Useful links for extra p

Brief 4 improvements : double /dueling/ fixed-q target / prioritized experience replay
https://medium.freecodecamp.org/improvements-in-deep-q-learning-dueling-double-dqn-prioritized-experience-replay-and-fixed-58b130cc5682

https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-8438a3e2b8df

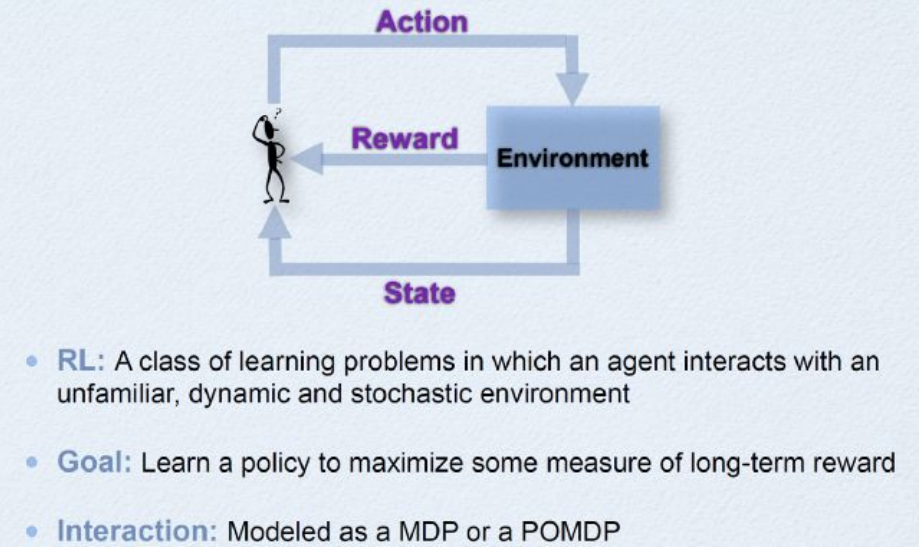
https://jaromiru.com/2016/11/07/lets-make-a-dqn-double-learning-and-prioritized-experience-replay/

Markov Decision Process

Markov property: " *The future is independent of the past given the present."*

Discounted Rewards: "A reward (payment) in the future is not worth quite as much as a reward now

qReinforcement learning



Q learning

Value function: $V^{\pi}(s)$ - the value of a state s under policy π. The expected return when starting in s and following π thereafter

Q Function: $Q^{\pi}(s, a)$ - the value of taking action a in state s under a policy π. The expected return when starting from s taking the action a and thereafter following policy π.

