Practical Deep RL

Volodymyr Mnih, DeepMind

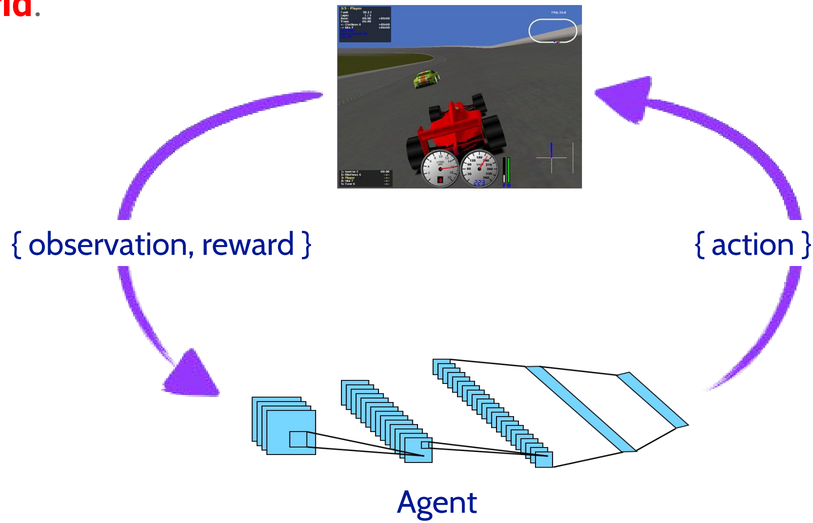
UCL

March 16, 2017

DeepMind

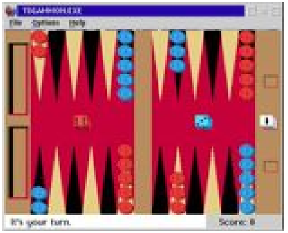
Deep Reinforcement Learning == AI?

- AI - building machines that are good at **sequential decision-making** **problems humans care about**.
- **Reinforcement learning** is a general framework for studying sequential decision-making.
- Deep learning:
  - Current best way of making computers **perceive the world**.
  - More generally it is a framework for learning in deep computational graphs.



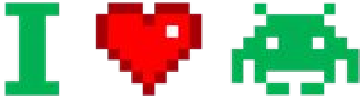
The Deep RL Boom

TD-Gammon (Tesauro, 1989-1995)

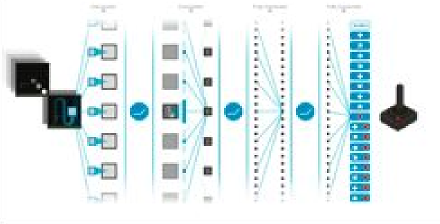


Slot car driving (Lange & Riedmiller, 2012)

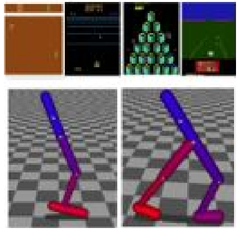
Arcade Learning Environment (Bellemare et al, 2013)



Deep Q-Networks (2013, 2015)



Trust region policy optimization (Schulman et al, 2015)



End-to-end training on real robots (Levine et al, 2015)

AlphaGo

Deep RL vs Deep Learning

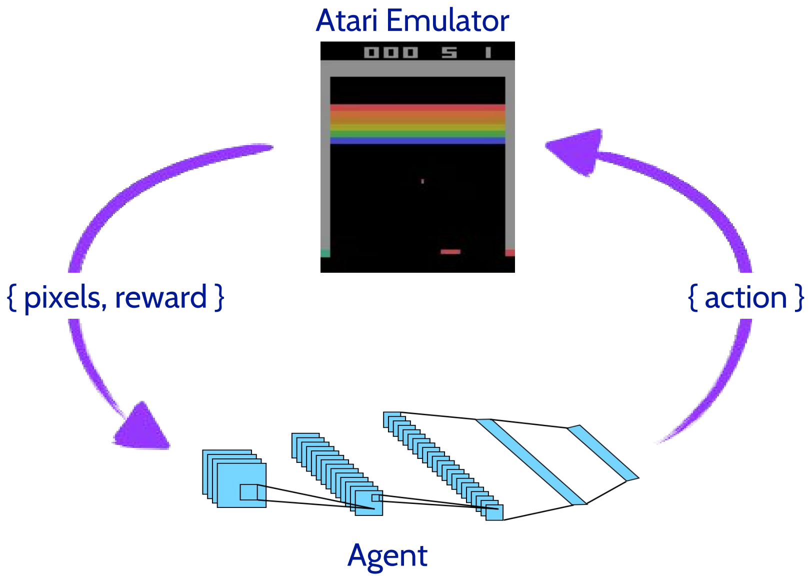
- How is deep RL different from standard deep learning?
- The data distribution is non-stationary.
  - Neural nets don’t like this. Most of the theory no longer applies.
- The data distribution is determined by the agent’s actions.
  - Exploration vs exploitation.
  - You can get stuck in local minima. Optimization really matters.
- Sparse/delayed feedback.
- Training neural nets with RL was thought to be inherently unstable (Tsitsiklis & Van Roy, 1997).

Outline

- DQN in more detail.
- Faster agents through parallel training.
- Better data efficiency through unsupervised RL.
- Some practical advice.

Deep Q Networks (DQN)

- Represent the action value (Q) function using a convolutional neural network.
- Train using end-to-end Q-learning.
- Can we do this in a stable way?



DQN

Initialize **target network** $\theta^- \leftarrow \theta$

For each time step ***t***

Take action a_t, and observe r_t, s_{t+1}

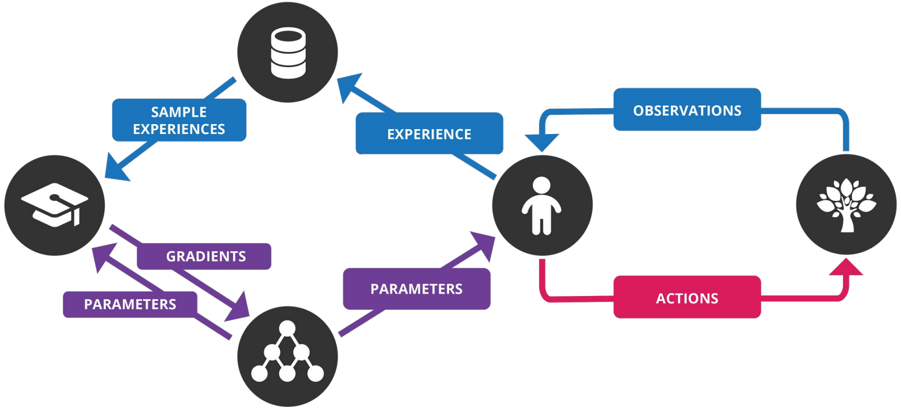
Sample (s,a,r,s') from **replay memory**

Generate **target** $r + \delta\gamma \max_{a'} Q(s', a'; \theta^-)$

Take SGD step following $\theta_{t+1} \leftarrow \theta_t - \eta \frac{\partial L(\theta)}{\partial \theta_t}$

Update **target network** if t % k : $\theta^- \leftarrow \theta$

Store (s_t, a_t, r_t, s_{t+1}) in **replay memory**



DQN

- High-level idea - make Q-learning look like supervised learning.

- Apply Q-updates on batches of past experience instead of online:
    - Experience replay (Lin, 1993).
    - Previously used for better data efficiency.
    - Makes the data distribution more stationary.

- Use an older set of weights to compute the targets (**target network**):
    - Keeps the target function from changing too quickly.

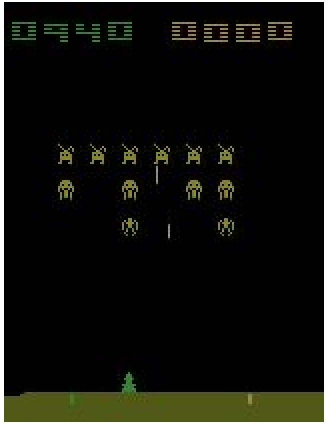
$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

Target Network Intuition

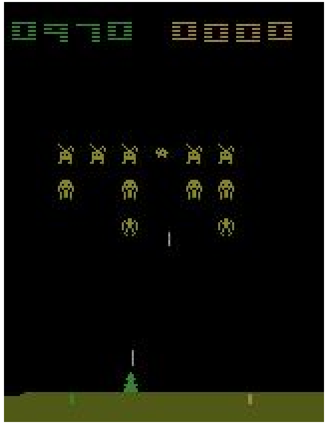
- Changing the value of one action will change the value of other actions and similar states.
- The network can end up chasing its own tail because of bootstrapping.
- Somewhat surprising fact - bigger networks are less prone to this because they alias less.

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

s



s'



Neural Fitted Q Iteration

- NFQ (Riedmiller, 2005) trains neural networks with Q-learning.
- Alternates between collecting new data and fitting a new Q-function to all previous experience with batch gradient descent.

NFQ_main() {

input: a set of transition samples D; output: Q-value function Q_N

  k=0

  init_MLP() $\rightarrow Q_0$;

  Do {

    generate_pattern_set $P = \{(input^l, target^l), l = 1, \ldots, \#D\}$ where:

      $input^l = s^l, u^l,$

      $target^l = c(s^l, u^l, s'^l) + \gamma \, min_b Q_k(s'^l, b)$

    Rprop_training$(P) \rightarrow Q_{k+1}$

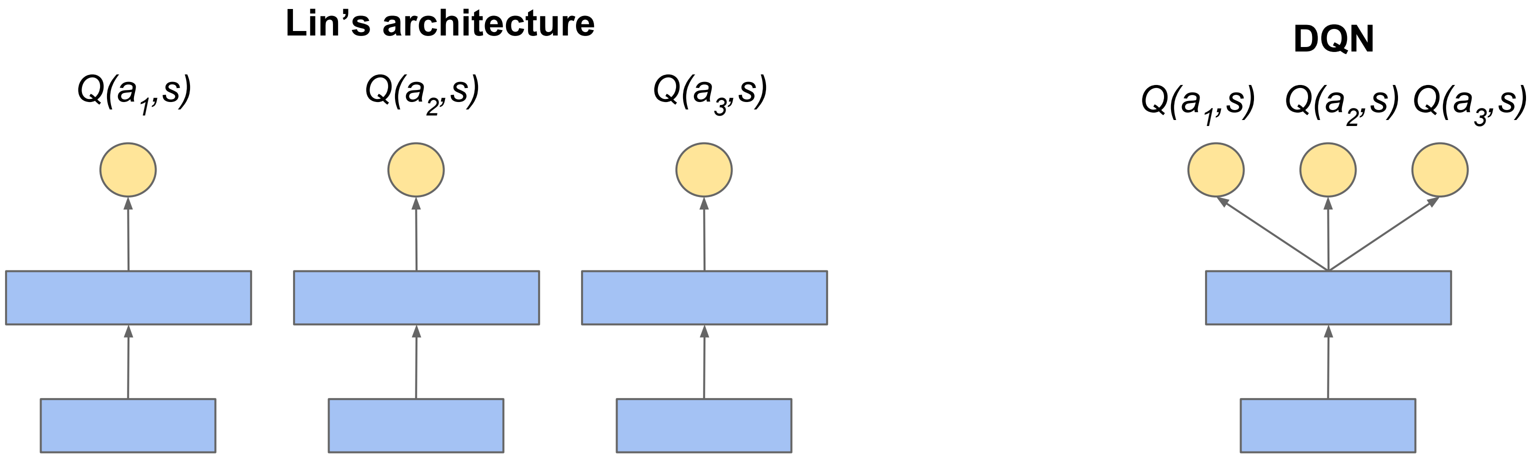
    k:= k+1

  } WHILE $(k < N)$

- DQN can be seen as an online variant of NFQ.

