REINFORCEMENT LEARNING

FUNDAMENTALS AND APPLICATIONS

Final Project Guidelines

HOW TO USE THIS DOCUMENT: This document is intended as a set of **loose** guidelines for anyone who would like a bit of guidance/structure for their final project. People are doing all sorts of different projects, and in the final assignment document there are three choices, so not everything will apply to everyone.

In this document I direct attention to various example papers to show you how people publishing in this field tend to approach their writeups. In other words: this document shows you the details RL paper authors tend to include and how they include them. You may find value in simply browsing the examples and taking bits and pieces as inspiration for how you structure your own project writeup.

The writeup for your final project might have the following sections:

Introduction / Literature Review, where you introduce the basics of reinforcement learning that you will be using throughout your project, describe and discuss papers used as reference in your work, and so on.

Methodology / Architecture, where you describe the specifics of your task, your state space, your reward function, (if applicable) your deep RL architecture chosen, **your experiment plan,** and so on.

Experimental Results, where you present and discuss the results for your experiment plan. Graphs and tables are encouraged - you should know how to make them and a bit about how to interpret them by now.

Conclusion, where you summarize what was done, what was found, and what was learned.

Introduction / Literature Review

In the Introduction, you will

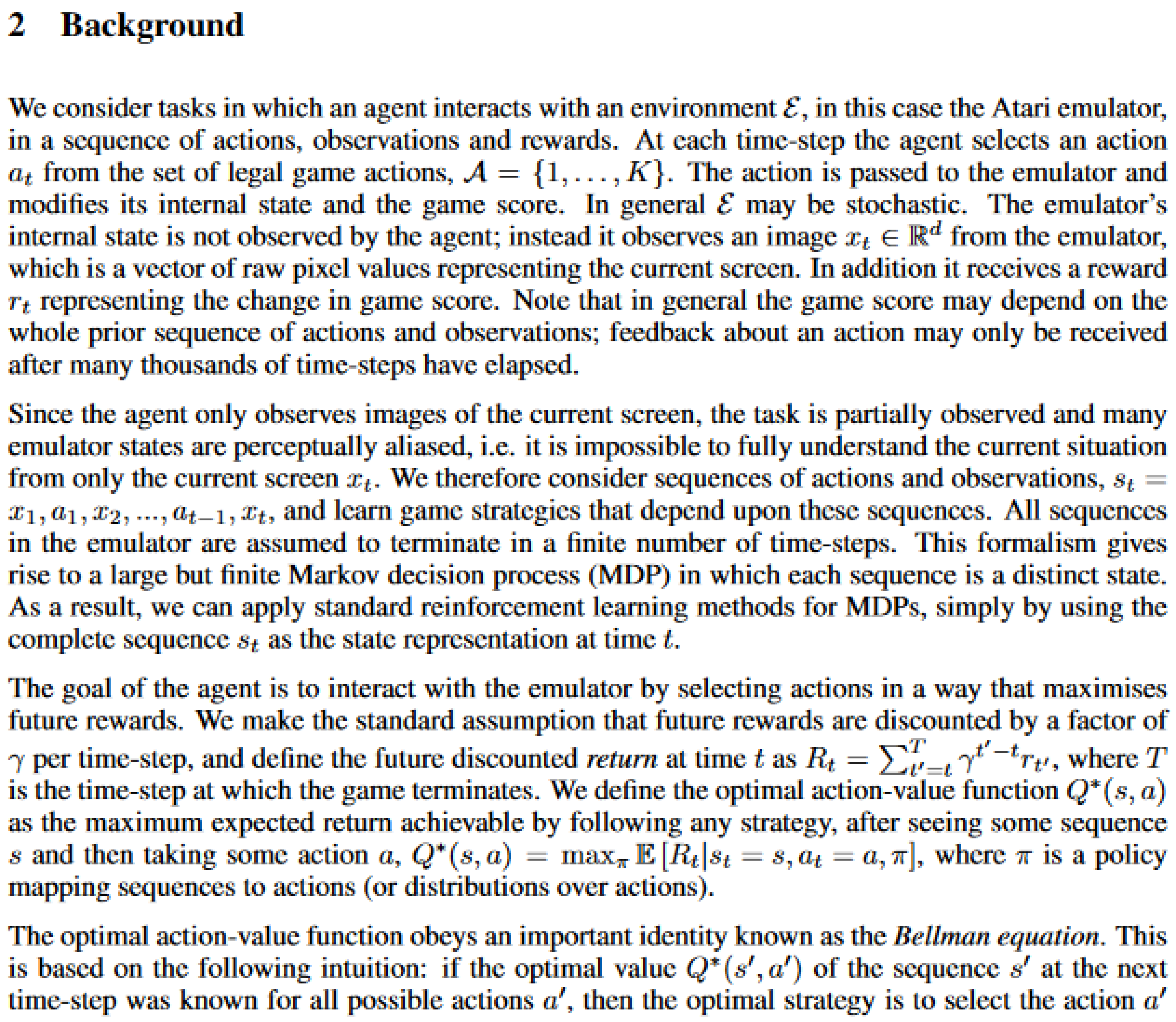
(a) go over the fundamental reinforcement learning concepts you will be using as the basis for your project,
(b) summarize/describe related papers that you read in relation to the work, and
(c) provide a high-level, conceptual description of the methodology (e.g., what neural network architecture, the tabular formulation, etc).

You should finish this section with a summary that succinctly tells the reader what you are doing and why you are doing it.

In reading RL papers, you may have noticed that they almost always have some summarization of reinforcement learning, as well as the particular method used (e.g., the equation for discounted future returns, Q-Learning, etc.). It is a very good idea to use these sections as inspiration for your own paper structure. below. It will be easy to gloss over this area and do too little, so do try to be complete. All you are doing is showcasing your mastery of the fundamental concepts we covered in the first half of the course.

Example: Playing Atari

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
https://arxiv.org/pdf/1312.5602.pdf



2 Background

We consider tasks in which an agent interacts with an environment $\mathcal{E}$, in this case the Atari emulator, in a sequence of actions, observations and rewards. At each time-step the agent selects an action a_t from the set of legal game actions, $\mathcal{A} = \{1, \ldots, K\}$. The action is passed to the emulator and modifies its internal state and the game score. In general $\mathcal{E}$ may be stochastic. The emulator's internal state is not observed by the agent; instead it observes an image $x_t \in \mathbb{R}^d$ from the emulator, which is a vector of raw pixel values representing the current screen. In addition it receives a reward r_t representing the change in game score. Note that in general the game score may depend on the whole prior sequence of actions and observations; feedback about an action may only be received after many thousands of time-steps have elapsed.

Since the agent only observes images of the current screen, the task is partially observed and many emulator states are perceptually aliased, i.e. it is impossible to fully understand the current situation from only the current screen x_t. We therefore consider sequences of actions and observations, $s_t = x_1, a_1, x_2, ..., a_{t-1}, x_t$, and learn game strategies that depend upon these sequences. All sequences in the emulator are assumed to terminate in a finite number of time-steps. 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.

The goal of the agent is to interact with the emulator by selecting actions in a way that maximises future rewards. We make the standard assumption that future rewards are discounted by a factor of γ per time-step, and define the future discounted *return* at time t as $R_t = \sum_{t'=t}^{T} \gamma^{t'-t} r_{t'}$, where T is the time-step at which the game terminates. We define the optimal action-value function $Q^*(s, a)$ as the maximum expected return achievable by following any strategy, after seeing some sequence s and then taking some action a, $Q^*(s, a) = \max_\pi \mathbb{E}\left[R_t | s_t = s, a_t = a, \pi\right]$, where π is a policy mapping sequences to actions (or distributions over actions).

