

Deep Reinforcement Learning - 4

Kalle Prorok, Umeå 2020

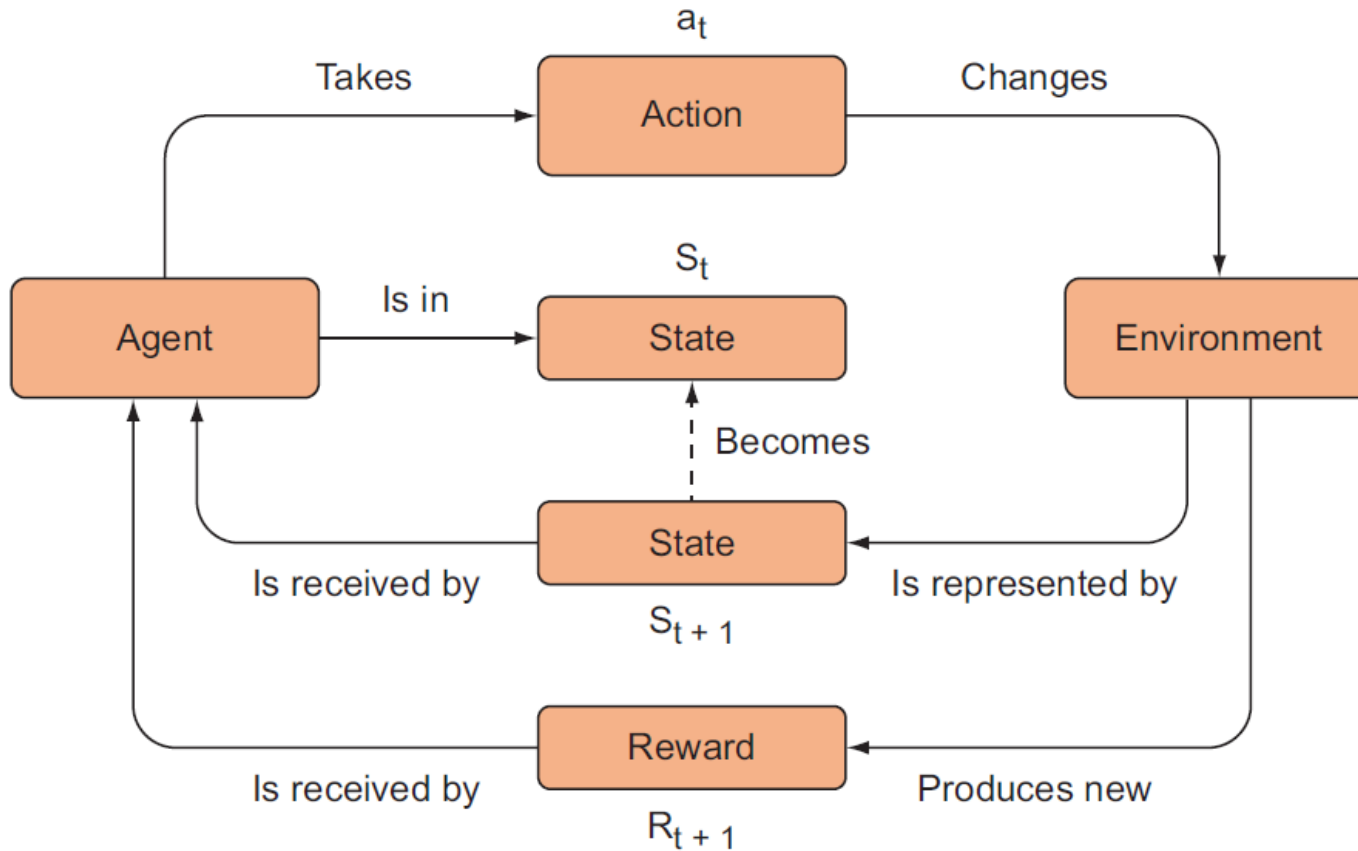
Parts based on the Manning book

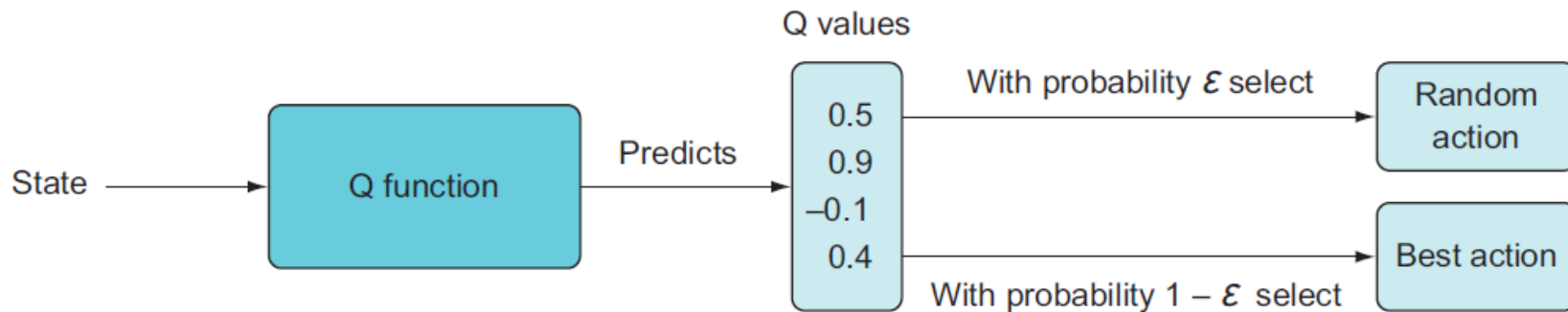
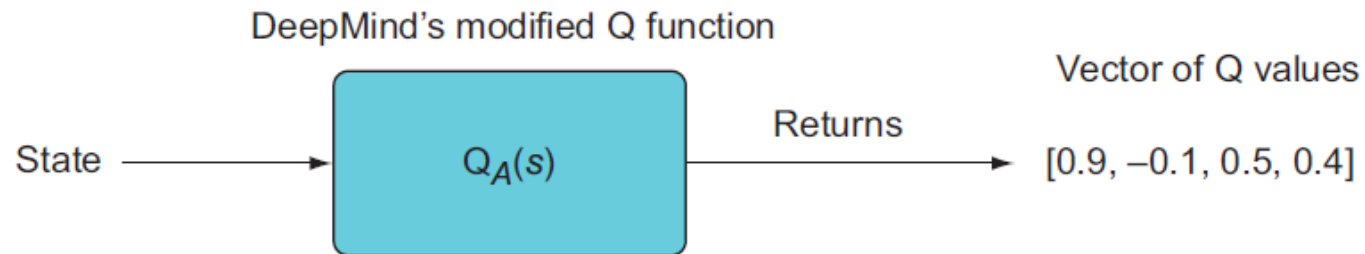
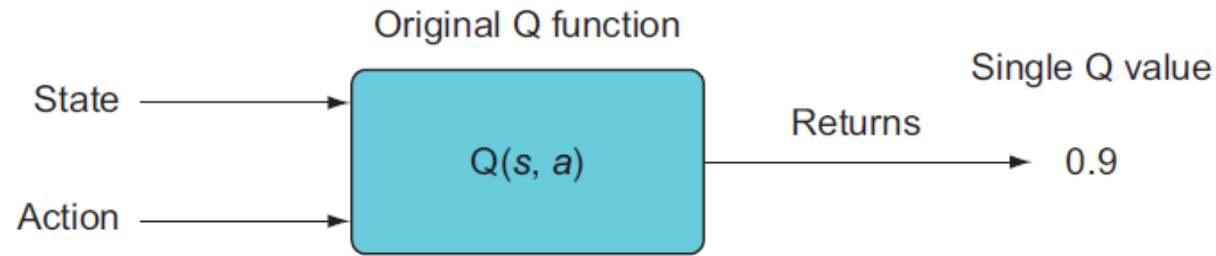
Deep Reinforcement Learning in Action

