Deep Reinforcement Learning

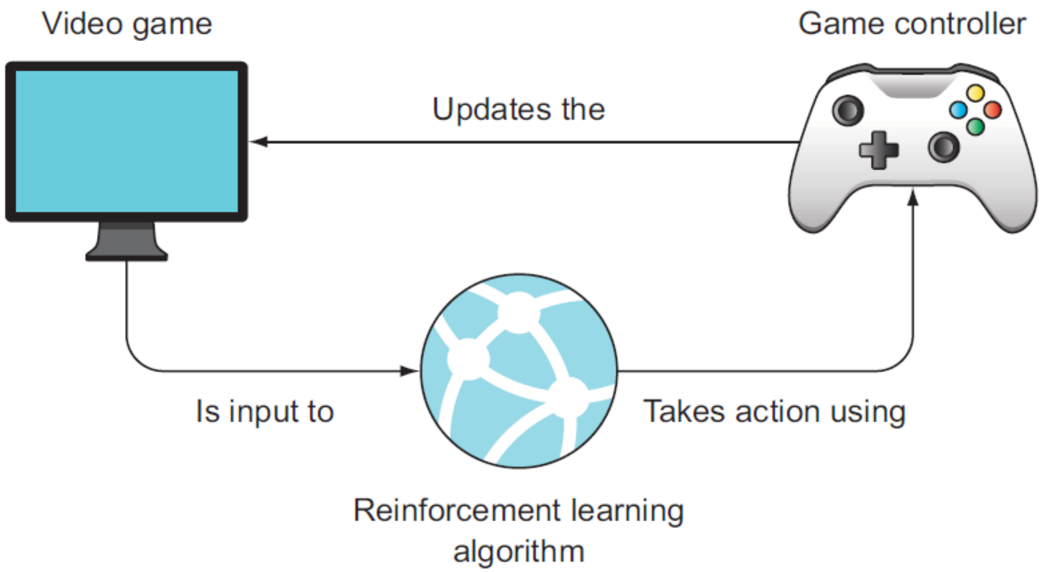
Kalle.Prorok@gmail.com , Umeå 2020

Parts based on the book Deep Reinforcement Learning in Action By Alexander Zai and Brandon Brown

And the free Sutton-Barto book Reinforcement Learning: An Introduction

Reinforcement learning

Deep Reinforcement Learning: Pong from Pixels



Video game

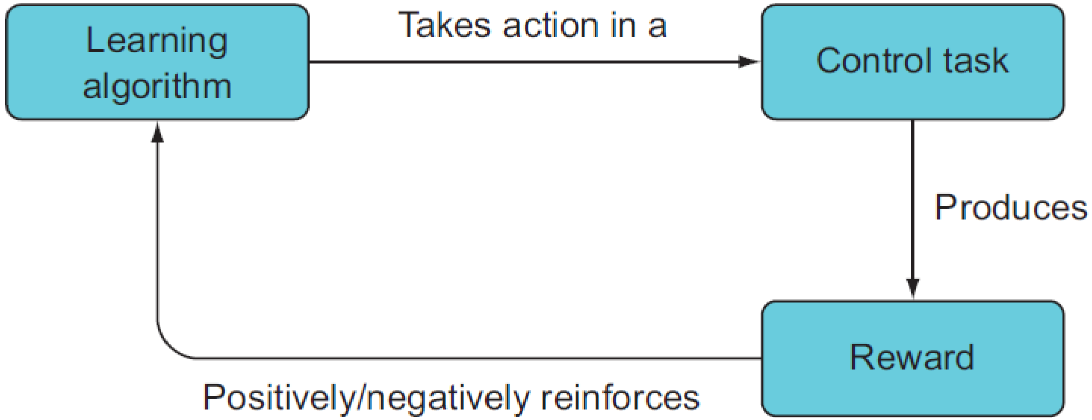
Game controller

Updates the

Is input to

Takes action using

Reinforcement learning algorithm



Learning algorithm

Takes action in a

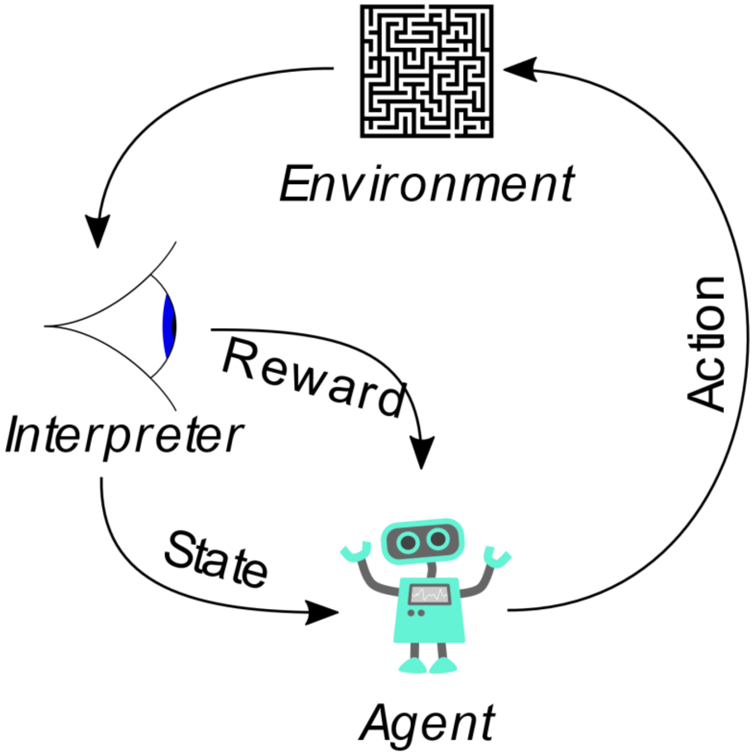
Control task

Produces

Reward

Positively/negatively reinforces

REINFORCEMENT LEARNING



Ex. Smart Elevators

NEW WAY TO PROGRAM

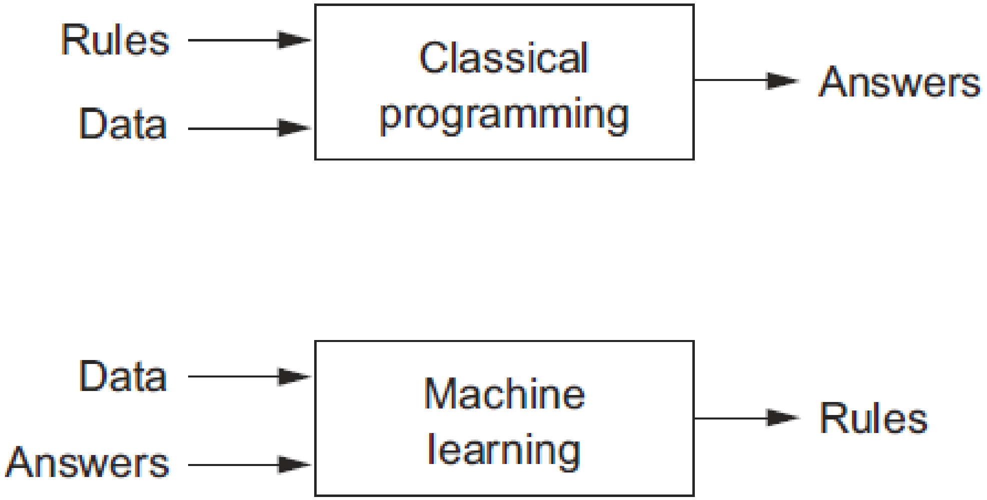
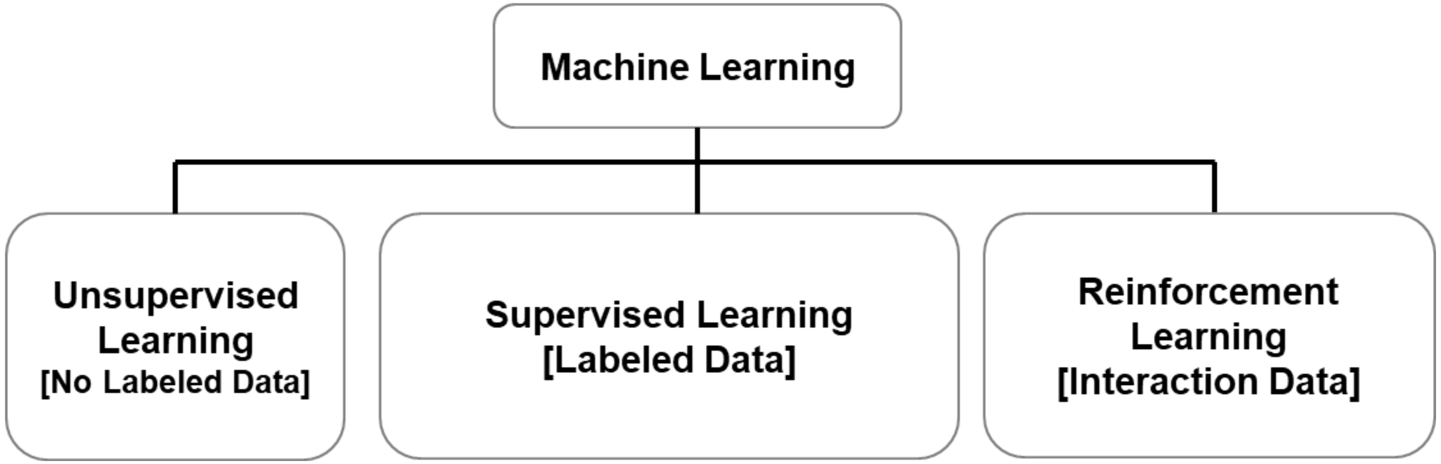


Figure 1.2 **Machine learning: a new programming paradigm**

Ur F Chollet: Deep Learning with Python

Reinforcement Learning vs Machine Learning vs Deep Learning



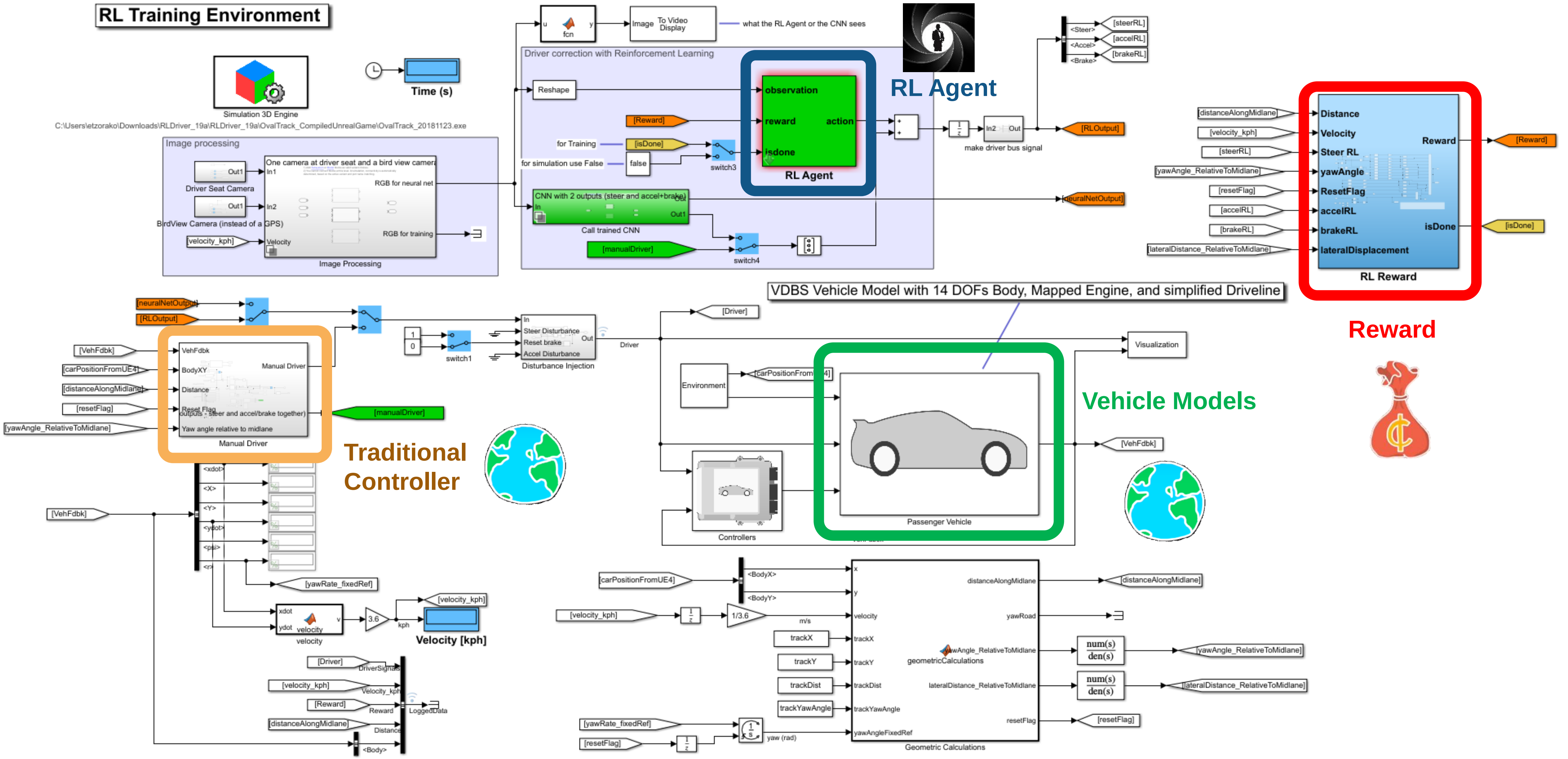
Machine Learning

Unsupervised Learning
[No Labeled Data]

Supervised Learning
[Labeled Data]

Reinforcement Learning
[Interaction Data]

Problem Setup



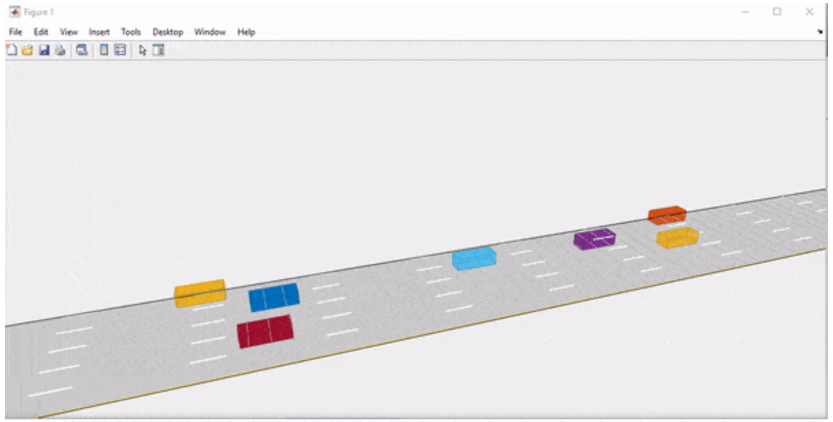
RL Agent

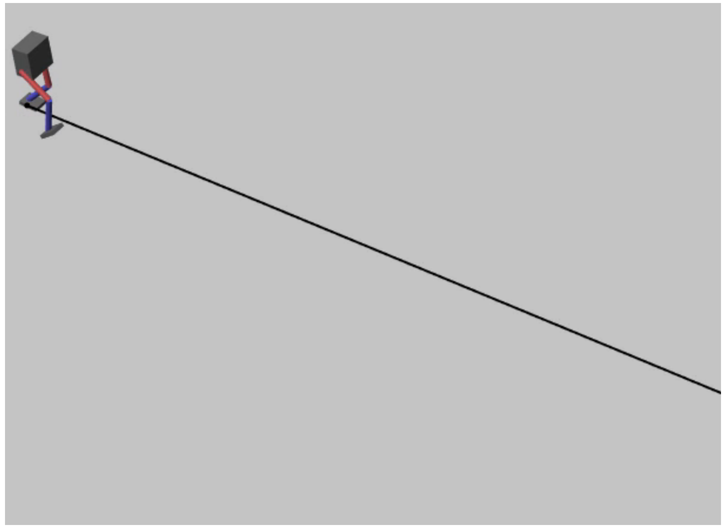
Reward

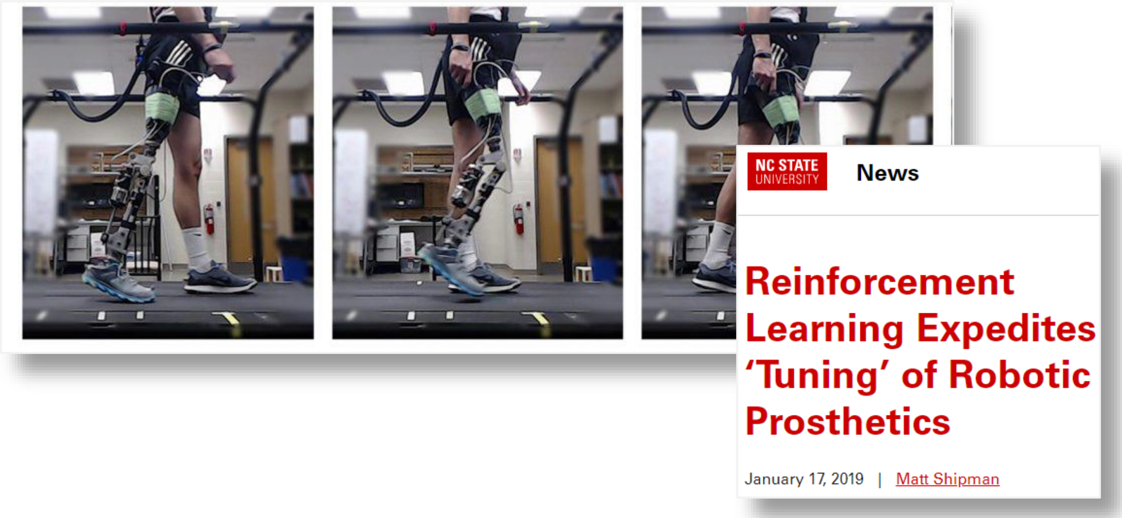
Vehicle Models

Traditional Controller

Apply Reinforcement Learning to **Control System Design** & **Decision Making Systems**



