Course outline

Introduction

Introduction to reinforcement learning

Statistics and machine learning background

Markov Decision Processes

Value functions

Policy optimization

Model-based reinforcement learning

AlphaGo to AlphaZero

Practical concerns

Introduction

Experience

s	state
s'	next state
a	action
r	reward

Measurements of total reward

$\mathbf{E}[f(x)]$	expectation of $f(x)$
γ	discount factor [0, 1)
G_t	discounted return after time t
$V_\pi(s)$	value function
$Q_\pi(s, a)$	action-value function

Taking actions

$a \sim \pi(s)$	sampling action from a stochastic policy
$a = \pi(s)$	deterministic policy
$\pi^\star$	optimal policy
θ, ω	function parameters

This course

Developed over two years

- 10 courses, ~90 students

Two day introduction to reinforcement learning at Data Science Retreat in Berlin

- no familiarity with reinforcement learning
- familiarity with supervised learning

Goals for today

- introduction to supervised learning
- landscape of modern reinforcement learning - terminology, key algorithms
- challenges in modern reinforcement learning
- where to go next

Where to go next

My personal collection of reinforcement learning resources

ADGEfficiency/rl-resources

Open AI's Spinning Up in Deep RL

lecture - notes

Sutton & Barto - An Introduction to Reinforcement Learning (2nd Edition)

textbook pdf

Where are we today?

AlphaGo was a milestone achievement for reinforcement learning

10 years ahead of schedule

Fleeting glimpses of application in industry

Google data centres

Where are we today

Modern trends

deep neural networks as function approximators

improvements on old algorithms (Q-Learning -> DQN)

new algorithms (MCTS)

access to GPU compute

Challenges

hidden information

instability across random seeds

sample efficiency

access to simulators

State of the art

Atari

Performance - Hessel et. al (2017) Rainbow - paper

AlphaZero

DeepMind - built on top of MCTS

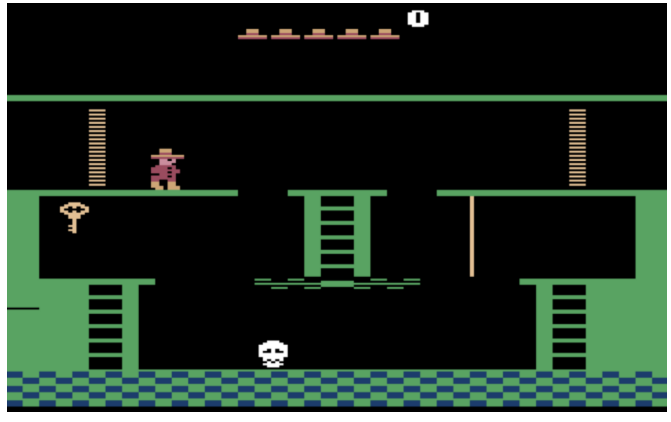
Open AI Five

built on top of PPO

World Models

agent's learning inside dream environments

Atari



Montezuma's Revenge - a difficult exploration problem

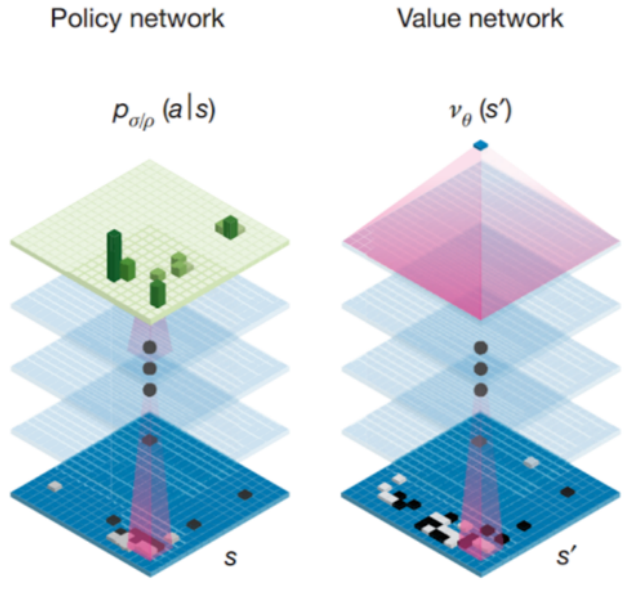
The Atari Learning Environment (ALE) is a key reinforcement learning benchmark

learning from pixels

discrete space

stack four frames to make Markovian

AlphaGo to AlphaZero



AlphaGo to AlphaZero

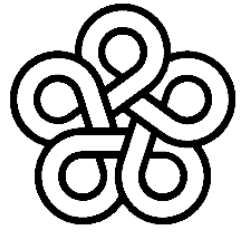
Generalization of superhuman performance across Go, Chess and Shogi

- self play
- planning using Monte Carlo Tree Search
- deep convolutional residual networks

AlphaGo -> AlphaGoZero -> AlphaZero

- removed dependence on human expert data

Open AI Five



Open AI Five

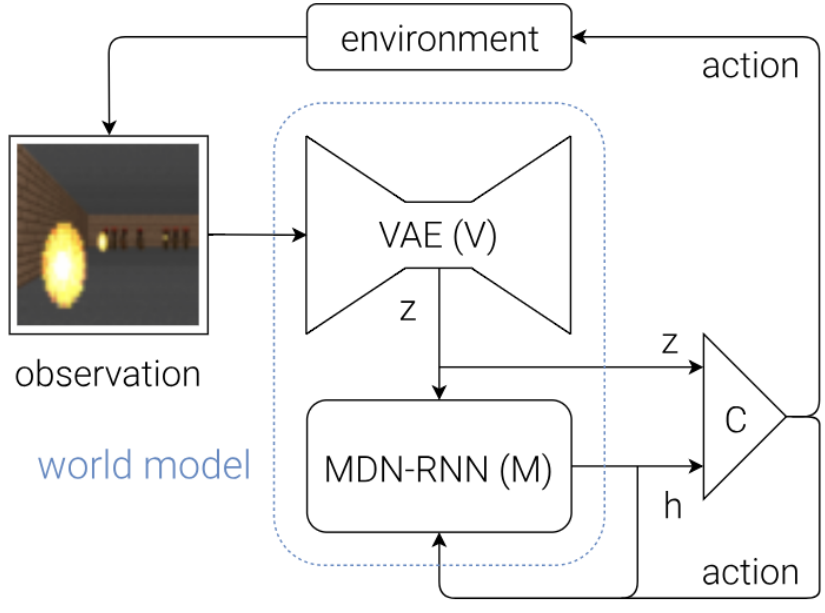
Real time strategy game

- hidden information team game
- large & continuous action space

Recently beat the world champions OG 2-0

- learning from n-d arrays
- simplified version of the full game

World Models



World Models

High quality interactive blog post

Model based reinforcement learning

- autoencoder to compress image
- LSTM memory to compress time
- CMA-ES for control

Notable organizations

DeepMind

Open AI

Berkley

Notable people*

Rich Sutton - University of Alberta / DeepMind

co-author of the Bible of reinforcement learning

David Silver - DeepMind

lead author on the AlphaGo papers

John Schulman - Berkeley / Open AI

lead author TRPO, PPO

Sergey Levine - Berkeley / Open AI / Google

robotics, teaches CS294

**notable due to research and visibility through teaching*

About me

Adam Green - adam.green@adgefficiency.com - adgefficiency.com

chemical engineer by training

four years as an energy engineer

two years as an energy data scientist

Enjoy building computational models

Introduction to reinforcement learning

what else could/should I try before reinforcement learning?

is my problem a reinforcement learning problem?

overview of the reinforcement learning landscape

Learning through action

Sequential decision making

- seeing cause and effect
- consequences of actions
- what to do to achieve goals

Substrate independent

- occurs in silicon and biological brains

Key features

- trial & error search
- delayed reward

What is reinforcement learning?

Computational approach to learning through action

- end to end problem
- goal directed agent
- interacting with an environment

Learning to maximize a reward signal

- map environment observation to action
- actions effect the future

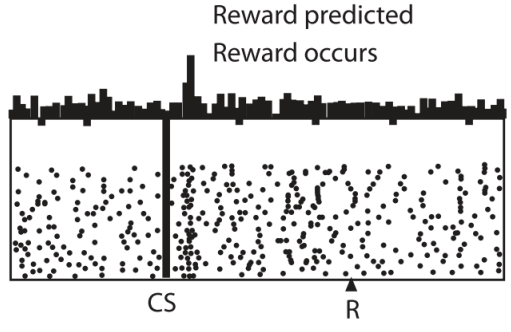
Biological inspiration

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

Sutton & Barto

Dopamine as a value function

Concentration of dopamine associated with the start of the stimulus



Expected reward not experienced -> no dopamine

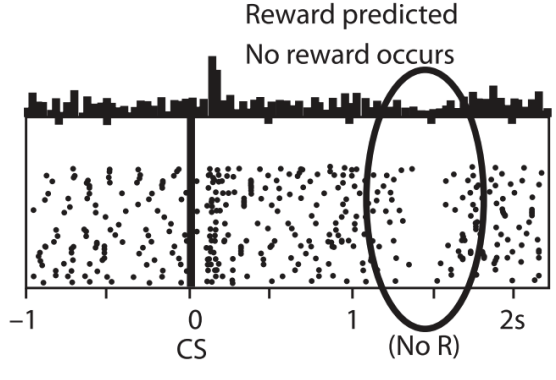


Fig. 3. from [Glimcher (2011) Understanding dopamine and reinforcement learning](https://www.pnas.org/content/108/Supplement_3/15647)

Applications of reinforcement learning

Sequential decision making problems

control physical systems

interact with users

solve logistical problems

play games

learn sequential algorithms

chess

optimization of refinery

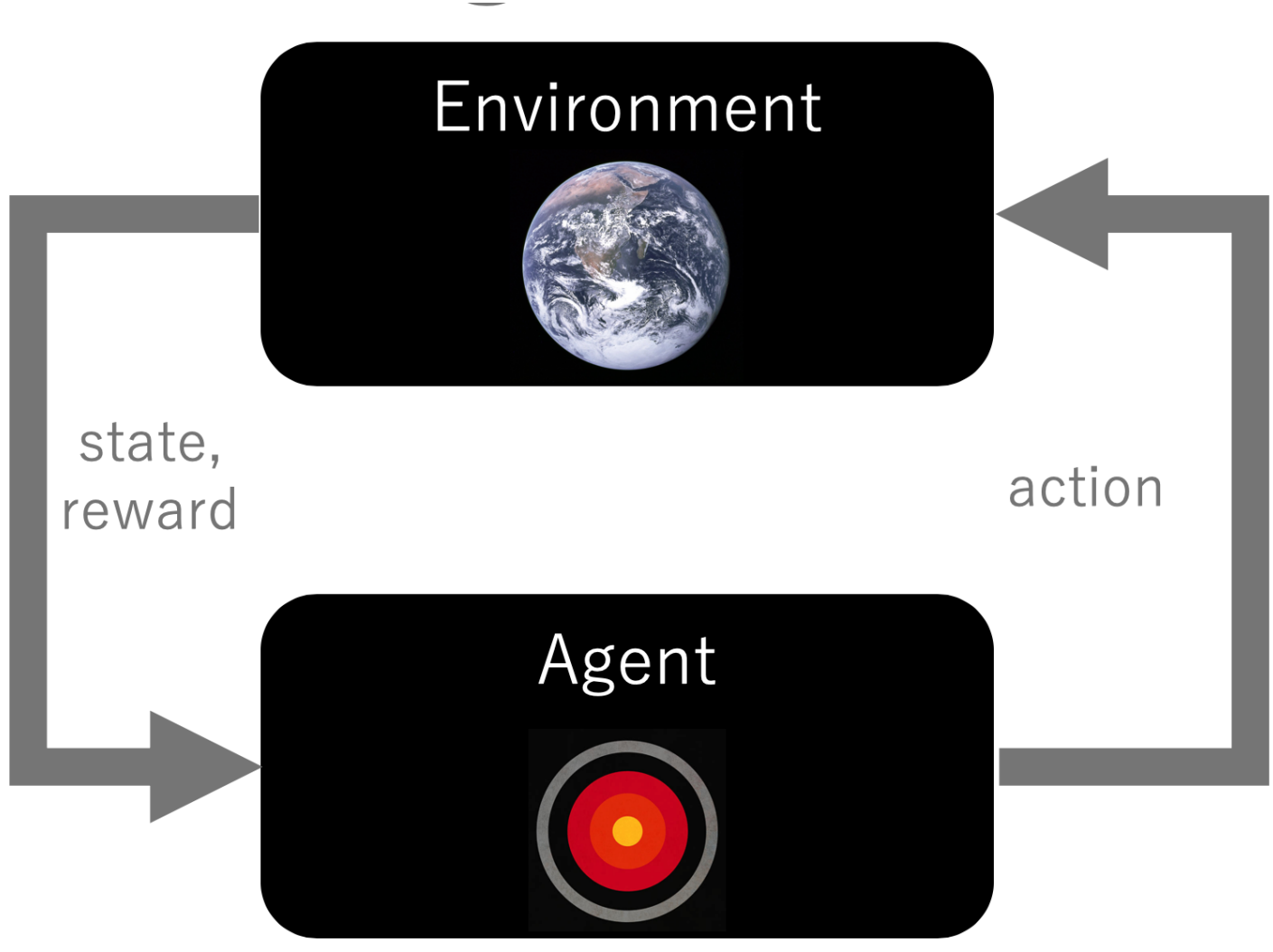
calf learning to walk

robot battery charging

making breakfast

David Silver - http://www0.cs.ucl.ac.uk/staff/d.silver/web/Resources_files/deep_rl.pdf

Sutton & Barto


Environment
state,
reward
action
Agent

Is my problem a reinforcement learning problem?

What is the action space

- what can the agent choose to do
- does the action change future states

What is the reward function

- does it incentive behaviour

It is a complex problem

- linear programming or cross entropy may offer a simpler solution

Can I sample efficiently / cheaply

- do you have a simulator

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!

Instability across random seeds

- results will succeed and fail over different random seeds (same hyperparameters!)
- this means you need to do a lot more experiments

Machine learning is an empirical science

- the ability to do more experiments directly correlates with progress
- this most true in reinforcement learning

The control problem

Trying to take good actions

- generate and test
- trial and error learning
- evolution

Prediction and control

- supervised and reinforcement
- being able to predict allows us to take good actions

Control methods

Without gradients

- guess and check
- cross entropy method
- evolutionary methods
- constrained optimization

Gradient based

- optimal control - model based
- model based reinforcement learning
- model free reinforcement learning

Constrained optimization

Guaranteed convergence to global optimum for convex system of equations

mixed integer linear programs can be used to model many business problems

```
minimize cost function

subject to equality ( == ) constraints
subject to inequality ( >= or <= ) constraints
```

Evolutionary methods

Biologically inspired - selection, reproduction and mutation

- easily parallelizable
- good in partially observed environments

CMA-ES

- good algorithm up to a few thousand parameters
- used in World Models to learn parameters of a linear controller

Optimal control

ICML 2018: Tutorial Session: Optimization Perspectives on Learning to Control - lecture - slides

Exists both in parallel and overlapping with reinforcement learning

focuses on continuous spaces and environment models

includes PID control (95% of all industrial control is PI), Model Predictive Control (MPC)

Linear Quadratic Regulator

linear transition dynamics, quadratic cost

The four grades of competence

Each is a successive application of generate and test with improved competence

competence = ability to act well

comprehension = understanding

Dennet - From Bach to Bacteria and Back

The four grades of competence

1 - Darwinian

pre-designed and fixed competence

no learning within lifetime

global improvement via local selection

2 - Skinnerian

the ability to adjust behaviour through reinforcement

learning within lifetime

hardwired to seek reinforcement

The four grades of competence

3 - Popperian

learns models of the environment

local improvement via testing behaviours offline

crows, cats, dogs, dolphins & primates

4 - Gregorian

builds thinking tools

arithmetic, democracy & computers

systematic exploration of solutions

local improvement via higher order control of mental searches

only humans

The four grades of competence

Darwinian

fixed competences, tested by selection

Skinnerian

adjusts behaviour through reinforcement

Popperian

learns models of the environment, tests using environment models

Gregorian

builds thinking tools

Reward hypothesis

Maximising expected return (which is what agent's do) is making an assumption about the nature of our goals

Goals can be described by the maximization of expected cumulative reward

do humans optimize for expected value?

what about multi-modal distributions?

The Reward Engineering Principle

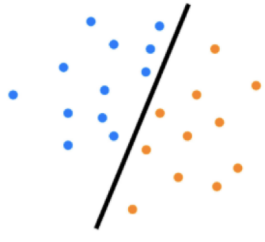
As reinforcement-learning based AI systems become more general and autonomous, the design of reward mechanisms that elicit desired behaviours becomes both more important and more difficult.

The Reward Engineering Principle - [Dewey (2014) Reinforcement Learning and the Reward Engineering Principle] (http://www.danieldewey.net/reward-engineering-principle.pdf)

Context within machine learning

Supervised

learning
known
patterns



Random forests
Support vector machines

Feedforward neural net
Convolutional neural net
Recurrent neural net

Unsupervised

learning
unknown
patterns

Clustering algorithms

Generative adversarial nets
(GANS)

Reinforcement

taking actions
generating data
learning patterns

Value function methods

Policy gradients

Planning (i.e. model based)

Contrast with supervised learning

Supervised learning

given a dataset with labels

test on unseen data

Reinforcement learning

need to generate data by taking actions

need to label data

Contrast with supervised learning

Freedom from the constraint of a dataset

hard to access high quality datasets

easy to access high quality simulators

Democratization

Requirement of a dataset is replaced with requirement of a simulator

sample inefficiency limits learning from small number of samples

Reinforcement learning data

Reinforcement learning involves

generating data by taking actions

creating targets for data

One sample of data = experience/transition tuple (s, a, r, s')

no implicit target

The dataset we generate is the agent's memory

list of experienced transitions

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

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

$$\ldots$$

Statistics and machine learning background

what are the advantages and disadvantages of lookup tables?

which of bias/variance is over/under fitting?

how is reinforcement learning different from supervised learning?

Expectations

The mean - weighted average of all possible values

$$expectation = probability * magnitude$$

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

Expectations allow us to approximate by sampling

to approximate the average time it takes us to get to work

measure how long it takes us for a week and average each day

$$expectation = average(samples)$$

Expectations

Expectations are convenient in reinforcement learning

The expectation gives us something to aim at

reinforcement learning optimizes for total expected reward

The expectation allows us to approximate using samples

often all we can do is sample

working with expectations allows us to use samples as approximations for the true expectation

IID

Fundamental assumption in statistical learning

independent and identically distributed

assuming that the training set is independently drawn from a fixed distribution

Independent

our samples are not correlated/related to each other

Identically distributed

the distribution across our data set is the same as the 'true' distribution

Conditionals

Probability of one thing given another

probability of next state s' given state s & action a

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

reward received conditioned on taking action a in state s, then transitioning to state s'

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

sampling an action from the policy - action a conditioned on state s

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

Lookup tables

State with two dimensions

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

One row per element of the discretized state space

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
- discretization

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

Neural networks

Neural networks

Used to approximate functions

- forward pass maps from input to output
- forward pass maps from features to target

Learn these functions by creating targets

$$loss = approximation - target$$

- this is supervised learning

Use the loss to find gradients

- loss is a measure of error
- use gradients to change weights in direction that reduces error

Neural networks

Feedforward / fully connected

general purpose function approximator

Convolution

learning from pixels

Recurrent

learning from sequences

All use features to predict a target

Feedfoward

Bootstrapping

Create targets using bootstrapping

function creates targets for itself

The Bellman Equation is bootstrapped equation

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

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

Bootstrapping causes

bias - the agent has a chance to fool itself

instability - weight updates depend on previous weights

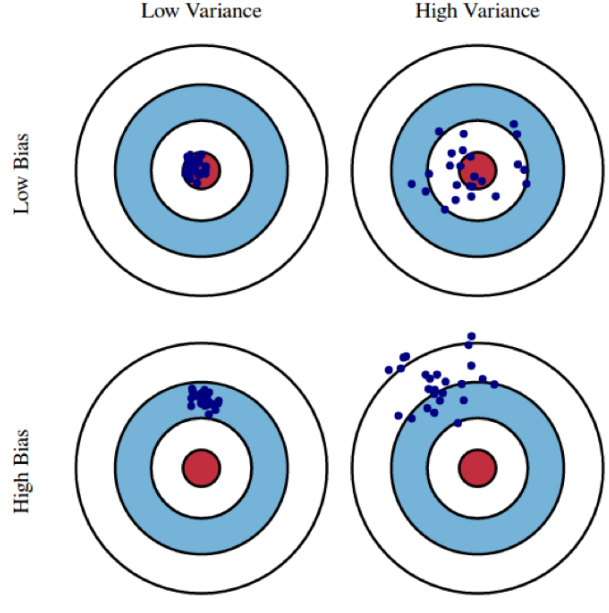
Variance & bias

Variance

if I did this again, would I get the same result

Bias

am I getting the correct result



Variance & bias in supervised learning

Model generalization error = bias + variance + noise

Variance = deviation from expected value = overfitting

error from sensitivity to noise in data set

seeing patterns that aren't there

Bias = deviation from true value = underfitting

error from assumptions in the learning algorithm

missing relevant patterns

Variance & bias in reinforcement learning

Variance = deviation from expected value

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

Bias = deviation from true value

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

Random seeds

Used to control randomness in computers

Using the same random seed should result in the same random numbers

Instability of reinforcement learning across different random seeds is a key message of this course

