

a glance at reinforcement learning

adam green

adam.green@adgefficiency.com

adgefficiency.com



Course Materials

https://github.com/ADGEfficiency/dsr_rl

google dsr_rl

- lecture notes hosted on GitPages (`PITCHME.md`)
- practical work - using an existing reinforcement learning library to run experiments.
- [introduction to tensorflow](#) and [python tricks](#) notebooks



Agenda - Today

today - morning

one - background & terminology

two - introduction to reinforcement learning

three - value functions & DQN

today - afternoon

DQN practical

≡

Agenda - Tomorrow

tomorrow - morning

four - improvements to DQN

five - policy gradients & Actor Critic

six - AlphaGo & AlphaGo Zero

seven - practical concerns

eight - a quick look at the state of the art

tomorrow - afternoon

Misc advice + portfolio projects



About Me

Education

B.Eng Chemical Engineering

MSc Advanced Process Design for Energy

DSR Batch 9

Industry

Energy Engineer at ENGIE UK

Energy Data Scientist at Tempus Energy



Goals for today and tomorrow

Introduction to concepts, ideas and terminology

Familiarity with important literature

Experience with running reinforcement learning experiments

Guidance on reinforcement learning project ideas

Working with existing code bases



Goals for today and tomorrow

To really learn RL, you will need to dedicate significant amount of time (same as if you want to learn NLP, convolution, GANs etc)

These slides are designed as both a **future reference** and slides for today



Where to go next

Textbook Sutton & Barto - An Introduction to Reinforcement Learning (2nd Edition is in progress)

Video lectures David Silver's 10 lecture series on YouTube

Literature review Li (2017) Deep Reinforcement Learning: An Overview



one
nomenclature & statistics background
a few things about training neural
networks

≡

Nomenclature

Nomenclature in RL can be inconsistent

- value function methods, action = a
- policy gradient methods, action = u

Following Thomas & Okal (2016) A Notation for Markov Decision Processes

| symbol | variable |
|------------------|--|
| s | state |
| s' | next state |
| a | action |
| r | reward |
| G_t | discounted return after time t |
| γ | discount factor [0, 1) |
| $a \sim \pi(s)$ | sampling action from a stochastic policy |
| $a = \pi(s)$ | deterministic policy |
| π^* | optimal policy |
| $V_\pi(s)$ | value function |
| $Q_\pi(s, a)$ | value function |
| θ, ω | function parameters (weights) |

≡

Expectations

Weighted average of all possible values (the mean)

`expected_value = probability * magnitude`

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

Expectations allow us to approximate by sampling

- if we want to approximate the average time it takes us to get to work
- we can measure how long it takes us for a week and get an approximation by averaging each of those days

≡

Conditionals

Probability of one thing given another

probability of next state and reward given state & action

$$P(s'|s, a)$$

reward received from a state & action

$$R(r|s, a, s')$$

sampling an action from a stochastic policy conditioned on being in state s

$$a \sim \pi(s|a)$$

≡

Variance & bias in supervised learning

Model generalization error = **bias + variance + noise**

Variance

- error from sensitivity to noise in data set
- seeing patterns that aren't there -> overfitting

Bias

- error from assumptions in the learning algorithm
- missing relevant patterns -> underfitting

Variance & bias in RL

Variance = deviation from expected value

- how consistent is my model / sampling
- can often be dealt with by sampling more
- high variance = sample inefficient

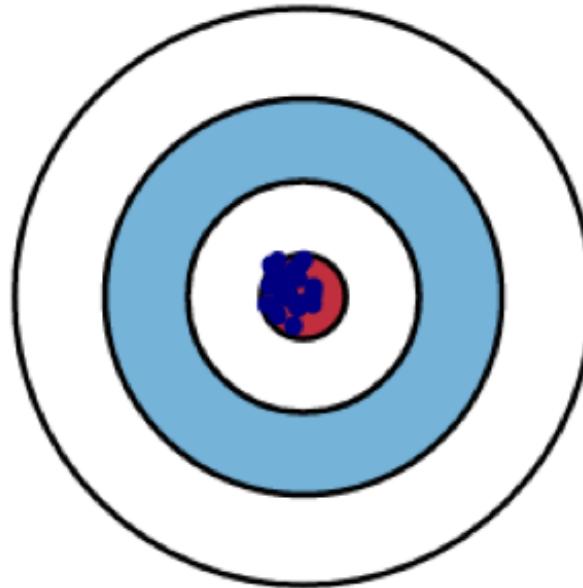
Bias = expected deviation vs true value

- how close to the truth is my model
- approximations or bootstrapping tend to introduce bias
- biased away from an optimal agent / policy

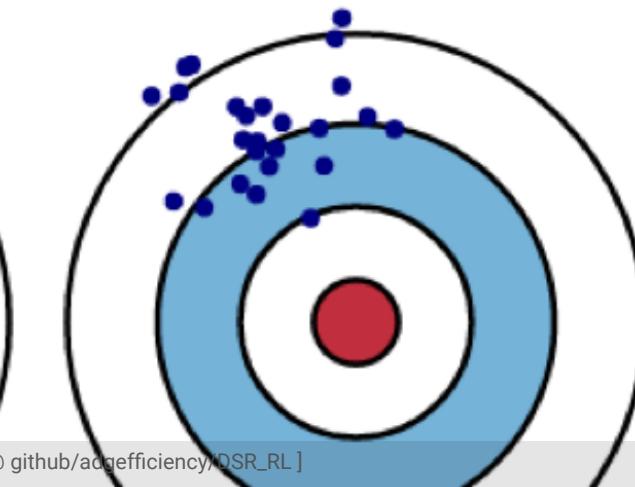
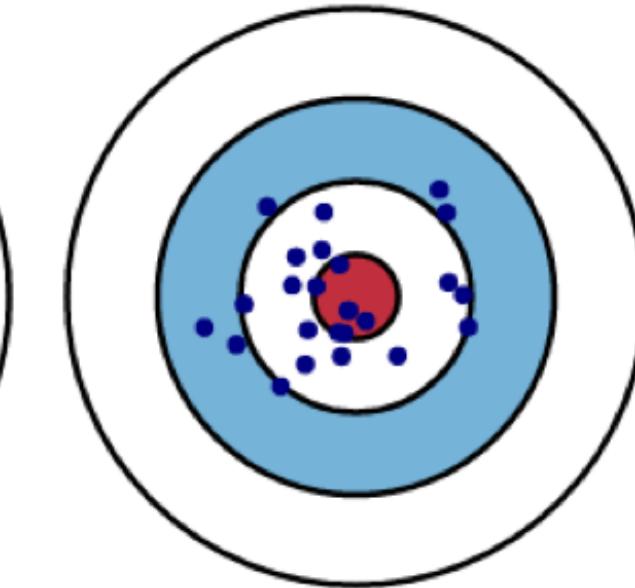
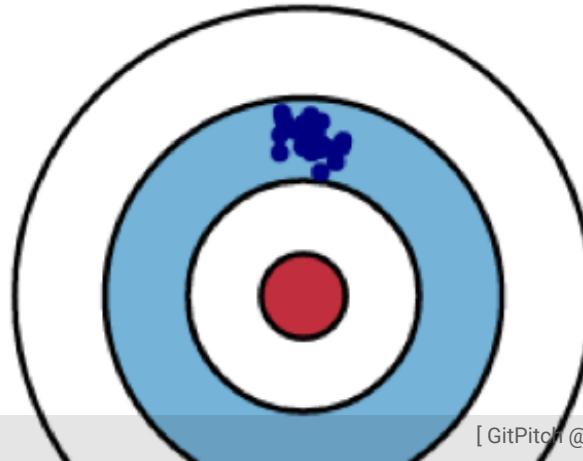
III

Low Variance High Variance

Low Bias



High Bias



Bootstrapping

Doing something on your own

- i.e. funding a startup with your own capital
- using a function to improve / estimate itself

The Bellman Equation is bootstrapped equation

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

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

Bootstrapping often introduces bias

- the agent has a chance to fool itself

≡

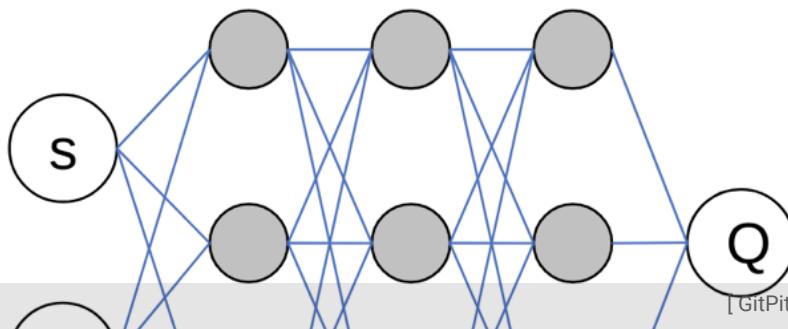
Function approximation

| state | temperature | pressure | estimate |
|-------|-------------|----------|----------|
| | s1 | s2 | |
| 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 tables

Two dimensions in the state variable

```
state = np.array([temperature, pressure])
```

| state | temperature | pressure | estimate |
|-------|-------------|----------|-----------|
| 0 | high | high | unsafe |
| 1 | low | high | safe |
| 2 | high | low | safe |
| 3 | low | low | very safe |

Lookup tables

Advantages

Stability

Each estimate is independent of every other estimate

Disadvantages

No sharing of knowledge between similar states/actions

Curse of dimensionality

High dimensional state/action spaces means lots of entries

Linear functions

$$V(s) = 3s_1 + 4s_2$$

Advantages

Less parameters than a table

Can generalize across states

Disadvantages

The real world is often non-linear

≡

Non-linear functions

Most commonly neural networks

Advantages

Model complex dynamics

Convolution for vision

Recurrency for memory / temporal dependencies

Disadvantages

Instability

Difficult to train

≡

iid

Fundamental assumption in statistical learning

Independent and identically distributed

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

≡

one
nomenclature & statistics background
a few things about training neural
networks

≡

A few things about training neural networks

Learning rate

Batch size

Scaling / preprocessing

Larger batch size

- larger learning rate
- decrease in generalization
- increase in batch normalization performance

≡

Learning rate

Controls the strength of weight updates performed by the optimizer (SGD, RMSprop, ADAM etc)

$$\theta^{t+1} = \theta^t - \alpha \frac{\partial E(x, \theta^t)}{\partial \theta}$$

where $E(x, \theta^t)$ is the error backpropagated from sample x

Small learning rate

- slow training

High learning rate

- overshoot or divergence

≡

Learning rate

Always intentionally set it

```
from keras.models import Sequential  
  
# don't do this!  
  
model.compile(optimizer='rmsprop', loss='mse')  
  
# do this  
  
from keras.optimizers import RMSprop  
  
opt = RMSprop(lr=0.001)  
  
model.compile(optimizer=opt, loss='mse')
```

≡

Batch size

Modern reinforcement learning trains neural networks using batches of samples

Below we have a dataset with four samples, of shape (14, 2)

```
>>> import numpy as np  
  
>>> data = np.arange(4*28).reshape(4, -1, 2)  
  
>>> data.shape  
  
(4, 14, 2)
```

The first dimension is the batch dimension - this is foundational in TensorFlow

≡

Batch size

Smaller batches can fit onto smaller GPUs

- if a large sample dimension we can use less samples per batch

Batches allow us to learn faster

- weights are updated more often during each epoch

Batches give a less accurate estimate of the gradient

- this noise can be useful to escape local minima

Larger batch size -> larger learning rate

- more accurate estimation of the gradient (better distribution across batch)
- we can take larger steps

≡



Batch size

Observed that larger batch sizes decrease generalization performance

Poor generalization due to large batches converging to *sharp minimizers*

- areas with large positive eigenvalues $\nabla^2 f(x)$
- Hessian matrix (matrix of second derivatives) where all eigenvalues positive = positive definite = local minima

Batch size is a **hyperparameter that should be tuned**

<https://stats.stackexchange.com/questions/164876/tradeoff-batch-size-vs-number-of-iterations-to-train-a-neural-network>

Scaling aka pre-processing

Neural networks don't like numbers on different scales

- improperly scaled inputs or outputs can cause issues with gradients
- anything that touches a neural network needs to be within a reasonable range

We can estimate statistics like min/max/mean from the training set

- these statistics are as much a part of the ML model as weights
- in reinforcement learning we have no training set

Scaling aka pre-processing

Standardization = removing mean & scale by unit variance

$$\phi(x) = x - \frac{\mu(x)}{\sigma(x)}$$

Our data now has mean of 0, variance of 1

Normalization = min/max scaling

$$\phi(x) = \frac{x - x_{min}}{x_{max} - x_{min}}$$

Our data is now between 0 and 1

≡

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 normalization

Batch norm. is additional preprocessing of data as it moves between network layers

- used in very deep convolutional/residual nets

We use the mean and variance of the batch to normalize activations

- standardization is actually used!
- reduces sensitivity to weight & bias initialization
- allows higher learning rates
- originally applied before the activation - but this is a topic of debate

Batch normalization before or after relu - Reddit

≡

Ian Goodfellow Lecture (3:20 onward)

Batch renormalization

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

- the estimated statistics are worse
- vanilla batch norm. uses two different methods for normalization during training & testing
- batch renormalization uses a single algorithm for both training & testing

≡

Recap

Three sources of generalization error

- ?
- ?
- ?

Missing relevant patterns in data = ?

Seeing patterns that aren't there = ?

One advantage & disadvantage of lookup tables

- advantage = ?
- disadvantage = ?

iid = ? and ? distributed

≡

Recap

Three sources of generalization error

- bias
- variance
- noise

Missing relevant patterns in data = bias

Seeing patterns that aren't there = variance

One advantage & disadvantage of lookup tables

- advantage = stability
- disadvantage = no aliasing between states, curse of dimensionality

iid = independent and identically distributed

≡

two
introduction to reinforcement learning
four central challenges
Markov Decision Processes

≡

Related methods

Evolutionary methods

- better able to deal with sparse error signals
- easily parallelizable
- tend to perform better than RL if state variable is hidden

Cross entropy method is often recommended as an alternative

Constrained optimization such as linear programming

Any other domain specific algorithm for your problem

Machine learning



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 to learn or approximate
functions

- classifier learns the function of image -> cat
- regressor learns the function of market_data -> stock_price

≡

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
- recurrent

≡

Model free reinforcement learning



Applications

RL is decision making

- ▶ **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, ...



Biological inspiration

Sutton & Barto - Reinforcement Learning: An Introduction

Of all the forms of machine learning, reinforcement learning is the closest to the kind of learning that humans and other animals do, and many of the core algorithms of reinforcement learning were originally inspired by biological learning systems

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

A new level of intelligence

Founder & CEO of DeepMind Demis Hassabis on the brilliance of AlphaGo in it's 2015 series



Reinforcement Learning

is

learning through action



learns to maximise return





Contrast with supervised learning

Supervised learning

- are given a dataset with labels
- we are constrained by this dataset
- test on unseen data

Reinforcement learning

- are given no dataset and no labels
- we can generate more data by acting
- test using the same environment

Data in RL

- the agent's experience (s, a, r, s')
- it's not clear what we should do with this data
- no implicit target

=

Reinforcement learning dataset

The dataset we generate is the agent's memory

$[experience,$

$experience,$

\dots

$experience]$

$[(s_0, a_0, r_1, s_1),$

$(s_1, a_1, r_2, s_2),$

\dots

$(s_n, a_n, r_n, s_n)]$

What should we do with this dataset?

≡

two introduction to reinforcement learning four central challenges Markov Decision Processes

≡

Four central challenges

one - exploration vs exploitation

two - data quality

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

- we don't want to waste time exploring poor quality states
- we don't want to miss high quality states



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

Time step matters

- too small = rewards are delayed = credit assignment harder
- too large = coarser control

≡

Data quality

iid = independent sampling & identical distribution

RL breaks both in multiple ways

Independent sampling

- all the samples collected on a given episode are correlated (along the state trajectory)
- our agent will likely be following a policy that is biased (towards good states)

Identically distributed

- learning changes the data distribution
- exploration changes the data distribution
- environment can be non-stationary

≡

Reinforcement learning will
always break supervised
learning assumptions about
data quality

≡

Credit assignment

The reward we see now might not be because of the action we just took

Reward signal can be

- **delayed** - benefit/penalty of action only seen much later
- **sparse** - experience with reward = 0

Can design a more dense reward signal for a given environment

- reward shaping
- changing the reward signal can change behaviour

Sample efficiency

How quickly a learner learns

How often we reuse data

- do we only learn once or can we learn from it again
- can we learn off-policy

How much we squeeze out of data

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

Requirement for sample efficiency depends on how expensive it is to generate data

- cheap data -> less requirement for data efficiency
- expensive / limited data -> squeeze more out of data

≡

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
four central challenges

Markov Decision Processes

≡

Markov Decision Processes

Mathematical framework for reinforcement learning

Markov property

Can be a requirement to guarantee convergence

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

$$P(s_{t+1}|s_t, a_t) = P(s_{t+1}|s_t, a_t \dots s_0, a_0)$$

≡

Formal definition of a MDP

An MDP can be defined a tuple

$$(\mathcal{S}, \mathcal{A}, \mathcal{R}, P, R, d_0, \gamma)$$

Set of states \mathcal{S}

Set of actions \mathcal{A}

Set of rewards \mathcal{R}

State transition function $P(s'|s, a)$

Reward transition function $R(r|s, a, s')$

Distribution over initial states d_0

≡

Discount factor γ

Object oriented definition of a MDP

Two objects - the agent and environment

Three signals - state, action & reward

```
class Agent
```

```
class Environment
```

```
state = env.reset()
```

```
action = agent.act(state)
```

```
reward, next_state = env.step(action)
```

≡

Environment

Real or virtual

- modern RL uses virtual environments to generate lots of experience

Each environment has a state space and an action space

- these spaces can be discrete or continuous

Environments can be

- episodic (finite length, can be variable or fixed length)
- non-episodic (infinite length)

The MDP framework unites both in the same way by using the idea of a final absorbing state at the end of episodes

≡

Discretization

Too coarse

- non-smooth control output

Too fine

- curse of dimensionality
- computational expense

Requires some prior knowledge

Lose the shape of the space

≡

State

Information for the agent to **choose next action** and to **learn from**

State is a flexible concept - it's a n-d array

```
state = np.array([temperature, pressure])
```

```
state = np.array(pixels).reshape(height, width)
```

Observation

Many problems your agent won't be able to see everything that would help it learn - i.e. non-Markov

This then becomes a POMDP - partially observed MDP

```
state = np.array([temperature, pressure])  
  
observation = np.array([temperature + noise])
```

Observation can be made more Markov by

- concatenating state trajectories together
- using function approximation with a memory element (LSTMs)

Agent

Our agent is the **learner and decision maker**

It's goal is to maximize total discounted reward

An agent always has a policy

Reward

Scalar

Delayed

≡

Sparse

Reward hypothesis

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

Goals can be described by the maximization of expected cumulative reward

Do you agree with this?

- happiness
- status
- reputation

Think about the role of emotion in human decision making. Is there a place for this in RL?



Reward engineering

The Reward Engineering Principle: As reinforcement-learning-based AI systems become more general and autonomous, the design of reward mechanisms that elicit desired behaviours becomes both more important and more difficult.

Reinforcement Learning and the Reward Engineering Principle



Policy $\pi(s)$

$$\pi(s)$$

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

$$\pi_\theta(s | a)$$

A policy is rules to select actions

- act randomly
- always pick a specific action
- the optimal policy - the policy that maximizes future reward

Policy can be

- parameterized directly (policy gradient methods)
- generated from a value function (value function methods)

≡

Prediction versus control

Prediction / approximation

- predicting return for given policy

Control

- the optimal policy
- the policy that maximizes expected future discounted reward

≡

On versus off policy learning

On policy

- learn about the policy we are using to make decisions

Off policy

- evaluate or improve one policy while using another to make decisions

Control can be on or off policy

- use general policy iteration to improve a policy using an on-policy approximation

≡

Why would we want to learn off-policy?

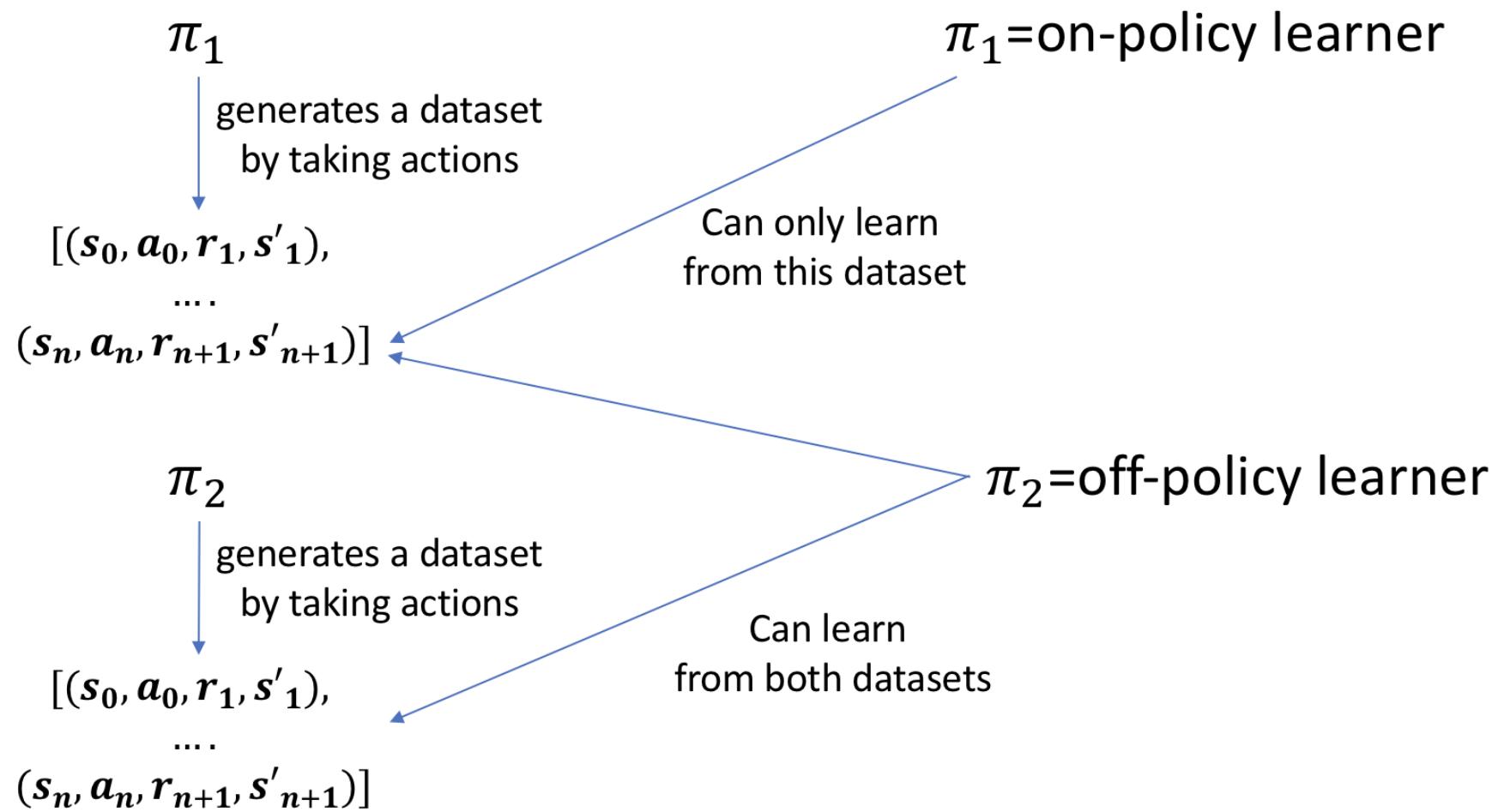
We can learn about policies that we don't have

