Reinforcement Learning

What is Reinforcement Learning?

Reinforcement Learning (RL) is an area of machine learning where an *agent* ought to take some *actions* in an *environment* so as to maximize some notion of cumulative *reward*.

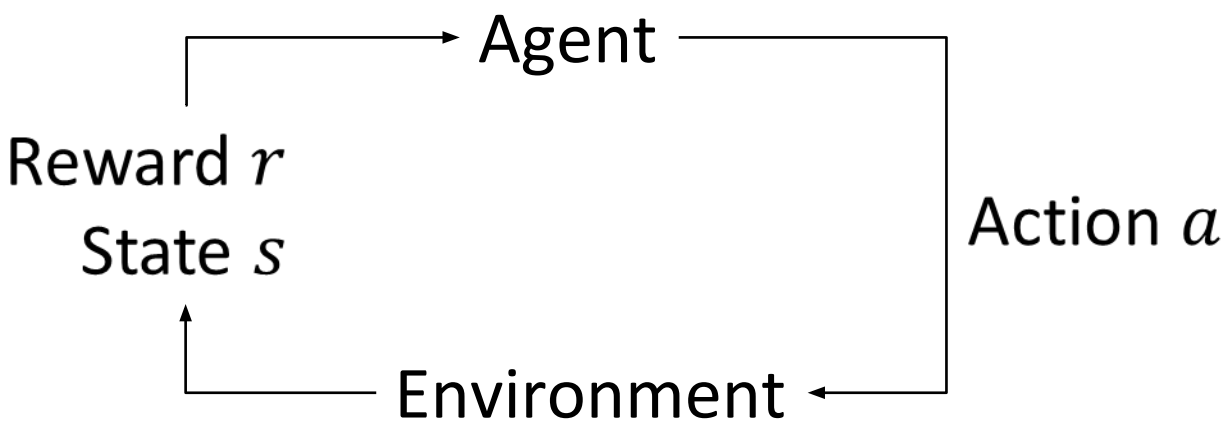
RL vs Supervised vs Unsupervised

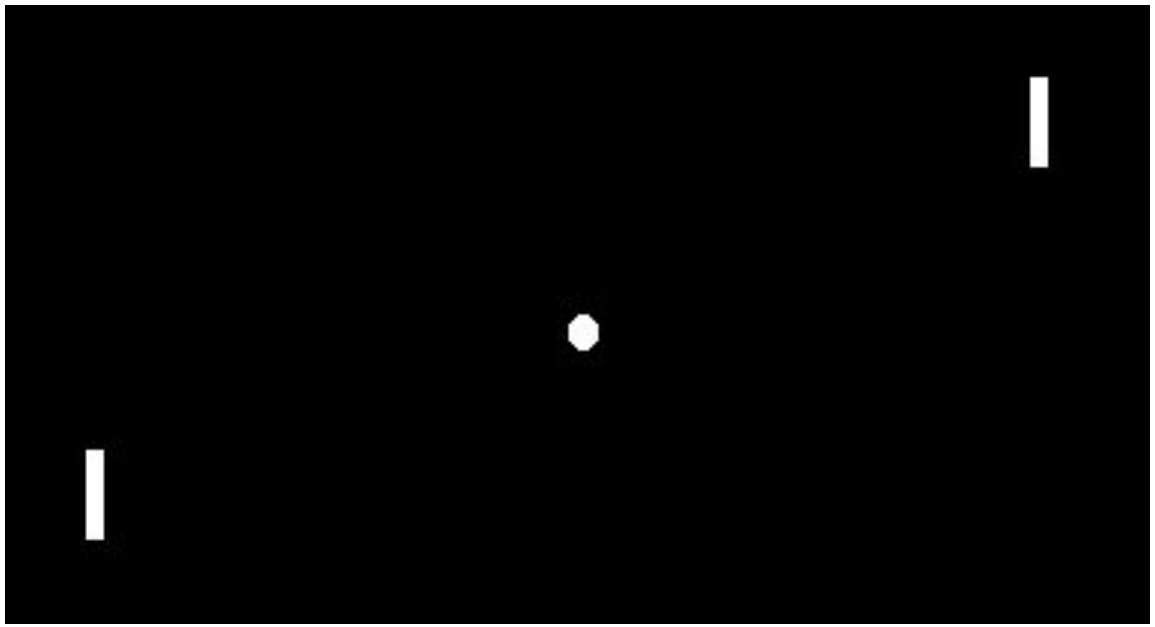
- Supervised - prediction tasks
- Unsupervised – understand data
- RL – decision making

RL Examples

- Game playing AI. Teaching an agent to play games like Pong, Chess, Go, etc.
- Article recommendations. Teaching an agent to make better article suggestions to a user.
- Robotics. Teaching an agent to navigate a course.

RL System





RL System

Consider an AI playing the ATARI game **Pong.**

Environment- The setup of the game

Agent - One of the player

Action - Move up / Move down

State - Current positions of the paddles and the ball

Reward - +1 for winning

-1 for losing

0 otherwise

Objective

The objective of an agent in an RL setup is to achieve maximum (cumulative) reward.

In the Pong example, the objective is to keep winning to achieve maximum reward.

Objective

The objective of an agent in an RL setup is to achieve maximum (cumulative) reward.

In the Pong example, the objective is to keep winning to achieve maximum reward.

$$E_{x \sim p(x|\theta)}[f(x)]$$

Objective

The objective of an agent in an RL setup is to achieve maximum (cumulative) reward.

In the Pong example, the objective is to keep winning to achieve maximum reward.

