Deep reinforcement learning and efficient exploration

EPFL
Lab of Artificial Intelligence (LAI)
Supervised by Prof. Boi Faltings, Igor Kulev, and Goran Radanovic'

Ruben Fiszel

Spring 2016

Contents

1 Introduction

Deep learning and reinforcement learning have been used with great success to build smart agents. Tasks that seemed to be out of reach for the machine can now be learned by an AI. DeepMind uses it to play games for the Atari 2600 platform from the raw pixels only. They are able to reach super-human performance on some titles and they extend continuously the limits. Recently, the AlphaGo project from Deepmind beat the Go World champion, Lee Sedol. Go was believed to be the proof of the human superiority over the machine because of its high branching factor that made any tree search over all possible states inefficient. In this project, we will investigate some core ideas around deep reinforcement learning.

We explore the use of deep reinforcement learning and efficient exploration, and compare td-lambda and q-learning to solve problem modelized by Markov Decision Process (MDP). First, we introduce all the concepts necessary to understand deep reinforcement learning, then we explore its use for 2048 (no attempt of deep reinforcement learning seems to have been made before despite the popularity of the game) and a deterministic variant 6561. We then study efficient exploration and its use for 2048 and 6561. Finally, we introduce a variant of a recently published technique of deep reinforcement learning.

The code

All the results achieved here are done through a deep reinforcement learning library in scala on top of another, the neural network library deeplearning4j. Every algorithm have been reimplemented even the smallest. This deep reinforcement learning library was made for the sole purpose of this project and is the first of its kind for Scala and the JVM. A special care was given to generalize all the code so it would be easily adaptable to any MDP.

The deeplearning4j library was not in a mature state during the period of this project which was an additional technical difficulty. One might wonder about this choice of framework. The reason is that, bugs around the usage of neural networks are very tough to spot. Indeed, everything can run just fine in appearance. We were convinced that Scala provided a level of abstraction that was crucial for the project.

2 Preliminaries

Deep Learning

Deep learning coud be defined as the adjustment of the parameters of a neural network with more than 1 hidden layer such that it approximates at best a "true" underlying function. To train a neural network, you need to feed him numerous inputs that are labelled with targets, the outputs returned by the "true" function. By example, to make a classifier, you would train the network by giving him examples of images and their classification made by trusted "supervisors". By using backpropagation, which is an efficient way to find the gradient of the network for each data, you can adjust the parameters of the network in a direction which will decrease the error between the neural network's current output and the target. This manner of training by using the gradient is called gradient descent. If you provide data labeled by supervisors as in the previous example, it is classified as supervised learning.

Neural network

An artificial neural network is a family of models inspired by the biological neural network inside the brain. Here, we focus on multi-layered feedforward network of nodes which approximate a given function. Essentially, a neural network is the decomposition of a parametric function into simple mathematic operations such as additions, multiplication and activation functions (We use only ReLu). Each node has multiple inputs and one single output, which is also the input of the other nodesof the next layer. The nodes are grouped into layers. Layers are arranged one after the other such that there is no intra-layer connection between nodes of the same layer but every output of the nodes of one layer are used as the input of the nodes of the next layer.

Inside each node, the operations applied are

$$f(\mathbf{X})_{\mathbf{W},b} = a(\mathbf{W}^\intercal \mathbf{X} + b)$$

where f is the output function, a is the activation function, W is the vectors of weights attributed to each input, b is the bias. It's a parametric function, with parameters W and b. W and B are adjusted during training but fixed during the evaluation.

This model is very simple but it enables, after training, to approximate very complex functions, such as the ones that need "deep" understanding of the data. That under-

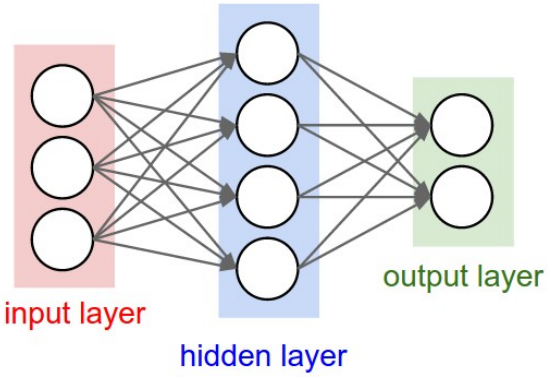


Figure 2.1: A neural network with only 1 hidden layer

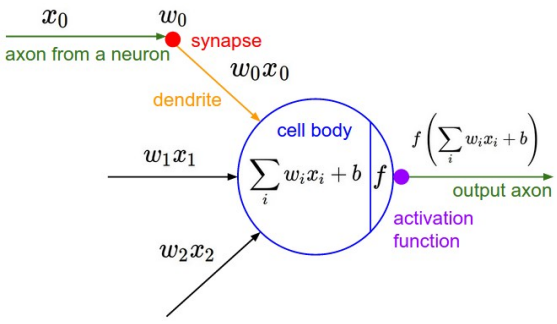


Figure 2.2: A mathematical model of a neuron

standing is in fact the ability to abstract and transform the data such that non-trivial patterns appears. This is one form of intelligence and what makes neural network very powerful models.

The input layer nodes just output the data fed to the neural network, and the output of the nodes of the output layer are the output of the neural network. The number of nodes in a layer is the size of the layer. The neural network must have its input layer of the same size than his input dimension and its output layer of the same size that its output dimension. Dimension here is the number of numbers (here Float) used to define the data. An image has commonly a dimension of width*height*3. 3 being the different value in the RGB (Red-Green-Blue) encoding.

Once input and targets are given to the neural network, the parameters are adjusted by gradient descent. Gradient descent use the gradient found by backpropagation to adjust the parameters to minimize the error between the output of the neural network and the targets. We use stochastic gradient descent that is a variant of gradient descent which is faster to converge thanks to statistical properties of random mini-batch of training.

Neural networks parameters are randomly initialized instead of being set to 0 at start. This enable to have different convergence depending of the seed and it has been shown that this random initialization lead to faster convergence.

Markov Decision Process

A very common way, to model problems and games where an agent interact with an environment is through a Markov Decision Process.

A Markov decision process is a 5-tuple $(S, A, P_\cdot(\cdot,\cdot), R_\cdot(\cdot,\cdot), \gamma)$, where

- S is a finite set of states,
- A is a finite set of actions (alternatively, A_sy is the finite set of actions available from state s),
- $P_a(s, s') = \Pr(s_{t+1} = s' \mid s_t = s, a_t = a)$ is the probability that action a in state s at time t will lead to state s' at time $t + 1$,
- $R_a(s, s')$ is the immediate reward (or expected immediate reward) received after transition to state s' from state s,
- $\gamma \in [0, 1]$ is the discount factor, which represents the difference in importance between future rewards and present rewards.

A common simple MDP is the Multi-Armed-Bandit where a gambler can choose from a different number N of slot machine. Each machine has a distribution of reward that is unknown to the gambler and specific to that machine. The MAB has only one state s, N actions to choose from on that state. For all action a, $P_a(s, s) = 1$ and the rewards' distribution, are specific to each machine thus to each A. The MAB is important in the literature because it symbolize the dilemma in reinforcement learning

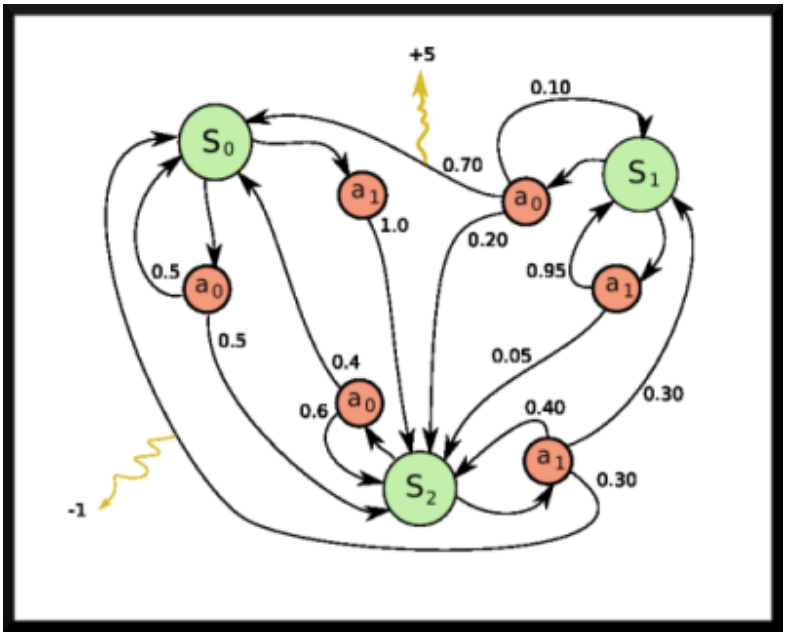


Figure 2.3: An example of a MDP

between exploitation and exploration. The crucial tradeoff the gambler faces at each trial is between "exploitation" of the machine that has the highest expected payoff and "exploration" to get more information about the expected payoffs of the other machines.