- learn the optimal policy from data generated by a random policy

We can reuse data

- on-policy algorithms have to throw away experience after the policy is improved

Maybe the less we need to learn from deep learning is large capacity learners with large and diverse datasets - Sergey Levine



≡

Environment model

Our agent can learn an environment model

Predicts environment response to actions

- predicts s' , r from s, a

```
def model(state, action):  
    # do stuff  
    return next_state, reward
```

Sample vs. distributional model

Model can be used to simulate trajectories for planning

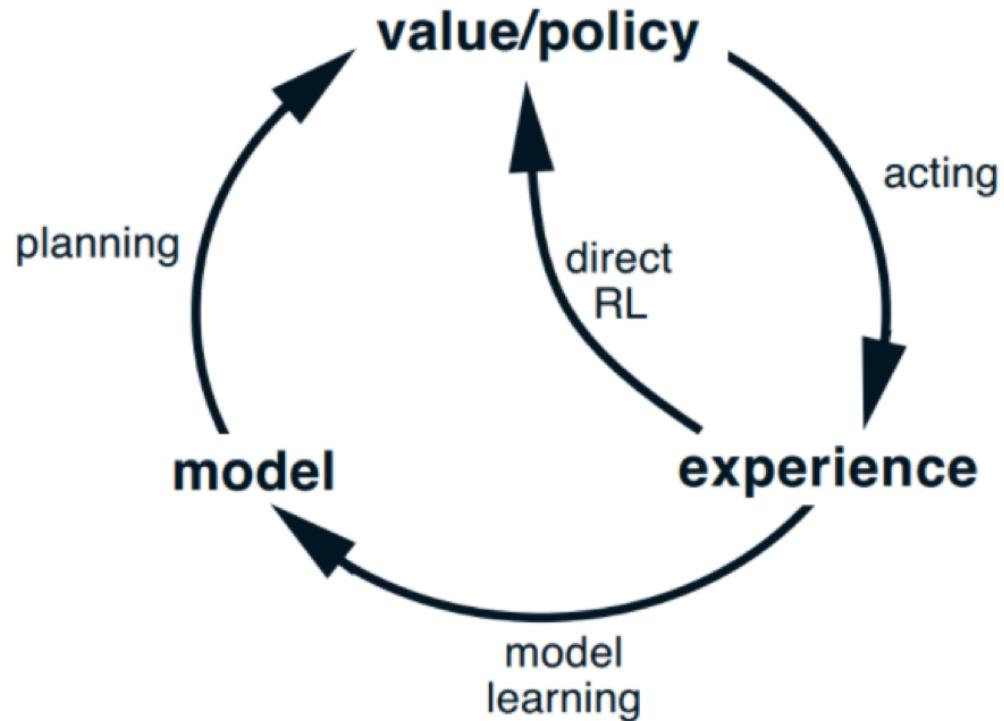
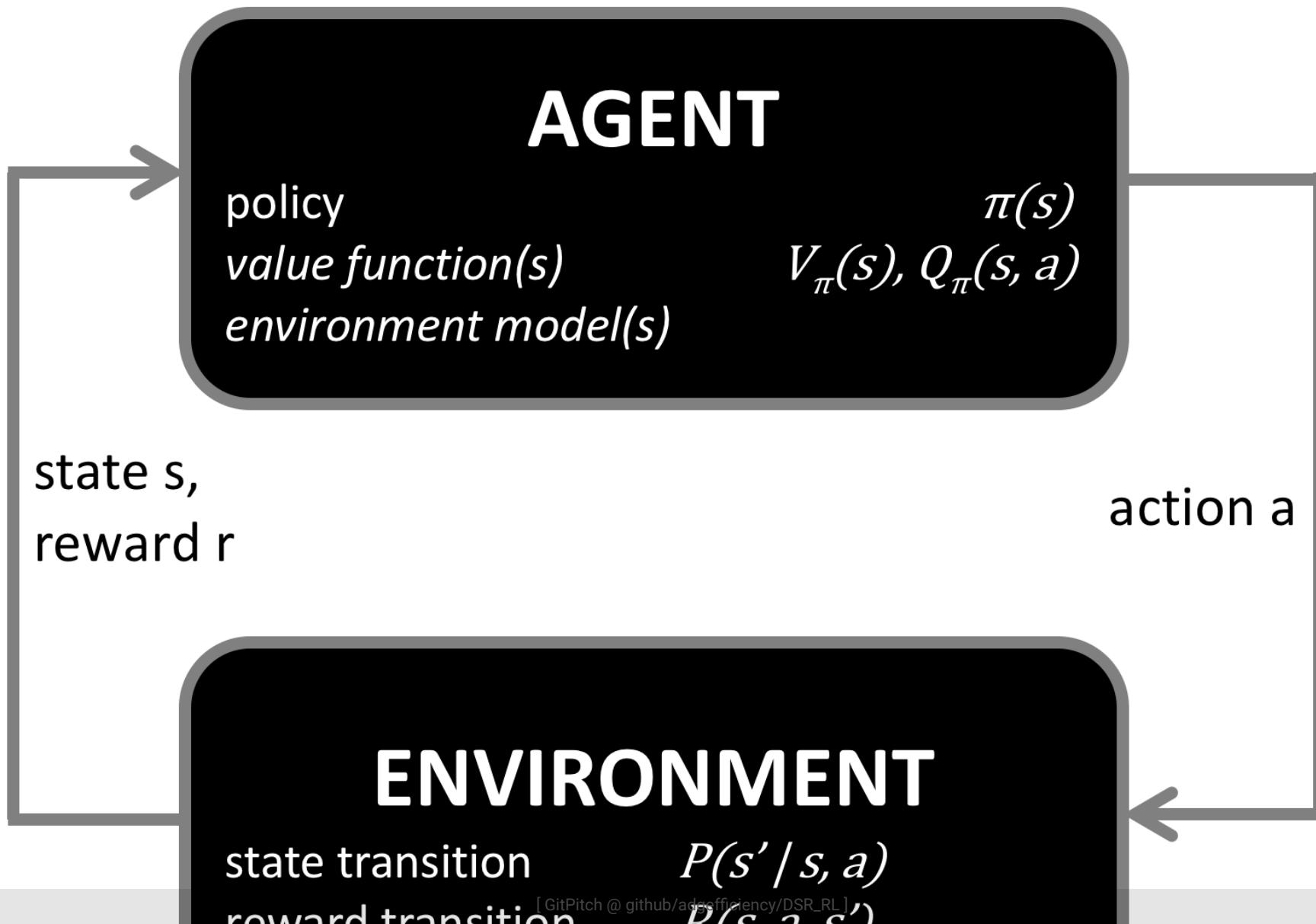


Figure 8.2: Relationships among learning, planning, and acting.

Sutton & Barto - Reinforcement Learning: An Introduction

learns to maximise return



Return

Goal of our agent is to maximize reward

Return (G_t) is the total discounted future reward

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

reward + discount * reward + discount^2 *
reward ...

Why do we discount future rewards?

≡

Discounting

Future is uncertain

- stochastic environment

Matches human thinking

- hyperbolic discounting

Finance

- time value of money

Makes the maths works

- return for infinite horizon problems finite
- discount rate is $[0, 1)$
- can make the sum of an infinite series finite

≡

Discounting

Can use discount = 1 for

- games with tree-like structures (without cycles)
- when time to solve is irrelevant (i.e. a board game)



Recap

How does reinforcement learning break iid?

- ?
- ?

What is the credit assignment problem?

- ?

An MDP is composed of two objects & three signals - what are they?

- ?
- ?

What is off-policy learning?

≡

Recap

How does reinforcement learning break iid?

- we don't sample experience independently - sampling biased by the agent & environment
- our experience is not independent - based on trajectory in the MDP

What is the credit assignment problem?

- working out which action gave us which rewards

An MDP is composed of two objects & three signals - what are they?

- agent & environment
- state, action, reward

≡

What is off-policy learning?

three
value functions

Bellman Equation

approximation methods

SARSA & Q-Learning

DQN

≡



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

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

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

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

=

Value function

$$V_\pi(s)$$

how good is this state

Action-value function

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

how good is this action

≡

Value function

$$V_\pi(s) = \mathbf{E}[G_t | s_t]$$

Expected return when in state s , following policy π

Action-value function

$$Q_\pi(s, a) = \mathbf{E}[G_t | s_t, a_t]$$

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

≡

Value functions are oracles

Prediction of the future

- predict expected future discounted reward
- always conditioned on a policy

We don't know this function

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

≡

Using a value function

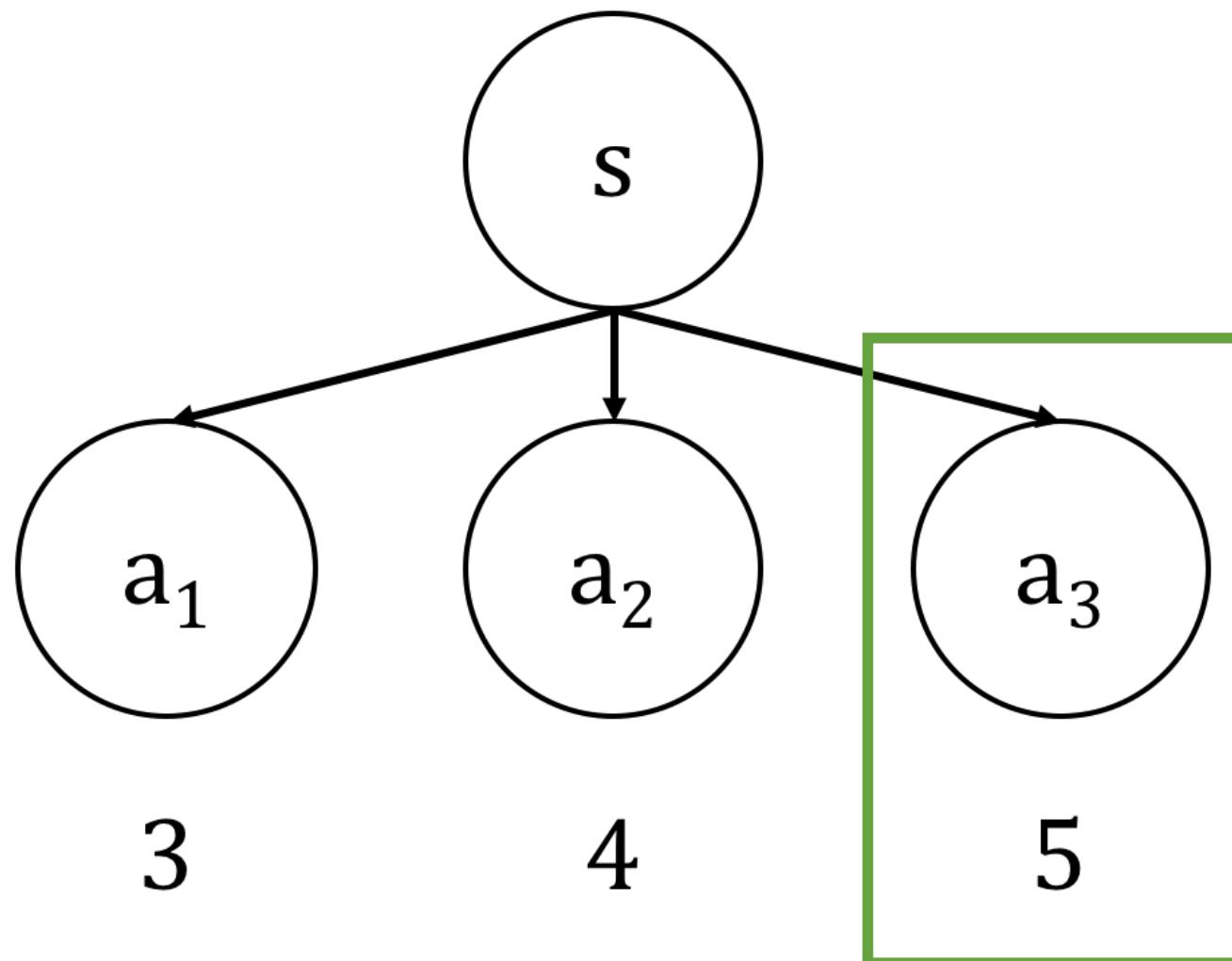
- given the optimal value function $Q_*(s, a)$
- we are in state s
- our set of actions $\mathcal{A} = a_1, a_2, a_3$

How can we act?

- use the value function to determine which action has the highest expected return
- select the action with the largest $Q(s, a)$ - ie take the $\arg \max_a Q(s, a)$

≡ This is known as a *greedy policy*

Using a value function



≡

```
def greedy_policy(state):
    # get the Q values for each state_action pair
    q_values = value_function.predict(state)

    # select action with highest Q
    action = np.argmax(q_values)

    return action
```



Approximation versus improvement

A good approximation of the optimal value function helps us to improve our policy

1 - improving the predictive power of our value function

2 - improving the policy

$$V_{k+1}(s) = \max_a \sum_{s',r} P(s',r|s,a)[r + \gamma V_k(s')]$$

These two steps are done sequentially in a process known as **policy iteration**

- approximate our policy (i.e. $V_\pi(s)$)
- act greedy wrt value function
- approximate our (better) policy

=

Generalized policy iteration

Letting policy evaluation and improvement processes interact

Policy iteration

- sequence of approximating value function then making policy greedy wrt value function

Value iteration

- single iteration of policy evaluation done inbetween each policy improvement

Both of these can achieve the same result

The policy and value functions interact to move both towards their optimal values - this is one source of non-stationary learning in RL

=

Policy iteration (using iterative policy evaluation)

1. Initialization

$V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$

2. Policy Evaluation

Repeat

$$\Delta \leftarrow 0$$

For each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \sum_{s',r} p(s', r | s, \pi(s)) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number)

3. Policy Improvement

policy-stable \leftarrow true

For each $s \in \mathcal{S}$:

$$\text{old-action} \leftarrow \pi(s)$$

$$\pi(s) \leftarrow \operatorname{argmax}_a \sum_{s',r} p(s', r | s, a) [r + \gamma V(s')]$$

If $\text{old-action} \neq \pi(s)$, then *policy-stable* \leftarrow false

If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

≡

Value iteration

Initialize array V arbitrarily (e.g., $V(s) = 0$ for all $s \in \mathcal{S}^+$)

Repeat

$$\Delta \leftarrow 0$$

For each $s \in \mathcal{S}$:

$$v \leftarrow V(s)$$

$$V(s) \leftarrow \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

$$\Delta \leftarrow \max(\Delta, |v - V(s)|)$$

until $\Delta < \theta$ (a small positive number)

Output a deterministic policy, $\pi \approx \pi_*$, such that

$$\pi(s) = \arg \max_a \sum_{s',r} p(s',r|s,a) [r + \gamma V(s')]$$

Sutton & Barto



Value function approximation

To approximate a value function we can use one of the methods we looked at in the first section

- lookup table
- linear function
- non-linear function

Linear functions are appropriate with some agents or environments

Modern reinforcement learning is based on using neural networks

≡

three
value functions

Bellman Equation

approximation methods

SARSA & Q-Learning

DQN

≡

Richard Bellman



Invented dynamic programming in 1953.

Also introduced the curse of dimensionality

- number of states S increases exponentially with number of dimensions in the state

≡

On the naming of dynamic programming

I was interested in planning, in decision making, in thinking. But planning, is not a good word for various reasons. I decided therefore to use the word, ‘programming.’ I wanted to get across the idea that this was dynamic, this was multistage, this was time-varying...



Bellman Equation

Bellman's contribution is remembered by the Bellman Equation

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

The Bellman equation relates the expected discounted return of the **current state** to the discounted value of the **next state**

The Bellman equation is a recursive definition - it is bootstrapped

We can apply it to value functions

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

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

≡

How do we use the Bellman Equation?

Create **targets** for learning

- train a neural network by minimizing the difference between the network output and the correct target
- improve our approximation of a value function we need to create a targets for each sample of experience
- minimize a loss function

$$\text{loss} = \text{target} - \text{approximation}$$

For an experience sample of (s, a, r, s')

$$\text{loss} = r + Q(s', a) - Q(s, a)$$

=

This is also known as the **temporal difference error**

three
value functions
Bellman Equation
approximation methods
SARSA & Q-Learning
DQN

≡

Approximation methods

Look at three different methods for approximation

1. dynamic programming
2. Monte Carlo
3. temporal difference

We are **creating targets** to learn from

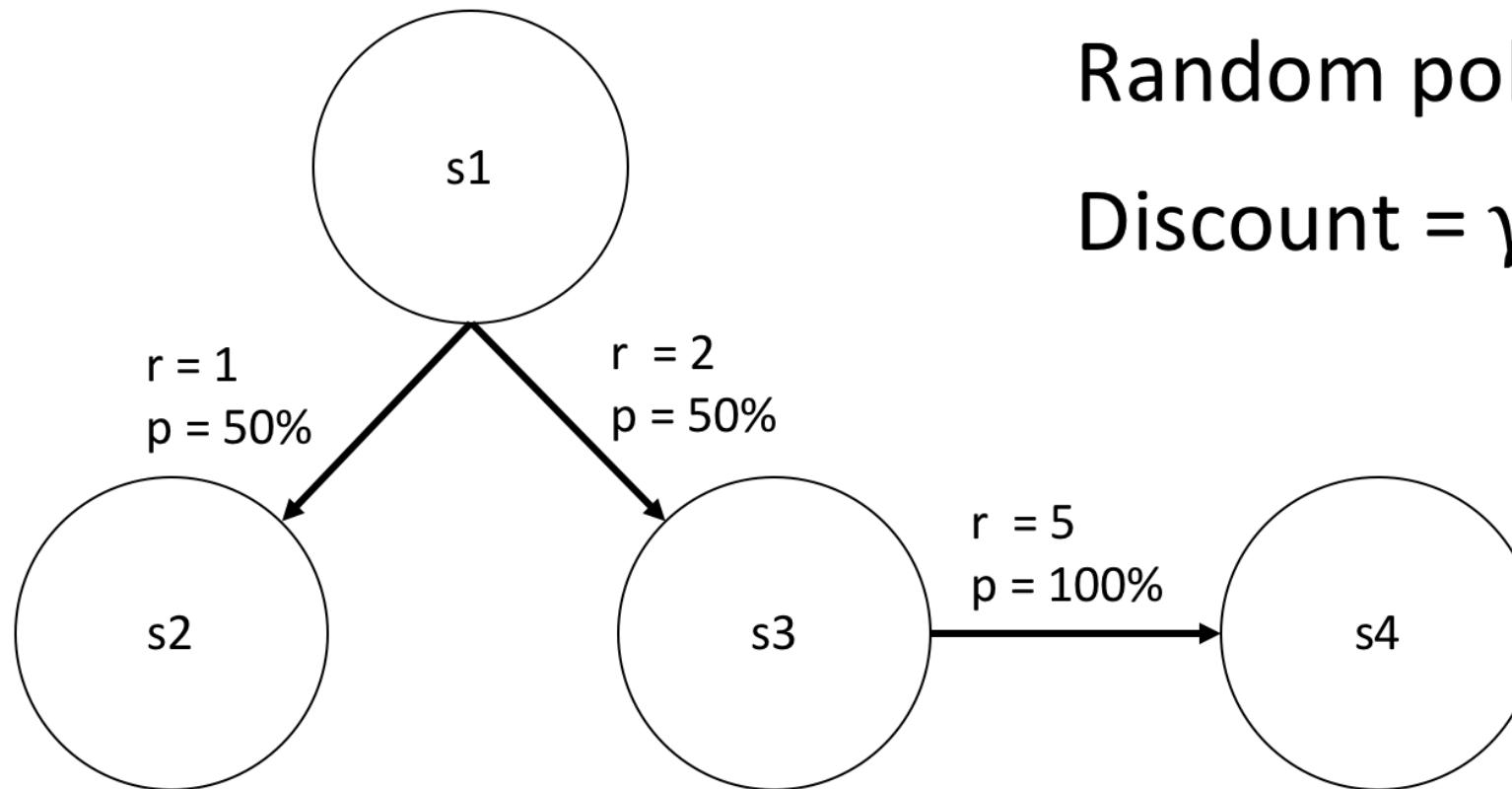
Policy improvement can be done by either policy iteration or value iteration for all of these different approximation methods

Dynamic programming

Imagine you had a perfect environment model

- the state transition function $P(s'|s, a)$
- the reward transition function $R(r|s, a, s')$

Can we use our perfect environment model for value function approximation?



Random policy
Discount = $\gamma = 0.9$

Note that the probabilities here depend both on the environment and the policy

≡

Dynamic programming backup

We can perform iterative backups of the expected return for each state

The return for all terminal states is zero

$$V(s_2) = 0$$

$$V(s_4) = 0$$

We can then express the value functions for the remaining two states

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

$$V(s_3) = 1 \cdot [5 + 0.9 \cdot 0] = 5$$

≡

Dynamic programming

Our value function approximation depends on

- our policy (what actions we pick)
- the environment (where our actions take us and what rewards we get)
- our current estimate of $V(s')$

A dynamic programming update is expensive

- our new estimate $V(s)$ depends on the value of all other states (even if the probability is zero)

≡

Dynamic programming summary

Requires a **perfect environment model**

- we don't need to sample experience at all
- we don't ever actually take actions)

We make **full backups**

- the update to the value function is based on the probability distribution over all possible next states

Bootstrapped

- we use the recursive Bellman Equation to update our value function

Limited utility in practice but they provide an **essential foundation** for understanding reinforcement learning

≡

Monte Carlo

Monte Carlo methods = finding the expected value of a function of a random variable

No model

- we learn from actual experience

We can also learn from **simulated experience**

- we don't need to know the whole probability distribution
- just be able to generate sample trajectories

No bootstrapping

- we take the average of the true discounted return

≡

Episodic only

Monte Carlo approximation

Estimate the value of a state by averaging the true discounted return observed after each visit to that state

As we run more episodes, our estimate should converge to the true expectation

Low bias & high variance - why?