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 $Q^*(s', a')$ of the sequence s' at the next time-step was known for all possible actions a', then the optimal strategy is to select the action a'

Continued next page…

maximising the expected value of $r + \gamma Q^*(s', a')$,

$$Q^*(s,a) = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q^*(s', a') \Big| s, a \right] \tag{1}$$

The basic idea behind many reinforcement learning algorithms is to estimate the action-value function, by using the Bellman equation as an iterative update, $Q_{i+1}(s,a) = \mathbb{E}\left[r + \gamma \max_{a'} Q_i(s', a') | s, a\right]$. Such *value iteration* algorithms converge to the optimal action-value function, $Q_i \rightarrow Q^*$ as $i \rightarrow \infty$ [23]. In practice, this basic approach is totally impractical, because the action-value function is estimated separately for each sequence, without any generalisation. Instead, it is common to use a function approximator to estimate the action-value function, $Q(s,a;\theta) \approx Q^*(s,a)$. In the reinforcement learning community this is typically a linear function approximator, but sometimes a non-linear function approximator is used instead, such as a neural network. We refer to a neural network function approximator with weights θ as a Q-network. A Q-network can be trained by minimising a sequence of loss functions $L_i(\theta_i)$ that changes at each iteration i,

$$L_i\left(\theta_i\right) = \mathbb{E}_{s,a \sim \rho(\cdot)} \left[\left(y_i - Q\left(s,a;\theta_i\right) \right)^2 \right], \tag{2}$$

where $y_i = \mathbb{E}_{s' \sim \mathcal{E}} \left[r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) | s, a\right]$ is the target for iteration i and $\rho(s,a)$ is a probability distribution over sequences s and actions a that we refer to as the *behaviour distribution*. The parameters from the previous iteration θ_{i-1} are held fixed when optimising the loss function $L_i\left(\theta_i\right)$. Note that the targets depend on the network weights; this is in contrast with the targets used for supervised learning, which are fixed before learning begins. Differentiating the loss function with respect to the weights we arrive at the following gradient,

$$\nabla_{\theta_i} L_i\left(\theta_i\right) = \mathbb{E}_{s,a \sim \rho(\cdot); s' \sim \mathcal{E}} \left[\left(r + \gamma \max_{a'} Q(s', a'; \theta_{i-1}) - Q(s,a;\theta_i) \right) \nabla_{\theta_i} Q(s,a;\theta_i) \right]. \tag{3}$$

Rather than computing the full expectations in the above gradient, it is often computationally expedient to optimise the loss function by stochastic gradient descent. If the weights are updated after every time-step, and the expectations are replaced by single samples from the behaviour distribution ρ and the emulator $\mathcal{E}$ respectively, then we arrive at the familiar *Q-learning* algorithm [26].

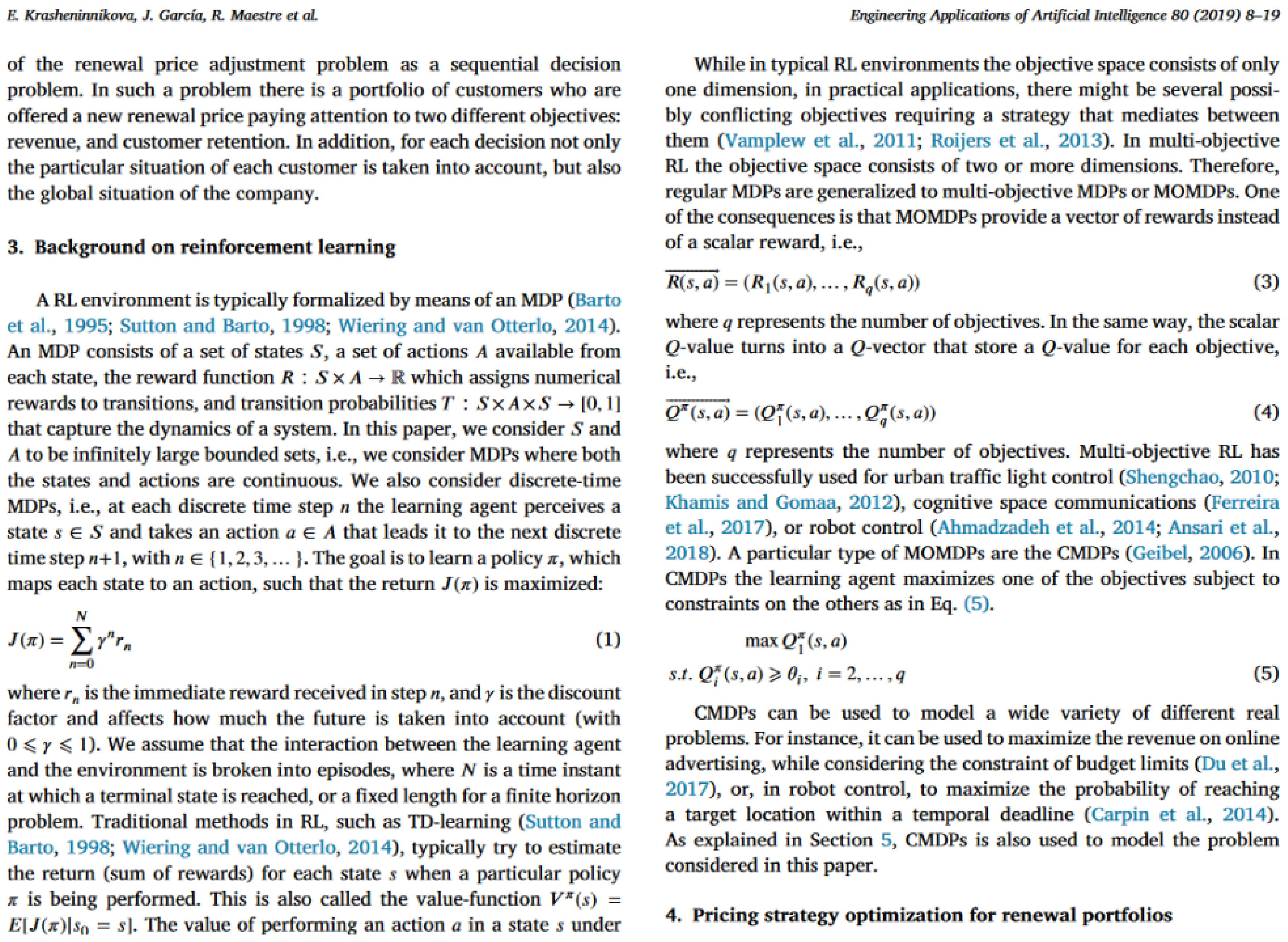
Note that this algorithm is *model-free*: it solves the reinforcement learning task directly using samples from the emulator $\mathcal{E}$, without explicitly constructing an estimate of $\mathcal{E}$. It is also *off-policy*: it learns about the greedy strategy $a = \max_a Q(s,a;\theta)$, while following a behaviour distribution that ensures adequate exploration of the state space. In practice, the behaviour distribution is often selected by an ϵ-greedy strategy that follows the greedy strategy with probability $1-\epsilon$ and selects a random action with probability ϵ.

