Introduction

This document provides a detailed high-level description of our implementation of Deep Q Learning for Starcraft II minigames. As a starting point, we will use the implementation of DQN in [1], which was used to play Atari 2600 games. However, since our agent has to play SC2, [2] will define the actions, environment and other details specific to SC2. The reason we are not attempting to reimplement the baseline agents in [2] exactly is because they use more advanced (newer) methods like Asynchronous Advantage Actor Critic (A3C) as described in [3]. (Should we try A3C? I'm not sure if it's easier or harder). In fact, we might diverge from certain implementation details in the papers as we deem appropriate. Any deviations will be noted and justified in this document. If time allows, we will augment our implementation with more advanced techniques.

About A3C

A3C is an asynchronous method that trains on a multi-core CPU rather than a GPU. On the Atari 2600 games, it trains much faster than than the original DQN and is more stable:

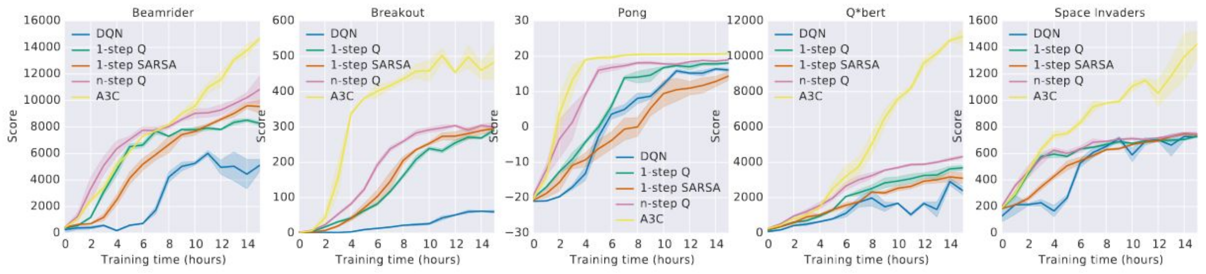


Figure 1. Learning speed comparison for DQN and the new asynchronous algorithms on five Atari 2600 games. DQN was trained on a single Nvidia K40 GPU while the asynchronous methods were trained using 16 CPU cores. The plots are averaged over 5 runs. In the case of DQN the runs were for different seeds with fixed hyperparameters. For asynchronous methods we average over the best 5 models from 50 experiments with learning rates sampled from $LogUniform(10^{-4}, 10^{-2})$ and all other hyperparameters fixed.