Bias & variance of 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

Monte Carlo algorithm

Algorithm for a lookup table based Monte Carlo approximation



Interesting feature of Monte Carlo

Computational expense of estimating the value of state s is independent of the number of states \mathcal{S}

This is because we use experienced state transitions



Monte Carlo

Learn from actual or simulated experience

- no environment model

No bootstrapping

- use true discounted returns sampled from the environment

Episodic problems only

- no learning online

Ability to **focus** on interesting states and ignore others

High variance & low bias

≡

Temporal difference

Learn from actual experience

- like Monte Carlo
- no environment model

Bootstrap

- like dynamic programming
- learn online

Episodic & non-episodic problems



Temporal difference

Use the Bellman Equation to approximate $V(s)$ using $V(s')$

Temporal difference error

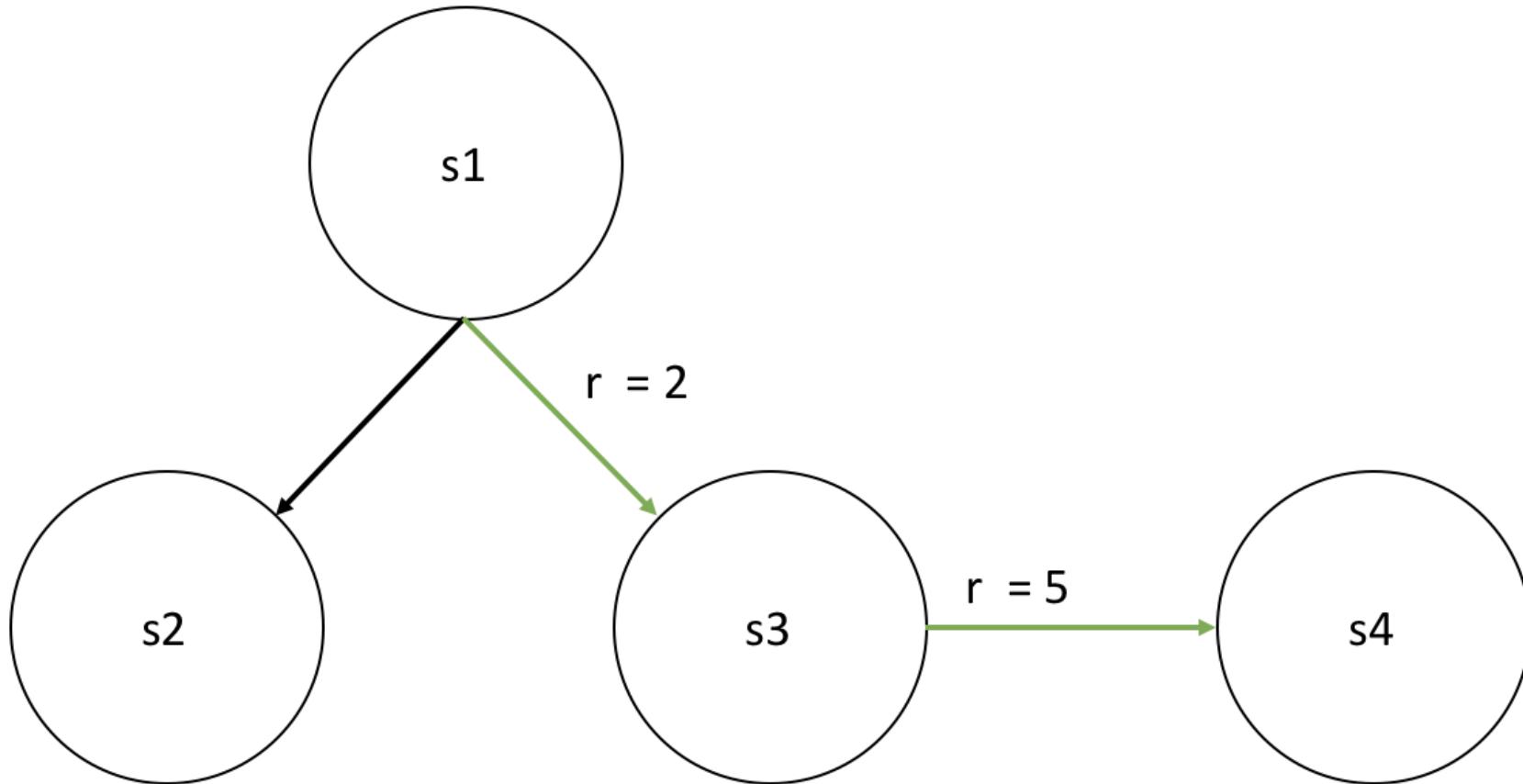
$$\begin{aligned} \text{actual} &= r + \gamma V(s') \\ \text{predicted} &= V(s) \end{aligned}$$

$$\begin{aligned} \text{error} &= \text{actual} - \text{predicted} \\ \text{error} &= r + \gamma V(s') - V(s) \end{aligned}$$

Update rule for a table TD(0) approximation

≡

Temporal difference backup



=

$$V(s_1) \leftarrow V(s_1) + \alpha[r_{23} + \gamma V(s_3) - V(s_1)]$$

You are the predictor

Example 6.4 from Sutton & Barto

Imagine you observe the following episodes

- format of (State Reward, State Reward)
- i.e. A 0 B 0 = state A, reward 0, state B, reward 0

| Episode | Number times observed |
|----------|-----------------------|
| A 0, B 0 | 1 |
| B 1 | 6 |
| B 0 | 1 |

What are the optimal predictions for $V(A)$ and $V(B)$?

You are the predictor

We can estimate the expected return from state B by averaging the rewards

$$V(B) = 6/8 \cdot 1 + 2/6 \cdot 0 = 3/4$$

What about $V(A)$?

- We observed that every time we were in A we got 0 reward and ended up in B
- Therefore $V(A) = 0 + V(B) = 3/4$

or

- We observed a discounted return of 0 each time we saw A
- therefore $V(A) = 0$

≡

Which is the Monte Carlo approach, which is the TD approach?

You are the predictor

Estimating $V(A) = 3/4$ is the answer given by TD(0)

Estimating $V(A) = 0$ is the answer given by Monte Carlo

The MC method gives us the lowest error on fitting the data (i.e. minimizes MSE)

The TD method gives us the **maximum-likelihood estimate**

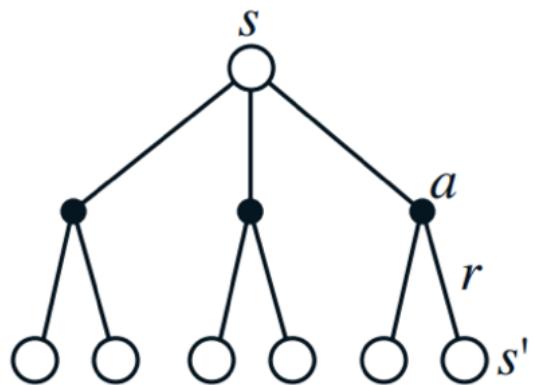
You are the predictor

The maximum likelihood estimate of a parameter is the parameter value whose probability of generating the data is greatest

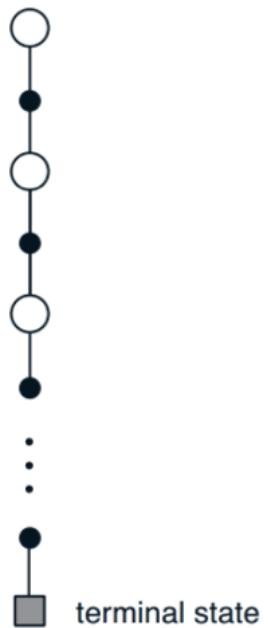
We take into account the transition probabilities, which gives us the **certainty equivalence estimate** - which is the estimate we get when assuming we know the underlying model (rather than approximating it)

Recap

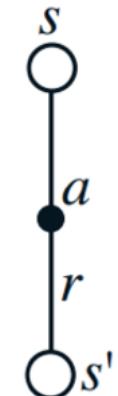
Dynamic Programming



Monte Carlo



Temporal Difference



three
value functions
Bellman Equation
approximation methods
SARSA & Q-Learning
DQN

≡

SARSA & Q-Learning

Approximation is a tool - **control** is what we really want

SARSA & Q-Learning are both based on the **action-value function** $Q(s, a)$

The practical today is based on DQN - the DeepMind implementation of Q-Learning

Why might we want to learn $Q(s, a)$ rather than $V(s)$?

≡

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

Imagine a simple MDP

$$\mathcal{S} = s_1, s_2, s_3$$

$$\mathcal{A} = a_1, a_2$$

Our agent finds itself in state s_2

We use our value function $V(s)$ to calculate

$$V(s_1) = 10$$

$$V(s_2) = 5$$

$$V(s_3) = 20$$

≡

Which action should we take?

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

Now imagine we had

$$Q(s_2, a_1) = 40$$

$$Q(s_2, a_2) = 20$$

It's now easy to pick the action that maximizes expected discounted return

$V(s)$ tells us how good a state is. We require the state transition probabilities for each action to use $V(s)$ for control

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

≡

SARSA

SARSA is an **on-policy** control method

- we approximate the policy we are following
- we improve the policy by being greedy wrt to our approximation

We use every element from our experience tuple (s, a, r, s')

- and also a' - the next action selected by our agent

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma Q(s', a') - Q(s, a)]$$

Why is SARSA on-policy?

≡

SARSA

SARSA is on-policy because we learn about the action a' that our agent choose to take

Our value function is always for the policy we are following

- the state transition probabilities depend on the policy

But we can improve it using general policy iteration (GPI)

- approximate $Q(s, a)$ for our current policy
- act greedily towards this approximation of $Q(s, a)$
- approximate $Q(s, a)$ for our new experience
- act greedily towards this new approximation
- repeat

≡

Q-Learning

Q-Learning allows **off-policy control**

- use every element from our experience tuple (s, a, r, s')

We take the **maximum over all possible next actions**

- we don't need to know what action our agent took next (i.e. a')

This allows us to learn the optimal value function while following a sub-optimal policy

$$Q(s, a) \leftarrow Q(s, a) + \alpha[r + \gamma \max_a Q(s', a') - Q(s, a)]$$

Don't learn Q_π - learn Q^* (the optimal policy)

≡

SARSA & Q-Learning



Q-Learning

Selecting optimal actions in Q-Learning can be done by an *argmax* across the action space

$$\text{action} = \underset{a}{\operatorname{argmax}} Q(s, a)$$

The *argmax* limits Q-Learning to **discrete action spaces only**

For a given approximation of $Q(s, a)$ acting greedy is deterministic

How then do we explore the environment?

≡

ϵ -greedy exploration

A common exploration strategy is the **epsilon-greedy policy**

```
def epsilon_greedy_policy():
    if np.random.rand() < epsilon:
        # act randomly
        action = np.random.uniform(action_space)

    else:
        # act greedy
        action = np.argmax(Q_values)

    return action
```

ϵ is decayed during experiments to explore less as our agent learns (i.e. to exploit)

Exploration strategies

Boltzmann (a softmax)

- temperature being annealed as learning progresses

Bayesian Neural Network

- a network that maintains distributions over weights -> distribution over actions
- this can also be performed using dropout to simulate a probabilistic network

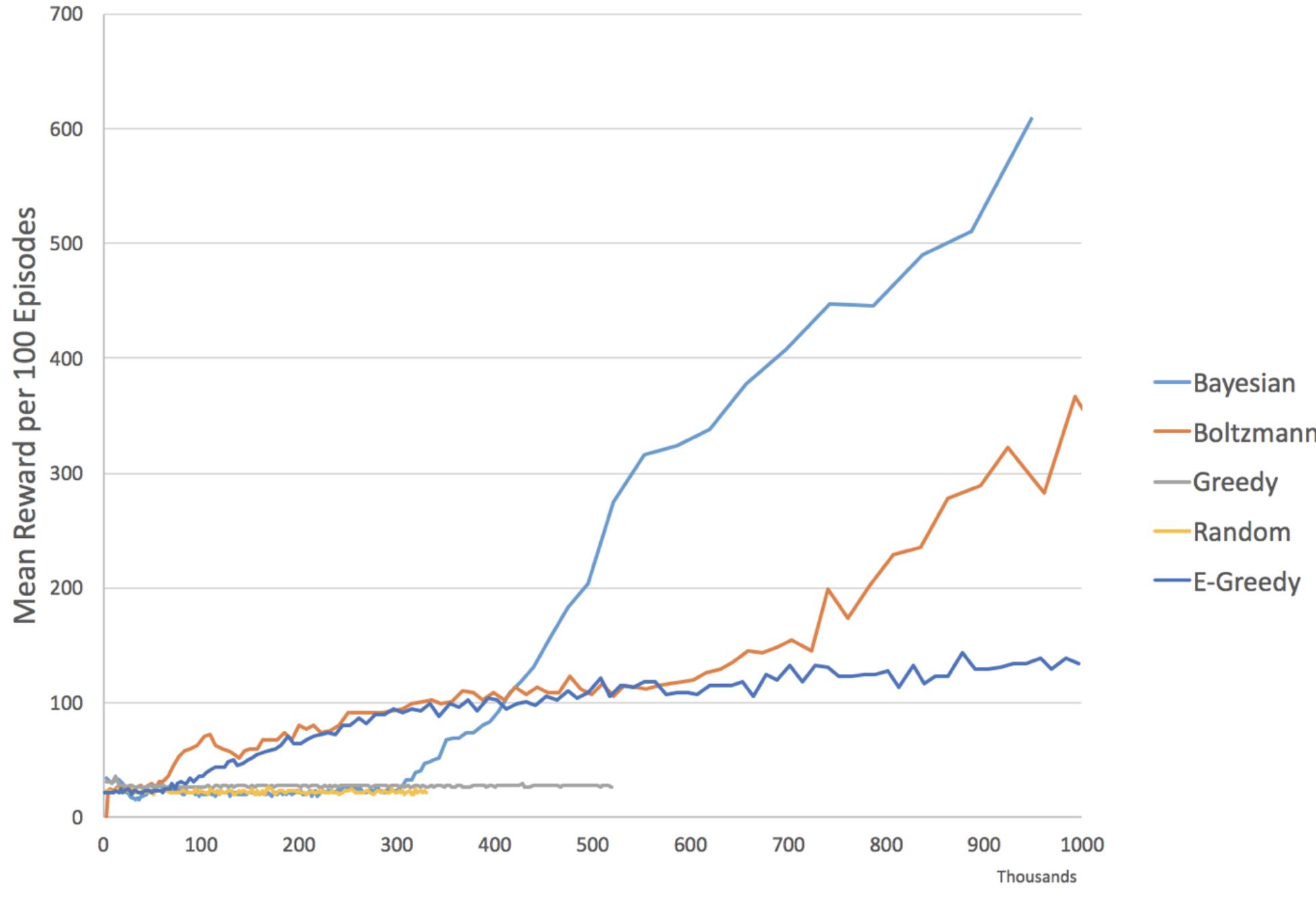
Parameter noise

- adding adaptive noise to weights of network

Action-Selection Strategies for Exploration

≡

Action Selection Methods - Cartpole Performance



Problems with vanilla Q-Learning

Correlations in our dataset (the list of experience tuples)

- combine this with bootstrapping and instability occurs

Small changes $Q(s, a)$ estimates can drastically change the policy

$$Q(s_1, a_1) = 10$$

$$Q(s_1, a_2) = 11$$

Then we do some learning and our estimates change

$$Q(s_1, a_1) = 12$$

$$Q(s_1, a_2) = 11$$

Now our policy is completely different!

=

Deadly triad

Sutton & Barto discuss the concept of the **deadly triad**

Three mechanisms that combine to produce instability and divergence

1. off-policy learning - to learn about the optimal policy while following an exploratory policy
2. function approximation - for scalability and generalization
3. bootstrapping - computational & sample efficiency



Deadly triad

It's not clear what causes instability

- dynamic programming can diverge with function approximation (so even on-policy learning can diverge)
- prediction can diverge
- linear functions can be unstable

Divergence is an emergent phenomenon

Up until 2013 the deadly triad caused instability when using Q-Learning with complex function approximators (i.e. neural networks)

Then came DeepMind & DQN

≡

three
value functions
Bellman Equation
approximation methods
SARSA & Q-Learning
DQN

≡

DQN

In 2013 a small London startup published a paper

- an agent based on Q-Learning
- superhuman level of performance in three Atari games

In 2014 Google purchased DeepMind for around £400M

This is for a company with

- no product
- no revenue
- no customers
- a few world class employees



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



Significance

End to end deep reinforcement learning

- Q-Learning with neural networks was historically unstable

Learning from high dimensional input

- raw pixels

Ability to **generalize**

- same algorithm, network structure and hyperparameters

Reinforcement learning to play Atari

State

- Last four screens concatenated together
- Allows information about movement
- Grey scale, cropped & normalized

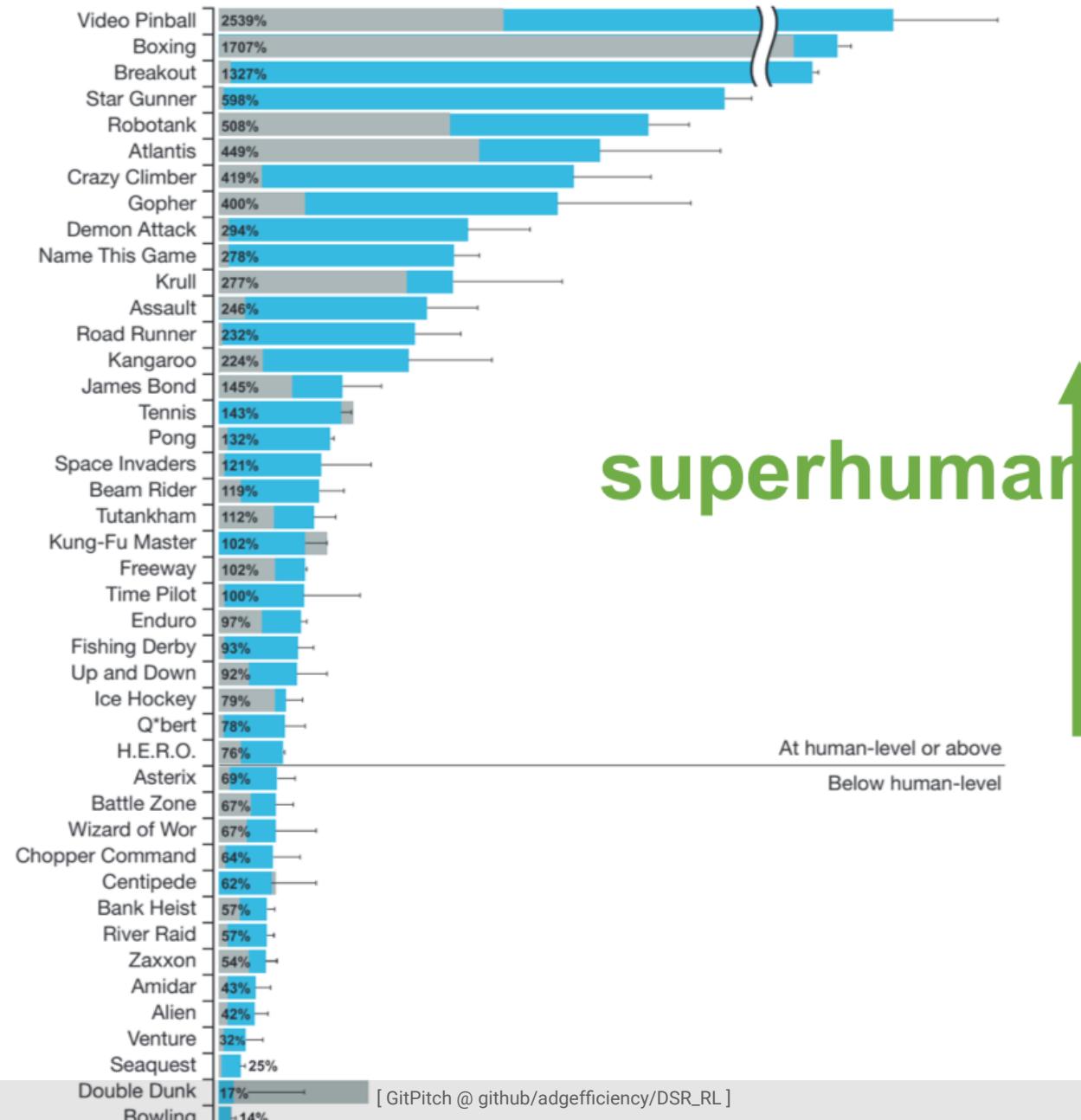
Reward

- Game score
- Clipped to $[-1, +1]$

Actions

- Joystick buttons (a discrete action space)

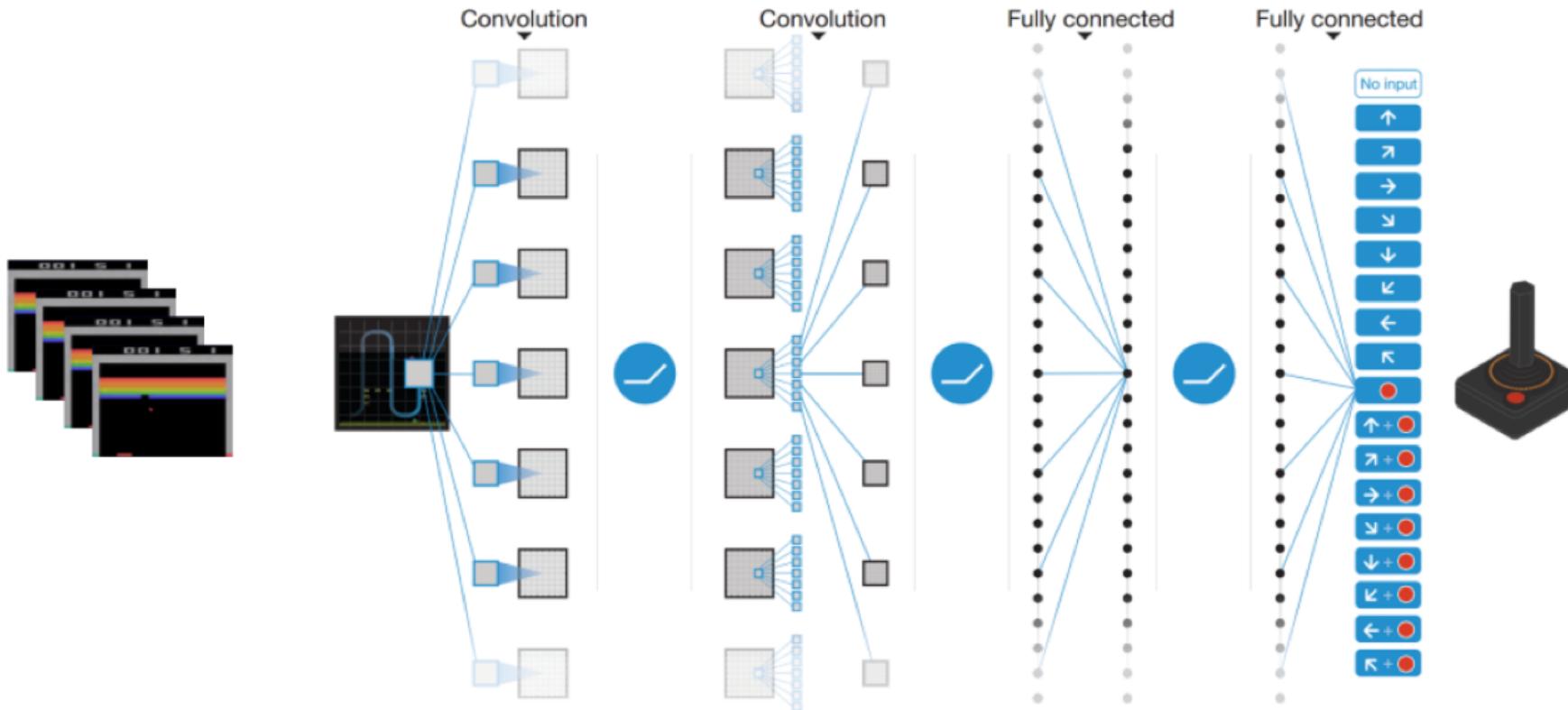
≡



superhuman

At human-level or above

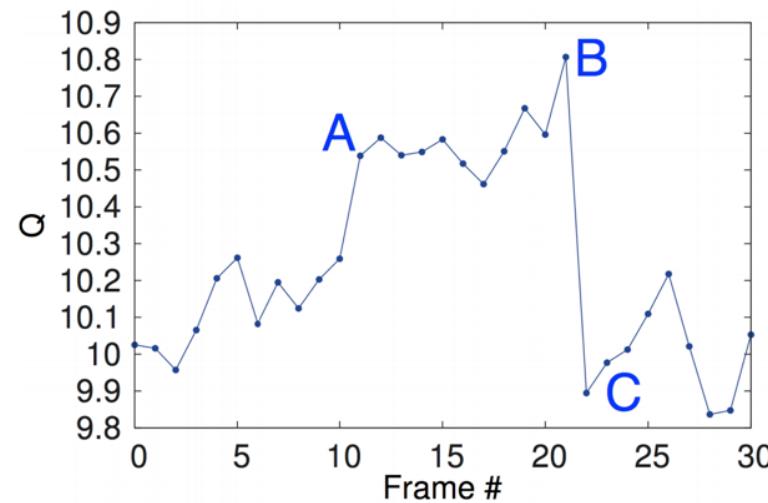
Below human-level



| Layer | Input | Filters | Filter size | Stride | Activation |
|-------------------|-----------|---------|-------------|--------|------------|
| 1 Convolution | 84x84x4 | 16 (32) | 8x8 | 4 | ReLU |
| 2 Convolution | 20x20x32 | 32 (64) | 4x4 | 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](#)



