Deep Reinforcement Learning - 3

Kalle Prorok, Umeå 2020

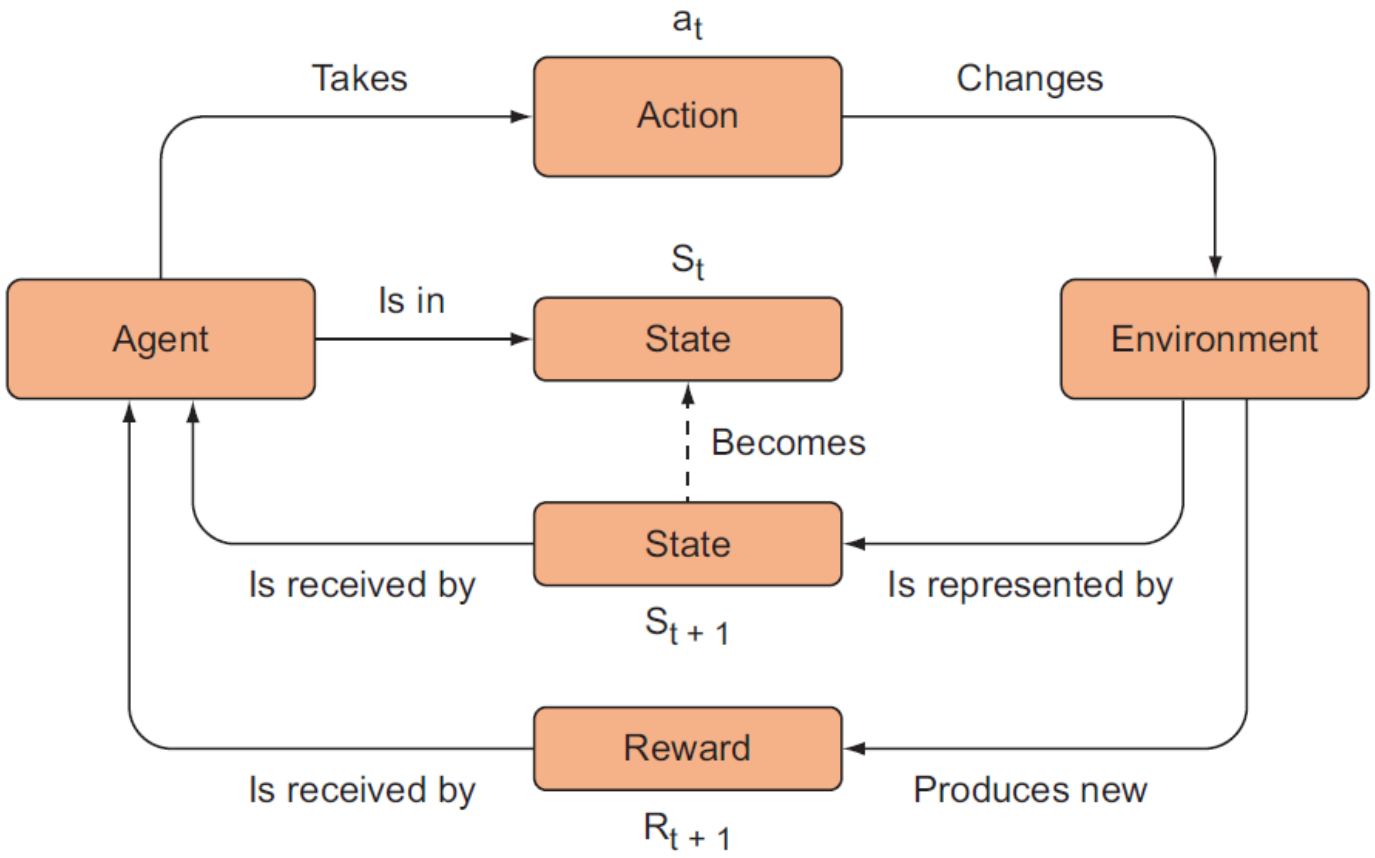
Parts based on the Manning book

Deep Reinforcement Learning in Action

By Alexander Zai and Brandon Brown

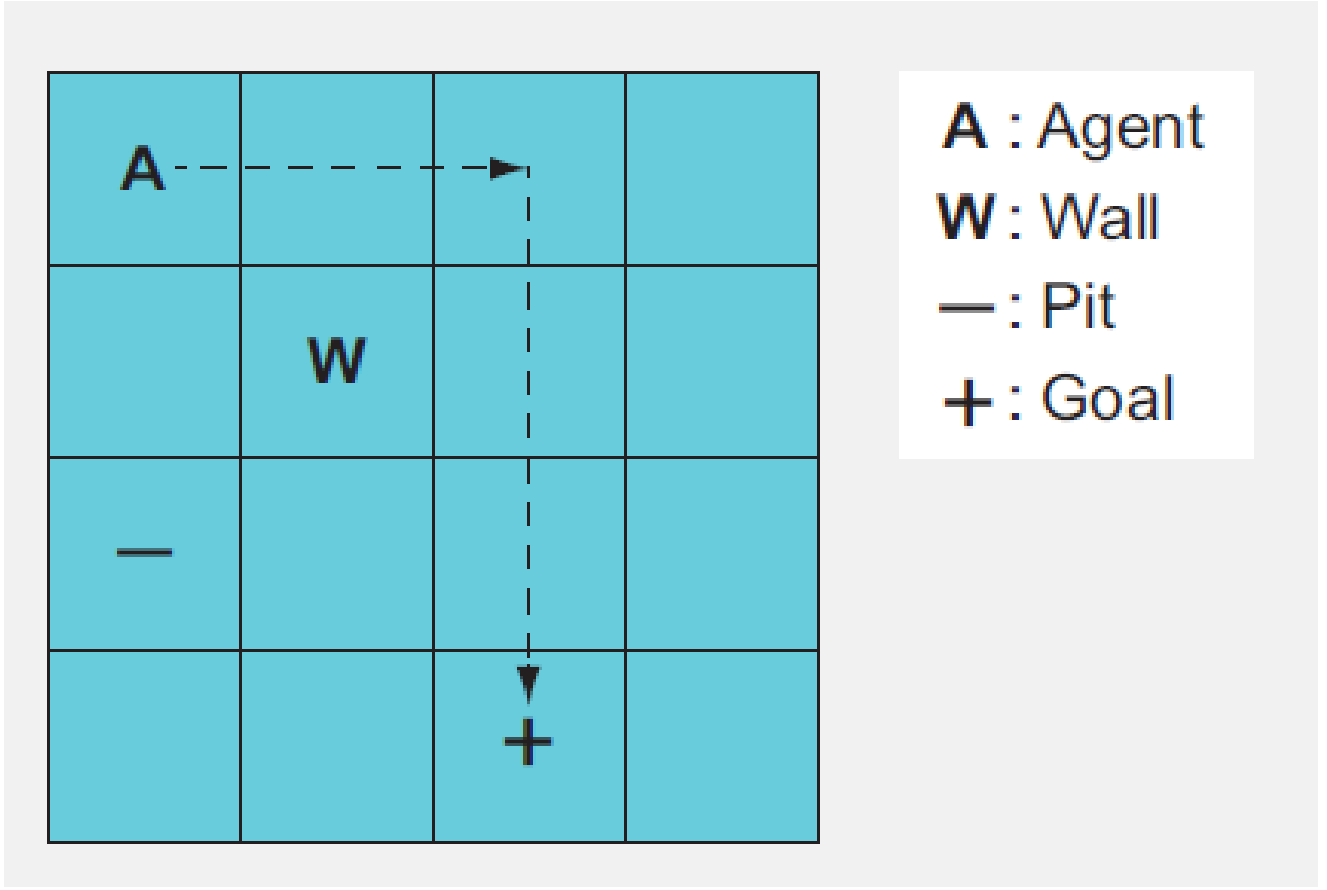
And the Sutton-Barto book

Agent – Action – Environment – State/Reward



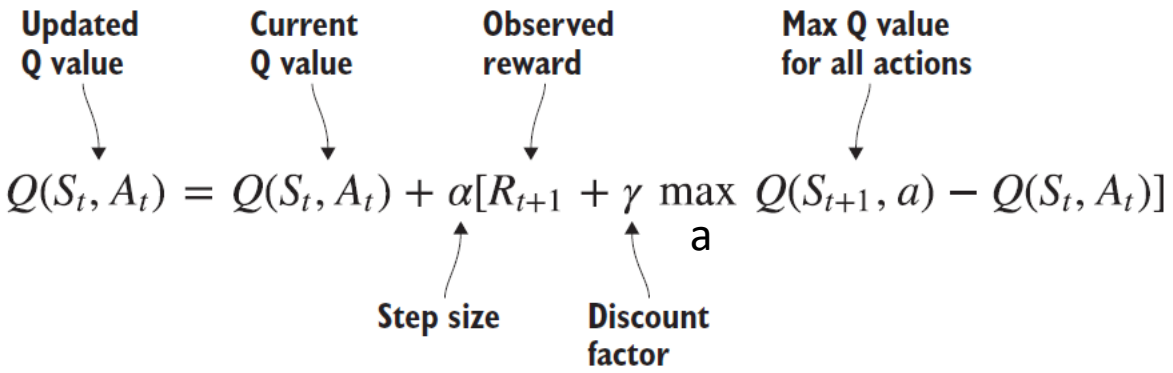
A simple version of Gridworld

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



Q-learning algorithm update rule

Math



Pseudocode

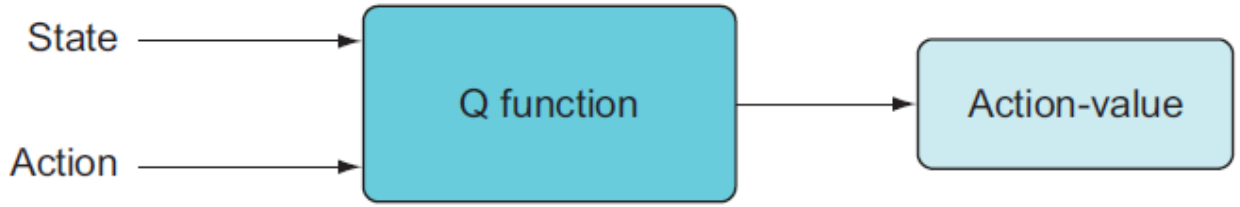
```
def get_updated_q_value(old_q_value, reward, state, step_size, discount):
     term2 = (reward + discount * max([Q(state, action) for action in
             actions])
     term2 = term2 - old_q_value
     term2 = step_size * term2
     return (old_q_value + term2)
```

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.

