a glance at reinforcement learning

DATA SCIENCE RETREAT®

SINCE 2014

Adam Green

adgefficiency.com

adam.green@adgefficiency.com

Course notes

https://github.com/ADGEfficiency/DSR_RL

Agenda

Today
one background & terminology
two introduction to reinforcement learning
three value functions

DQN practical

Tomorrow
four policy gradients
five AlphaGo
six practical concerns
seven beyond the expectation, auxiliary loss functions, inverse RL

education

B.Eng. – Chemical Engineering
MSc - Advanced Process Design for Energy

energy engineer

2011 - 2016

GDF SUEZ

ENGIE

energy data scientist

2017 - present

TEMPUS

Today

Learning

Getting excited about reinforcement learning

Seeing state of the art concepts are not too far out of each

Knowing where to look next

These notes are designed as a future reference

First time doing this as a 2 day course – we can take it slow -> ask questions!

Today

Quizzes

answers are on the next slide

if no one knows the answer – read ahead (but still answer!)

Practical

I've built an agent based on DQN using TensorFlow

as a group we will run experiments with it

+ some general Python & TensorFlow stuff

Also happy to chat through project ideas

one

nomenclature & definitions

background & terminology

Where to start

Sutton & Barto – Reinforcement Learning: An Introduction

the bible of reinforcement learning

2nd edition in progress

RL Course by David Silver

10 lecture series by lead DeepMind programmer

Li (2017) Deep Reinforcement Learning: An Overview

These are all in the course repo
https://github.com/ADGEfficiency/DSR_RL

Nomenclature

s	state
s'	next state
a	action
r	reward
G_t	discounted return
γ	discount factor

$a \sim \pi(s)$	stochastic policy
$a = \pi(s)$	deterministic policy
π^*	optimal policy
$V_\pi(s)$	value function
$Q_\pi(s,a)$	action-value function
θ, w	function parameters
$\mathbb{E}[f(x)]$	expectation of $f(x)$

(s, a, r, s')	experience
(S, A, P, R, γ, H)	Markov Decision Process
$x \epsilon [-1, +1)$	interval including -1, excluding $+1$

Definitions

Expectation

weighted average of all possible values (i.e. the mean)

$$\mathbb{E}[f(x)] = \sum p(x) \cdot f(x)$$

Conditionals

probability of one thing given another

$P(s', r \mid s, a)$	probability of next state & reward, (given state, action)
$R(r \mid s, a)$	reward received (given state, action)
$a \sim \pi(s \rfloor a)$	action a sampled from policy π in state s

Variance & bias in supervised learning

Model generalization error = bias + variance + noise

Variance

error from sensitivity to noise in data set

model sees patterns that aren't there -> overfitting

Bias

error from assumptions in the learning algorithm

model can miss relevant patterns -> underfitting

Variance & bias in reinforcement learning

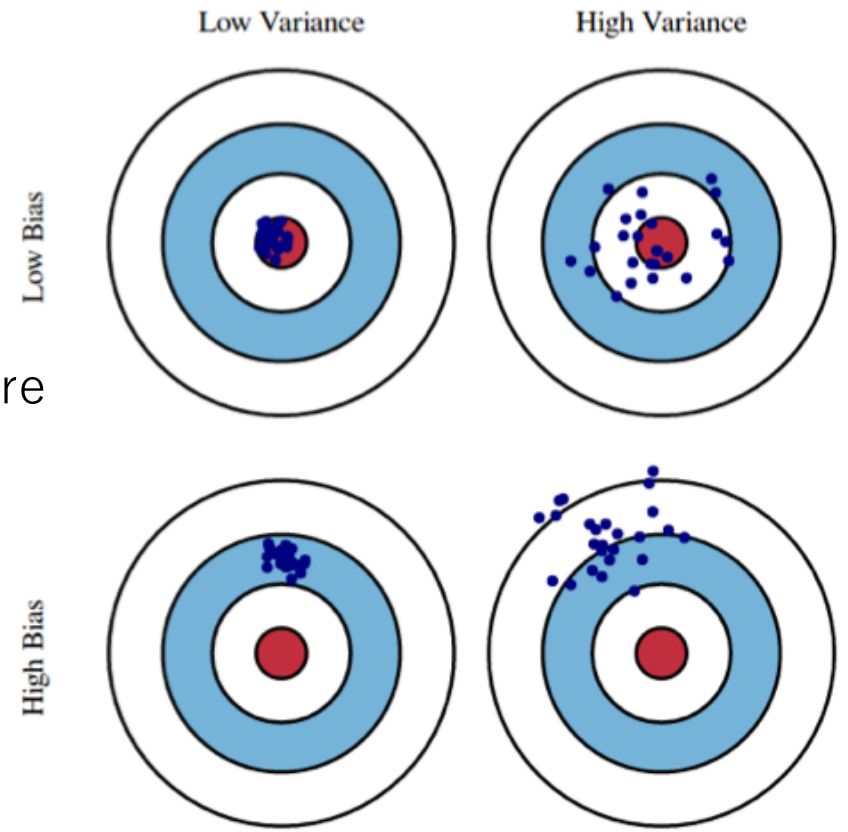
Variance

deviation from expected value

how consistent is my model / sampling

can often be dealt with by sampling more

sample inefficient



Bias

expected deviation vs true value

how close to the truth is my model

approximations or bootstrapping tend to introduce bias

biased away from an optimal agent / policy

Bootstrapping

Doing something on your own

i.e. funding a startup with your own money

Using a function to improve itself

The Bellman Equation is a bootstrapped equation

$$V(s) = r + \gamma V(s')$$

$$Q(s,a) = r + \gamma Q(s',a)$$

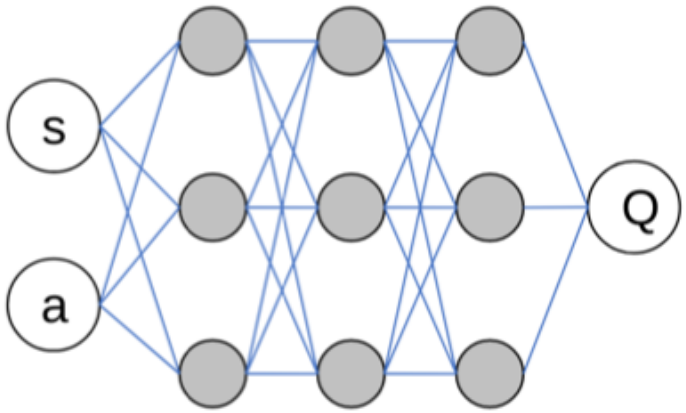
Function approximation

state	temperature s1	pressure s2	estimate
0	100	100	-1
1	100	90	1
2	90	100	0
3	90	90	1

Lookup table

$$V_\pi(s) = 3s_1 + 4s_2$$

Linear function



Non-linear function

(neural network)

Lookup table

state	temperature s1	pressure s2	estimate
0	100	100	-1
1	100	90	1
2	90	100	0
3	90	90	1

Advantages

stability

each estimate is independent of every other estimate (stability benefit)

Disadvantages

no aliasing between similar states

no sharing of knowledge between similar states or state actions

curse of dimensionality

need one entry per state – high dimensional state spaces means lots of entries

Linear function

$$V_\pi(s) = 3 * temperature + 4 * pressure$$

$$V_\pi(s) = 3s_1 + 4s_2$$

Advantages

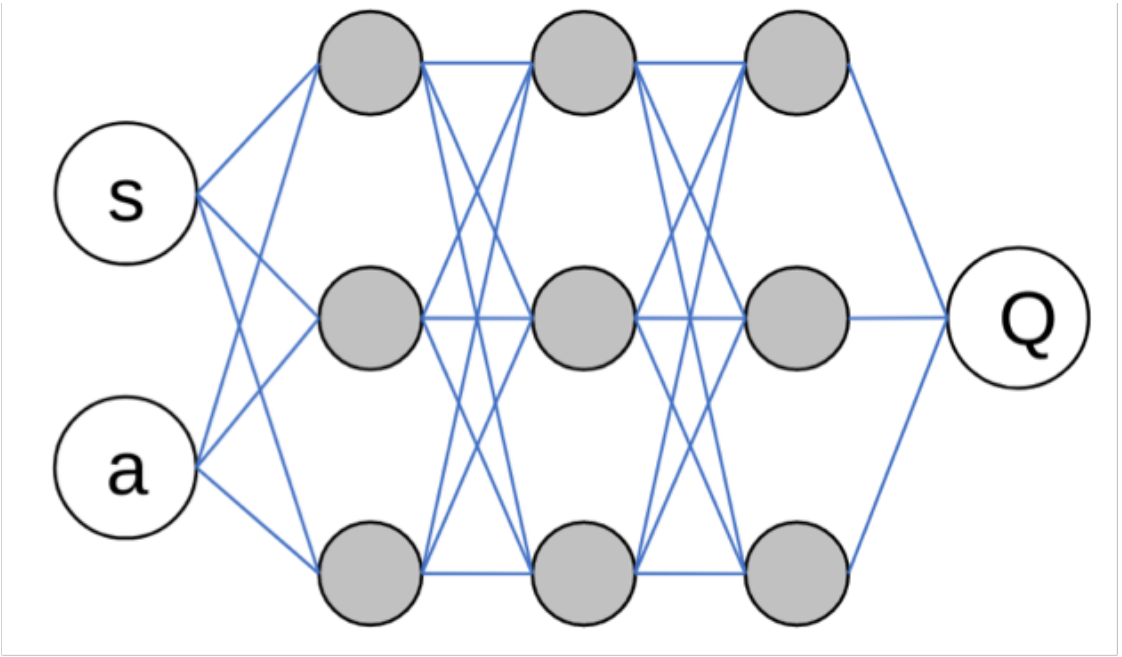
less parameters than a table

generalize across states

Disadvantages

many real world problems are non-linear

Non-linear function



Advantages

model complex dynamics

convolution for vision

recurrency for memory

Disadvantages

instability

difficult to train

Function approximation

		action	
		a1	a2
state	s1	-1	100
	s2	0	1

Lookup table

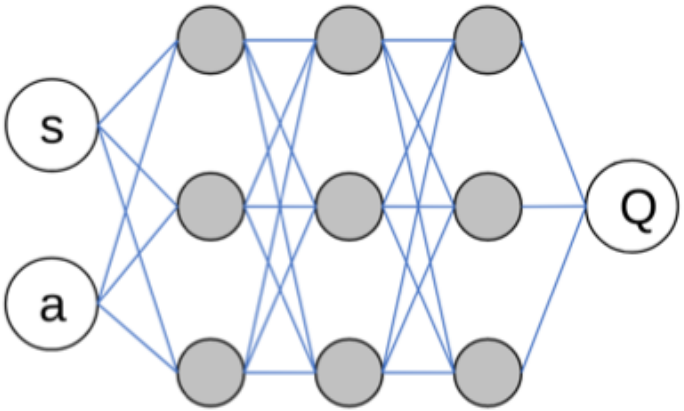
curse of dimensionality

$$V_\pi(s) = 3s_1 + 4s_2$$

Linear function

less parameters

generalize across states



Non-linear function

model complex dynamics

convolution for vision

recurrence for memory

iid

Fundamental assumption in statistical learning

independent and **i**dentically **d**istributed

In statistical learning one always assumes the training set is independently drawn from a fixed distribution

Batch size

Modern reinforcement learning trains neural networks using batches of samples

1 epoch = 1 pass over all samples

i.e. 128 samples, batch size=64

-> two forward & backward passes across net

Batch training

Np.arrange(200).reshape(10, -1)

Train.reshape(10, 2, 10) (10 2x10 images)

Train.reshape(10, 5, 4) (10 5x4 images)

Batch size

Smaller batch sizes = less memory on GPU

Batches train faster – weights are updated more often for each epoch

The cost of using batches is a less accurate estimate of the gradient

this noise can be useful to escape local minima

Batch normalization

Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift

Sergey Ioffe
Google Inc., *sioffe@google.com*

Christian Szegedy
Google Inc., *szegedy@google.com*

arXiv:1502.03167v3 [cs.LG] 2 Mar 2015

Batch Renormalization: Towards Reducing Minibatch Dependence in Batch-Normalized Models

Sergey Ioffe
Google Inc., *sioffe@google.com*

arXiv:1702.03275v2 [cs.LG] 30 Mar 2017

Batch norm.

Mean & variance of the batch is used to normalize activations

$$\frac{x_i - \mu_{\mathcal{B}}}{\sigma_{\mathcal{B}}}$$

scaled_x = x – batch_mean / batch_variance

Keep the distribution of the layer inputs constant

Reduces sensitivity to weight & bias initialization

Can use higher learning rates

Batch renorm.

Vanilla batch norm. struggles with small or non-iid batches

Vanilla batch norm. uses two different methods for normalization for training & testing

Batch renormalization attempts to fix this by using a single algorithm for both training & testing

Quiz

Generalization error = ? + ? + ?

Variance = over/underfitting?

Bias = over/underfitting?

Quiz

Advantage of lookup table versus functions

Advantage of functions over lookup table

What is iid?

Vanilla batch norm. struggles with * or * batches – why?

Quiz

Generalization error = variance + bias + noise

Variance

overfitting

seeing patterns that aren't there

Bias

underfitting

missing patterns

Quiz

Advantage of lookup table versus functions = stability

Advantage of functions over lookup table = less parameters

iid = independent & identically distributed

Vanilla batch norm. struggles with small or non-iid batches – because estimate of mean & std is harder

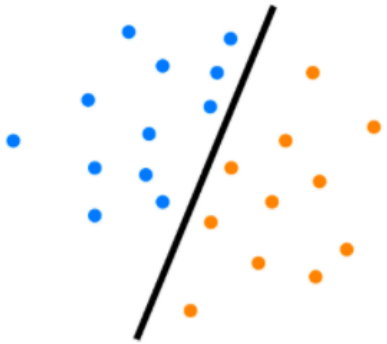
two

introduction to reinforcement learning

Machine learning

Supervised

learning
known
patterns



Random forests
Support vector machines

Feedforward neural net
Convolutional neural net
Recurrent neural net

Unsupervised

learning
unknown
patterns

Clustering algorithms

Generative adversarial nets
(GANS)

Reinforcement

taking actions
generating data
learning patterns

Value function methods

Policy gradients

Planning (i.e. model based)

Model free reinforcement learning

Value functions

Parameterize a value function

$V_\pi(s, \theta)$

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

Policy gradients

Parameterize a policy

$\pi(s, \theta)$

Actor-Critic

Parameterize both a value function & policy

$V_{\pi(s,\theta)}$

$Q_{\pi(s,a,\theta)}$

$\pi(s, \theta)$

We won’t cover

Model based RL

- model is used for planning

Evolutionary algorithms

- non-gradient based approach
- deal with sparse error signals
- easily parallelizable

General optimization methods

- i.e. cross entropy method

Applications – David Silver

- Control physical systems: walk, fly, drive, swim, ...
- Interact with users: retain customers, personalise channel, optimise user experience, ...
- Solve logistical problems: scheduling, bandwidth allocation, elevator control, cognitive radio, power optimisation, ..
- Play games: chess, checkers, Go, Atari games, ...
- Learn sequential algorithms: attention, memory, conditional computation, activations, ...

David Silver – Deep Reinforcement Learning

Applications

robotics

self driving cars

smart contracts on blockchains

recommender systems

neural network design

manufacturing

energy

Reinforcement learning is not

NOT an alternative method to use instead of a random forest, neural network etc

“I’ll try to solve this problem using a convolutional nn or RL” this is nonsensical

Neural networks (supervised techniques in general) are a tool that reinforcement learners can use

Deep reinforcement learning

Deep learning

neural networks with multiple layers

Deep reinforcement learning

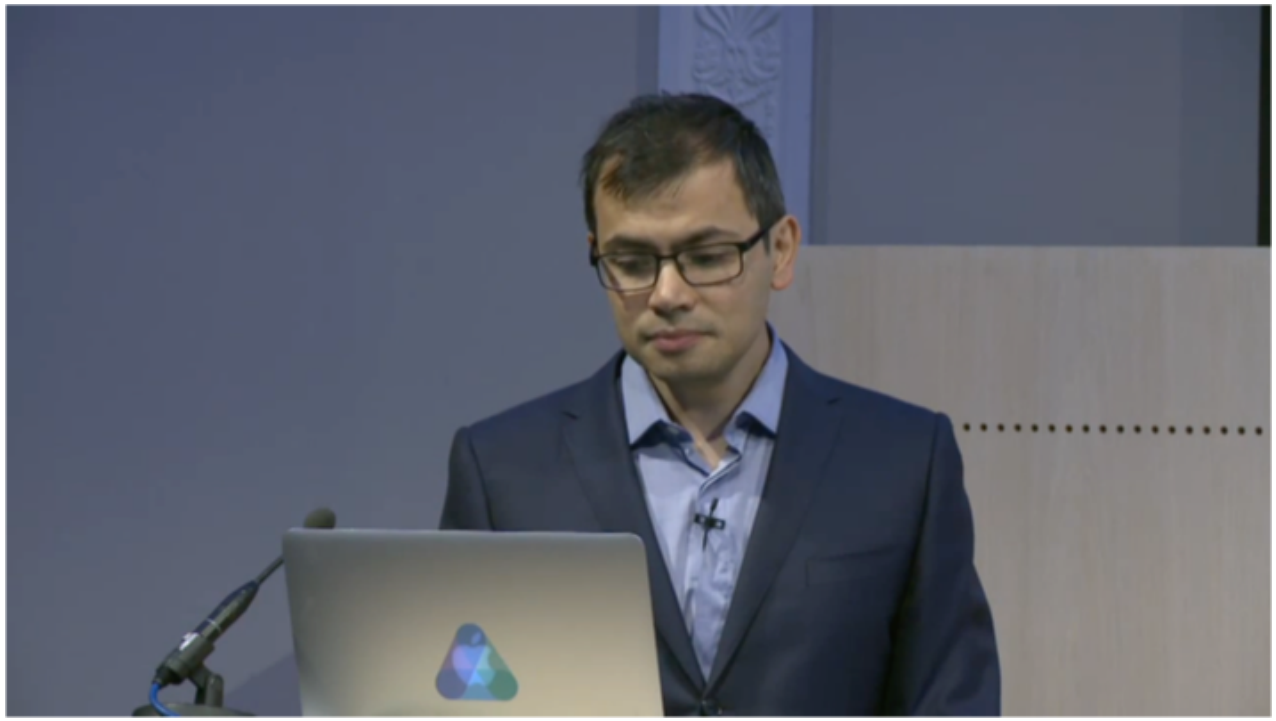
using multiple layer networks to approximate policies or value functions

Feedforward, convolutional or recurrent neural networks are all used

A new level of intelligence

Demis Hassabis is Founder & CEO of DeepMind

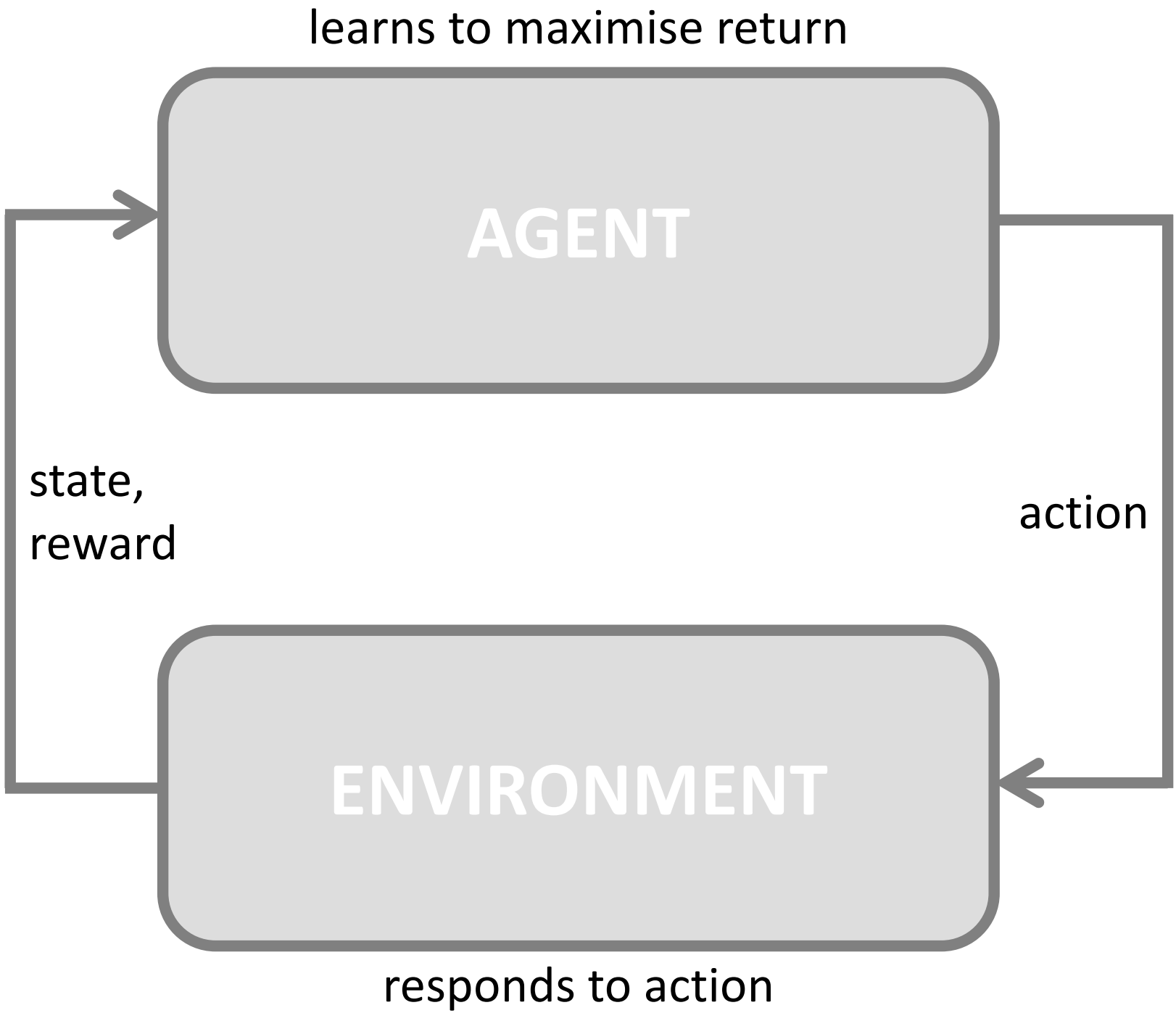
the brilliance of AlphaGo in it's 2015 series

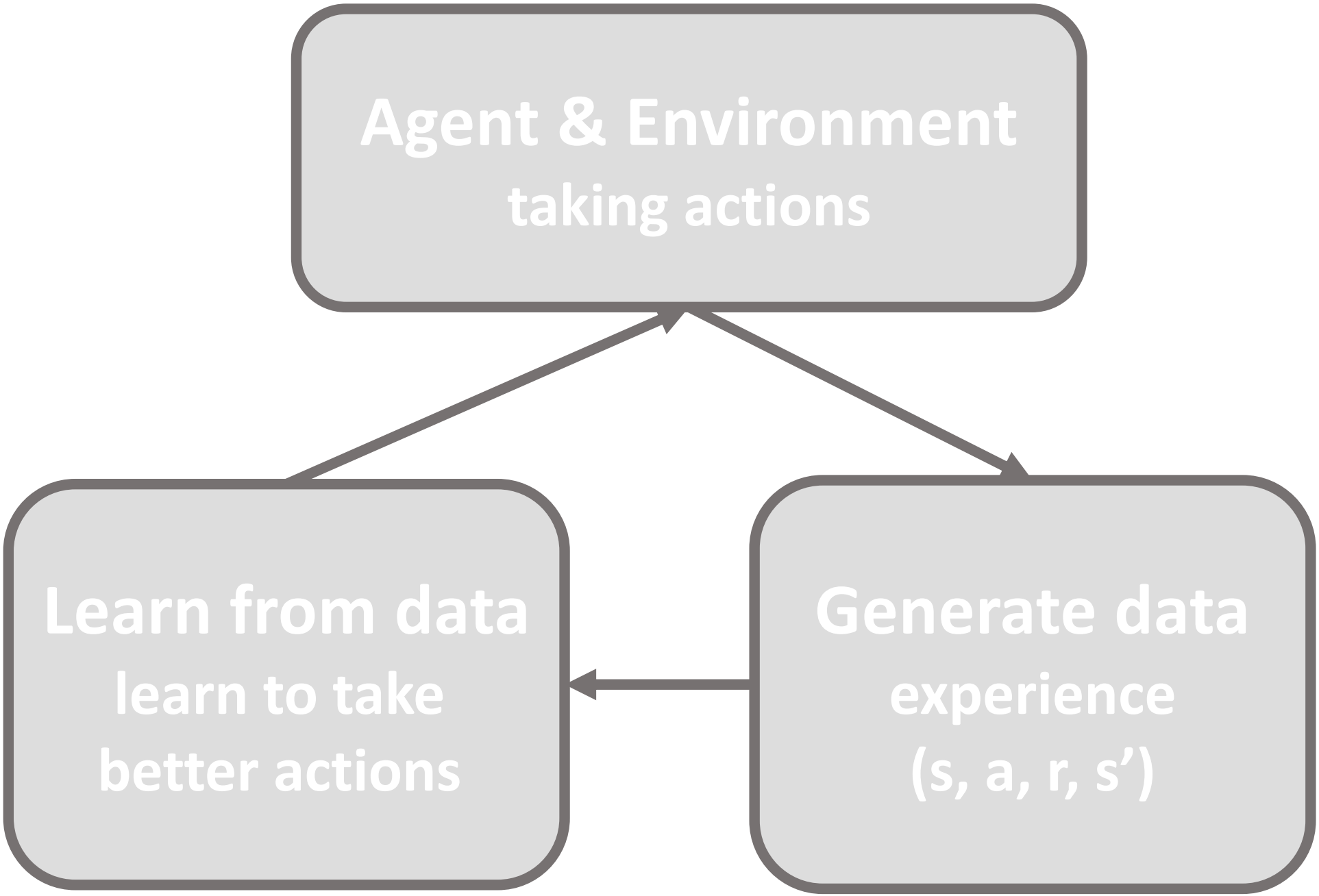


Reinforcement learning

is

learning through action


learns to maximise return
AGENT
state,
reward
action
ENVIRONMENT
responds to action


Agent & Environment
taking actions
Learn from data
learn to take
better actions
Generate data
experience
(s, a, r, s')

Biological inspiration

Neurobiological evidence that reward signals during perceptual learning may influence the characteristics of representations within the primate visual cortex (Mnih et. al 2015)

Habit formation

Cue -> Routine -> Reward

State -> Action -> Reward

Data in reinforcement learning

Data is generated by the learner

supervised learning only as good as the dataset

reinforcement learning generate more data through action

Data is the agent's experience (s, a, r, s')

It's not clear what we should do with this data

no implicit target as in supervised learning

Reinforcement learning dataset

$$[(\boldsymbol{experience}),$$
$$(\boldsymbol{experience}),$$
$$(\boldsymbol{experience}),$$
$$\ldots.$$
$$(\boldsymbol{experience})]$$

sequences of experience

Reinforcement learning dataset

$$[(\boldsymbol{s_0}, \boldsymbol{a_0}, \boldsymbol{r_1}, \boldsymbol{s'_1}),$$
$$(\boldsymbol{s_1}, \boldsymbol{a_1}, \boldsymbol{r_2}, \boldsymbol{s'_2}),$$
$$(\boldsymbol{s_2}, \boldsymbol{a_2}, \boldsymbol{r_3}, \boldsymbol{s'_3}),$$
$$\ldots.$$
$$(\boldsymbol{s_n}, \boldsymbol{a_n}, \boldsymbol{r_{n+1}}, \boldsymbol{s'_{n+1}})]$$

what should we do with this?