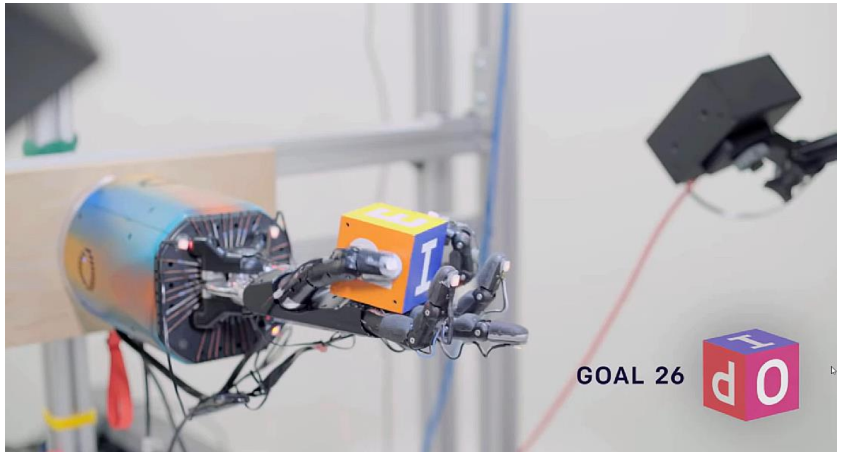




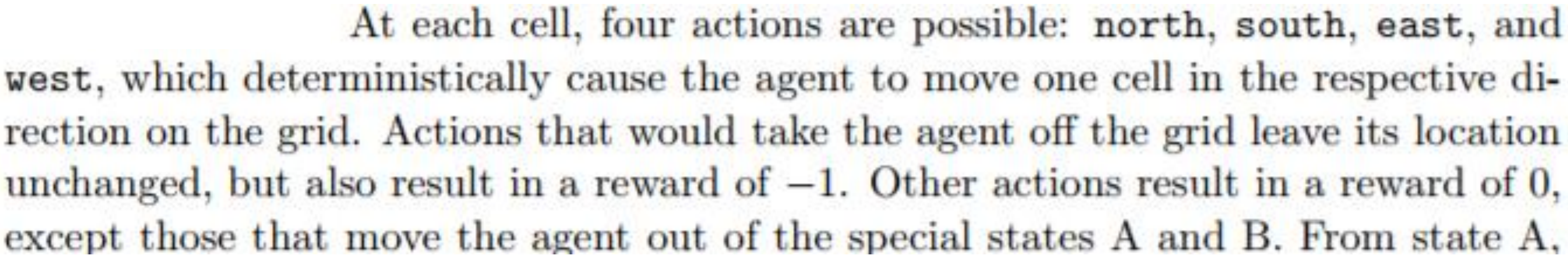
NC STATE UNIVERSITY News

Reinforcement Learning Expedites 'Tuning' of Robotic Prosthetics

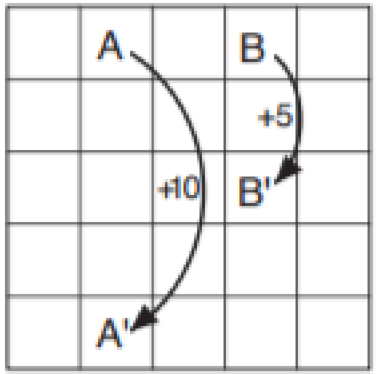
January 17, 2019 | Matt Shipman



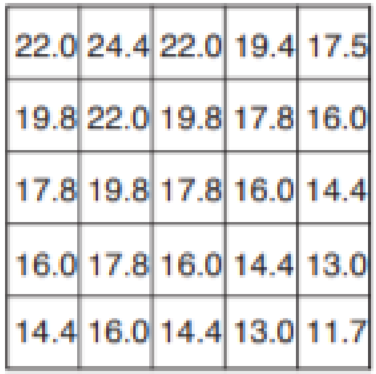
STATE-TABLES WITH VALUES



At each cell, four actions are possible: **north**, **south**, **east**, and **west**, which deterministically cause the agent to move one cell in the respective direction on the grid. Actions that would take the agent off the grid leave its location unchanged, but also result in a reward of −1. Other actions result in a reward of 0, except those that move the agent out of the special states A and B. From state A,

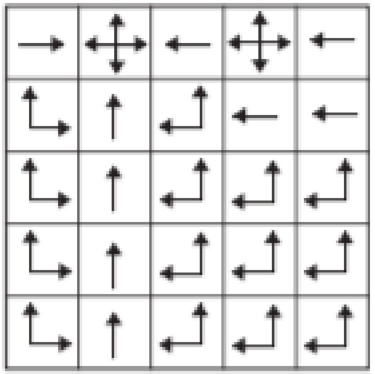


Gridworld



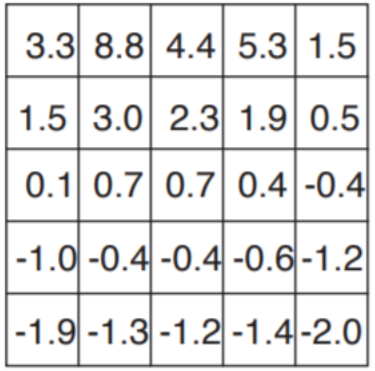
22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

v_*



π_*

Random policy



3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

From Sutton RL 1998

Policy for actions

Dynamic programming

- *Goal decomposition* as it solves complex high-level problems by decomposing them into smaller and smaller subproblems until it gets to a simple subproblem that can be solved
- Train a robot vacuum to move from one room in a house to its dock
  - stay in/exit this room?
  - move toward the door” or “move away from the door
- You need to know your house extremely well

DeepReinforcementLearningInAction-master/Chapter 1/Ch1_Introduction.ipynb

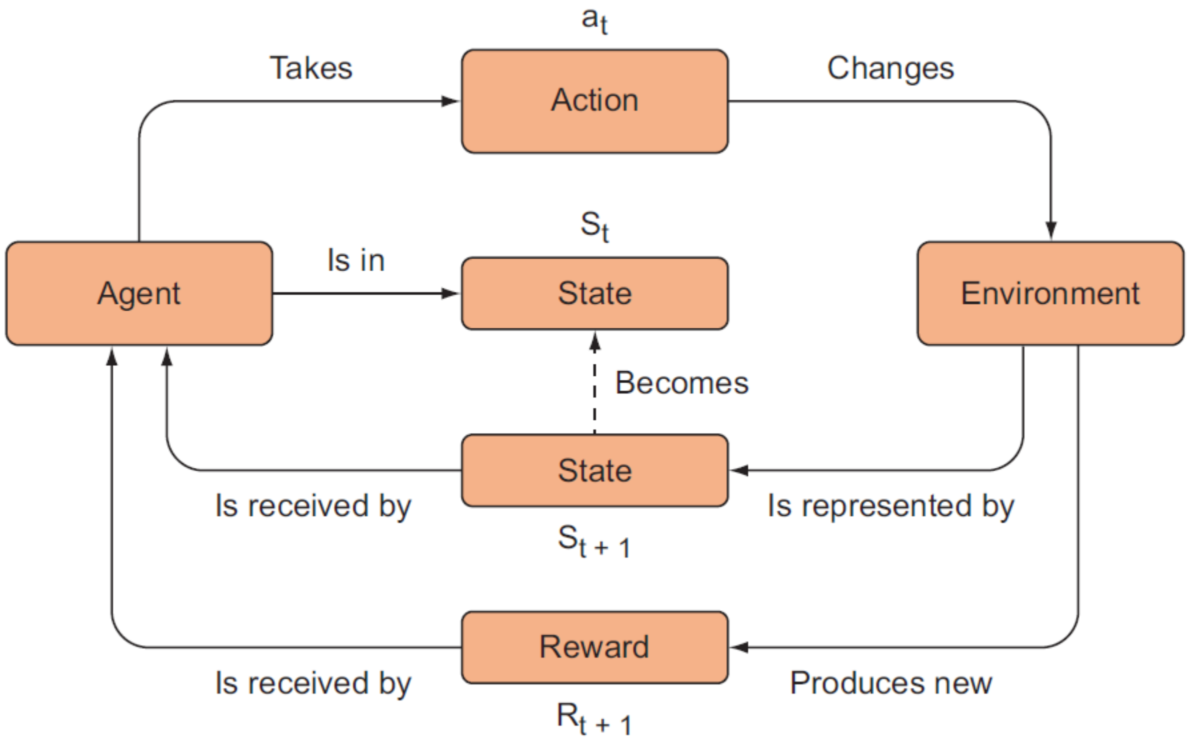
The Bellman equation for v

$$
\begin{aligned}
v_\pi(s) &\doteq \mathbb{E}_\pi[G_t \mid S_t = s] \\
&= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1} \mid S_t = s] \\
&= \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r \,|\, s, a) \Big[r + \gamma \mathbb{E}_\pi[G_{t+1} | S_{t+1} = s'] \Big] \\
&= \sum_a \pi(a|s) \sum_{s', r} p(s', r \,|\, s, a) \Big[r + \gamma v_\pi(s') \Big], \quad \text{for all } s \in \mathcal{S},
\end{aligned}
$$

Monte Carlo

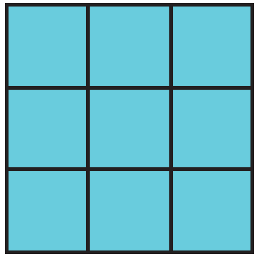
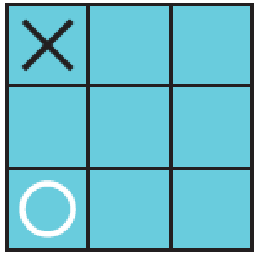
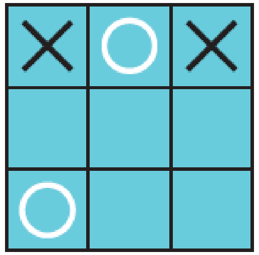
- The trial and error strategy
- Go to a party at a house that you’ve never been to before, you might have to look around until you find the bathroom on your own

Agent


a_t
Takes
Action
Changes
S_t
Agent
Is in
State
Environment
Becomes
State
Is received by
Is represented by
S_{t+1}
Reward
Is received by
Produces new
R_{t+1}

An action lookup table for Tic-Tac-Toe

Game play lookup table

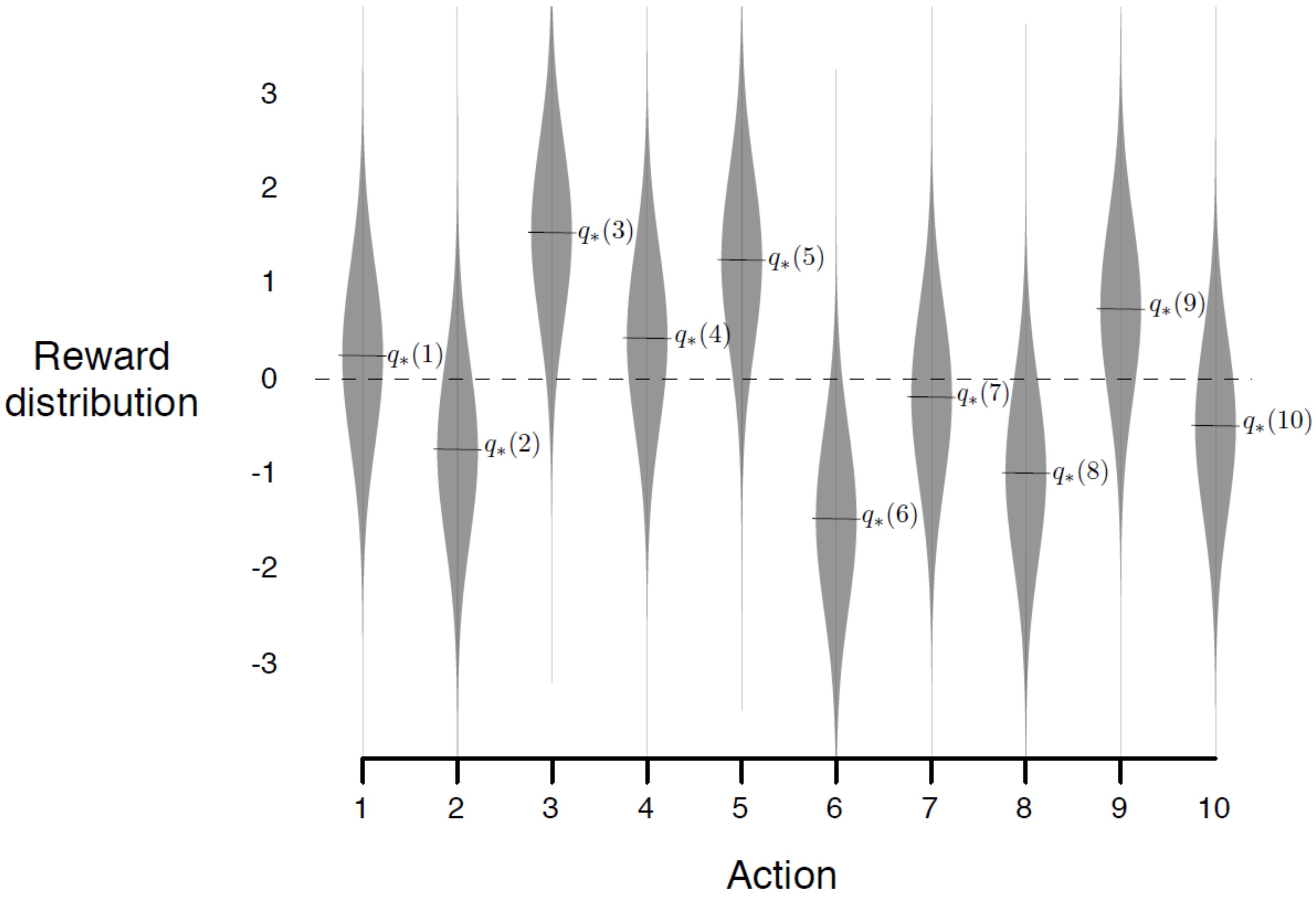
Key Current state	Value Action to take
	Place X in top left
	Place X in top right
	Place X in bottom right

...

there are 255,168 valid board positions..

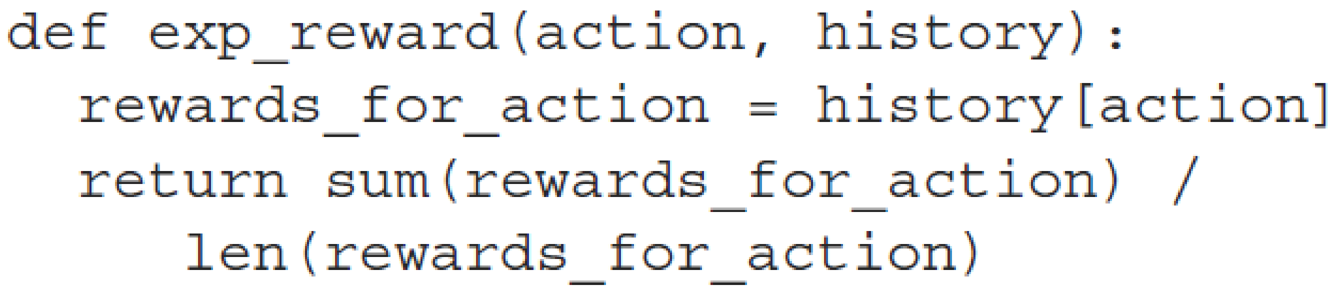
The multi-arm bandit problem

- 10 slot machines with a sign that says “Play for free! Max payout $10!
- Each have a different average payout
  - the reward at each play is probabilistic
  - so try to figure out which one gives the most rewards on average
- ” play a few times, choosing different levers(a) and observing our rewards R for each action. Then we want to only choose the lever with the largest observed average reward.”


Reward distribution
3
2
1
0
-1
-2
-3
$q_*(1)$
$q_*(2)$
$q_*(3)$
$q_*(4)$
$q_*(5)$
$q_*(6)$
$q_*(7)$
$q_*(8)$
$q_*(9)$
$q_*(10)$
1
2
3
4
5
6
7
8
9
10
Action

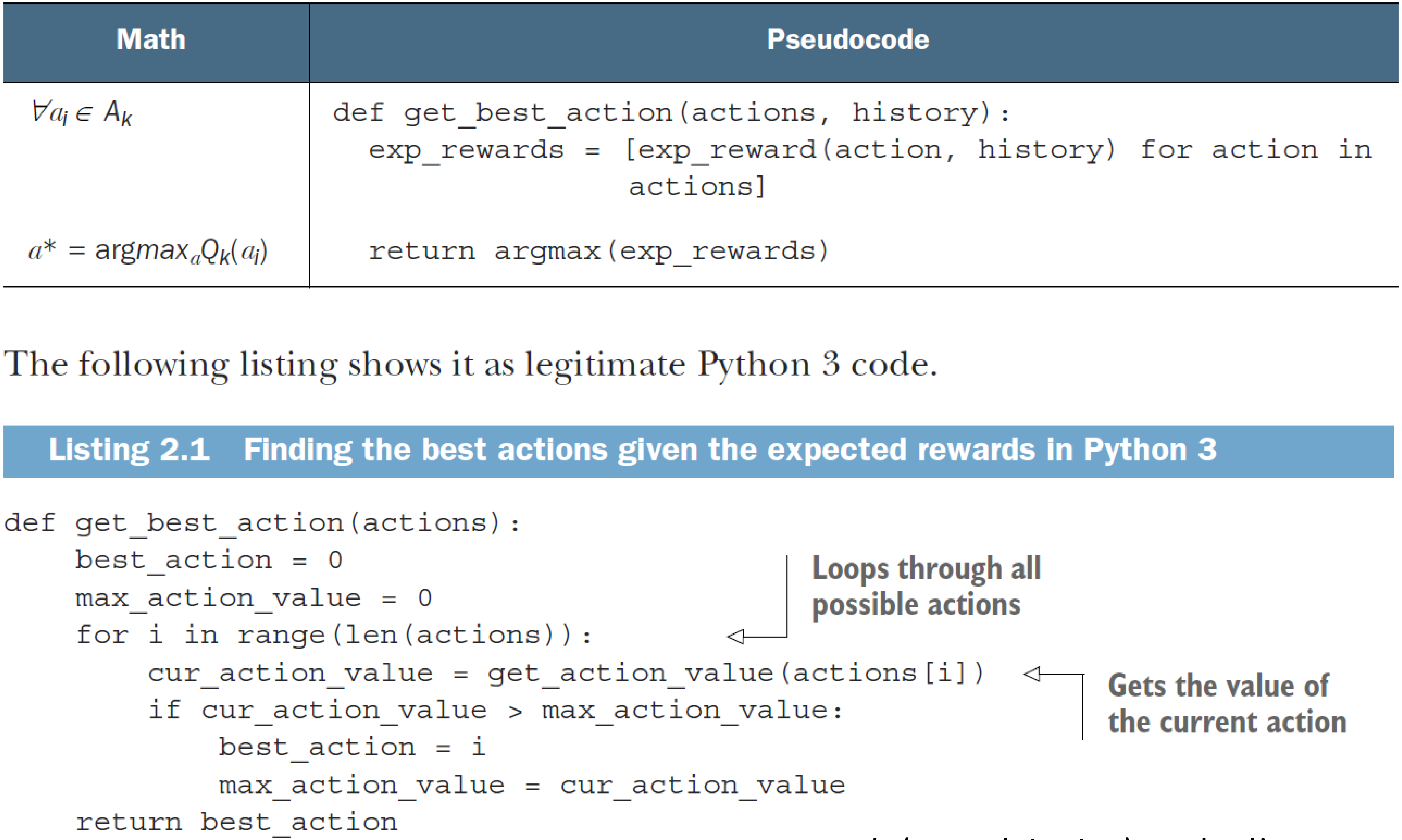
expected reward at play k for taking an action (*a*)

$$Q_k(a) = \frac{R_1 + R_2 + \ldots + R_k}{k_a}$$



```
def exp_reward(action, history):
  rewards_for_action = history[action]
  return sum(rewards_for_action) /
     len(rewards_for_action)
```

action-value function; the value of taking a particular action



Math	Pseudocode
$\forall a_i \in A_k$	def get_best_action(actions, history): exp_rewards = [exp_reward(action, history) for action in actions]
$a^* = \mathrm{argmax}_a Q_k(a_i)$	return argmax(exp_rewards)

The following listing shows it as legitimate Python 3 code.