By Alexander Zai and Brandon Brown

And the Sutton-Barto book

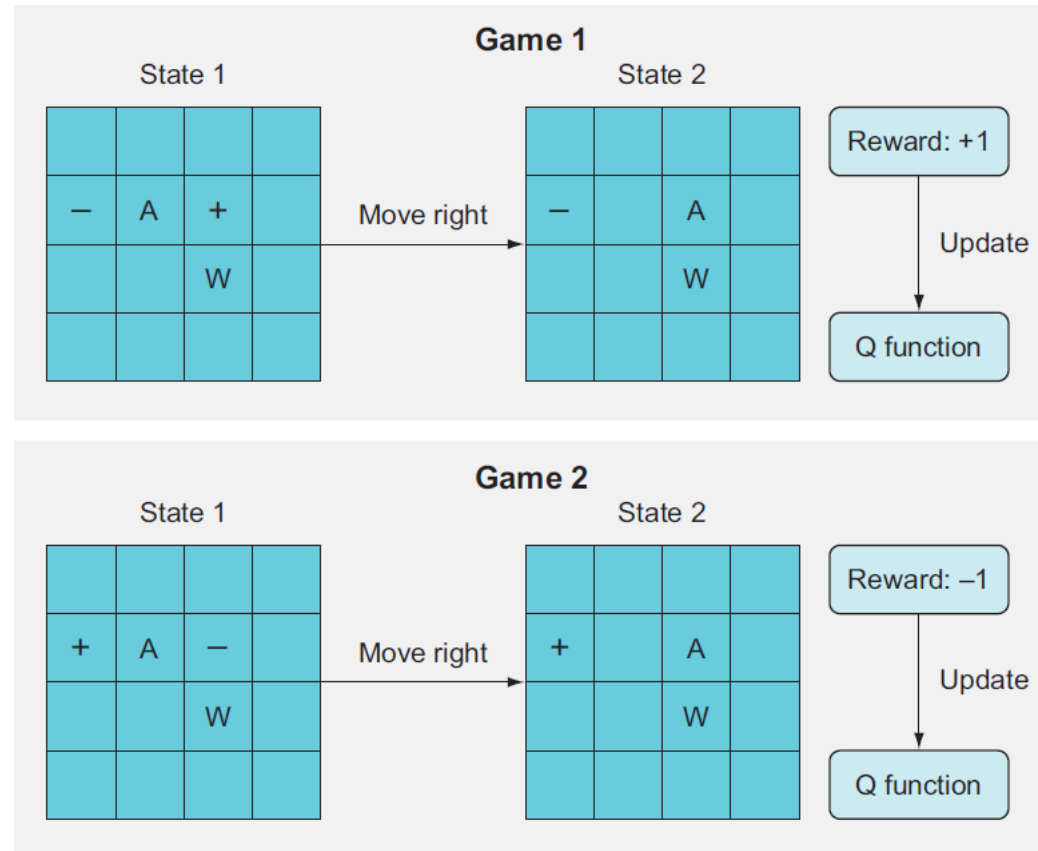
Agent – Action – Environment – State/Reward





DeepMind used a modified vector-valued Q function that accepts a state and returns a vector of state-action values, one for each possible action given the input state. The vector-valued Q function is more efficient, since you only need to compute the function once for all the actions

