UCL Université catholique de Louvain

École polytechnique de Louvain (EPL)

epl

Deep Q-Learning for Robocode

Dissertation presented by
Baptiste DEGRYSE

for obtaining the Master's degree in
Civil Engineering

Supervisor(s)
John LEE

Reader(s)
Cédrick FAIRON, Thomas FRANÇOIS , Adrien BIBAL

Academic year 2016-2017

Abstract

Q-learning can be used to find an optimal action-selection policy for any given finite Markov Decision Process. The Q-network is a neural network that approximate the value of an action in vast state space. This work will study the deep Q-learning, a combination of the Q-learning and neural networks, and evaluate the impact of meta parameters. The applicative context of this work is the game Robocode and we evaluate the impact of different state representation, rewards, actions, neural network architectures in order to build an artificial intelligence. The hybrid architecture combining a deep feedforward neural network with two convolutional layers for the processing of the log of the states was successful. Adding a long short-term memory layer in parallel, and removing the random memory replay in order to have a sequential training was not suited. The resulting artificial intelligence outperformed the other public machine learning projects thanks to its simplicity of actions enabling learning complex behavior. This project can be used to gain intuition on an efficient state and action representation, on the importance of a complete reward function, as well as the advantages of an hybrid architecture to get the best of each network specificities.

Acknowledgements

I would like to thank John Lee, my supervisor who has been great all year long, even when nothing was working as expected. I am also grateful for the time that the jury will spend on my master's thesis. A very special thanks to JoAnn Lucas who corrected my approximate and self taught English grammar. I also really appreciated the review of Thomas François which made some precious comments. Finally, I would like to thank Blanche Jonas for the help on the master thesis requirements, and the chapter style.

Contents

1 | Introduction

Machine learning raises interest in many domains, for instance in big data analysis, autonomous cars and intuitive artificial intelligence. It allows building a complicated task that we could not implement by ourselves. Learning is carried out by instantiating a model from examples. But we must find many examples to learn a function. One of the machine learning algorithms that is very common in video games is the Q-learning (a particular formalization of reinforcement learning) which allows collecting the data needed to train our model by playing the game. Among the most exciting models are the neural networks, with their specific structures achieving state-of-the-art performance in facial recognition [25], language generation [22], question answering [34], sequential image generation [17].

But the problem with this model is that the internal behavior of the network is hard to interpret and the training can take some time if the meta-parameters are not well tuned. The tuning and the choice of the architecture has still an important intuitive part, which should be reduced as much as possible. Once we cannot reduce it anymore, we should at least learn to have an intuition using all the available information and find as easily as possible the reason of a strange behavior. One of the first competitive game Artificial Intelligence (AI) using a Q-network (neural network used to approximate a Q-function) is the TD-Gammon [41] which was made in 1994. The Q-function is an evaluation of the quality of an action given a state. This AI learned by playing against itself and achieved a master-level play only using the reinforcement learning algorithm. The second amazing article is the AI that learned to play some Atari games by only feeding the network with the pixels of the screen [29]. This intelligence is generic and did not need any modifications during the various training session on the other games. The AlphaGo player [36] uses more complex combinations of networks to master the game of Go, which was supposed to be at least a decade away. The challenging part of this board game is the huge search space and the lack of criteria for a quality assessment of a board state.

This master's thesis will explore the deep Q-learning algorithm for the game Robocode. This game simulates a tank battle, where each tank must be programmed to think by himself. The deep Q-learning is the algorithm used in the Atari article [29] which uses a neural network to estimate the Q-function trained by reinforcement learning in order to play a game. We will train an artificial intelligence for the Robocode game, and get familiar with the neural networks and the deep Q-learning algorithm. We will then study the impact of other network structures, meta-parameters and input/output choices. We will also evaluate the best actions estimated by the network, the Q-score and the score of the Robocode game, as well as an evaluation against the sample bots. The research will be on the state representation, action choices and network hybrid architecture. This thesis is not the first machine learning work on Robocode, but the previous works were not really exploiting all the machine learning tools. Work has been conducted by a group of students [15], combining genetic algorithm for some actions, a neural network for movement prediction and reinforcement learning for the actions choice. Unfortunately the bot does not perform well; other simpler implementations [37] achieve better results. Many videos about machine learning for Robocode are available on Youtube [38], [31], [8], [28], [9]. Robocode is not the only game that has been used for neural network research. A bot has been made for Mario using the NEAT algorithm [39] and has perfect results on a non-trivial world [7].

We will start with a description of the game Robocode and see why this game is interesting, then we will go through a brief technical overview of the concepts used in this thesis. After the state of the art, we will explore the methods that will be used for the evaluation, and discuss some of their particular combinations. We will finally evaluate each model, and the consequences of our choices. The AI coming out of this thesis probably outperformed all of the previous works using machine learning on Robocode that were made public. Superiority resulted from simplicity of the actions, giving our bot the opportunity to learn advanced behaviors.

2 | Robocode

Build the best, destroy the rest!

Robocode's moto

Since neural networks need vast quantities of data, and since we learn better while having fun, the Robocode game was completely appropriate for this thesis.

> "Robocode is a programming game where the goal is to code a robot battle tank to compete against other robots in a battle arena. So the name Robocode is a short for "Robot code". The player is the programmer of the robot, who will have no direct influence on the game. Instead, the player must write the AI of the robot telling it how to behave and react on events occurring in the battle arena. Battles are running in real-time and on-screen." [24]

Figure 2.1: Robocode's logo [3]

This game is also used in universities for the AI study [30]. It is also possible to run the battles without interface to boost the computation time.

2.1 Why Robocode

This game is appropriate for machine learning in many ways:

1. It is a fun way to work on AI. It has been shown that we learn more and faster if we are having fun. The master's thesis is a important part of the master, and should be taken seriously in terms of amount and level of work. If the option is available to have fun while attempting to master challenging concepts, this will be more effective than studying Q-learning in a formal environment.

2. We have access to infinite data since it is a programming game. We are not restricted to players performances, and there is a significant database of public competitive robots.

3. It is a well known software for which there are competitions, articles, a user community, previous work on machine learning, studies in AI.

4. Robocode has enough complexity to be meaningful.

Before discussing the difficulties of the game, we will first introduce vocabulary, necessary to ensure clarity. The game aims at controlling a little tank, and kill the opponent's one. The tank can go forward, backward, and rotate on himself. He has a *gun* and he can move it independently from the body, with 360° of freedom. The tank only sees enemies when using it's *radar*, which is also free to move independently from the gun and the body. The bot can shoot *bullets*, propelled in straight line from the gun's current position, and the speed of the bullet is inversely proportional to the bullet power. The damage dealt to the enemy by the bullet is proportional to its power. The shooter loses a fraction of life during the shooting command and gains more life if the bullet hits an opponent, still proportional to the bullet's power. In AI, an *environment* is the context that has the *agent* in order to take a decision.

The game has multiple difficulties:

- We do not see the enemy bullets, the environment is thus partially observable. This makes learning harder because we cannot anticipate everything in the next state since we do not have the complete current state.
- The vision is limited by the radar; the partial part of the state might not contain the same information depending of our actions.
- The targeting requires estimation, calculation and interpretation by the player to guess were the enemy will go because the bullets take time to travel.
- The time is limited each turn, so the evaluation must be fast otherwise the game does not let the robot execute an action this turn.
- The possible response actions are huge, since the tank can turn, aim, scan up to any angle smaller than a limit, and move up to any distance at the same time, making the space of actions hard to define.

The mechanisms of a game come from the number of possibilities to resolve difficulties. This is why we are interested by this game. We will now detail a little more the rules of the game.

2.2 Rules

The goal is to kill the opponent, and stay alive. But the score is not only dependent on the win, it also quantifies and counts the damage inflicted during the battle. A *round* is over when there is only one tank left on the battlefield. A *battle* is a group of rounds, and gives a better evaluation of the level difference between the two or more tanks. A *turn* is the time during which the robot has to find his next action. There is roughly between 300 and 700 turns in a round, depending on the quality of the fight.

The game has many rules, which can be found on the robowiki [2]. The principal rules are the following:

- The shooter loses a fraction of their life when shooting, proportional to the bullet power (the only available information to detect that the enemy has shot).
- The robot has a varying degree of gun heat, which increase depending on the size of the bullet. A robot cannot shoot if the gun heat is too high.
- The robot receives events when he hits an enemy, or get hit by a bullet, ...
- The robot goes slower while turning.

- There are no obstacles in the battlefield.

For this master's thesis, we will only work on the one versus one battles, in a squared battlefield of size 600 and the battles will be made of twenty-five rounds.

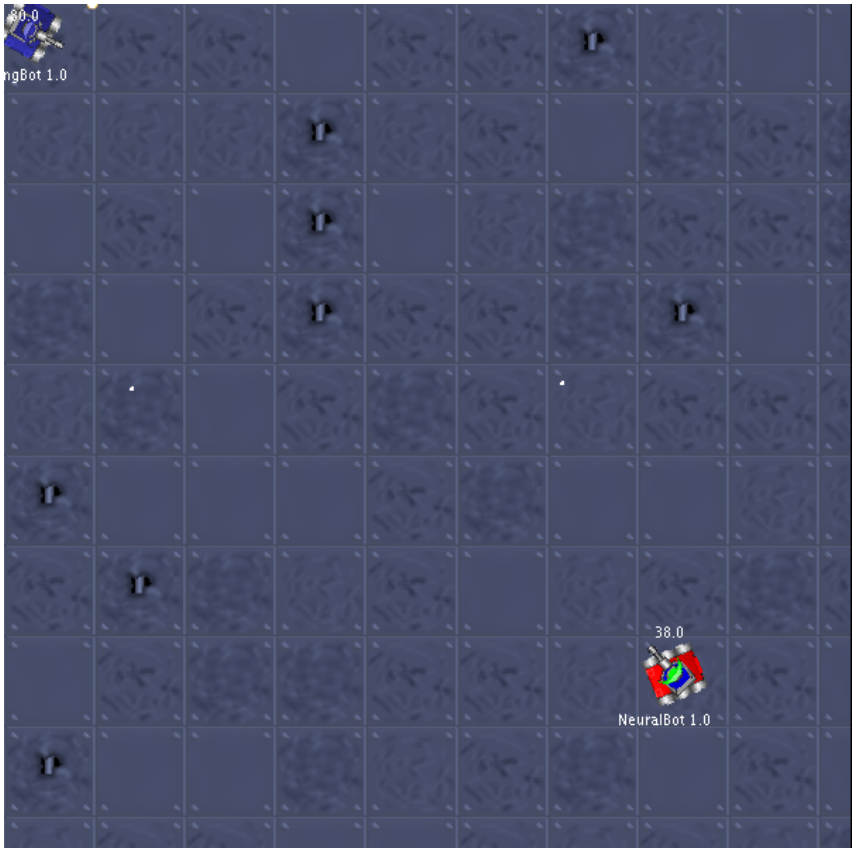


Figure 2.2: Snapshot of a robocode round: we can see that blue tank on the top left has shot a small white bullet around three squares on top of the red tank or neural bot. He also has shot a bullet under the blue tank.

3 | State of the Art

> Toute l'histoire du code vient de la Coda.
> Parce que code code codaa!!
>
> *Fabien Degryse - my father*

Deep learning is a hot topic in the scientific community since the TD-gammon success back in 1994 already [41]. The Atari games that are played by only feeding the pixels of the game frames into the network are another beautiful example [29], combining the image processing power of convolutional layers with the dense structure of the Q-network. But AlphaGo made it popular for the broad public, by defeating a master player in the game of Go [36]. We will start with a little reminder, then see some applications of the neural networks. Atari games and AlphaGo will follow, with the main ideas of the articles that we will reuse in this thesis. The final part of this state of the art will give an overview of the previous machine learning projects implemented for Robocode.

3.1 General reminder

In this thesis, we will use classical concepts, such as Markov Decision Process (MDP), or Q-learning. The reader who is familiar with these concepts may skip this section and proceed to the next section. We introduce here the MDP because it is used by the Q-learning algorithm.

3.1.1 Markov Decision Process (MDP)

This can be used to describe an environment [45], and the decision making process where

- S is a set of states (the states of the game)
- A is the set of possible actions (the possible actions for the robot)
- $P_a(s, s')$ is the probability of going from state s to state s' (in chess, it is always equal to 1)
- $R(s, s')$ is the reward for going from state s to state s' (often the difference between the score at state s and the score at state s')
- $\gamma \in [0, 1]$ is the discount factor, which represent the difference in importance between future rewards and present rewards. (If the future is badly evaluated, γ should be close to 0)

A *policy* is the agent's strategy to choose an action at each state. Symbol π denotes a policy. Our goal here will be to find the policy that will maximize the cumulative reward ($R = \sum_{t=0}^{n} \gamma^t r_{t+1}$).

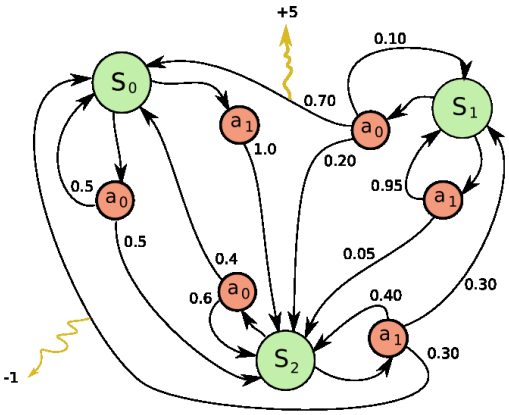


Figure 3.1: Example of Markov decision process with three states and two actions. The reward is given by the yellow arrow, and the numbers at the beginning of the arrows are the probability of taking this transition given that we have chosen the a_i.

3.1.2 Q-Learning

The Q-learning is a kind of reinforcement learning in which the algorithm tries to learn the Q-value associated to the $<$ state, action $>$ pair. There are multiple approaches to reinforcement learning (RL) [35]:

- Policy-based RL : search for the optimal policy π^*, maximizing the future reward.
- Value-based RL : estimate the optimal value function $Q^*(s, a)$, this is the maximum value achievable under any policy.
- Model-based RL : use a transition model to find the best policy.

In order to find the best policy, we need to learn the model on which we are playing. Unlike backgammon that has a model defined by statistics, Robocode is a little bit more complex to describe. The best approach for us is the Value-based RL since it let us improve without having to learn or build the model.

We will thus use a model free reinforcement learning strategy, which tries sometimes random new actions, and update the expected reward based on the past experiences. This is a very powerful and common way to explore the environment in game AI. It was used in the AlphaGo and in the Atari games articles.

3.1.3 Neural networks

There are multiple kinds of neural networks. The most popular are feedforward networks, convolutional neural networks and Long short-term memory networks. In order to understand how the networks process data, we will start by a quick tour around the activation functions.

Activation functions

The activation functions are used by the neurons inside the networks in order to approximate the target function. They are the building blocks that sum up to make the resulting function. The three main activation functions are the sigmoid (σ), rectified linear unit (ReLU) and hyperbolic tangent (tanh) (figure 3.2). The sigmoid and the tanh functions are appropriate for the LSTM network, or for flat network (maximum 2 hidden layers) but they suffer from the vanishing

gradient problem if the network is deeper. We will thus mainly use the ReLU function in this thesis.