Agent sees new enemy

Agent has fired torpedo

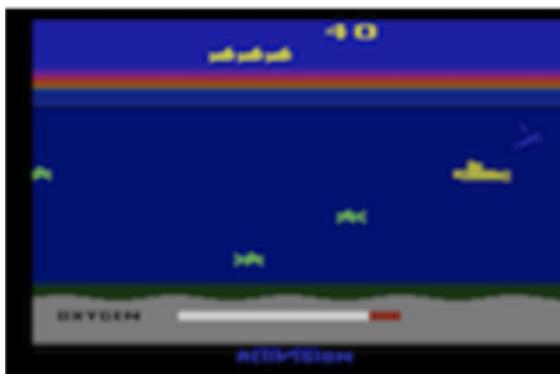
Reward received

Value function gets excited

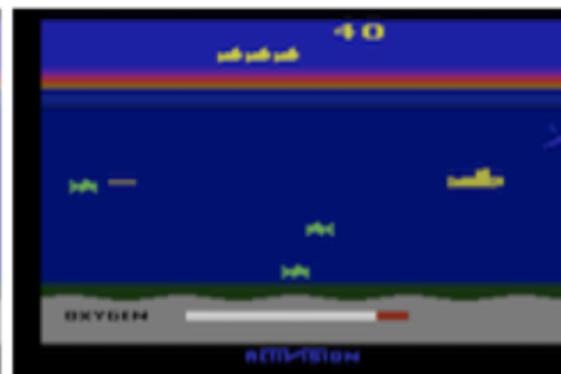
Value function is very excited

Value function back to normal

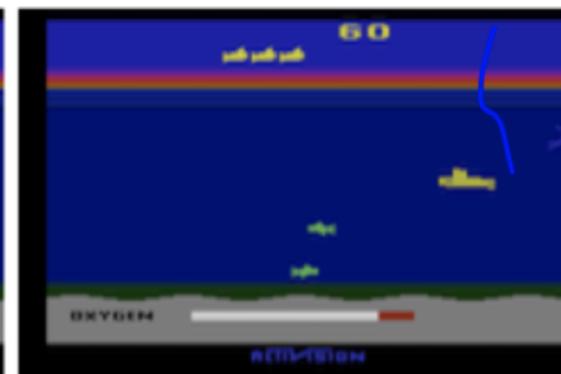
A



B



C



Two key innovations in DQN

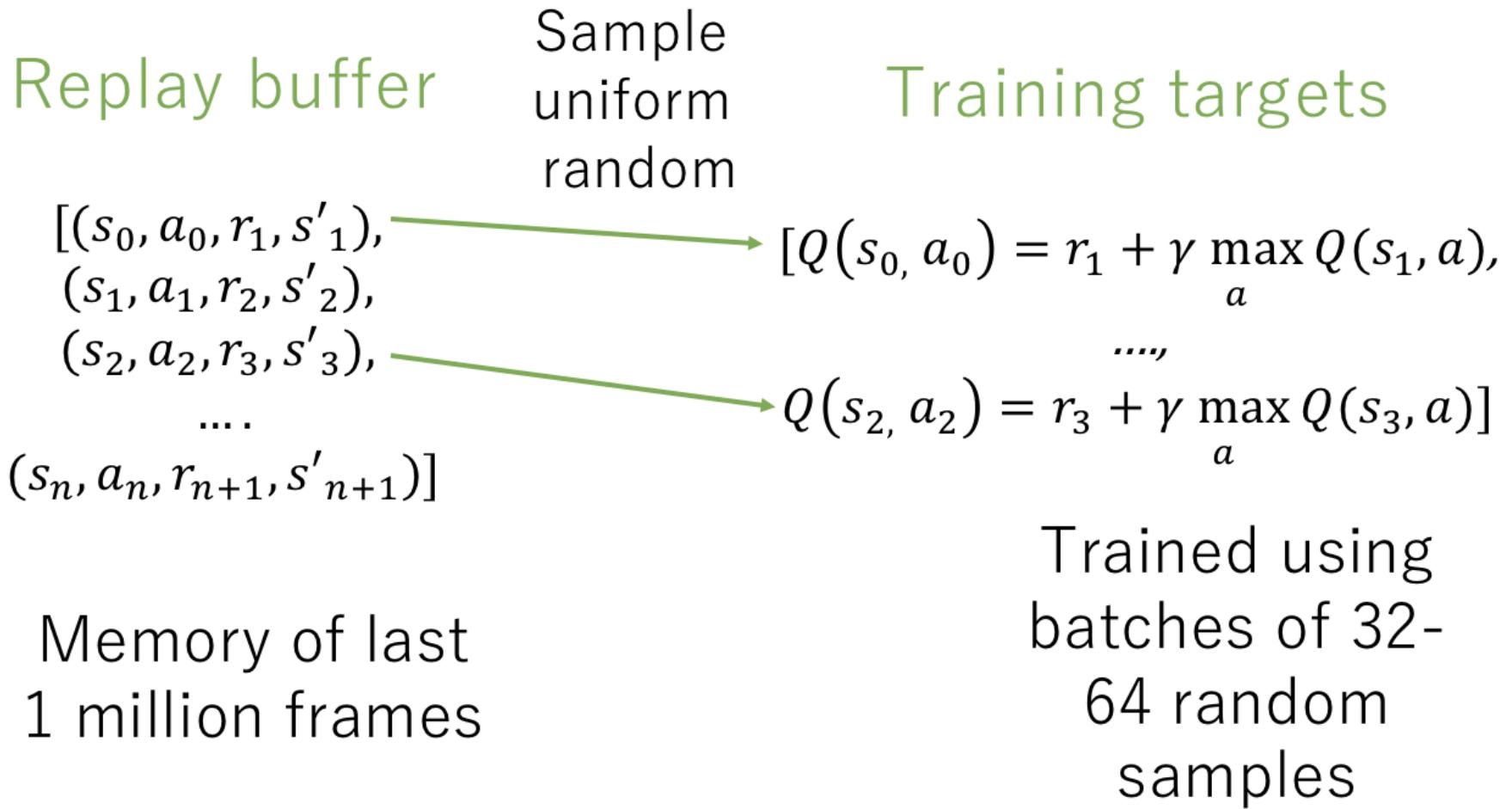
Experience replay

Target network

Both improve learning **stability**



Experience replay



≡

Experience replay

Experience replay helps to deal with our non-iid dataset

- randomizing the sampling of experience -> more independent
- brings the batch distribution closer to the true distribution -> more identical

Data efficiency

- we can learn from experience multiple times

Allows seeding of the memory with high quality experience

Biological basis for experience replay

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

Mnih et. al (2015)

≡

Target network

Second innovation behind DQN

Parameterize two separate neural networks (identical structure)
- two sets of weights θ and θ^-

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

target **approx.**

Original Atari work copied the online network weights to the target network every 10k - 100k steps

Can also use a small factor tau (τ) to smoothly update weights at each step

=

Target network

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

- bigger networks less prone (less aliasing aka weight sharing)

Stable training

- no longer bootstrapping from the same function, but from an old & fixed version of $Q(s, a)$
- reduces correlation between the target created for the network and the network itself

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

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 $a_t = \operatorname{argmax}_a Q(\phi(s_t), a; \theta)$

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

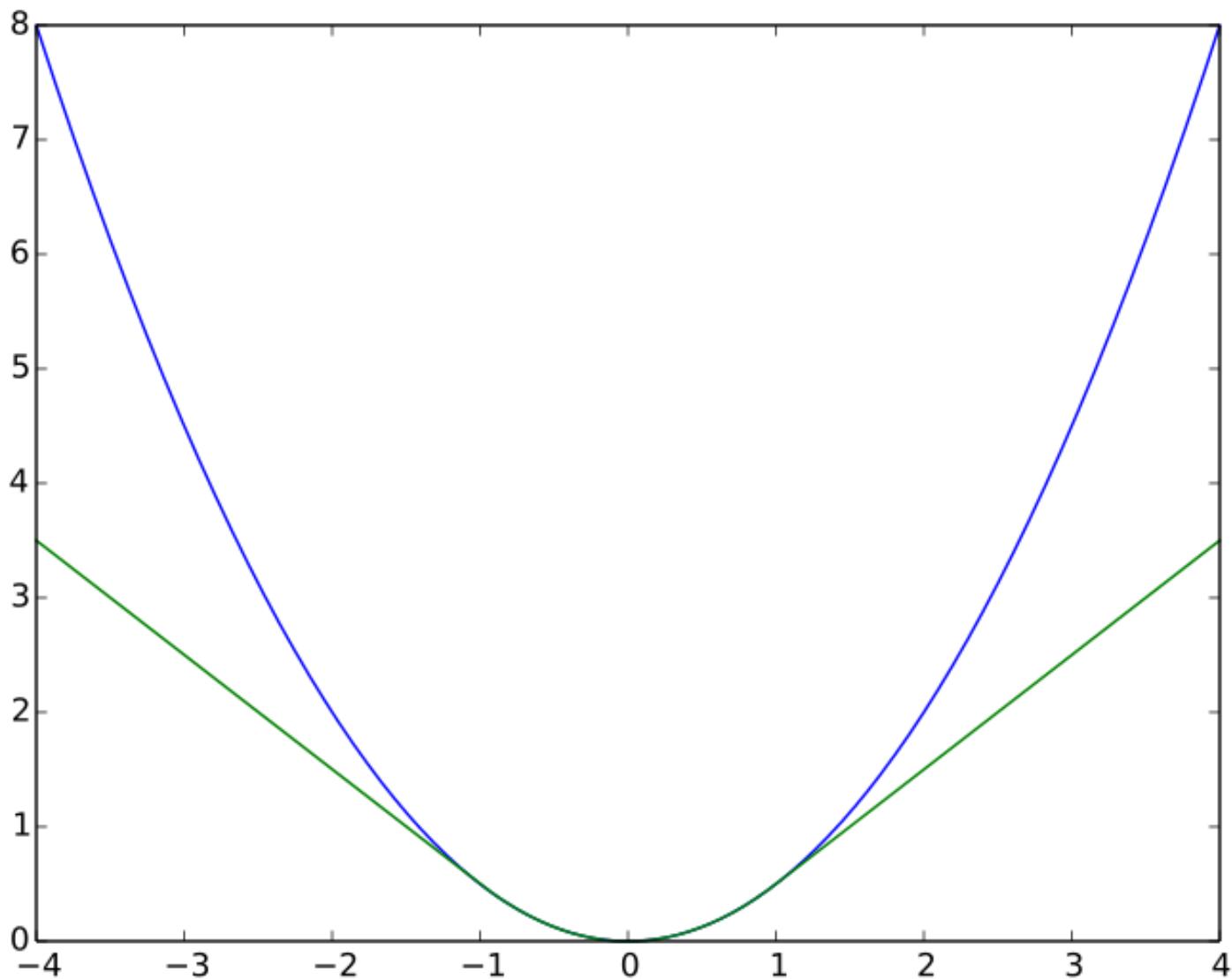
 Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the network parameters θ

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

End For

End For

≡



≡

Timeline

1986 - Backprop by Rumelhart, Hinton & Williams in multi layer nets

1989 - Q-Learning (Watkins)

1992 - Experience replay (Lin)

2010 - Tabular Double Q-Learning

...

2010's - GPUs used for neural networks

2013 - DQN

2015 - DQN (Nature)

≡

Recap

What does a value function predict?

What is general policy iteration?

How does the Bellman Equation help us learn?

For each of the three approximation methods (DP, MC, TD)

- needs an environment model?
- bootstrapped?
-

If we wanted to control using $V(s)$, what else would we need?

What makes Q-Learning off-policy?

≡

What is the deadly triad?

Recap

What does a value function predict?

- future expected discounted reward

What is general policy iteration?

- process of letting policy evaluation and policy improvement interact

How does the Bellman Equation help us learn?

- allows us to create bootstrapped targets

If we wanted to control using $V(s)$, what else would we need?

- the state transition function

What makes Q-Learning off-policy?

=

Practical



Practical

Experiments with a working DQN agent & the Open AI CartPole environment

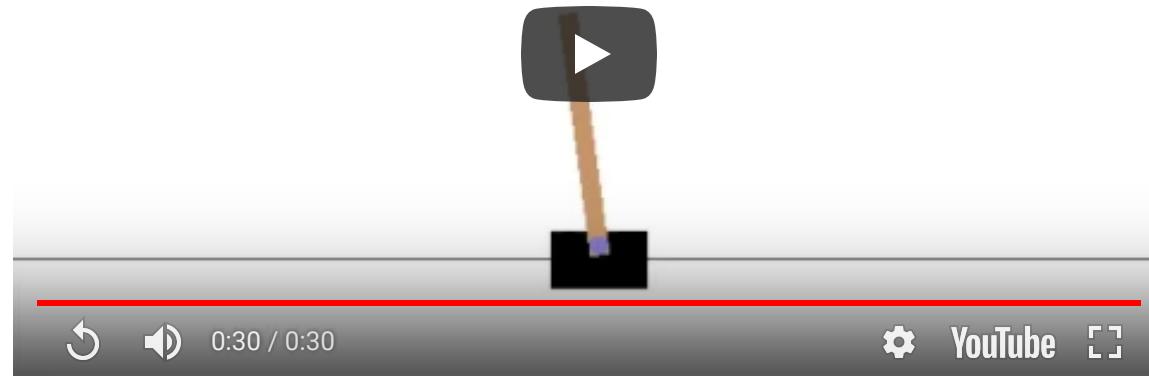


Practical

- you won't be handed a set of notebooks to shift-enter through
- given an existing code base and be expected to figure out how it works
- learn to read other peoples code in the wild
- useful for understanding open source projects
- using a working system allows you to understand the effect of hyperparameters and feel how hard RL can be!

≡

CartPole



CartPole-v0 defines "solving" as getting average reward of 195.0 over 100 consecutive trials.

Hyperparameters

Hyperparameters are configured using a dictionary

- using dictionaries to setup agents/experiments allows you to easily save them to a text file
- you can also explode a dictionary into a function

```
config = {'param1': 10, 'param2': 12}

def expt(param1, param2):
    return param1 * param2

>>> expt(**config)
120
```

Hyperparameters

What do you think the effect of changing each of these hyperparameters will be?

```
config_dict = {'discount': 0.97,  
              'tau': 0.001,  
              'total_steps': 500000,  
              'batch_size': 32,  
              'layers': (50, 50),  
              'learning_rate': 0.0001,  
              'epsilon_decay_fraction': 0.3,  
              'memory_fraction': 0.4,  
              'process_observation': False,  
              'process_target': False}
```



four
eligibility traces
prioritized experience replay
DDQN
Distributional Q-Learning
Rainbow

≡

Unified View

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

Temporal-
difference
learning

height
(depth)
of backup

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

Monte
Carlo

width
of backup

Dynamic
programming

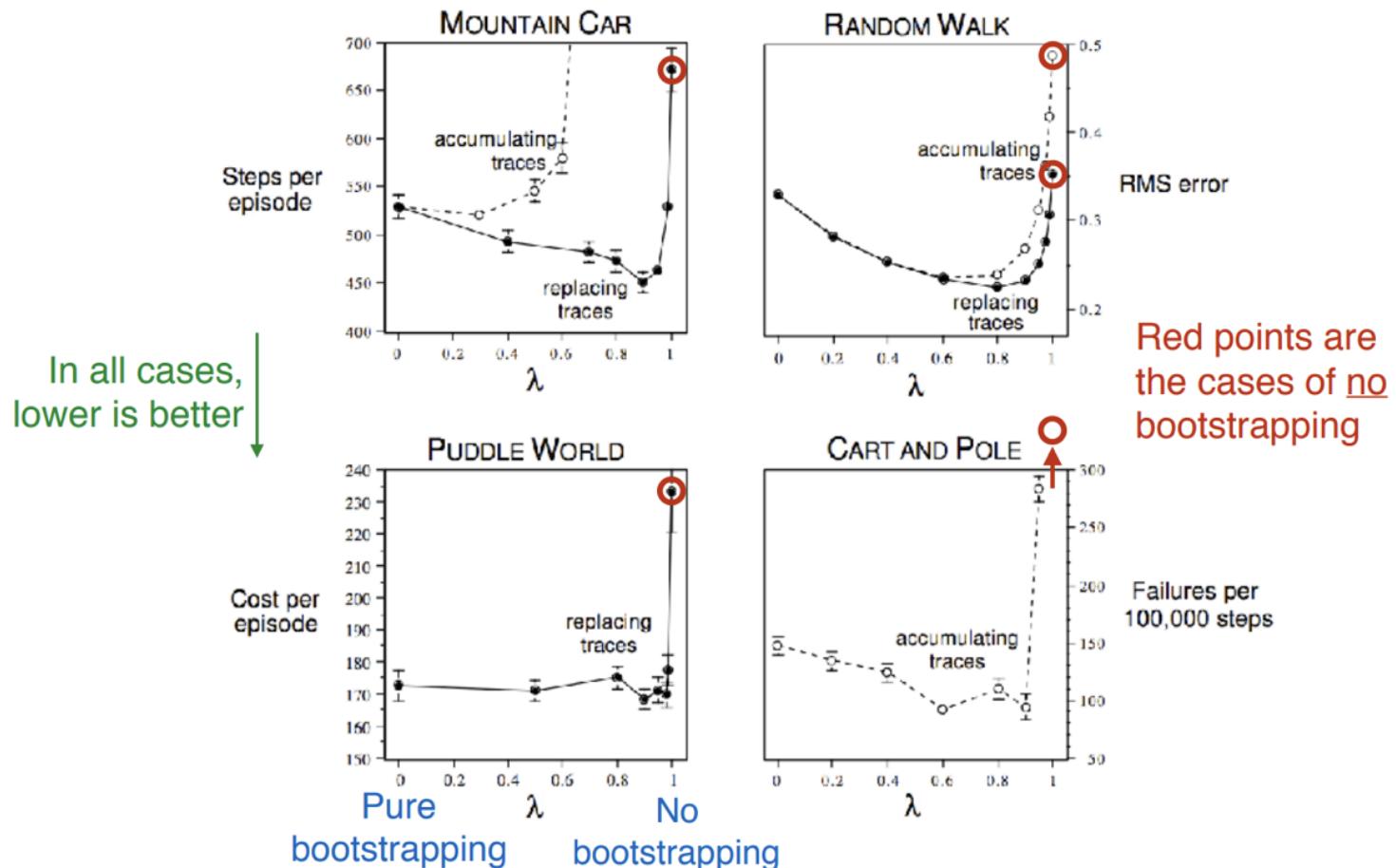
look ahead only
one step – but at
every possible
next state

Exhaustive
search

look ahead all the
way to the end
AND
every possible

4 examples of the effect of bootstrapping

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



$\lambda = 0 = \text{TD}(0)$

$\lambda = 1 = \text{full Monte Carlo}$

Conclusion – full Monte Carlo isn't
very good!

Eligibility traces

Family of methods between Temporal Difference & Monte Carlo

Eligibility traces allow us to **assign TD errors** to different states

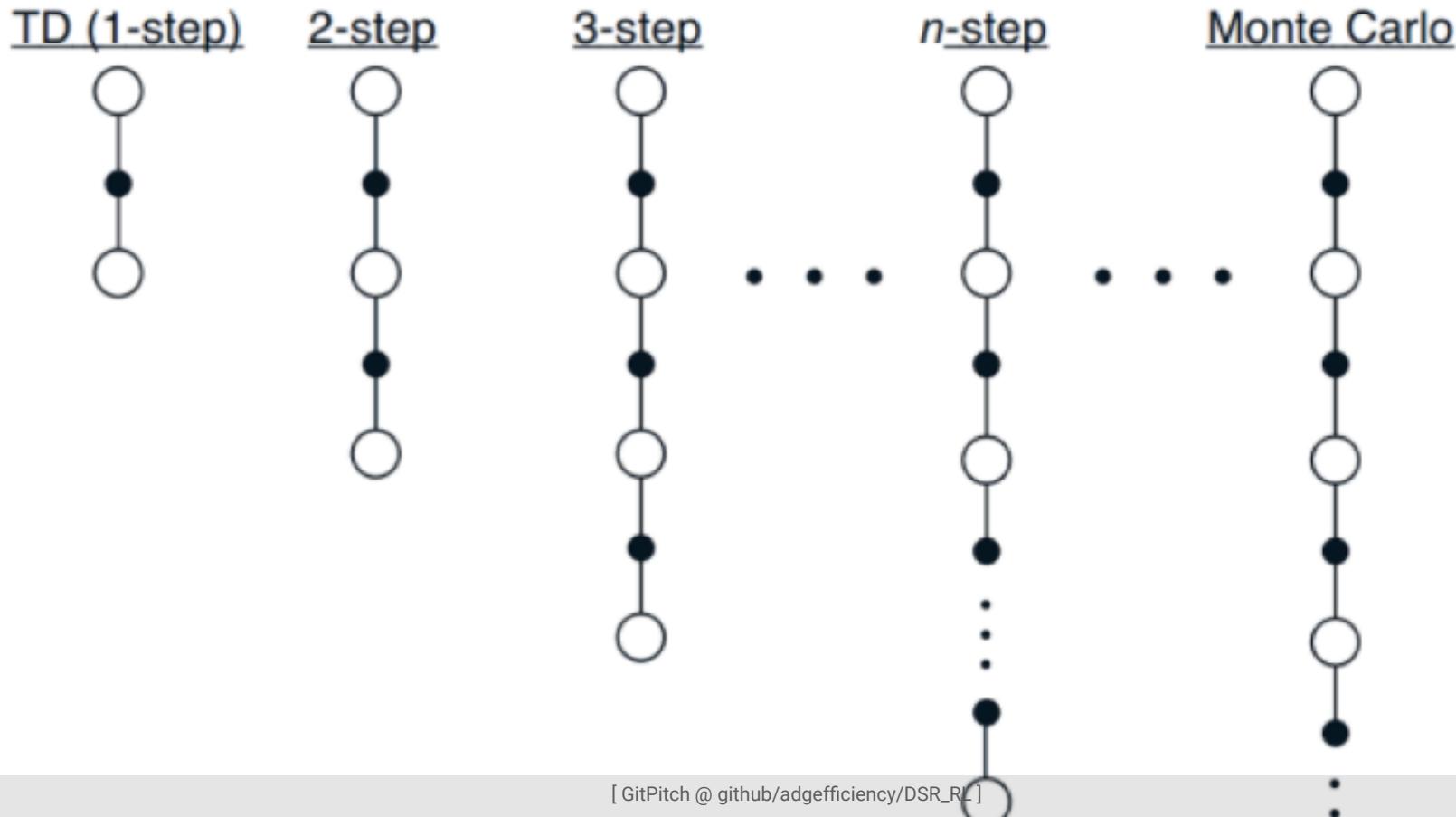
- can be useful with delayed rewards or non-Markov environments
- requires more computation
- squeezes more out of data