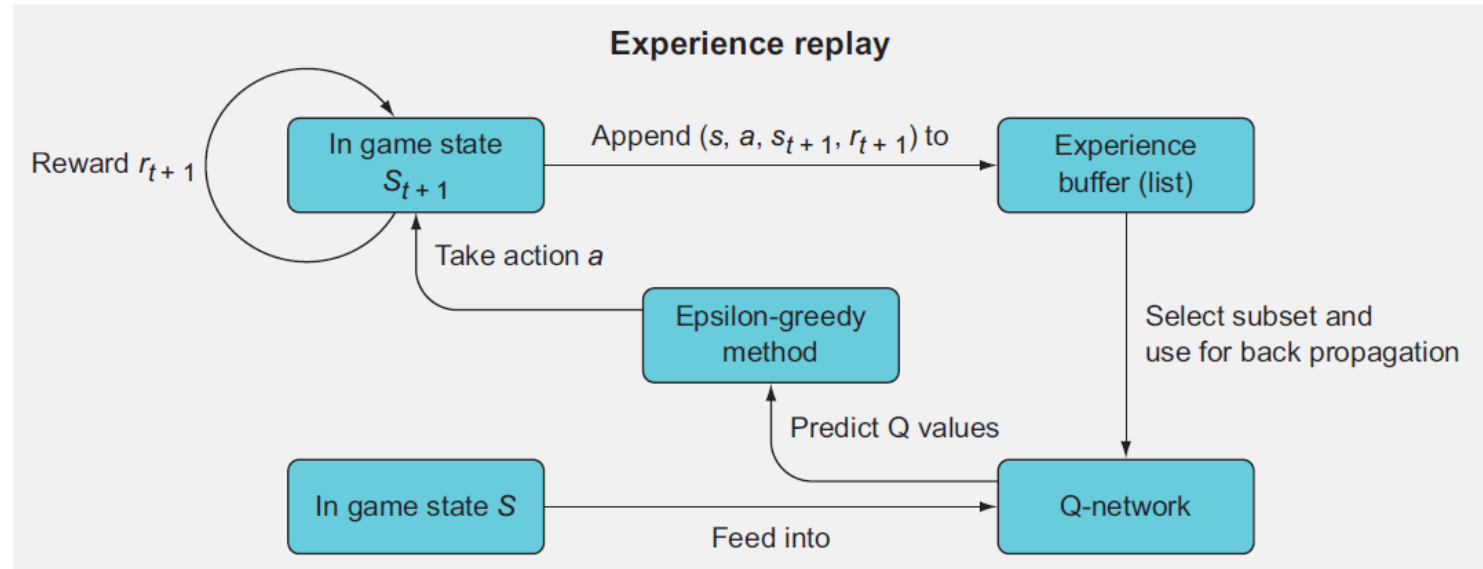
Catastrophic forgetting



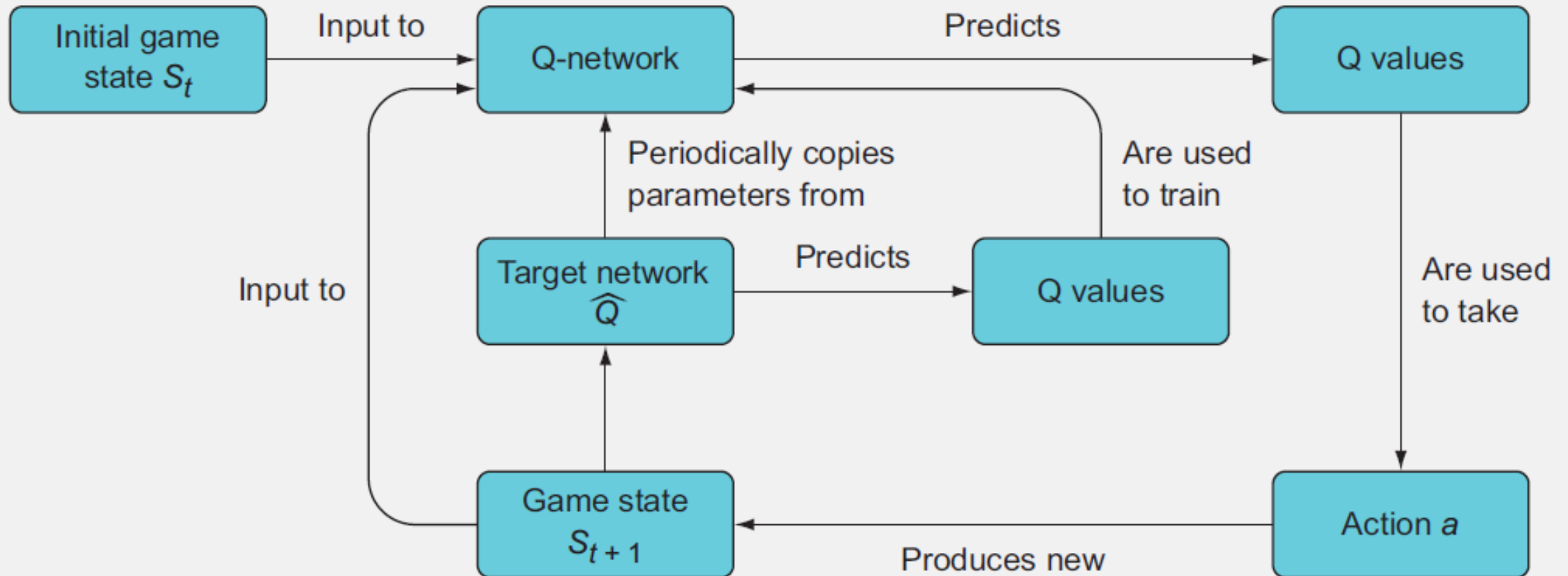
two game states are very similar and yet lead to very different outcomes, the Q function will get “confused” and won’t be able to learn what to do

Preventing catastrophic forgetting: Experience replay

- employ mini-batching by storing past experiences and then using a random subset of these experiences to update the Q-network, rather than using just the single most recent experience

