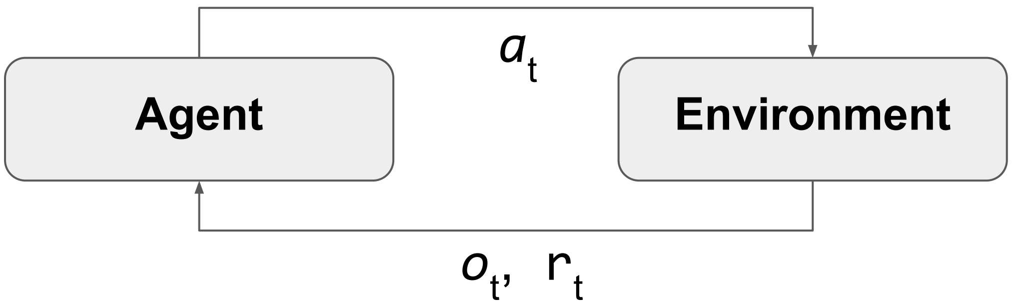
Deep Reinforcement Learning

Recap

The Reinforcement Learning Problem

A learning system — the **agent** — must learn to act in the universe it's embedded in — the **environment** — to maximize a scalar feedback signal — the **reward**.



Not a solution nor an algorithm: it's a problem specification.

Policies

The agent's behaviour is defined by a **policy** $\pi(A_t|S_t)$, i.e. by a prob. distribution over actions, conditioned on the current **state**

The agent's **objective** is to find an **optimal** policy $\pi^*(A_t|S_t)$ that maximises the **return**, i.e. the sum of rewards collected from any state S_t onwards:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

Where the **discount** γ modulates the relative weight of immediate vs distant rewards.

Values

If the environment or the agent is **stochastic**, the return is a random variable, thus the objective is typically formulated as maximizing **expected returns**, or **values**.

$$V(s) = E[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s] \quad \longrightarrow \quad \text{State Value}$$

$$Q(s, a) = E[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} | S_t = s, A_t = a] \quad \longrightarrow \quad \text{State-Action Value}$$

The expectations are over both the environment and agent stochasticity.

Models

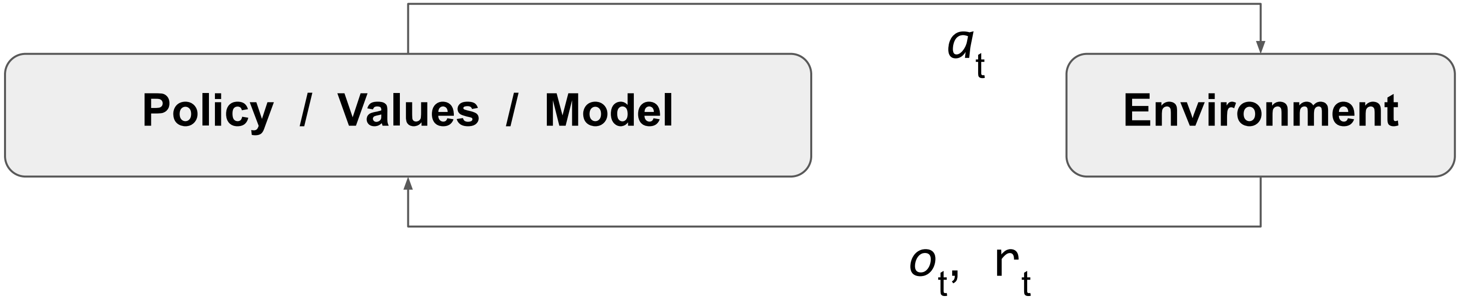
In RL we often assume that we do not know the environment dynamics:

- values and policies are learned **model-free** from interaction

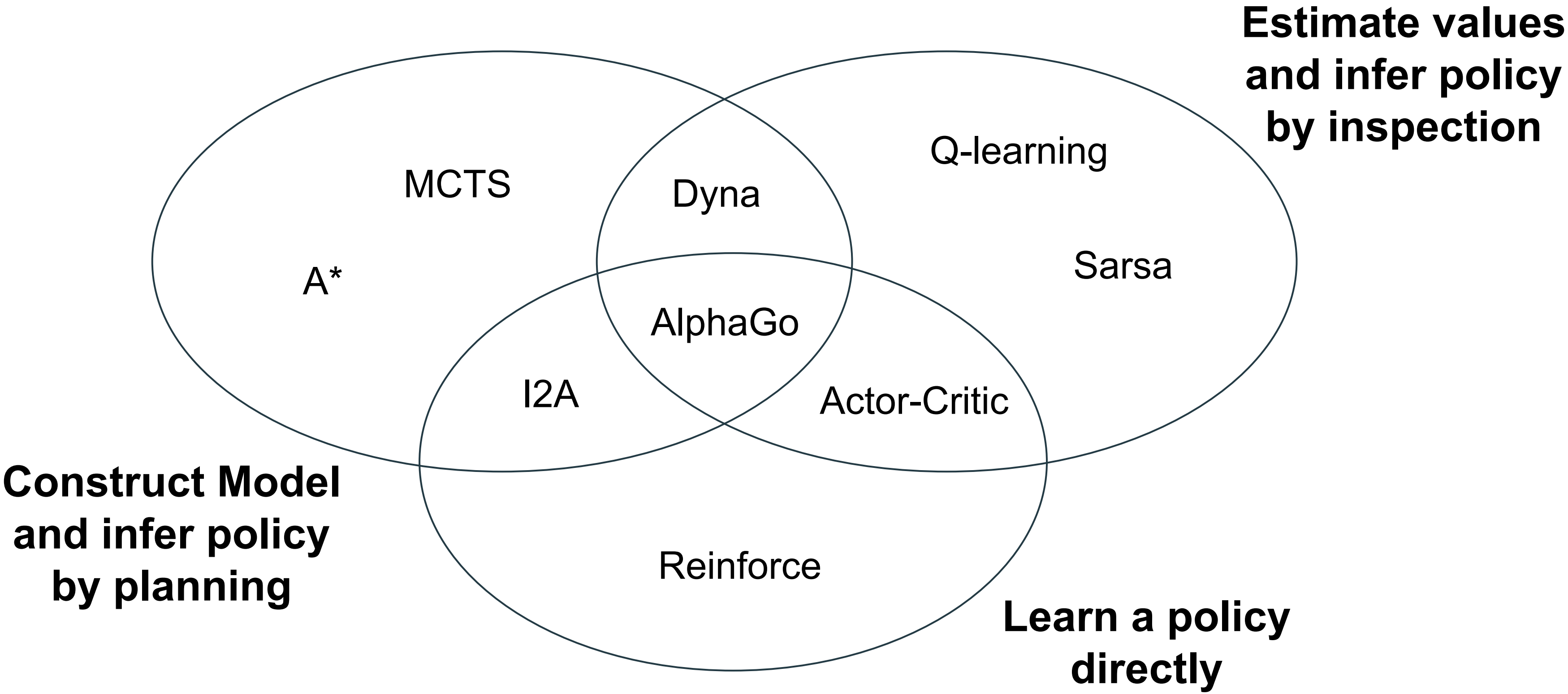
Sometimes, however, we may use the interaction data to learn a **model** of the environment, inferring, from data, how actions affect environment state and rewards.

RL Solutions: The Big Picture

- Learn **policy** directly - **execute** it
- Estimate **values** of each action - infer policy by **inspection**
- Construct a **model** - infer policy by **planning**



RL Solutions: The Big Picture



Deep Reinforcement Learning

Policy $\pi : S_t \longrightarrow p(A_t)$

Value $v : S_t \longrightarrow V(S_t)$

Model $m : S_t, A_t \longrightarrow S_{t+1}, R_{t+1}$

Ultimately these are just **functions**,

and we need a flexible way of **representing** and **learning** them

Deep Reinforcement Learning

Policy $\pi : S_t \longrightarrow p(A_t)$
Value $v : S_t \longrightarrow V(S_t)$
Model $m : S_t, A_t \longrightarrow S_{t+1}, R_{t+1}$

Deep Learning

Ultimately these are just **functions**,
and we need a flexible way of **representing** and **learning** them

Why Deep Learning?

Deep learning is not the only answer, but it's a pretty convenient one

- **Table Look Up**?
  $\longrightarrow$ Learn about each state separately
  $\longrightarrow$ No **generalization**, does not scale to large state spaces

- **Linear**?
  $\longrightarrow$ $y_t = \mathbf{w}^T S_t + b$
  $\longrightarrow$ Can work very well if you have a rich state representation S_t
  $\longrightarrow$ Hard-coding a suitable **rich state representation** S_t is difficult!

Stable and efficient Deep Reinforcement Learning

Ref: Watkins et al.

Q-learning

Optimal policies and their values satisfy the **Bellman's optimality equation**:

$$Q^*(s,a) = E[R_{t+1} + \gamma\ max_{a'} Q^*(S_{t+1}, a')|S_t = s, A_t = a]$$

Given estimates of the values Q, we can **sample** this equation and turn it into **incremental updates** to our estimates of Q*:

$$Q^*(S_t, A_t) \leftarrow Q^*(S_t, A_t) + \alpha\ (\underbrace{R_t + \gamma\ max_{a'} Q^*(S_{t+1}, A_{t+1})}_{G_t} - Q^*(S_t, A_t))$$

Deep Q-learning

Previously we assumed a separate value estimate Q^* for each state action pairs *s, a*.

Instead, we can use a **neural network** to learn a **mapping** between state-action pairs *(s, a)* and their (optimal) values, and exploit **generalization**

The Q-learning updates reduces to **stochastic semi-gradient descent update** on:

$$Loss(\theta) = (\underbrace{R_t + \gamma\, max_{a'} Q^*_\theta(S_{t+1}, A_{t+1})}_{\textit{Target}\text{: do not back-prop gradients here}} - Q^*_\theta(S_t, A_t))^2$$

Ref: Lin et al.

Experience Replay

Stochastic gradient descent assumes that the updates we apply are **i.i.d.**

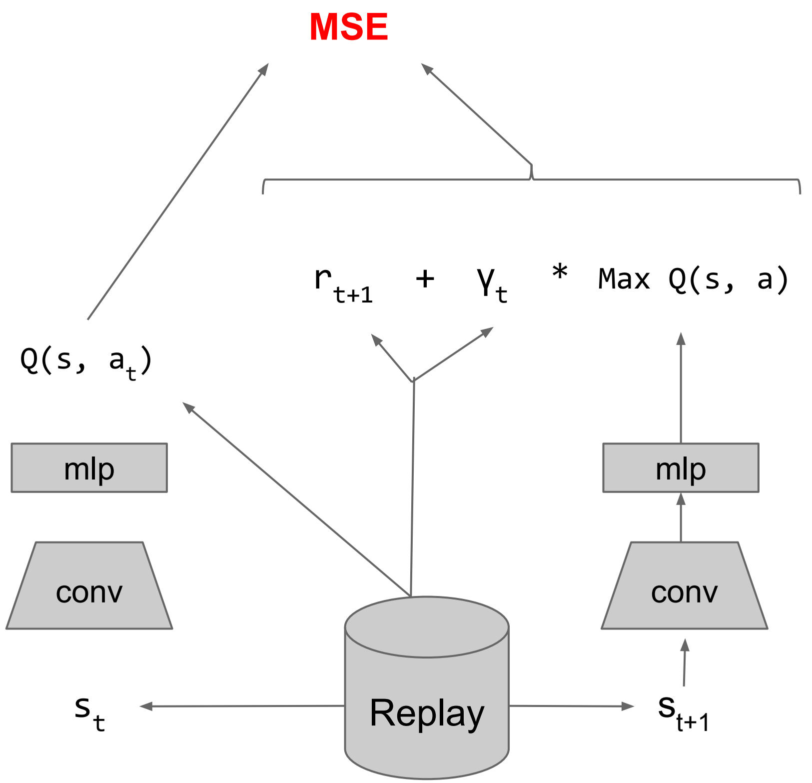
In Reinforcement Learning this is largely false

- e.g. values of states in the same episode of a game are **strongly correlated**.

Compute the Q-learning loss not on the most recent transition in the environment but instead on an old one sampled from a **large memory buffer**

Ref: Mnih et al.

DQN



video

Ref: Schaul et al.