An episode is a history of state and actions that begin at a state s_0 and finish at an end state. An end state is a state where no actions are available. An episode is of the form: $s_0, a_0, \ldots, s_t, a_t, r_t, s_{t+1}, a_{t+1}, r_{t+1}, s_{t+2}, \ldots$

A distinction between discrete and continuous here is used to qualify the action space. In chess by example, the action space is discrete since you choose 1 move among a discrete set of move. For a robot, the intensity signal to transmit to his motors could be continuous (eg: from 0.0 to 1.0). This is called continuous control. We focus here on discrete control MDP.

Reinforcement Learning

Reinforcement learning is using the interaction of an agent with its environment for its training. The goal is to learn a strategy that is as close as possible to the optimal strategy. A strategy which at each state choose an action is called a policy. In the context of MDP, finding the optimal strategy is finding the optimal policy π^* that maximize the expected sum of rewards of an episode.

To find and learn the optimal policy, we use an auxilliary function: the value function

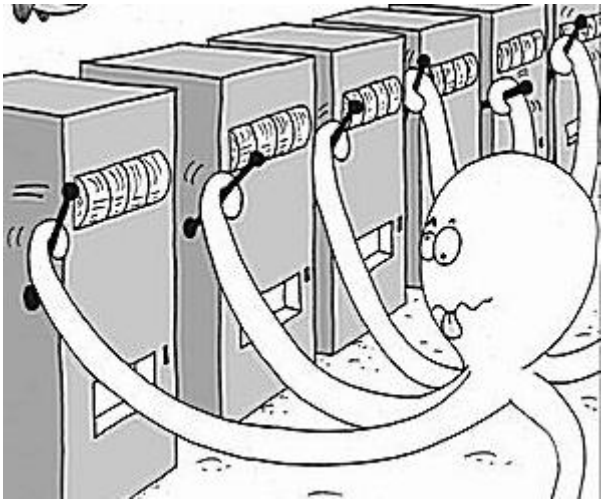


Figure 2.4: A smart agent playing MAB

V^π defined as

$$V^\pi(s) = E\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s\}$$

where π is the policy followed. The expectation is over both the agent's stochastic policy and the environment dynamics. This equation is transformed into a **Bellman equation**:

$$\begin{aligned} V^\pi(s) &= E_\pi\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s\} \\ &= E_\pi\{r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V(s')\right] \end{aligned}$$

This equation can be solved as long as we have boundary conditions and the policy π. Here the boundary conditions are the values of the end states. It can be the final score in a 1-player game, 0 when the rewards are incrementally given with each transition, or +1 and -1 for victory and loss in a 2-player setting. Gamma γ is the discount factor, it represents the preference for getting the same value (reaching an end-state) sooner than later. Once we have this function, the optimal policy is

$$\pi^*(s) = \arg\max_a E_s[R_a(s,s') + V(s')|s,a]$$

where s' is a state reachable from s after the action a is done. There is a cyclical dependency between π^* and V^{π^*}. This is solved by **Policy Iteration**: We start out with some diffuse initial policy (with a randomly initialized Value function) and evaluate the Value function of every state under that policy by turning the Bellman equation into an update.

The value function effectively diffuses the rewards backwards through the environment dynamics and the agent's expected actions, as given by its current policy. Once we estimate the values of all states we will update the policy to be greedy with respect to the Value function. We then re-estimate the Value of each state, and continue iterating until we converge to the optimal policy (the process can be shown to converge).

The intuition behind the convergence is that when the neural network starts the training, it does not know anything about the game. Its policy will be random. An observation, that is crucial for Monte-Carlo methods[1], is that if you play randomly from a given state, you are more likely to have a better score if the state has a high expected discount reward sum. By example, it means better hand at poker should win more often even when you play randomly. This simple fact is what enables convergence as playing randomly still give relevant informations about each state. Good states will be more likely to have a higher positive adjustment. Gradually during the training, the agent plays a strategy that is less random (inherited from the initial parameters of the neural network) and more close to the optimal policy. This is convergence.

Reinforcement learning works with neural networks by assigning targets during the policy update of the training. The targets are calculated for V^π or Q^π in an unsupervided manner from the **Policy Iteration**. The combination of using a deep neural network as the function approximator and reinforcement learning is called deep reinforcement learning.

Q function

The Q function is an alternative auxilliary function that calculate directly the expected value of following an action from a state s instead of calculating the sum of rewards reachable from a state s'.

$$\begin{aligned} Q^\pi(s,a) &= E_\pi\{r_t + \gamma V^\pi(s_{t+1}) \mid s_t = s, a_t = a\} \\ &= \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s')\right] \end{aligned}$$

One benefit of using the Q function is that the optimal policy become simpler to formulate: $\pi^*(s) = \arg\max_a Q^*(s,a)$. It also enable us to use q-learning which is an efficient update rule Q function that we will see later.

V is still present in the equation but the update of Q only use V when it is trivially defined for the updates involving the end-states. An example where q-learning seem to be more efficient are Atari games where the action are the possible input of the controller and the states are the raw pixels.

Both functions, Q and V, can be updated with rules that do not require to explore each action of each state. $TD(\lambda)$ (Temporal-difference lambda learning) for Value function, SARSA or Q-learning for Q function are update rules that converge to the optimal policy when applied with an ϵ-greedy strategy, but only guaranteed for an infinite amount of time.

By update rule, we mean the operation that modify the parameters of the Q or V functions from a single or a list of transitition of the form $s_t, a_t, r_t, s_{t+1}, a_{t+1}$ (now you understand the SARSA acronym). In a tabular RL, it change directly the value of the cell in the table, but when used with neural networks, it assigns target for gradient descent.

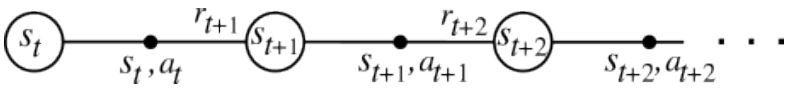


Figure 2.5: SARSA

All the update rules we present here are based on temporal difference learning.

Temporal difference learning

Temporal difference learning is a method of reinforcement learning that makes the observation that two predictions at time t and $t+1$ should differ only by the reward obtained in-between (but you must take account of the discount factor). The difference is called the temporal difference error.

For example, for the SARSA update rule, the TD error is:

$$\underbrace{r_t + \gamma Q(s_{t+1}, a_{t+1})}_{target} - \underbrace{Q(s_t, a_t)}_{current}$$

SARSA

SARSA is the straightfoward application of Temporal difference:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[\underbrace{r_t + \gamma Q(s_{t+1}, a_{t+1})}_{target} - \underbrace{Q(s_t, a_t)}_{current} \right]$$

α is the learning rate: It scales the update and it is the most important parameter. We use 0.005.

TD-Lambda

TD-lambda[1] is an update rule for Value functions that uses the fact that instead of using only the temporal difference error with the next state, you can use the TD error of all the following (or preceding, both views are equivalent) states with an exponentially decreasing weight of λ.