3 Related Work

Perhaps the best-known success story of reinforcement learning is *TD-gammon*, a backgammon-playing program which learnt entirely by reinforcement learning and self-play, and achieved a super-human level of play [24]. TD-gammon used a model-free reinforcement learning algorithm similar

Example: Insurance Price Optimization and Tabular Q-Learning

https://www.sciencedirect.com/sdfe/reader/pii/S0952197619300107/pdf



of the renewal price adjustment problem as a sequential decision problem. In such a problem there is a portfolio of customers who are offered a new renewal price paying attention to two different objectives: revenue, and customer retention. In addition, for each decision not only the particular situation of each customer is taken into account, but also the global situation of the company.

3. Background on reinforcement learning

A RL environment is typically formalized by means of an MDP (Barto et al., 1995; Sutton and Barto, 1998; Wiering and van Otterlo, 2014). An MDP consists of a set of states S, a set of actions A available from each state, the reward function $R : S \times A \to \mathbb{R}$ which assigns numerical rewards to transitions, and transition probabilities $T : S \times A \times S \to [0, 1]$ that capture the dynamics of a system. In this paper, we consider S and A to be infinitely large bounded sets, i.e., we consider MDPs where both the states and actions are continuous. We also consider discrete-time MDPs, i.e., at each discrete time step n the learning agent perceives a state $s \in S$ and takes an action $a \in A$ that leads it to the next discrete time step $n+1$, with $n \in \{1, 2, 3, \dots\}$. The goal is to learn a policy π, which maps each state to an action, such that the return $J(\pi)$ is maximized:

$$J(\pi) = \sum_{n=0}^{N} \gamma^n r_n \qquad (1)$$

where r_n is the immediate reward received in step n, and γ is the discount factor and affects how much the future is taken into account (with $0 \leqslant \gamma \leqslant 1$). We assume that the interaction between the learning agent and the environment is broken into episodes, where N is a time instant at which a terminal state is reached, or a fixed length for a finite horizon problem. Traditional methods in RL, such as TD-learning (Sutton and Barto, 1998; Wiering and van Otterlo, 2014), typically try to estimate the return (sum of rewards) for each state s when a particular policy π is being performed. This is also called the value-function $V^\pi(s) = E[J(\pi)|s_0 = s]$. The value of performing an action a in a state s under

While in typical RL environments the objective space consists of only one dimension, in practical applications, there might be several possibly conflicting objectives requiring a strategy that mediates between them (Vamplew et al., 2011; Roijers et al., 2013). In multi-objective RL the objective space consists of two or more dimensions. Therefore, regular MDPs are generalized to multi-objective MDPs or MOMDPs. One of the consequences is that MOMDPs provide a vector of rewards instead of a scalar reward, i.e.,

$$\overrightarrow{R(s,a)} = (R_1(s,a), \dots, R_q(s,a)) \qquad (3)$$

where q represents the number of objectives. In the same way, the scalar Q-value turns into a Q-vector that store a Q-value for each objective, i.e.,

$$\overrightarrow{Q^\pi(s,a)} = (Q_1^\pi(s,a), \dots, Q_q^\pi(s,a)) \qquad (4)$$

where q represents the number of objectives. Multi-objective RL has been successfully used for urban traffic light control (Shengchao, 2010; Khamis and Gomaa, 2012), cognitive space communications (Ferreira et al., 2017), or robot control (Ahmadzadeh et al., 2014; Ansari et al., 2018). A particular type of MOMDPs are the CMDPs (Geibel, 2006). In CMDPs the learning agent maximizes one of the objectives subject to constraints on the others as in Eq. (5).

$$\begin{aligned} &\max Q_1^\pi(s,a) \\ s.t.\ & Q_i^\pi(s,a) \geqslant \theta_i,\ i = 2, \dots, q \end{aligned} \qquad (5)$$

CMDPs can be used to model a wide variety of different real problems. For instance, it can be used to maximize the revenue on online advertising, while considering the constraint of budget limits (Du et al., 2017), or, in robot control, to maximize the probability of reaching a target location within a temporal deadline (Carpin et al., 2014). As explained in Section 5, CMDPs is also used to model the problem considered in this paper.

4. Pricing strategy optimization for renewal portfolios

Metholodogy/Architecture

- Task description (what are the technical attributes of your task? How do you want the agent to behave within the environment?)
- Action representation(s) and rationale.
    - Discrete? Continuous? Both?
- State representation(s) and rationale.
- Reward function(s) and rationale (how do the reward functions help your agent learn the desired beahviours?)
- Value estimate procedure (e.g., your tabular framing of the problem and why you chose tabular over model-based, your deep RL architecture + why you chose it + how it works, etc.)

Here are some examples

Example: Deep Reinforcement Learning for 2048

Dedieu, A., & Amar, J. (2017). Deep reinforcement learning for 2048. In *Conference on Neural Information Processing Systems (NIPS), 31st, Long Beach, CA, USA*.

https://www.mit.edu/~amarj/files/2048.pdf

2 Presentation and Visualization of 2048

In this section, we present the game 2048 with the visualization tool we have built to play it. We define our representation of the space and actions spaces useful for later Reinforcement Learning.

2.1 Description

2048 is a single-player sliding block puzzle game developed by Gabriele Cirulli in 2014. The game represents a 4×4 grid where the value of each cell is a power of 2. An action can be any of the 4 movements: *up, down, left right*. When an action is performed, all cells move in the chosen direction. Any two adjacent cells with the same value (power of 2) along this direction merge to form one single cell with value equal to the sum of the two cells (i.e. the next power of 2). The objective of the game is to combine cells until reaching 2048. As an example let us illustrate a move in the game in figure 1.

Consider the game position in Figure 1a. A reasonable move to make is to shift ***right*** so that the two 32 cells will merge to form a 64. Note that the two 2 also merges. Once the cells have merged, a 2 appears with a uniform distribution over the vacant cells (in this case, in the top-left cell). Notice that moving ***left*** would have merged the same cells. However in practice it is better to **always maintain the largest value cell in a corner**. Learning this policy may require several experiments for a human. The question remains: can a neural network learn it?

2.2 State Representation

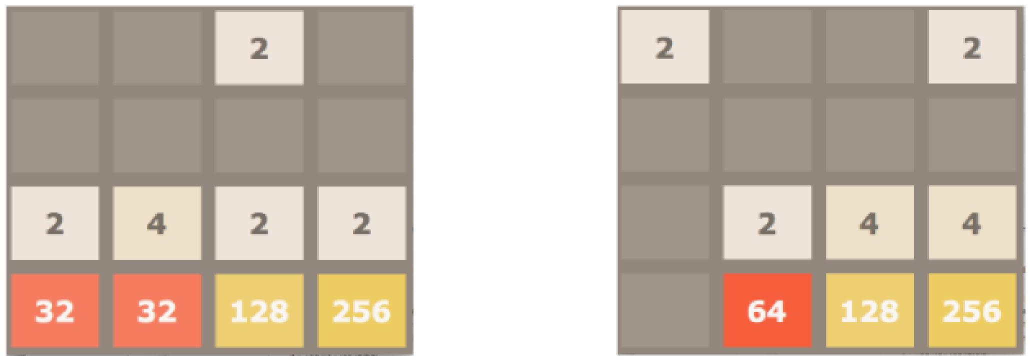
We represent our state space using a binary vector. The value of each cell can be any power of 2 in the range 1 (representing the empty cell) to 2^{15}. We therefore chose to represent each cell by a vector of $\{0, 1\}^{16}$, for a total state space size included in $\{0, 1\}^{256}$.