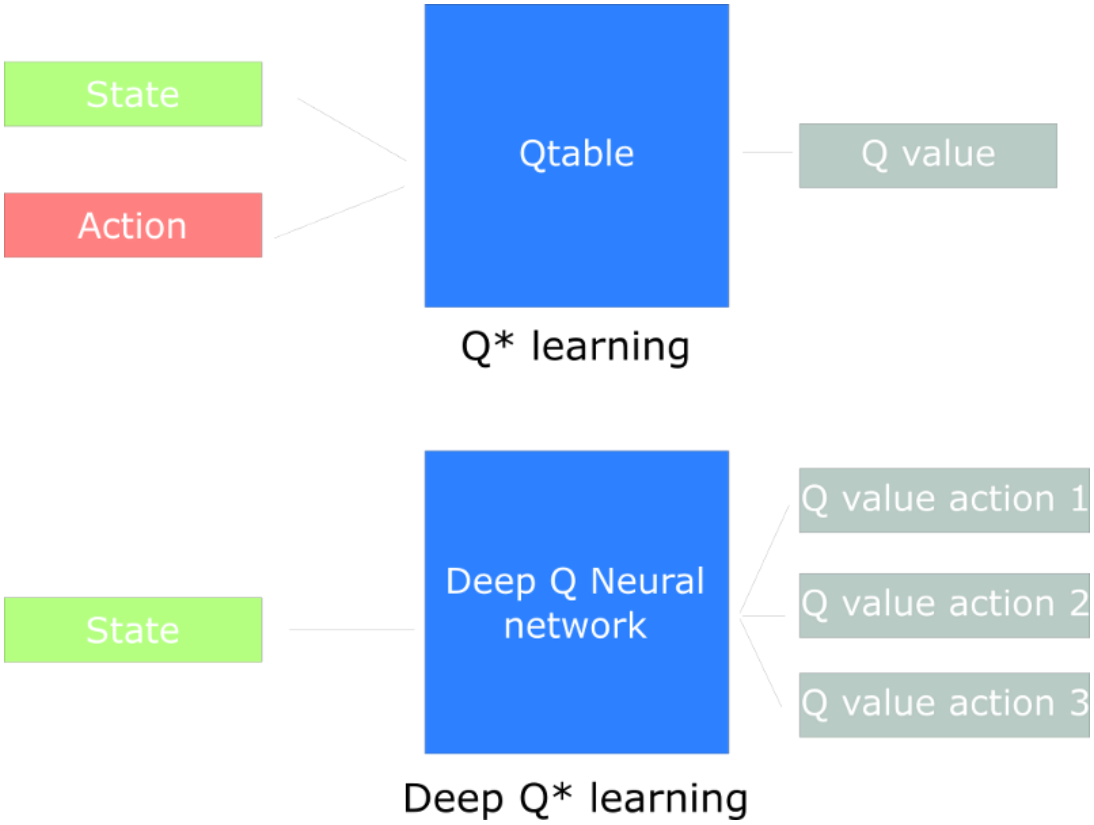
The TD method is called a "bootstrapping" method, because the value is updated partly using an existing estimate and not a final reward.

$$V(s) \leftarrow V(s) + \alpha(\overbrace{r + \gamma V(s')}^{\text{The TD target}} - V(s))$$

Q-Table is just a fancy name for a simple lookup table where we calculate the maximum expected future rewards for action at each state.

Exploitation and Exploration

Deep Q learning VS Q learning



From Basic DQN to Double Dueling DQN + PER

Fixed target network

As a consequence, there is a big correlation between the TD target and the parameters (w) we are changing.
Therefore, it means that at every step of training, our Q values shift but also the target value shifts. So, we're getting closer to our target but the target is also moving. It's like chasing a moving target! This lead to a big oscillation in training.
If not fixed, it is chasing a shifting target

Double DQN

$$\underbrace{Q(s,a)}_{\text{TD target}} = r(s,a) + \gamma Q(s', \underbrace{argmax_a Q(s',a)}_{\text{DQN Network choose action for next state}})$$

Target network calculates the Q value of taking that action at that state

One problem in the DQN algorithm is that the agent tends to overestimate the Q function value, due to the *max* in the formula used to set targets:

$$Q(s,a) \rightarrow r + \gamma max_a Q(s',a)$$

To demonstrate this problem, let's imagine a following situation. For one particular state there is a set of actions, all of which have the same true Q value. But the estimate is inherently noisy and differs from the true value. Because of the *max* in the formula, the action with the highest positive error is selected and this value is subsequently propagated further to other states. This leads to positive bias – value overestimation. This severe impact on stability of our learning algorithm[1].
A solution to this problem was proposed by Hado van Hasselt (2010)[2] and called Double Learning. In this new algorithm, two Q functions – Q_1 and Q_2 – are independently learned. One function is then used to determine the maximizing action and second to estimate its value. Either Q_1 or Q_2 is updated randomly with a formula:

$$Q_1(s,a) \rightarrow r + \gamma Q_2(s', argmax_a Q_1(s',a))$$

or

$$Q_2(s,a) \rightarrow r + \gamma Q_1(s', argmax_a Q_2(s',a))$$

It was proven that by decoupling the maximizing action from its value in this way, one can indeed eliminate the maximization bias[2].
By decoupling the action choice from the target Q-value generation, we are able to substantially reduce the overestimation, and train faster and more reliably.