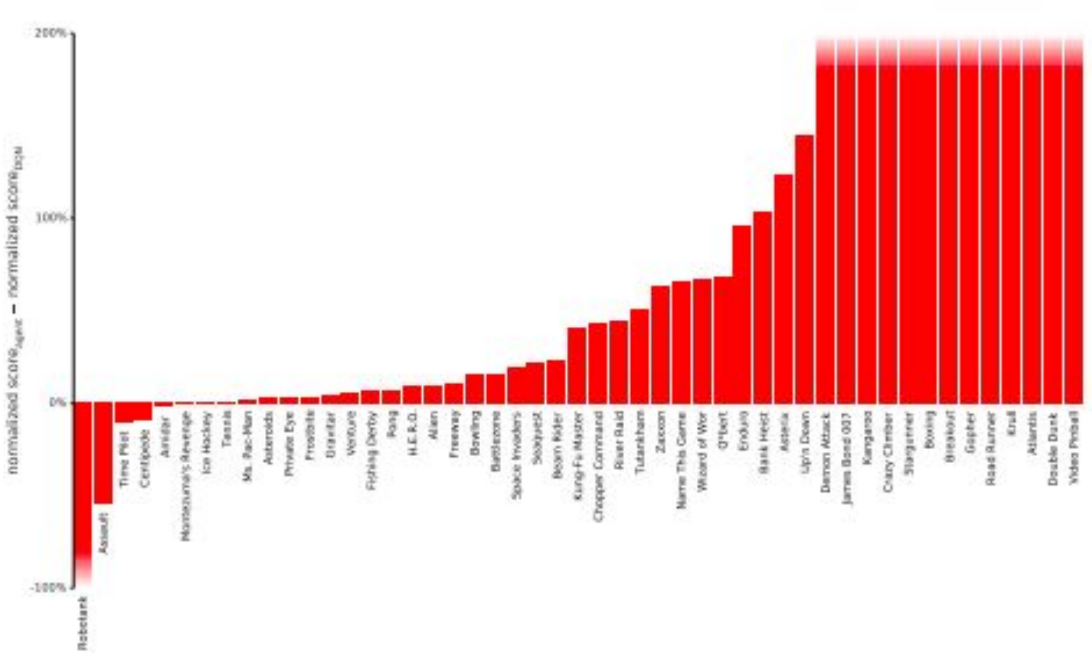
Prioritized Experience Replay

An experience replay is also useful because it allows:

- to use data multiple times to update our network
- to prioritize novel or “surprising” data from which there is much to learn over less interesting data that we already fully understand

As a result it can massively improve **data-efficiency**

Prioritized Replay



$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}$$

Proportional

p_i = loss

Rank-based

p_i = rank_i

Ref: van Hasselt et al.

Overestimation Bias

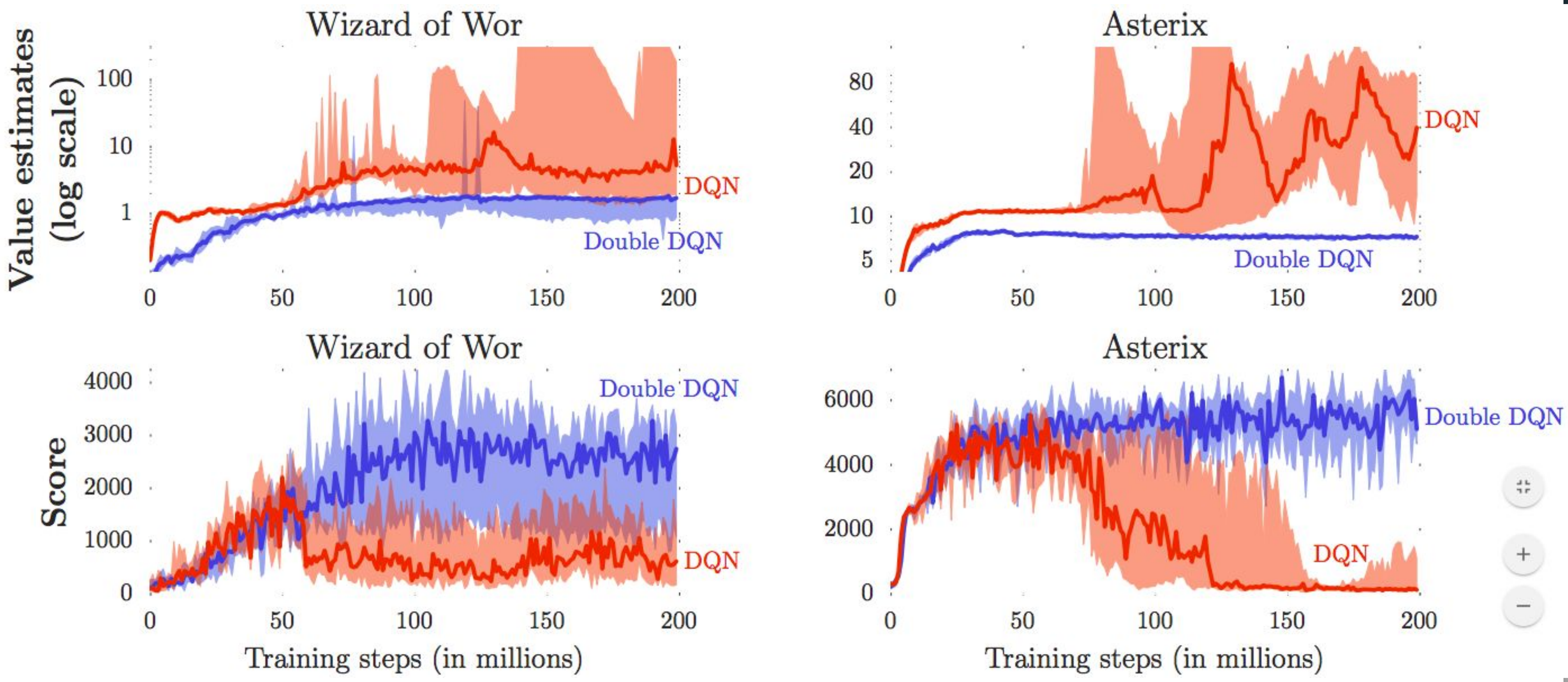
At the heart of the Q-learning operator there is the process of selecting the max value among all actions to construct the target.

$$Loss(\theta) = (R_t + \gamma\ max_{a'} Q^*_\theta(S_{t+1}, A_{t+1}) - Q^*_\theta(S_t, A_t))^2$$

Danger! If the value estimates are noisy I will tend to select the noisiest estimate, And over time I will end up **overestimating** the action values

$Q_\theta(s, argmax_{a'}\ Q_\eta(s, a')))$ ← **Double Q-learning**

Overestimation Bias



Multi-step deep Q-learning

Using our own estimates Q_θ to construct targets $R_t + \gamma \max_a Q_\theta(S_{t+1}, A_{t+1})$ is called **bootstrapping**.

We don't however need to bootstrap necessarily after a single step.

We can construct **multi-step targets**:

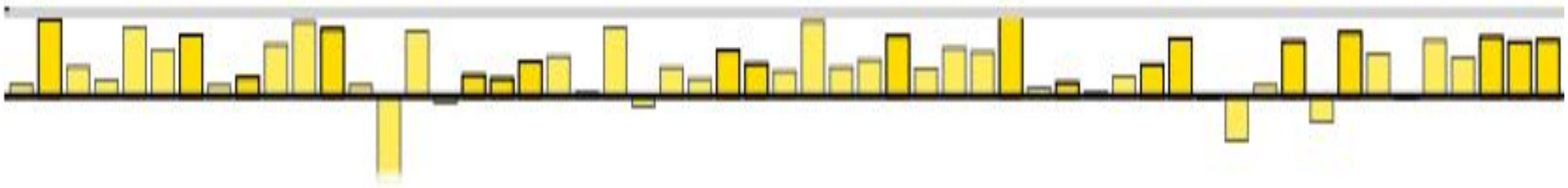
$$R_t + \gamma R_{t+1} + \ldots + \gamma^{K-1} R_{t+K-1} + \gamma^K \max_a Q_\theta(S_{t+1}, A_{t+1})$$

Ref: Hessel et al.

Multi-step deep Q-learning

Multi-step updates propagate information about observed rewards faster through state space, and by choosing appropriately the bootstrap length we can trade-off **bias** and **variance**

Improvements of K=3 over K=1 in a variant of DQN on a collection of Atari games



The Deadly Triad

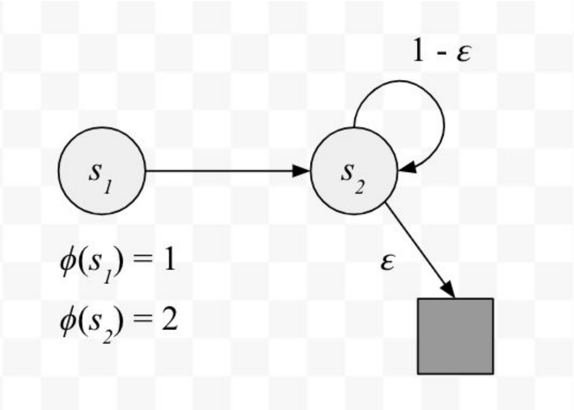
1. **Off-Policy**
    a. Estimate values of a policy on data generated from a different policy
2. **Bootstrapping**
    a. Using estimates to compute the targets for updating those same estimates
3. **Function Approximation**
    a. E.g. using neural networks

Ref: Tsitsiklis and Van Roy

The Deadly Triad

Sutton and Barto refer to the combination of these three features as the **Deadly Triad**, because can result in **divergence**, with the parameters of the network blowing up to infinity

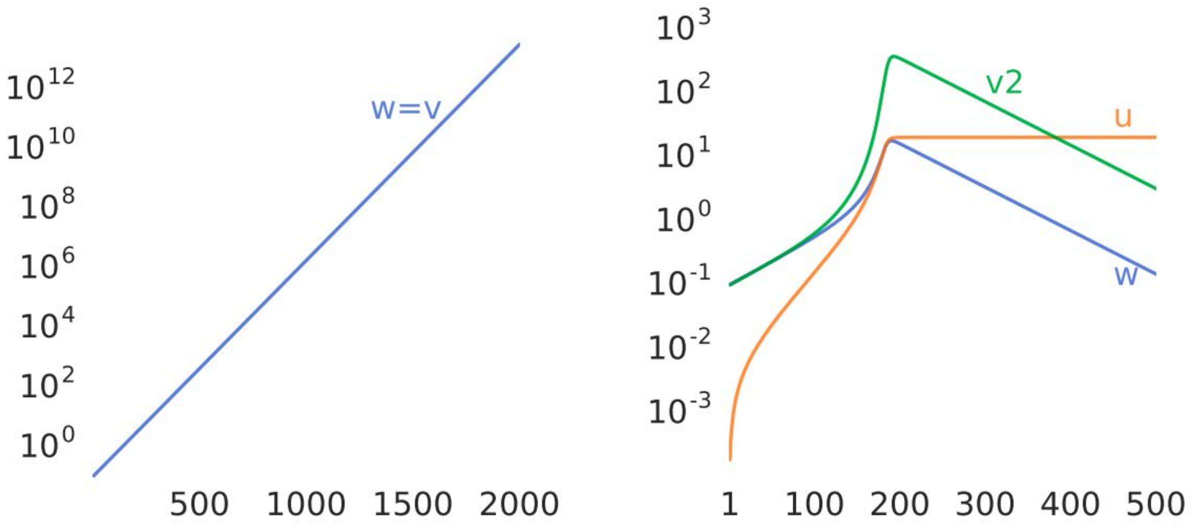
$$\left(R_t + \gamma v_\theta(S_{t+1}) - v_\theta(S_t)\right)^2$$



If I update **off-policy,** for instance because I use an experience replay, I may update **v(s$_1$) more often than v(s$_2$)** making **w** blow to infinity instead of converging to the optimal value of 0.

Ref: van Hasselt et al.

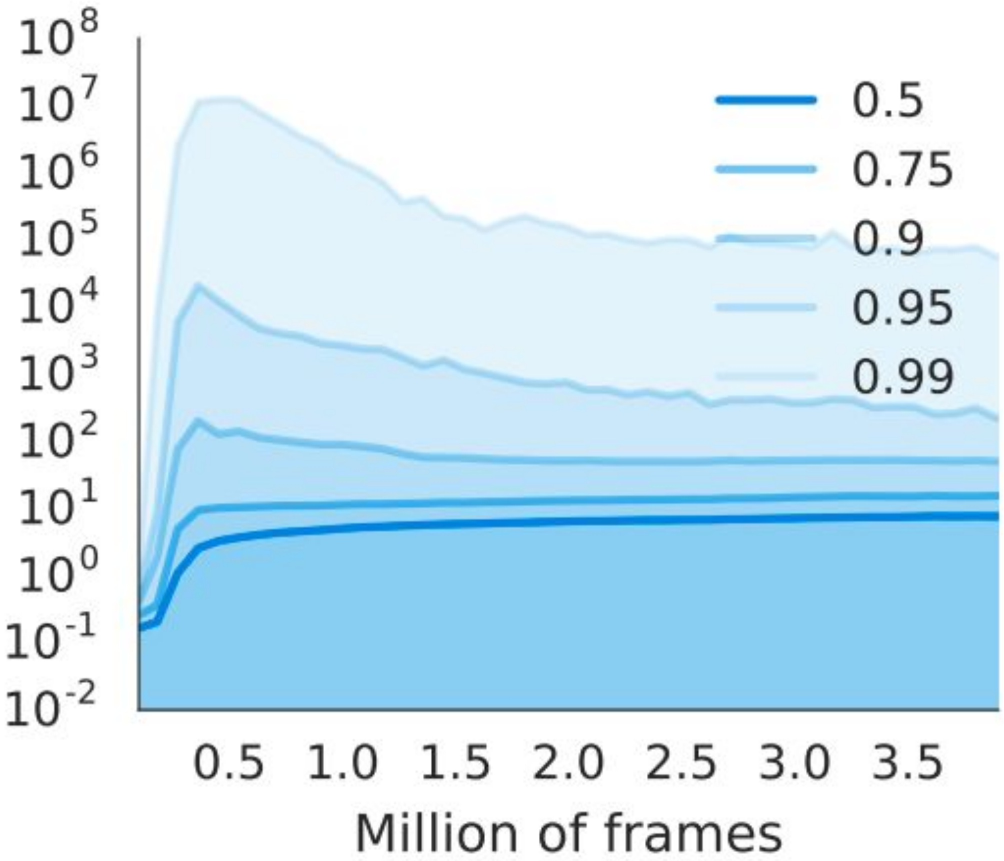
Deep RL and the Deadly Triad



(b) $v(s) = w\phi(s)$ diverges. (c) $v(s) = w(\phi(s) + u)$ converges.