Sequence of events for a game of Gridworld

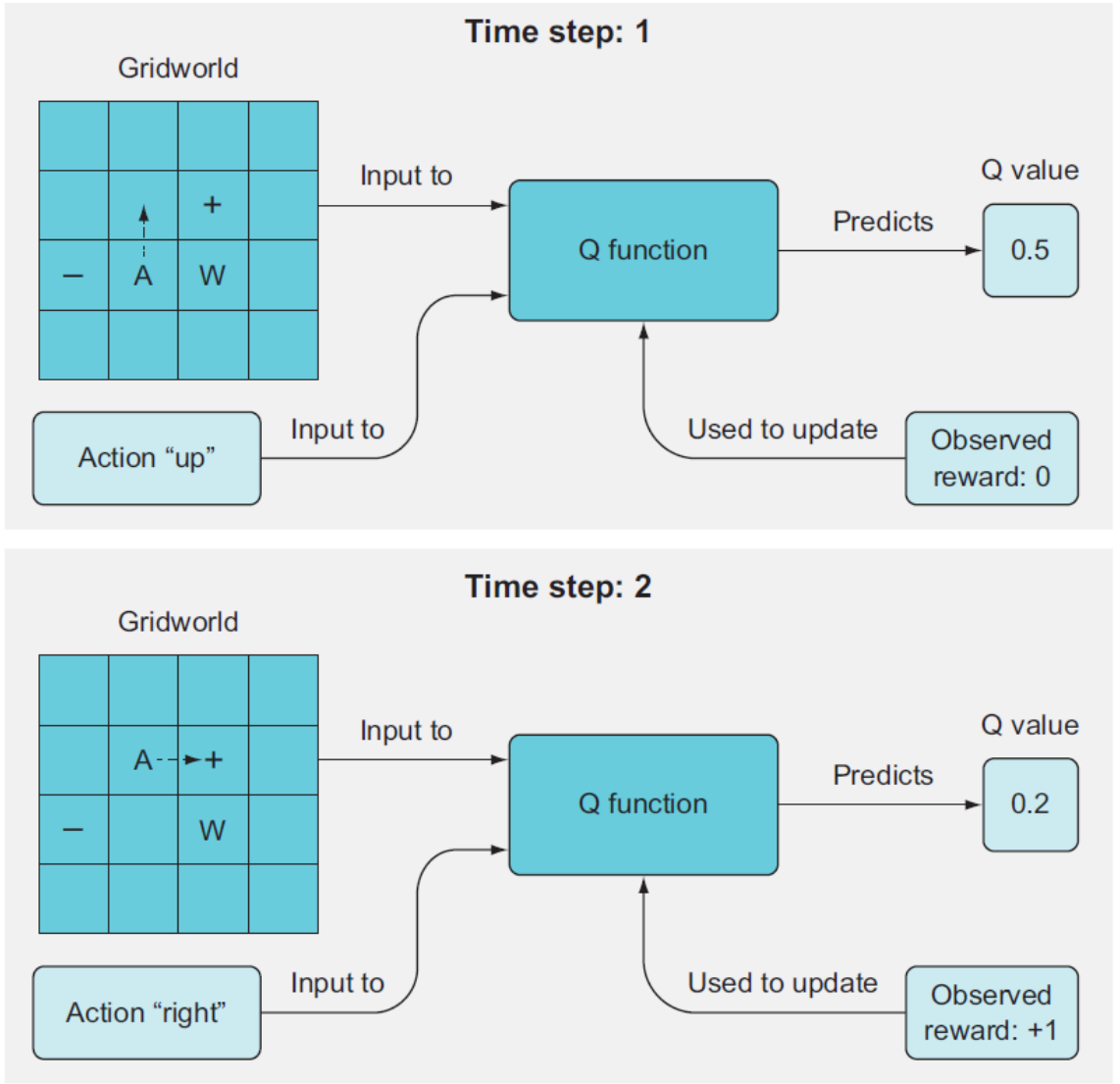
- We start the game in some state that we'll call S_t. The state includes all the information about the game that we have. For our Gridworld example, the game state is represented as a $4 \times 4 \times 4$ tensor.
- We feed the S_t data and a candidate action into a deep neural net and it produces a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

