Capstone Project Proposal: Learn to play Atari through Deep Reinforcement Learning

---

Domain Background

In reinforcement learning (RL) an agent interacts with the environment over a number of discrete time steps. At each time step the agent experiences a state s of the environment and decides to perform an action a. The action to be performed in a particular state is defined by the policy $\pi(a|s)$, and the quality of the policy is expressed in terms of the return R_t through the action value function $Q^\pi(s,a) = \mathbb{E}[R_t|s_t = s, a]$. In Q-learning the learning task involves finding the optimal value function $Q^*(s,a)$, which returns the maximum action. Once $Q*(s,a)$ in known, the plan is to act greedly with respect to a, namely through $argmax_a Q^*(s,a)$. If the state space is very large a function approximator, such as a neural network (NN), is used to model the optimal policy $Q^*(s,a) \approx Q(s,a,\theta)$. This type of learning agent, which is called deep Q-network (DQN), led to the first algorithm that outperformed humans in playing Atari [**?**]. Nevertheless DQN can suffer from a problematic convergence and large training times. Convergence issued have been addressed through experience replay, and by adopting a separated target network to prevent training on correlated samples [**?**]. Still, this solution cannot guarantee good performances in all RL problems. More recently policy gradient (PG) methods have been revived as they have shown for some applications superior converge while reducing training time compared to DQN [**?**]. In this class of algorithms a NN is used to approximate the policy that maximises the expected reward $\pi^*(a|s) \approx \pi(s,a,\theta)$. Policy parameters learning is based on the gradient of the performance measure with respect to the policy parameters. PG methods however also do not work well universally and may require long time to find the correct hyperparameters. Perhaps the simplest advantage that policy parameterization may have over action-value is that the policy may be a simpler function to approximate. Problems vary in the complexity of their policies and action-value functions. For some, the action-value function is simpler and thus easier to approximate. For others, the policy is simpler. In the latter case a policy-based method will typically learn faster and yield a superior asymptotic policy [**?**].

Problem Statement

The aim of this project is to train a RL agent to play the Atari 2600 game Pong [**?**] through two different reinforcement learning approaches described in the previous Section. In Pong the agent and an opponent control a paddle and the game consists of bouncing a ball past the other player. At each time step the agent observes a pixel image and chooses between two actions, Up or Down, then he receives a reward based on whether or not the ball went past the opponent with the goal of maximising the return at the end of each game. In the original DeepMind implementation [**?**, **?**] the state consists of the last 4 84×84 frames observed by the agent which

results on $256^{84\times84\times4} \approx 1.4\times10^{6}7970$ different states. In such a large state space the state-value function $Q^{\pi}(s,a)$, or the policy $\pi(s,a)$, needs to be approximated by a NN.

Datasets and Inputs

In a reinforcement learning problem the agent is not trained on a labelled data set, as in supervised learning, but instead he is required to understand the quality of his actions based on the rewards that he receives within an episode of the environment. In this project OpenAI Gym is used to replicate Atari 2600 Pong environment [**?**]. Each input is an array of shape (210, 160,3), and the environment returns a reward of +1 if the ball went past the opponents, -1 if the ball is missed, or 0 otherwise.

Solution Statement

The agent shall learn how to play Pong, or in other words, how to maximise the score in a Pong game, through the DQN and PG reinforcement learning methods. A learning agent is one who is able to increase his score with the number of episodes played.

Benchmark Model

The benchmark model is defined by a random agent who select his action by sampling uniformly between up or down. The average score of the random agent will be evaluated on 1000 episodes. It is expected that the learning agent initially obtains a score similar to the random agent, but then improves with the number of episodes played.

Evaluation Metrics

Each Pong game finishes when one of the opponent reaches a score of 21. An episode is defined as a set of 10 Pong games so that the maximum score that can be obtained per episode is 210. The performance of the algorithms will be evaluated based on the evolution of the score at the end of each episode, as a function of an increasing number of played episodes which shall be at least 1000.

Project Design

Following a similar approach to [**?**] a RL algorithm is organised accordingly to 3 Python classes described in Listings **??** and **??**:

- the `Agent` who performs actions, observes the environment, and learns from the feedback of the environment;
- the agent's `Brain` which contains the definition of the NN architecture, training, prediction, and weight update procedures
- the `Environment` in which the agent learns, which returns the new state and the reward that follow an agent action in a loop that ends at the end of an episode

The NN architecture for both the DQN and the PG methods is the same as in [**?**] composed of 2 convolutional layers and a fully connected layer, and will be implemented using *Keras* and *Tensorflow* Python libraries. A pre-processing step of the images is included which involve cropping, downsampling and conversion to black and white.
The key difference between the two methods is the way the agent is training. In DQN NN weights are updated using the output of a target network, obtained from memory replay, into the Bellman equation. In PG learning a new loss function given by the product of the discounted reward and the likelihood need to be defined in `_build_train_fn`.

```
class Agent:
    def act():
        ''' perform action according to epsilon-greedy scheme '''

    def observe():
        ''' add state to memory and update epsilon '''

    def replay(self):
        ''' sample experience from memory and update weights '''

class Brain:
    def _createModel():
        ''' define NN architecture '''

    def train():
        ''' fit NN model '''

    def predict():
        ''' predict actions from state '''

    def updateTargetModel():
        ''' assign weights to the Target Network '''

class Environment():
    def run(self, agent):
        while True:
            # Agent performs action
            a = agent.act(s)
            # environment return new state and reward
            s_next, r, done, info = self.env.step(a)
            # store state in memory and update epsilon
            agent.observe((s, a, r, s_next))
            # sample batch of states from memory and performe weigth Q-learning
            agent.replay()
            if done:
```

```
            # end loop at the end of episode
              break
```

Listing 1: DQN algorithm framework

```
class Agent:
    def act():
        ''' sample action according to policy probability '''

    def observe():
        ''' add state, action, and reward to memory '''

    def learn(self):
        ''' learn policy through policy gradient update '''

class Brain:
    def _createModel():
        ''' define NN architecture '''

    def _build_train_fn(self):
        ''' Create a train function. This is needed to implement the PG loss function '''

    def discount_rewards(self, rewards):
        '''  Calculate discounted reward '''

    def train():
        ''' Call states and reward from memory,  compute discount reward, and fit CNN model '''

    def predict():
        ''' return actions probability  '''


class Environment():
    def run(self, agent):
        while True:
            # Agent performs action
            a = agent.act(s)
            # environment return new state and reward
            s_next, r, done, info = env.step(a)
            # add reward to episode score
            score += r
            # store state in memory
            agent.observe(s, a, r)
            if done:
                # at the end of episode perform policy gradient learning
                agent.train()
```

Listing 2: Policy Gradient algorithm framework

References

[1] V. Mnih et al. Playing Atari With Deep Reinforcement Learning. *NIPS Deep Learning Workshop*, 2013

[2] V. Mnih et al. Human-level control through deep reinforcement learning. *Nature* 518 (7540):529–533, 2013

[3] V. Mnih et al. Asynchronous Methods for Deep Reinforcement Learning. *International Conference on Machine Learning*, 2016

[4] R.S. Sutton and A.G. Barto. *Reinforcement Learning: An Introduction*. Second Edition. Complete Draft, 2017

[5] http://gym.openai.com/envs/Pong-v0/

[6] https://jaromiru.com/2016/10/03/lets-make-a-dqn-implementation/