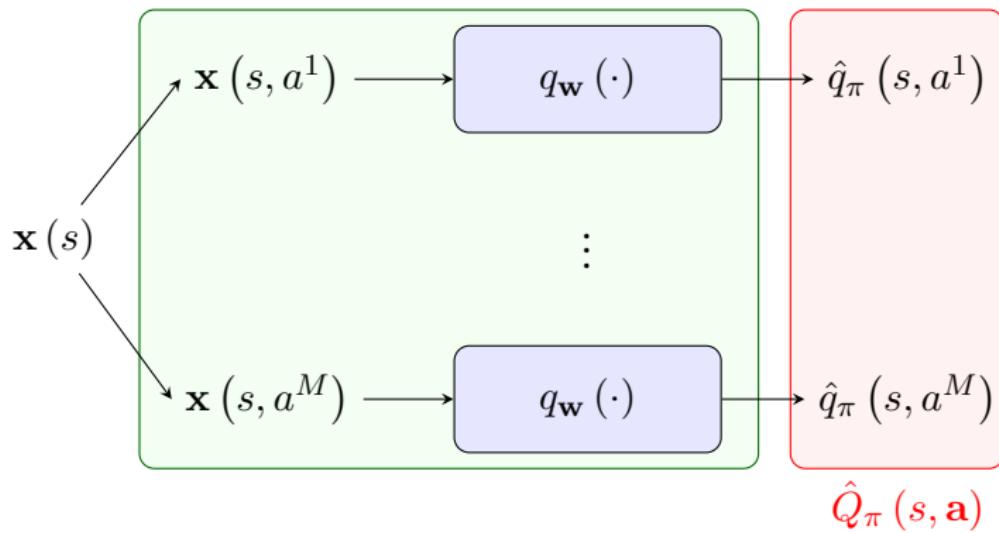
Action-Value Approximator: Form II

We may stack this approximator for various actions: *we can look at the end-to-end setting as a general approximator*



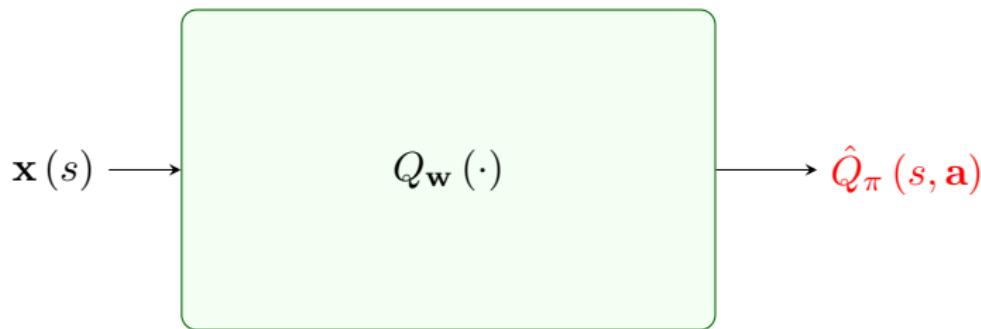
Action-Value Approximator: Form II

We may stack this approximator for various actions: *we can look at the end-to-end setting as a general approximator*



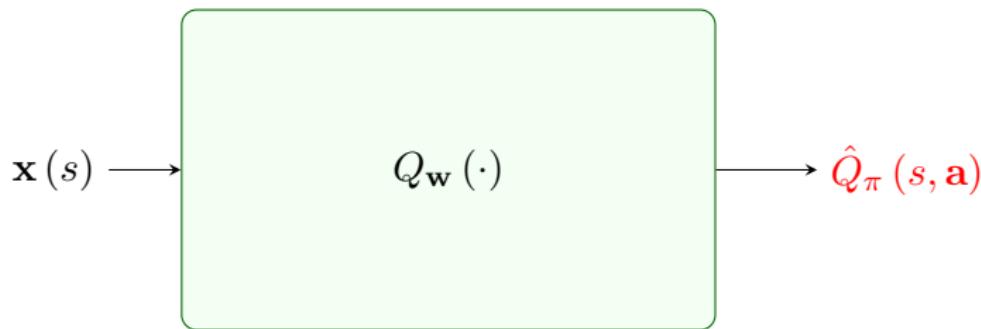
Action-Value Approximator: Form II

We may stack this approximator for various actions: *maybe, we can consider a general approximation model in this case*



Action-Value Approximator: Form II

We may stack this approximator for various actions: *maybe, we can consider a general approximation model in this case*



Let's make an agreement

$\hat{Q}_\pi(s, \mathbf{a})$ and $Q_{\mathbf{w}}(s, \mathbf{a})$ represent the complete **vector** of action-values
and $\hat{Q}_\pi(s, a)$ and $Q_{\mathbf{w}}(s, a)$ denote the **entry corresponding to a**

Training Approximation Model

The LS training can further be applied here:

Training Approximation Model

The LS training can further be applied here: *with the help of genie, we train the action-value approximator as*

$$\mathbf{w}^* = \min_{\mathbf{w}} \mathbb{E}_{\pi} \left\{ |Q_{\mathbf{w}}(S, A) - Q_{\pi}(S, A)|^2 \right\}$$

Training Approximation Model

The LS training can further be applied here: *with the help of genie, we train the action-value approximator as*

$$\mathbf{w}^* = \min_{\mathbf{w}} \mathbb{E}_{\pi} \left\{ |Q_{\mathbf{w}}(S, A) - Q_{\pi}(S, A)|^2 \right\}$$

which is iteratively solved via gradient descent using update rule

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbb{E}_{\pi} \left\{ (Q_{\pi}(S, A) - Q_{\mathbf{w}}(S, A)) \nabla Q_{\mathbf{w}}(S, A) \right\}$$

Training Approximation Model

The LS training can further be applied here: *with the help of genie, we train the action-value approximator as*

$$\mathbf{w}^* = \min_{\mathbf{w}} \mathbb{E}_{\pi} \left\{ |Q_{\mathbf{w}}(S, A) - Q_{\pi}(S, A)|^2 \right\}$$

which is iteratively solved via gradient descent using update rule

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \mathbb{E}_{\pi} \left\{ (Q_{\pi}(S, A) - Q_{\mathbf{w}}(S, A)) \nabla Q_{\mathbf{w}}(S, A) \right\}$$

Again, we could use Monte Carlo or TD to find an estimator of $Q_{\pi}(S, A)$

LS Training of Action-Value Approximator

Say we have sampled a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We can convert this trajectory into a sequence of (S_t, A_t, G_t)

LS Training of Action-Value Approximator

Say we have sampled a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We can convert this trajectory into a sequence of (S_t, A_t, G_t)

- Using Monte Carlo we can update by

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(G_t - Q_{\mathbf{w}}(S_t, A_t)) \nabla Q_{\mathbf{w}}(S_t, A_t)$$

LS Training of Action-Value Approximator

Say we have sampled a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We can convert this trajectory into a sequence of (S_t, A_t, G_t)

- Using Monte Carlo we can update by

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(G_t - Q_{\mathbf{w}}(S_t, A_t)) \nabla Q_{\mathbf{w}}(S_t, A_t)$$

- Using TD we can update as

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - Q_{\mathbf{w}}(S_t, A_t)) \nabla Q_{\mathbf{w}}(S_t, A_t)$$

LS Training of Action-Value Approximator

Say we have sampled a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

We can convert this trajectory into a sequence of (S_t, A_t, G_t)

- Using Monte Carlo we can update by

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(G_t - Q_{\mathbf{w}}(S_t, A_t)) \nabla Q_{\mathbf{w}}(S_t, A_t)$$

- Using TD we can update as

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - Q_{\mathbf{w}}(S_t, A_t)) \nabla Q_{\mathbf{w}}(S_t, A_t)$$

and we can find $v_{\mathbf{w}}(S_{t+1})$ as

$$v_{\mathbf{w}}(s) = \sum_{m=1}^M \pi(a^m|s) Q_{\mathbf{w}}(s, a^m)$$

LS Training of Action-Value Approximator

We can use all approaches we developed before

- only **difference** is that we replace the simple update by gradient descent
- it gets back to tabular RL if we use **tokenization** and **linear approximation**

LS Training of Action-Value Approximator

We can use all approaches we developed before

- only **difference** is that we replace the simple update by gradient descent
- it gets back to tabular RL if we use **tokenization** and **linear approximation**

Let's see a simple example: remember the backward view of TD_{λ}

LS Training of Action-Value Approximator

We can use all approaches we developed before

- only **difference** is that we replace the simple update by gradient descent
- it gets back to tabular RL if we use **tokenization** and **linear approximation**

Let's see a simple example: remember the backward view of TD_{λ}

- we **trace** the eligibility of each **state-action**

LS Training of Action-Value Approximator

We can use all approaches we developed before

- only **difference** is that we replace the simple update by gradient descent
- it gets back to tabular RL if we use **tokenization** and **linear approximation**

Let's see a simple example: remember the backward view of TD_{λ}

- we **trace** the eligibility of each **state-action**
- we **propagate** the update to previous state-action pairs

LS Training of Action-Value Approximator

We can use all approaches we developed before

- only **difference** is that we replace the simple update by gradient descent
- it gets back to tabular RL if we use **tokenization** and **linear approximation**

Let's see a simple example: remember the backward view of TD_λ

- we **trace** the eligibility of each **state-action**
- we **propagate** the update to previous state-action pairs

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all **state-action** pairs
- 2: **for** all state-action pairs (s, a) **do**
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: **end for**
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

This was with tabular RL which uses tokenization and linear approximation:

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

This was with tabular RL which uses tokenization and linear approximation: let's see if we can represent it in terms of approximation model components

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

This was with tabular RL which uses **tokenization** and **linear approximation**: let's see if we can represent it in terms of approximation model components

- with **tokenization**, we have NM -dimensional feature

$$\mathbf{x}(s^n, a^m) = \mathbf{1}\{s = s^n, a = a^m\}$$

LS Training of Action-Value Approximator

`ElgTrace($S_t, A_t, E(\cdot) | \lambda$):`

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

This was with tabular RL which uses **tokenization** and **linear approximation**: let's see if we can represent it in terms of approximation model components

- with **tokenization**, we have NM -dimensional feature

$$\mathbf{x}(s^n, a^m) = \mathbf{1}\{s = s^n, a = a^m\}$$

- with **linear** model, we have \mathbf{w} which is of the same dimension

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

This was with tabular RL which uses **tokenization** and **linear approximation**: let's see if we can represent it in terms of approximation model components

- with **tokenization**, we have NM -dimensional feature

$$\mathbf{x}(s^n, a^m) = \mathbf{1}\{s = s^n, a = a^m\}$$

- with **linear** model, we have \mathbf{w} which is of the same dimension
- with **linear** model, we have

$$\nabla Q_{\mathbf{w}}(s, a) = \mathbf{x}(s, a)$$

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

Let's define a vector E_w which has the NM entries of $E(S_t, A_t)$

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

Let's define a vector E_w which has the NM entries of $E(S_t, A_t)$

- we can write the update rule in line 4 as $E_w \leftarrow \gamma \lambda E_w$

LS Training of Action-Value Approximator

`ElgTrace($S_t, A_t, E(\cdot) | \lambda$):`

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

Let's define a vector E_w which has the NM entries of $E(S_t, A_t)$

- we can write the update rule in line 4 as $E_w \leftarrow \gamma \lambda E_w$
- and the update rule in line 5 as

$$E_w \leftarrow E_w + \mathbf{1}\{s = S_t, a = A_t\}$$

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

Let's define a vector E_w which has the NM entries of $E(S_t, A_t)$

- we can write the update rule in line 4 as $E_w \leftarrow \gamma \lambda E_w$
- and the update rule in line 5 as

$$\begin{aligned} E_w &\leftarrow E_w + \mathbf{1}\{s = S_t, a = A_t\} \\ &\quad \leftarrow E_w + \mathbf{x}(S_t, A_t) \end{aligned}$$

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

Let's define a vector E_w which has the NM entries of $E(S_t, A_t)$

- we can write the update rule in line 4 as $E_w \leftarrow \gamma \lambda E_w$
- and the update rule in line 5 as

$$\begin{aligned} E_w &\leftarrow E_w + \mathbf{1}\{s = S_t, a = A_t\} \\ &\leftarrow E_w + \mathbf{x}(S_t, A_t) \\ &\leftarrow E_w + \nabla Q_w(S_t, A_t) \end{aligned}$$

LS Training of Action-Value Approximator

ElgTrace($S_t, A_t, E(\cdot) | \lambda$):

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

Merging the two lines, we get into

$$E_w \leftarrow \gamma \lambda E_w + \nabla Q_w(S_t, A_t)$$

LS Training of Action-Value Approximator

`ElgTrace($S_t, A_t, E(\cdot) | \lambda$):`

- 1: Eligibility tracing function has NM components, i.e., $E(s, a)$ for all state-action pairs
- 2: for all state-action pairs (s, a) do
- 3: Update $E(s, a) \leftarrow \gamma \lambda E(s, a)$
- 4: end for
- 5: Update $E(S_t, A_t) \leftarrow E(S_t, A_t) + 1$

Merging the two lines, we get into

$$E_w \leftarrow \gamma \lambda E_w + \nabla Q_w(S_t, A_t)$$

This is the more general form of eligibility tracing

- we can use it for any approximation
- our trace is of the size of the feature vector

LS Training via TD_λ

So, we could evaluate via training as

SGD_TD_QEval(λ):

- 1: Initiate with some initial w and learning rate α
 - 2: **for** episode $k = 1 : K$ **do**
 - 3: Sample a trajectory

$$S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$$

- ```

4: for $t = 0 : T - 1$ do
5: Compute $\Delta = R_{t+1} + \gamma v_w(S_{t+1}) - Q_w(S_t, A_t)$ # forward propagation
6: Compute $\nabla = \nabla Q_w(S_t, A_t)$ # backpropagation
7: $E_w \leftarrow \lambda \gamma E_w + \nabla$
8: Update weights as
 $w \leftarrow w + \alpha \Delta E_w$
9: end for
10: end for

```

## Example: Mountain Car

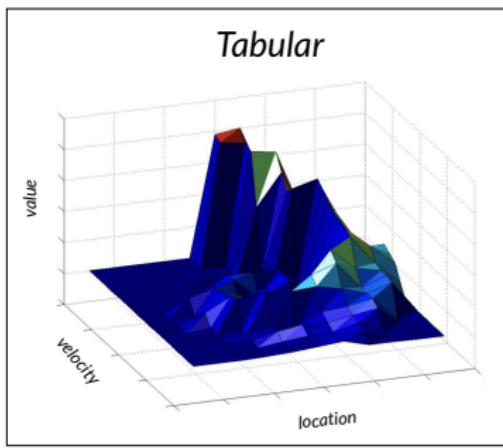


We can compare **tabular RL** against the one with **function approximation**

## Example: Mountain Car



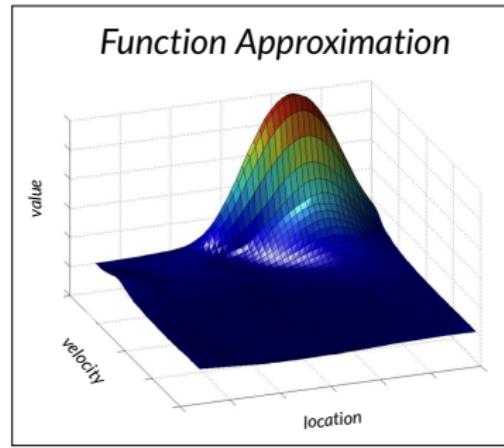
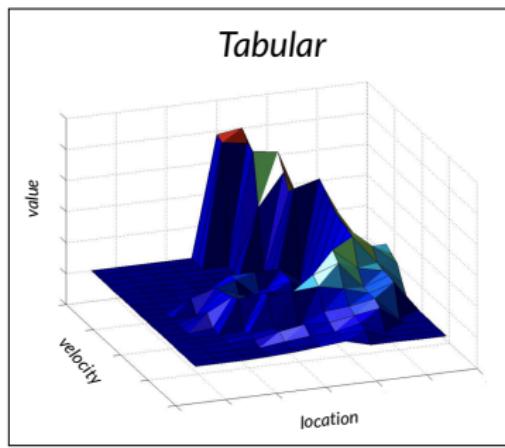
We can compare **tabular RL** against the one with **function approximation**



## Example: Mountain Car



We can compare **tabular RL** against the one with **function approximation**



# Back to Model-free RL: Control

- + But, we have in general ***prediction*** and ***control*** problems. In which one are we going to use function ***approximation***?
- Well, we can use in both

## Recall

We have two major problems in ***model-free*** RL

- ***Prediction*** in which for a given policy  $\pi$  we evaluate values by sampling the environment
- ***Control*** in which after each interaction, we improve our policy aiming to converge to the ***optimal policy***