Note that an alternative approach could consist in representing the state space by an array of size 16 with each entry being an integer in $[0, 15]$. This model presents the advantage of introducing a metric distance between the cell values After considering both alternatives but selected the first model since the binary representation is more commonly used with neural networks in the literature

Figure 1: Illustration of the dynamics of the game. The grid is shifted ***right***, and a 2 randomly appears on the grid

(a) Before the move

(b) After the move



Further, the action space will be represented as an array of size 4. Note that a difficulty specific to 2048 and occasionally for other Atari games is that in many situations, some moves are prohibited by the configuration of the grid. We will suggest later how to deal with these issues.

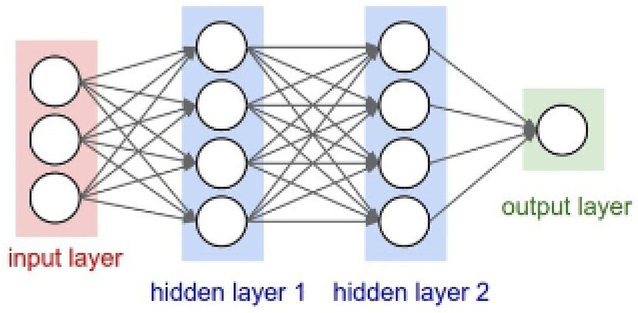
3 Using the Policy Network with Reinforcement Learning

In this section, we present the our Policy Network controlling the actions in 2048. We explain the game playing with **front-propagation** algorithm and the learning process by **back-propagation**. We develop 2 methodologies encouraging exploration: an ϵ-greedy and a probabilistic learning.

3.1 Front propagation with Policy Network

We aim at searching for the optimal policy π^* which maps the state space to the decision space in order to maximize the expected reward. We implement a policy network with 3 layers. The two hidden layers are chosen of size 200 and 100. Hence our Neural Network is encoded over 3 matrices $\mathbf{W_1}, \mathbf{W_2}, \mathbf{W_3}$. The output of the current state is a vector of size 4, such that each coordinate is associated with one of the 4 possible moves: *up, down, left, right*. The neural architecture can be represented in figure 2.

Figure 2: Our policy network architecture



Following the two hidden layers, we use the ReLu $h : x \to (x)^+$ activation function which guarantees non vanishing gradients. The final layer computes a **softmax** transformation so that the output values are in $[0, 1]$ and sum to 1. Hence the **front-propagation** algorithm is:

Input: A state $\mathbf{x} \in \{0, 1\}^{256}$
Output: A vector $\mathbf{p} \in [0, 1]^4$ such that $\mathbf{1}^T\mathbf{p} = 1$

1. Define the first hidden layer $\mathbf{h_1} = \max(\mathbf{W_1}^T\mathbf{x}, 0)$
2. Define the second hidden layer $\mathbf{h_2} = \max(\mathbf{W_2}^T\mathbf{x}, 0)$
3. Define $\mathbf{p} = S(\mathbf{W_3}^T\mathbf{x})$ with $S(\mathbf{h})_i := \frac{\exp(\mathbf{h}_i)}{\sum_k \exp(\mathbf{h}_k)} \in [0, 1]$

At every iteration we store all states, actions made, outputs and hidden vectors in matrices to update the weights using **back-propagation** algorithm For a vector $\mathbf{v}$ we note $\mathbf{V}$ the matrix with all sequence of vectors during a game (or several games). We then use the output $\mathbf{p}$ to sample an action and update the game state We present two alternative approaches in the remainder of the section which will guide our action sampling.

Unfeasible moves: The chosen action may not be feasible given the configuration of the game. We then eliminate such unfeasible moves before selecting the right one. We do not store the associated probability: if we were doing so, the algorithm would be unable to know that the move is unfeasible and it would encourage or discourage an action not chosen. Thus we only keep track of the feasible moves and rescale the probability for the future learning.

3.1.1 ϵ-greedy learning

In the first model, we use the coordinates of $\mathbf{p}$ to infer a ranking on the 4 different decisions. The highest value is associated with the best decision to make, i.e. we use this information to make the optimal decision $u^* = \arg\max_u \mathbf{p}_u$. We introduce some randomness with by sampling a random decision with probability ϵ in order to provide an explorative learning. The intuition is that most often, the policy follows its current belief of the optimal decision making, but it alternatively chooses to explore a new decision with probability ϵ. ϵ can be a fixed parameter or a function of the number of game plays decreasing to zero as in Silver et al. [2016]. A decreasing epsilon would allow better convergence properties.

3.1.2 A probabilistic approach

In this second model, we consider the coefficients of $\mathbf{p}$ as the probabilities that a move may be the optimal decision. Hence we should sample a decision u with probability $\mathbf{p}_u$. This setup is inherently explorative in situations of uncertainty. Therefore we would make optimal decisions with high probability unless the best decision is unclear.

3.2 Back-propagation

We know how to use our policy network to play a game. We now focus on the learning phase. Since our two prior perspectives use $\mathbf{p}$ in two different ways in the decision making process, the update rules of the Neural Net must be adapted to each situation.

3.2.1 Defining the reward

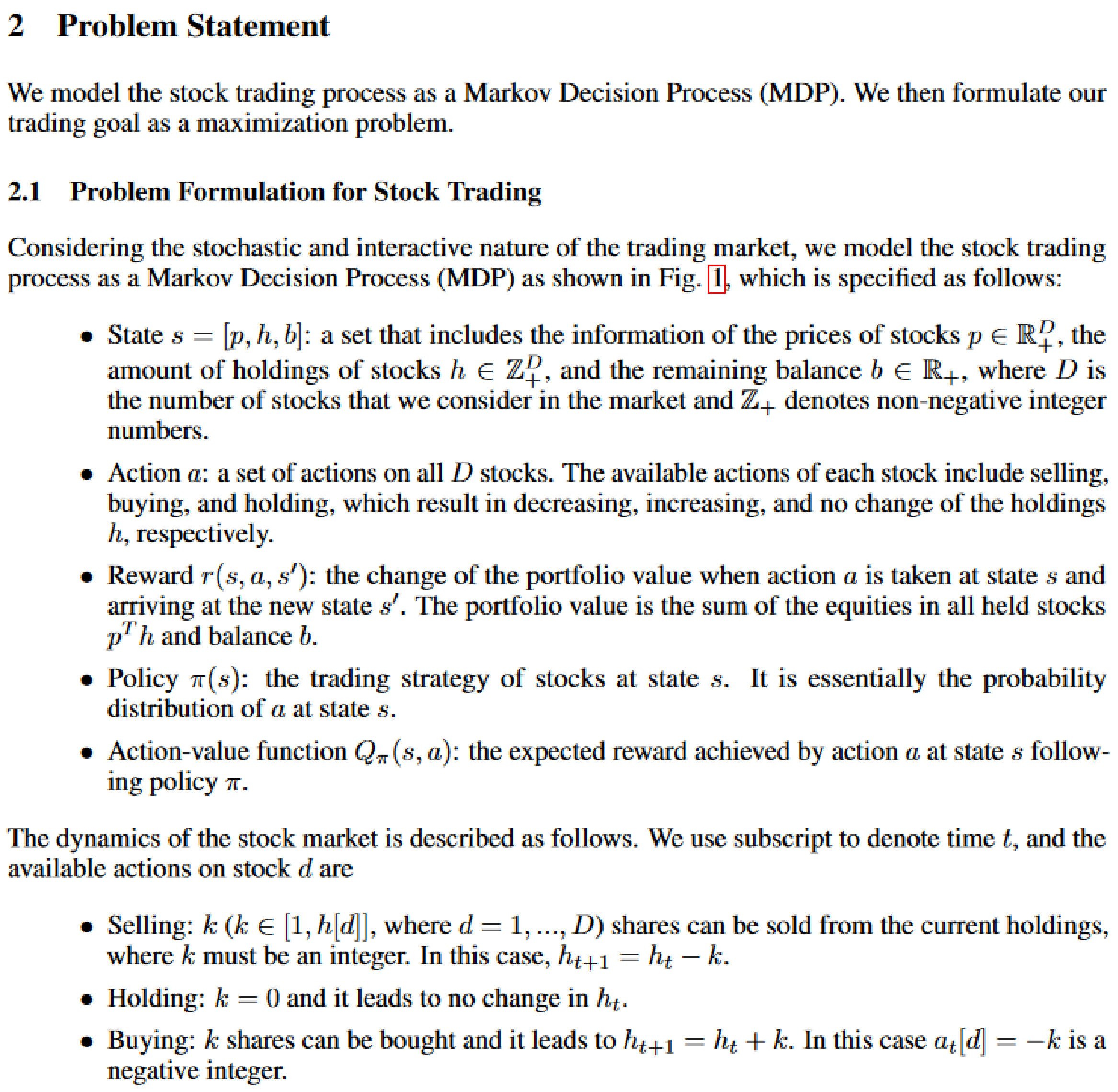
The purpose of defining a **reward** to separate the actions leading to better games played from the others. In a win/lose game such as most of the Atari Mnih et al. [2013], the reward is simply defined as 1 if they win and 0 otherwise. For 2048 there is no such clear situation. We could define an time-homogeneous reward for all moves in a game, corresponding to the highest value on the grid at the end. However this is a very discrete reward which gives too little information regarding the episode played. In practice we prefer to keep the sum of the cells of the grid at the end of the game, to encourage maximizing the length of an episode.