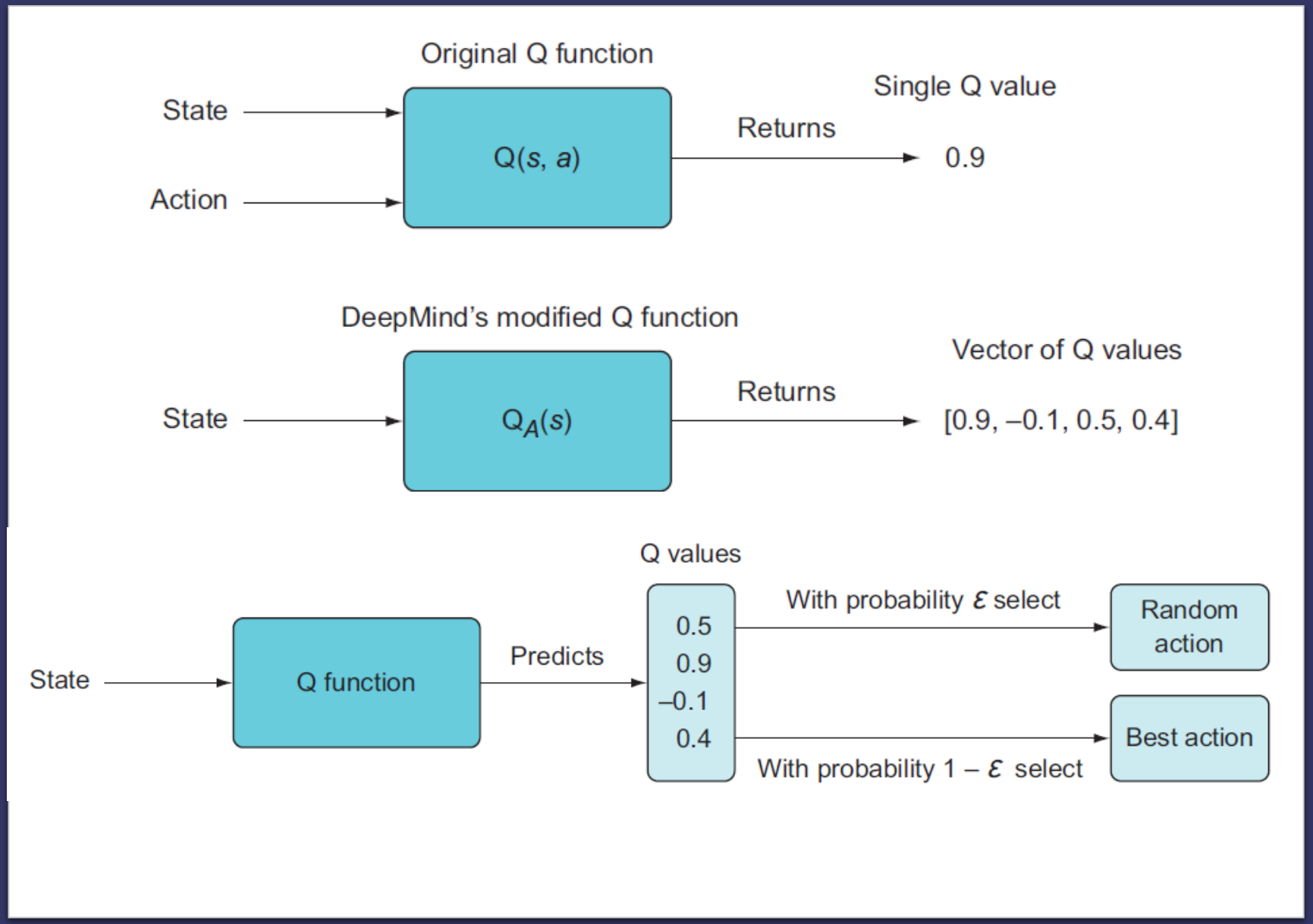
- perhaps because our neural network predicted it is the highest value action or perhaps we take a random action. We'll label the action A_t. We are now in a new state of the game, which we'll call S_{t+1}, and we receive or observe a reward, labelled R_{t+1}. We want to update our learning algorithm 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)

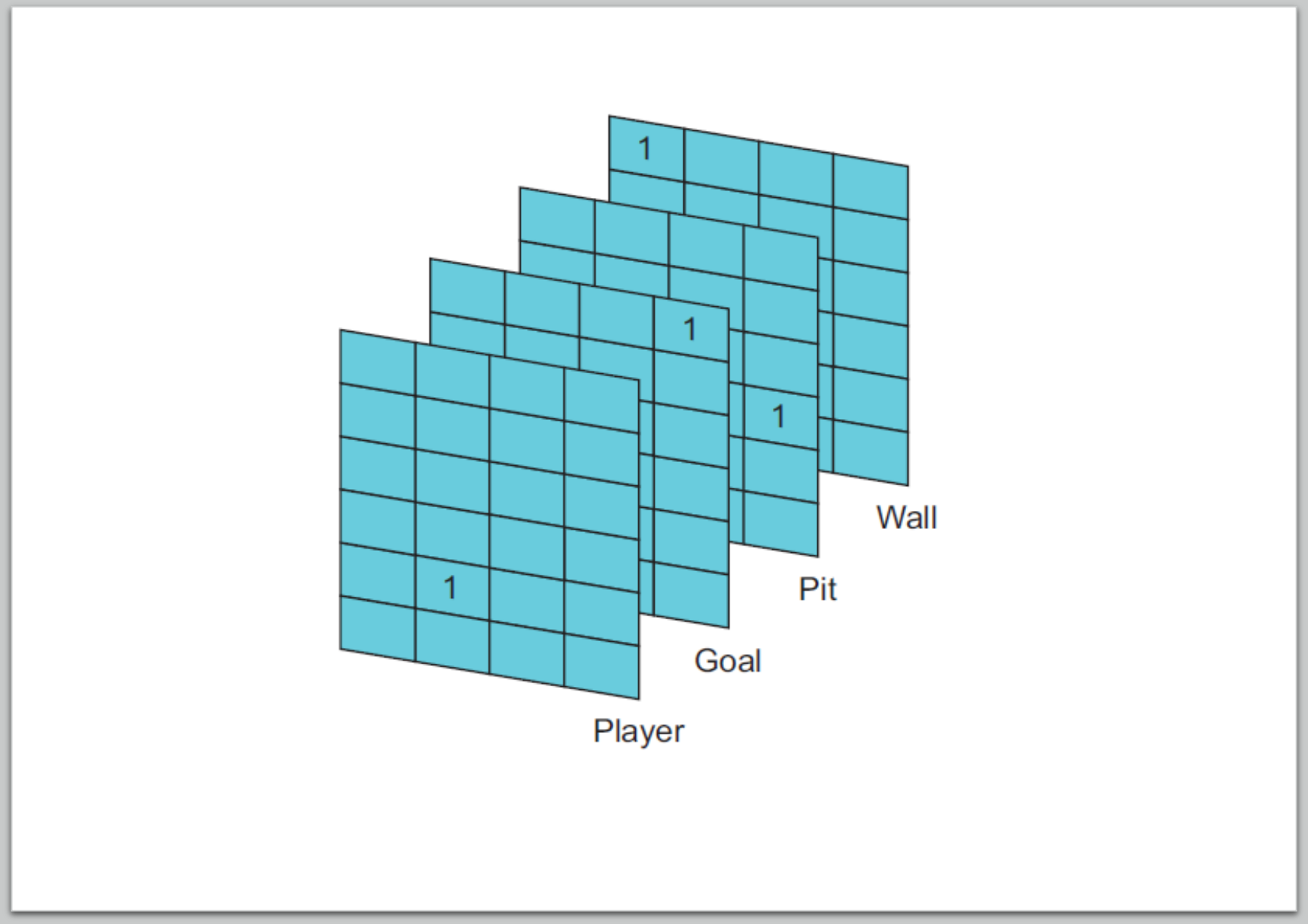
- Now we run the algorithm using S_{t+1} as input and figure out which action our algorithm predicts has the highest value. We'll call this value max $Q(S_{t+1},a)$. To be clear, this is a single value that reflects the highest predicted Q value, given our new state and all possible actions there
- We'll 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