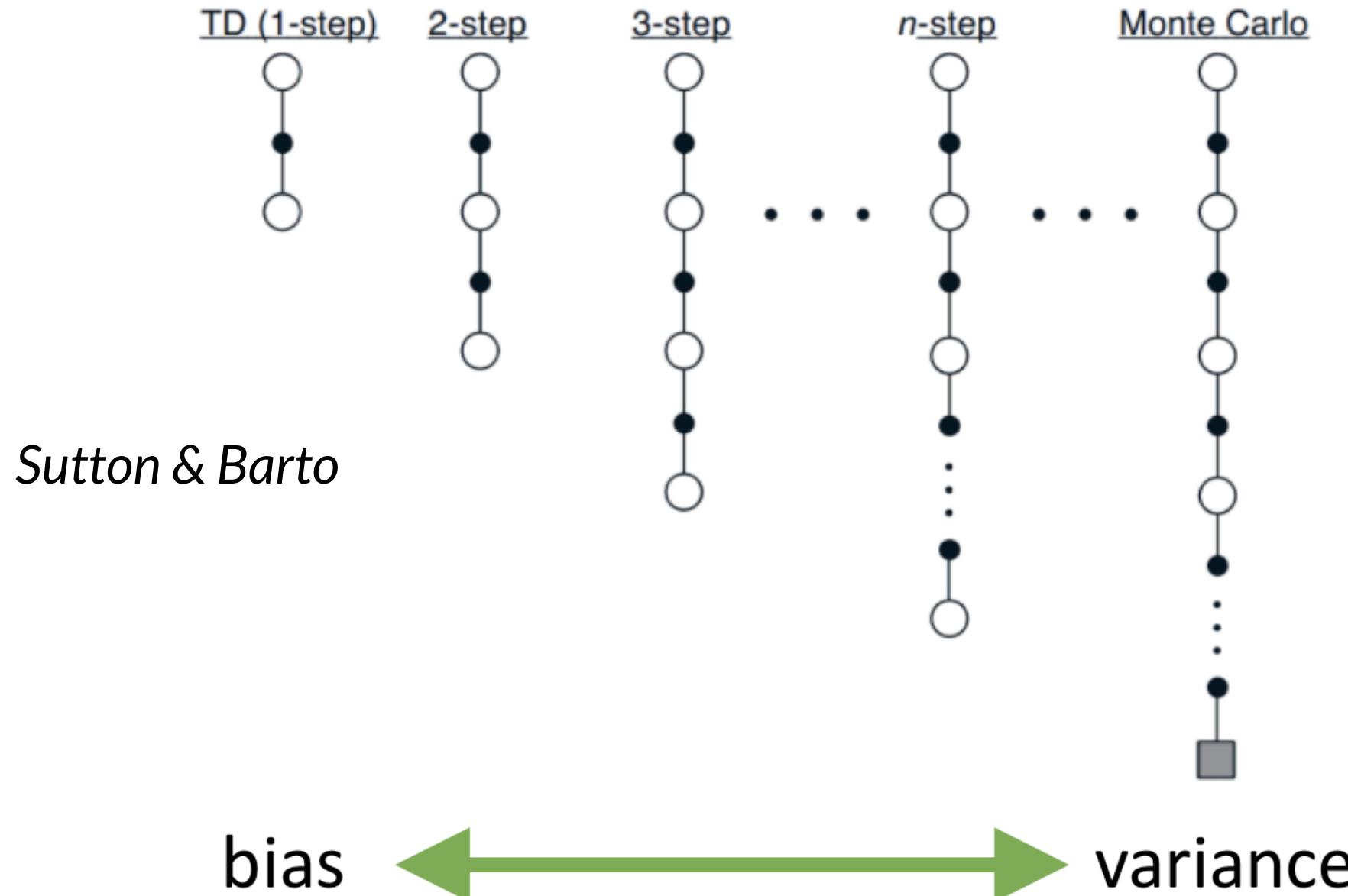
Allow us to tradeoff between bias and variance



The space between TD and MC

In between TD and MC exist a family of approximation methods known as **n-step returns**





≡

Forward and backward view

We can look at eligibility traces from two perspectives

The **forward** view is helpful for understanding the theory

The **backward** view can be put into practice

The forward view

We can decompose return into **complex backups**

- looking forward to future returns
- can use a combination of experience based and model based backups

$$R_t = \frac{1}{2}R_t^2 + \frac{1}{2}R_t^4$$

$$R_t = \frac{1}{2}TD + \frac{1}{2}MC$$

The backward view

The backward view approximates the forward view

- forward view is not practical (requires knowledge of the future)

It requires an additional variable in our agents memory

- **eligibility trace** $e_t(s)$

At each step we decay the trace according to

$$e_t(s) = \gamma \lambda e_{t-1}(s)$$

Unless we visited that state, in which case we accumulate more eligibility

$$e_t(s) = \gamma \lambda e_{t-1}(s) + 1$$

=

The backward view

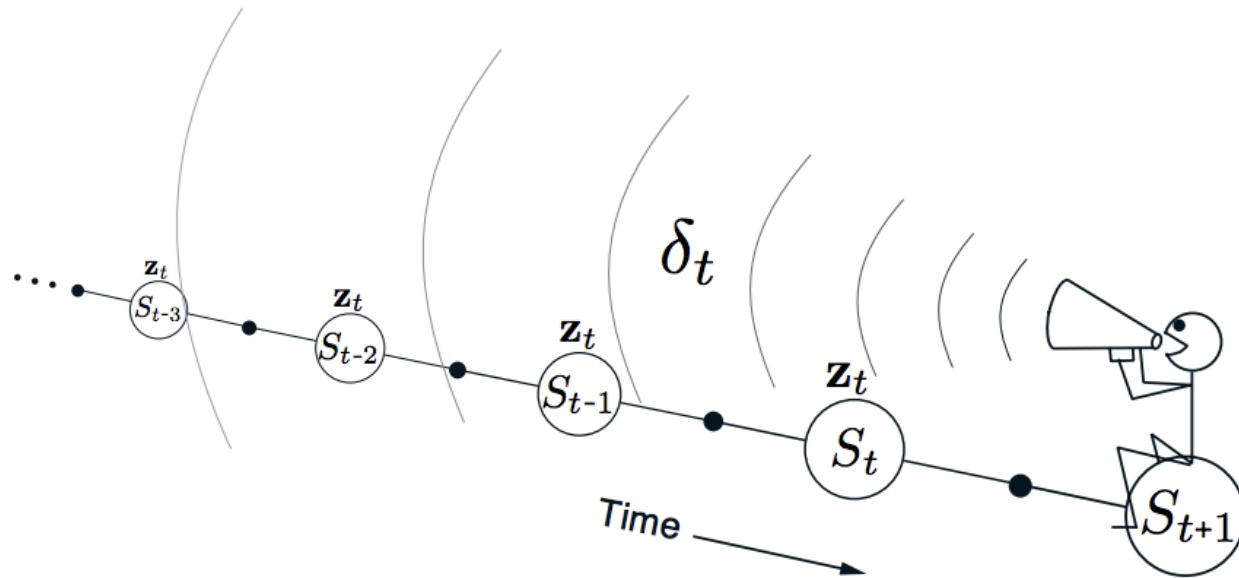


Figure 12.5: The backward or mechanistic view. Each update depends on the current TD error combined with the current eligibility traces of past events.

Sutton & Barto



Traces in a grid world



four
eligibility traces
prioritized experience replay
DDQN
Distributional Q-Learning
Rainbow

≡

Published as a conference paper at ICLR 2016

PRIORITIZED EXPERIENCE REPLAY

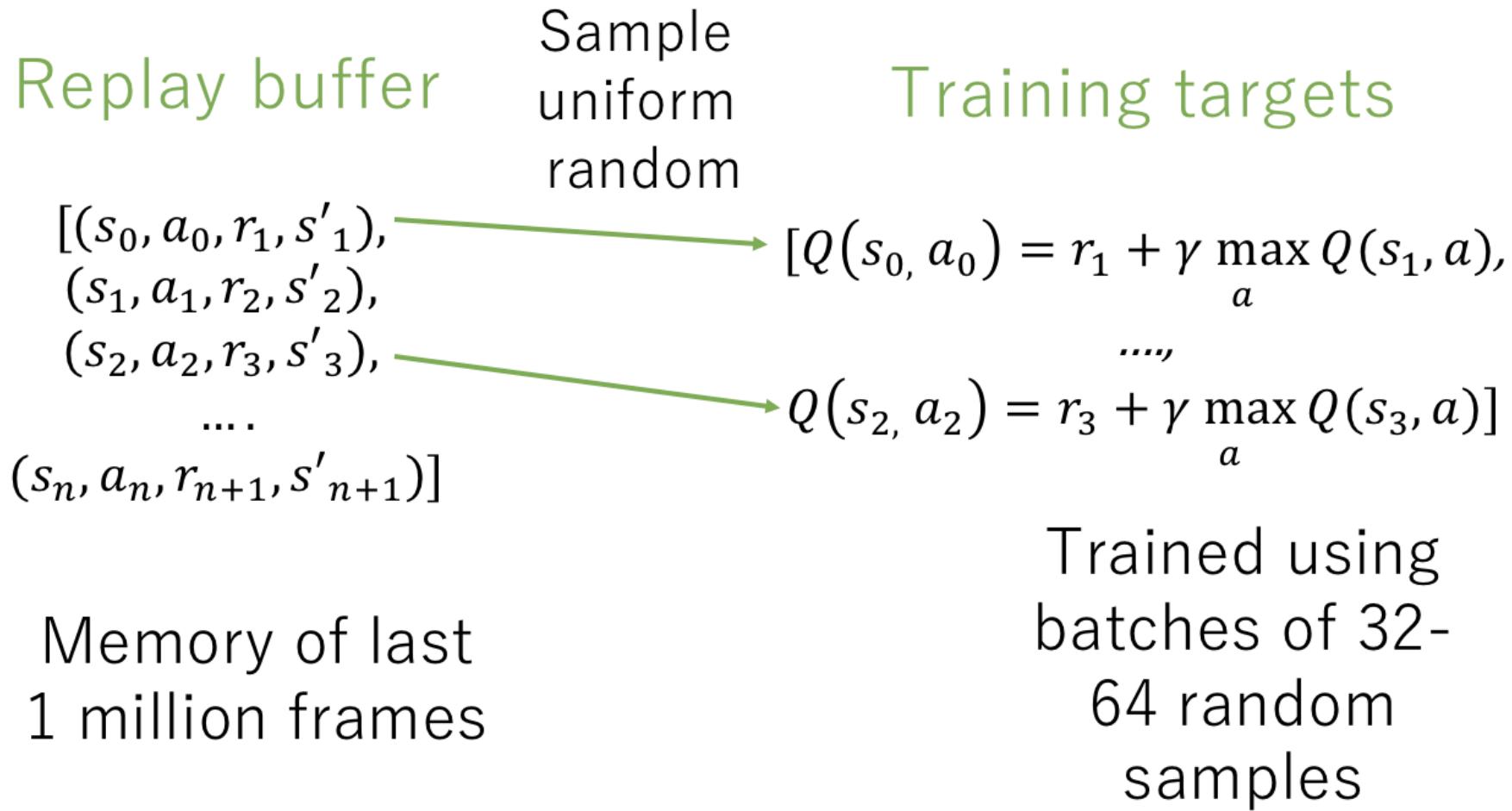
Tom Schaul, John Quan, Ioannis Antonoglou and David Silver

Google DeepMind

{schaul, johnquan, ioannisa, davidsilver}@google.com



Naive experience replay



≡

Prioritized Experience Replay

Naive experience replay randomly samples experience

- learning occurs at the same frequency as experience

Some experience is more useful for learning than others

- we can measure how useful experience is by the temporal difference error

$$\text{error} = r + \gamma Q(s', a) - Q(s, a)$$

TD error measures surprise

- this transition gave a higher or lower reward than our value function expected

≡

Prioritized Experience Replay

Non-random sampling introduces two problems

1. loss of diversity - we will only sample from high TD error experiences
2. introduce bias - non-independent sampling

Schaul et. al (2016) solves these problems by

1. loss of diversity -> make the prioritization stochastic
2. correct bias -> use importance sampling

four
eligibility traces
prioritized experience replay

DDQN

Distributional Q-Learning

Rainbow

≡

Deep Reinforcement Learning with Double Q-learning

Hado van Hasselt and Arthur Guez and David Silver
Google DeepMind

arXiv:1509.06461v3 [cs.LG] 8 Dec 2015



DDQN

DDQN = Double Deep Q-Network

- first introduced in a tabular setting in 2010
- reintroduced in the context of DQN in 2016

DDQN aims to overcome the **maximization bias** of Q-Learning

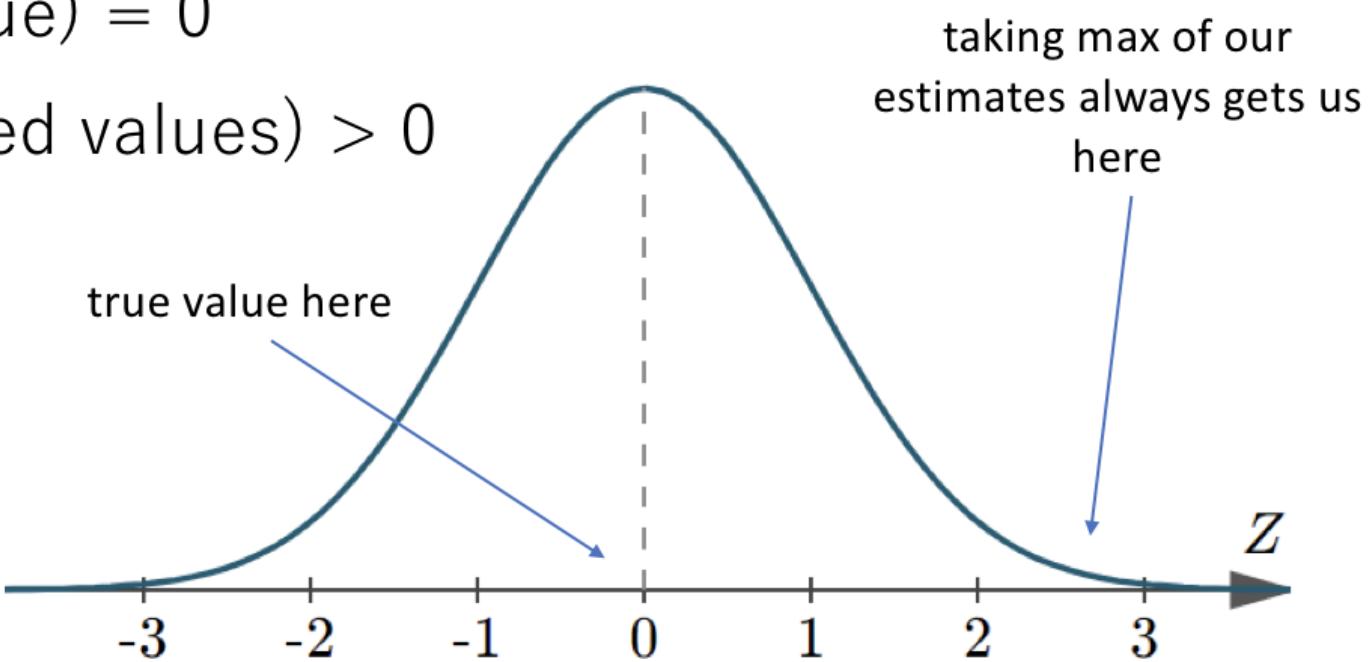
Maximization bias

Imagine a state where $Q(s, a) = 0$ for all a

Our estimates are normally distributed above and below 0

$$\max(\text{true value}) = 0$$

$$\max(\text{estimated values}) > 0$$



≡

DDQN

The DDQN modification to DQN makes use of the target network as a different function to approximate $Q(s,a)$

Original DQN target

$$r + \gamma \max_a Q(s, a; \theta^-)$$

DDQN target

$$r + \gamma Q(s', \operatorname{argmax}_a Q(s', a; \theta); \theta^-)$$

- select the action according to the online network
- quantify the value that action using the target network

≡

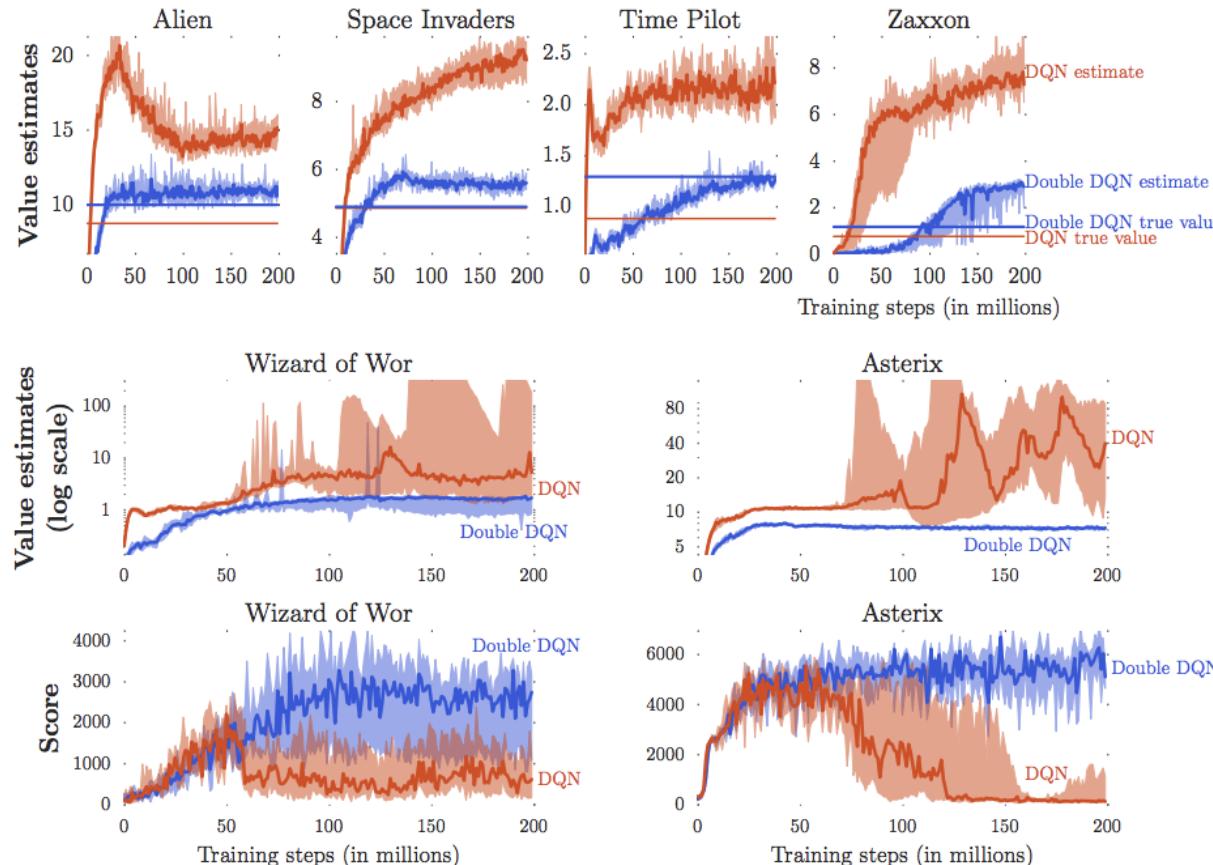


Figure 3: The **top** and **middle** rows show value estimates by DQN (orange) and Double DQN (blue) on six Atari games. The results are obtained by running DQN and Double DQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over seeds and we average the two extreme values to obtain the shaded area (i.e., 10% and 90% quantiles with linear interpolation). The straight horizontal orange (for DQN) and blue (for Double DQN) lines in the top row are computed by running the corresponding agents after learning concluded, and averaging the actual discounted return obtained from each visited state. These straight lines would match the learning curves at the right side of the plots if there is no bias. The **middle** row shows the value estimates (in log scale) for two games in which DQN's overoptimism is quite extreme. The **bottom** row shows the detrimental effect of this on the score achieved by the agent as it is evaluated during training: the scores drop when the overestimations begin. Learning with Double DQN is much more stable.

four
eligibility traces
prioritized experience replay
DDQN
Distributional Q-Learning
Rainbow

≡

A Distributional Perspective on Reinforcement Learning

Marc G. Bellemare^{* 1} Will Dabney^{* 1} Rémi Munos¹

21 Jul 2017



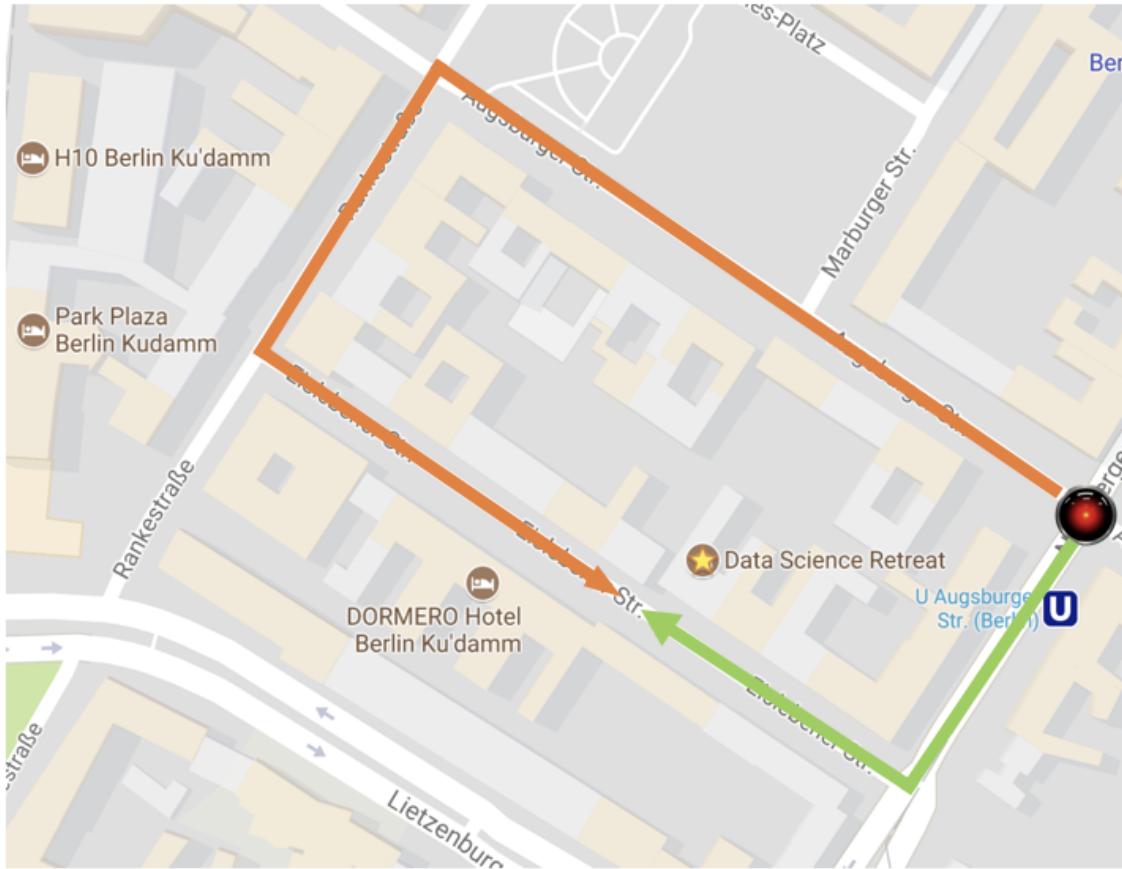
Beyond the expectation

All the reinforcement learning we have seen focuses on the expectation (i.e. the mean)

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

In 2017 DeepMind introduced the idea of the value distribution
State of the art results on Atari (at the time - Rainbow is currently SOTA)

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!