In our learning process, we want to compare games played, to reproduce the better ones. Hence we simulate a given number N_{batch} of episodes using the same Policy Network. At the end of each batch, we mean-center and standardize the rewards. The better games played are therefore the ones associated with positive reward and conversely At a higher level, our normalization approach proceeds in a very natural manner in the sense that it is continuously improving itself, rather than aiming for a specific goal. Given the near impossibility of reaching 2048 from a random initialization, the incremental improvements should perform better.

We update the weights in the neural net according to the **back-propagation** algorithm, which is essentially a gradient ascent methodology. The algorithm has two different implementations to

Example: Stock Market MDPs

Xiong, Z., Liu, X. Y., Zhong, S., Yang, H., & Walid, A. (2018). Practical deep reinforcement learning approach for stock trading. *arXiv preprint arXiv:1811.07522*.
https://arxiv.org/pdf/1811.07522.pdf?ref=https://githubhelp.com



2 Problem Statement

We model the stock trading process as a Markov Decision Process (MDP). We then formulate our trading goal as a maximization problem.

2.1 Problem Formulation for Stock Trading

Considering the stochastic and interactive nature of the trading market, we model the stock trading process as a Markov Decision Process (MDP) as shown in Fig. 1, which is specified as follows:

- State $s = [p, h, b]$: a set that includes the information of the prices of stocks $p \in \mathbb{R}_+^D$, the amount of holdings of stocks $h \in \mathbb{Z}_+^D$, and the remaining balance $b \in \mathbb{R}_+$, where D is the number of stocks that we consider in the market and $\mathbb{Z}_+$ denotes non-negative integer numbers.
- Action a: a set of actions on all D stocks. The available actions of each stock include selling, buying, and holding, which result in decreasing, increasing, and no change of the holdings h, respectively.
- Reward $r(s, a, s')$: the change of the portfolio value when action a is taken at state s and arriving at the new state s'. The portfolio value is the sum of the equities in all held stocks $p^T h$ and balance b.
- Policy $\pi(s)$: the trading strategy of stocks at state s. It is essentially the probability distribution of a at state s.
- Action-value function $Q_\pi(s, a)$: the expected reward achieved by action a at state s following policy π.

The dynamics of the stock market is described as follows. We use subscript to denote time t, and the available actions on stock d are

- Selling: k ($k \in [1, h[d]]$, where $d = 1, ..., D$) shares can be sold from the current holdings, where k must be an integer. In this case, $h_{t+1} = h_t - k$.
- Holding: $k = 0$ and it leads to no change in h_t.
- Buying: k shares can be bought and it leads to $h_{t+1} = h_t + k$. In this case $a_t[d] = -k$ is a negative integer.

3 A Deep Reinforcement Learning Approach

We employ a DDPG algorithm to maximize the investment return. DDPG is an improved version of Deterministic Policy Gradient (DPG) algorithm [12]. DPG combines the frameworks of both Q-learning [13] and policy gradient [14]. Compared with DPG, DDPG uses neural networks as function approximator. The DDPG algorithm in this section is specified for the MDP model of the stock trading market.

The Q-learning is essentially a method to learn the environment. Instead of using the expectation of $Q(s_{t+1}, a_{t+1})$ to update $Q(s_t, a_t)$, Q-learning uses greedy action a_{t+1} that maximizes $Q(s_{t+1}, a_{t+1})$ for state s_{s+1}, i.e.,

$$Q_\pi(s_t, a_t) = \mathbb{E}_{s_{t+1}}[r(s_t, a_t, s_{t+1}) + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1})]. \quad (2)$$

With Deep Q-network (DQN), which adopts neural networks to perform function approximation, the states are encoded in value function. The DQN approach, however, is intractable for this problem due to the large size of the action spaces. Since the feasible trading actions for each stock is in a discrete set, and considering the number of total stocks, the sizes of action spaces grow exponentially, leading to the "curse of dimensionality" [15]. Hence, the DDPG algorithm is proposed to deterministically map states to actions to address this issue.

As shown in Fig. 2, DDPG maintains an actor network and a critic network. The actor network $\mu(s|\theta^\mu)$ maps states to actions where θ^μ is the set of actor network parameters, and the critic network $Q(s, a|\theta^Q)$ outputs the value of action under that state, where θ^Q is the set of critic network parameters. To explore better actions, a noise is added to the output of the actor network, which is sampled from a random process $\mathcal{N}$.

Similar to DQN, DDPG uses an experience replay buffer R to store transitions and update the model, and can effectively reduce the correlation between experience samples. Target actor network

Li, Y., Ni, P., & Chang, V. (2020). Application of deep reinforcement learning in stock trading strategies and stock forecasting. *Computing*, *102*(6), 1305-1322.
https://research.tees.ac.uk/ws/files/9947462/VC_comp_deep_reinforcement_learning_accepted.pdf

3 ARCHITECTURE OF OUR DEPLOYMENT

This section describes the architecture of our deployment, since it can help us to get analysis quicker, better and with higher accuracy. As shown in Figure 1. When the raw data comes in, the data analysis phase is performed first to ensure that it is not extreme data. At the same time, the original data is passed into the data processing phase. Data will then be the input for the Deep Q Network part. DQN is a kind of network that uses a neural network to predict Q value and continuously updates the neural network to learn the max Q value. There are two neural networks (NN) in DQN: one is Target-Network with relatively fixed parameters, which is used to obtain the target value; the other is called Current Q-Network, which is used to evaluate the Current Q value. The training data is extracted randomly from Replay Memory, which records the actions (a), rewards (r), and results of the next state $(s, a, r, s^{'})$. As the Environment changes, Networks will update its parameters regularly and Replay Memory will change accordingly. The Loss function is the result of subtracting the value of Q in Target-Network from the value of Q in Current q-network. The values between modules are changed iteratively until the optimal value of Q is achieved and output operation is carried out.