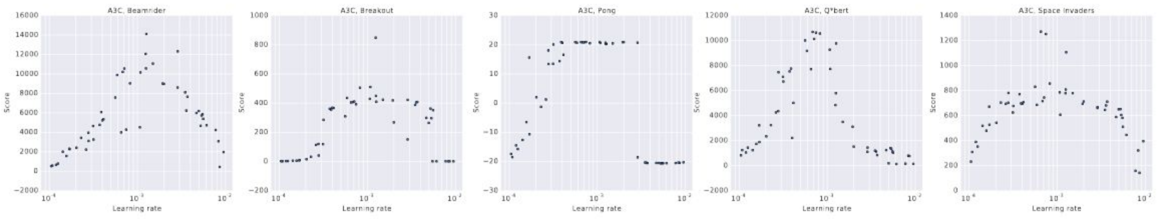
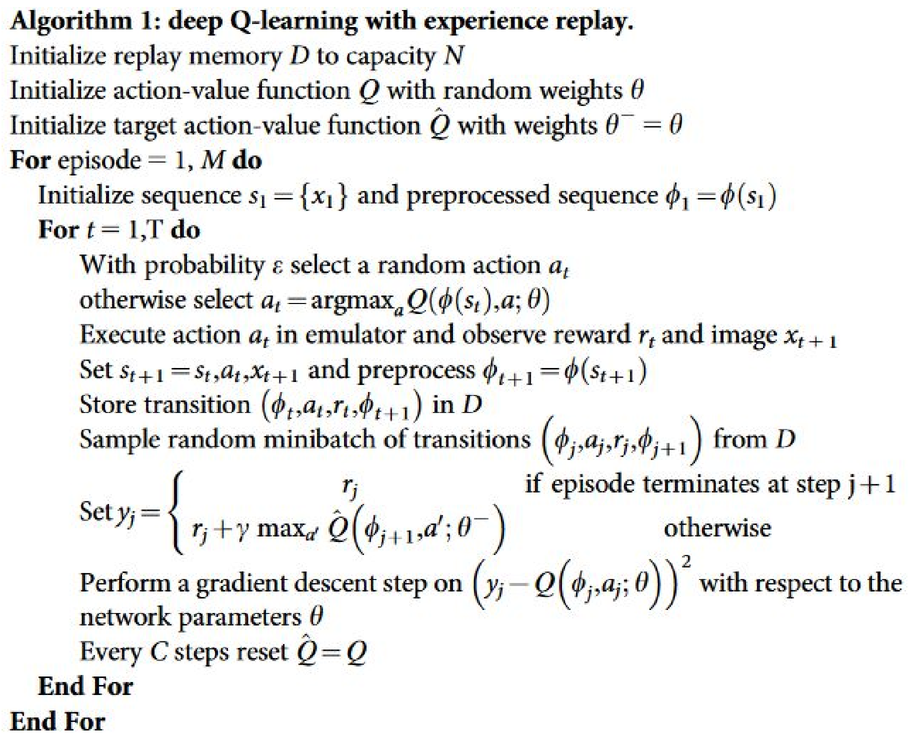


Figure 2. Scatter plots of scores obtained by asynchronous advantage actor-critic on five games (Beamrider, Breakout, Pong, Q*bert, Space Invaders) for 50 different learning rates and random initializations. On each game, there is a wide range of learning rates for which all random initializations acheive good scores. This shows that A3C is quite robust to learning rates and initial random weights.

To my understanding, the network architecture for the original DQN and A3C aren't too different except for the final layer outputs. The main differences are in the learning algorithm. A3C launches multiple instances of the network on different CPU cores which then learns asynchronously. Each instance interacts with the environment differently to stabilize the training so that experience replay is no longer completely necessary (it could still improve training performance).

Learning Algorithm

For baselines in [2], Deepmind uses Asynchronus Advantage Actor Critic (A3C) [3] to learn the parameters of the policy (A.K.A. the artificial neural network). We will not use A3C for our DQN, at least not for our first version. Instead we will use the original DQN learning algorithm in [1]:



```
**Algorithm 1: deep Q-learning with experience replay.**
Initialize replay memory $D$ to capacity $N$
Initialize action-value function $Q$ with random weights $\theta$
Initialize target action-value function $\hat{Q}$ with weights $\theta^- = \theta$
For episode = 1, $M$ do
    Initialize sequence $s_1 = \{x_1\}$ and preprocessed sequence $\phi_1 = \phi(s_1)$
    For $t = 1$,T do
        With probability $\varepsilon$ select a random action $a_t$
        otherwise select $a_t = \text{argmax}_a Q(\phi(s_t), a; \theta)$
        Execute action $a_t$ in emulator and observe reward $r_t$ and image $x_{t+1}$
        Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
        Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $D$
        Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $D$
        Set $y_j = \begin{cases} r_j & \text{if episode terminates at step j+1} \\ r_j + \gamma \max_{a'} \hat{Q}(\phi_{j+1}, a'; \theta^-) & \text{otherwise} \end{cases}$
        Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ with respect to the
        network parameters $\theta$
        Every $C$ steps reset $\hat{Q} = Q$
    End For
End For
```

Before going into the algorithm in greater depth, let’s explain three important concepts: (1) experience replay, (2) epsilon-greedy exploration and (3) separate target network.