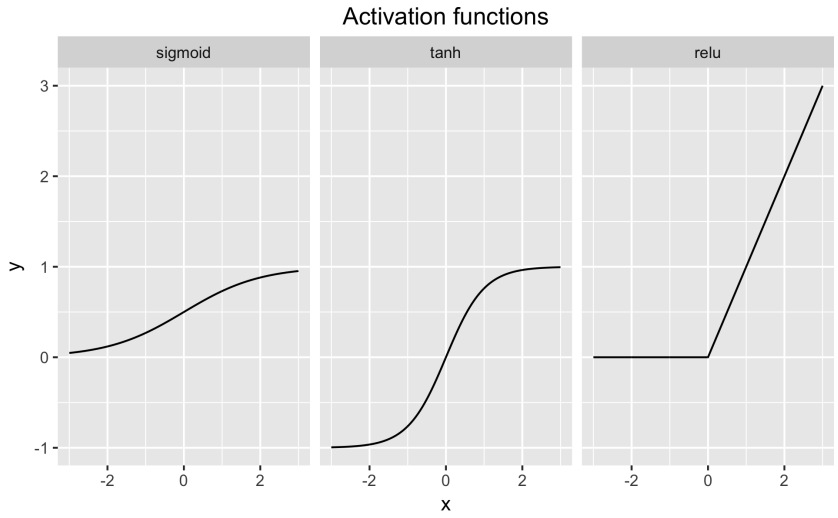


Figure 3.2: The three activation functions that we will use in this thesis. The sigmoid function has its image between 0 and 1, the tanh function is like the sigmoid, but between -1 and 1. The ReLU activation function is the only one that does not suffer from the vanishing gradient problem.

Feedforward neural network (FNN)

In very short, a FNN is a network without cycle [44] (figure 3.3). It is relevant for relatively small input dimension, and does not keep memory. It can have one or multiple hidden layers, with many hidden units. The layers are usually dense, which means that each unit of layer n has a link to every units of layer $n+1$. We will mostly use the rectified linear unit (ReLU) activation function, which avoids the vanishing gradient problem in deep neural network (many hidden layers). The more specific details will be discussed later.

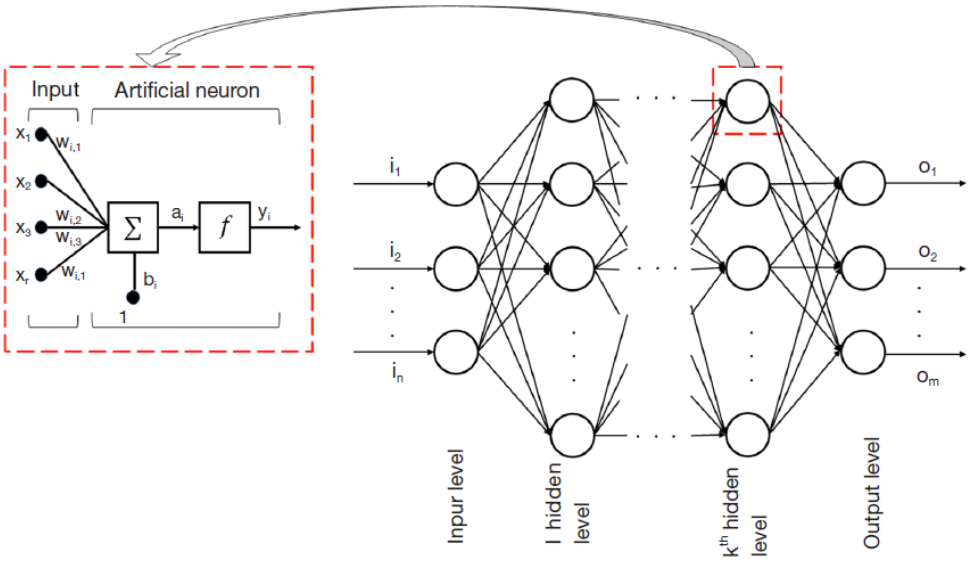


Figure 3.3: An example of feedforward neural network [33]. The structure is very permissive about the number of layers, of neurons per layer, input and output's size. We have the detail of a neuron's internal behavior. The neuron is getting the sum of the outputs (x_r) pondered by the weight ($w_{i,r}$) of the connections from the previous layer, then add the bias (b_i) and use this result (a_i) as input of the activation function (f). The output (y_i) is fed forward to the next layer of neurons.

Convolutional neural network

This is a special case of the FNN, with less connections in order train efficiently dimension reduction, which has become popular in computer vision. The goal of the convolutional layer(s) is to extract patterns before the analysis by the FNN. This is widely used for preprocessing an image to detect salient low- or high-levels patterns like edges, corners, textures, etc. In our example [10] (figure 3.4), we can see the 3 dimensional input and the result of the processing by the 2 filters. The input is an image of size 7x7, and the 3 depth dimensions are the color red, green and blue of each pixel. The convolutional layer outputs 2 features also called activation maps, which could represent lines, or other kind of patterns. This example uses a stride of 2,2, which means that the center of the blue matrix moves by step of 2 horizontally and vertically. This explains the 3x3 size of the output volume instead of 5x5. Since the goal is to reduce the size of the FNN's input, it is common to downsample the output of the convolutional layer (figure 3.5). The price to pay is the loss of some information, but the gain is dimensionality reduction. A complete CNN can have a structure with multiple convolutional layers, each of them followed by a downsampling function, and finally the FNN that has to take a decision about the extracted patterns (figure 3.6).

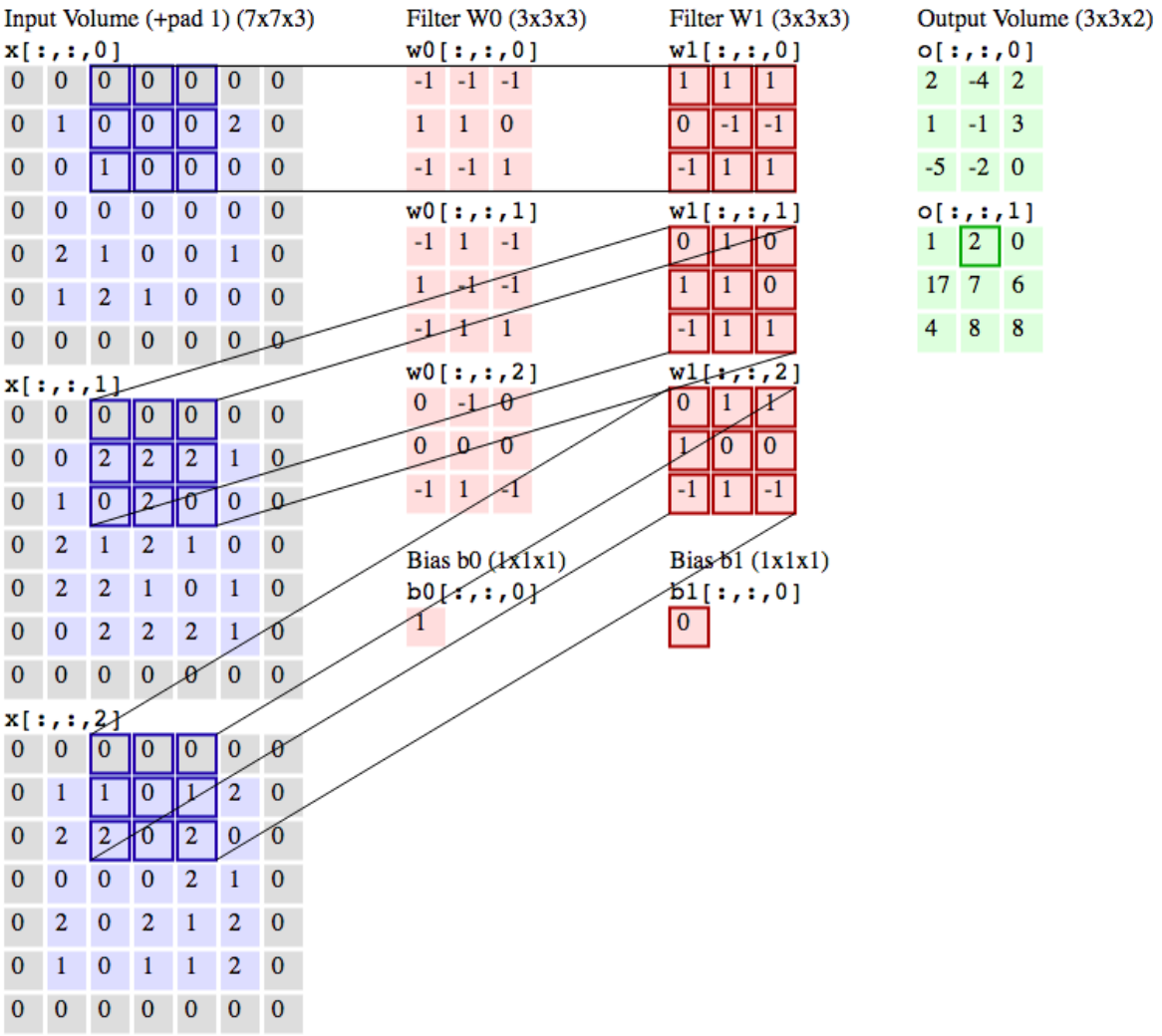


Figure 3.4: An example of a convolutional layer [20]. The input is a squared image of 5 x 5 pixels, with the three channels (red, blue and green). Adding the padding gives a 7x7x3 volume, which is filtered (red) into tho features (green) using matrix convolution. The filter $w0[:,:,i]$ and the bias $b0$ made the output $o[:,:,0]$. The current convolution is multiplying pairwise the 3x3 blue and red matrices, taking the sum of it plus the bias $b1$, gives the $o[0,1,1]$ result. The stride is 2,2 which means that the blue matrix moves by step of 2, horizontally and vertically.

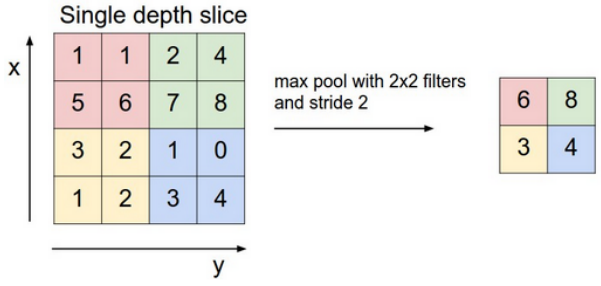


Figure 3.5: An example of max pooling [20]. Downsampling using the function max over each block of four numbers.

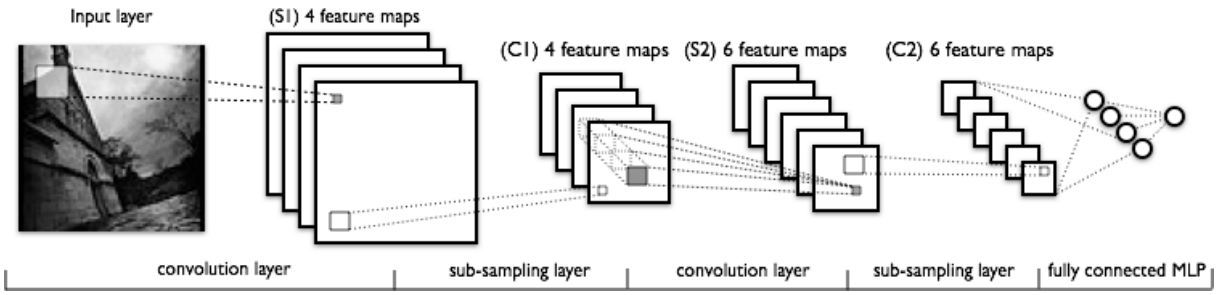


Figure 3.6: An example of the structure of a convolutional neural network [20], with a structure made of two convolution layers, each followed by a sub-sampling layer. The last part is a fully connected multilayer perceptron.

Recurrent neural network (RNN)

Unlike the FNN and CNN, the RNN can have a cycle in the connections (figure 3.7). This cycle is made because of the feedback connection. This does not change much if we present it as a contribution of the input of the next state. In some ways, we can unfold the loop to get multiple FNN feeding some information from time t to the network at time $t+1$.

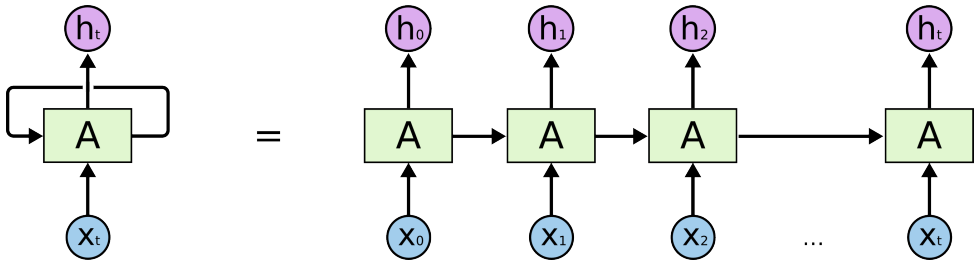


Figure 3.7: Recurrent neural network [32], on the left: the compact representation of a RNN. On the right: the same network, but in a unfolded view. The arrow from the left to the right contains some information from previous states

The advantage of a recurrent network is that it can keep some notions of memory. While it can be used for stream analysis, it can also be used to explore dynamically a static input. They are used in speech recognition, language modeling, translation, image captioning and way more. The simplest RNN can keep short term memories (figure 3.8)

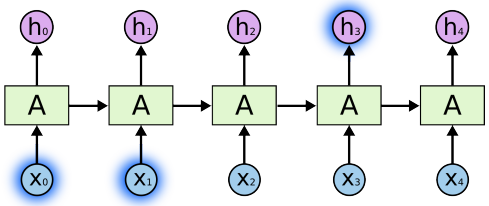


Figure 3.8: RNN with short term dependencies [32], there is no problem for the highlighted output to take the two highlighted inputs into account.

But it struggles with the long term dependencies (figure 3.9) even if it should theoretically be possible. The problem comes with the activation function, which is not exactly the identity

and thus changing the initial information over time. The in-depth analysis can be found in the paper [6].

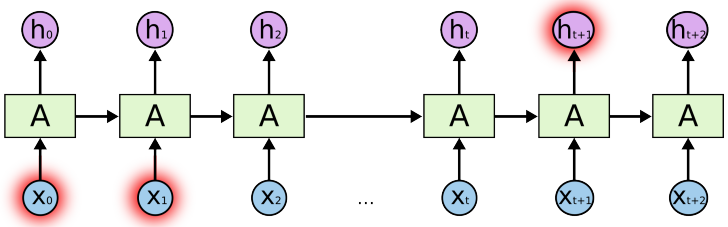


Figure 3.9: RNN with long term dependencies [32], it is very hard to train a network that do not forget all the information about the two highlighted inputs.

