Deep Reinforcement Learning for 2048

Jonathan Amar
Operations Research Center
Massachusetts Insitute of Technology
amarj@mit.edu

Antoine Dedieu
Operations Research Center
Massachusetts Insitute of Technology
adedieu@mit.edu

Abstract

In this paper, we explore the performance of a Reinforcement Learning algorithm using a Policy Neural Network to play the popular game 2048. After proposing a modelization of the state and action spaces, we review our learning process, and train a first model without incorporating any prior knowledge of the game. We prove that a simple Probabilistic Policy Network achieves a 4 times higher maximum score than the initial random policy. We then try to improve the learning process with Approximate Dynamic Programming. Finally we test the performances of our network by coupling it with Monte-Carlo Tree Search in order to encourage optimal decisions using an explorative methodology.

1 Introduction

1.1 Motivation

Reinforcement Learning has enjoyed a great increase in popularity over the past decade by controlling how agents can take optimal decisions when facing uncertainty. The first best-known story is probably *TD Gammon*, a Reinforcement Learning algorithm which achieved a master level of play at backgammon Tesauro [1995]. Recently Deep Learning has scaled Reinforcement Learning methods to a new range of problems and thus to media success Google's DeepMind developed algorithms that were able to successfully play Atari games Mnih et al. [2013] and defeat the world Go champion Silver et al., 2016. This recent AI accomplishment is considered as a huge leap in Artificial Intelligence since the algorithm should search through an enormous state space before making a decision.

After reviewing the state of the art in RL, we chose to work on an average complexity game to obtain results within the project timeline. We have studied the game of **2048** which is played on a 4×4 grid with each cell being a power of 2. The objective of the game is to reach the score of 2048 by shifting the grid along any of the four directions *up, down, right, left* and merging two consecutive cells with same value. Such a game presents an intuitive human strategy that consists of keeping larger elements in a corner. Starting from a random algorithm, we aim at learning such a strategy and test if we can outperform the human player.

1.2 Literature Review and Model

Gathering techniques from previous literature in RL and combining them with elements of DP, we present in this section the main ideas we implemented in our work. We start with a few theoretical elements. In Dynamic Programming, we define the **value-to-go** function as the current expected cumulated rewards we may obtain starting from our current state and following an optimal policy. We can relax this notion with the value-to-go function under a given policy. These functions allow us to define the recursive **Bellman equation** structuring our strategy: optimal decisions allow us to maximize our expected long-term reward. The second key idea in RL is to learn and approximate

31st Conference on Neural Information Processing Systems (NIPS 2017), Long Beach, CA, USA.

the value function using simulations under multiple policies in order to simultaneously understand the rules and the best strategies characterizing the game. We do so by inferring the impact of the long term reward from decisions made at a prior given stage: essentially we will learn how to make the decisions that will surely maximize the long-term reward.

We approached the problem without incorporating prior knowledge of the game: we worked with parametric models approximating the decision-making functions, using a Neural Network initialized with random weights. Inspired by Silver et al. [2016], we decided to implement a **Policy Network** that explores the 4 different decisions (some of them being infeasible) by using a probabilistic model. At the end of each game, we compute the final reward and propagate it to every visited state-actions pairs. After running multiple games, we modify the weights in our NN in order to encourage making decisions with a positive impact on the long term reward, and discourage those with a negative impact.

In parallel, we tried to integrate initial knowledge of the game by implementing an Approximate Dynamic Program to best fit the value function using basis functions which we believed represented how "good" the current state is. We saw no significant performance improvement with this method. Finally, when it comes to making optimal decisions and dropping the learning context, we decided to implement a **Monte Carlo Tree search** strategy. At a given state, MCTS exploits our learnt Policy Network by exploring future possible states ,MCTS then makes the decision that it believes will lead to highest reward.

We structure the rest of the paper in the following order. Section 2 presents the game 2048 and our visualization tool. In Section 3, we study two policy-learning methodologies. We then suggest a couple of tricks to increase our learning rate in Section 4. Finally given the learnt policy, we introduce efficient strategies in order to maximize our winning ratios, and study the performance of our algorithms in Section 5.