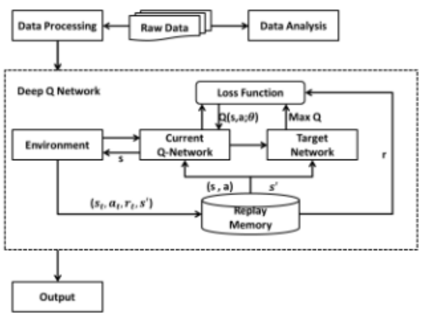


Fig. 1 The architecture of our deployment for experiments and data analysis

4 DESCRIPTION OF THE EXPERIMENT

To verify the feasibility of Deep Reinforcement Learning in stock market investment decisions, ten stocks in the 'Historical daily prices and volumes of all US stocks' dataset were selected by randomization for experiments. The dataset is available from kaggle, which provides the full historical daily price and volume data for all US-based stocks and ETFs (Exchange-Trade Funds) trading on the NYSE (New York Stock Exchange), NASDAQ (National Association of Securities Dealers Automated Quotation), and NYSE MKT (New York Stock Exchange Market). The three classic models in DRL, Deep Q-Network (DQN), Double Deep Q-Network (DDQN), and Dueling Double Deep Q-Network (Dueling DDQN) were selected for computational analysis and then performance comparison. Each stock was split into a training set and testing set and then fed into three Deep Reinforcement Learning models. The effects of the models were visually examined by simulating trading, and the benefits of the three models were compared horizontally. The training results and test results could then be compared and analyzed simultaneously.

4.2.1 Deep Q Network

Deep Q Network is one of the most classical and excellent algorithms in DRL. Before DQN appear in the deep reinforcement learning research found that using the nonlinear mapping deep network to represent the value function is not stable or even no convergence, and deep network training samples requirements are independent of each other but before and after the intensive study of a correlation between state data, these problems make directly from the high-dimensional data learning control strategy can be difficult. However, deep Q network introducing experience replay technology, target network and other methods, to overcome the above problems, use DQN build the Agent on the Atari 2600 game[28] directly before four frames of the graphics for the input, output control instruction for end-to-end training. In the test, DQN shown that it can be comparable to those of human players, and even outperforming experienced human experts in less difficult, non-strategic games.

The algorithm blends the Q-learning algorithm and neural network benefits. This is achieved by 1) increasing the experience reply function, from the previous state transition (experience) in the random sample training and 2) overcoming the correlated data and non - stationary distribution problems. In DQN, the Q value represents the current learned experience. The key of DQN model is to learn the q-value function, and finally be able to converge and accurately predict the Q value of each action in various states. The Q value calculated according to the formula is a score obtained by the agent through interaction with the environment and its own experience (namely, the target Q value). Finally, update the old Q value $Q^{'}(s_t, a_t)$ with the target Q value $r_{t+1} + \gamma max_{a'} Q(s_t^{'}, a_t^{'}; \theta)$. The corresponding relationship between the target Q value and the old Q value is exactly the corresponding the correlation between the result value and the output value in the supervised learning neural network. The experience pool can save the transfer samples (s_t, a_t, r_t, s_{t+1}) fetched and stored by each time step agent and the current environment into a memory unit, some are randomly fetched for training at the time needed. The loss function of DQN is presented below:

$$L(\theta) = E[(TargetQ - Q(s_t, a_t; \theta))^2]$$

(a_t represents the action situation of the Agent, s_t refer to the current state of this Agent, r_t is a real number that means the reward of selected action, θ indexes the mean square error of the network parameter, and $Q^{'}$, $s_t^{'}$, $a_t^{'}$ signals the renovated value of Q, s_t and a_t.)

Under certain conditions, Q learning algorithm only needs to use greedy strategy to ensure convergence, so Q learning is a more commonly used model independent reinforcement learning algorithm.

4.2.2 Double DQN

In Q learning and deep Q learning, optimal Q value will be used to select and measure an action. Choosing an overestimated value will lead to overestimation of the real value of Q. Van Hasselt et al. [30] found and proved that the traditional DQN method had the problem of overestimating the Q value, and the error would accumulate with the increase of the number of iterations. The proposal of Double Deep Q learning is to solve the problem caused by overestimation.

Deep Q learning can be regarded as a new neural network plus an old neural network. They have the same structure, but their internal parameters are updated with time difference. Since the optimal Q value predicted by the neural network is inherently wrong, and the error will become larger and larger with the iteration, Double DQN introduces another neural network to optimize the influence of error. Target network and main network can effectively reduce the number of participants. The specific operation is to modify the generating of the Target Q value is:

$$TargetDQ = r_{t+1} + \gamma Q(s_t, argmax_{a'} Q(s_t^{'}, a_t^{'}; \theta); \theta^{'})$$

4.2.3 Dueling DQN

In many DRL tasks, the value functions of the state action pairs are different under the influence of different actions. However, in some states, the size of the value function is independent of the action. Based on this situation, Wang et al.[34] proposed Dueling DQN, and add it into the DQN network pattern.

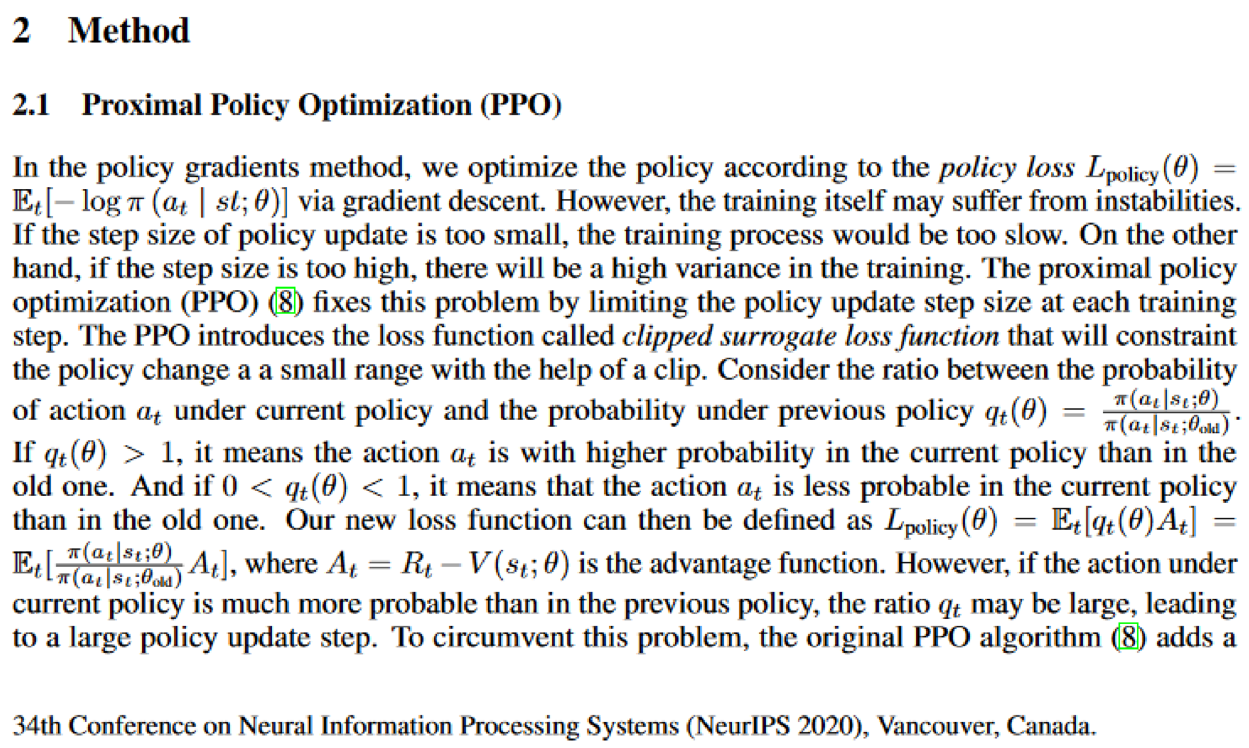
Dueling DQN combines Dueling Network with DQN.Dueling net assigns its eigenvalues extracted from the convolutional network layer to its two branches. The first part is the state value function $V(s_t; \theta, \beta)$, which stands for the value of the current state environment itself; the second part is the action advantage function $A(s_t, a_t; \theta, \alpha)$ of the dependent state, which refer to the extra value of an Action (A). Finally, the final Q value $Q(s_t, a_t; \theta, \alpha, \beta)$ can be obtained by re-aggregating the two paths, $V(s_t; \theta, \beta)$ and $A(s_t, a_t; \theta, \alpha)$ together. In the above function, a_t stands for an action of the Agent, θ is the convolutional layer parameter, s_t represents a state of the Agent, and α and β are the two-way fully connected layer parameters.

