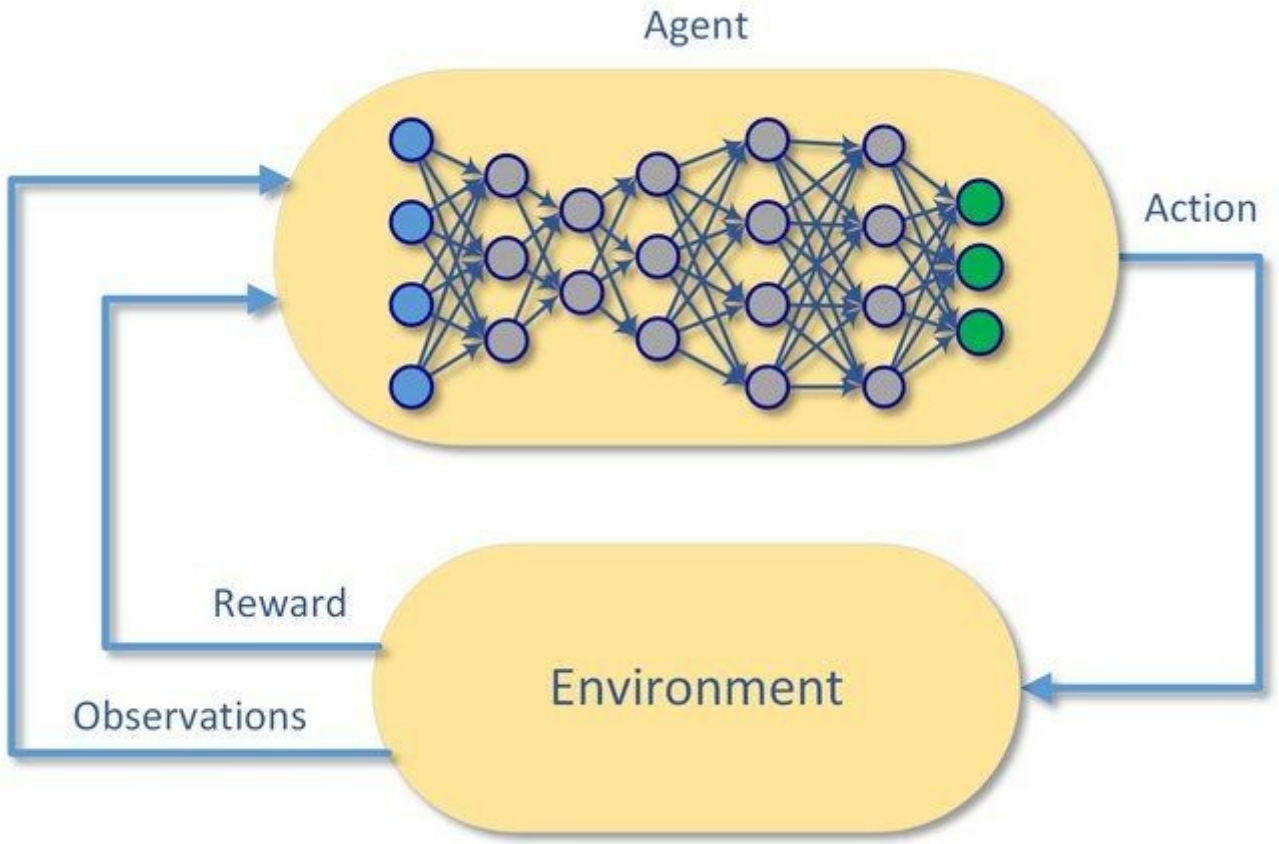
Deep Reinforcement Learning



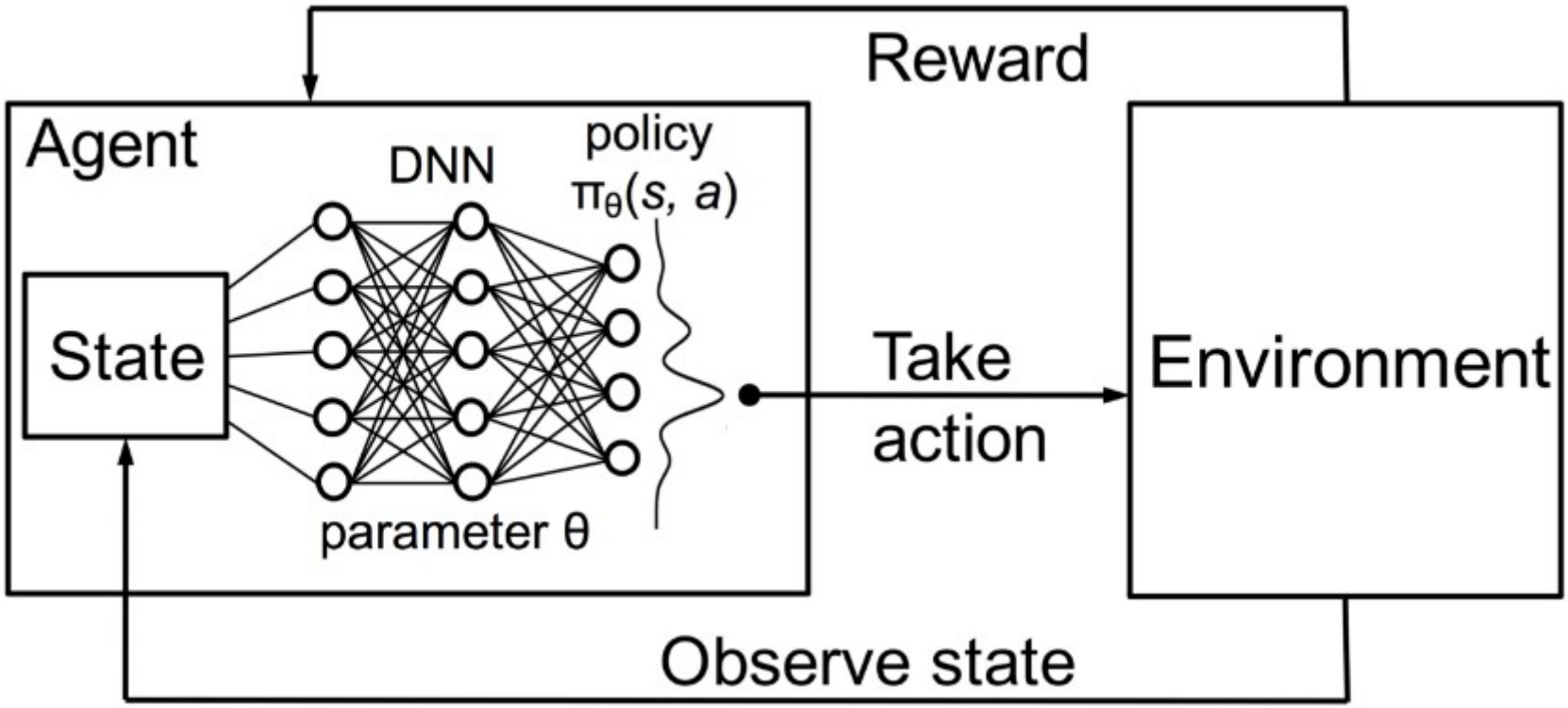
Reinforcement Learning involves…

- Sensing something of the environment
- Trial and error search
- A way to balance best predicted action (exploitation) vs. exploration
- Linking state -> action -> reward + new state
  - I was in state *s*. I took action *a.* I ended up in state *s'*. And I received reward *r.*
- Working out how to maximise long term rewards (*return*), sometimes at the cost of short term rewards
- Uncertainty in sensing and linking action to reward
  - Observations may be inaccurate
  - Action may not lead to the expected state
- Learning leads improved choice of actions (policy, π) over time

Elements of RL

- ***Environment***: all observable and unobservable information relevant to us
- ***Observation***: sensing the environment
- ***State***: the perceived (or perceivable) environment
- ***Agent***: senses environment, decides on action, receives and monitors rewards
- ***Action***: may be discrete (e.g. turn left) or continuous (accelerator pedal)
- ***Policy, π*** (how to link state to action; often based on probabilities)
- ***Reward*** signal: aim is to accumulate maximum reward (*return*) over time
- ***Memory***: A database of *state, action, reward, new state* for each action taken
- ***Value*** function of a state: prediction of likely/possible long-term reward from a particular *state*
- ***Q***: prediction of likely/possible long-term *return* from performing a particular *action*. Frequently includes the idea of *discount*, such that future rewards have less value the further out they are in time. The discount rate is symbolised as *gamma, γ.*
- ***Advantage***: the difference an action makes to long term reward compared with other possible actions.
- ***Target***: the value we wish to update our network to predict (equivalent to observed value or label in normal supervised learning)

High level view



Collecting Experience:

I was in state *s*
I took action *a*
I landed in state *s'*
I received reward *r*
(I reached *terminal* state)

We store each step as:

[State, action, reward, next state, terminal]

