(Deep) Reinforcement Learning

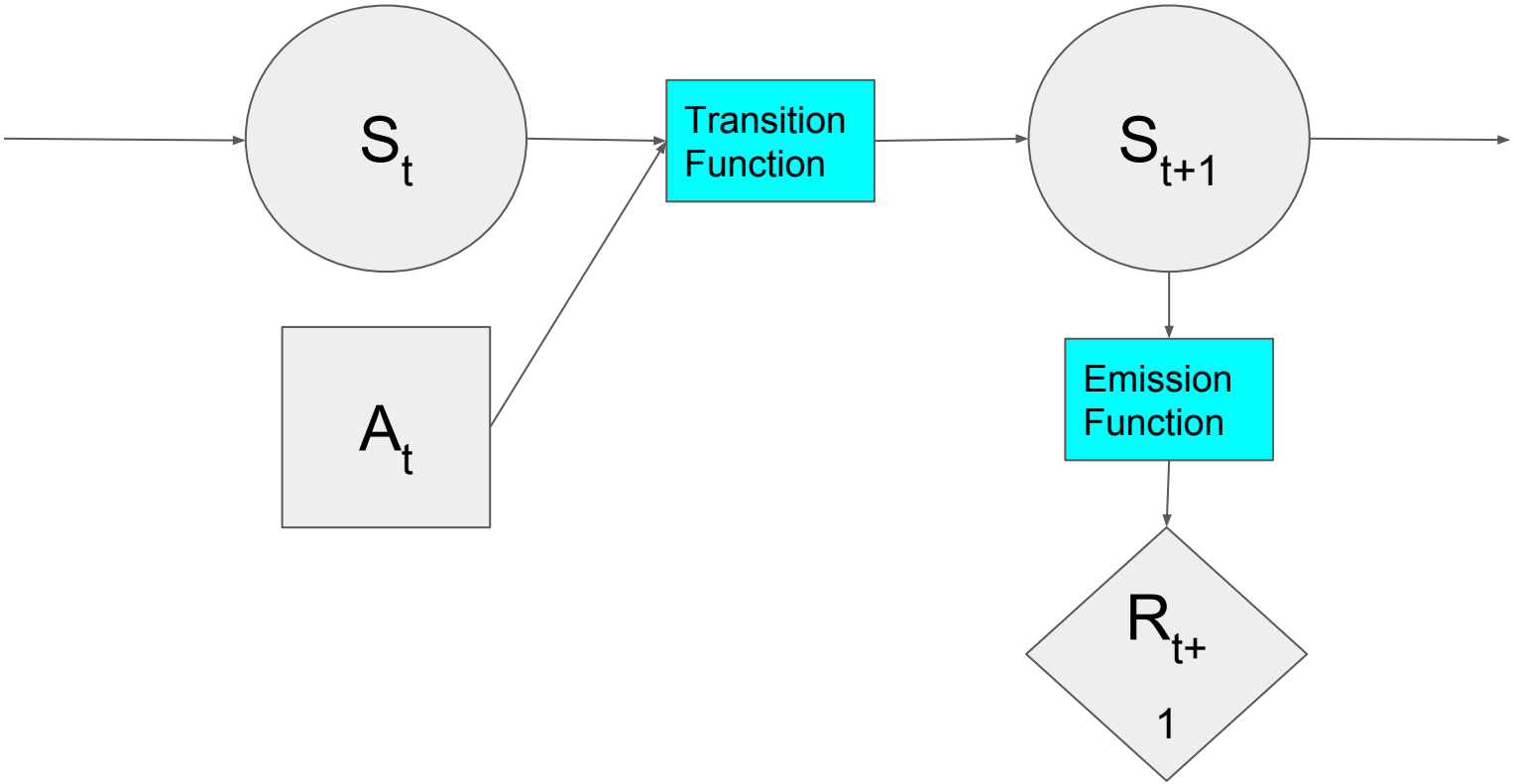
Markov Decision Processes

A MDP is:

- A discrete time dynamic system, where the state of the environment evolves over time.
- The state at time t+1 depends on the environment state and the action taken by the controller at time t.
- At each time point, a reward function gives feedback about the utility of the current state for the controller.