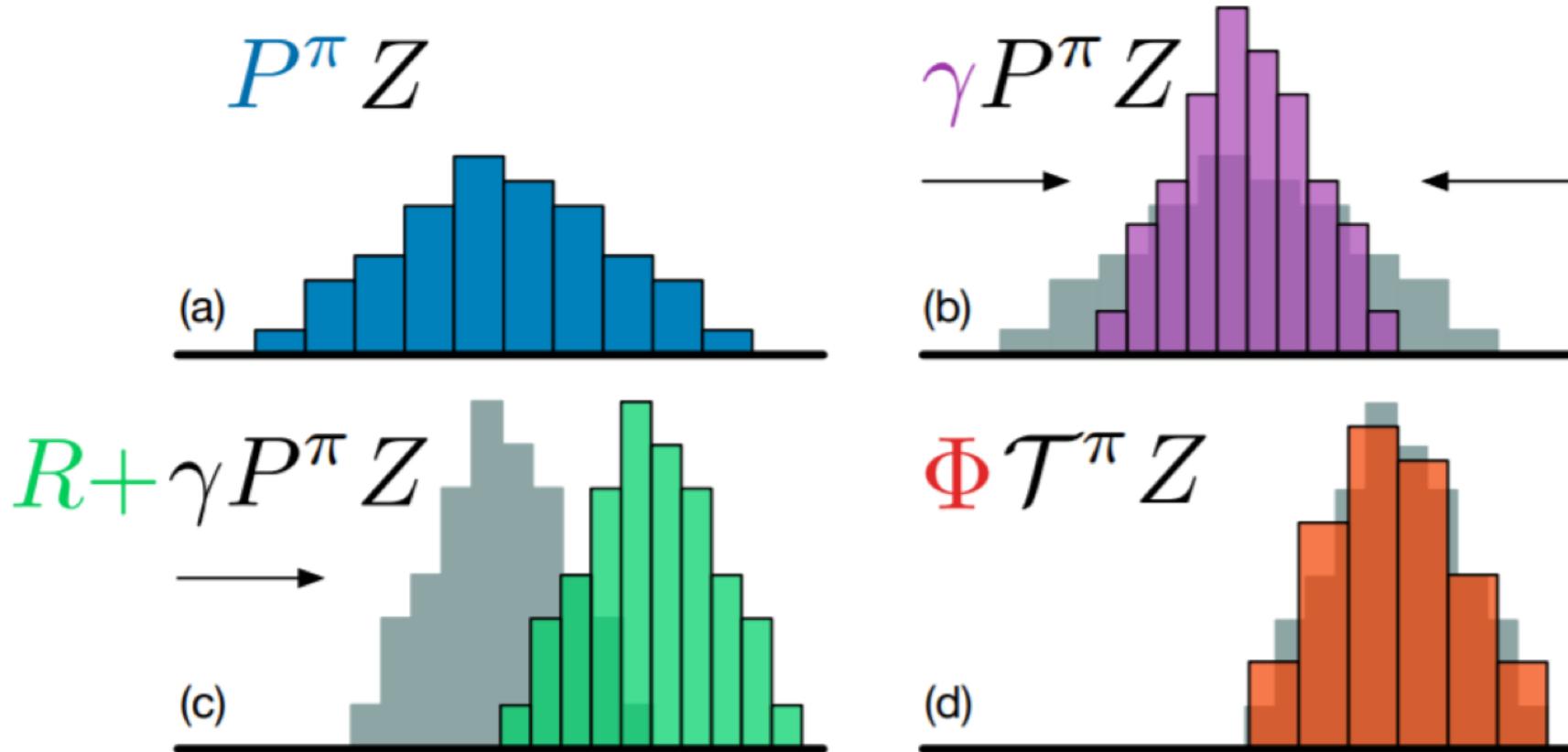


Figure 1. A distributional Bellman operator with a deterministic reward function: (a) Next state distribution under policy π , (b) Discounting shrinks the distribution towards 0, (c) The reward shifts it, and (d) Projection step (Section 4).

| | 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).

four
eligibility traces
prioritized experience replay
DDQN
Distributional Q-Learning
Rainbow

≡

Rainbow: Combining Improvements in Deep Reinforcement Learning

Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, David Silver

(Submitted on 6 Oct 2017)

The deep reinforcement learning community has made several independent improvements to the DQN algorithm. However, it is unclear which of these extensions are complementary and can be fruitfully combined. This paper examines six extensions to the DQN algorithm and empirically studies their combination. Our experiments show that the combination provides state-of-the-art performance on the Atari 2600 benchmark, both in terms of data efficiency and final performance. We also provide results from a detailed ablation study that shows the contribution of each component to overall performance.

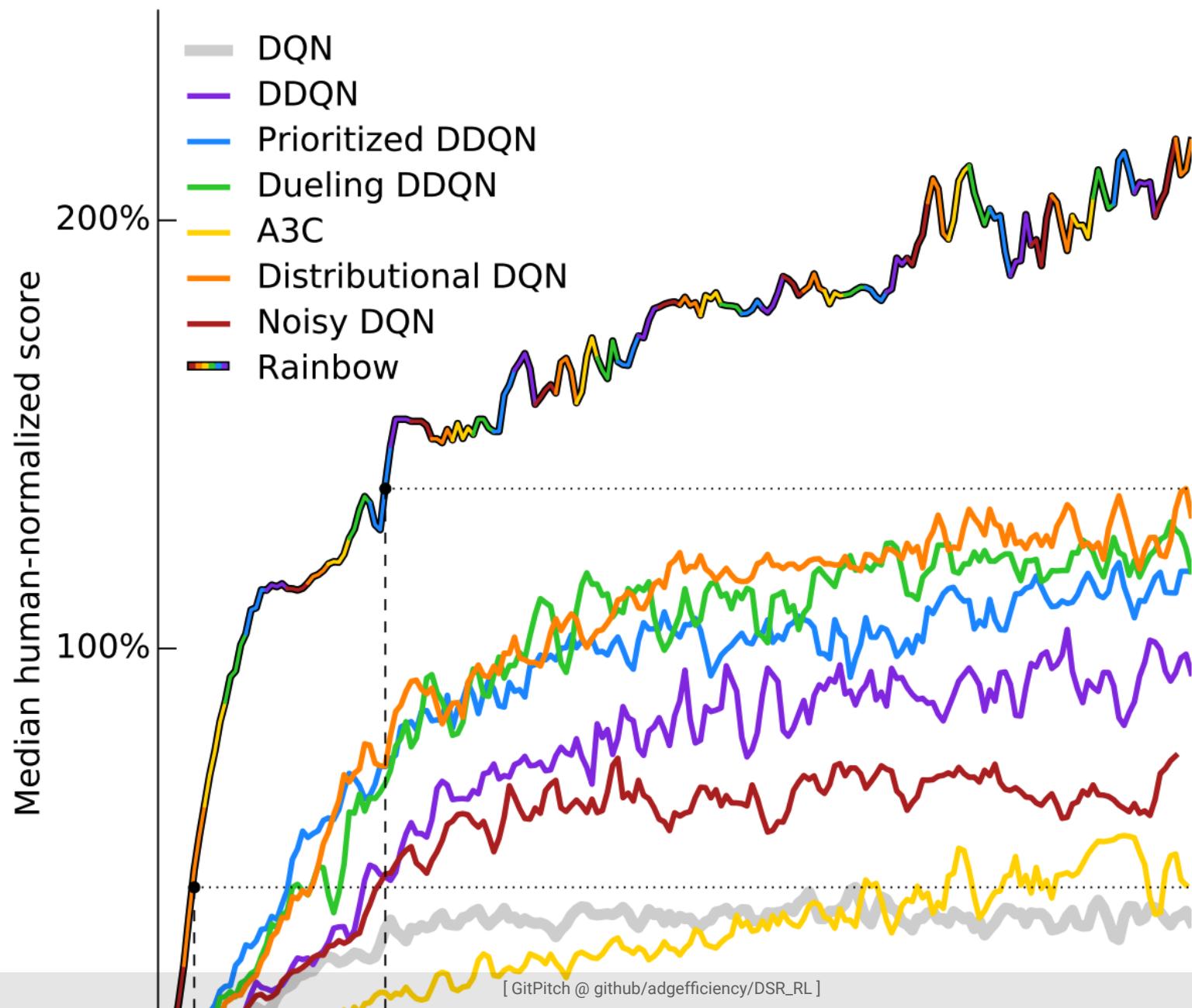


Rainbow

All the various improvements to DQN address different issues

- DDQN - overestimation bias
- prioritized experience replay - sample efficiency
- dueling - generalize across actions
- multi-step bootstrap targets - bias variance tradeoff
- distributional Q-learning - learn categorical distribution of $Q(s, a)$
- noisy DQN - stochastic layers for exploration

Rainbow combines these improvements



Evaluation Methodology. We evaluated all agents on 57 Atari 2600 games from the arcade learning environment (Bellemare et al. 2013). We follow the training and evaluation procedures of Mnih et al. (2015) and van Hasselt et al. (2016). The average scores of the agent are evaluated during training, every 1M steps in the environment, by suspending learning and evaluating the latest agent for 500K frames. Episodes are truncated at 108K frames (or 30 minutes of simulated play), as in van Hasselt et al. (2016).

| Parameter | Value |
|--|-----------------------|
| Min history to start learning | 80K frames |
| Adam learning rate | 0.0000625 |
| Exploration ϵ | 0.0 |
| Noisy Nets σ_0 | 0.5 |
| Target Network Period | 32K frames |
| Adam ϵ | 1.5×10^{-4} |
| Prioritization type | proportional |
| Prioritization exponent ω | 0.5 |
| Prioritization importance sampling β | $0.4 \rightarrow 1.0$ |
| Multi-step returns n | 3 |
| Distributional atoms | 51 |
| Distributional min/max values | $[-10, 10]$ |

=

Table 1: Rainbow hyper-parameters

Learning speed. As in the original DQN setup, we ran each agent on a single GPU. The 7M frames required to match DQN’s final performance correspond to less than 10 hours of wall-clock time. A full run of 200M frames corresponds to approximately 10 days, and this varies by less than 20% between all of the discussed variants. The litera-

| Agent | no-ops | human starts |
|----------------------|--------|--------------|
| DQN | 79% | 68% |
| DDQN (*) | 117% | 110% |
| Prioritized DDQN (*) | 140% | 128% |
| Dueling DDQN (*) | 151% | 117% |
| A3C (*) | - | 116% |
| Noisy DQN | 118% | 102% |
| Distributional DQN | 164% | 125% |
| Rainbow | 223% | 153% |

Recap

Eligibility traces allow us to trade bias and variance

Eligibility traces assign the temporal difference to different states

What two problems does prioritized experience replay introduce?

- lack of diversity, solved by making sampling stochastic
- introduces bias, solved using importance sampling

What problem does DDQN address?

- maximization bias



five motivations for policy gradients

introduction

the score function

REINFORCE

Actor-Critic

≡

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 we generated a policy from a value function