The reason for this is that you can always reformulate the Value function with an horizon of n step instead of only 1:

$$\begin{aligned} V^{\pi}(s) &= E_{\pi}\{r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots \mid s_t = s\} \\ &= E_{\pi}\{r_t + \gamma V^{\pi}(s_{t+1}) \mid s_t = s\} \\ &= E_{\pi}\{r_t + \gamma r_{t+1} + \ldots + \gamma^{n-1} r_{t+n-1} + \gamma^n V^{\pi}(s_{t+n}) \mid s_t = s\} \end{aligned}$$

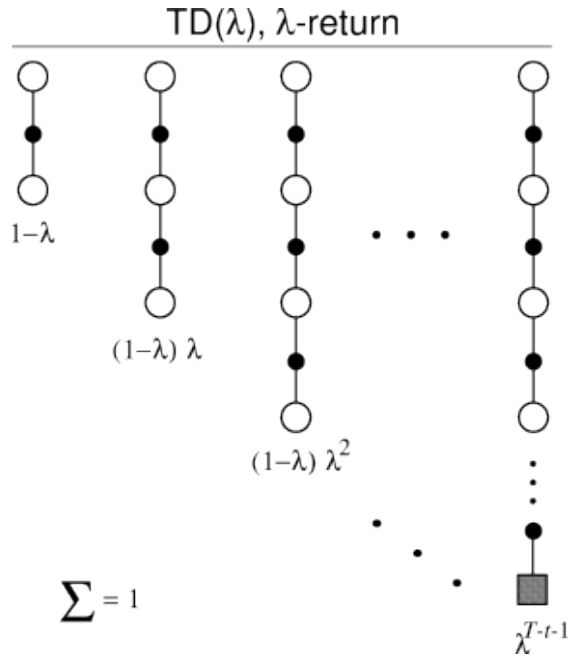


Figure 2.6: TD-lambda

$$V(s_t) \leftarrow V(s_t) + \alpha(1-\lambda)\sum_{i=t}^{N} \lambda^{i-t}\left(r_i + \gamma V(s_{i+1}) - V(s_i)\right)$$

As you can see, TD-λ is theoretically equivalent but less "short-sighted" than the original update rule since it takes into account future temporal-difference. It results in faster convergence for most case.

Q-learning

Q-learning is an update rule for Q functions. If you are interested by control rather than prediction, Q-learning is shown empirically[1] to converge faster than the SARSA update rule:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

The researchers at DeepMind have found 3 significant improvements to the raw q-learning.

- Experience replay[2,3]: As described below in Policy iteration, the transition are normally extracted from the last episode. The modification done here is that transitions are stored in a memory D and then randomly sampled from D. It helps avoid overfitting since a part of the transitions will not directly be from the last episode. It also stabilize the learning. This is loosely inspired by the brain, and in particular the way it syncs memory traces in the hippocampus with the cortex. This improvement has improved significantly the results on atari 2600[2]
- Double DQN and target network [4]: every X step of training, a clone of the current network is stored such that the function represented by this clone is Q' and the update rule become:
- Clamping TD Error[2]: If the TD Error exceed some threshold, it is clamped to that treshold.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma Q'(s_{t+1}, \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right]$$

It improves and stabilize the learning (since TD-error is done on a clone that change less immediatly).

When Q-learning uses a neural network, it is called DQN (Deep Q Networks).

Policy iteration

We will focus here on *Policy iteration* that enable episodic learning following the given procedure:

1. Randomly initialize a Neural network to approximate the Q or V function.
2. Sample a full episode following a policy $P_1(Q)$ or $P_1(V)$ that use at least partially the current Q or V function in an optimistic and greedy manner.
3. Update the Q or V function from extracted transition of the last sample using an update rule.
4. Sample a full episode following the completely greedy policy $P_2(Q)$ or $P_2(V)$ to observe the current score.
5. Go to step 2 unless convergence is observed or too much time elapsed.

Note: P_1 is the policy used to explore and learn in the most efficient manner the environnment. P_2 is the policy that from the gathered information attempt to be as close as possible to the optimal policy.

As we see, the fascinating thing about reinforcement learning is that the learning impacts the directed exploration of the environment.

On step 1. The random initialization of parameters the neural network serves as a random initialization of the Q or V function. On step 2. We stay vague about the exact policy because using the right policy enables us to learn more information about the environment and converge faster. Optimistic here means that we use that current V or Q function as the best current estimator of Q^* and V^*. Greedy means that we must exploit sufficiently often the best candidates that P_2 would return, so that it will converge.

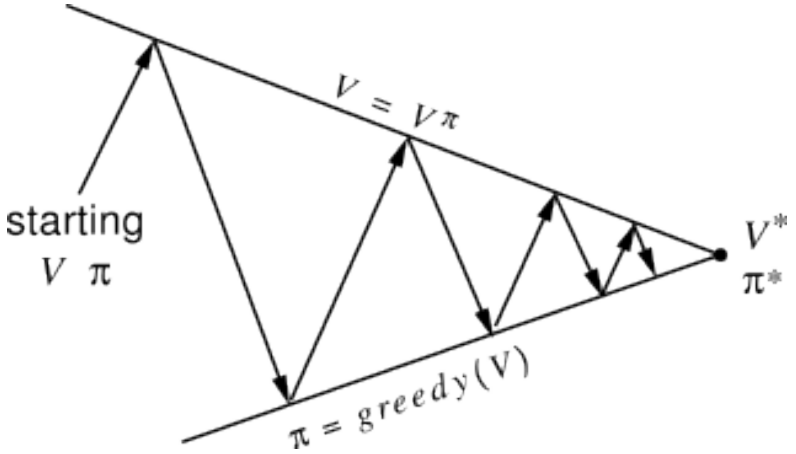


Figure 2.7: Convergence symbiose

As we see on the figure above, the policy and value function convergence depend of each other. An increase in correctness of one imply an increase on the correctness of the other.

Self-play

If you can assume that at each state of an adversarial game, some moves are superior to others independently of the adversary, then a strong strategy that can lead to high level of play is to play as if the opponent was perfect. It is not necessary optimal since that if you know that the adversary makes mistakes, some moves to "bait" him are more likely to win.

Nevertheless, at high level of plays, your adversary does not make mistakes. Self-play enables to incrementally improve the training by playing against an opponent (yourself) that plays incrementally better moves. To bootstrap the self-play and accelerate the

training, you can train on a corpus of expert moves to reach a good level to start on. This is the approach taken by Deepmind for their AlphaGo agent.

TD-gammon, Giraffe and AlphaGo

One of the first success of deep reinforcement learning is TD-gammon[1], Tesauro's application of reinforcement learning to Backgammon. The agent required very little backgammon knowledge, yet learned to play extremely well, near the level of the world's strongest grandmasters. The learning algorithm in TD-Gammon was a straightforward combination of the TD(λ) algorithm and a nonlinear function approximator using a multilayer neural network trained by backpropagating TD errors. It is believed that because of the high stochacity of Backgammon, it was a perfect application for reinforcement learning of a value function. Indeed, it mades the value function very smooth (no high jump between similar states). This is because since the boards are not deterministic after one move, the value is an expectation which is an average over probabilities. This average as a smoothing effect.

The training was not episodic but continuous with self-play: After each transition, an update of the neural network was made. When the self-play reached an end-state, it was simply resetted to a starting position. There was no reward except at the end state: +1 for a win, -1 for a loss. That made the value function an estimator of the probability of victory from each position. This pattern is similar for every adversarial zero-sum games where only win or loss matters. By contrast, in Abalone, the number of balls lost matters too and are used as negative rewards.

More recently, Matthew Lai was able to achieve similar results with chess and his agent, Giraffe[6], trained by deep reinforcement learning from a very raw representation of the board only. This was very impressive in the sense that it was able to reach a similar elo (a measure of the agents strength) with the best chess agents (like GNU Chess) that had been fine-tuned for decades with expert knowledge.

And lastly, a team at DeepMind developped AlphaGo[7], a Go's agent that was able to beat the World champion. This challenge was believed to be at least a decade away from AI because of the high branching factor, far higher than the one of chess. They used deep reinforcement learning with convolutionnal layers. Convolutionnal layers are very often used to treat images with neural networks because they are able to extract recurrent patterns in smaller patches of images. Here, the patches are smaller region of the Go's board. Contrary to the two others, and like all project of DeepMind, they used q-learning.

Exploitation and exploration dilemma

The dilemma is between choosing what you know and getting something close to what you expect ('exploitation') and choosing something you are not sure about and possibly learning more ('exploration'). A branch is the tree of possible path emerging from the choice of an action on a state. A path is a list of action going from one state to an end-state. Having more exploitation enables exploring more deeply the branches that are currently the most promisings. But not enough exploration, then you might ignore better paths that are completely different from the current best one.