A control system wishes to act so as to maximize future cumulative rewards.

Markov Decision Process



Terminology

A policy, $\pi : S \rightarrow A$, is a function that specifies what action to perform in any state.

A utility function, $U : s,\pi \rightarrow \mathbb{R}$ or $U_\pi : S \rightarrow \mathbb{R}$, gives the expected future discounted cumulative rewards of performing a policy starting at the state *s*.

$$U_\pi(s) = P(s'|s,\pi(s)) \left(R(s') + \gamma\, U_\pi(s') \right)$$

- $\gamma \in [0,1]$ is the *discount* rate which tells us how much we should prioritize immediate rewards over future rewards.
- $\gamma < 1$ also forces convergence on (most) utility functions.

How should we control this system?

We want to find and implement the optimal policy, π^*, where π^* is such that for all $s \in S$, $U(s,\pi^*) \geq U(s,\pi')$

- π^* will maximize expected future cumulative discounted rewards.
- At each state, s, π^* will perform the action that maximizes

$$U_{\pi^*}(s) = P(s'|s,\pi^*(s)) \left(R(s') + \gamma U_{\pi^*}(s') \right)$$ Bellman Equation for U

How can we find or approximate π^*?

π^* will exist, but may not be unique.

Known Environment

Where we know the transition function (how the environment evolves over time based on previous state and actions) and reward function, we can learn the optimal policy by:

- Value iteration
  - Find utility values of optimal policy by iterating values of actions in states through the Bellman equations. Actions then based on the found utilities: $\pi_*(s) = \text{argmax}_{a \in A(s)} U^{\pi*}[s]$. (We perform the action which maximizes discounted cumulative rewards)
- Policy iteration
  - Find optimal policy by iteratively improving policy (simpler iteration step and terminates in fewer iterations than value iteration - effectively stops value iteration once actions for states are ranked correctly, rather than when expected discounted utility well estimated).

For a good overview, read *AI: A Modern Approach* (Hastie et al), chapter 17.

Value Iteration

repeat
  $\delta \leftarrow 0$
  for each state s **in** S **do**
    $U'[s] \leftarrow R(s) + \gamma \max_{a \in A(s)} \Sigma_{s'} P(s'|s,a)\, U[s']$
    if $|U'[s] - U[s]| > \delta$ **then** $\delta \leftarrow |U'[s] - U[s]|$
  $U \leftarrow U'$
until $\delta < \varepsilon(1-\gamma)/\gamma$
return U

Inputs:

- mdp, an MDP with discrete states S, discrete actions A(s), transition model P(s |s,a), rewards R(s), discount γ ,
- ε the maximum error allowed in the utility of any state

Local variables:

- U, U , vectors of utilities for states in S, initially zero
- δ, the maximum change in the utility of any state in an iteration

Optimal policy not dependent on optimal utility estimates…

- We just need to choose the right action, not know exactly how good different actions are. Once we have ranking of action utilities correct, we are fine.

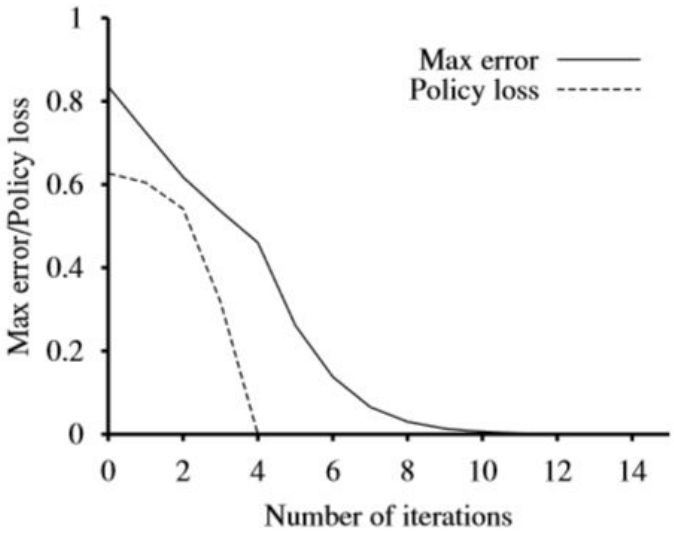


Figure 17.6 The maximum error $||U_i - U||$ of the utility estimates and the policy loss $||U^{\pi_i} - U||$, as a function of the number of iterations of value iteration.