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

A rough map of the reinforcement learning landscape

Model free

Value functions

Learn value functions
Generate policy from a value function

Discrete action spaces only

Q-Learning
DQN

SOTA* = Rainbow

Actor-Critic

Learn value functions
Learn a policy

Continuous and discrete action spaces

REINFORCE
A2C & A3C
TRPO

SOTA = Proximal Policy Optimization

Model based

Dynamic programming

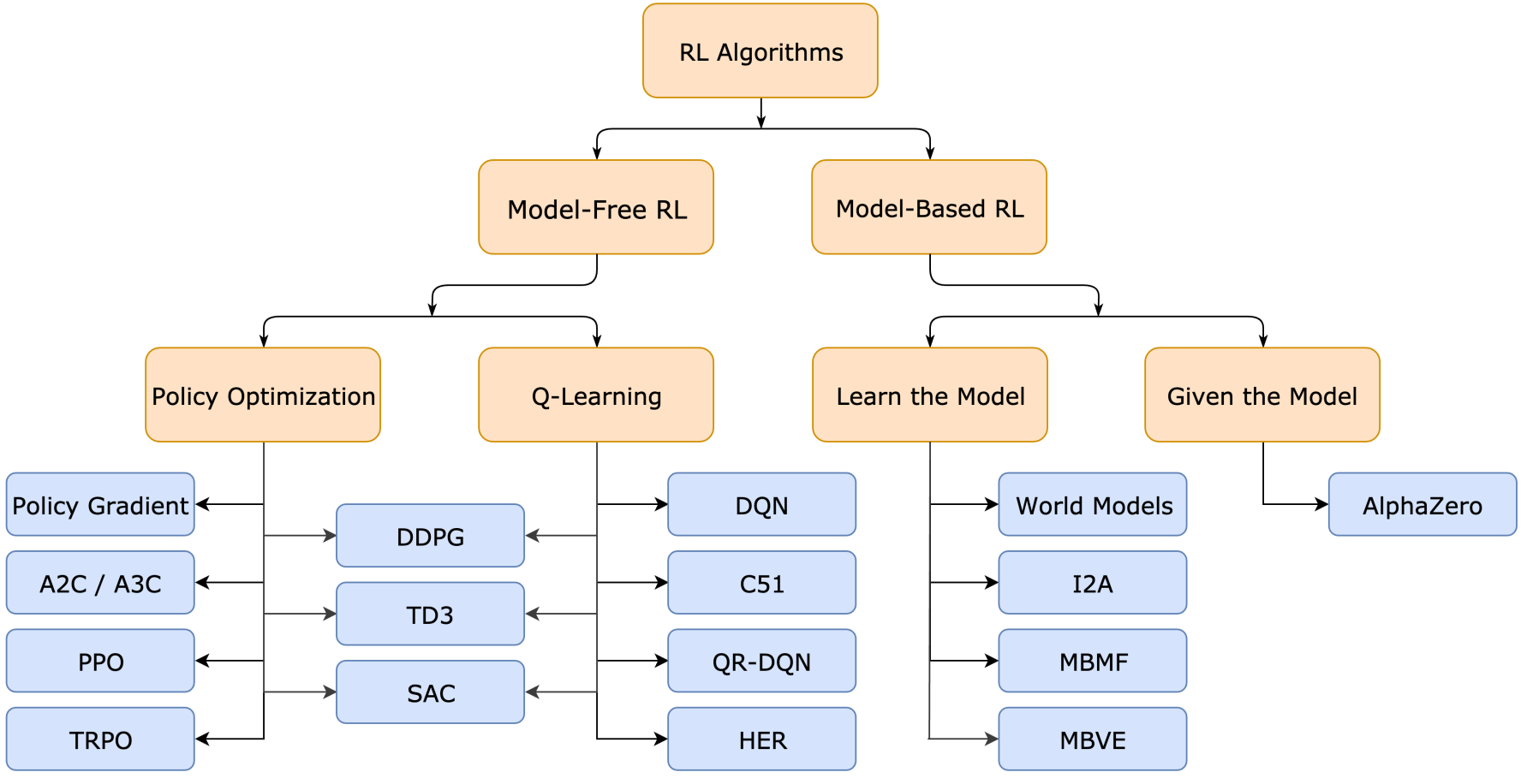
Monte Carlo Tree Search

Learn environment model
Plan using environment model

AlphaGo
World Models

SOTA = Alpha Zero

* SOTA = state of the art



Open AI Spinning Up

Four central challenges

Exploration versus 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 versus exploitation

How stationary are the environment state transition and reward functions?

How stochastic is my policy?

Design of reward signal versus exploration required

Time step matters

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

Data quality

Reinforcement learning breaks both assumptions in IID

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

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

policy is biased in sampling experience

distribution of experience changes as we learn

Credit assignment

which action gave me this reward

Sample efficiency

learning quickly, squeezing information from data

Markov Decision Processes

what's wrong with discretizing an action space?

what is the credit assignment problem?

what are the two main tasks of an agent?

why is on-policy learning less sample efficient?

Markov Decision Processes (MDPs)

Framework from dynamical systems theory

optimal control of partially observed MDP's

Mathematical framework for sequential decision making

framework within which agent's live

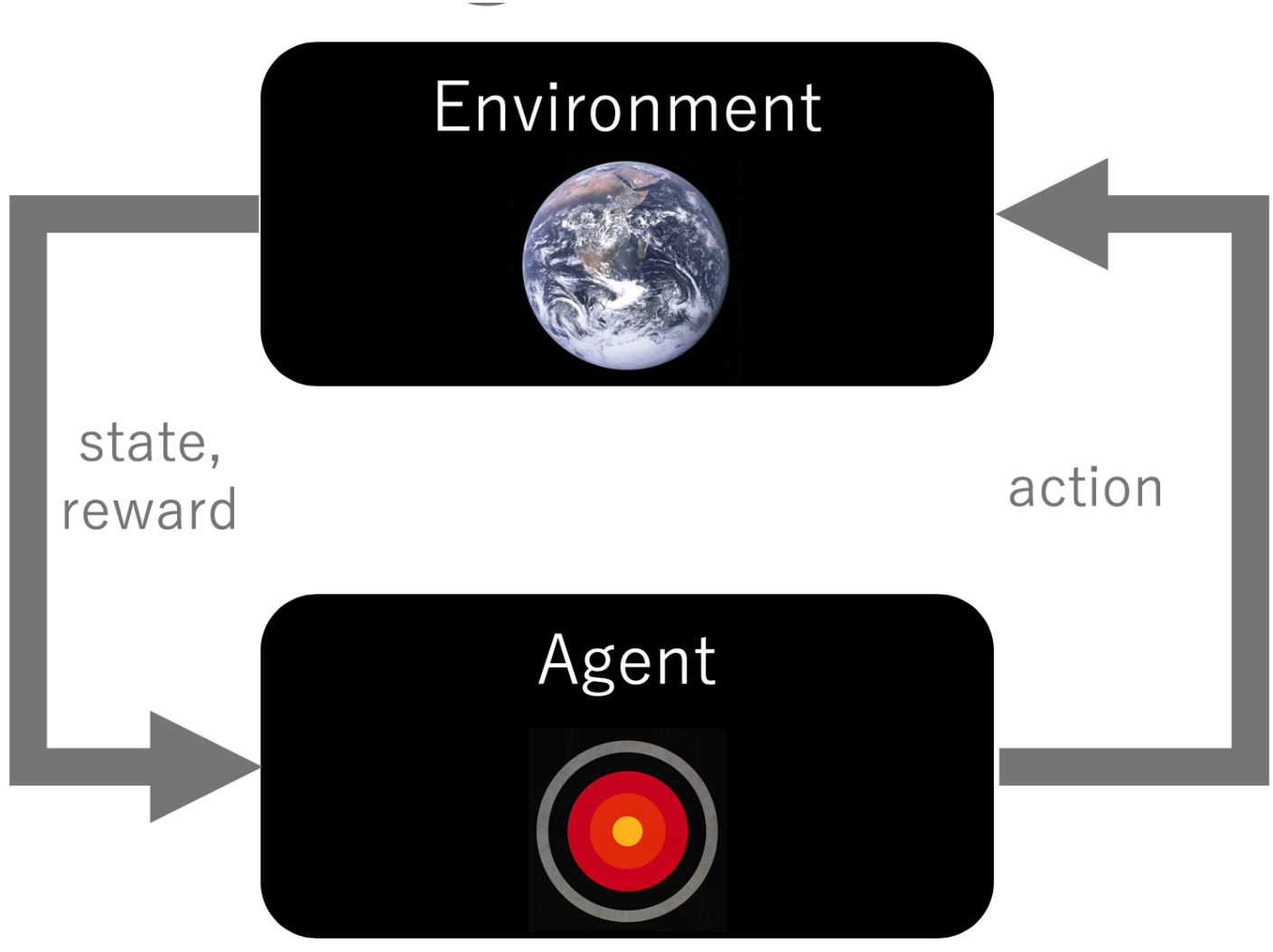
Markov property

additional information will not improve our decision

can make decisions using only the current state

don't need the history of the process

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


Environment
state,
reward
action
Agent

Formal definition of a MDP

An MDP is defined as 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

```
done = False
while not done:

    #  select an action based on the observation
    action = agent.act(observation)

    #  take a step through the environment
    next_observation, reward, done, info = env.step(action)

    #  store the experienced transition
    agent.remember(observation, action, reward, next_observation, done)

    #  improve the policy
    agent.learn()
```

Environment

Real or virtual

virtual environments allow cheap sampling of experience

Each environment has a state space and an action space

discrete or continuous

Environment

Episodic

finite horizon, with a variable or fixed length

Non-episodic

infinite horizon

Both exist in the same mathematical framework

discounting to keep returns finite

a post episode 'absorbing state' turns episodic into infinite horizon

Discretiziation

Requires some prior knowledge

Lose the shape of the space

Too coarse

non-smooth control output

Too fine

curse of dimensionality

computational expense

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

State and observation

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

ground truth of the system

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

information the agent sees about the system

used to choose actions and to learn

Observation can be made more Markov by

concatenating state trajectories together

using function approximation with a memory element (LSTMs)

Reward and return

Reward (r) is a scalar

delayed

sparse

Return (G_t) is the total discounted future reward

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

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

geometric series

Agent

Goal to maximize total reward

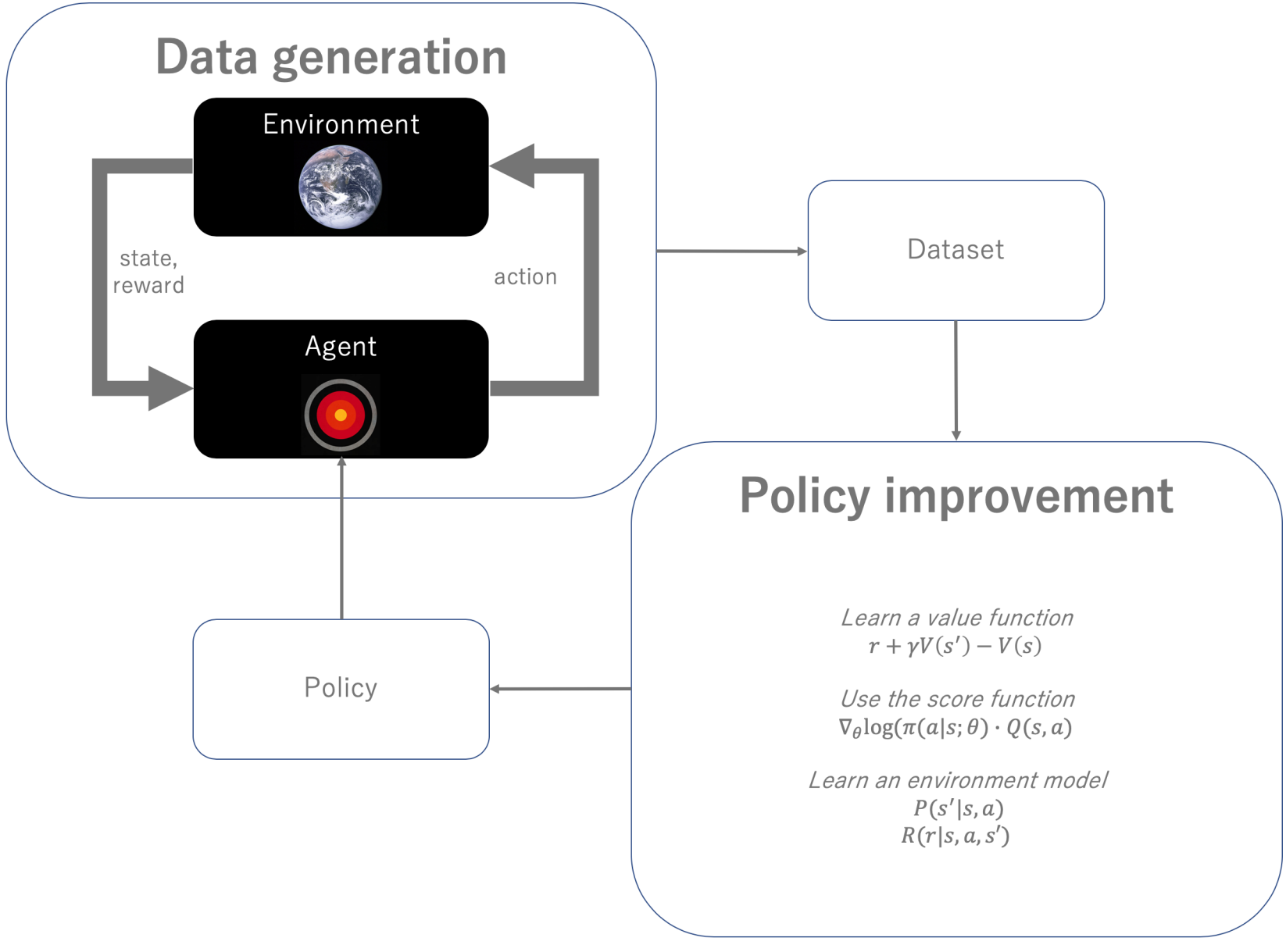
- often optimize for the expectation of future discounted reward

Our agent has a lot to do!

- decide what actions to take
- learn how to take better actions

Reinforcement learning is the interaction of these two processes

- prediction versus control
- general policy iteration
- data generated by actions used in learning


Data generation
Environment
state,
reward
action
Agent
Dataset
Policy improvement
Learn a value function
$r + \gamma V(s') - V(s)$
Use the score function
$\nabla_\theta \log(\pi(a|s;\theta) \cdot Q(s,a)$
Learn an environment model
$P(s'|s,a)$
$R(r|s,a,s')$
Policy

Policy

$$\pi(s) = \pi(s, a; \theta) = \pi_\theta(s|a)$$

Rules to select actions

Any rule is valid

act randomly

always pick a specific action

We want the optimal policy - $\pi_*(s)$

the policy that maximizes future reward

Policy

```
def random_policy():
    return env.action_space.sample()
```

Parametrized directly

policy gradient methods

Generated from a value function

value function methods

Deterministic or stochastic

often explore using a stochastic policy

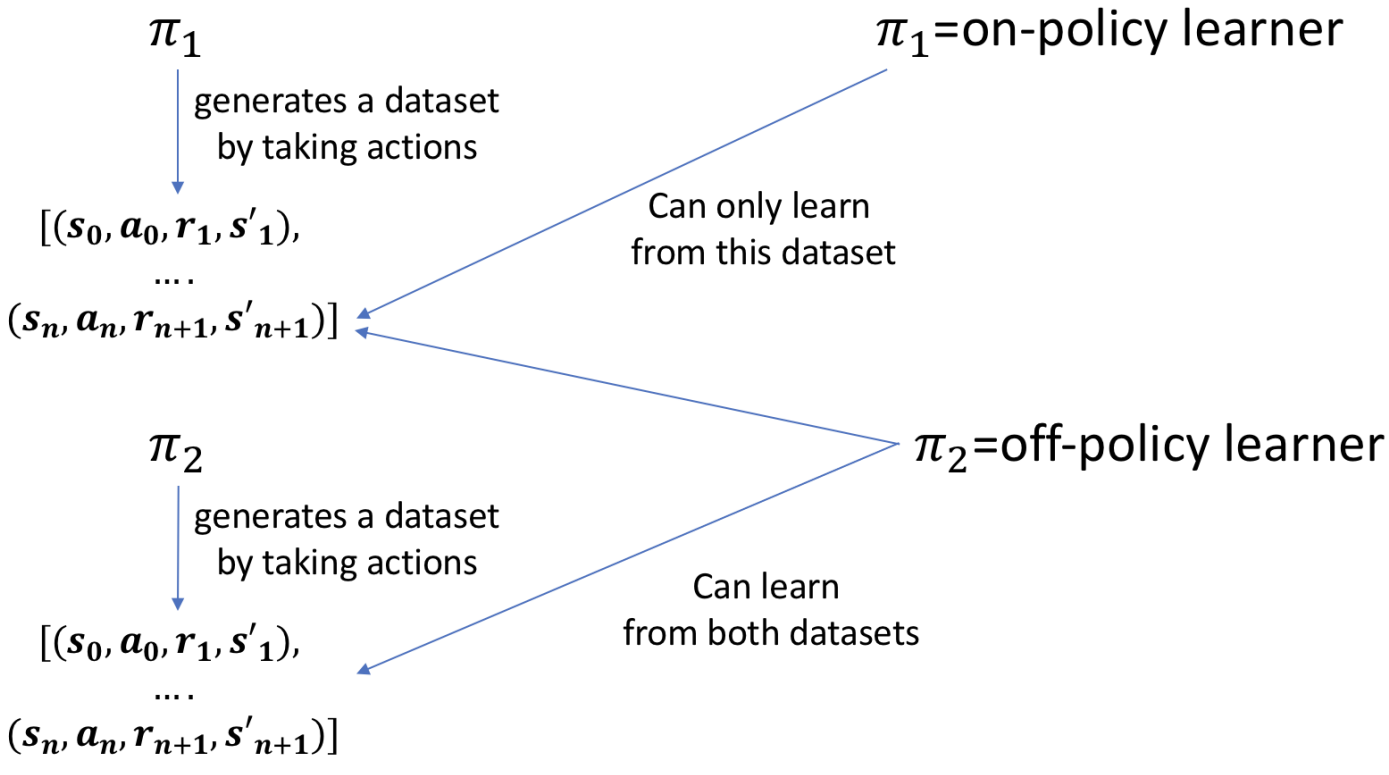
On versus off policy learning

On policy

- learn about the policy we are using to make decisions
- limited to data generated using our policy

Off policy

- evaluate or improve one policy while using another to make decisions
- use data generated by other policies
- less stable


π_1
generates a dataset
by taking actions
$[(s_0, a_0, r_1, s'_1),$
$\ldots.$
$(s_n, a_n, r_{n+1}, s'_{n+1})]$
π_1=on-policy learner
Can only learn
from this dataset
π_2
generates a dataset
by taking actions
$[(s_0, a_0, r_1, s'_1),$
$\ldots.$
$(s_n, a_n, r_{n+1}, s'_{n+1})]$
π_2=off-policy learner
Can learn
from both datasets

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

off policy learning increases the diversity of the datasets we can learn from

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

Value functions

what do value functions predict?

how do we use the Bellman equation to act?

how do we use the Bellman equation to learn?

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

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...

[On the naming of dynamic programming](ttp://arcanesentiment.blogspot.com.au/2010/04/why-dynamic-programming.html)

Value function

$V_\pi(s)$

how good is this state

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

expected return when in state s, following policy π

Action-value function

$Q_\pi(s, a)$

how good is this action

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

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

Value functions are oracles

Prediction of the future

predict return

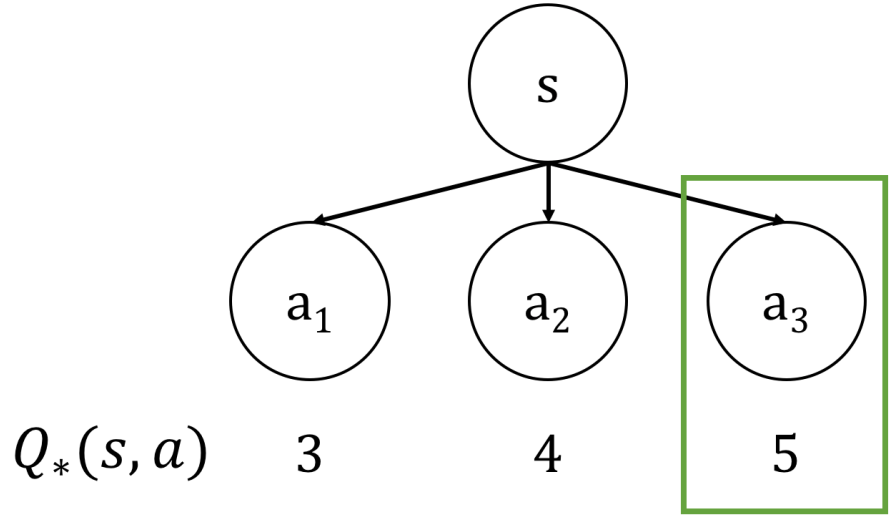
Always conditioned on a policy

return depends on future actions

We don't know this function

agent must learn it

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




```
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
```

Prediction versus control

Prediction / approximation

predicting return for given policy

Control

the optimal policy

the policy that maximizes expected future discounted reward

Prediction helps us to do control

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

Tables and linear functions are appropriate with some functions

- depends on agent and environment

Modern reinforcement learning is based on using neural networks

- convolution for learning from pixels
- feedforward for learning from n-d arrays

Approximation methods

Let's look at three different methods for approximation

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

We are creating targets to learn from

we are labelling our data

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?

