4 - Improvements in Deep Q-Learning: Dueling Double DQN, Prioritized Experience Replay, fixed Q-targets

Deep Q-Learning first introduced in 2014. There are a few strategies developed since then that have dramatically improved it:

- Fixed Q-targets
- Double DQNs
- Dueling DQN (DDQN)
- Prioritized Experience Replay

Fixed Q-targets

Theory:

In Deep Q-Learning, calculating TD error is done by calculating the difference between the TD target (Q_target) and the current Q_value (estimation of Q)

$$\underbrace{\Delta w}_{\text{Change in weights}} = \underset{\text{learning rate}}{\alpha}[\underbrace{(\underbrace{R + \gamma\, max_a\, \hat{Q}(s', a, w)}_{\text{Maximum possible Qvalue for the next\_state (= Q\_target)}}) - \underbrace{\hat{Q}(s, a, w)}_{\text{Current predicted Q-val}}}_{\text{TD Error}}]\ \underbrace{\nabla_w \hat{Q}(s, a, w)}_{\text{Gradient of our current predicted Q-value}}$$

But this doesn't actually make much sense, because we don't have any idea of the real TD target, it is an estimation. Using the Bellman equation, we see that TD target is the reward of taking that action at that state and adding the discounted highest Q_value for the next state.

So, we are using the same parameters (weights) for estimating the target and the Q value. As a result, there is a big correlation between the TD target and the parameters (w) that are changing.

- At every point in training, Q-values shift but that causes a shift in the target value, so we are chasing a moving target

This leads to a big oscillation in training

DeepMind came up with the idea of fixed Q-targets:
- Use a separate network with a fixed parameter (w-) for estimating the TD target
- Every tau steps, update the fixed parameters

In actual implementation, this is how its done:
1. Instantiating networks
    1. Instantiate DQNetwork
    2. Instantiate target Network
2. Create function update_target_graph() that will take DQNetwork parameters and copy them to TargetNetwork
    1. Get parameters of DQNetwork and target network
    2. Copy DQNetwork parameters to target_network parameters and list op_holder
    3. Return op_holder
3. During training, update target network every tau steps (tau is tunable hyper parameter)

$$\underbrace{\Delta w}_{\text{Change in weights}} = \underbrace{\alpha}_{\text{learning rate}} [\underbrace{(\underbrace{R + \gamma \, max_a \, \hat{Q}(s', a, \boxed{w^-}}_{\text{Maximum possible Qvalue for the next\_state (= Q\_target)}}) - \underbrace{\hat{Q}(s, a, \boxed{w})}_{\text{Current predicted Q-val}}}_{\text{TD Error}}] \, \underbrace{\nabla_w \hat{Q}(s, a, w)}_{\text{Gradient of our current predicted Q-value}}$$

At every T steps:

$$\underbrace{w^- \leftarrow w}_{\text{Update fixed parameters}}$$

In pic above, T is actually tau

Double DQNs

This method is used to handle problem of overestimating Q-values.

Recall calculating TD target with bellman equation:

$$\underbrace{Q(s,a)}_{\text{Q target}} = \underbrace{r(s,a)}_{\text{Reward of taking that action at that state}} + \underbrace{\gamma max_a Q(s',a)}_{\text{Discounted max q value among all. possibles actions from next state.}}$$

In above scheme, we are assuming that the best action for the next state is the action with the highest Q-value.

We know that accuracy of q values depends on

- What action we tried
- What neighboring states we explored

So, at the beginning of training, it’s probably not the highest q-value that is the best — we don’t have enough information from exploring the state space to know the best action to take. We don’t want to get false positives by choosing the maximum q-value action every time.

Solution: When computing the Q target, use two networks to decouple the action selection rom the target Q value generation.

- Use DQN network to select best action for next state (simple action with highest q value)
- Use target network to calculate the target Q value of taking that action at the next state.

$$\underbrace{Q(s,a)}_{\text{TD target}} = r(s,a) + \gamma \underbrace{Q(s', \underbrace{argmax_a Q(s',a)}_{\text{DQN Network choose action for next state}})}_{\text{Target network calculates the Q value of taking that action at that state}}$$

In this scheme, Double DQN helps reduce overestimation of q values and results in faster training and more stable learning.