Furthermore, some branches are not better than the current best branches but they are very different from the states the neural network has previously visited. Those states are important to explore because they contain a lot of informations to learn from. A bad path is still relevant, even just to learn that it is bad. It is information that might be very important in more delicate cases later on. Indeed, Diversifying the input is crucial for the ability of a neural network to generalize. Not visiting them could be detrimental for the learning.

Can you avoid completely avoid exploration ? [6,7] One way to avoid exploration is to have a dataset of starting position. Then you can apply self-play from those positions. Those starting positions are not the starting positions of the MDP. They are positions that would be difficult to reach by simple self-play. With a good (big and diversified) enough dataset, you will be able to learn enough informations to play optimally without further exploration. AlphaGo[7] and Giraffe[6] do this from a dataset of games of experts. AlphaGo does not start with a blank random neural network but initialize it with supervised learning on a dataset of millions of expert moves.

Another way you can avoid exploration is by solving a MDP that has enough stochastic elements that have the effect to make the state-space smooth and force the exploration with a greedy policy. It is similar to a forced ϵ-greedy strategy

Epsilon greedy policy

An epsilon greedy policy is a policy that is greedy (choose the best estimated action) with probability $1 - \epsilon$ and a random action with probability ϵ. This enable the agent to have a minimal exploration. Usually, epsilon is decreased with time from a high value (can be as high as 1), to a minimal value (commonly 0.01) because you want to decrease the exploration with time and explore more profoundly the most promising branches.

```
algorithm eps-greedy:
    input: List of [Action, List[Reward, Next_State, Odd]] l, current state s,
    current episode t
    output: Action
    thresh = max(0.01, 1/f(t)) //f is any increasing positive function
```

```
if (randomFloat() < thresh)
    return uniform_pick(l)
else
    // if Q function
    return arg max action [Q(s, action)] from l
    //if V function
    return arg max action  [expectation(reward + V(next_state) wrt odd)] from l
```

Note: As you can see here, the V function is more complicated to evaluate during exploration. It is slower compared to Q because you have to evaluate every possible next_state

3 Application of deep reinforcement learning to the game 2048

The game 2048

The popularity of the game 2048 might be explainable by its simplicity and addictivity. 2048 is played on a 4×4 grid, with numbered tiles that slide smoothly when a player moves them in the different 4 directions. Every turn, a new tile will randomly appear in an empty spot on the board with a value of either 2 or 4. Tiles slide as far as possible in the chosen direction until they are stopped by either another tile or the edge of the grid. If two tiles of the same number collide while moving, they will merge into a tile with the total value of the two tiles that collided. The resulting tile cannot merge with another tile again in the same move. The player's score starts at zero, and is incremented whenever two tiles combine, by the value of the new tile.

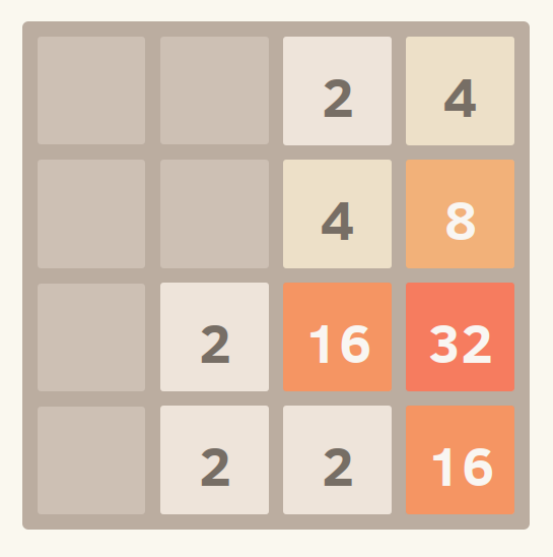


Figure 3.1: A 2048 board

Note that the colors of the tiles above are purely aesthetic and change with the score of the tile. The game is won when a tile with a value of 2048 appears on the board, hence the name of the game. After reaching the 2048 tile, players can continue to play (beyond

the 2048 tile) to reach higher scores. When the player has no legal moves (there are no empty spaces and no adjacent tiles with the same value), the game ends.

Minimax and expectimax

In theory, the optimal strategy for those stateful zero-sum games is the minimax algorithm. The minimal algorithm is a tree search of all the possible plays from both players and find move that start the branch with the best returns. Of course the opponent will try to play as best as he can. Thus, at each level of the tree, you want to branch to the state that maximize your score, while the opponent will branch to the state that minimize it. The alternance between minimizing and maximizing at each level explains its name. For board games that have only win and loss possibilities (like chess), only the end states are evaluable with -1 for a loss and +1 for a victory.

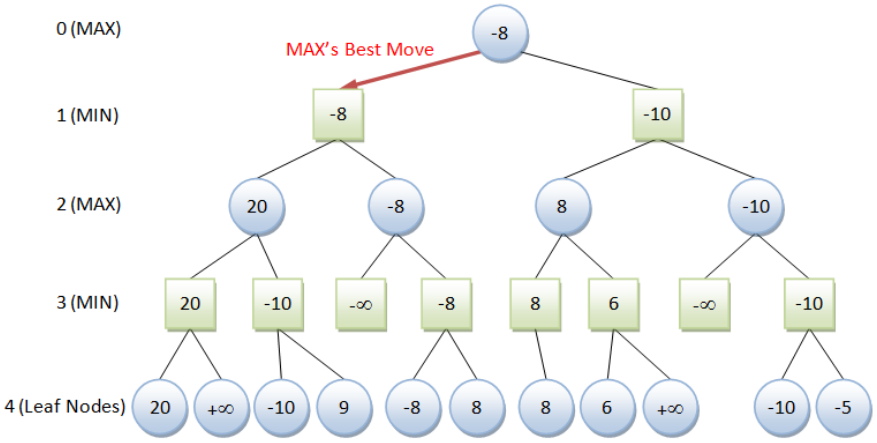


Figure 3.2: Minimax

Since it is a one player game and you only want to maximize your score, you can use a variant of minimax which is expectimax. Expectimax will treat the "environment turn" by maximizing the expectation of a branch by averaging all the next possible branch according to their probabilites. In the case of 2048, every possible branchs are equiprobable so you can simply calculate the average.

For games with high branching factors, it become untractable to build the full tree until the end of the game. To be able to still implement minimax, there is a pre-determined depth at which the states are evalued by a leaf function that approximate $V\pi$. If this function is an heuristic set by hand, those values reflect the relative qualities of the states and are set by hand. For board game like 2048 or 6561, the value of the state is not necessarily the value of a board (sum of the tiles). In fact, it is strongly inefficient to use the value of the board, as it makes the agent act too greedily and short-sighted.

Nevertheless, the rewards for reinforcement learning can be the difference of value of the boards. It is the same reward system than [8] that was able to beat all records on 2048 but using n-tuple networks.

In the case of deep reinforcement learning, we can use our V approximator in a minimax algorithm.

Leaf evaluation

The three main strategies for 2048 to use together is to:

- Have large values on the edges, because they are the one you combine last
- Keep as much empty squares (2x2, 3x3) as possible, because you should keep the free tiles in a certain way that maximize that number of squars.
- Having tiles in monotonic order (increasing and decreasing) in rows and columns. Having them in monotonic order enable you to combine them one after the other with the same direction.

The heuristic set by hand is just a weighted average of those features.

We could use those features as input of our neural network, but we aim to make the neural network learn from scratch. The neural network can be very powerful in the sense that it is not limited to a weighted average of heuristics that he would learn and can combine them in a more complex manner. It is also more specific since he can learn patterns that are only present in rare cases.

Learn the heuristic

To get a sense of the strength of neural networks, we first built a neural network that would be able to learn the heuristic by feeding it boards with their heuristic value as target. We were able to achieve very precise results after 100 000 random examples. We could replace the heuristic by the output of the neural network without any significant difference in the performance of the agent.

Results of TD-Lambda vs Q-learning on 2048

Then we applied reinforcement learning with the given encoding:

12 grid of 4x4 as a list of 16 bits for each possible values of the tiles. We set reasonnably 12 as the maximum power of 2 reachable so there were 12 grids, thus a string of $12 * 16$ bits. We set to 1 a bit corresponding to a tile in a grid if a tile of that corresponding

position is present and of the same value than the grid. It does seem like a wasteful encoding but it follows the One-hot encoding pattern. It showed to be efficient in practice.