The long short-term memory neural net (LSTM)

The LSTM is a special case of the recurrent neural network. They solve the problem of the long term dependencies by keeping an internal memory state [32]. They were introduced by Hochreiter & Schmidhuber in 1997 [18]. The inside of a RNN with a tanh as activation function with a single hidden layer looks like figure 3.10.

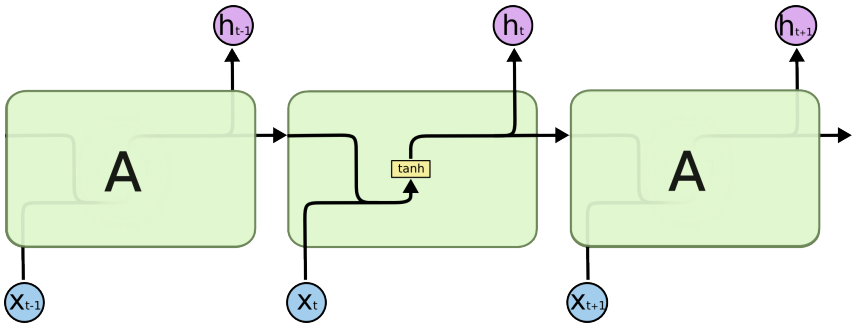


Figure 3.10: The repeating modules in a standard RNN [32]. The input at time t is concatenated with the recurrent connection from t-1. They both go through a tanh layer, building the output of the current step and giving this output to the next layer.

But the LSTM uses four layers instead of one, each of them having its own purpose (figure 3.11).

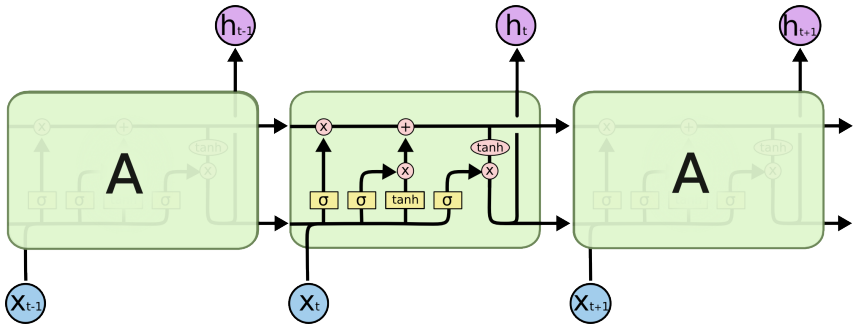


Figure 3.11: The repeating modules in a LSTM [32]. From left to right, the first σ layer is the forget gate layer, and it is trained to wisely chose when and what to forget. The second σ layer multiplies the tanh input layer, and controls what to write in the memory cell from the input and the previous output. The third layer is the activation function, modifying the input before the saving. This layer is similar to the layer that we had in the RNN example. The fourth σ layer controls what to output from the memory cell.

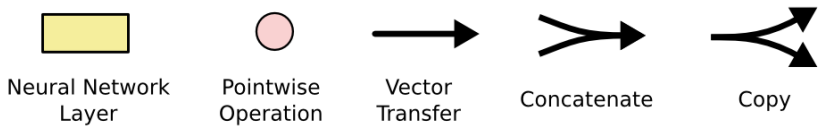


Figure 3.12: Notations of the above figures [32].

For reminder, the image of the σ function $\in [0, 1]$, and the image of the tanh function $\in [-1, 1]$. Here, instead of having a hard time to remember things, we have made a highway for the information to go through time, staying easily unchanged. If we initialize this network with a high bias, the first default behavior will be to remember things.

3.1.4 Deep reinforcement learning

This subsection briefly describes the algorithm used in the "Playing Atari with Deep Reinforcement Learning" article [29]. It is a combination of MDP, Q-learning and neural network.

Reinforcement learning

We want our agent to perform better with time, so we need to define rewards that he will get depending on the choices he makes. The thing is that it does generally not depend only on the last action, and that is where the gamma of the MDP becomes useful. The actions are defined using SARSA (state action reward state action) with the notation <s,a,r,s',a'>, with s, a and r for the time step t, and s', a' for the time step $t + 1$. Let γ be the discount factor, a quality of the certainty of the future rewards. For every action, the agent can hope for the expected reward using the Bellman equation [35]:

$$\begin{aligned} Q^{\pi}(s, a) &= \mathbb{E}[r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ... + |s, a] \\ &= \mathbb{E}_{s',a'}[r + \gamma Q^{\pi}(s', a')|s, a] \end{aligned} \tag{3.1}$$

which can give us the optimality equation:

$$Q^*(s, a) = \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s', a')|s, a] \tag{3.2}$$

The Q-function can thus be iteratively approximated with a simple algorithm as follow

initialize Q[num_states, num_actions] arbitrarily
observe initial state s
repeat
  select and carry out an action a
  observe reward r and new state s'
  $Q[s, a] = Q[s, a] + \alpha(r + \gamma \max_{a'} Q[s', a'] - Q[s, a])$
  $s = s'$
until *terminated;*

Figure 3.13: The basic Q-learning pseudo algorithm [27]

Deep Q-network

The problem of this algorithm is that our variable *num_states* is too large to fit into memory. And it becomes even worse if you decide to give some previous states as log. This is why we need neural networks, to approximate the state space, and output the Q value for us. We could simply use the network as a table, outputting the Q-value for a state and an action as input. This will require multiple forward passes in order to find the best action. Another better approach would be to give only the state as input, and output the Q-value for every action, doing only a single forward pass through the network.

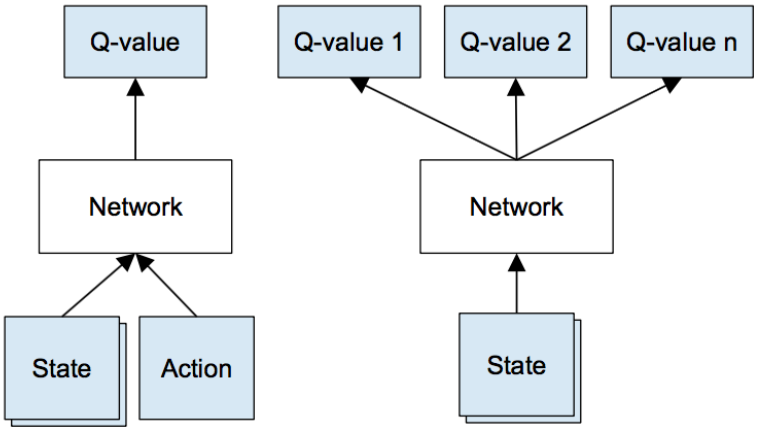


Figure 3.14: The naïve approach on the left, and the better one on the right [27]

The question now is how do we train the network to store the right $Q[s, a]$ values. This can be done in four steps given a transition $<s,\ a,\ r,\ s'>$

1. Do a first feedforward with s to get the current estimation $Q[s, :]$ of the network
2. Do a second feedforward with s' to get the network's current estimation for the next step $Q[s', :]$ and take the maximum Q-value for this next step $(max_{a'} Q[s', a'])$
3. Set the value of the action played $Q[s, a] = r + \gamma \max_{a'} Q[s', a']$, and let all the other predicted values unchanged
4. Train the network with new array

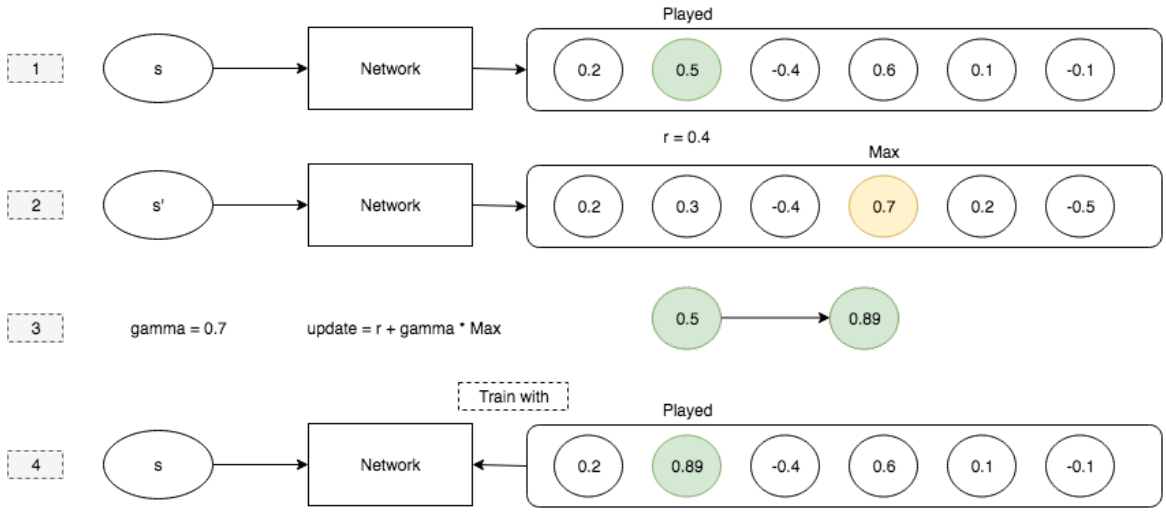


Figure 3.15: An example of update. The state s is fed into the network which predicts the Q-values of each actions. The second action is played, so we can verify if the predicted value is correct. We find a closer estimation of the real Q-value using $Q[s,a] = r + \gamma \max_{a'} Q[s',a']$ and then train the network using this new value for the played action, and letting the other predictions unchanged.

Exploration vs Exploitation dilemma

In order to try new strategies, but also exploit the existing ones, we need some randomness [13]. The usual way to do this is by using the epsilon-greedy strategy, that means go for the most promising Q-value with probability $(1 - \epsilon)$, and take a random action with probability ϵ. A common ϵ value is 0.1, but it can also start higher and decrease over time.

3.2 The Unreasonable Effectiveness of Recurrent Neural Networks

Andrew Karpathy, PhD student at Stanford University lists many applications of recurrent networks in a blog article [22]. This article is worth reading for anyone who wants to use LSTM networks.

3.2.1 Sequential processing in absence of sequence

DeepMind, the most famous research team in neural networks, published two papers on the sequential processing of images. The first article [4] is about reducing the input dimension of big images by focusing only on a part of the image (figure 3.16, on the left). It is inspired from our natural way of looking at an image. The model is called deep recurrent attention model (DRAM), and it uses a controller to choose itself what to look at, and a RNN to feed the chosen subframe.

The second article [17] also uses the attention mechanism, but in order to generate an image this time. They introduce the Deep Recurrent Attentive Writer (DRAW), which generate images similar to their training images. The figure 3.16 shows the result of a training with house number on the right. The figure 3.17 shows DRAW with the MNIST dataset. The output of this writer cannot be differentiated with the training set by naked eyes. A video of these images was published with the article and can be found on youtube [16].

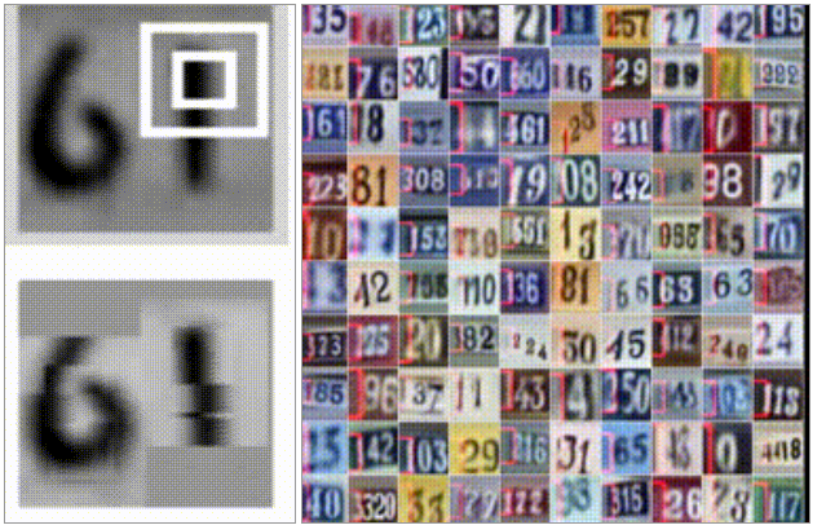


Figure 3.16: On the left, the network reading the image by focusing sequentially [4], and the images on the right are generated by a RNN coloring a canvas [17].

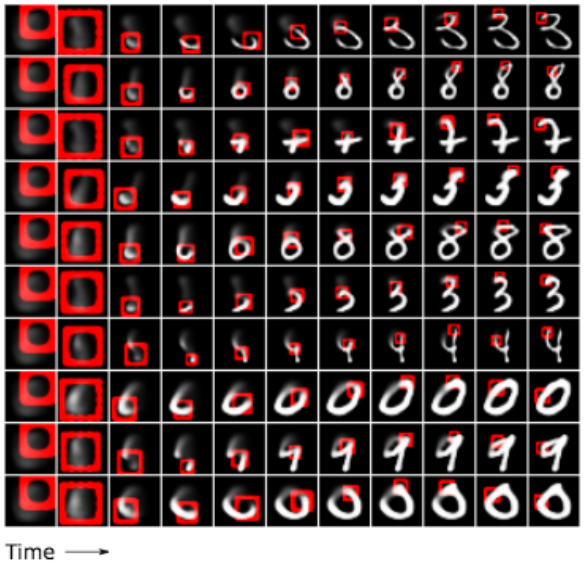
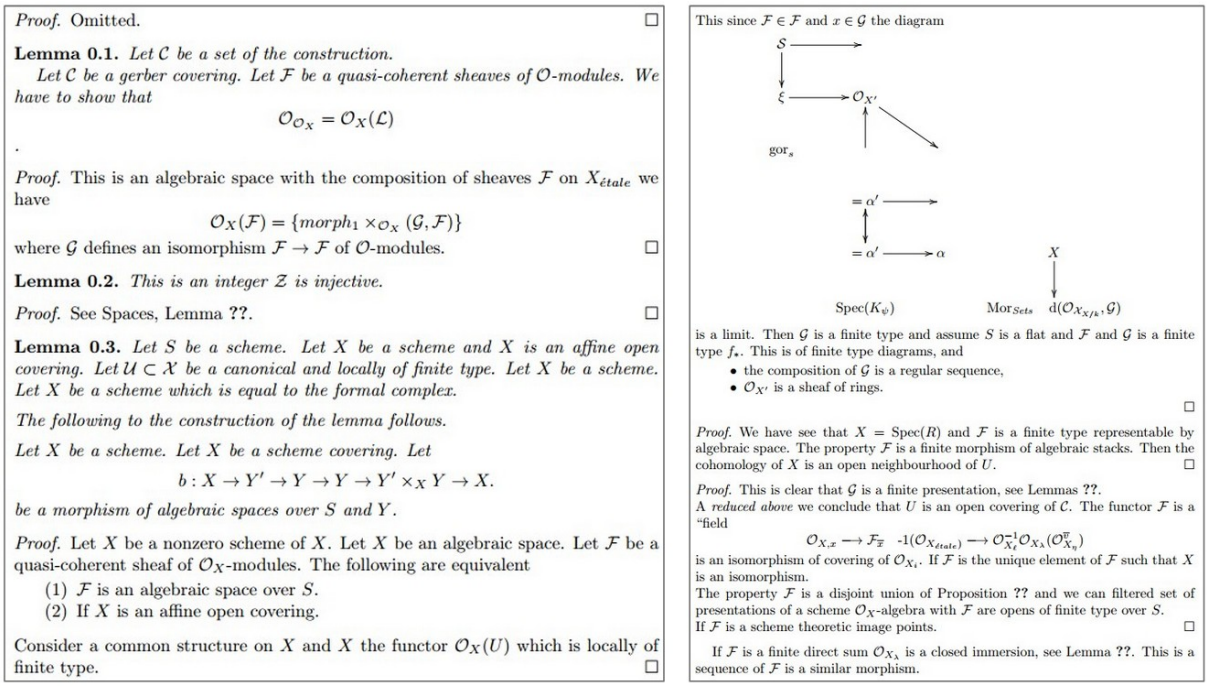


Figure 3.17: DRAW trained with the MNIST dataset [17].