# Back to Model-free RL: Control

- + But, we have in general ***prediction*** and ***control*** problems. In which one are we going to use function ***approximation***?
- Well, we can use in both

## Recall

We have two major problems in ***model-free*** RL

- ***Prediction*** in which for a given policy  $\pi$  we evaluate values by sampling the environment
- ***Control*** in which after each interaction, we improve our policy aiming to converge to the ***optimal policy***

Let's now go to the ***control***

## Recap: Online Control $\epsilon$ -Greedy

We have seen a typical control loop via  $\epsilon$ -greedy algorithm

X\_Control():

- 1: Initiate two random policies  $\pi$  and  $\bar{\pi}$
- 2: **while**  $\pi \neq \bar{\pi}$  **do**
- 3:    $\hat{q}_\pi = \text{X\_QUpdate}(\pi)$  and  $\pi \leftarrow \bar{\pi}$
- 4:    $\bar{\pi} = \epsilon\text{-Greedy}(\hat{q}_\pi)$
- 5: **end while**

## Recap: Online Control $\epsilon$ -Greedy

We have seen a typical control loop via  $\epsilon$ -greedy algorithm

X\_Control():

- 1: Initiate two random policies  $\pi$  and  $\bar{\pi}$
- 2: **while**  $\pi \neq \bar{\pi}$  **do**
- 3:    $\hat{q}_\pi = X\_QUpdate(\pi)$  and  $\pi \leftarrow \bar{\pi}$
- 4:    $\bar{\pi} = \epsilon\text{-Greedy}(\hat{q}_\pi)$
- 5: **end while**

We can realize this loop using function approximation

- ① Start with an initial **approximator**
- ② Use the **approximator** to **improve policy** via  $\epsilon$ -Greedy
- ③ Use **SGD** to update the **approximation** model **from observation**

# Recap: Online Control $\epsilon$ -Greedy

We have seen a typical control loop via  $\epsilon$ -greedy algorithm

```
X_Control():
```

- 1: Initiate two random policies  $\pi$  and  $\bar{\pi}$
- 2: **while**  $\pi \neq \bar{\pi}$  **do**
- 3:    $\hat{q}_\pi = X\_QUpdate(\pi)$  and  $\pi \leftarrow \bar{\pi}$     via function approximation
- 4:    $\bar{\pi} = \epsilon\text{-Greedy}(\hat{q}_\pi)$
- 5: **end while**

We can realize this loop using function approximation

- ① Start with an initial **approximator**
- ② Use the **approximator** to **improve policy** via  $\epsilon$ -Greedy
- ③ Use **SGD** to update the **approximation** model **from observation**

# Recap: Online Control $\epsilon$ -Greedy

We have seen a typical control loop via  $\epsilon$ -greedy algorithm

```
X_Control():
```

- 1: Initiate two random policies  $\pi$  and  $\bar{\pi}$
- 2: **while**  $\pi \neq \bar{\pi}$  **do**
- 3:    $\hat{q}_\pi = X\_QUpdate(\pi)$  and  $\pi \leftarrow \bar{\pi}$     via function approximation
- 4:    $\bar{\pi} = \epsilon\text{-Greedy}(\hat{q}_\pi)$
- 5: **end while**

We can realize this loop using function approximation

- ① Start with an initial **approximator**
- ② Use the **approximator** to **improve policy** via  $\epsilon$ -Greedy
- ③ Use **SGD** to update the **approximation** model **from observation**

We need an **action-value approximator** in this case

let's formally define the **Q-Network** then

# Q-Net $\equiv$ Action-Value Approximator – Form II

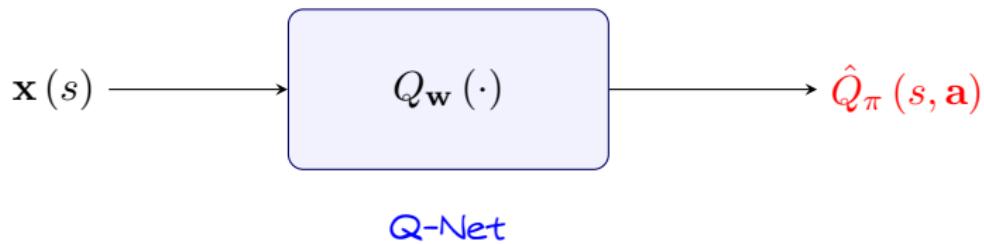
## Q-Network

*In the context of RL, the action-value approximation model that maps features to the vector of all action values is often referred to as Q-Net*

# Q-Net $\equiv$ Action-Value Approximator – Form II

## Q-Network

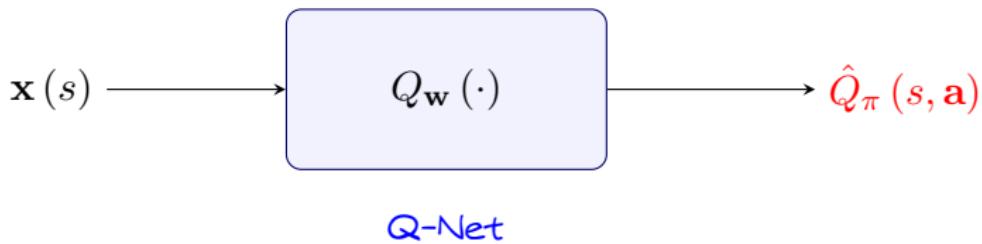
In the context of RL, the action-value approximation model that maps features to the vector of all action values is often referred to as Q-Net



# Q-Net $\equiv$ Action-Value Approximator – Form II

## Q-Network

In the context of RL, the action-value approximation model that maps features to the vector of all action values is often referred to as Q-Net



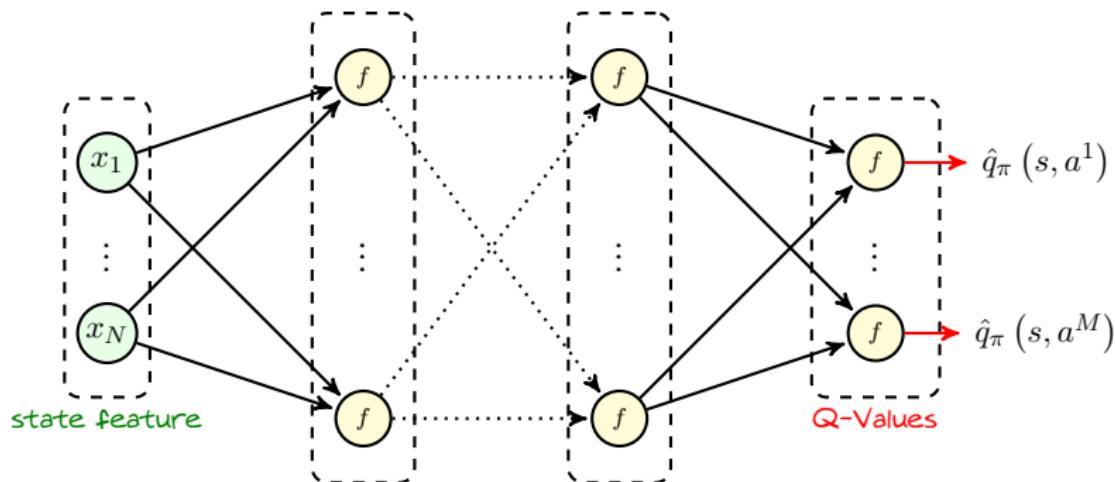
## Deep Q-Network

If we set Q-Net to be a DNN; then, it is usually called

Deep Q-Network  $\equiv$  DQN

## Example: MLP as DQN

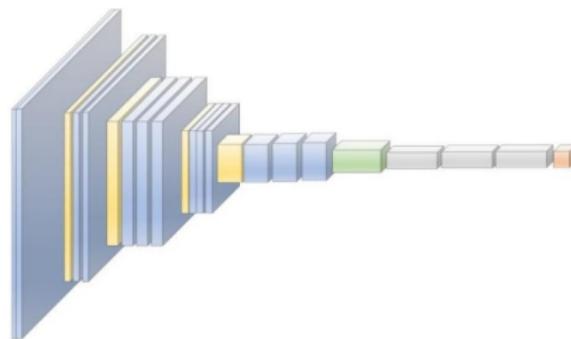
DQN can be simply an MLP



- Give the feature as an **input vector**
- The **MLP estimates all action-values**

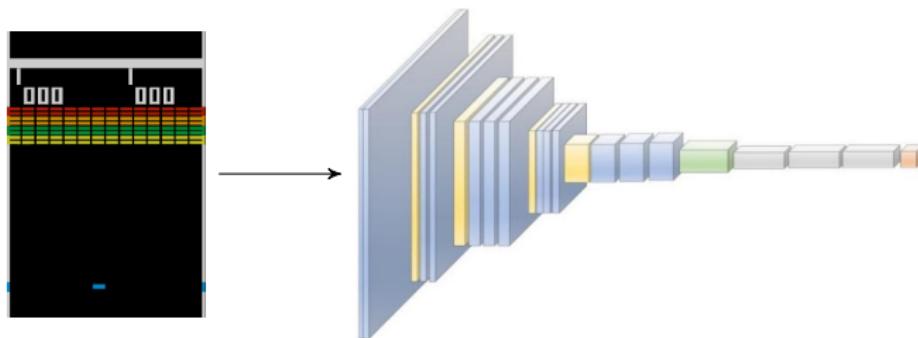
## Example: CNN as DQN

*It can be a convolutional neural network  $\equiv$  CNN*



## Example: CNN as DQN

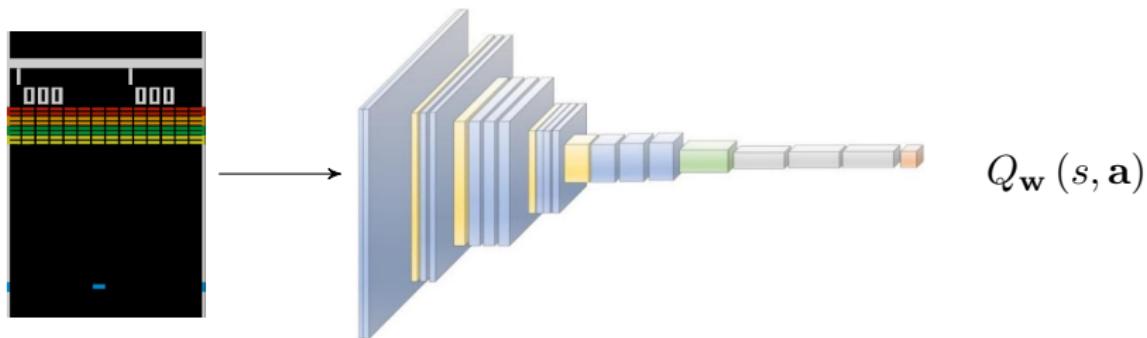
*It can be a convolutional neural network  $\equiv$  CNN*



- Give the feature as an **input tensor**, e.g., when feature is a frame

## Example: CNN as DQN

*It can be a convolutional neural network  $\equiv$  CNN*



- Give the feature as an **input tensor**, e.g., when feature is a frame
- The **CNN estimates all action-values**

# Building Control Loop with Q-Net

We can use Q-Net to build a control loop: *similar to tabular RL, we can have on-policy or off-policy approaches*

# Building Control Loop with Q-Net

We can use Q-Net to build a control loop: *similar to tabular RL, we can have on-policy or off-policy approaches*

- Deep *on-policy* RL
  - ↳ We can use a DQN to realize an *on-policy* control loop, e.g., SARSA
  - ↳ They are less in use as compared to *off-policy* versions

# Building Control Loop with Q-Net

We can use Q-Net to build a control loop: *similar to tabular RL, we can have on-policy or off-policy approaches*

- Deep *on-policy* RL
  - ↳ We can use a DQN to realize an *on-policy* control loop, e.g., SARSA
  - ↳ They are less in use as compared to *off-policy* versions
- Deep *off-policy* RL
  - ↳ We may use a DQN to realize an *off-policy* control loop, e.g., Q-learning

# Building Control Loop with Q-Net

We can use Q-Net to build a control loop: *similar to tabular RL, we can have on-policy or off-policy approaches*

- Deep *on-policy* RL
  - ↳ We can use a DQN to realize an *on-policy* control loop, e.g., SARSA
  - ↳ They are less in use as compared to *off-policy* versions
- Deep *off-policy* RL
  - ↳ We may use a DQN to realize an *off-policy* control loop, e.g., Q-learning
  - ↳ Due to some reasons deep *off-policy* RL is more popular
    - ↳ We can use *experience replay* in this case
    - ↳ They are therefore more *sample-efficient*
    - ↳ *No worries!* We will see these reasons ☺

# Building Control Loop with Q-Net

We can use Q-Net to build a control loop: *similar to tabular RL, we can have **on-policy** or **off-policy** approaches*

- Deep **on-policy** RL
  - ↳ We can use a DQN to realize an **on-policy** control loop, e.g., SARSA
  - ↳ They are less in use as compared to **off-policy** versions
- Deep **off-policy** RL
  - ↳ We may use a DQN to realize an **off-policy** control loop, e.g., Q-learning
  - ↳ Due to some reasons deep **off-policy** RL is more popular
    - ↳ We can use **experience replay** in this case
    - ↳ They are therefore more **sample-efficient**
    - ↳ **No worries!** We will see these reasons ☺

Let's start with **on-policy** algorithms

## Remark: Value Network

- + *But, when I look at internet, I usually see the term **value network**! Don't we have this concept?!*
- Sure! We have already worked with **value networks**

## Remark: Value Network

- + But, when I look at internet, I usually see the term **value network**! Don't we have this concept?!
- Sure! We have already worked with **value networks**

### Value Network

Any approximation model that gets **features** and returns **values** is a value function; this value could be a **state value** or an **action-value**

- Q-Net is a **value network**
- $v_w(\cdot)$  that we used for prediction was also **value network**

## Remark: Value Network

- + But, when I look at internet, I usually see the term **value network**! Don't we have this concept?!
- Sure! We have already worked with **value networks**

### Value Network

Any approximation model that gets **features** and returns **values** is a value function; this value could be a **state value** or an **action-value**

- Q-Net is a **value network**
  - $v_w(\cdot)$  that we used for **prediction** was also **value network**
- 
- + Then, do we have any other sort of networks?!
  - Yes! We could have **policy networks**: we will see them in the next chapter

## Recap: Going On-Policy

Let's look back at our TD-based prediction algorithm

**SGD\_TD\_QEval**( $\lambda$ ) :

- ```

1: Initiate with some initial  $\mathbf{w}$  and learning rate  $\alpha$ 
2: for episode  $k = 1 : K$  do
3:   Sample a trajectory  $S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$ 
4:   for  $t = 0 : T - 1$  do
5:     Compute  $\Delta_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - Q_{\mathbf{w}}(S_t, A_t)$       # forward propagation
6:     Compute  $\nabla_t = \nabla Q_{\mathbf{w}}(S_t, A_t)$                                 # backpropagation
7:      $E_{\mathbf{w}} \leftarrow \lambda \gamma E_{\mathbf{w}} + \nabla_t$ 
8:     Update weights as

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta_t E_{\mathbf{w}}$$

9:   end for
10:  end for

```

Recap: Going On-Policy

Let's look back at our TD-based prediction algorithm

SGD_TD_QEval(λ) :

- ```

1: Initiate with some initial \mathbf{w} and learning rate α
2: for episode $k = 1 : K$ do
3: Sample a trajectory $S_0, A_0 \xrightarrow{R_1} S_1, A_1 \xrightarrow{R_2} \dots \xrightarrow{R_{T-1}} S_{T-1}, A_{T-1} \xrightarrow{R_T} S_T$
4: for $t = 0 : T - 1$ do
5: Compute $\Delta_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - Q_{\mathbf{w}}(S_t, A_t)$ # forward propagation
6: Compute $\nabla_t = \nabla Q_{\mathbf{w}}(S_t, A_t)$ # backpropagation
7: $E_{\mathbf{w}} \leftarrow \lambda \gamma E_{\mathbf{w}} + \nabla_t$
8: Update weights as

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta_t E_{\mathbf{w}}$$

9: end for
10: end for

```

The key point in going on policy is to evaluate  $v_w(S_{t+1})$  using our *actual policy*

## SARSA: Going On-Policy

We can modify line *line 5*: we compute  $v_{\mathbf{w}}(S_{t+1})$  as

$$v_{\mathbf{w}}(S_{t+1}) = \sum_{m=1}^M \pi(a^m | S_{t+1}) Q_{\mathbf{w}}(S_{t+1}, a^m)$$

## SARSA: Going On-Policy

We can modify line *line 5*: we compute  $v_{\mathbf{w}}(S_{t+1})$  as

$$v_{\mathbf{w}}(S_{t+1}) = \sum_{m=1}^M \pi(a^m | S_{t+1}) Q_{\mathbf{w}}(S_{t+1}, a^m)$$