Policy Iteration

Alternate the following two steps, beginning from some arbitrary initial policy π_0:

1. Policy evaluation

   Given policy π_i, calculate the utility of each state if π_i were to be executed using: $U_i(s)=R(s)+\gamma \Sigma_{s'} P(s'|s,\pi_i(s)) U_i(s')$

2. Policy improvement

   Calculate a new policy π_{i+1}, using: $\pi_{i+1}(s) = \text{argmax}_{a \in A(s)} \Sigma_{s'} P(s'|s,a) U_i[s']$.

The algorithm terminates when the policy improvement step yields no change in the utilities.

Unknown Environment

What if we know do not know the transition and reward functions?

- Model based methods: Proceed to model the transition and reward functions from data we collect. Estimate optimal policy based on these models.
- Model free methods: Directly model expected reward of actions given states.

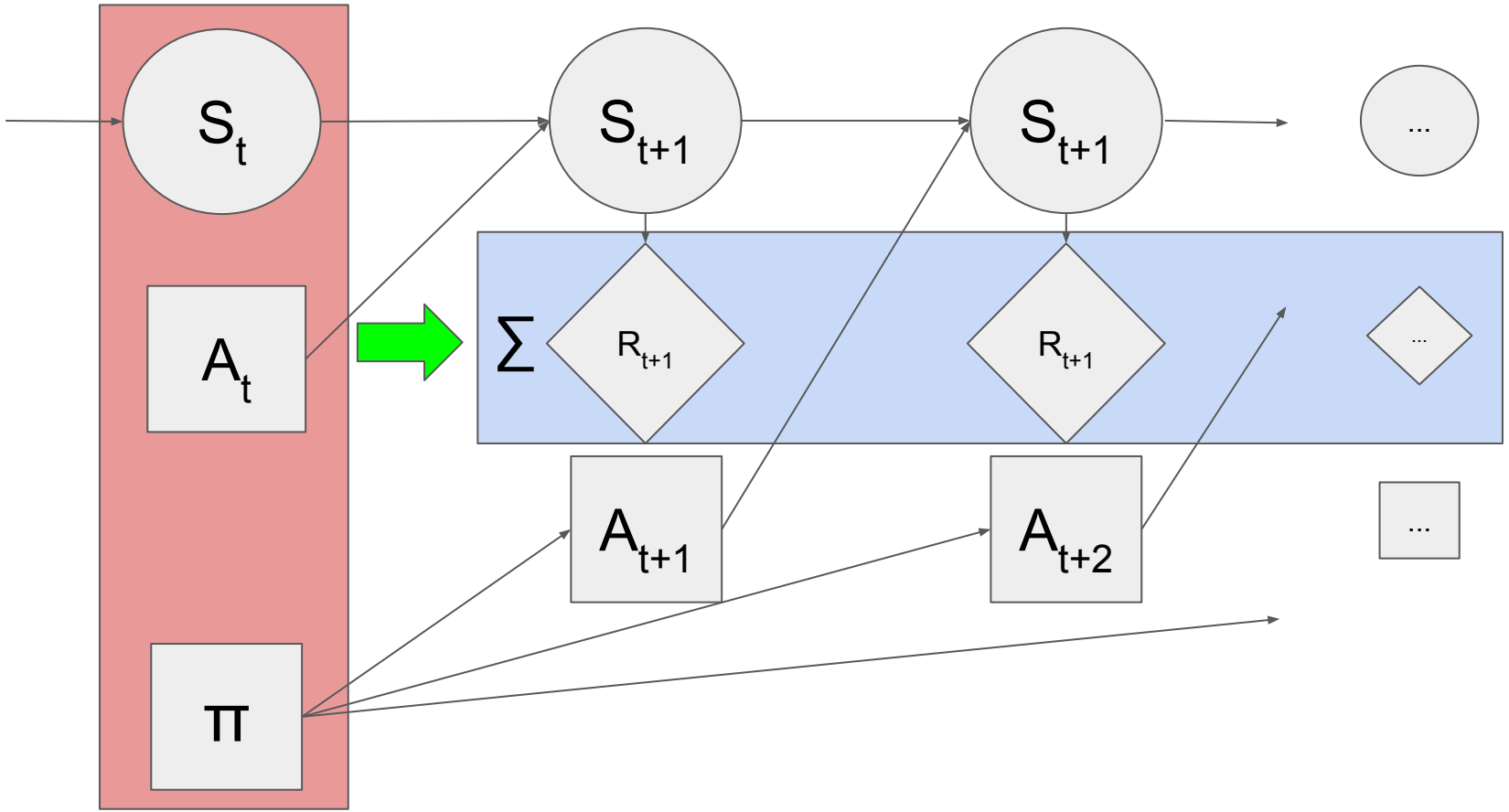
Model Based Methods

We could attempt, via exploring the effects of actions in different states, to learn via standard statistical (supervised learning) methods a model for the environment (transition and reward functions).

Then we can use value or policy iteration to work out the best policy given our current estimation of the environment.

We will consider alternatives that avoids explicitly modeling the environment. Instead we seek to model the Q-function that directly relates expected cumulative rewards to state/action pairs given a particular policy.

The Q_π-Function



Fitting Q to a fixed policy π

Assume our environment has only finitely many states before a terminal state.

We can perform ‘runs’ in this environment acting according to π. At the end of a run, for each state, action pair $<s,a>$ encountered, we have a sample of the utility of performing that a in s and than performing actions as dictated by π thereafter. This is labelled data that can be used to train Q.

BUT: We would only ever obtain samples for actions that our policy tells us to perform. Q will not converge to Q_π.

Fitting Q to a fixed policy π: ε-learning

We can act according to a sequence of randomized π policies, π_{R1}, π_{R2}, … such that that $\pi_{Rn} \rightarrow \pi$ as $n \rightarrow \infty$.