Listing 2.1 Finding the best actions given the expected rewards in Python 3

```
def get_best_action(actions):
    best_action = 0
    max_action_value = 0
    for i in range(len(actions)):
        cur_action_value = get_action_value(actions[i])
        if cur_action_value > max_action_value:
            best_action = i
            max_action_value = cur_action_value
    return best_action
```

Loops through all possible actions

Gets the value of the current action

a *greedy* (or exploitation) method!

Dilemma/Balance

Exploration

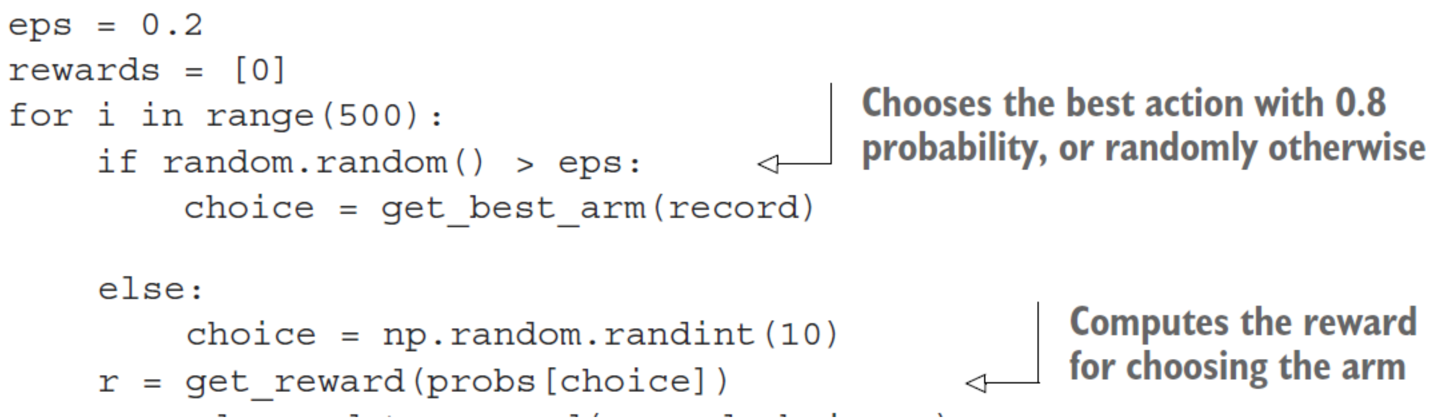
- Play the game and observe the rewards we get for the various machines
- To learn more
- Random selected action?

Exploitation

- Use our current knowledge about which machine seems to produce the most rewards, and keep playing that machine
- Greedy

Epsilon-greedy strategy

- with a probability, ε, we will choose an action, *a*, at random, and the rest of the time (probability 1 – ε) we will choose the best lever based on what we currently know from past plays



```
eps = 0.2
rewards = [0]
for i in range(500):
    if random.random() > eps:
        choice = get_best_arm(record)

    else:
        choice = np.random.randint(10)
    r = get_reward(probs[choice])
```

Chooses the best action with 0.8 probability, or randomly otherwise

Computes the reward for choosing the arm

Softmax selection policy

- Treating patients with heart attacks, choose 1 treatment out of 10
  - doesn’t know which one is the best yet
  - randomly choosing a treatment could result in patient death, not just losing some money. We really want to make sure we don’t choose the worst treatment

$$\Pr(A) = \frac{e^{Q_k(A)/\tau}}{\sum_{i=1}^{n} e^{Q_k(i)/\tau}}$$

```
def softmax(av, tau=1.12):
    softm = np.exp(av / tau) / np.sum( np.exp(av / tau) )
    return softm
```

τ is a parameter called *temperature* that scales the probability distribution of actions. Usually reduced over time, “cooling metal”.

OPTIMISTIC INITIAL VALUE

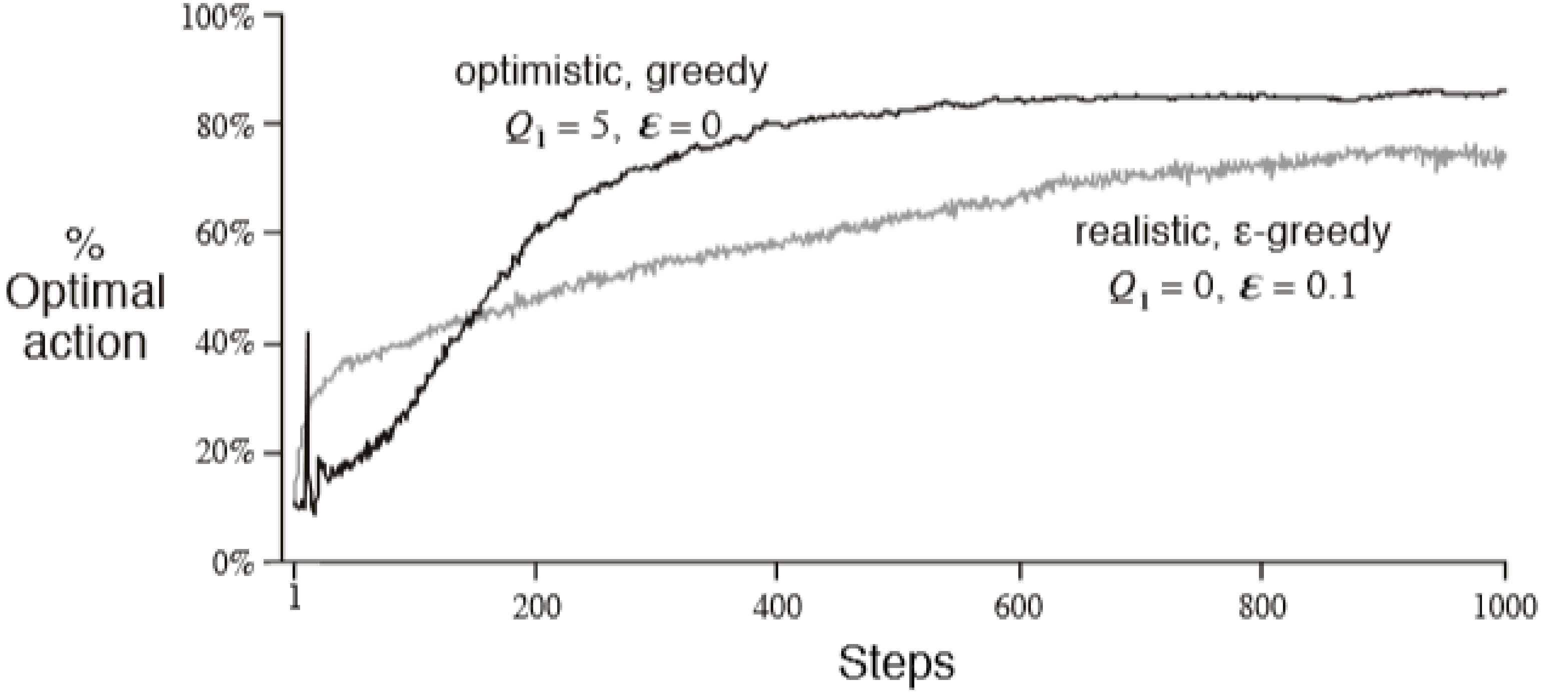
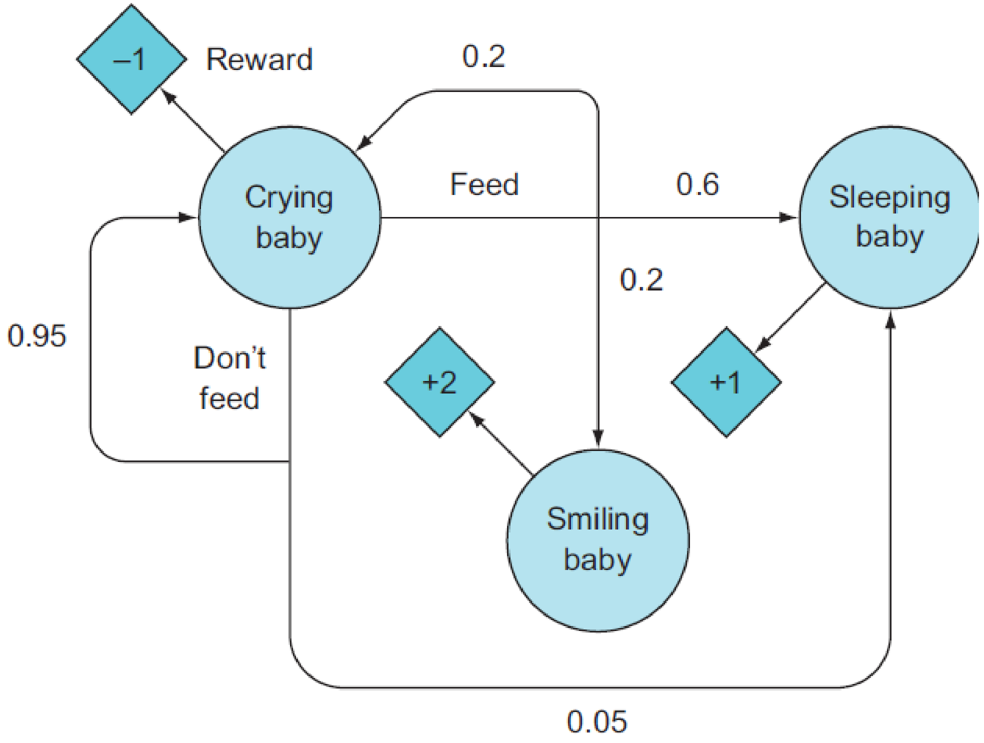


Figure 2.3: The effect of optimistic initial action-value estimates on the 10-armed testbed. Both methods used a constant step-size parameter, $\alpha = 0.1$.

The Markov property

- the current state alone contains enough information to choose optimal actions to maximize future rewards
- A game (or any other control task) that exhibits the Markov property is said to be a *Markov decision process* (MDP):

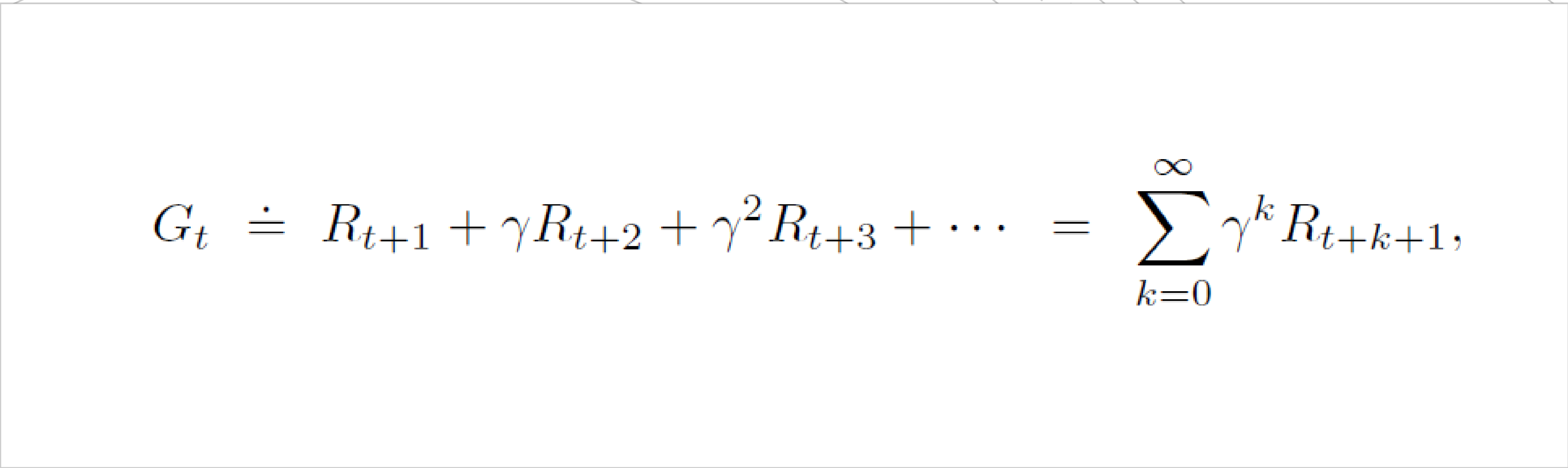


Markov property or not?

- Driving a car
- Deciding whether to invest in a stock or not
- Choosing a medical treatment for a patient
- Diagnosing a patient’s illness
- Predicting which team will win in a football game
- Choosing the shortest route (by distance) to some destination
- Aiming a gun to shoot a distant target

Episodic or Continuous task?

- most games are *episodic*, meaning that there are multiple chances to take actions before the game is over, and many games like chess don’t naturally assign points to anything other than winning or losing the game
- a continuously learning RL algorithm has the advantage of being able to adapt to changing market conditions in real time
  - Need discount to avoid *infinite* reward



$$G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

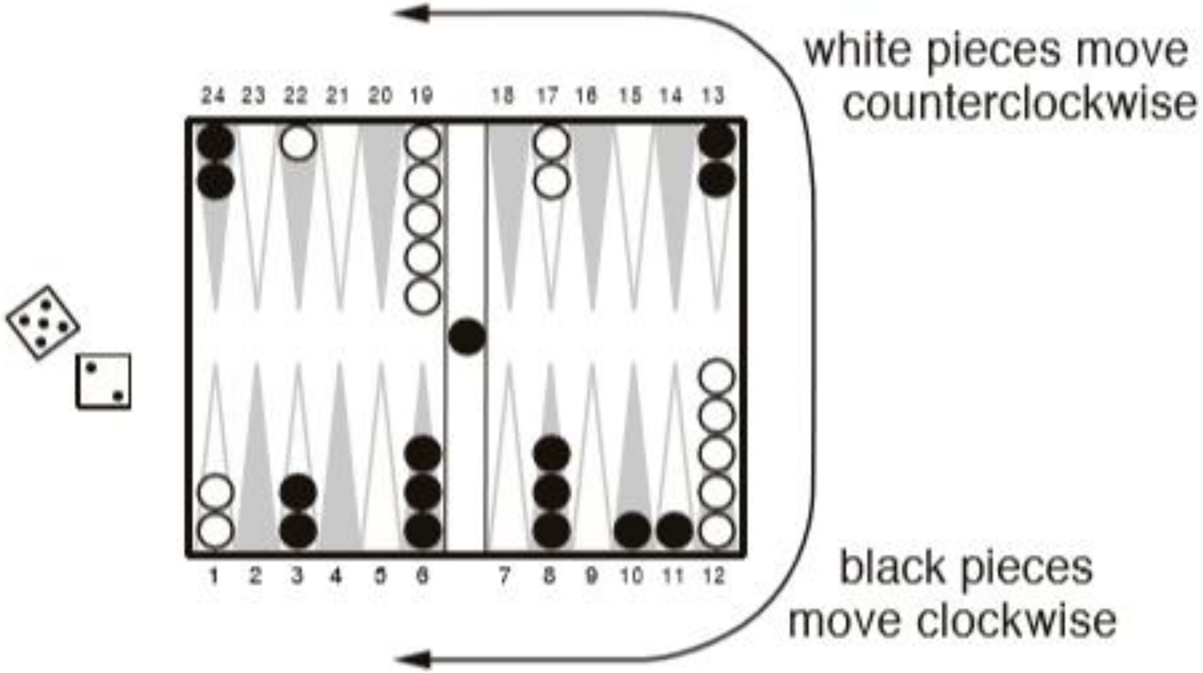
maximize the expected discounted return Gt

SARSA

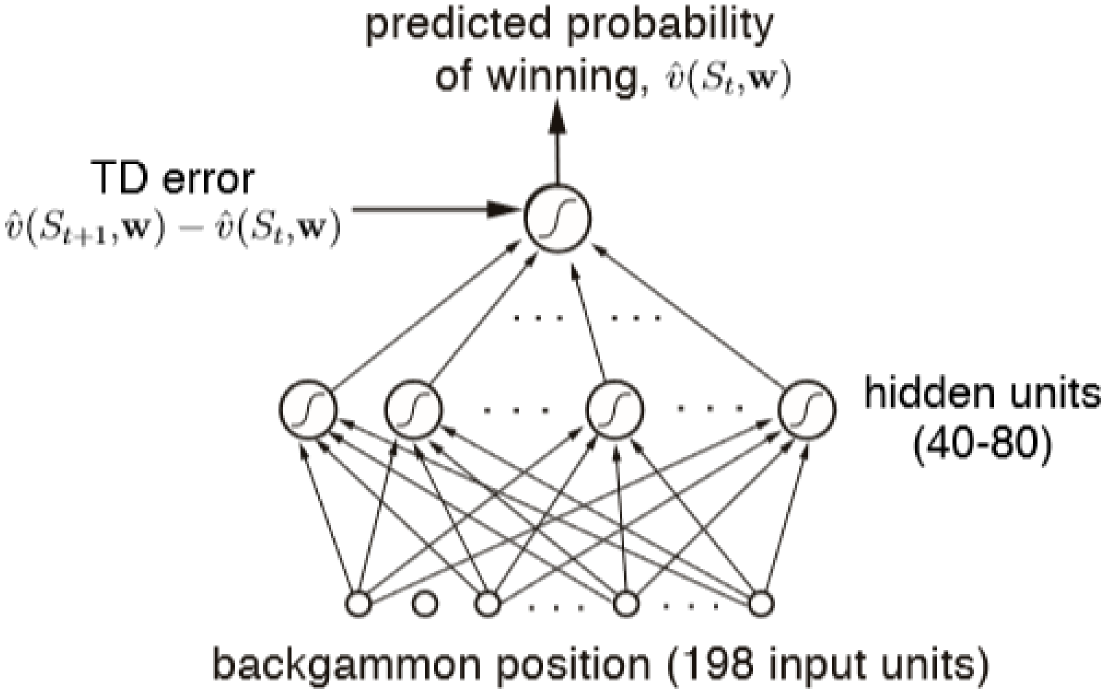
- $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$