Soft-divergence

This form of **soft-divergence** Where values grow to unreasonably high values but then converge back to sensible estimates, is more common in practice than actual divergence.



Target Networks

Target Networks are a common remedy to this type of **inappropriate generalization**

We can keep **two copies** of the neural network used to estimate Q values:

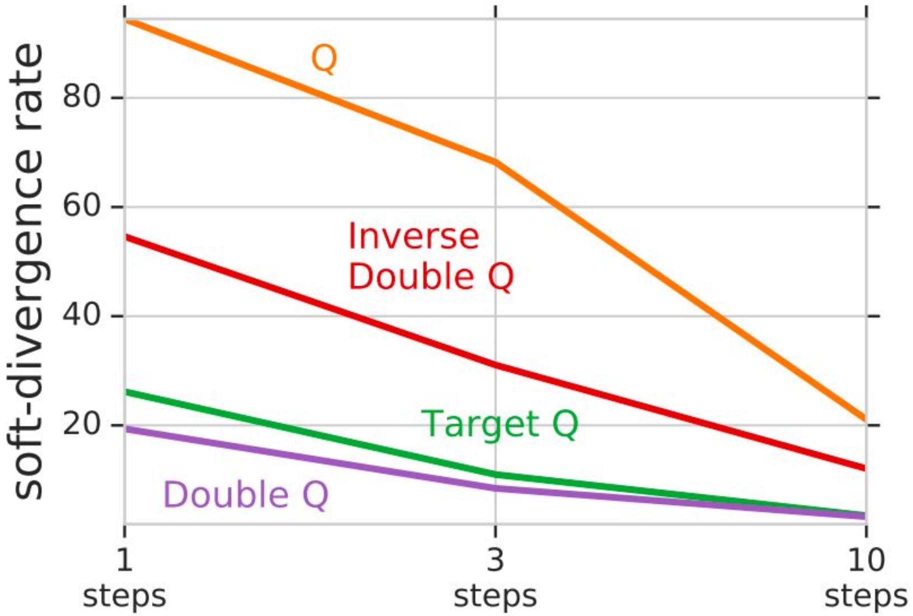
- The parameters of one are updated at each step,
- The parameters of the other are a slow copy of the first one.

This **reduces** the opportunity for **feedback loops**.

Ref: van Hasselt et al.

Multi-step returns and divergence

Alternatively we can also use Multi-step returns to reduce the reliance of our updates on bootstrapping:



(a) Multi-step returns

DL-aware RL and RL-aware DL

It's important to understand how reinforcement learning algorithm interact with your favourite form of function approximation

1. Experience Replay
2. Target networks

$\rightarrow$ **DL-aware reinforcement learning**

Conversely, rather than using neural nets developed for SL, we want to develop architecture with the right inductive biases for RL $\rightarrow$ **RL aware Deep Learning**

Ref: Wang et al.

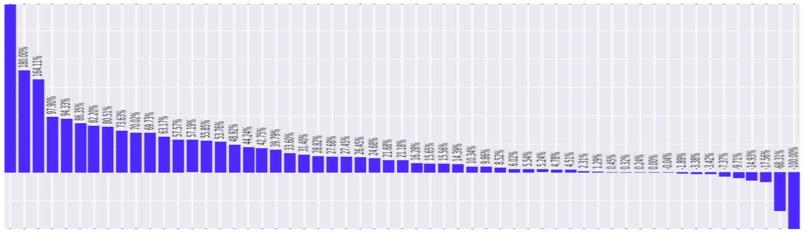
Dueling Networks

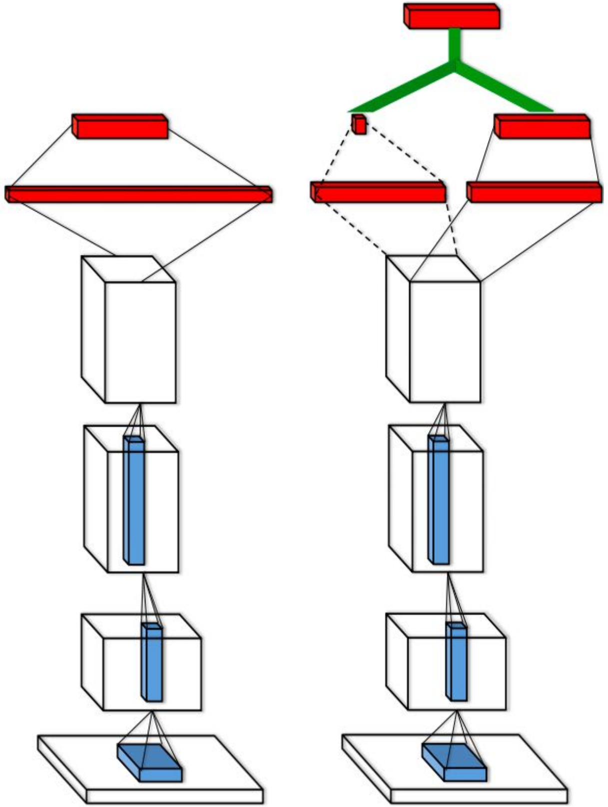
Value-Advantage decomposition of Q-values

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

Dueling Network Architecture

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





video

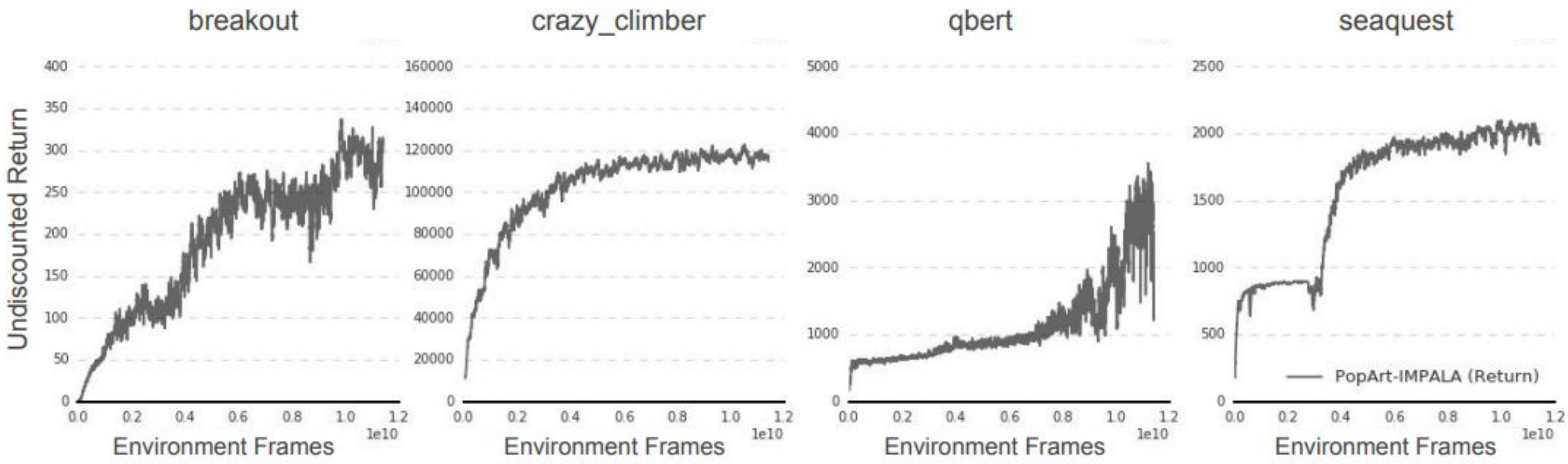
Reward Scaling

An RL task is define by its rewards

- Go: win (1), loss (0)
- Atari: change in score (+-1 in Pong +=1000s in MsPacman)

The magnitude of updates scales directly with the magnitude/frequency of rewards,

And we cannot just normalize the loss because it's **non-stationary**


breakout
crazy_climber
qbert
seaquest
Undiscounted Return
Environment Frames
1e10
PopArt-IMPALA (Return)

Ref: van Hasselt et al., Hessel et al.

PopArt

1. **Adatively Rescale Targets** (ART): rescale predictions and gradient updates according to an adaptive normalization, updated according to its own objective

2. **Preserve Outputs Precisely** (POP): after each update of the normalization statistics, apply an inverse update to the parameters of the network to avoid invalidating the unnormalized predictions (needed for instance to bootstrap)

Art

$$V(s) = \mu + \sigma * U^n(s)$$

$$\mu = (1 - \beta)\,\mu + \beta\, G$$
$$v = (1 - \beta)\, v + \beta\, G^2$$
$$\sigma = v - \mu^2$$

Adaptively Rescale Targets

$$U^n(s) = W^T x + b$$

$$\sigma \rightarrow \sigma', \quad \mu \rightarrow \mu'$$

$$W' = \sigma/\sigma' * W$$
$$b = (\sigma * b + \mu - \mu')\,/\sigma'$$

Preserve Output Precisely

Pop

$$V(s) = \mu + \sigma * U^n(s)$$

$$\mu = (1 - \beta)\,\mu + \beta\, G_{vtrace}$$
$$v = (1 - \beta)\, v + \beta\, G^2_{vtrace}$$
$$\sigma = v - \mu^2$$

Adaptively Rescale Targets

$$U^n(s) = W^T x + b$$

$$\sigma \rightarrow \sigma', \quad \mu \rightarrow \mu'$$

$$W' = \sigma/\sigma' * W$$
$$b = (\sigma * b + \mu - \mu')\,/\sigma'$$

Preserve Output Precisely

Normalized value learning

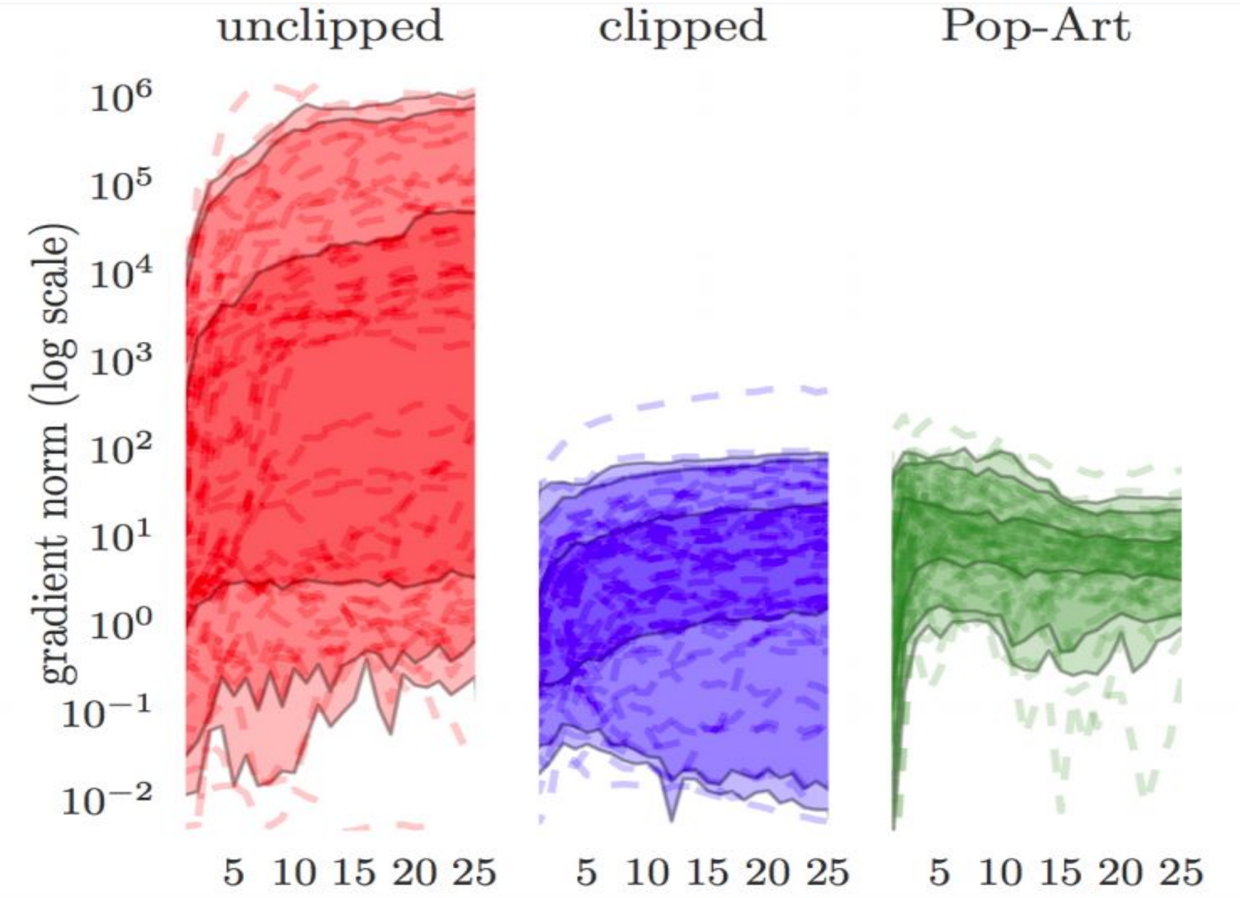
We can rewrite our value learning update

$$\left(R_t + \gamma v_\theta(S_{t+1}) - v_\theta(S_t)\right)^2$$

in normalized space, by:

- using the unnormalized value V to bootstrap
- then normalizing the target before computing the loss for U

$$\left([R_t + \gamma V_\theta(S_{t+1}) - \mu]/\sigma - U_\theta(S_t)\right)^2$$



unclipped
clipped
Pop-Art
gradient norm (log scale)
10^6
10^5
10^4
10^3
10^2
10^1
10^0
10^−1
10^−2
5 10 15 20 25
5 10 15 20 25
5 10 15 20 25

video

Exploration

At the heart of RL there is the trade-off between

- **Exploitation**: acting according to what we believe to be optimal
- **Exploration**: purposefully selecting alternative courses of actions in order to gather more information (that might lead us to revise our current beliefs)

Dithering

Many popular RL agent only explore through dithering around the current policy

- ε-greedy
- entropy regularization

In order to scale effectively to very complex domains and improve our data-efficiency we need better then this

Ref: Fortunato et al.