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?

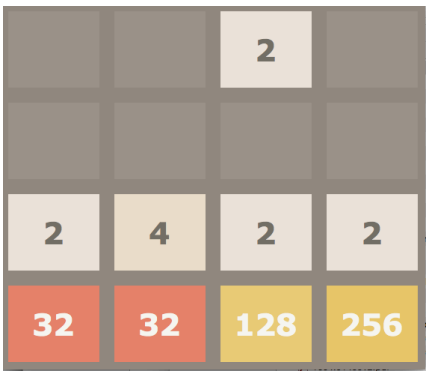
2.2 State Representation

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 $\{0,1\}^{256}$.

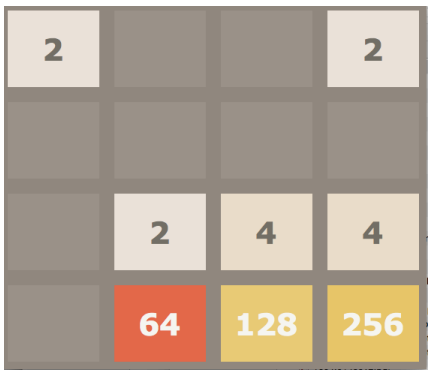
Note that an alternative approach could consist in representing the state space by an array of size 16 with each entry being an integer in $[0, 15]$. This model presents the advantage of introducing a metric distance between the cell values After considering both alternatives but selected the first model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

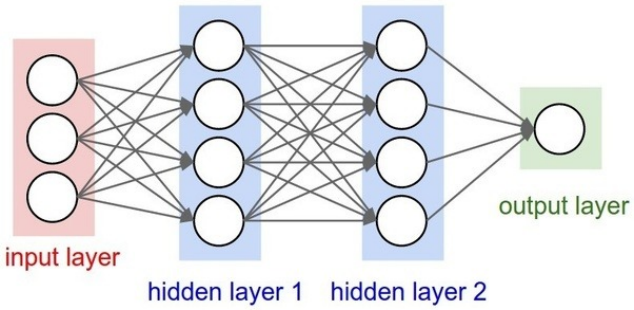
3 Using the Policy Network with Reinforcement Learning

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

update the Policy Network by Reinforcement Learning, according to the action-sampling strategy used.

3.2.2 Back propagation for ϵ-greedy Learning

In the ϵ-greedy case, we aim at updating the policy Network to achieve higher reward. The key idea behind is as follows. Assume from a vector $\mathbf{p}$ we have taken an action $\mathbf{a}$ corresponding to the highest value of $\mathbf{p}$. We associate a reward r to this action. If the reward is positive we try to encourage the action, i.e. we want $\mathbf{p}$ to approximate the vector $\mathbf{a}$. For negative rewards we do the opposite. More generally the gradient ascent goes along $r \times \mathbf{a} - \mathbf{p}$. Consequently the **back-propagation** formulas are given by:

Input: The matrices of weights $\mathbf{W_1}, \mathbf{W_2}, \mathbf{W_3}$, states $\mathbf{X}$, hidden states $\mathbf{H}_1$ $\mathbf{H}_2$, probabilities $\mathbf{P}$, actions taken $\mathbf{A}$ and rewards $\mathbf{R}$ for the N_{batch} games
Output: The updated matrices of weights $\mathbf{W_1}, \mathbf{W_2}, \mathbf{W_3}$

1. Define the rewarded vector $\mathbf{V} = \mathbf{R} * (\mathbf{A} - \mathbf{P})$
2. Define $d\mathbf{W}_3 = \mathbf{V}^T \mathbf{H}_2$
3. Define $d\mathbf{h}_2 = \mathbf{V}\mathbf{W}_3$ and $d\mathbf{W}_2 = d\mathbf{h}_2 \left[\mathbf{h}_2 < 0\right]^T \mathbf{H}_1$
4. Define $d\mathbf{h}_1 = d\mathbf{h}_2\mathbf{W}_2$ and $d\mathbf{W}_1 = d\mathbf{h}_1 \left[\mathbf{h}_1 < 0\right]^T \mathbf{X}$