$$a = \underset{a}{\operatorname{argmax}} Q(s, a)$$

In policy gradients we **parameterize a policy directly**

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

≡

John Schulan - Berkley, Open AI



≡

Motivation - stochastic policies



A deterministic policy (i.e. always rock) is easily exploited

≡

A stochastic policy means exploration is built into the policy

Motivation - high dimensional action spaces

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

We also lose shape of the action space by discretization

≡

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)]
```

Motivation - optimize return directly

When learning value functions our optimizer is working towards improving the predictive accuracy of the value function

- our gradients point in the direction of predicting return

This isn't what we really care about - we care about maximizing return

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)

≡

Motivation - simplicity

Sometimes it's easier to pick an action

- rather than to quantify return for each action, then pick action

Policy gradients are more general and versatile

More compatible with recurrent neural networks

Policy gradients versus value functions

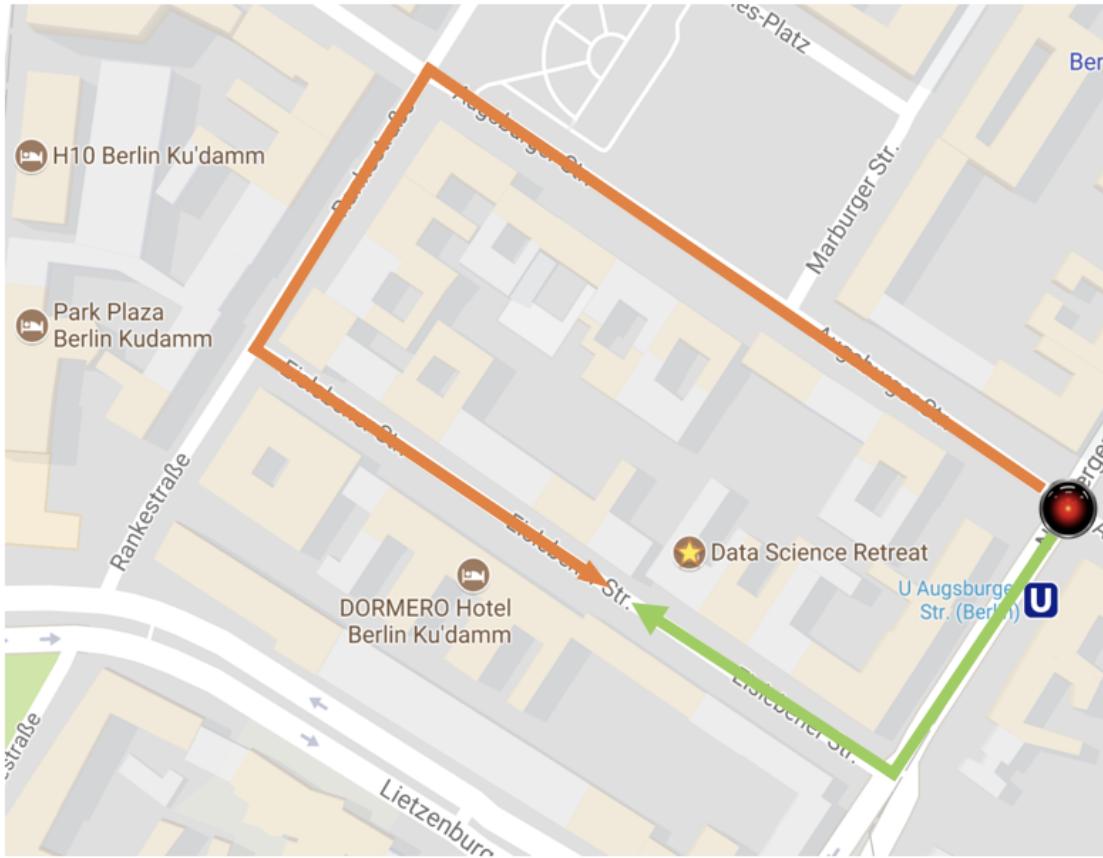
Policy gradients

- optimize return directly
- work in continuous and discrete action spaces
- works better in high-dimensional action spaces

Value functions

- optimize value function accuracy
- off policy learning
- exploration
- better sample efficiency

≡



$$Q(s, a_1) = 10 \text{ min}$$
$$Q(s, a_2) = 5 \text{ min}$$

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

VS

$a_2 \sim \pi(s)$

five motivations for policy gradients

introduction

the score function

REINFORCE

Actor-Critic

=

Deterministic Policy Gradient

Parameterizing policies

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

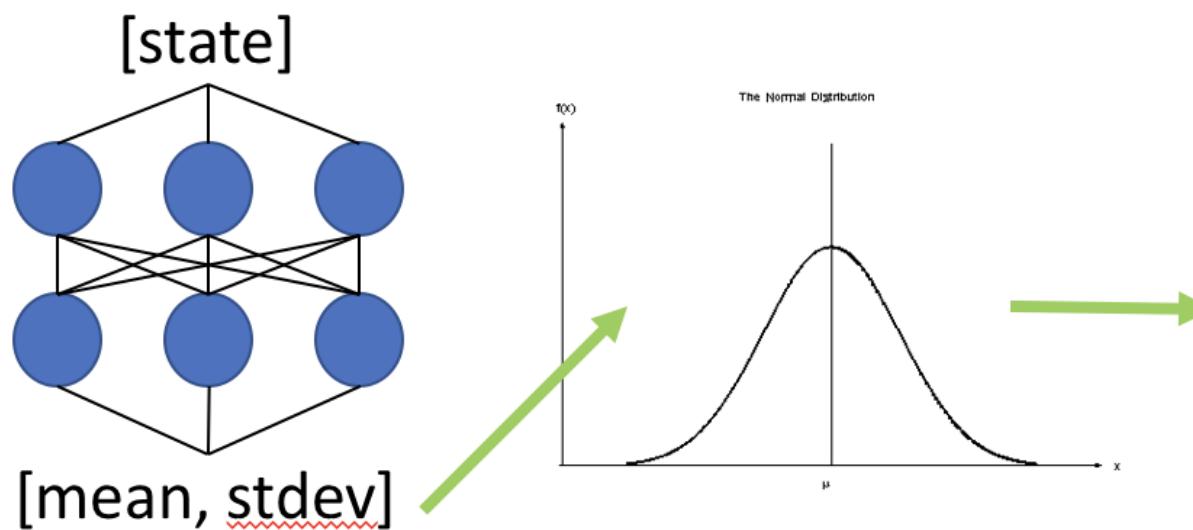


Parameterizing policies

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

continuous action space

output layer = mean & stdev



Sample an
action from
the
distribution

Policy gradients without equations

We have a parameterized policy

- a neural network that outputs a distribution over actions

How do we improve it - how do we learn?

- 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 the expected reward*

≡

Policy gradients with a few equations

Our policy $\pi(a_t|s_t; \theta)$ is a **probability 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*

$$\nabla_{\theta} \mathbf{E}[G_t] = \mathbf{E}[\nabla_{\theta} \log \pi(a|s) \cdot G_t]$$

We can figure out how to change our parameters without actually knowing the reward function itself

≡

The score function in statistics

The **score function** comes from using the log-likelihood ratio trick

The score function allows us to get the gradient of a function by **taking an expectation**

Expectataions are averages

- use sample based methods to approximate them

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

≡

Deriving the score function

$$\begin{aligned}\nabla_{\theta} E_x[f(x)] &= \nabla_{\theta} \sum_x p(x) f(x) && \text{definition of expectation} \\ &= \sum_x \nabla_{\theta} p(x) f(x) && \text{swap sum and gradient} \\ &= \sum_x p(x) \frac{\nabla_{\theta} p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\ &= \sum_x p(x) \nabla_{\theta} \log p(x) f(x) && \text{use the fact that } \nabla_{\theta} \log(z) = \frac{1}{z} \nabla_{\theta} z \\ &= E_x[f(x) \nabla_{\theta} \log p(x)] && \text{definition of expectation}\end{aligned}$$

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



The score function in reinforcement learning

$$\nabla_{\theta} \mathbf{E}[G_t] = \mathbf{E}[\nabla_{\theta} \log \pi(a|s) \cdot G_t]$$

gradient of return = expectation of the gradient of the policy * return

The RHS is an expectation - we can estimate it by sampling

The expectation is made up of things we can sample from

- we can sample from our policy
- we can sample the return (from experience)

≡

Training a policy

We use the score function to get the gradient, then follow the gradient

gradient = log(probability of action) * return

gradient = log(policy) * return

The score function limits us to on-policy learning

- we need to calculate the log probability of the action taken by the policy

Policy gradient intuition

$$\nabla_{\theta} \mathbf{E}[G_t] = \mathbf{E}[\nabla_{\theta} \log \pi(a|s) \cdot G_t]$$

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

- how probable was the action we picked
- we want to reinforce actions we thought were good

G_t

- how good was that action
- we want to reinforce actions that were actually good

≡

REINFORCE

Different methods to approximate the return G_t

We can use a Monte Carlo estimate - this is known as
REINFORCE

Using a Monte Carlo approach comes with all the problems we
saw earlier

- high variance
- no online learning
- requires episodic environment

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

≡

Baseline

We can introduce a baseline function

- this reduces variance without introducing bias
- a natural baseline is the value function (weights w).

$$\log \pi(a_t | s_t; \theta) \cdot (G_t - B(s_t; w))$$

This also gives rise to the concept of **advantage**

- how much better this action is than the average action (policy & env dependent)

$$A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$$

≡

Actor-Critic



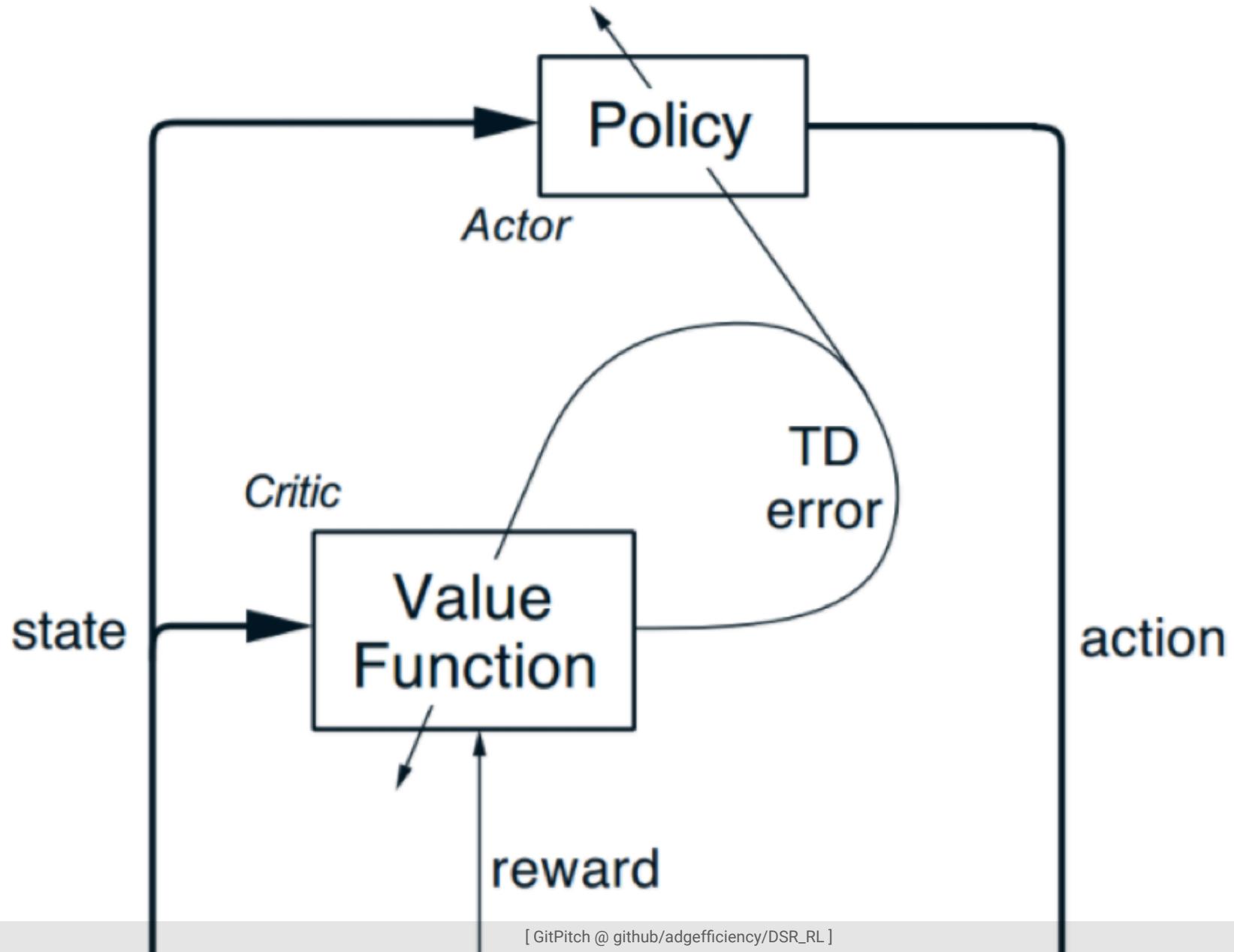
Actor-Critic

Actor-Critic brings together value functions and policy gradients

We parameterize two functions

- **actor** = policy
- **critic** = value function

We update our actor (i.e. the behaviour policy) in the direction suggested by the critic



Actor-Critic Algorithm



five
motivations for policy gradients
introduction
the score function
REINFORCE
Actor-Critic

=

Deterministic Policy Gradient

Deterministic Policy Gradient Algorithms

David Silver

DeepMind Technologies, London, UK

DAVID@DEEPMIND.COM

Guy Lever

University College London, UK

GUY.LEVER@UCL.AC.UK

Nicolas Heess, Thomas Degrif, Daan Wierstra, Martin Riedmiller

DeepMind Technologies, London, UK

*@DEEPMIND.COM

Proceedings of the 31st International Conference on Machine Learning, Beijing, China, 2014. JMLR: W&CP volume 32. Copyright 2014 by the author(s).



Deterministic Policy Gradient

Actor Critic

Deterministic policy

- more efficient than stochastic

Continuous action spaces

Off-policy learning

Uses experience replay

Uses target networks

≡

Stochastic vs deterministic policies

Stochastic policy is a probability distribution over actions

Actions are selected by sampling from this distribution

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

$$a \sim \pi_\theta(a|s)$$

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
- estimates $Q(s, a)$

≡

Gradients



Updating policy weights

The gradient

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

The update function

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

α learning rate

Q^w action value function parameterized by weights w

≡

DPG results



five
motivations for policy gradients

introduction

the score function

REINFORCE

Actor-Critic

=

Deterministic Policy Gradient

Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih¹

VMNIH@GOOGLE.COM

Adrià Puigdomènech Badia¹

ADRIAP@GOOGLE.COM

Mehdi Mirza^{1,2}

MIRZAMOM@IRO.UMONTREAL.CA

Alex Graves¹

GRAVESEA@GOOGLE.COM

Tim Harley¹

THARLEY@GOOGLE.COM

Timothy P. Lillicrap¹

COUNTZERO@GOOGLE.COM

David Silver¹

DAVIDSILVER@GOOGLE.COM

Koray Kavukcuoglu¹

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

We saw earlier that experience replay is used to make learning more stable & 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

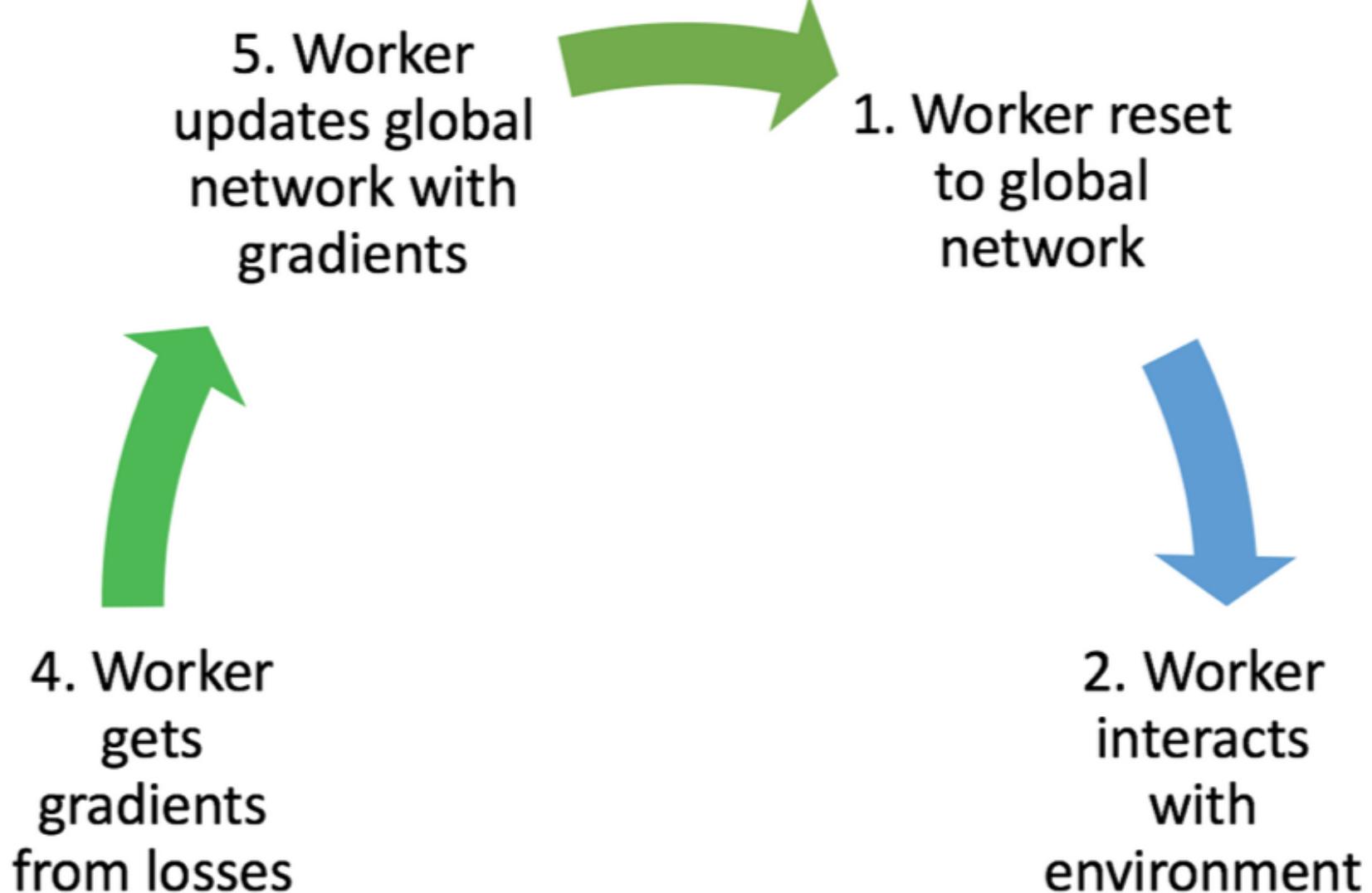
Advantage = the advantage function

$$A_\pi(s_t, a_t) = Q_\pi(s_t, a_t) - V_\pi(s_t)$$

How much better an action is than the average action followed by the policy

≡

A3C algorithm



≡

Recap

Motivations for policy gradients

- optimize what we care about directly
- high dimensional spaces

How do we parameterize a continuous action

- output of network is the mean & variance of a Gaussian

What does the log-likelihood trick allow us to do

- get the gradient of an unknown function

Two intuitions behind the score function

- make probable actions more probable
- make high return actions more probable

=

six

AlphaGo

AlphaGo Zero

Residual networks



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

David Silver^{1*}, Aja Huang^{1*}, Chris J. Maddison¹, Arthur Guez¹, Laurent Sifre¹, George van den Driessche¹, Julian Schrittwieser¹, Ioannis Antonoglou¹, Veda Panneershelvam¹, Marc Lanctot¹, Sander Dieleman¹, Dominik Grewe¹, John Nham², Nal Kalchbrenner¹, Ilya Sutskever², Timothy Lillicrap¹, Madeleine Leach¹, Koray Kavukcuoglu¹, Thore Graepel¹ & Demis Hassabis¹

NATURE | VOL 529 | 28 JANUARY 2016





≡

IBM Deep Blue

First defeat of a world chess champion by a machine in 1997



Deep Blue vs AlphaGo

Deep Blue was handcrafted by programmers & chess grandmasters

AlphaGo *learnt* from 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



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

Learning

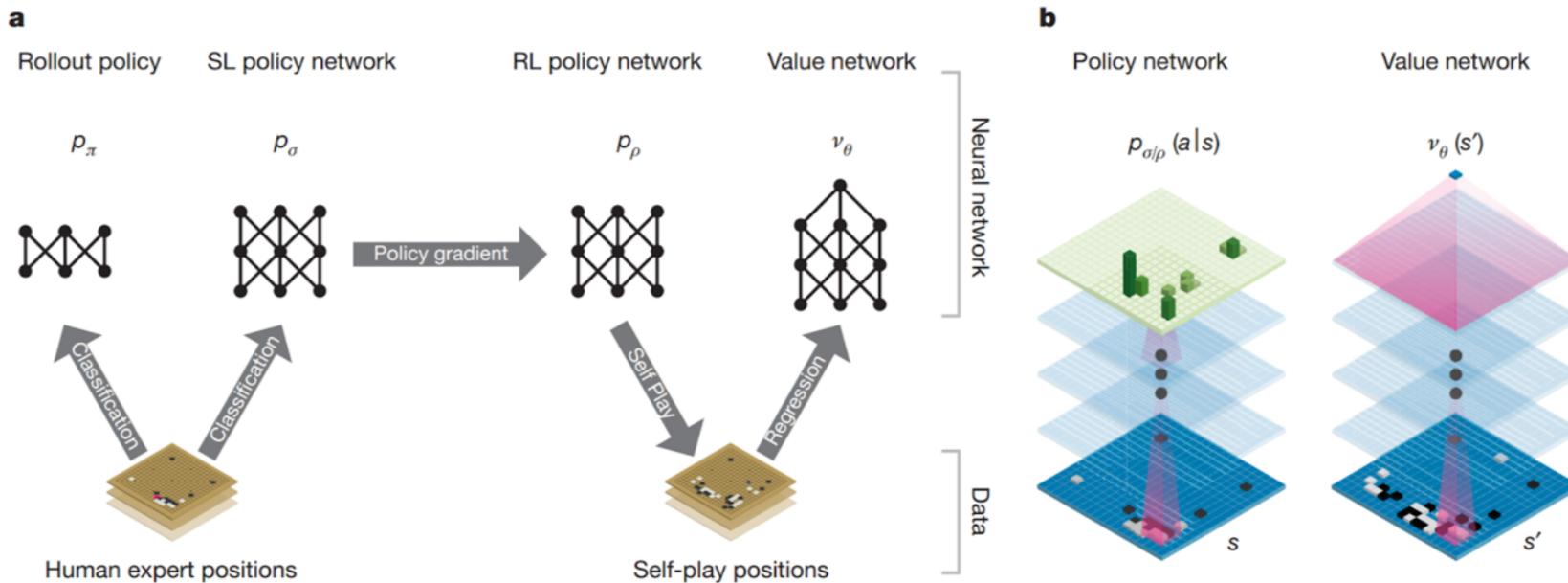


Figure 1 | Neural network training pipeline and architecture. **a**, A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a data set of positions. A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, 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

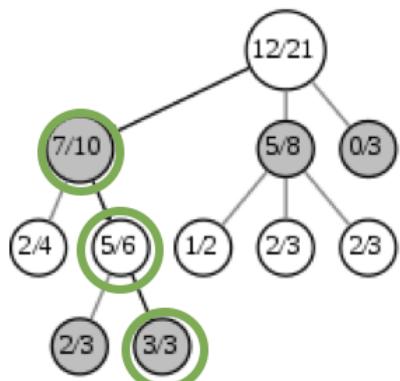
Planning algorithm

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



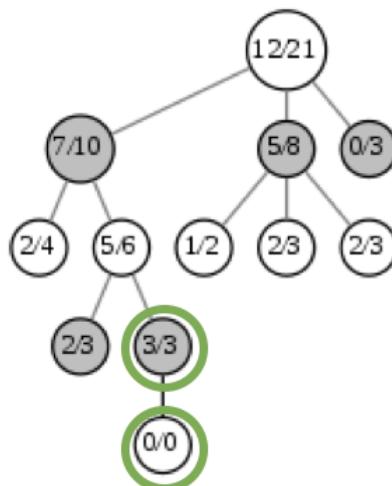
Monte Carlo Tree Search

Selection



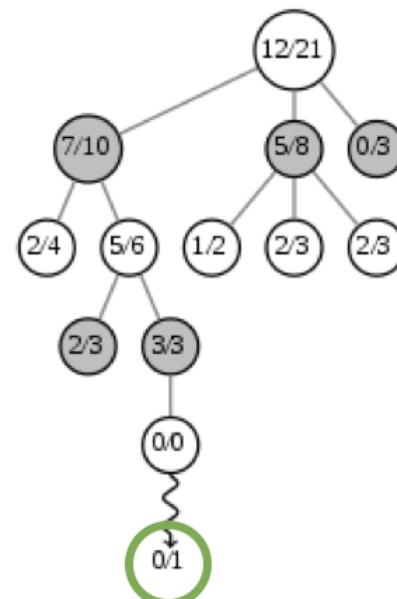
Select nodes based
on statistics
collected so far

Expansion



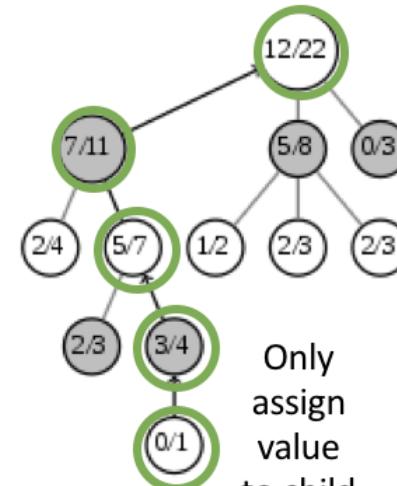
If leaf node:
create & the select
a new child node

Simulation



Rollout to the end
of the game
(simulated)

Backpropagation



Backup the result
through the tree to
the root state

Monte Carlo Tree Search in AlphaGo

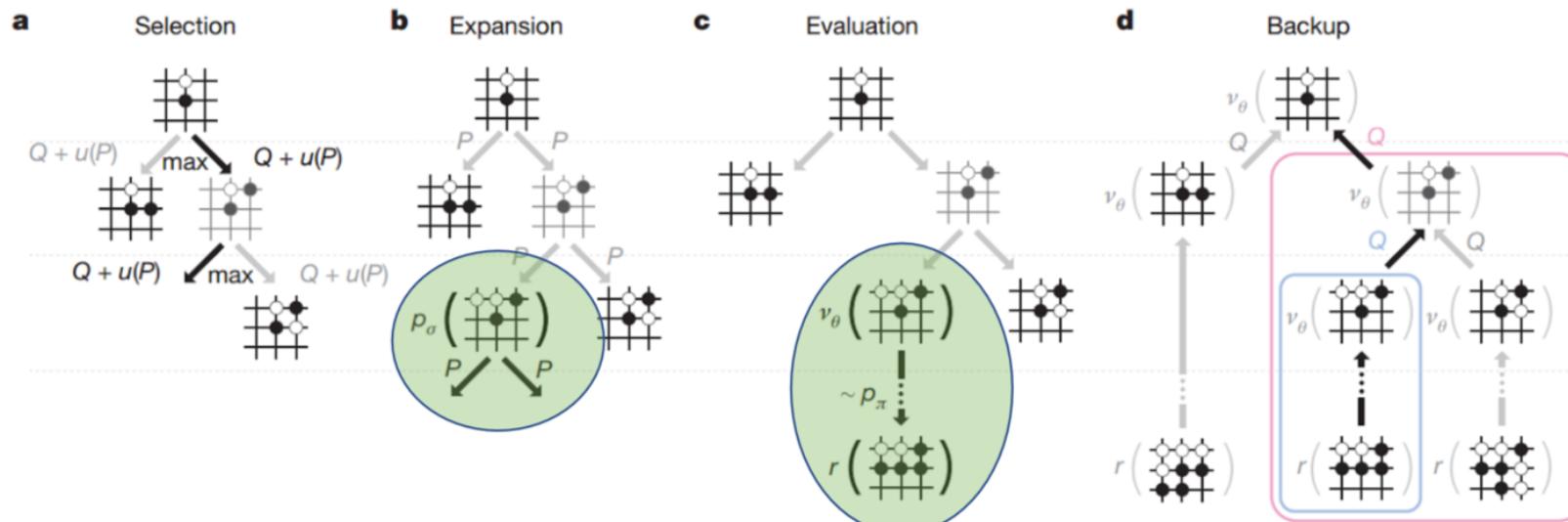


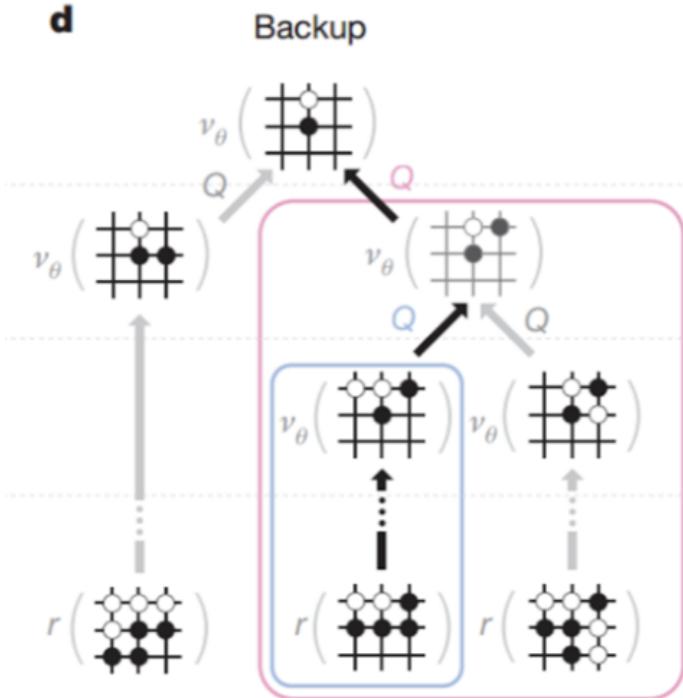
Figure 3 | Monte Carlo tree search in AlphaGo. **a**, Each simulation traverses the tree by selecting the edge with maximum action value Q , plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy 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.

Monte Carlo Tree Search in AlphaGo



Monte Carlo Tree Search in AlphaGo



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

Monte Carlo Tree Search in AlphaGo



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 $\approx 50\text{ W}$) 1080 ti = 250 W

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

six

AlphaGo

AlphaGo Zero

Residual networks

≡

ARTICLE

doi:10.1038/nature24270

Mastering the game of Go without human knowledge

David Silver^{1*}, Julian Schrittwieser^{1*}, Karen Simonyan^{1*}, Ioannis Antonoglou¹, Aja Huang¹, Arthur Guez¹, Thomas Hubert¹, Lucas Baker¹, Matthew Lai¹, Adrian Bolton¹, Yutian Chen¹, Timothy Lillicrap¹, Fan Hui¹, Laurent Sifre¹, George van den Driessche¹, Thore Graepel¹ & Demis Hassabis¹

¹DeepMind, 5 New Street Square, London EC4A 3TW, UK.

*These authors contributed equally to this work.

354 | NATURE | VOL 550 | 19 OCTOBER 2017



Key ideas in AlphaGo Zero

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



AlphaGo Zero learning curves



AlphaGo Zero innovations

Learns using only self play