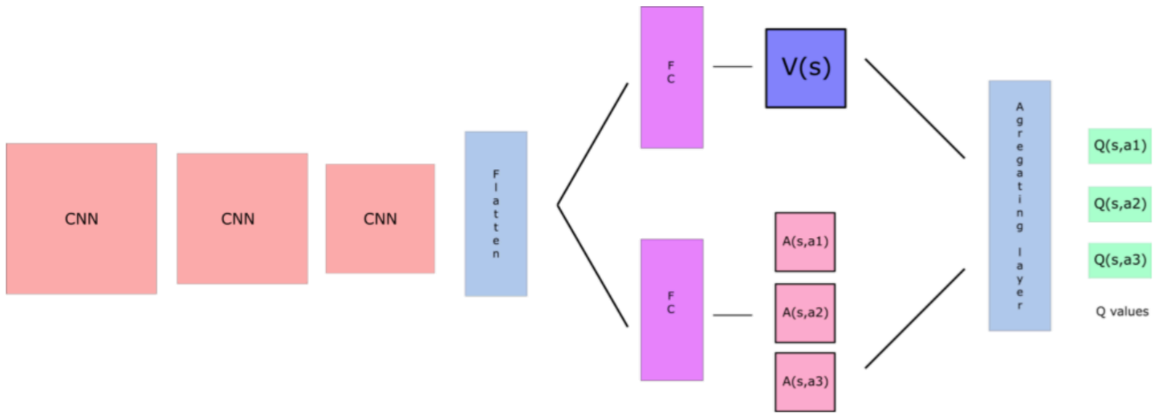
Dueling DQN (aka DDQN)

Theory:
Q-values correspond to how good it is to be at that state and taking an action at the state Q(s,a).

Q(s,a) can be decomposed as follows:
- V(s) : the value of being at that state
- A(s,a) : the advantage of taking that action at that state (how much better it is to take this action vs. all other possible actions at that state.

$$Q(s,a) = A(s,a) + V(s)$$

With Dueling DQN, the idea is separating the estimator for each of these elements. In the architecture, this is shown as follows:

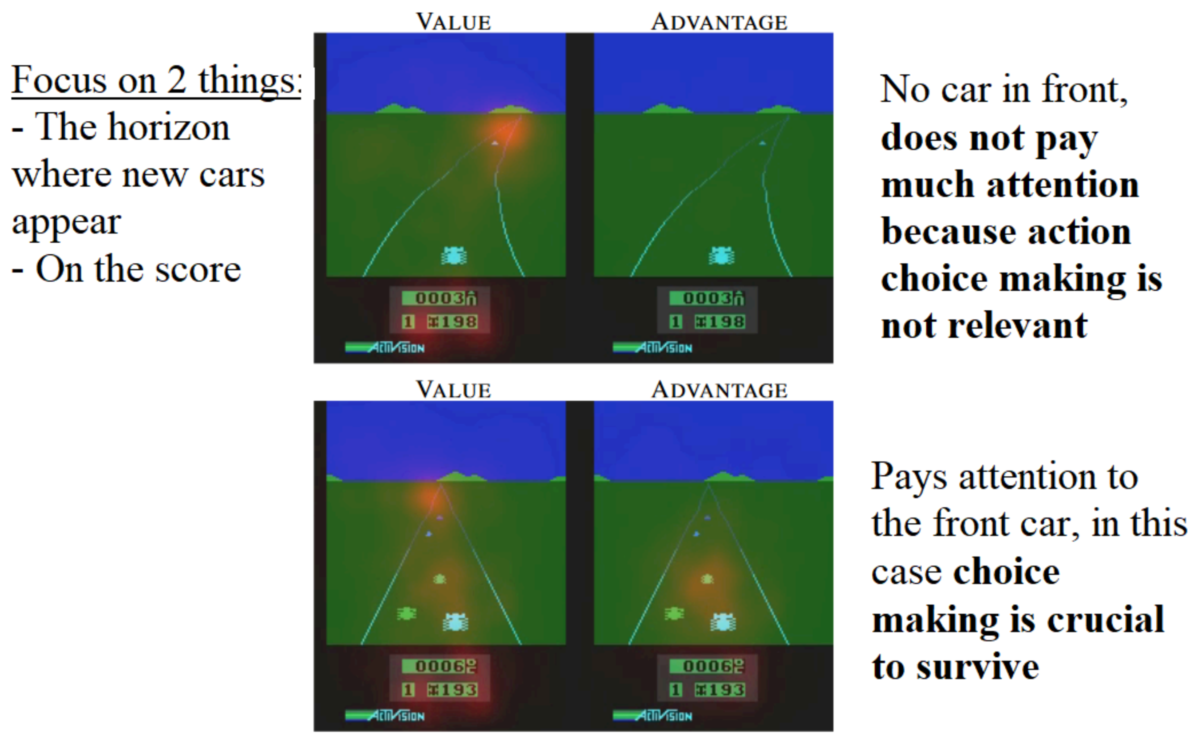


The top stream is a fully-connected layer responsible for estimating the state Value V(s). The lower stream is a fully-connected layer that estimates the advantage for each action A(s,a). Both streams are then joined back together with an aggregate layer. This results in our estimate of Q(s,a).

The reason it makes more sense to separate the estimation of the state and action advantages, before ultimately recombining them is a way for the DDQN to figure out what states are valuable or not without having to learn the effect of each action at each state.

With the normal design, we have to calculate the value of each action at that state. But this is wasteful if that state is bad (for example, all following actions lead to death).

- So, for states where actions don’t affect the environment in a relevant way, decoupling the estimation of V(s) we avoid unnecessary calculation.



Focus on 2 things:
- The horizon where new cars appear
- On the score

No car in front, **does not pay much attention because action choice making is not relevant**

Pays attention to the front car, in this case **choice making is crucial to survive**

Above is saliency maps of trained advantage and value streams produced by computing Jacobians of the streams with respect to the input video. In the first time-step (top pair of images) we see that the value stream pays close attention to the horizon where new cars appear and the score. The advantage stream, however, does not pay attention to anything - it doesn’t need to when there are no cars in front of it. In the second pair of images, however, the advantage stream does show saliency to the area in front of the player, as there is a car immediately in front and the choice of action is very relevant.

How to do the aggregation?

We can’t simply combine the streams by adding the two streams together. The problem with this is that given Q(s,a) we won’y be able to find A(s,a) and V(s). This is a problem for back-propagation. Instead, force the advantage function estimator to have 0 advantage at the chosen action.

$$Q(s,a;\underbrace{\boxed{\theta}}_{\text{Common network parameters}},\underbrace{\boxed{\alpha}}_{\text{Advantage stream parameters}},\underbrace{\boxed{\beta}}_{\text{Value stream parameters}}) = V(s;\theta,\beta) + \left(A(s,a;\theta,\alpha) - \underbrace{\frac{1}{A}\sum_{a'} A(s,a';\theta,\alpha)}_{\text{Average advantage}}\right)$$

Prioritized Experience Replay

PER is based off idea that some experiences may be more important than others for training, but might occur less frequently.

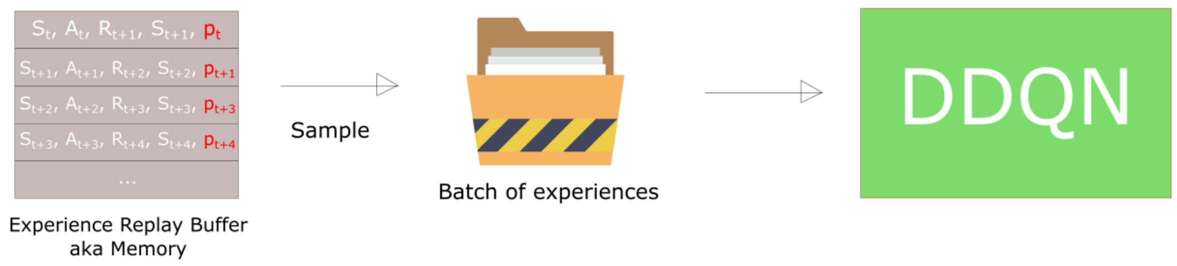
Because we sample from the batch uniformly, we are essentially selecting experiences randomly and rich experiences that occur rarely have very low chance of being selected.