For each state we are given a set (vector) of observations

Q, and Q learning

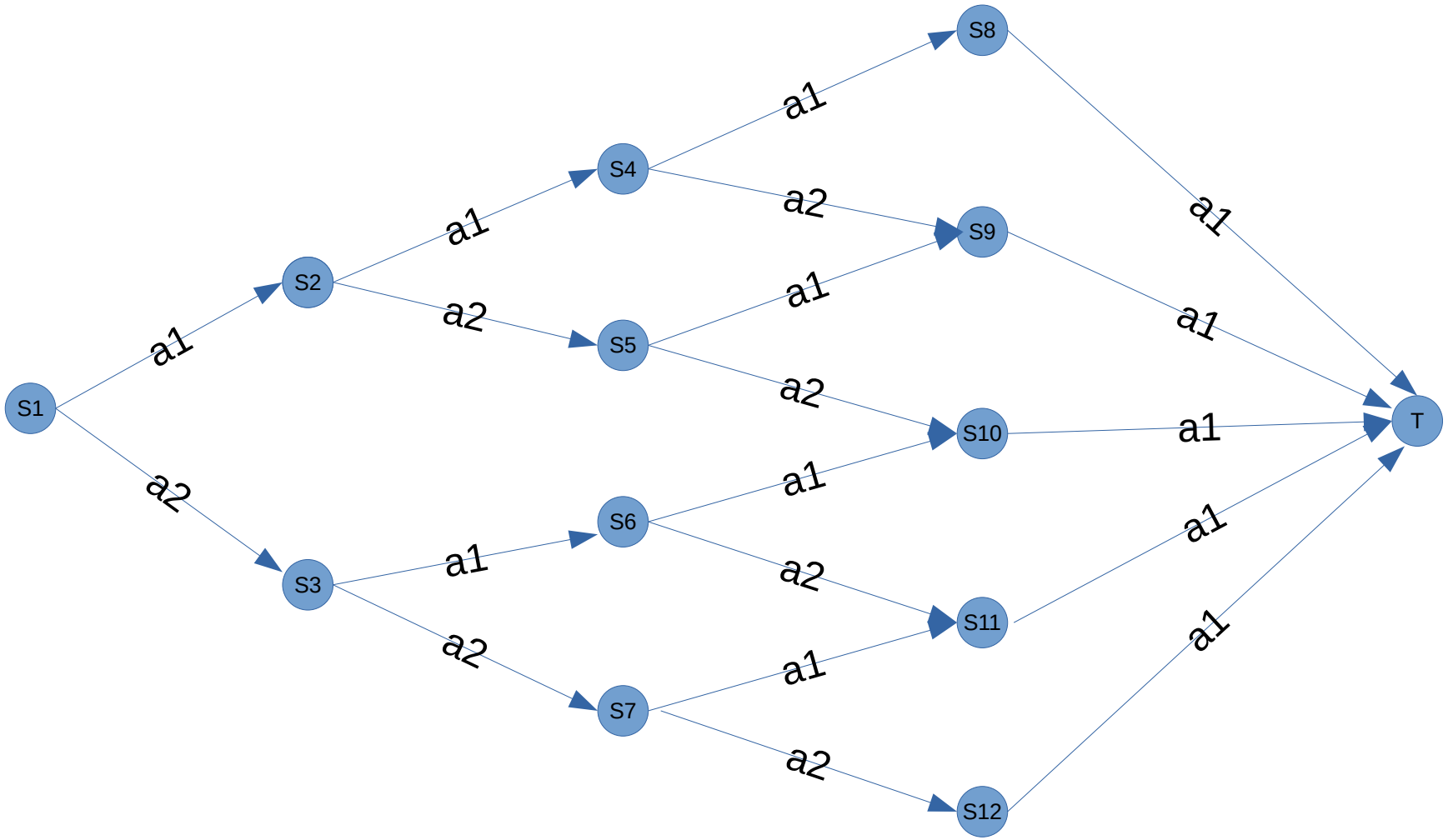
Q

- Q is the expected maximum long term reward from any particular action from a given state.