3.2.2 Text Generator

With very simple settings, using a LSTM of 2 or 3 layers, it is possible to teach a network to generate English like sentences. This was done for fun by Andrew Karpathy, and resulted in a nice Shakespeare generator (the LSTM was trained with Shakespeare). But he also tried with Wikipedia articles, for which the LSTM was correctly using the markdown syntax. He then tried with latex text of algebraic geometry. The output almost compiled, and gave some interesting proofs... (figure 3.18) It even tried to make a graph, and note the omitted proof on the top left.



Proof. Omitted. □

Lemma 0.1. *Let $\mathcal{C}$ be a set of the construction.*
Let $\mathcal{C}$ be a gerber covering. Let $\mathcal{F}$ be a quasi-coherent sheaves of $\mathcal{O}$-modules. We have to show that

$$\mathcal{O}_{\mathcal{O}_X} = \mathcal{O}_X(\mathcal{L})$$

.

Proof. This is an algebraic space with the composition of sheaves $\mathcal{F}$ on $X_{\acute{e}tale}$ we have

$$\mathcal{O}_X(\mathcal{F}) = \{morph_1 \times_{\mathcal{O}_X} (\mathcal{G}, \mathcal{F})\}$$

where $\mathcal{G}$ defines an isomorphism $\mathcal{F} \to \mathcal{F}$ of $\mathcal{O}$-modules. □

Lemma 0.2. *This is an integer $\mathcal{Z}$ is injective.*

Proof. See Spaces, Lemma ??. □

Lemma 0.3. *Let S be a scheme. Let X be a scheme and X is an affine open covering. Let $\mathcal{U} \subset \mathcal{X}$ be a canonical and locally of finite type. Let X be a scheme. Let X be a scheme which is equal to the formal complex.*

The following to the construction of the lemma follows.

Let X be a scheme. Let X be a scheme covering. Let

$$b : X \to Y' \to Y \to Y \to Y' \times_X Y \to X.$$

be a morphism of algebraic spaces over S and Y.

Proof. Let X be a nonzero scheme of X. Let X be an algebraic space. Let $\mathcal{F}$ be a quasi-coherent sheaf of $\mathcal{O}_X$-modules. The following are equivalent

(1) $\mathcal{F}$ is an algebraic space over S.
(2) If X is an affine open covering.

Consider a common structure on X and X the functor $\mathcal{O}_X(U)$ which is locally of finite type. □

This since $\mathcal{F} \in \mathcal{F}$ and $x \in \mathcal{G}$ the diagram

$$\begin{array}{ccc} \mathcal{S} & \longrightarrow & \\ \downarrow & & \\ \xi & \longrightarrow & \mathcal{O}_{X'} \\ & \text{gor}_s & \uparrow \\ & & \\ & = \alpha' & \longrightarrow \\ & \updownarrow & \\ & = \alpha' & \longrightarrow \alpha \end{array} \qquad \begin{array}{c} X \\ \downarrow \\ \text{d}(\mathcal{O}_{\mathcal{X}_{X/k}}, \mathcal{G}) \end{array}$$

$$\text{Spec}(K_\psi) \qquad \text{Mor}_{Sets}$$

is a limit. Then $\mathcal{G}$ is a finite type and assume S is a flat and $\mathcal{F}$ and $\mathcal{G}$ is a finite type $f_\star$. This is of finite type diagrams, and

- the composition of $\mathcal{G}$ is a regular sequence,
- $\mathcal{O}_{X'}$ is a sheaf of rings.

□

Proof. We have see that $X = \text{Spec}(R)$ and $\mathcal{F}$ is a finite type representable by algebraic space. The property $\mathcal{F}$ is a finite morphism of algebraic stacks. Then the cohomology of X is an open neighbourhood of U. □

Proof. This is clear that $\mathcal{G}$ is a finite presentation, see Lemmas ??.
A *reduced above* we conclude that U is an open covering of $\mathcal{C}$. The functor $\mathcal{F}$ is a "field

$$\mathcal{O}_{X,x} \longrightarrow \mathcal{F}_{\overline{x}} \quad \text{-1}(\mathcal{O}_{X_{\acute{e}tale}}) \longrightarrow \mathcal{O}_{X_i}^{-1}\mathcal{O}_{X_\lambda}(\mathcal{O}_{X_\eta}^{\overline{x}})$$

is an isomorphism of covering of $\mathcal{O}_{X_i}$. If $\mathcal{F}$ is the unique element of $\mathcal{F}$ such that X is an isomorphism.
The property $\mathcal{F}$ is a disjoint union of Proposition ?? and we can filtered set of presentations of a scheme $\mathcal{O}_X$-algebra with $\mathcal{F}$ are opens of finite type over S.
If $\mathcal{F}$ is a scheme theoretic image points. □

If $\mathcal{F}$ is a finite direct sum $\mathcal{O}_{X_\lambda}$ is a closed immersion, see Lemma ??. This is a sequence of $\mathcal{F}$ is a similar morphism.

Figure 3.18: Latex output of a LSTM [22].

Then A. Karpathy tried to train the LSTM with Linux source code (in C), and the result is quite interesting. Of course, it does not do much, but there are very few syntactic mistakes, some random lines of comments, and it really looks like normal code (figure 3.19). We can see that the function is declared void, but returns something. This is a common mistake due to the long-term interaction.

```
/*
 * If this error is set, we will need anything right after that BSD.
 */
static void action_new_function(struct s_stat_info *wb)
{
  unsigned long flags;
  int lel_idx_bit = e->edd, *sys & ~((unsigned long) *FIRST_COMPAT);
  buf[0] = 0xFFFFFFFF & (bit << 4);
  min(inc, slist->bytes);
  printk(KERN_WARNING "Memory allocated %02x/%02x, "
    "original MLL instead\n"),
    min(min(multi_run - s->len, max) * num_data_in),
    frame_pos, sz + first_seg);
  div_u64_w(val, inb_p);
  spin_unlock(&disk->queue_lock);
  mutex_unlock(&s->sock->mutex);
  mutex_unlock(&func->mutex);
  return disassemble(info->pending_bh);
}
```

Figure 3.19: An example of generated code by the LSTM [22].

3.2.3 Other applications

The RNNs are also performing great in the NLP/Speech domain. They can transcribe speech to text, perform machine translation, generate handwritten text... As well as in computer vision, video classification, and visual question answering [34] (figure 3.20).

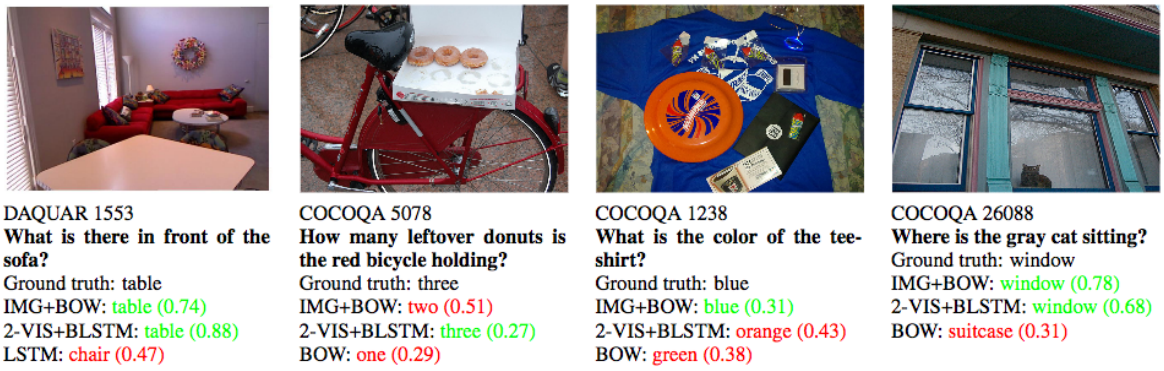


Figure 3.20: Visual question answering: sample questions and responses of a variety of models. Correct answers are in green and incorrect in red. The numbers in parentheses are the probabilities assigned to the top-ranked answer by the given model [34].

3.3 Atari games - DeepMind

A central work [29] related to this thesis deals with deep Q-learning in the Atari games. The DeepMind company, which is also behind the AlphaGo player, made a very generic AI. The goal of this paper was to make a generic AI for multiple games, without changing any meta-parameters, or architectures between the different training sessions. They used a deep convolutional network to evaluate a high dimensional input (the pixels of the game), and then evaluate the future reward of every actions as output. Since one screen is not enough to represent the whole state of the game, the input will be the few last frames, with their associated actions.

3.3.1 Experience replay

In order to stabilize learning, they will keep an experience memory, and train the network with random samples of this memory. It has three benefits:

1. Each step of experience is potentially used in many weight updates, which is good for the data efficiency.

2. Learning directly from consecutive samples is inefficient due to the strong correlation between them.

3. It avoids to have unwanted feedback loops in learning that could bring the network in a local minima.

The resulting algorithm (figure 3.21) is the improvement of the algorithm reported in the figure 3.13. The training was done on 10 million frames, with the memory containing the last 1 million frames.

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$
  end for
end for

Figure 3.21: The deep Q-learning with experience replay [29].

3.3.2 Reward Clipping

To avoid to change the reward scale from game to game, they used only -1 for any negative changes in game score, $+1$ for positive changes in the score and 0 for neutral actions. This loses the differentiation of the magnitude of the rewards, but it gains to be generic.

3.3.3 Epsilon strategy

They used the linear annealing ϵ-greedy policy, with 0.1 as minimum after one million frames. In other words, they started by playing only random actions, and slowly increase the probability of choosing the most promising action according to the network.

3.3.4 Frame skipping

To limit the computation time, they only feed one frame every k frames, which does not change much in the game, but make it possible to play$k$ times more games for the same computation time.

3.4 AlphaGo

The game of Go has an enormous search space and the value of the states are really hard to evaluate. This makes Go the most challenging game from the artificial intelligence point of view. AlphaGo is the intelligence that has beaten the European Go champion 5-0 using a value network to evaluate the states, a policy network to chose the next move and a Monte Carlo simulation to look ahead efficiently [36]. This is the first time that a computer has defeated a professional human player in a full sized game.

3.4.1 The policy network

This network is used to predict the next move, and was trained in two ways: at first using the log of expert human players in supervised learning, then by reinforcement learning (playing against

himself) to improve by optimizing the final outcome in games of self-play. This adjusts the policy towards the correct goal of winning games, rather than maximizing predictive accuracy.

3.4.2 The value network

This network is trained with the data generated by the reinforcement learning of the policy network and tries to estimate who will win the game give a board state. The figure 3.22 details the different learning part and neural networks used to predict the next move.

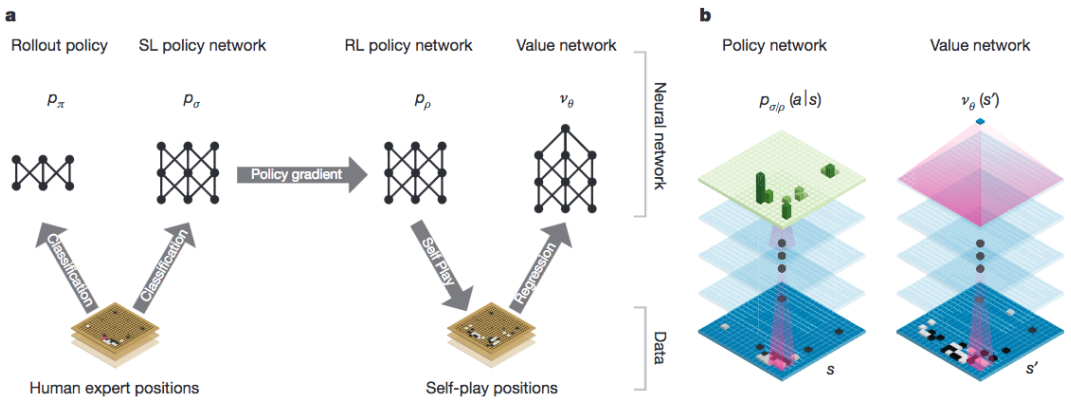


Figure 3.22: *Neural network training pipeline and architecture:* ***a,*** *A fast rollout policy p_π and supervised learning (SL) policy network p_σ are trained to predict human expert moves in a date set of positions. A reinforcement learning (RL) policy network p_ρ is initialized to the SL policy network, and is then improved by policy gradient learning to maximize the outcome (that is, winning more games) against previous versions of the policy network. A new data set is generated by playing games of self-play with the RL policy network. Finally, a value network v_θ is trained by regression to predict the expected outcome (that is, whether the current player wins) in positions from the self-play data set.* ***b,*** *Schematic representation of the neural network architecture used in AlphaGo. The policy network takes a representation of the board position s as its input, passes it through many convolutional layers with parameters σ (SL policy network) or ρ (RL policy network), and outputs a probability distribution $p_\rho(a|s)$ over legal moves a, represented by a probability map over the board. The value network similarly uses many convolutional layers with parameters θ, but outputs a scalar value $v_\theta(s')$ that predicts the expected outcome in position s'.* Figure and caption taken from the official article [36].

3.4.3 Monte Carlo tree search

Monte Carlo algorithms take benefits from randomness to limit the computational complexity of a function. They achieve slightly less stable result with some non-null probability, but are most of the time as competitive as deterministic algorithms. This is used to efficiently combine the two neural networks, by searching ahead in a probabilistic beam over the actions (figure 3.23). Searching in a beam means that we do not only take the best choice, but the N best choices. Here, since we are using probabilities, the N best choices are chosen randomly with their respective probabilities.