In real cases, the action dominant flow is commonly set as the individual action advantage function $Q(s_t, a_t; \theta, \alpha, \beta)$ minus the average of all action advantage functions $\frac{1}{|A|} \sum_{a'} A(s_t, a_t^{'}; \theta, \alpha)$ in a certain state.

The advantage of this method is that, when there is no sample to a, a can also be updated, data can be used more efficiently and training can be accelerated. In this way, the relative order of the main functions of each action in this state can be guaranteed to remain unchanged, and the range of Q value can be reduced to reduce the redundancy, so as to improve the overall stability of the algorithm.

Example: PPO for CartPole and MiniGrid

https://yipeichan.github.io/AdvML_deepRL.pdf

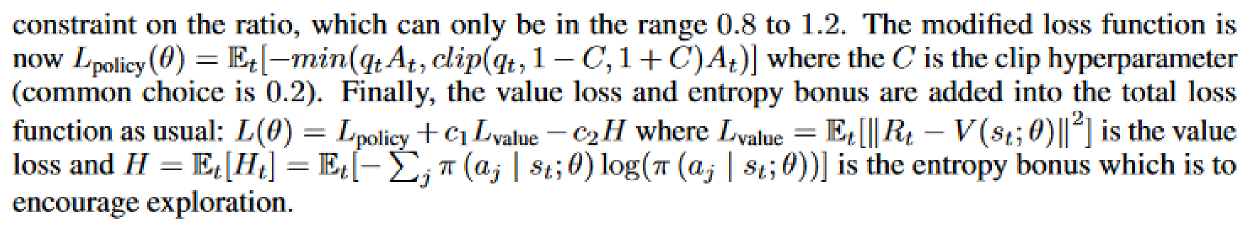


2 Method

2.1 Proximal Policy Optimization (PPO)

In the policy gradients method, we optimize the policy according to the *policy loss* $L_{\text{policy}}(\theta) = \mathbb{E}_t[-\log \pi\left(a_t \mid st;\theta\right)]$ via gradient descent. However, the training itself may suffer from instabilities. If the step size of policy update is too small, the training process would be too slow. On the other hand, if the step size is too high, there will be a high variance in the training. The proximal policy optimization (PPO) [8] fixes this problem by limiting the policy update step size at each training step. The PPO introduces the loss function called *clipped surrogate loss function* that will constraint the policy change a a small range with the help of a clip. Consider the ratio between the probability of action a_t under current policy and the probability under previous policy $q_t(\theta) = \frac{\pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta_{\text{old}})}$. If $q_t(\theta) > 1$, it means the action a_t is with higher probability in the current policy than in the old one. And if $0 < q_t(\theta) < 1$, it means that the action a_t is less probable in the current policy than in the old one. Our new loss function can then be defined as $L_{\text{policy}}(\theta) = \mathbb{E}_t[q_t(\theta)A_t] = \mathbb{E}_t[\frac{\pi(a_t|s_t;\theta)}{\pi(a_t|s_t;\theta_{\text{old}})}A_t]$, where $A_t = R_t - V(s_t;\theta)$ is the advantage function. However, if the action under current policy is much more probable than in the previous policy, the ratio q_t may be large, leading to a large policy update step. To circumvent this problem, the original PPO algorithm [8] adds a

34th Conference on Neural Information Processing Systems (NeurIPS 2020), Vancouver, Canada.



constraint on the ratio, which can only be in the range 0.8 to 1.2. The modified loss function is now $L_{\text{policy}}(\theta) = \mathbb{E}_t[-min(q_tA_t, clip(q_t, 1-C, 1+C)A_t)]$ where the C is the clip hyperparameter (common choice is 0.2). Finally, the value loss and entropy bonus are added into the total loss function as usual: $L(\theta) = L_{\text{policy}} + c_1 L_{\text{value}} - c_2 H$ where $L_{\text{value}} = \mathbb{E}_t[\|R_t - V(s_t;\theta)\|^2]$ is the value loss and $H = \mathbb{E}_t[H_t] = \mathbb{E}_t[-\sum_j \pi\left(a_j \mid s_t;\theta\right)\log(\pi\left(a_j \mid s_t;\theta\right))]$ is the entropy bonus which is to encourage exploration.

3 Experiments and Results

3.1 CartPole Environment

In the first experiment, we plan to study the classic control problem `CartPole-v0` environment provided by the OpenAI Gym (9). The environment is shown in the Figure1. In this environment, a pole is attached by an un-actuated joint to a cart, which moves along a frictionless track. The pendulum starts upright, and the goal is to prevent it from falling over by increasing and reducing the cart's velocity. The RL agent needs to output the appropriate action according to the observation it receives at each timestep.
The cart-pole environment mapping is:

- Observation: a four dimensional vector s_t representing the cart position, cart velocity, pole angle, pole velocity at tip.
- Action: there are two actions $+1$, -1 in the action space.
- Reward: A reward of $+1$ is provided for every timestep that the pole remains upright. The episode ends when the pole is more than 15 degrees from vertical, or the cart moves more than 2.4 units from the center.

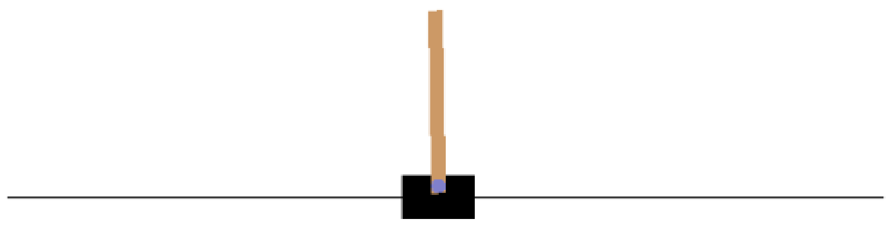


Figure 1: **CartPole environment in OpenAI Gym.** Screenshot from the OpenAI Gym (9).

3.2 MiniGrid Environment

We plan to study the PPO algorithm in solving problems provided by the environment MiniGrid (10). This environment follows the standard OpenAI Gym (9) API thus it is a good choice for the RL benchmarking. In this environment, the RL agent will receive $7 \times 7 \times 3 = 147$ dimensional vector for observation and would need to determine the action from action space $\mathcal{A}$ in which there are 6 possibilities.