We do not use minimax, and only directly use the policy:

$$argmaxaR(s,a) + V(s')$$

The curves obtained here are done through averaging of multiple seeds in order to be statistically significant:

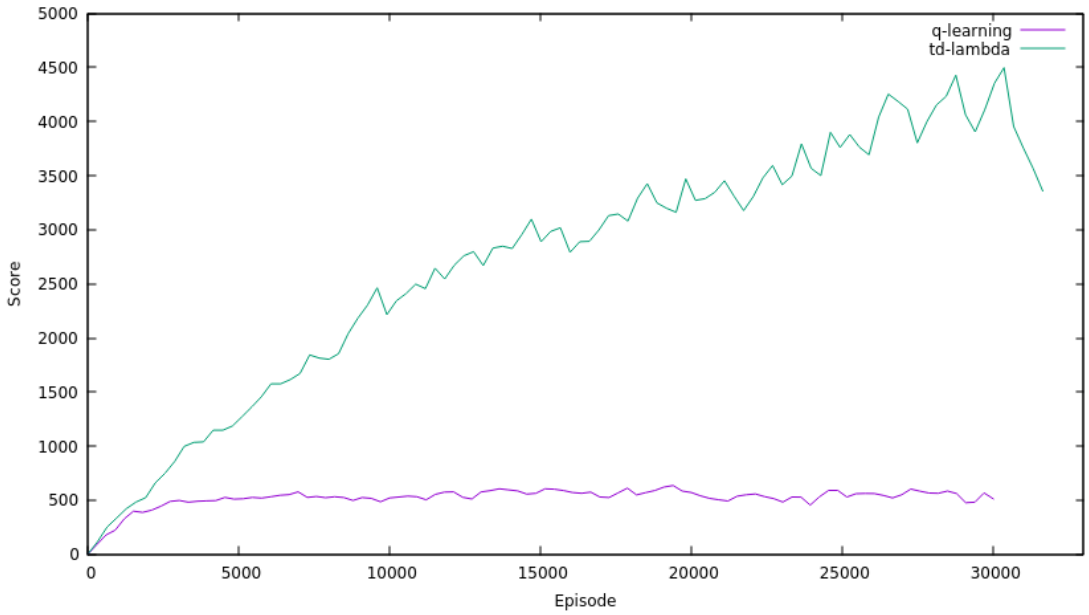


Figure 3.3: Q-learning vs TD-Lambda 2048

For 2048, TD-lambda seems to be much more efficent than q-learning. Maybe it can be explained by the stateful nature of 2048. All the information necessary should be contained in the different configurations of the board.

The results are far from an optimal policy results, which could reach a score around 2M in theory. But the increase in score are exponential with the number of moves. TD-Lambda achieve a success rate for getting the tile of 2048 after convergence of about 26%.

It can be hard to define a human benchmark, but we were not able to reach the tile of 2048 ourselves by hand even after a bit of training and expert knowledge.

Does the performance of TD-lambda hold for 6561 ?

4 Application of deep reinforcement learning to the game 6561

The game 6561

The game 6561 was developped for the AI codecup competition. The game 6561 is a collaborative variant of 2048 but it is played successively by 2 players. It ends like 2048 when no move could change the board of the game or, at the difference of 2048, when the number of turn reach the limit of 1000. Each tile of the grid can be empty or filled with a piece of one of the three available colors Red, Blue, Gray and with a value assigned to it that is one of the power of 3 starting at 1. Each turn the other player make a move. There is 5 distinct phases that alternate one after the other until the game end:

1. The current player place a blue piece of value 1
2. The current player place a red piece of value 1
3. The current player place a gray piece of value 1
4. and 5. The current player turn the board in one of the 4 axis direction: Right, Left, Up, Down.

(current player alternates each turn)

When the board is turned, each tile goes as far as it can in the direction of the turn. If a tile meet another tile of the same value in that direction, they combine according to the following rules: - If they are of the same color, they create a new piece of 3 times the value of the 2 original pieces - If they are of different color, they cancel each other.

The goal of the game is to achieve the maximum score throughout the game. The score is the sum of the value of the pieces on the board.

Strategy

In a collaborative game like 6561, it doesn't make sense to minimize the score since the opponent will also attempt to maximize the common score. Thus, there is two usable reasonnable variant of minimax in that case: "maximax" and expectimax.

Maximax will simply assume that the opponent will choose the branch which maximize the score. You can observe that by applying maximax, the algorithm shift to a planning problem. The goal is equivalent to finding the best state in a one player setting.

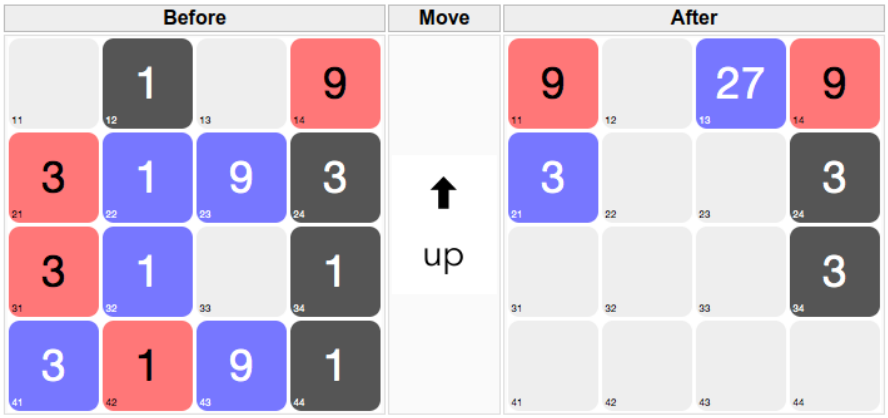


Figure 4.1: A 6561 transition

Expectimax assume that there is a probability measure such that the opponent has a probability p_i to choose each branch. In expectimax, you choose the branch the maximize the expectation of the branch with respect to that probability measure.

When using expectimax, you need to know what type of opponent you play against so you can calculate the right probabilities. The best agent of the codecup competition simply assume that each branch is as likely. It's a poor assumption against high skilled players but it's a very reasonnable assumption against poor players. In a setting of a competition like codecup, where there is many different profile of players, it's efficient. Furthermore, a branch that is strong even if the next move is played randomly can mean that this branch is inherently good. It's a property exploited by monte-carlo search tree, where all the moves of each branch are played randomly. In games where Monte-Carlo nethods are efficients, you can conclude that expectimax with equal probabilities given to each branch leads to effective plays.

The heuristic used in the best current agent try to maximize the heuristic of one color, same heuristics than 2048, giving a big bonus to all the heuristic of the maximum current color and a negative value to the other color's heuristics. Those heuristics include sum, monotonicity (order of the tiles in a row), possible merges and potential cancellations. The effect of this strategy is that it will combine as much as possible one color, and clear the board of the other color.

With short horizons, it's inefficient as you get better score by keeping all three colors on the board. With long horizons, it's very efficient as you can only manage to keep one color with high-valued tile on the board.

Results of TD-lambda and Q-learning on 6561

This is modified version of 6561 with an horizon of 100 turns intead of a 1000. The issue with an horizon of a 1000 is that the optimal strategy is to focus on only one color and is not as interesting as having to manage 3 colors. It requires also a longer training.

The encoding is the same than 2048 but we require 3 times more grid, for each color.