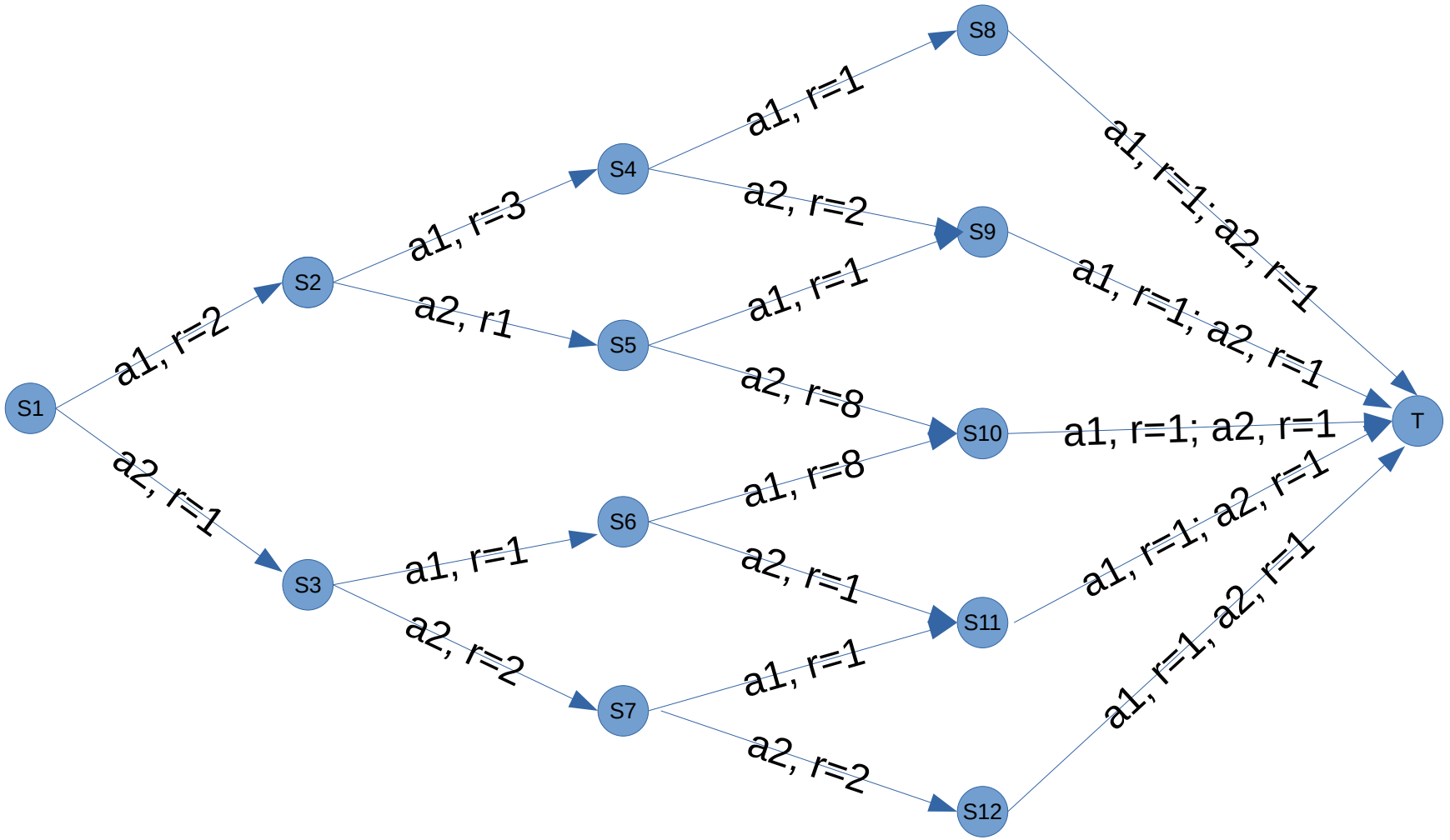
An example

- Imagine a game where I have to make three choices between two alternative options.
- With each choice I get a reward.
- Each choice leads me down an alternative path.
- We can represent all these choices and paths in a network graph.

A simple state, action, next state graph



A simple state, action, reward, next state graph



Use state(s), action (a), reward (r), next state (s') format.
For example, taking action 1 from state s2:
s,a,r,s' = [s2, a1, 3, s4]

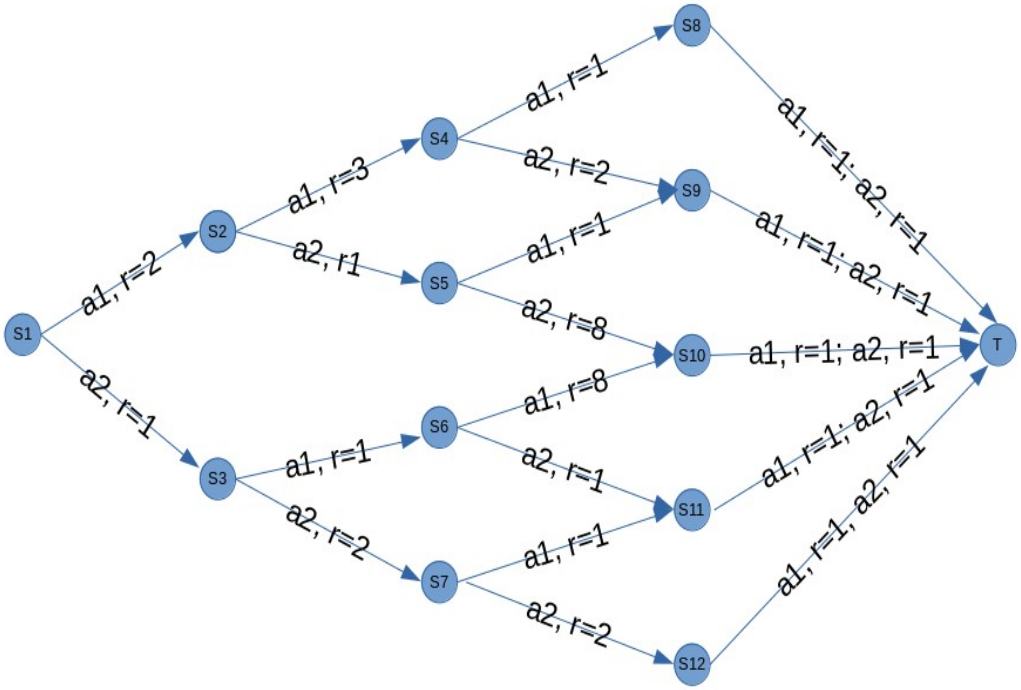
Bellman equation

- $Q(s,a) = r + \gamma.maxQ(s',a')$
- The Q of any state and action is the immediate reward achieved + the discounted maximum Q value (the best action taken) of next best action, where gamma (γ) is the discount rate.
- What is handy about the Bellman equation is that we can build up our knowledge of Q by only referring to the current step and the next step.