The MiniGrid environment mapping is:

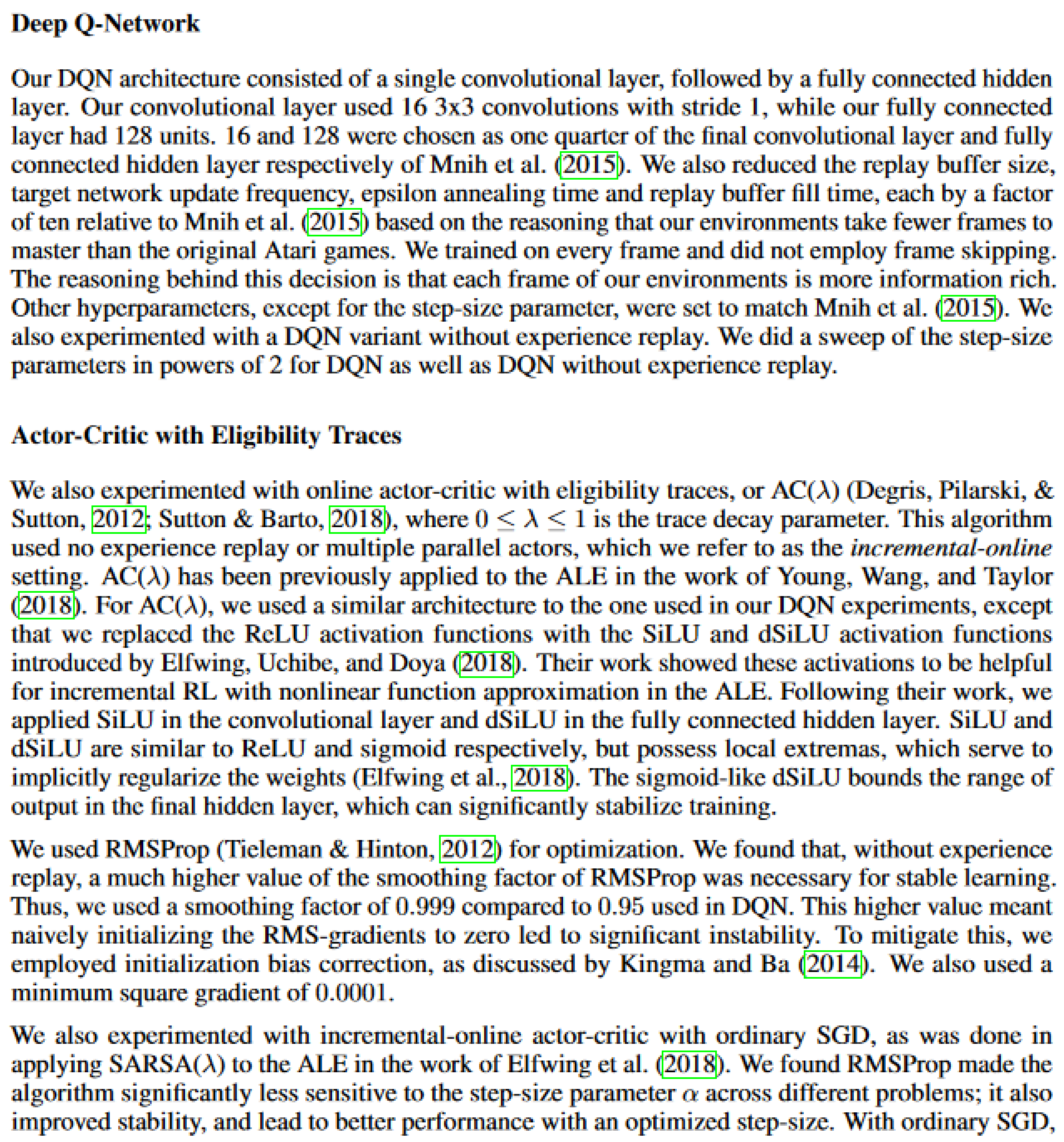
- Observation: a 147 dimensional vector s_t
- Action: there are six actions $0, \cdots, 6$ in the action space. Each representing the following,

  - Turn left
  - Turn right
  - Move forward
  - Pick up an object
  - Drop the object being carried
  - Toggle (open doors, interact with objects)
- Reward: Get 1 when reaching the goal with small penalty subtracted from the return for the number of steps to reach the goal.

We plan to study the following two testing cases for this project. They are the *empty-env-6x6* as shown in Figure3 and the *empty-env-8x8* as shown in Figure5. In these two environments, the RL is expected to go from the starting point (upper left in the figure) to the destination (lower right green box). The reward scheme in this MiniGrid environment is as follows. In each of the step, the agent will receive a small negative reward as penalty. The ideas behind this design is to encourage the agent to take the shortest possible path. The agent gets reward of 1 when it succeed, and 0 for failure.

The RL agents will be implemented with a recent released python package `TF-Agents` (11).

Example: MinAtar with DQN + Actor-Critic with Eligibility Traces



Deep Q-Network

Our DQN architecture consisted of a single convolutional layer, followed by a fully connected hidden layer. Our convolutional layer used 16 3x3 convolutions with stride 1, while our fully connected layer had 128 units. 16 and 128 were chosen as one quarter of the final convolutional layer and fully connected hidden layer respectively of Mnih et al. (2015). We also reduced the replay buffer size, target network update frequency, epsilon annealing time and replay buffer fill time, each by a factor of ten relative to Mnih et al. (2015) based on the reasoning that our environments take fewer frames to master than the original Atari games. We trained on every frame and did not employ frame skipping. The reasoning behind this decision is that each frame of our environments is more information rich. Other hyperparameters, except for the step-size parameter, were set to match Mnih et al. (2015). We also experimented with a DQN variant without experience replay. We did a sweep of the step-size parameters in powers of 2 for DQN as well as DQN without experience replay.

Actor-Critic with Eligibility Traces

We also experimented with online actor-critic with eligibility traces, or AC(λ) (Degris, Pilarski, & Sutton, 2012; Sutton & Barto, 2018), where $0 \leq \lambda \leq 1$ is the trace decay parameter. This algorithm used no experience replay or multiple parallel actors, which we refer to as the *incremental-online* setting. AC(λ) has been previously applied to the ALE in the work of Young, Wang, and Taylor (2018). For AC(λ), we used a similar architecture to the one used in our DQN experiments, except that we replaced the ReLU activation functions with the SiLU and dSiLU activation functions introduced by Elfwing, Uchibe, and Doya (2018). Their work showed these activations to be helpful for incremental RL with nonlinear function approximation in the ALE. Following their work, we applied SiLU in the convolutional layer and dSiLU in the fully connected hidden layer. SiLU and dSiLU are similar to ReLU and sigmoid respectively, but possess local extremas, which serve to implicitly regularize the weights (Elfwing et al., 2018). The sigmoid-like dSiLU bounds the range of output in the final hidden layer, which can significantly stabilize training.

We used RMSProp (Tieleman & Hinton, 2012) for optimization. We found that, without experience replay, a much higher value of the smoothing factor of RMSProp was necessary for stable learning. Thus, we used a smoothing factor of 0.999 compared to 0.95 used in DQN. This higher value meant naively initializing the RMS-gradients to zero led to significant instability. To mitigate this, we employed initialization bias correction, as discussed by Kingma and Ba (2014). We also used a minimum square gradient of 0.0001.

We also experimented with incremental-online actor-critic with ordinary SGD, as was done in applying SARSA(λ) to the ALE in the work of Elfwing et al. (2018). We found RMSProp made the algorithm significantly less sensitive to the step-size parameter α across different problems; it also improved stability, and lead to better performance with an optimized step-size. With ordinary SGD,