3.2.3 Back propagation for the Probabilistic Network

We wish to obtain a policy network that will yield optimal decisions, i.e. make high values actions more likely. In the stochastic decision-making setting, **back-propagation** algorithm can be written as for the ϵ-greedy case. However, since we are learning a probability distribution hence we use the formula:

$$\nabla_\theta \mathbb{E}\left[f(\mathbf{x})\right] = \mathbb{E}\left[f(\mathbf{x}) \nabla_\theta \log p(\mathbf{x})\right]$$

Consequently we use the reward vector $\mathbf{V} = \mathbf{R} * (\mathbf{A} - \log \mathbf{P})$ in the previous algorithm.

3.2.4 Learning rate

Given the computed $d\mathbf{W}_1, d\mathbf{W}_2, d\mathbf{W}_3$, we update our neural net by gradient ascent with learning rate α. We use a large learning rate (say 0.01) for the first moves to allow large variations before reducing it. As recommended in Karpathy [2016] we use the RMSProp algorithm to compute our gradient descent, that is we introduce a display rate ρ (say 0.99) and compute:

$$\eta_{t+1} = \rho \eta_t + (1-\rho) \|d\mathbf{W}_i\|_2^2$$

$$\mathbf{W}_i = \mathbf{W}_i + \frac{\alpha}{\sqrt{\eta_{t+1} + 1e-5}} d\mathbf{W}_i, \ i = 1, 2, 3$$

3.3 Learning Records

We compare the two learnings methods and motivate the choice of the Probabilistic Network. We fix a size of batches $N_{batch} = 10$ games, a learning rate of $\alpha = 0.01$

3.3.1 ϵ-greedy learning

In figure 3, we show the poor performances of ϵ-greedy learnt policy over more than 600 batches. We keep track of the averaged sum of the grid over a batch and see no real improvements over the initial random policy.

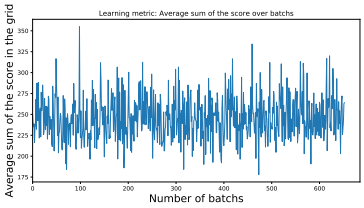


Figure 3: Average score over batches with ϵ-greedy learning

From our understanding, this method is not adapted to the structure of the game: playing randomly a move in 2048 can disturb all the previously built strategy and have a huge influence on the game.

3.3.2 Probabilistic Network

The probabilistic network proved to be more efficient than the random policy for the same metric. In particular, we notice a gap in the average length of an episode after several hundreds of iterations in figure 4.

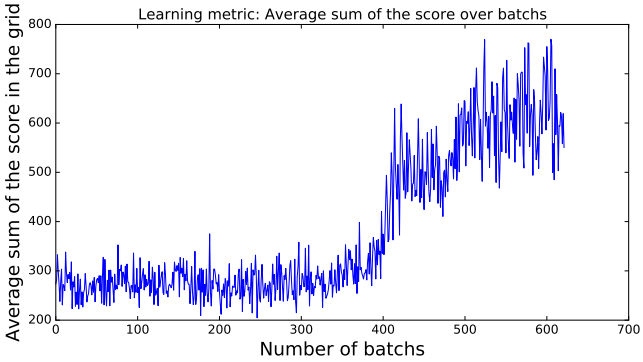


Figure 4: Average score over batches with Probabilistic Network

Understanding the learning The next plots in figure 5 illustrate the learning procedure of the neural network and give better insight for further improvements. At every iteration, for the ranking and the probabilistic methods, we would like the vectors $\mathbf{p}$ returned by the forward algorithm to be close to a vector with three 0 and a single 1. Hence we measure the distance between each vector $\mathbf{p}$ and this ideal certainty case:

$$d(\mathbf{p}) = 1 - p_{u^*} + \sum_{i \neq u^*} p_i \text{ where } u^* = \arg\max_u \mathbf{p}_u$$

We average the results over one game and average again over batches. We see similar results to previous figure 4. After several hundreds of games, the Network starts learning, thus it is more accurate in its probability predictions, leading to less variance.