$$Q(S_t\,,\,A_t\,) + \alpha\,*[R_{t+1}\; + \gamma\,*\max_{a}\,Q(S_{t+1}\,,A) - Q(S_t\,,\,A_t)]$$

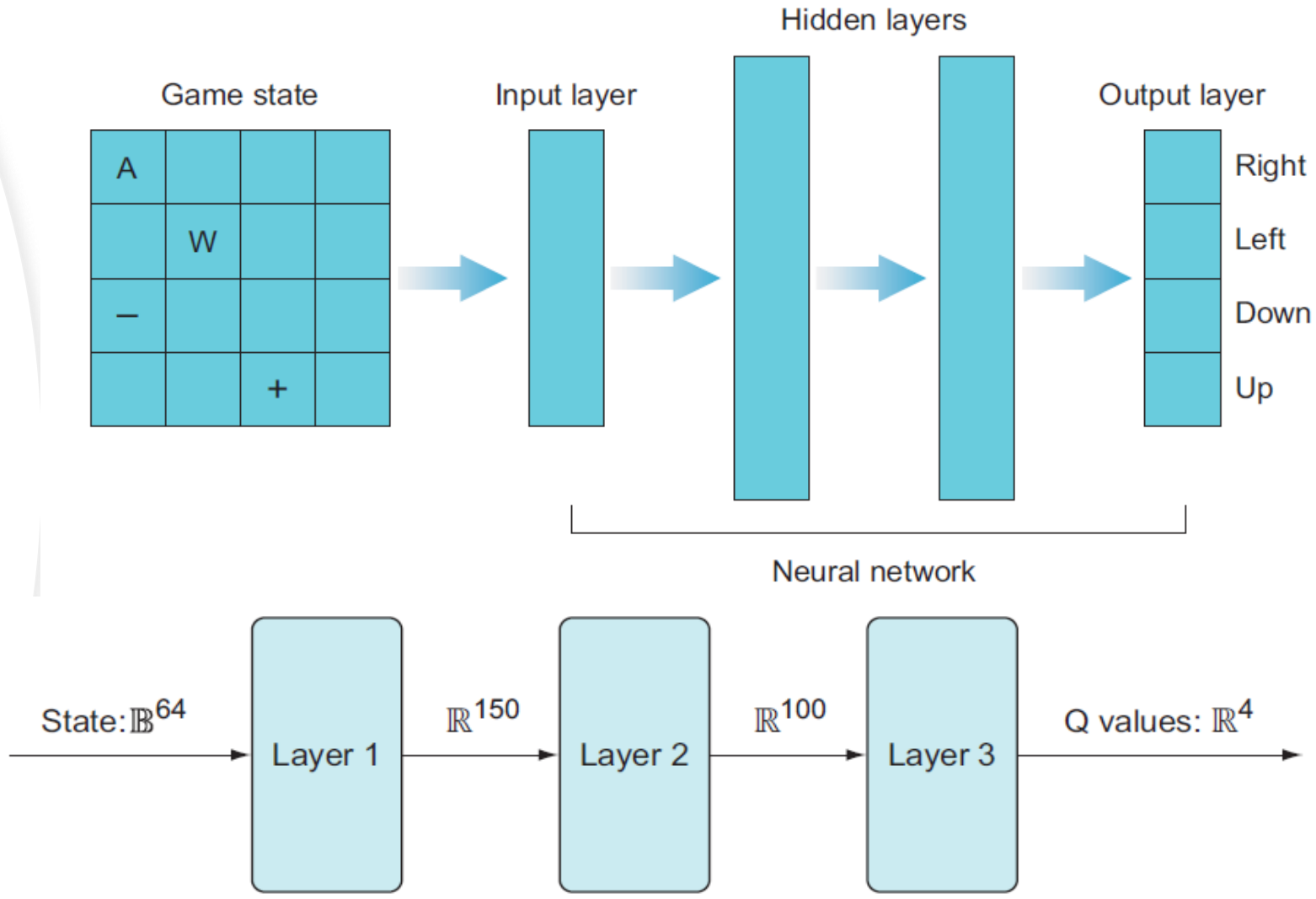


DeepMind used a modified vector-valued Q function that accepts a state and returns a vector of state-action values, one for each possible action given the input state. The vector-valued Q function is more efficient, since you only need to compute the function once for all the actions