Prioritized Experience Replay with DDQN

Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}(S_j, \arg\max_a Q(S_j, a)) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

Some explanation:

- The magnitude of the TD-error indicates how unexpected a certain transition was
- The TD-error can be a poor estimate about the amount an agent can learn from a transition when rewards are noisy
- Problems with greedily selecting experiences:
  - High-error transitions are replayed too frequently
  - Low-error transitions are almost entirely ignored
  - Expensive to update entire replay memory, so errors are only updated for transitions that are replayed
  - Lack of diversity leads to over-fitting
- A stochastic sampling method is introduced which finds a balance between greedy prioritization and random sampling (current method)
- Two variants of $P(i)=p_i^\alpha / \sum_k p_k^\alpha$
- were studied, where P is the probability of sampling transition i, $p_i>0$ is the priority of transition i, and the exponent α determines how much prioritization is used, with $\alpha=0$ the uniform case
  - Variant 1: proportional prioritization, where $p_i=|\delta_i|+\epsilon$ is used and ϵ is a small positive constant that prevents the edge-case of transitions not being revisited once their error is zero. δ is the TD-error
  - Variant 2: rank-based prioritization, with $p_i=\frac{1}{\text{rank}(i)}$ where $\text{rank}(i)$ is the rank of transition i when the replay memory is sorted according to δ_i
- **Key insight** The estimation of the expected value of the total discounted reward with stochastic updates requires that the updates correspond to the same distribution as the expectation. Prioritized replay introduces a bias that changes this distribution

uncontrollably. This can be corrected by using importance-sampling (IS) weights $w_i=(1/N*1/P(i))^{\beta}$

- that fully compensate for the non-uniform probabilities $P(i)$ if $\beta=1$. These weights are folded into the Q-learning update by using $w_i \times \delta_i$, which is normalized by $1/\max_i w_i$
- IS is annealed from β_0 to 1, which means its affect is felt more strongly at the end of the stochastic process; this is because the unbiased nature of the updates in RL is most important near convergence
- To remind myself, PER is not simply heapq
- this is a form of stratified sampling that has the added advantage of balancing out the minibatch (there will always be exactly one transition with high magnitude δ, one with medium magnitude, etc)

How to calculate the priority?

$$p = (error + \epsilon)^{\alpha}$$

(Error is TD error)Epsilon ϵ is a small positive constant that ensures that no transition has zero priority. Alpha, $0 \leq \alpha \leq 1$, controls the difference between high and low error. It determines how much prioritization is used. With $\alpha = 0$ we would get the uniform case.

Why do we need experience replay?

- The problem with a bunch of consecutive, highly correlated updates for similar states is that training a Neural Network with that kind of input is problematic; a Neural Network is prone to "forgetting" things it has learned previously for certain inputs if you don't repeatedly keep showing it those kinds of inputs.