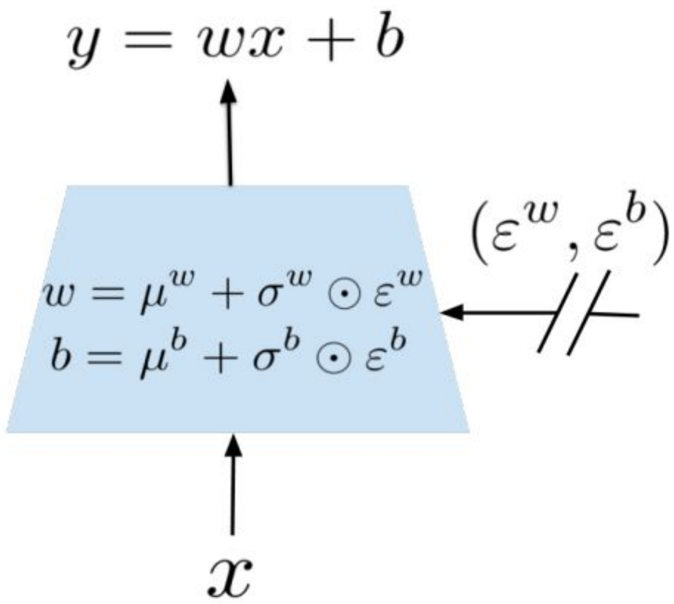
Noisy Networks

Inject additional noisy inputs:

- The net can learn to ignore noise over time

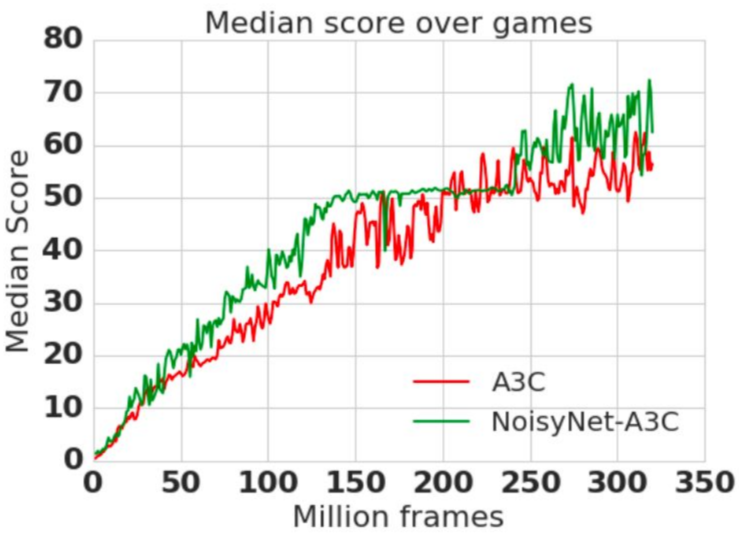
Which naturally results into:

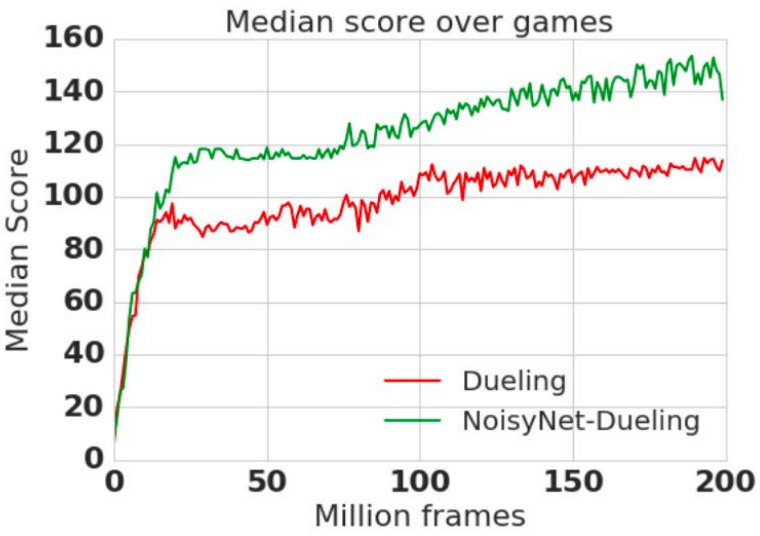
- Annealing of the amount of exploration
- The network will be more deterministic in states that have been seen many times
- The values/policy will have higher variance in novel / surprising states



Noisy Networks

The same principle can be used across different learning algorithms:





Planning and Model-based RL

Planning is the process of using additional **compute**, but not additional environment **data** to improve the agent's policy or predictions.

- Experience replay is also a form of planning!

Alternatively, we can plan by learning a **parametric model** of the environment.

Why is learning models hard?

Learning models from raw observations in complex environments is hard because:

- Observations are **high dimensional**
- Often most of the pixels in an observation are **irrelevant**

How do we make the model **focus** on what we care about for planning?


001
2
1

Planning with inaccurate models

1. But **don’t trust** fully the model

2. Learn a fully **abstract** model that is **trained** to be suitable **for planning**

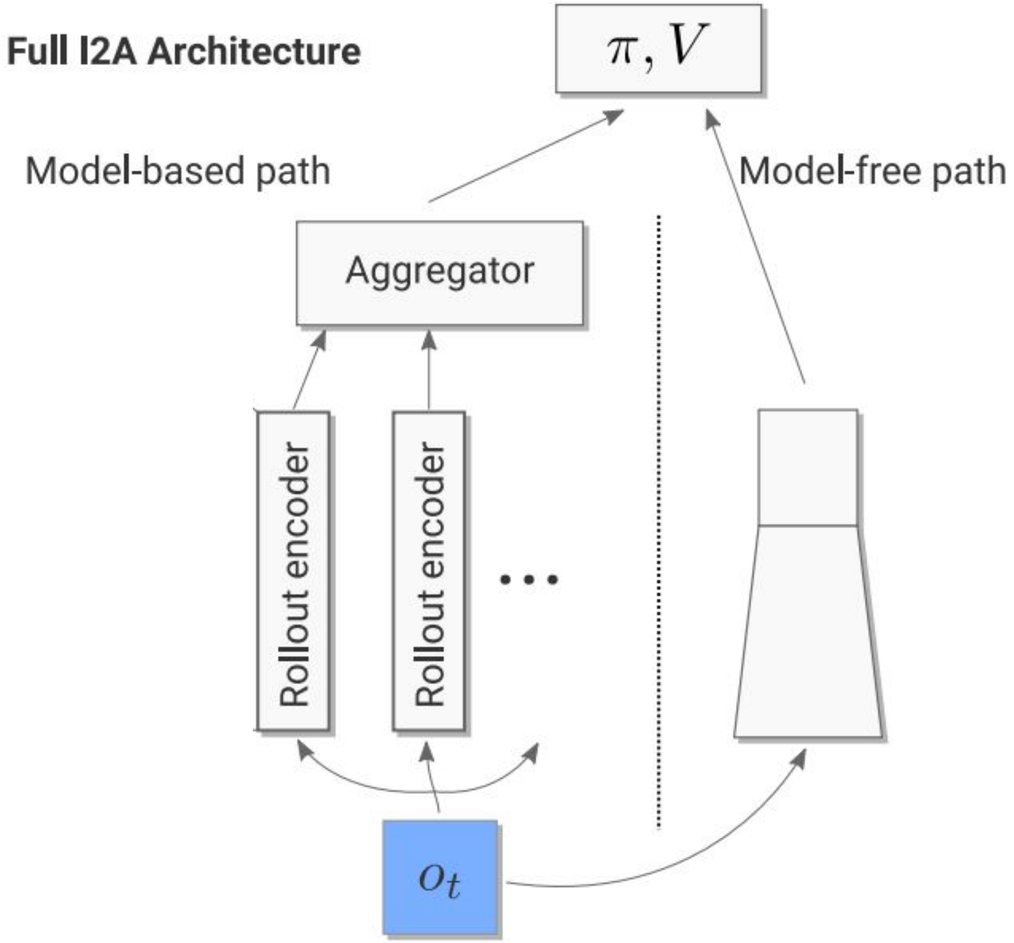
Planning with inaccurate models

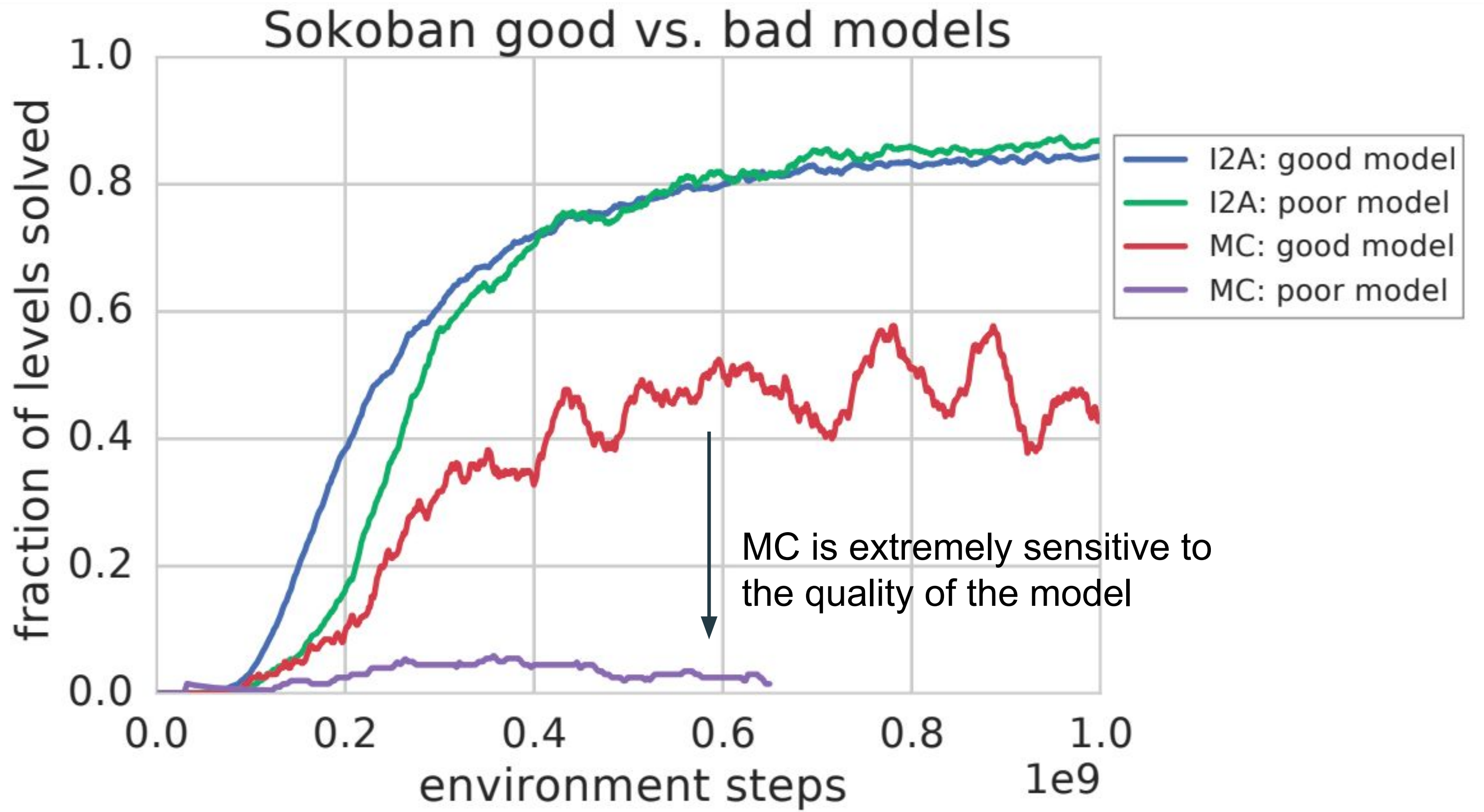
1. But **don't trust** fully the model

2. Learn a fully **abstract** model that is **trained** to be suitable **for planning**

I2A

Ref: Weber et al.