- On-policy algorithm

TD-GAMMON (TESAURO 1992)



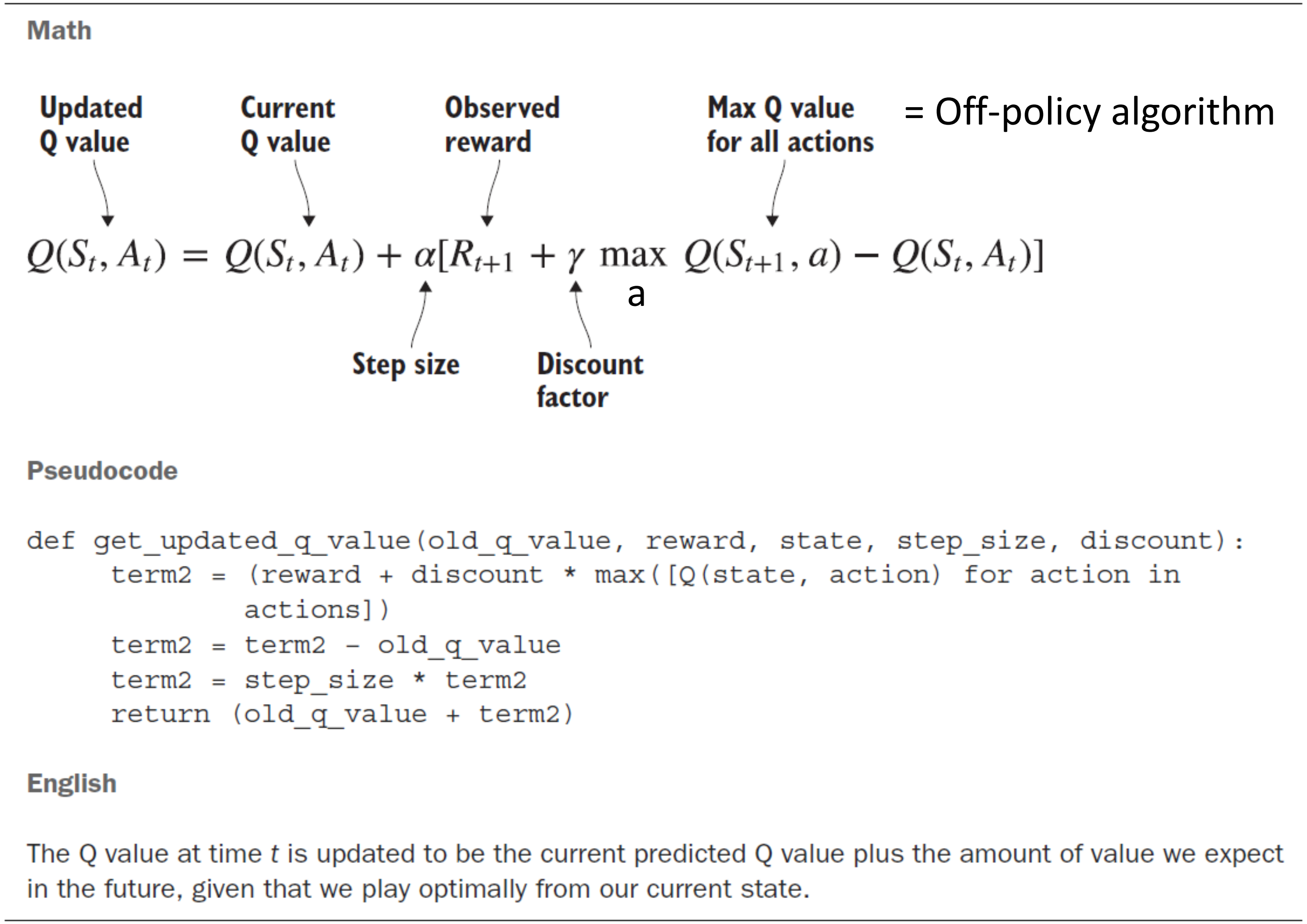
white pieces move counterclockwise
black pieces move clockwise
24 23 22 21 20 19 18 17 16 15 14 13
1 2 3 4 5 6 7 8 9 10 11 12



predicted probability of winning, $\hat{v}(S_t,\mathbf{w})$
TD error
$\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})$
hidden units (40-80)
backgammon position (198 input units)

Figure 16.1: The TD-Gammon neural network

Q-learning algorithm update rule



Math

$$\overset{\text{Updated Q value}}{Q(S_t, A_t)} = \overset{\text{Current Q value}}{Q(S_t, A_t)} + \underset{\text{Step size}}{\alpha}[\overset{\text{Observed reward}}{R_{t+1}} + \underset{\text{Discount factor}}{\gamma} \max_{a} \overset{\text{Max Q value for all actions}}{Q(S_{t+1}, a)} - Q(S_t, A_t)]$$

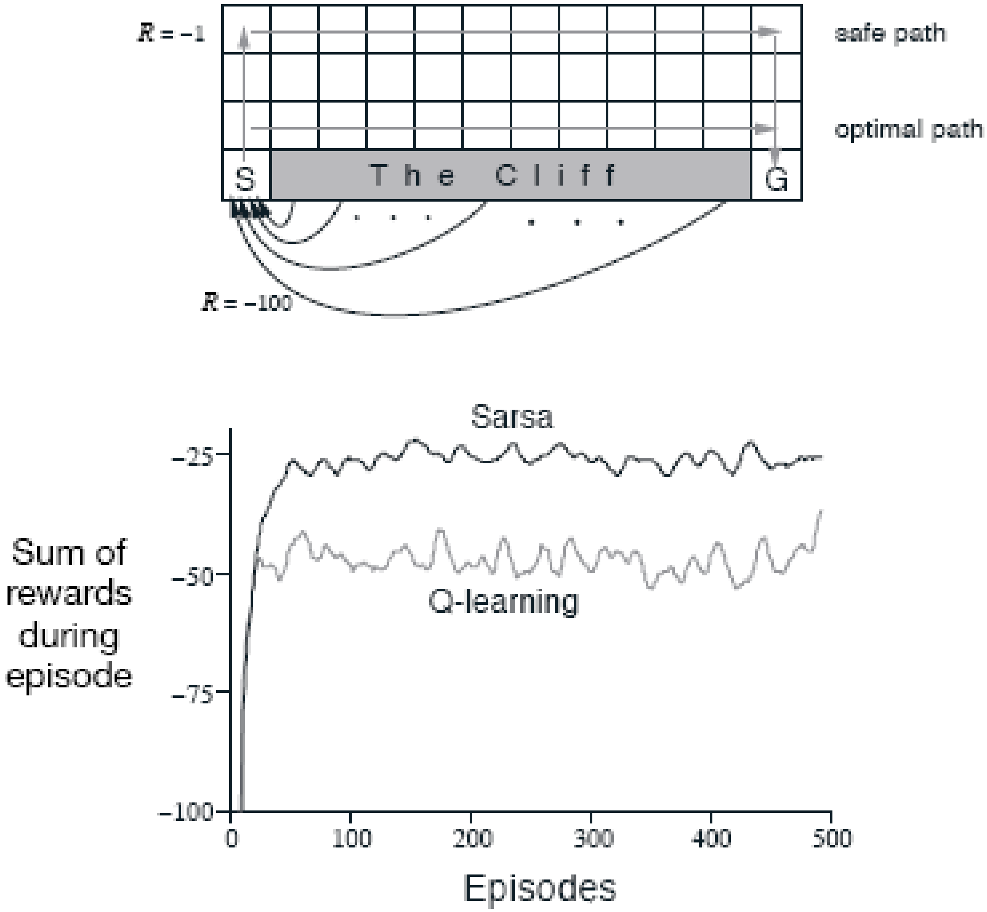
= Off-policy algorithm

Pseudocode

```
def get_updated_q_value(old_q_value, reward, state, step_size, discount):
    term2 = (reward + discount * max([Q(state, action) for action in
            actions])
    term2 = term2 - old_q_value
    term2 = step_size * term2
    return (old_q_value + term2)
```

English

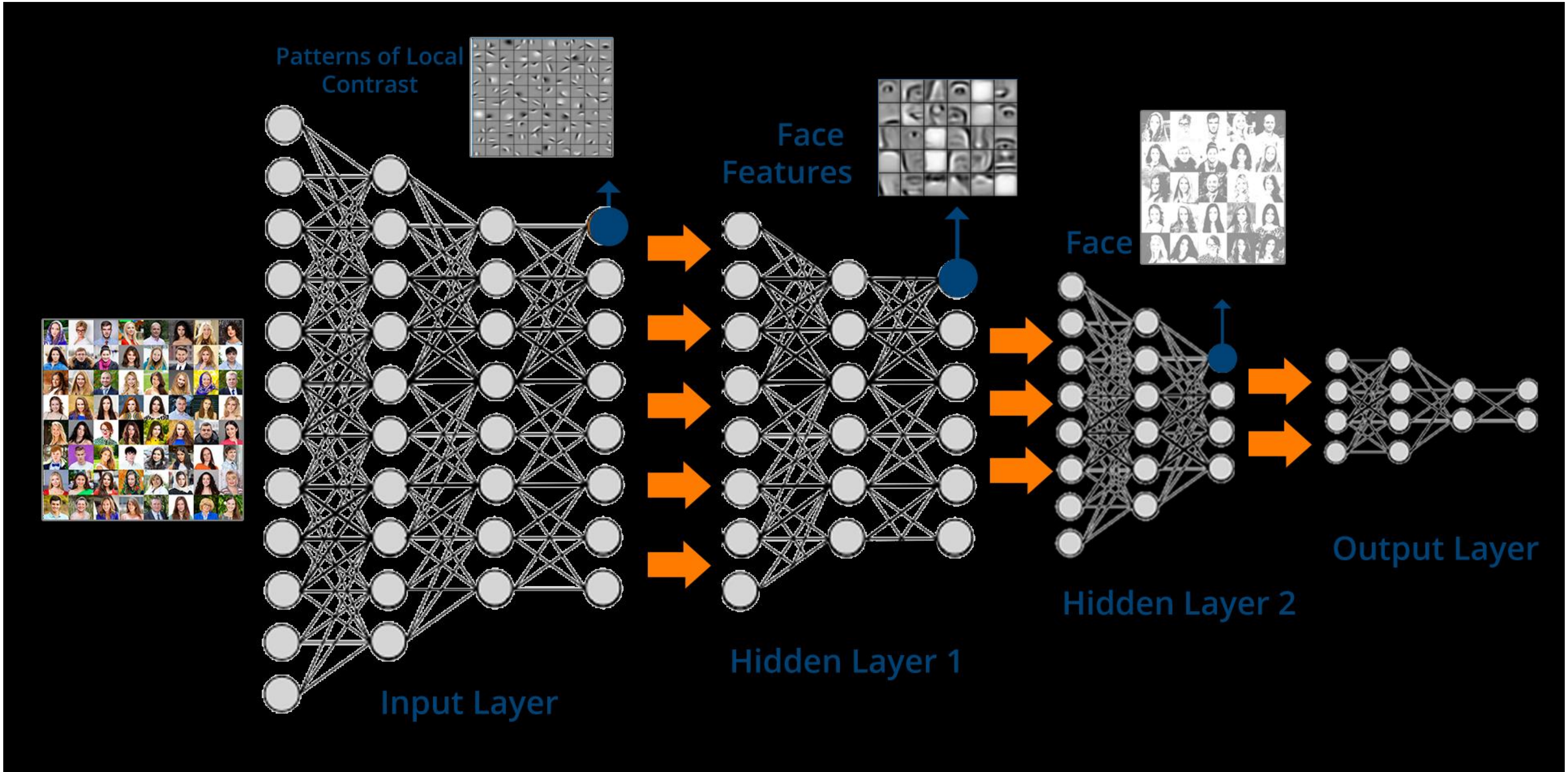
The Q value at time *t* is updated to be the current predicted Q value plus the amount of value we expect in the future, given that we play optimally from our current state.



On-policy algorithm

Off-policy algorithm

ANN DEE NING


Patterns of Local Contrast
Face Features
Face
Input Layer
Hidden Layer 1
Hidden Layer 2
Output Layer

RL ♥ DL

- Combine RL and DL
- Replace discrete tables with interpolation function
  - Can generalize better, don’t need to visit every state
  - Learns faster/better
  - Problems: Deadly Triad, Catastrophic forgetting, Complexity..
- Why now?
  - Powerful computers
  - Multicore, GPU board (Nvidia Geforce RTX 2080 Ti)
  - Libraries; TensorFlow, Matlab, PyTorch

Contextual bandits

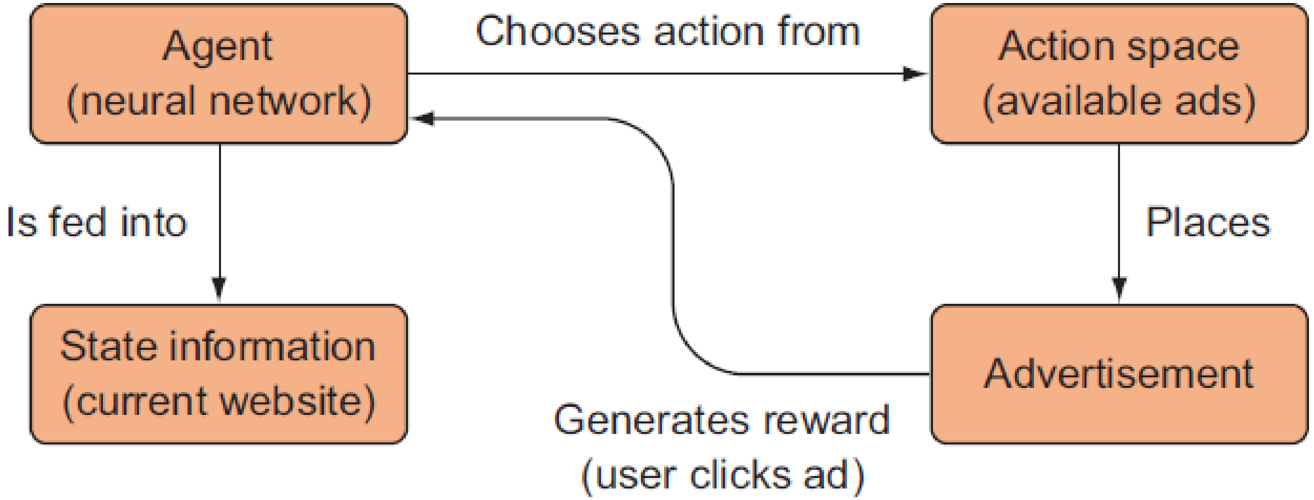
- Example: advertisement placement
    - maximize the probability that you will click ads depending on what site you visit
    - Let’s say we manage 10 e-commerce websites, each focusing on selling a different broad category of retail items such as computers, shoes, jewelry, etc.

We want to increase sales by referring customers who shop on one of our sites to another site that they might be interested in. When a customer checks out on a particular site in our network, we will display an advertisement to one of our other sites in hopes they'll go there and buy something else.

