Deep Q-Learning Improvements

December 13, 2020 7:26 PM

- Several improvements to the original Deep Q-Learning algorithm have been suggested.
- Double DQN, Prioritized Experience Replay, Dueling DQN

Double DQN

- Deep Q-Learning tends to overestimate action values. Double Q-Learning has been shown to work well in practice to help with this.
- Focusing on the TD Target, the max operation is necessary to find best possible value we can get for the next state.

$$\Delta\mathbf{w} = \alpha\Big(R + \gamma \max_a \hat{q}(S', a, \mathbf{w}) - \hat{q}(S, A, \mathbf{w})\Big)\nabla_{\mathbf{w}}\hat{q}(S, A, \mathbf{w})$$

$$\downarrow$$

$$R + \gamma\hat{q}\Big(S', \arg\max_a \hat{q}(S', a, \mathbf{w}), \mathbf{w}\Big)$$

- We can rewrite the TD target, like above. When we do this we can see that it possible for the arg max to make a mistake especially in the early stages.
- **Why?** Cause Q values are still evolving and we are not likely to choose a the best action. In that case we are choosing among a set of noisy numbers.
- How do we make our choice more robust?
- Double Q-Learning: Select the best action using one set of parameter w and evaluate it using a different set of parameters w' (w prime). Two separate function approximations.

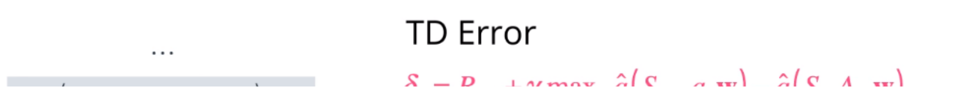
$$R + \gamma \hat{q}\left(S', \arg\max_{a} \hat{q}(S', a, \overset{\text{select best action}}{\mathbf{w}}), \underset{\text{evaluate that action}}{\mathbf{w}'}\right)$$

- The two functions must agree with the same action. If w' does not agree with w, then the Q-value returned is not that high.
- In the long run, this prevents the algorithm from propagating incidental higher rewards that may have been achieved by chance and don't reflect long term returns.
- **Where do we get these parameters w' from?** We can use w- (w mins) for w', since w- is frozen for a while and different enough from w which is updated often.
- Overall, making this small change will prevent this the q-values from exploding in early stages of learning or fluctuating later on.
- If you'd like to dig deeper into how Deep Q-Learning overestimates action values, please read this **research paper.**

Prioritized Experience Replay

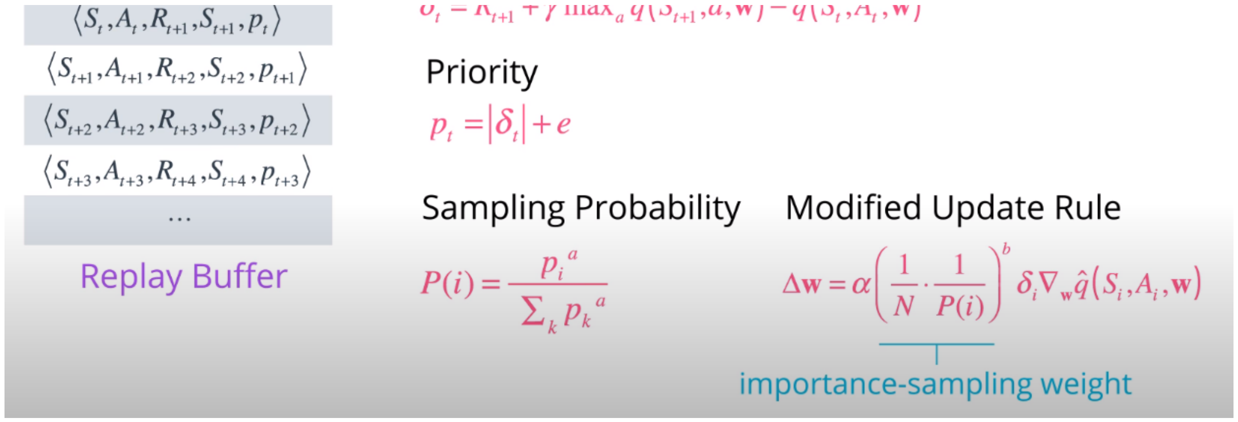
- Deep Q-Learning samples experience transitions *uniformly* from a replay memory.
- **Prioritized experienced replay** is based on the idea that the agent can learn more effectively from some transitions than from others, and the more important transitions should be sampled with higher probability.
- Recall we interact with the environment to collect experience tuples, save them in a buffer and sample a batch to learn from later. This helps us break the correlation between consecutive experiences and stabilizes our learning algorithms.