Sokoban good vs. bad models
fraction of levels solved
1.0
0.8
0.6
0.4
0.2
0.0
0.0
0.2
0.4
0.6
0.8
1.0
1e9
environment steps
I2A: good model
I2A: poor model
MC: good model
MC: poor model
MC is extremely sensitive to the quality of the model

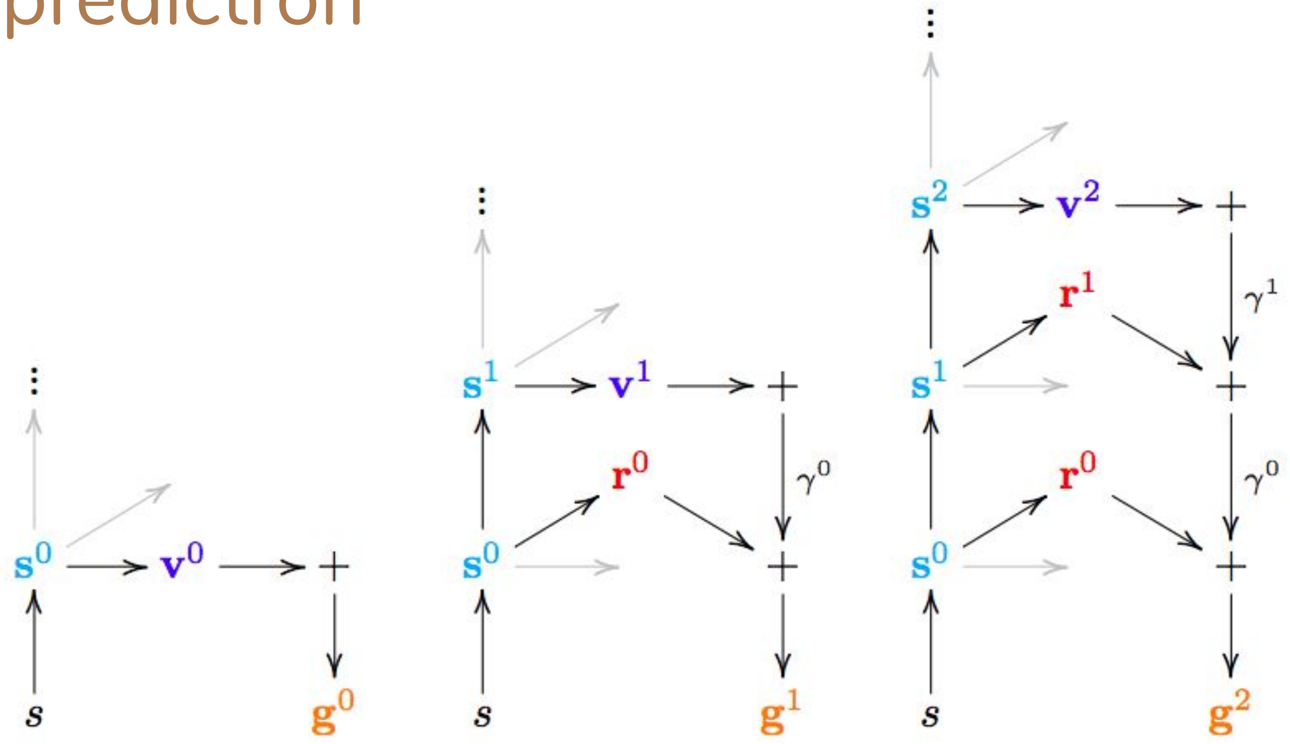
Planning with inaccurate models

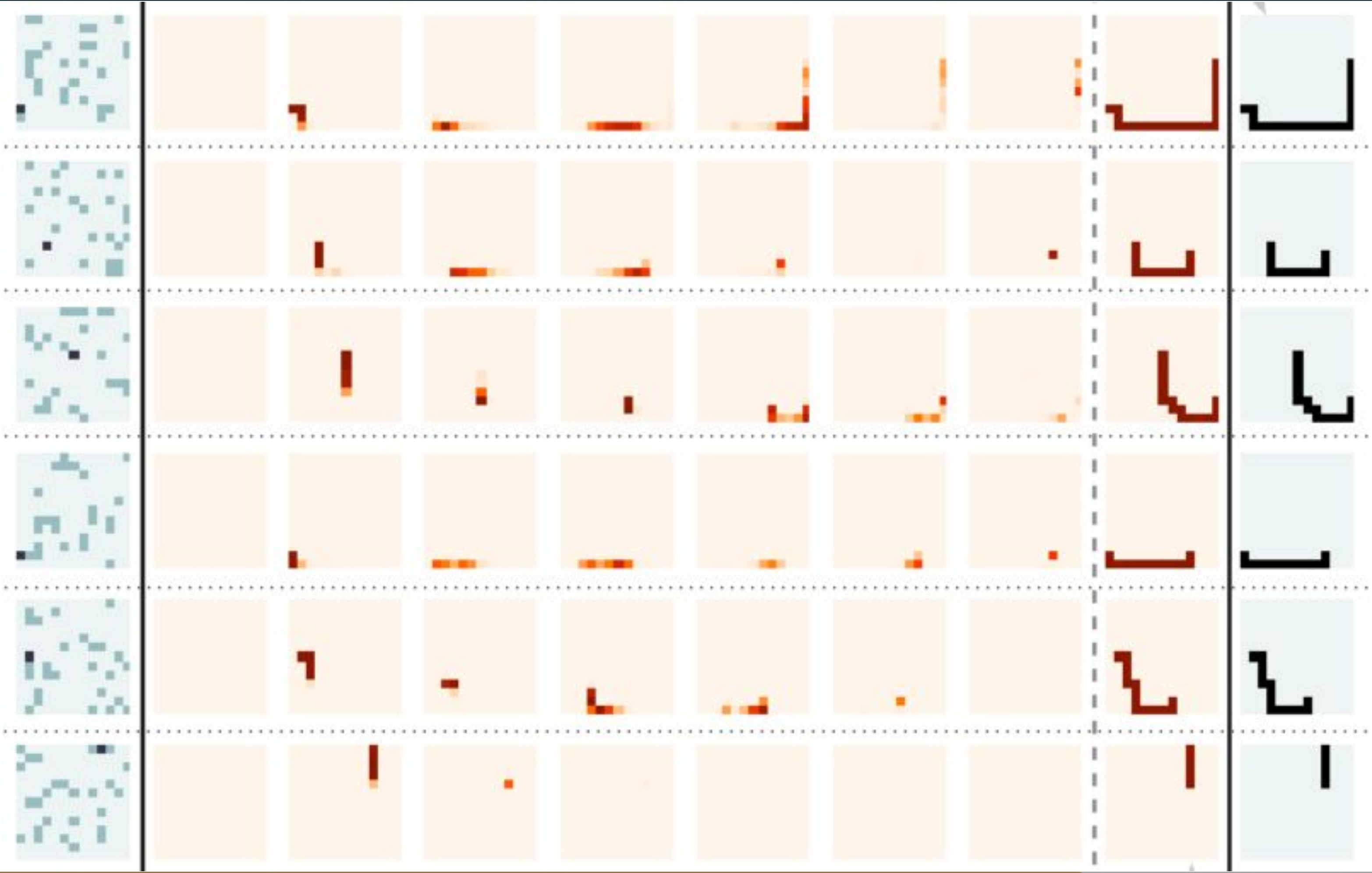
1. But **don’t trust** fully the model

2. Learn a fully **abstract** model that is **trained** to be suitable **for planning**

Ref: Silver et al.

The predictron





Learning State Representations

State representations

In **fully observable** environments the last observation fully encodes everything that the agent needs to know to select the next action $\pi^*(A_t | S_t) = \pi^*(A_t | O_t)$

Often, the state is not given to the agent, the agent must itself compress the **history** of all previous observations into a compact **state** $S_t = f\,(O_1, O_2, \ldots, O_t)$

The agent then needs to **jointly** learn both a state representation $S_t = f\,(H_t)$, and a policy $\pi\,(A_t | S_t)$ conditioned on such state representation.

Incremental state representations

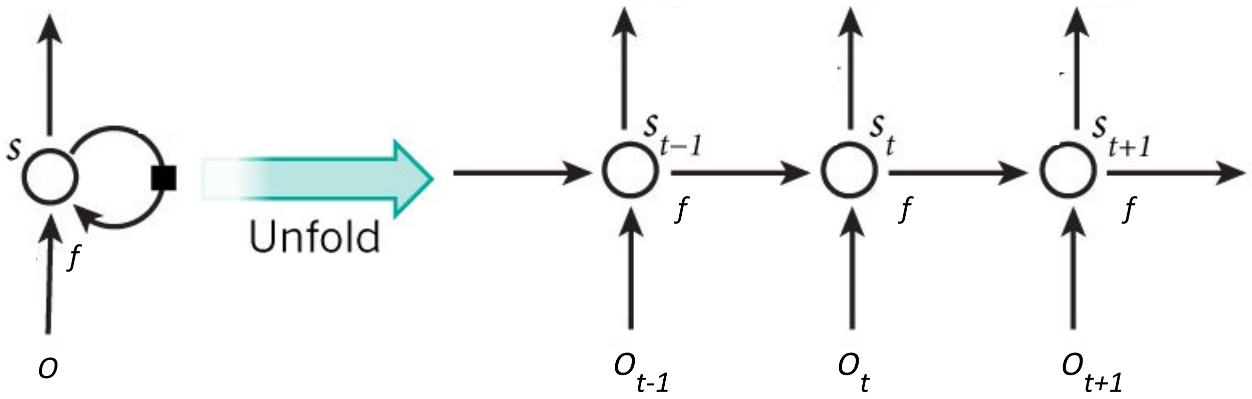
A crucial requirement for a good state representation is to be **incremental**:

$f : S_t, O_t \longrightarrow S_{t+1}$

So that the **cost** of computing the state is **constant** during the agent's lifespan

Recurrent state representations

This is exactly the functional form of recurrent neural networks (RNNs).



Recurrent state representations

We can therefore exploit years of deep learning research on recurrent neural network architectures:

- Long Short Term Memory networks (LSTMs)
- Gated Recurrent Units (GRUs)

video

Auxiliary Tasks

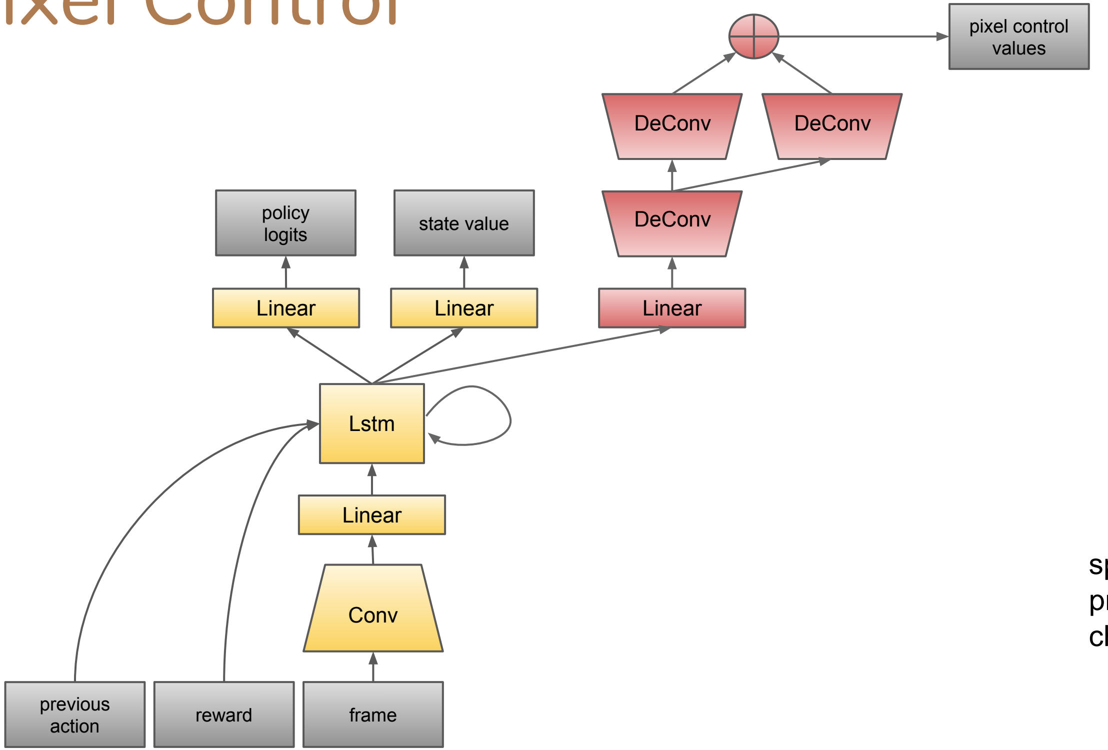
Learning about many things at once can be a powerful way of building better and more robust representations when using powerful Deep Learning models.

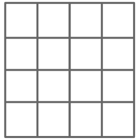
In deep RL this is sometimes done by defining **pseudo reward** signals on top of the usual stream of experience, and predicting/controlling these signal as **auxiliary tasks**.

These auxiliary tasks can be unsupervised and exploit the rich **structure** present in the stream of observations, even before the agent manages to collect its first reward.