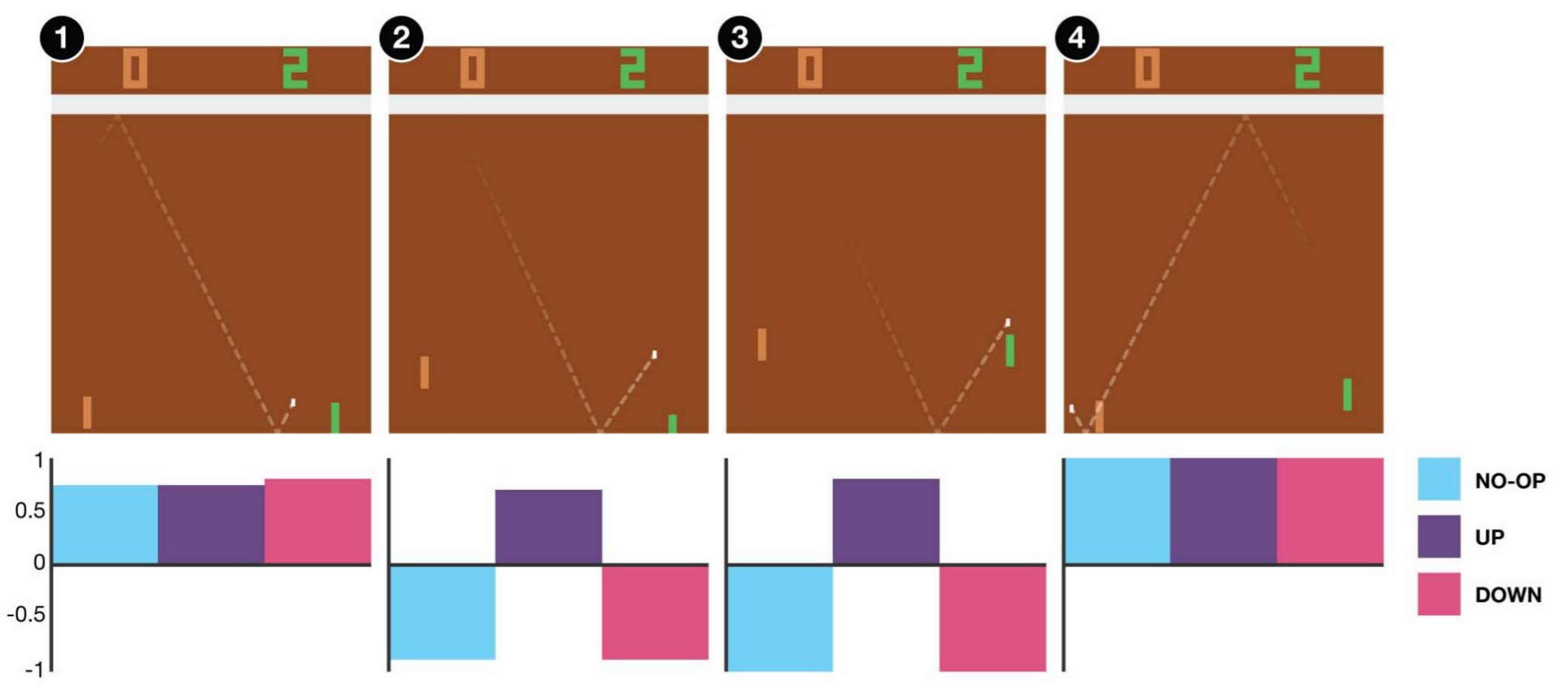
Lin’s Networks

- Long-Ji Lin’s thesis “Reinforcement Learning for Robots using Neural Networks” (1993) also trained neural nets with Q-learning.
- Introduced experience replay among other things.
- Lin’s networks did not share parameters among actions.



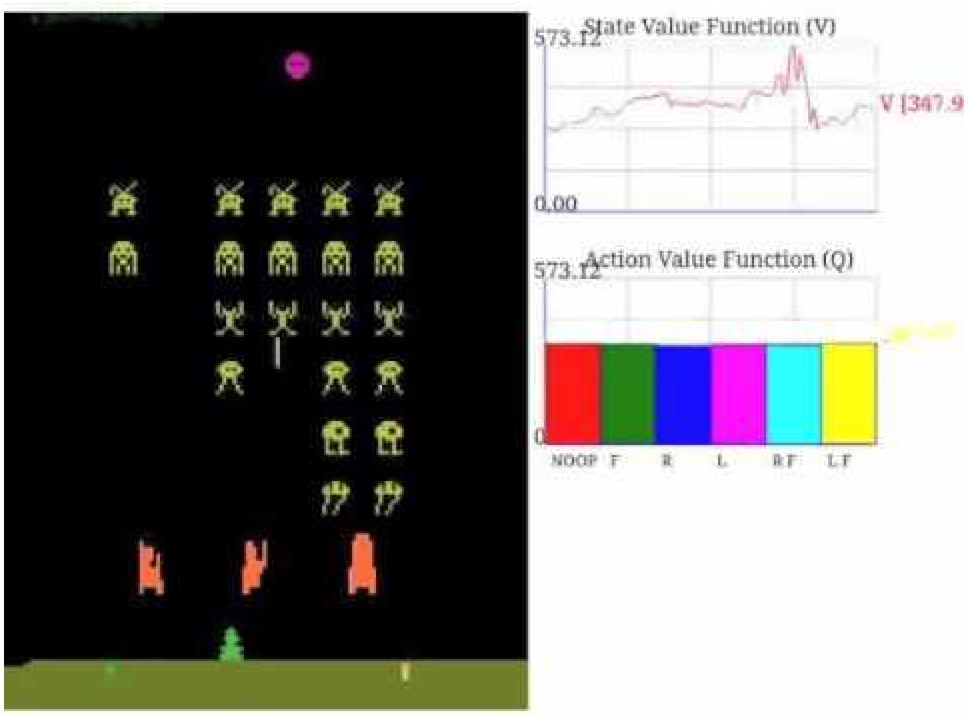
DQN Playing ATARI

Action Values on Pong

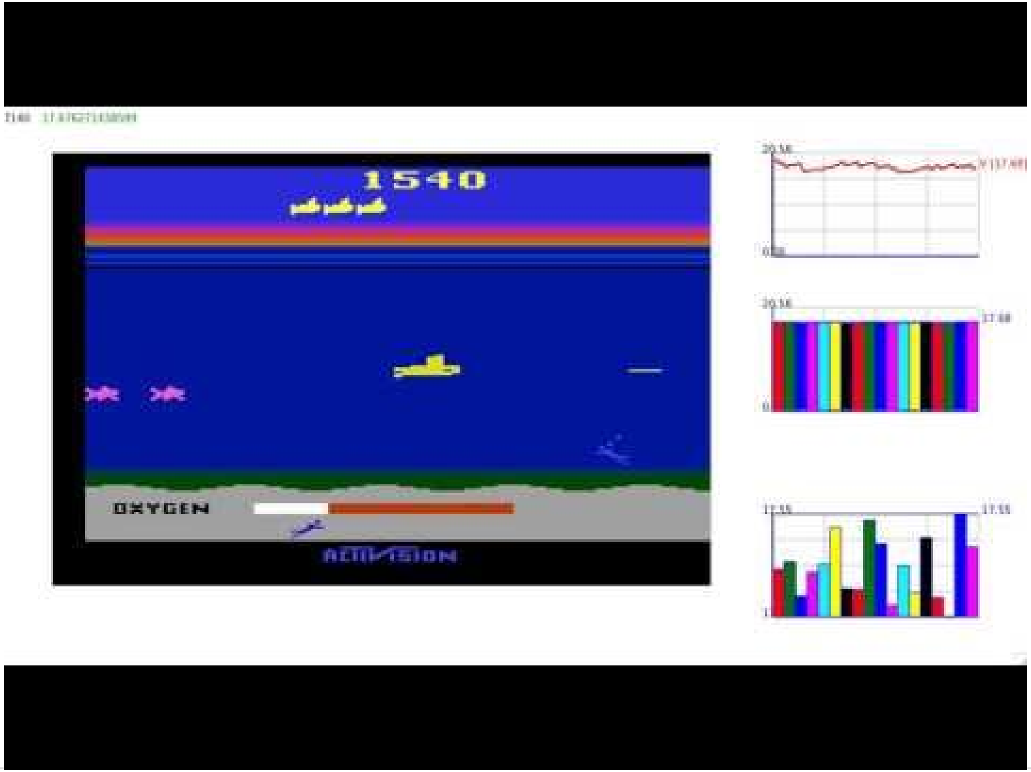


DeepMind

Learned Value Functions



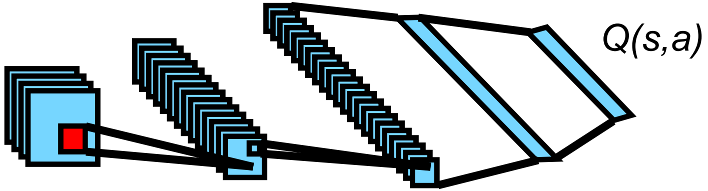
Sacrificing Immediate Rewards

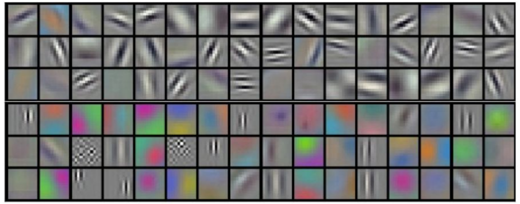


DeepMind

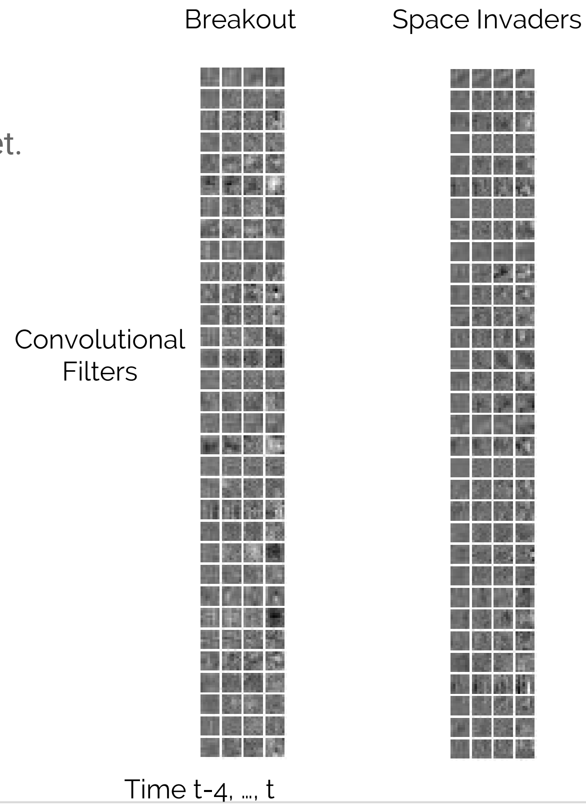
DQN Atari Filters

- Visualizing the convolutional filters learned by DQN.
- Surprisingly little structure compared to convolutional filters on ImageNet.



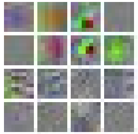


ImageNet filters, Krizhevsky et al. (2012)

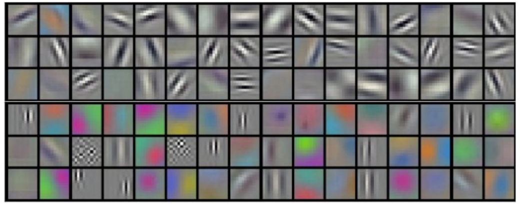


Labyrinth Filters

- Filters learned by A3C on a 3D environment.
- Visually richer environment produces more structured and interesting filters.