Four challenges

one	exploration vs exploitation
two	data
three	credit assignment
four	sample efficiency

Exploration vs exploitation

Do I go to the restaurant in Berlin I think is best – or do I try something new?

Exploration = finding information

Exploitation = using information

Agent needs to balance between the two

Exploration vs exploitation

How stationary are the environment state transition and reward functions? How stochastic is my policy?

Design of reward signal vs. exploration required

Algorithm does care about the time step

too small = rewards are delayed = credit assignment harder

Data – correlated samples

All the samples collected on a given episode are correlated (along the state trajectory)

This breaks the iid assumption of independent sampling

Data – non-stationary distribution

Environment can be non-stationary

Learning changes the data we see

Exploration changes the data we see

All of these break the identically distributed assumption of iid

Reinforcement learning will always break supervised learning assumptions about data quality

Credit assignment

Which actions give us which reward

Reward signal is often

delayed benefit of action only seen much later

sparse experience with reward = 0

Sometimes we can design a more dense reward signal for a given environment

Sample efficiency

How quickly a learner learns

How often we reuse data

do we only learn once or can we learn from it again

on vs off policy

How much we squeeze out of data

i.e. learn a value function, learn a environment model

Four challenges

exploration vs exploitation

how good is my understanding of the range of options

data

biased sampling, non-stationary distribution

credit assignment

which action gave me this reward

sample efficiency

learning quickly, squeezing information from data

two

introduction to reinforcement learning

Markov Decision Processes

Markov Decision Processes

Mathematical framework for the reinforcement learning problem

A lot of theory (which we won't cover today) proves that certain algorithms will converge to unbiased or optimal values in MDPs

Markov property

Future is conditional only on the present

Can make prediction or decisions using only the current state

Any additional information about the history of the process will not improve our decision

Markov property

$$p(s_{t+1}|s_1, a_1 \ldots s_t, a_t)$$

$$=$$

$$p(s_{t+1}|s_t, a_t)$$

State vs observation

Not all problems are true MDPs

- it's more of an ideal - often unrealistic in practice

Often we are limited to partially observed MDPs

- POMDP
- the agent can only see some of the state variables

Lose convergence guarantees

- still might be able to train a good agent

Formal definition of an MDP

Set of states	S
Set of actions	A
State transition function	$P(s' \mid s, a)$
Reward function	$R(r \mid s, a)$
Initial state distribution	$P(s_0)$
Discount factor	γ
Horizon	H

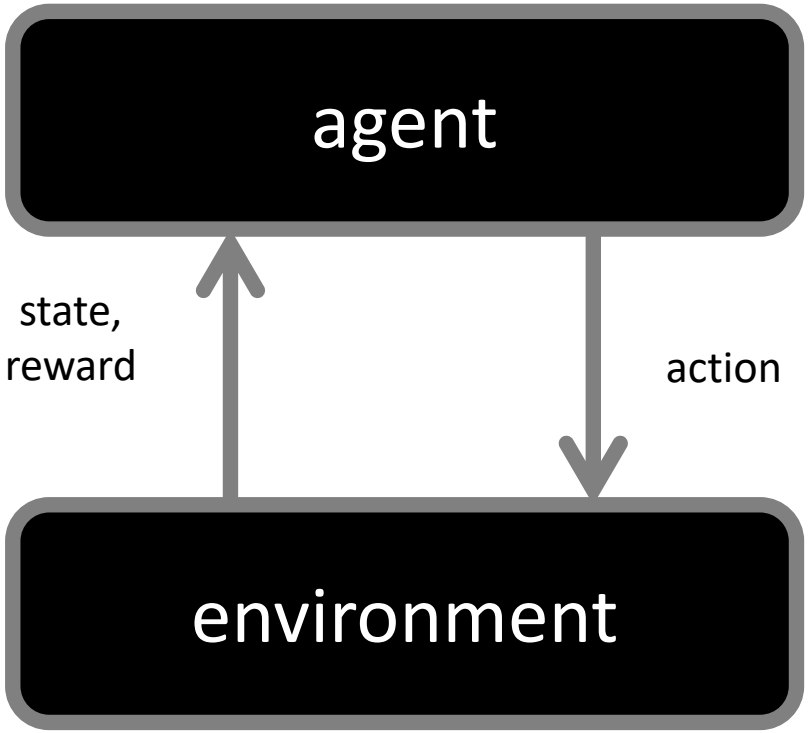
Informal definition of an MDP

Two objects

agent + environment

Three signals

state, action, reward



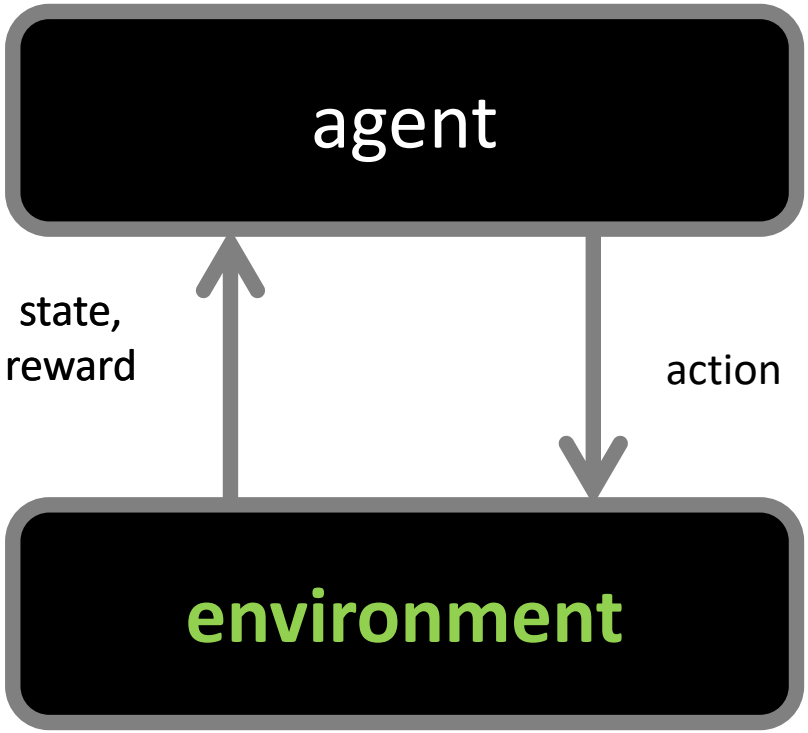
Environment

Real or virtual

Discrete or continuous

action space

state space



Episodic vs. non-episodic

episodic is a special case of non-episodic (the absorbing state)

Discrete vs continuous spaces

Discrete action space

Continuous action space

Car accelerator from 0-100

We could discretize this as

[0, 1, 2, 3 … 99, 100]

Discrete vs continuous spaces

This can be a key consideration

some algorithms can only work with discrete spaces

Q-Learning requires a discrete action space to argmax across

Possible to discretize a continuous action space

Discretiziation

Too coarse

non-smooth control output

Too fine

curse of dimensionality

computationally expensive

Discretization requires some prior knowledge

State

Flexible concept

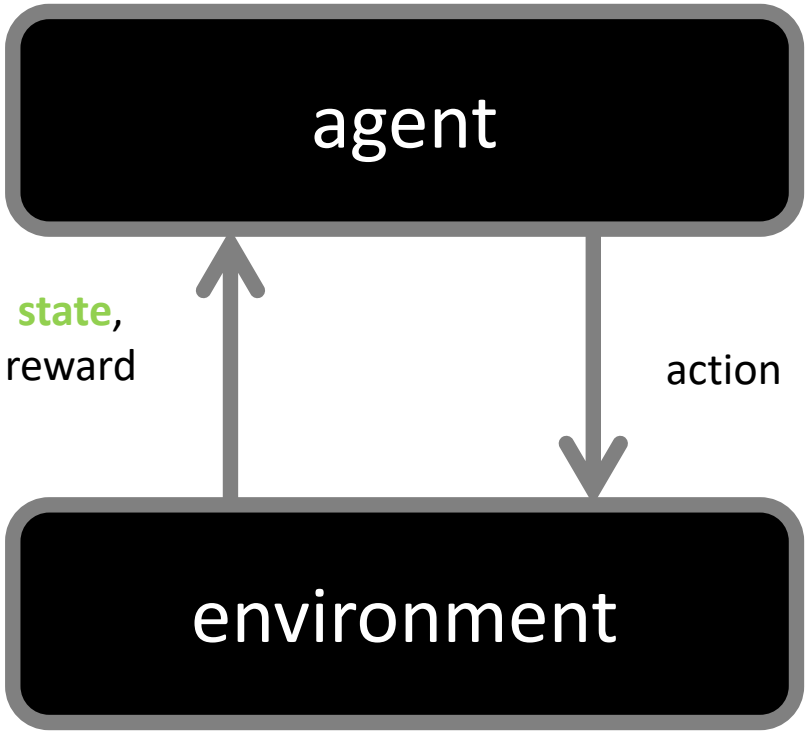
n-d array

Information for agent to

learn from

choose next action

Fully observed vs hidden



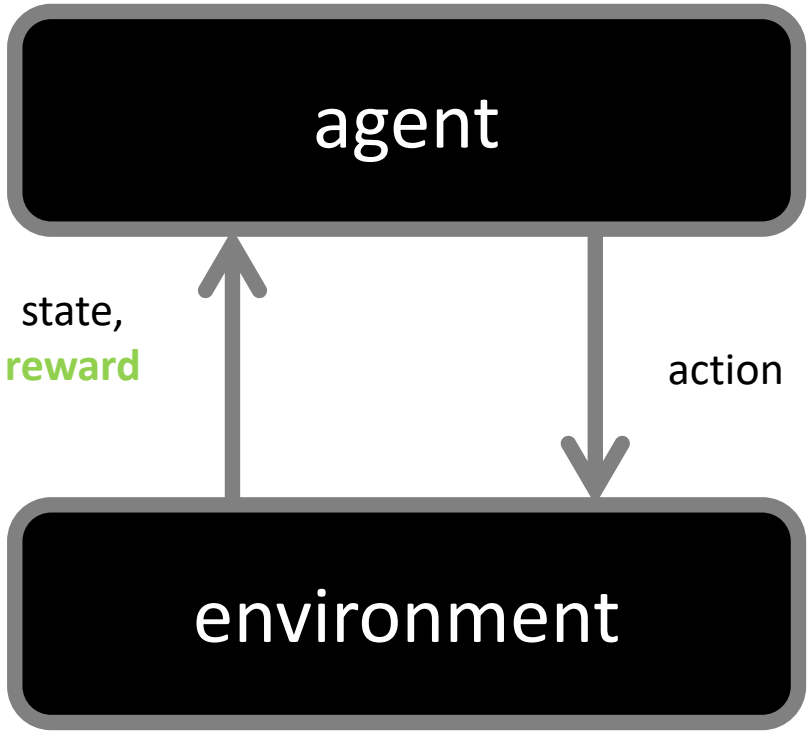
Reward

Flexible concept

scalar

delayed

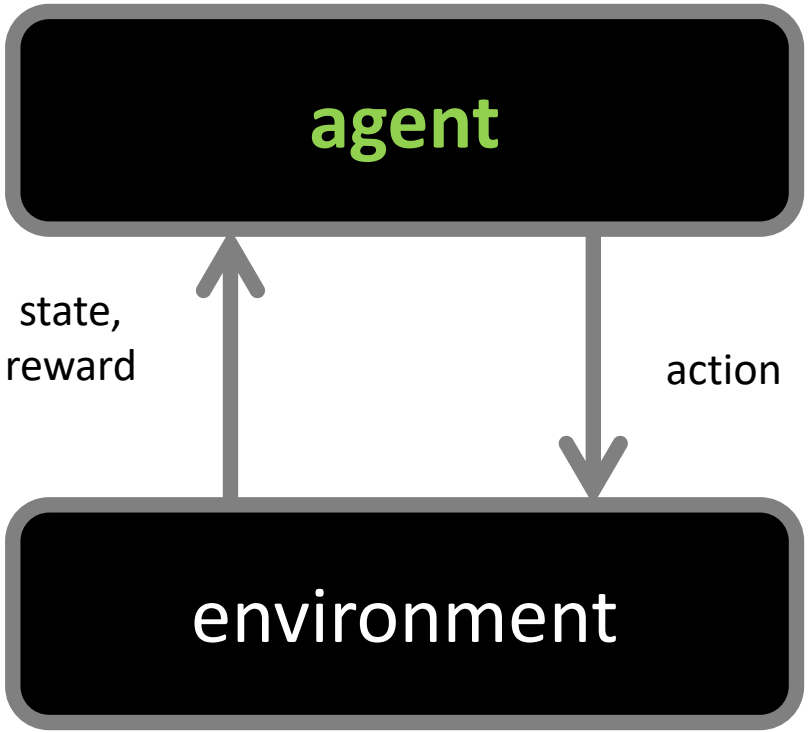
sparse



Agent

Learner & decision maker

maximize cumulative discount reward



Agent always has a policy

even if it's a bad one!

Reward Hypothesis

Maximising return is making an assumption about the nature of our goals

We are assuming that:

Goals can be described by the maximization of expected cumulative reward

Do you agree with this?

Policy - $\pi(s)$

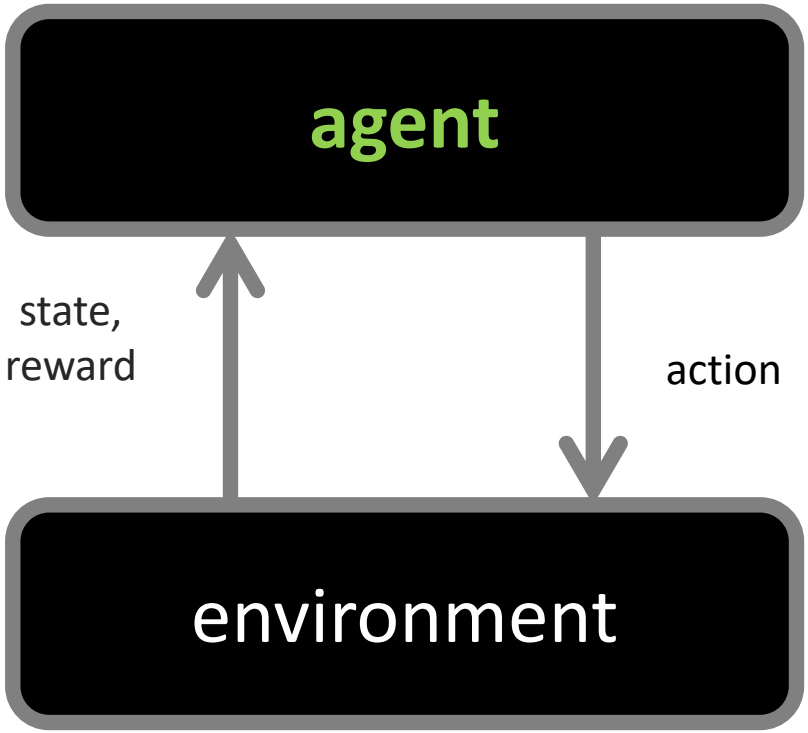
Rules to select actions

Always follow a policy

act randomly

always pick specific action

optimal π^*



Deterministic or stochastic

Random policy - $\pi(s)$

```
def random_policy (state):
    #  randomly sample across the action space
    action = np.random.uniform(action_space)
    #  we don't use the state to choose an action
    return action
```

Policy - $\pi(s)$

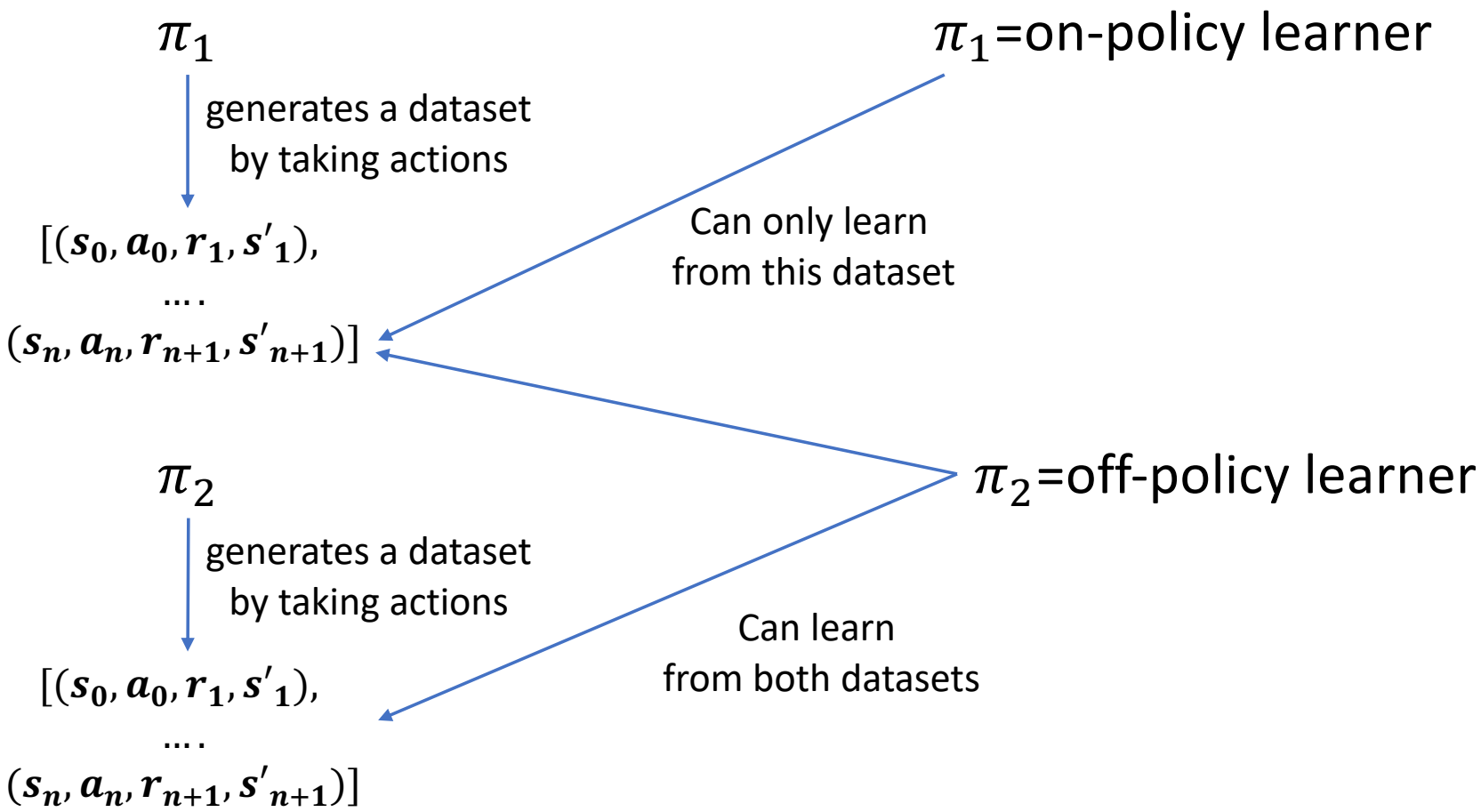
Policy can be

parameterized directly (policy gradient methods)

generated from a value function (value function methods)

On vs. off-policy learning

On vs off policy learning

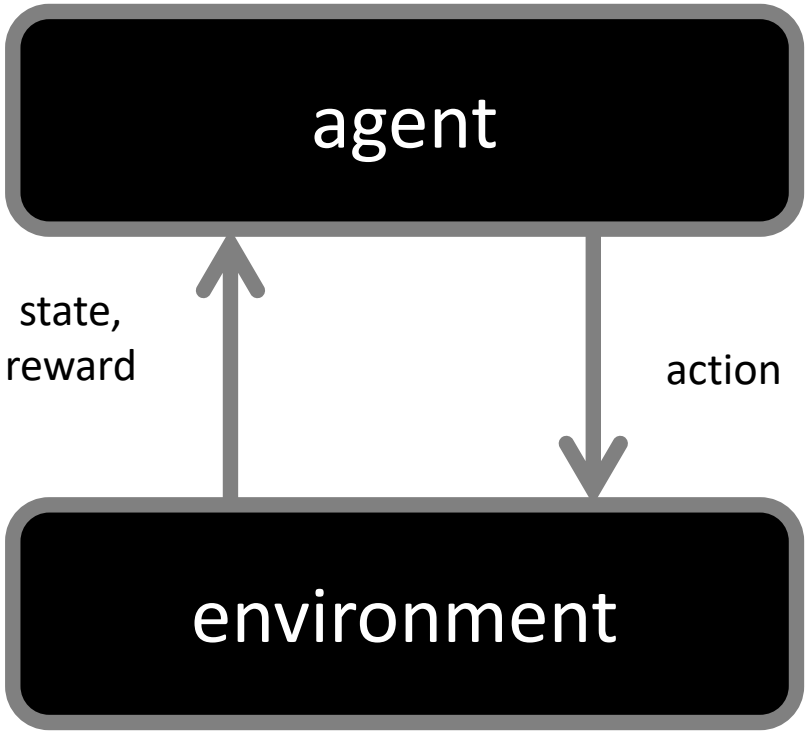


Environment model

Predicts environment response to actions

predicts s', r from s, a

not predicting return!



Sample vs. distributional model

Model is learnt

Model can be used for planning

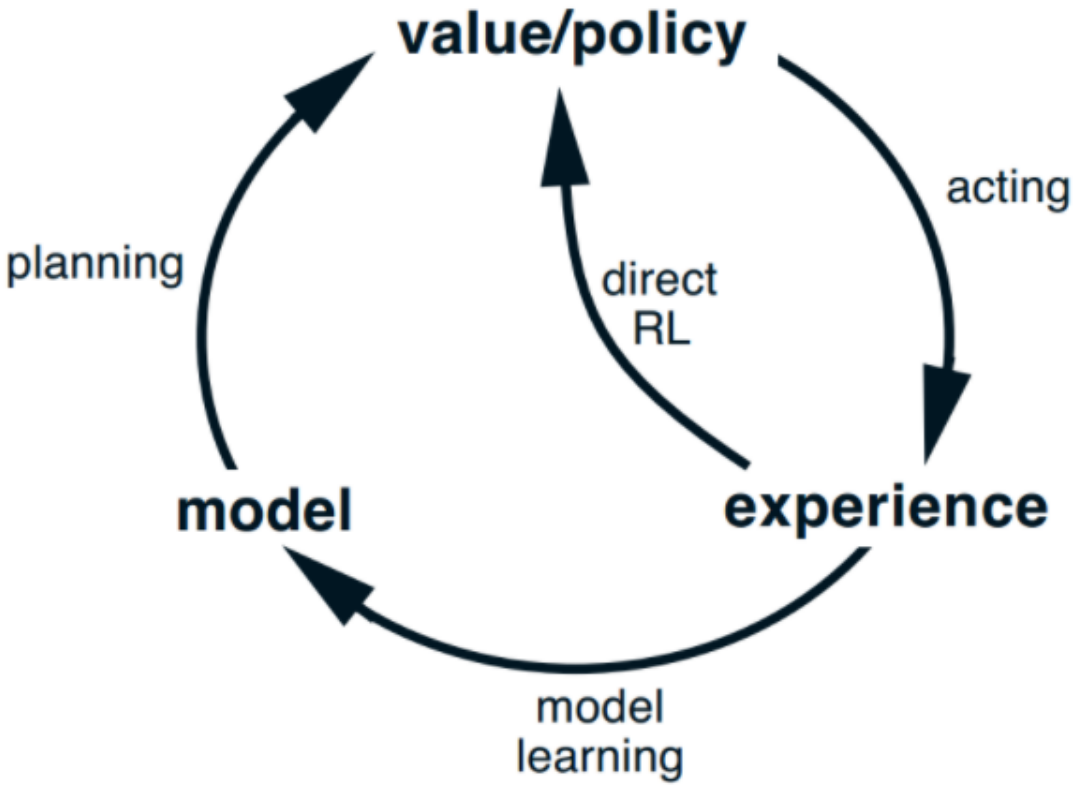
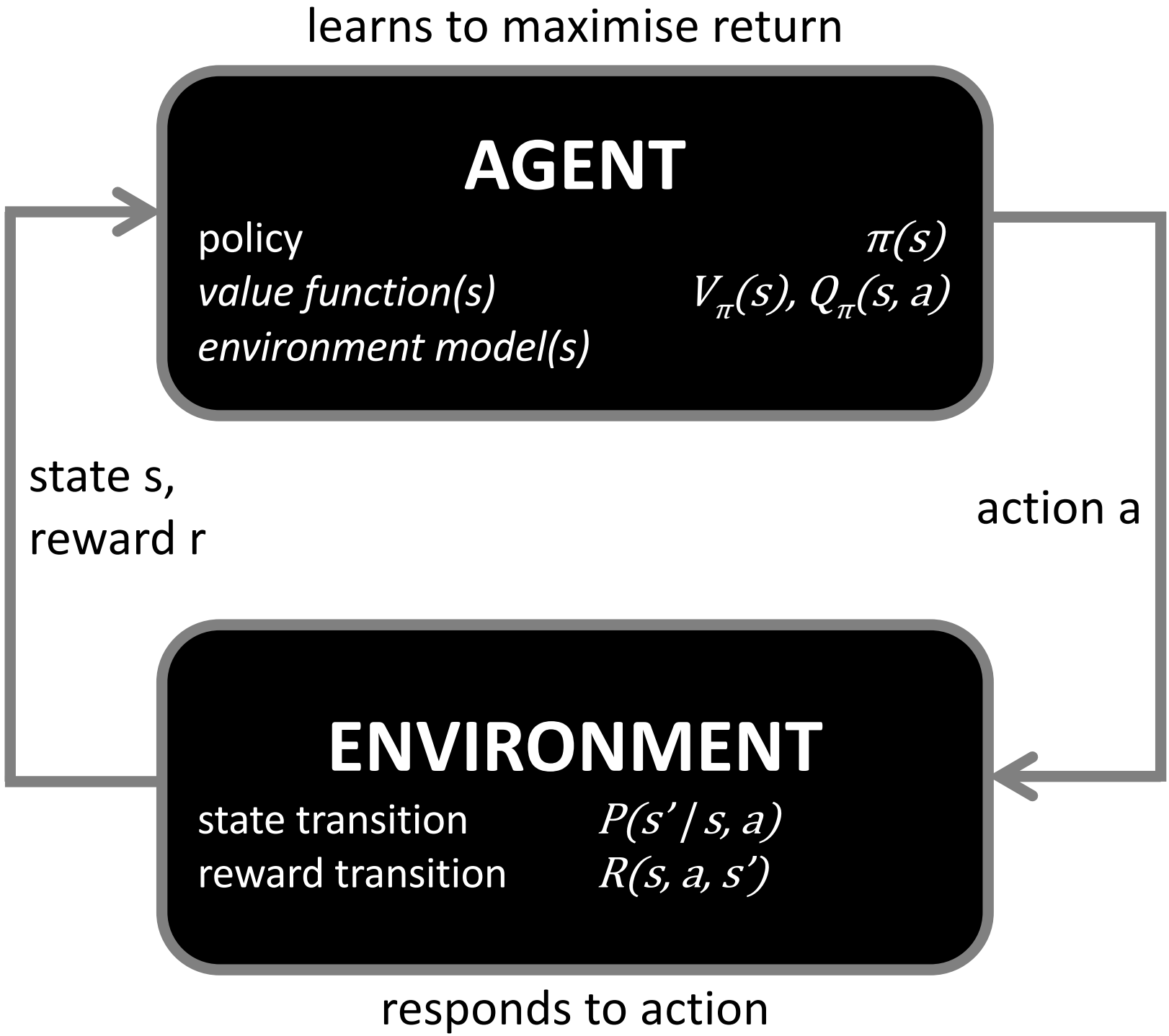


Figure 8.2: Relationships among learning, planning, and acting.