Alternatively, we could place an ad for another product on the same site. Our problem is that we don't know which sites we should refer users to. We could try placing random ads, but we suspect a more targeted approach is possible.

state spaces

- we know the user is buying something on a particular site, which may give us some information about that user's preferences and could help guide our decision about which ad to place



Building networks with PyTorch

A simple two-layer neural network with an Optimizer:

```
model = torch.nn.Sequential(
   torch.nn.Linear(10, 150),
   torch.nn.ReLU(),
   torch.nn.Linear(150, 4),
   torch.nn.ReLU(),
)
```

A training loop:

```
for step in range(100):
   y_pred = model(x)
   loss = loss_fn(y_pred, y_correct)
   optimizer.zero_grad()
   loss.backward()
   optimizer.step()
```

Our own Python class:

```
from torch.nn import Module, Linear
class MyNet(Module):
   def __init__(self):
      super(MyNet, self).__init__()
      self.fc1 = Linear(784, 50)
      self.fc2 = Linear(50, 10)
   def forward(self, x):
      x = F.relu(self.fc1(x))
      x = F.relu(self.fc2(x))
   return x
model = MyNet()
```

Exercice 1

- https://colab.research.google.com/
- https://github.com/nutte2/DeepRL
- bandits

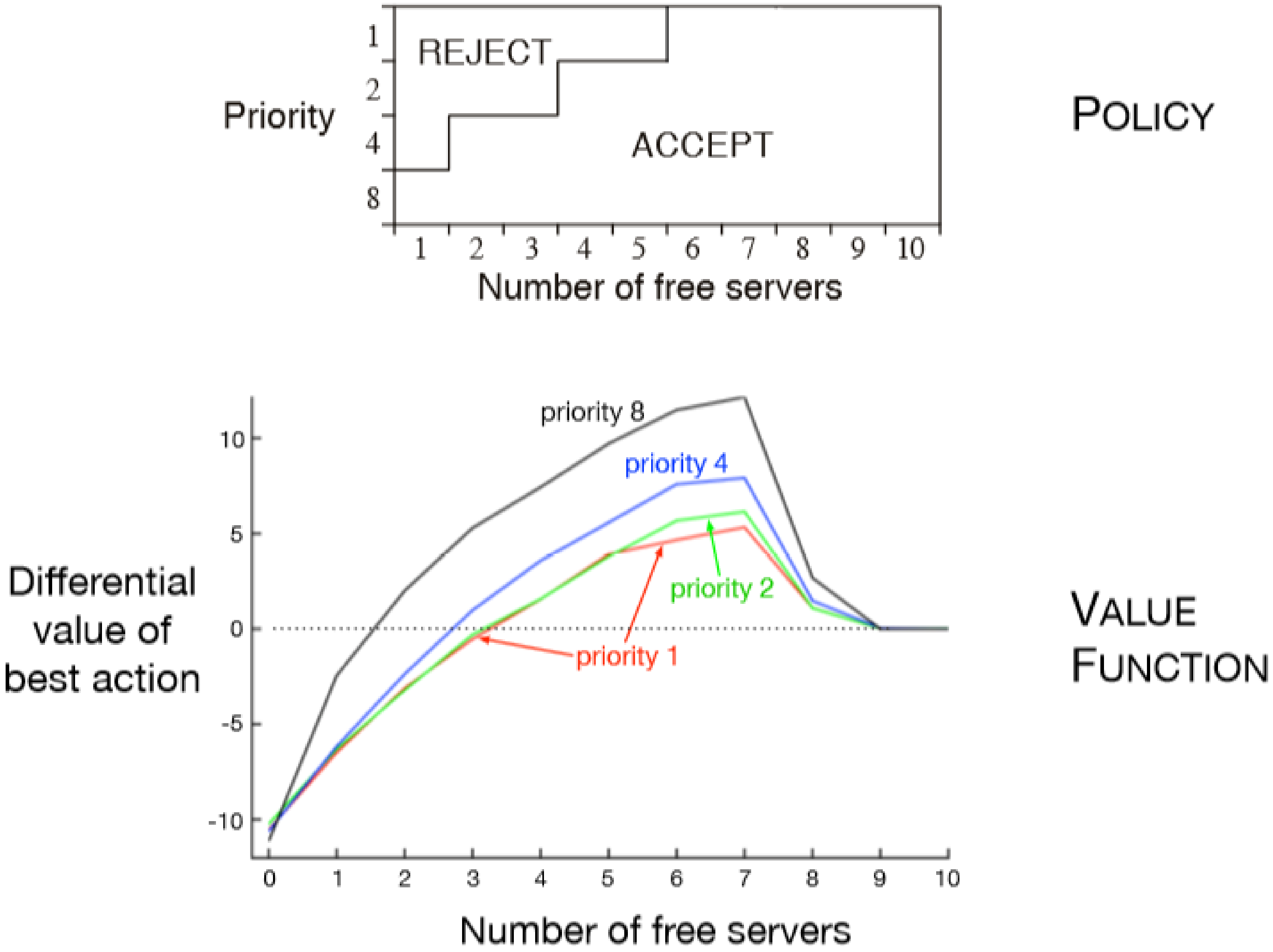


Figure 10.5: The policy and value function found by differential semi-gradient one-step Sarsa on the access-control queuing task after 2 million steps. The drop on the right of the graph is probably due to insufficient data; many of these states were never experienced. The value learned for $\bar{R}$ was about 2.31. ■

Example 5.3: Solving Blackjack It is straightforward to apply Monte Carlo ES to blackjack. Because the episodes are all simulated games, it is easy to arrange for exploring starts that include all possibilities. In this case one simply picks the dealer's cards, the player's sum, and whether or not the player has a usable ace, all at random with equal probability. As the initial policy we use the policy evaluated in the previous blackjack example, that which sticks only on 20 or 21. The initial action-value function can be zero for all state–action pairs. Figure 5.2 shows the optimal policy for blackjack found by Monte Carlo ES. This policy is the same as the "basic" strategy of Thorp (1966) with the sole exception of the leftmost notch in the policy for a usable ace, which is not present in Thorp's strategy. We are uncertain of the reason for this discrepancy, but confident that what is shown here is indeed the optimal policy for the version of blackjack we have described.

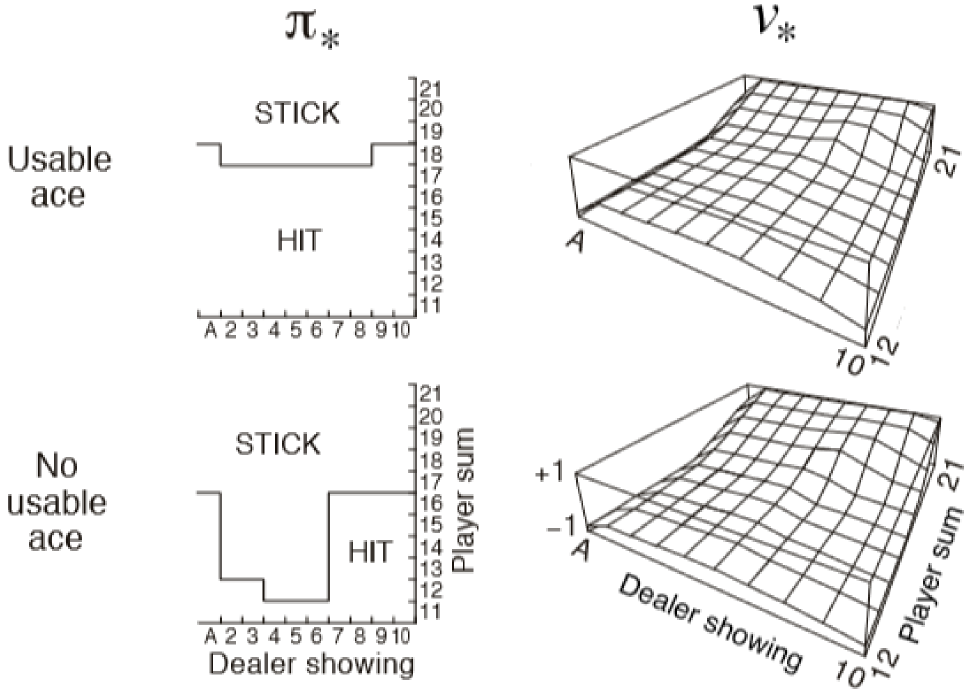


Figure 5.2: The optimal policy and state-value function for blackjack, found by Monte Carlo ES (Figure 5.4). The state-value function shown was computed from the action-value function found by Monte Carlo ES.

■

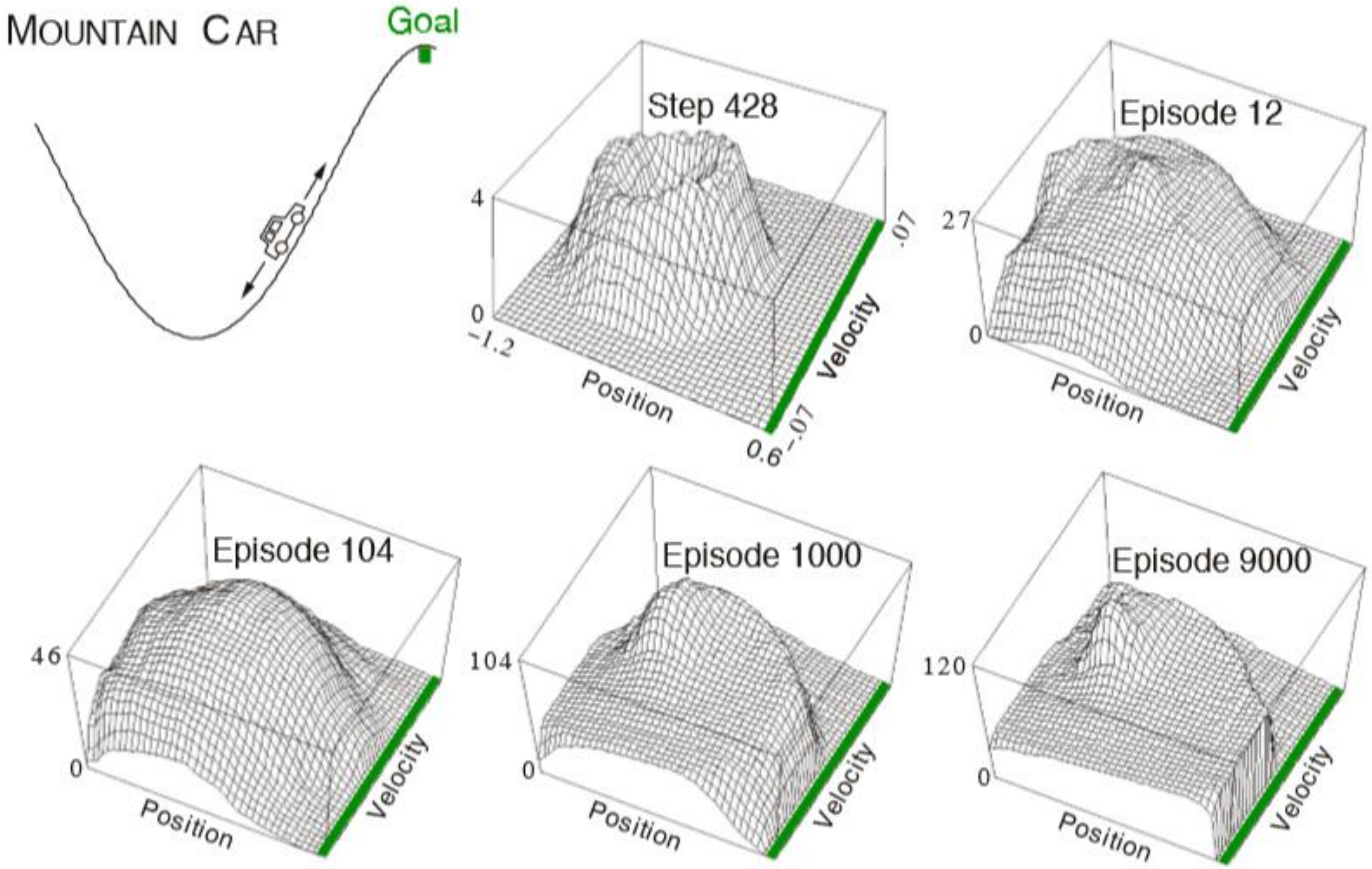
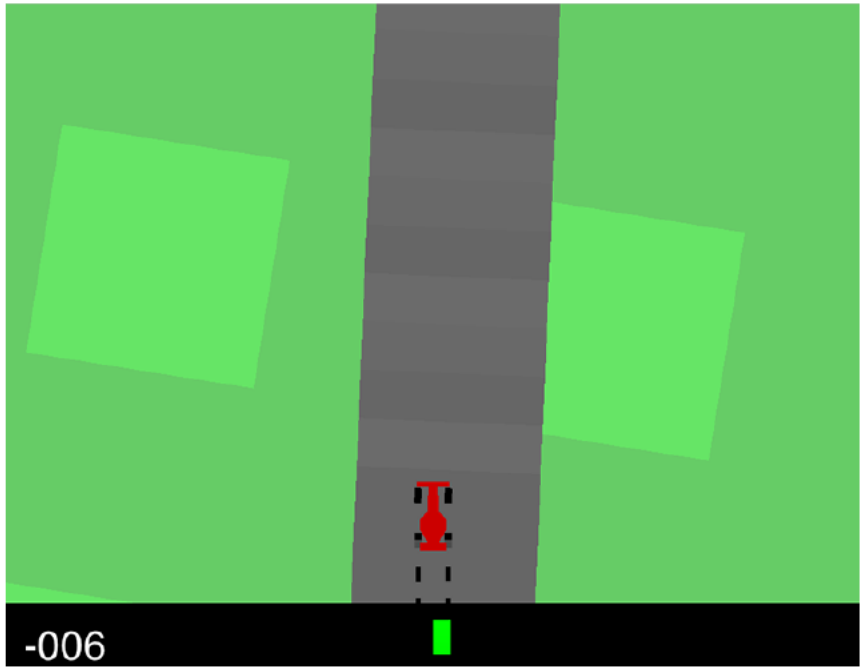


Figure 10.1: The Mountain Car task (upper left panel) and the cost-to-go function ($-\max_a \hat{q}(s, a, \mathbf{w})$) learned during one run.

Open AI Gym

```
import gym
env = gym.make('CarRacing-v0')
env.reset()
env.step(action)
env.render()
```



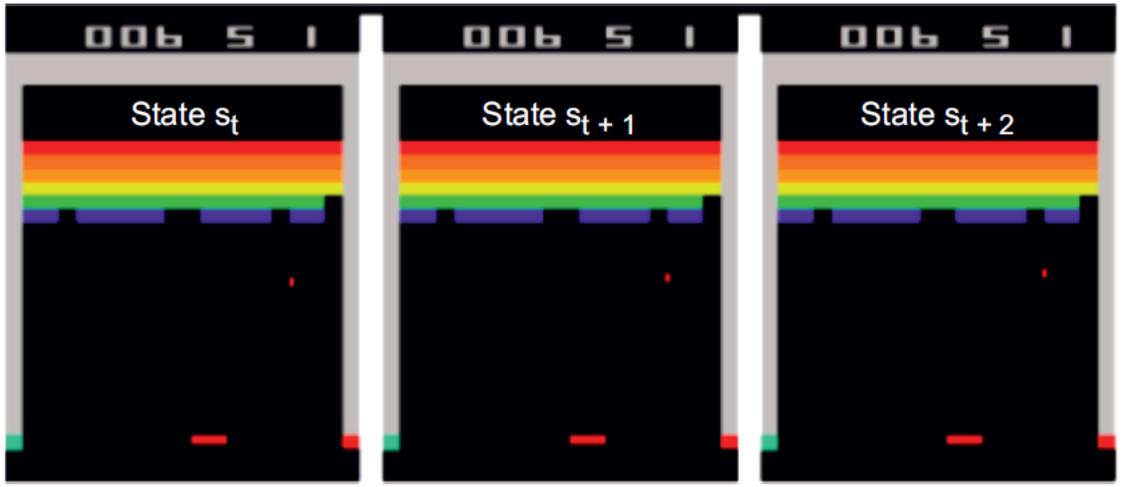
Exercice 2

- https://colab.research.google.com/
- https://github.com/nutte2/DeepRL
- mountcar

GOOGLE DEEPMINDS ALPHAGO ZERO

- Beat the previous version of AlphaGo (Final score: 100–0).
- Learn to perform this task from scratch, without learning from previous human knowledge (i.e. recorded game play).
- World champion level Go playing in just 3 days of training.
- Do so with an order of magnitude less neural networks (4 TPUs vs 48 TPUs).
- Do this with less training data (3.9 million games vs 30 millions games).
- https://deepmind.com/blog/article/alphago-zero-starting-scratch
- The Evolution of AlphaGo to MuZero

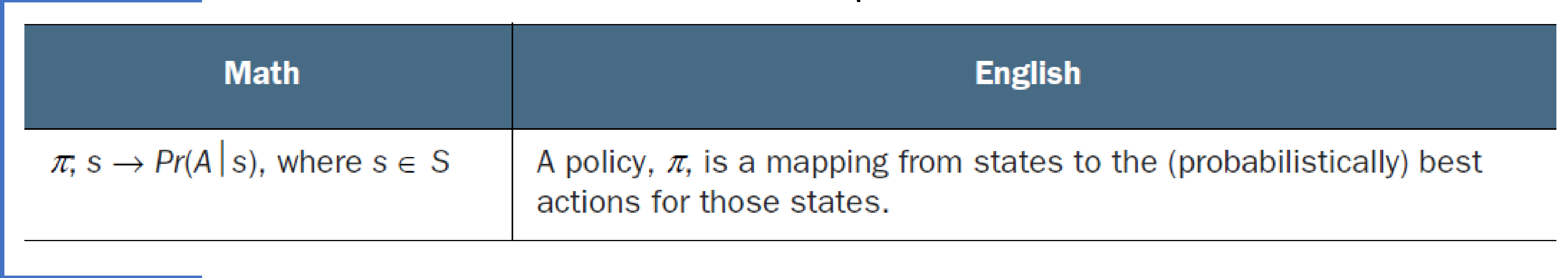
DeepMind’s DQN algorithm for Atari games



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

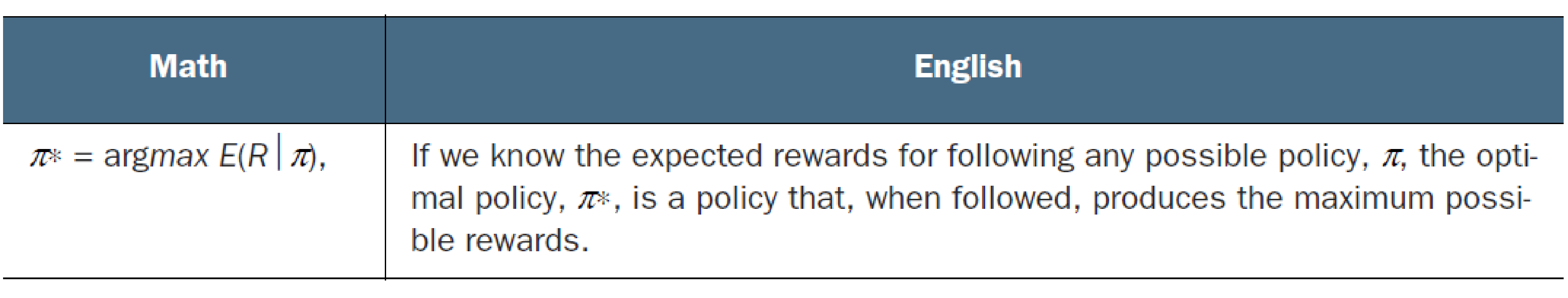
Policy functions

- How exactly do we use our current state information to decide what action to take?
- In words, a policy, π, is the strategy of an agent in some environment.
- A policy is a function that maps a state to a probability distribution over the set of possible actions in that state



Math	English
π, $s \rightarrow Pr(A \mid s)$, where $s \in S$	A policy, π, is a mapping from states to the (probabilistically) best actions for those states.