Bellman Equation

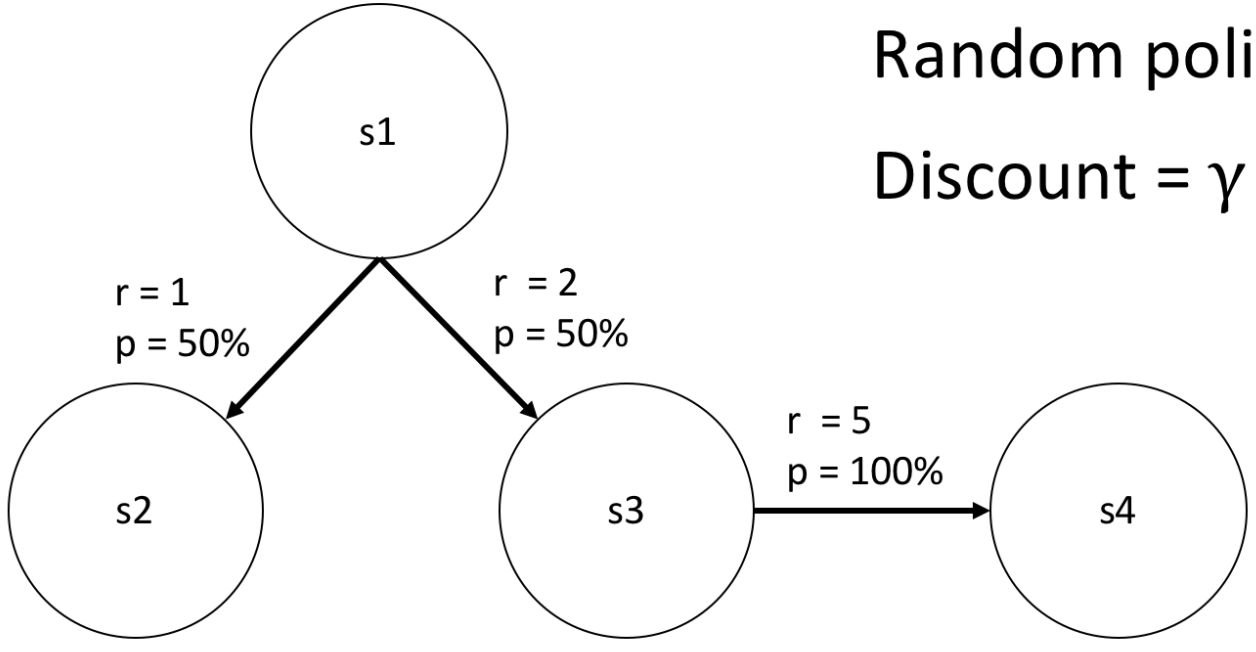
Bellman's contribution is remembered by the Bellman Equation

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

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

value of being in state S = reward r + discounted value of next state s'

bootstrapped



Random policy

Discount = $\gamma = 0.9$

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

probabilities here depend both on the environment and the policy

Dynamic programming backup

The return for all terminal states is zero

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

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

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

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

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

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
- choosing which states to update can help (asynchronous dynamic programming)

Policy iteration

Iterate between

improving value function accuracy

improving the policy

Value iteration

Do it all in a single step

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

Dynamic programming summary

Requires a perfect environment model

access to the full distribution over next states and rewards

Full backups

update based on the probability distribution over all possible next states

expectation based backups

Bootstrapped

Bellman Equation to update our value function

Monte Carlo

Monte Carlo methods

finding the expected value of a function of a random variable

No environment model

learn from sampled transitions

No bootstrapping

we take the average of the true discounted return

Episodic only

need to measure the experienced discounted return

Monte Carlo

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?

Monte Carlo

High variance

we need to sample enough episodes for our averages to converge

can be a lot for stochastic or path dependent environments

Low bias

we are using actual experience

no chance for a bootstrapped function to mislead us

Lookup table Monte Carlo

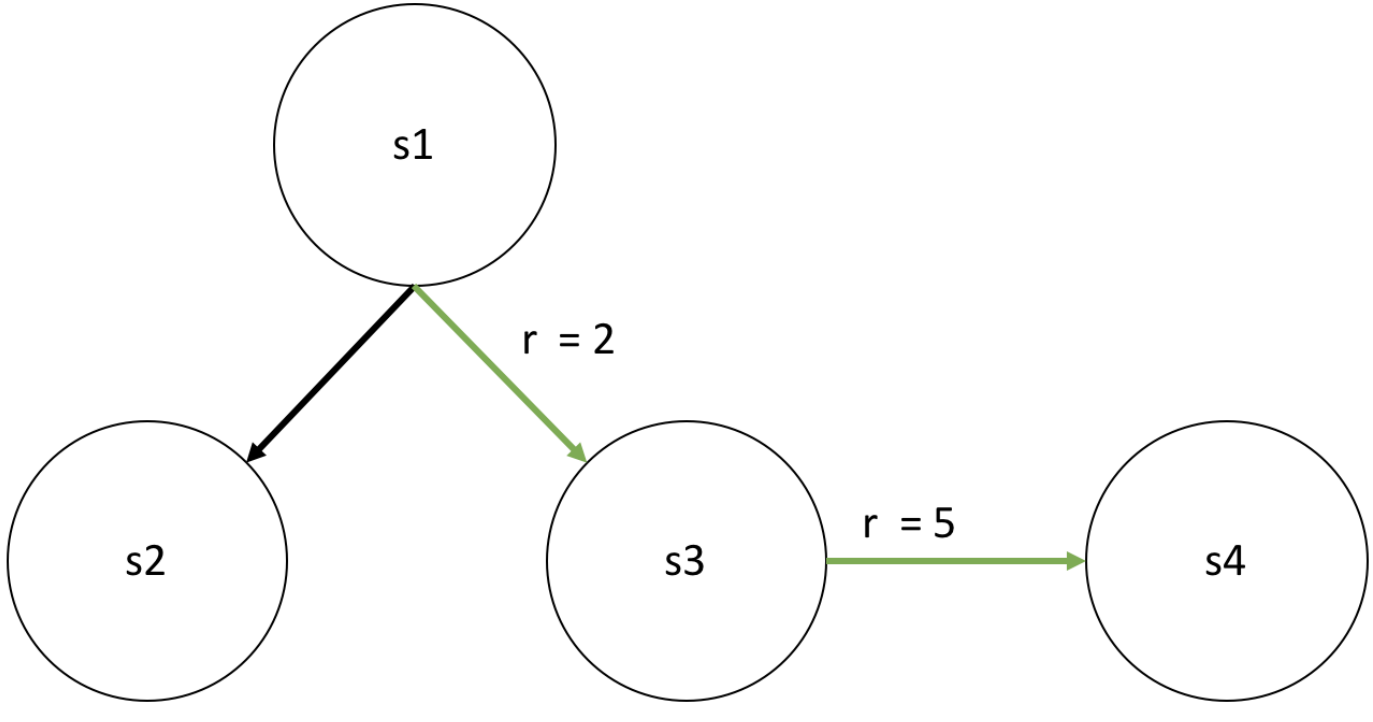
```
returns = defaultdict(list)

episode = run_full_episode(env, policy)

for experience in episode:
    returns = episode.calc_return(experience)

    returns[experience.state].append(return)

value_estimate = np.mean(returns[state])
```

Monte Carlo can focus

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

because we use experienced state transitions

can focus on high value states in large state spaces (ignore the rest of the space)



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 samples of experience

- like Monte Carlo
- no environment model

Bootstrap

- like dynamic programming
- learn online

Episodic & non-episodic problems

Temporal difference error

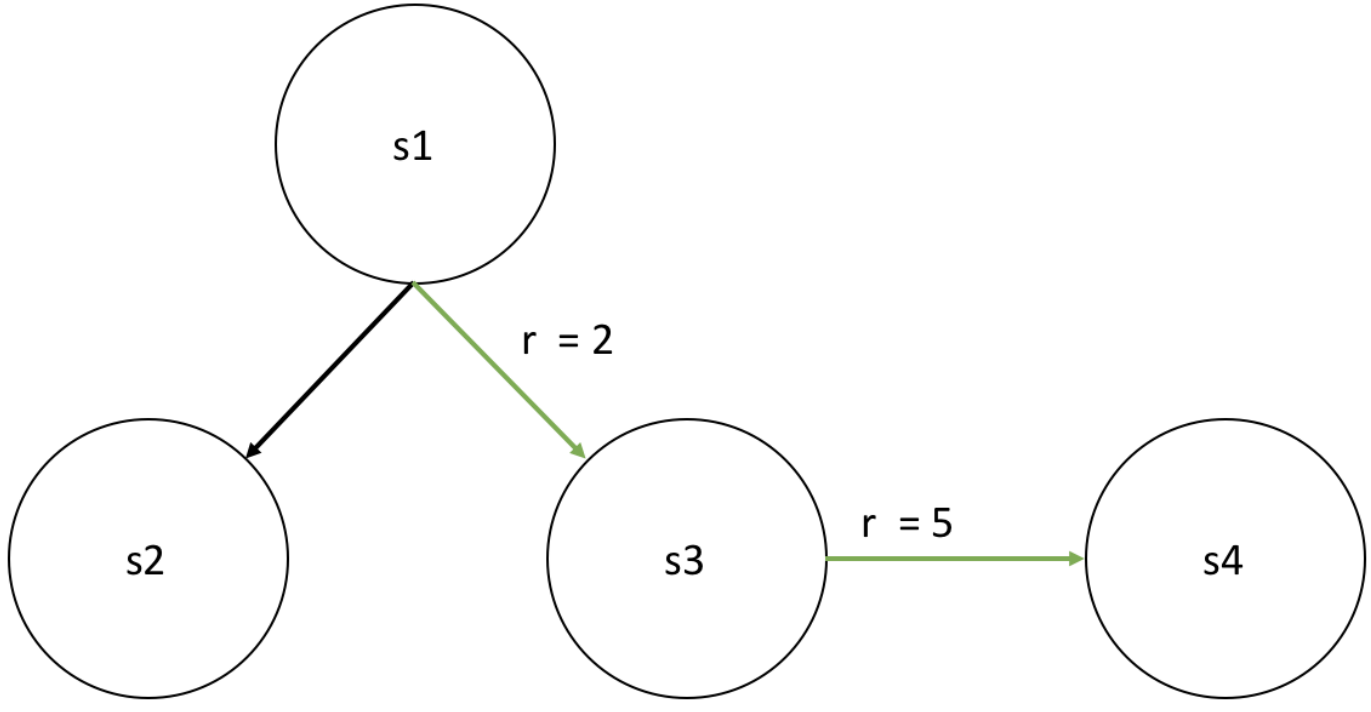
$$\text{error} = r + \gamma V(s') - V(s)$$

Update rule for a table

$$V(s_t) \leftarrow V(s_t) + \alpha[R_{t+1} + \gamma \cdot V(s_{t+1}) - V(s_t)]$$

Update rule for a neural network

$$\nabla_\theta (r + \gamma Q(s', a; \theta) - (Q(s, a; \theta))^2$$



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

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

$$error = target - approximation$$

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

$$error = r + Q(s', a) - Q(s, a)$$

Q-Learning

Function approximation and data labelling are a means to an end

- control is what we really want
- optimal actions

Q-Learning

- off-policy control
- based on the action-value function $Q(s, a)$

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 the value of all possible next states

$$V(s_1) = 10$$

$$V(s_2) = 5$$

$$V(s_3) = 20$$

$V(s)$ 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

need additional info (state transition probabilities) to select an action

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

select best action by $argmax$ across the action space

SARSA

SARSA = state, action, reward, next state, next action

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

on-policy control

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

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

Why is the value function learnt by SARSA on-policy?

we learn about the action a' that our agent choose to take

Q-Learning

Off-policy control

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

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

We take the maximum over all possible next actions

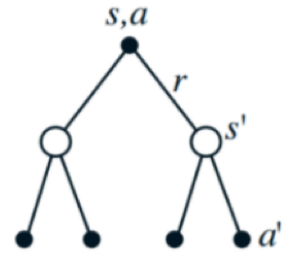
we don't need to know what action our agent took next (i.e. a')

learn the optimal value function while following a sub-optimal policy

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

SARSA versus Q-Learning

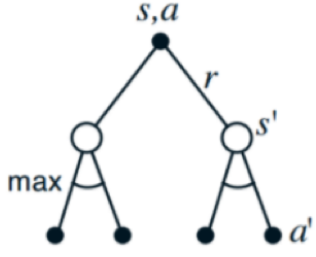
$q_\pi(a,s)$



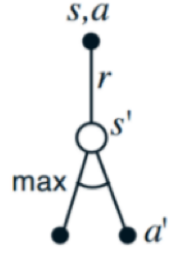
Q-policy evaluation

Sarsa

$q_*(a,s)$



Q-value iteration



Q-learning

Q-Learning

Select optimal actions by $argmax$ across the action space

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

the $argmax$ limits Q-Learning to discrete action spaces only

Issues with the argmax

positively biased (see DDQN)

aggressive

Lets talk about the *argmax*

Small changes in $Q(s, a)$ 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!

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

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

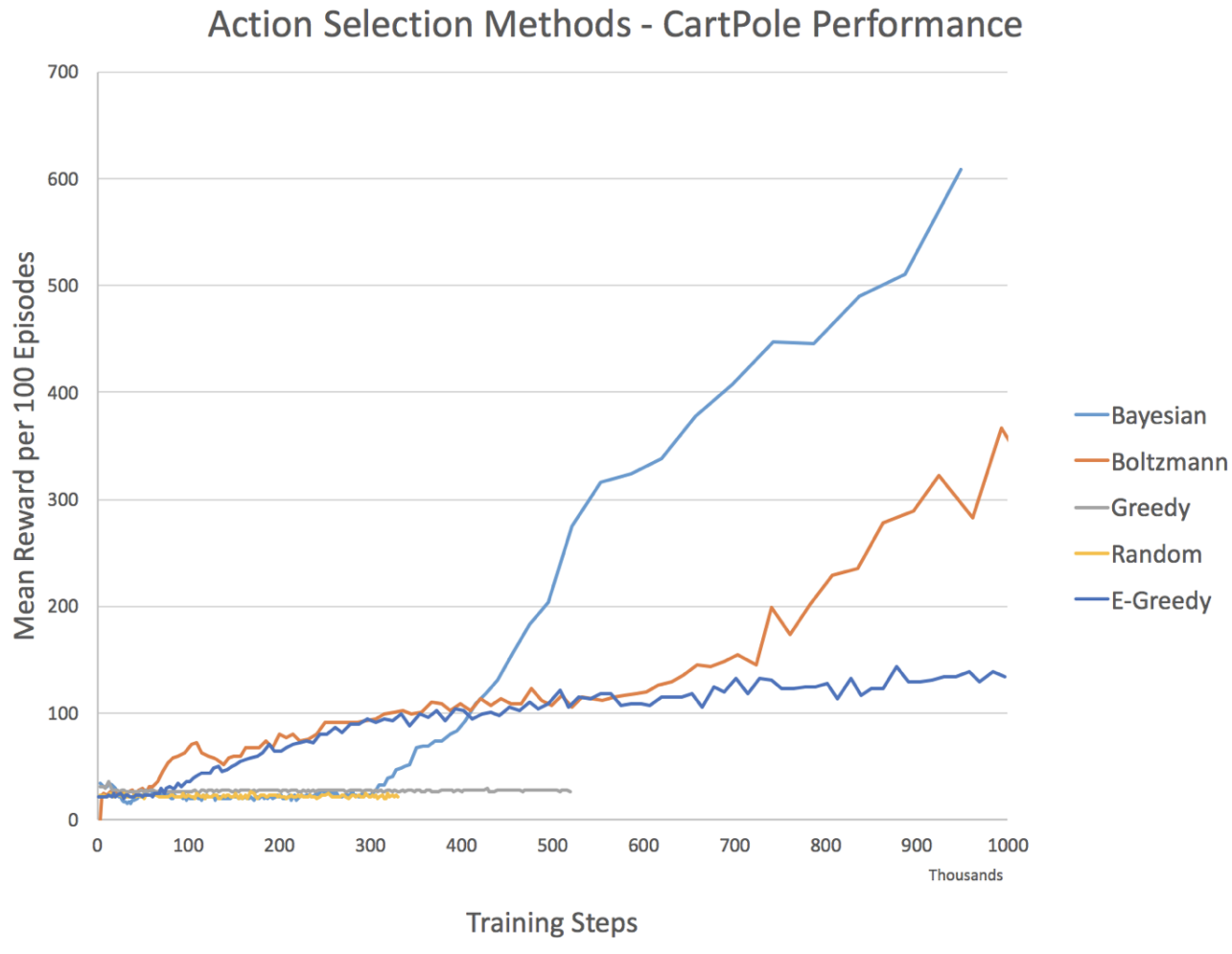
this can also be performed using dropout to simulate a probabilistic network

Parameter noise

adding adaptive noise to weights of network

[Action-Selection Strategies for Exploration](https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf)

[Plappert et al. (2018) Paramter Space Noise for Exploration](https://arxiv.org/pdf/1706.01905.pdf)



[Action-Selection Strategies for Exploration](https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-7-action-selection-strategies-for-exploration-d3a97b7cceaf)

Deadly triad

Emergent phenomenon that produces instability. Driven by:

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

It's not clear what causes instability

dynamic programming can diverge with function approximation - on-policy divergence

prediction can diverge

linear functions can be unstable

Sutton & Barto

Up until 2013 the deadly triad limited the use of neural networks with Q-Learning

Then came DeepMind & DQN...

DQN

why so significant?

which two key innovations stabilized learning?

what is the shape of the output layer?

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

doi:10.1038/nature14236

Human-level control through deep reinforcement learning

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

Significance

Deep reinforcement learning

convolution

End to end

learning from raw pixels

Ability to generalize

same algorithm, many games

Reinforcement learning to play Atari

State

last four screens concatenated together - allows infomation about movement

grey scale, cropped & normalized

Reward

game score

clipped to [-1, +1]

Actions

joystick buttons (a discrete action space)

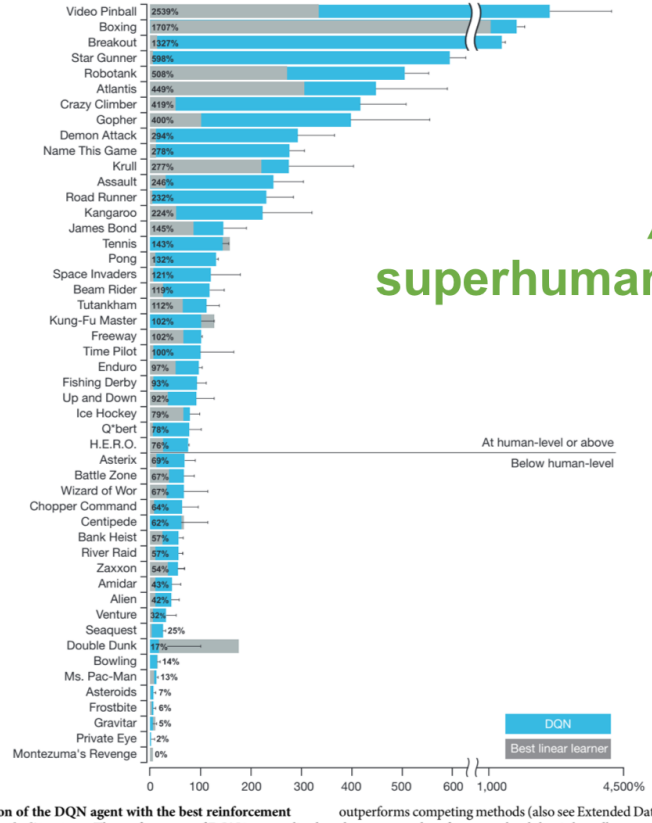
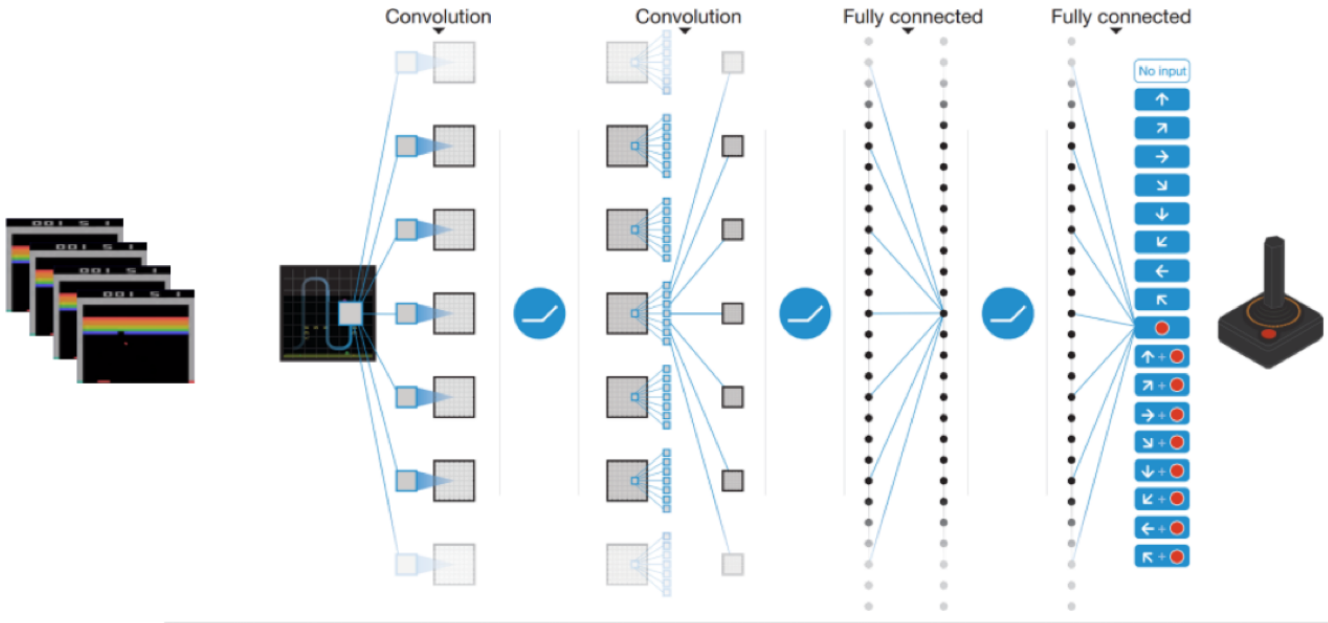