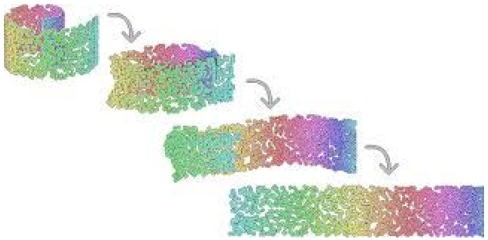
Labyrinth agent filters



ImageNet filters, Krizhevsky et al. (2012)

t-SNE

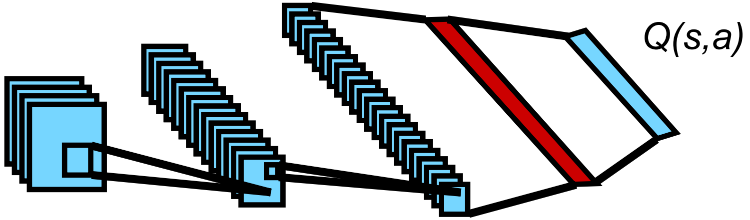
- Problem - visualizing high dimensional data by projecting it to 2D.
- Multi-Dimensional Scaling (MDS):
  - Project to 2D while maintaining pairwise distances.
  - Crowding problem - impossible to maintain all distances.
- t-distributed Stochastic Neighbor Embedding (Van Der Maaten and Hinton, 2008).
  - Try to preserve neighbors but allow some nearby points in the high dimensional space to be far apart in the low dimensional space.
  - Nearby points in a t-SNE plot are nearby in the original space.
  - Most distant points in a t-SNE plot are distant in the original space, but a few could be nearby.

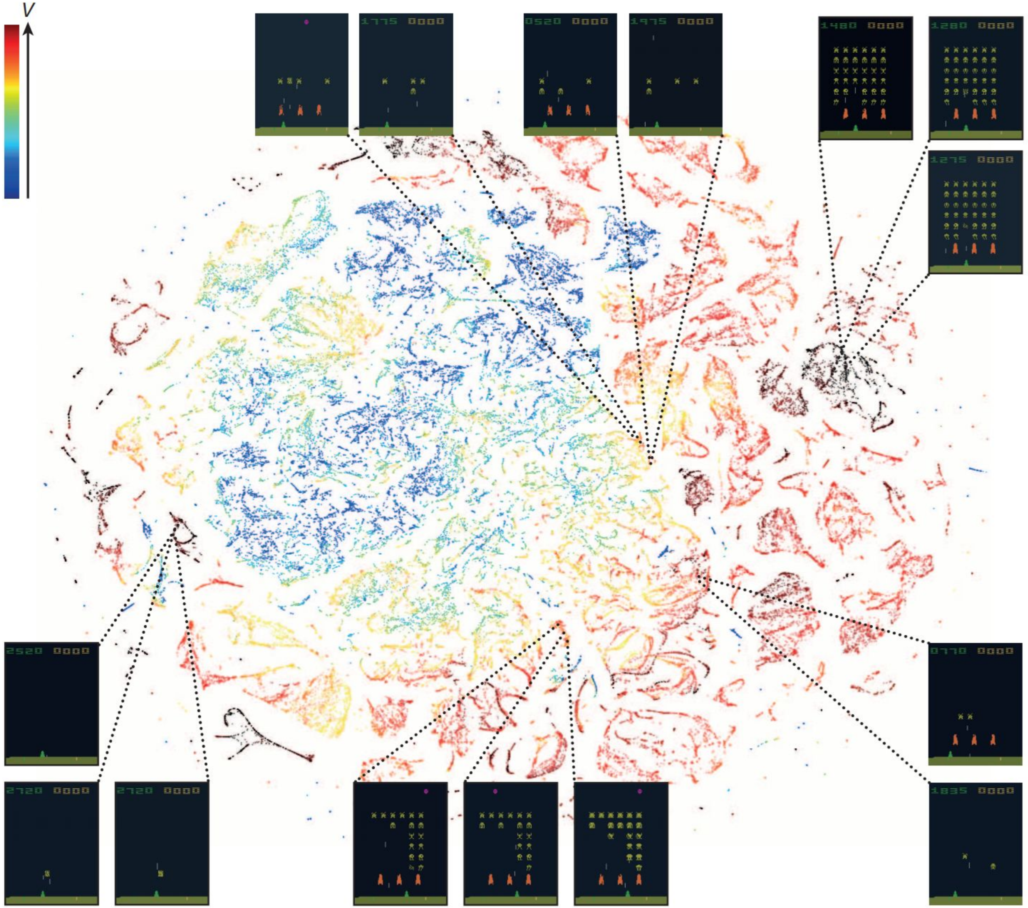


Gashler et al., 2011

Space Invaders Representation

- t-SNE embedding of the last hidden layer of the DQN network.

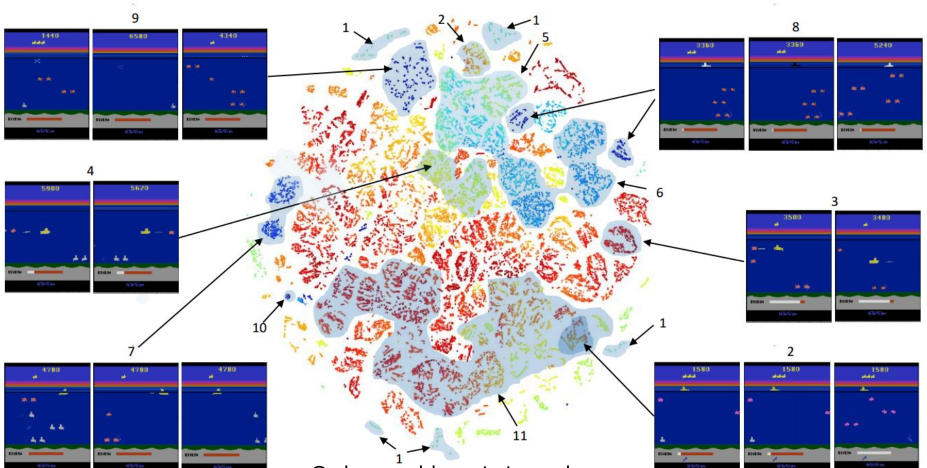




DeepMind

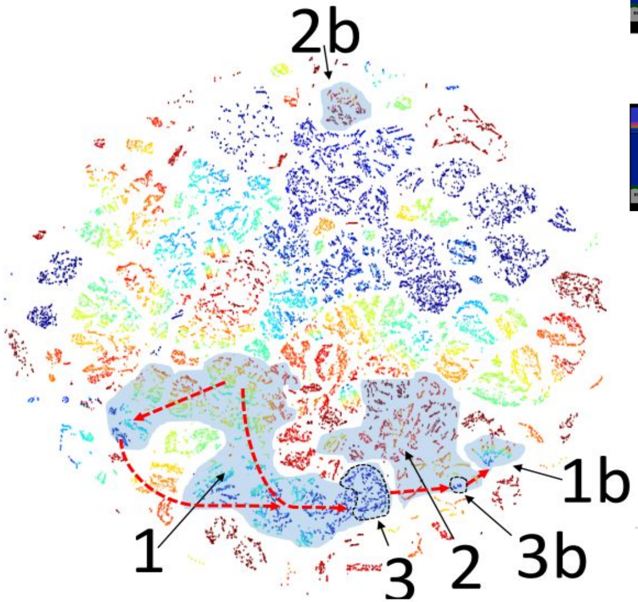
Understanding DQN (Zahavy et al., 2015)

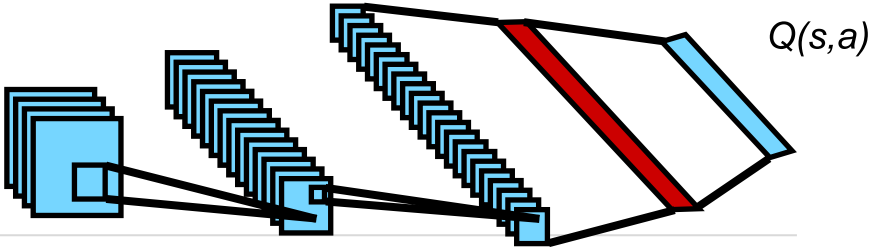
- More detailed t-SNE analysis of DQN.
- Argues that DQN learns hierarchical state aggregation and hierarchical policies.



Coloured by state value.

Coloured by remaining oxygen.



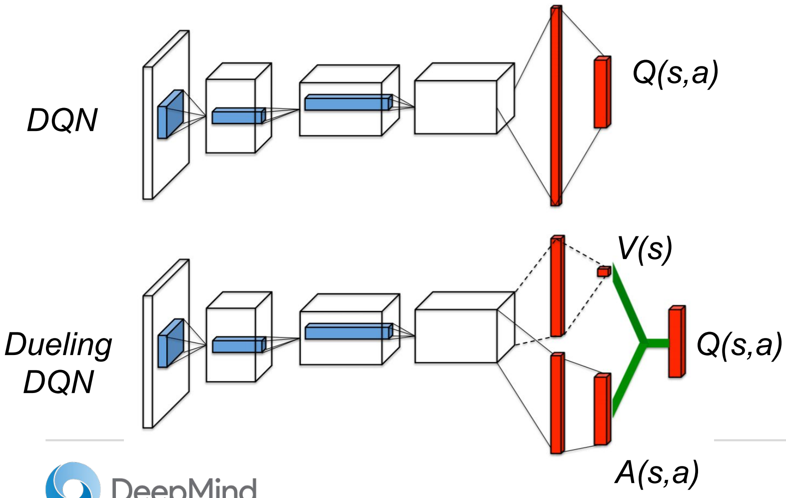


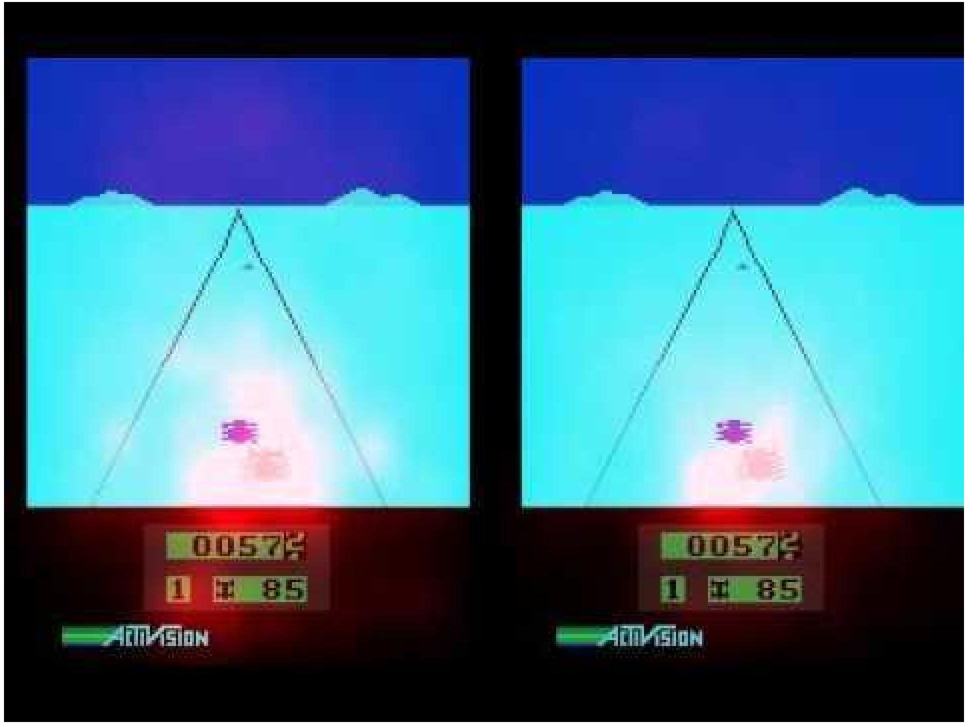
Saliency Maps

- Value-Advantage decomposition of Q:

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

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



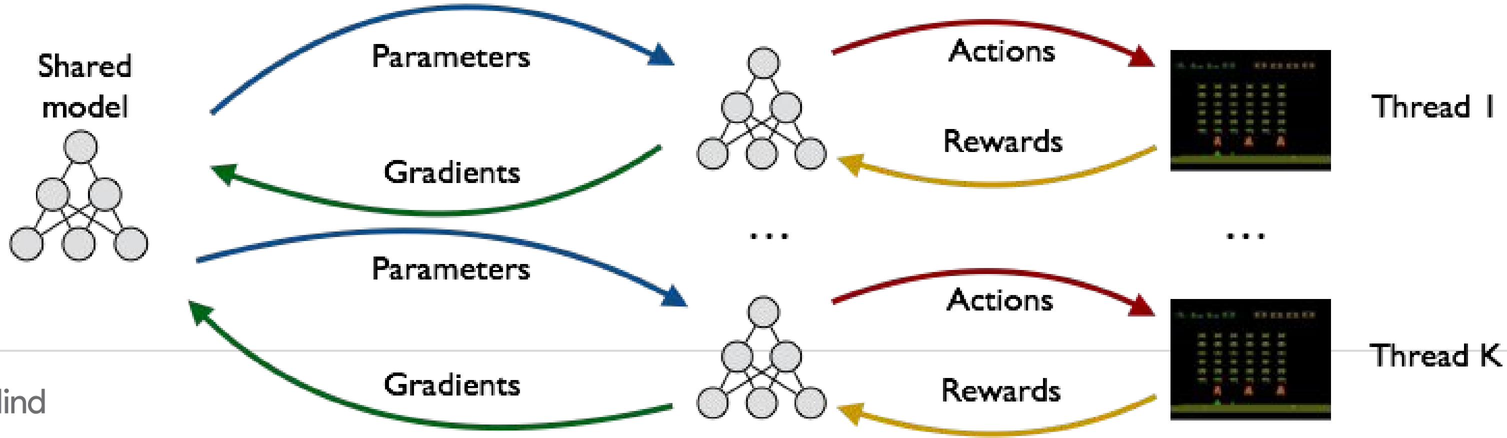


Beyond DQN

- DQN is very robust, but has some limitations:
  - Computationally expensive.
    - But see Prioritized DQN (Schaul et al., 2016), Dueling Prioritized DQN (Wang et al, 2016)
  - Has not worked as well with recurrent networks.
  - Does not handle continuous actions (see DDPG Lillicrap et al., 2015).
- Is there a method that allows:
  - Fast training on a single machine - hours for simple Atari games.
  - Freedom to use on or off-policy methods.
  - Flexibility - discrete or continuous actions, feedforward or recurrent models, etc.

AsyncRL

- Asynchronous training of RL agents:
    - Parallel actor-learners implemented using **CPU threads** and shared parameters.
    - Online **Hogwild!**-style asynchronous updates (Recht et al., 2011, Lian et al., 2015).
    - No replay? Parallel actor-learners have a similar stabilizing effect.
    - Choice of RL algorithm: on-policy or off-policy, value-based or policy-based.



1-step Q-Learning

- Parallel actor-learners compute online 1-step update

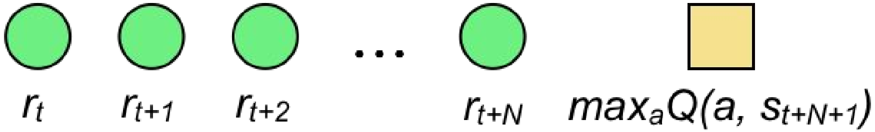
$$y \leftarrow r + \gamma \max_{a'} Q(s', a'; \theta^-)$$

$$\Delta\theta \leftarrow \Delta\theta + \frac{\partial \left(y - Q(s, a; \theta)\right)^2}{\partial \theta}$$

- Gradients accumulated over minibatch before update

N-step Q-Learning

- Q-learning with a uniform mixture of backups of length 1 through N.



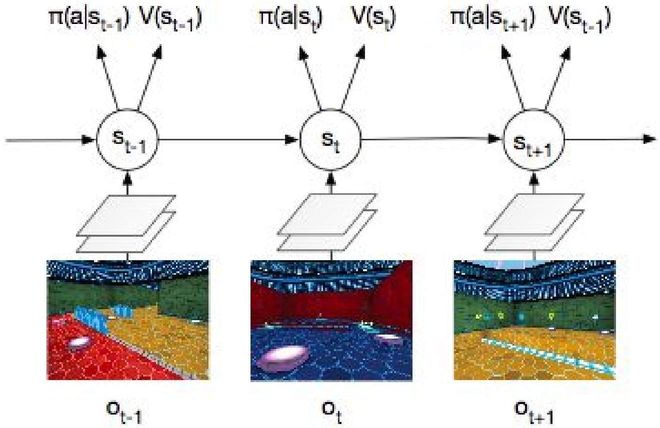
$$y \leftarrow \sum_{k=0}^{N-1} \gamma^k r_{t+k} + \gamma^N \max_{a'} Q(s_{t+N}, a'; \theta^-)$$

$$\Delta\theta \leftarrow \Delta\theta + \frac{\partial \left(y - Q(s_t, a_t; \theta)\right)^2}{\partial\theta}$$

- Variation of “Incremental multi-step Q-learning” (Peng & Williams, 1995).

Async Advantage Actor-Critic (A3C)

- The agent learns a policy and a state value function.
- Policy gradient multiplied by an estimate of the advantage. Similar to Generalized Advantage Estimation (Schulman et al, 2015).



$$\nabla_\theta \log \pi(a_t|s_t,\theta)\left(\sum_{k=0}^{N}\gamma^k r_{t+k} + \gamma^{N+1}V(s_{t+N+1}) - V(s_t)\right)$$

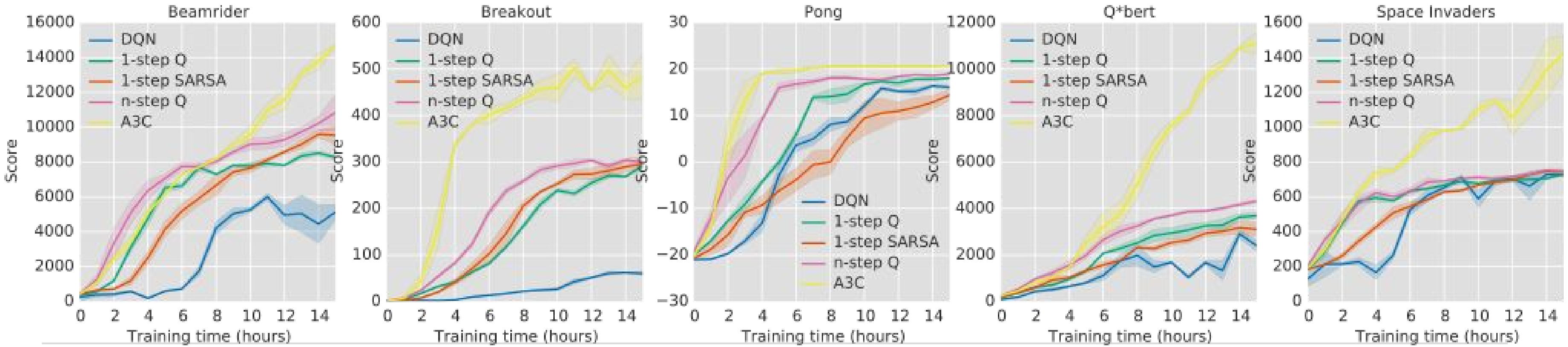
Async Advantage Actor-Critic (A3C)

Algorithm S3 Asynchronous advantage actor-critic - pseudocode for each actor-learner thread.

// Assume global shared parameter vectors θ and θ_v and global shared counter $T = 0$
// Assume thread-specific parameter vectors θ' and θ'_v
Initialize thread step counter $t \leftarrow 1$
repeat
  Reset gradients: $d\theta \leftarrow 0$ and $d\theta_v \leftarrow 0$.
  Synchronize thread-specific parameters $\theta' = \theta$ and $\theta'_v = \theta_v$
  $t_{start} = t$
  Get state s_t
  repeat
    Perform a_t according to policy $\pi(a_t|s_t;\theta')$
    Receive reward r_t and new state s_{t+1}
    $t \leftarrow t + 1$
    $T \leftarrow T + 1$
  until terminal s_t **or** $t - t_{start} == t_{max}$
  $R = \begin{cases} 0 & \text{for terminal } s_t \\ V(s_t, \theta'_v) & \text{for non-terminal } s_t \text{// Bootstrap from last state} \end{cases}$
  for $i \in \{t-1, \ldots, t_{start}\}$ **do**
    $R \leftarrow r_i + \gamma R$
    Accumulate gradients wrt θ': $d\theta \leftarrow d\theta + \nabla_{\theta'} \log \pi(a_i|s_i;\theta')(R - V(s_i;\theta'_v))$
    Accumulate gradients wrt θ'_v: $d\theta_v \leftarrow d\theta_v + \partial\left(R - V(s_i;\theta'_v)\right)^2 / \partial\theta'_v$
  end for
  Perform asynchronous update of θ using $d\theta$ and of θ_v using $d\theta_v$.
until $T > T_{max}$

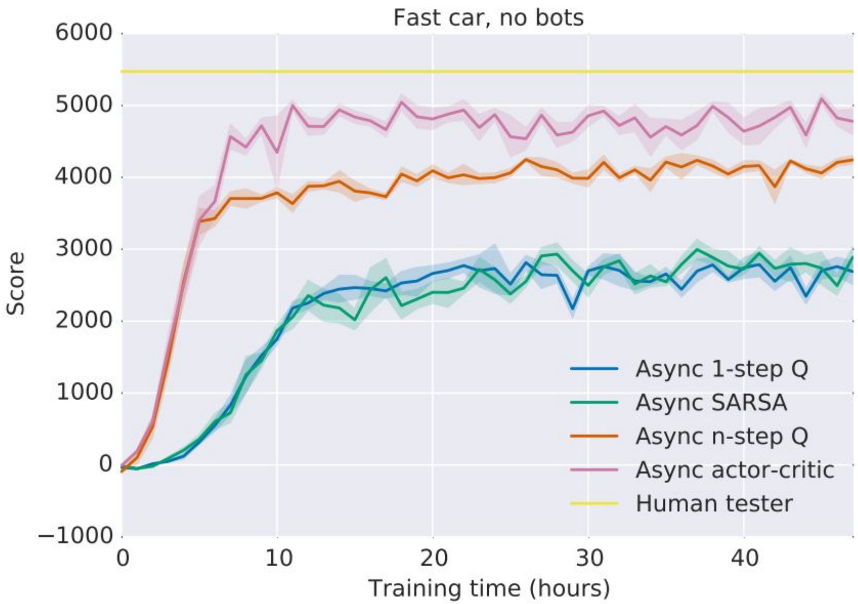
AsyncRL - Learning Speed

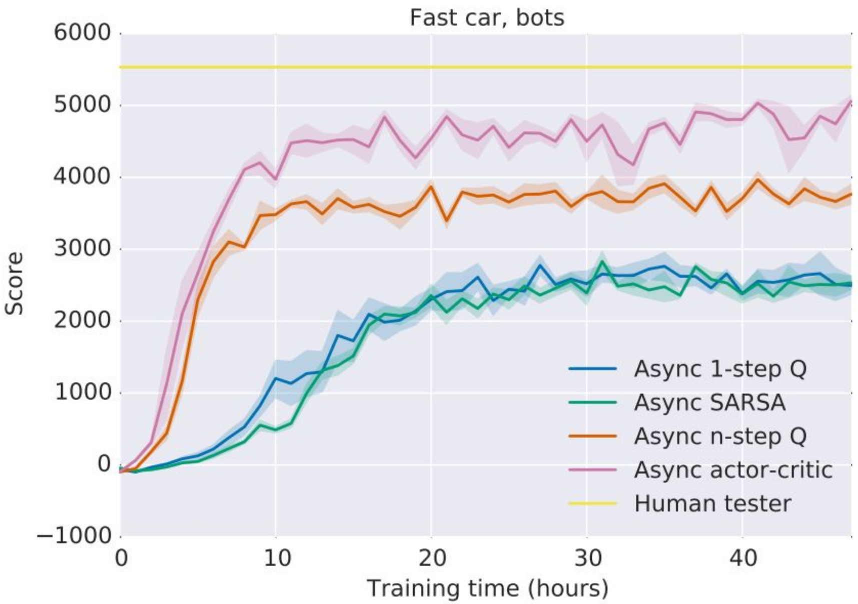
- New asynchronous methods trained on 16 CPU cores compared to DQN (blue) trained on a K40 GPU.
- n-step methods can be much faster than single step methods.
- Async advantage actor-critic tends to dominate the value-based methods.