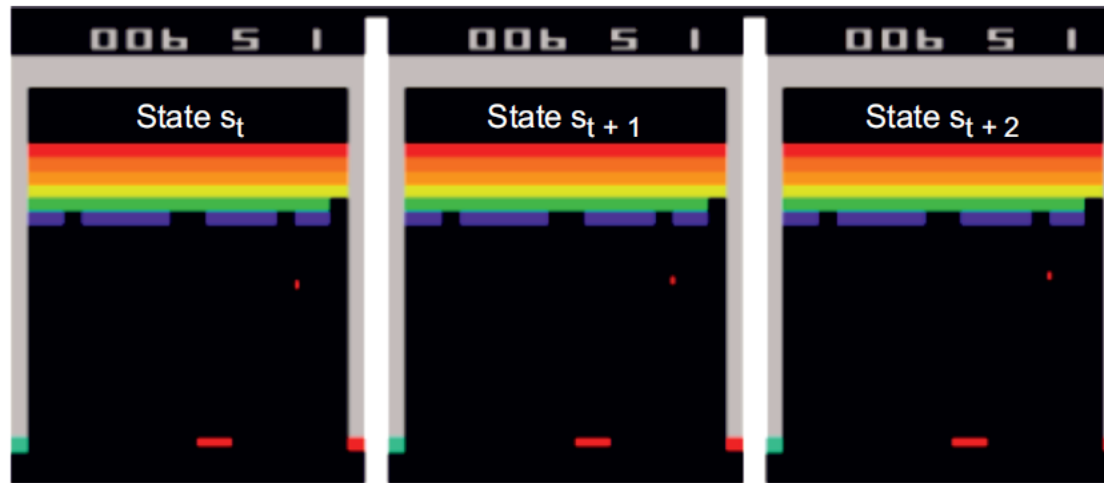


Q-learning with a target network



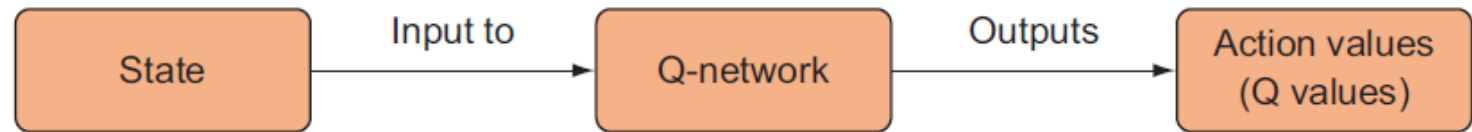
Improving stability with a target network

DeepMind's DQN algorithm for Atari games



Four (three here) 84×84 grayscale images at each step, which would lead to 256^{28228} unique game states. DQN CNN have 1792 parameters

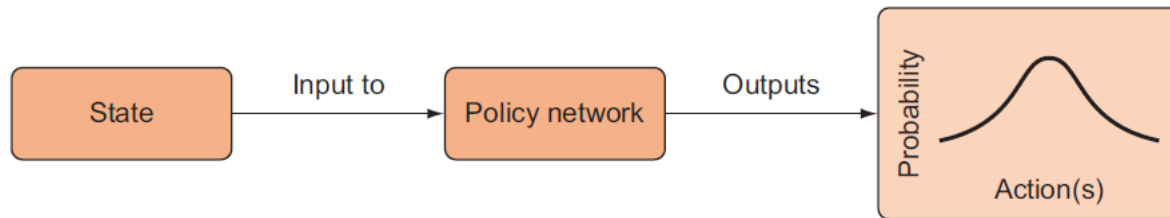
The Q value is the expected (weighted average) of rewards



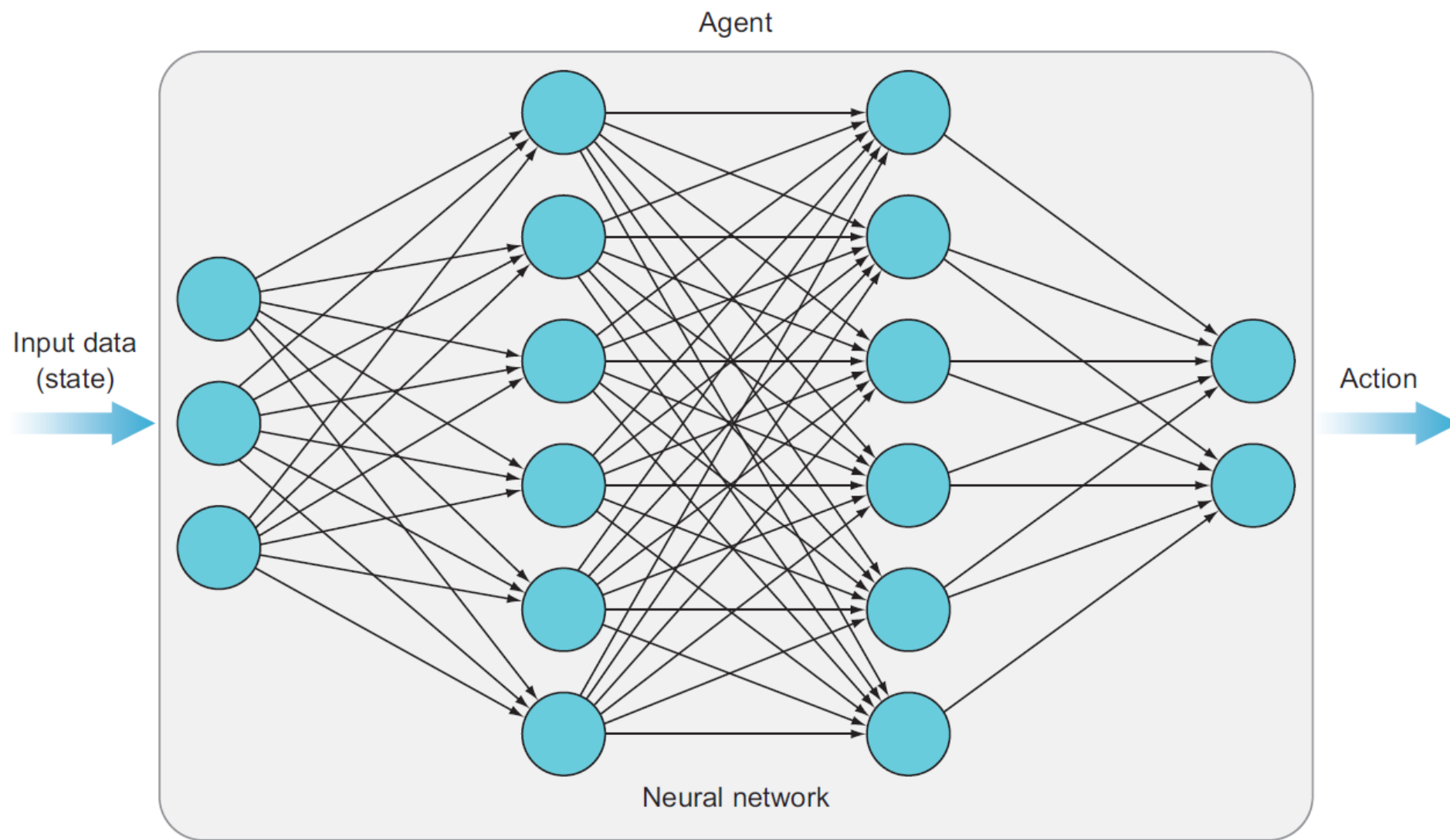
- A Q-network takes a state and returns Q values (action values) for each action. We can use those action values to decide which actions to take