$$E_{x \sim p(x|\theta)}[f(x)]$$

$$x - Action$$

$$f(x) - Reward\ for\ action\ x$$

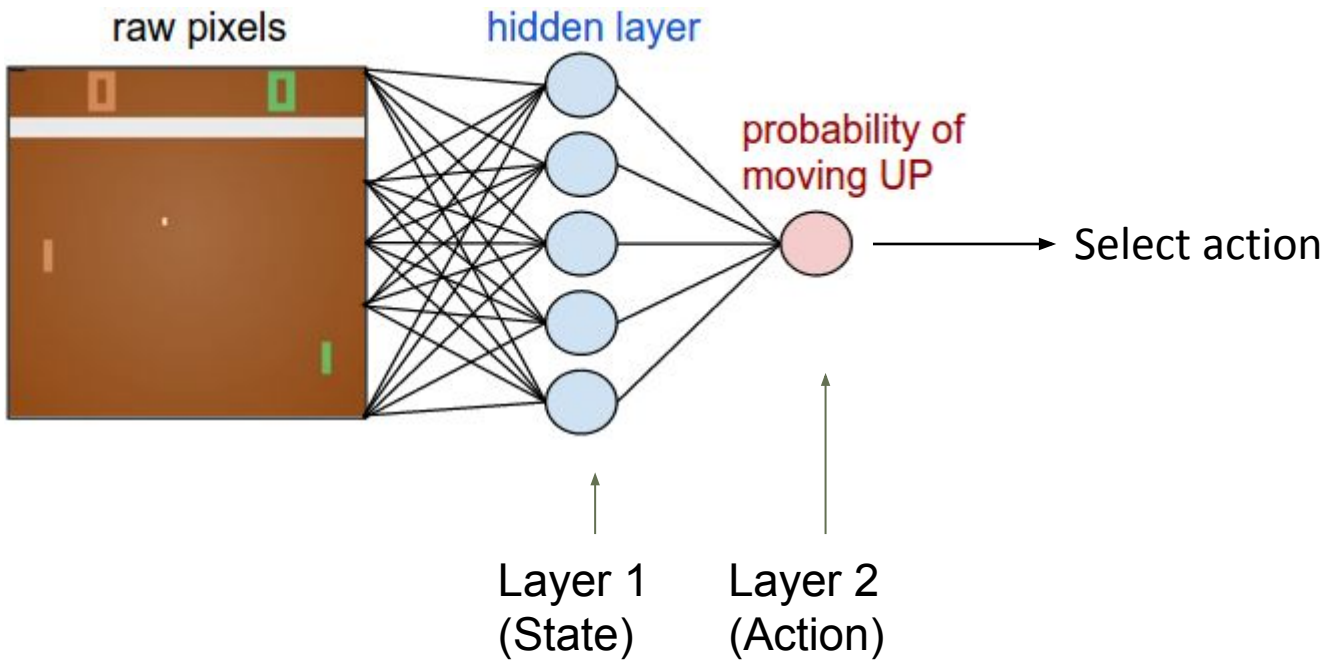
$$p(x|\theta) - Policy\ (Decision\ maker)$$

Model

A fully connected 2 layer Neural Network.

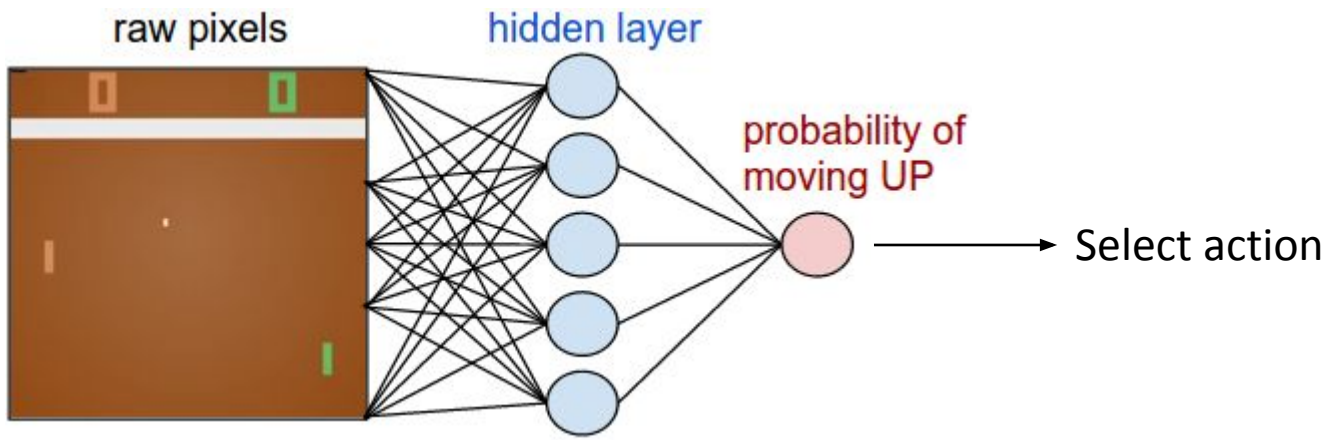
The first layer will take as input a vector of H x W x 3 pixels and capture the state of the game.

The second layer will capture the action needed to perform.



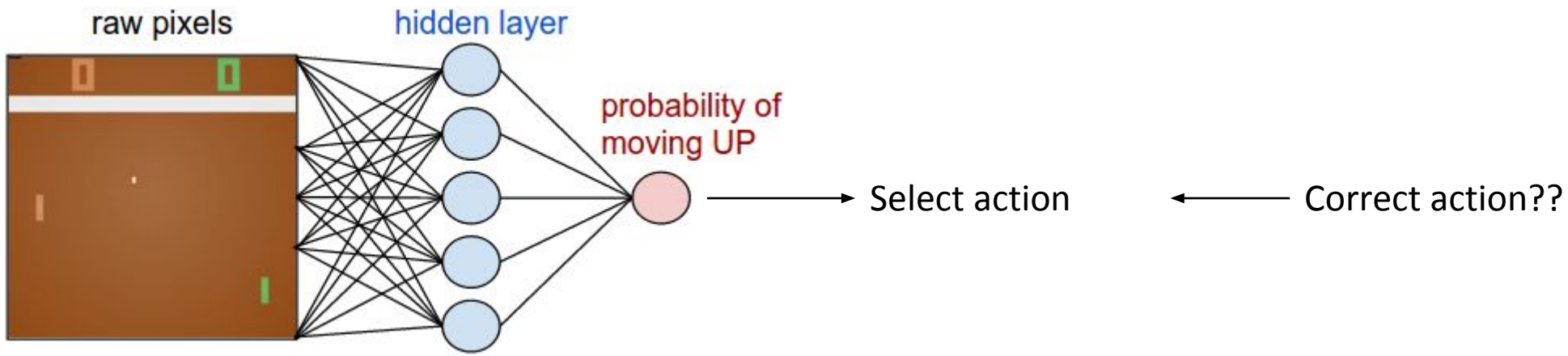
How to train?

Can we train in a Supervised learning fashion?