DeepMind

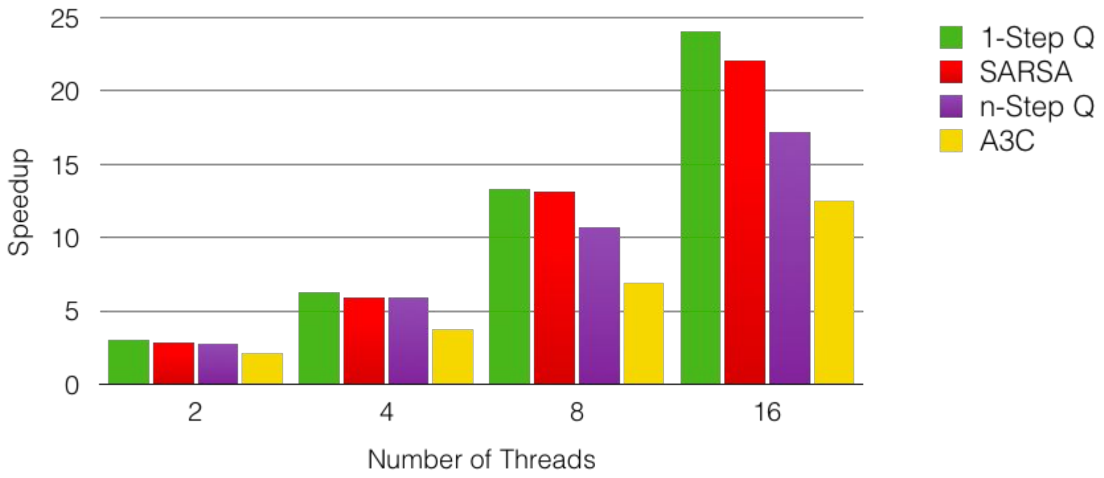
AsyncRL - TORCS





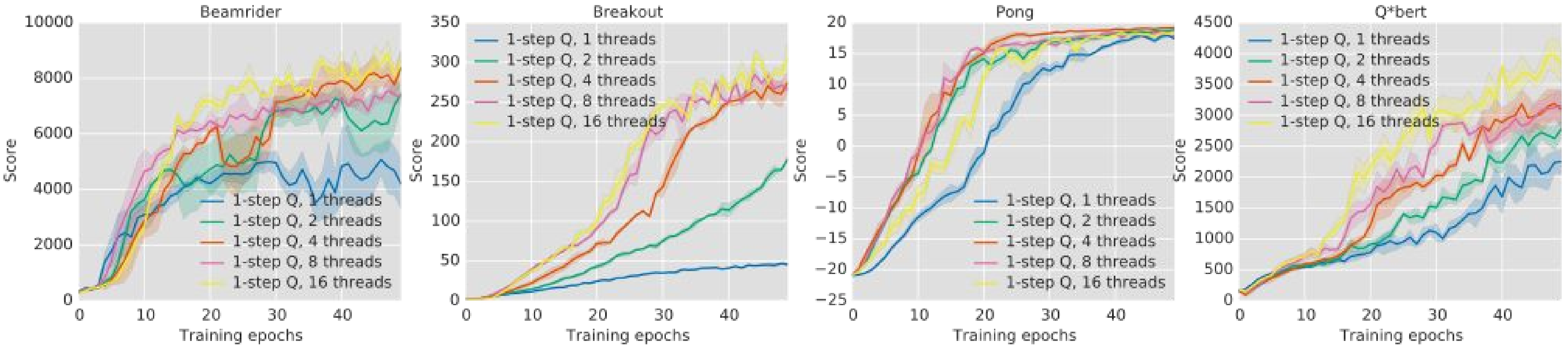
AsyncRL - Scalability

- Average speedup from using K threads to reach a reference score averaged over 7 Atari games.
- **Super-linear** speed-up for 1-step methods.



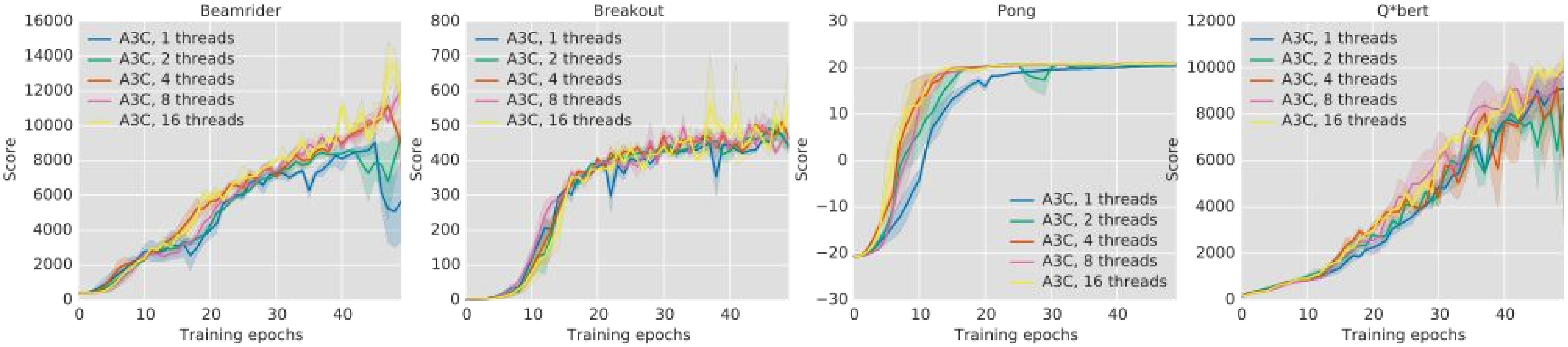
Data Efficiency of 1-Step Q-learning

- Better **data efficiency** from multiple actor-learners plus a speedup from parallel training.
  - 1 thread (blue) 16 threads (yellow)



Data Efficiency of A3C

- No data-efficiency gains. Sub-linear speedup from parallel training.
  - 1 thread (blue) 16 threads (yellow)

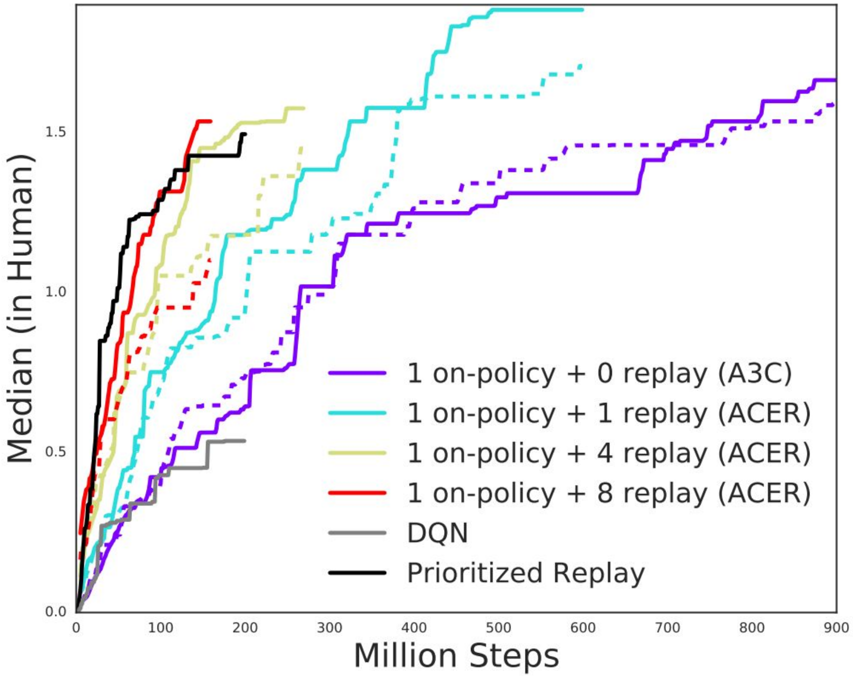


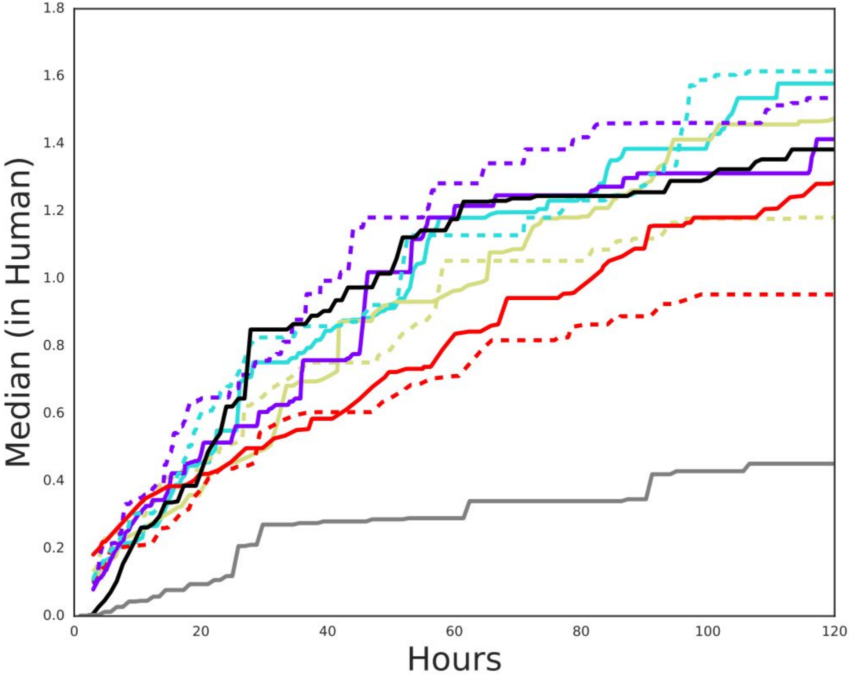
A3C - ATARI Results

Method	Training Time	Mean	Median
DQN	8 days on GPU	121.9%	47.5%
Gorilla	4 days, 100 machines	215.2%	71.3%
D-DQN	8 days on GPU	332.9%	110.9%
Dueling D-DQN	8 days on GPU	343.8%	117.1%
Prioritized DQN	8 days on GPU	463.6%	127.6%
A3C, FF	1 day on CPU	344.1%	68.2%
A3C, FF	4 days on CPU	496.8%	116.6%
A3C, LSTM	4 days on CPU	623.0%	112.6%

A3C and Data Efficiency

- Speed may not be the best metric.
- A3C is not very data efficient.
- ACER (Wang et al., 2017) and PGQ (O'Donoghue et al., 2017) combine replay with A3C for improved data efficiency.