Q: $Q(s,a) = r + \gamma.\max Q(s',a')$

Bellman equation; The Q of any state and action is the immediate reward achieved + the discounted maximum Q value (the best action taken) of next best action, where gamma (γ) is the discount rate.

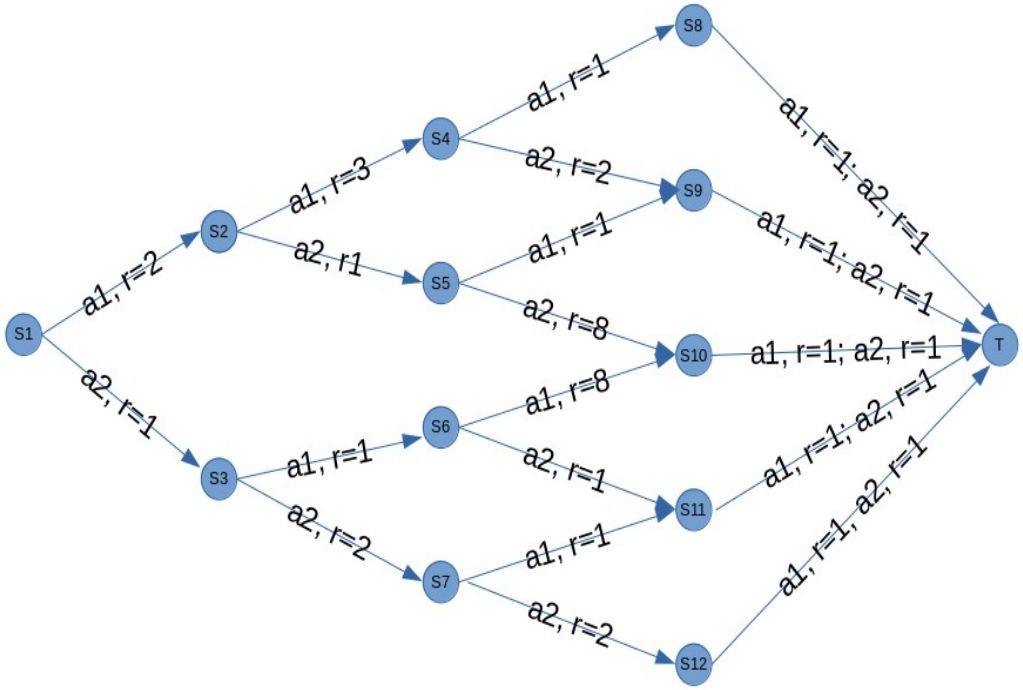
- Q is best thought through from the terminal state (T) backwards)
- S8 to S12 all go to the terminal state with a reward of 1. There is no possibility of further reward. So Q is simply the reward, 1.
- Taking action 1 from state S4, leads to sate S8, with a reward of 1.
  - We know state S8 has a Q of 1 (whichever action is taken. If we use a discount rate (γ) of 0.9, the Q of [S4, a1] is the immediate reward (1) + 0.9 * Q(S8) = 1.9
- Taking action 2 from state S4, leads to sate S9, with a reward of 2.
  - We know state S9 has a Q of 1 (whichever action is taken. If we use a discount rate (γ) of 0.9, the Q of [S4, a2] is the immediate reward (2) + 0.9 * Q(S9) = 2.9



Q: $Q(s,a) = r + \gamma.\max Q(s',a')$

Bellman equation; The Q of any state and action is the immediate reward achieved + the discounted maximum Q value (the best action taken) of next best action, where γ is the discount rate.

- Now working back to S2, let's look at action 1 [S2, a1] :
  - Action 1 from state S2 leads to state S4.
  - State S4 has Q values of 1.9 and 2.9 depending on the action taken in state S4
  - For the Bellman equations we take the best future Q of the next state, which is 2.9
  - So the Q of [S2, a1] = immediate reward + γ * best Q available in the next state = 3 + (0.9 * 2.9) = 5.61



Can you work out all the Q values (for each state/action pair)?