If we represent the average distance to certainty over a single game, at the very beginning of the game and after 500 iterations we see two very different plots. In Figure 6a all probabilities are almost equal to 0.25 hence $d(\mathbf{p})$ is always very close to its maximum of 1.5. In Figure 6b there is much less variance over the games. Most distances are close to 0 meaning that the Policy Network knows what is the good decision to make. Note that the noisy episodes come from the apparition of new high cells. The learning seems insufficient for such moves.

3.4 Achieving 2048?

Let us finally recall that the goal of the game is to achieve 2048. Unfortunately, we were not able to reach this score. But our Policy Network achieved a significant number of times **1024** whereas

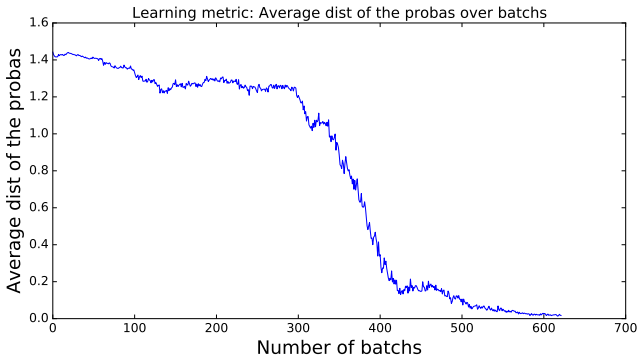
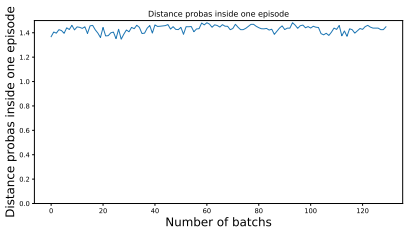


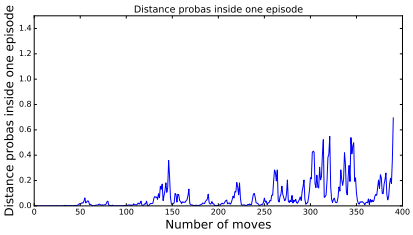
Figure 5: Average probability distance to ideal case over batches with Probabilistic Network

Figure 6: Comparison at two stages of the learning process.

(a) Before learning



(b) After learning



the random network never reached **256**. This shows the performance of a Reinforcement Learning algorithm, even with a simple model over random decisions. This motivates the next sections to try to use more advanced methods to improve the performance of our model.

4 Rapid Learning through better Initialization

While running our Reinforcement Learning algorithm, we have noticed that starting from a random policy net incurs a large cost to our learning rate by increasing its time. Our RL algorithm explores too much before understanding the underlying physics and strategies of the game. This initial cost motivated the implementation of a warm start, in the sense that we wish to feed our policy net basic human knowledge of the game.

4.1 Feeding Data

Our first idea was to hand-play several games, that we consider strategic. Typically we played using basic strategies, i.e. maintaining an ordering of the cells by keeping the largest values in a corner. As mentioned in Section 3, we update the weights from our Neural Net based on the normalized rewards from a batch of played games. Our idea was to repeatedly add these hand played games to the batch. Therefore the policy network will try to fit the human strategy used in our hand-played games while it remains better than the current strategy inferred by the network.

While this initialization method feels promising, it does not scale well in practice due to the lack of data. We suggest that the net gets stuck in a local minimum, and could not deal with the true uncertainty of the game in a reasonable way. Indeed while it learns the human way to play, it is also repeatedly learning that 2-cells are showing up in the same spot as described by the hand-layed games. Therefore the NN is simultaneously learning a strategy and an erroneous distribution characterizing the 2-cell appearance. Similarly to DeepMind's AlphaGo approach, the best approach would be to have a huge dataset with thousands of human games so that the Policy Network can learn how to play like a human.