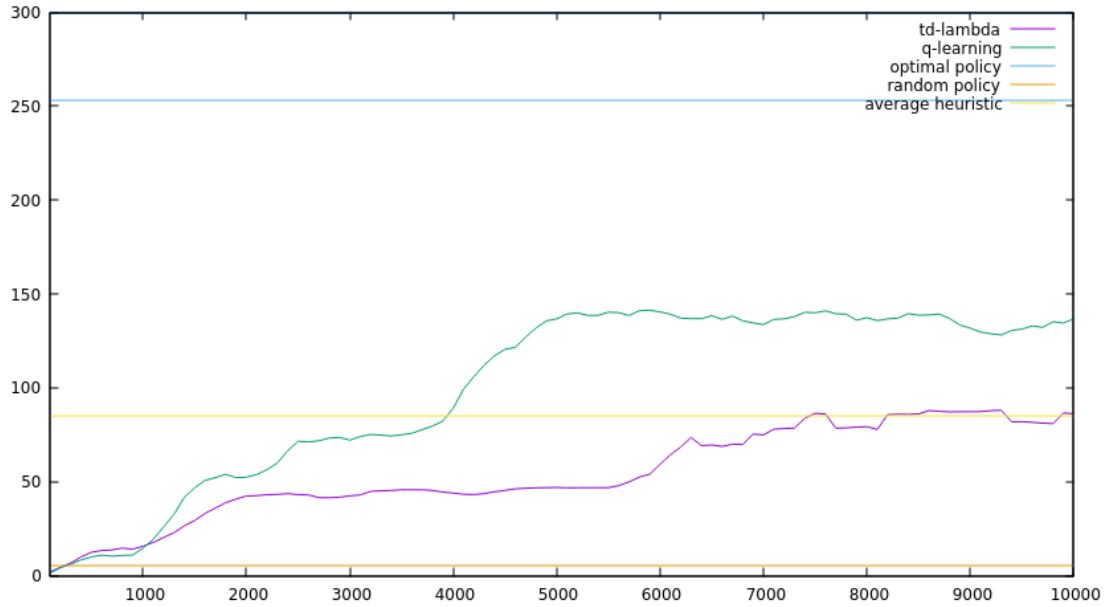


Figure 4.2: Q-learning vs TD-Lambda 6561

6561 is a hard deterministic problem in the sense that from testing, a single bad move is often hard to recover and will make high scores impossible to reach. This was tested by applying one single random move with different policy to see how hard it impacted their score. From this, we can conclude that it is quite impressive that agents learn to play to moderate or high-level. It means that they understand and apply one underlying pattern accross all move.

We can see that both agents improve drastically from a simple random agent and even beat the heuristic. Note that the heuristic is originally intended for an horizon of a 1000 and the comparison is not completly “fair”.

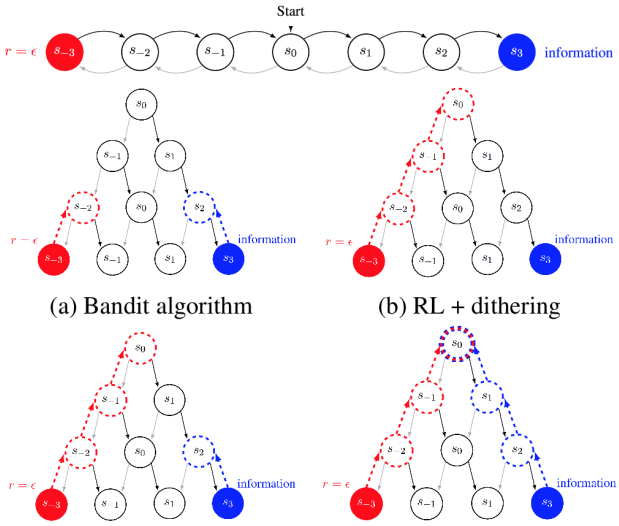
Contrary to 2048, for 6561, q-learning seems to be more efficient. It was surprising considering that the q-function had an output of size 52 (one for each possible action: turns + placement of a tile of a given color) and seemed to be “hard” to learn.

5 Deep exploration with bootstrapping DQN

Deep exploration

Information should be seeked during exploration. Information that is the most relevant is information you lack, or as defined by the information entropy equation, the quantity of surprise, or novelty. The problem with ϵ-greedy is that it is "dithering"[9]. *epsilon*-greedy is used by most agents of reinforcement learning but leaves much room to exploration. Indeed, it is not enough that the exploration is directed toward information or reward, it must also be deep. By deep exploration we define exploration that is directed over multiple time steps; it can also be called "planning to learn" or "far-sighted" exploration.

For exploitation, this means that an efficient agent must consider the future rewards over several time steps and not simply the myopic rewards. In exactly the same way, efficient exploration may require taking actions which are neither immediately rewarding, nor immediately informative.



(a) Bandit algorithm (b) RL + dithering

(c) RL + shallow exploration (d) RL + deep exploration

Figure 5.1: Deep exploration

This depicts the planning and look ahead trees for several algorithmic approaches in this example MDP. The action "left" is gray, the action "right" is black. Rewarding states are depicted as red, informative states as blue. Dashed lines indicate that the agent can plan ahead for either rewards or information. Unlike bandit algorithms, an RL agent can plan to exploit future rewards. Only an RL agent with deep exploration can plan to learn. The Bandit algorithm is algorithms developped to solve the MAB. In the context of MAB, there is no need for planning since there every action has a direct result and do not need planning.

Thus, the improvement of deep exploration is that it will seek informations that are located multiple state ahead. A dithering exploration like ϵ-greedy will only be efficient if it doesn't have to seek for information..

The chain MDP

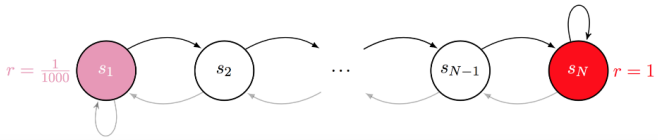


Figure 5.2: The chain MDP

It starts at s_1 and end after N+9 transitions have been realized. The last state on the right is s_N. From each state, you can choose to go Right, on the s_{t+1} state or Left on the s_{t-1} state. On the left-most state, s_1, going Left keeps you at the same state, but rewards you $r = 1/1000$. On the right-most state, s_N, going Left keeps you at the same state, but rewards you $r = 1$. The optimal path is to always go right, and get the sum of reward 10.

The difficulty is that an ϵ-greedy will have to do 2^n episode before finding the optimal path. Indeed, it is similar to a random walk. As N grow big, it becomes clear that it is a very inefficient exploration. Being able to "solve" in a reasonnabe amount of steps the MDP is a necessary condition of existence of deep exploration.

Bootstrapping DQN

The main idea behind bootstrapping DQN is to use K neural networks instead of one. One of those K neural network is randomly picked to do a conventionnal ϵ-greedy exploration. The transitions from that episode are stored in a memory D (experience replay). Then the transitions are randomly sampled from that memory D (it is an experience replay) and are randomly sent, following a random distribution X, for learning to each of the K

neural networks. We use a binomial distribution ($p = 0.5$) meaning that each transition sampled from the memory D has a probability p of being used by each head. As the number of transition grow big, the divergence grow between each head.

```
algorithm explore-bootstrapping-dqn:
    input: List of [Action, Reward, Next State] l_ars, current state s,
    current episode t, list of Head l_h
    output: Action
    head = uniform_pick l_h
    return head.epsilon_greedy(l_ars, s, t)


algorithm learn-bootstrapping-dqn:
    input: list of Transitions l_t, list of Head l_h, distribution x
    for head in l_h:
        if x.nextBoolean:
            head.learn(l_t)
```

Instead of using K separate neural network, it is possible to share the first C layers then only have K different heads. For Atari 2600 games, it is very useful because the first C layers should be about transforming the raw pixels into a more abstracted form. This should be common for all head.

The name come from bootstrapping in statistic. Indeed, each head is similar to an estimator trained on a different dataset $\tilde{D}$ that is of the same cardinality than the original D. It is similar to resampling.

The intuition into why it is working is that each head has a different random initalisation that is kept diverging because of their different learning. Those different heads have different "belief" about the environment and will choose different paths. At least one of the head should have a positive bias for a given uncertainty. In the chain MDP, there only need one head that assign a stronger value to the uncertainty of going right, than the fixed reward of going left. The more head there is, the more likely that one head will explore unknown paths. For the atari games, the authors of [9] have shown that it accelerated the learning process while having minimal overhead.

Results

We were able to reproduce and get comparable results than [9] with K=10 and the chain MDP of every length until 100.