- ε-learning: follow π with probability 1-ε, otherwise perform random. $\varepsilon \rightarrow 0$ as $n \rightarrow \infty$.

In this case:

- We obtain samples from actions not specified by π
- Q will converge to π

Fitting Q to an evolving policy

We do not know the optimal policy.

- We must try to learn it at the same time as we learn Q.
- We work with the evolving policy, π_Q, which is based on an evolving Q model: we pick the best actions as determined by Q. So we update the policy based on the evolving Q model.
- Our Q model is now trying to converge to a target that is itself moving *based on the evolution in the Q model*.

Problem:

- Instability.

Temporal Difference Q-Learning

Our naive approach is to obtain labelled training data for $\hat{Q}$ after a complete run, based on the results of that run. This ignores the relationship between states given in the Bellman equations.

- We lose significant information ignoring this constraint.

Using the Bellman equations we can get information we can use to update $\hat{Q}$ immediately after an action, since the following holds:

$$Q(s,a) = E_{s'}[\, R(s') + \gamma \max_{a} Q(s',a) \,]$$

Temporal Difference Q-Learning

$\hat{q}=\hat{Q}(s,a)$

$q=Q(s,a)$

$\tilde{q}=R(s') + \gamma \max_a Q(s',a)$

We would *like* to train $\hat{Q}$ on the loss between the model estimates, $\hat{q}$, and the true Q values, q. But we cannot, since we do not know the latter.

We can train our model on the loss between $\hat{q}$, the model estimates, and $\tilde{q}$, the estimates of the true Q values that include some objective ground truth (the known new state transitioned to, and the reward obtained at that new state) and an estimate from our Q model (the estimated utility of continuing on from that new state).

SARS´ Learning

Whenever we make action *a* in state *s*, we arrive at a new state, s´. Essentially s´ is a sample the next state we can expect to arrive at when performing action *a* in state *s*. (Over time the new states ‘sampled’ will converge to expectation.) We learn the reward of arriving at that state, $r=R(s')$, and can estimate the expected future utility of being in that state from the $\hat{Q}$ model, $\gamma \max_a \hat{Q}(s',a)$.