In *on-policy* approach, we *act* before we update

our policy leads us to next action  $A_{t+1}$

## SARSA: Going On-Policy

We can modify line *line 5*: we compute  $v_{\mathbf{w}}(S_{t+1})$  as

$$v_{\mathbf{w}}(S_{t+1}) = \sum_{m=1}^M \pi(a^m | S_{t+1}) Q_{\mathbf{w}}(S_{t+1}, a^m)$$

In *on-policy* approach, we *act* before we update

our policy leads us to next action  $A_{t+1}$

---

So, we could *move on our policy* and write

$$\pi(a | S_{t+1}) = \begin{cases} 1 & a = A_{t+1} \\ 0 & a \neq A_{t+1} \end{cases}$$

## SARSA: Going On-Policy

We can modify line *line 5*: we compute  $v_{\mathbf{w}}(S_{t+1})$  as

$$v_{\mathbf{w}}(S_{t+1}) = \sum_{m=1}^M \pi(a^m | S_{t+1}) Q_{\mathbf{w}}(S_{t+1}, a^m)$$

In *on-policy* approach, we *act* before we update

our policy leads us to next action  $A_{t+1}$

So, we could *move on our policy and write*

$$\pi(a | S_{t+1}) = \begin{cases} 1 & a = A_{t+1} \\ 0 & a \neq A_{t+1} \end{cases} \rightsquigarrow v_{\mathbf{w}}(S_{t+1}) = Q_{\mathbf{w}}(S_{t+1}, A_{t+1})$$

## SARSA with Q-Net

SGD\_SARSA() :

- 1: Initiate with  $w$  and learning rate  $\alpha$
  - 2: **for**  $episode = 1 : K$  or until  $\pi$  stops changing **do**
  - 3:   Initiate with a random state-action pair  $(S_0, A_0)$
  - 4:   **for**  $t = 0 : T - 1$  where  $S_T$  is either terminal or terminated **do**
  - 5:     Act  $A_t$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
  - 6:     Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_w(S_t, A_t))$
  - 7:     Draw the new action  $A_{t+1}$  from  $\pi(\cdot | S_{t+1})$  and move on policy

$$S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}, A_{t+1}$$

- ```

8:     Set  $\Delta \leftarrow R_{t+1} + \gamma Q_w(S_{t+1}, A_{t+1}) - Q_w(S_t, A_t)$       # forward propagation
9:     Update  $w \leftarrow w + \alpha \Delta \nabla Q_w(S_t, A_t)$                       # backpropagation
10:    end for
11: end for

```

SARSA with Q-Net and Eligibility Tracing

We have seen that *with function approximation eligibility tracing reduces to*

$$E_{\mathbf{w}} \leftarrow \gamma \lambda E_{\mathbf{w}} + \nabla Q_{\mathbf{w}} (S_t, A_t)$$

SARSA with Q-Net and Eligibility Tracing

We have seen that *with function approximation eligibility tracing reduces to*

$$E_{\mathbf{w}} \leftarrow \gamma \lambda E_{\mathbf{w}} + \nabla Q_{\mathbf{w}}(S_t, A_t)$$

Let's fit it into our SARSA control loop!

SARSA with Q-Net and Eligibility Tracing

We have seen that *with function approximation eligibility tracing reduces to*

$$E_{\mathbf{w}} \leftarrow \gamma \lambda E_{\mathbf{w}} + \nabla Q_{\mathbf{w}}(S_t, A_t)$$

Let's fit it into our SARSA control loop!

Eligibility Tracing \propto Backpropagation

If we use a DQN, we should **backpropagate** to compute the *eligibility tracing*: this point is **intuitive** as both approaches naturally follow the same logic

SARSA(λ) with Function Approximation

SGD_SARSA(λ):

- 1: Initiate with w and learning rate α
 - 2: **for** episode = 1 : K or until π stops changing **do**
 - 3: Initiate with a random state-action pair (S_0, A_0)
 - 4: **for** $t = 0 : T - 1$ where S_T is either terminal or terminated **do**
 - 5: Act A_t and observe $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
 - 6: Update policy to $\pi \leftarrow \epsilon\text{-Greedy}(Q_w(S_t, a))$
 - 7: Draw the new action A_{t+1} from $\pi(\cdot | S_{t+1})$ and move on policy

$$S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}, A_{t+1}$$

- ```

8: Set $\Delta \leftarrow R_{t+1} + \gamma Q_w(S_{t+1}, A_{t+1}) - Q_w(S_t, A_t)$ # forward propagation
9: $E_w \leftarrow \lambda\gamma E_w + \nabla Q_w(S_t, A_t)$ # backpropagation
10: Update $w \leftarrow w + \alpha \Delta E_w$
11: end for
12: end for

```

## Recap: Going Off-Policy

In off-policy control: we *behave* with a policy  $\pi$  but *update* by a *target policy*  $\bar{\pi}$

- We could use *importance sampling* to evaluate *target policy*
- We mainly focused on *Q-learning approach*

## Recap: Going Off-Policy

In off-policy control: we *behave* with a policy  $\pi$  but *update* by a *target policy*  $\bar{\pi}$

- We could use *importance sampling* to evaluate *target policy*
- We mainly focused on *Q-learning approach*

### Q-Learning

Q-learning is an *off-policy TD control* algorithm, where we sample with  $\epsilon$ -greedy policy but update with greedy policy

## Recap: Going Off-Policy

In off-policy control: we *behave* with a policy  $\pi$  but *update* by a target policy  $\bar{\pi}$

- We could use *importance sampling* to evaluate target policy
- We mainly focused on *Q-learning approach*

### Q-Learning

Q-learning is an *off-policy TD control* algorithm, where we sample with  $\epsilon$ -greedy policy but update with greedy policy

Key property of Q-learning is that we don't really need *importance sampling*: we could directly *update* as

$$\hat{q}_{\bar{\pi}}(S_t, A_t) \leftarrow \hat{q}_{\bar{\pi}}(S_t, A_t) + \alpha \left( R_{t+1} + \gamma \max_m \hat{q}_{\bar{\pi}}(S_{t+1}, a^m) - \hat{q}_{\bar{\pi}}(S_t, A_t) \right)$$

# Q-Learning with Q-Net

We can therefore extend our Q-learning algorithm to

SGD\_Q-Learning() :

- 1: Initiate with  $w$  and learning rate  $\alpha$
- 2: **for** episode = 1 :  $K$  or until  $\pi$  stops changing **do**
- 3:   Initiate with a random state  $S_0$
- 4:   **for**  $t = 0 : T - 1$  where  $S_T$  is either terminal or terminated **do**
- 5:     Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_w(S_t, a))$
- 6:     Draw action  $A_t$  from  $\pi(\cdot | S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 7:      $\Delta \leftarrow R_{t+1} + \gamma \max_m Q_w(S_{t+1}, a^m) - Q_w(S_t, A_t)$  # forward propagation
- 8:     Update  $w \leftarrow w + \alpha \Delta \nabla Q_w(S_t, A_t)$  # backpropagation
- 9:   **end for**
- 10: **end for**

We could potentially use *eligibility tracing* as well!

# Incremental Algorithms: Challenges

What we have developed by now are the so-called *incremental approaches*

## Incremental Algorithms

*Incremental algorithms* use the *actual sample* at each time to *update* the Q-Net

# Incremental Algorithms: Challenges

What we have developed by now are the so-called *incremental approaches*

## Incremental Algorithms

*Incremental algorithms* use the *actual sample* at each time to *update* the Q-Net

Though easy to implement, *incremental approaches* are *not* efficient

- ① They are extremely *sample-inefficient*
  - ↳ Once we use a sample, we are *over* with it

# Incremental Algorithms: Challenges

What we have developed by now are the so-called *incremental approaches*

## Incremental Algorithms

*Incremental algorithms* use the *actual sample* at each time to *update* the Q-Net

Though easy to implement, *incremental approaches* are *not* efficient

### ① They are extremely *sample-inefficient*

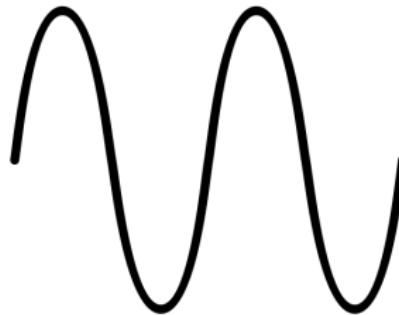
- ↳ Once we use a sample, we are *over* with it
- ↳ But, in deep learning we used to use the same samples *several times*
  - ↳ We make a training loop with *multiple epochs*
  - ↳ In each epoch, we go through the *whole dataset*

## Incremental Algorithms: Challenges

Though easy to implement, **incremental approaches** are **not efficient**

- ② They use samples that are **strongly correlated**

↳ In sample  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}, A_{t+1}$  states  $S_t$  and  $S_{t+1}$  are closely related

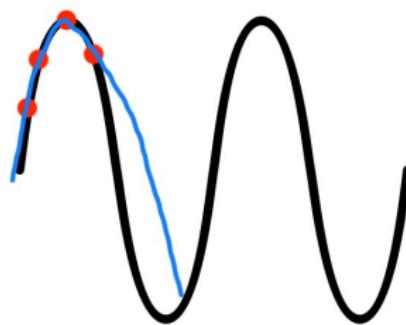


## Incremental Algorithms: Challenges

Though easy to implement, **incremental approaches** are **not efficient**

- ② They use samples that are **strongly correlated**

- ↳ In sample  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}, A_{t+1}$  states  $S_t$  and  $S_{t+1}$  are closely related
    - ↳ If state is the location; then, locations in time  $t$  and  $t + 1$  are very close

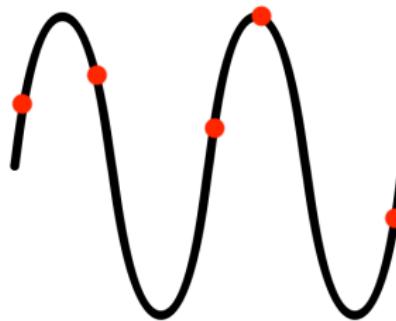


# Incremental Algorithms: Challenges

Though easy to implement, **incremental approaches** are **not efficient**

## ② They use samples that are **strongly correlated**

- ↳ In sample  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}, A_{t+1}$  states  $S_t$  and  $S_{t+1}$  are closely related
  - ↳ If state is the location; then, locations in time  $t$  and  $t + 1$  are very close
- ↳ But, we know that we need **independent** samples
  - ↳ With correlated samples we can easily **stick to a local fitting**
  - ↳ Our Q-Net **does not generalize** very well



## Solution: Batch Training

The solution to these challenges is to use *batch training*

- ① Save every sample in a *database*

↳ We can only save the values we need, i.e., estimator values

## Solution: Batch Training

The solution to these challenges is to use *batch training*

- ① Save every sample in a *database*

- ↳ We can only save the values we need, i.e., estimator values

- ② In each iteration *sample* from *database*

- ↳ We can reuse (reply) our *sampled experiences*

- ↳ We can *reduce* the *correlation* between succeeding samples

## Solution: Batch Training

The solution to these challenges is to use *batch training*

- ① Save every sample in a *database*
  - ↳ We can only save the values we need, i.e., estimator values
- ② In each iteration *sample* from *database*
  - ↳ We can reuse (reply) our *sampled experiences*
  - ↳ We can *reduce* the *correlation* between succeeding samples

This is what we call *experience reply*

- This needs us to control *off-policy*
  - ↳ We are using samples of *previous non-improved policies*
  - ↳ We want to update the value of a *different target policy*

## Solution: Batch Training

The solution to these challenges is to use *batch training*

- ① Save every sample in a *database*
  - ↳ We can only save the values we need, i.e., estimator values
- ② In each iteration *sample* from *database*
  - ↳ We can reuse (reply) our *sampled experiences*
  - ↳ We can *reduce* the *correlation* between succeeding samples

This is what we call *experience reply*

- This needs us to control *off-policy*
  - ↳ We are using samples of *previous non-improved policies*
  - ↳ We want to update the value of a *different target policy*
- This is why deep *off-policy* algorithms are *more sample-efficient*

## Solution: Batch Training

The solution to these challenges is to use *batch training*

- ① Save every sample in a *database*
  - ↳ We can only save the values we need, i.e., estimator values
- ② In each iteration *sample* from *database*
  - ↳ We can reuse (reply) our *sampled experiences*
  - ↳ We can *reduce* the *correlation* between succeeding samples

This is what we call *experience reply*

- This needs us to control *off-policy*
  - ↳ We are using samples of *previous non-improved policies*
  - ↳ We want to update the value of a *different target policy*
- This is why deep *off-policy* algorithms are *more sample-efficient*

We next study this in details ☺

# Vanilla Deep Q-Learning

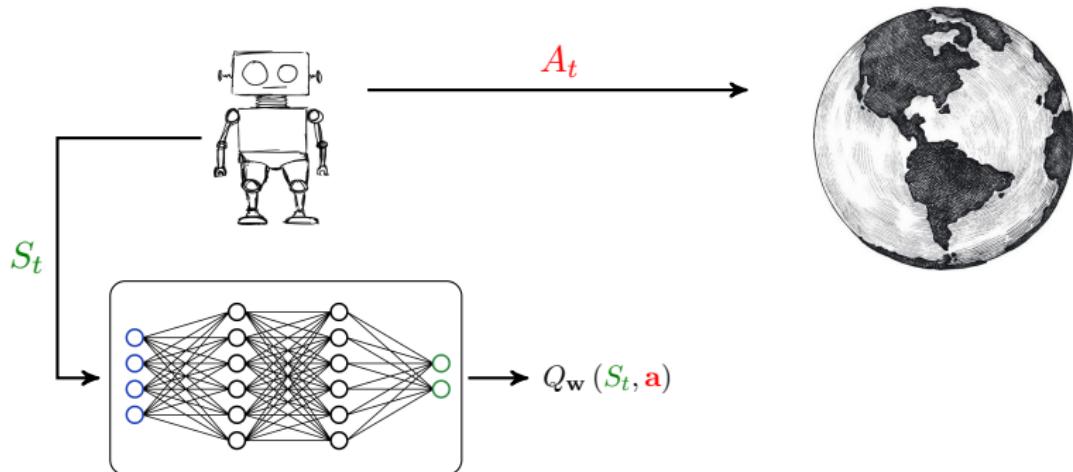
SGD\_Q-Learning() :

- 1: Initiate with  $w$  and learning rate  $\alpha$
- 2: **for** episode = 1 :  $K$  or until  $\pi$  stops changing **do**
- 3:   Initiate with a random state  $S_0$
- 4:   **for**  $t = 0 : T - 1$  where  $S_T$  is either terminal or terminated **do**
- 5:     Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_w(S_t, a))$
- 6:     Draw action  $A_t$  from  $\pi(\cdot | S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 7:      $\Delta \leftarrow R_{t+1} + \gamma \max_m Q_w(S_{t+1}, a^m) - Q_w(S_t, A_t)$  # forward propagation
- 8:     Update  $w \leftarrow w + \alpha \Delta \nabla Q_w(S_t, A_t)$  # backpropagation
- 9:   **end for**
- 10: **end for**