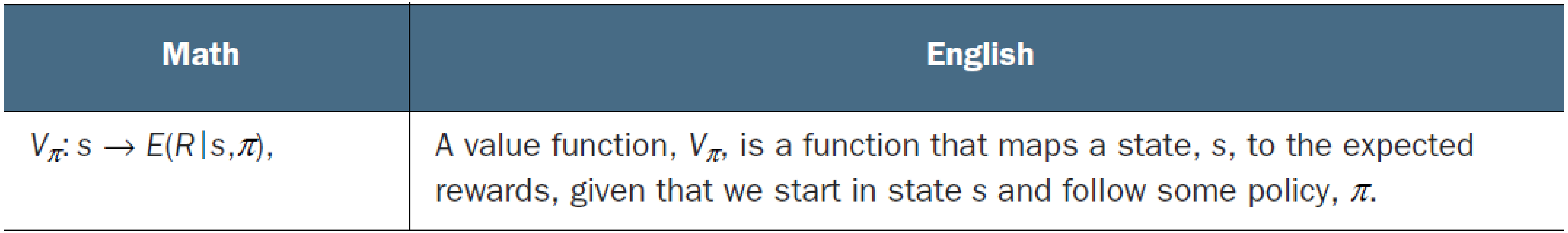
- The *optimal policy*—it's the strategy that maximizes rewards.



Math	English
$\pi* = \text{argmax } E(R \mid \pi)$,	If we know the expected rewards for following any possible policy, π, the optimal policy, $\pi*$, is a policy that, when followed, produces the maximum possible rewards.

Value functions

- *Value functions* are functions that map a state to the *expected value* (the expected reward) of being in some state

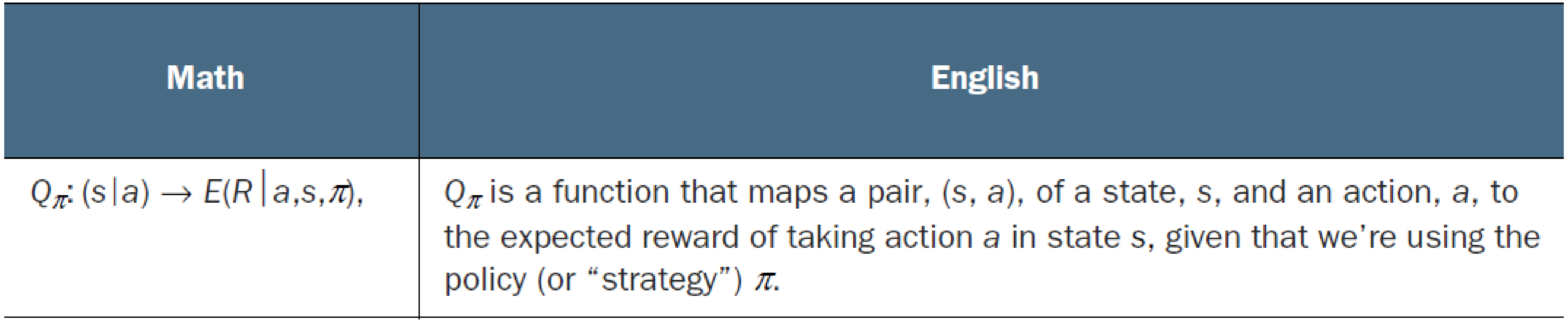


Math	English
$V_\pi: s \rightarrow E(R \mid s, \pi)$,	A value function, V_π, is a function that maps a state, s, to the expected rewards, given that we start in state s and follow some policy, π.

- Accepts a state, *s*, and returns the expected reward of starting in that state and taking actions according to our policy, π
- Actual values depends on the policy

Q function or Q value

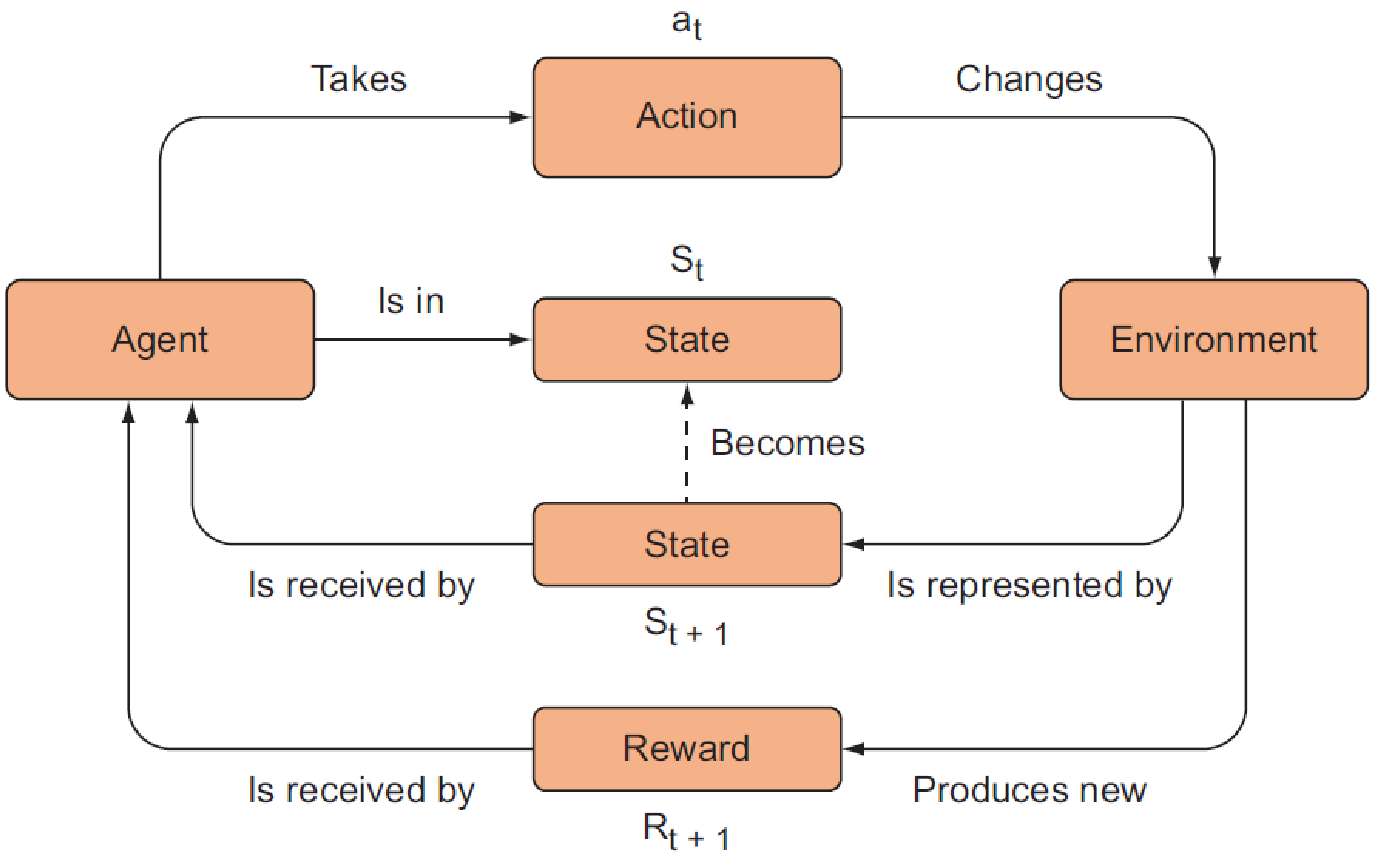
- Estimates of the expected reward for taking an action given a state



Math	English
$Q_\pi: (s\|a) \rightarrow E(R\|a,s,\pi)$,	Q_π is a function that maps a pair, (*s*, *a*), of a state, *s*, and an action, *a*, to the expected reward of taking action *a* in state *s*, given that we're using the policy (or "strategy") π.

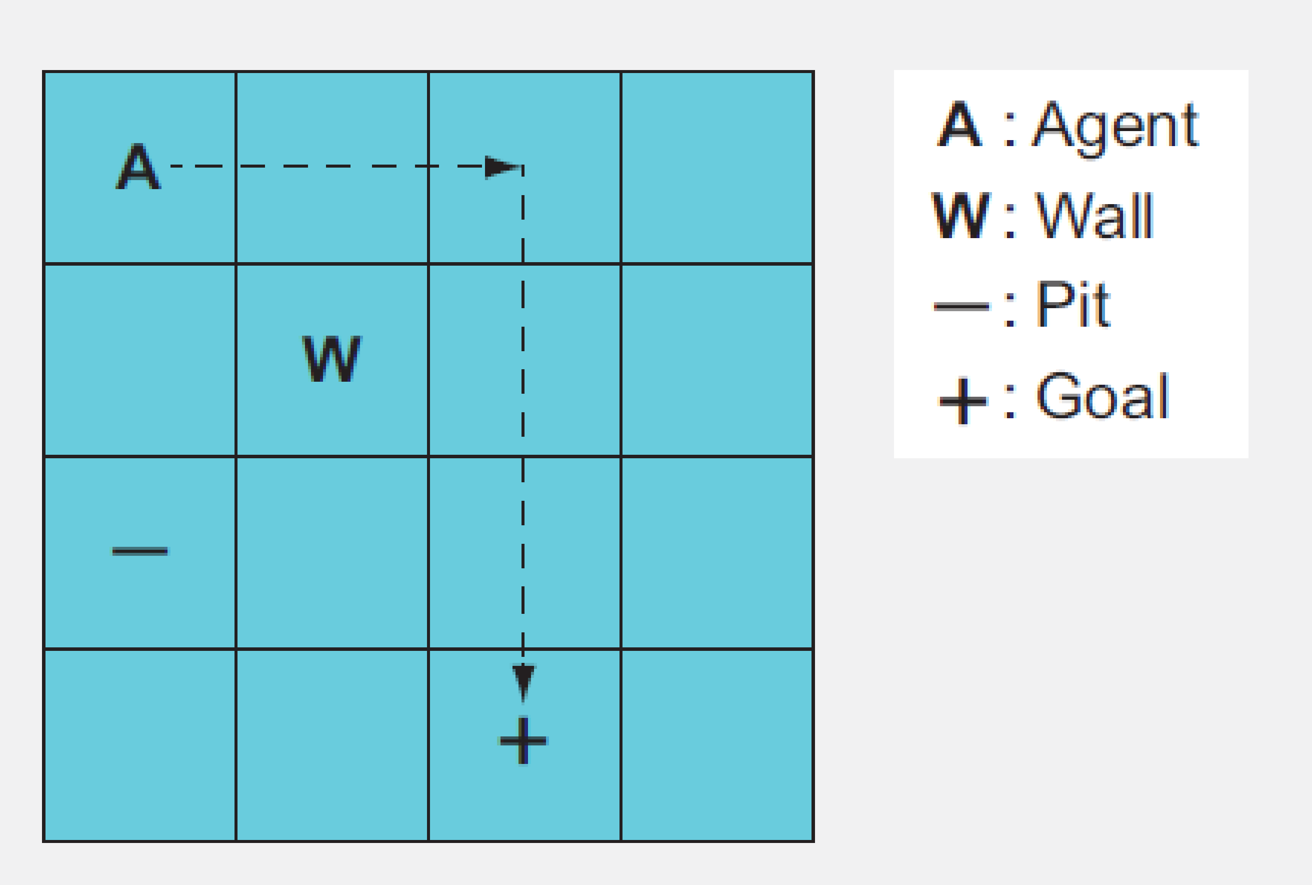
- Q-learning (Watkins 1989, 1992)
- https://en.wikipedia.org/wiki/Q-learning

Agent – Action – Environment – State/Reward



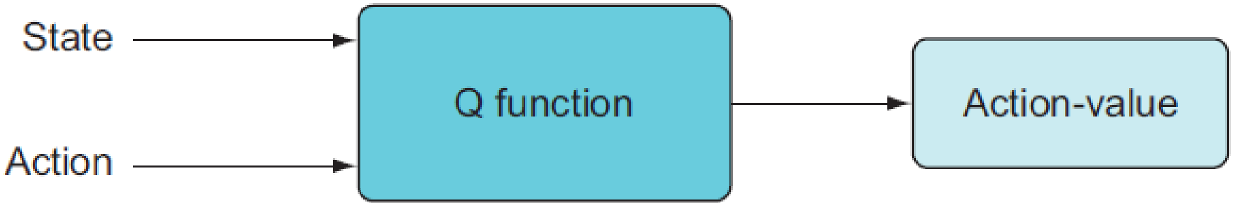
A simple version of Gridworld

- Train a DRL agent to navigate the Gridworld board to the goal
- Following the most efficient route



Sequence of events for a game of Gridworld

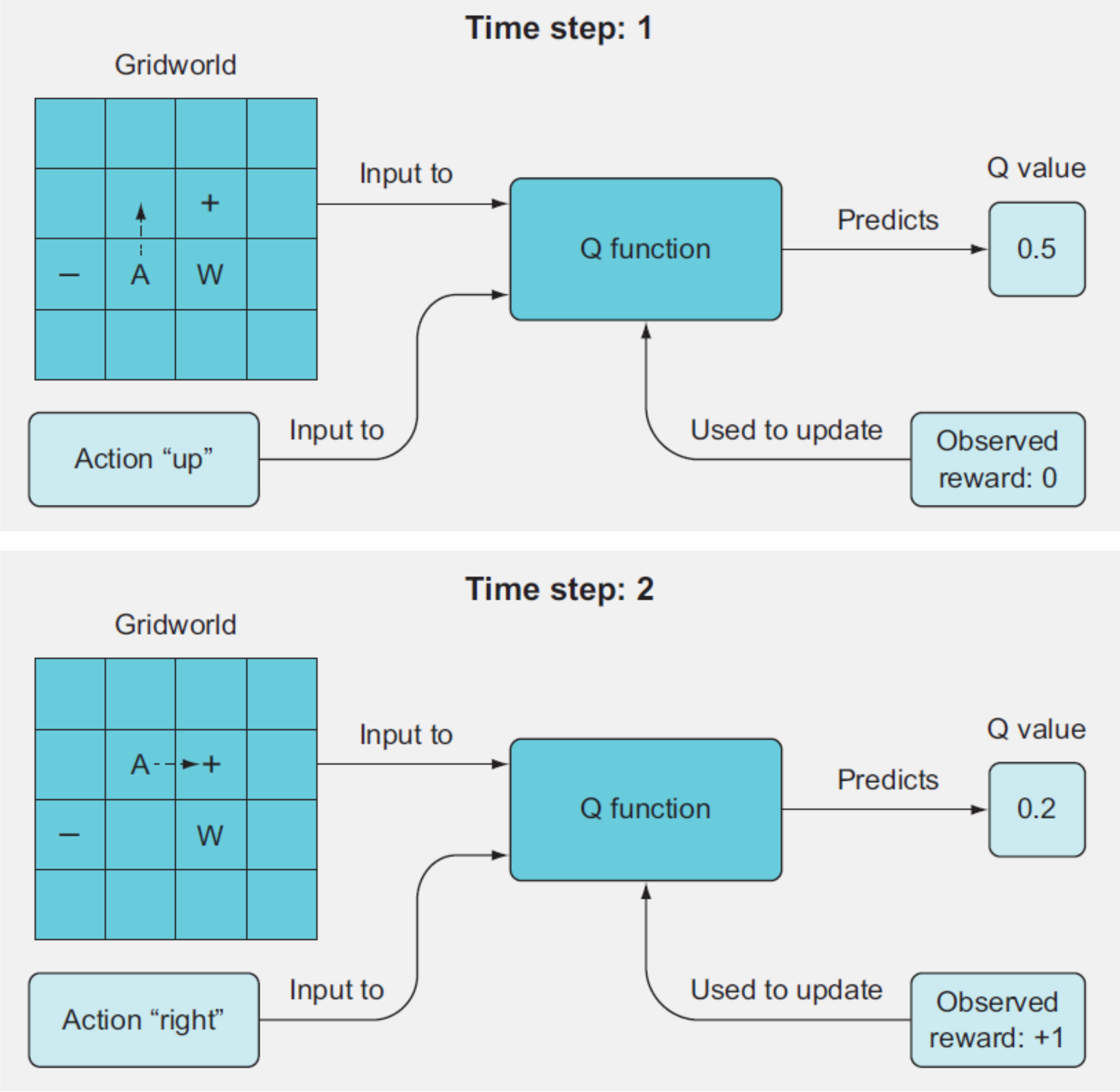
- We start the game in some state that we'll call S_t. The state includes all the information about the game that we have. For our Gridworld example, the game state is represented as a 4 × 4 × 4 tensor.
- We feed the S_t data and a candidate action into a deep neural net and it produces a prediction of how valuable taking that action in that state is



- The algorithm is not predicting the reward we will get after taking a particular action; it's predicting the expected value (the expected rewards), which is the long-term average reward we will get from taking an action in a state and then continuing to behave according to our policy π.