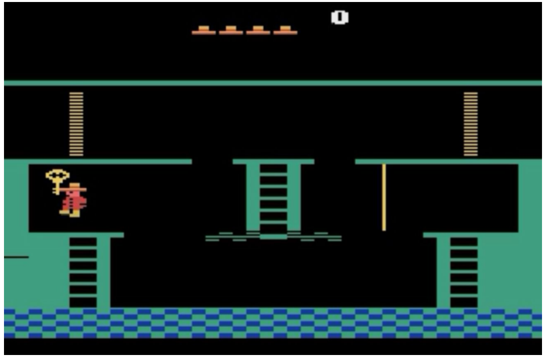


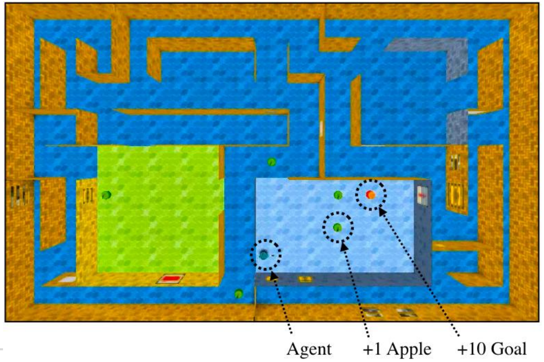


A3C - Procedural Maze Navigation in 3D

Unsupervised Reinforcement Learning

- The best deep RL methods are still very data hungry. Especially with **sparse rewards**.
- Obvious solution - Learn about the environment.
- We can augment an RL agent with **auxiliary prediction and control tasks** to improve data efficiency.
- The UNREAL agent - UNsupervised REinforcement and Auxiliary Learning.
    - “Reinforcement Learning with Unsupervised Auxiliary Tasks”, Jaderberg et al. (2017)

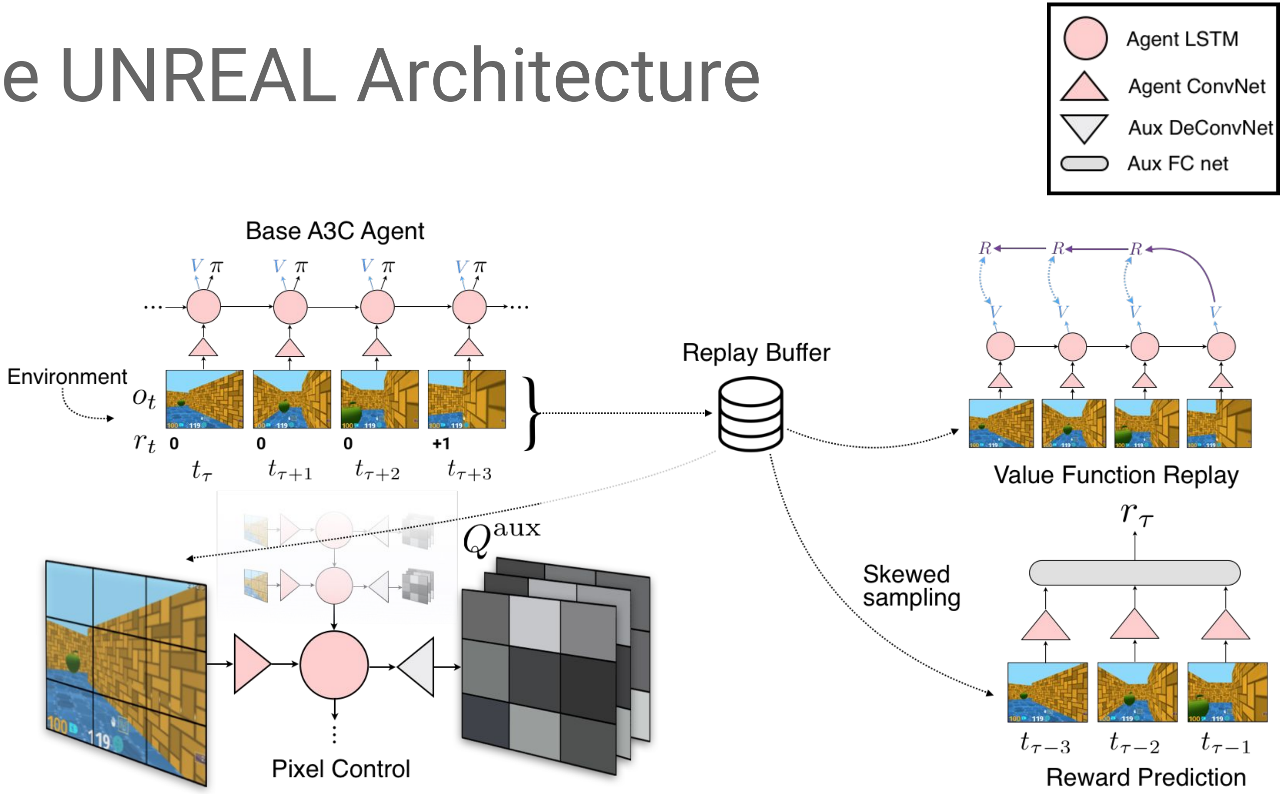




DeepMind

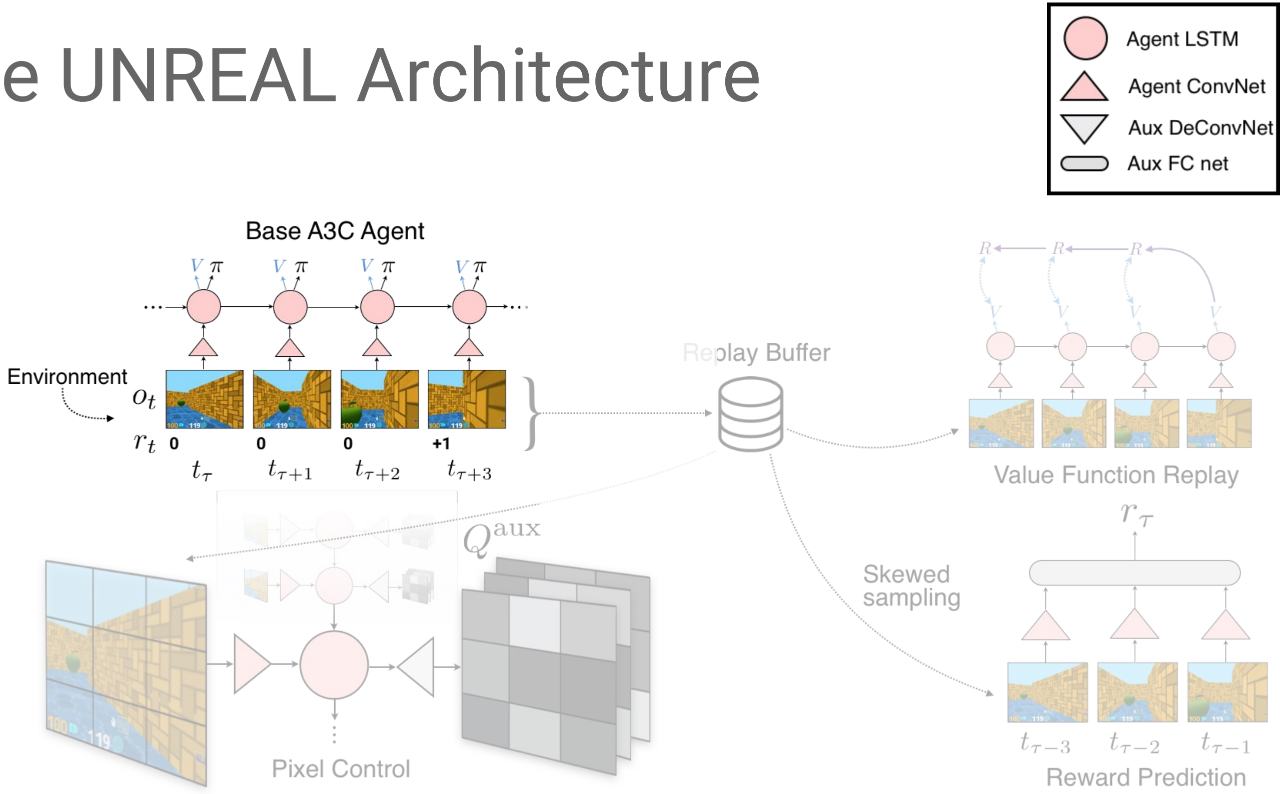
The UNREAL Architecture

- UNREAL augments an LSTM A3C agent with 3 auxiliary tasks.
- Can be used on top of DQN, DDPG, TRPO or other agents.



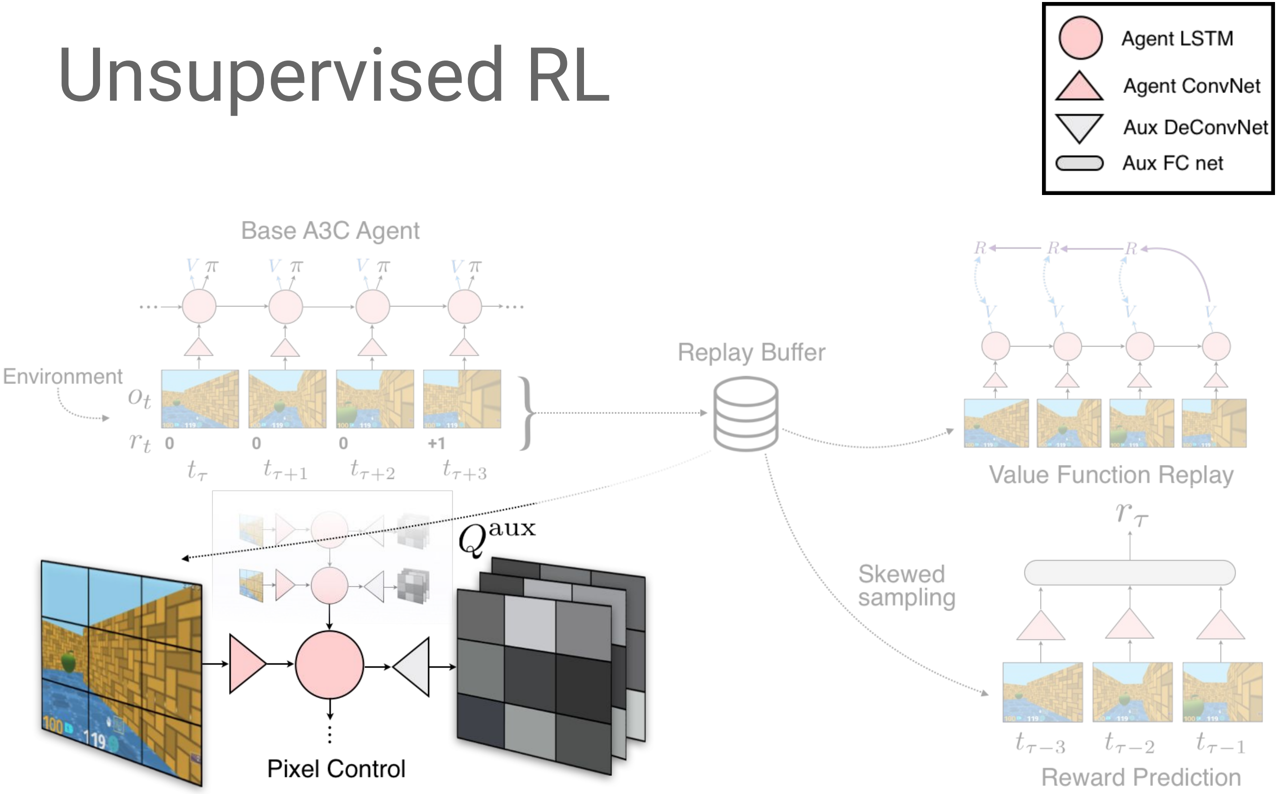
The UNREAL Architecture

- Base A3C LSTM agent learns from the environment's scalar reward signal.
- UNREAL acts using the base A3C agent's policy.



