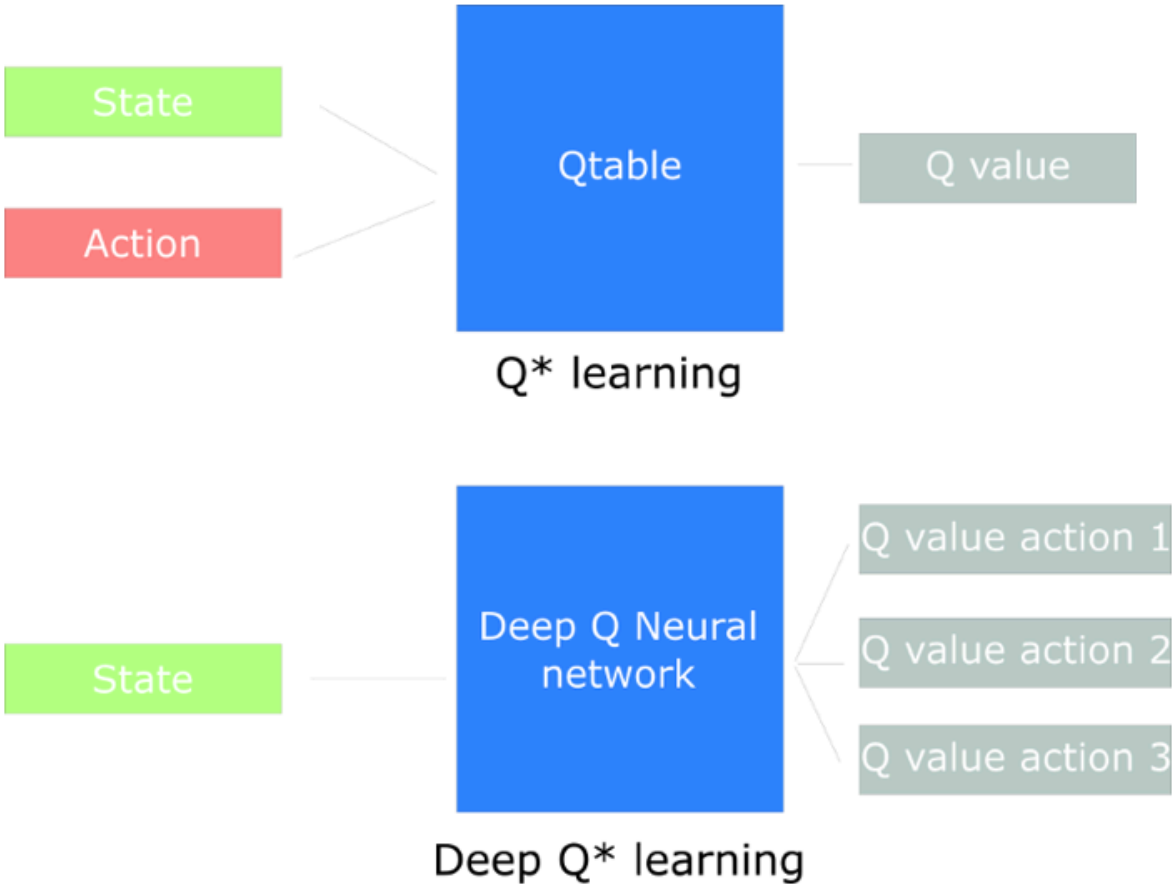
3 - Introduction to Deep Q-Learning

Adding 'Deep' to Q-Learning

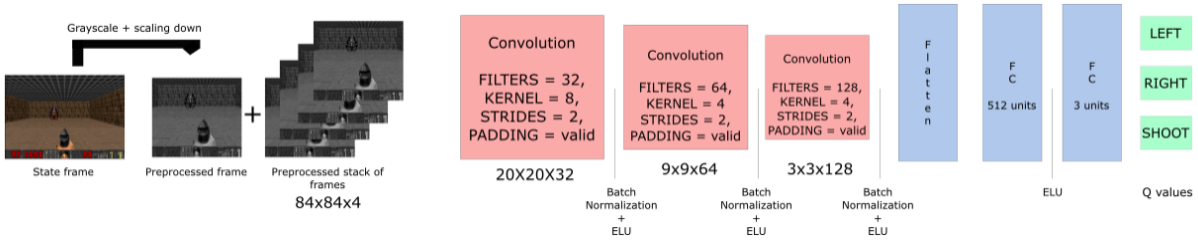
Q-table is basically a cheatsheet that we create over time. Problem with this is that it isn't scalable. For games with gigantic environments and a gigantic **state space** (millions of different states) we need better strategy that is more efficient.

In this case, a neural network that approximates different Q-values for each action is better. Input for this neural network = the given state:



How does Deep Q-Learning Work?

Architecture:



This big architecture describes overall design. It's similar to a convolutional neural network.