4.2 Approximate Dynamic Program

We then decided to teach our policy network our human perspective of the game and its physics. We implemented an Approximate Dynamic Programming ADP (de Farias and Van Roy [2003]) methodology where our algorithm moves according to basis functions that we believe should guide the decision making. We encoded indicator functions that illustrate that the grid is well ordered, i.e. that there is positive gradient of cell values directed towards a corner. We show in figure 7 an example of ordered grid. Here are the functions:

- $\mathbf{1}\{\text{Largest cell is in a corner}\}$
- $\mathbf{1}\{\text{Descending cells along vertical axis from largest corner}\}$
- $\mathbf{1}\{\text{Descending cells along horizontal}\}$
- $\mathbf{1}\{\text{Descending cells along diagonal}\}$

Figure 7: Example of ordered grid, gradient towards the bottom left corner.

Theory for ADP: Let us give a few theoretical elements motivating the ADP. Given the reward system, let us denote by $J(\mathbf{x}_t)$ the value function applied at state the current state x_t. The Bellman recursive equation reflects that the best decision to make u_t at time t and state x_t will maximize the expected values for the next state:

$$J(\mathbf{x}_t) = \max_u \mathbb{E}_{\mathbf{x}_{t+1}}[J(\mathbf{x}_{t+1}) \mid \mathbf{x}_t, u]$$

Without loss of generality, here we have considered a non discounted termination reward.

The key idea behind ADP is to approximate the value function by a linear combination of basis function which we will denote $\phi_k(x), k \in \{1, \dots, K\}$. Similarly to the policy network which approximates the best policy, we are here implementing a value-function approximation using basis functions rather than a neural net. We will thus proceed our learning of the best approximation using a gradient descent method.

At a batch τ, let us assume we have an linear approximation with weights $\boldsymbol{\lambda}$:

$$\tilde{J}_\tau(.) = \sum_k \lambda_{\tau,k}\phi_k(.)$$

During this batch we play by trying to maximize the next state's value, i.e. we choose at every turn the policy solving the equation:

$$\tilde{u}_\tau(x_t) = \arg\max_u \mathbb{E}_{x_{t+1}}[\tilde{J}_\tau(x_{t+1}) \mid x_t, u]$$

Following the induced policy, we collect rewards r for different games in the batch. We update the weights $\boldsymbol{\lambda}$ by encouraging better games and penalizing others. We proceed by gradient descent in order to best fit the true reward function using a least square estimator LSE. More specifically, for $i \in \{\text{batch and visited states}\}$:

$$\Delta := \vec{\nabla}_\lambda \sum_i (y(i) - \lambda\phi(i))^2 = 2\sum_i \phi(i)(\lambda\phi(i) - y(i))$$

And we update the weights with a learning rate α:

$$\lambda_{\tau+1} := (1-\alpha)\lambda_\tau + \alpha\Delta$$

We initialized the weights to 1, and run the ADP for several batches until we are satisfied with the performance of the learnt strategy. The idea is to then fit the policy network to the strategy learnt through the ADP, until a convergence criterion is met.

4.3 Results

Since the ADP's general performance was not satisfactory, we decided not to pursue along this path. Besides this approach went against our initial motivation of learning how to play without giving any intelligence of the game, for generalizability purposes.

Further we have noticed that some weights were being updated to negative values, which would go against human perspective of the game. This may be due to correlation between different functions though, and therefore a negative weight on ϕ_k may balance a positive weight on $\phi_{k'}$. Consequently we believe that the ADP structure from the functions we implemented may not be the right match for this reward.

5 Efficient Game-play Methods

In this section, we use the learnt strategies and now focus only on maximizing our reward at every game, by exploiting optimally the policies.

5.1 Switching the Probabilistic Policy Net

A fully exploitative strategy seems the most appropriate to maximize the reward. Thus we drop the exploration factors from our strategy. More specifically we compute the output $\mathbf{p}$ of the neural network, until now viewed as a ranking or probabilities and make the decision $u^* = \arg\max_u \mathbf{p}_u$ with probability 1 (rather than $1 - \epsilon$).

The choice of dropping the explorative factor was motivated by the specificity of the game. At an advanced stage of the game, a suboptimal decision could lead very quickly to termination of the game, and thus a much smaller reward than expected.