- Some of the experiences may be more important than others, but may also occur infrequently. Since buffers are practically limited in capacity, older important experiences may get lost.
- **What criteria should we use to assign priority? TD delta error.**
- The larger the error, the more we can learn from the experience. Next idea is to store the magnitude of the TD error as p within the tuples we are storing. When creating batches, we can use this value to compute a sampling probability P(i)
- When a tuple is picked, we can update its priority with a newly computed TD error using the latest q values.
- This method has been proven to reduce the number of batches needed to learn a value function.
- Couple problems: Starvation
  - If the TD-Error is 0, then the probability of picking action a will also be zero
  - 0 doesn't mean we don't have anything to learn from the tuple, it just means our sample was close due to the limited samples visited at that point.
  - Add a small constant e to every priority value
- Improper distributing using the priority to replay a small subset of experiences , resulting in an overfitting to that subset.
- To solve this we use parameter 'a', a constant that controls the randomness.
  - 'a' = 0, pure randomness
  - 'a' = 1, uses priorities
- We have to make one adjustment to our update rule. Remember that our original Q learning update is derived from an expectation over all experiences. When using an stochastic update rule, the way we sample these experiences must match the underlying distribution they came from.
- Thus idea is preserved when we sample experience tuples uniformly from the replay buffer, but this is violated when we use non-uniform sampling (aka priority)
- We must add hyperparameter 'b' to scale the change in weights. These are more important to the end of learning when your q values begin to converge. So you can scale increase 'b' from a low value to one over time.



TD Error

$\delta = R + \gamma \max \hat{q}(S, a, w) - \hat{q}(S, A, w)$



$\langle S_t, A_t, R_{t+1}, S_{t+1}, p_t \rangle$

$\langle S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, p_{t+1} \rangle$

$\langle S_{t+2}, A_{t+2}, R_{t+3}, S_{t+3}, p_{t+2} \rangle$

$\langle S_{t+3}, A_{t+3}, R_{t+4}, S_{t+4}, p_{t+3} \rangle$

…

Replay Buffer

$$\delta_t = R_{t+1} + \gamma \max_a q(S_{t+1}, a, \mathbf{w}) - q(S_t, A_t, \mathbf{w})$$