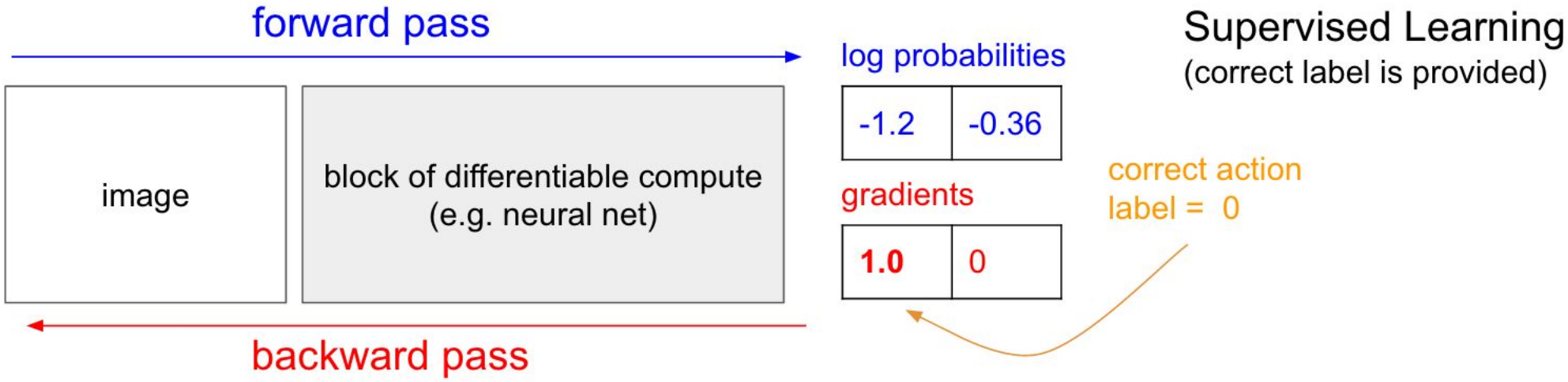


How to train?

Can we train in a Supervised learning fashion?



How to train? (cont.)



In supervised learning we need a *correct label* to perform backward pass to train the system.

Solution

What do we do if we do not have the correct label in the Reinforcement Learning setting?

Solution

What do we do if we do not have the correct label in the Reinforcement Learning setting?

Use Policy Gradients

Policy Gradient - Step 1

Run a forward pass on the network with the present state.

Let’s say, our policy network calculated probability of going UP as 30% (logprob -1.2) and DOWN as 70% (logprob -0.36).

Policy Gradient - Step 2

Sample an action from the probability distribution obtained from Step 1. Execute the action.

We will now sample an action from this distribution; E.g. suppose we sample DOWN, and we will execute it in the game.

Policy Gradient - Step 3

Calculate the gradient assuming the sampled action is the correct action.

The sampled action might not be correct. But that is ok. We can wait for a bit and see!

Note: We are not updating the network yet. We are only calculating the gradients.

Policy Gradient - Step 4

Repeat Steps 1 - 3 till the game ends.

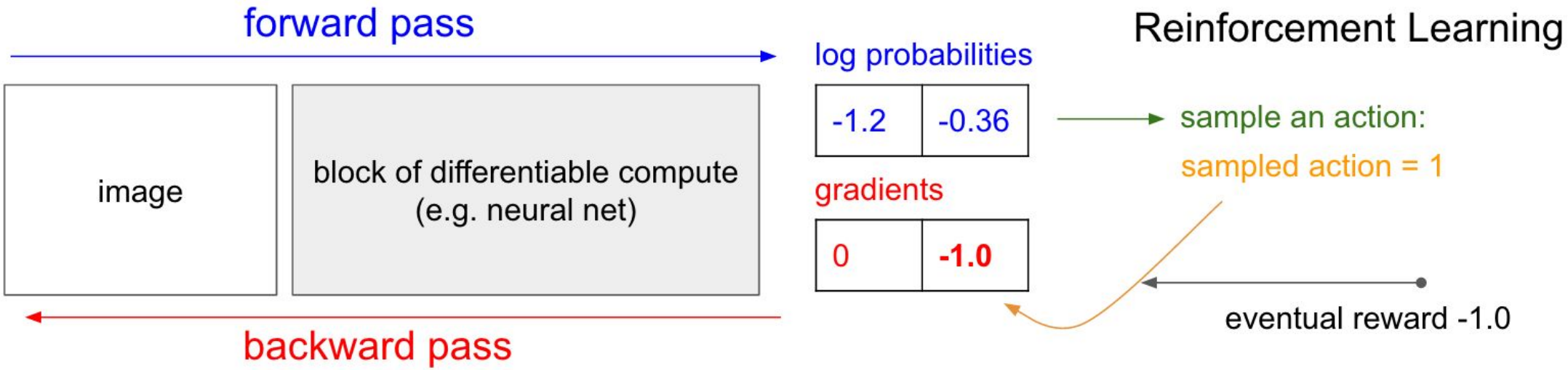
Update the network using the calculated gradients and the **“eventual reward”** received.

In Pong the eventual reward will be +1 (winning) or -1 (losing).

Policy Gradient - Intuition

Let’s say going DOWN ended up in losing the game (-1 reward). So if we combine that with the gradient and do backprop we will find a gradient that ***discourages*** the network to take the DOWN action for that input in the future.

Policy Gradient



Deriving Policy Gradients

Policy Gradients are a special case of a more general *score function gradient estimator*.

The general case is that we trying to maximize an expression of the form:

$$E_{x \sim p(x\,|\theta)}\,[f(x)]$$

$$\mathrm{E}[X] = \sum_{i=1}^{k} x_i\, p_i = x_1 p_1 + x_2 p_2 + \cdots + x_k p_k.$$

$$\begin{aligned}
\nabla_\theta E_x[f(x)] &= \nabla_\theta \sum_x p(x) f(x) && \text{definition of expectation} \\
&= \sum_x \nabla_\theta p(x) f(x) && \text{swap sum and gradient} \\
&= \sum_x p(x) \frac{\nabla_\theta p(x)}{p(x)} f(x) && \text{both multiply and divide by } p(x) \\
&= \sum_x p(x) \nabla_\theta \log p(x) f(x) && \text{use the fact that } \nabla_\theta \log(z) = \frac{1}{z} \nabla_\theta z \\
&= E_x[f(x) \nabla_\theta \log p(x)] && \text{definition of expectation}
\end{aligned}$$

Policy Gradient (cont.)