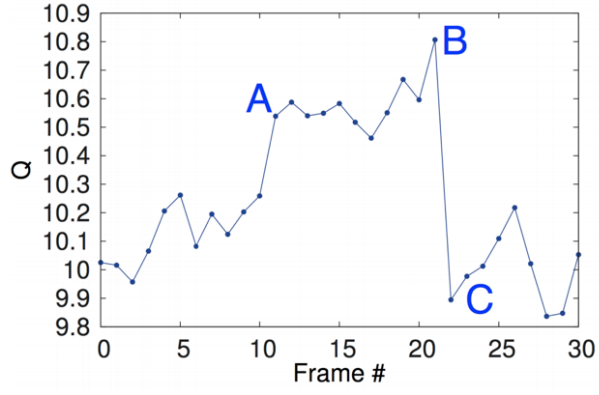
Figure 3 | Comparison of the DQN agent with the best reinforcement learning methods[15] in the literature. The performance of DQN is normalized with respect to a professional human games tester (that is, 100% level) and random play (that is, 0% level). Note that the normalized performance of DQN, expressed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



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

2013 – Playing Atari with Deep Reinforcement learning

2015 – Human-level control through deep Reinforcement learning



Agent sees new enemy

Value function gets excited

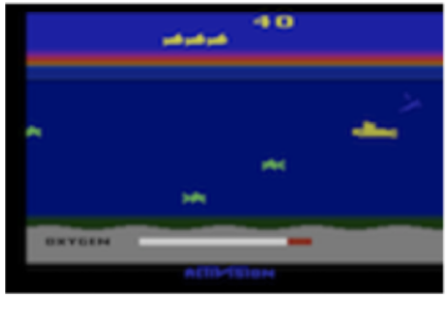
Agent has fired torpedo

Value function is very excited

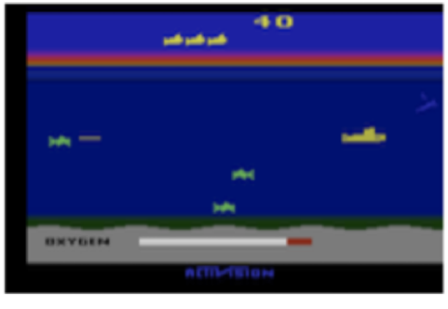
Reward received

Value function back to normal

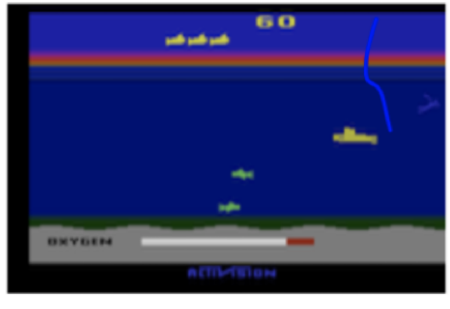
A



B



C



Two key innovations in DQN

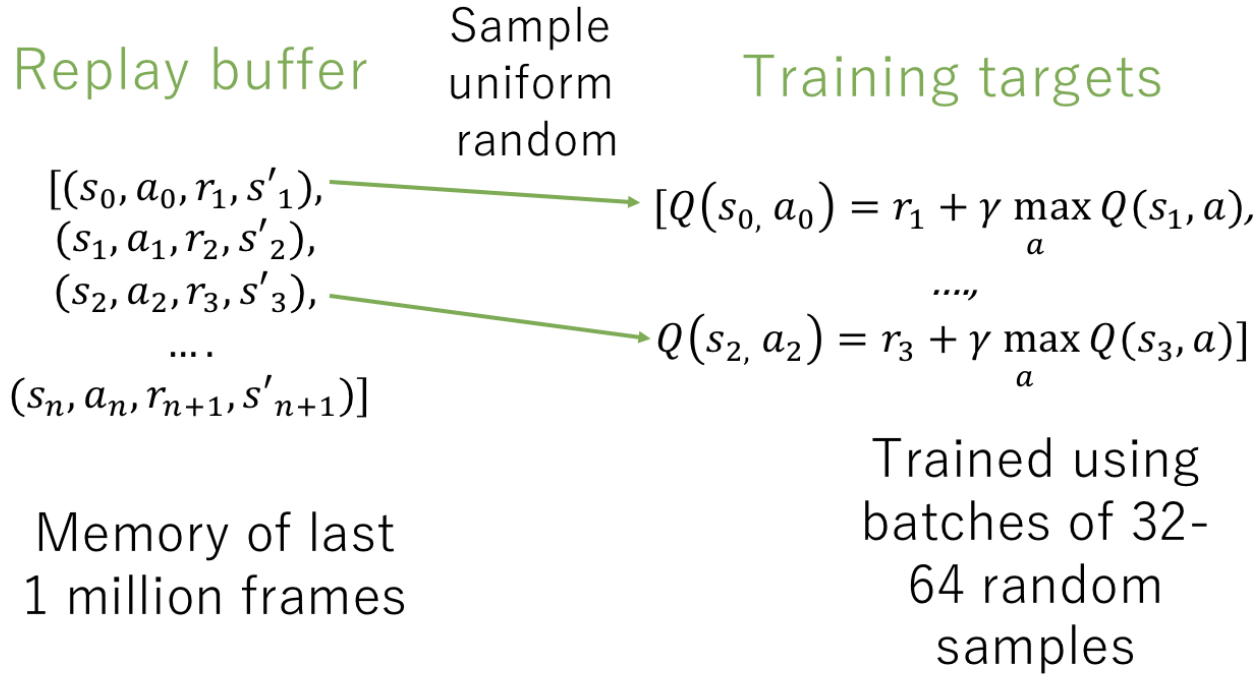
Experience replay

Target network

both improve learning stability

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

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

Can only do this because we can learn off-policy!

Biological basis for experience replay

Hippocampus may support an experience replay process in the brain

time compressed reactivation of recently experienced trajectories during off-line periods

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

Mnih et. al (2015)

Target network

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

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)

$$L(\theta_i) = [r + \gamma \cdot max_{a'} Q(s', a; \theta_i^-) - Q(s, a; \theta_i)]$$

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

DQN algorithm

Algorithm 1: deep Q-learning with experience replay.

Initialize replay memory D to capacity N

Initialize action-value function Q with random weights θ

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

For episode = 1, M **do**

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

  For t = 1,T **do**

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

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

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

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

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

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

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

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

  End For

End For

Mnih et. al (2015)

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 - Prioritized experience replay

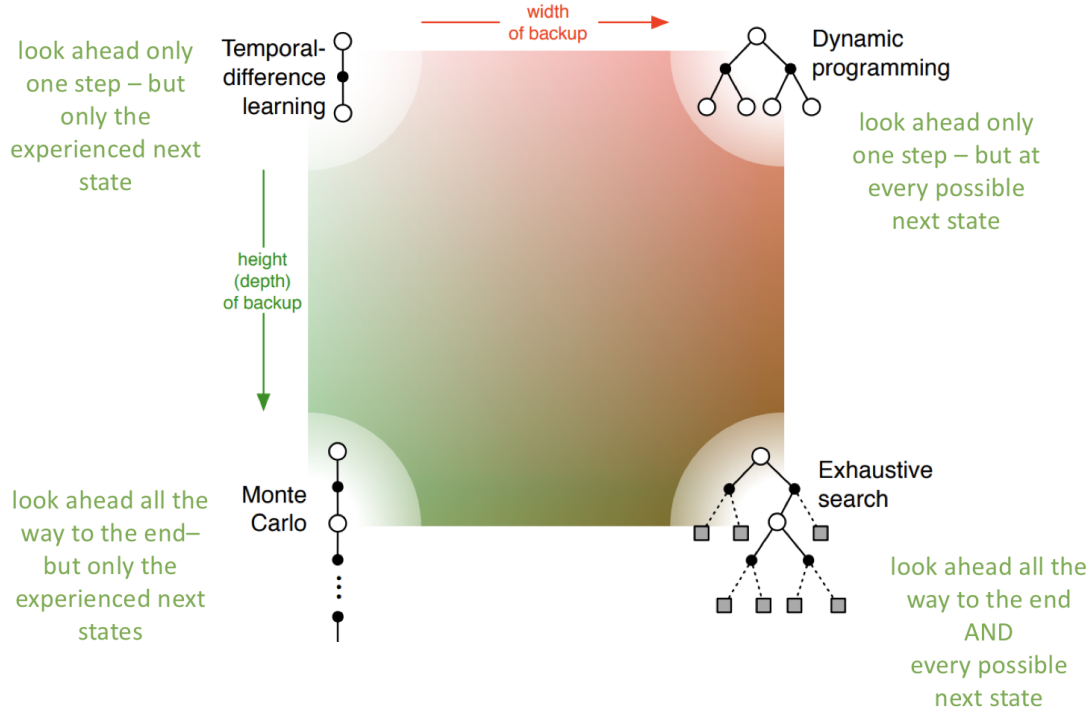
2016 - Double DQN (DDQN)

2017 - Distributional Q-Learning

2018 - Rainbow

Beyond DQN

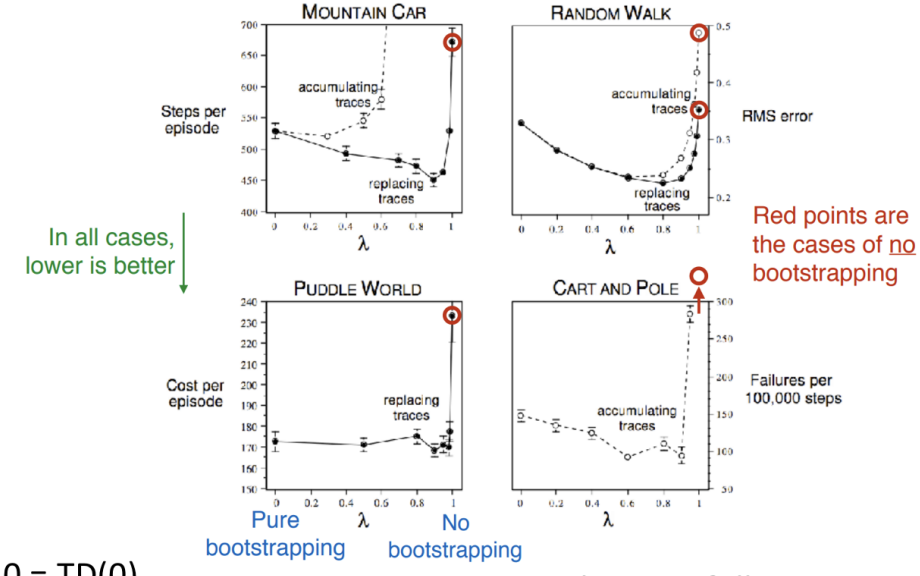
Unified View



[Sutton - Long-term of AI & Temporal Difference Learning](https://www.youtube.com/watch?v=EeMCEQa85tw)

4 examples of the effect of bootstrapping

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



λ = 0 = TD(0)

λ = 1 = full Monte Carlo

Conclusion – full Monte Carlo isn't very good!

[Sutton - Long-term of AI & Temporal Difference Learning](https://www.youtube.com/watch?v=EeMCEQa85tw)

Eligibility traces

Family of methods between Temporal Difference & Monte Carlo

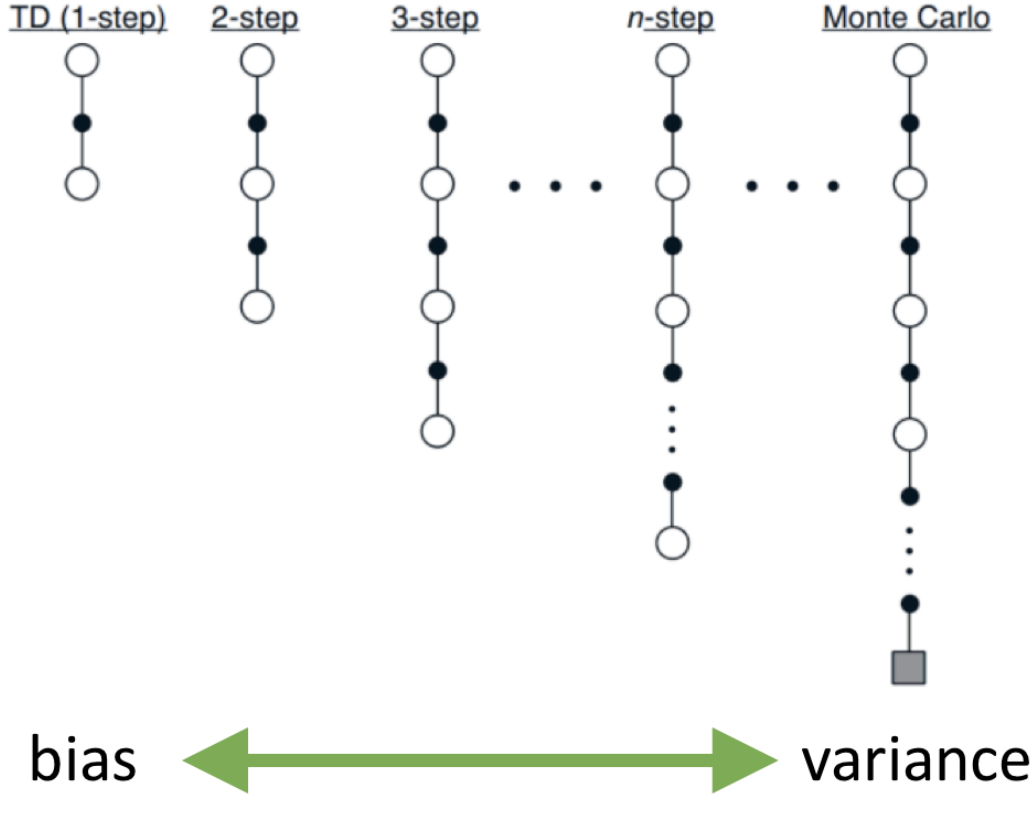
Eligibility traces allow us to assign TD errors to different states

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

Allow us to trade-off between bias and variance

n-step returns

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



Sutton & Barto

Forward and backward view

We can look at eligibility traces from two perspectives

The forward view

helpful for understanding the theory

The backward view

can be put into practice

The forward view

assign future returns to the current state

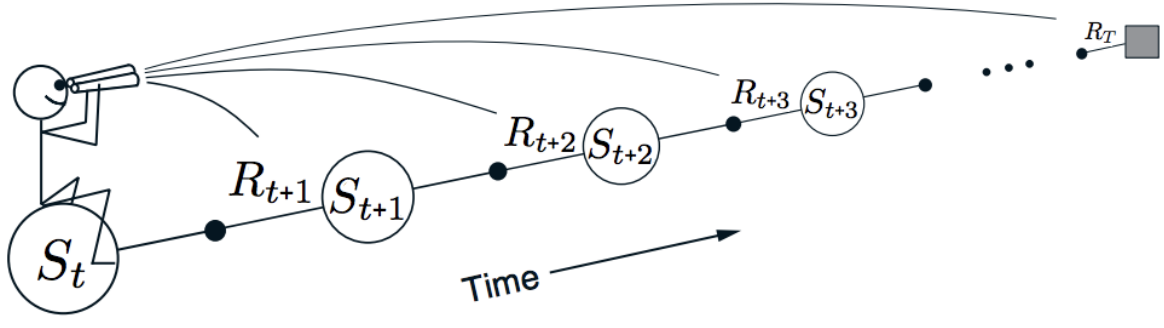


Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

Sutton & Barto

The backward view

assign TD error to previous states

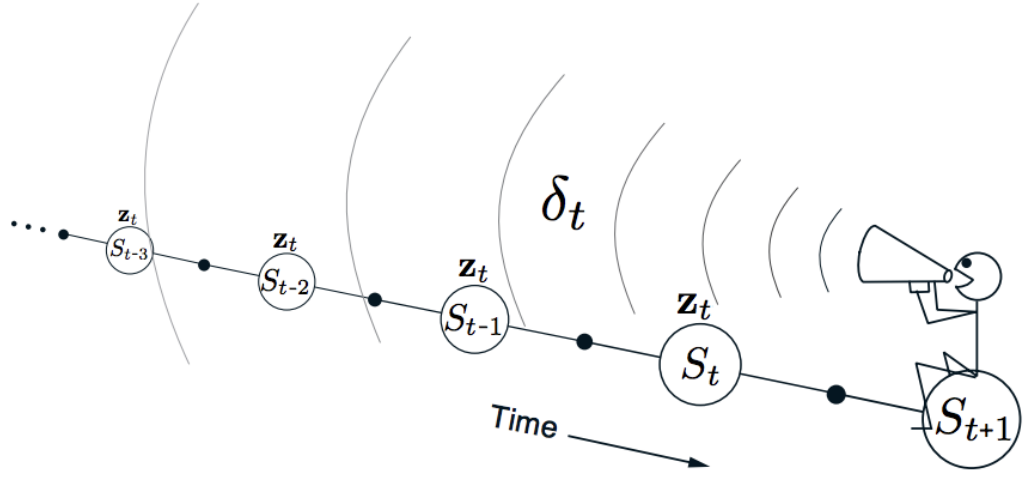


Figure 12.5: The backward or mechanistic view. Each update depends on the current TD error combined with the current eligibility traces of past events.

Sutton & Barto

The backward view

The backward view approximates the forward view

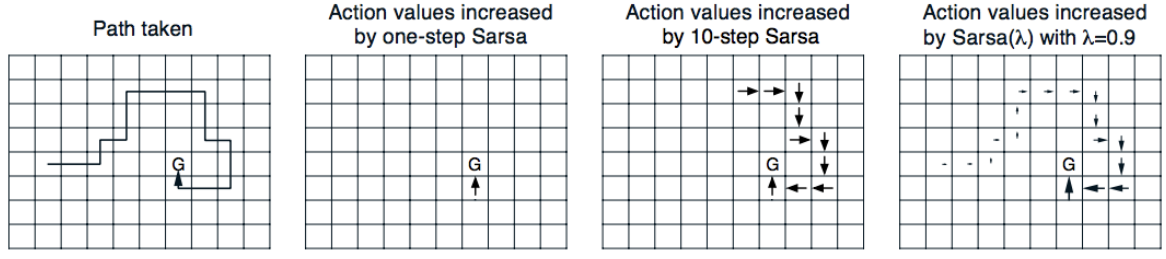
forward view is not practical (requires knowledge of the future)

We need to remember which states we visited in the past

it requires an additional variable in our agents memory

eligibility trace $e_t(s)$

Example 12.1: Traces in Gridworld The use of eligibility traces can substantially increase the efficiency of control algorithms over one-step methods and even over n-step methods. The reason for this is illustrated by the gridworld example below.