In **deep** Q-learning, we use a DQN to perform offline control via Q-learning

$$\text{deep Q-learning} \equiv \text{DQL}$$

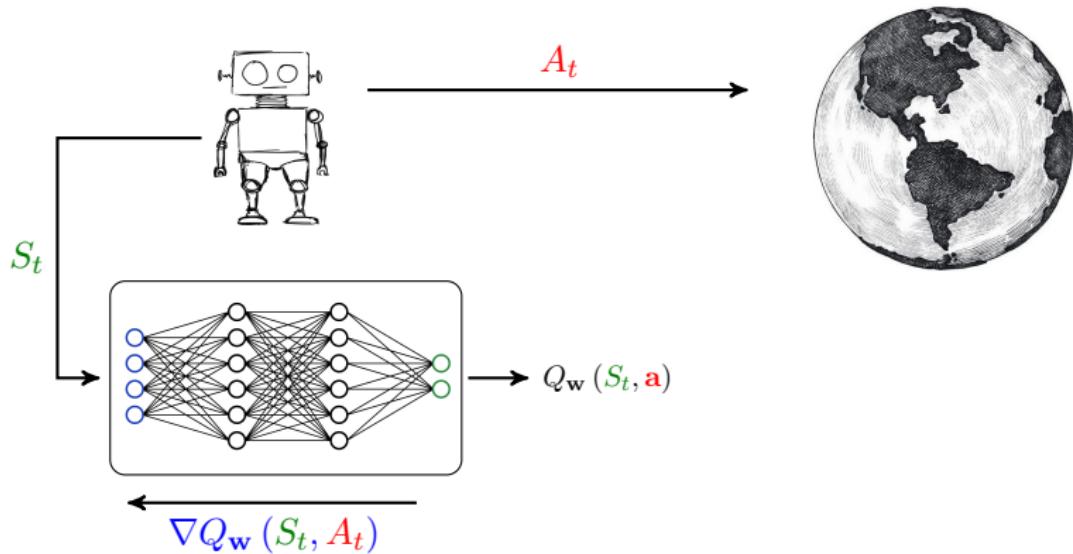
# Vanilla DQL: Visualization



We update the weights on the DQN as  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta$

$$\Delta \leftarrow -Q_{\mathbf{w}}(S_t, A_t)$$

# Vanilla DQL: Visualization

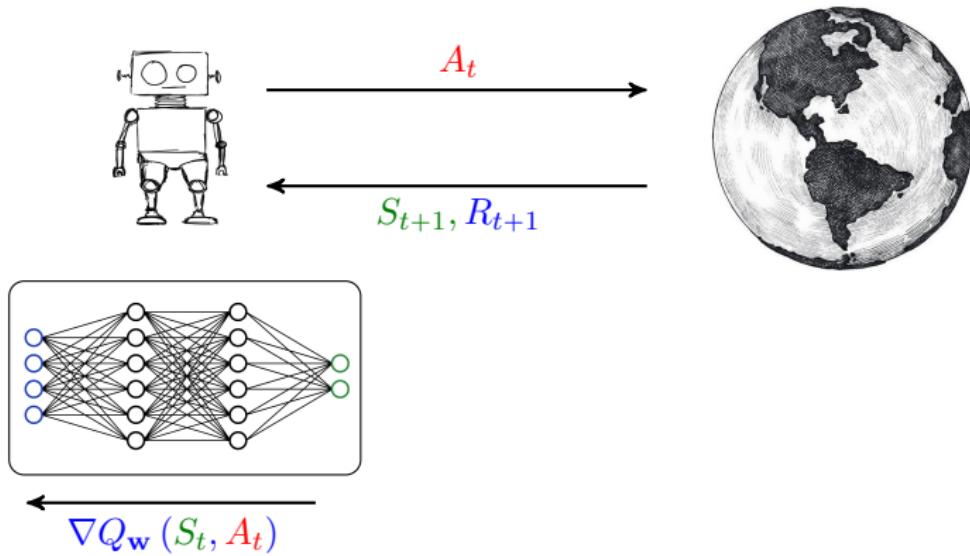


We update the weights on the DQN as  $w \leftarrow w + \alpha \Delta \nabla Q_w(S_t, A_t)$

$$\Delta \leftarrow$$

$$-Q_w(S_t, A_t)$$

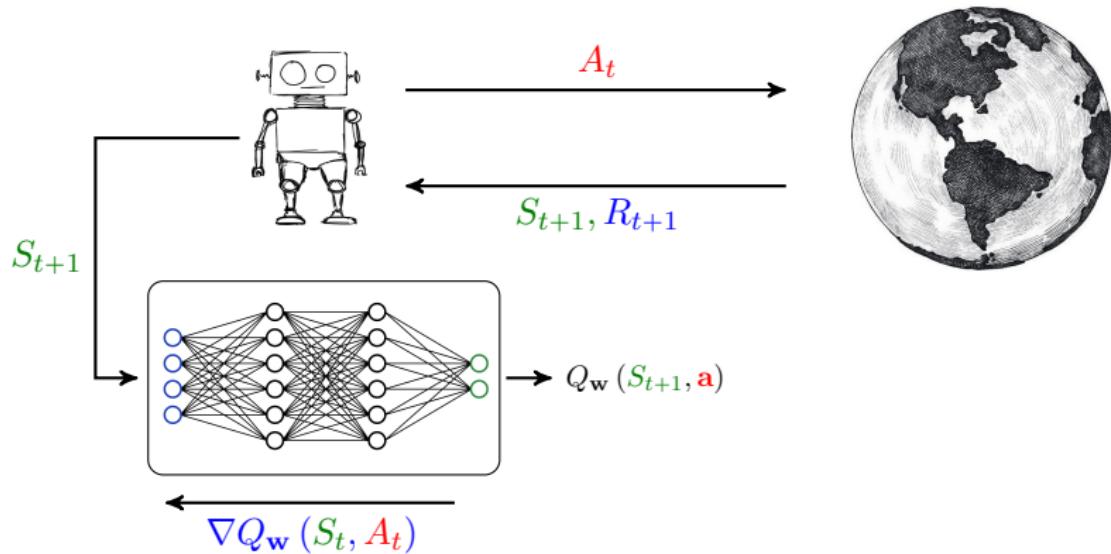
# Vanilla DQL: Visualization



We update the weights on the DQN as  $w \leftarrow w + \alpha \Delta \nabla Q_w(S_t, A_t)$

$$\Delta \leftarrow -Q_w(S_t, A_t)$$

# Vanilla DQL: Visualization



We update the weights on the DQN as  $w \leftarrow w + \alpha \Delta \nabla Q_w(S_t, A_t)$

$$\Delta \leftarrow [R_{t+1} + \gamma \max_m Q_w(S_{t+1}, a^m)] - Q_w(S_t, A_t)$$

# Vanilla Deep Q-Learning: Challenges

Vanilla DQL does **not** perform **impressive**: it suffers from two major challenges

- ① We deal with **strongly correlated** samples
  - ↳ We handle this issue via **experience reply**

Let's first get to **experience replay**

# Vanilla Deep Q-Learning: Challenges

Vanilla DQL does **not** perform *impressive*: it suffers from two major challenges

- ① We deal with *strongly correlated* samples

- ↳ We handle this issue via *experience reply*

- ② The labels in the sample data-points change in each iteration

$$\Delta \leftarrow \boxed{R_{t+1} + \gamma \max_m Q_{\mathbf{w}}(S_{t+1}, a^m)} - Q_{\mathbf{w}}(S_t, A_t)$$

- ↳ Here, we can look at each sampled state as a data sample with label

$$y_t = R_{t+1} + \gamma \max_m Q_{\mathbf{w}}(S_{t+1}, a^m)$$

Let's first get to *experience replay*

# Vanilla Deep Q-Learning: Challenges

Vanilla DQL does **not** perform **impressive**: it suffers from two major challenges

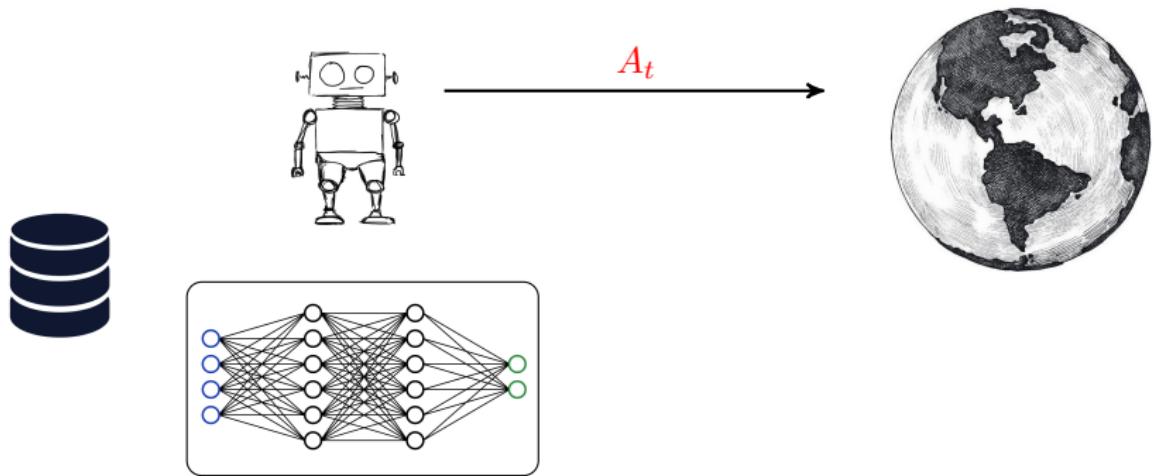
- ① We deal with **strongly correlated samples**
  - ↳ We handle this issue via **experience reply**
- ② The labels in the sample data-points change in each iteration
 
$$\Delta \leftarrow \boxed{R_{t+1} + \gamma \max_m Q_{\mathbf{w}}(S_{t+1}, a^m)} - Q_{\mathbf{w}}(S_t, A_t)$$
  - ↳ Here, we can look at each sampled state as a data sample with label

$$y_t = R_{t+1} + \gamma \max_m Q_{\mathbf{w}}(S_{t+1}, a^m)$$

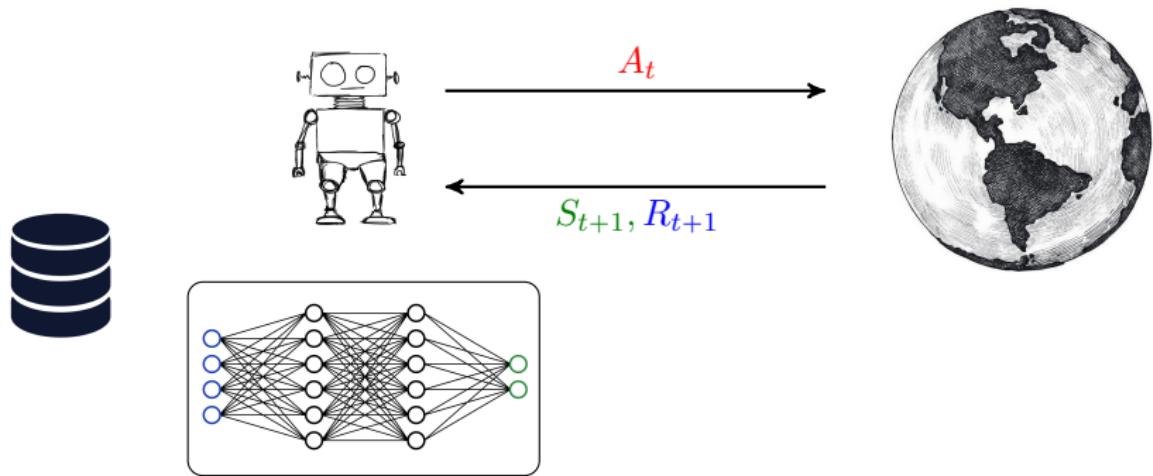
- ↳ After each update of  $\mathbf{w}$  this label changes
- ↳ This results in **divergence** or high **error variance**
- ↳ We are going to over-come this issue by using a **target network**

**Let's first get to experience replay**

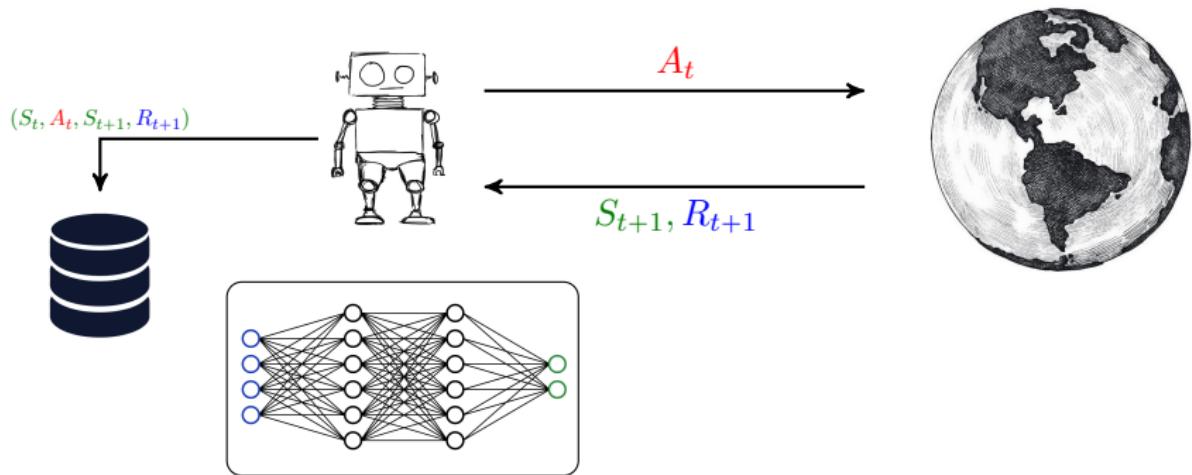
# Experience Replay



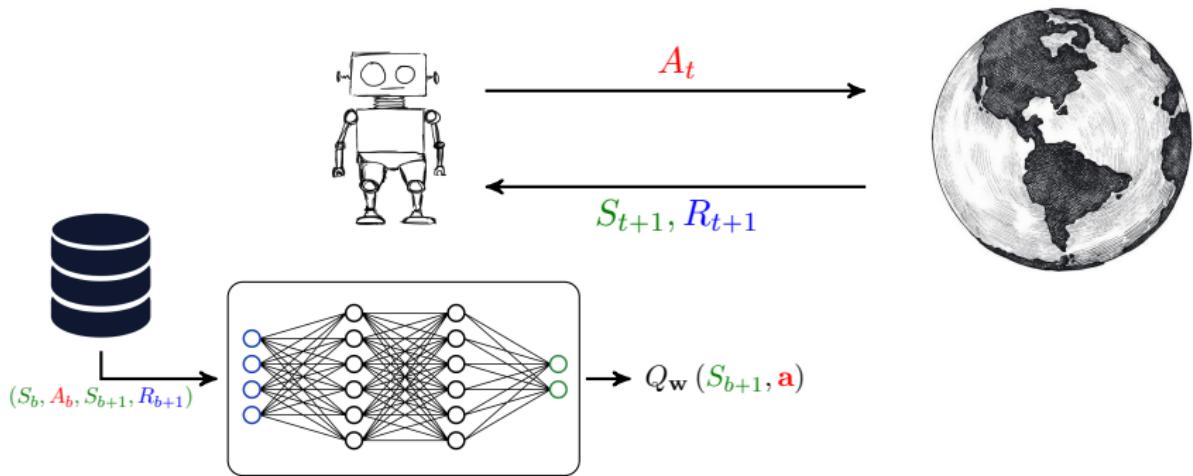
# Experience Replay



# Experience Replay



# Experience Replay



We update DQN by mini-batches  $\mathbf{w} \leftarrow \mathbf{w} + \sum_b \alpha \Delta_b \nabla Q_{\mathbf{w}}(S_b, A_b)$

$$\Delta_b \leftarrow \boxed{R_{b+1} + \gamma \max_m Q_{\mathbf{w}}(S_{b+1}, \mathbf{a}^m)} - Q_{\mathbf{w}}(S_b, A_b)$$

# DQL with Experience Replay

DQL\_v1() :

- 1: Initiate with  $\mathbf{w}$ , empty replay buffer  $\mathbb{D}$  and learning rate  $\alpha$
- 2: **for** episode = 1 :  $K$  or until  $\pi$  stops changing **do**
- 3:   Initiate with a random state  $S_0$
- 4:   **for**  $t = 0 : T - 1$  where  $S_T$  is either terminal or terminated **do**
- 5:     Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_{\mathbf{w}}(S_t, \mathbf{a}))$
- 6:     Draw action  $A_t$  from  $\pi(\cdot | S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 7:     Add sample  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$  to the replay buffer  $\mathbb{D}$
- 8:   **for** iteration  $\ell = 1 : L$  **do**
- 9:     Sample mini-batch  $\mathbb{B} = \{S_b, A_b \xrightarrow{R_{b+1}} S_{b+1} \text{ for } b = 1 : B\}$  from  $\mathbb{D}$
- 10:      $\Delta_b \leftarrow R_{b+1} + \gamma \max_m Q_{\mathbf{w}}(S_{b+1}, \mathbf{a}^m) - Q_{\mathbf{w}}(S_b, A_b)$
- 11:     Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_{b=1}^B \Delta_b \nabla Q_{\mathbf{w}}(S_b, A_b)$
- 12:   **end for**
- 13:   **end for**
- 14: **end for**

## DQL with Experience Replay

In general, we can iterate **multiple mini-batches**, i.e.,  $L > 1$

- *This can improve the convergence speed*
- *The trained DQN may however stick to a bad local minima*

In practice we typically set  $L = 1$

- *with a relatively large batch-size we can see good convergence results*

## DQL with Experience Replay

In general, we can iterate **multiple mini-batches**, i.e.,  $L > 1$

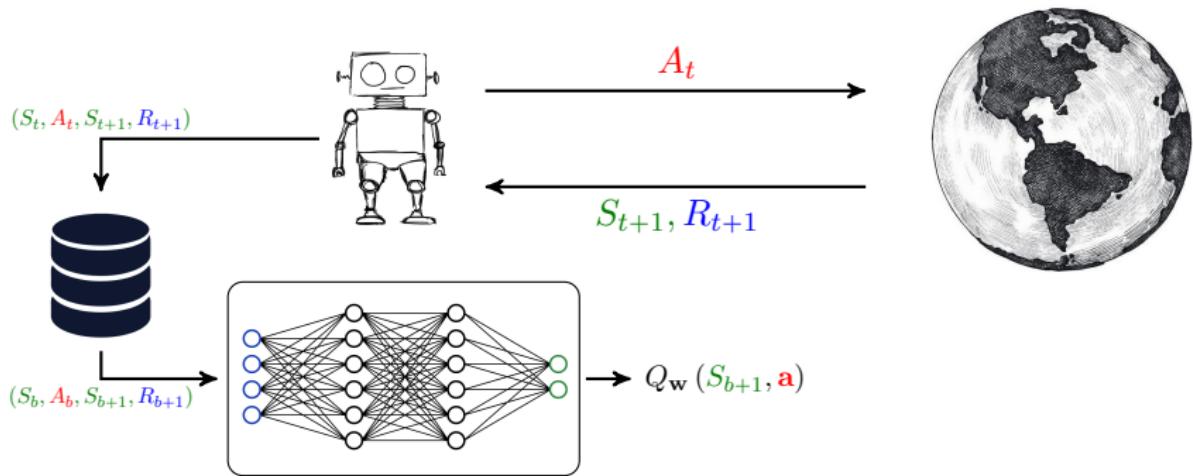
- *This can improve the convergence speed*
- *The trained DQN may however stick to a bad local minima*

In practice we typically set  $L = 1$

- *with a relatively large batch-size we can see good convergence results*

- 
- + *What about the second challenge? I didn't get really what was the issue at the first place!*
  - Let's break it down!