Ref: Jaderberg et al.

Pixel Control





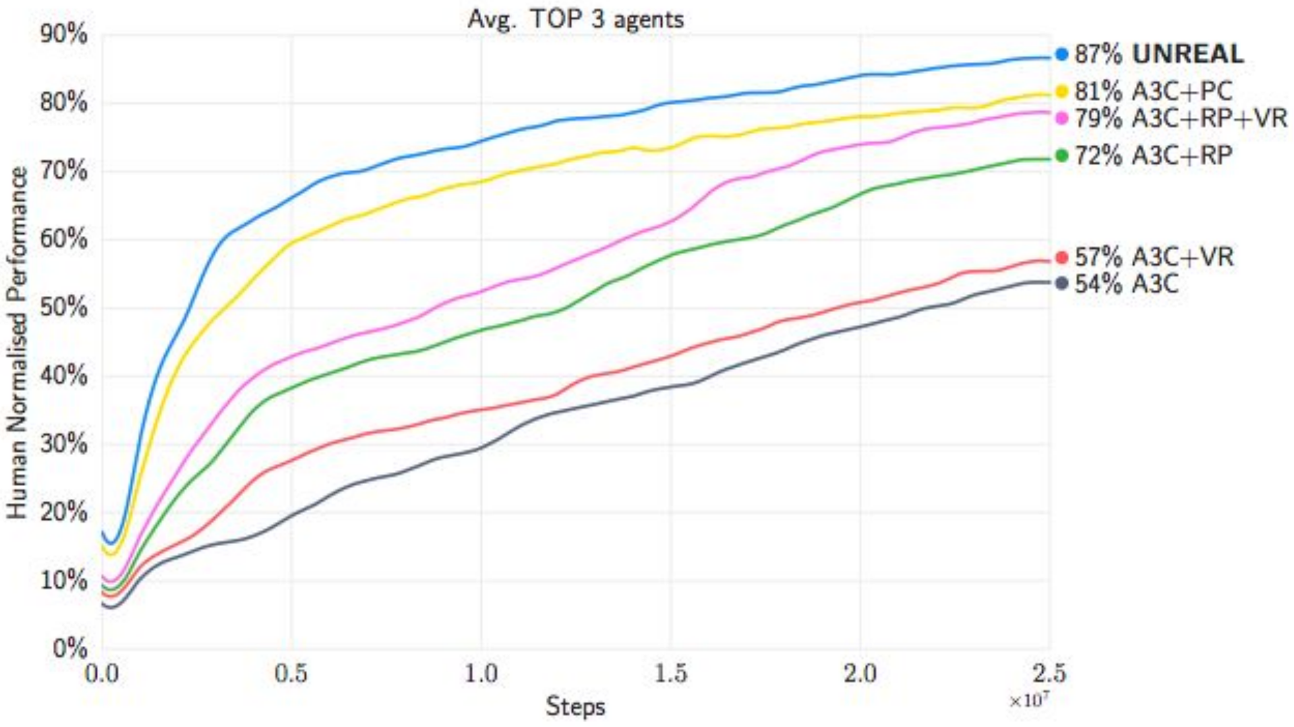
spatial grid of sudo rewards proportional to the average change of intensity of **pixels**.

Other auxiliary tasks

There are many other ways we can make our learning signal richer

- Immediate reward prediction
- Next state prediction
- Learn at multiple time-scales (predict values for different discounts)

Labyrinth Performance



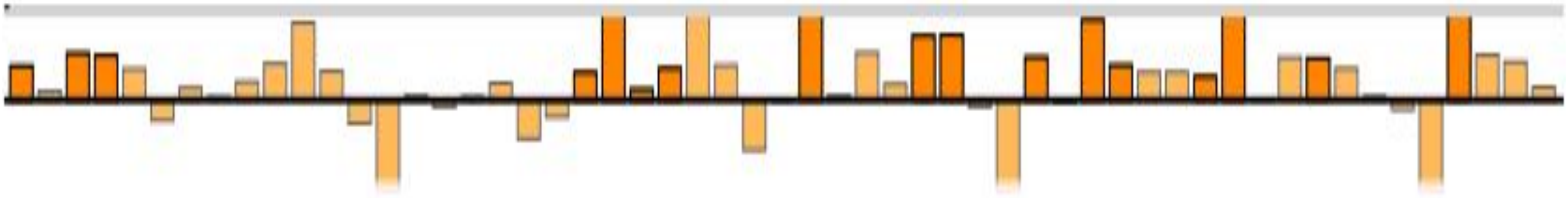
Ref: Dabney et al.

Distributional RL

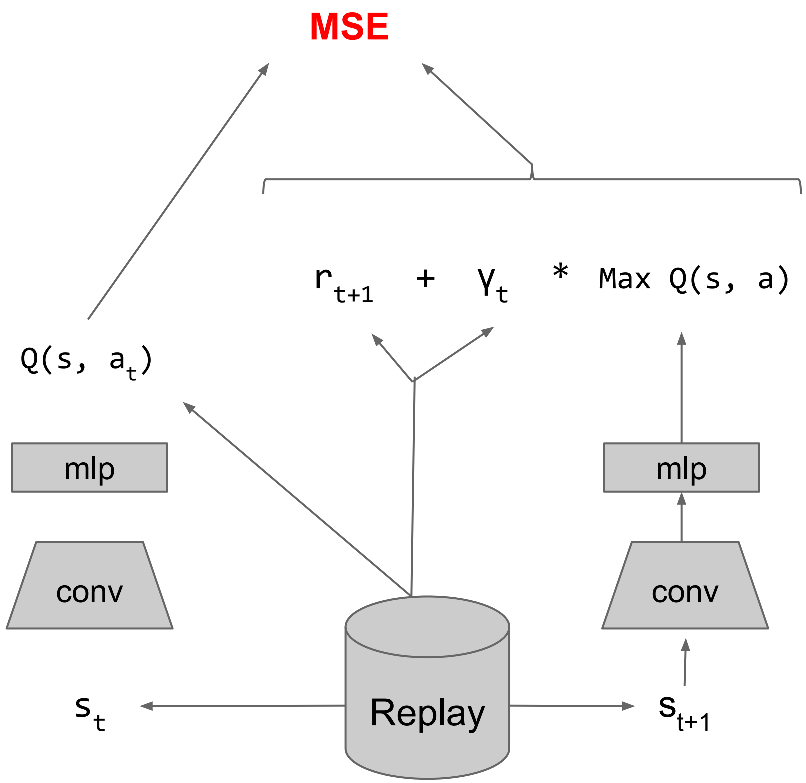
Instead of hand-crafting auxiliary predictions,
we can predict the entire **distribution** of returns

→ Distributional reinforcement learning

Improvement of distributional DQN over DQN on a collection of Atari games

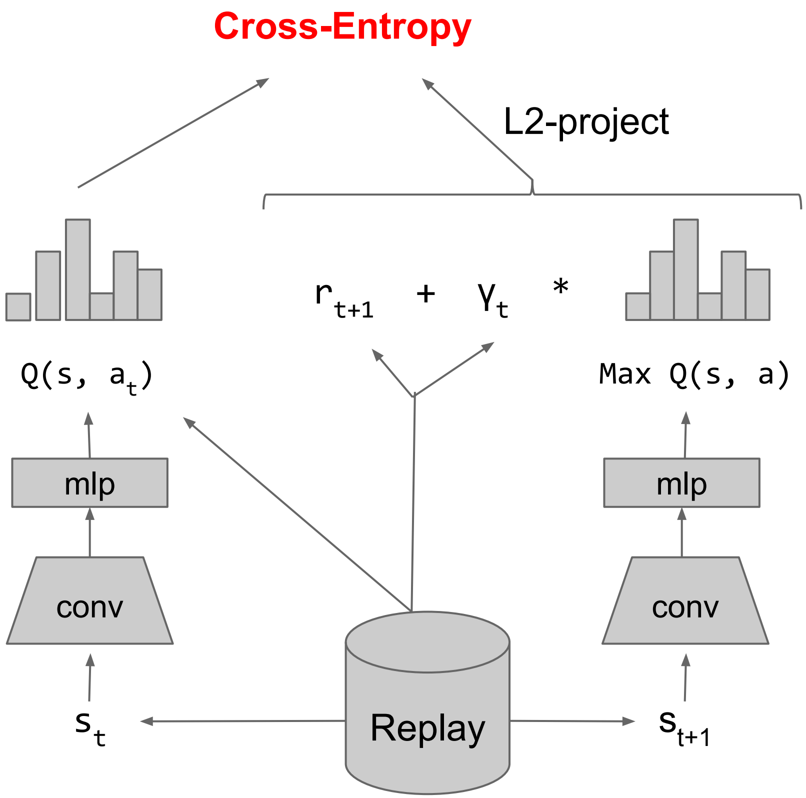


Std. DQN



Distrib. DQN

- Richer signal
- Well scaled loss
- Multi-modal



Distributional RL

Beyond providing a very rich signal for the agent to learn from, learning distributions of returns can be useful for **risk-sensitive** RL

For instance, we may want the agent to be **risk-averse**, and avoid courses of actions that have high mean but non-trivial probability of catastrophic outcomes

Scalable Deep RL

Credits: Sutton

The Bitter Lesson

1) AI researchers always try to build knowledge into their agents

2) This always helps **in the short term**

3) In the long run leads to diminishing returns and may even inhibits further progress

4) breakthrough eventually arrives by an approach based on **learning** and **search**.

Principles of scalability

1. Leverage **learning** and **search** rather than **human knowledge**
2. Computation **order of number of parameters**
3. Computation **independent of temporal span**

Scaling up resources

Deep Learning

More Data + More Compute ⟶ Better Performance

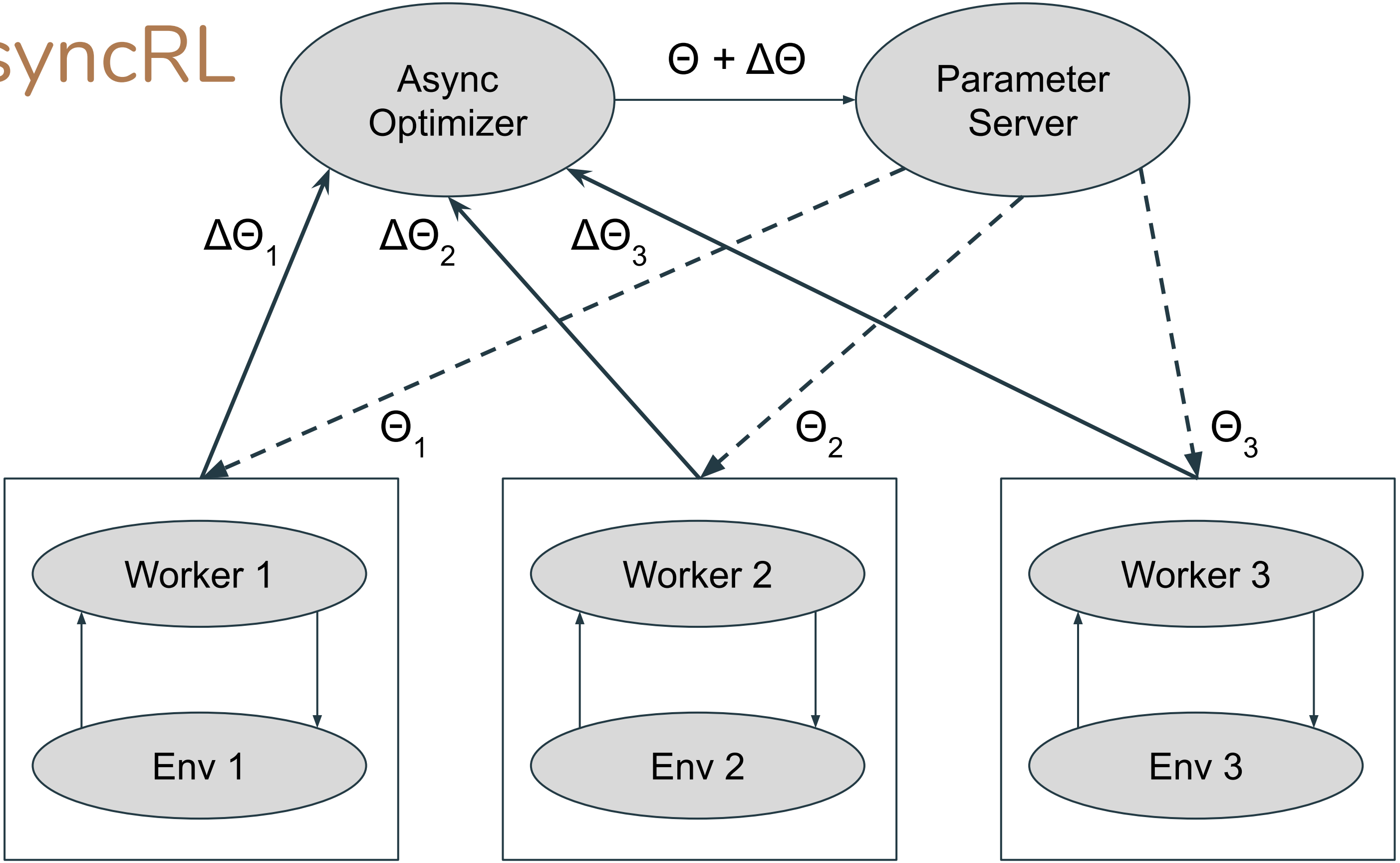
Does a similar principle hold for Deep Reinforcement Learning?