Sutton & Barto


learns to maximise return
AGENT
policy
$\pi(s)$
value function(s)
$V_\pi(s)$, $Q_\pi(s, a)$
environment model(s)
state s,
reward r
action a
ENVIRONMENT
state transition
$P(s' \mid s, a)$
reward transition
$R(s, a, s')$
responds to action

Return

Agent's goal is to maximise expected future reward

Return (G_t) is the **total discounted future reward**

after time t, until end of episode ($k = \infty$)

$$G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

*return = reward + discount * reward + discount^2 *reward*

$\gamma \in [0,1)$

Why would we discount future rewards?

Discounting

Future is uncertain

stochastic environment

Matches human thinking

hyperbolic discounting

Finance

time value of money

Discounting

Makes return for infinite horizon problems finite

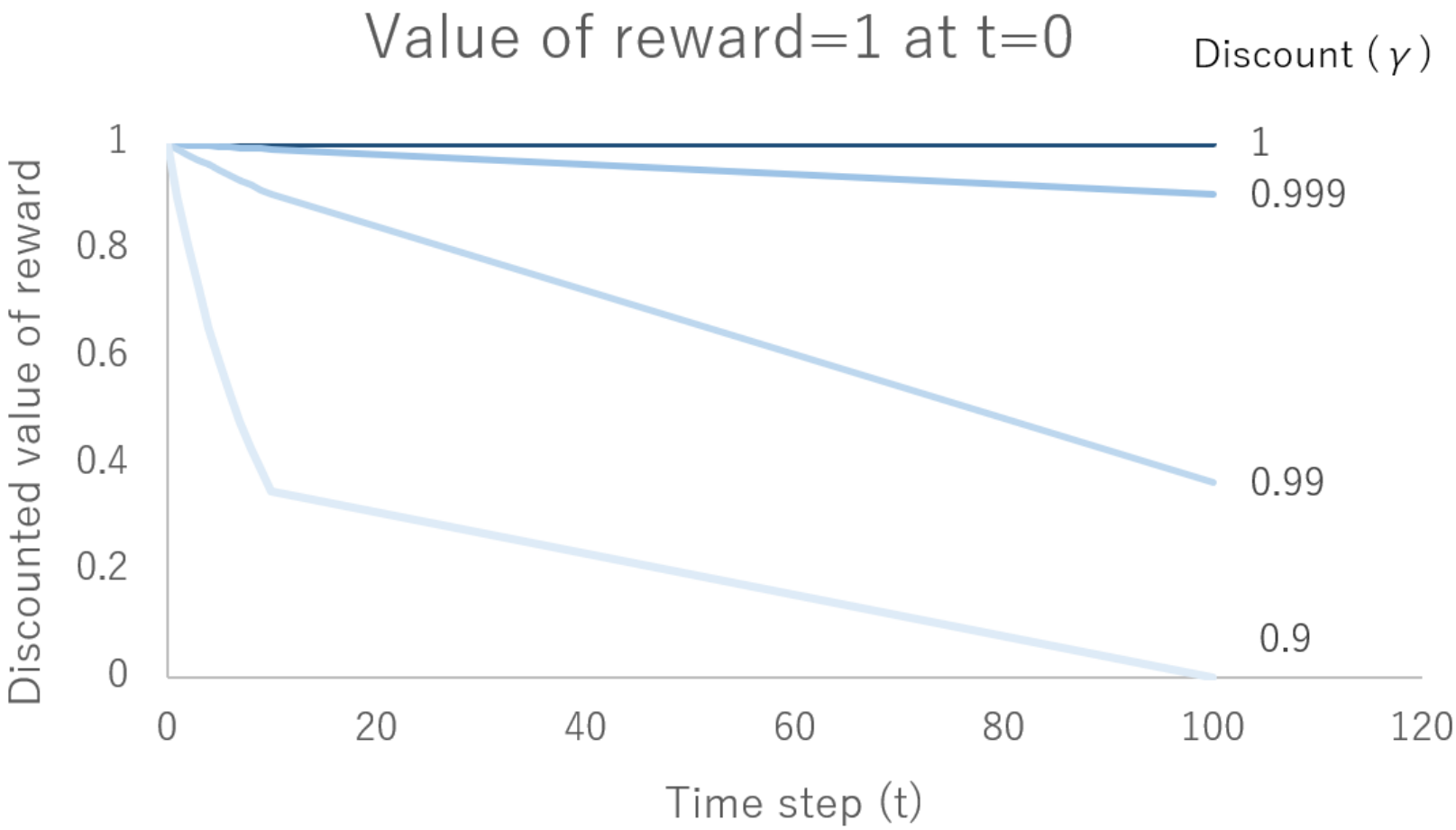
an infinite series with a finite sum (geometric series)

for discount [0,1)

Many games with tree-like structures (without cycles) can use a discount of 1. Also when time to solve is irrelevant (ie a board game)

https://www.jair.org/media/5507/live-5507-10333-jair.pdf

How much is a reward of 1 worth after t time steps



break

recap quiz

Two objects ?

Three signals ?

Policy ?

Model ?

Return ?

recap

Two objects	agent & environment
Three signals	state, reward & action
Policy	rules to select actions
Model	predicts next state & reward
Return	future discounted reward

three - value function methods

introduction to value functions

Bellman equation

Value function

how good is this state

Action-value function

how good is this action

Value function

$V_\pi(s)$

how good is this state

Action-value function

$Q_\pi(s, a)$

how good is this action

Value functions

Value function

$$V_\pi(s) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} = \mathbb{E}_\pi[G_t | S_t = s]$$

expected return *when in state* s*, following policy* π

Action-value function

$$Q_\pi(s,a) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

expected return *when in state* s*, taking action* a*, following policy* π

Value functions

Value functions are predictions of the future

- prediction of return
- prediction of future expected discounted reward
- always conditioned on a policy

But we don't know this function

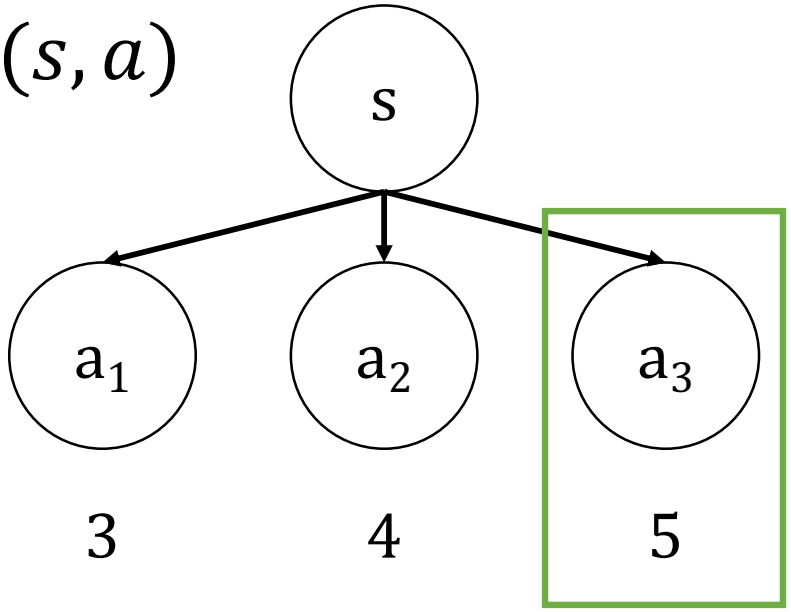
- agent must learn it
- once we learn it – how will it help us to act?

Generating a policy from a value function

We can generate the optimal policy π^* from the optimal value function $Q_*(s, a)$

Select an action by $\underset{a}{\operatorname{argmax}}\, Q_*(s, a)$

Select action with largest Q_*



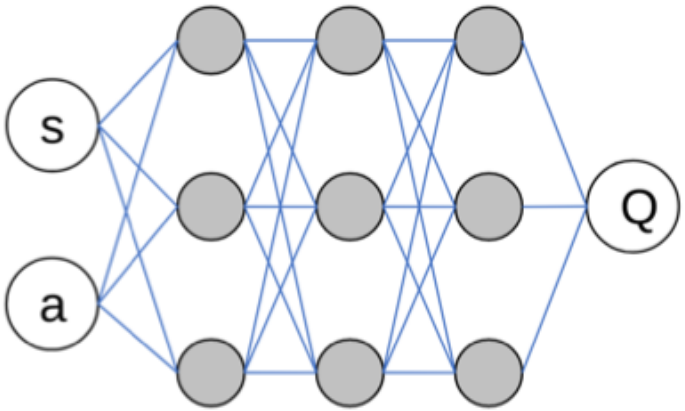
Greedy policy - $\pi(s)$

```
def greedy_policy(state):
    #  get every possible state action
    #  this can be computationally expensive!
    state_actions = all_action_combinations(state)

    #  predict Q(s,a) for each state action
    #  this can be computationally expensive!
    Q_values = Q.predict(state_actions)

    #  get the action with highest Q
    action = np.argmax(Q_values)
    return action
```

Value function approximation

state	temperature s1	pressure s2	estimate
0	100	100	-1
1	100	90	1
2	90	100	0
3	90	90	1

$$V_\pi(s) = 3s_1 + 4s_2$$



Lookup table

curse of dimensionality

Linear function

less parameters

generalize across states

Non-linear function

model complex dynamics

convolution for vision

recurrency for memory

Richard Bellman

Dynamic programming in 1953

Bellman expectation equation

$$G_\pi(s) = r + \gamma G_\pi(s')$$

return = reward + discounted return from next state

Also introduced the curse of dimensionality

number of states grows exponentially with state variables

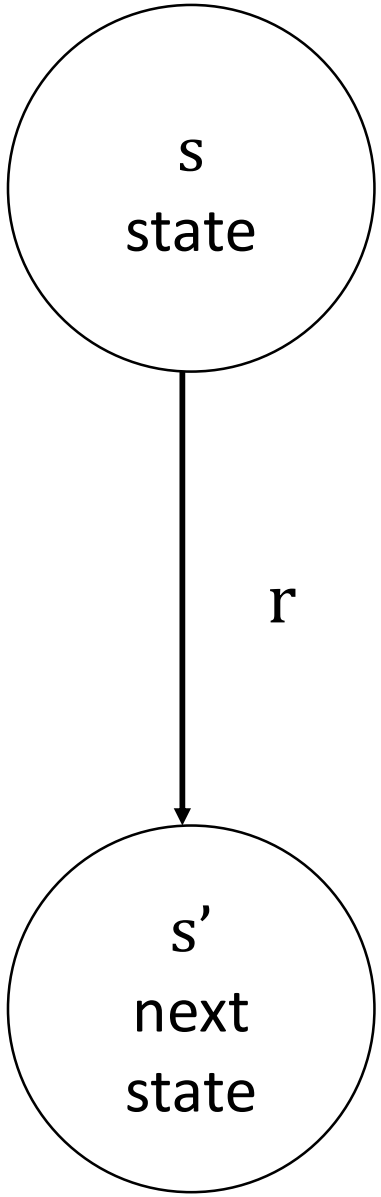
What happens if we combine a value function with the Bellman equation

$$V_\pi(s) = \sum_{k=0}^{\infty} \gamma^k r_{t+k+1}$$

$$V_\pi(s) = r + \gamma V_\pi(s')$$

$$G_\pi(s) = r + \gamma G_\pi(s')$$

note the conditioning on a certain policy



Recursive relationship between states

$V(s)$
value of being in state

$=$

r
reward we get after leaving state

$+$

$\gamma V(s')$
discounted value of next state

Why is the Bellman equation important?

Bellman equation

Create a bootstrapped target to improve value function approximations

the target is used by supervised learners

We can improve our approximation by

$$\text{loss} = \min[\underbrace{r + Q(s', a)}_{\text{target}} - \underbrace{Q(s, a)}_{\text{current function approximation}}]$$

Bellman equation

Value function approximation is crucial for policy improvement

Policy approximation vs policy improvement

prediction vs control

Two problems

1. improving our approximation of the value function
2. improving our policy (ie our actions)

Prediction, control, planning

The prediction problem, or policy evaluation, is to compute the state or action value function for a policy

The control problem is to find the optimal policy

Planning constructs a value function or a policy with a model

recap quiz

Why is a value function conditioned on a policy

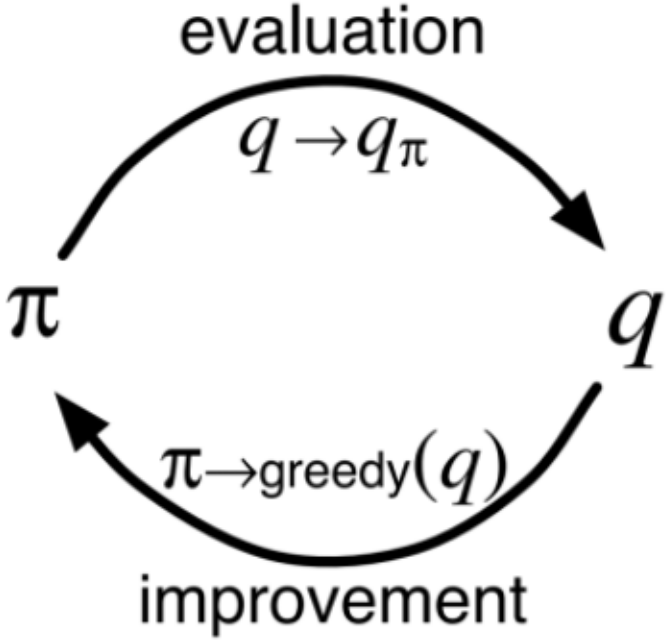
i.e. $V_\pi(s)$ or $Q_\pi(s, a)$

If we have the true optimal value function – how can we use it to act?

Approximation & improvement

Three different methods for approximation (there are more!)

- dynamic programming
- Monte Carlo
- temporal difference



One method for policy improvement

- change policy to favour the best action

Sutton & Barto

three – value function methods

introduction to value functions & Bellman equation

value function approximation

dynamic programming

Dynamic programming

Imagine we had access to a perfect environment model

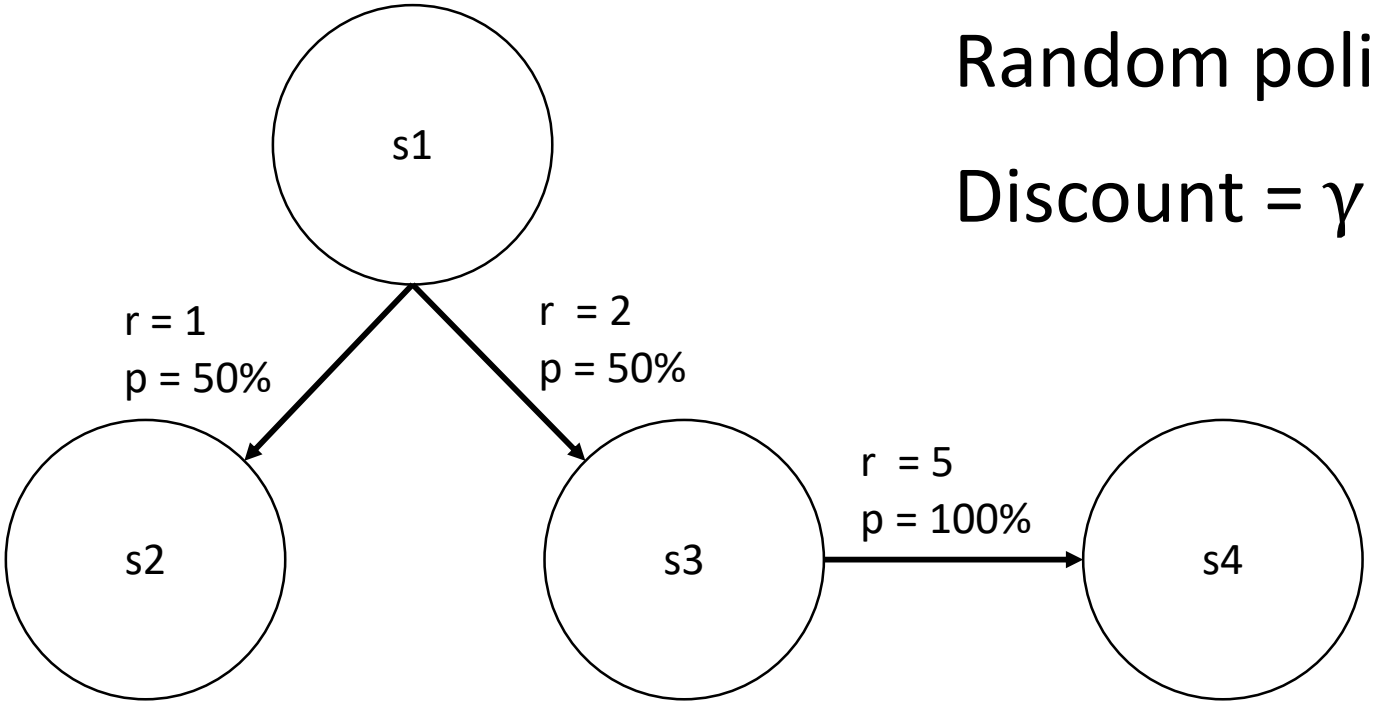
know state transition probabilities $P(s|s', a)$

know the reward function $R(s, a, s')$

How would you use this perfect environment model for value function approximation?

use the Bellman equation

Dynamic programming backup



The state transition probabilities p depend both on the environment & policy

Dynamic programming backup

We can now perform iterative backups of the value of each state backwards in time

note – return for all terminal states are zero

$$V(s_4) = 0$$
$$V(s_2) = 0$$

$$V(s_3) = P_{34}[r_{34} + \gamma V(s_4)]$$

$$V(s_1) = P_{12}[r_{12} + \gamma V(s_2)] + P_{13}[r_{13} + \gamma V(s_3)]$$

Dynamic programming backup

$$V(s_4) = 0$$
$$V(s_2) = 0$$

$$V(s_3) = P_{34}[r_{34} + \gamma V(s_4)] = 1 * [5 + 0.9 * 0] = 5$$

$$V(s_1) = P_{12}[r_{12} + \gamma V(s_2)] + P_{13}[r_{13} + \gamma V(s_3)]$$

$$V(s_1) = 0.5 * [1 + 0.9 * 0] + 0.5 * [2 + 0.9 * 5] = 3.75$$

Dynamic programming

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s\right] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\Big[r(s,a,s') + \gamma v_\pi(s')\Big],
\end{aligned}
$$

Sutton & Barto

Value of state s — POLICY — MODEL — BOOTSTRAPPED ESTIMATE

Value function estimate $V(s)$ depends on

our policy (what action we pick in state s)

the environment model (state & reward transitions)

our estimate of $V(s')$ (bootstrapping)

Dynamic programming

$$
\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi\left[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots \mid S_t = s\right] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1}) \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} p(s'|s,a)\Big[r(s,a,s') + \gamma v_\pi(s')\Big],
\end{aligned}
$$

Sutton & Barto

Value of state s — POLICY — MODEL — BOOTSTRAPPED ESTIMATE

V(s) depends on on the value of all possible s'

even when P(s|s', a)=0

Computationally expensive – every state is backed up by every other state

Dynamic programming

Requires a perfect environment model

we don't learn from experience at all!

Bootstrapped

value function improves itself

Reliable convergence for tabular value function, finite, fully observed MDPs

three – value function methods

introduction to value functions & Bellman equation

value function approximation

dynamic programming

Monte Carlo

Monte Carlo

No model – learn from actual experience

experience = sequences of (s, a, r, s')

No bootstrapping

$V(s)$ = average of true discounted returns experienced

Episodic only

because we need to know the true discounted return

no within episode learning

Monte Carlo

Estimate value of state by averaging returns observed after visit to that state

estimate by sampling observed returns

As more episodes run, estimate should converge to the expected value

expected return of state s following policy π

Low bias & high variance – why?