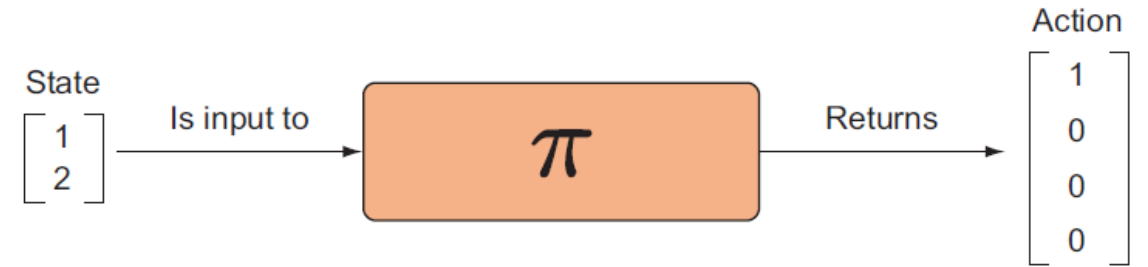
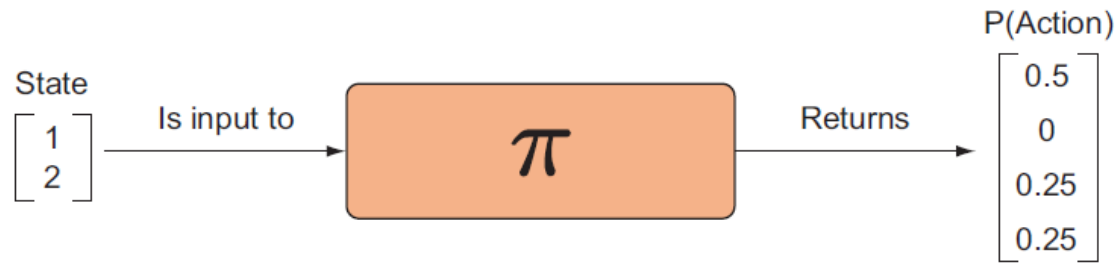
Instead train a neural network to output an action directly?

- A policy network tells us exactly what to do given the state we're in. No further decisions are necessary
- All we need to do is randomly sample from the probability distribution $P(A|S)$, and we get an action to take



- This class of algorithms is called *policy gradient methods*





Eventually converges to a *degenerate probability distribution*.
it is difficult to get this working in the fully differentiable
manner that we are accustomed to with deep learning; avoid

Stochastic policy gradient

How to train the network?

Previously we trained the deep Q-network with a minimizing mean squared error (MSE) loss function with respect to its predicted Q values and the target Q value.

Instead use $\pi_{\theta}(s)$; returns action probability distribution:
[0.25, 0.25, 0.25, 0.25], (for a start)



Episode= List of (State,action,resulting reward):

Gridworld episode: $\epsilon = (S0,3,-1),(S1,1,-1),(S2,3,+10)$

How to learn from? –
moves seems good?

Reinforce those
actions that led to a
nice positive reward,
particularly 3 in S2

Modify θ such that
 $\pi_s(a_3 | \theta) > 0.25$. Or
Minimize $1 - \pi_s(a_3 | \theta)$

Surrogate objective

- Sometimes probabilities may be extremely tiny or very close to 1, and this runs into numerical issues when optimizing on a computer with limited numerical precision.
- Use $-\log \pi_s(a_3 | \theta)$ (where \log is the natural logarithm), instead of $1 - \pi_s(a_3 | \theta)$
- Ranges from $(-\infty, 0)$, approaches 0 as $\pi_s(a_3 | \theta)$ approaches 1
 - $\log(1) = 0$, $\log(0.9) = -0.1$, $\log(0.1) = -2.3$,
 $\log(0.01) = -4.6$, $\log(0) = -\text{INF}$

Credit assignment



It makes sense that the last action right before the reward deserves more credit for winning the game than does the first action in the episode.



Multiply the magnitude of the update by the discount factor



The final objective function that we will tell PyTorch to minimize is:



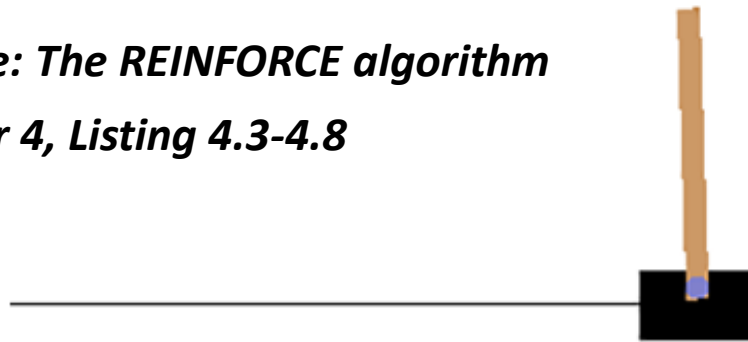
$$- \gamma_t * G_t * \log \pi_s(a | \theta), \gamma_t = \gamma^{(T-t)},$$