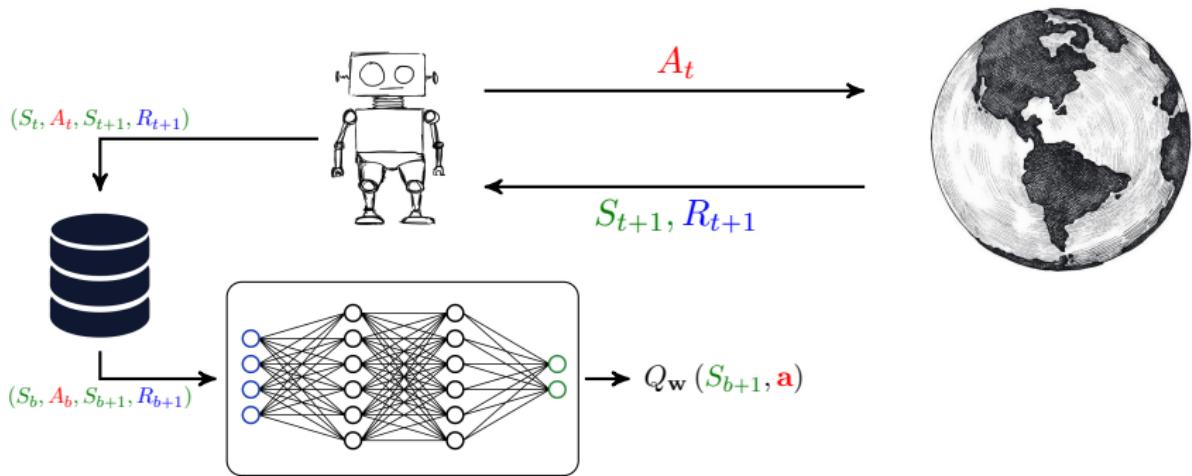
# Varying Labels



We can look at the training procedure as supervised learning with label

$$y_b = R_{b+1} + \gamma \max_m Q_w(S_{b+1}, a^m)$$

# Varying Labels

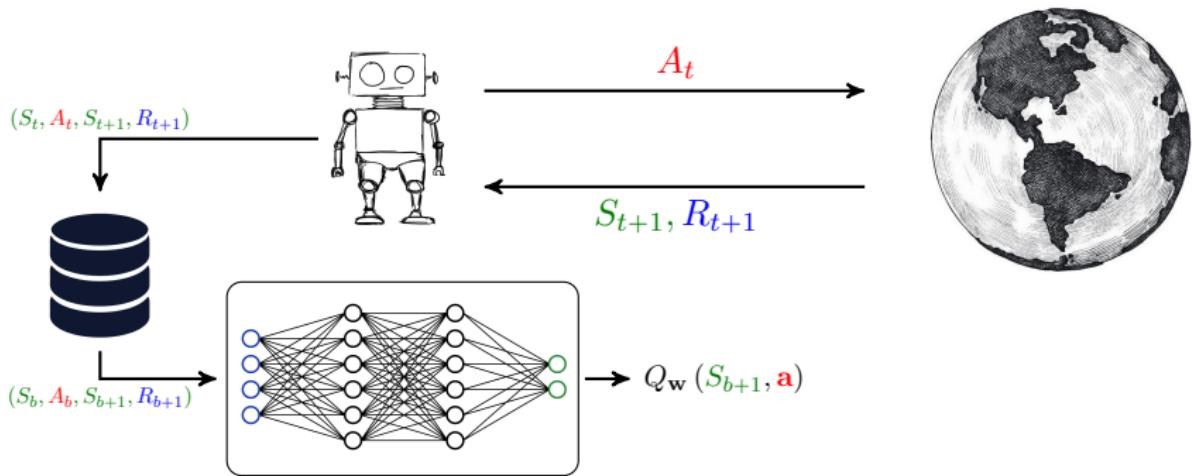


We are then updating  $w$  gradually, such that

$$\Delta_b = y_b - Q_w(S_b, A_b)$$

shrinks:

# Varying Labels

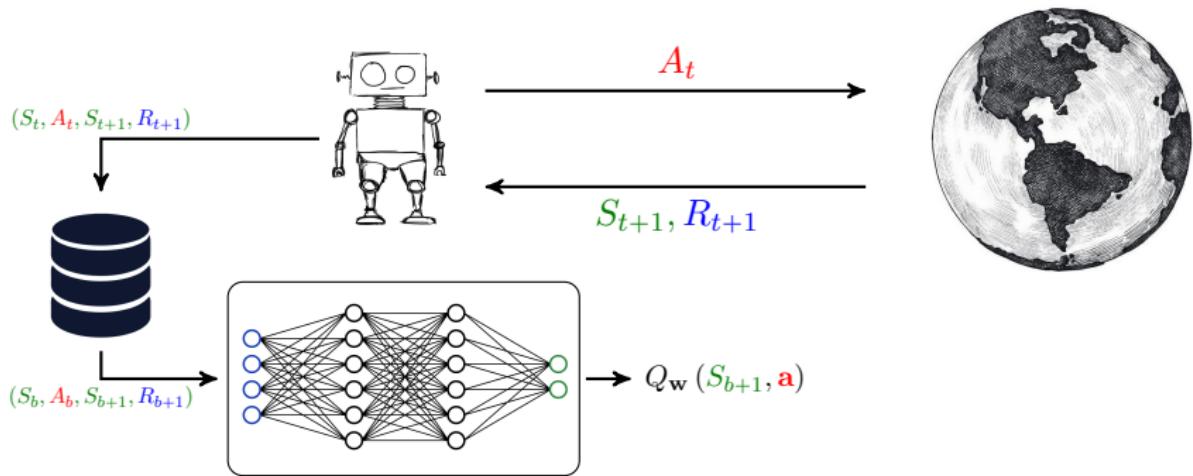


We are then updating  $w$  gradually, such that

$$\Delta_b = y_b - Q_w(S_b, A_b)$$

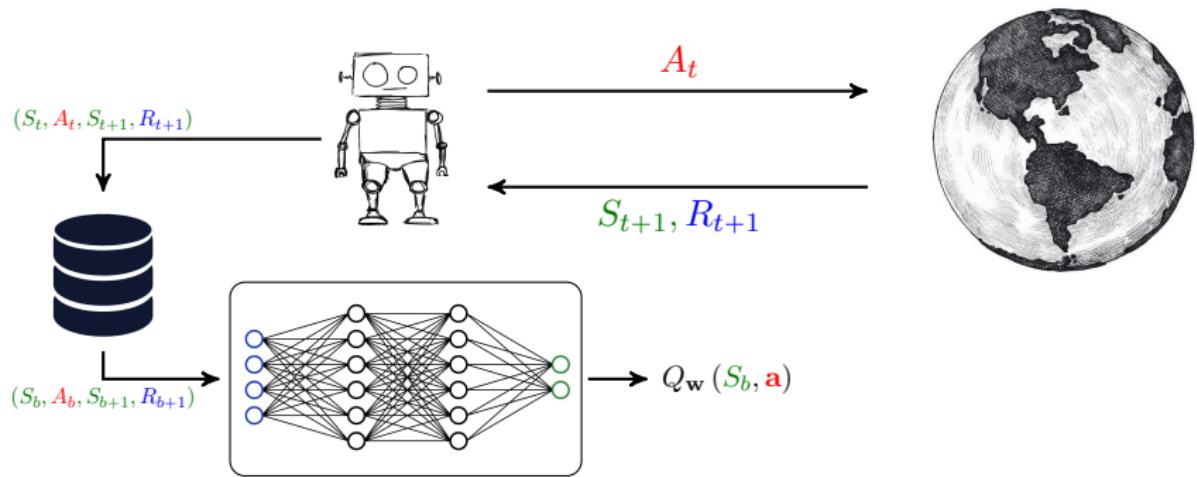
shrinks: but, each time we update  $w$ , the label  $y_b$  also changes!

# Varying Labels

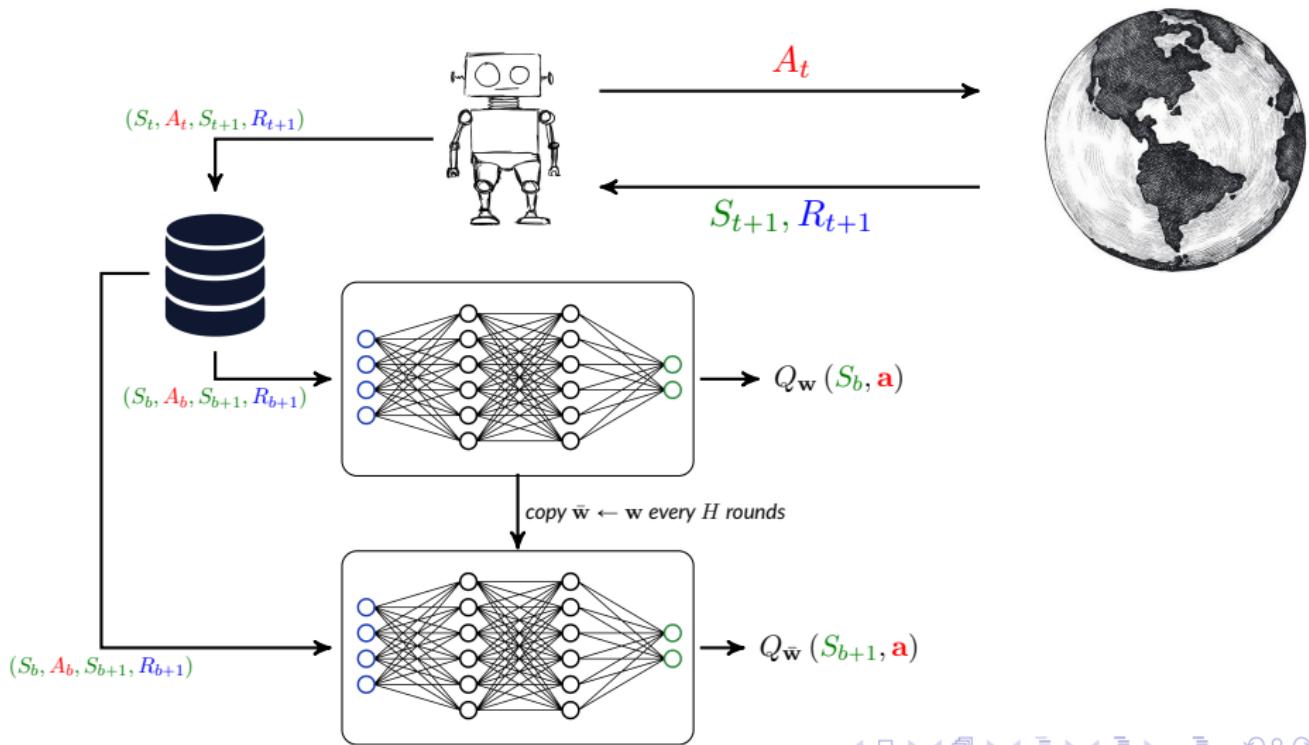


This is an issue: in standard SGD, we have **fixed labels** and therefore  
by **multiple iterations** the NN **gradually** converges to a good approximator

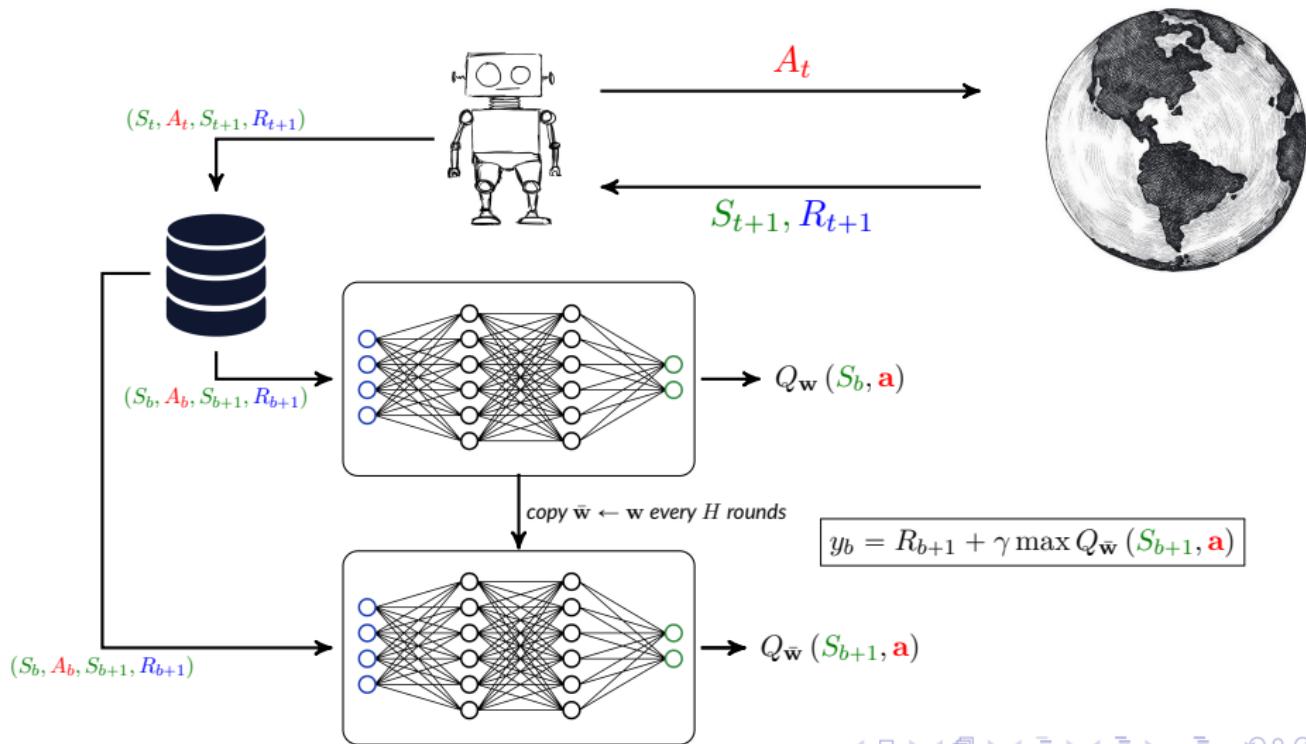
# Target Network: Simple Remedy



# Target Network: Simple Remedy



# Target Network: Simple Remedy



# DQL: Classic Algorithm

DQL():

- 1: Initiate with  $w$ , empty replay buffer  $\mathbb{D}$ , learning rate  $\alpha$ , and a random state  $S_t$
- 2: **while** interating **do**
- 3:   Update  $\bar{w} \leftarrow w$
- 4:   **for**  $h = 1 : H$  **do**
- 5:      $S_t \leftarrow S_{t+1}$  if  $S_t$  is a terminal state **then** replace  $S_t$  with a random state
- 6:     Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_w)$  and draw  $A_t$  from  $\pi(\cdot | S_t)$
- 7:     Add  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$  to the replay buffer  $\mathbb{D}$
- 8:     **for** iteration  $\ell = 1 : L$  **do**
- 9:       Sample mini-batch  $\mathbb{B} = \{S_b, A_b \xrightarrow{R_{b+1}} S_{b+1} \text{ for } b = 1 : B\}$  from  $\mathbb{D}$
- 10:        $\Delta_b \leftarrow R_{b+1} + \gamma \max_m Q_{\bar{w}}(S_{b+1}, a^m) - Q_w(S_b, A_b)$
- 11:       Update  $w \leftarrow w + \alpha \sum_{b=1}^B \Delta_b \nabla Q_w(S_b, A_b)$
- 12:     **end for**
- 13:   **end for**
- 14: **end while**

# DQL: Classic Algorithm

DQL():