We take an action,

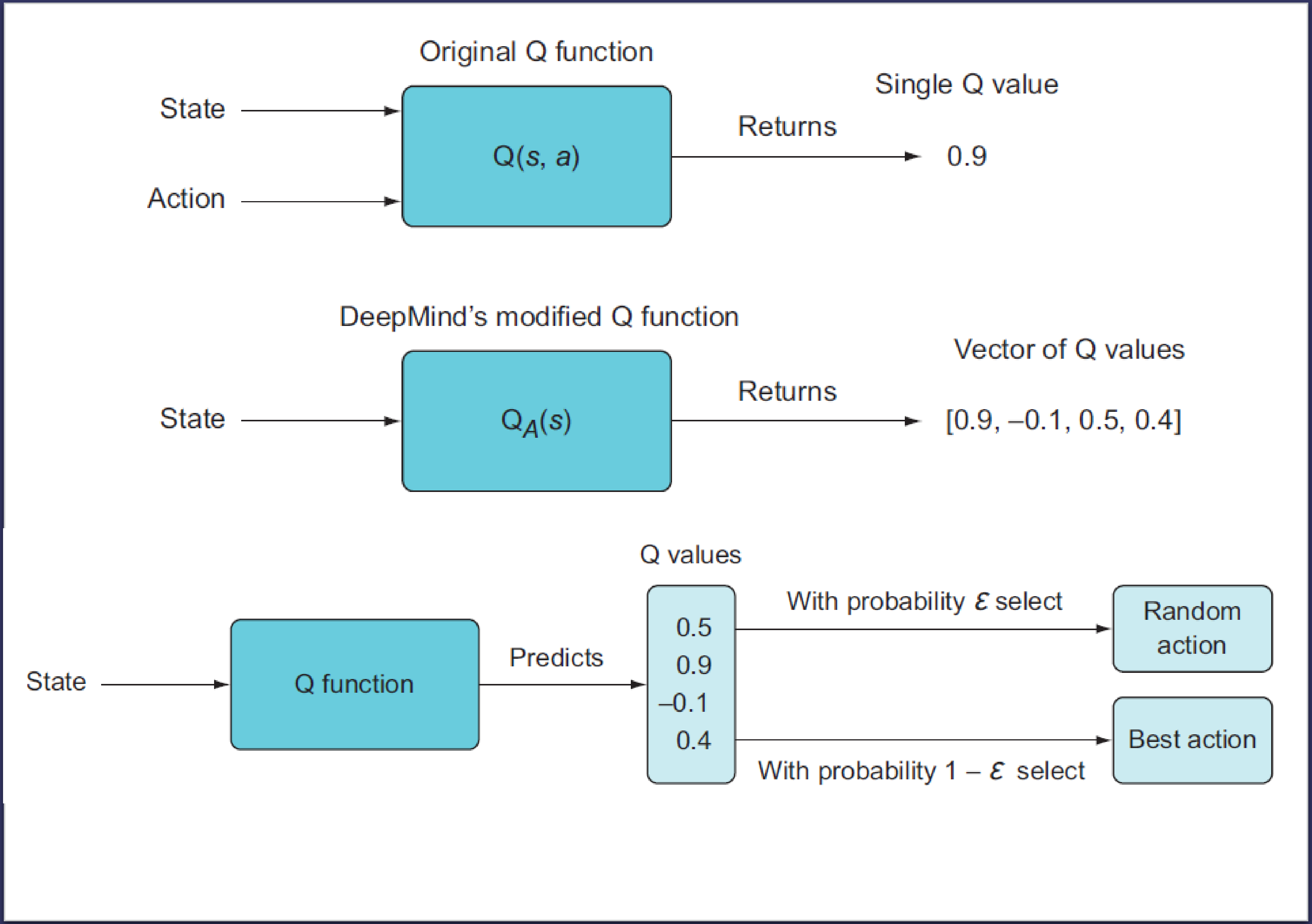
- perhaps because our neural network predicted it is the highest value action or perhaps we take a random action. We'll label the action A_t. We are now in a new state of the game, which we'll call S_{t+1}, and we receive or observe a reward, labelled R_{t+1}. We want to update our learning algorithm to reflect the actual reward we received, after taking the action it predicted was the best. Perhaps we got a negative reward or a really big reward, and we want to improve the accuracy of the algorithm's predictions



Updating Q(S_t ,a)

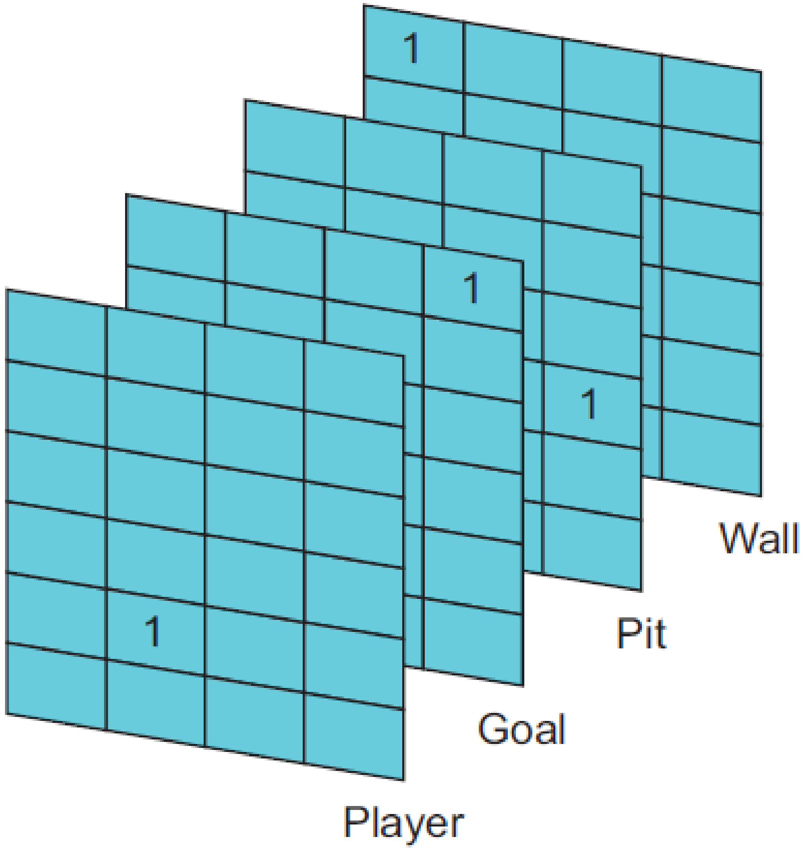
- Now we run the algorithm using S_{t+1} as input and figure out which action our algorithm predicts has the highest value. We'll call this value max $Q(S_{t+1},a)$. To be clear, this is a single value that reflects the highest predicted *Q* value, given our new state and all possible actions there
- We'll perform one iteration of training using some loss function, such as mean squared error, to minimize the difference between the predicted value from our algorithm and the target prediction of

$$Q(S_t , A_t) + \alpha *[\mathrm{R}_{t+1} + \gamma *\overset{a}{\mathrm{max}}\, Q(S_{t+1} ,A)– Q(S_t , A_t)]$$

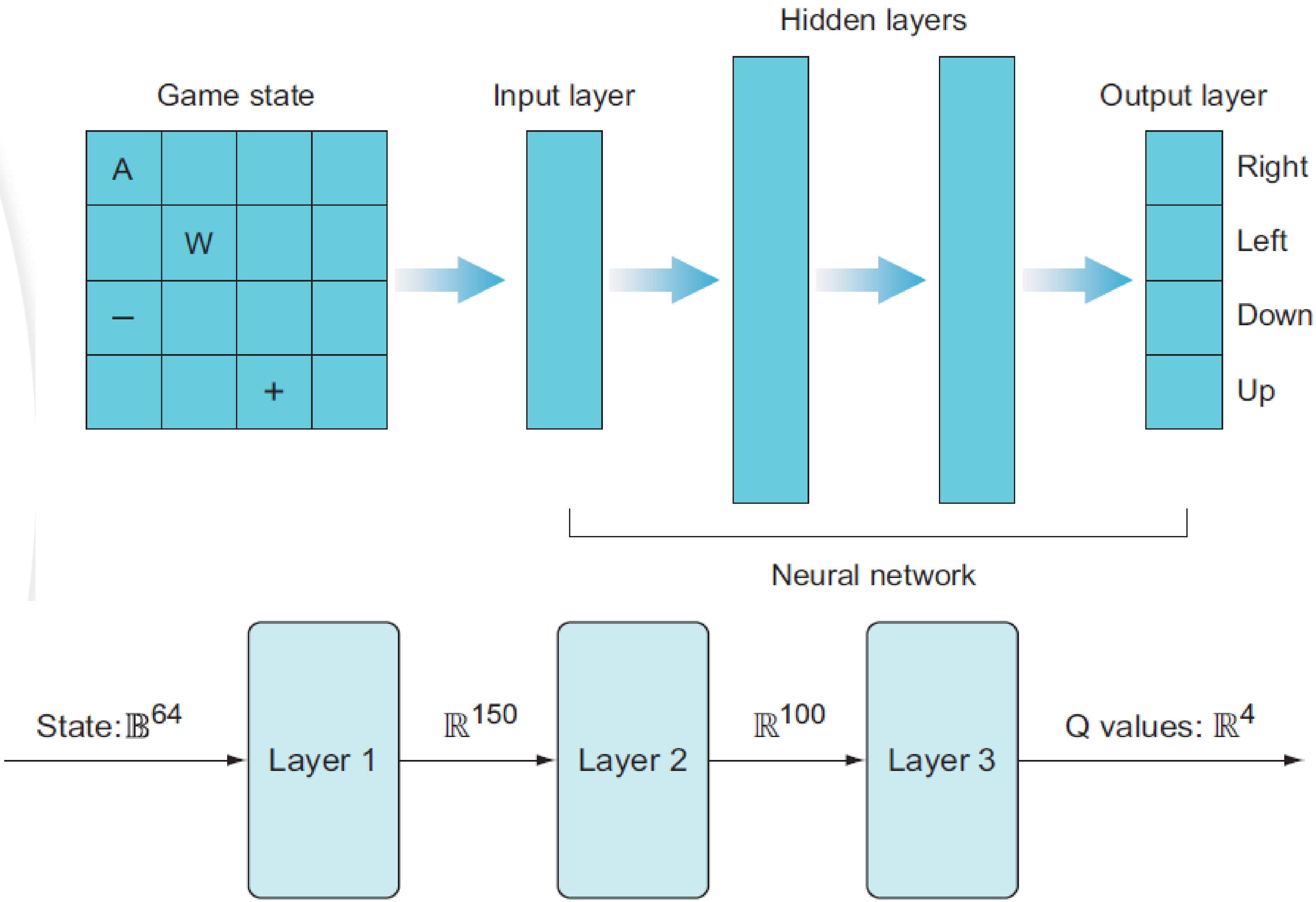


DeepMind used a modified vector-valued Q function that accepts a state and returns a vector of state-action values, one for each possible action given the input state. The vector-valued Q function is more efficient, since you only need to compute the function once for all the actions

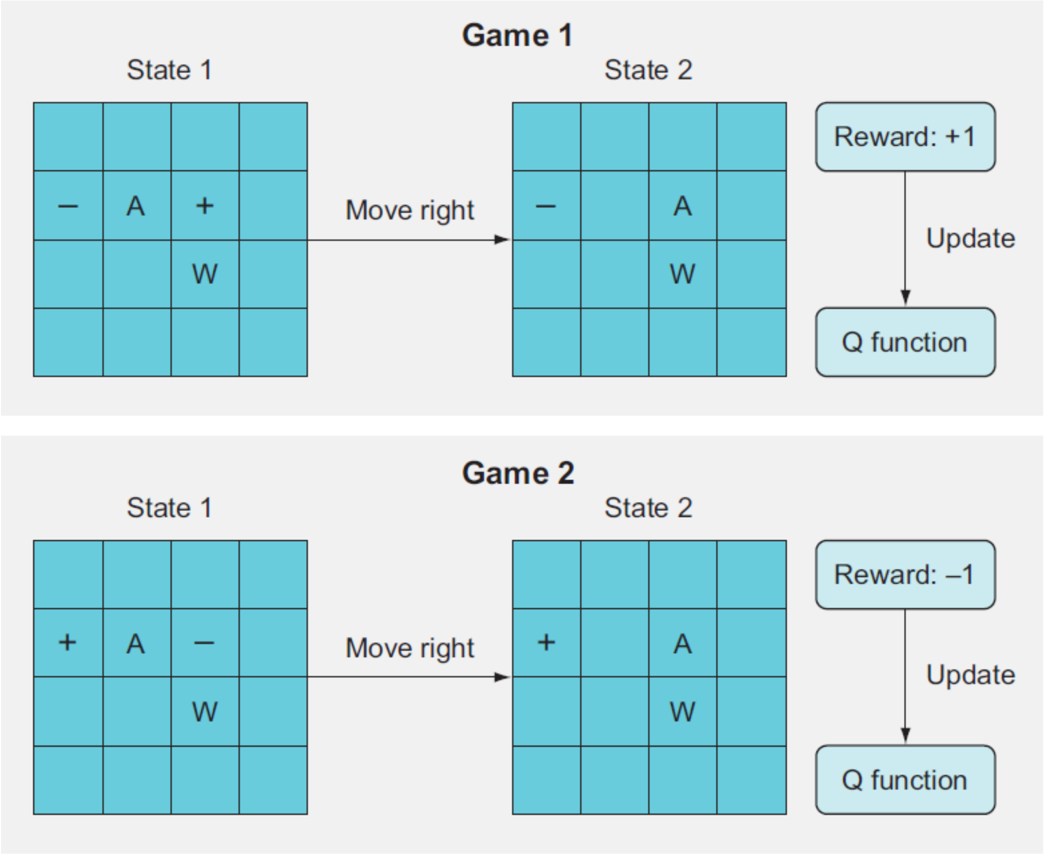
The Gridworld board is represented as a numpy array. It is a 4 x 4 x 4 tensor, composed of 4 “slices”



The general architecture for the model we will build.



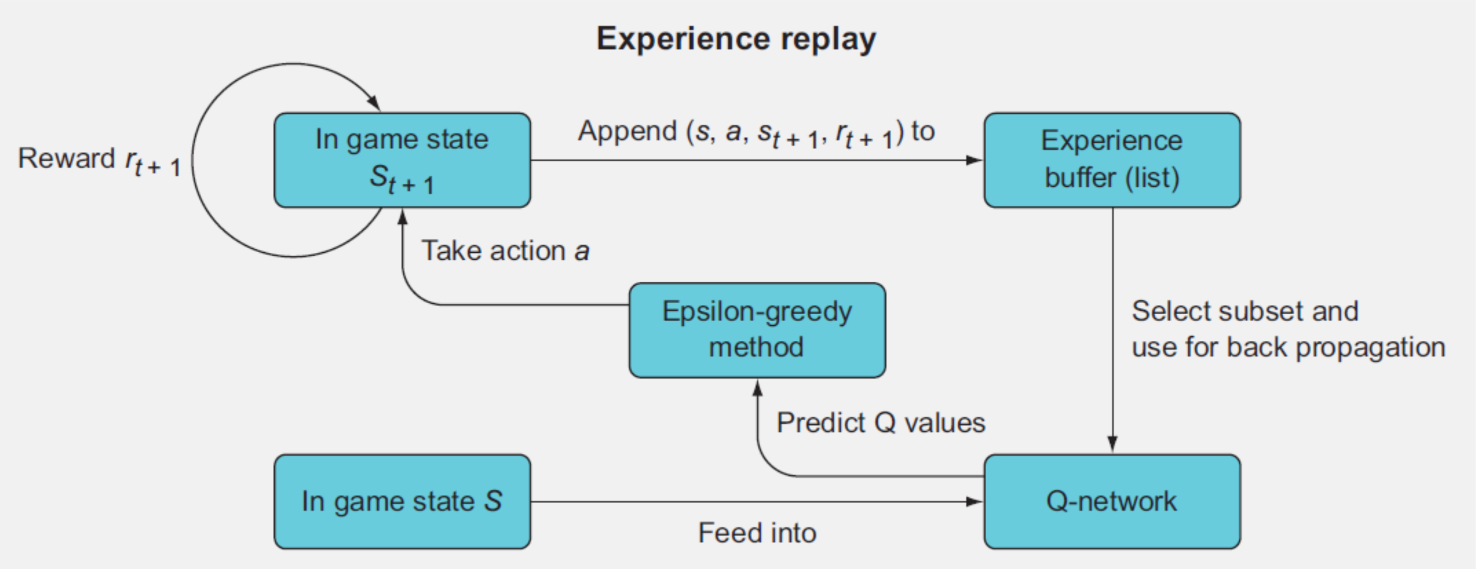
Catastrophic forgetting

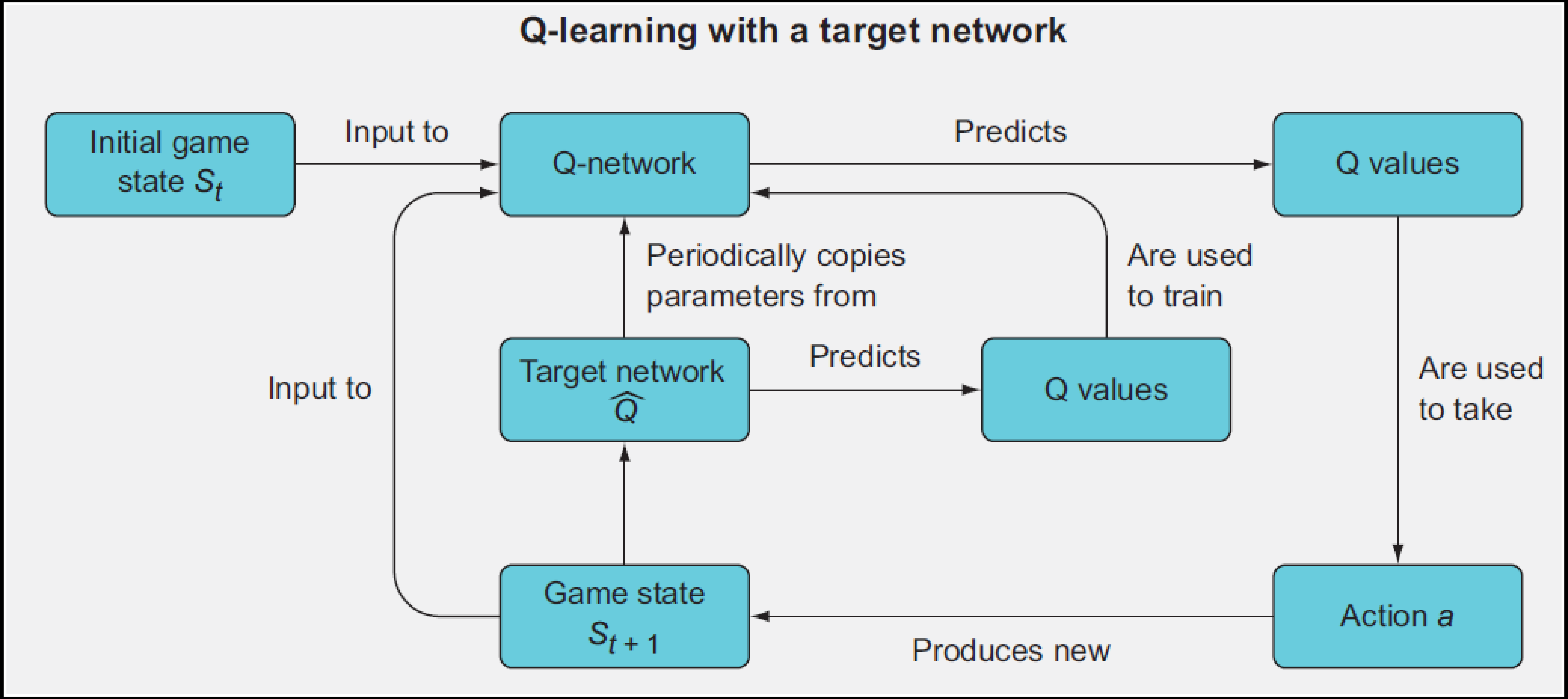


two game states are very similar and yet lead to very different outcomes, the Q function will get “confused” and won’t be able to learn what to do