Preprocessing

Goal of preprocessing is to reduce the complexity of the states. First step is making image grayscale because color does not add any important information (at least in the case of Doom). This saves a lot of data because we go from three color channels (RGB) to 1 (grayscale). Then, we remove other non-important information by cropping the frame. In this example, the roof is cropped out because it is not important.

Next part of preprocessing is to create a stack of four sub-frames, this is to solve **temporal limitation**.

Relevant reading: https://medium.com/emergent-future/simple-reinforcement-learning-with-tensorflow-part-8-asynchronous-actor-critic-agents-a3c-c88f72a5e9f2
~ idea of using LSTM neural networks for handling problem of temporal limitation.

Essentially, we want our agent to have an idea of how objects in the current frame are moving. Passing a single frame gives no indication of motion, but a little bit of context does. So simply stacking frames will be a sufficient solution for temporal limitation.

Using CNNs

Next, these frames are processed by a CNN, which look at spatial relationships in images and across frames. Note that ELU is now considered activation of choice even above RELU in convolutional layers (it attempts to combine good parts of RELU and Leaky RELU, doesn't have dying RELU problem).

After convolution layers, there is layer to flatten output of convolutions into column vector, which is then passed to a single fully-connected layer with ELU. Finally, an output layer (fully-connected layer with linear activation function) is used to produce Q-value estimation for each action.

Experience Replay:

Experience replay is irefully for handling two things:

- Avoid forgetting previous experience
- Reduce correlations between experiences.

Avoid forgetting previous experiences

Big problem: variability of the weights. Cause: high correlation between actions and states.

General Reinforcement learning process is receiving a tuple (state, action, reward, new_state) at each time step. We learn from this tuple (with deep q-learning this is feeding it into neural network), and then throw away the experience.

Problem is that as time goes by and we get more samples from interactions with the environment to our neural network, agent will forget previous experiences as they are overwritten with new experiences.

Example is a game like Mario. Learning how to solve the water stage will make the agent forget how to solve the first level.

Solution to this is learning the previous experience multiple times.

“Replay Buffer” - stores experience tuples while interacting with the environment. We sample a batch of tuples from the buffer to feed our neural network. Replay buffer is essentially a folder where every sheet is an experience tup[le. It is fed by interacting with the environment, and then a random sheet from the folder is fed to the neural network.

As a result, network does not only learn from what it has immediately done.

Reducing Correlation Between Experiences

Another problem results from the high correlation caused by every action affecting the subsequent state. This implies that we should not train our agent in order, as that would risk the agent being influenced by the effect of this correlation.

An example of thes is a first person shooter where the agent has two actions, shoot left and shoot right, which act upon a monster that can appear on the left or right.

Let’s assume that we know that if we shoot a monster, the probability of the next monster coming from the same direction is 70% (action implying future states).

Then, when we start training, if the agent sees a monster on the right, it will

probably continue seeing more and more monsters on the right, which could lead to the agent not seeing a lot of left examples. Then, after training is over, the agent might shoot the right gun even when there is a monster on the left. This is not a rational choice, instead it's the agent attempting to predict the monster based solely on its past experience of shooting predominantly on the right side.

To deal with this, we need to stop learning while interacting with the environment. Instead, we should try different things and explore with the state space. Save the resulting experiences in the replay bugger.

Then, recall these experiences and learn from them. After that, go back to play with the updated value function.

As a consequence, we will have a better set of examples to generalize patterns from - we won't be focusing on just one region of the state space. As a result, we will avoid reinforcing the same action over and over again. This approach is a form of Supervised Learning.

An alternative way to deal with this problem is "Prioritized experience replay" which is about presenting rare or "important" tuples to the neural network more often.

Deep Q-Learning Algorithm

Recall Bellman equation used to update Q value for a given state and action.

To modify this equation for deep q-learning, we instead update the weights of the neural network with the TD error (different between Q_target, the maximum possible value form the next state, and Q_value, the current prediction of the Q-value)

$$\Delta w = \alpha[(R + \gamma \, max_a \, \hat{Q}(s', a, w)) - \hat{Q}(s, a, w)] \, \nabla_w \hat{Q}(s, a, w)$$

Change in weights

learning rate

Maximum possible Qvalue for the next_state (= Q_target)

Current predicted Q-val

TD Error

Gradient of our current predicted Q-value

Overall process:

- Initialize Doom Environment E
- Initialize replay buffer M with capacity N
- Initialize DQN weights w

```
    for episode in max_episodes:
        s = environment state
        for steps in max_steps:
            choose action a from state s using epsilon greedy
            take action a, get r (reward), and s' (next state)
            store experience tuple <s, a, r, s'> in M
            s = s' ( move to new state)

            get random mini batch of experience tuples from M
            Set Q_target = reward(Sara) + gamma*maxQ(s')
            Update w = alpha * (Q_target - Q_value) * gradient of current
predicted Q value
```