- 1: Initiate with  $\mathbf{w}$ , empty replay buffer  $\mathbb{D}$ , learning rate  $\alpha$ , and a random state  $S_t$
- 2: **while** interating **do**
- 3:   Update  $\bar{\mathbf{w}} \leftarrow \mathbf{w}$
- 4:   **for**  $h = 1 : H$  **do**
- 5:      $S_t \leftarrow S_{t+1}$  if  $S_t$  is a terminal state **then** replace  $S_t$  with a random state
- 6:     Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_{\mathbf{w}})$  and draw  $A_t$  from  $\pi(\cdot | S_t)$
- 7:     Add  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$  to the replay buffer  $\mathbb{D}$
- 8:     **for** iteration  $\ell = 1 : L$  **do**
- 9:       Sample mini-batch  $\mathbb{B} = \{S_b, A_b \xrightarrow{R_{b+1}} S_{b+1} \text{ for } b = 1 : B\}$  from  $\mathbb{D}$
- 10:        $\Delta_b \leftarrow R_{b+1} + \gamma \max_m Q_{\bar{\mathbf{w}}}(S_{b+1}, a^m) - Q_{\mathbf{w}}(S_b, A_b)$
- 11:       Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_{b=1}^B \Delta_b \nabla Q_{\mathbf{w}}(S_b, A_b)$
- 12:     **end for**
- 13:   **end for**
- 14: **end while**

# DQL: Classic Algorithm

DQL():

- 1: Initiate with  $\mathbf{w}$ , empty replay buffer  $\mathbb{D}$ , learning rate  $\alpha$ , and a random state  $S_t$
- 2: **while** interating **do**
- 3:   Update  $\bar{\mathbf{w}} \leftarrow \mathbf{w}$
- 4:   **for**  $h = 1 : H$  **do**
- 5:      $S_t \leftarrow S_{t+1}$  if  $S_t$  is a terminal state **then** replace  $S_t$  with a random state
- 6:     Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_{\mathbf{w}})$  and draw  $A_t$  from  $\pi(\cdot | S_t)$
- 7:     Add  $(S_t, A_t \xrightarrow{R_{t+1}} S_{t+1})$  to the replay buffer  $\mathbb{D}$
- 8:     **for** iteration  $\ell = 1 : L$  **do**
- 9:       Sample mini-batch  $\mathbb{B} = \{S_b, A_b \xrightarrow{R_{b+1}} S_{b+1} \text{ for } b = 1 : B\}$  from  $\mathbb{D}$
- 10:        $\Delta_b \leftarrow R_{b+1} + \gamma \max_m Q_{\bar{\mathbf{w}}}(S_{b+1}, a^m) - Q_{\mathbf{w}}(S_b, A_b)$
- 11:       Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \sum_{b=1}^B \Delta_b \nabla Q_{\mathbf{w}}(S_b, A_b)$
- 12:     **end for**
- 13:   **end for**
- 14: **end while**

# A Revolution: Google DeepMind

## LETTER

doi:10.1038/nature14236

### Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>1\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Fidjeland<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadik<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharshan Kumaran<sup>1</sup>, Daan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

# A Revolution: Google DeepMind

## LETTER

doi:10.1038/nature14236

### Human-level control through deep reinforcement learning

Volodymyr Mnih<sup>1\*</sup>, Koray Kavukcuoglu<sup>1\*</sup>, David Silver<sup>1\*</sup>, Andrei A. Rusu<sup>1</sup>, Joel Veness<sup>1</sup>, Marc G. Bellemare<sup>1</sup>, Alex Graves<sup>1</sup>, Martin Riedmiller<sup>1</sup>, Andreas K. Fidjeland<sup>1</sup>, Georg Ostrovski<sup>1</sup>, Stig Petersen<sup>1</sup>, Charles Beattie<sup>1</sup>, Amir Sadik<sup>1</sup>, Ioannis Antonoglou<sup>1</sup>, Helen King<sup>1</sup>, Dharshan Kumaran<sup>1</sup>, Daan Wierstra<sup>1</sup>, Shane Legg<sup>1</sup> & Demis Hassabis<sup>1</sup>

*Just to have a clue about how revolutionary it had been*

#### Human-level control through deep reinforcement learning

V Mnih, K Kavukcuoglu, D Silver, AA Rusu, J Veness, MG Bellemare, A Graves, M Riedmiller...

nature, 2015 • nature.com

#### Abstract

The theory of reinforcement learning provides a normative account, deeply rooted in psychological and neuroscientific perspectives on animal behaviour, of how agents may optimize their control of an environment. To use reinforcement learning successfully in situations approaching real-world complexity, however, agents are confronted with a difficult task: they must derive efficient representations of the environment from high-dimensional sensory inputs, and use these to generalize past experience to new

SHOW MORE ▾



☆ Save

✉ Cite

Cited by 30437

Related articles

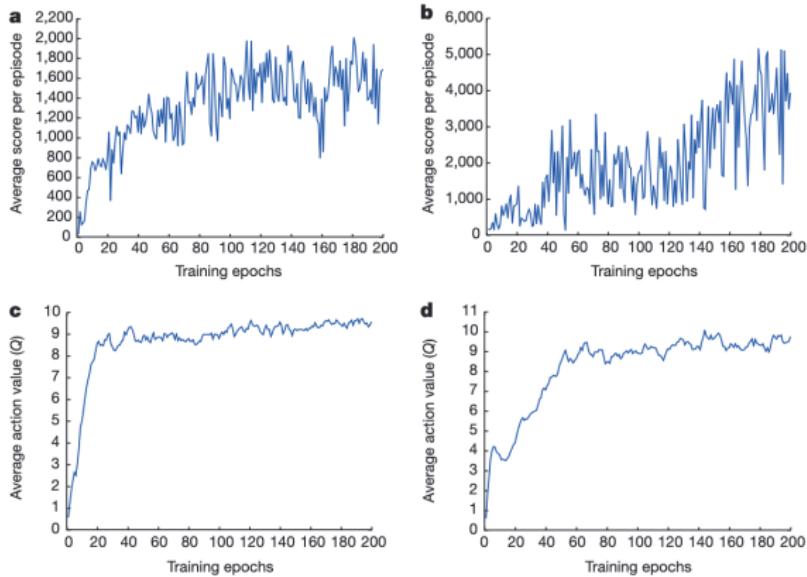
All 57 versions

Web of Science: 14719



# A Revolution: Google DeepMind

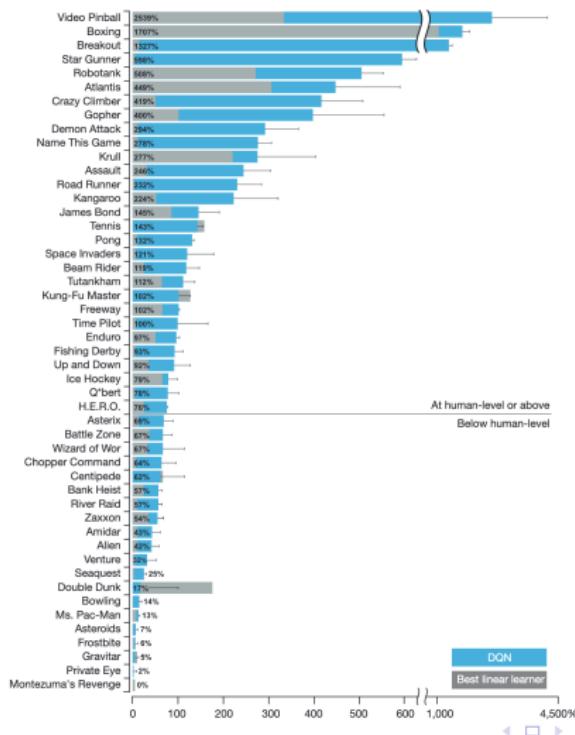
DQL could show *extraordinary performance*



You may also watch the demo of the *Breakout game* on Youtube

# A Revolution: Google DeepMind

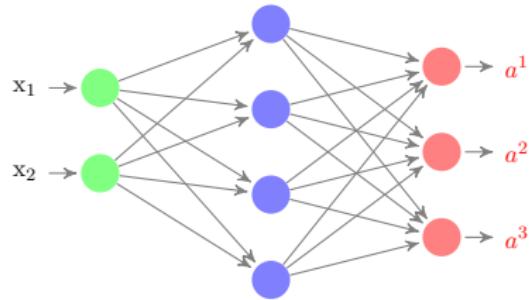
DQL was the first *universal* algorithm that could be applied to *any* environment



## Example: Mountain Car



Let's try to imagine DQL in the mountain car example with a simple DQN



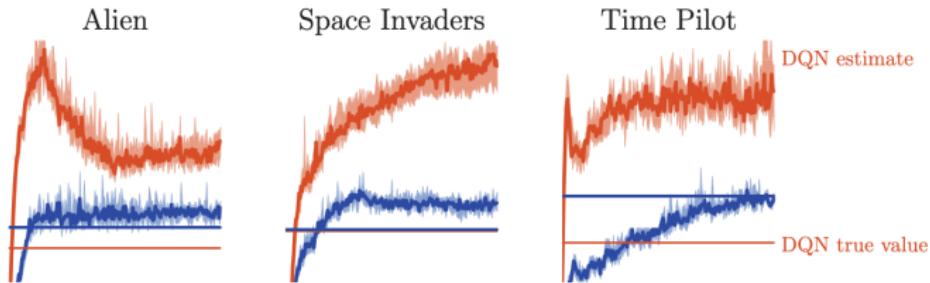
A more concrete example will be solved in the next Tutorial

## DQL: Bias Problem

Q-learning is known to estimate action-values with **bias**:

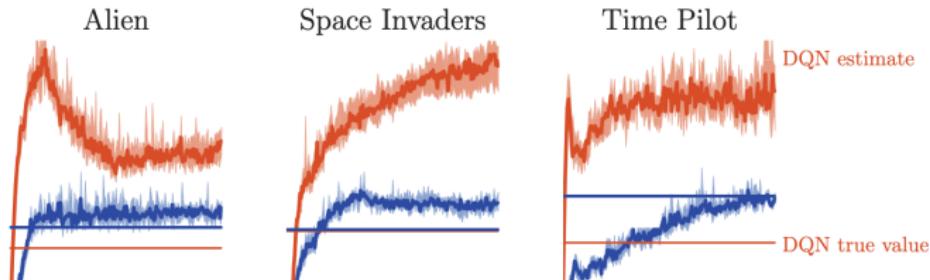
## DQL: Bias Problem

Q-learning is known to estimate action-values with bias: take a look at few examples of DQL algorithm playing Atari games



## DQL: Bias Problem

Q-learning is known to estimate action-values with bias: take a look at few examples of DQL algorithm playing Atari games



- + Why is it happening? Is it simply because of TD approach?
- It's severer than only TD: it comes from the max operator in the update!
- + How does it come?
- Well, It's best understood through an example

## Bias Problem: Basic Example

Let's consider a simple example: *assume we are dealing with two computers, namely A and B. These computers are used to run the same program*

- Computer A runs the program in exactly  $X_A = 8$  seconds
- Computer B runs the program in exactly  $X_B = 5$  seconds

## Bias Problem: Basic Example

Let's consider a simple example: *assume we are dealing with two computers, namely A and B. These computers are used to run the same program*

- Computer A runs the program in exactly  $X_A = 8$  seconds
- Computer B runs the program in exactly  $X_B = 5$  seconds

We however do not know these values: we can only run sample runs

- Our time measurement is **noisy**, i.e., we compute on Computer  $i$

$$\hat{X}_i = X_i + \varepsilon$$

↳ This error is **random** and can **increase or decrease**  $X_i$

## Bias Problem: Basic Example

Let's consider a simple example: assume we are dealing with two computers, namely **A** and **B**. These computers are used to run the **same program**

- Computer A runs the program in exactly  $X_A = 8$  seconds
- Computer B runs the program in exactly  $X_B = 5$  seconds

We however do not know these **values**: we can only run **sample runs**

- Our time measurement is **noisy**, i.e., we compute on Computer  $i$

$$\hat{X}_i = X_i + \varepsilon$$

↳ This error is **random** and can **increase or decrease**  $X_i$

## Ultimate Goal

We want to compute the maximum runtime between the two computers

# Bias Problem: Basic Example

We have access to noisy samples

$$\hat{X}_A^{(1)}, \dots, \hat{X}_A^{(K)}, \hat{X}_B^{(1)}, \dots, \hat{X}_B^{(K)}$$

# Bias Problem: Basic Example

We have access to noisy samples

$$\hat{X}_A^{(1)}, \dots, \hat{X}_A^{(K)}, \hat{X}_B^{(1)}, \dots, \hat{X}_B^{(K)}$$

If  $K$  is only moderately large, we could almost surely say that

$$\max \left\{ \hat{X}_A^{(1)}, \dots, \hat{X}_A^{(K)}, \hat{X}_B^{(1)}, \dots, \hat{X}_B^{(K)} \right\} > \max \{ X_A, X_B \} = 8$$

- Within enough number of samples there is for sure a positive error sample
- This error sample renders an over-estimation

# Bias Problem: Basic Example

We have access to noisy samples

$$\hat{X}_A^{(1)}, \dots, \hat{X}_A^{(K)}, \hat{X}_B^{(1)}, \dots, \hat{X}_B^{(K)}$$

If  $K$  is only moderately large, we could almost surely say that

$$\max \left\{ \hat{X}_A^{(1)}, \dots, \hat{X}_A^{(K)}, \hat{X}_B^{(1)}, \dots, \hat{X}_B^{(K)} \right\} > \max \{ X_A, X_B \} = 8$$

- Within enough number of samples there is for sure a positive error sample
- This error sample renders an **over-estimation**

A crucial point is that if we repeat this experiment several times, we always get an **over-estimate**; therefore,

after averaging over multiple instances, we are still **biased!**

## Solution to Max-Bias: Double Measurements

There is a very simple and intuitive solution to this problem: we can collect two sequences of samples

Sequence 1:  $\hat{X}_{A,1}^{(1)}, \dots, \hat{X}_{A,1}^{(K)}, \hat{X}_{B,1}^{(1)}, \dots, \hat{X}_{B,1}^{(K)}$

Sequence 2:  $\hat{X}_{A,2}^{(1)}, \dots, \hat{X}_{A,2}^{(K)}, \hat{X}_{B,2}^{(1)}, \dots, \hat{X}_{B,2}^{(K)}$

## Solution to Max-Bias: Double Measurements

There is a very simple and intuitive solution to this problem: we can collect two sequences of samples

Sequence 1:  $\hat{X}_{A,1}^{(1)}, \dots, \hat{X}_{A,1}^{(K)}, \hat{X}_{B,1}^{(1)}, \dots, \hat{X}_{B,1}^{(K)}$

Sequence 2:  $\hat{X}_{A,2}^{(1)}, \dots, \hat{X}_{A,2}^{(K)}, \hat{X}_{B,2}^{(1)}, \dots, \hat{X}_{B,2}^{(K)}$

We find the index of the maximizer in Sequence 1, i.e.,

$$(i, k) = \operatorname{argmax} \left\{ \hat{X}_{A,1}^{(1)}, \dots, \hat{X}_{A,1}^{(K)}, \hat{X}_{B,1}^{(1)}, \dots, \hat{X}_{B,1}^{(K)} \right\}$$

## Solution to Max-Bias: Double Measurements

There is a very simple and intuitive solution to this problem: we can collect two sequences of samples

Sequence 1:  $\hat{X}_{A,1}^{(1)}, \dots, \hat{X}_{A,1}^{(K)}, \hat{X}_{B,1}^{(1)}, \dots, \hat{X}_{B,1}^{(K)}$

Sequence 2:  $\hat{X}_{A,2}^{(1)}, \dots, \hat{X}_{A,2}^{(K)}, \hat{X}_{B,2}^{(1)}, \dots, \hat{X}_{B,2}^{(K)}$

We find the index of the maximizer in Sequence 1, i.e.,

$$(i, k) = \operatorname{argmax} \left\{ \hat{X}_{A,1}^{(1)}, \dots, \hat{X}_{A,1}^{(K)}, \hat{X}_{B,1}^{(1)}, \dots, \hat{X}_{B,1}^{(K)} \right\}$$

But we take the sample from Sequence 2, i.e.,

$$\hat{X}_{\max} = \hat{X}_{i,2}^{(k)}$$

## Solution to Max-Bias: Double Measurements

There is a very simple and intuitive solution to this problem: we can collect two sequences of samples

Sequence 1:  $\hat{X}_{A,1}^{(1)}, \dots, \hat{X}_{A,1}^{(K)}, \hat{X}_{B,1}^{(1)}, \dots, \hat{X}_{B,1}^{(K)}$

Sequence 2:  $\hat{X}_{A,2}^{(1)}, \dots, \hat{X}_{A,2}^{(K)}, \hat{X}_{B,2}^{(1)}, \dots, \hat{X}_{B,2}^{(K)}$