5.2 Monte-Carlo Tree Simulation

After improving our decision making by removing the stochastic aspect of our Neural Net, we still noticed that some decisions were suboptimal and quickly led to a loss. We balanced that effect by using the more classical approach of tree search Silver et al. [2016]. Starting from a state x, we consider the different actions u, and the different state-actions visited during the tree search in $\{(s,a)_i\}$. We store in two tables the **action-value** $Q(s,a)$ and **number-of-visits** $N(s,a)$ and the **prior probabilities** $P(s,a)$ initialized at $\mathbf{p}(s)_a$,the output from the policy network. We traverse the tree N_{tree} times, each time with a depth L. While we are traversing the tree, we choose the next action from state s_t:

$$a_t := \arg\max_a Q(s_t,a) + \alpha \frac{P(s_t,a)}{1+N(s_t,a)}$$

The second term is an exploration bonus for actions that have been tested less often. During the tree search, we simulate random apparitions of 2s. We perform the tree search until reaching a leaf node s_L. The leaf node is evaluated by a rollout using the policy network and given the value

$$V(s_L) = \text{maximum atteined during the rollout}/2048$$

At the end of the simulation, we update our values of the visited states

$$N(s,a) \leftarrow \sum_{i=1}^{n} \mathbf{1}(s,a,i)$$

$$Q(s,a) \leftarrow \frac{1}{N(s,a)} \sum_{i=1}^{n} \mathbf{1}(s,a,i) V(s_L^i)$$

where $\mathbf{1}(s,a,i)$ is the indicator function of the event (s,a) was visited during tree-simulation i. The leaf node reached on ith simulation is denoted s_L^i.

Once the tree search is complete, i.e. $n = N_{\text{tree}}$, we make the decision which was visited the most often form the current state x. The strength of MCTS is its ability to focus its exploration on good branches of the tree. The MCTS simulation can be quite computationally expensive, especially if the leaf nodes values are rolled out until termination of the game.

5.3 Performance

6 Discussion

Given our unsatisfactory results, we reflect on different choices we made during our strategy design that may have setback the performances

6.1 Designing the correct reward

A substantial question was in designing the reward function. In our work, we considered back-propagating the rewards of the games played through the policy network. Past literature Silver et al. [2016] assign a binary reward from $\{0, 1\}$ depending on whether the game led to a win. The reward is then linked to every state-action visited during the game-play of our reinforcement learning. Our initial choice was to incur a reward of 1 if we achieved the goal of 2048, and otherwise incur a 0 reward.

In our setup however, the probability of wining using a random policy net is too small, and this binary reward does not separate very similar games. We preferred using rewards encouraging a more natural, progressive learning. Our policy network will at a high level try to fit the value function reached under an optimal policy. Given that we do not consider a time discount factor for our rewards, the value function represents the expected termination reward one can hope to reach knowing our current state. From the Bellman-Equation, the best action to make is the one maximizing the expected value function of the updated state. By propagating the Bellman equation, the value function evaluated at the initial state should represent the best achievable reward under an optimal policy. More specifically, the value-to-go function *J(.)* represents the expected number of rounds one can wish to play at a current state *x*.

Other references studied a temporal discount factor. This would allow to set more importance on the decisions made at an early stage of the game when updating our neural network weights. Also we considered having a stage dependent value function, rather than representative of the total run-through. These weren't properties we wished our value function to have.

6.2 Adjacent nets and Partitioned learning

After realizing that our policy network learns quickly enough a decent policy over the first moves (i.e. until reaching 512), and made decisions with quasi-certainty, we suggested to split our decision making in two policy networks. More specifically the moves made at the beginning were made with very high probability. The longer the game lasted, the lower were these probabilities The unbalance of the extremal values compared to the explorative ones incurred computational issues when calculating the gradient in our back-propagation.

Finally future extensions will consider deeper layers in our Neural Net for the reinforcement learning stage. We also consider building a network to approximate the value function or the Q function.

References

D. P. de Farias and B. Van Roy. The linear programming approach to approximate dynamic programming. *Operations research*, 51(6):850–865, 2003.

A. Karpathy. *Reinforcement Learning: Pong from Pixels*, 2016. http://karpathy.github.io/2016/05/31/rl/ [Accessed: Whenever].

V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3): 58–68, 1995.