How can we leverage more compute to solve extremely complex RL problems faster?

- Multi-threading?
- GPUs?
- Distributed Computing?

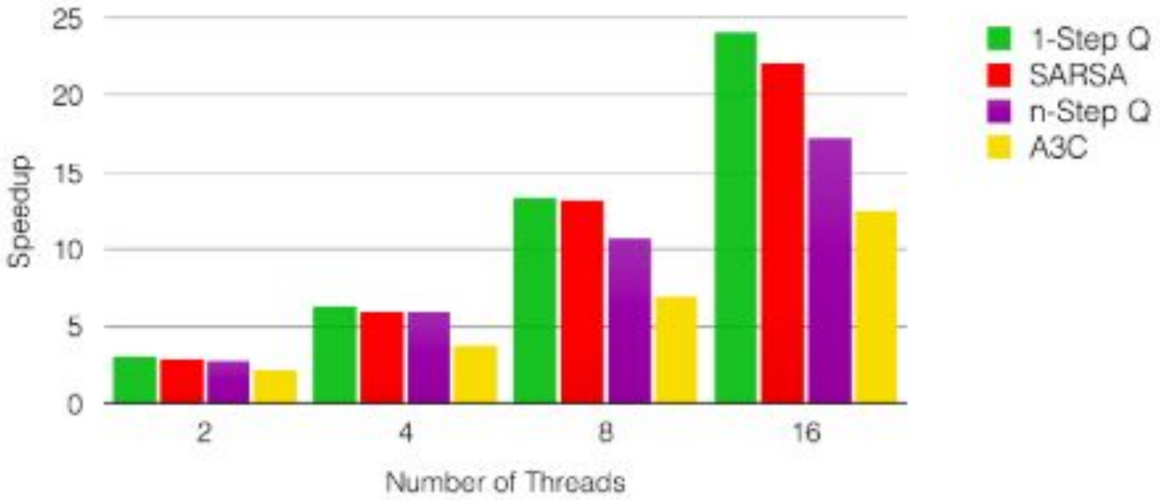
AsyncRL

Credits: Mnih et al.



Credits: van Hasselt et al.

AsyncRL



Actor-Learner decomposition

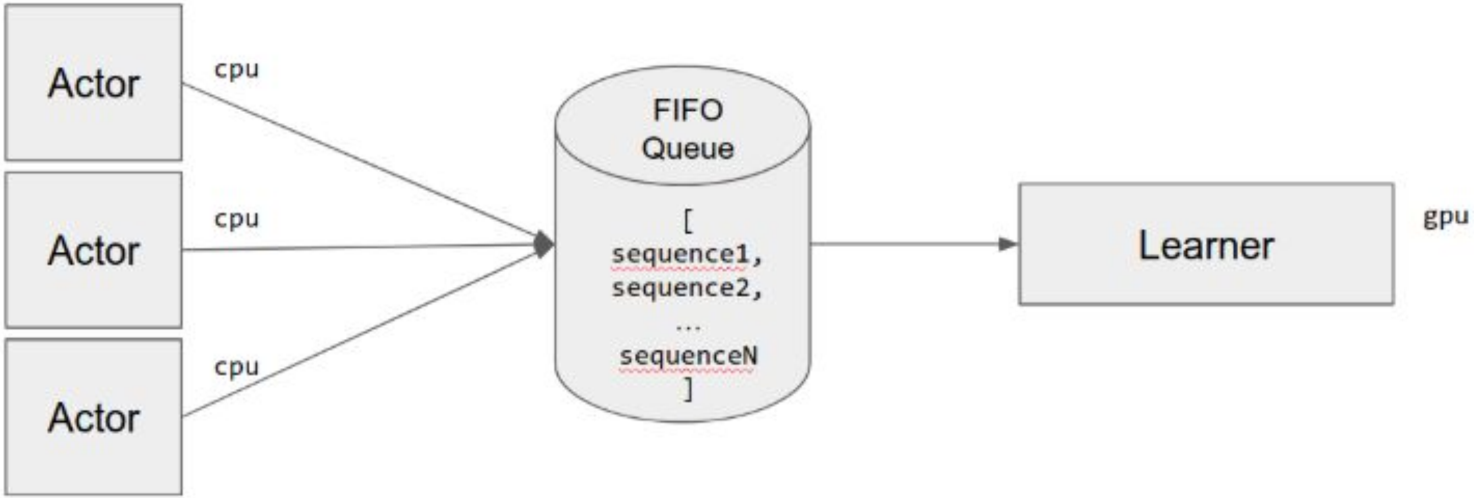
The main limitation of **asynchronous RL** is that gradients become **stale** very fast!

This means that as we scale up the number of parallel agents updating the same neural network the learning process starts becoming **unstable**

As a result, the most effective way of scaling up deep RL has been to not parallelize the full agent learning process but decompose **acting** from **learning**

- Experience generated by parallel actor doesn't become stale as fast as gradients
- Learning can be performed very efficiently in batch on modern hardware

IMPALA



Actor
Actor
Actor
cpu
cpu
cpu
FIFO
Queue
[
sequence1,
sequence2,
...
sequenceN
]
Learner
gpu

Off-policy learning

Off-policy learning is the problem of learning about some policy π from data generated according to a different policy μ.

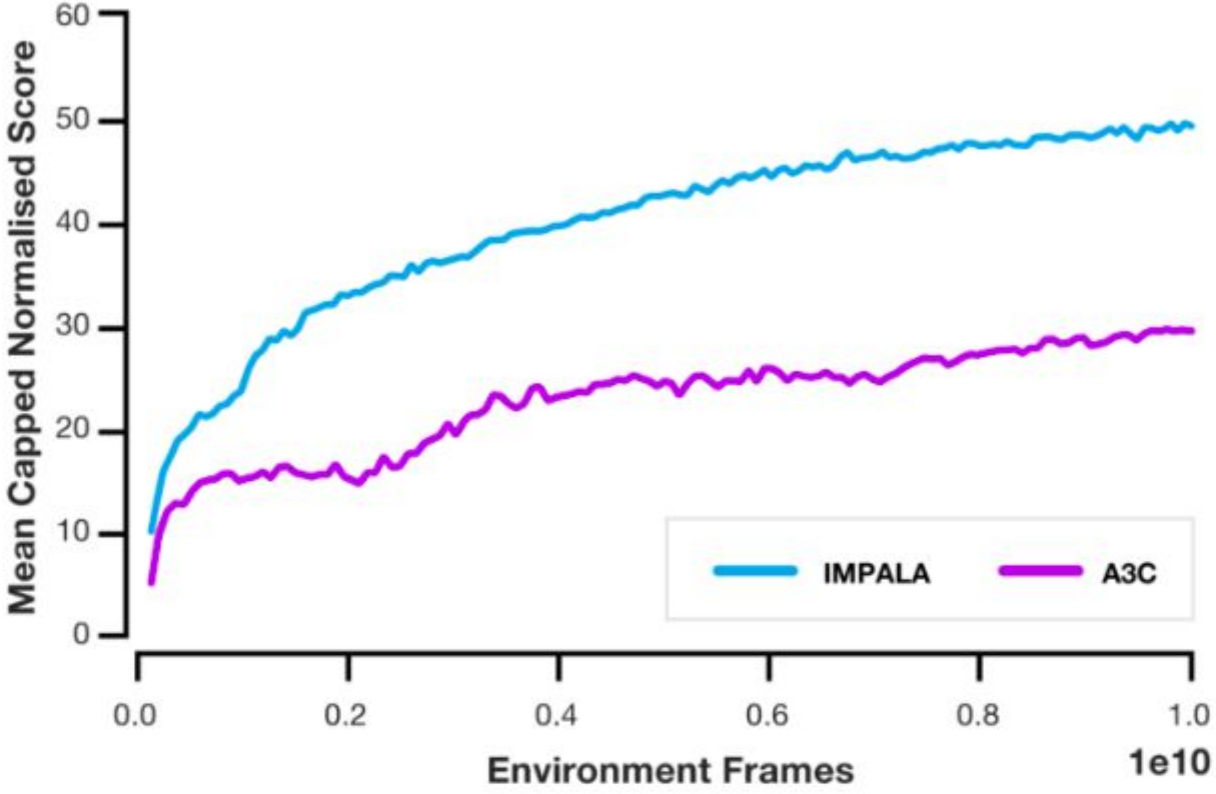
Some algorithms are designed to work off-policy

- e.g. Q-learning

Others aren't and in some cases may fail **catastrophically** when $\mu \neq \pi$

Importance sampling corrections can then be used to help with off-policy learning

Credits: Espeholt et al.



Multi-Task learning in Deep RL

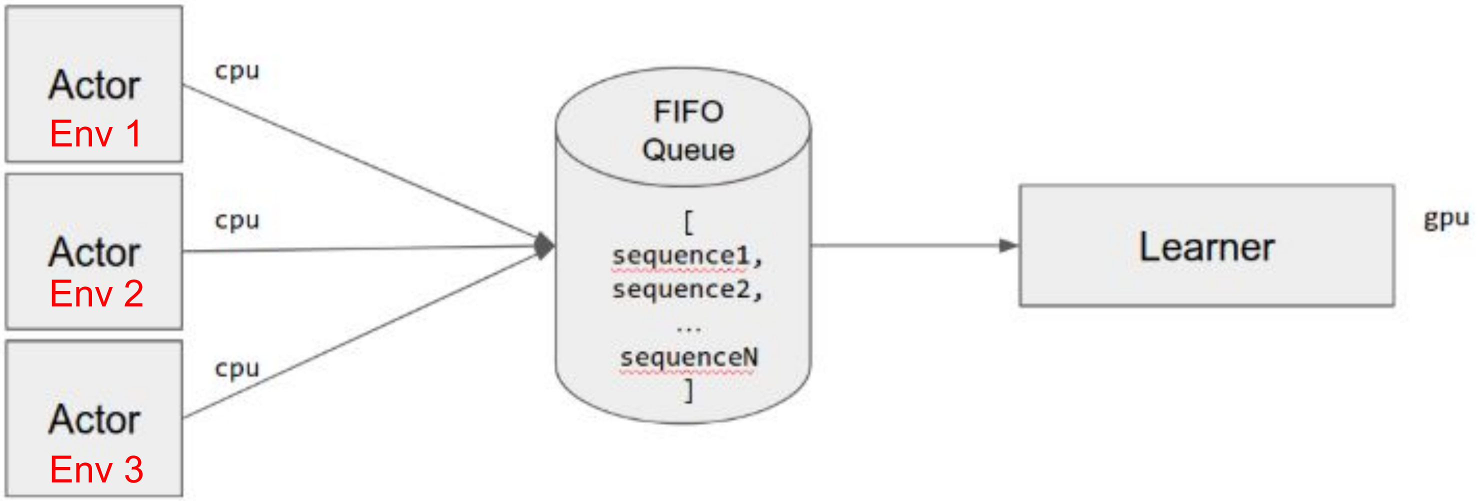
Curriculum learning

Some of the tasks could be easier or focus on specific skills that you need to master to solve the more complex ones!

Deep Exploration

We can get deep and much more meaningful exploration
by executing policies which are good for other tasks

MultiTask Learning



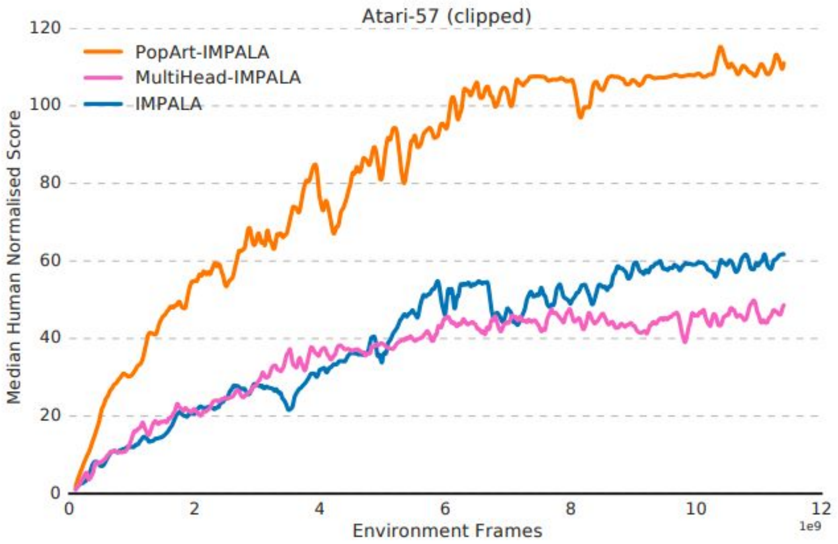
Different actors can interact with **different environments in parallel**!