With this 4-tuple (s,a,r,s´) we can calculate $\hat{q}$ and $\tilde{q}$ and fit $\hat{Q}$ using TD Q-learning.

DQNs

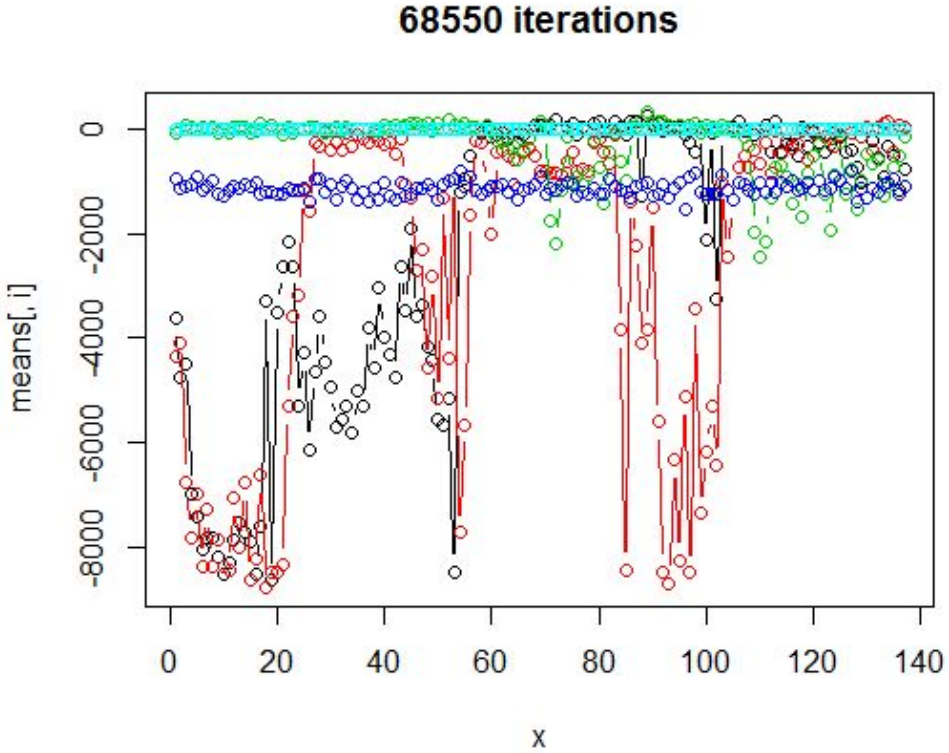
A deep Q-Network model is just Q-Learning with Q being a deep ANN.

- Lots of showpiece image based DRL systems, such as playing Atari games. The deep ANN is here a CNN based model.
- AlphaGo was a DRL system.

Note that they are difficult to work with.

- Convergence is not guaranteed - local minima exist on the loss surface, even ignoring the fact we are chasing a moving target! Indeed *divergence* is possible :(
- Common to need lots of data.
- Instability in learning is common.

Instability in Learning



Experiential Replay

We can update Q whenever we get a new s, a, r, s´ 4-tuple.

- Since the updates come from consecutive states played by the agent, the updates are correlated, which can lead to instability as the model over-learns how deal with whatever sub-space of the environment it's currently in, forgetting how to deal with other sub-spaces.

IDEA: Store encountered sars´ 4-tuples in memory and sample a batch of them after each action.

- Data used in fitting (much) less correlated.
- Lots more data used to learn. But same data reused a lot too (perhaps more instability in convergence and overfitting issues).
- Note that we do not store the estimated $\tilde{q}$ values. We recalculate them each time (ideally our estimates of the ground truth will improve as Q improves).

Preferential Experiential Replay

We learn more from *unexpected* data. I.e. from data which our model gets badly wrong.

Instead of just sampling a batch of memories. Let’s give preference to ones that where the error was large.

- When sampling the experiential replay batch, give weights based on loss function last time the sars´ 4-tuple was used.

Double Deep Q Networks (DDQNs)

Instability can really damage the learning process.

IDEA: Use two DQNs, Q and Q'. One decides the action to be taken, the other estimates the $\tilde{q}$ values.

$$\tilde{q} = R(s') + \gamma\, Q'(s', \text{argmax}_a Q(s', a)) \,]$$

Q is updated every step. Q' is periodically updated with a copy of Q's weights.