T length of the episode, $\gamma = 0.99$

Open AI Gym

```
import gym
env = gym.make('CartPole-v0')
```

Exercise: The REINFORCE algorithm
Chapter 4, Listing 4.3-4.8



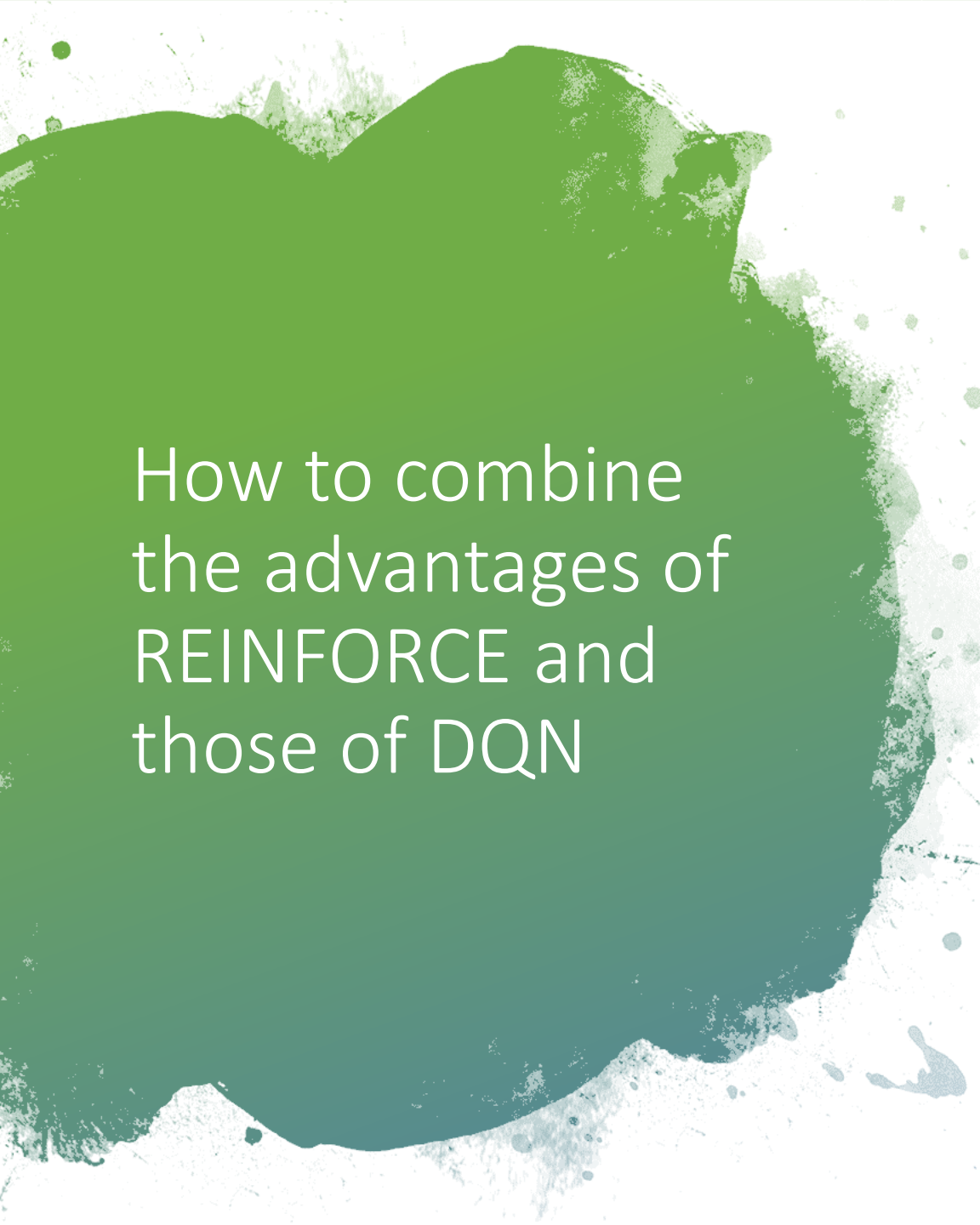
*Tackling
more
complex
problems
with
actor-critic
methods*



INTRODUCING A *CRITIC* TO IMPROVE
SAMPLE EFFICIENCY AND DECREASE
VARIANCE



USING THE ADVANTAGE FUNCTION
TO SPEED UP CONVERGENCE



How to combine the advantages of REINFORCE and those of DQN

- REINFORCE algorithm is generally implemented as an *episodic algorithm*
 - We only apply it to update our model parameters after the agent has completed an entire episode (and collected rewards along the way)
 - By sampling a full episode, we get a pretty good idea of the true value of an action because we can see its downstream effects rather than just its immediate effect, i.e. Monte Carlo approach.
- Our deep Q-network did well in the non-episodic setting and it could be considered an online-learning algorithm, but it required an experience replay buffer in order to effectively learn

A new kind of policy gradient method called *distributed advantage actor-critic* (DA2C)



Have

the online-learning advantages of DQN without a replay buffer

the advantages of policy methods where we can directly sample actions from the probability distribution over actions, thereby eliminating the need for choosing a policy (such as the epsilon-greedy policy)



We want to

improve the sample efficiency by updating more frequently.

decrease the variance of the reward we used to update our model.



Use the value learner to reduce the variance in the rewards that are used to train the policy

Advantage

- instead of minimizing the REINFORCE loss that included direct reference to the observed return, R , from an episode, we instead add a baseline value such that the loss is now:

- $V(S) - R$, is termed the *advantage*.
- the advantage quantity tells you how much better an action is, relative to what you expected.

**Log probability of
action given state**

Return

State value


$$\text{Loss} = -\log (\pi(a \mid S)) \cdot (R - V_{\pi}(S))$$

Advantage example/explained



Imagine that we're training a policy on a Gridworld game with discrete actions and a



small discrete state-space, such that we can use a vector where each position in the vector represents a distinct state, and the element is the average rewards observed after that state is visited. This look-up table would be the $V(S)$.



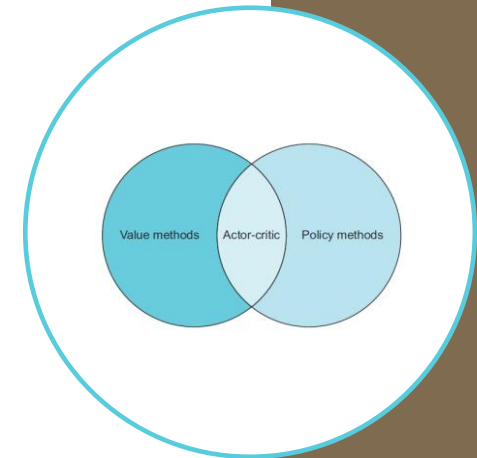
We might sample action 1 from the policy and observe reward +10, but then we'd use our value look-up table and see that on average we get +4 after visiting this state, so the advantage of action 1 given this state is $10 - 4 = +6$.