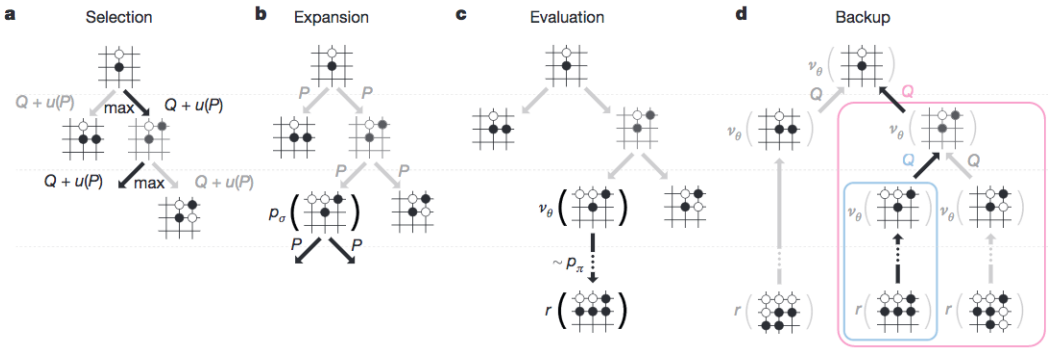


Figure 3.23: *Monte Carlo tree search in AlphaGo:* ***a,*** *Each simulation traverses the tree by selecting the edge with maximum action value Q, plus a bonus u(P) that depends on a stored prior probability P for that edge.* ***b,*** *The leaf node may be expanded; the new node is processed once by the policy network trained by self learning and the output probabilities are stored as prior probabilities P for each action.* ***c,*** *At the end of a simulation, the leaf node is evaluated in two ways: using the value network; and by running a rollout to the end of the game with the fast rollout policy p_π, then computing the winner with function r.* ***d,*** *Action values Q are updated to track the mean value of all evaluations r(.) and v(.) in the subtree below that action.* Figure and caption taken from the official article [36].

3.5 Machine Learning in Robocode: Reinforcement Learning of a robot with Functional Approximation Using a Neural Network

There are already some machine learning bots in robocode. But there is no structure or public information that contains them all. Here is the most interesting one, made last year by a contributor of the open source library dl4j that will be used for this thesis.

Learning here was done in two ways: once with a lookup table, and the second time using a neural network [38] [37]. The lookup table stores the pair <state, action> and the Q-value of the action. It is updated using Bellman's equation 3.2. During the second part, the bot uses the neural network to predict the Q-value from the pair *<state,action>* and does not take advantage of the multiple output as defined in figure 3.14. This does not really matter since the bot is only able to perform 4 actions: turn clockwise/counter-clockwise around the enemy, or shot while going forward/backward.

This project does not allow the bot to aim by himself or manually handle events like hitting walls or others. The bot does not inherit from the AdvancedRobot class and is thus restricted to one action at the time. The good point from this inheritance is that aiming is perfect due the game rules of basic robots, which means that the bullet is aimed to touch the exact center of the enemy. The compromise of the high level of accuracy is that the bot cannot aim preventively which means that he will struggles against bots that are always moving.

The state is defined with 4 numbers: the x and y coordinates reduced to 1-8, the distance to the enemy reduced to 1-4 and the absolute bearing reduced to 1-4. The action is represented by a number between 1 and 4. For example, the state 8413 means:

- $8 : 700 < x < 800$
- $4 : 300 < y < 400$
- $1 : 0 < distanceToEnemy < 250$

- $3 : 180 < bearing < 270$

This bot achieved a perfect score playing against the Spin bot over 100 rounds with the best settings. The training was made on 200k rounds, and the network had only one hidden layer.

4 | Method

> Success is not final, failure is not fatal: it is the courage to continue that counts.
>
> *Winston Churchill*

We will go through all the settings that will be used for the evaluation, then discuss about the appropriated associations of settings in order to get the more promising results.

4.1 Architecture

We must choose an efficient architecture in order to implement ease of learning. In this section we will go through different architectures made for different purposes. It also allows us to have a global view of the project before diving into the details.

4.1.1 Saving states

Robocode executes battles, but it requires some calculation even if we can disable the UI to speed things up. One normal way to avoid to recalculate many times the same things is to save the results in files, for easier later use. But it has to be generic and complete so that we can reuse it for any state representation.

Intercepting information

The state that we would like to save contains multiple information:

- The internal state of the bot (energy, position, gun heat, et. al).
- Its actions (forward, turn left, et. al).
- The events that it receives (bullet hits an enemy, et. al).

however, we do not have access to the code of the other bots. We thus need to find a way around. The solution was to inherit from the robot, and intercept all the information (figure 4.1) and save everything in external files. The game actually does not allow us to package a robot that does not inherit from the Robot or AdvancedRobot class, as a secure way to avoid cheating. But this is not completely safe, and once the bot is accepted by the game, it is possible to change the internal structure of the jar file to bypass the initial restriction.

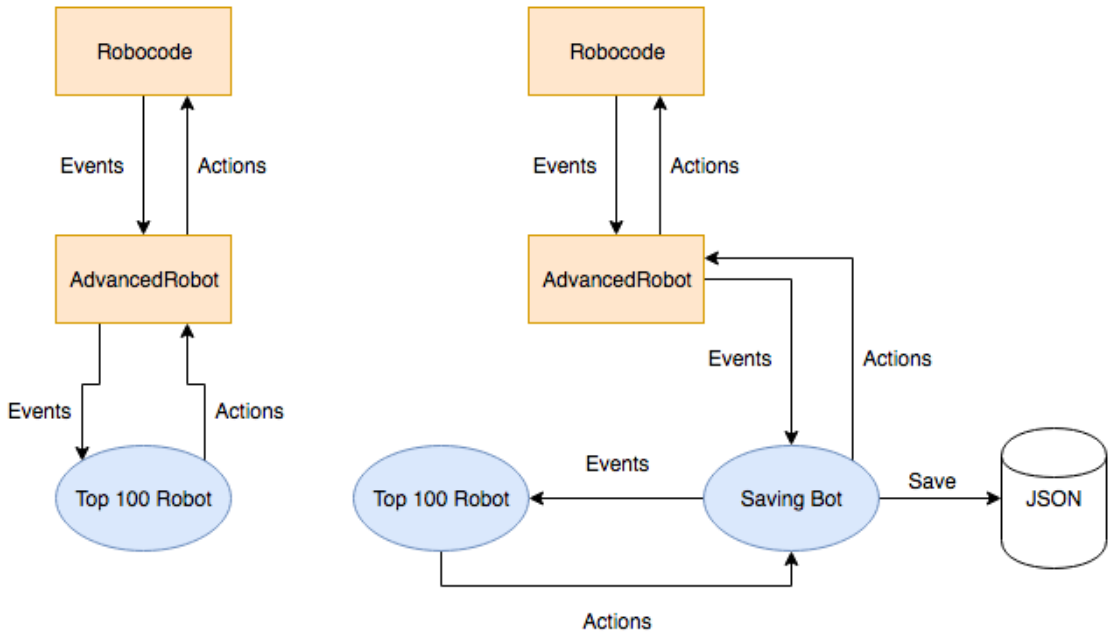


Figure 4.1: Left: the normal information path. Right: the information path in order to save the actions and events from a bot for which we do not have the code

Performances

One round (a fight until a robot dies) is approximately 300 turns (actions). One turn can be written in one JSON dictionary of 6 lines. At first, one round was saved by file, but 3000 calls to read files might be slow. The second try was to save 1000 rounds per file, but this made a java out-of-memory exception. The final choice was a compromise, reaching 100 rounds per file.

4.1.2 Network training

There will mainly be two phases, the offline training on the saved battles, and the online training using reinforcement learning and memory replay. As reminder, the concept of memory replay is to sample random past memories to train the network with a non-sequential experience in order to avoid poor local maxima. The architecture can be seen in figure 4.2. The training from other bots is a smarter way to explore strategies, but it suffers from the dimensionality reduction of the action space. We have to limit the possible actions of our robot due to the machine learning algorithm. But the other bots that have been implemented by hand do not have this restriction. This implies that our AI is not able to exactly copy the behavior of other bots. It is not similar to the offline training to the game of Go, in which the network is able to play exactly the same moves than the humans. We have the same problem by trying to learn from the memory of the opponent.

A way to avoid this problem is to make the bot play against itself (figure 4.3) in order to double the number of perfect memories, balance the number of positive and negative rewards since there is one winner and one loser and it also allows the bot to always play against a opponent of the exact same strength.

When we will train the LSTM neural network which has memories, we will clone the network at the beginning of the round, save the memories as one block, and train the network sequentially on both of them.

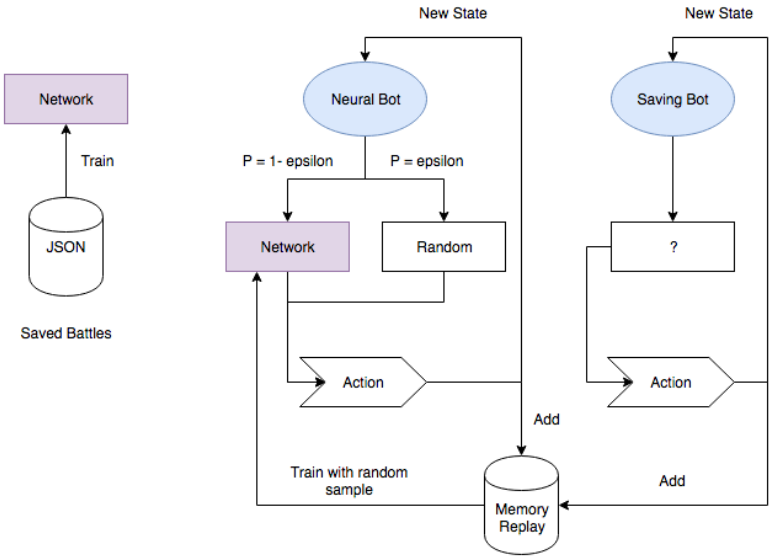


Figure 4.2: The structure of the neural bot during a battle. The network has been trained offline before, and continues to be trained using the memory replay mechanism.

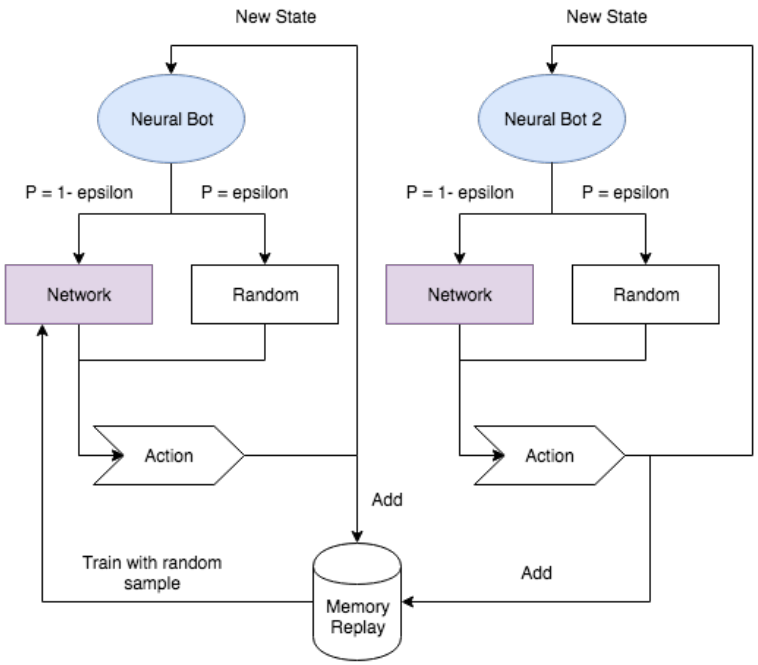


Figure 4.3: The two networks are actually the same network, playing for the two bots. Since the actions of the second neural bot are in the same space than the neural bot, we can keep them in the memory replay. We thus have as many won games than lost games, for a better representation of each part of the game.

Double Q-network

The Q-network has the problem to over-estimate the real Q-value [43]. A classical way to stabilize it is to use the double Q-network instead of using only one network. The idea is that we use an older version of the network to output the data used for the training. This older version is sometimes updated to the target network in order to stay consistent. This forces the target network to stay around the same values, and not over-evaluate with some bad samples. The resulting training algorithm is shown in figure 4.4.

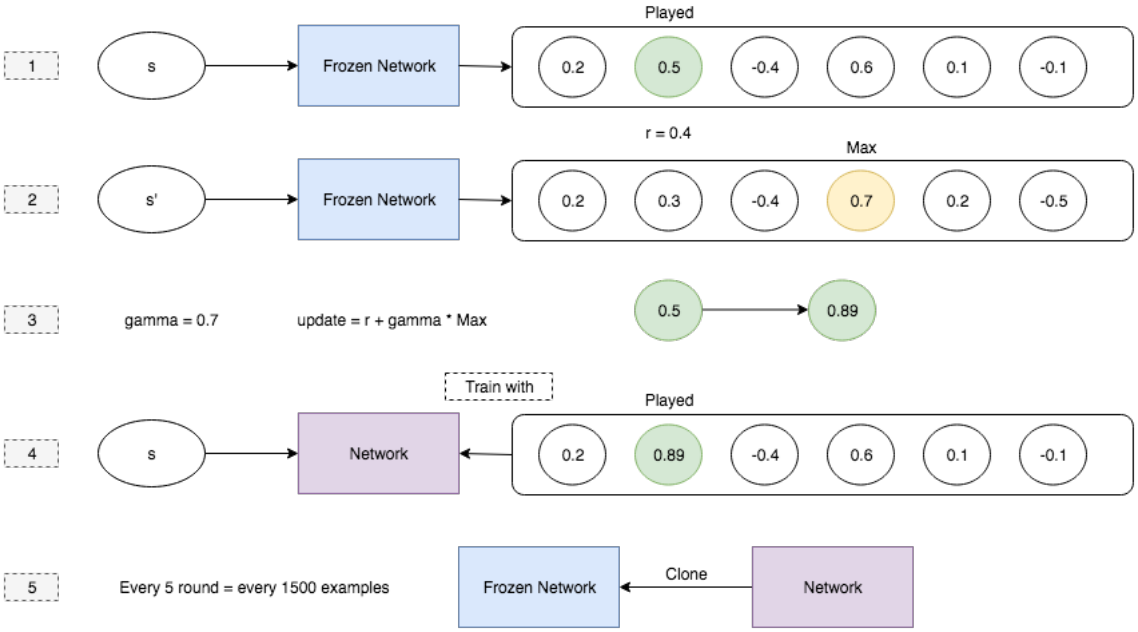


Figure 4.4: Training a network using the double Q-network technique. The output are generated by the frozen network, which is periodically updated to the target network.

4.2 State representation

There is a well know saying in machine learning: "garbage in, garbage out." This means that a model cannot have good performances if the input is not adapted to the problem it has to solve. This is one of the most critical point of the method. We cannot simply give the image as input since the robot is not allowed to have full information about the battlefield. Now comes the choice of the state representation. We will explore some representations unlikely to succeed, but easy to test, and other representations that are more promising. We will then reduce the responsibility of the network to the complex behaviors since some of the actions are very easy to implement perfectly by hand. Even if it would be a great machine learning demonstration, it is useless to learn the straightforward behaviors.

4.2.1 Raw

The first representation is not fully elaborated. It is the state as it is saved. So technically it contains all the information. The problem is that is does not take care of the symmetry since the positions are given in Cartesian coordinates. If the bot learns to deal with an enemy on the top right corner, it still must learn how to deal with an enemy in the three other corners and so

on since the representation is not rotational invariant. The representation of the raw data is given by figure 4.5.