Priority

$$p_t = |\delta_t| + e$$

Sampling Probability

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

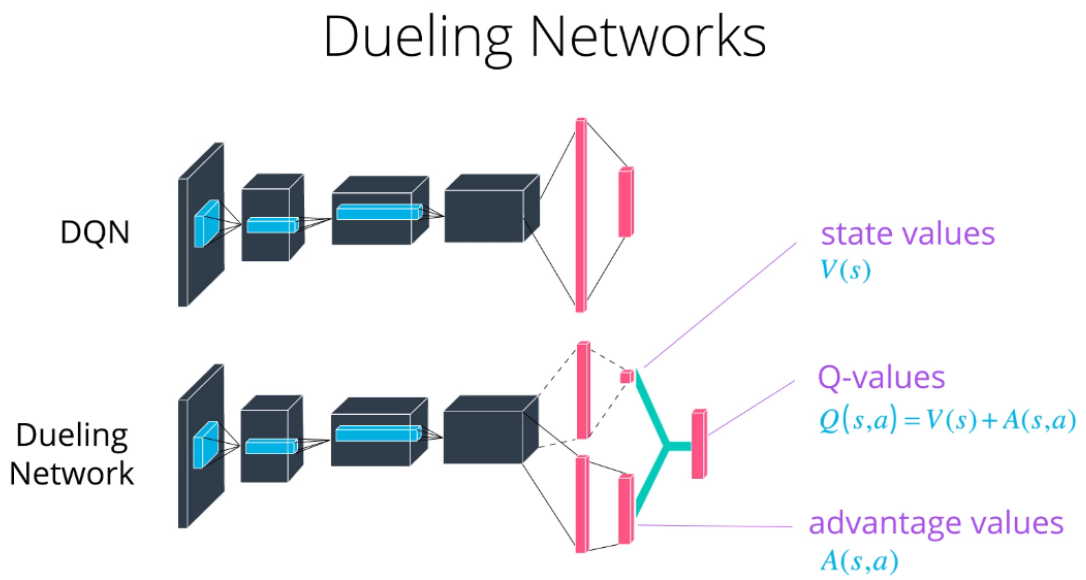
Modified Update Rule

$$\Delta \mathbf{w} = \alpha \underbrace{\left(\frac{1}{N} \cdot \frac{1}{P(i)} \right)^b}_{\text{importance-sampling weight}} \delta_i \nabla_{\mathbf{w}} \hat{q}(S_i, A_i, \mathbf{w})$$

- You can read more about prioritized experience replay by perusing this **research paper**.

Deuling DQN

- Typical DNN Network vs Dueling DQN



- Core idea of using dueling network is to use two streams that estimates 2 different thing:
  - state value function
  - Advantage for each action

- These streams may share some layers in the beginning such as convolutional layers, then branch off to their our fully connected layers.
- Finally, the desired Q values are obtained by combining the state and advantage values.
- The intuition behind this is that the value of most states don't vary a lot across actions, it makes sense to try and directly estimate them. But we still need to capture the difference actions make in each state.
- The advantage function comes in to help Q learning adapt to this architecture.
- You can read more about Dueling DQN by perusing this research paper.

More Extensions

- Many more extensions have been proposed, including:
  - Learning from multi-step bootstrap targets (as in A3C)
  - Distributional DQN
  - Noisy DQN

- Each of the six extensions address a ***different*** issue with the original DQN algorithm.
- Researchers at Google DeepMind recently tested the performance of an agent that incorporated all six of these modifications. The corresponding algorithm was termed **Rainbow.**
- It outperforms each of the individual modifications and achieves state-of-the-art performance on Atari 2600 games!

DQN
DDQN
Prioritized DDQN
Dueling DDQN
A3C
Distributional DQN
200%

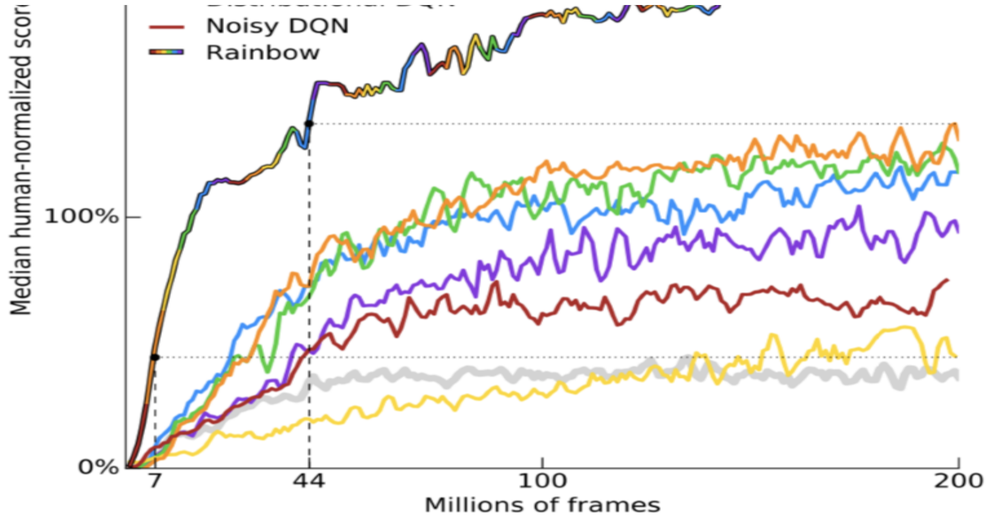


Figure 1: **Median human-normalized performance** across 57 Atari games. We compare our integrated agent (rainbow-colored) to DQN (grey) and six published baselines. Note that we match DQN's best performance after 7M frames, surpass any baseline within 44M frames, and reach substantially improved final performance. Curves are smoothed with a moving average over 5 points.