# Machine Learning Engineer Nanodegree

## Capstone Proposal

## Proposal

I would like to develope an A3C-based agent successfully as described in 1. I plan to evaluate the algorithm's performance for Breakout, an ATARI game where baseline methods are Double DQN2 with Prioritized Experience Replay3. It is true there's hardly anything novel, but for the sake of personal learning, I think this is a challenging, interesting problem.

### Domain Background

The field of Reinforcement Learning has recently seen breakthrough results by approximataing Q action-value functions with Deep Neural Networks (DQN). DQN has been shown to beat expert human players on a number of the ATARI games by training only with raw visual inputs of the games and without any domain knowledge specific to the games given[4, 7]. Despite DQN's promising results, it is known to be unstable in its vanilla form. There are three main causes to this. First, the samples an agent experieces are highly correlated (non-iidness). Second, a small update to the Q action value function can lead to a large change to the data the agent experiences (non-stationarity). Third, the action values (Q) and the target (reward + gamma * max Q) are correlated, just like a cat chasing its own tail (non-stationarity). To remedy the instability, Experience Replay(EP) and the trick of using a separate target were developed, and achieved a stabilizing effect.

However, EP has several drawbacks. First, it requires more memory and computation because the agent must keep track of the storage of experiences and in the case of Prioritized EP, the task of updating the priorities of the samples can be computationally heavy. Second, EP forces us to do off-policy learning where on-policy learning algorithms like SARSA, actor-critic methods is impossible.

Asynchronous Advantage Actor Critic (A3C) is an algorithm proposed to solve the issues mentioned above. A3C is deemed outperforming DQN, in terms of the simplicity of implementation, the training time, and performance. Unlike DQN with EP that uses a single agent interacting with the environment, A3C runs multiple agents where each agent interacts, in parallel (asynchronously), with a different instance of the environment. This scheme has the effect of reducing the correlations between experience samples, and making the learning more stationary because the variety of experiences will increase. Practically, the parallel scheme allows the training time to be faster than in DQN, and the training can be performed on a single computer with a standard multi-core CPU.

### Problem Statement

The problem to solve is building a reinforcement learning agent to play autonomously Breakout providing minimal game-specific knowledge to the agent: the input data are visual images and the number of actions available The goal of Breakout is that the agent must move a paddle so the ball bounces off it to break bricks. The more bricks it breaks at given time, the better. Concretely, I aim to build an agent taht learns to acquire over an arbitrary amount of 500 points.

### Datasets and Inputs

1. Environment
    - openAI's gym: https://gym.openai.com/envs/Breakout-v0
    - state: continuous. Game Memory (RAM).
    - action: discrete. ["LEFT", "RIGHT", "NOOP"]
    - observation: visual input: 210 x 160 x 3.
    - ALE environment also has a peculiar hidden feature called action repeat probability where every time an agent takes an action, the environment ignores the action with a 25% chance and simply repeat the previous action taken.
    - OpenAI environment seems to enforce the frame skipping their own way where K becomes a variable on {1, 2, 3} to add stochasticity. Once an agent takes an action, it may be repeated once, twice or three times. There is no

component-wise maximum value frame encoding. I conjecture OpenAI fixed the flickering issue when they created the environment.

2. Input & preprocessing:

    - raw input: pixel-based image: 210 (width) x 160 (height) x 3 (RGB, 128 color palette). We assume this much of resolution is not necessary to learn an optical policy and can cause a computation bottleneck in real-time play. Therefore, we preprocess the raw sensory input to a lower dimension.
    - frame encoding: 4 encoded a frame by taking component-wise maximum value over the current raw input frame and the previous on each color channel. For example, if the current frame has (R:34, G:50, B:102) and the previous (R:4, G:100, B:202), the encoded current frame is (R:34, G:100, B:202). This was necessary to remove flickering (pixels not showing when they should) that is present in some ATARI games.
    - downsampling: downsample the RGB frame to 84 x 84 image. In 4, linear scaling (bilinear interpolation) was done to downsample without keeping the original image aspect ratio. In 7, the image was downsampled to 110 x 84 with the same aspect ratio and an additional step of cropping to 84 x 84 was taken.
    - greyscale conversion: grey scale conversion was done through the built-in ALE (Arcade Learning Environment) grayscale conversion method.
    - frame skipping: to reduce the computational burden and addreses partial oberservability, he agent ouputs its decision every kth frame. The k in our experiment will be 4, the same as [4, 7]. During the k number of frames, the agent repeats the same action decided k times.
    - final input: K most recent consecutive frames will be stacked to form a final input of 4 x 84 x 84. The agent makes a decision by looking k steps backwards that likely include motion-related information. Note the conseutive final inputs are overlapping by (k-1) frames.

3. Output:

    - action ["LEFT", "RIGHT", "NOOP"]

## Solution Statement

The solution is an A3C algorithm. The A3C model will be trained on my local machine: Dell XPS 9560 with quad-core i7 CPU and NVIDIA GTX 1050 (4GB memory). The trained model will be evaluated on my machine whose result will be posted publicly on the OpenAI's scoreboard. 500 points is on par with some of the highest scores posted there.

Existing implementations to refernece:

- https://github.com/devsisters/DQN-tensorflow
- https://dbobrenko.github.io/2016/11/03/async-deeprl.html
- https://github.com/spragunr/deep_q_rl/

## Benchmark Model

1. DQN with Experience Replay This will be trained on my local machine. Since training DQN succesfully may take days, I will try to use Google Cloud or EC2 for training it. If possible, I may try to evaluate double DQN(D-DQN), D-DQN with prioritized experience replay, or dueling D-DQN.

2. CEM (Cross Entropy Method) This is an evolutionary algorithm that is considered a well-performing baseline. The implementation will share the implementations of 5 and 6.

## Evaluation Metrics

1. training time to achieve an appropriate score level (e.g. 100 points)
2. total average points per episode (moving average over 100 episodes) acquired by a sucessfully trained model.

## Project Design

I will follow the strategy of writing the final report from the beginning and keep adding details until the report is complete. Therefore the workflow will be simliar to howe one would go about reading the final report from one section to another.

1. related work I will read papers that feature relevant studies (6 referenced plus any more papers) to understand both the theoretical grounds and the technical details needed for implementation. The result
2. environment setup I will set up necessary dependencies to run an agent on an OpenAI gym environment. I will also acquire an account from OpenAI to be able to post results on their scoreboard.
3. system setup The system to implement will be A3C, DQN and CEM. The system will be written in Python 3.
4. training The training will be done on my local machine. If I conclude DQN is hard to train on my local machine, I will run it on Amazon EC2 or Google Cloud Compute Engine.
5. evaluation The evaluation will be done on my local machine where the evaluated metrics will be visualized in python-based plots.
6. reporting The end-result will be shown on Jupyter notebook. I will report findings and share areas of improvement for further work.