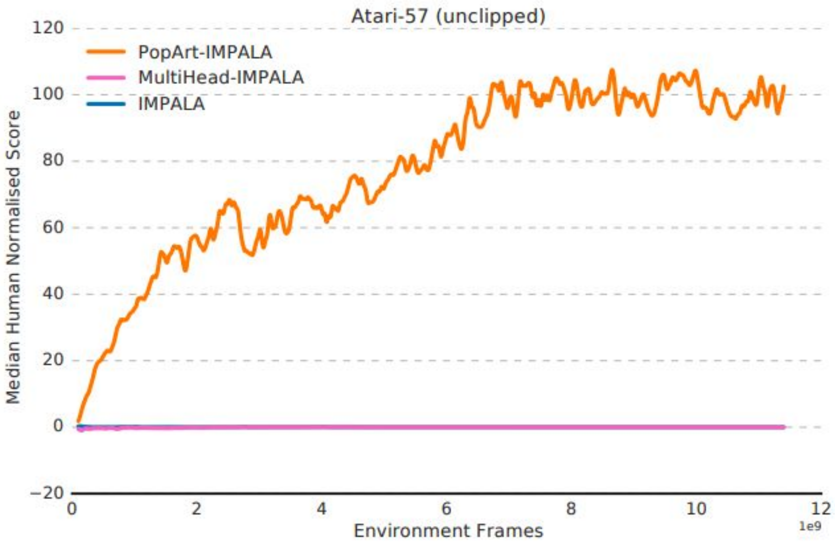
MultiTask RL

Multi-task learning comes with its own set of challenges:

- Catastrofic interference
- Different magnitude of updates across tasks

MultiTask Learning





Hyper-parameter Tuning

As our agents become more powerful they also become more complex

- Many hyper-parameters
- Subtle interactions
- Combinatorial space

Adaptive solutions

- Meta-learning
- Population Based Training

Meta-Gradient RL

The basic deep RL algorithm can be sketched as:

```
while True:
 step agent and environment → D
 get loss → L(D, θ)
 compute an update Δθ ⋉ grad_θ(L)
 θ = θ - Δθ
```

Meta-Gradient RL

The loss is also a differentiable function of some hyper-parameters, e.g. γ

```
while True:
 step agent and environment → D
 get loss → L(D, θ, γ)
 compute an update Δθ(γ) ⋉ grad_θ(L)
 θ = θ - Δθ(γ)
```

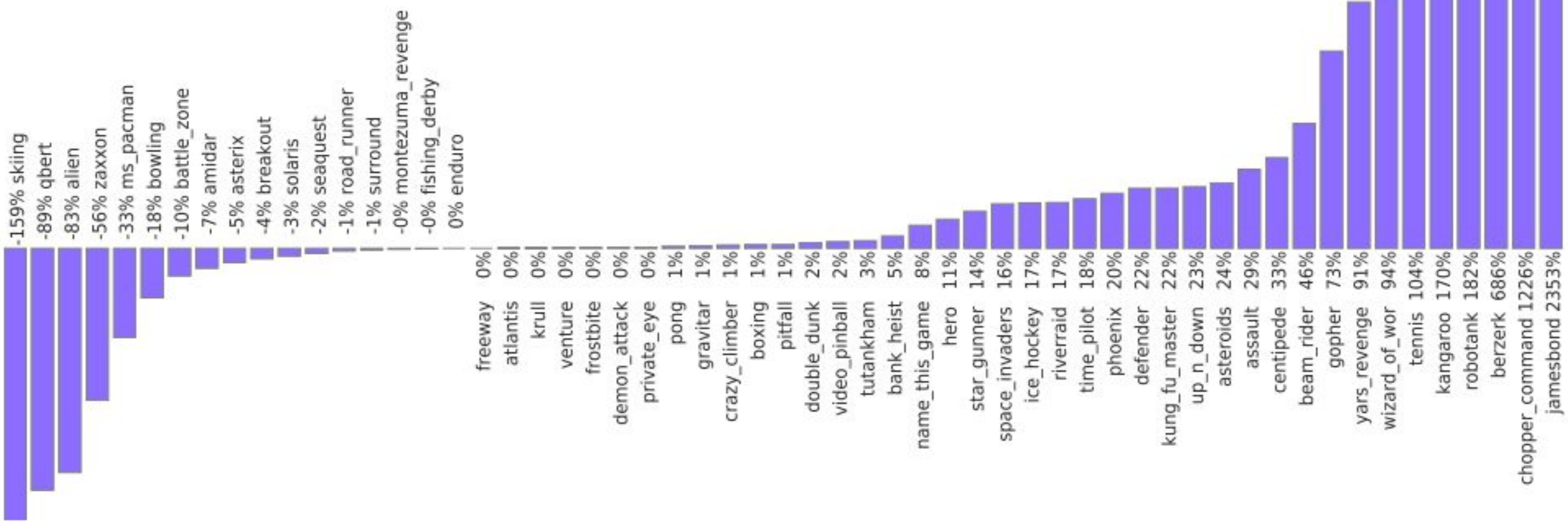
Credits: Xu et al.

Meta-Gradient RL

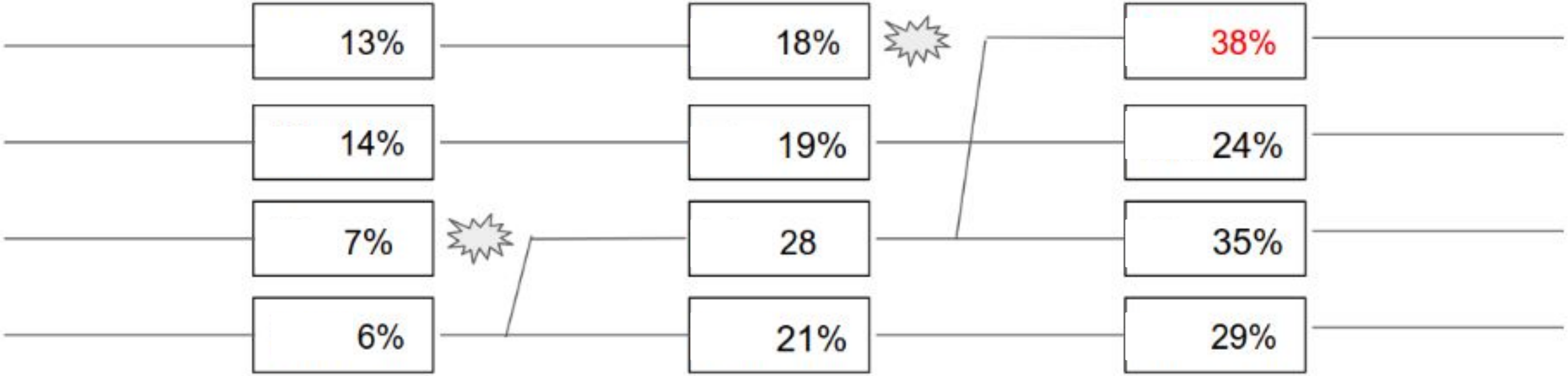
```
while True:
  step agent and environment → D_1
  get loss → L(D_1, θ, γ)
  compute an update Δθ(γ) ⋉ gradθ(L)
  step agent and environment → D_2
  get loss → L(D_1, θ - Δθ(η), 1.)
  compute an update Δγ ⋉ grad_γ(L)
  θ = θ - Δθ(γ)
  γ = γ - Δγ(γ)
```

Credits: Xu et al.

Meta-learning the discount

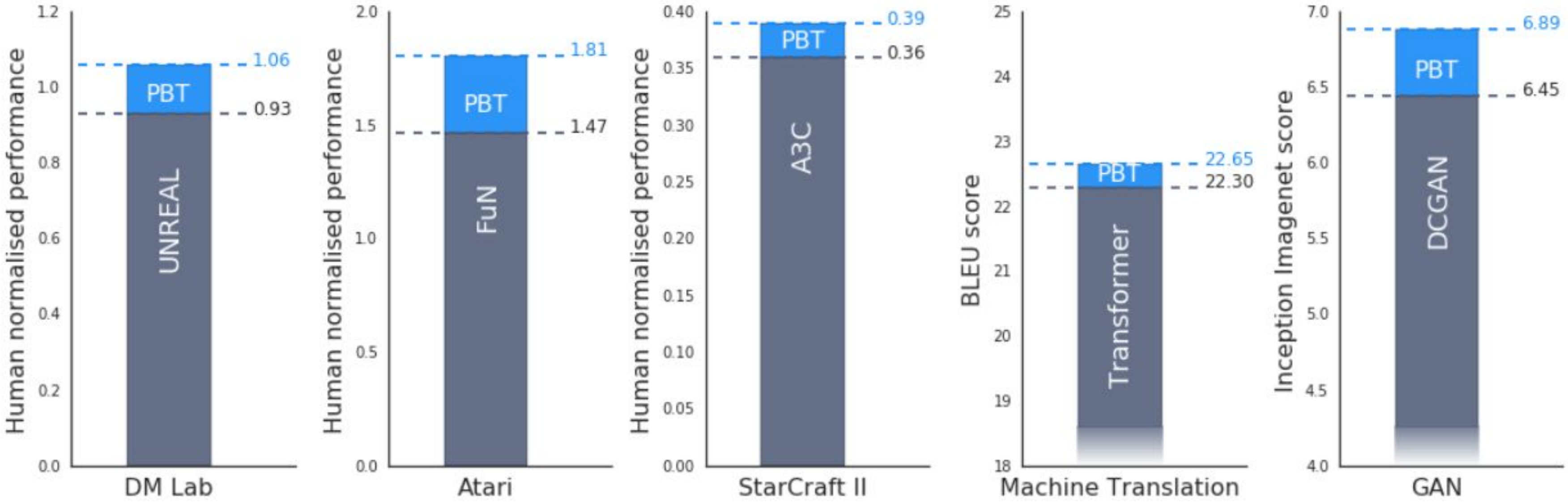


Population Based Training



Population based training searches for hyper-parameter **schedules**.

Credits: Jaderberg et al.



Model-Based RL at Scale

Search and **learning** are the most scalable techniques we have,

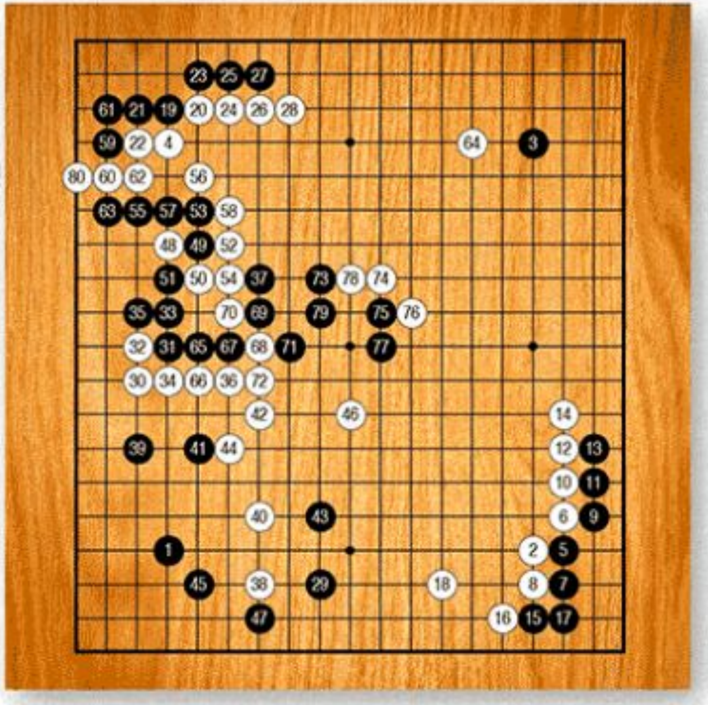
Model-based RL allows us to leverage both!

AlphaZero

Go is an ancient game very popular in China, Japan and Korea.

Has been played for 3000 years and was considered a grand challenge of AI

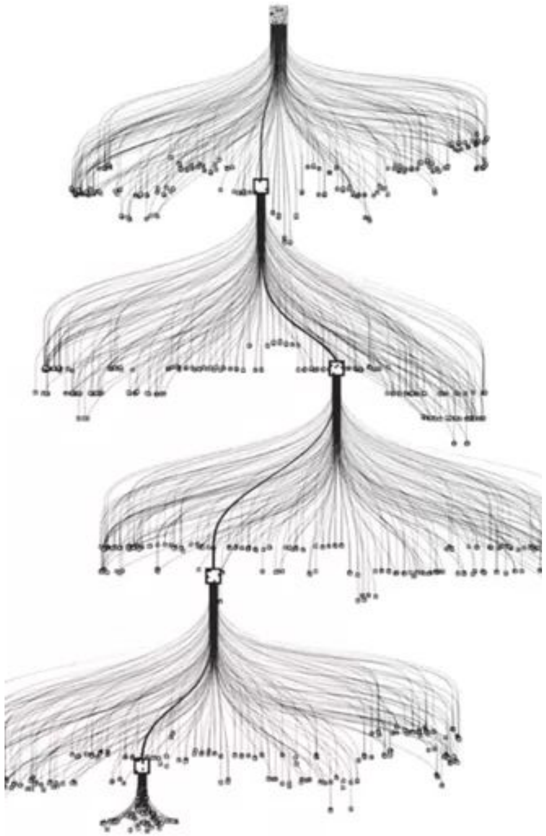
An RL algorithm **AlphaZero** beat the world champion last year after learning exclusively from self-play



AlphaZero

AlphaZero trains a deep neural network to estimate both a **policy** and a **value** function.

Then uses these in combination with a perfect model of Go performing **Monte Carlo Tree Search**.



The End