Supervised learning objective :

$$\sum_i \log(p(y_i|x_i))$$

Policy Gradient learning objective :

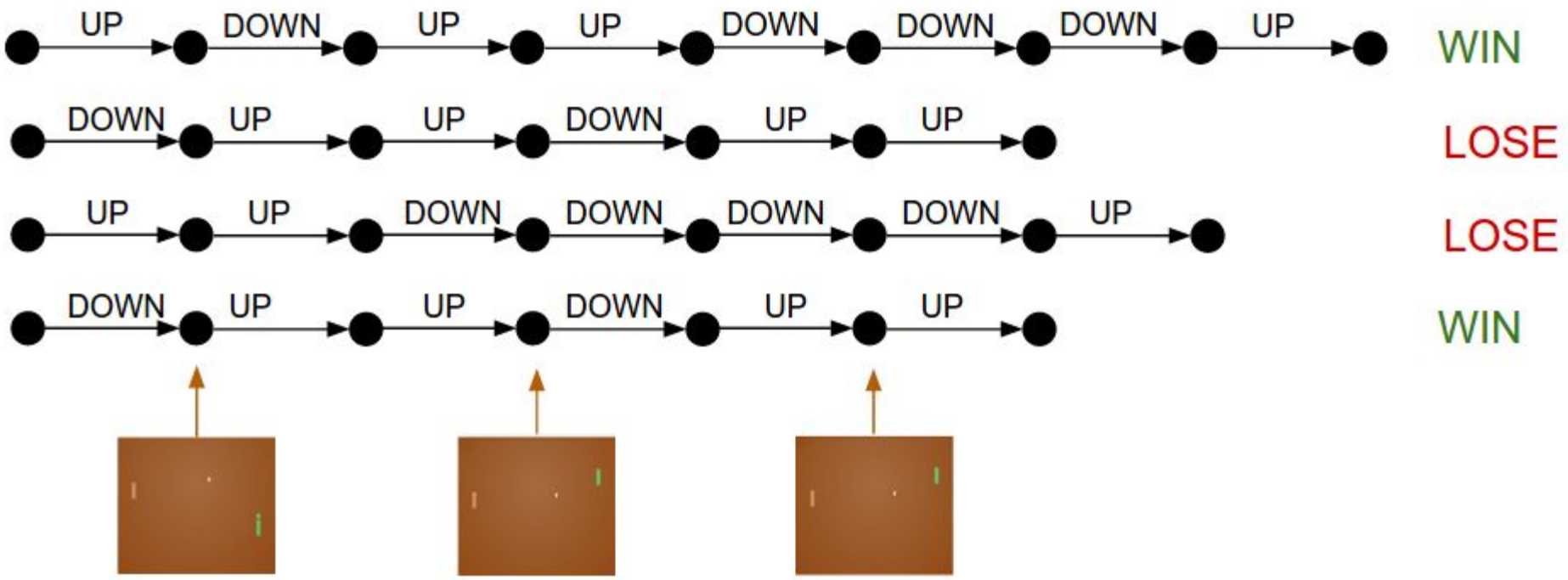
$$\sum_i A_i \, log(p(y_i|x_i))$$

A_i here is called an **advantage.** In case of Pong, A_i is +1 if we eventually win in the episode that contained x_i and -1 if we lost.

y_i here is the action we happened to sample from the distribution output of the network.

This will ensure that we maximize the log probability of actions that led to good outcome and minimize the log probability of those that didn't.

Policy Gradient (cont.)



More general Advantage functions

In a more general RL setting we would receive some reward r_t at every time step

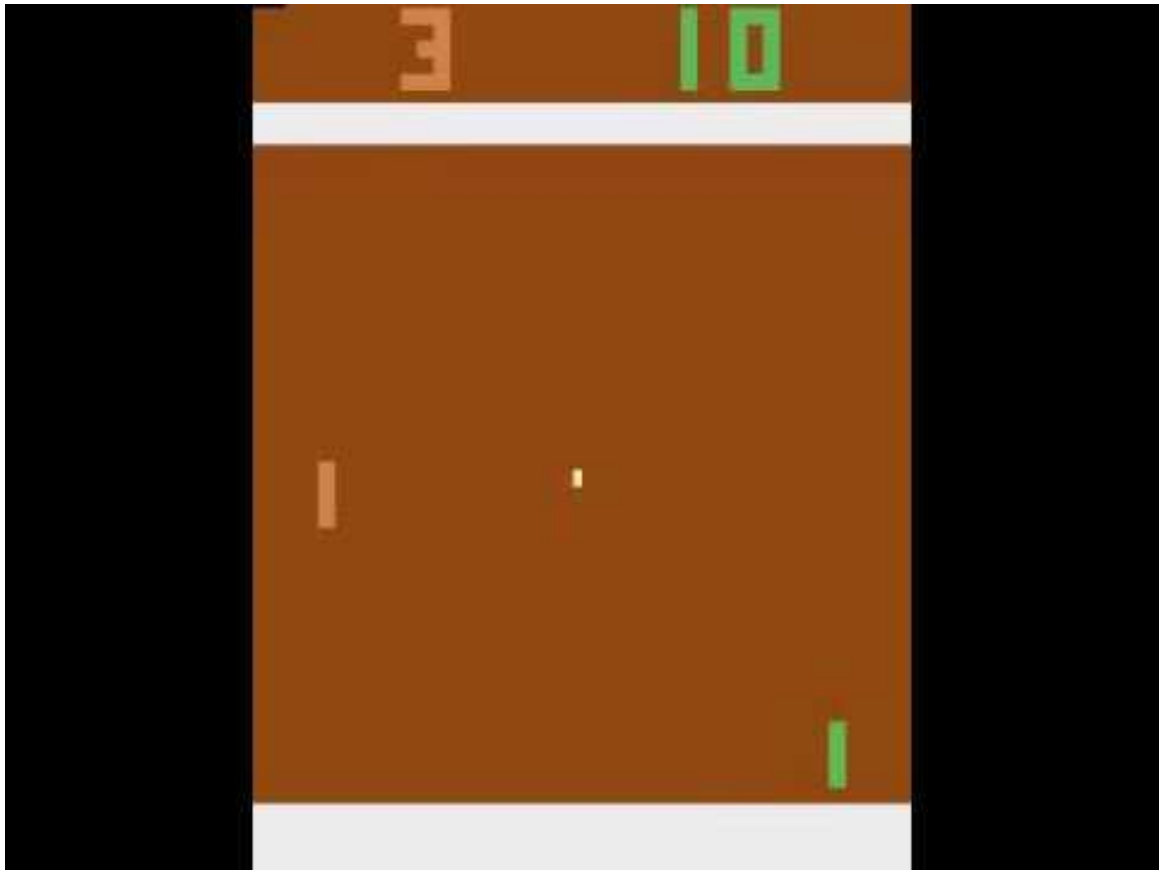
One common choice is to use a discounted reward, so the "eventual reward" would become:

$$R_t = \sum_{k=0}^{\infty} \gamma^k \, r_{t+k}$$

γ is a number between 0 and 1 called *discount factor*

The expression states that the strength with which we encourage a sampled action is the weighted sum of all rewards afterwards, but later rewards are exponentially less important.