We find the index of the maximizer in Sequence 1, i.e.,

$$(i, k) = \operatorname{argmax} \left\{ \hat{X}_{A,1}^{(1)}, \dots, \hat{X}_{A,1}^{(K)}, \hat{X}_{B,1}^{(1)}, \dots, \hat{X}_{B,1}^{(K)} \right\}$$

But we take the sample from Sequence 2, i.e.,

$$\hat{X}_{\max} = \hat{X}_{i,2}^{(k)}$$

This is an **unbiased estimator**: if we repeat this experiment several times

after averaging over multiple instances, we get close to  $X_A$

# DQL with Double Measurements

We can apply this idea to DQL:

## DQL with Double Measurements

We can apply this idea to DQL: we train *two DQNs simultaneously*

- We *find out* the action with maximum value using *one DQN*
- We *evaluate* the action-value of this action by the *other DQN*

This way we get an *unbiased* estimator of the optimal action-value

we refer to this approach as *double DQL*

## DQL with Double Measurements

We can apply this idea to DQL: we train **two DQNs simultaneously**

- We **find out** the action with maximum value using **one DQN**
- We **evaluate** the action-value of this action by the **other DQN**

This way we get an **unbiased** estimator of the optimal action-value

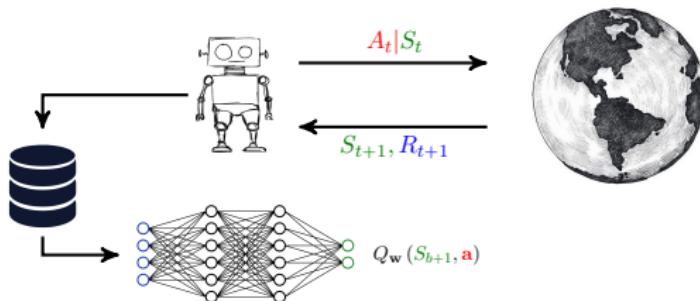
we refer to this approach as **double** DQL

### Attention

In general this does not mean that we are necessarily reaching to a better policy: we could have biased estimator and still play **optimally!**

# Double DQL

Selecting DQN



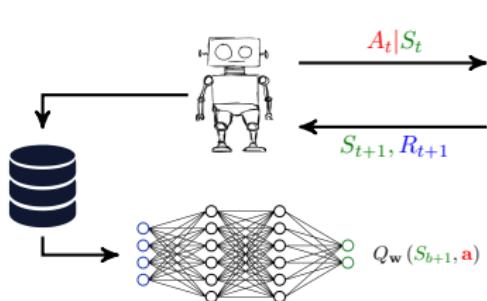
We train two DQNs on two *independent experience sets*

- With the *online* DQL we find

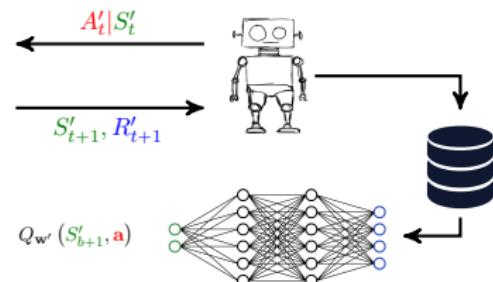
$$A_{t+1}^* = \operatorname{argmax} Q_w(S_{b+1}, a)$$

# Double DQL

Selecting DQN



Evaluating DQN

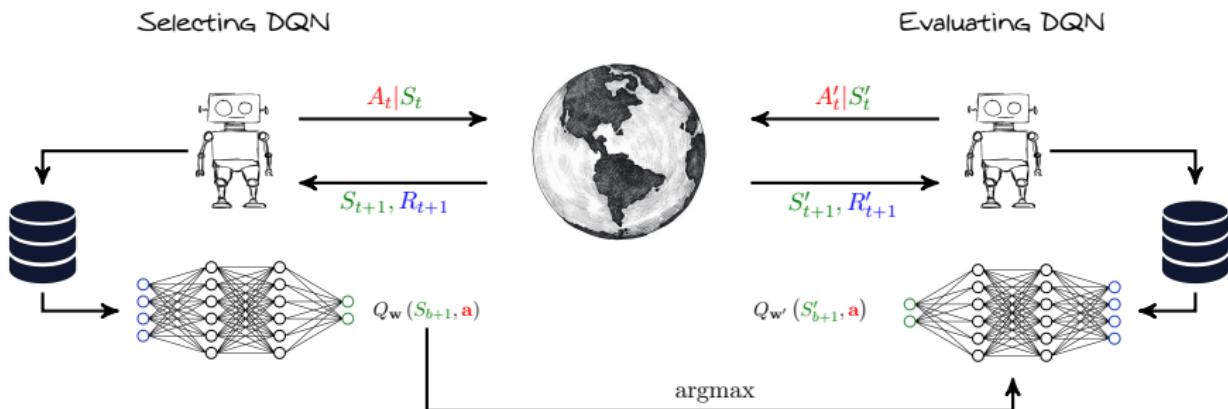


We train two DQNs on two *independent experience sets*

- With the *online* DQL we find

$$A_{t+1}^* = \operatorname{argmax} Q_w(S_{b+1}, a)$$

# Double DQL



We train two DQNs on two *independent experience sets*

- With the *online* DQL we find

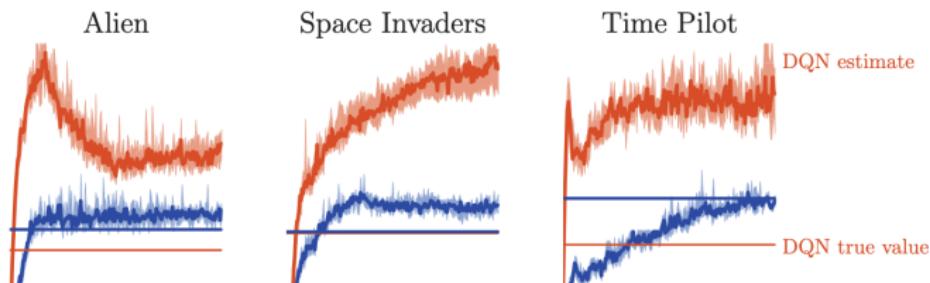
$$A_{t+1}^* = \operatorname{argmax} Q_w(S_{b+1}, a)$$

- With the *double* DQL we evaluate

$$y_b = R_{b+1} + \gamma Q_{w'}(S_{b+1}, A_{t+1}^*)$$

## Double DQL: Sample Results

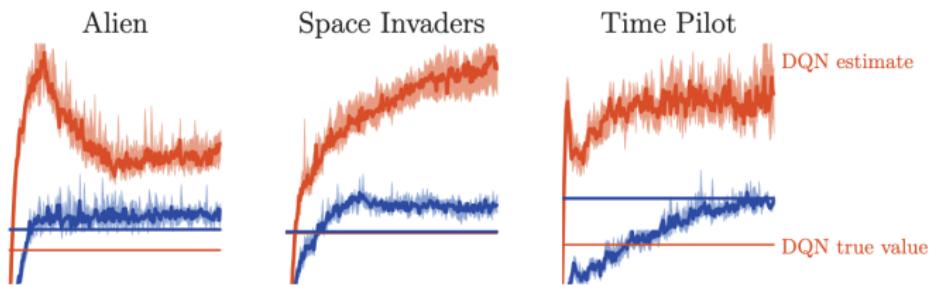
Let's take a look back to the earlier examples



Here, **blue curves** show double DQL

## Double DQL: Sample Results

Let's take a look back to the earlier examples

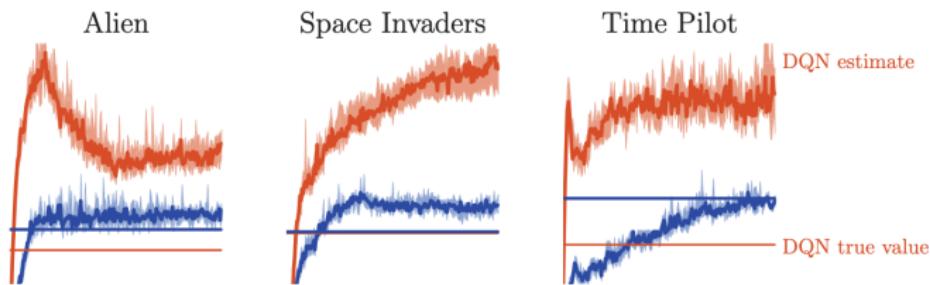


Here, **blue curves** show double DQL

- In all examples we are getting **less bias** as compared to DQL

## Double DQL: Sample Results

Let's take a look back to the earlier examples

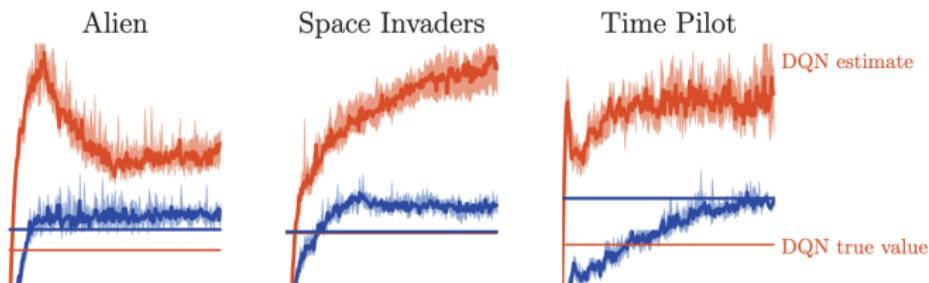


Here, **blue curves** show double DQL

- In all examples we are getting **less bias** as compared to DQL
- In two example it helps converging to **better policy**
  - ↳ Alien and Time Pilot

## Double DQL: Sample Results

Let's take a look back to the earlier examples



Here, **blue curves** show double DQL

- In all examples we are getting **less bias** as compared to DQL
- In two example it helps converging to **better policy**
  - ↳ Alien and Time Pilot
- In one example, it reduces bias but **does not impact** converging policy

## Other Variants

There are various extensions to classic DQL: *some famous ones are*

- *Double DQL*
  - ↳ *It tries to reduce the bias of value estimation*
- *Dueling DQL*
  - ↳ *It uses notion of advantage  $\equiv$  difference between action-value and value*
  - ↳ *It helps finding non-valuable actions*

# Other Variants

There are various extensions to classic DQL: *some famous ones are*

- *Double DQL*
  - ↳ *It tries to reduce the bias of value estimation*
- *Dueling DQL*
  - ↳ *It uses notion of advantage  $\equiv$  difference between action-value and value*
  - ↳ *It helps finding non-valuable actions*
- *Prioritized DQL*
  - ↳ *It gives priority to samples in the experience buffer*

# Other Variants

There are various extensions to classic DQL: *some famous ones are*

- *Double DQL*
  - ↳ *It tries to reduce the bias of value estimation*
- *Dueling DQL*
  - ↳ *It uses notion of advantage  $\equiv$  difference between action-value and value*
  - ↳ *It helps finding non-valuable actions*
- *Prioritized DQL*
  - ↳ *It gives priority to samples in the experience buffer*
- *Distributed DQL*
  - ↳ *It enables training DQN through pipelining*
  - ↳ *This let training of DQNs for massive problems*

# Distributed DQL: Gorila

*General Reinforcement Learning Architecture ≡ Gorila*

Gorila is implemented through four main generalization

- ① Parallel *actors* generating acting behavior
- ② Parallel DQNs trained by stored experience

# Distributed DQL: Gorila

*General Reinforcement Learning Architecture ≡ Gorila*

Gorila is implemented through four main generalization

- ① Parallel *actors* generating acting behavior
- ② Parallel DQNs trained by stored experience
- ③ Distributed storage of experience
- ④ A distributed DQN that specifies acting behavior policy

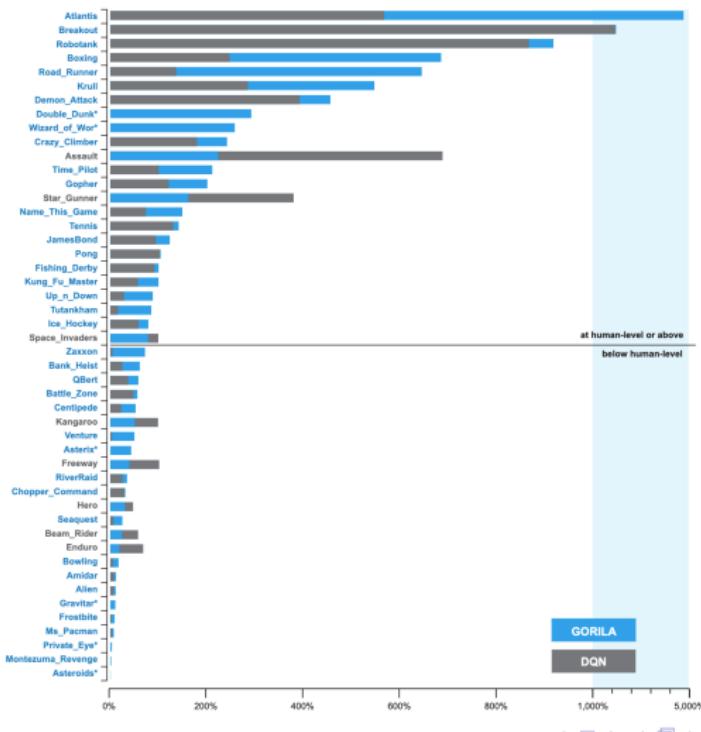
---

Gorila **massively** parallelize implementation of DQL

this enables implementation of DQL for *realistic* hard control loops

# Distributed DQL: Gorila

With significantly lower training time, Gorila starts to beat classic DQL



# Dealing with Continuous Actions

Once DQL was established a new **question** raised

**? How can we handle settings with *continuous action space*?**

*This is a very **practical** setting that show up in robotics, autonomous driving, etc*

# Dealing with Continuous Actions

Once DQL was established a new **question** raised

- ? How can we handle settings with **continuous action space**?

This is a very **practical** setting that show up in robotics, autonomous driving, etc

- + What about it? Why should be a **challenge** to use DQL with **continuous action space**?
- Well! How could we **maximize** action-value in this case?!

# Dealing with Continuous Actions

Once DQL was established a new **question** raised

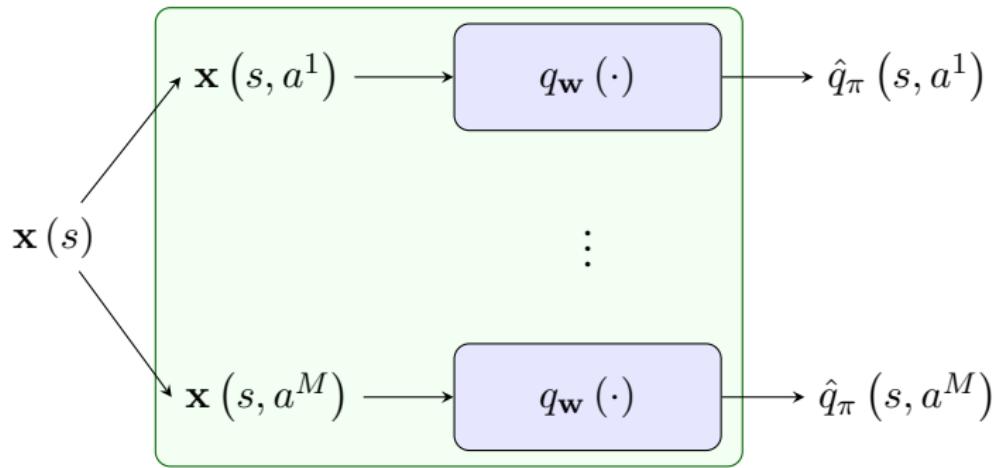
- ? How can we handle settings with **continuous action space**?

This is a very **practical** setting that show up in robotics, autonomous driving, etc

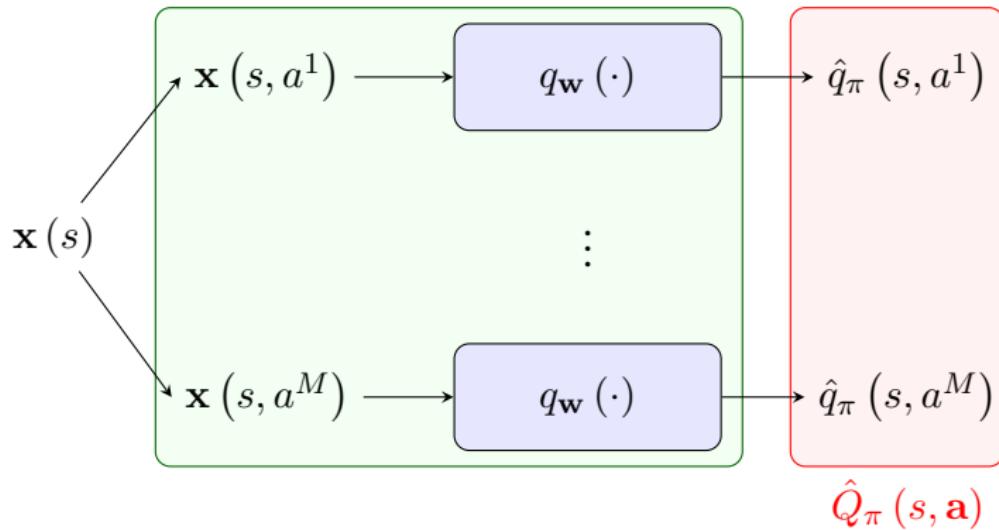
- + What about it? Why should be a **challenge** to use DQL with **continuous action space**?
- Well! How could we **maximize** action-value in this case?!