We also compared with the q-learning of 6561:

Surprisingly, We can see that in this case it is actually detrimental to use bootstrapping dqn. There might be multiple factors that can explain a negative effect. The first one

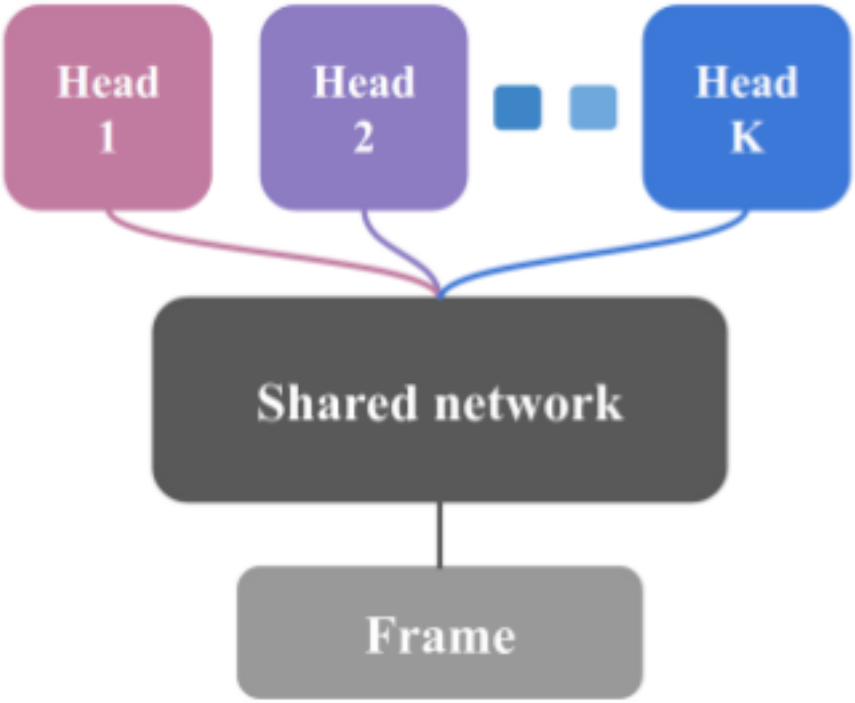


Figure 5.3: Archtecture for K heads

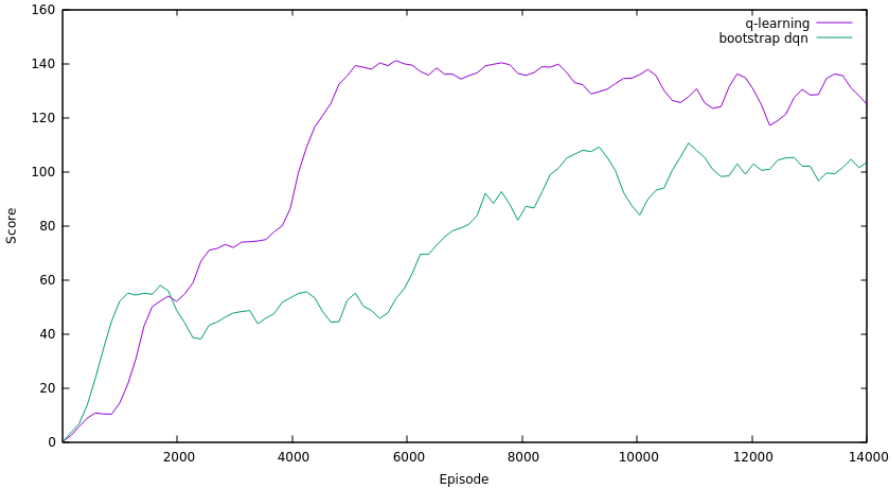


Figure 5.4: bootstrap dqn

is that 6561 MDP is much more complex than the chain MDP. Indeed, in that MDP it is sufficient that one head to be biased to go right to solve the MDP efficiently. If you ignore the pixel recognition, 6561 might also be more complex than the games of the atari platform. It might be possible that bootstrapping dqn does not scale that well with complexity.

The other important issue is that the final policy is made by combining all the heads. It might be very possible that disagreements, solved by most voted action between heads, lead to an inefficient combined policy.

It could also be an implementation issue that did not reveal itself under the test of the chain MDP and other tests.

6 Incentivizing exploration by quantifying novelty with an auxiliary neural network

Novelty and information

To be able to quantify the number of information or novelty, one way is to quantify how unexpected the next state s_{t+1} is for a given pair (s_t, a_t). You could use a tabular memory to count how often was a state visited but it will not be fully relevant: Some states are very close from previously explored states. Even though they have never been explored, they contain little information. You would want to attribute them less incentive as they do not contain very much information.

The idea behind [10] is to calculate the euclidean distance between the prediction of an auxilliary neural network of the next state s_{t+1} given the pair (s_t, a_t). This distance is used as a bonus in the greedy search of the next state to explore. It is used in conjunction with a q-learning neural network. It does not necessarily mean that this state is very informative for the q-learning neural network. But since both neural networks have learned from the same transitions, a surprising transition for the latter one should be a surprising transition for the former one.

Quantifying the error of prediction for the full state s_{t+1} which is of very high dimensions (as many as the number of pixel for the training on atari 2600 games) gives very unstable and inaccurate results. To be able to focus on the relevant part of the state, a dimension reduction algorithm must be applied to each state. We define $r(s)$ the function that reduce the dimension of a state. The prediction neural network is now given $(r(s_t), a_t)$ and must predict the state $r(s_{t+1})$.

$$\text{Novelty}(s_t, a_t, s_{t+1}) = \|r(s_{t+1}) - p(r(s_{t+1}), a_t)\|$$

where p is the prediction of the neural network.

To achieve a reduction of dimension, we can use another neural network, an autoencoder.

```
algorithm explore:
    input: List of [Action, Reward, Next State] l, current state s,
    current episode t
    output: Action
    arg max action [Q(s, action) + (1/f(t))*Novelty(s, action, next_state)] from l
    //f is any increasing positive function
```

Autoencoder

An entoencoder (AE) is a neural network that has for input and target the same data. The layers are decreasing in size then re-increasing in size. It forces the neural network to be able to find an encoding of the data that is of the dimension of the smallest layer. Once trained, it is possible to use directly the output of the $N - i$ layer, instead of the output layer (the N-th layer). By varying i, it is possible to vary the dimension reduction achieved.

To train an autoencoder, it is best to train him over the same diversity of input that it will face later. To achieve this[10], it is possible to use a "static AE" that is trained once and for all by exploring the MDP randomly before the learning start. Or it is possible to use a "dynamic AE" that train himself in parrallel of the learning with the same input.

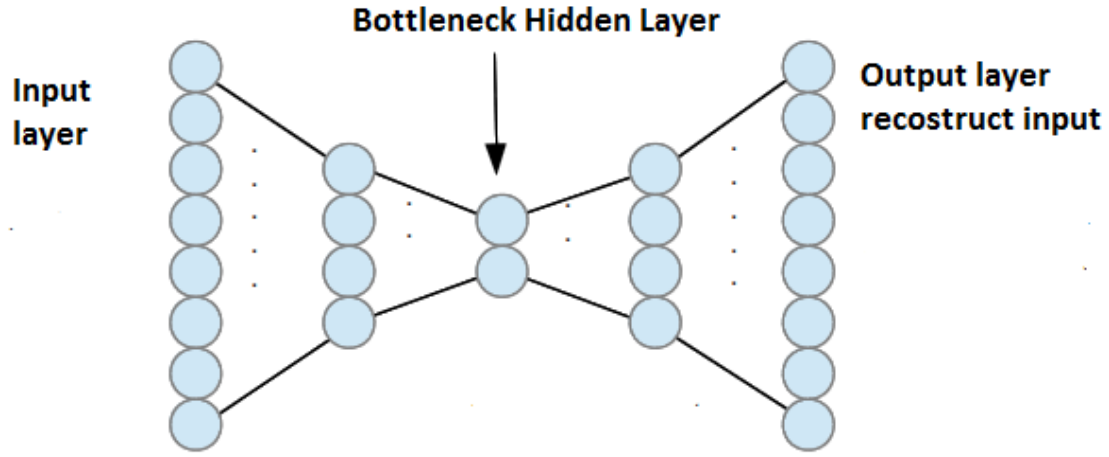


Figure 6.1: autoencoder

New approach with TD-lambda

TD-λ has different performances than Q-learning. In some case, it is more efficient. We adapted this novelty incentivising to TD-λ. Thus, our approach has benefits wherever td-λ is more efficient than q-learning. The adaptation is very similar with a bonus given during the exploration to novel transitions. This bonus is scaled by a factor β, another hyper-parameter that is specific to each MDP.

```
algorithm explore:
    input: List of [Action, [Reward, Next State, Odd]] l, current state s,
    current episode t, autoencoder ae
    output: Action
    arg max action [expectation(reward + V(s, action)
```