The first panel shows the path taken by an agent in a single episode. The initial estimated values were zero, and all rewards were zero except for a positive reward at the goal location marked by G. The arrows in the other panels show, for various algorithms, which action-values would be increased, and by how much, upon reaching the goal. A one-step method would increment only the last action value, whereas an n-step method would equally increment the last n action's values, and an eligibility trace method would update all the action values up to the beginning of the episode to different degrees, fading with recency. The fading strategy is often the best tradeoff, strongly learning how to reach the goal from the right, yet not as strongly learning the roundabout path to the goal from the left that was taken in this episode. ■

Sutton & Barto

one step method would only update the last $Q(s, a)$

n-step method would update all $Q(s, a)$ equally

eligibility traces updates based on how recently each $Q(s, a)$ was experienced

Prioritized Experience Replay

Naive experience replay

randomly samples experience

Some experience is more useful for learning than others

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

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

TD error is an indication of how useful a transition is for learning

measures surprise

Prioritized Experience Replay

Non-random sampling introduces two problems

Loss of diversity

- we will only sample from high TD error experiences
- solve by making the prioritization stochastic

Introduce bias

- non-independent sampling
- correct bias using importance sampling

DDQN

DDQN = Double Deep Q-Network

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

Tabular Q-Learning used two different approximations of $Q(s, a)$

- one to quantify the value of actions
- one to select actions

Double Q-Learning aims to overcome the maximization bias of Q-Learning

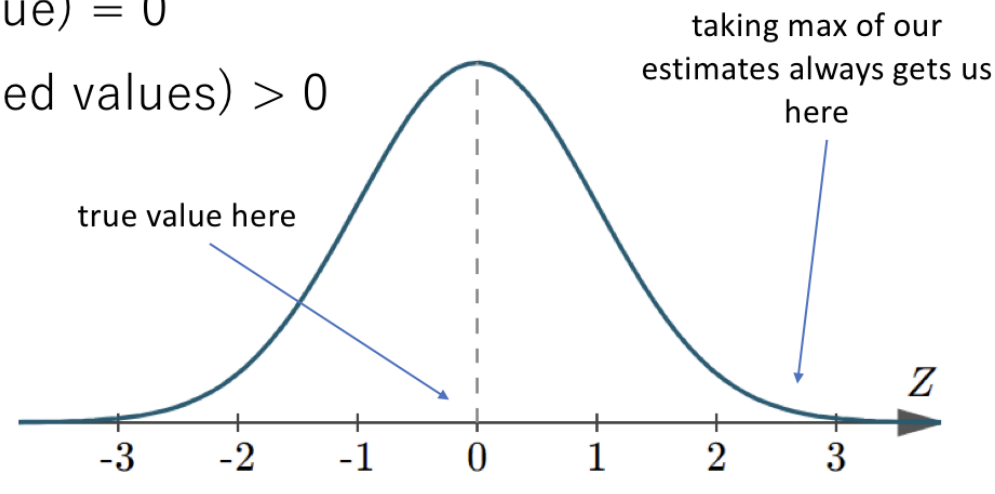
Maximization bias

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

Use the target network as a different approximation of $Q(s, a)$

Original DQN target

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

DDQN target

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

select the action according to the online network

quanitfy the value that action using the target network

Both Q networks still have noise - but the noise is decorrelated

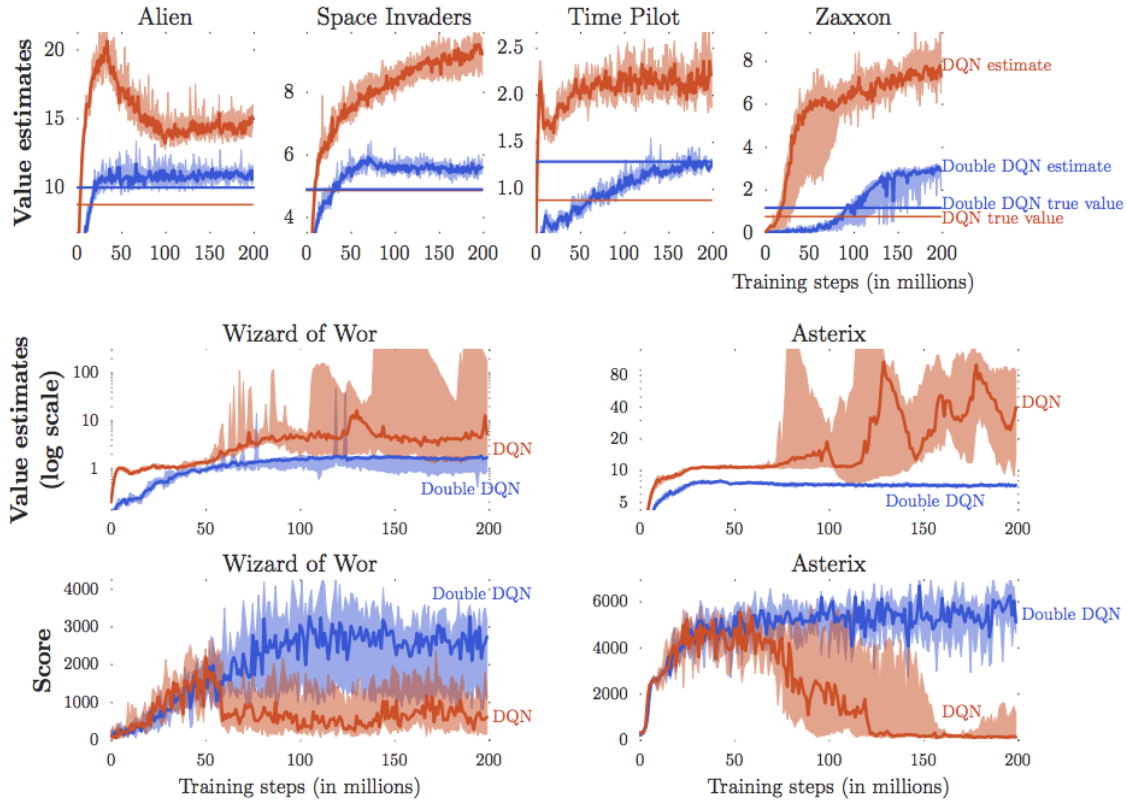


Figure 3: The **top** and **middle** rows show value estimates by DQN (orange) and Double DQN (blue) on six Atari games. The results are obtained by running DQN and Double DQN with 6 different random seeds with the hyper-parameters employed by Mnih et al. (2015). The darker line shows the median over 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.

van Hasselt et. al. (2015)

Rainbow

What if we combined all the improvements together?

All the various improvements to DQN address different issues

DDQN - overestimation bias

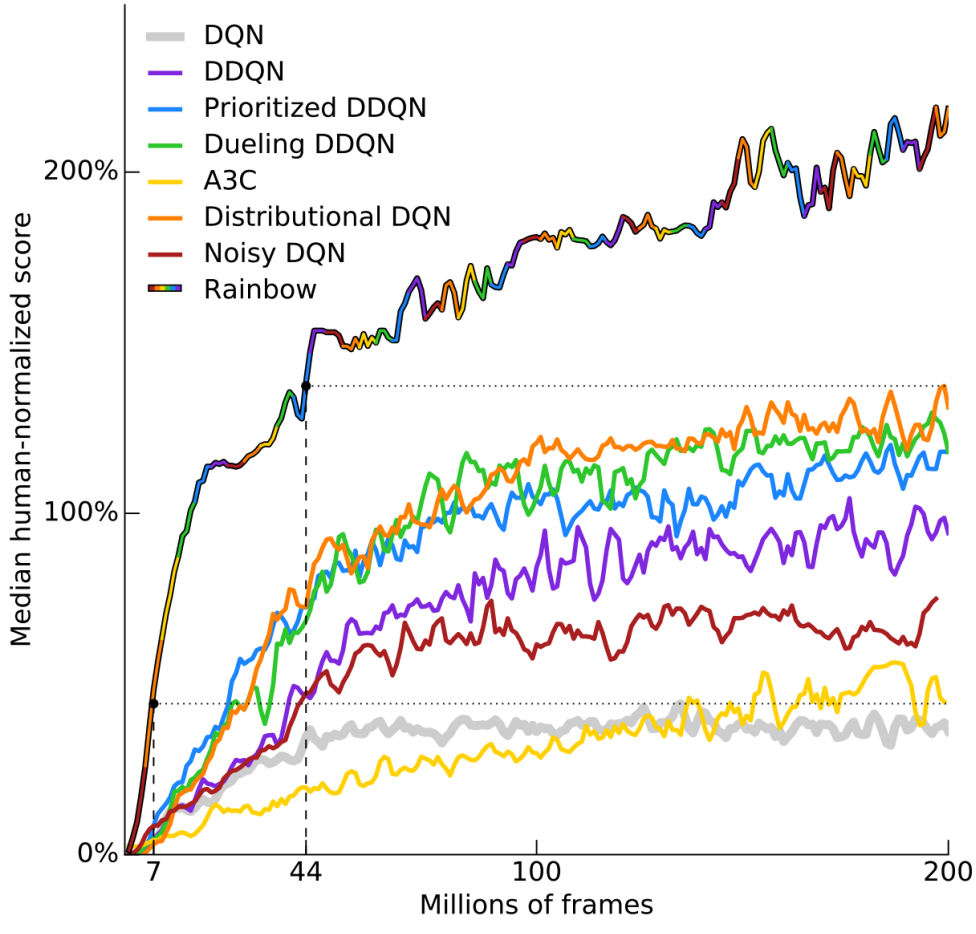
prioritized experience replay - sample efficiency

duelling - generalize across actions

multi-step bootstrap targets - bias variance trade-off

distributional Q-learning - learn categorical distribution of $Q(s, a)$

noisy DQN - stochastic layers for exploration



Median human-normalized performance across 57 Atari games. We compare our integrated agent (rainbow colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.

van Hasselt et. al. (2017)

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

van Hasselt et. al. (2017)

Policy optimization

how do I parametrize a continuous action space?

how is the score function used to find the policy gradient?

why use baselines?

what do the actor and critic do?

Policy gradients

Value function methods generate a policy from a learnt value function

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

Policy optimization parametrizes and improves the policy directly

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

Deep Reinforcement Learning (John Schulman, OpenAI)

Motivations for policy gradients

High dimensional action spaces

robotics

Optimize what we care about

optimize return

Less sample efficient

usually on-policy learning

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)]
```

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

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

Motivation - optimize return directly

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

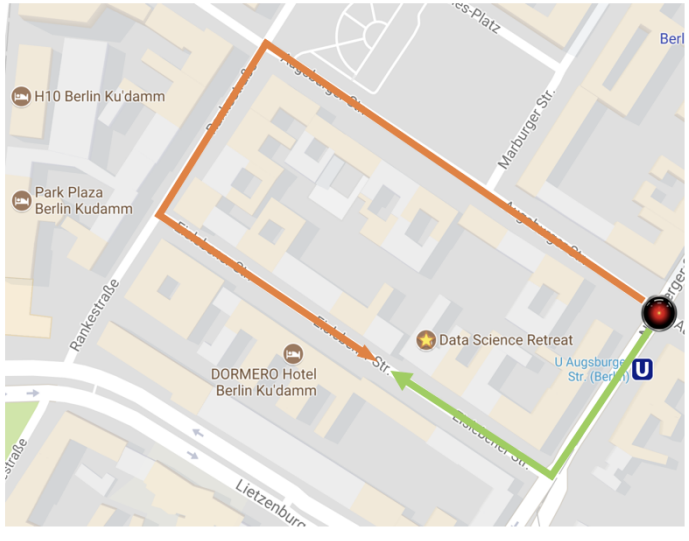
changing weights according to the gradient that maximizes future reward

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



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

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

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

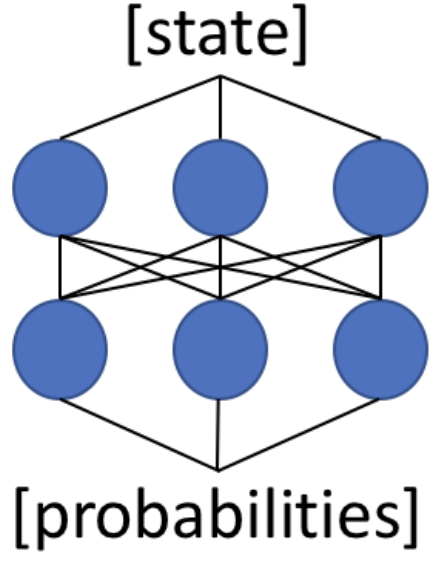
vs

$a_2 \sim \pi(s)$

Parametrizing policies

discrete action space

output layer = softmax

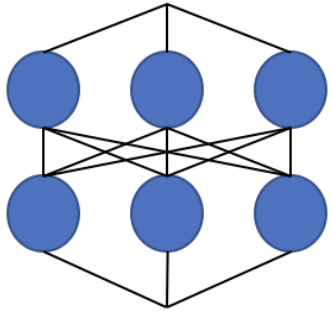


Parametrizing policies

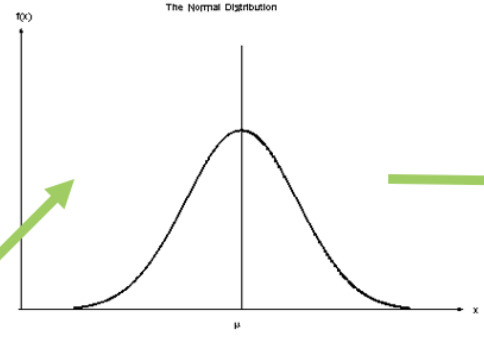
continuous action space

output layer = mean & stdev

[state]



[mean, stdev]



Sample an action from the distribution

Policy gradients without equations

We have a parametrized policy

a neural network that outputs a distribution over actions

How do we improve it - how should we change our parameters?

take actions that get more reward

favour probable actions

Reward function is not known

so we can't just take a gradient wrt reward

Policy gradients with a few equations

Our policy is a probability distribution over actions

How do we improve it?

take actions that get more reward

favour probable actions

Reward function is not known

but we can find the gradient of expected reward

find gradients to increase return

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

The score function

The score function allows us to get the gradient of the expectation of a function

i.e. expected reward

derived using the log-likelihood ratio trick

Expectations are averages

this allows us to use sample based methods (no need for full transition probabilities as in dynamic programming)

i.e. we can approximate the RHS using samples

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

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

Deriving the score function

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

start with the definition of expectation

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

take the gradient with respect to our neural network weights

$$\nabla_\theta \mathbf{E}_x[f(x)] = \nabla_\theta \sum_x p(x) \cdot f(x)$$

swap sum and gradient

$$\nabla_\theta \mathbf{E}_x[f(x) = \sum_x \nabla_\theta p(x) \cdot f(x)$$

multiply and divide by $p(x)$

using the fact that $\nabla_\theta log(p(x)) = \frac{\nabla_\theta p(x)}{p(x)}$

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

using the definition of an expectation $\sum_x p(x) \cdot f(x) = \mathbf{E}_x[f(x)]$

We end up with the ability to find the gradient of a function we don't have access too

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

but we must be able to sample from it - to approximate the expectation on the RHS

[Deep Reinforcement Learning: Pong from Pixels - Andrej Karpathy](http://karpathy.github.io/2016/05/31/rl/)

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

Training a policy

We use the score function to get a gradient

```
gradient = log(probability of action) * expected_return

gradient = log(policy) * expected_return
```

The score function limits us to on-policy learning

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

REINFORCE

Williams (1992) - Simple statistical gradient-following algorithms for connectionist reinforcement learning

Different methods to approximate the return G_t

- REINFORCE uses a Monte Carlo estimate
- we use actual sampled discounted reward

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

- high variance
- no online learning
- requires episodic environment

Baseline

We can introduce a baseline function

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

this reduces variance (smaller gradients) without introducing bias

a natural baseline is the value function (weights w)

This also gives rise to the concept of advantage

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

how much better this action is than the average action

Actor Critic

Actor Critic brings together value functions and policy gradients

We parametrize two functions

actor = policy

critic = value function

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

$$\nabla_\theta \mathbf{E}[G] = \mathbf{E}[\nabla_\theta log(\pi_\theta(a|s) \cdot Q(s,a)]$$

```
while not done:
    action = agent.act(observation)
    next_observation, reward, done, info = env.step(action)

    agent.update_critic()