Monte Carlo

High variance

- we need to sample enough episodes for our averages to converge
- can be a lot for stochastic or path dependent environments

Low bias

- we are using actual experience
- no chance for a bootstrapped function to mislead us

Monte Carlo

Initialize:

$\pi \leftarrow$ policy to be evaluated i.e. greedy policy

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

(lookup table)

Repeat forever:

(a) Generate an episode using π

(b) For each state s appearing in the episode:

$G \leftarrow$ return following the first occurrence of s

Append G to $Returns(s)$

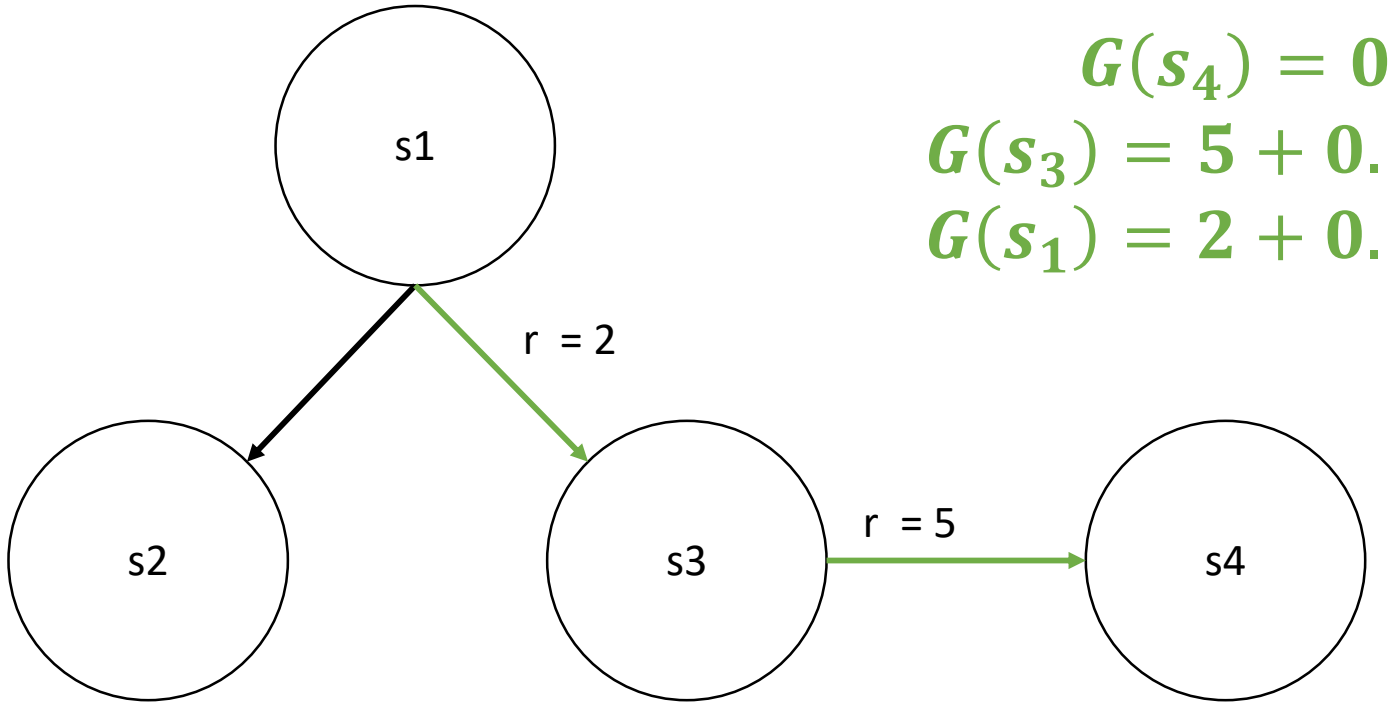
$V(s) \leftarrow \text{average}(Returns(s))$

David Silver

Interesting thing about Monte Carlo

Computational expense of estimating the value of state s is independent of the number of states S

this is because we use experienced state transitions



$$G(s_4) = 0$$
$$G(s_3) = 5 + 0.9 * 0$$
$$G(s_1) = 2 + 0.9 * 5$$

Monte Carlo

Learn from actual experience – no environment model

No bootstrapping – use true discounted returns

Episodic problems only – no learning online

High variance, low bias

three – value function methods

introduction to value functions & Bellman equation

value function approximation

- dynamic programming
- Monte Carlo
- temporal difference

Temporal difference

Learn from actual experience – no environment model

Bootstrap – learn online

Episodic & non-episodic problems

Usually converges faster than Monte Carlo

Temporal difference

Estimate bootstrap the approx. of $V(s)$ using $V(s')$

like dynamic programming

Sample from experienced trajectories

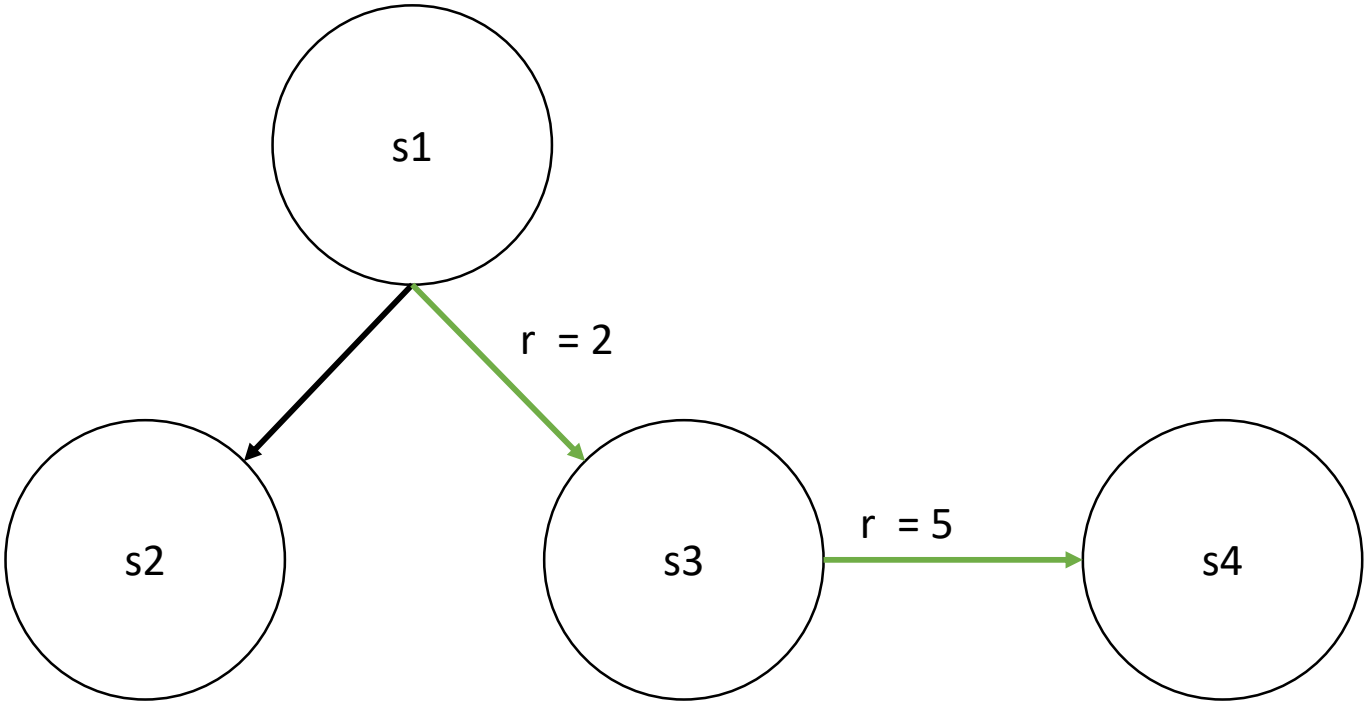
like Monte Carlo

TD(0)

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big]$$

TD error

TD(0) backup



$$V(s_1, \theta_{i+1}) = V(s_1 \theta_i) + \alpha[r_{23} + \gamma V(s_3, \theta_i) - V(s_1, \theta_i)]$$

Bootstrapping

Using the experienced trajectory

TD(0) algorithm

Input: the policy π to be evaluated
Input: a differentiable value function $\hat{v}(s, \boldsymbol{w})$, $\hat{v}(terminal, \cdot) = 0$
Output: value function $\hat{v}(s, \boldsymbol{w})$
initialize value function weight $\boldsymbol{w}$ arbitrarily, e.g., $\boldsymbol{w} = 0$
for *each episode* **do**
    initialize state s
    for *each step of episode, state* s *is not terminal* **do**
        $a \leftarrow \pi(\cdot|s)$
        take action a, observe r, s'
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha[r + \gamma\hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})]\nabla\hat{v}(s, \boldsymbol{w})$
        $s \leftarrow s'$
    end
end

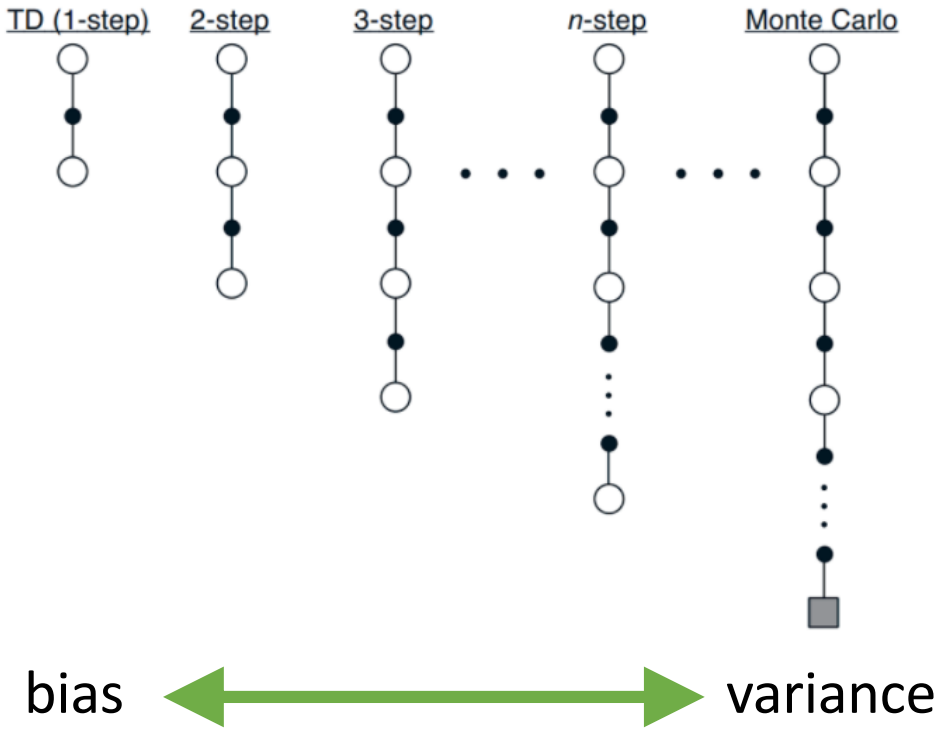
Algorithm 4: TD(0) with function approximation, adapted from Sutton and Barto (2017)

Ref = Li (2017)

Linking together

Temporal difference & Monte Carlo are two extremes of the same scale

Bias-variance trade



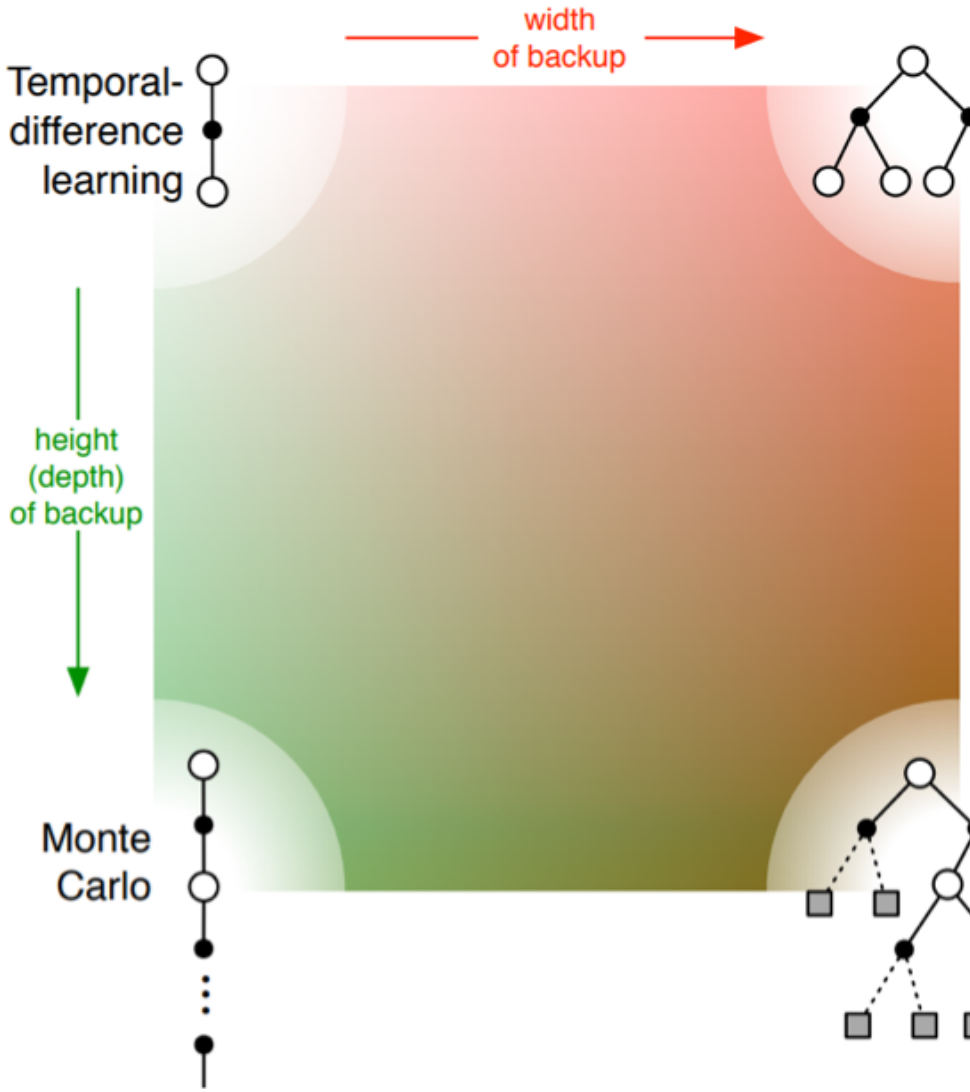
Sutton & Barto

Further reading

- n-step methods
- TD(λ)
- Eligibility traces

Unified View

look ahead only one step – but only the experienced next state



look ahead only one step – but at every possible next state

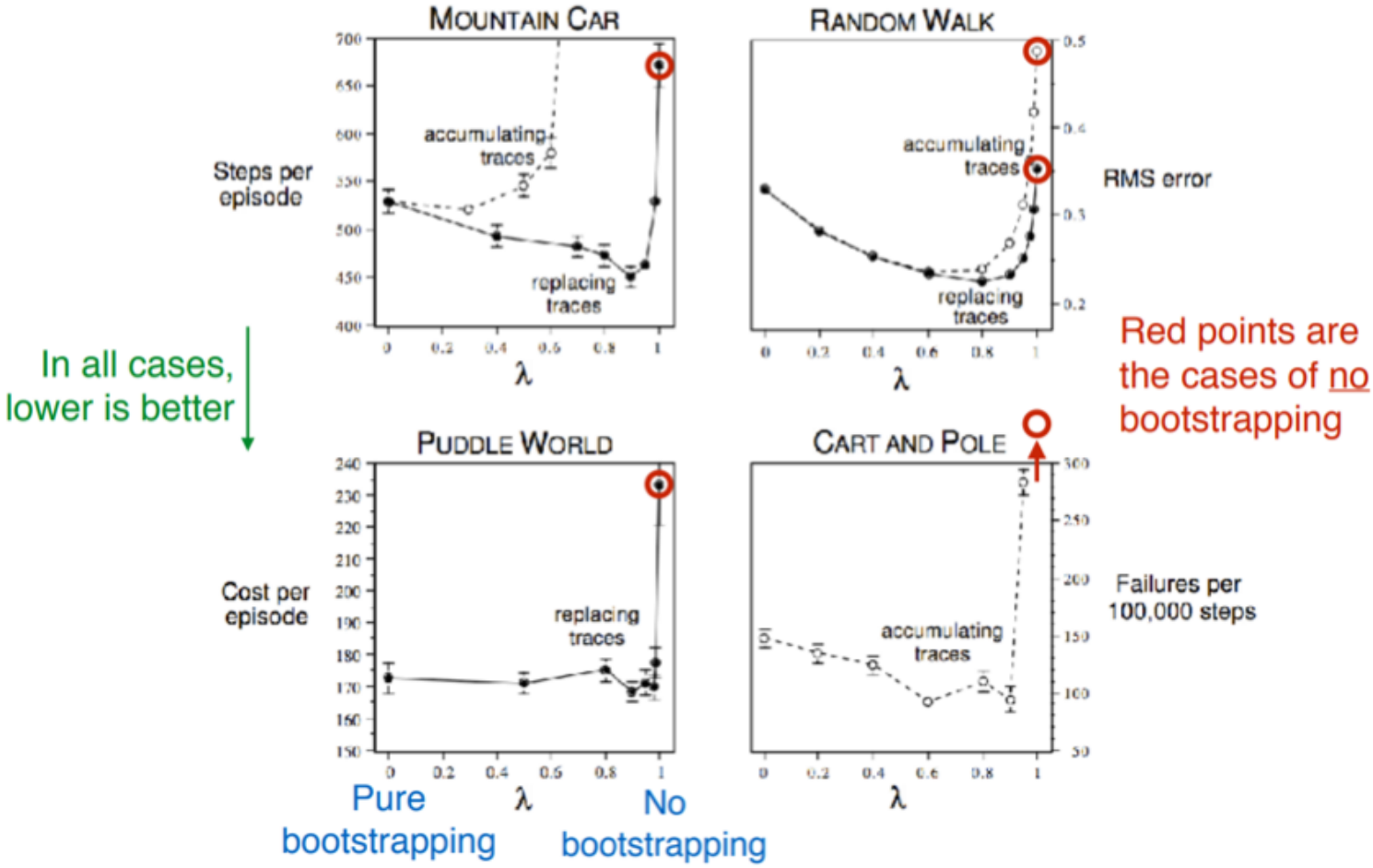
look ahead all the way to the end– but only the experienced next states

look ahead all the way to the end AND every possible next state

The Long-term of AI & Temporal-Difference Learning – Rich Sutton

4 examples of the effect of bootstrapping

suggest that λ=1 (no bootstrapping) is a very poor choice
(i.e., Monte Carlo has high variance)



λ = 0 = TD(0)

λ = 1 = full Monte Carlo

Conclusion – full Monte Carlo isn't very good!

break

recap quiz

	Model	Uses experience	Bootstraps
Dynamic programming			
Monte Carlo			
Temporal difference			

Why can’t Monte Carlo learn online (i.e. within episode)?

recap quiz

Dynamic programming

environment model, bootstraps

Monte Carlo

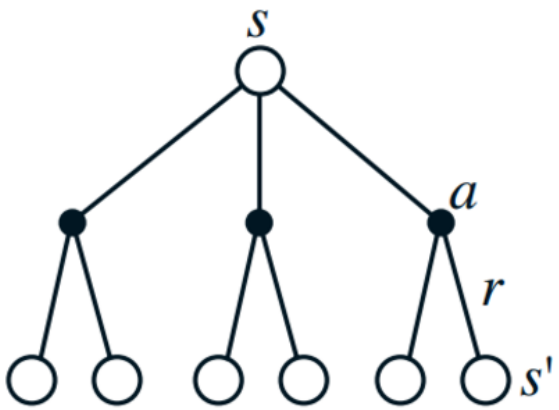
model free, samples actual returns, episodic only

Temporal difference

model free, bootstraps, samples actual experience

recap

Dynamic Programming



Monte Carlo

Temporal Difference

Sutton & Barto

three – value functions

policy improvement & control

Q-Learning

DeepMind Atari work

Playing Atari with Deep Reinforcement Learning

Volodymyr Mnih **Koray Kavukcuoglu** **David Silver** **Alex Graves** **Ioannis Antonoglou**

Daan Wierstra **Martin Riedmiller**

DeepMind Technologies

{vlad,koray,david,alex.graves,ioannis,daan,martin.riedmiller} @ deepmind.com

19 Dec 2013

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

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

Q-Learning

Popular modern method for reinforcement learning

first introduced in 1988/89 (Watkins)

Historically was unstable with non-linear function approximators

DeepMind used experience replay & target networks to improve learning stability

Why Q-Learning?

$V(s)$ vs $Q(s,a)$

Both model future expected discounted return

Which is more useful?

$V(s)$ tells us how good a state is

doesn't tell us how to get there

$Q(s,a)$ tells us how good an action is

SARSA & Q-Learning

Both temporal difference methods

SARSA

on-policy – policy approximation

experienced state, action, reward, next state, next action

Here we take our Q estimate for observed s', a'

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

SARSA & Q-Learning

Q-Learning

off policy – policy improvement / control

experienced state, action, reward, next state

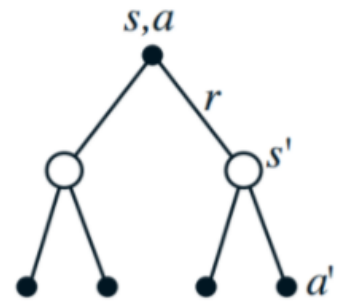
max across all actions from next state

Here we take the max Q for our observed s',

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

SARSA & Q-Learning

$q_\pi(a,s)$



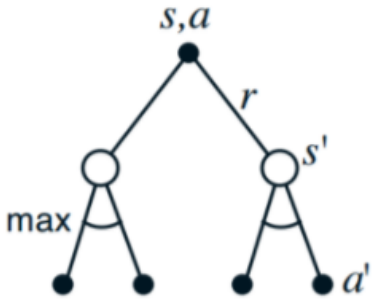
Q-policy evaluation

Sarsa

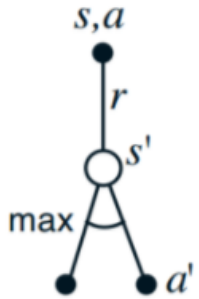
On policy

Policy evaluation

$q_*(a,s)$



Q-value iteration



Q-learning

Off policy

Policy improvement

Sutton & Barto

Q-Learning

Discrete action spaces only

argmax across all possible actions

Deterministic policy

we always pick the best action

How do we get exploration in Q-Learning?

ε-greedy exploration

$argmax[Q(s,a)]$ is deterministic

Use ε-greedy exploration to explore state space

```
def e_greedy_policy(state):
    if np.random.rand() < epsilon:
        #  act completely randomly
        action = np.random.uniform(action_space)
    else:
        #  act according to Q(s,a)
        action = greedy_policy(state)
    return action
```

ε decayed as learning improves greedy action selection

Problems with vanilla Q-Learning

Can be hard to get working

correlations in the data (state sequences)

small updates to weights may change the policy drastically

$$r + \gamma \max_{a'} Q(s', a'; \theta_i) - Q(s, a; \theta_i)$$

target **network output**

But if it works, works really well

sample efficient

Deadly triad

1 - Off policy learning

to free behaviour policy from target policy

2 - Function approximation

scalability & generalization

3 - Bootstrapping

computational & data efficiency (trade variance for bias)

Combine to produce instability & divergence

Sutton & Barto

Deadly triad

Unclear which is the root cause for instability

- dynamic programming can diverge with function approximation
- prediction can diverge (so it's not exploration or control)
- linear functions can be unstable (not non-linear functions)

It's the combination of the three

DeepMind Atari work

DQN = Deep Q-Network

Mnih et. al (2013) Playing Atari with Deep Reinforcement Learning

Mnih et. al (2015) Human-level control through deep reinforcement learning

Two key techniques to improve stability

1 – experience replay (Lin 1993)

2 – target network

Significance

End to end deep reinforcement learning

- learning from high dimensional input
- raw pixels & convolutional neural network

Ability to generalize

- same algorithm, network structure and hyperparameters
- intentionally sacrifice performance for ability to generalize

Reinforcement learning to play Atari

State = last four screens

allows information about movement

grey scale, cropped

Reward = game score

clipped to [-1,1]

Actions = joystick buttons (a discrete action space)

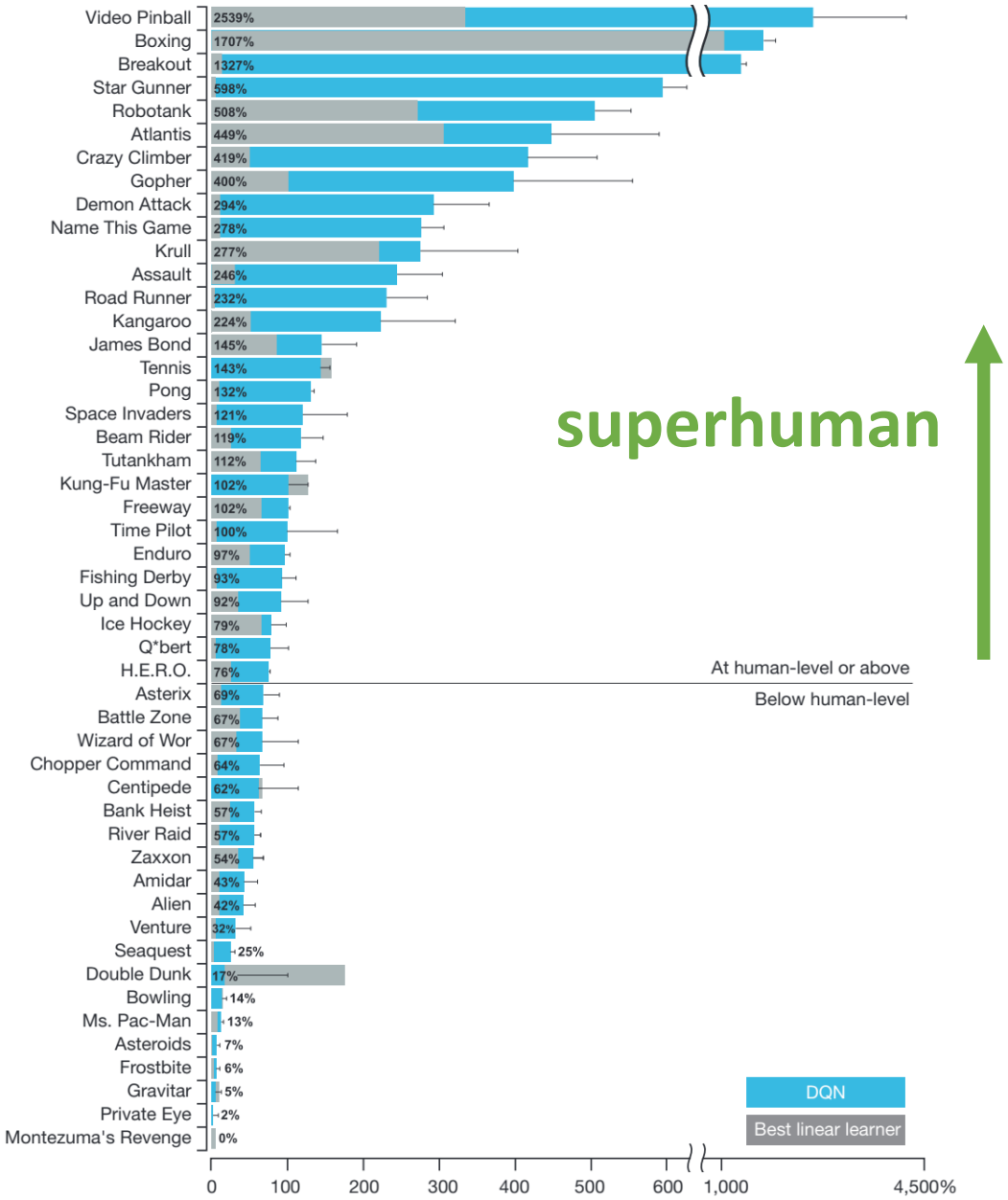
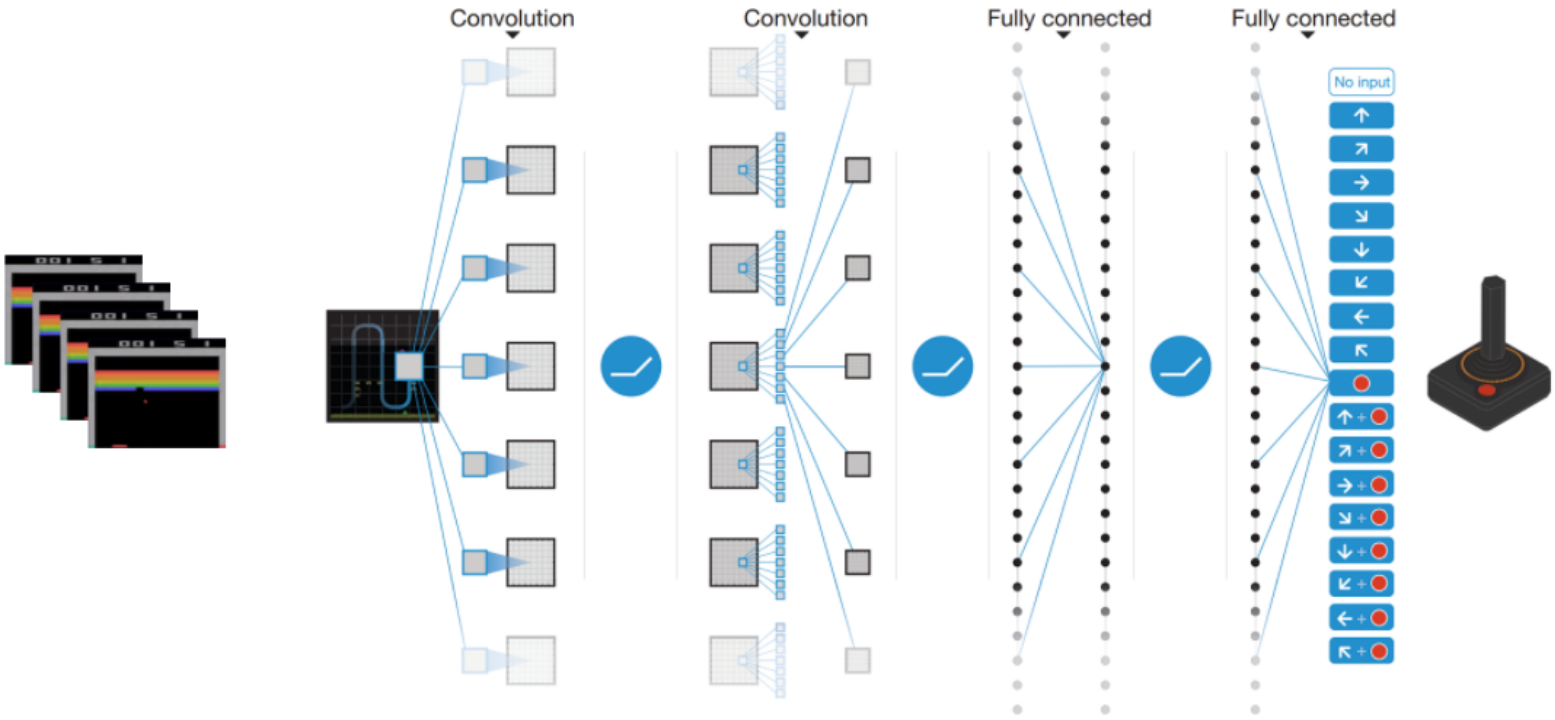


Figure 3 | Comparison of the DQN agent with the best reinforcement learning methods[15] in the literature. The performance of DQN is normalized with respect to a professional human games tester (that is, 100% level) and random play (that is, 0% level). Note that the normalized performance of DQN, expressed as a percentage, is calculated as: 100 × (DQN score − random play score)/(human score − random play score). It can be seen that DQN outperforms competing methods (also see Extended Data Table 2) in almost all the games, and performs at a level that is broadly comparable with or superior to a professional human games tester (that is, operationalized as a level of 75% or above) in the majority of games. Audio output was disabled for both human players and agents. Error bars indicate s.d. across the 30 evaluation episodes, starting with different initial conditions.