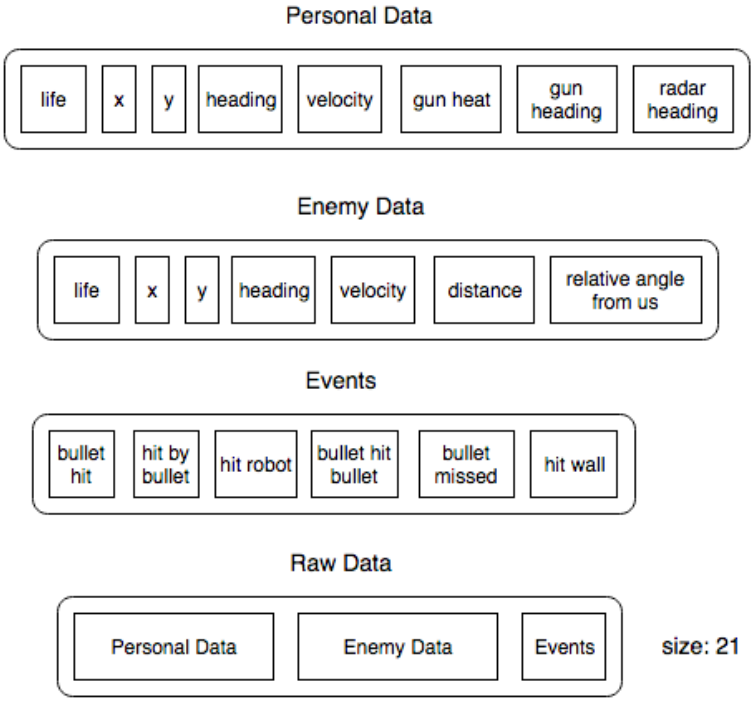


Figure 4.5: The representation of Raw data, test of input representation for the neural network.

4.2.2 Target Data

We would like to give a reward when the robot aimes towards the enemy, but this is impossible without using the radar. In a one versus one battle, the radar control is straightforward, it has to lock on the enemy and keep him inside the scanned area. We could then implement that part of the behavior, since it consists of 5 lines of code, that we can find in many robocode forums. This allows us to give an angular scan with the *enemy position*, and our *gun position*, so that the network has more information to start learning.

This representation is already better, since it does not give the network the information to differentiate symmetrical states. It is also more intuitive, and it becomes possible for a human to play with the input.

The target data contains:

- The *body scan*, which is like parking sensors in car bumpers, every $\pi/6$ rad, with a maximal distance of one fourth of the battlefield's size. This scan does not see the enemy tank, only the walls, and stays relative to the tank heading. The turning angle is also $\pi/6$ rad in order to make the impact of the action obvious in the input.

- The *enemy position* which contains a 1 if the enemy is toward that direction, 0 otherwise. This scan is relative to the tank heading. This scan has more precision (every $\pi/18$ rad), which is the gun turning step size. This vector is followed by a scalar which correspond to the normalized distance to the enemy.

- The *gun scan* which contains a 1 if the gun points toward that direction, 0 otherwise. This

scan is also relative to the tank heading, letting the optimal position (right angle between the gun and the heading) easy to find.

- The *bullet scan*, containing the distance of the farthest bullet in every direction, and 0 if no bullet are present.
- The events.
- Our life.
- The enemy's life.
- The offset to the position - added explicitly during the evaluation. The offset is a scalar representing the minimum angle to add to the accessible scan angle in order to be aligned to the enemy bot.
- The velocity decomposed in two, from polar coordinates (velocity toward us, and velocity perpendicular to the angular scan) - added explicitly during the evaluation.
- The actions.

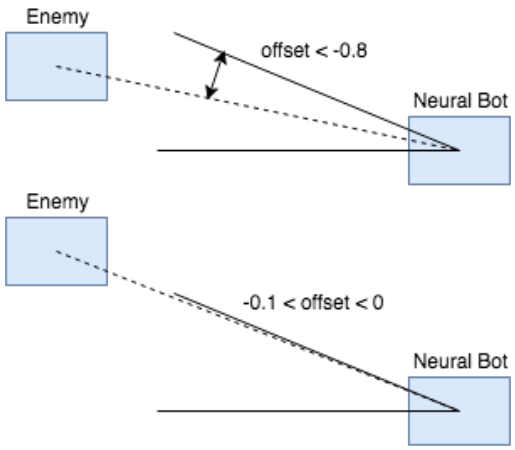


Figure 4.6: Example of two different offsets. If the offset is high, and the distance to the enemy is high too, the bullet will probably not touch a static enemy. If the offset is low, the bullet will touch a static enemy.

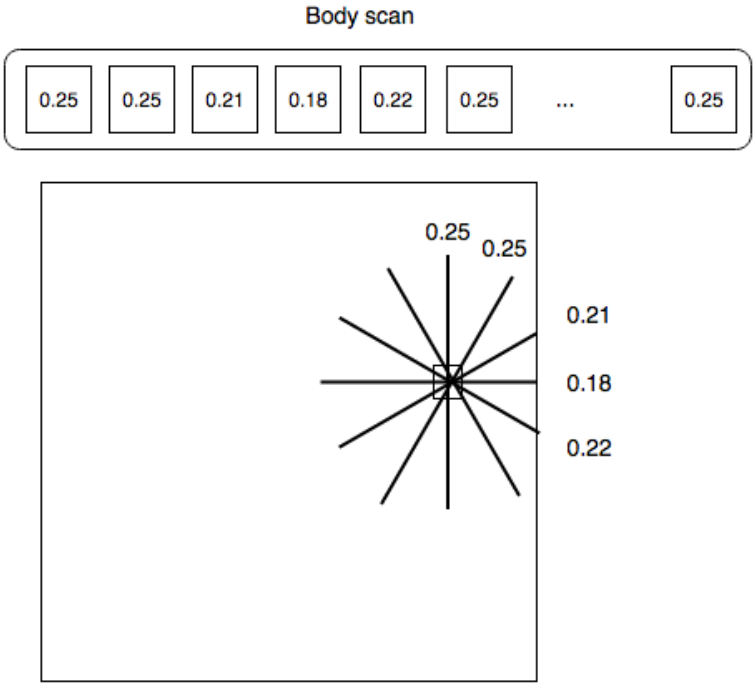


Figure 4.7: The *body scan* representation, which is rotation invariant. The square represents the battle field, and the lines are the inputs

4.2.3 Precise Target Data

This is the same as above, but with a more precise scan. Instead of a 10 degree step, it is a 2 degree step, going from 3 times 36 (enemy, gun and bullet scan) for the input size to 3 times 180, exploding the size of the input (from 136 to 571), but getting more precise. This test aims to address poor aiming precision from long range. There is also a need for finer granularity regarding the gun movement, and they will be defined in the next section. This growth in the input space will reduce considerably the speed of the training.

4.2.4 Log

We have no need for the log to find the velocity, unlike in the Atari games that needed the previous frames to have all the information. But it might be useful in order to keep events a bit longer and allow more precise reactions. It allows us to use a network without memory, since it is kept as input. The problem is that it makes the input dimension grow linearly. This raises the option of convolutional layers to extract patterns out of the evolution of the log.

4.3 Action representation

The choice of the actions must be a favorable compromise between the number of outputs for the Q-network and the freedom that it allows. They also need to have a well defined impact on the input. For example, if the rotation angle is 30°, the *body scan* should be made of 30° steps. In the thesis work, we will use four different action sets.

4.3.1 Advanced Actions

This representation was a bit too complete which allowed the robot to do everything, and all the combinations of actions. The action space was only made of 243 different action combinations, which is great for the learning from other robots, but a bit too expensive to explore.

4.3.2 Very Basic Actions

Since the *complete state* representation, there is no radar movement (controlled by hard coded code). Shooting is also kind of a trivial question, this is why it also got removed for the sake of dimensionality reduction. Idle was also useless, since it can be replaced with a movement and its opposite [21]. All of that shrank the number of outputs to 6.

4.3.3 Good Aiming Actions

Since the robot lacked precision, a natural evolution is to allow it to aim by steps smaller that 10°. The changes are adding two other steps similarly to another deep Q-learning project [19], one step of 6° and another one of 2°. The scan in the input must then change from one vector of size 36 to a vector of size 180. The new action size is 10.

4.4 Events representation

The events are often binary, letting us to represent them with 1 if the event is happening, 0 otherwise. If an information need to be kept, the info will be normalized and placed instead of the 1.

4.5 Network output

This section does not require much discussion; it is very similar to previous work.

4.5.1 Evaluate all actions at once

Instead of evaluating the pairs $<state,\ action>$, we will evaluate take the state as input and output the estimated Q-value of all the actions. This allows us to do only one feedforward pass and get the best action (figure 3.14).

4.6 Reward

This is a critical section, since it is the goal for which the network is learning. We could simply give a positive reward at the end of the game if we win, and -1 otherwise [29] [21]. This reward would be dragged back thanks to the $Q(i) = r + \gamma * maxQ(i+1)$ update formula. But we can also give more accurate rewards since we do not have to be generic like the Atari article. The following reward functions are the result of an evolution through many different reward strategies. For the sake of brevity, the evolution will not be exposed in the following section.

4.6.1 Specific evaluation

We can give rewards for every behavior that should make a competitive robot. Following this goal, we will use two different kinds of rewards: one from the difference of two state scores, and the other as bonus to one event.

State score

This evaluation contains information about the gun position. It gives a high score for a gun aimed at the enemy, and a low score if this is not the case. Since the *target data* does not allow the robot to aim well, an appreciation of the optimal distance is also added to the state score. It means that a positive change in either of those states will give a reward and its opposite for the reverse change. For example, the bot will get $+0.3$ for turning the gun toward the enemy and get -0.3 if he is turning in the wrong direction.

State bonus

The bonus is a way to directly give reward in reaction to an event. The win event yields the $+1$ reward, we sadly have no way to give malus for a defeat since the bot cannot compute anything once dead. The bot also gains rewards when he hits the enemy with a bullet, and gets malus when touched by a bullet. He also gains points when he stays well placed with his gun toward the enemy. The last bonus is for the relative angle of the gun to the body. The optimal value being $+90°$ and -90° so that the forward and backward moves allow easier dodge.

4.6.2 Basic evaluation

The *specific evaluation* (subsection 4.6.1) was maybe too specific. So this evaluation uses only a bonus for the win, for hitting with a bullet and a malus for getting hit, which is actually the only real matter.

4.7 Network

4.7.1 Deep forward network

The most basic network that we will try is the deep forward network with the ReLU activation function. ReLU stands for rectified linear unit which is equal to 0 if x < 0, and equal to x otherwise ($f(x) = \max(0, x)$). This activation function has the advantage that the gradient does not vanish for the deeper layers compared to saturating functions like the tanh and sigmoid functions. It allows us to build narrow but deep networks that can have fairly complex behaviors by composing ReLU functions. The meta-parameters of the network are:

- A learning rate of 0.01, this can be tuned using the UI of the DL4J library. The log of the update of the parameter ratio should be between -2 and -4 (< -4 means that the learning rate is too low, > -2 means that it is too high) [12]. In other words, we can have an appreciation of the quantity of change of our network. If the network does not change much, the update of the parameter ratio will be low, and it means that we could raise the learning rate value.

- The optimization algorithm: stochastic gradient descent, which is a standard in neural networks. It must be stochastic since we will train on one sample at the time. At first, we will not consider mini-batch training, which consists in grouping the experience samples in order to train the network with larger data block. It has the advantage to reduce noise on the loss function, and stabilize learning a little bit.

- The Mean Squared Error (MSE) loss function: again, this is classical for this kind of network. Another reason is that it allows us to train the network with the Q learning algorithm 3.15 and respect the Bellman equation 3.2.

- The updater is somehow arbitrary, and we will study the impact of changing this meta-parameter. It has been shown that the momentum must be around 0.9 to be efficient [40].

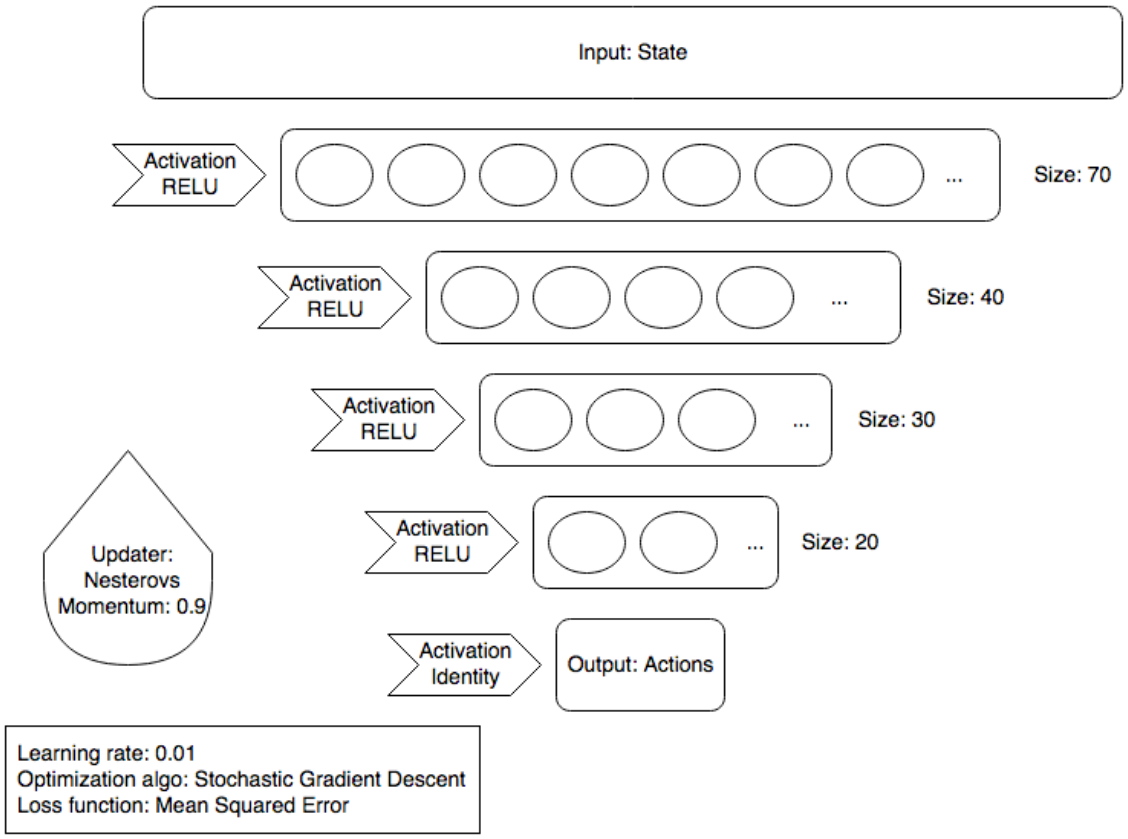


Figure 4.8: The deep forward network with his metaparameters.

4.7.2 Deeper network

We will then try a deeper neural network because it can more easily learn complex functions, but it slows training down a little bit since there are more parameters to update. We will arbitrarily add three layers in order to keep a clean decay of layer size.