Results

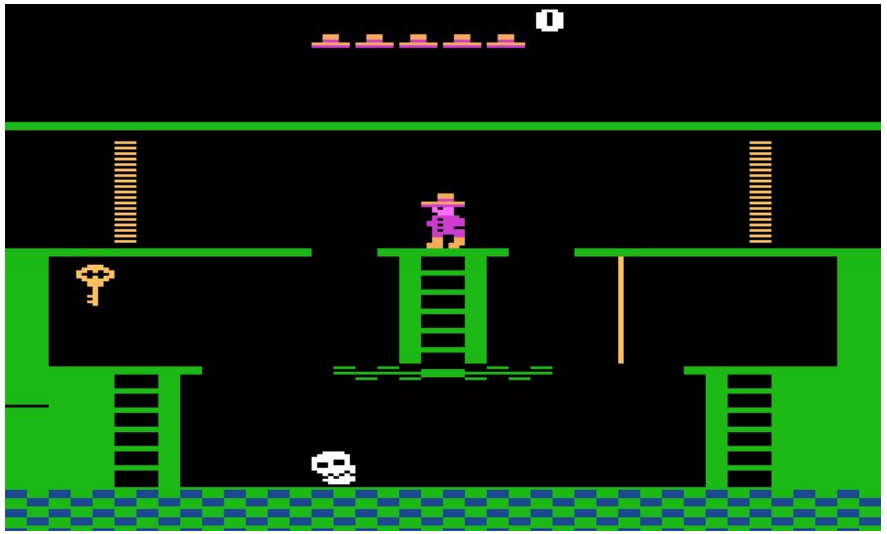


Disadvantages

In Policy Gradients, the models has to actually experience positive award and experience it often to shift the policy towards high rewarding moves.

As the action space increases the policy samples billions of random actions 99% of which will result in negative reward. It becomes harder for the policy to “stumble upon” the positive rewarding situation.

It becomes increasingly difficult for the model to learn a good policy which can maximize the cumulative reward.



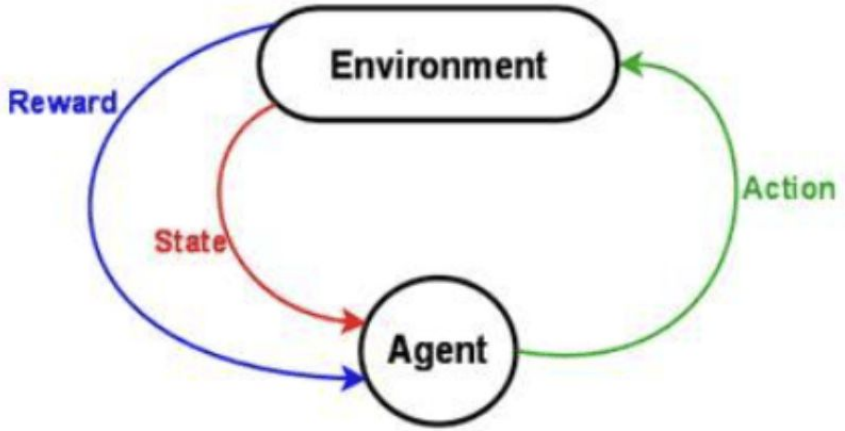
Montezuma's Revenge

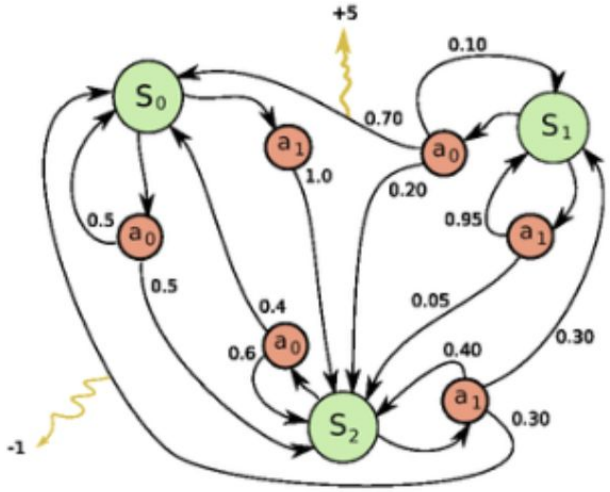
References

These slides are inspired by Andrej Karparthy's Deep Reinforcement Learning blog post

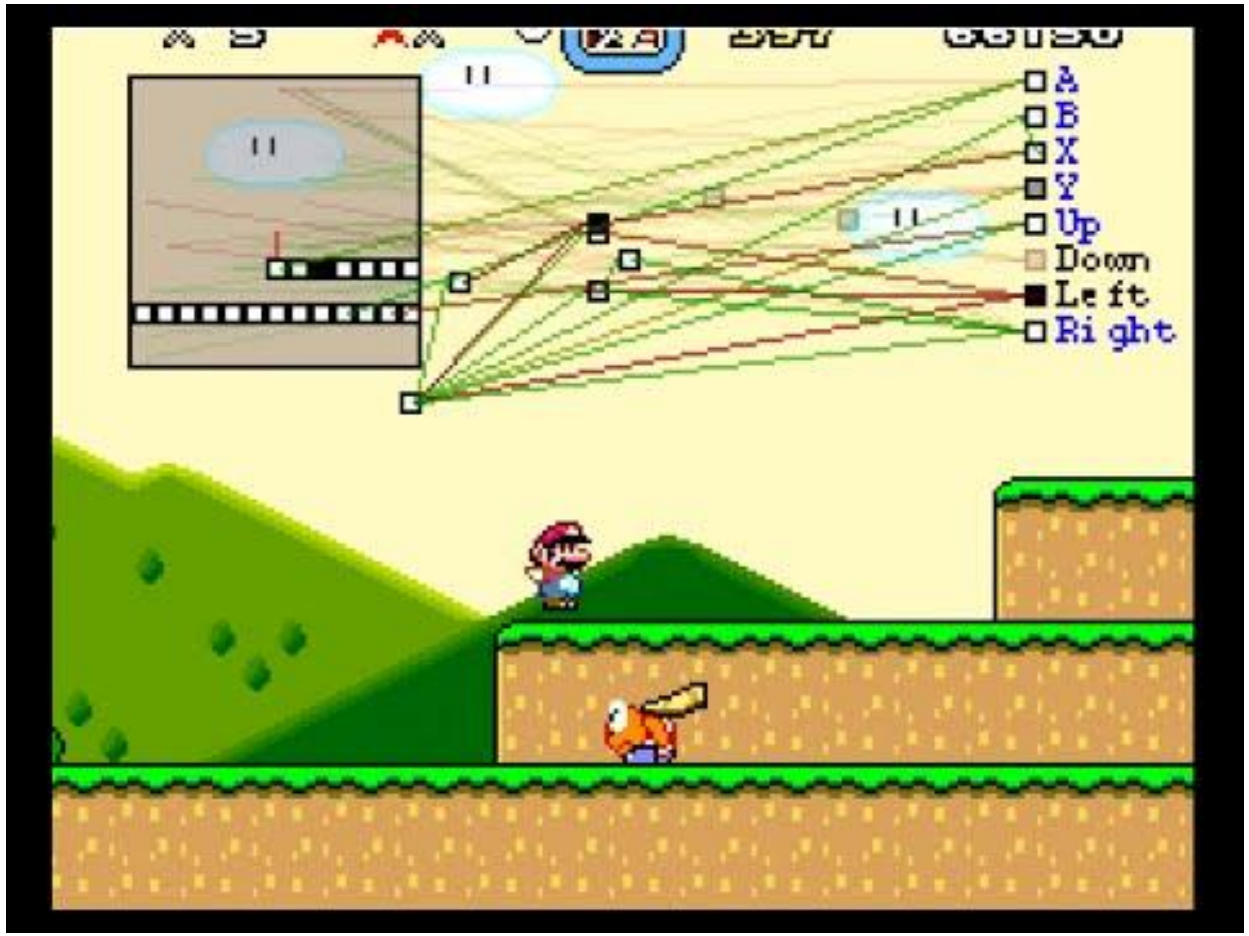
http://karpathy.github.io/2016/05/31/rl/