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





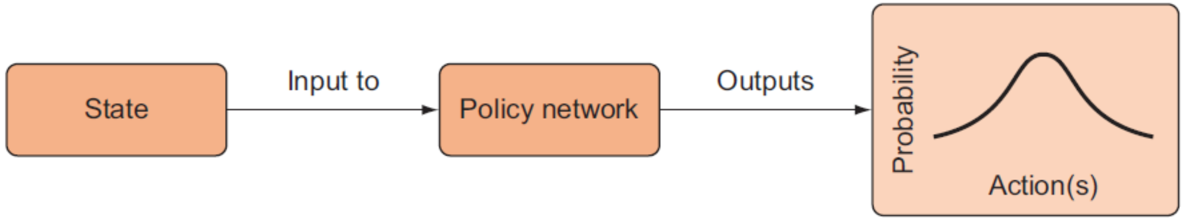
Improving stability with a target network

Exercice 3

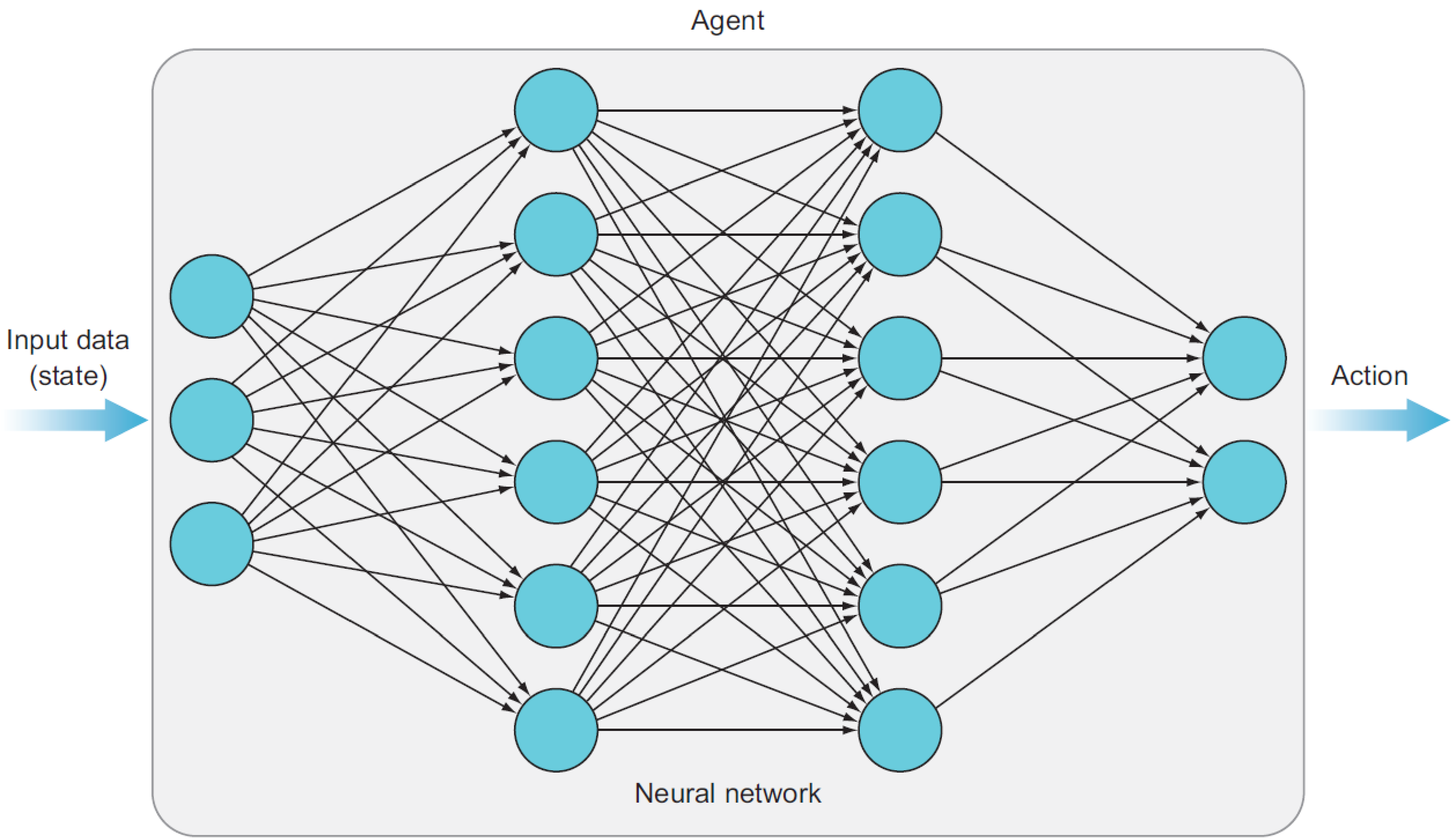
- https://colab.research.google.com/
- https://github.com/nutte2/DeepRL
- grids

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*

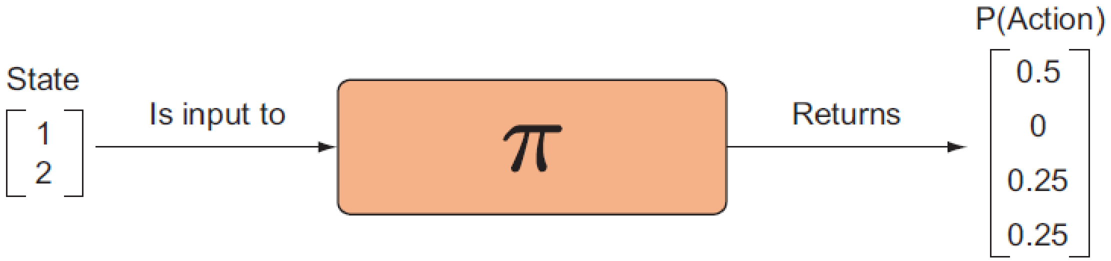


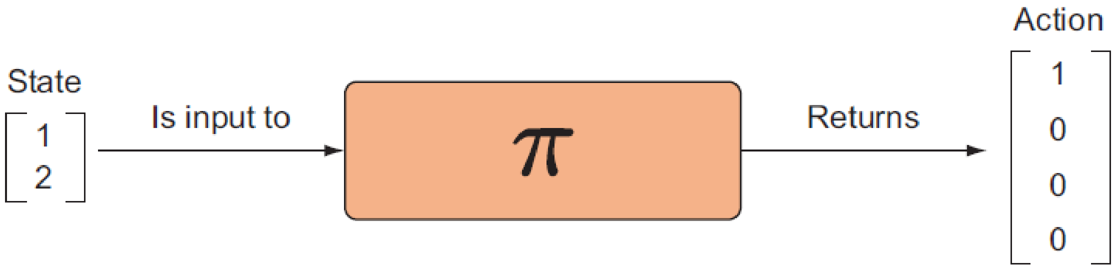
Agent

Input data (state)

Action

Neural network

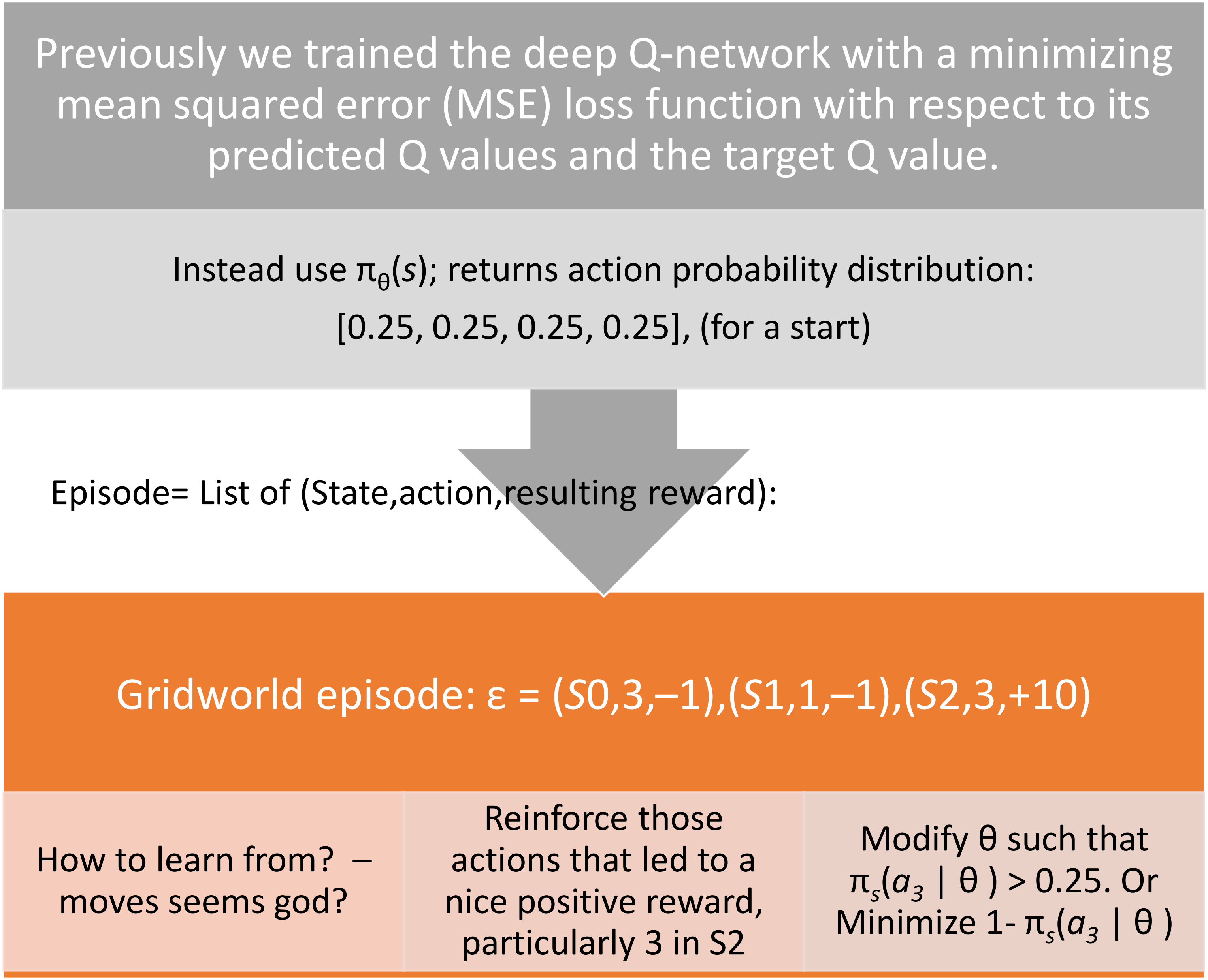




Eventually converges to a *degenerate probability distribution.* it is difficult to get this working in the fully differentiable manner that we are accustomed to with deep learning; avoid

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: $\varepsilon = (S0,3,-1),(S1,1,-1),(S2,3,+10)$

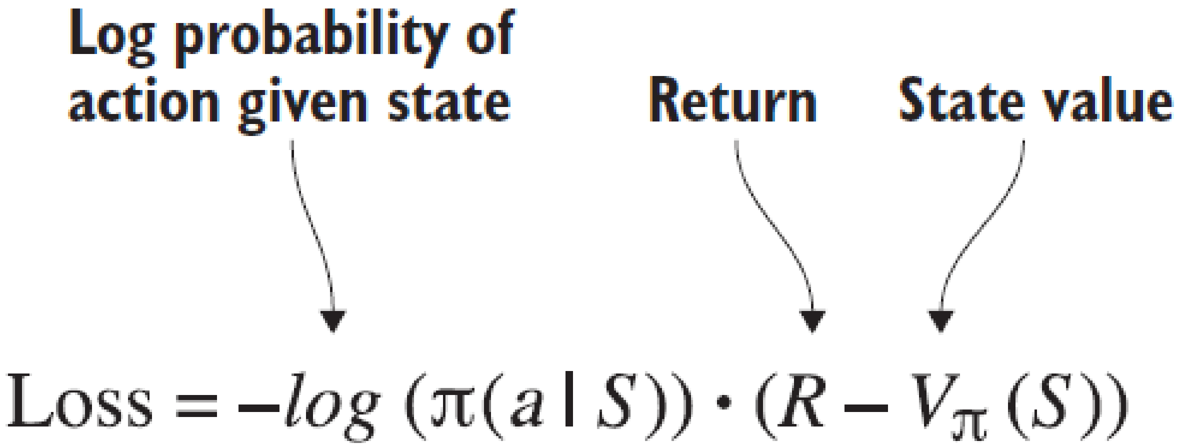
How to learn from? – moves seems god?

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

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

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:



Log probability of action given state

Return

State value

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

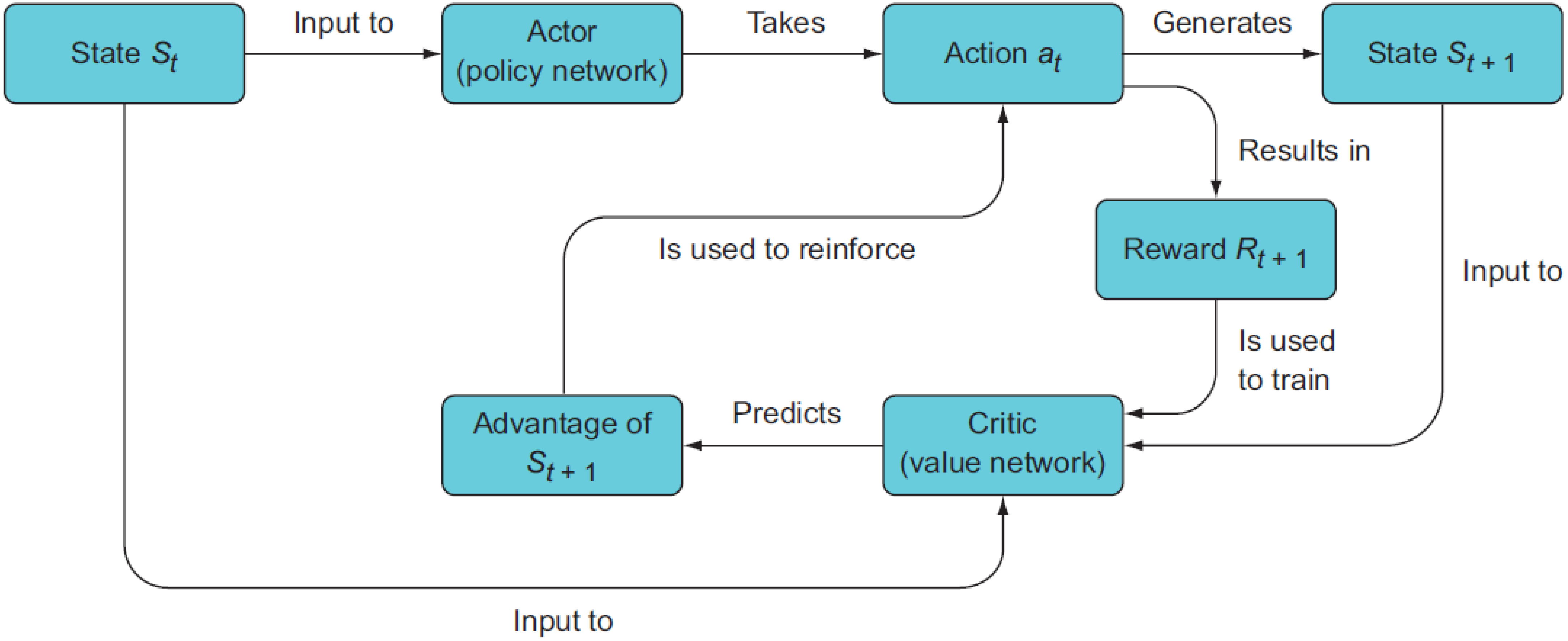
- V(*S*) – *R*, is termed the *advantage*.
- the advantage quantity tells you how much better an action is, relative to what you expected.

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

General overview of actor-critic models



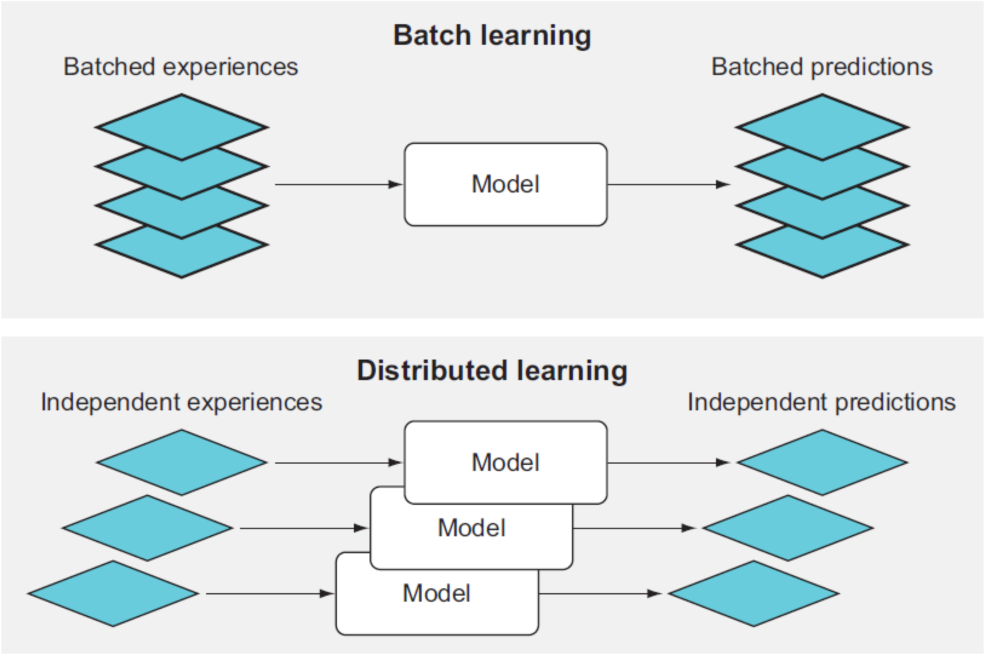
Distributed training

- **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

Some DRL links

- Deep Reinforcement Learning: Pong from Pixels
  - http://karpathy.github.io/2016/05/31/rl/
  - https://arxiv.org/pdf/1312.5602.pdf
- the DRL book's GitHub repository: http://mng.bz/JzKp
- AAMAS conference 9-13 may: https://aamas2020.conference.auckland.ac.nz/
- Tensorflow TF-Agents
  - https://colab.research.google.com/github/tensorflow/agents/blob/master/docs/tutorials/1_dqn_tutorial.ipynb
- Real application: Adaptive Power System Emergency Control using Deep Reinforcement Learning
  - https://arxiv.org/pdf/1903.03712.pdf