Value function approximation

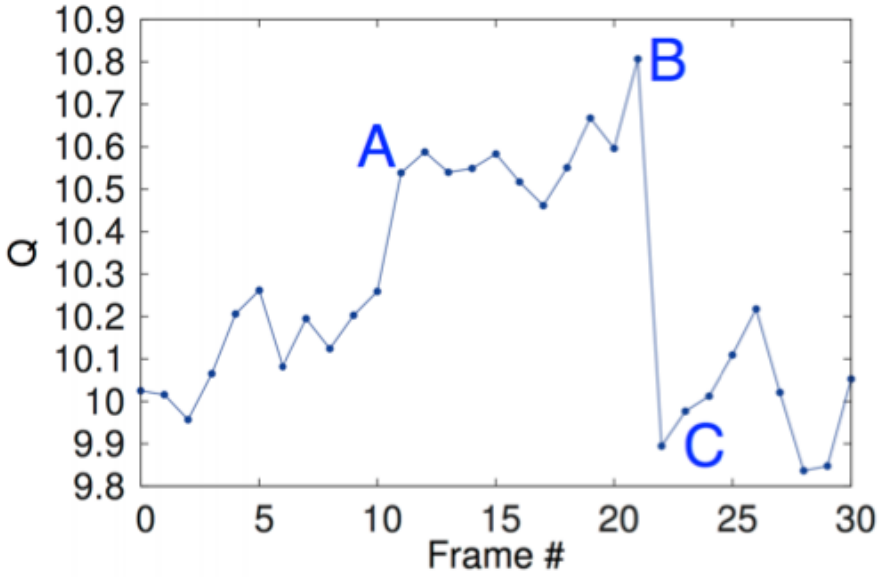


	Layer	Input	Filters	Filter size	Stride	Activation
1	Convolution	84x84x4	16 *(32)*	8×8	4	ReLU
2	Convolution	20x20x32	32 *(64)*	4×4	2	ReLU
3	*Convolution*	*64*	*64*	*3x3*	*1*	*ReLU*
4	Fully connected	256 *(512)*				Linear

2013 – Playing Atari with Deep Reinforcement learning

2015 – Human-level control through deep Reinforcement learning

One forward pass to estimate $Q(s, a)$ for all a



Agent sees new enemy

Value function gets excited

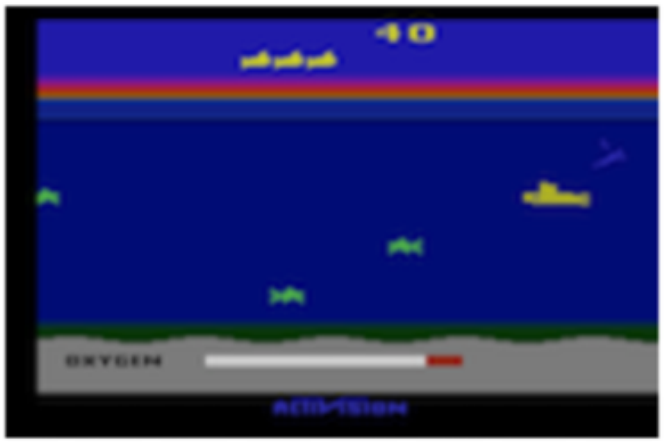
Agent has fired torpedo

Value function is very excited

Reward received

Value function back to normal

A



B

C

Experience replay

Replay buffer

Sample uniform random

$[(s_0, a_0, r_1, s'_1),$
$(s_1, a_1, r_2, s'_2),$
$(s_2, a_2, r_3, s'_3),$
$\ldots.$
$(s_n, a_n, r_{n+1}, s'_{n+1})]$

Memory of last 1 million frames

Training targets

$[Q(s_0, a_0) = r_1 + \gamma \max_a Q(s_1, a),$
$\ldots,$
$Q(s_2, a_2) = r_3 + \gamma \max_a Q(s_3, a)]$

Trained using batches of 32-64 random samples

Experience replay

Averaging of behavior distribution

more independent sampling

avoids the within trajectory correlation of s to s'

breaking correlations in data

Data efficiency

learn from experience multiple times

Allows seeding with human expert experience

Biological inspiration

Hippocampus may support an experience replay process in the brain

Time compressed reactivation of recently experienced trajectories during offline periods

Provides a mechanism where value functions can be efficiently updated through interactions with the basal ganglia

Minh et. al (2015)

PRIORITIZED EXPERIENCE REPLAY

Tom Schaul, John Quan, Ioannis Antonoglou and David Silver
Google DeepMind
{schaul,johnquan,ioannisa,davidsilver}@google.com

Prioritize training on experiences with high temporal difference error

Balance between greedy replay (on TD error) vs uniform random (like vanilla experience replay)

Faster learning – and also less i.i.d learning

Target network

Second innovation behind DQN

Use a separate Q network to create the target

parameterizing with two sets of weights $\boldsymbol{\theta}$ & $\boldsymbol{\theta_i^-}$

$$L_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}\left[r + \gamma \max_{a'} Q(s', a'; \boldsymbol{\theta_i^-}) - Q(s, a; \boldsymbol{\theta_i})^2\right]$$

target **approx.**

Update target network with new weights every 10k – 100k steps

Target network

Changing value of one action changes value of all actions & similar states

bigger networks less prone (less aliasing)

Stable training

no longer bootstrapping from the same function, but from an old & fixed version of $Q(s, a)$

Reduces the correlation between target and network output

Stability techniques

Game	With replay, with target Q	With replay, without target Q	Without replay, with target Q	Without replay, without target Q
Breakout	316.8 **100x**	240.7	10.2	3.2
Enduro	1006.3 **1000x**	831.4	141.9	29.1
River Raid	7446.6 **5x**	4102.8	2867.7	1453.0
Seaquest	2894.4 **10x**	822.6	1003.0	275.8
Space Invaders	1088.9 **3x**	826.3	373.2	302.0

Minh – Deep Q-Networks – Deep RL Bootcamp 2017

DQN Algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

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

For episode = 1, M **do**

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

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

With probability ε select a random action a_t
otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$ — **ε-greedy action selection**

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

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

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

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

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

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

Every C steps reset $\hat{Q} = Q$ — **target network update**

End For

End For

Minh et. al (2015)

Curse of dimensionality

Hits us twice in Q-Learning

Action selection

need to calculate $Q(s, a)$ for each state action

select action with by $argmax[Q(s, a)]$ for all possible a

Training

need to calculate $Q(s', a)$ for each action

to calculate max value of the next state s'

Reward clipping

DQN clips rewards to [-1, +1]

Keeps $Q(s, a)$ in a reasonable range

helps keep gradients under control

Lose information

now we are essentially reward counting

Atari work timeline

2013DQN – target network, experience replay

2015 Prioritized experience replay

2016 Double DQN (DDQN)

2016 Asynchronous Advantage Actor-Critic (A3C)

2017 Distributional Q-Learning

Maximization bias

argmax allows Q-Learning to learn off-policy

to learn the optimal policy while following a sub-optimal policy

But the maximum is

- aggressive
- positively biased

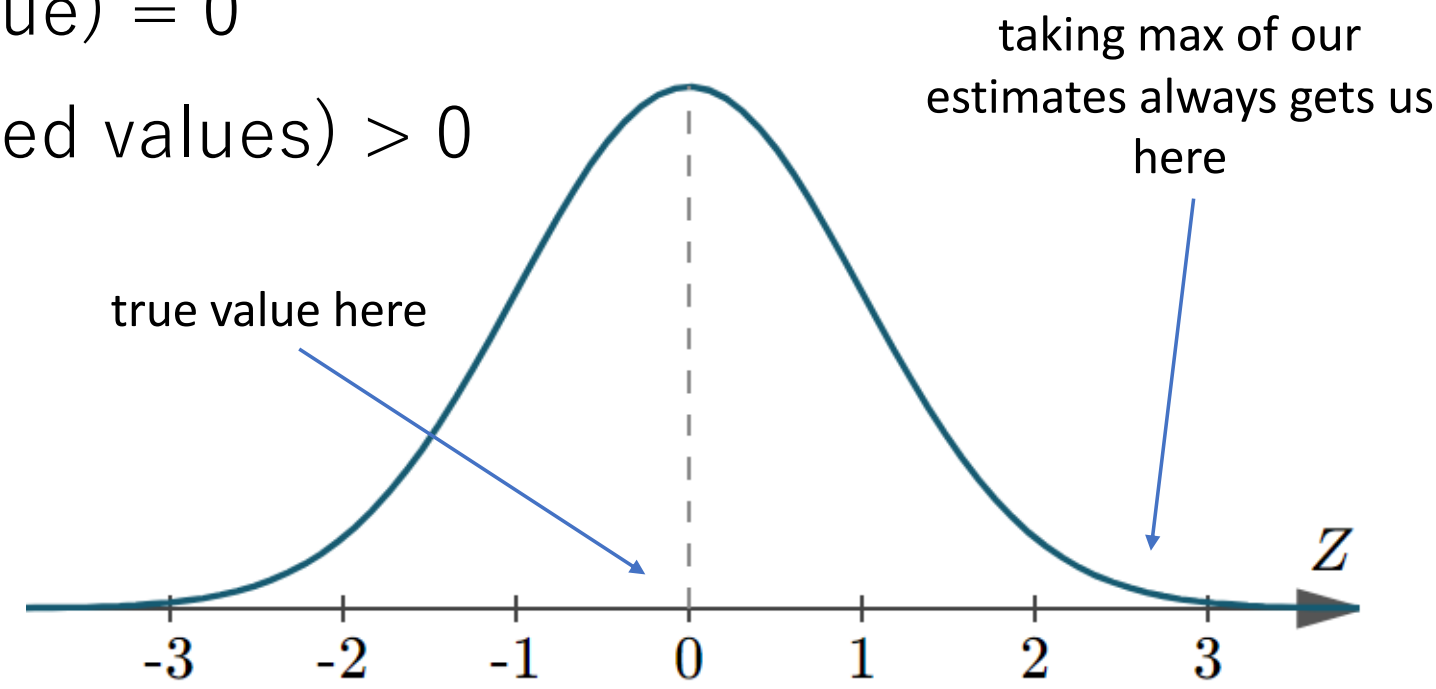
Maximization bias

State where $Q(s, a) = 0$ (for all possible a)

our estimates are distributed above & below 0

max(true value) = 0

max(estimated values) > 0



Double Q-Learning

Algorithm 1 Double Q-learning

Initialize Q^A,Q^B,s
repeat
  Choose a, based on $Q^A(s,\cdot)$ and $Q^B(s,\cdot)$, observe r, s'
  Choose (e.g. random) either UPDATE(A) or UPDATE(B)
  if UPDATE(A) **then**
    Define $a^* = \arg\max_a Q^A(s',a)$
    $Q^A(s,a) \leftarrow Q^A(s,a) + \alpha(s,a)\left(r + \gamma Q^B(s',a^*) - Q^A(s,a)\right)$
  else if UPDATE(B) **then**
    Define $b^* = \arg\max_a Q^B(s',a)$
    $Q^B(s,a) \leftarrow Q^B(s,a) + \alpha(s,a)(r + \gamma Q^A(s',b^*) - Q^B(s,a))$
  end if
  $s \leftarrow s'$
until end

Policy based on both Q

Select a Q to update

Maintain **two** parameterized Q functions (A & B)

avg(A, B) to select actions

improve A by bootstrapping off B (or vice versa)

Vlad Mnih - 2017 Deep RL bootcamp

Emphasis on experimentation / grid searching

Use Huber loss instead of squared loss on Bellman error

Use RMSProp instead of vanilla SGD

Make sure your terminal state target = reward!

Change network structure to predict V(s), A(s,a) and calculate Q(s,a)

Better than just predicting Q(s,a) !

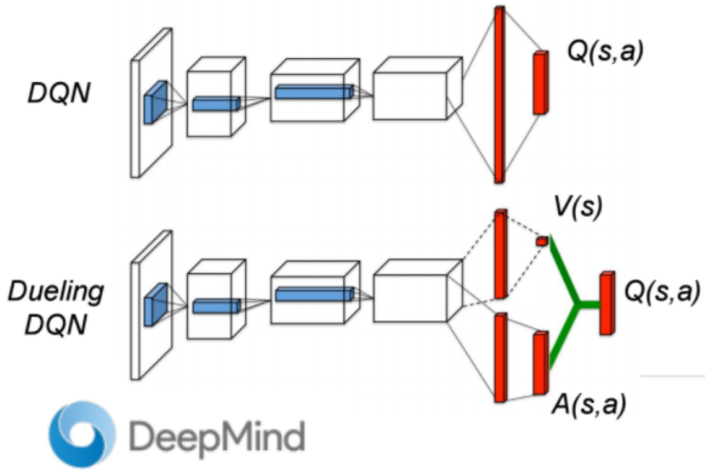
Dueling DQN

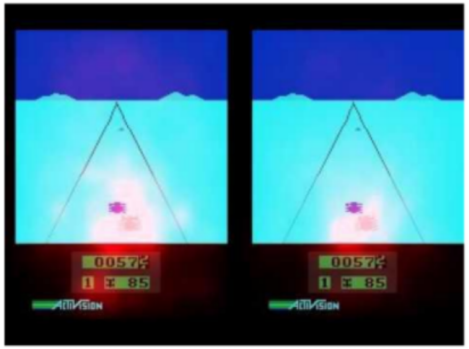
- Value-Advantage decomposition of Q:

$$Q^{\pi}(s,a) = V^{\pi}(s) + A^{\pi}(s,a)$$

- Dueling DQN (Wang et al., 2015):

$$Q(s,a) = V(s) + A(s,a) - \frac{1}{|\mathcal{A}|}\sum_{a=1}^{|\mathcal{A}|} A(s,a)$$





Atari Results

	30 no-ops		Human Starts	
	Mean	Median	Mean	Median
Prior. Duel Clip	**591.9%**	**172.1%**	**567.0%**	**115.3%**
Prior. Single	434.6%	123.7%	386.7%	112.9%
Duel Clip	**373.1%**	**151.5%**	**343.8%**	**117.1%**
Single Clip	341.2%	132.6%	302.8%	114.1%
Single	307.3%	117.8%	332.9%	110.9%
Nature DQN	227.9%	79.1%	219.6%	68.5%

"Dueling Network Architectures for Deep Reinforcement Learning", Wang et al. (2016)

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

Action-value fctn = value fctn + advantage fctn

break (lunch hopefully!)

recap quiz

Why is Q-Learning off policy?

Why discrete action spaces only?

What is the deadly triad?

Why use experience replay?

Why use a target network?

recap – Q-Learning

Off policy control - discrete action spaces only

Maximization bias - addressed by Double Q-Learning

Deadly triad – off policy, non-linear functions, bootstrapping

Experience replay – more iid, data efficiency

Target network – reduces correlation between target & network

four – policy gradients

motivations

Reinforcement learning

Value functions

learn a value function

use value function to act

$V_\pi(s, \theta)$

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

Policy gradients

learn policy

use policy to act

$\pi(s, \theta)$

Actor-Critic

Parameterize both a value function & policy

$V_{\pi(s, \theta)}$

$Q_{\pi(s, a, \theta)}$

$\pi(s, \theta)$

Policy gradients

Previously looked at generating a policy from a value function

argmax across the action space

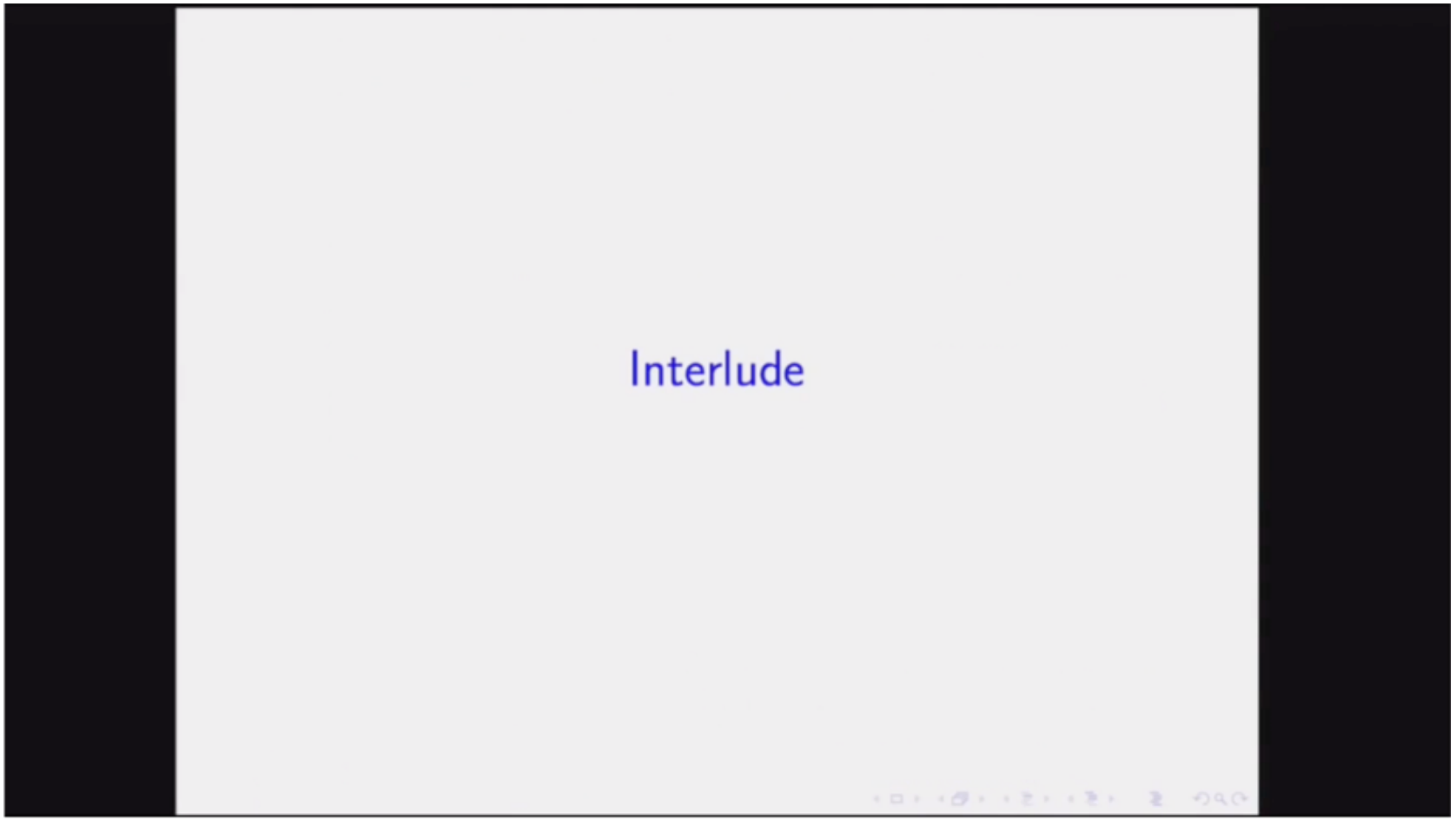
Policy gradient = parameterize a policy directly

$$\pi(a_t|s_t;\theta)$$

probability of action a_t given state s_t (stochastic)

a function parameterized by θ (i.e. neural network weights)

John Schulman – Berkley, Open AI



Motivations for policy gradients

Stochastic policies

High dimensional or continuous spaces

Optimize what we care about directly

More stable convergence

Motivation - stochastic policies

Deterministic policy (always rock) is easily exploited

Stochastic policy also gets us exploration for free

Motivation – curse of dimensionality

Q-Learning requires a discrete action space to argmax across

Lets imagine controlling a robot arm in three dimensions in the range [0, 90] degrees

This corresponds to approx. 750,000 actions a Q-Learner would need to argmax across

Discretizing continuous action spaces

In [8]:
```
#  a robot arm operating in three dimensions with a 90 degree range
single_dimension = np.arange(91)
single_dimension
```

Out[8]:
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50,
       51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67,
       68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84,
       85, 86, 87, 88, 89, 90])
```

In [32]:
```
#  we can use the combinations tool from the Python standard library
from itertools import product
all_dims = [single_dimension.tolist() for _ in range(3)]
all_actions = list(product(*all_dims))
print('num actions are {}'.format(len(all_actions)))
print('expected_num_actions are {}'.format(len(single_dimension)**3))

