**Challenges:**

That’s way too many pixels, definitely more than we need.

can be demanding in terms of computation and memory requirements

**Preprocessing:**

1. let’s crop and down sample our images to 84x84 squares. We’ll use significantly less memory while keeping all necessary information.
2. convert the colors to the grayscale

Assuming that for each observation we are going to store 4 last frames, our input shape will be 84x84x4.

Note that every single frame will be a part of 4 consecutive observations, so instead of copying it every time, we can use a reference to a single point in memory. We’ll significantly (by 4 times!) reduce our memory usage.

We can simply use the algo mentioned in the below link:

<https://github.com/openai/baselines/blob/master/baselines/common/atari_wrappers.py>

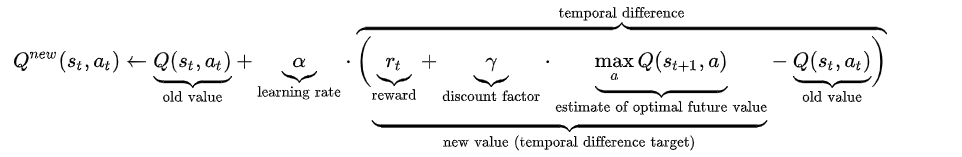
**Q-Function:**

"Q" refers to the function that the algorithm computes - the expected rewards for an action taken in a given state.

Q : S x A -> R

Explanation (Taken from Wikipedia):

Before learning begins, Q is initialized to a possibly arbitrary fixed value. Then, at each time t the agent selects an action a(t), observes a reward r(t), enters a new state s(t+1) (that may depend on both the previous state and the selected action), and Q is updated. The core of the algorithm is a Bellman equation as a simple value iteration update, using the weighted average of the old value and the new information:



**Model architecture:**

Instead of using the old architecture where a separate forward pass was needed to calculate the Q value for every single action, we will use an architecture known as double Q learning. We are going to decouple the action choice from the target Q-value generation. To do so we are going to have two separate networks. We will use our primary network to select an action and a target network to generate a Q-value for that action. To synchronize our networks, we are going to copy weights from the primary network to the target one every n training steps.

**Training details:**

A different network was trained on each game: the same network architecture, learning algorithm and hyperparameter settings were used across all games, showing that our approach is robust enough to work on a variety of games while incorporating only minimal prior knowledge. The purpose of changing the reward values to +1, -1 and 0 for positive reward, negative reward and no reward was to limits the scale of the error derivatives and makes it easier to use the same learning rate across multiple games. However, this could affect the performance of our agent since it cannot differentiate between rewards of different magnitude.

We use RMSprop algorithm with mini batches of size 32.

* **RMSprop:**

It is unpublished optimization algorithm designed for neural networks. There are two ways to introduce RMSprop. First, is to look at it as the adaptation of rprop algorithm for mini-batch learning. It was the initial motivation for developing this algorithm. Another way is to look at its similarities with Adagrad. Please check the below link for more details:

<https://towardsdatascience.com/understanding-rmsprop-faster-neural-network-learning-62e116fcf29a>

The behavior policy during training was e-greedy with e annealed linearly from1.0 to 0.1 over the first million frames and fixed at 0.1 thereafter. we also use a simple frame-skipping technique. More precisely, the agent sees and selects actions on every kth frame instead of every frame, and its last action is repeated on skipped frames. This technique allows the agent to play roughly k times more games without significantly increasing the runtime. We use k = 54 for all games.

**Evaluation Procedure:**

The trained agents were evaluated by playing each game 30 times for up to 5min each time with different initial random conditions and an e-greedy policy with e50.05. This procedure is adopted to minimize the possibility of overfitting during evaluation. The random agent served as a baseline comparison and chose a random action at 10Hz.

The professional human tester used the same emulator engine as the agents and played under controlled conditions. The human tester was not allowed to pause, save, or reload games. To make everything equal between the agent and human tester even audio was disabled.

**Algorithm**:

The action is passed to the emulator and modifies its internal state and the game score. In general, the environment may be stochastic. The emulator’s internal state is not observed by the agent; instead, the agent observes an image from the emulator, which is a vector of pixel values representing the current screen. In addition, it receives a reward r(t) representing the change in game score. Because the agent only observes the current screen, the task is partially observed. Therefore, sequences of actions and observations are input to the algorithm, which then learns game strategies depending upon these sequences. All sequences in the emulator are assumed to terminate in a finite number of timesteps. This formalism gives rise to a large but finite Markov decision process (MDP) in which each sequence is a distinct state. As a result, we can apply standard reinforcement learning methods for MDPs, simply by using the complete sequence s(t) as the state representation at time t.

We define the optimal action-value function as the maximum expected return achievable by following any policy, after seeing some sequence s and then taking some action a. The optimal action-value function obeys an important identity known as the Bellman equation. This is based on the following intuition: if the optimal value of the sequence at the next time-step was known for all possible actions then the optimal strategy is to select the action maximizing the expected value.

*The basic idea behind many reinforcement learning algorithms is to estimate the action-value function by using the Bellman equation as an iterative update. Such value iteration algorithms converge to the optimal action-value function as the number of iterations reaches to infinity.*

**why DQN might not converge?**

Regular DQN tends to overestimate Q-values of potential actions in a given state. It wouldn’t cause any problems if all the actions were equally overestimated, but the case is, that once one specific action becomes overestimated, it’s more likely to be chosen in the next iteration making it very hard for the agent to explore the environment uniformly and find the right policy.

The below link is really a great one to understand how to make a good agent to play Atari games:

<https://gsurma.medium.com/atari-reinforcement-learning-in-depth-part-1-ddqn-ceaa762a546f>