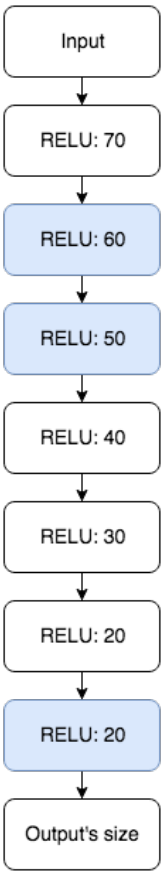
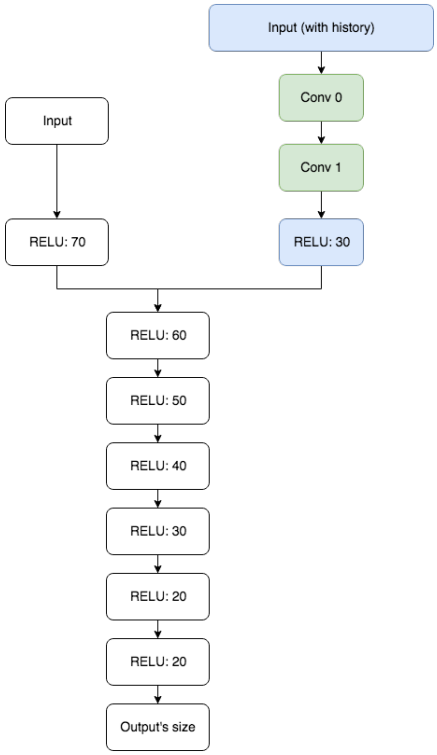


Figure 4.9: The deeper network architecture. We added the three blue layers, using the same parameters as previously. They are all using the ReLU activation function, and the size is slowly decreasing until the output prediction.

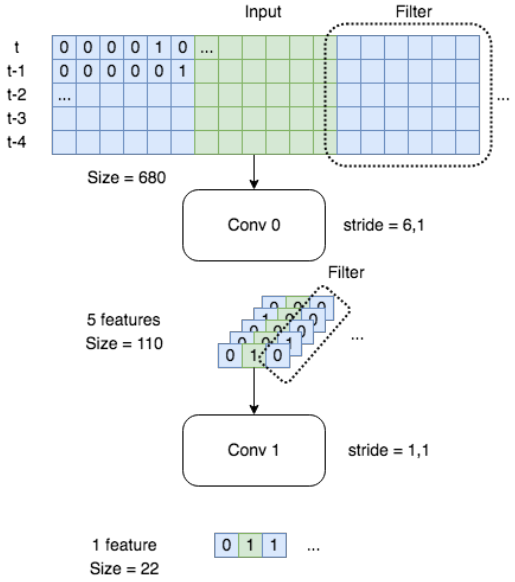
4.7.3 Convolutional hybrid architecture

In this architecture, we will use two inputs which will be processed in parallel before getting merged into the third layer of the previous neural network. We would like to add some log in the input, but it strongly increase the size of the input which is already big. A great tool to process an important input size in neural networks is the convolutional layer. But we would like to keep the initial precision on the current state.

We will thus change our architecture to build the best of both worlds (figure 4.10 (a)). The log will be made of the 5 last states, making the convolutional input of size $136 * 5 = 680$ that are given in one channel, in a matrix 136 x 5. This input goes through layer *Conv 0* which outputs 5 different features from the convolution of a filter 6 x 5 over the input (figure 4.10 (b)). This reduces the number of data from 680 to 110 ($5 * \lfloor 136/6 \rfloor$). This information is then used to feed the *Conv 1* layer, which has a filter of size 1 and depth 5, used to merge all five features into one. Its output is thus of size $22 = \lfloor 136/6 \rfloor$, containing all relevant information about the log. This is then fed into the fully connected ReLU layer of size 30 and merged to the third layer of our previous network.



(a) The previous network is in white. We merge the information extracted about the log in the third layer of the network. It has been well reduced thanks to the two convolutional layers (green). The activation function of the convolutional layers are the identity function.



(b) The *Conv 0* layer has a filter of size 6x5, a lateral stride of 6, and outputs five features, reducing the size of the input from 680 to 110. The colors show the data path. The layer *Conv 1* merges the five features letting the relevant information in a vector of size 22.

Figure 4.10: The new architecture of the neural network

We will also use this architecture with the *precise target data*, changing the input size from 136 to 571, letting the log input of size $5 * 571 = 2855$. We will also let the *Conv 0* layer output 10 features instead of 5, and the layer *Conv 1* output 3 features instead of 1. The rest of the network will remain unchanged.

4.7.4 Convolutional - LSTM hybrid architecture

We already have precise information about the current state, the global view about the small log, we simply need long term memory which would theoretically result in a very promising network. The new architecture is shown in figure 4.11 and combines the best of all worlds. The resulting sacrifice is that we cannot use the random memory replays anymore since we must train the memory of the LSTM layer on sequential training. This network should not suffer of the vanishing gradient since the LSTM layer is at the end of the back-propagation of the gradient, which has been well propagated by the ReLU layers. The LSTM layer has 50 neurons, and is followed by a ReLU layer which can choose the important information from the memory.

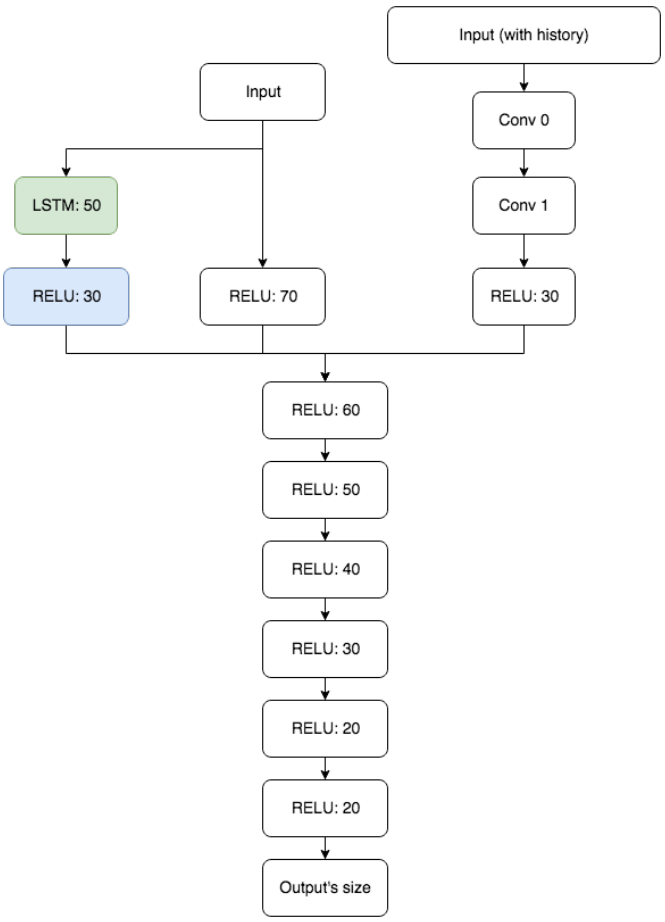


Figure 4.11: The architecture of the convolutional - LSTM hybrid computational graph. The LSTM (in green) and the ReLU layers (blue) have been added to the previous network (white). This allows to have long term memory while keeping the previous performances.

4.8 Network updater

Let dx denote the gradient and x be the vector of parameters. Once we have dx and x, there are multiple ways to update the weights.

4.8.1 Vanilla

The simplest way is to just update the parameters by multiplying the gradient with the learning rate (α):

$$x = x - \alpha * dx$$

but this has the drawback to oscillate a lot before finding the minimum. Nevertheless, it has the guarantee to converge toward a smaller loss function [1].

4.8.2 Momentum

A solution to the oscillating problem is to add some momentum (μ) to the derivative. The momentum forces the update to be partially in the same direction than previously and it also

allows learning to accelerate when the gradient points consistently in roughly the same direction:

$$v = \mu * v - \alpha * dx$$

giving the parameter update

$$x = x + v$$

.

4.8.3 Nesterov momentum

This is close to the momentum update, the only difference is the gradient that is taken from the new position instead of the current position [1] (figure 4.12).

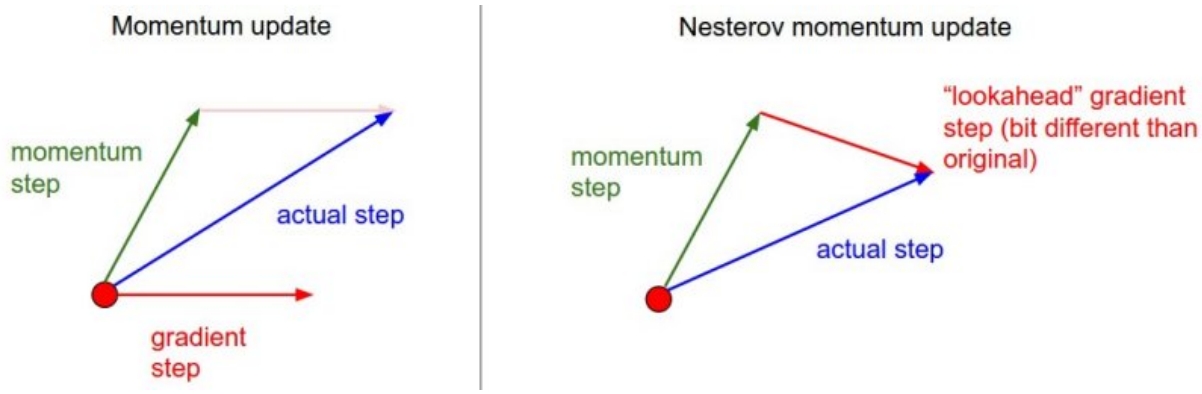


Figure 4.12: The gradient used for the momentum is taken from the new x position instead of from the current x position [1].

This method has shown to be slightly more efficient in practice, and also has stronger theoretical convergence for convex functions [5]. We will use this updater for most networks.

4.8.4 Adam

Adam stands for adaptive moment estimation [23], and uses the second-order moment to find the update [42]. The main component of the Adam algorithm are the average of the gradient (m_t = mean) and the squared gradient (v_t = the uncentered variance). There are 2 meta-parameters: $\beta_1, \beta_2 \in [0, 1[$ which control respectively the exponential decay rate of moving averages. The used learning rate is:

$$\alpha_t = \alpha * \frac{\sqrt{1 - \beta_2^t}}{1 - \beta_1^t}$$

and the update of the parameter θ:

$$x_t = x_{t-1} - \alpha_t * \frac{m_t}{\sqrt{v_t} + \hat{\epsilon}}$$

with the $\hat{\epsilon} = 10^{-8}$, a small constant to avoid division by 0. The complete algorithm (figure 4.13) shows the parameter updates over time, and the initialization. Adam has the advantage of being computationally efficient, and dealing well with sparse gradients and with non-stationary objectives.

Algorithm 1: *Adam*, our proposed algorithm for stochastic optimization. See section 2 for details, and for a slightly more efficient (but less clear) order of computation. g_t^2 indicates the elementwise square $g_t \odot g_t$. Good default settings for the tested machine learning problems are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t we denote β_1 and β_2 to the power t.

Require: α: Stepsize
Require: $\beta_1, \beta_2 \in [0, 1)$: Exponential decay rates for the moment estimates
Require: $f(\theta)$: Stochastic objective function with parameters θ
Require: θ_0: Initial parameter vector
  $m_0 \leftarrow 0$ (Initialize 1st moment vector)
  $v_0 \leftarrow 0$ (Initialize 2nd moment vector)
  $t \leftarrow 0$ (Initialize timestep)
  while θ_t not converged **do**
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep t)
    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)
    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
  end while
  return θ_t (Resulting parameters)

Figure 4.13: The complete Adam algorithm [42]. Please note the change in the notation. In this algorithm, they used θ for the parameters instead of x.

4.9 Discussion

It does not make sense to try all the settings one after the other, without first focusing somewhat on the more promising settings. We will try to differentiate the settings that could work together, and which of them should be tested first.

4.9.1 Architecture

The idea of exploring the strategies from the other bots is very powerful, but sadly, we do not really have the same freedom than the other bots since we are restricted to a small action space. We could try to let our bot use more that 200 actions, it would still be very incomplete compared to the huge combination of actions that are possible for the other bots. Furthermore, it would be really hard to explore in depth every action and evaluate their respective rewards. We will thus let this setting aside, focusing on the Q-learning part, without using the saved states nor the online experience of the other bot. We will still keep the *memory replay*, since it improves learning stability a lot, and does not suffer from dimensionality reduction from the infinite action space to our restricted action space. The bot can thus play exactly the same actions and get the same personal state transition than in the saved memory. This is not the case when we try to use the memories of the saving bot. We will also use the *double Q network*.

4.9.2 State representation

The *raw state* representation is not really promising, since it uses Cartesian coordinates and does not take advantage of the rotation problem although it has the advantage to avoid preprocessing. The *log* is a nice concept, but slows learning down. It must be tested, but combined with a

state representation that has already made improvement. The *target data* is probably the most promising one, since it has all the elementary information and uses complete information. The *precise target data* must be tried since it could lead to a better aiming bot. The offset in the *target data* will try to build the same quality of information without duplicating the size of the input.

4.9.3 Action representation

The *advanced actions* is certainly too big to have efficient learning. Even if it is very attractive to just say: the best choice is to choose everything. The *very basic actions* is the most promising one, since it is reduced to the smallest number, while keeping a reasonably large field of maneuver. The *good aiming* action must be coupled with the *precise target* state representation otherwise the actions will not be visible in the input state which is not recommended.

4.9.4 Rewards

The *specific evaluation* is the easiest evaluation, but has the disadvantage that we must tune the parameters, and find the criteria that will lead to a better bot. The positive side is that our network has a smooth function that is way easier to approximate. The *basic evaluation* should theoretically work, but the time needed to make it work is uncertain. We will try it just to get a feedback on this hypothesis.

4.9.5 Network updater

The *vanilla* updater is limited and it is fairly straight forward to find a stronger updater; the vanilla updater will not be used. It was introduced to get to the *momentum* updater which is also used to get to the *Nesterov* updater. We will thus only consider the *Nesterov* and the *Adam* updater.

5 | Evaluation

> The most exciting phrase to hear in science, the one that heralds new discoveries, is not "Eureka!" but "That's funny..."
>
> *Isaac Asimov*

In this chapter, we will go through health checks, test meta-parameters and try to evolve toward a smarter neural bot. We will analyze different settings during the Robot performances section, then we will go through the evaluation of the different architectures. We will conclude with an comparative analysis of the model performances of the architecture section.

5.1 Network health

The DL4J library has a very complete user interface to analyze many aspects of the neural networks [11]. It was used to debug the early problems found in the network, and adjust meta-parameters such as the learning rate, and make sure that the average activation were remaining steady during the learning. We will not expose this part of the thesis here for the sake of brevity.

5.2 Network predictions

There is no great learning curve to observe in the sense that the output error drop drastically in the 10 first iterations ($10 * 25 rounds * 300 turns = 75000 samples$) . This is typical of this kind of learning [29], and the informative analysis is the evolution of the Q score (equation 3.2), which should continue to grow during the learning. Another indicator is the difference between the estimated Q score and the real Q score which is an evaluation of the network performance. All of those will be studied in the next section.