    agent.update_policy()
```

Advantage Actor Critic (A2C)

Advantage

$$A_\pi(s,a) = Q_\pi(s,a) - V_\pi(s)$$

how much better this action is than the average action

We don't have to learn an additional network

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

$$A_\pi(s,a) = r + \gamma V_\pi(s') - V_\pi(s)$$

Asynchronous Advantage Actor Critic (A3C)

Multiple agents learning in parallel

- update based on fixed length's of experience (say 20 timesteps)
- share parameters between value and policy networks
- asynchronous updates of parameters

Different policies can be run in each environment

- exploration

Natural Policy Gradients, TRPO and PPO

All three of these papers build on the same idea - that we want to constrain policy updates to get more stable learning

Natural Policy gradients - rely on a computationally intense second order derivative method (inverse of the Fisher Infomation matrix)

TRPO - uses the KL-divergence to hard constrain policy updates (avoids calculating the Fisher Infomation matrix, but uses Conjugate Gradient to solve a constrained optimization problem)

PPO - uses clipped probability ratios to constrain policy updates

By constraining the policy update, we can learn off-policy

Model based reinforcement learning

All modern reinforcement learning is model based

Poor sample efficiency means simulation is required

A simulator is an environment model!

Lots of research aimed at developing model-free algorithms

Model-based is less well developed but has advantages

for a perfect, distributional model can use dynamic programming

learning a model is a key challenge

Why learn and use environment models?

We use low dimensional mental models to represent the world around us.

Our brain learns abstract representations of spatial and temporal infomation. Evidence also suggests that perception itself is governed by an internal prediction of the future, using our mental models.

This predictive model can be used to perform fast reflexive behaviours when we face danger.

Kinds of models

Sample versus distributional

Local versus global

If you have a perfect model

If you need to learn the model

Monte Carlo Tree Search

Ha & Schmidhuber (2018) World Models

paper - blog post - blog post appendix

Learning within a dream - agent learns using a generative environment model

- vision that compresses pixels into a latent space
- LSTM memory that makes predictions of the latent space
- low dimensional controller that maps from vision + LSTM to action

Solved a previously unsolved car racing task (plus VisDoom)

- continuous action space
- learning from pixels

Component	Model	Parameter Count
Vision	VAE	4,348,547
Memory	MDN-RNN	422,368
Controller	CMA-ES	867

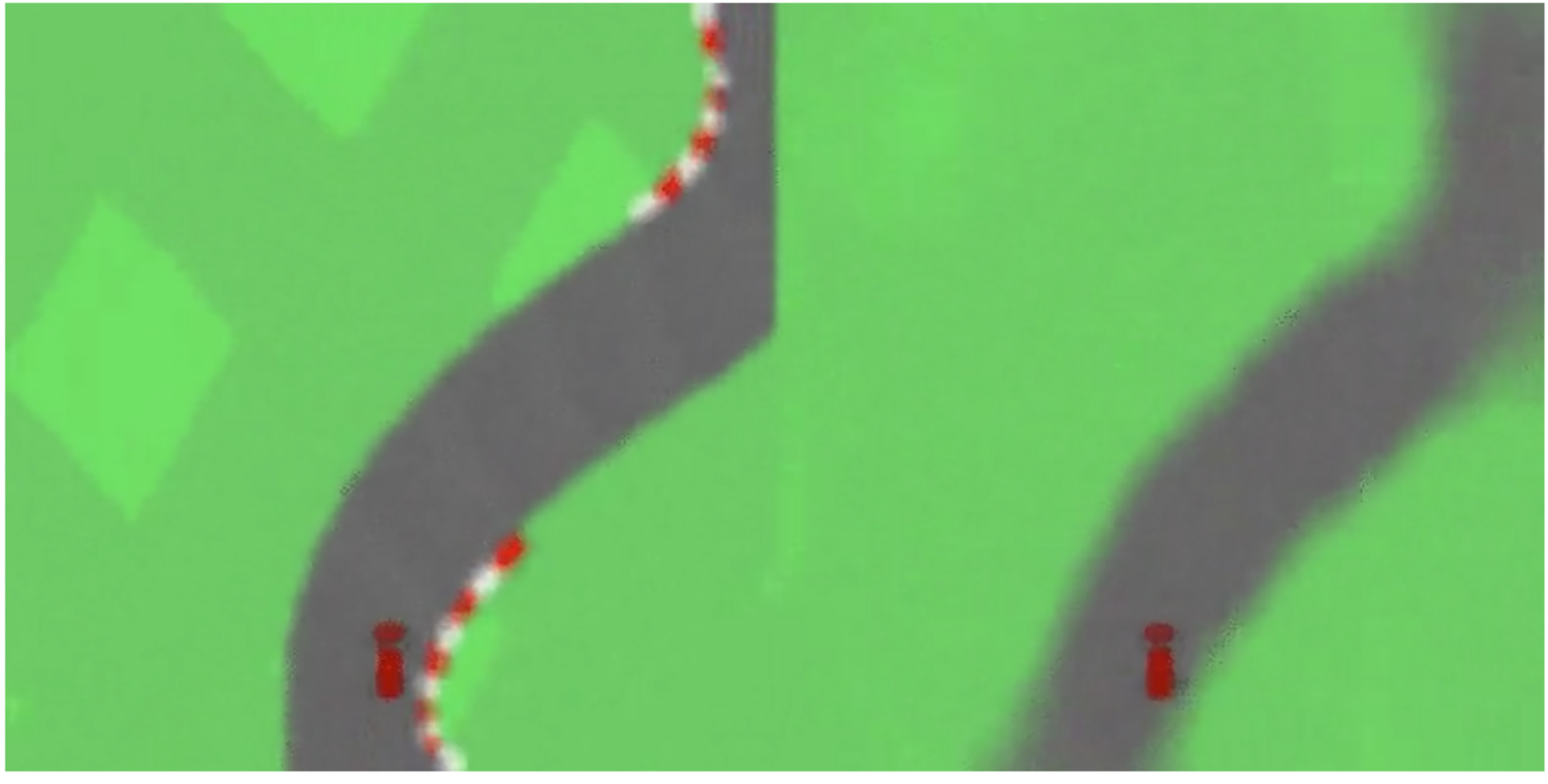
Compression

Compression used to

- reduce the dimensionality of the observation into a latent representation (z)
- compress the latent representation over time

Allows a compact linear policy to be used for control

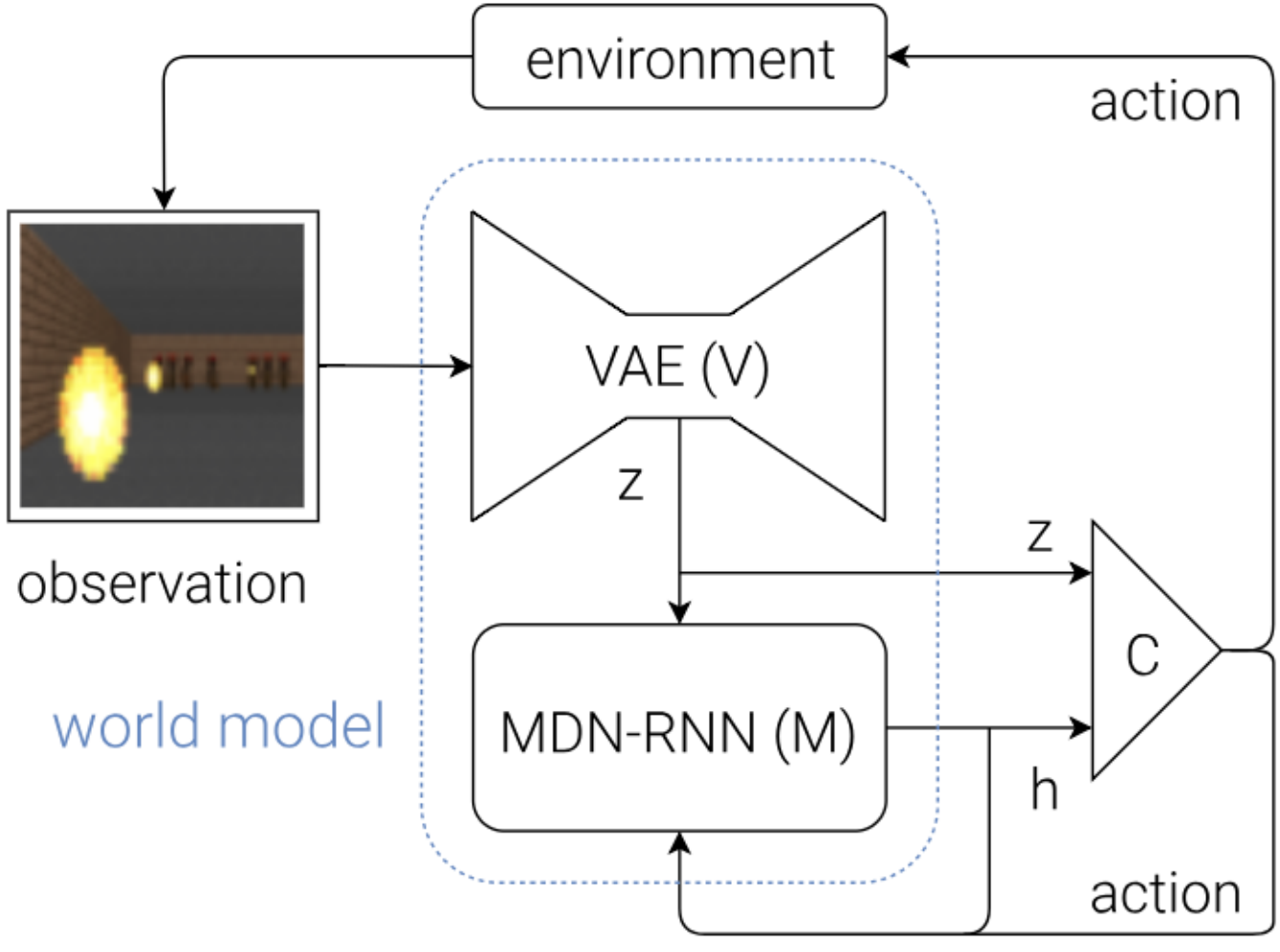
- the policy parameters are learnt using an evolutionary algorithm



Actual observations from the environment.

What gets encoded into z_t.

Actual and reconstructed observations produced by the autoencoder. Note the reconstructed observation is not used by the controller - it is shown here to compare the quality with the actual observation.


environment
action
VAE (V)
z
observation
z
C
world model
MDN-RNN (M)
h
action

At each time step, our agent receives an **observation** from the environment.

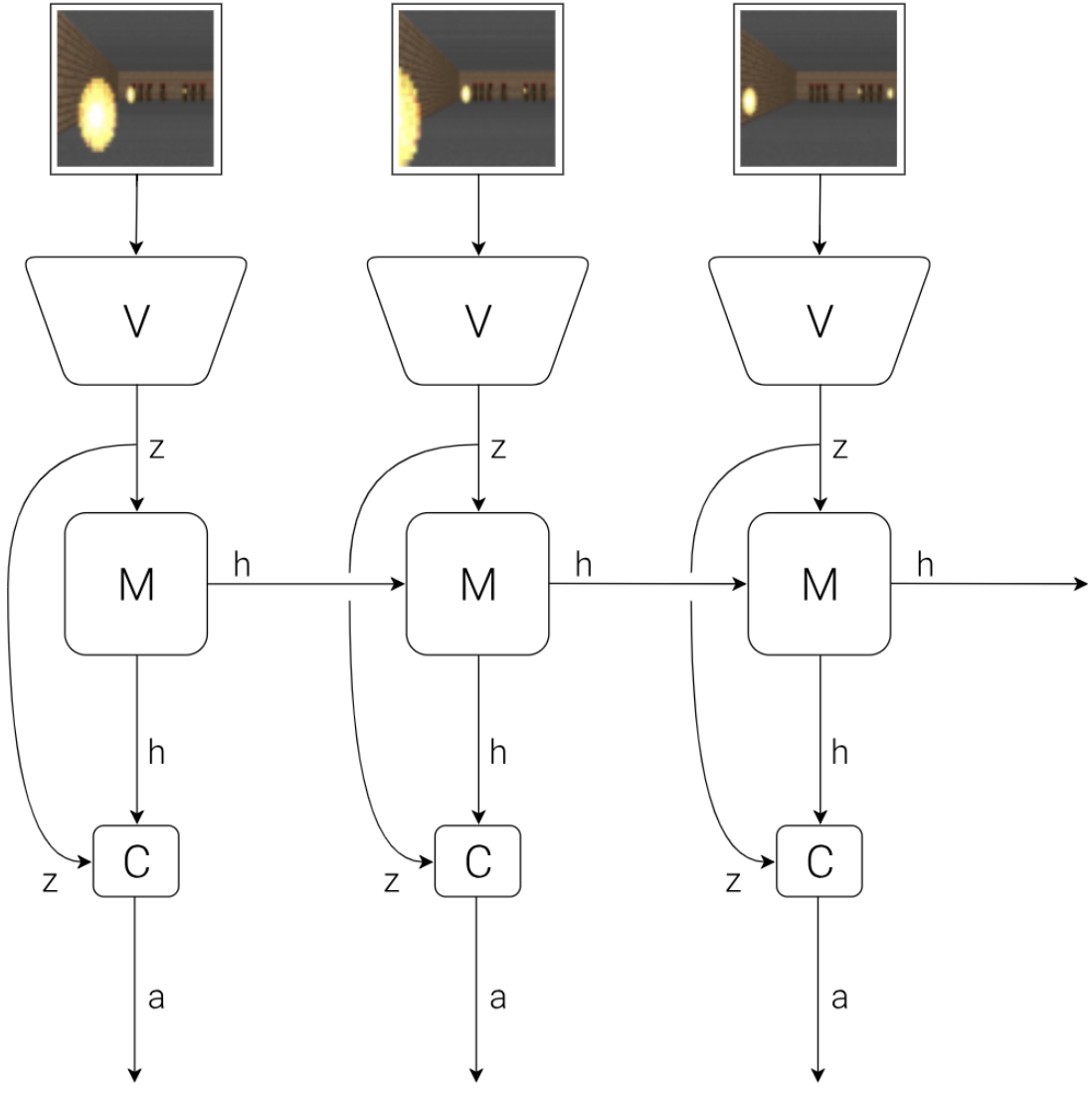
World Model

The **Vision Model (V)** encodes the high-dimensional observation into a low-dimensional latent vector.

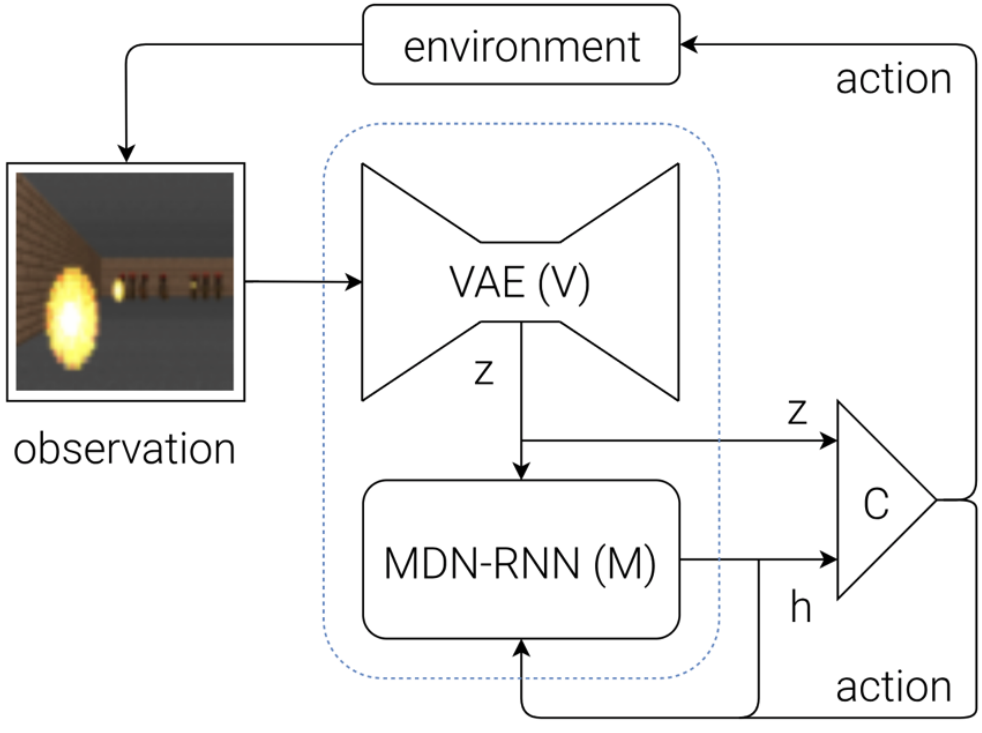
The **Memory RNN (M)** integrates the historical codes to create a representation that can predict future states.

A small **Controller (C)** uses the representations from both **V** and **M** to select good actions.

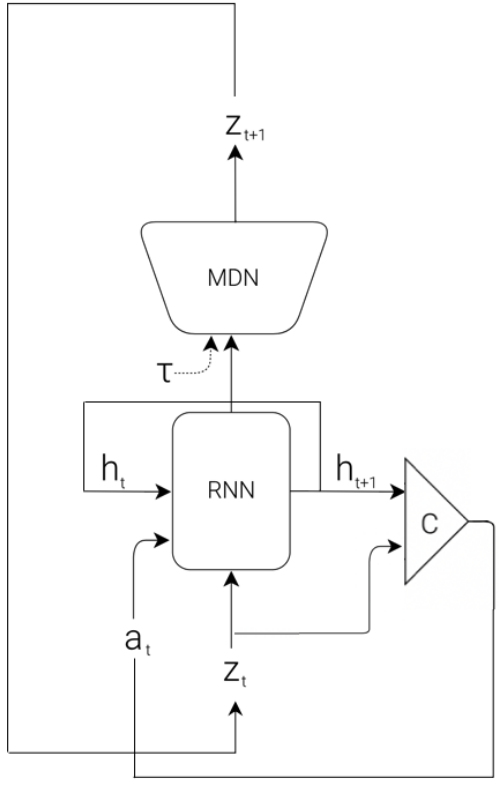
The agent performs **actions** that go back and affect the environment.



The agent consists of three components - Vision (V), Memory (M), and Controller (C)



Learning using the real environment



Learning inside a dream

Agent can learn both from the real environment and from inside it's own dream

Vision - Variational Auto Encoder

A network that compresses and reproduces the observation of the environment

the reconstructed observation is not used

the compressed version of the observation (aka the latent representation) is used as input to both the memory and the controller

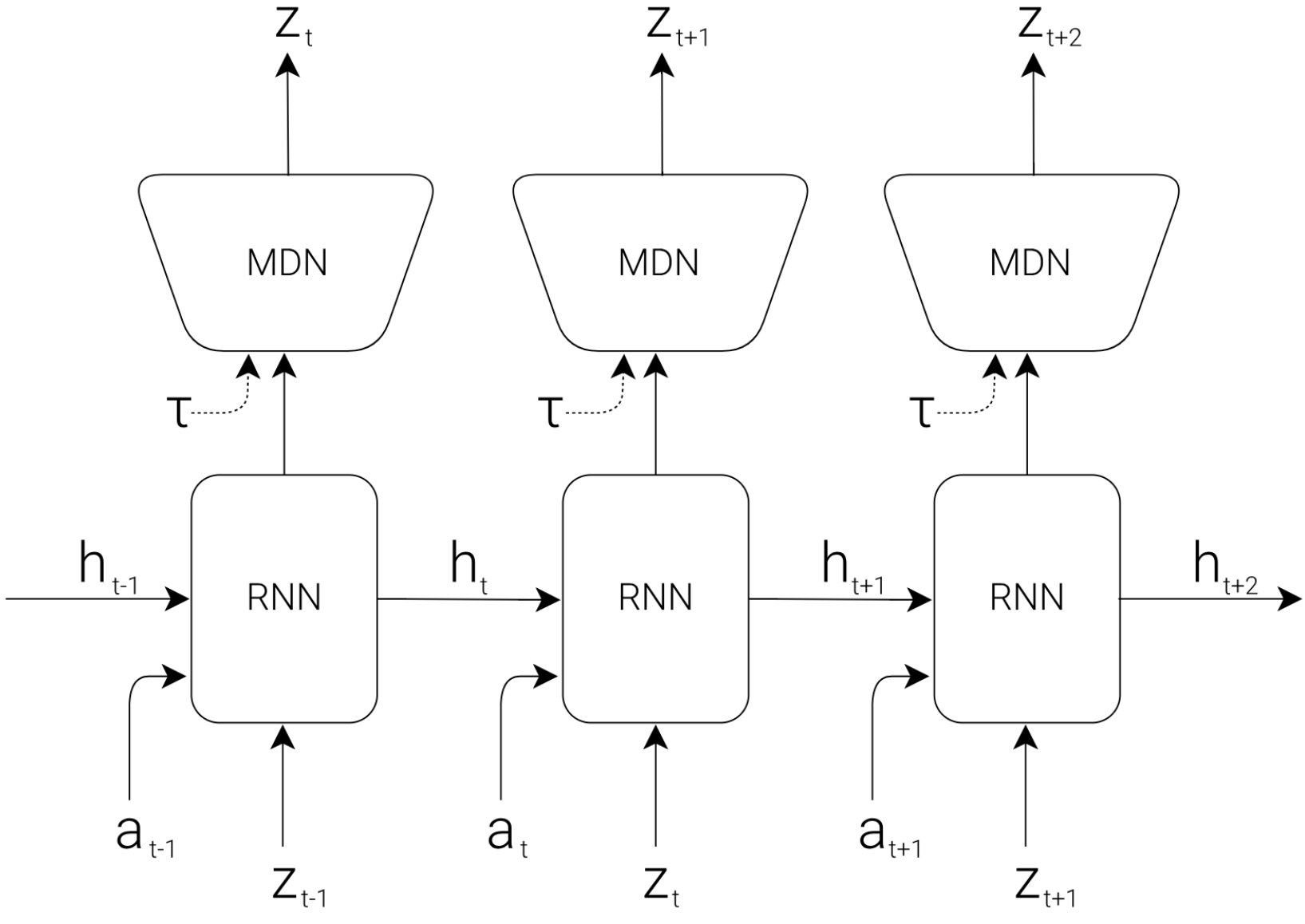
Memory - Mixed Density Recurrent Network

Predicts the latent representation of the observation

- not the raw observation
- compresses over time
- a predictive model of the future vectors that V is expected to produce.

Mixed density

- stochastic
- outputs probability density
- outputs a mixture of Gaussians



Outputs the parameters of a mixture of Gaussians used to sample a prediction of the compressed representation of state z

Controller - CMA-ES

Controller is linear

- single layer that maps memory hidden state and latent representation of observation to action
- allows simpler optimization

CMA-ES = covariance matrix adaptation evolution strategy

- derivative free
- good for up to a few thousand parameters

Performance

Method	Average Score over 100 Random Tracks
DQN [53]	343 ± 18
A3C (continuous) [52]	591 ± 45
A3C (discrete) [51]	652 ± 10
ceobillionaire's algorithm (unpublished) [47]	838 ± 11
V model only, z input	632 ± 251
V model only, z input with a hidden layer	788 ± 141
Full World Model, z and h	$\mathbf{906 \pm 21}$

AlphaGo to AlphaZero


AlphaGo Official Trailer

IBM Deep Blue

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

Deep Blue
IBM chess computer

Garry Kasparov
World Chess Champion

Deep Blue versus AlphaGo

Deep Blue

- hand-crafted by programmers & chess grandmasters