Calculated Q values

Gamma	0.9				
State	**Action**	**Reward**	**Next state**	**Best Q of next state**	**Q**
S1	1	2	S2	8.9	10.01
	2	1	S3	8.9	9.01
S2	1	3	S4	2.9	5.61
	2	1	S5	8.9	9.01
S3	1	1	S6	8.9	9.01
	2	2	S7	2.9	4.61
S4	1	1	S8	1	1.9
	2	2	S9	1	2.9
S5	1	1	S9	1	1.9
	2	8	S10	1	8.9
S6	1	8	S10	1	8.9
	2	1	S11	1	1.9
S7	1	1	S11	1	1.9
	2	2	S12	1	2.9
S8	1	1	T	0	1
	2	1	T	0	1
S9	1	1	T	0	1
	2	1	T	0	1
S10	1	1	T	0	1
	2	1	T	0	1
S11	1	1	T	0	1
	2	1	T	0	1
S12	1	1	T	0	1
	2	1	T	0	1

If we follow the best Q:

S1 → a1 → S2 (reward = 2)
S2 → a2 → S5 (reward = 1)
S5 → a2 → S10 (reward = 8)
S10 → a1 or a2 → T (reward = 1)

Total reward = 12

Note: At S2 we took an action with lower immediate reward.

This is an example of a *Q table* but in complex environments we cannot collect information on all states/actions (this is also impossible if the state contains continuous variables, such as velocity). Neural nets will often rely on *similarity* between a current state and one encountered before.

Updating Q in a network

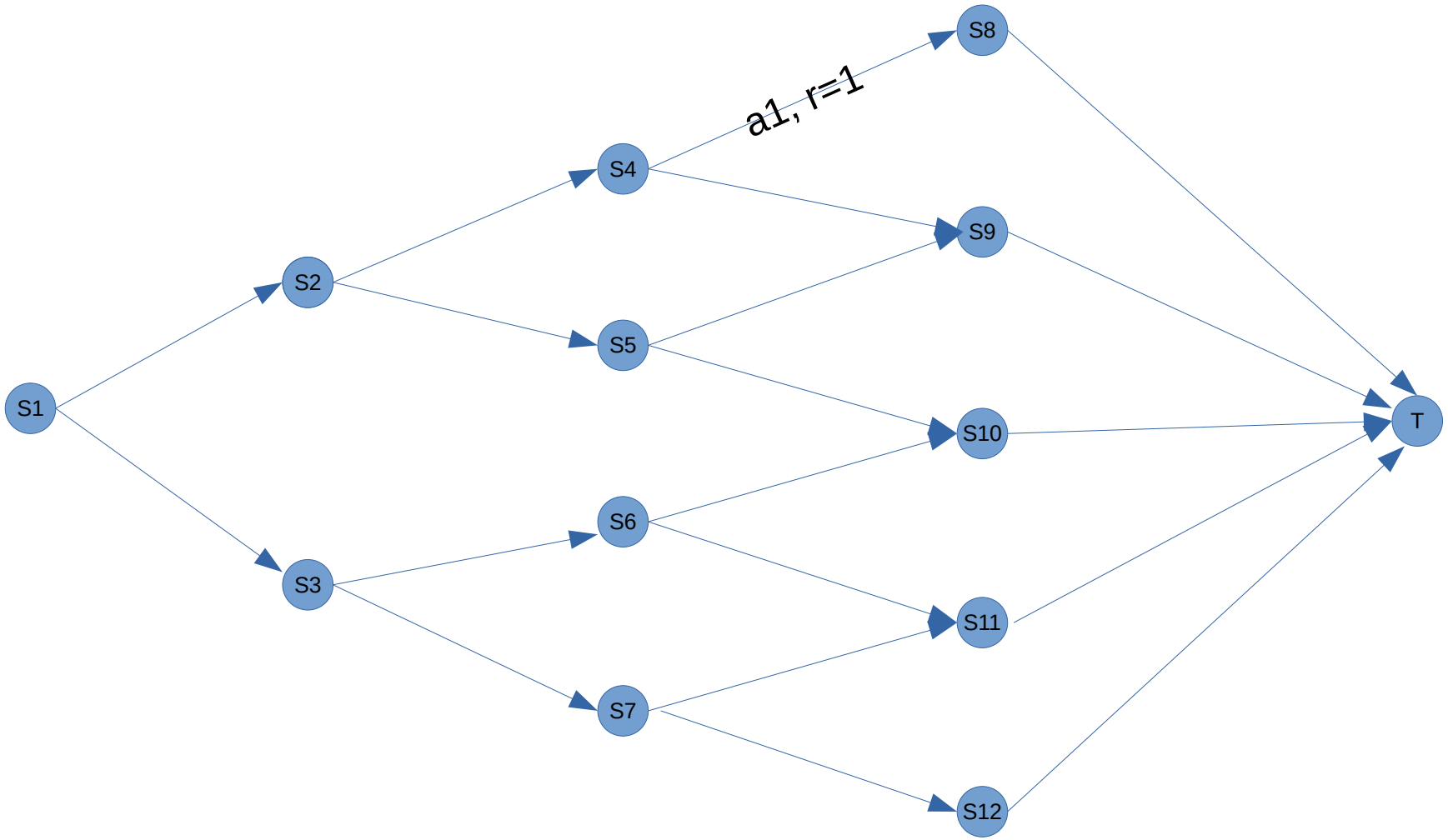
- Updating Q in a network is performed by random sampling* from a memory of s/a/r/s’
- We update Q based on the reward and best Q of next step. We only need to look one step ahead and update based on that. This is *temporal differential learning*. Updates can happen while learning (no need to complete episodes).
- Early on the network will have a very incomplete understanding of Q. There may be limited knowledge of rewards, and some rewards will not have been sampled sufficiently to update network.
- The performance of the network is especially poor early on. During this stage it is usual to have a significant level of random exploration to help ensure a wider range of s/a/r/s’ are experienced
- As more data (experience) is collected, and as more training iterations are performed, predicted Q will become closer to actual Q.