Why: Reduce the 'tail chasing' aspects of instability.

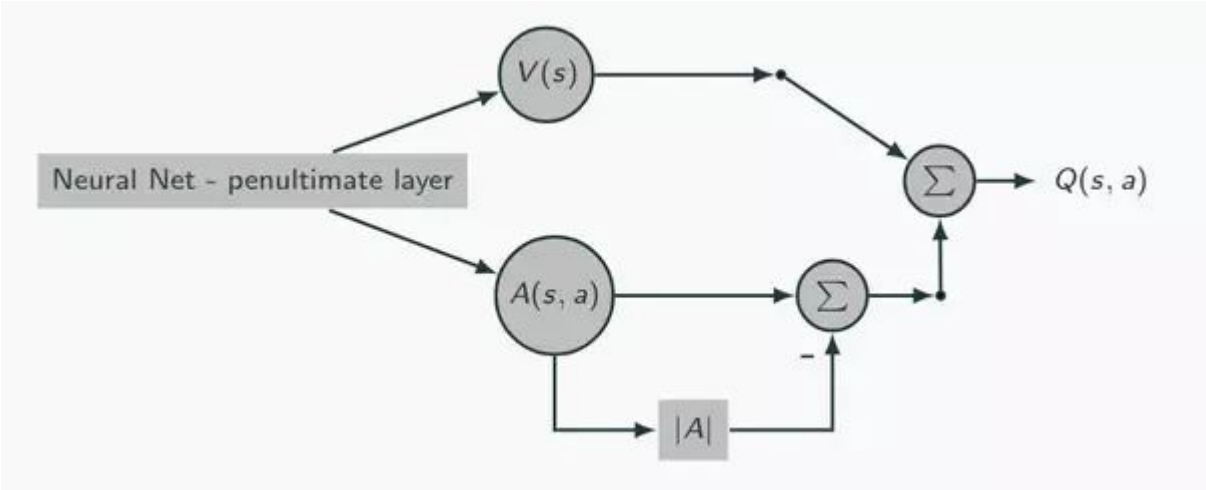
- Give Q a period to converge to a stationary target and propagate stationary information through the Bellman equation.

Dueling DQNs
(Advantage DQNs)

Separate the Q function into V and A components:

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

- V is the Value of the state
  - Really just the mean of the Q-values for the actions.
- A is the Advantage of taking an action in that state



Dueling DQNs
(Advantage DQNs)

The purpose of the Q-value estimates in a DQN is twofold:

1. Deciding how to act
   - We want to act so as to maximize Q-values (expected future utility)
2. Propagating information backwards through the Bellman equation
   - We need to have good estimates about the value to us of states that actions take us to in order to know which action to take.

Intuitively dueling DQNs can help in both of these.

Dueling DQNs
(Advantage DQNs)

Advantage One: When deciding how to act concentrate on picking the best action

- What we want to know is the best action to take.
- Certain actions will be better than others over a wide range of similar states.
- The differences in the Q-values for actions at these different states, because some states are better or worse, is not very important to the decision about which action to take.

Dueling DQNs
(Advantage DQNs)

Advantage Two: Update information about Q-values common to all actions in a state regardless of action performed.