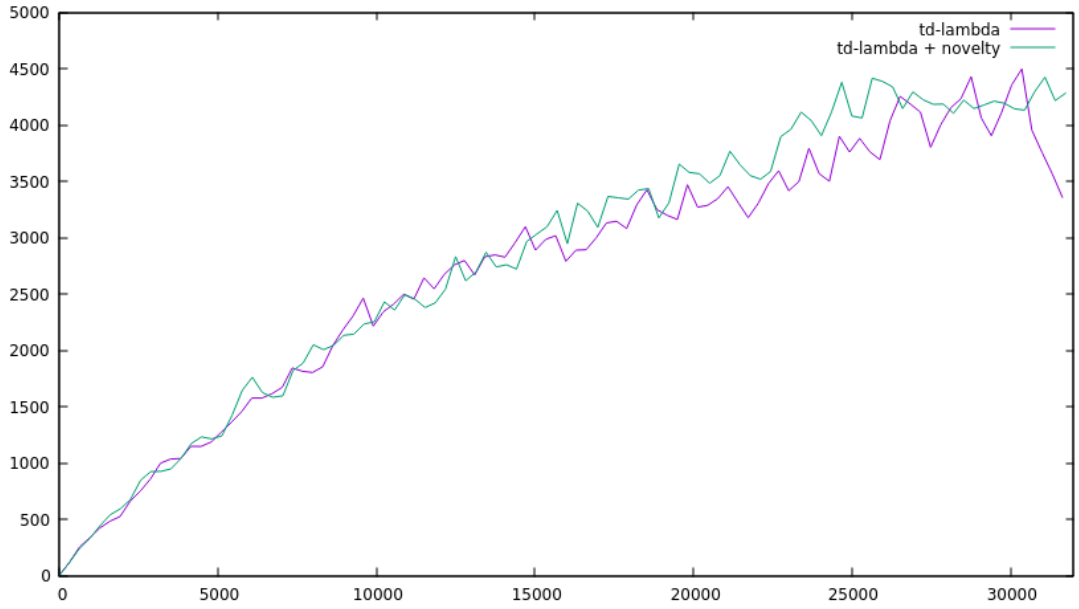
```
+ (1/f(t))*Novelty(ae(s), action, ae(next_state))) wrt to odd]
from 1 //f is any increasing positive function
```

Results

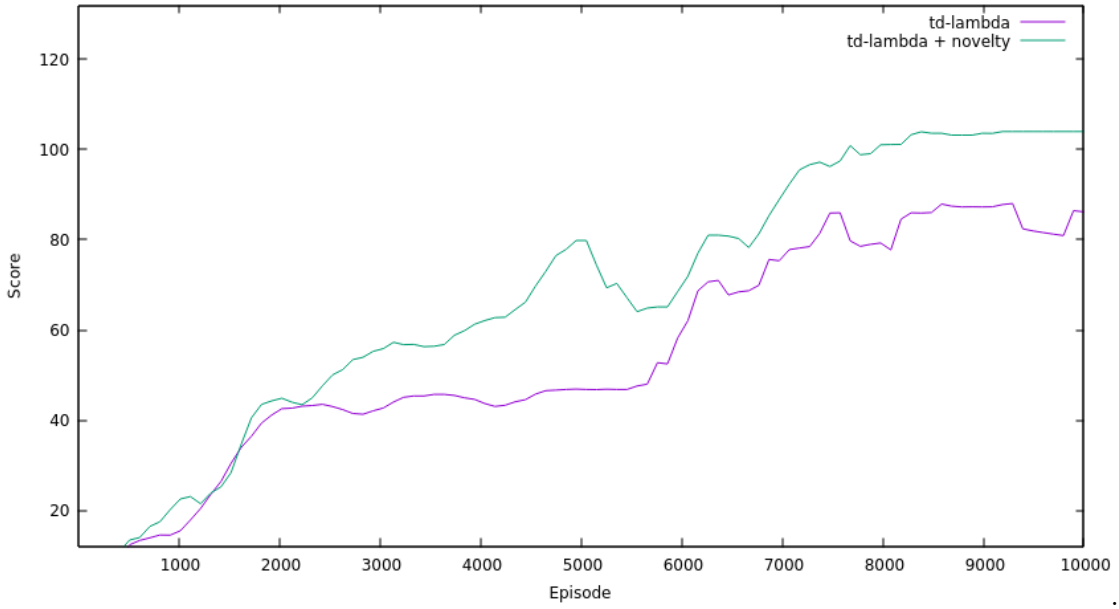
On the chain MDP with a length of N=20, this new approach was able to find the optimal path at 3132 episode (median of 3 seeds).

It is less efficient than the previous approach, bootstrapping DQN, on this particular MDP but it would have to be tested on multiple MDP to be significant. Without this bonus, and with only ϵ-greedy, the optimal path takes more than 50 000 episodes to be found.

Results on 2048:



This approach did not improve significantly the performance of td-lambda on 2048.
Results on 6561:



Nevertheless, it has increased the maximum score of convergence of $TD - \lambda$ on 6561.

One possible explanation is that deep exploration is not benefitial to 2048 because of the stochacity of the environment. Indeed, it is hard to plan when half of the moves are random. On the other hand, for 6561 since it is completely deterministic, it seems natural that deep exploration would benefit the learning because you can plan all moves and get determinist results.

7 Conclusion

We compared q-learning and td-lambda and have shown that their performances vary grandly with the MDP they are applied to. There does not seem to be any obvious pattern to predict which MDPs is better fit to which algorithm. Experimenting with various methods of learning and various learning rate seem to still be necessary.

As often said about neural networks, adjusting them seems to be more art than science. To be able to learn through reinforcement learning efficiently, there is multiple factors to consider. The right hyperparameters, the right encoding, the right rewards, the ability of the approximator to learn, here neural networks, but also the policy used to explore and intereact with the environment. Deep reinforcement learning has trouble finding optimal policy that require a high amount of planning when looking multiple step ahead for rewards and information. There exist efficient explorations for tabular RL, but they do not adapt well to neural networks. To answer this, one can use deep exploration. Deep exploration can increase the performance of an agent but it seems to vary between MDP and the method used to achieve deep exploration.

Nevertheless, even though ϵ-greedy is simple and improvable, better explorations like deep explorations do not necesarily translate to an increase in the efficiency of the learning. The reasons why can be hard to isolate. A pattern observable here is that deterministic MDP are more likely to benefit from the planning effect of deep exploration.

Not only neural networks are hard to use in proofs of convergence, because of the high uncertainty of their behavior, they are also very tough to debug. But neural networks are very promising tools, with a lot of margin to improve. Their use to solve all kind of problem has only begun. The future of AI and deep reinforcement learning seem tied.

Last but not least, I would like to thank my supervisors for the opportunity to do this very enriching project.

References

[1] Sutton, Richard, Barto, Andrew, Reinforcement Learning: An Introduction, MIT Press, 1998.

[2] Mnih, Volodymyr et al, Human-level control through deep reinforcement learning, Nature. 518(7540) (2015) 529–533.

[3] Schaul, Tom, Quand, John, Antonoglou, Ioannis, Silver, David, Prioritized experience replay, (2015). http://arxiv.org/abs/1511.05952.

[4] Van Hasselt, Hado, Guez, Arthur, Silver, David, Deep reinforcement learning with double q-learning, (2015). http://arxiv.org/abs/1602.04621.

[5] Tesauro, Gerald, Temporal difference learning and td-gammon, Communications of the ACM. 38(3) (1995) 58–68.

[6] Matthew Lai, Giraffe: Using Deep Reinforcement Learning to Play Chess, (2015). http://arxiv.org/abs/1509.01549.

[7] David Silver, Demis Hassabis et al., Mastering the game of Go with deep neural networks and tree search, Nature. 529 (2016) 484–489.

[8] Wojciech Jaśkowski, Mastering 2048 with Delayed Temporal Coherence Learning, Multi-State Weight Promotion, Redundant Encoding and Carousel Shaping, (2016). http://arxiv.org/abs/1604.05085.

[9] Ian Osband, Charles Blundell, Alexander Pritzel, Benjamin Van Roy, Deep Exploration via Bootstrapped DQN, (2016). http://arxiv.org/abs/1602.04621.

[10] Bradly C. Stadie, Sergey Levine, Pieter Abbeel, Incentivizing Exploration In Reinforcement Learning With Deep Predictive Models, (2015). http://arxiv.org/abs/1507.00814.