- no learning from human expert games
- no feature engineering
- learn purely from board positions

Single neural network

- combine the policy & value networks

MCTS only during acting (not during learning)

Use of residual networks

≡

AlphaGo Zero acting & learning



Search in AlphaGo Zero

Policy evaluation

Policy is evaluated through self play

This creates high quality training signals - the game result

Policy improvement

MCTS is used during acting to create the improved policy

The improved policy generated during acting becomes the target policy during training

Keynote David Silver NIPS 2017 Deep Reinforcement Learning
Symposium AlphaZero



six

AlphaGo

AlphaGo Zero

Residual networks



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 $H(x)$

Instead of learning $F(x) = H(x)$

We learn the residual $F(x) = H(x) - x$

And can get $H(x) = F(x) + x$

≡

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](#)



Recap

How does AlphaGo reduce search width

- policy network to focus on high probability states

How does AlphaGo reduce search depth

- a value network

MCTS is a planning algorithm - what does AlphaGo use for an environment model?

- the linear fast rollout policy + game rules

Innovations in AlphaGo Zero

- combining the policy and value network
- using MCTS during acting to create targets to learn from

≡

seven practical concerns



Should I use reinforcement learning for my problem?

What is the action space

- what can the agent choose to do
- does the action change the environment
- continuous or discrete

What is the reward function

- does it incentivize behaviour

It is a complex problem

- classical optimization techniques such as linear programming or cross entropy may offer a simpler solution

=

Reinforcement learning is hard

Debugging implementations is hard

- very easy to have subtle bugs that don't break your code

Tuning hyperparameters is hard

- tuning hyperparameters can also cover over bugs!

Results will succeed and fail over different random seeds (same hyperparameters!)

Machine learning is an empirical science, where the ability to do more experiments directly correlates with progress

Mistakes I've made so far

Normalizing targets - a high initial target that occurs due to the initial weights can skew the normalization for the entire experiment

Doing multiple epochs over a batch

Not keeping batch size the same for experience replay & training

Not setting `next_observation = observation`

Not setting online & target network variables the same at the start of an experiment

Not gradient clipping

- clip the norm of the gradient (I've seen between 1 - 5)

Mistakes DSR students have made in RL projects

Since I started teaching in Batch 10 we have had three RL projects

Saving agent brain

- not saving the optimizer state

Using too high a learning rate

- learning rate is always important!!!

Building both an agent and environment



Hyperparameters

Policy gradients

- learning rate
- clipping of distribution parameters (stochastic PG)
- noise for exploration (deterministic PG)
- network structure

Value function methods

- learning rate
- exploration (i.e. epsilon)
- updating target network frequency
- 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





Home

Answer

Notifications

Search Quora

Reinforcement Learning

Artificial Intelligence

+1



How can I test if the training process of a reinforcement learning algorithm work correctly?

Answer

Request

Follow 7

Comment 1

Downvote



<https://www.quora.com/How-can-I-test-if-the-training-process-of-a-reinforcement-learning-algorithm-work-correctly>



Best practices

Quick experiments on small test problems

- CartPole for discrete action spaces
- Pendulum for continuous action spaces

Compare to baselines - a random agent is a good idea

Be careful not to overfit these simple problems

- use low capacity neural networks

Interpret & visualize learning process

- state visitation, value functions

Make it easier to get learning to happen (initially)

- input features, reward function design

Best practices

In reinforcement learning we often don't know the true min/max/mean/standard deviation of observations/actions/rewards/returns

Standardize data

- if observations in unknown range, estimate running average mean & stdev
- use the min & max if known

Rescale rewards - but don't shift mean

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

Batch size matters

≡

Best practices

Compute useful statistics

- explained variance (for seeing if your value functions are overfitting),
- computing KL divergence of policy before and after update (a spike in KL usually means degradation of policy)
- entropy of your policy

Visualize statistics

- running min, mean, max of episode returns
- KL of policy update
- explained variance of value function fitting
- network gradients

Gradient clipping is helpful - dropout & batchnorm not so much

≡

Amid Fish

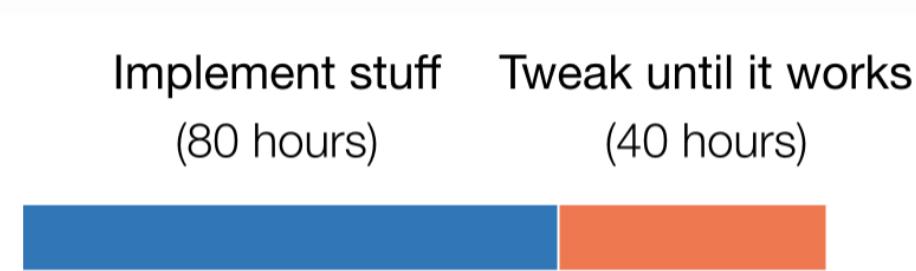
Lessons Learned Reproducing a Deep Reinforcement Learning Paper

Apr 6, 2018

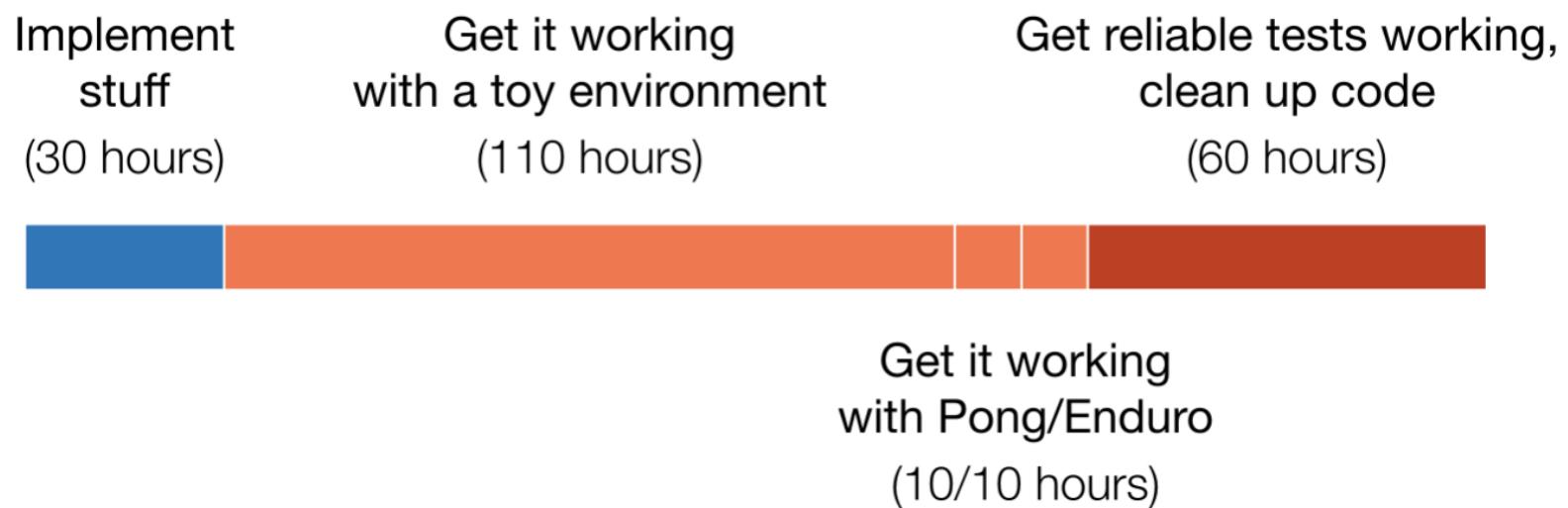
<http://amid.fish/reproducing-deep-rl>



The more interesting surprise was in how many hours each stage actually took. The main stages of my initial project plan were basically:



Here's how long each stage *actually* took.



In total, the project took:

- **150 hours of GPU time and 7,700 hours (wall time × cores) of CPU time** on Compute Engine,
- **292 hours of GPU time** on FloydHub,
- and **1,500 hours (wall time, 4 to 16 cores) of CPU time** on my university's cluster.

I was horrified to realise that in total, that added up to **about \$850** (\$200 on FloydHub, \$650 on Compute Engine) over the 8 months of the project.



Reinforcement learning can be so unstable that you need to repeat every run multiple times with different seeds to be confident.



Matthew Rahtz of Amid Fish

It's not like my experience of programming in general so far where you get stuck but there's usually a clear trail to follow and you can get unstuck within a couple of days at most.

It's more like when you're trying to solve a puzzle, there are no clear inroads into the problem, and the only way to proceed is to try things until you find the key piece of evidence or get the key spark that lets you figure it out.



Debugging

Debugging in four steps

1. evidence about what the problem might be
2. form hypothesis about what the problem might be (evidence based)
3. choose most likely hypothesis, fix
4. repeat until problem goes away

Most programming involves rapid feedback

- you can see the effects of changes very quickly
- gathering evidence can be cheaper than forming hypotheses

In RL (and supervised learning with long run times) gathering evidence is expensive

- suggests spending more time on the hypothesis stage

≡

Get more out of runs

Reccomends keeping a detailed work log

- what output am I working on now
- think out loud - what are the hypotheses, what to do next
- record of current runs with reminder about what each run is supposed to answer
- results of runs (i.e. TensorBoard)

Log all the metrics you can

- policy entropy for policy gradient methods



Matthew Rahtz of Amid Fish

RL specific

- end to end tests of training
- gym envs: -v0 environments mean 25% of the time action is ignored and previous action is repeated. Use -v4 to get rid of the randomness

General ML

- for weight sharing, be careful with both dropout and batchnorm - you need to match additional variables
- spikes in memory usages suggest validation batch size is too big
- if you are struggling with the Adam optimizer, try an optimizer without momentum (i.e. RMSprop)

TensorFlow

≡

- `sess.run()` can have a large overhead. Try to group session calls

Cool open source RL projects

[gym](#) - Open AI

[baselines](#) - Open AI

[rllab](#) - Berkley

[Tensorforce](#) - reinforce.io



Recap

Quick experiments on small test problems

- make learning easy
- automate experiments

Visualize the learning process

Multiple random seeds

Preprocess/scale observations/targets etc

Deep RL is hard and sample inefficient

≡

eight

Deep RL doesn't work yet

auxillary loss functions

inverse reinforcement learning

world models

≡

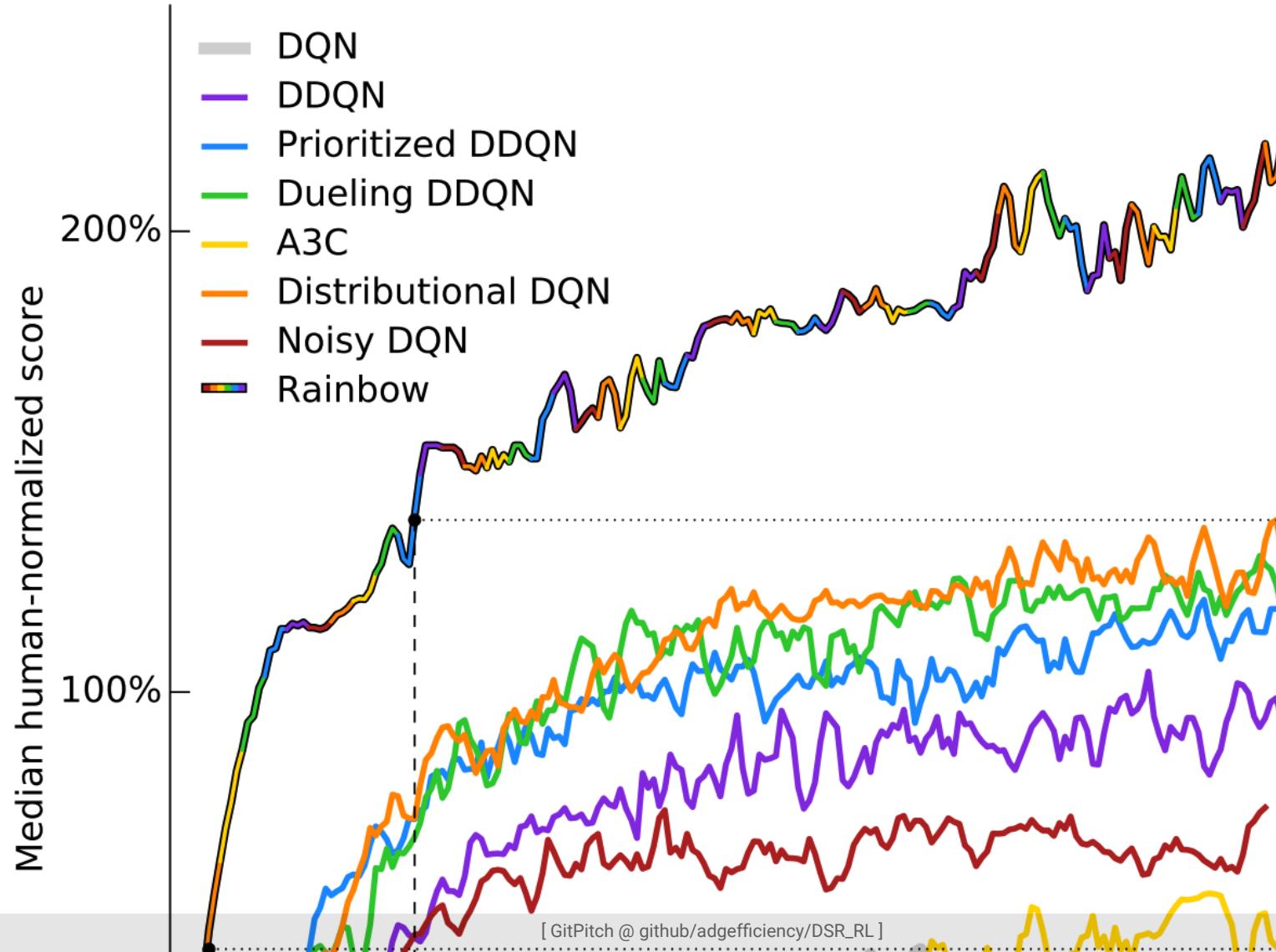
In a world where everyone has opinions, one man...also has opinions

Deep Reinforcement Learning Doesn't Work Yet

Feb 14, 2018



Modern RL is sample inefficient



Other methods often work better

Many problems are better solved by other methods

- allowing the agent access to a ground truth model (i.e. simulator)
- model based RL with a perfect model

In a similar vein, you can easily outperform DQN in Atari with off-the-shelf Monte Carlo Tree Search. Here are baseline numbers from [Guo et al, NIPS 2014](#). They compare the scores of a trained DQN to the scores of a UCT agent (where UCT is the standard version of MCTS used today.)

| Agent | <i>B.Rider</i> | <i>Breakout</i> | <i>Enduro</i> | <i>Pong</i> | <i>Q*bert</i> | <i>Seaquest</i> | <i>S.Invaders</i> |
|--------------|----------------|-----------------|---------------|-------------|---------------|-----------------|-------------------|
| DQN | 4092 | 168 | 470 | 20 | 1952 | 1705 | 581 |
| <i>-best</i> | 5184 | 225 | 661 | 21 | 4500 | 1740 | 1075 |

| Agent | <i>B.Rider</i> | <i>Breakout</i> | <i>Enduro</i> | <i>Pong</i> | <i>Q*bert</i> | <i>Seaquest</i> | <i>S.Invaders</i> |
|------------|----------------|-----------------|---------------|-------------|---------------|-----------------|-------------------|
| UCT | 7233 | 406 | 788 | 21 | 18850 | 3257 | 2354 |

Requirement of a reward function

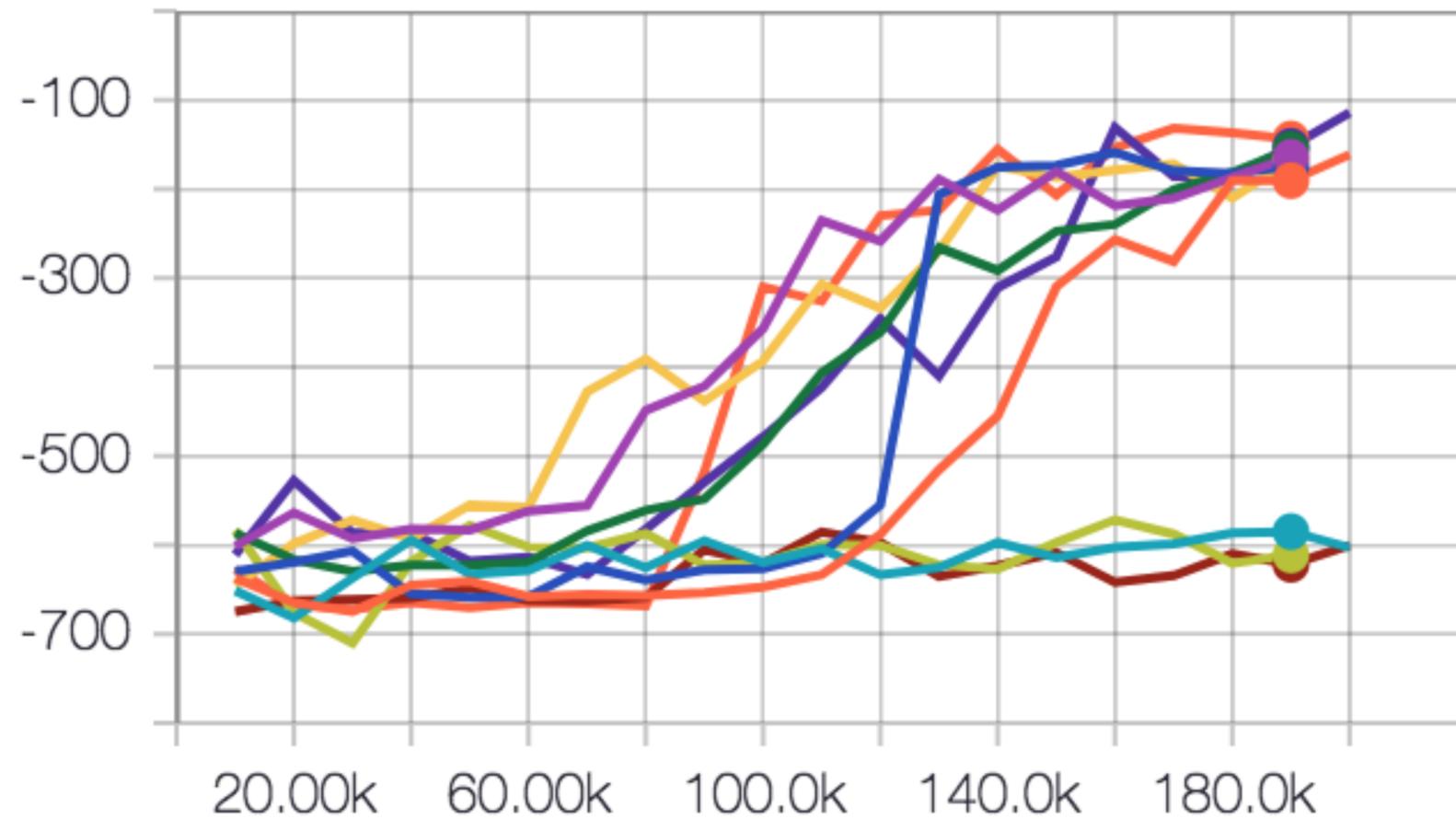
Reward function design is difficult

- need to encourage behaviour
- need to be learnable

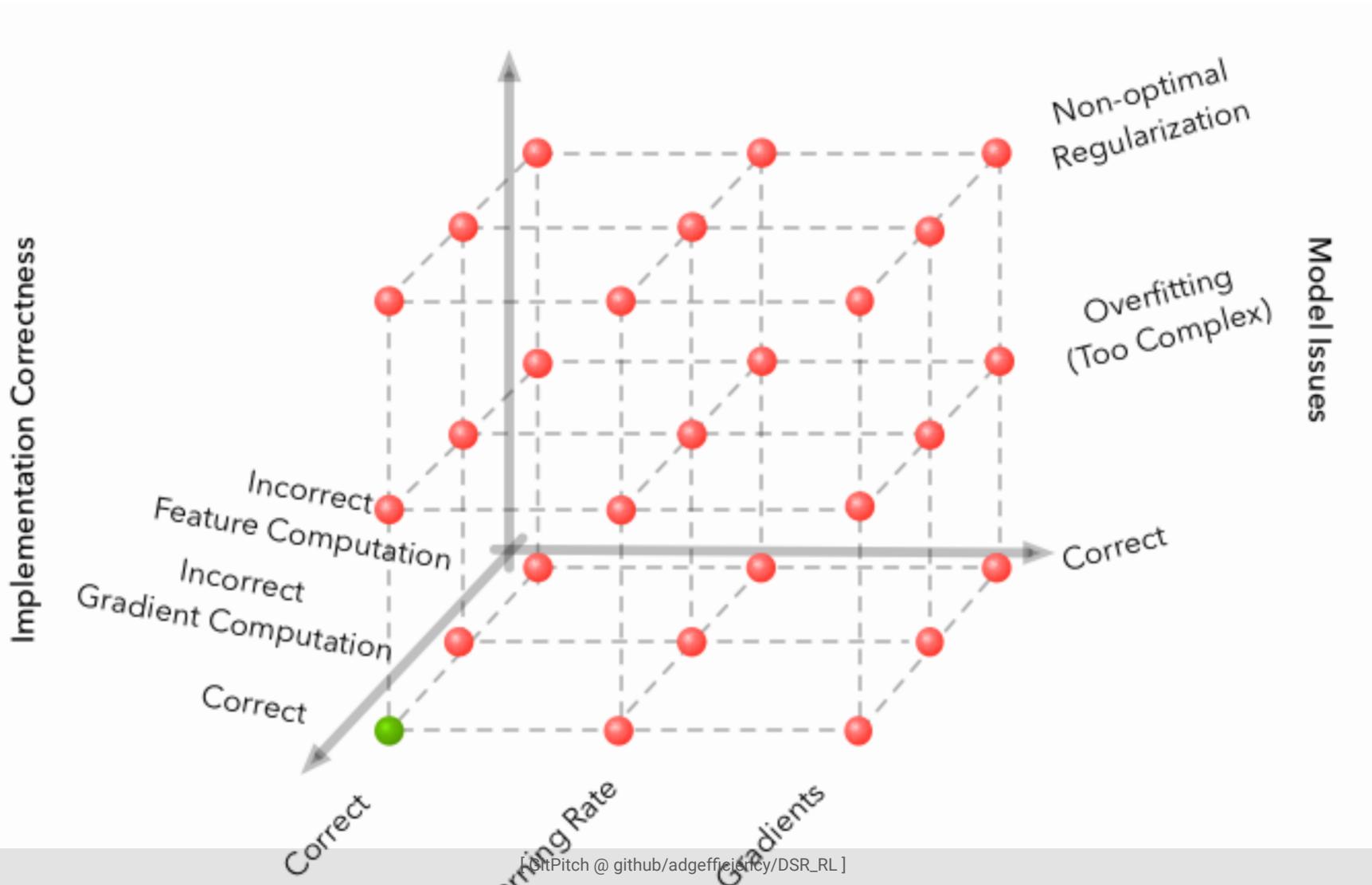
Shaping rewards to help learning can change behaviour

Unstable and hard to reproduce results

episode_reward/test



Machine learning adds more dimensions to your space of failure cases



Going forward & the future

The way I see it, either deep RL is still a research topic that isn't robust enough for widespread use, or it's usable and the people who've gotten it to work aren't publicizing it. I think the former is more likely.

Make learning easier

- ability to generate near unbounded amounts of experience
- problem is simplified into an easier form
- you can introduce self-play into learning
- learnable reward signal
- any reward shaping should be rich

The future

- local optima are good enough (is any human behaviour globally optimal)
- improvements in hardware help with sample inefficiency

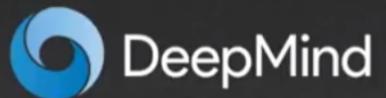
≡



Deep Reinforcement Learning and Real World Challenges

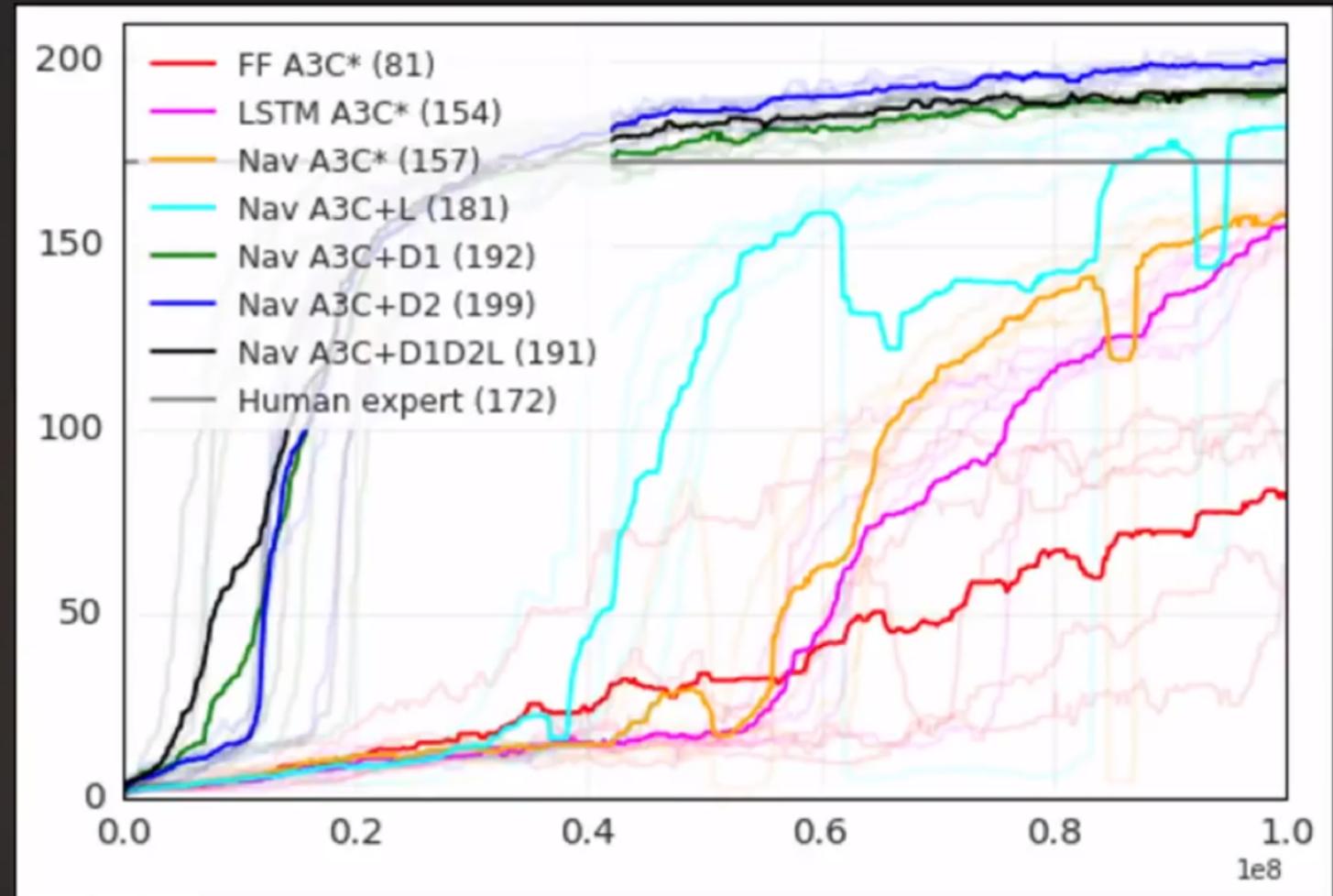
Raia Hadsell
Research Scientist, DeepMind

www.raiahadsell.com



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

Results: Auxiliary tasks speed up RL ten-fold!



DeepMind

Inverse Reinforcement Learning

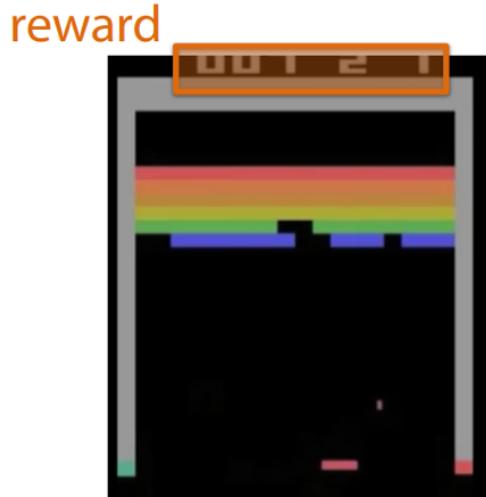
Chelsea Finn

Deep RL Bootcamp



Where does the reward come from?

Computer Games



Mnih et al.'15

Real World Scenarios

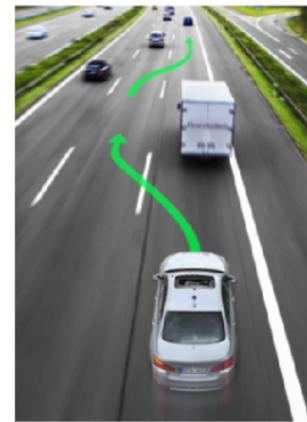
robotics



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

Closing thoughts

Exploration versus exploitation

Test your models on simple problems

Reinforcement learning is sample inefficient

Deep RL is hard

Reward engineering is key

Thank you

Adam Green

≡

adgefficiency.com