#  we can look at the first few combinations of actions
all_actions[0:10]
```

```
num actions are 753571
expected_num_actions are 753571
```

Out[32]:
```
[(0, 0, 0),
 (0, 0, 1),
 (0, 0, 2),
 (0, 0, 3),
 (0, 0, 4),
 (0, 0, 5),
 (0, 0, 6),
 (0, 0, 7),
 (0, 0, 8),
 (0, 0, 9)]
```

In [33]:
```
#  and the last few
all_actions[-10:]
```

Out[33]:
```
[(90, 90, 81),
 (90, 90, 82),
 (90, 90, 83),
 (90, 90, 84),
 (90, 90, 85),
 (90, 90, 86),
 (90, 90, 87),
 (90, 90, 88),
 (90, 90, 89),
 (90, 90, 90)]
```

Motivation – optimize return directly

Value function methods optimize the value function accuracy

who cares about accurate value functions?

gradients point in the direction of value function accuracy

this can be quite a complex problem to solve

Motivation – optimize return directly

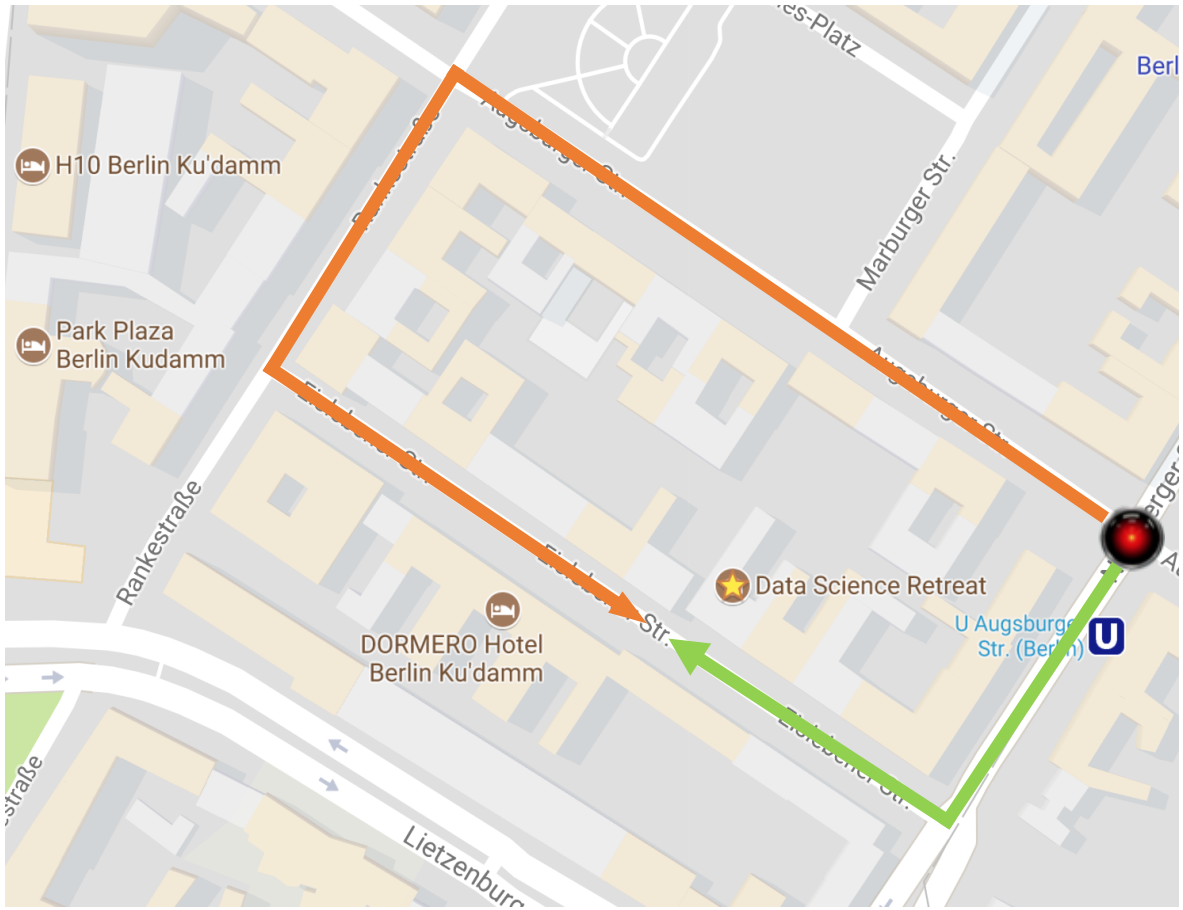
Policy methods optimize return directly

changing weights according to the gradient that maximizes future reward

aligning gradients with our objective (and hopefully a business objective)

can be easier to just select an action – rather than quantify return

Motivations for policy gradients



$Q(s, a_1) = 10 \text{ min}$

$Q(s, a_2) = 5 \text{ min}$

$\underset{a}{\text{argmax}} \rightarrow a_2$

vs

$a_2 \sim \pi(s)$

Often the a policy can be more compact & simpler than using a value function

Policy gradients

Motivations

- stochastic policies
- high dimensional or continuous spaces
- optimize what we care about directly
- more stable convergence
- exploration for free

Drawbacks

- almost always converge to local optimum
- sample efficiency

four – policy gradients

motivations

introduction

score function

REINFORCE

Policy gradients without equations

We have a parameterized policy

i.e. a neural network that outputs a distribution over actions

How do we improve it?

change parameters to take actions that get more reward

change parameters to favour probable actions

Reward function is not known

but we can calculate the gradient of the expectation of reward

Policy gradients with some equations

Imagine we have a policy $\pi(a_t|s_t;\theta)$

probability distribution over actions (stochastic)

we can get the probability from our distribution

How do we improve it?

change parameters to take actions that get more reward

change parameters to favour probable actions

Return function is not known

but gradient of expected return can be calculated

$$\nabla_\theta \, \mathbb{E}[G_t] = \mathbb{E}[\nabla_\theta \log \pi(a] \, s) G_t]$$

Policy gradients

Policy gradients are an optimization problem

1 - pick (any) policy quality measure

i.e. average reward per step

2 - pick an optimization algorithm

i.e. gradient descent

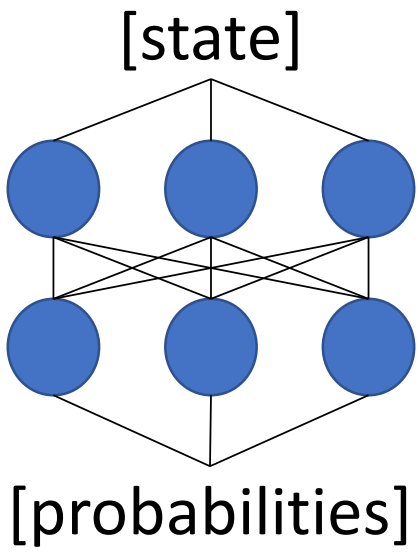
change weights to maximize policy quality

Parameterizing policies - discrete

The type of policy you parameterize depends on the action space

discrete action space

output layer = softmax

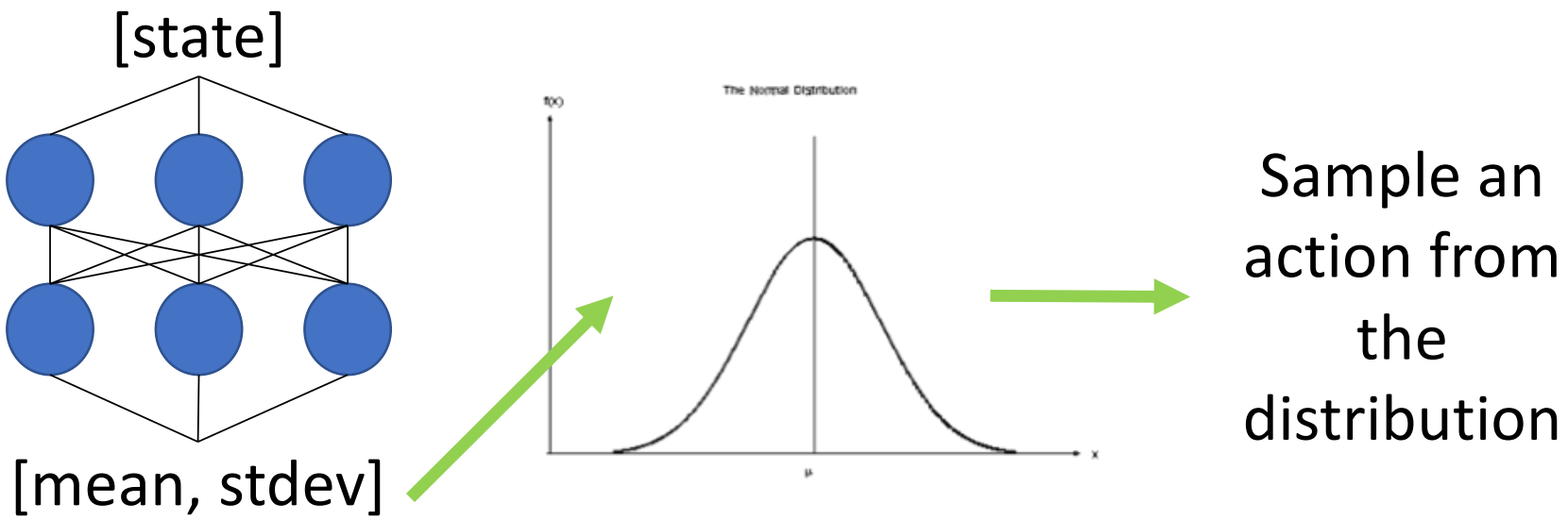


Parameterizing policies - continuous

The type of policy you parameterize depends on the **action space**

continuous action space

output layer = mean & stdev



Score function in statistics

Log-likelihood ratio trick -> score function

$$\nabla_\theta \mathbb{E}[f(x)] = \mathbb{E}[\nabla_\theta \log P(x|\theta) f(x)]$$

gradient of the expectation of *f(x)*

expectation of the gradient of the log probability of *x* * *f(x)*

Score function derivation

$$\nabla_\theta E_x[f(x)] = \nabla_\theta \sum_x p(x) f(x) \qquad \text{definition of expectation}$$

$$= \sum_x \nabla_\theta p(x) f(x) \qquad \text{swap sum and gradient}$$

$$= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) \qquad \text{both multiply and divide by } p(x)$$

$$= \sum_x p(x) \nabla_\theta \log p(x) f(x) \qquad \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z$$

$$= E_x[f(x) \nabla_\theta \log p(x)] \qquad \text{definition of expectation}$$

http://karpathy.github.io/2016/05/31/rl/

Score function in reinforcement learning

Log-likelihood ratio trick -> score function

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\left[\nabla_\theta \log \pi(a_t|s_t;\theta) G_t\right]$$

gradient of return wrt policy parameters

expectation for policy of the gradient of the policy wrt policy parameters

*

return

Training a policy

Gradient descent on expected return of the policy

Log-likelihood ratio trick -> score function

$$\nabla_\theta \mathbb{E}[f(x)] = \mathbb{E}[\nabla_\theta \, log \, P(x|\,\theta) f(x)]$$

$$\nabla_\theta J(\theta) = \mathbb{E}_{\pi_\theta}\,[\nabla_\theta \log \pi(a_t|s_t;\theta) G_t]$$

log(probability of action) * return

log(policy) * return

gradient

=

log(probability of action) * return

=

log(policy) * return

Policy gradient intuition

$$\nabla_\theta \log \pi(a_t|s_t;\theta) G_t$$

How probable was the action we picked $\log \pi(a_t|s_t;\theta)$

want to reinforce actions we thought would be good

How good was that action G_t

reinforce actions that were actually good

Policy Gradient Theorem

$$\nabla_\theta J(\theta) = \mathbb{E}_\pi \left[\nabla_\theta \log[\pi_\theta(s,a)] \cdot Q^{\pi_\theta}(s,a)\right]$$

Expectation for policy π

Gradient of the log of the policy

Discounted return for policy π from state s, taking action a

Note – θ = function parameters:

i.e. π_θ = policy π parameterized by weights θ

$$Q^{\pi_\theta} \; - not \; Q^* \; !$$

Improving a policy

$$\nabla_\theta \log \pi(a_t|s_t;\theta) G_t$$

Score function constrains policy gradients to on-policy learning only

need to calculate the log probability of the action for the policy that took that action

Score function allows us to take expectations

because we can sample from our policy

Approximating return

We can use different methods to approximate the quality of an action

Monte Carlo REINFORCE = use actual returns

$$\nabla_\theta \log \pi(a_t|s_t;\theta)\boldsymbol{G_t}$$

Why might using a Monte Carlo return be a problem?

Problems with MC REINFORCE

Monte Carlo is high variance

Monte Carlo requires episodic problems

Monte Carlo requires lots of experience = slow to converge

How can we get some of the advantages of Temporal Difference methods?

Baseline

Monte Carlo REINFORCE = use actual returns

$$\nabla_\theta \log \pi(a_t|s_t;\theta)\boldsymbol{G_t}$$

We can introduce a baseline function

reduces variance without changing the expectation

REINFORCE with a baseline

$$\nabla_\theta \log \pi(a_t|s_t;\theta)(\boldsymbol{G_t - B(s_t)})$$

Approximating return

REINFORCE with a baseline

reduces variance without introducing bias

$$\nabla_\theta \log \pi(a_t|s_t;\theta)(\boldsymbol{G_t - B(s_t)})$$

A natural baseline is the value function

$$\nabla_\theta \log \pi(a_t|s_t;\theta)(\boldsymbol{G_t - V(s_t;w)})$$

Gives rise to the concept of the **advantage function**

$$A(s_t, a_t) = G_t - V(s_t) = \boldsymbol{Q(s_t, a_t) - V(s_t)}$$

REINFORCE with baseline

Input: policy $\pi(a|s, \boldsymbol{\theta})$, $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes, $\alpha > 0$, $\beta > 0$
Output: policy $\pi(a|s, \boldsymbol{\theta})$
initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for *true* **do**
  generate an episode $s_0, a_0, r_1, \cdots, s_{T-1}, a_{T-1}, r_T$, following $\pi(\cdot|\cdot, \boldsymbol{\theta})$ **Run through entire episode following policy**
  for *each step* t *of episode* $0, \cdots, T-1$ **do**
    $G_t \leftarrow$ return from step t **calculate MC return**
    $\delta \leftarrow G_t - \hat{v}(s_t, \boldsymbol{w})$ **return - baseline**
    $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta\delta\nabla_{\boldsymbol{w}}\hat{v}(s_t, \boldsymbol{w})$ **update our baseline (parameters = *w*)**
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\gamma^t\boldsymbol{\delta}\nabla_{\boldsymbol{\theta}} log\pi(a_t|s_t, \boldsymbol{\theta})$ **update our policy (parameters = θ)**
  end
end

Algorithm 5: REINFORCE with baseline (episodic), adapted from Sutton and Barto (2017)

Note we are still constrained to episodic problems!

Li 2017

four – policy gradients

motivations

introduction

score function

REINFORCE

Actor-Critic

Reinforcement learning

Value functions

Parameterize a value function

$V_\pi(s, \theta)$

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

Policy gradients

Parameterize a policy

$\pi(s, \theta)$

Actor-Critic

Parameterize both a value function & policy

$V_\pi(s, \theta)$

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

$\pi(s, \theta)$

Actor-Critic

Parameterize

actor = policy

critic = value function

Actor updates the policy parameters in the direction suggested by the critic

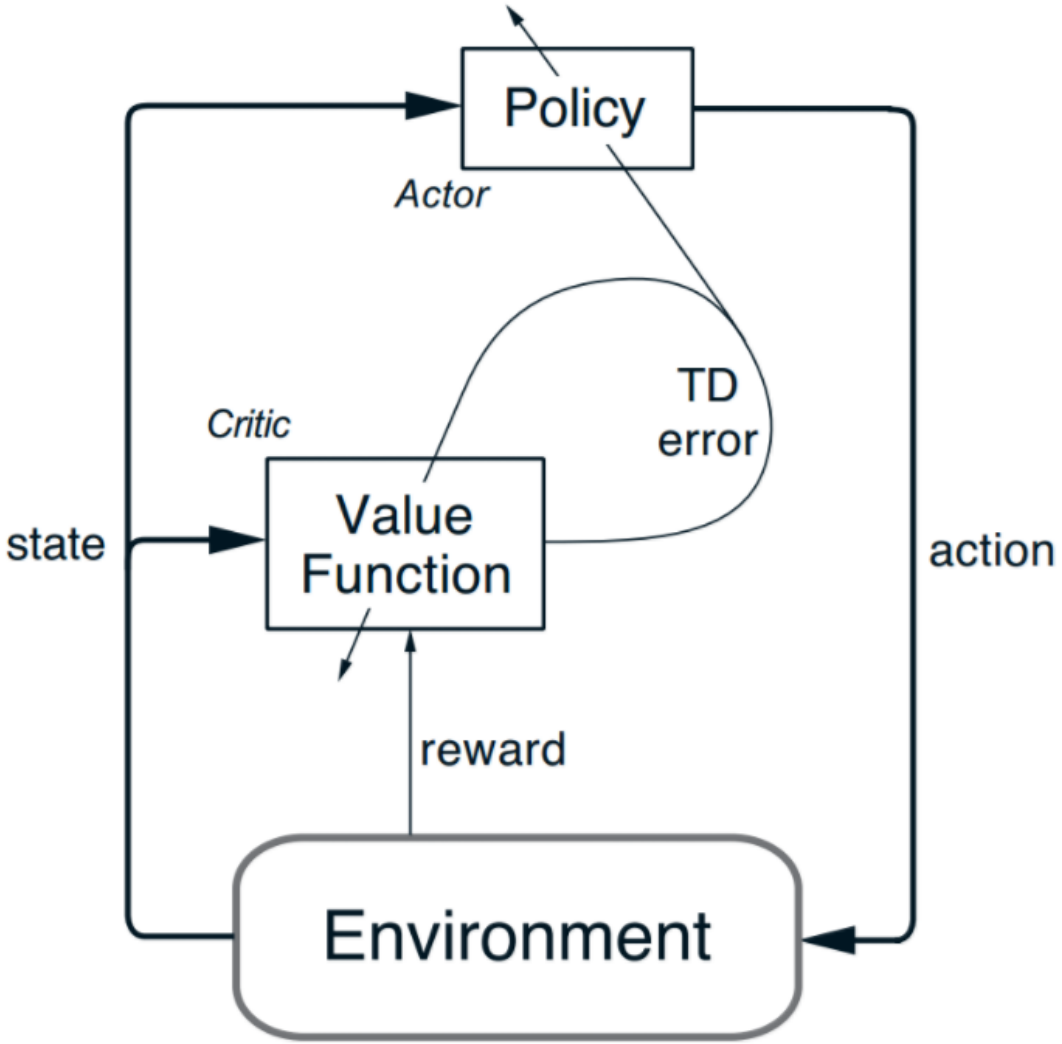


Figure 11.1: The actor–critic architecture.

Sutton & Barto

Actor-Critic algorithm

Input: policy $\pi(a|s, \boldsymbol{\theta})$, $\hat{v}(s, \boldsymbol{w})$
Parameters: step sizes, $\alpha > 0$, $\beta > 0$
Output: policy $\pi(a|s, \boldsymbol{\theta})$
initialize policy parameter $\boldsymbol{\theta}$ and state-value weights $\boldsymbol{w}$
for *true* **do**
    initialize s, the first state of the episode
    $I \leftarrow 1$
    for s *is not terminal* **do**
        $a \sim \pi(\cdot|s, \boldsymbol{\theta})$
        take action a, observe s', r
        $\delta \leftarrow r + \gamma \hat{v}(s', \boldsymbol{w}) - \hat{v}(s, \boldsymbol{w})$ (if s' is terminal, $\hat{v}(s', \boldsymbol{w}) \doteq 0$)
        $\boldsymbol{w} \leftarrow \boldsymbol{w} + \beta \delta \nabla_{\boldsymbol{w}} \hat{v}(s_t, \boldsymbol{w})$
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha I \delta \nabla_{\boldsymbol{\theta}} log \pi(a_t|s_t, \boldsymbol{\theta})$
        $I \leftarrow \gamma I$
        $s \leftarrow s'$
    end
end

within episode updating

TD error

update value function (critic)

update policy (actor)

Algorithm 6: Actor-Critic (episodic), adapted from Sutton and Barto (2017)

Li 2017

Deterministic Policy Gradients

Deterministic Policy Gradient Algorithms

David Silver DAVID@DEEPMIND.COM
DeepMind Technologies, London, UK
Guy Lever GUY.LEVER@UCL.AC.UK
University College London, UK
Nicolas Heess, Thomas Degris, Daan Wierstra, Martin Riedmiller *@DEEPMIND.COM
DeepMind Technologies, London, UK

Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32.

DPG

Actor Critic

Deterministic -> more efficient than stochastic

Continuous action spaces

Off policy

Uses experience replay

Uses target networks

Stochastic policy gradient (review)

Policy is a probability distribution over actions

we then select action by sampling from this distribution

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

Deterministic Policy Gradient (DPG) parameterizes a deterministic policy

$$a = \mu_\theta(s)$$

DPG Components

Actor

off-policy