PER, therefore, is about changing the sampling distribution by using a criterion to define the priority of each tuple of experience.

We want to take in priority experience where the r is a big difference between our prediction and the TD target, because it means that we have a lot to learn about it.

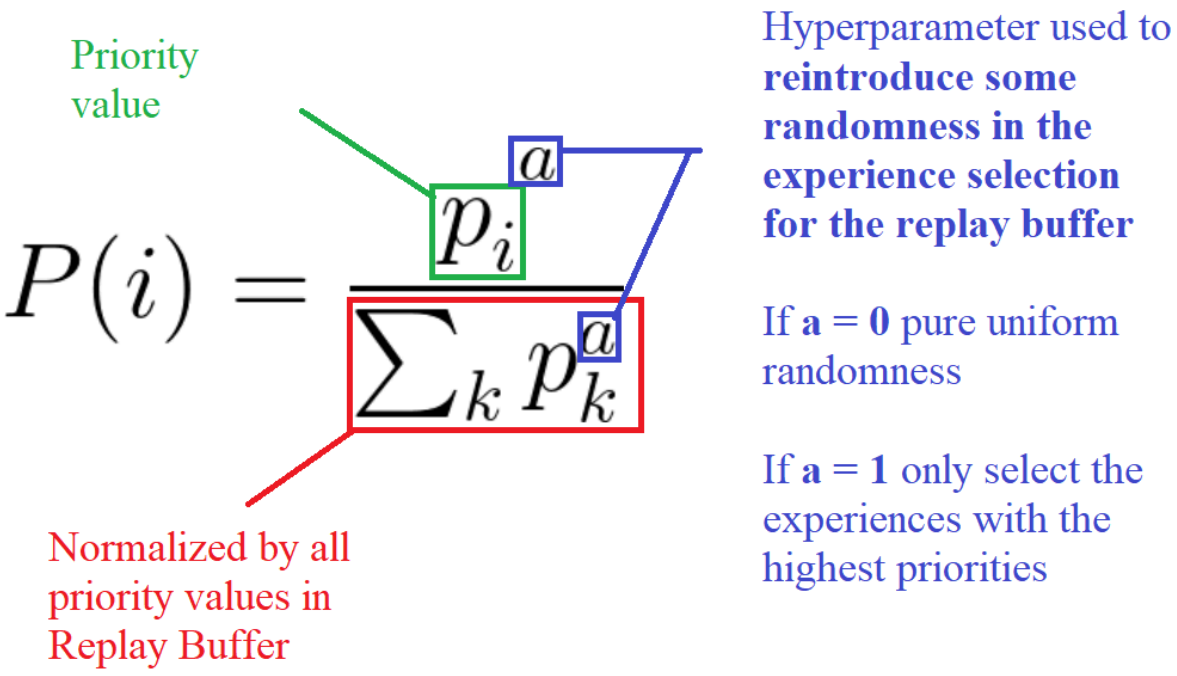
Use the absolute value of the magnitude of our TD error: Then put that priority in the experience of each replay bugger.

$$p_t = \underbrace{|\delta_t|}_{\text{Magnitude of our TD error}} + \underbrace{e}_{\text{Constant assures that no experience has 0 probability to be taken.}}$$


$S_t, A_t, R_{t+1}, S_{t+1}, p_t$
$S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, p_{t+1}$
$S_{t+2}, A_{t+2}, R_{t+3}, S_{t+3}, p_{t+3}$
$S_{t+3}, A_{t+3}, R_{t+4}, S_{t+4}, p_{t+4}$
...
Experience Replay Buffer
aka Memory
Sample
Batch of experiences
DDQN

There is one issue with this: it is a greedy method that will always lead to training the same experiences (those with big priorities), causing over-fitting.

Instead, we introduce stochastic prioritization, which generates the probability of being chosen for a replay:



$$P(i) = \frac{p_i^a}{\sum_k p_k^a}$$

Priority value
Hyperparameter used to **reintroduce some randomness in the experience selection for the replay buffer**
If **a = 0** pure uniform randomness
If **a = 1** only select the experiences with the highest priorities
Normalized by all priority values in Replay Buffer

I DON'T REALLY UNDERSTAND PER YET