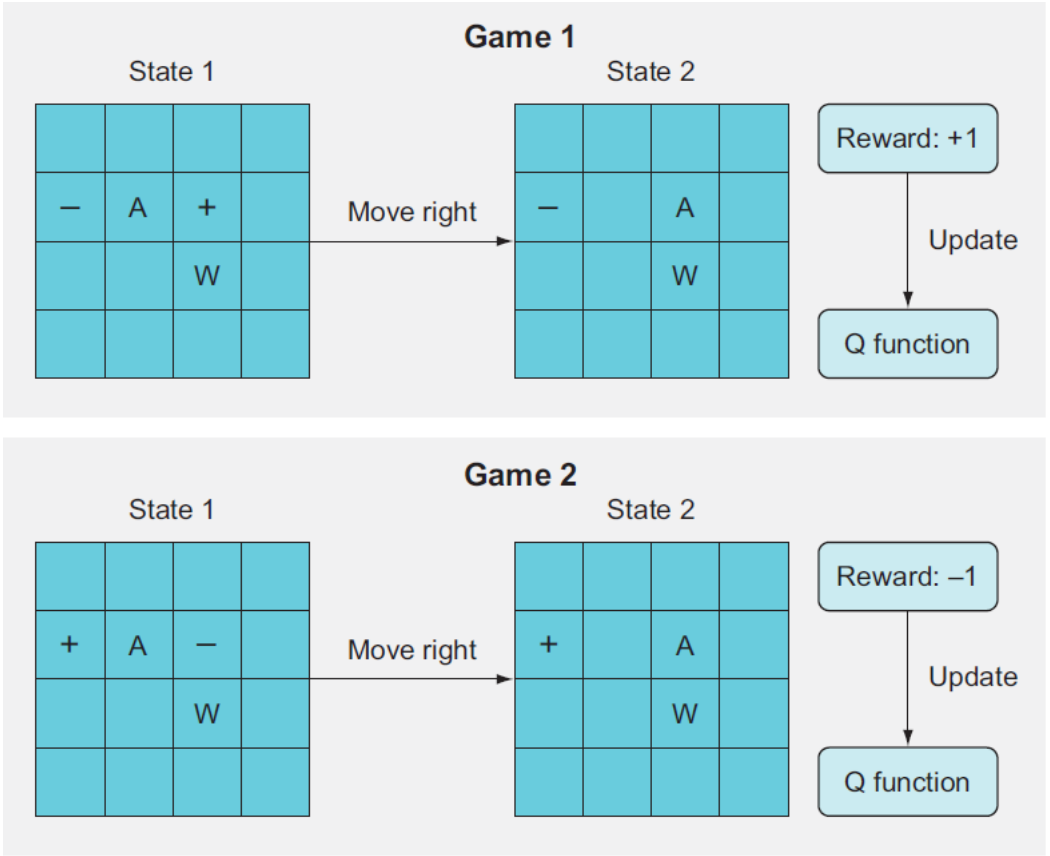
The Gridworld board is represented as a numpy array. It is a 4 x 4 x 4 tensor, composed of 4 “slices”



The general architecture for the model we will build.