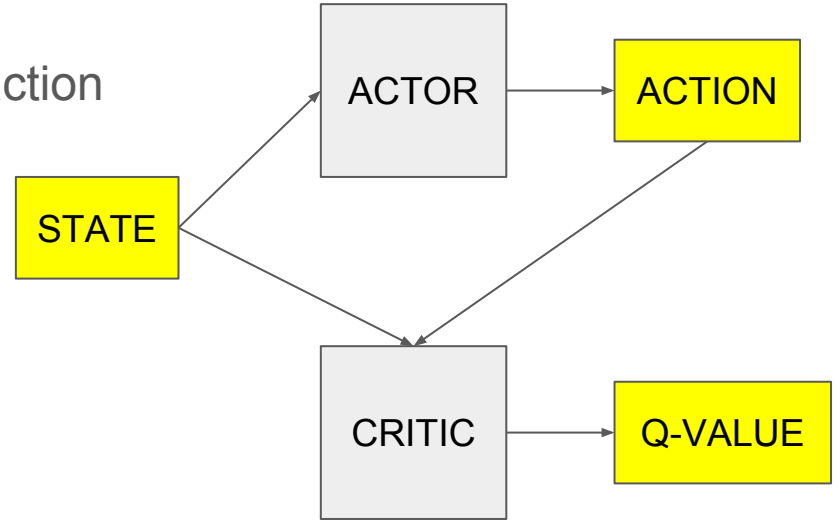
- DQNs only directly compare $\hat{q}=Q(s,a)$ to $\tilde{q}$.
- But the value component of the q-value is of importance to *any* actions taken in *s* (not just *a*).
- In a Dueling DQNs, parameters responsible for generating this value component of $\hat{q}$ are given a (pseudo-) training case whatever action is taken, and this should speed up their fitting and hence improve the estimates of q-values for *all* actions in this state.

Actor-Critic Models

Previous DQN based algorithms will not work where we have continuous actions. Actor-Critic algorithms will.

Actor: Outputs action based on state

Critic: Outputs Q based on state and action



Training an Actor-Critic model

Training the Critic

- The critic is simply a normal DQN (or Dueling DQN) model.
  - It is just the actions taken are not determined by maximizing the action argument in the DQN type model, but rather by the independent Actor model.

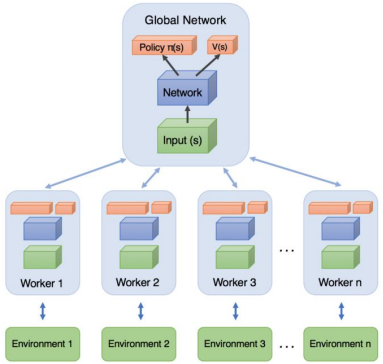
Training the Actor

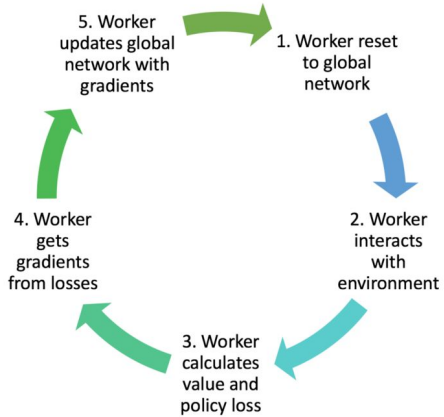
- We want to determine what changes in parameters in the Actor model will result in the largest increase in the Q-values estimated by the Critic model.
  - Analogous to training the generator network in a GAN

Asynchronous Advantage Actor-Critic (A3C) Networks

Asynchronous Advantage Actor-Critic (A3C) Networks

- Use Actor-Critic models with dueling DQN critic model.
- Global network estimates V and A values
- Multiple workers learn and update global network
  - Instead of experiential replay we just run multiple agents simultaneously to gather multiple uncorrelated cases for each training step.
  - In the paper that introduced this, each agent does its own training, and passes the gradients to the global network. This is for asynchronicity and hence speed.





Requirements

1. Ability to control the system *badly* for a long period of time.
   - Often not reasonable, so we need a simulator to learn reasonable control of an approximation of the system.
   - Fine tuning may be permitted when we move from the simulator to the true system. But real care needed to avoid instability based loss of performance during this fine-tuning.
2. Time and computational resources
   - Running *many* runs for each RL system training attempt.
   - Running many training attempts to tune hyper-parameters and model architecture.

Basic Advice

1. Choosing the environment statespace.
    a. You may want to work with video inputs - then just use CNN based models with the appropriate input dimensions. And expect to need to train over *many* runs!
    b. With structure data, try and make it as simple as possible!
2. Choosing the model structure and characteristics
    a. Start simple and work up. Try a linear model if you are using structured data, at least as a (more stable) baseline.
3. Evaluate performance
    a. If you can, have a high quality rule-based system for comparison. If you can’t, have a random action controller for comparison.

The Singularity Remains Distant

Long way from general AI

- To much required of individual agents
- Schematic transfer learning?