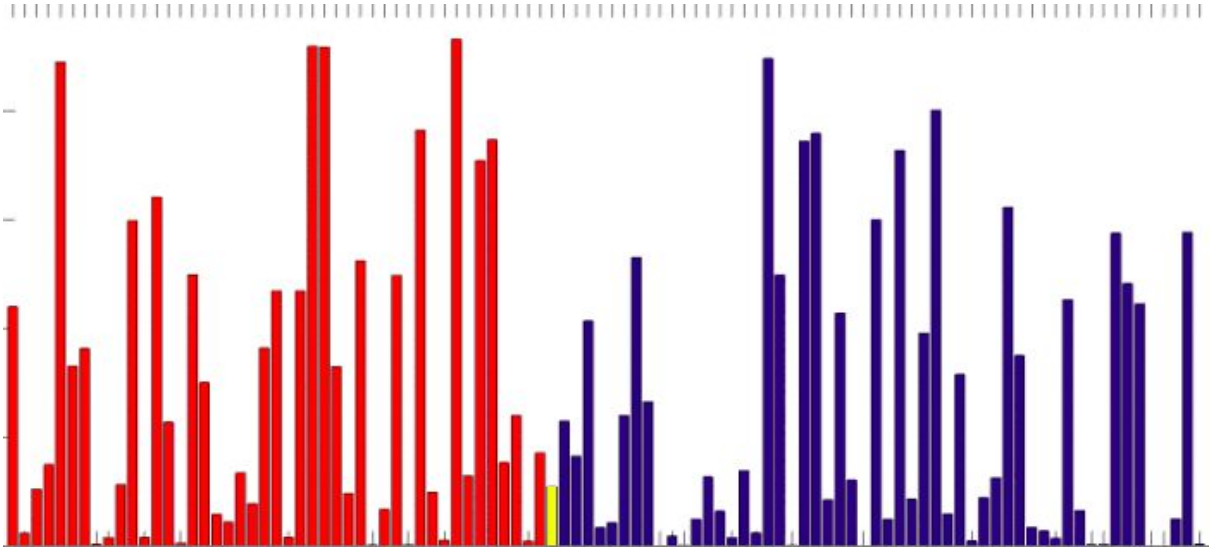
Why do we need prioritized experience replay?

- Some experience is more important than other

How do we deal with the bias introduced by PER?

- Importance Sampling Weight

The following graph explains the idea of using SumTree



Algorithm 1 Double DQN with proportional prioritization

Input: minibatch k, step-size η, replay period K and size N, exponents α and β, budget T.
Initialize replay memory $\mathcal{H} = \emptyset$, $\Delta = 0$, $p_1 = 1$
Observe S_0 and choose $A_0 \sim \pi_\theta(S_0)$
for $t = 1$ **to** T **do**
  Observe S_t, R_t, γ_t
  Store transition $(S_{t-1}, A_{t-1}, R_t, \gamma_t, S_t)$ in $\mathcal{H}$ with maximal priority $p_t = \max_{i<t} p_i$
  if $t \equiv 0 \mod K$ **then**
    for $j = 1$ **to** k **do**
      Sample transition $j \sim P(j) = p_j^\alpha / \sum_i p_i^\alpha$
      Compute importance-sampling weight $w_j = (N \cdot P(j))^{-\beta} / \max_i w_i$
      Compute TD-error $\delta_j = R_j + \gamma_j Q_{\text{target}}\left(S_j, \arg\max_a Q(S_j, a)\right) - Q(S_{j-1}, A_{j-1})$
      Update transition priority $p_j \leftarrow |\delta_j|$
      Accumulate weight-change $\Delta \leftarrow \Delta + w_j \cdot \delta_j \cdot \nabla_\theta Q(S_{j-1}, A_{j-1})$
    end for
    Update weights $\theta \leftarrow \theta + \eta \cdot \Delta$, reset $\Delta = 0$
    From time to time copy weights into target network $\theta_{\text{target}} \leftarrow \theta$
  end if
  Choose action $A_t \sim \pi_\theta(S_t)$
end for

1. Define global var, GAMMA, **INITIAL_EPSILON, FINAL_EPSILON** etc.
2. Create environment
3. Q evaluate network, ensure the shape of last layer is (ACTION_DIM, 1), the output layer is `q_values`, `q_values` will be used in `explore()` function to select next action
4. Q target network, exactly same structure as Q evaluate network
5. Define replace_op, copy Q-evaluate to Q-target network
6. Define loss, and train_op
7. Define Experience_Replay buffer, and store_memory function
8. Open session
9. [optional] tf.summary.FileWrite();
10. Define the training iteration,
    a. remember to set the reward to -1 when done!
    b. Every 300 iterations, run the `replace_op`
    c. For experience sampling, do not sample out invalid data
    d. Get the value of `q_target`, reward + gamma * Q(s,a), a is the action the agent actually chose
    e. Run `train_op` and `loss` to actually update the Q network

Change Double DQN

1. Feed `eval_net` and `target_net` with `next_state`
2.