*We can prioritise training examples with Prioritised Experience Replay

Example

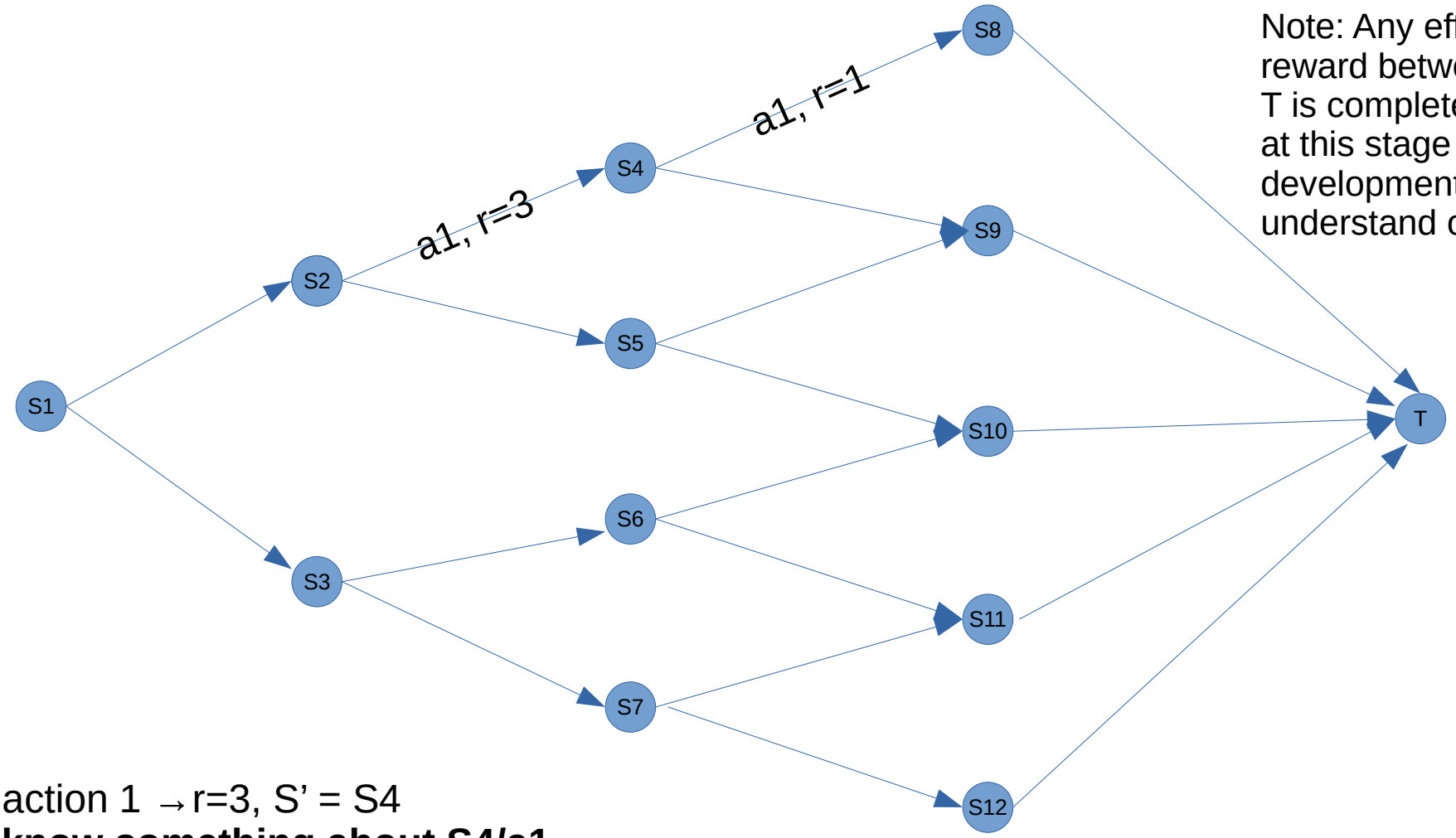
- We start with no idea of Q (we will set al at zero to make it easier to see what is going on, but a neural net will start with all random numbers).
- The following technique is a *boot-strapping* technique. It has been said that one of the biggest surprises in RL is that it works!
- The network provides the *target* value for best Q of next step. So the network is updating itself based on its own predicted values!