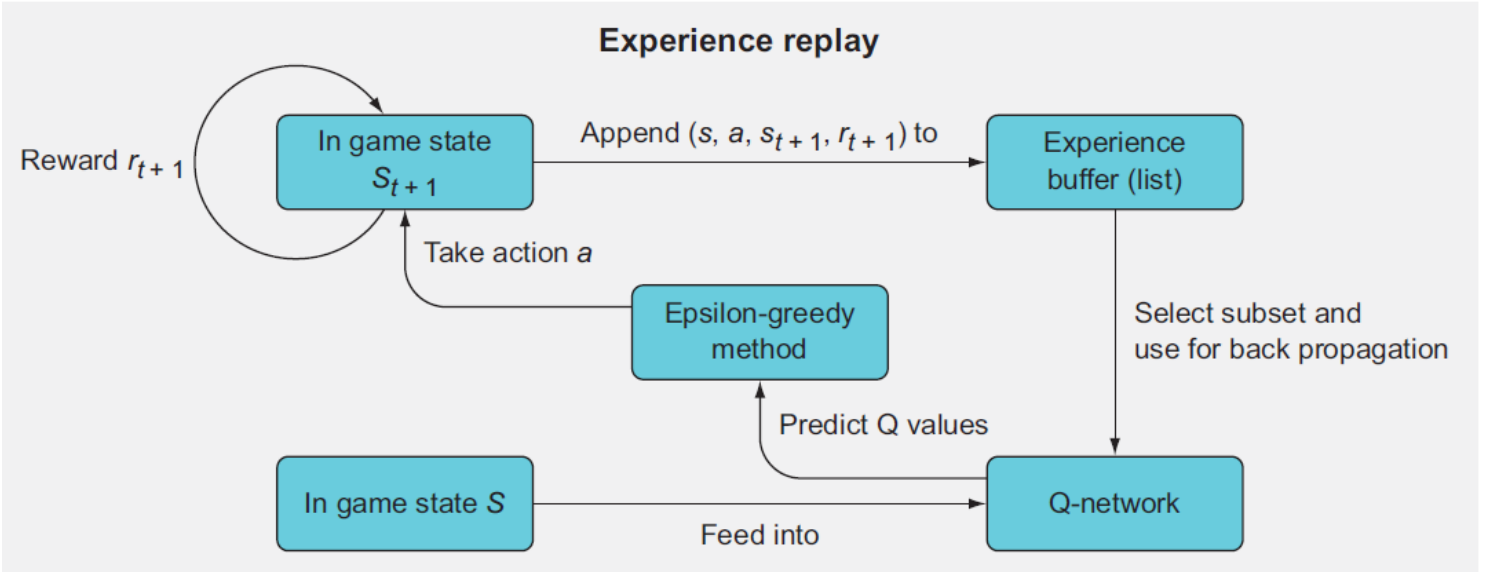
Catastrophic forgetting

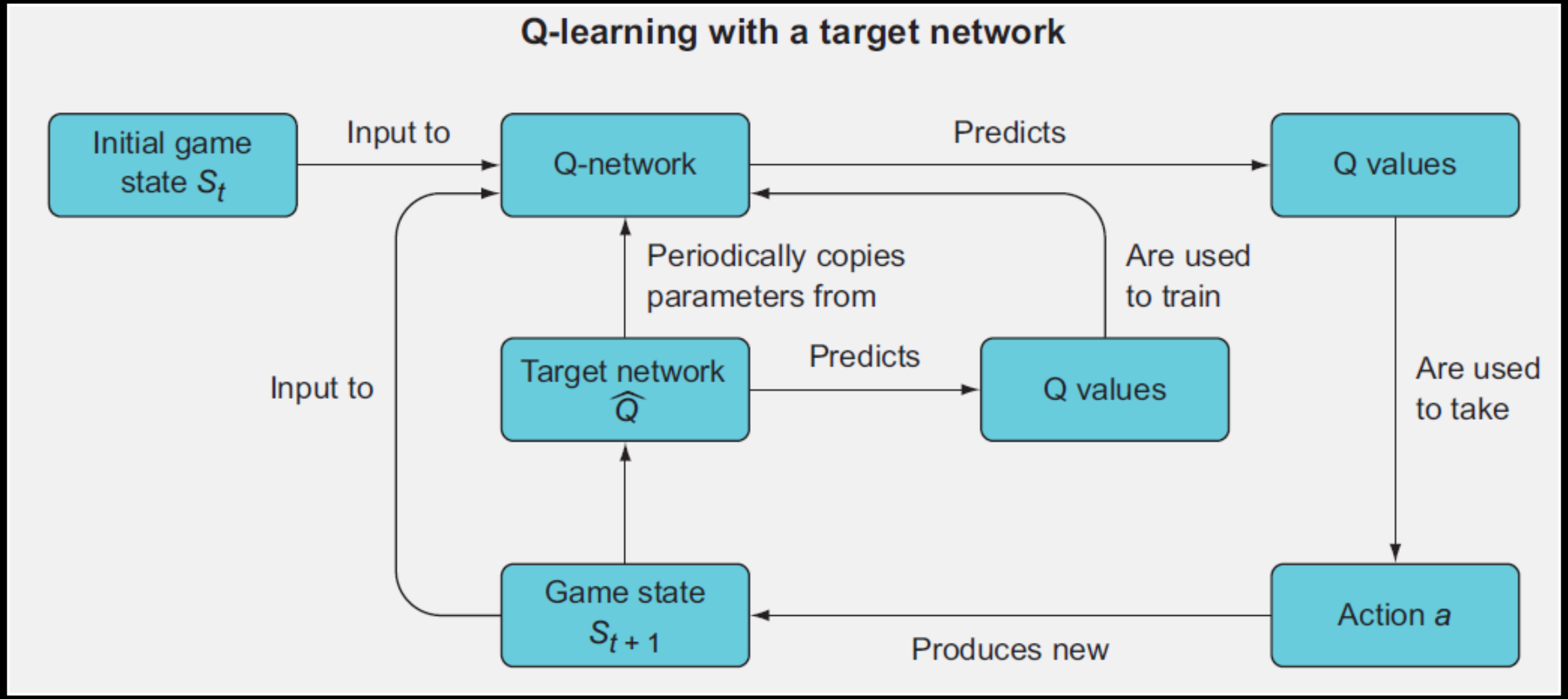


two game states are very similar and yet lead to very different outcomes, the Q function will get “confused” and won’t be able to learn what to do