Let's take a look to see the challenge clearly

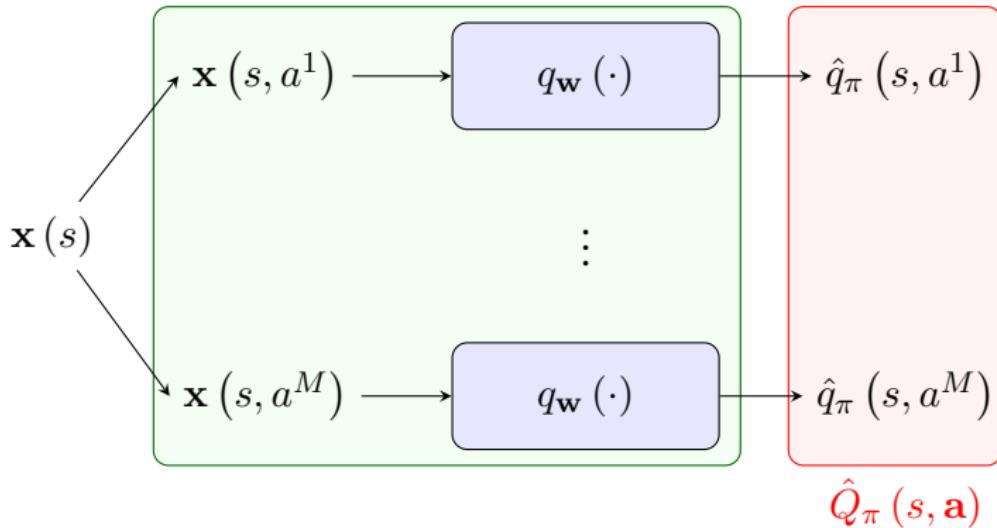
# DQN: Recalling the Output



# DQN: Recalling the Output



## DQN: Recalling the Output

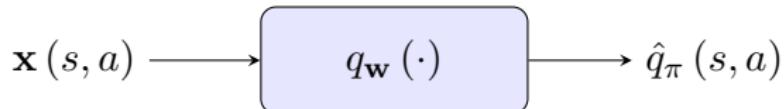


With **continuous actions**, we cannot **enumerate** the output!

- + Why don't we use action-value approximator of form I?!
- Well! Let's do this

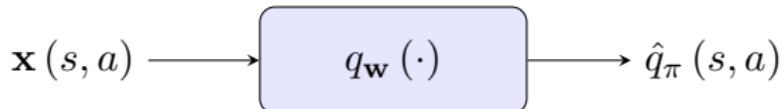
# Recall: Action-Value Approximator – Form I

*This is what we called Form I*



# Recall: Action-Value Approximator – Form I

This is what we called Form I

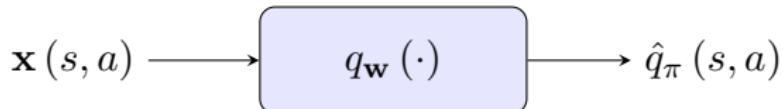


Let's now see how the DQL algorithm can be applied

↳ Let's consider the *vanilla* DQL

## Recall: Action-Value Approximator – Form I

This is what we called Form I



Let's now see how the DQL algorithm can be applied

↳ Let's consider the *vanilla* DQL

- 1: Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_{\mathbf{w}}(S_t, \mathbf{a}))$
- 2: Draw action  $A_t$  from  $\pi(\cdot | S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 3:  $\Delta \leftarrow R_{t+1} + \gamma \max_m Q_{\mathbf{w}}(S_{t+1}, \mathbf{a}^m) - Q_{\mathbf{w}}(S_t, A_t)$
- 4: Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \nabla Q_{\mathbf{w}}(S_t, A_t)$

We obviously can replace  $Q_{\mathbf{w}}(\cdot)$  with  $q_{\mathbf{w}}(\cdot)$

# DQL with Action-Value Approximator – Form I

- 1: Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_{\mathbf{w}}(S_t, \mathbf{a}))$
- 2: Draw action  $A_t$  from  $\pi(\cdot|S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 3:  $\Delta \leftarrow R_{t+1} + \gamma \max_m Q_{\mathbf{w}}(S_{t+1}, \mathbf{a}^m) - Q_{\mathbf{w}}(S_t, A_t)$
- 4: Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \nabla Q_{\mathbf{w}}(S_t, A_t)$

We can replace these updates with

- 1: Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(q_{\mathbf{w}}(S_t, \mathbf{a}))$
- 2: Draw action  $A_t$  from  $\pi(\cdot|S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 3:  $\Delta \leftarrow R_{t+1} + \gamma \max_a q_{\mathbf{w}}(S_{t+1}, \mathbf{a}) - q_{\mathbf{w}}(S_t, A_t)$
- 4: Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \nabla q_{\mathbf{w}}(S_t, A_t)$

Well! We need to optimize over a **continuous** variable!

# DQL with Action-Value Approximator – Form I

- 1: Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(Q_{\mathbf{w}}(S_t, \mathbf{a}))$
- 2: Draw action  $A_t$  from  $\pi(\cdot | S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 3:  $\Delta \leftarrow R_{t+1} + \gamma \max_m Q_{\mathbf{w}}(S_{t+1}, \mathbf{a}^m) - Q_{\mathbf{w}}(S_t, A_t)$
- 4: Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \nabla Q_{\mathbf{w}}(S_t, A_t)$

We can replace these updates with

- 1: Update policy to  $\pi \leftarrow \epsilon\text{-Greedy}(q_{\mathbf{w}}(S_t, \mathbf{a}))$
- 2: Draw action  $A_t$  from  $\pi(\cdot | S_t)$  and observe  $S_t, A_t \xrightarrow{R_{t+1}} S_{t+1}$
- 3:  $\Delta \leftarrow R_{t+1} + \gamma \max_a q_{\mathbf{w}}(S_{t+1}, \mathbf{a}) - q_{\mathbf{w}}(S_t, A_t)$
- 4: Update  $\mathbf{w} \leftarrow \mathbf{w} + \alpha \Delta \nabla q_{\mathbf{w}}(S_t, A_t)$

Well! We need to optimize over a **continuous** variable!

- + Where exactly we need it?
- In lines 1 and 3: once for  $\epsilon$ -greedy update and once for off-policy control

## DQL with Action-Value Approximator – Form I

This is an essential challenge: we need to **optimize** over a continuous variable

- We may grid the action space
  - ↳ It's **computationally expensive**: we are back at square one!
- We may apply gradient descent
  - ↳ It makes a two-tier loop: it's again **computationally expensive**!

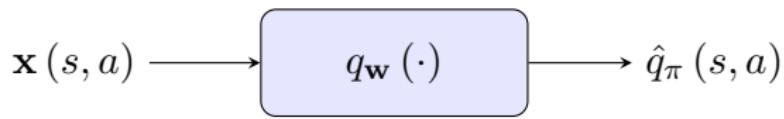
## DQL with Action-Value Approximator – Form I

This is an essential challenge: we need to **optimize** over a continuous variable

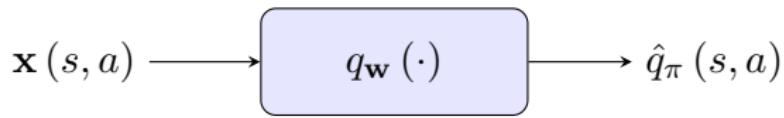
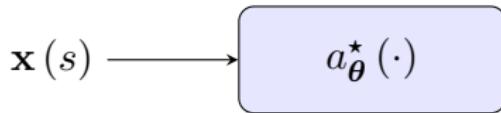
- We may grid the action space
    - ↳ It's **computationally expensive**: we are back at square one!
  - We may apply gradient descent
    - ↳ It makes a two-tier loop: it's again **computationally expensive**!
- 
- + It sounds like **impossible**!
  - Only impossible is impossible

In practice, we solve the target optimization via a **DNN**!

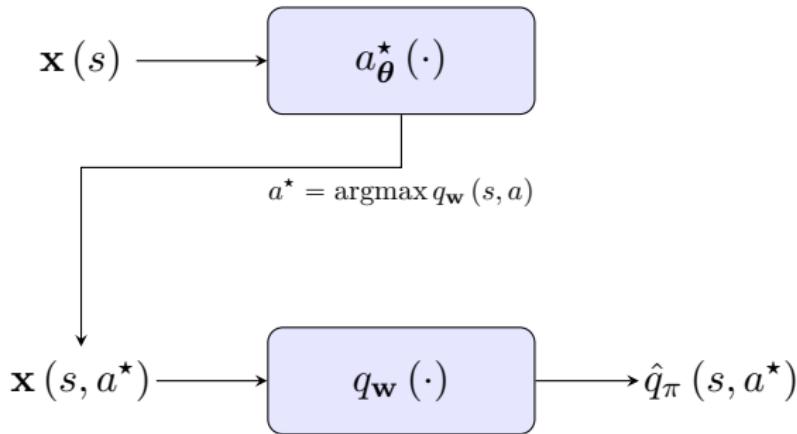
# Learning Optimal Action



# Learning Optimal Action



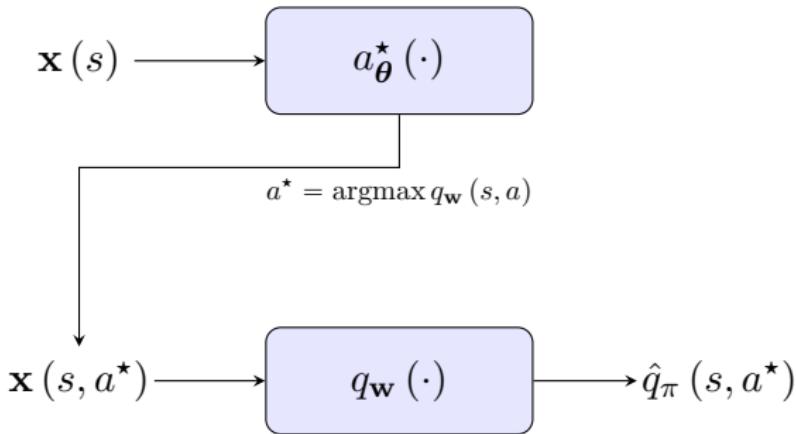
# Learning Optimal Action



We could then update in DQL algorithm as

$$\Delta \leftarrow R_{t+1} + \gamma q_w(S_{t+1}, a^\star) - q_w(S_t, A_t)$$

# Learning Optimal Action



We could then update in DQL algorithm as

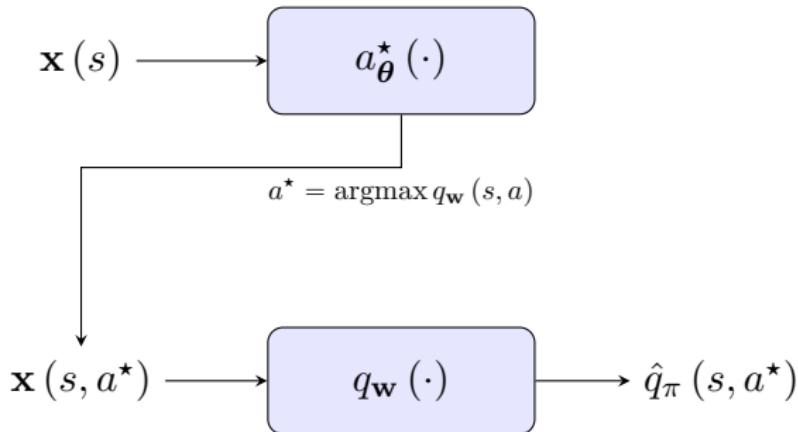
$$\Delta \leftarrow R_{t+1} + \gamma q_w(S_{t+1}, a^*) - q_w(S_t, A_t)$$

and for  $\epsilon$ -greedy improvement, we could

act  $a^*$  with probability  $1 - \epsilon$  and random with probability  $\epsilon$

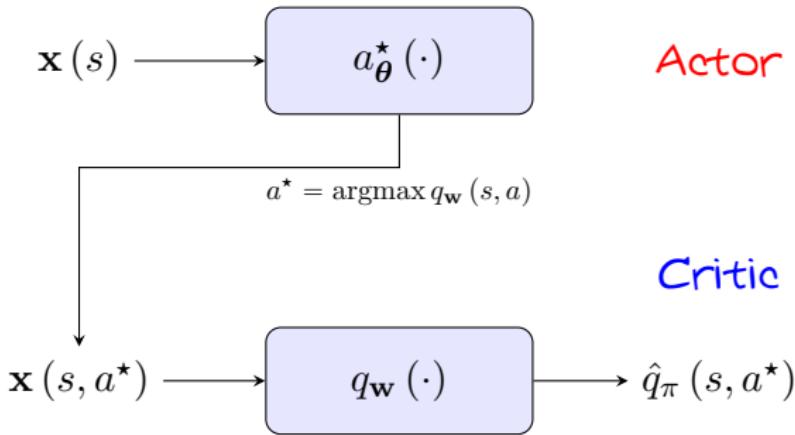
# Learning Optimal Action

- + Say we trained the network after some time; then, what do we do?
- We act  $a^*$  for each state  $s$



# Learning Optimal Action

- + Say we trained the network after some time; then, what do we do?
- We act  $a^*$  for each state  $s$

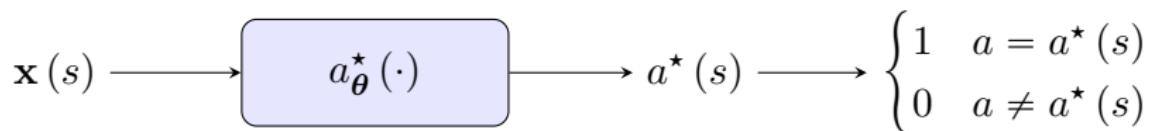


This is a particular example of **actor-critic** algorithm!

↳ We will discuss these algorithms

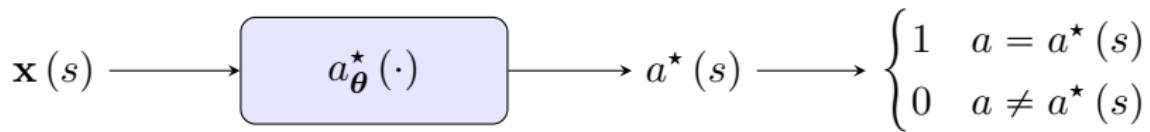
# Solution to Continuous Action: Policy Networks

Our *actor* learns an optimal greedy policy

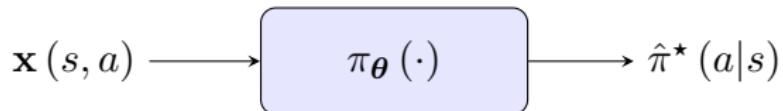


# Solution to Continuous Action: Policy Networks

Our **actor** learns an optimal greedy policy



This is a simplified form of the so-called **policy network**

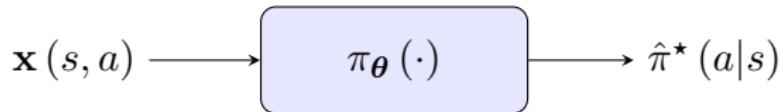


# Solution to Continuous Action: Policy Networks

Our **actor** learns an optimal greedy policy



This is a simplified form of the so-called **policy network**



## Policy Network

Policy network is an approximation model that maps state-action features to the optimal policy

# Solution to Continuous Action: Policy Networks

- + *What is the point in doing DQL anymore when we have a*

## Solution to Continuous Action: Policy Networks

- + What is the point in doing DQL anymore when we have a ? We already learn the optimal policy that we are looking for!
- Great! This is what we do in policy gradient algorithms

## Solution to Continuous Action: Policy Networks

- + What is the point in doing **DQL** anymore when we have  $a$ ? We already learn the **optimal policy** that we are looking for!
- Great! This is what we do in **policy gradient algorithms**

**Policy networks** are used in two sets of deep RL approaches

- **Policy gradient** approaches
  - ↳ We do **not** use a value network and directly approximate optimal policy
  - ↳ This is what we study next
- **Actor-critic** approaches
  - ↳ We **keep the value network** to examine the approximated optimal policy
  - ↳ This is the **most practically-robust** approach we can use