RL as a Markov Decision Process

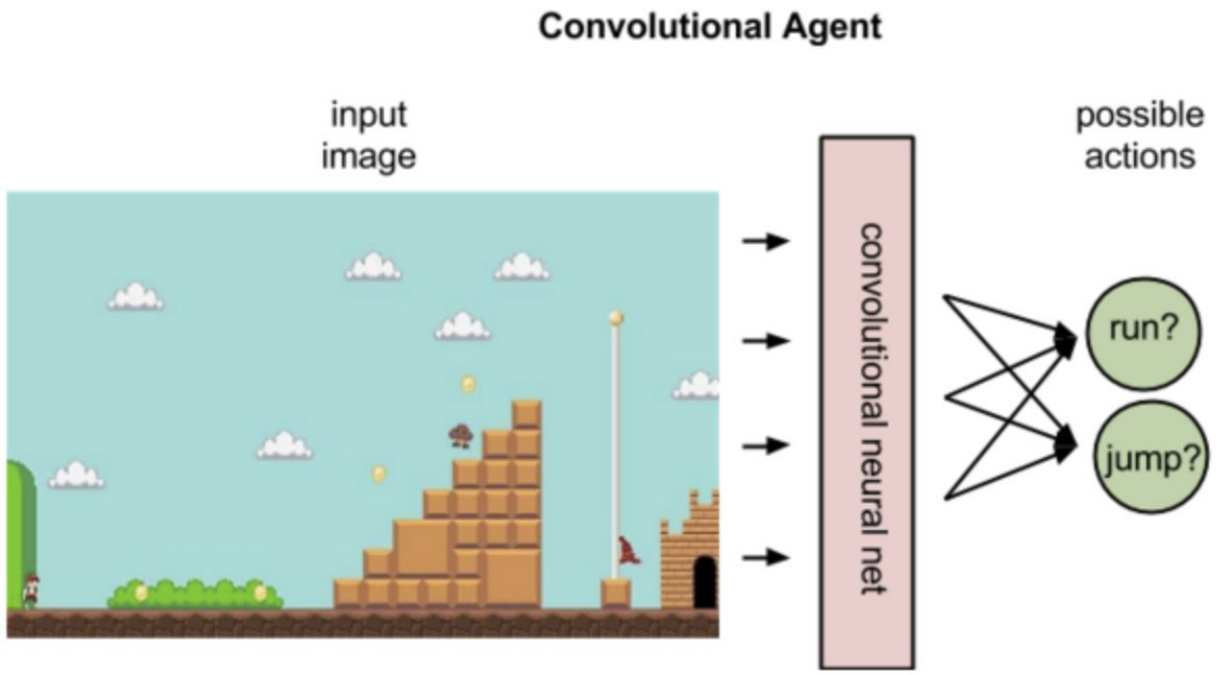




Eg. RL for MarI/O



Goal: Learn a Policy



$$a = \pi(s)$$

Rewards

- Total reward in a markov chain:

$$R = r_1 + r_2 + r_3 + \ldots + r_n$$

- Total Future Reward from time-step t:

$$R_t = r_t + r_{t+1} + r_{t+2} + \ldots + r_n$$

- Discounted Future Reward:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

Q-Learning:

In Q-learning we define a function **Q(s, a)** representing the maximum discounted future reward when we perform action a in state s, and continue optimally from that point on.

In other words: *The best possible score at the end of the game after performing action a in state s*

Policy

- Our goal is to find the optimum policy: that maximizes the reward
- If have our Q function, deriving optimal policy is easy:

$$\pi(s) = argmax_a \, Q(s, a)$$

But how do we get the Q function?

- Recall:

$$R_t = r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) = r_t + \gamma R_{t+1}$$