Unsupervised RL

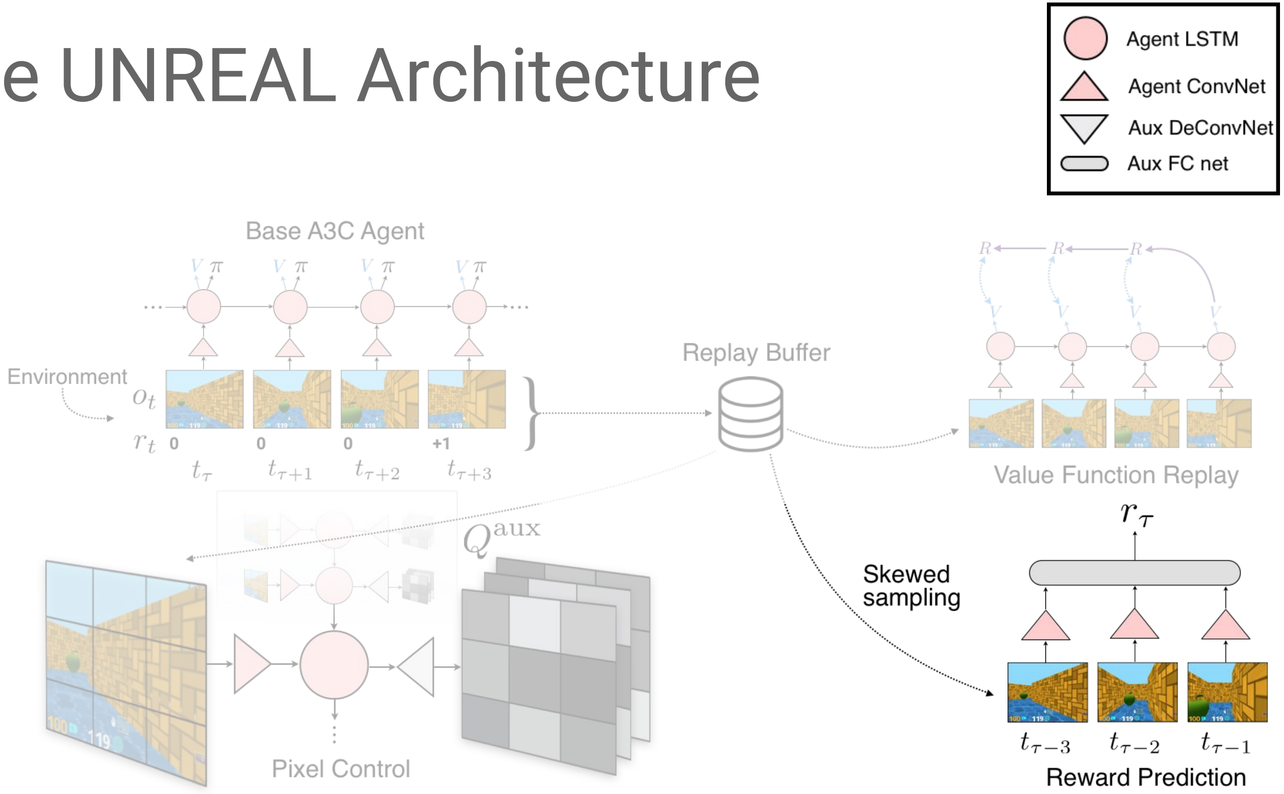
- Augment A3C with many **auxiliary control tasks**.
- Learning to control many aspects of the environment.
- Pixel control - learn to maximally change parts of the screen.
- Feature control (not used by UNREAL) - learn to control the internal representations.



The UNREAL Architecture

Focusing on rewards:

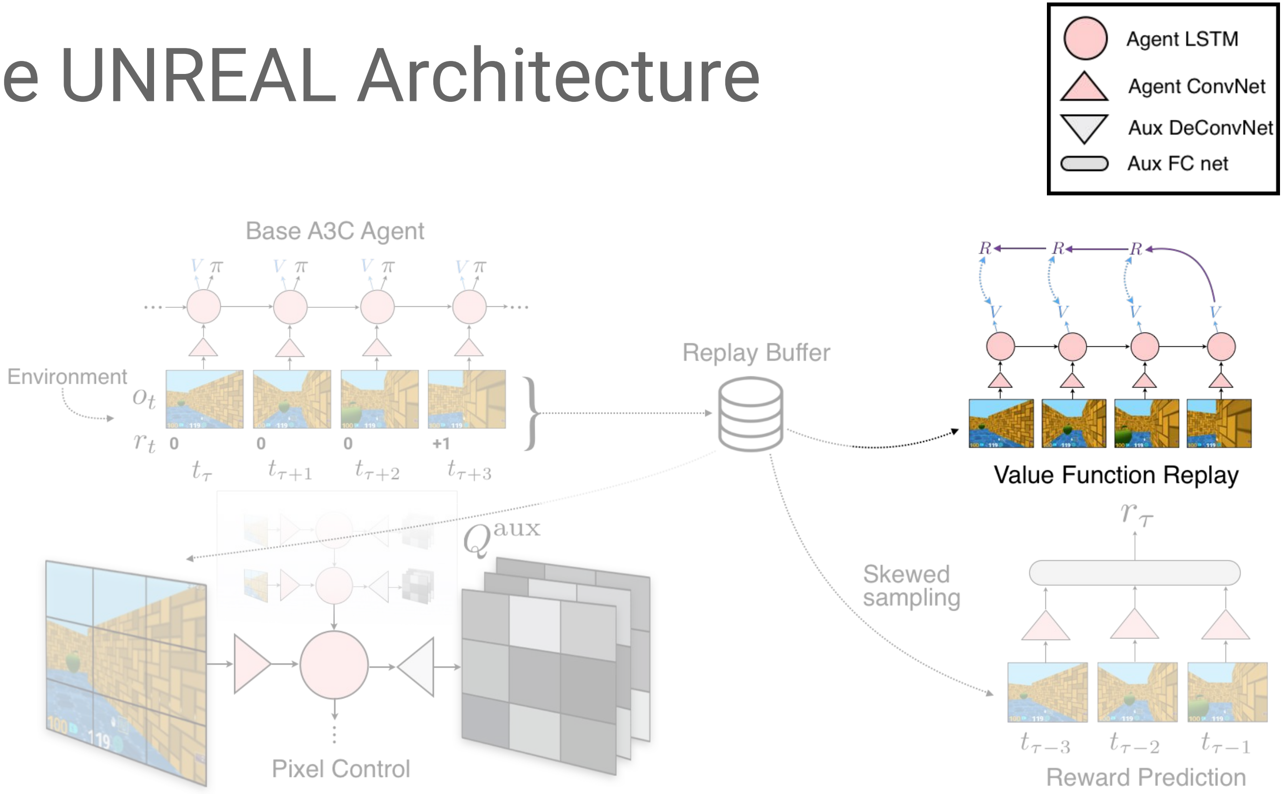
- Rebalanced reward prediction.
- Shape the agent's CNN by classifying whether a sequence of frames will lead to reward.
- No need to worry about off-policy learning.



The UNREAL Architecture

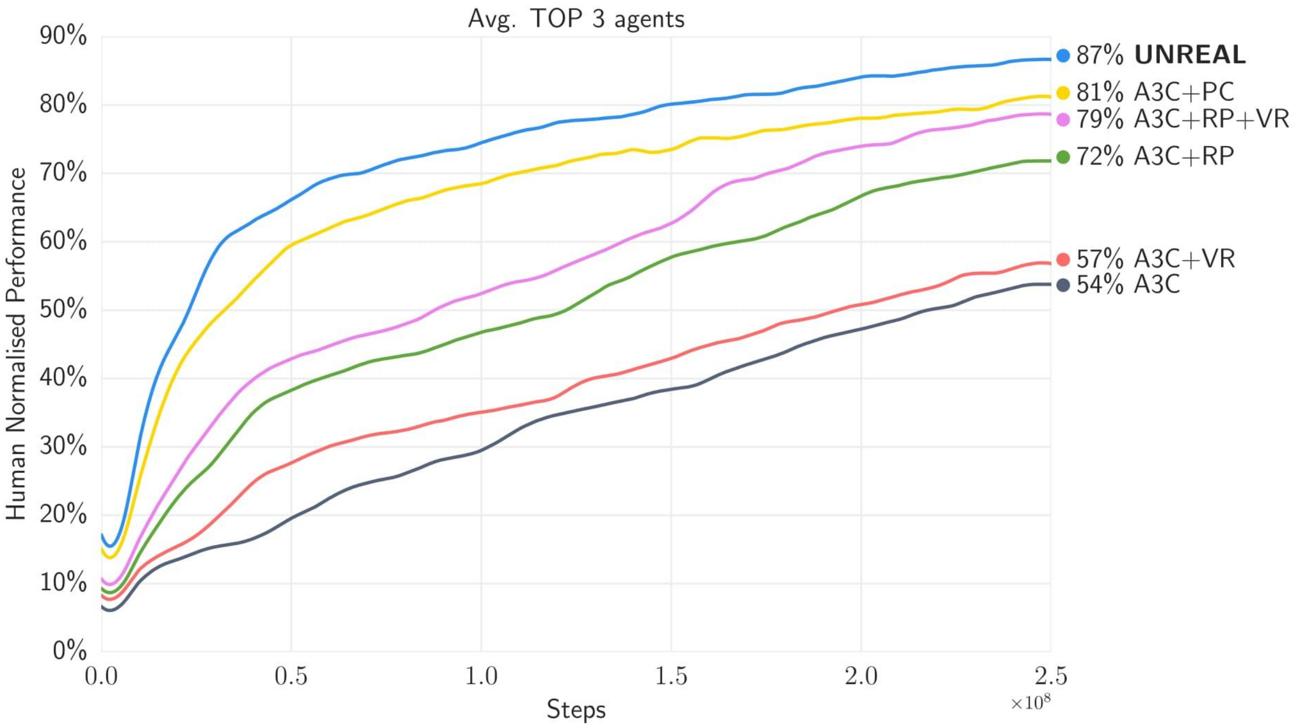
Focusing on rewards:

- Value function replay.
- Faster learning of the value function.