Our first random sample is action 1 from state S4



S4, action 1 → r=1, s'=s8
No knowledge of s8 so we set **Q of S4/a1 to = 1** (just the reward)

Our second random sample is action 1 from state S2



Note: Any effect of reward between S8 and T is completely ignored at this stage in development of our understand of Q values.

S2, action 1 →r=3, S' = S4

We know something about S4/a1

For S4 the max Q is 1 (S4, a1. We don't know about action 2 yet, so assume it is zero here)

S2, action 1: Q = reward + γ maxQ S4 = 3 + (0.95 * 1) = 3.95 (if γ = 0.95)

Carrying on…..

- Repeat this bootstrapping
- Not surprisingly early on in the development of Q network the predictions of best action are not good!
- So, it is common to:
  - Perform a large number of random actions to store in memory before even starting to train the network.
  - Only gradually increase the proportion of actions that are selected by the network (rather than random choice) after training begins.

DQN Basic Method

1) Initiate parameters for Q(s,a) with random weight.

2) Perform a number of random actions before training starts

3) With reducing probability select a random action, or best current action, *argmax Q(s,a)*

4) Execute action a, and get reward *r* and observations of next state *s'*.

5) Store transition (s,a,r,s') in the memory

6) Sample random mini batch of transitions from the memory.

7) For every transition in the buffer, calculate a *target Q* using the network to predict best next Q (from state s')

$$target\ Q = r + \gamma(max\ Q)$$

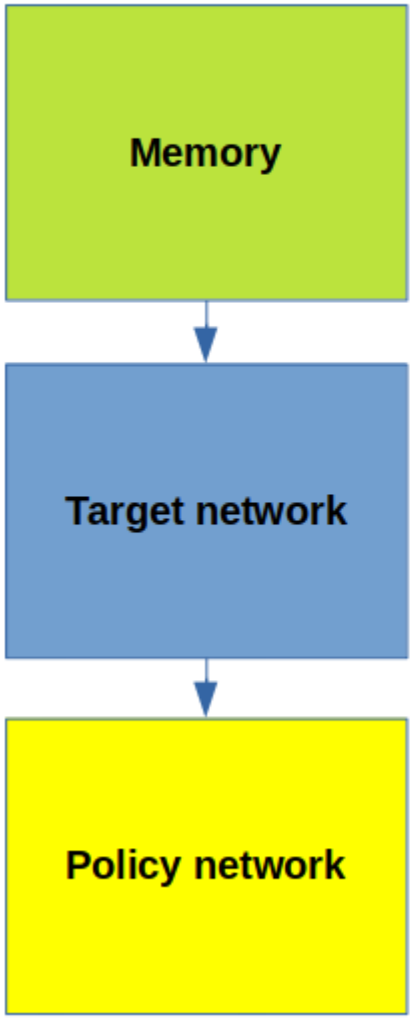
8) Get network prediction of current Q(s,a), and update network by minimising the loss between predicted and target Q for s/a pair.

9) Repeat from (2)

Double DQN

- Double DQN contains two networks. This amendment is to decouple training of Q for current state and next state which are closely correlated when comparing input features.
- The ***policy network*** is used to select action (action with best predicted Q) when playing the game.
- When training, the predicted best action (best predicted Q) for S' is taken from the ***policy network***, but the ***policy network*** is updated using the predicted Q value of the next state from the ***target network*** (which is updated from the ***policy network*** less frequently). So, when training, the action from S' is selected using Q values from the ***policy network***, but the the ***policy network*** is updated to better predict the Q value of that action from the ***target network***. The ***policy network*** is copied across to the ***target network*** every *n* steps (e.g. 1000).
- (It is called a target network because it provides the target for updating, just as the labels or observed values provide the target for a supervised learning neural network)

Train *policy network* every game step



Random sample from *memory*:
S = Initial state observations
A = Action taken
R = Reward
S' = new state observations
T = Is S' a terminal state?

(Use 1-10 samples per training step)

Use *target network* to get best Q for next state (S')
in S/A/R/S'/T sampled from *memory*.

Calculate new Q for S/A from Bellman equation using reward from S/A/R/S'/T sampled from memory, and using best next Q from *target network*.

Target Q(s,a) = r + γ.maxQ(S',a')

Train *policy network* on new *Target Q*.

Train *target network* every *n* game steps (e.g. every 1,000 steps)