<u>Experience Replay</u>

Training deep neural networks are difficult in general because training might not be stable (e.g. training only works converges and achieves good performance for certain weight initializations and learning rates). In short, experience replay makes the training process more stable for DQN. Each experience <s,a,r,s’> are stored in a replay memory. As the replay memory fills up (some amount of memory we allocate) we can discard the oldest experiences. During training, we use random sampling of the replay memory to train out network rather than the most recent state transition. This breaks the correlation between subsequent training samples that might psuh the network into a local minimum.

Epsilon-Greedy Exploration

During the beginning of training, the network performs exploration over the action space since the values of the network are randomly initialized. However, as the Q-function converges, the networks returns better and more consistent Q-values, and exploration decreases. This may lead to converging to a local minimum as the action sequences that lead to the global minimum are never explored. With e-greedy exploration, the network chooses a random action with probability e. This e can be decreased over time such that at the beginning the network focuses on exploration and later focuses on exploitation of known good state-actions.

Policy Representation (Defining Actions)

The policy representation will determine the implementation of the last layer in our DQN which equate to actions taken by our agent.

An action in SC2, such as selecting a unit and moving it, is actually a composite of simpler actions. To move a unit, the player has to select it, decide whether the action should be queued by holding shift, and then click on a point on the screen or minimap to execute it. [2]

Rather than defining three separate actions for the above task, PySC2 packages them together into an atomic compound action. In terms of code, an action is composed of a function identifier and a sequence of parameters. For example, consider selecting units by drawing a rectangle: select_rect(select_add, (x1, y1), (x2, y2)). “select_add” is a binary argument specifying whether the newly selected units should be added to currently selected units. The coordinates are integers that define the position of mouse click and release to draw the rectangle. [2]

Since an action is composed of the action identifier and its arguments, a joint distribution over the identifier and its arguments would require millions of values, which would not be good. Thus, Deepmind first simplifies by representing the policy in an auto-regressive manner by using the chain rule: [2]

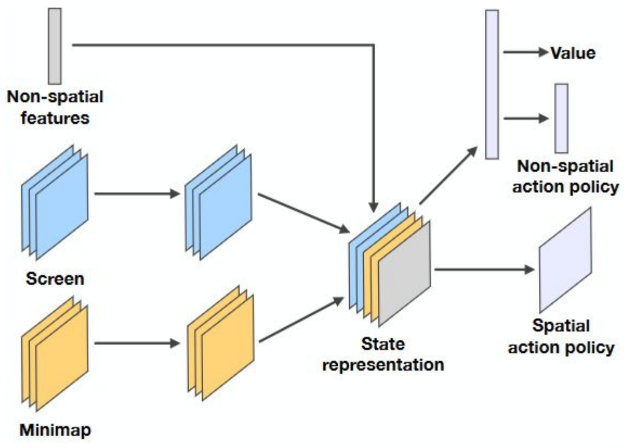
$$\pi_\theta(a|s) = \prod_{l=0}^{L} \pi_\theta(a^l | a^{<l}, s).$$

Then, they make a further simplification by using policies that choose the function identifier and its arguments independently: [2]

$$\pi_\theta(a|s) = \prod_{l=0}^{L} \pi_\theta(a^l|s)$$

See the section “4.2 Policy Representation” in [2] for more details.

To my understanding, the final layers of our neural network will be composed of output values for a (1) non-spatial action policy, and (2) spatial action policy:



(b) FullyConv

Policy Representation Summary

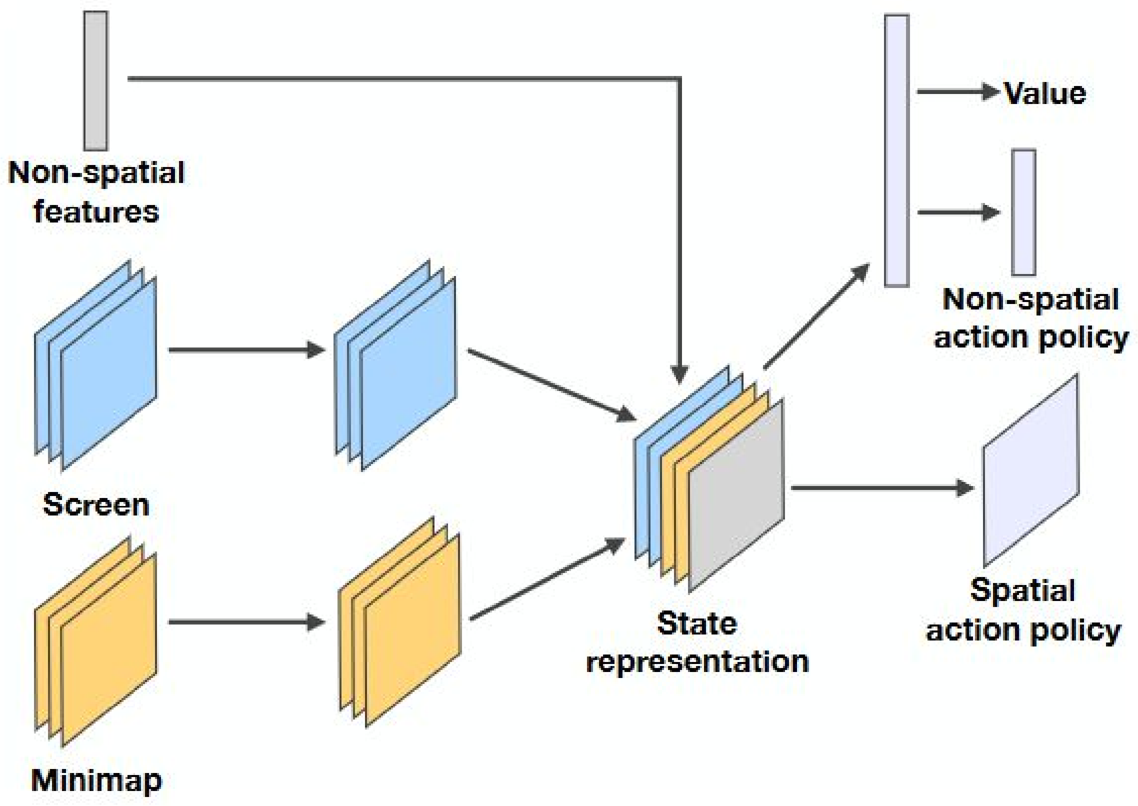
Each predicted value (between 0 and 1 for probability) in the non-spatial action policy represents a different function identifier or binary function argument. The spatial action policy represents coordinates used in certain function identifiers. All the more formal math jargon in the paper [2] just means to say that each predicted value in the layers is predicted independently by the network rather than, for example, predicting a joint probability for multiple function identifiers/arguments combinations.

However, I (Terry) am still a bit uncertain about how the policy selects the next best action to carry out. I suppose the policy first looks at the predicted values for all function identifiers in the non-spatial action policy, and selects the one with the highest value. Then the policy looks at the required function arguments. For binary arguments, the value is set to true if the predicted value > 0.5. For spatial coordinates, the spatial coordinate(s) with the highest value(s) are selected depending on the number of coordinates required (e.g. clicking to move on the screen requires one coordinate, but drawing a rectangle requires two coordinates). That being said, I am unsure if the order of selected spatial coordinates is important (e.g. drawing a rectangle: not important because a different order still draws the same rectangle; other actions: I’m not sure). This will require more investigation. If the order of coordinates are important, the policy also needs to decide the order.

Furthermore, the legal actions depend on the current state, so illegal actions need to be masked and the probability distribution over the legal actions need to be renormalized.

We also may want to restrict the action space ourselves.

Agent Architecture



(b) FullyConv

Complete Description of our DQN Implementation (Step by Step)

1. Code the DQN in Tensorflow (or another framework)

2. Implementing the training algorithm in PySC2

References

Main papers:
[1] Human-level control through deep reinforcement learning
[2] Starcraft II: A New Challenge for Reinforcement Learning

Other mentioned papers:
[3] Asynchronous Methods for Deep Reinforcement Learning

Blog posts:
[4] Demystifying Deep Reinforcement Learning