function that maps state to action

exploratory

Critic

on-policy (critic of the current policy)

function that estimates Q(s,a)

Gradients

Stochastic

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim p^\pi, a\sim\pi_\theta}\ \nabla_\theta \log \pi_\theta(a|s) \cdot Q^\pi(s,a)$$

Deterministic

On policy

$$\nabla_\theta J(\pi_\theta) = \mathbb{E}_{s\sim p^\pi}\ \nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\Big|_{a=\mu_\theta(s)}$$

Off policy

$$\nabla_\theta J_\beta(\pi_\theta) = \mathbb{E}_{s\sim p^\beta}\ \nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\Big|_{a=\mu_\theta(s)}$$

Updating policy weights

The gradient

$$\nabla_\theta J_\beta(\pi_\theta) = \mathbb{E}_{s\sim p^\beta}\ \nabla_\theta \mu_\theta(s)\, \nabla_a Q^\mu(s,a)\Big|_{a=\mu_\theta(s)}$$

The update function

$$\theta_{t+1} = \theta_t + \alpha\, \nabla_\theta \mu_\theta(s_t)\, \nabla_a Q^w(s_t,a_t)\Big|_{a=\mu_\theta(s)}$$

α learning rate

Q^w action value function parameterized by weights w

Results

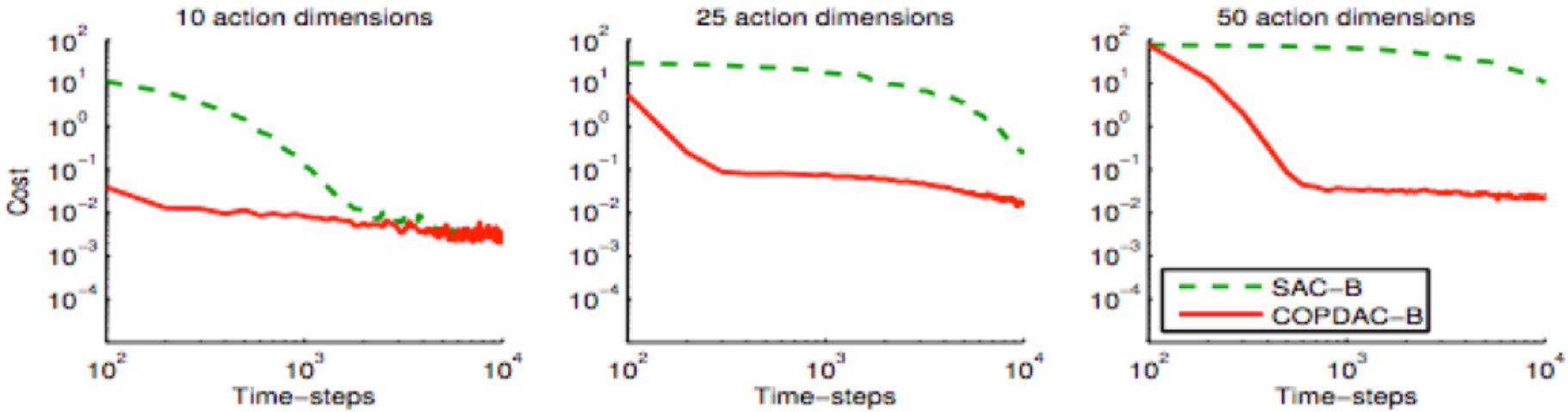


Figure 1. Comparison of stochastic actor-critic (SAC-B) and deterministic actor-critic (COPDAC-B) on the continuous bandit task.

Difference between stochastic (green) and deterministic (red) increases with the size of the action space

A3C

Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih[1] VMNIH@GOOGLE.COM
Adrià Puigdomènech Badia[1] ADRIAP@GOOGLE.COM
Mehdi Mirza[1,2] MIRZAMOM@IRO.UMONTREAL.CA
Alex Graves[1] GRAVESA@GOOGLE.COM
Tim Harley[1] THARLEY@GOOGLE.COM
Timothy P. Lillicrap[1] COUNTZERO@GOOGLE.COM
David Silver[1] DAVIDSILVER@GOOGLE.COM
Koray Kavukcuoglu [1] KORAYK@GOOGLE.COM

[1] Google DeepMind
[2] Montreal Institute for Learning Algorithms (MILA), University of Montreal

arXiv:1602.01783v2 [cs.LG] 16 Jun 2016

A3C

Asynchronous Advantage Actor-Critic

has obsoleted DQN as state of the art

works in continuous action spaces

We saw earlier that experience replay is used to reduce non-stationarity & decorrelate updates

but can only be used with off-policy learners

Asynchronous Advantage Actor-Critic

Asynchronous

multiple agents learning separately

experience of each agent is independent of other agents

learning in parallel stabilizes training

allows use of on-policy learners

runs on single multi-core CPU

learns faster than many GPU methods

Asynchronous Advantage Actor-Critic

Advantage

advantage function

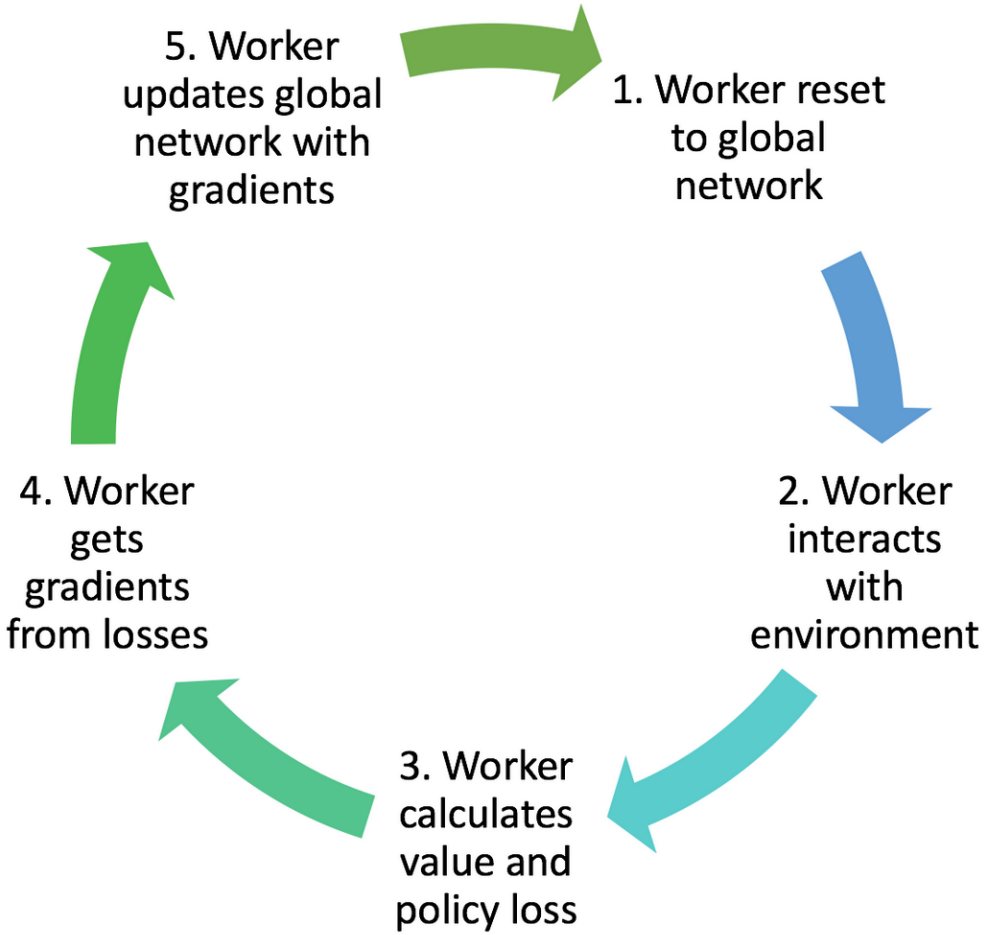
$$A(s_t, a_t) = G_t - V(s_t) = Q(s_t, a_t; \theta) - V(s_t; w)$$

Use an action-value function to approximate return

Agent can determine how much better an action is than the average action

average for the policy being followed

A3C



https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2

break

recap – policy gradients

What are the two main components of the score function?

Why are policy gradients on-policy?

What does A3C stand for?

recap – policy gradients

What are the two main components of the score function?

return and action probability

Why are policy gradients on-policy?

What does A3C stand for?

five – AlphaGo

Mastering the game of Go with deep neural networks and tree search

David Silver[1]*, Aja Huang[1]*, Chris J. Maddison[1], Arthur Guez[1], Laurent Sifre[1], George van den Driessche[1], Julian Schrittwieser[1], Ioannis Antonoglou[1], Veda Panneershelvam[1], Marc Lanctot[1], Sander Dieleman[1], Dominik Grewe[1], John Nham[2], Nal Kalchbrenner[1], Ilya Sutskever[2], Timothy Lillicrap[1], Madeleine Leach[1], Koray Kavukcuoglu[1], Thore Graepel[1] & Demis Hassabis[1]

NATURE | VOL 529 | 28 JANUARY 2016

AlphaGo

2015

first machine to beat human professional (Lee Sedol)

2017

beat no. 1 ranked player Ke Jie

Google's **Alpha Go** defeats world No. 1 Go player
동아일보 - 28 May 2017
The reddened eyes of world No. 1 **Go** player Ke Jie of nine dan rank were in tears. When the cameras took a close up of his face, he covered ...

IBM Deep Blue

First defeat of world chess champion by a machine in 1997

Deep Blue
IBM chess computer

Garry Kasparov
World Chess Champion

Deep Blue vs AlphaGo

Deep Blue was handcrafted

programmers & chess grandmasters

AlphaGo learnt

human moves & self play

AlphaGo evaluated fewer positions

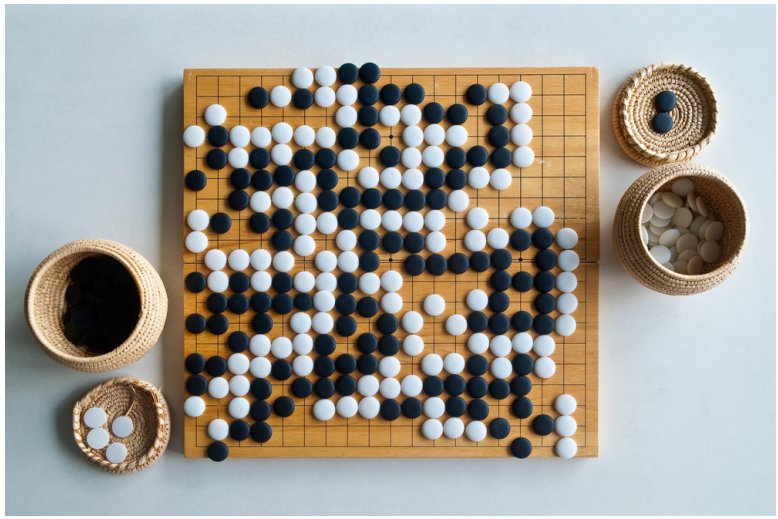
width policy network select states more intelligently

depth value function evaluate states more precisely

Why Go?

Long held as the most challenging classic game for artificial intelligence

- massive search space
- more legal positions than atoms in universe
- difficult to evaluate positions & moves
- sparse & delayed reward



AlphaGo

AlphaGo works to reduce both the width and depth of search

width by sampling high probability actions from policy

depth through position evaluation using $V(s)$

Components of the AlphaGo agent

Three policy networks $\pi(s)$

fast rollout policy network – linear function

supervised learning policy – 13 layer convolutional NN

reinforcement learning policy – 13 layer convolutional NN

One value function $V(s)$

convolutional neural network

Combined together using Monte Carlo tree search

Components of the AlphaGo agent

1 – train fast & supervised policy networks

predicting human moves

2 – train reinforcement learning policy network

initialize using supervised network weights

self play (align gradient towards winning)

3 – train value function

use data generated during self play

Learning

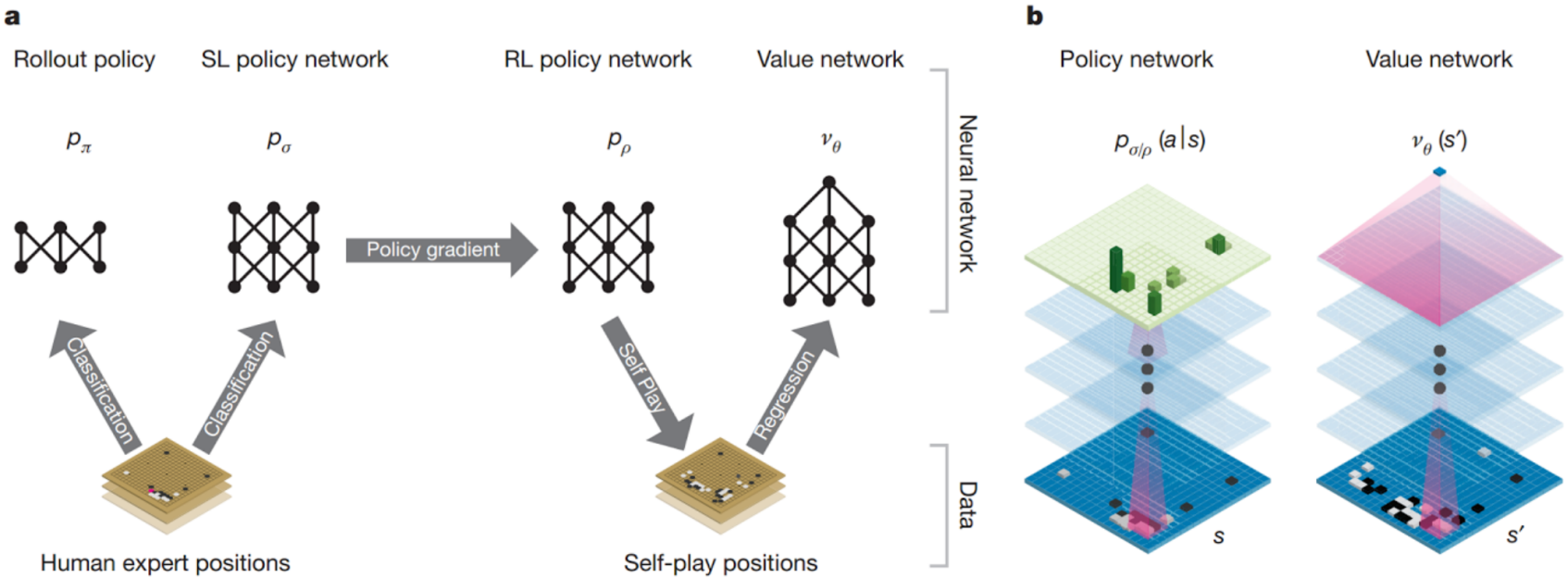


Figure 1 | Neural network training pipeline and architecture. a, A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions. A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network. Finally, a value network v_θ is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set. **b**, Schematic representation of the neural network architecture used in AlphaGo. The policy network takes a representation of the board position s as its input, passes it through many convolutional layers with parameters σ (SL policy network) or ρ (RL policy network), and outputs a probability distribution $p_\sigma(a|s)$ or $p_\rho(a|s)$ over legal moves a, represented by a probability map over the board. The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.

Monte Carlo Tree Search

Value & policy networks combined using MCTS

Basic idea = analyse most promising next moves

Planning algorithm

- simulated (not actual experience)
- roll out to end of game (a simulated Monte Carlo return)

Monte Carlo Tree Search

1 - pick a state to investigate further

using measures of state value & visit statistics

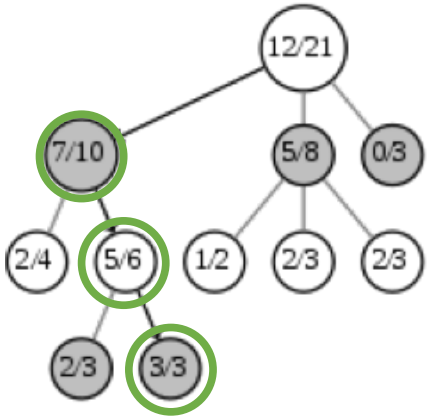
2 - rollout down from this state

use linear fast rollout policy

3 - repeat

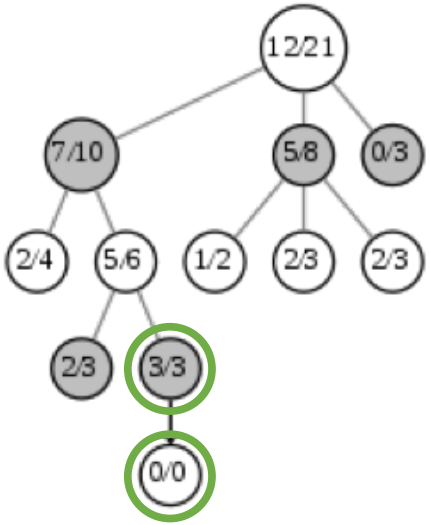
Monte Carlo Tree Search

Selection



Select nodes based on statistics collected so far

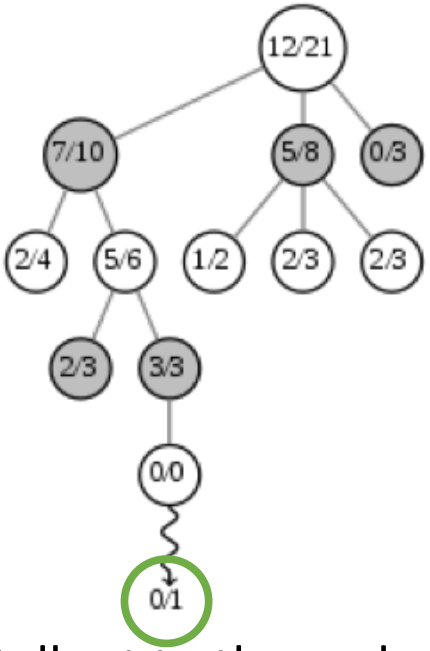
Expansion



If leaf node:

create & the select a new child node

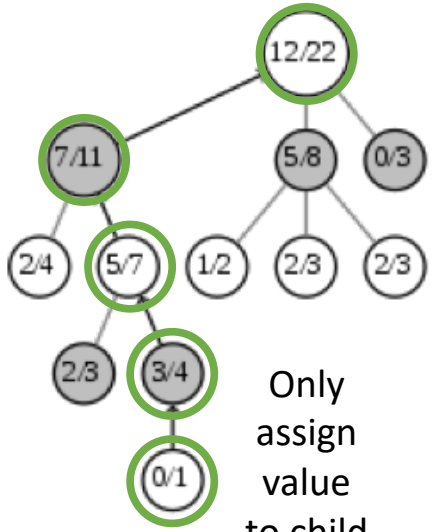
Simulation



Rollout to the end of the game

(simulated)

Backpropagation



Only assign value to child node

Backup the result through the tree to the root state

https://en.wikipedia.org/wiki/Monte_Carlo_tree_search

Monte Carlo Tree Search in AlphaGo

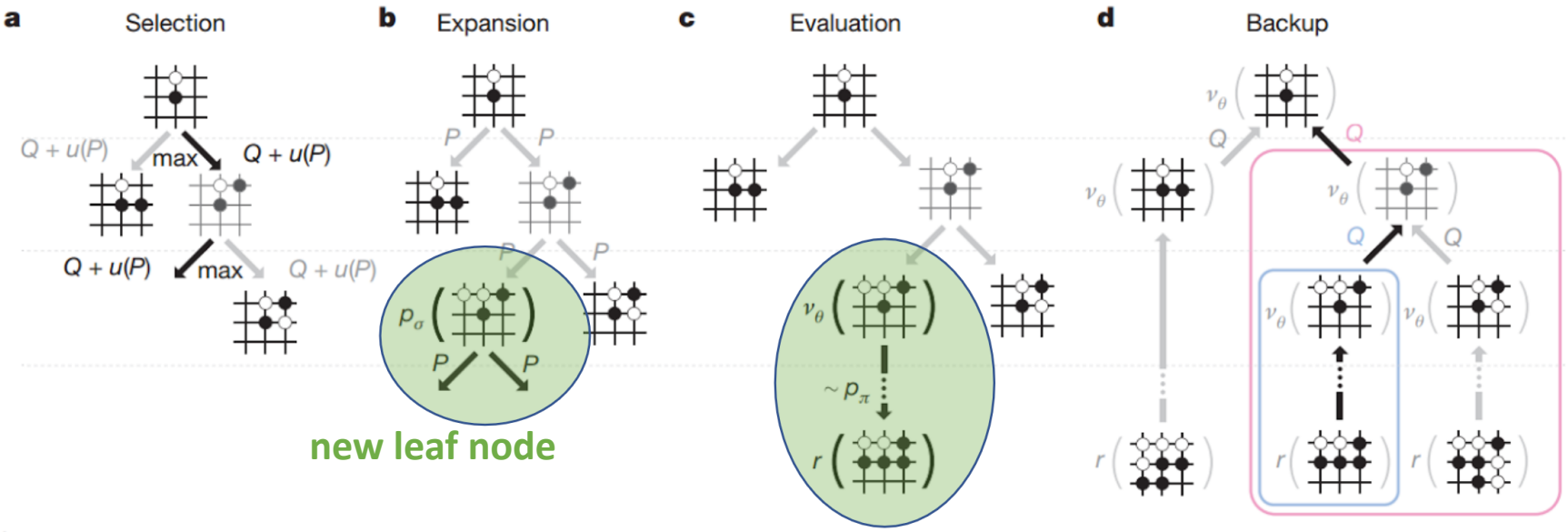
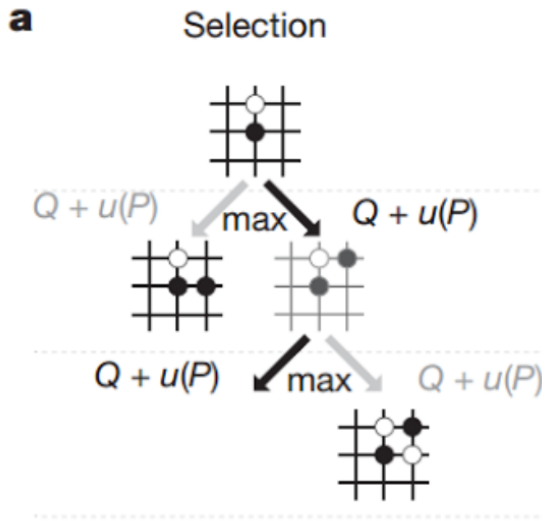


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network p_σ and the output probabilities are stored as prior probabilities P for each action. **c**, At the end of a simulation, the leaf node is evaluated in two ways: using the value network v_θ; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r. **d**, Action values Q are updated to track the mean value of all evaluations $r(\cdot)$ and $v_\theta(\cdot)$ in the subtree below that action.

MCTS in AlphaGo



Each edge (s, a) keeps track of statistics

action value	$Q(s, a)$
visit count	$N(s, a)$
prior probability	$P(s, a)$ SL policy network

Bonus $u(s, a)$ penalizes more visits to encourages exploration

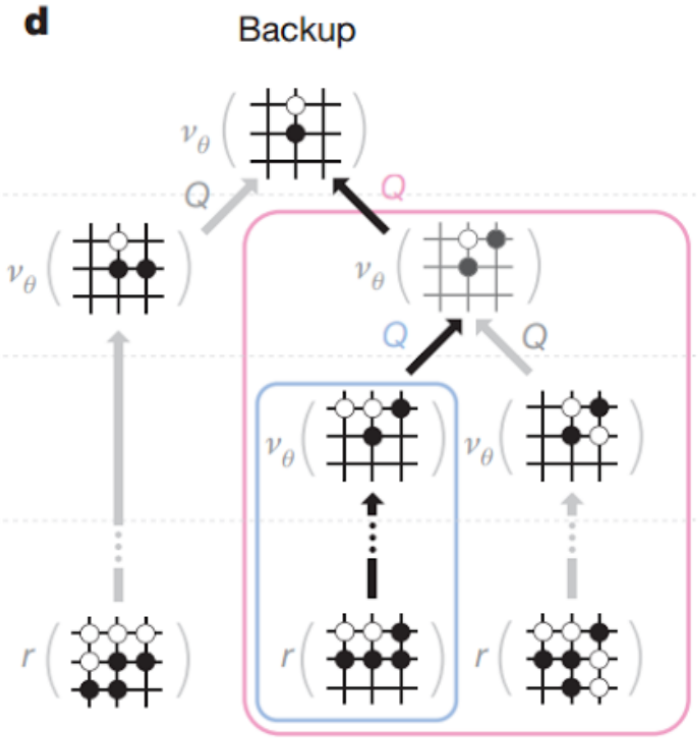
Action selected according too

$$a = \underset{a}{\text{argmax}}[Q(s, a) - u(s, a)]$$

where

$$u(s, a) \propto P(s, a)/[1 - N(s, a)]$$

Backpropagating statistics



After we finish our rollout – we calculate a state value for our leaf node s_L

$$V(s_L) = (1-\lambda)v_\theta(s) + \lambda z$$

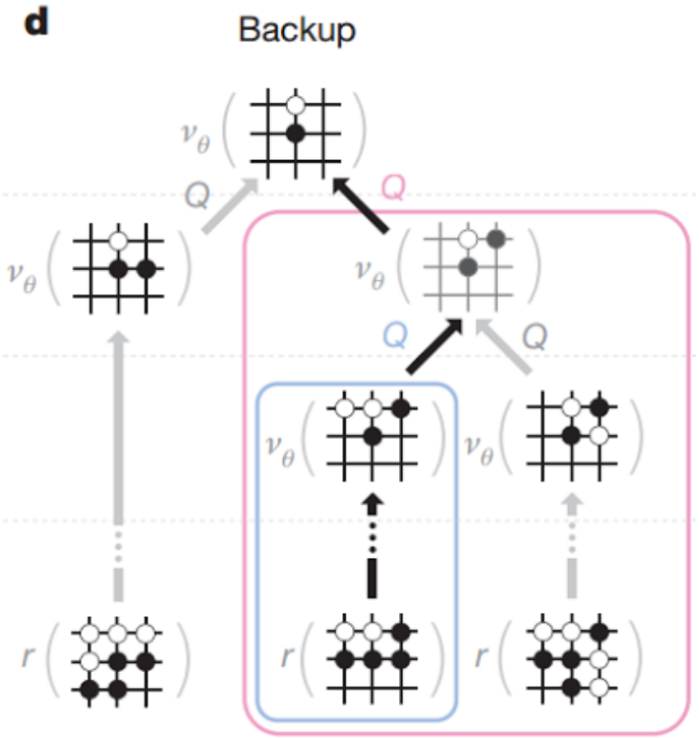
λ mixing parameter

v_θ value network estimate

z simulated result of rollout

We are combining the value network with the MCTS rollout

Backpropagating statistics



Then use our combined estimate $V(s_L)$ to update

action value $Q(s,a)$

visit count $N(s,a)$

$$Q(s,a) = \frac{1}{N(s,a)} \sum_{i=1}^{n} V(s_L^i)$$

We are averaging across all visits in the simulation

After all simulations finished - select the most visited action from the root state

AlphaGo, in context – Andrej Karpathy

Convenient properties of Go

fully deterministic

fully observed

discrete action space

access to perfect simulator

relatively short episodes

evaluation is clear

huge datasets of human play

energy consumption (human ≈ 50 W) 1080 ti = 250 W

https://medium.com/@karpathy/alphago-in-context-c47718cb95a5

AlphaGo Zero

ARTICLE

doi:10.1038/nature24270

Mastering the game of Go without human knowledge

David Silver[1]*, Julian Schrittwieser[1]*, Karen Simonyan[1]*, Ioannis Antonoglou[1], Aja Huang[1], Arthur Guez[1], Thomas Hubert[1], Lucas Baker[1], Matthew Lai[1], Adrian Bolton[1], Yutian Chen[1], Timothy Lillicrap[1], Fan Hui[1], Laurent Sifre[1], George van den Driessche[1], Thore Graepel[1] & Demis Hassabis[1]

[1]DeepMind, 5 New Street Square, London EC4A 3TW, UK.
*These authors contributed equally to this work.

AlphaGo Zero key ideas

simpler

search

adverserial

machine knowledge only

AlphaGo Zero results

Training time & performance

AG Lee trained over several months

AG Zero beat AG Lee 100-0 after 72 hours of training

Computational efficiency

AG Lee = distributed w/ 48 TPU

AG Zero = single machine w/ 4 TPU

AlphaGo Zero learning curve

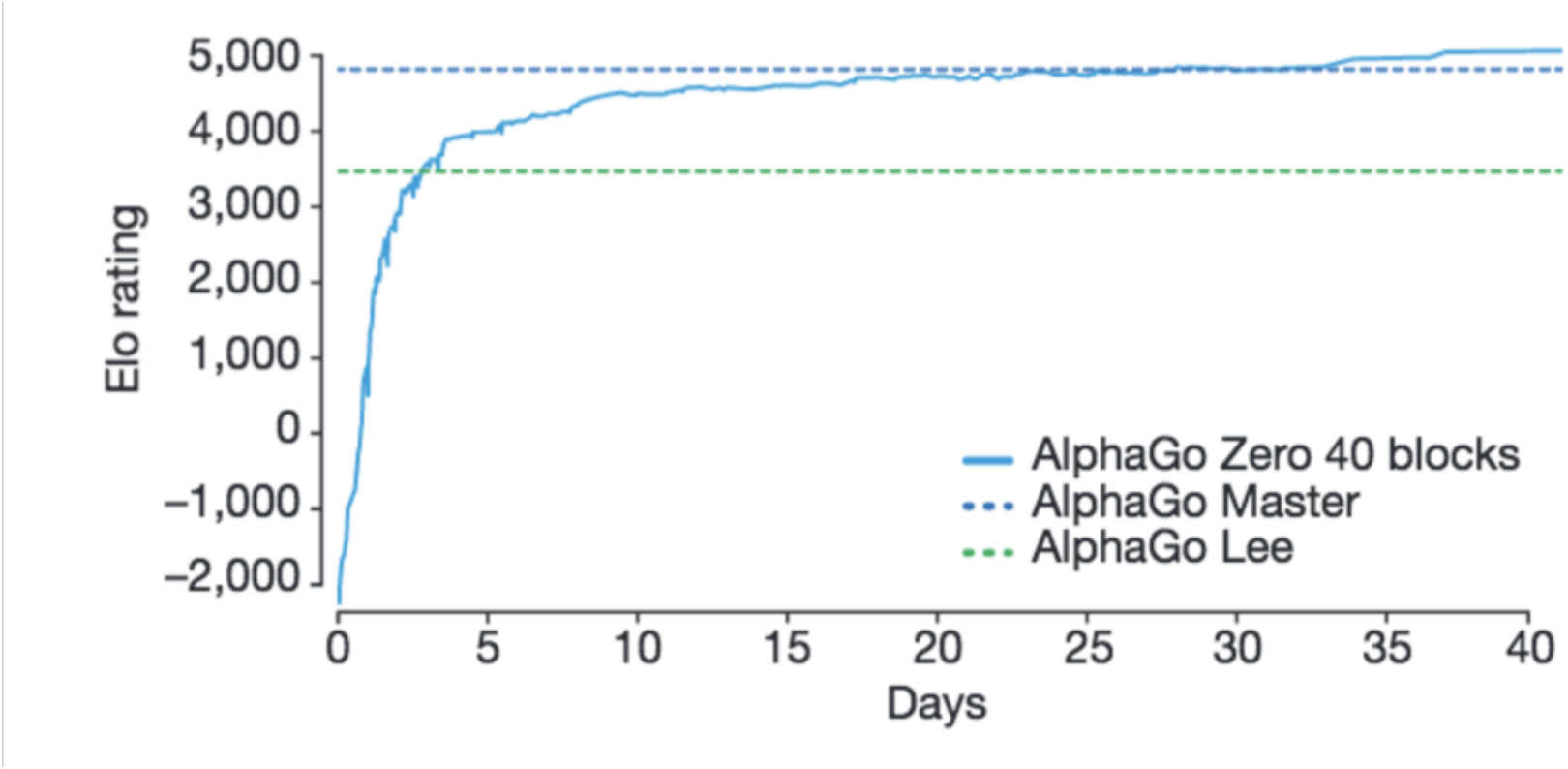


Figure 6 | Performance of AlphaGo Zero. a, Learning curve for AlphaGo Zero using a larger 40-block residual network over 40 days. The plot shows the performance of each player α_{θ_i} from each iteration i of our reinforcement learning algorithm. Elo ratings were computed from evaluation games between different players, using 0.4 s per search (see Methods)

AlphaGo Zero learning curves

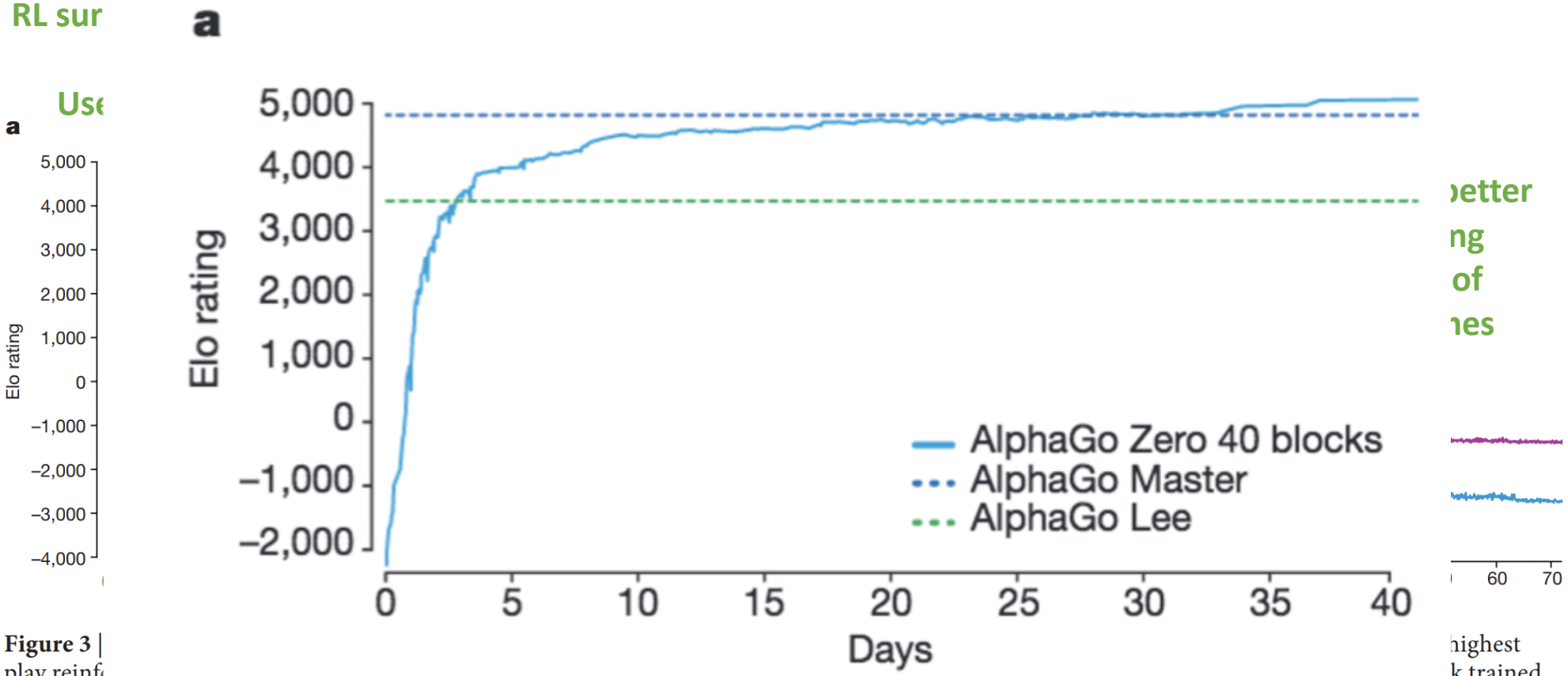


Figure 3 | play reinf MCTS pla AlphaGo Zero. Elo ratings were computed from evaluation games between different players, using 0.4 s of thinking time per move (see Methods). For comparison, a similar player trained by supervised learning from human data, using the KGS dataset, is also shown. **b**, Prediction accuracy on human professional moves. The plot shows the accuracy of the neural network f_{θ_i}, at each iteration of self-play i, in predicting human professional moves from the GoKifu dataset. The accuracy measures the