This means that when we took action 1, we got a reward that was significantly better than what we expected based on past rewards from that state, which suggests that it was a good action.

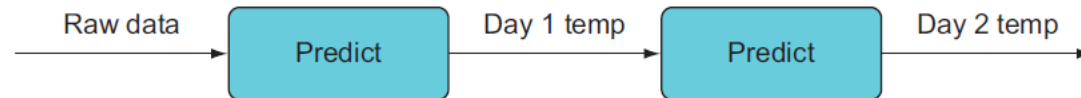
Rather than using a look-up table, we will use some sort of parameterized model

- We want a neural network that can be trained to predict expected rewards for a given state. So we want to simultaneously train a policy neural network and a state-value or action-value neural network.
- “actor” refers to the policy, because that’s where the actions are generated, and “critic” refers to the value function, because that’s what (in part) tells the actor how good its actions are.
- we’re using $R - V(S)$ to train the policy rather than just $V(S)$, this is called *advantage actor-critic*



Bootstrapping

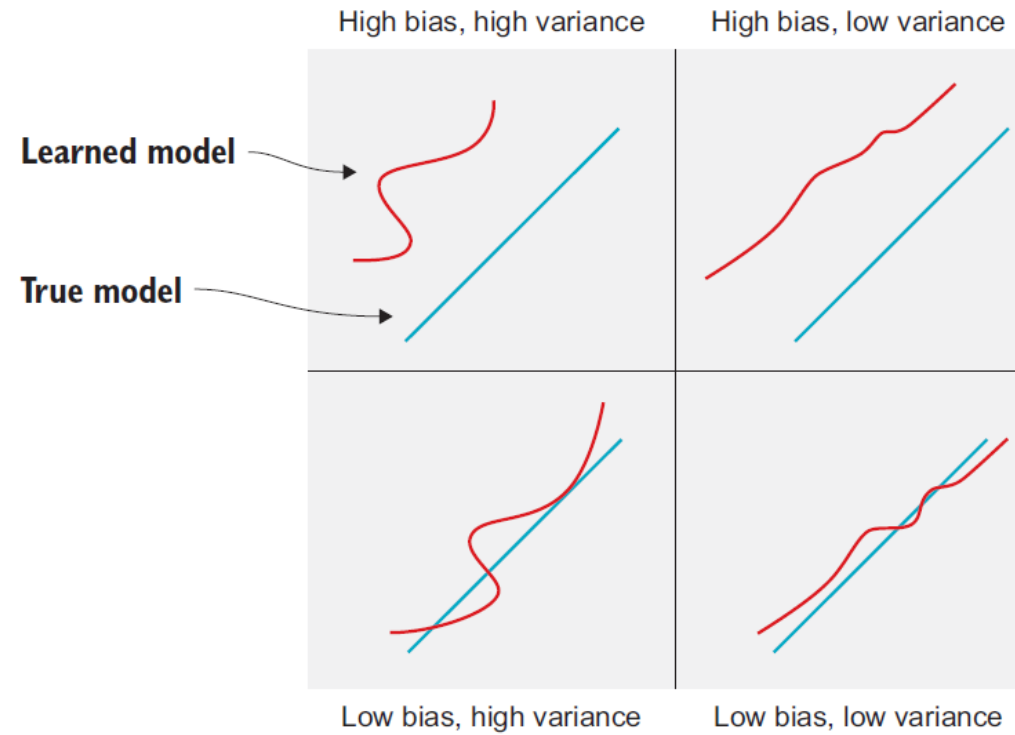
- If we ask you what the temperature will be like two days from now, you might first predict what the temperature will be tomorrow, and then base your 2-day prediction on that
- " make a prediction from a prediction"



- If your first prediction is bad, your second may be even worse, so bootstrapping introduces a source of *bias*. Bias is a systematic deviation

Variance

- A lack of precision in the predictions, which means predictions that vary a lot.
- In the temperature example, if we make our day 2 temperature prediction based on our day 1 prediction, it will likely not be too far from our day 1 prediction; lower *variance*