- big lookup table

AlphaGo

- learnt from human moves & self play
- reduced search width and depth using neural networks

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

Played and studied for thousands of years

- best human play is high level
- datasets of online matches

Components of the AlphaGo agent

Three policy networks $\pi(s)$

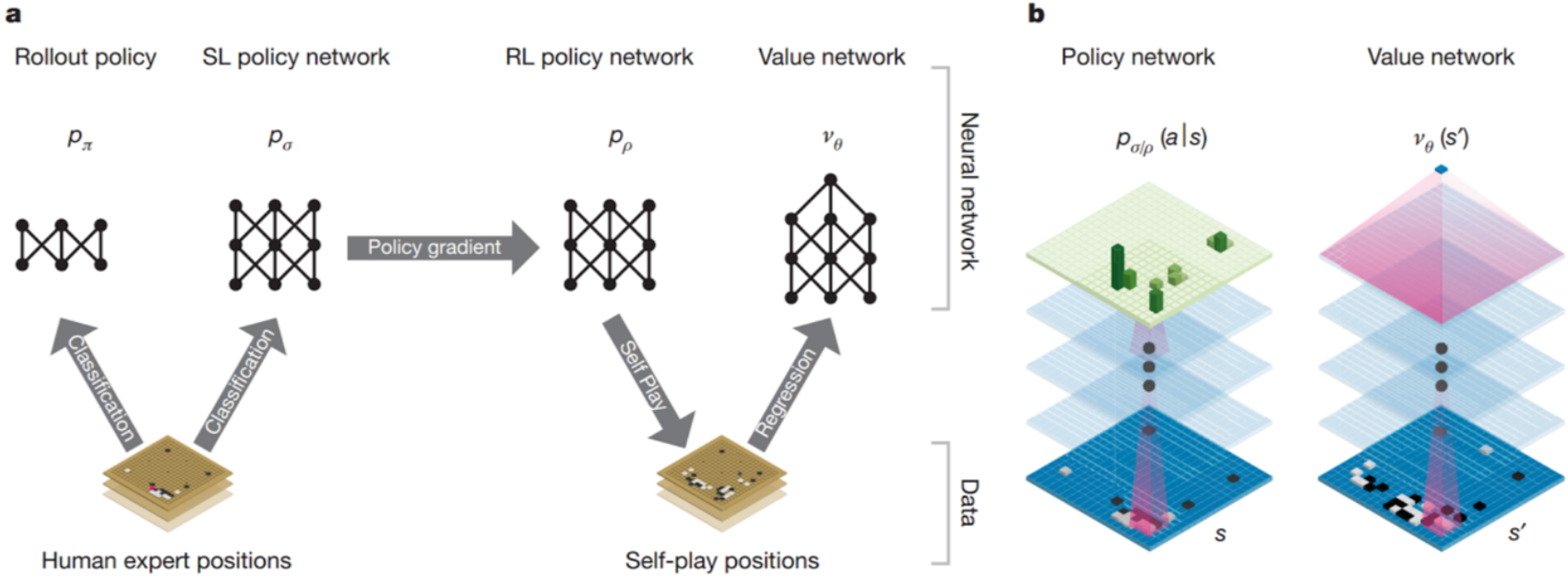
- fast rollout policy network – linear function
- supervised learning policy – 13 layer convolutional NN
- reinforcement learning policy – 13 layer convolutional NN

One value function $V(s)$

- convolutional neural network

Combined together using Monte Carlo tree search

Learning


a
Rollout policy
SL policy network
RL policy network
Value network
p_π
p_σ
p_ρ
v_θ
Policy gradient
Classification
Classification
Self Play
Regression
Neural network
Data
Human expert positions
Self-play positions
b
Policy network
Value network
$p_{\sigma/\rho}(a|s)$
$v_\theta(s')$
s
s′

Monte Carlo Tree Search

Value & policy networks combined using MCTS

Basic idea = analyse most promising next moves

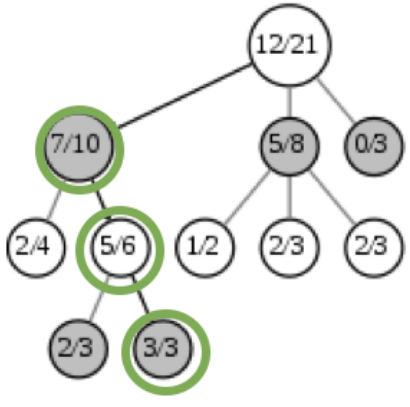
Planning algorithm

simulated (not actual experience)

roll out to end of game (a simulated Monte Carlo return)

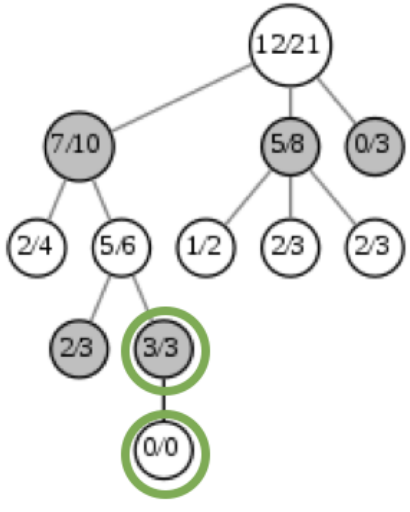
Monte Carlo Tree Search

Selection



Select nodes based on statistics collected so far

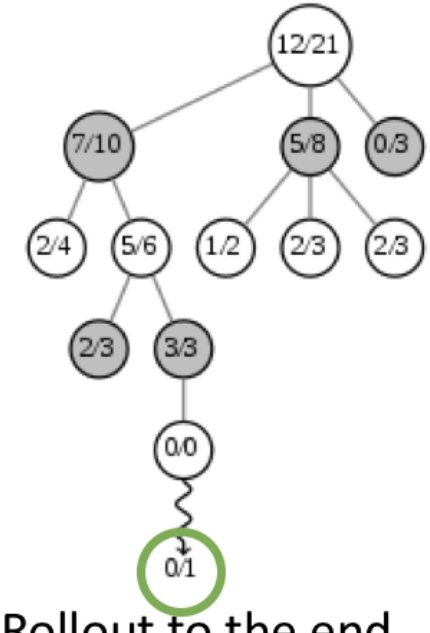
Expansion



If leaf node:

create & the select a new child node

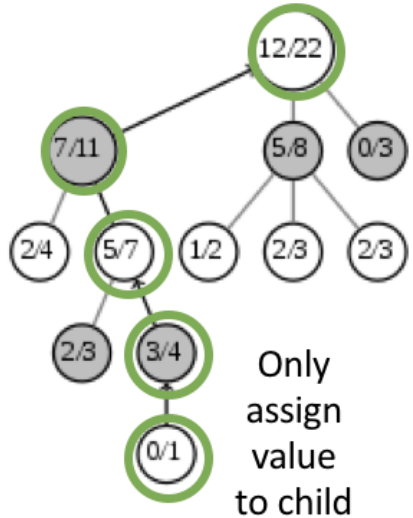
Simulation



Rollout to the end of the game

(simulated)

Backpropagation



Only assign value to child node

Backup the result through the tree to the root state

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

Monte Carlo Tree Search in AlphaGo

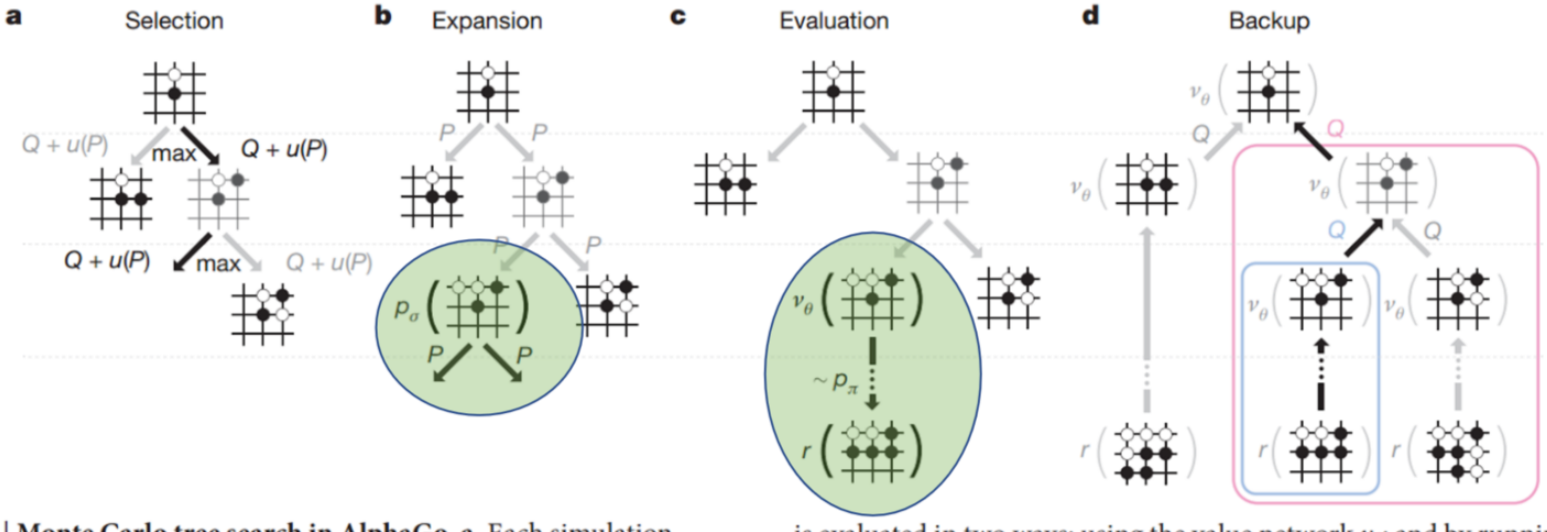
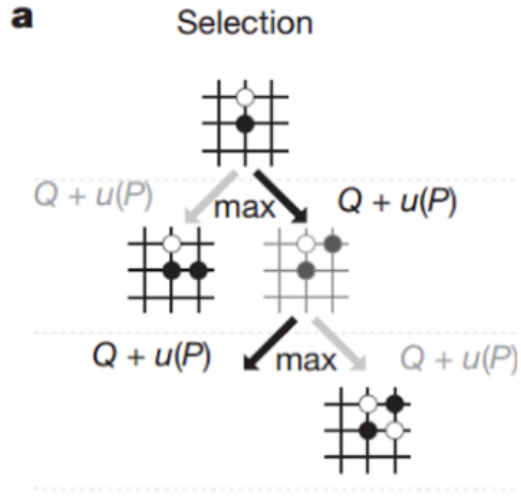


Figure 3 | Monte Carlo tree search in AlphaGo. a, Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus $u(P)$ that depends on a stored prior probability P for that edge. **b**, The leaf node may be expanded; the new node is processed once by the policy network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

Monte Carlo Tree Search in AlphaGo



Each edge (s, a) keeps track of statistics

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

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

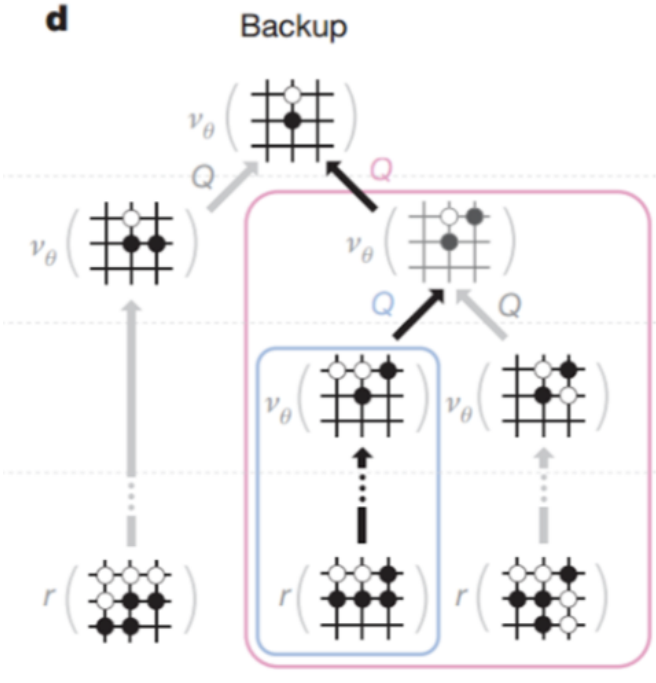
Action selected according too

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

where

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

Monte Carlo Tree Search in AlphaGo



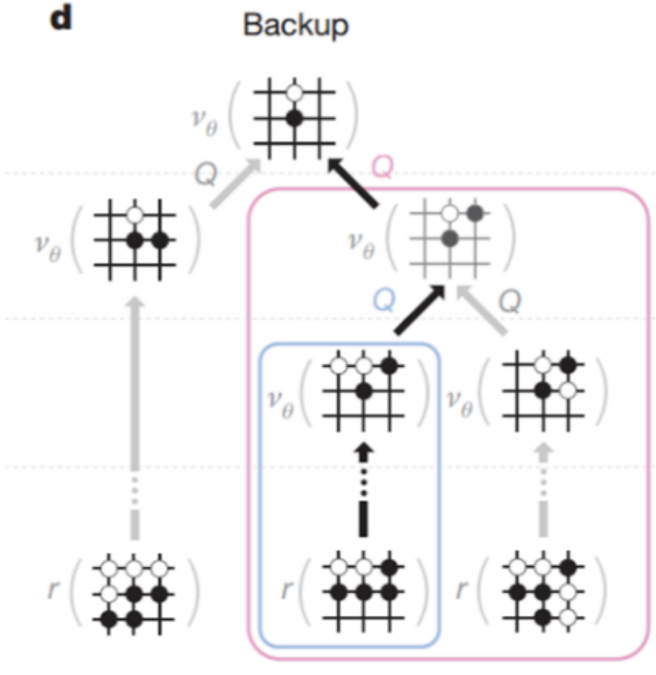
After we finish our rollout – we calculate a state value for our leaf node s_L

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

- 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



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

action value $Q(s,a)$

visit count $N(s,a)$

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

We are averaging across all visits in the simulation

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

AlphaGo Zero

Key ideas in AlphaGo Zero

Simpler

Search

Adversarial

Machine knowledge only

AlphaGo Zero Results

Training time & performance

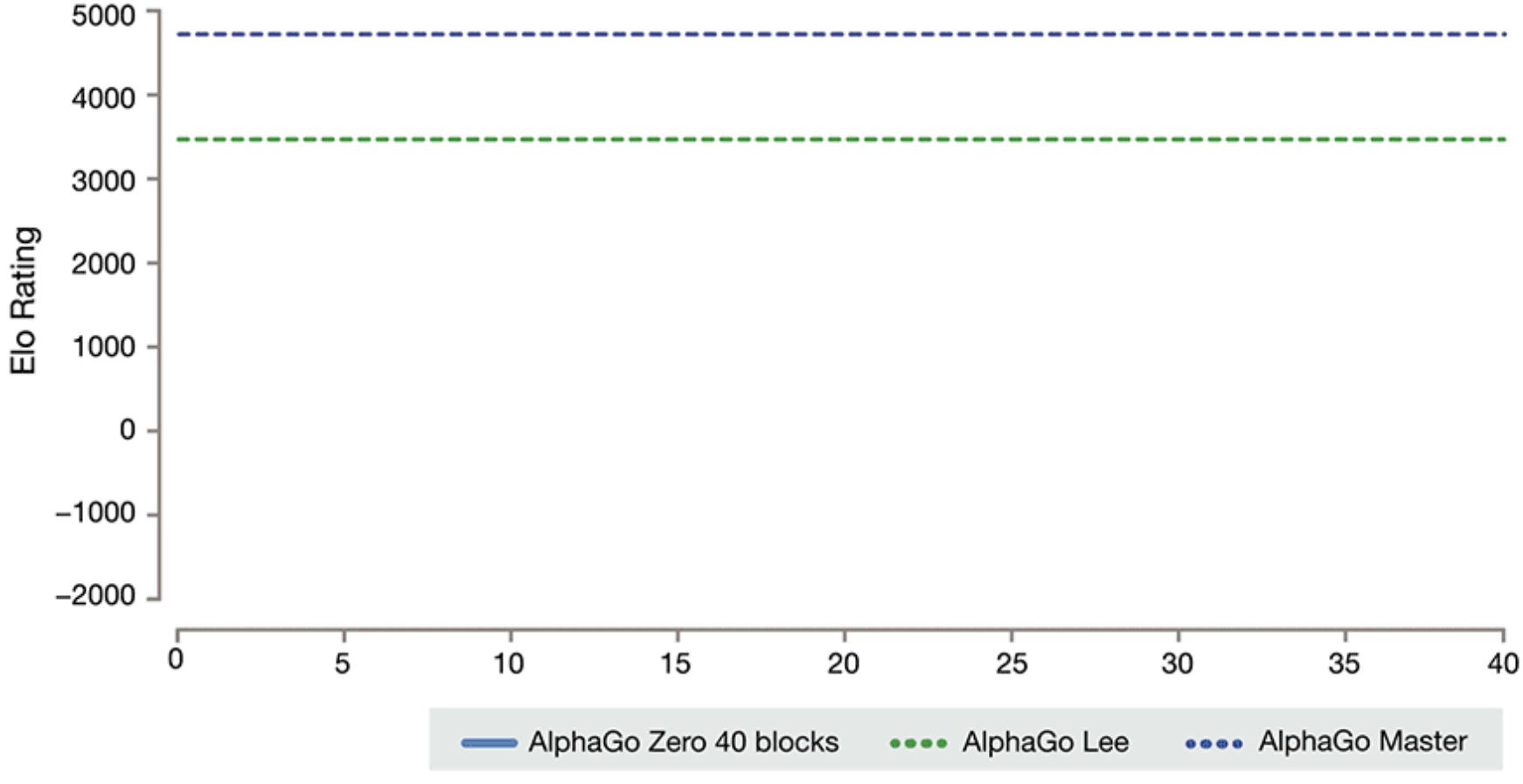
AG Lee trained over several months

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

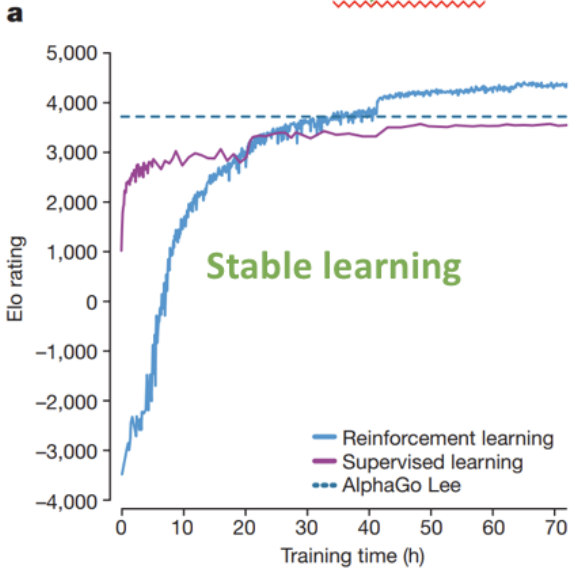


AlphaGo Zero: Learning from scratch

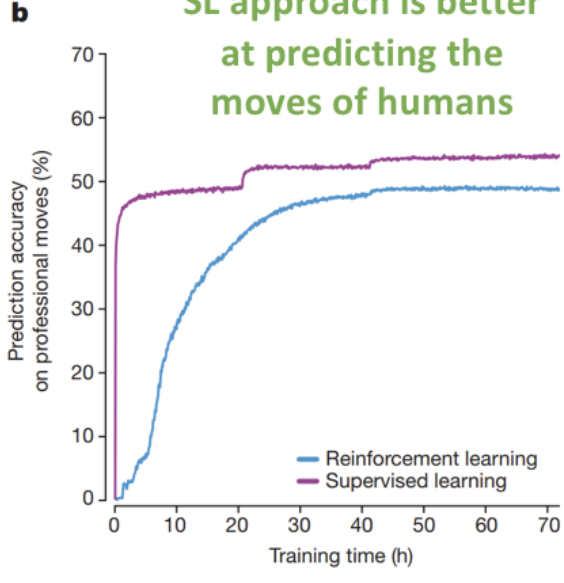
AlphaGo Zero learning curves

RL surpasses SL after around 1 day

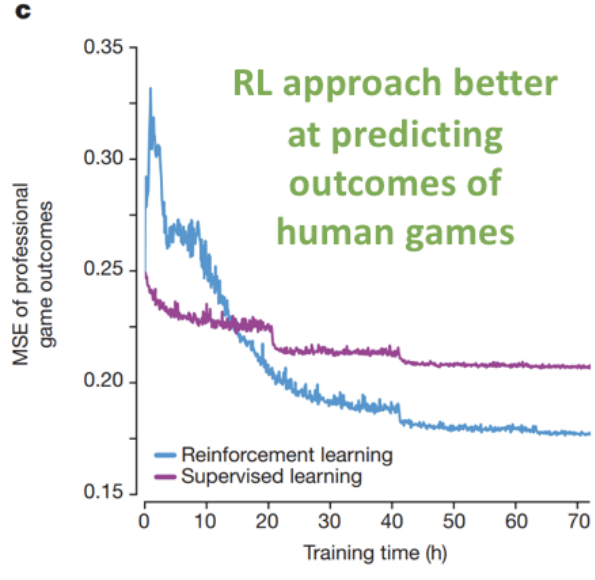
Use of SL limits AlphaGo Lee



SL approach is better at predicting the moves of humans



RL approach better at predicting outcomes of human games



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

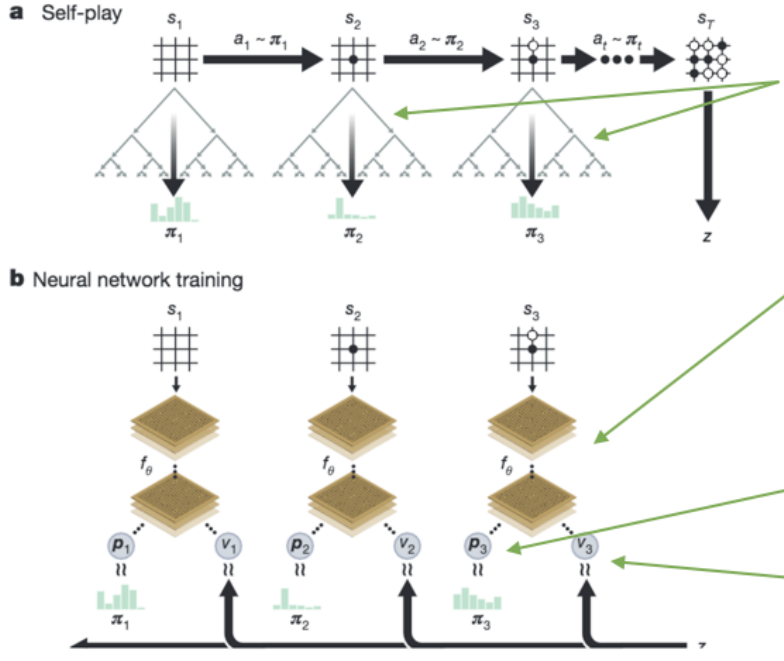


Figure 1 | Self-play reinforcement learning in AlphaGo Zero. a, The program plays a game $s_1, ..., s_T$ against itself. In each position s_t, an MCTS α_θ is executed (see Fig. 2) using the latest neural network f_θ. Moves are selected according to the search probabilities computed by the MCTS, $a_t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

Use MCTS during acting

Single neural network

two outputs:

1 – probability dist. over actions

2 – the game outcome

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

Search 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

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

[–] 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

AlphaZero

General version of AlphaGoZero that has mastered Chess and Shogi

AlphaGo, in context – Andrej Karpathy

Convenient properties of Go

fully deterministic

fully observed

discrete action space

access to perfect simulator

relatively short episodes

evaluation is clear

huge datasets of human play

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

Practical concerns

Should I use reinforcement learning for my problem?

It is a complex problem

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

evolutionary methods if you have > 10,000 parameters

Can I sample efficiently / cheaply

do you have a simulator

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 incentive behaviour

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

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 0 - 5)

Mistakes I've seen others make

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

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

Make it easier to get learning to happen (initially)

- input features, reward function design

Best practices

Be careful not to overfit these simple problems

use low capacity neural networks

Interpret & visualize learning process

state visitation, value functions

Always use multiple random seeds

Automate experiments - don't waste time watching them run!

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