highest
k trained
by supervised learning is also shown. **c**, Mean-squared error (MSE) of human professional game outcomes. The plot shows the MSE of the neural network f_{θ_i}, at each iteration of self-play i, in predicting the outcome of human professional games from the GoKifu dataset. The MSE is between the actual outcome $z \in \{-1, +1\}$ and the neural network value v, scaled by a factor of $\frac{1}{4}$ to the range of 0–1. The MSE of a neural network trained by supervised learning is also shown.

AlphaGo Zero innovations

1 – learns using only self play

no learning from human expert games

no feature engineering

learn purely from board positions

2 – single neural network

combine the policy & value networks

3 – MCTS only during acting (not during learning)

AlphaGo Zero acting & learning

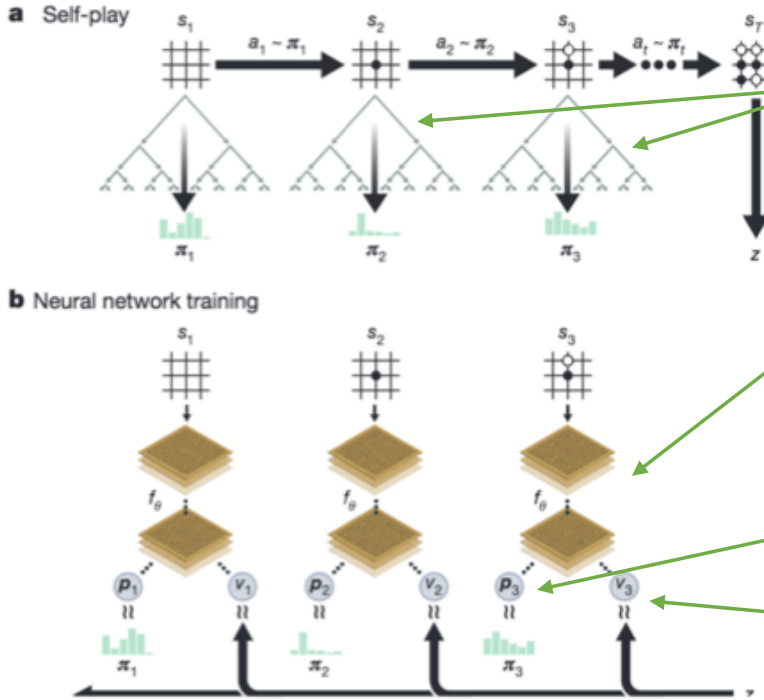


Figure 1 | Self-play reinforcement learning in AlphaGo Zero. a, The program plays a game $s_1, \ldots, s_T$ against itself. In each position s_t, an MCTS α_θ is executed (see Fig. 2) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t \sim \pi_t$. The terminal position s_T is scored according to the rules of the game to compute the game winner z. **b**, Neural network training in AlphaGo Zero. The neural network takes the raw board position s_t as its input, passes it through many convolutional layers with parameters θ, and outputs both a vector $\boldsymbol{p}_t$, representing a probability distribution over moves, and a scalar value v_t, representing the probability of the current player winning in position s_t. The neural network parameters θ are updated to maximize the similarity of the policy vector $\boldsymbol{p}_t$ to the search probabilities π_t, and to minimize the error between the predicted winner v_t and the game winner z (see equation (1)). The new parameters are used in the next iteration of self-play as in **a**.

Use MCTS during acting

Single neural network

two outputs:

1 – probability dist. over actions

2 – the game outcome

Model is trained to predict the probabilities as generated by MCTS during acting

Search

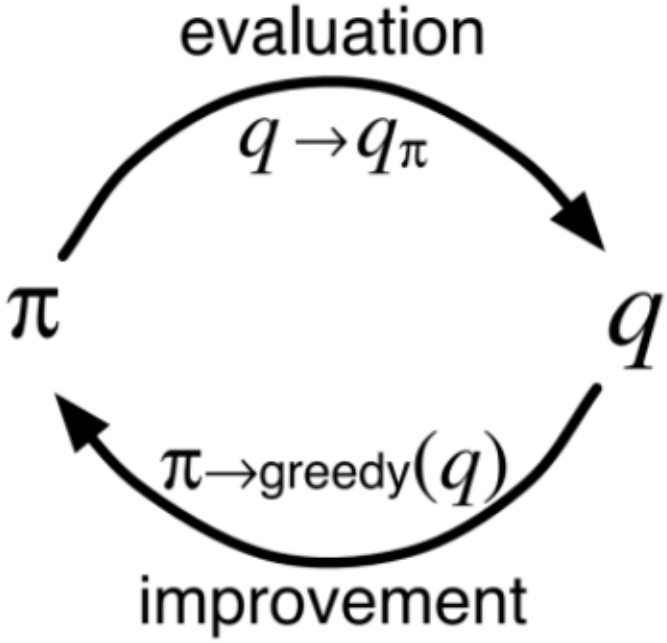
Policy evaluation

policy is evaluated through self play

this produces high quality training signals (ie the game result)

Policy improvement

MCTS search using acting to create target (ie the improved policy)



Keynote David Silver NIPS 2017 Deep Reinforcement Learning Symposium AlphaZero

Residual networks

Deep Residual Learning for Image Recognition

Kaiming He Xiangyu Zhang Shaoqing Ren Jian Sun
Microsoft Research
{kahe, v-xiangz, v-shren, jiansun}@microsoft.com

arXiv:1512.03385v1 [cs.CV] 10 Dec 2015

Convolutional network with skip connections

Layers are reformulated as residuals of the input

Residual networks

Trying to learn $\mathcal{H}(x)$

Instead of learning $\mathcal{F}(x) = \mathcal{H}(x)$

We learn the residual $\mathcal{F}(x) = \mathcal{H}(x) - x$

And can get $\mathcal{H}(x) = \mathcal{F}(x) + x$

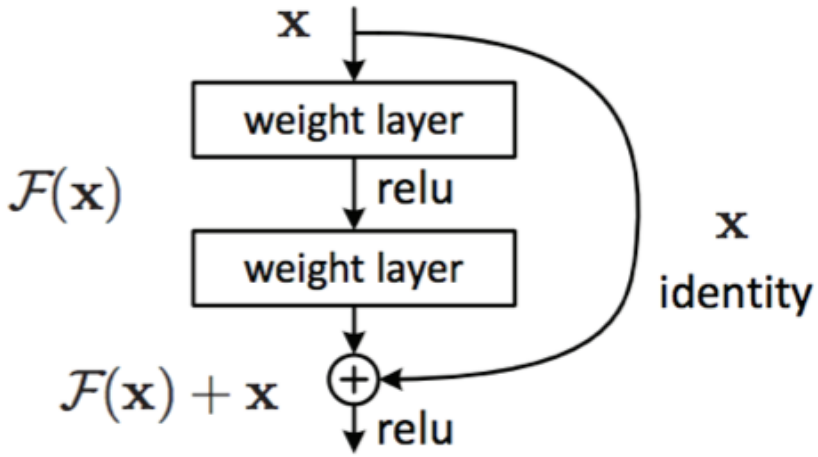


Figure 2. Residual learning: a building block.

Brief timeline

October 2015	beat European Go champion	5-0
March 2016	beat Lee Sedol (Federer of Go)	4-1
May 2017	beat Ke Jie (world no. 1)	3-0

superhuman

October 2017	AlphaGo Zero beats AlphaGo	100-0

DeepMind AlphaGo AMA

AMA: We are David Silver and Julian Schrittwieser from DeepMind's AlphaGo team. Ask us anything. (self.MachineLearning)

submitted 3 days ago * (last edited 1 day ago) by David_Silver

DeepMind - announcement

this post was submitted on 17 Oct 2017

245 points (97% upvoted)

shortlink: https://redd.it/76xjb5

DeepMind AlphaGo AMA

[–] **David_Silver** DeepMind [S] **9 points** 1 day ago

Creating a system that can learn entirely from self-play has been an open problem in reinforcement learning. Our initial attempts, as for many similar algorithms reported in the literature, were quite unstable. We tried many experiments - but ultimately the AlphaGo Zero algorithm was the most effective, and appears to have cracked this particular issue.

permalink source embed save save-RES parent report give gold reply hide child comments

[–] **David_Silver** DeepMind [S] **3 points** 1 day ago

In some sense, training from self-play is already somewhat adversarial: each iteration is attempting to find the "anti-strategy" against the previous version.

permalink source embed save save-RES parent report give gold reply

[–] **David_Silver** DeepMind [S] **13 points** 1 day ago

Actually we never guided AlphaGo to address specific weaknesses - rather we always focused on principled machine learning algorithms that learned for themselves to correct their own weaknesses.

Of course it is infeasible to achieve optimal play - so there will always be weaknesses. In practice, it was important to use the right kind of exploration to ensure training did not get stuck in local optima - but we never used human nudges.

permalink source embed save save-RES parent report give gold reply

https://www.reddit.com/r/MachineLearning/comments/76xjb5/ama_we_are_david_silver_and_julian_schrittwieser/

break

six – practical concerns

Key questions

What is the action space

- what can the agent choose between
- does the action change the environment state

What is the reward

Is it a complex problem

- cross entropy method often recommended as a simpler solution

Formulating action spaces

Continuous or discrete action space

determines which algorithms can be used

Constant vs variable action space

Important to be as computationally efficient as possible in creating state action combinations

i.e. for learning & training in Q-Learning

Reward scaling

Scaling of network inputs and targets to keep gradients under control

- supervised techniques are sensitive to scale
- normalization (scale to range 0.0 to 1.0)
- standardization (scale to mean=0, stdev=1)

True mean, min & max of value function not known

- this is a reinforcement learning specific problem
- supervised learning you can estimate these from training data

Learning values across many orders of magnitude

Hado van Hasselt **Arthur Guez** **Matteo Hessel** **Volodymyr Mnih** **David Silver**

Google DeepMind

29th Conference on Neural Information Processing Systems (NIPS 2016), Barcelona, Spain.

Learning values across many orders of magnitude

DQN clips rewards to [-1,+1]

- this becomes reward counting
- changes agent behavior

2016 DeepMind proposed adaptively normalizing targets

- helps agent to generalize across games
- helps with non-stationary problems

Mistakes I've made so far

Discount factor

what kind of behavior do I want (battery storage)

Monte Carlo policy gradient

using experience replay

shuffling the samples when training

using ε-greedy policy

Mistakes I've made so far

DQN

different batch size for experience replay & network training

multiple epochs over a batch

General

not setting next_observation = observation

Hyperparameters

Policy gradient methods

- learning rate
- clipping of distribution parameters (mean / stdev)

Value function methods

- learning rate
- epsilon – exploration
- update target network
- batch size
- space discretization

The Nuts and Bolts of Deep RL Research

John Schulman

OpenAI

August 26, 2017

John Schulman – Berkley Deep RL Bootcamp 2017
https://www.youtube.com/watch?v=8EcdaCk9KaQ

John Schulman advice

Quick experiments on small test problems

Interpret & visualize learning process

state visitation, value functions

Make it easier to get learning to happen (initially)

input features, reward function design

Always use multiple random seeds

John Schulman advice

Standardize data

if observations in unknown range, estimate running average mean & stdev

Rescale rewards - but don’t shift mean

Standardize prediction targets (i.e. value functions) the same way

John Schulman advice

Batch size matters

Policy gradient methods – weight initialization matters

determines initial state visitation (i.e. exploration)

DQN converges slowly

seven – beyond the expectation

A Distributional Perspective on Reinforcement Learning

Marc G. Bellemare [*1] **Will Dabney** [*1] **Rémi Munos** [1]

21 Jul 2017

Beyond the expectation

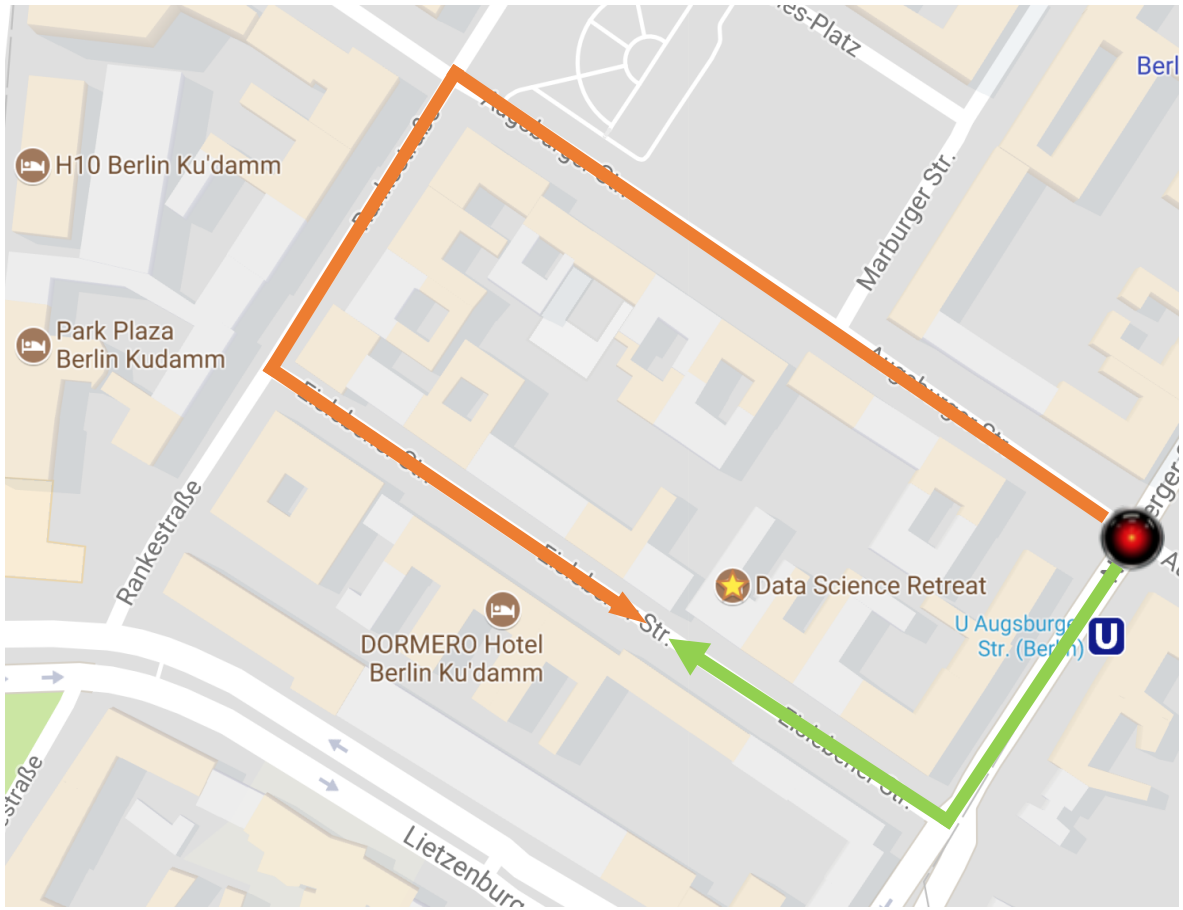
All the reinforcement learning today we have seen is about the **expectation** (mean expected return)

$$Q(s,a) = \mathbb{E}[G_t] = \mathbb{E}[r + \gamma Q(s',a)]$$

In 2017 DeepMind introduced the idea of the value **distribution**

State of the art results on Atari

Beyond the expectation



$Q(s, a_1) = 10 \text{ min}$

$Q(s, a_2) = 5 \text{ min}$

Expectation for uniform random policy $= 7.5$ min!

The expectation of 7.5 min will never occur in reality!

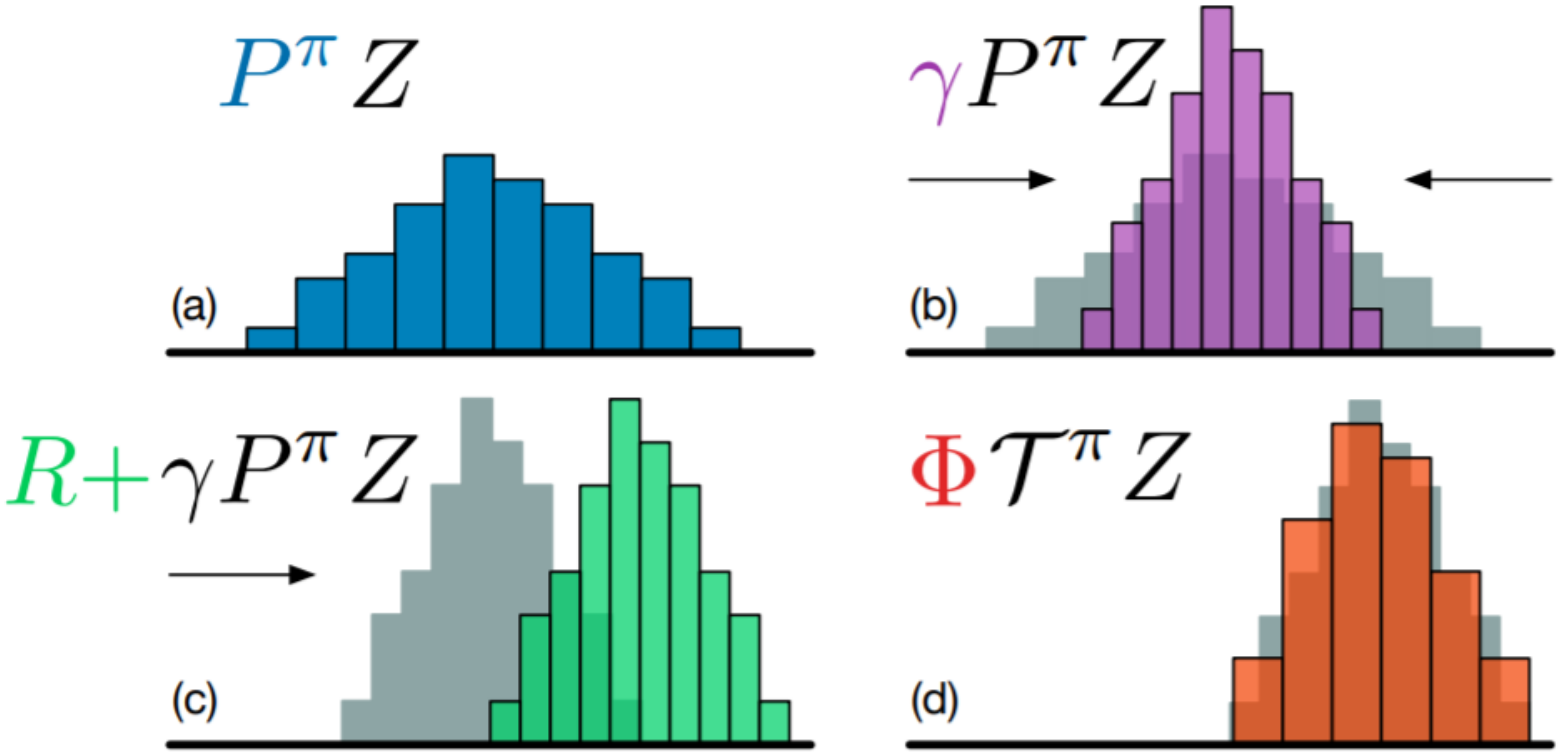


Figure 1. A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π, (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step (Section 4).

Bellemare et. al 2017

Beyond the expectation

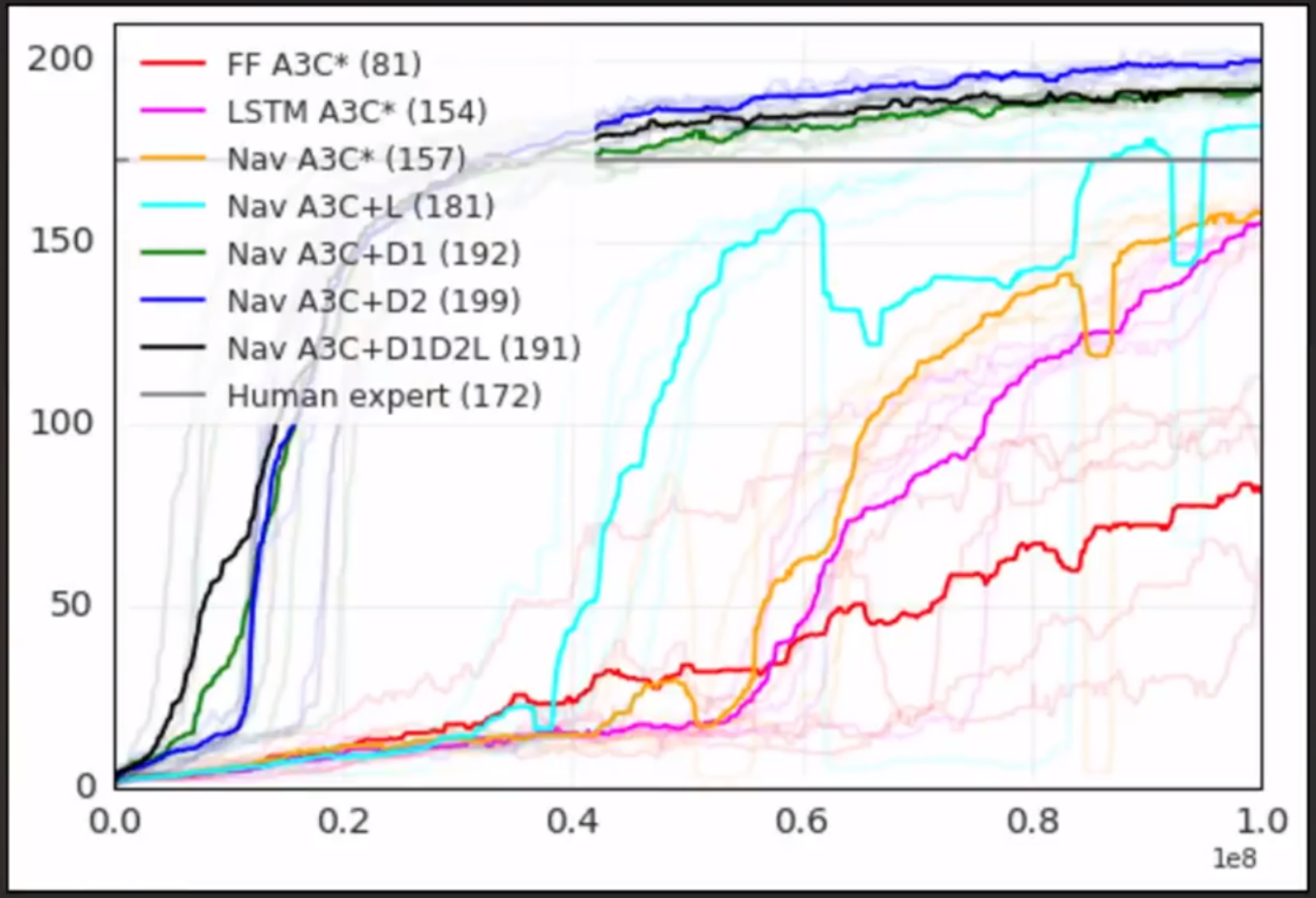
	Mean	Median	> H.B.	> DQN
DQN	228%	79%	24	0
DDQN	307%	118%	33	43
DUEL.	373%	151%	37	50
PRIOR.	434%	124%	39	48
PR. DUEL.	592%	172%	39	44
C51	**701%**	**178%**	**40**	**50**
UNREAL†	880%	250%	-	-

Figure 6. Mean and median scores across 57 Atari games, measured as percentages of human baseline (H.B., Nair et al., 2015).

† The UNREAL results are not altogether comparable, as they were generated in the asynchronous setting with per-game hyperparameter tuning (Jaderberg et al., 2017).

https://www.youtube.com/watch?v=mckulxKWyoc

Results: Auxiliary tasks speed up RL ten-fold!



DeepMind

seven – inverse RL

Inverse Reinforcement Learning

Chelsea Finn

Deep RL Bootcamp

Where does the reward come from?

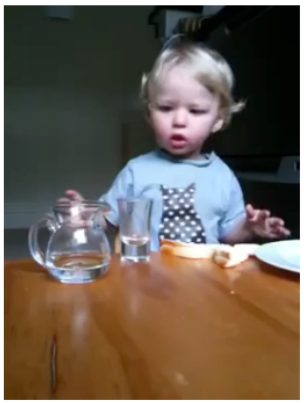
Computer Games

reward

Mnih et al. '15

Real World Scenarios

robotics



dialog

autonomous driving

what is the reward?
often use a proxy

frequently easier to provide expert data

Approach: infer reward function from roll-outs of expert policy

Inverse Optimal Control / Inverse Reinforcement Learning:
infer reward function from demonstrations

(IOC/IRL) (Kalman '64, Ng & Russell '00)

given:
- state & action space
- Roll-outs from π^*
- dynamics model (sometimes)

goal:
- recover reward function
- then use reward to get policy

Challenges

underdefined problem

difficult to evaluate a learned reward

demonstrations may not be precisely optimal

Chelsea Finn – Berkley Deep RL Bootcamp 2017

thank you

Adam Green

adgefficiency.com

adam.green@adgefficiency.com