$$Q(s, a) = r \ + \gamma max_{a'} Q(s', a')$$

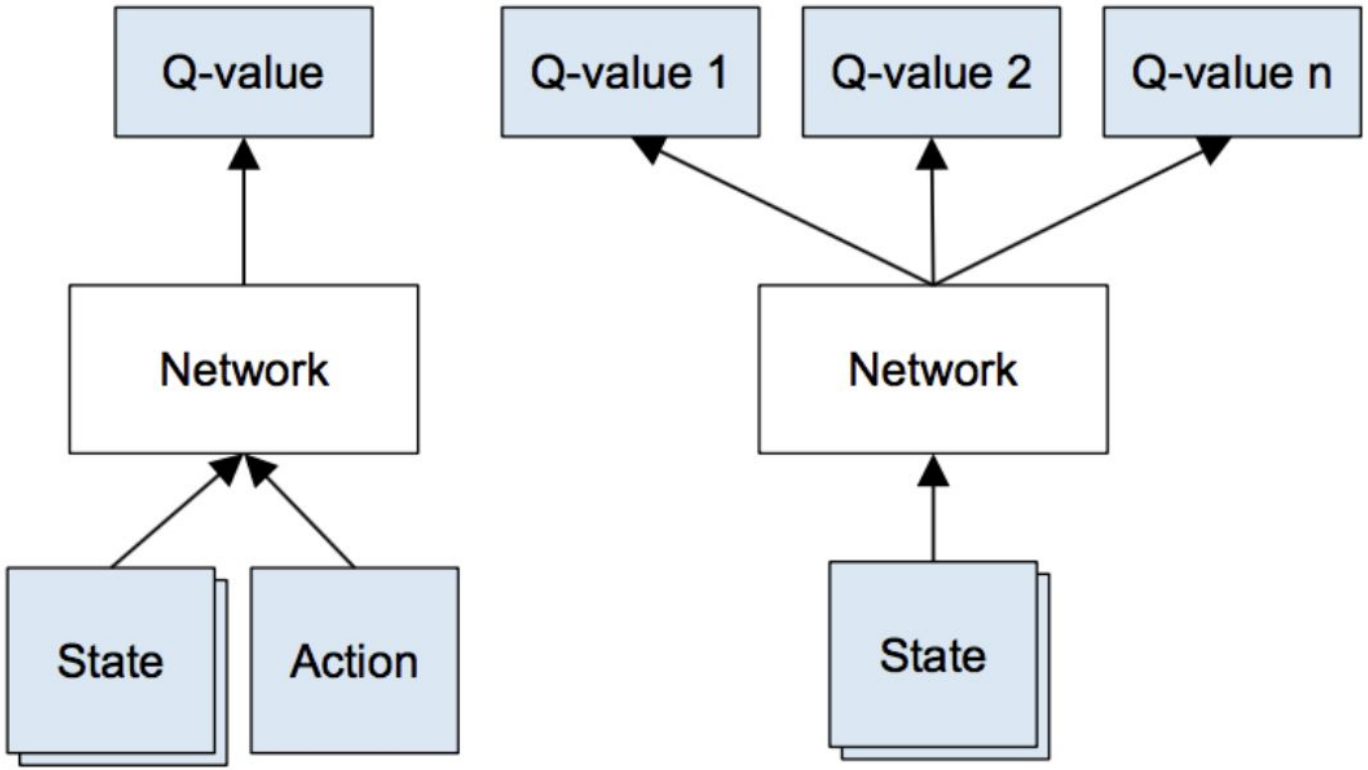
Learning- update rule

$$Q(s_t, a_t) \leftarrow (1-\alpha) \cdot \underbrace{Q(s_t, a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \overbrace{\left(\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1}, a)}_{\text{estimate of optimal future value}} \right)}^{\text{learned value}}$$

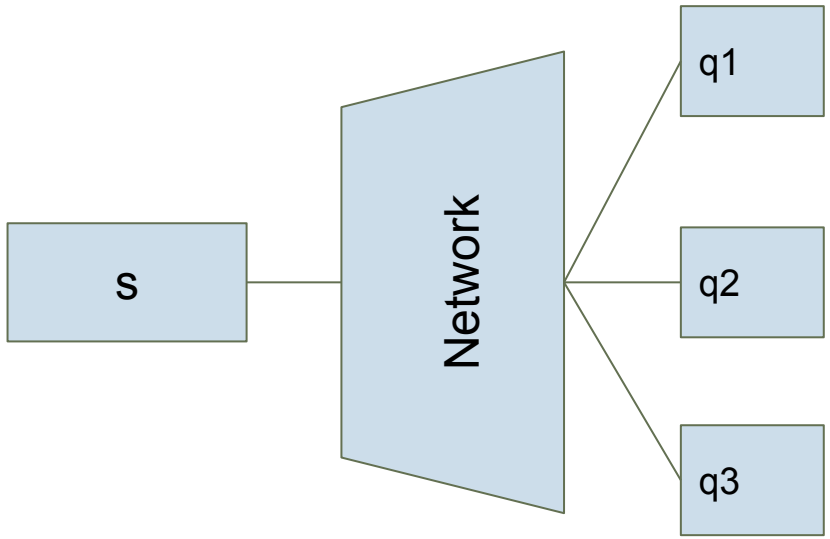
Q-learning Algorithm

initialize $Q[num\_states, num\_actions]$ arbitrarily
observe initial state s
repeat
    select and carry out an action a
    observe reward r and new state s'
    $Q[s,a] = Q[s,a] + \alpha(r + \gamma \max_{a'} Q[s',a'] - Q[s,a])$
    $s = s'$
until terminated

How to implement the Q-value function?

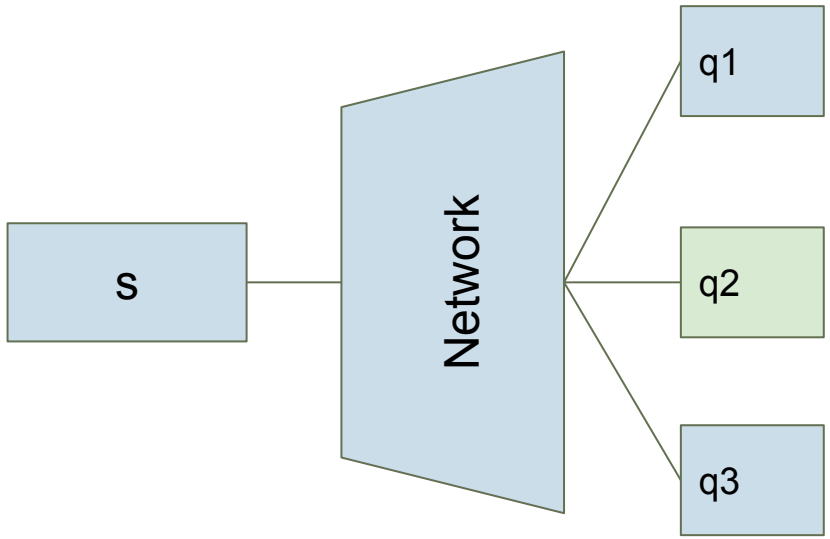


Deep Q-learning

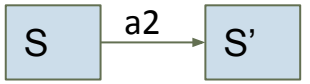


- Do a feedforward pass for the current state s to get predicted Q-values for all actions.

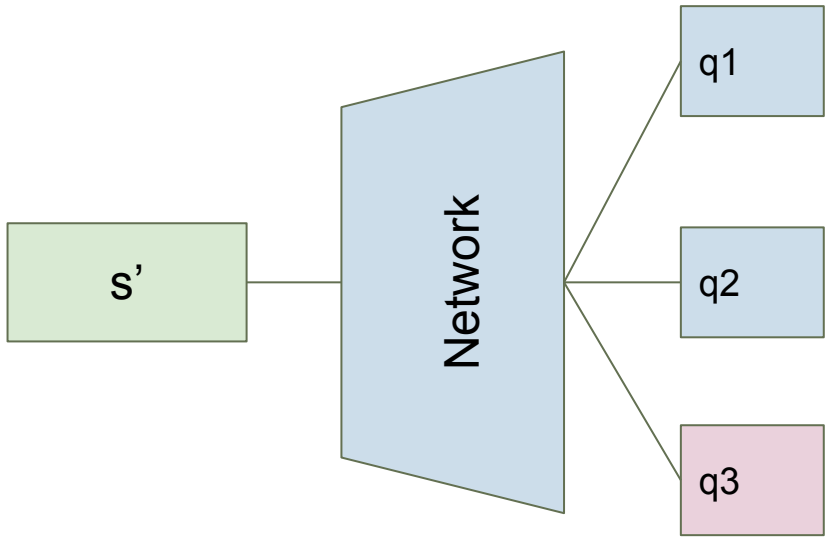
Deep Q-learning



- Do a feedforward pass for the current state s to get predicted Q-values for all actions.
- Sample an action, note the immediate reward r, and move to state s’.