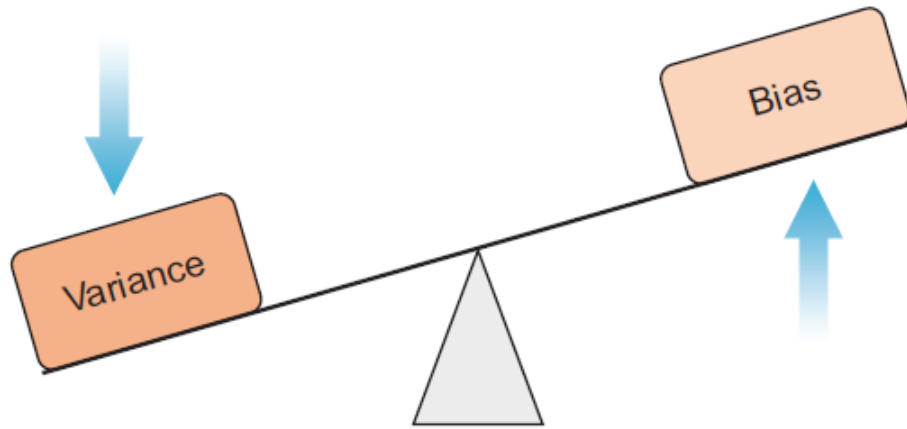
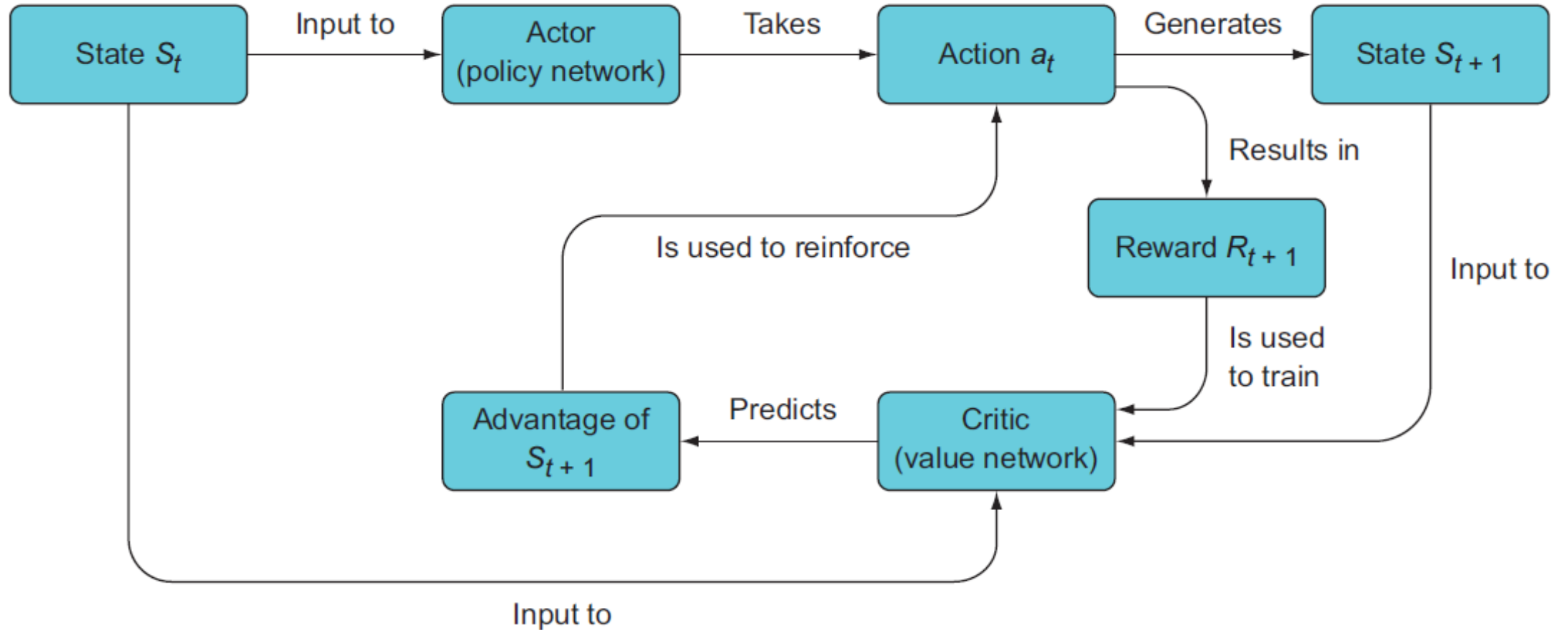


Figure 5.5 The bias-variance tradeoff. Increasing model complexity can reduce bias, but it will increase variance. Reducing variance will increase bias.

Regularization

- machine learning models are often *regularized* by imposing a penalty on the magnitude of the parameters during training; i.e., parameters that are significantly bigger or smaller than 0 are penalized.
- Regularization essentially means modifying your machine learning procedure in a way to mitigate overfitting

General overview of actor-critic models



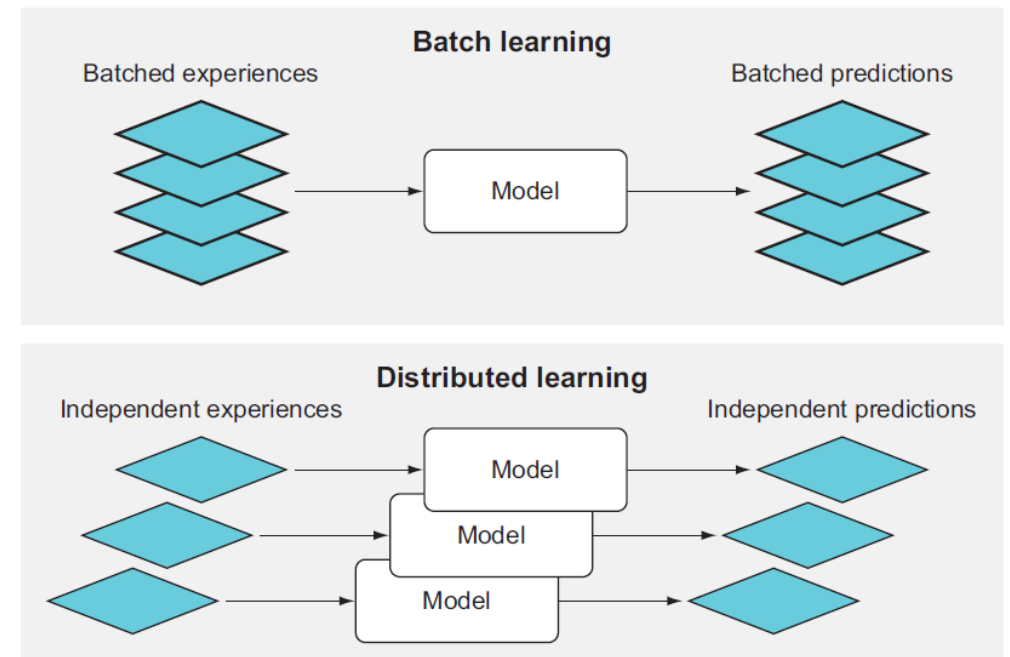
Distributed training

- **distributed** advantage actor-critic (DA2C)

One way to use RNNs (LSTMs) without an experience replay is to run multiple copies of the agent in parallel, each with separate instantiations of the environment.

By distributing multiple independent agents across different CPU processes, we can collect a varied set of experiences and therefore get a sample of gradients that we can average together to get a lower variance mean gradient.

This eliminates the need for experience replay and allows us to train an algorithm in a completely online fashion, visiting each state only once as it appears in the environment



Multiprocessing in Python