Best practices

Batch size matters

Policy gradient methods – weight initialization matters determines initial state visitation (i.e. exploration)

DQN converges slowly

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

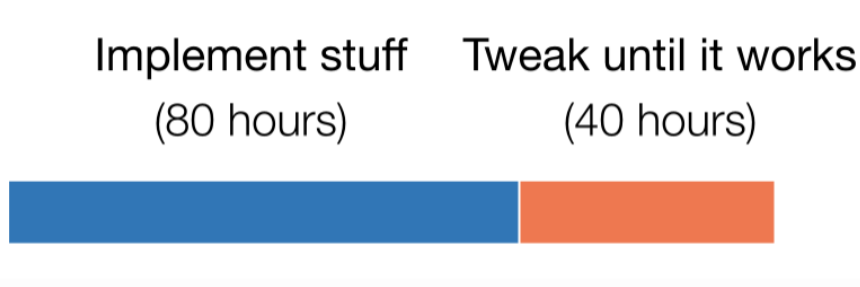
Visualize statistics

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

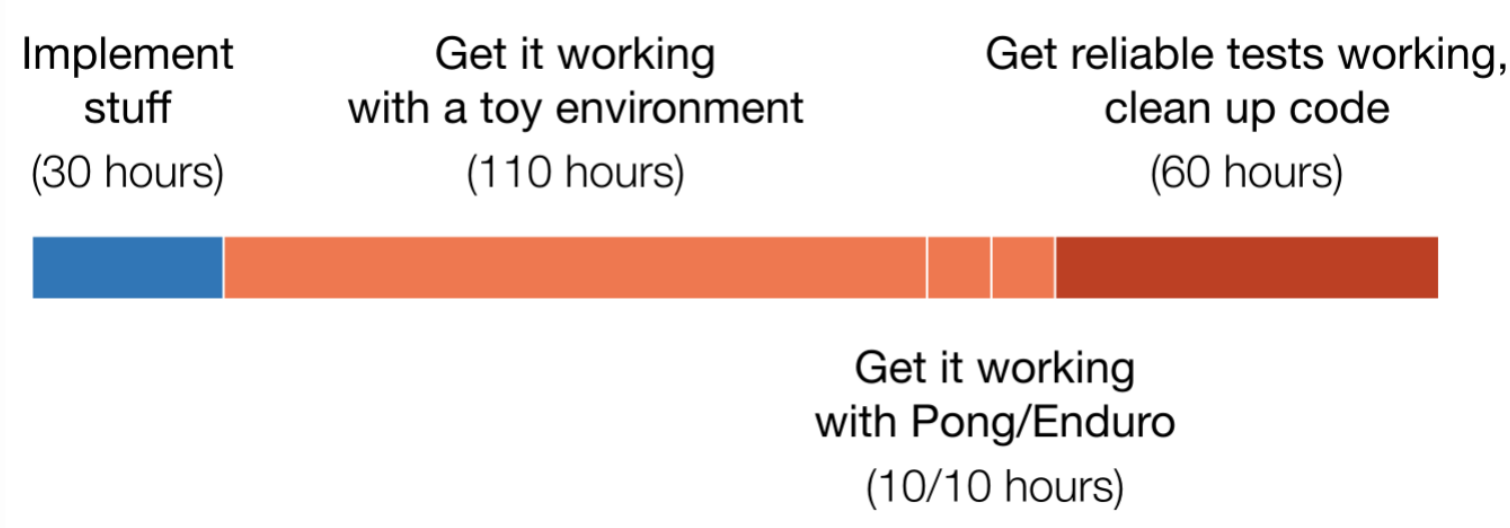
Gradient clipping is helpful - dropout & batchnorm not so much

Lessons Learned Reproducing a Deep Reinforcement Learning Paper - Amid Fish

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.

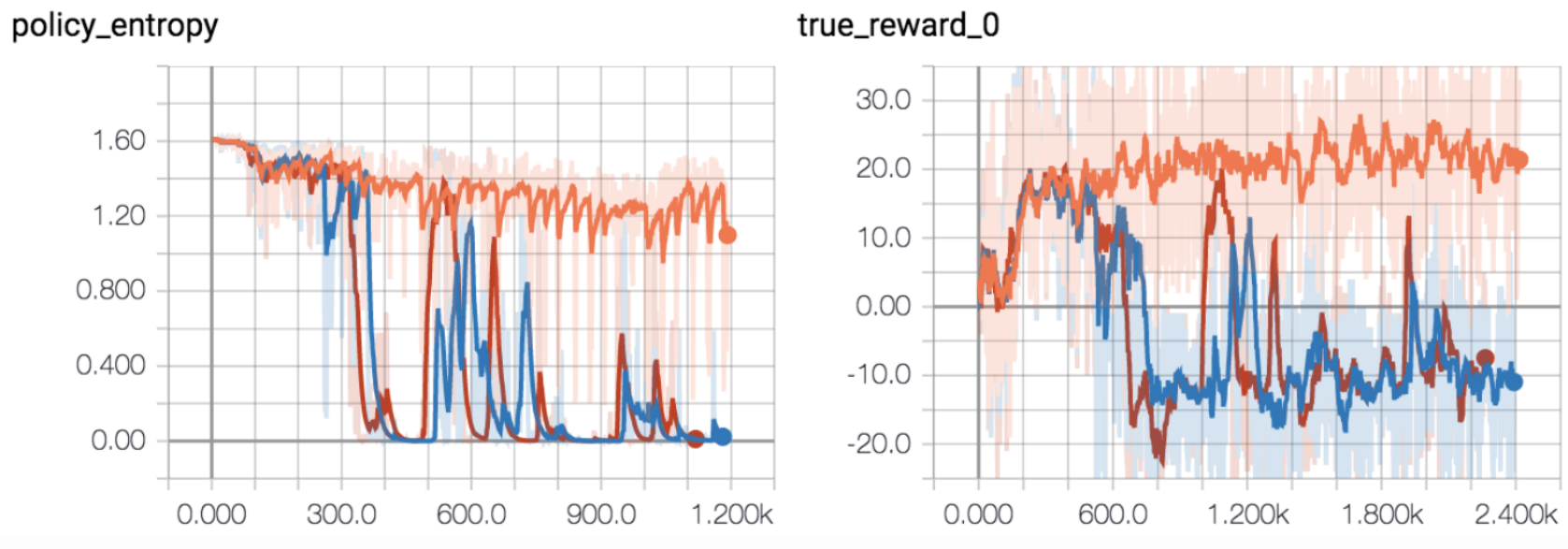


In total, the project took:

- **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.

For example, once I thought everything was basically working, I sat down to make end-to-end tests for the environments I'd been working with. But I was having trouble getting even the simplest environment I'd been working with, training a dot to move to the centre of a square, to train successfully. I went back to the FloydHub job that had originally worked and re-ran three copies. It turned out that the hyperparameters I thought were fine actually only succeeded one out of three times.



It's not uncommon for two out of three random seeds (red/blue) to fail.

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

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

switch from experimenting a lot and thinking little to **experimenting a little and thinking a lot**

reserve experiments for after you've really fleshed out the hypothesis space

Get more out of runs

Recommends keeping a detailed work log

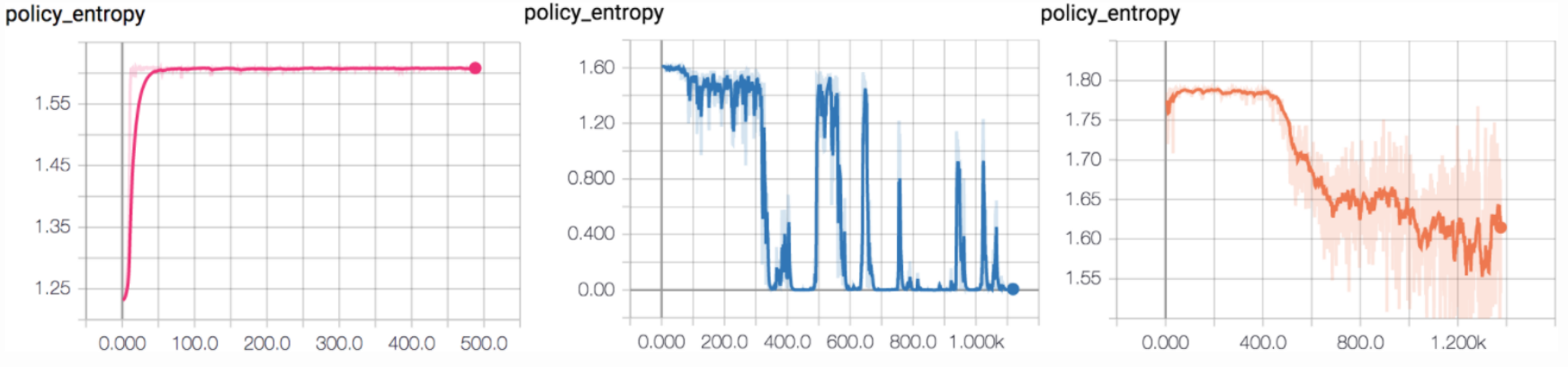
- what output am I working on now
- think out loud - what are the hypotheses, what to do next
- record of current runs with reminder about what each run is susposed to answer
- results of runs (i.e. TensorBoard)

Try to predict future failures

Get more out of runs

Log all the metrics you can

policy entropy for policy gradient methods



Examples of unhealthy and healthy policy entropy graphs. Failure mode 1 (left): convergence to constant entropy (random choice among a subset of actions). Failure mode 2 (centre): convergence to zero entropy (choosing the same action every time). Right: policy entropy from a successful Pong training run.

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)

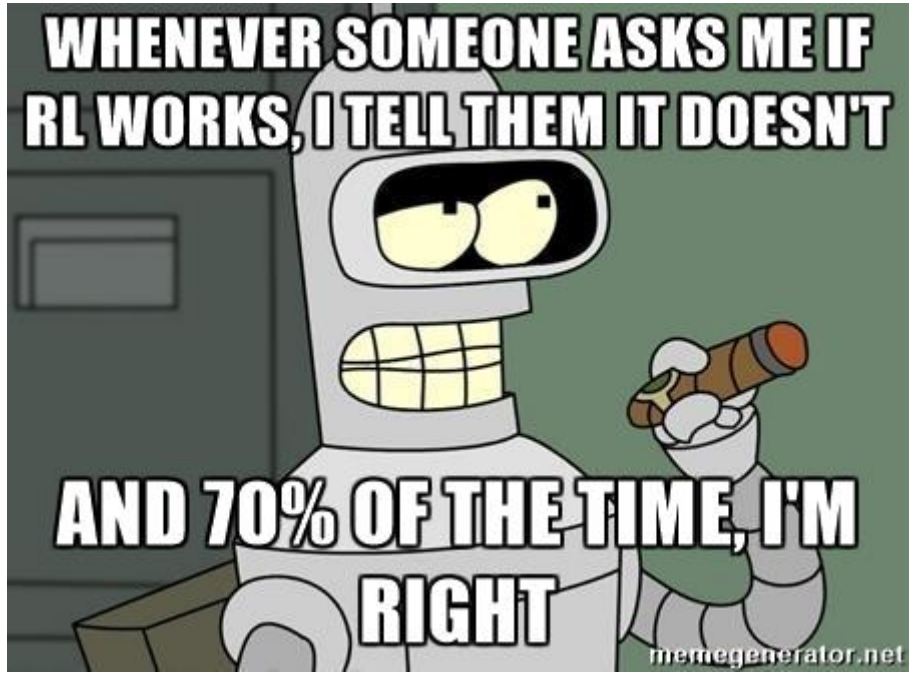
TensorFlow

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

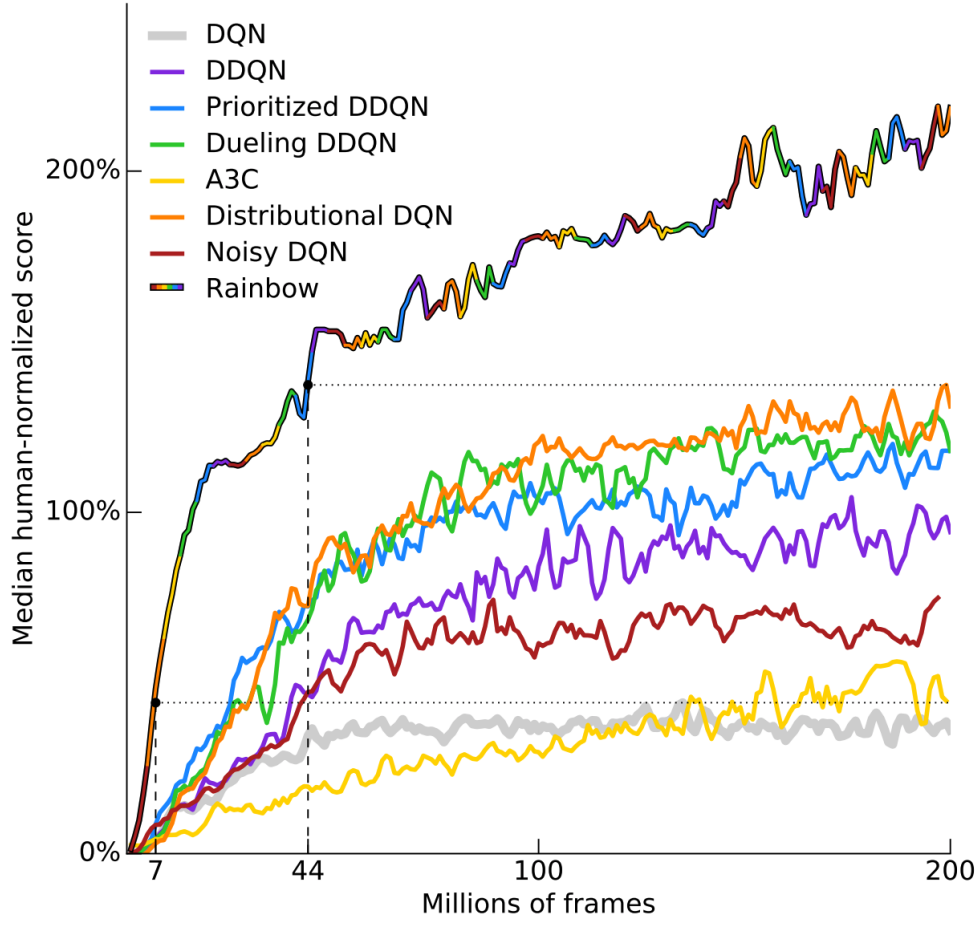
use the `allow_growth` option to avoid TF reserving memory it doesn't need

don't get addicted to TensorBoard - let your expts run!

Deep Reinforcement Learning Doesn't Work Yet - Sorta Insightful


WHENEVER SOMEONE ASKS ME IF RL WORKS, I TELL THEM IT DOESN'T
AND 70% OF THE TIME, I'M RIGHT
memegenerator.net

Modern RL is sample inefficient



Modern RL is sample inefficient

To pass the 100% median performance

Rainbow = 18 million frames = 83 hours of play

Distributional DQN = 70 million

DQN = never (even after 200 million frames!)

We can ignore sample efficiency if sampling is cheap

In the real world it can be hard or expensive to generate experience

It's not about learning time - it's about the ability to sample

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

Agent	*B.Rider*	*Breakout*	*Enduro*	*Pong*	*Q*bert*	*Seaquest*	*S.Invaders*
DQN	4092	168	470	20	1952	1705	581
-best	5184	225	661	21	4500	1740	1075
UCC	5342 (20)	175(5.63)	558(14)	19(0.3)	11574(44)	2273(23)	672(5.3)
-best	10514	351	942	21	29725	5100	1200
-greedy	5676	269	692	21	19890	2760	680
UCC-I	5388(4.6)	215(6.69)	601(11)	19(0.14)	13189(35.3)	2701(6.09)	670(4.24)
-best	10732	413	1026	21	29900	6100	910
-greedy	5702	380	741	21	20025	2995	692
UCR	2405(12)	143(6.7)	566(10.2)	19(0.3)	12755(40.7)	1024 (13.8)	441(8.1)

The generalizability of RL means that except in rare cases, domain specific algorithms work faster and better

Requirement of a reward function

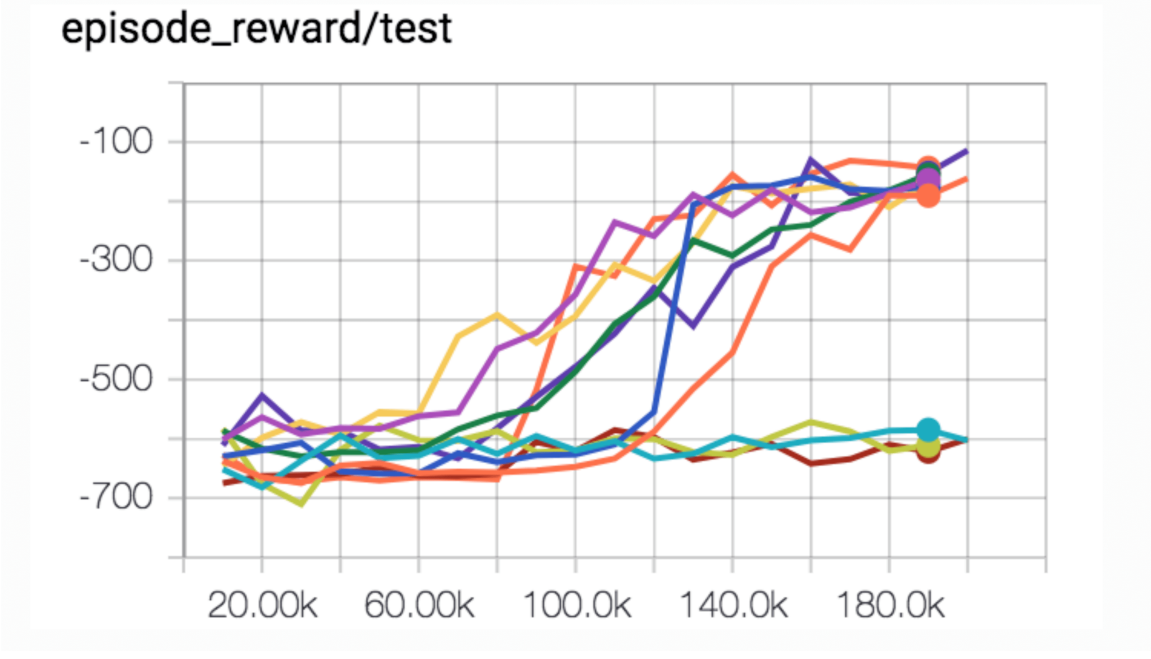
Reward function design is difficult

need to encourage behaviour

need to be learnable

Shaping rewards to help learning can change behaviour

Unstable and hard to reproduce results



Only difference is the random seed!

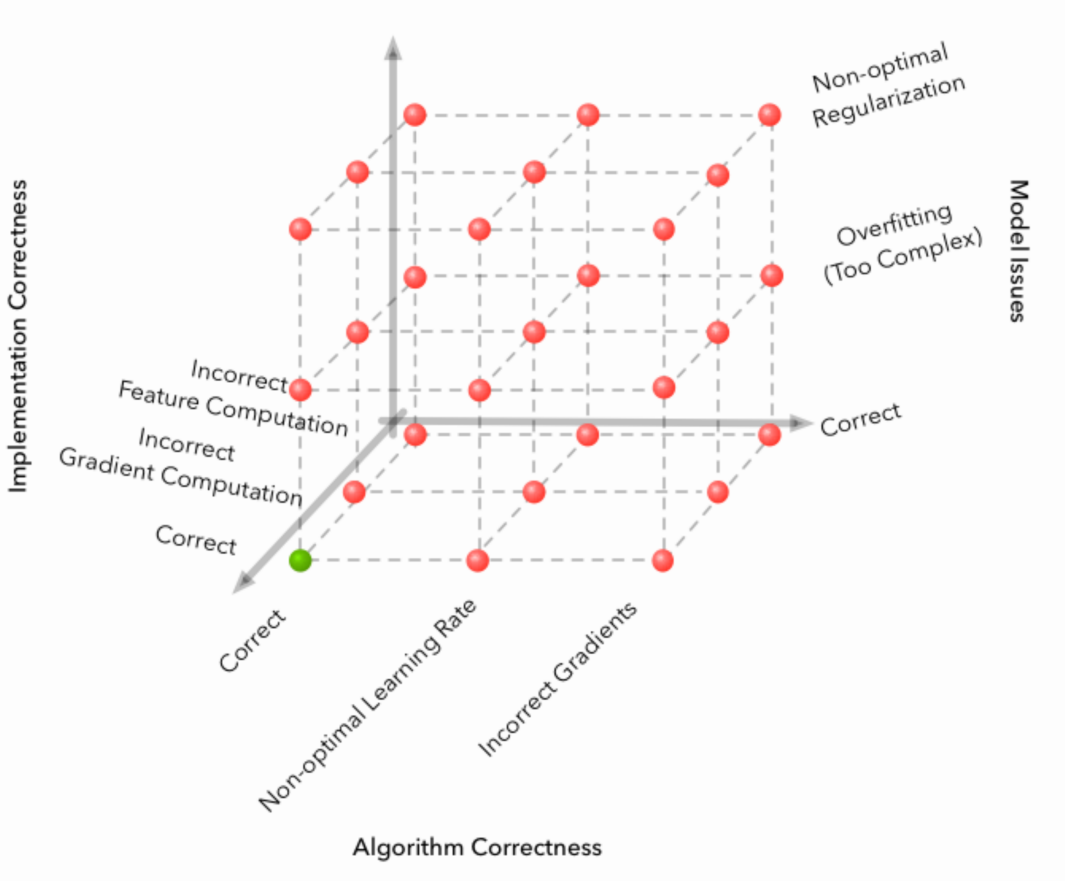
30% failure rate counts as working

Unstable and hard to reproduce results

Machine learning adds more dimensions to your space of failure cases

RL adds an additional dimension - **random change**

A sample inefficient and unstable training algorithm heavily slows down your rate of productive research


Implementation Correctness
Model Issues
Algorithm Correctness
Non-optimal
Regularization
Overfitting
(Too Complex)
Correct
Incorrect
Feature Computation
Incorrect
Gradient Computation
Correct
Correct
Non-optimal Learning Rate
Incorrect Gradients

[Supervised learning] wants to work. Even if you screw something up you'll usually get something non-random back.

RL must be forced to work. If you screw something up or don't tune something well enough you're exceedingly likely to get a policy that is even worse than random. And even if it's all well tuned you'll get a bad policy 30% of the time, just because.

Long story short your failure is more due to the difficulty of deep RL, and much less due to the difficulty of designing neural networks - Hacker News comment from Andrej Karpathy, back when he was at OpenAI

Going forward & the future

Make learning eaiser

ability to generate near unbounded amounts of experience

problem is simplified into an eaiser 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

more learning signal - hallucinating rewards, auxillary tasks, model learning

model learning fixes a bunch of problems - difficulty is learning one

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.*

Many things need to go right for RL to work - success stories are the exception, not the rule