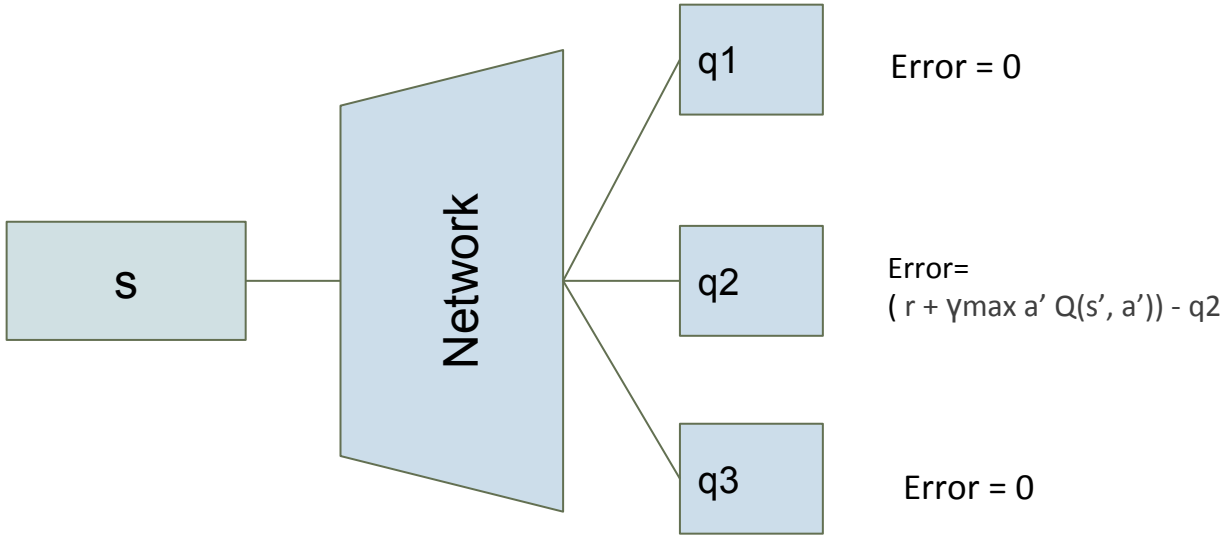


Deep Q-learning



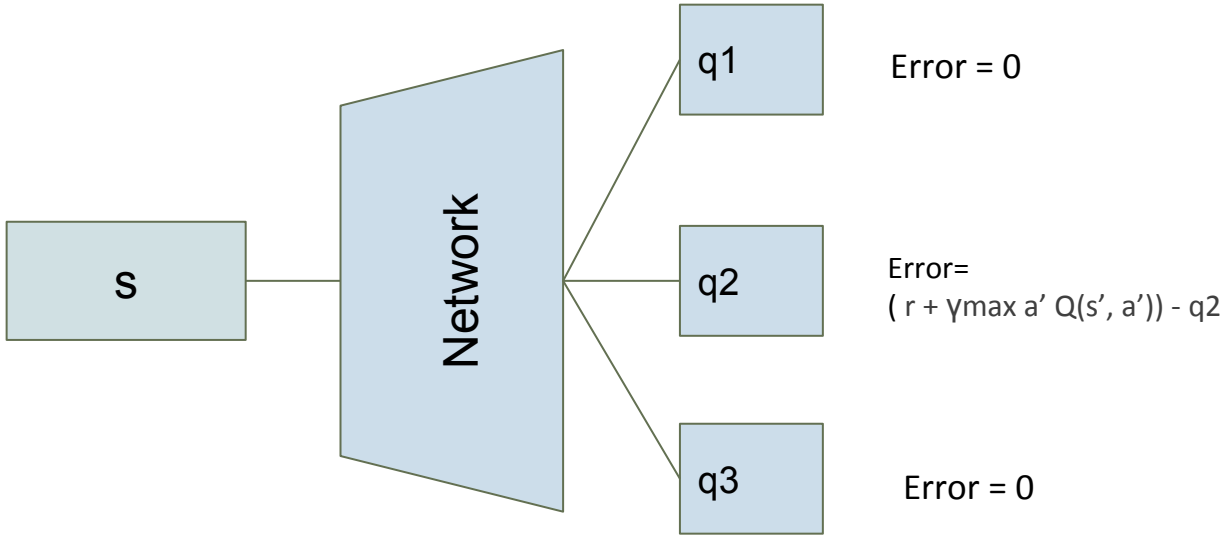
- Do a feedforward pass for the next state s’ and calculate maximum overall network outputs max a’ Q(s’, a’).

Deep Q-learning



- Set Q-value target for action to r + γmax a' Q(s', a') (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.

Deep Q-learning



- Set Q-value target for action to r + γmax a' Q(s', a') (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
- Update the weights using backpropagation.

Summary: Deep Q-learning

$$L = \frac{1}{2}[\underbrace{r + max_{a'}Q(s', a')}_{\text{target}} - \underbrace{Q(s, a)}_{\text{prediction}}]^2$$

- Do a feedforward pass for the current state s to get predicted Q-values for all actions.
- Do a feedforward pass for the next state s’ and calculate maximum overall network outputs max a’ Q(s’, a’).
- Set Q-value target for action to r + γmax a’ Q(s’, a’) (use the max calculated in step 2). For all other actions, set the Q-value target to the same as originally returned from step 1, making the error 0 for those outputs.
- Update the weights using backpropagation.

Conclusion

- Deep Q-learning became popular after the paper "Playing Atari with Deep Reinforcement Learning" from Deepmind.

- Deep Q learning in practice requires several tricks/hacks to work properly - experience replay, target network, error clipping, reward clipping.
  - NOTE - READ and WRITE Explanations

- Many improvements to deep Q-learning have been proposed since its first introduction – Double Q-learning, Prioritized Experience Replay, Dueling Network Architecture and extension to continuous action space to name a few.

How does Q-learning stand against PG?

- Like most RL algorithms, both Q-learning and PG require careful hyper-parameter tuning to work well.
  - Reward definition, discount factor, learning rate, policy network, exploration-exploitation etc.
- PG is preferred because it is end-to-end: there's an explicit policy and a principled approach that directly optimizes the expected reward.
- The authors of the original DQN paper who have shown Policy Gradients to work better than Q Learning when tuned well