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

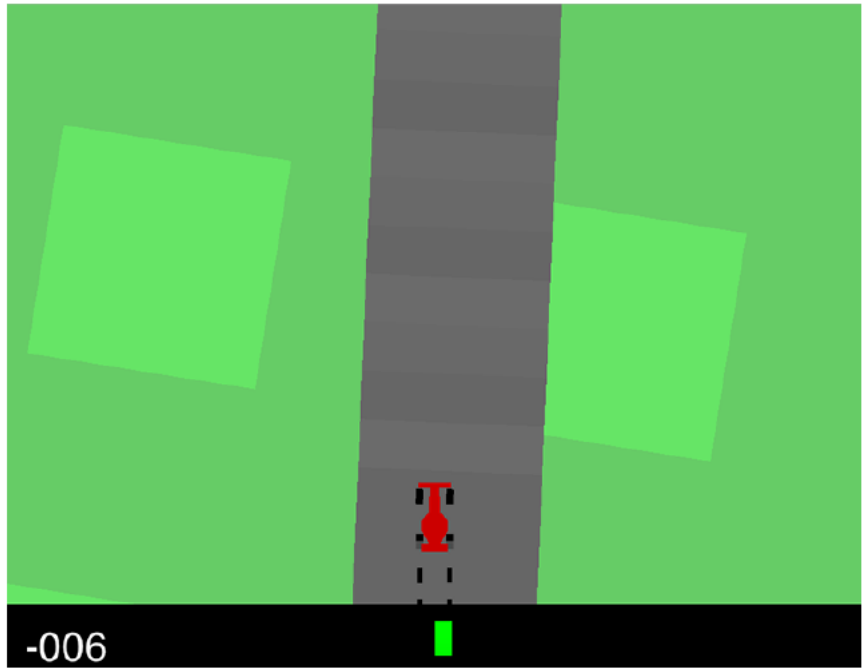


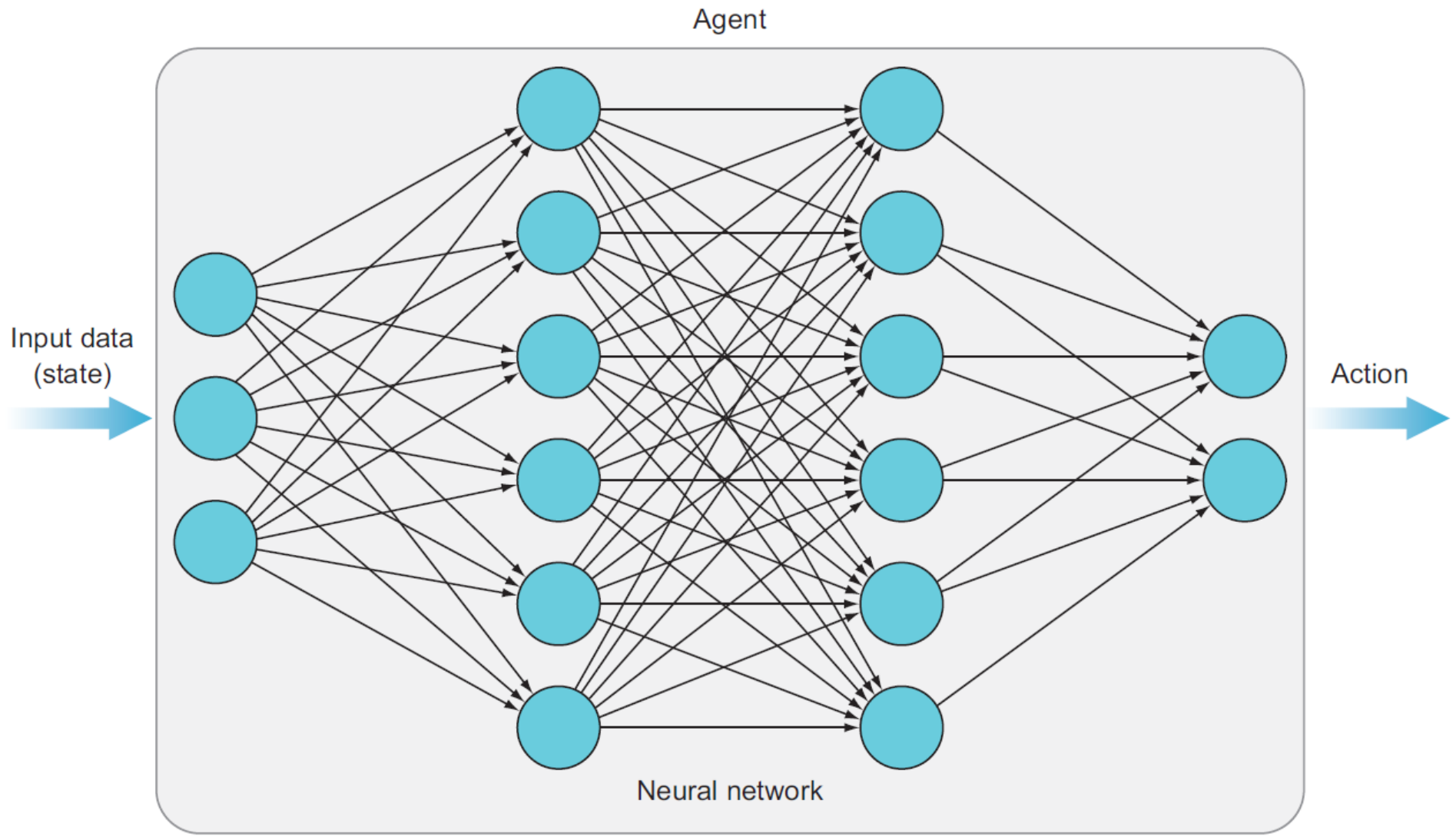


Improving stability with a target network

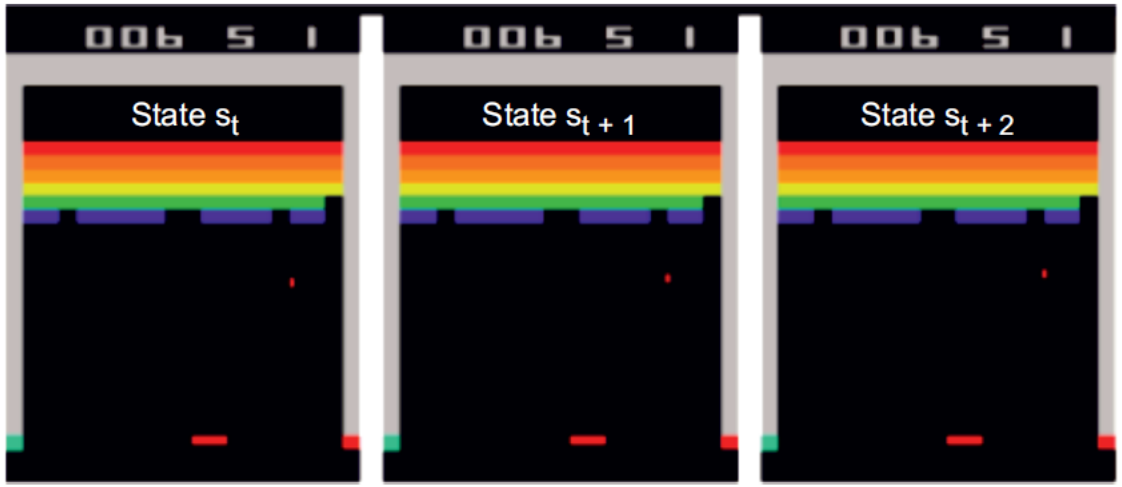
Open AI Gym

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




Agent
Input data
(state)
Action
Neural 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

Some DRL links

- Deep Reinforcement Learning: Pong from Pixels
  - http://karpathy.github.io/2016/05/31/rl/
  - https://arxiv.org/pdf/1312.5602.pdf
- 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