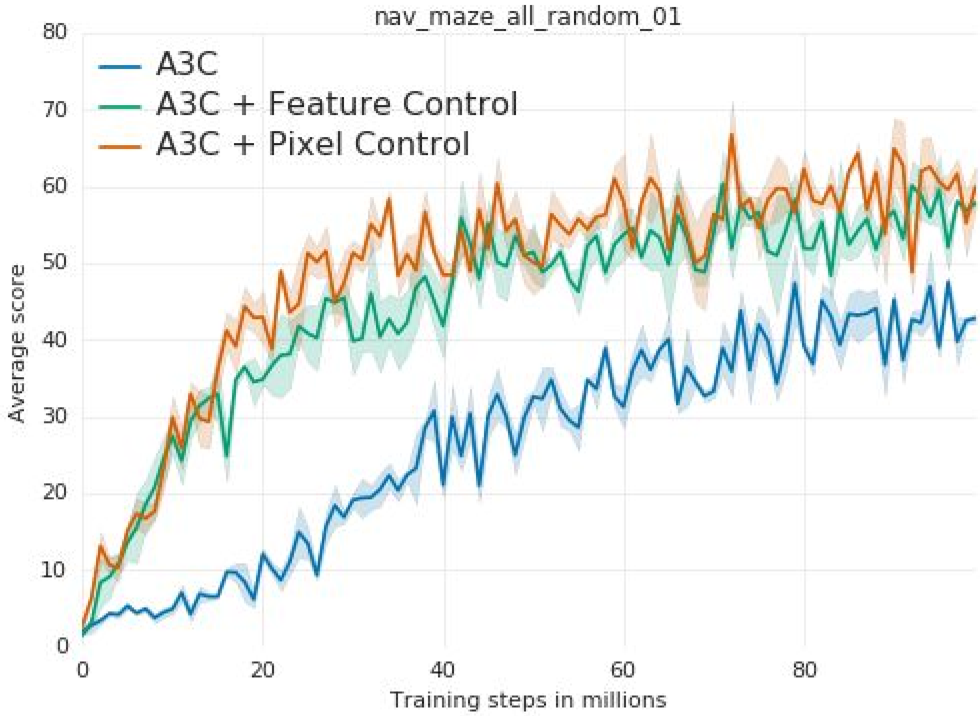


DeepMind Lab Results

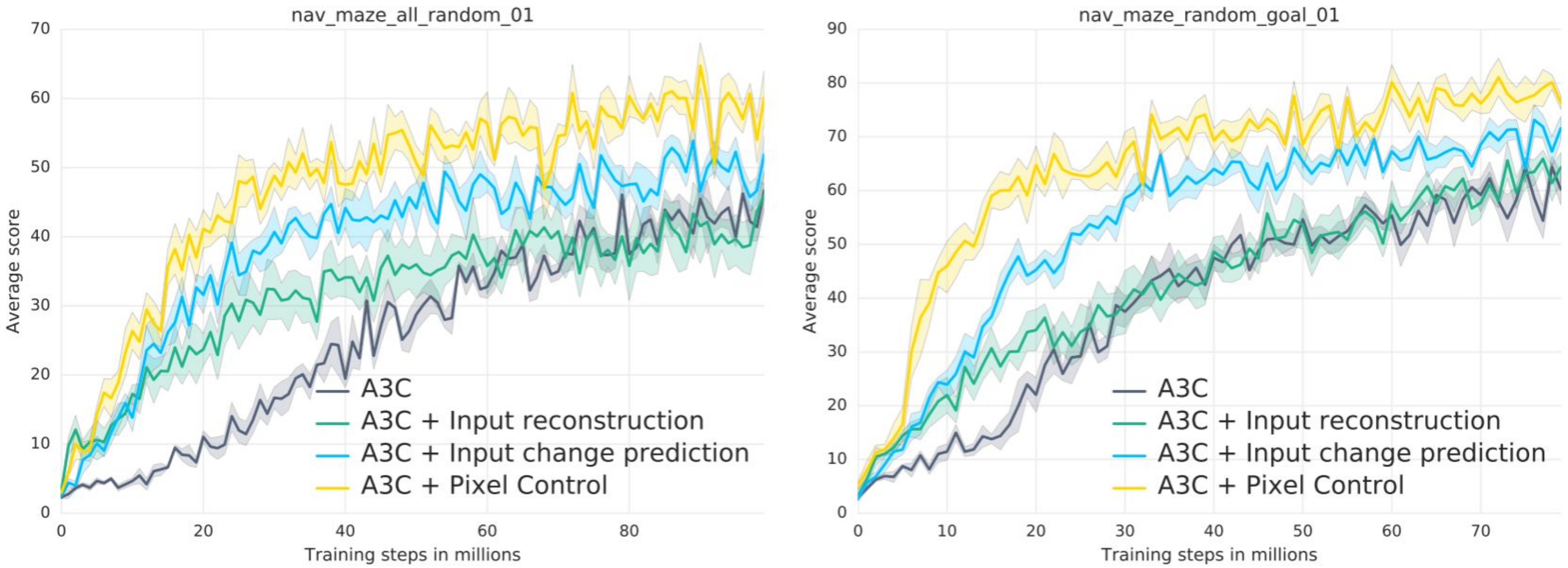
- Average human-normalized performance on 13 3D environments from DeepMind Lab.
- Tasks include random maze navigation and laser tag.
- Roughly a 10x improvement in data efficiency over A3C.
- 60% improvement in final performance.



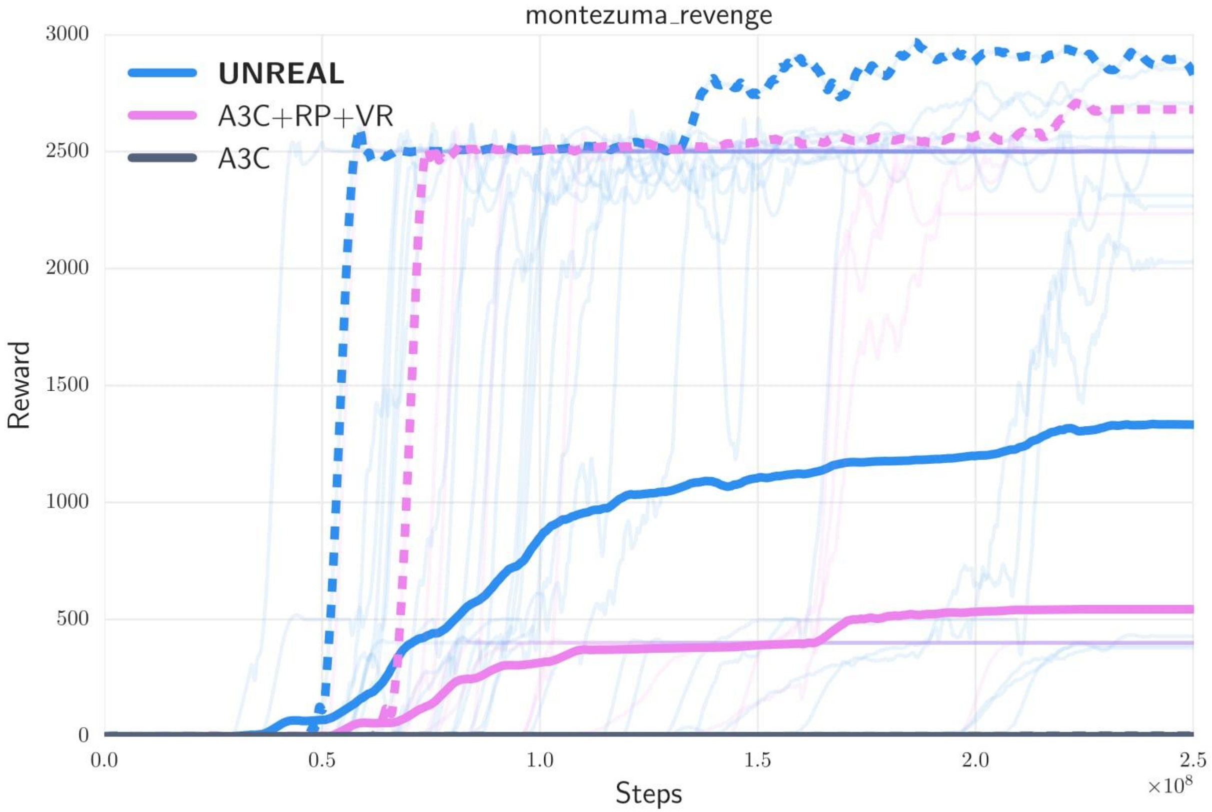
Feature Control



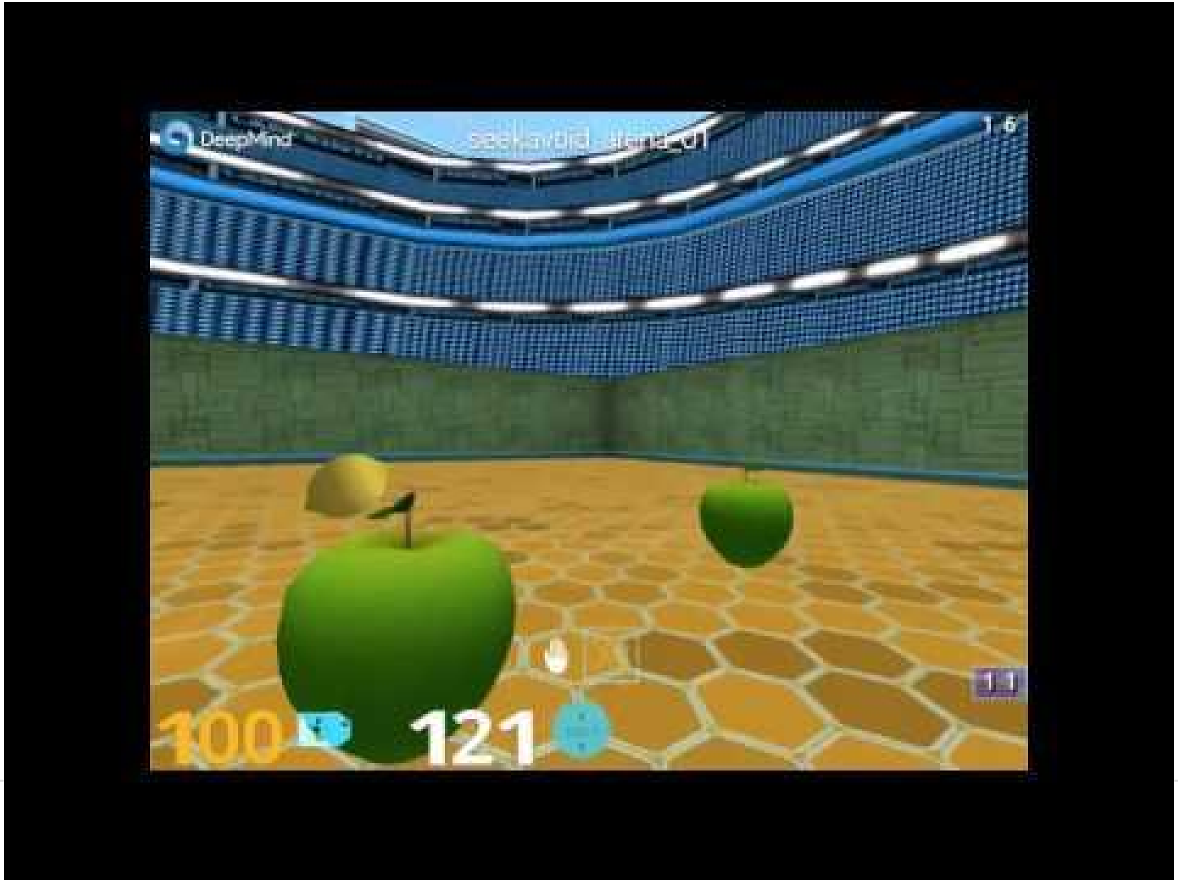
Unsupervised RL Baselines



Montezuma’s Revenge



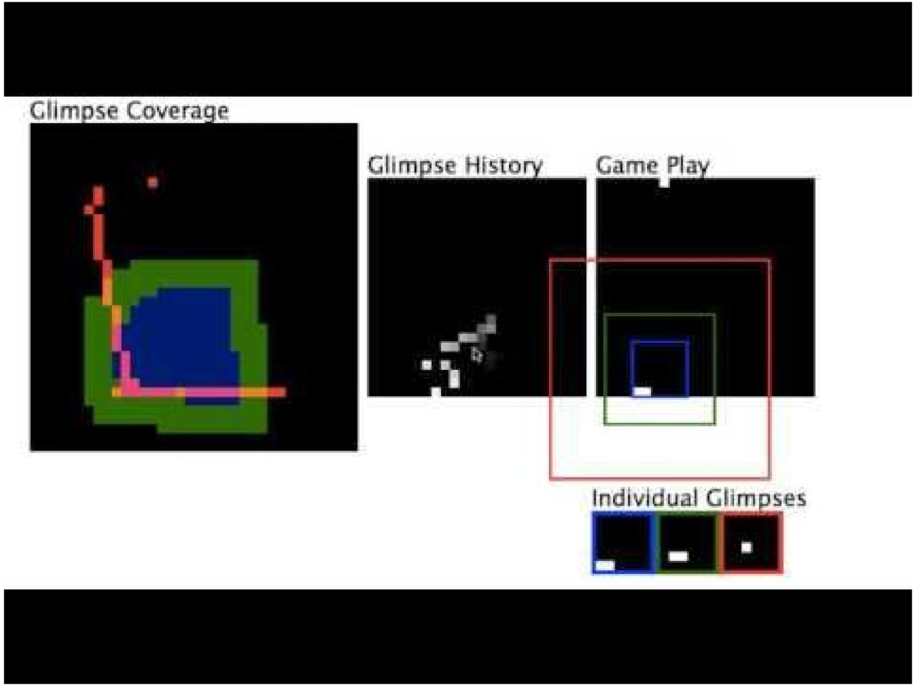
UNREAL playing DeepMind Lab



Practical Advice - Getting Started

- Start with a simple problem.
  - Something solvable in under a minute on your local machine.
  - Make it similar to the problem you really want to solve.
  - Ideally it should have knobs for controlling its difficulty.
- Plot the training curves (averaged over multiple episodes).
- Visualize the policy.
- Visualize the value function.
- Visualize everything you can think of.

The game of Catch



Practical Advice - Neural Nets

- Doing early experiments with a small network can help iterate faster.
  - This can also backfire (DQN and target networks).
- Reasonable strategy:
  - Run a few progressively larger nets to find what's sufficient for experimenting.
  - Periodically try larger nets to max out performance and verify assumptions.
- Be careful with initialization:
  - Visualize the initial policy to make sure it gets some rewards.
- Try RMSProp and/or Adam.
- Test deep learning tricks before incorporating them: dropout, batch norm, etc.
- See John Schulman's excellent guide - http://joschu.net/docs/nuts-and-bolts.pdf

Questions?