5.3 Robot performances

Since it is difficult to understand the strategy of the bot by simply looking at the Robocode score, we need some other kind of information, like the actions choices evolution over training, which is not a perfect piece of information because the strategy might be very different even with the same action ratios. Nevertheless, it is still an efficient way to check the evolution of the strategy. We will now focus on the meta-parameters that will be used to build our neural bot.

In this section, we analyze multiple models. We will start by testing the *basic evaluation* with the first neural network versus a bot with limited behavior (subsection 5.3.1). This model did not work properly, by lack of feedback on the actions. We will thus consider the *explicit evaluation* which helped learning a lot, and try a second exploration phase which did not lead to better results (subsection 5.3.2). We will then change a little bit the input, and let our bot learn against two different enemies (subsection 5.3.3). This was the first model to win against an opponent,

and we will thus start analyzing the score of our bot versus some enemies to get something like cross validation. We will finally try the Adam updater, but without success (subsection 5.3.4).

5.3.1 Basics

Goal of this model. We will start with this model to evaluate if a raw reward is enough for the learning.

Settings. The first evaluation is about testing the simplest case. In this learning session, the opponent is taken from the sample bots given with the Robocode game. Its name is *Corners*, but it is hidden behind the Saving Bot name since we are using the latter as an intermediate to have access to more information. The behavior of the *Corners* bot is simple: the bot goes in a corner of the battlefield, then it stays there and scan the battlefield. If it sees an enemy, it shoots. We will let the network use the very basic actions (see subsection 4.3.2 for more informations) and getting the basic evaluation as reward (subsection 4.6.2). The input will be the target data (subsection 4.2.2). This training was done in 2276 iterations, which means $2276 * 25 * 300 = 17.070.000$ training samples. The reinforcement learning was carried out using the epsilon strategy, decaying linearly from 1 to 0.1 during the first 360 iterations (the step is 0.0001 every round), and staying at 0.1 for the rest of the training. The graph of the actions does not take into account the randomly generated actions, it only uses the output of the network.

Results. The details are shown in figure 5.2 for the chosen actions, and in figure 5.3 for the Q score and the Robocode score. We can see that there is not much learning, and that the actions seem to stay relatively random. Furthermore, the estimated Q score is very far from the real Q score.

Conclusion. The poor results are predicted to be related to the complexity of the reward function that has to be learned. The robot does not have the position of the enemy bullet as input, and is rewarded for actions that are very far away in time. For example, the robot has no data regarding it's aim and it must learn that its gun position 30 turns ago was actually the reason of the reward, and not the forward movement that was done during the reward. The problem of this reward function is its irregularity, and a possible solution could be to add slope to explore efficiently the search space (figure 5.1).

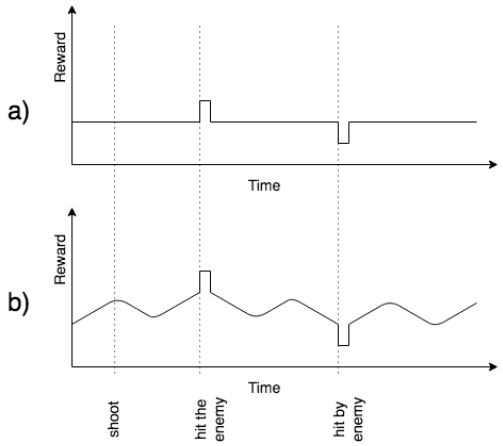
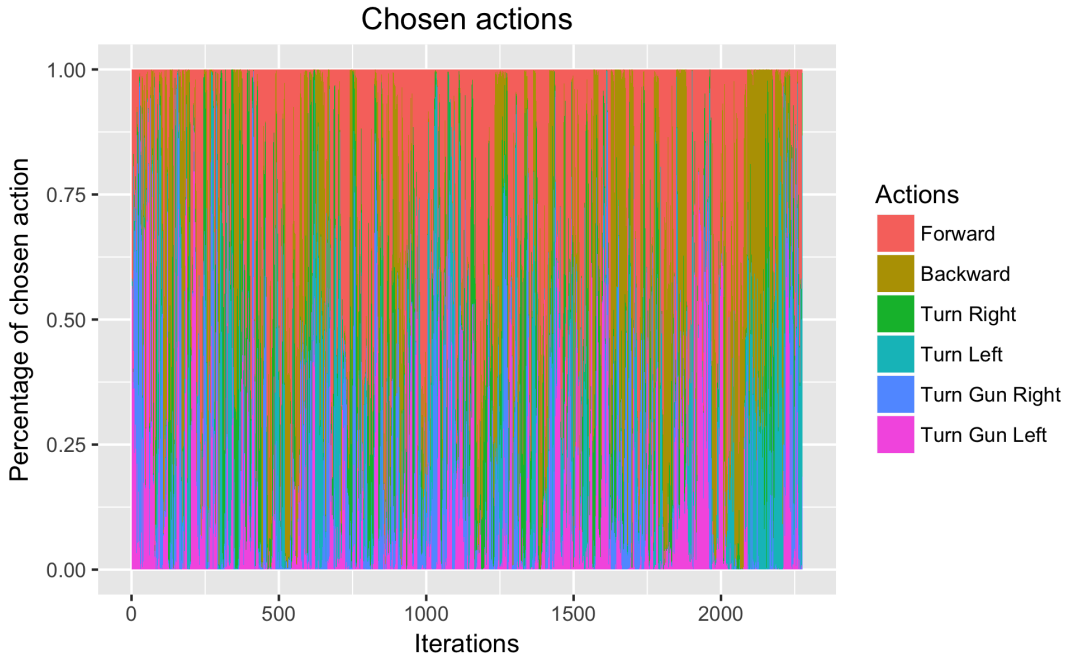
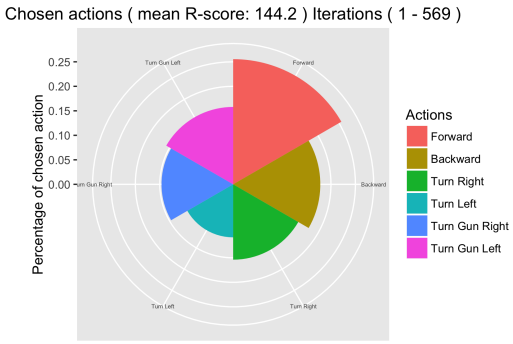


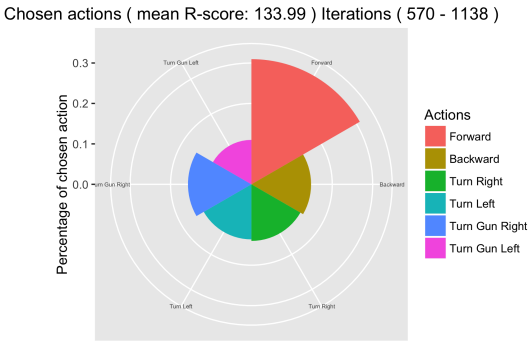
Figure 5.1: Example of two different reward functions for the same events over time. **a)** The reward function has steps, and it is hard to find which action at what moment was the cause of the reward. **b)** The reward function allows the bot to learn faster since there is more information about the possible source of the reward.



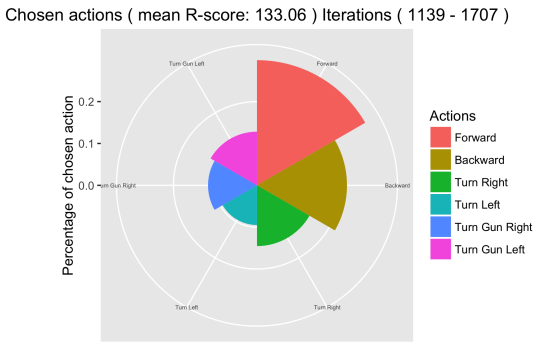
(a) Percentage of the chosen actions over the training iterations. Each iteration actually consist of 25 rounds times 300 turns. The network is trained with a random memory sample each turn.



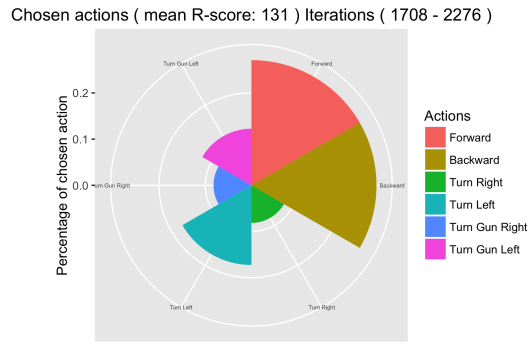
(b) The average percentage of the chosen actions over the first quarter of the training.



(c) The average percentage of the chosen actions over the second quarter of the training.



(d) The average percentage of the chosen actions over the third quarter of the training.



(e) The average percentage of the chosen actions over the fourth quarter of the training.

Figure 5.2: The actions analysis during the training over 17.070.000 samples. We can see that there is no improvement, or actions that are more developed that others. It means that the bot is changing a lot its strategy between each battle. A visual check stressed that it still plays randomly. This is due to the lack of non-zero slope in the reward function.

(a) The score of the reward function Q, true and estimated.

(b) The score of Robocode, for which the network has no information.

Figure 5.3: The Q score is very badly approximated, and the approximation does not improve over time. The Robocode score remains low when compared to the score range which is often achieved when played randomly.

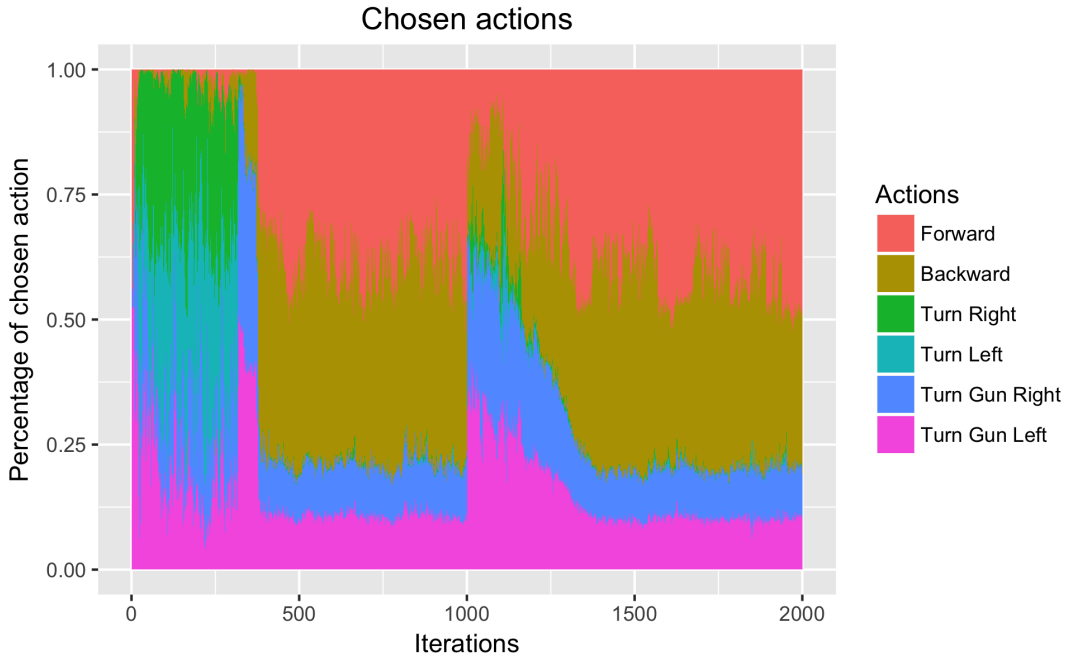
5.3.2 Explicit reward versus the *Corners* bot

Goal of this model. This model is used to evaluate the impact of a specific reward for the neural network. We will also evaluate the impact of epsilon in the evolution of the strategy.

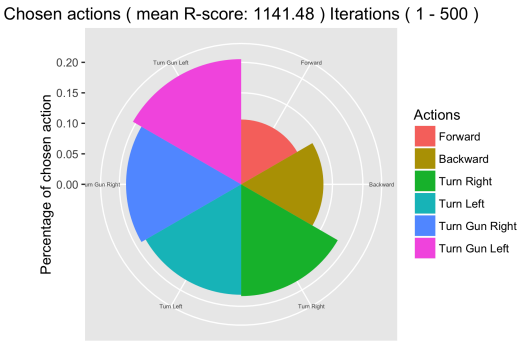
Settings. We will now train our bot using the *specific evaluation* (subsection 4.6.1) and the *very basic actions* (subsection 4.3.2) against the *Corners* bot and see if our bot can successfully aim toward a static bot, and dodge the bullets. We will use the same linear epsilon decay as the previous model, but we will restore an exploration phase in the middle of the training (figure 5.5). Since it is in the middle of the training, we will set it to 0.55 instead of 1, and go down with a slower decay (0.00005 instead of 0.0001 per round) to get back to 0.1 with the same amount of iterations (360).

Results. As we can see in figure 5.4, the network explores different actions during the beginning of the training, then reaches some equilibrium from which it does not escape. This neural bot is more exciting to watch based on the level of inherent strategy; sometimes it wins and sometimes it loses. The main behavior during its lost games is when it tries to shoot from far away. The Q score in figure 5.5 shows us that there is a nice progression during the beginning of the learning, then it stabilizes around some value. This is also visible in the Robocode score, and it is maybe due to the epsilon value. In fact, the first linear decay of epsilon from 1 to 0.1 takes 360 iterations, which is where it stops learning.

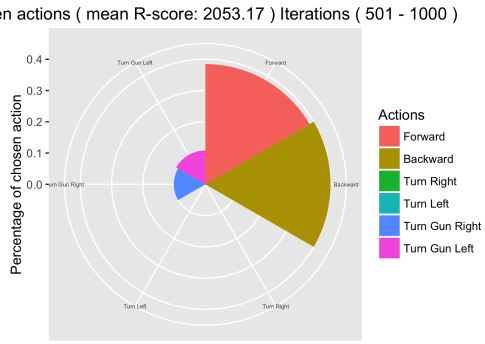
Conclusion. The problem of shooting far away is maybe due to the input choice. When the bot shoots from far away, it has to wait for a bullet to touch the wall to get the information that it is not aiming correctly. It has no other way to check it given the current input. The only evolution that it could try is to stay closer to the other bot. We thus need to try another input with some offset information. We could also ask ourselves if this model has reached its potential or if it is trapped in a local maximum. This is why the second exploration phase, made by restoring a higher epsilon, is useful. We can see in figure 5.5 that it makes the Q score drop a little, but this is normal since half of the actions are now random. The Robocode score does the same, which was also expected. Considering the second exploration phase, the neural bot still returns to its previous behavior, and the Robocode and Q score stabilize again, stressing the fact that learning has reached his potential with this setting. Thus, we need to try a new setting, with some information about the offset of the enemy (figure 4.6), to let it know that it will not touch the opponent even if the scanning data is aligned.



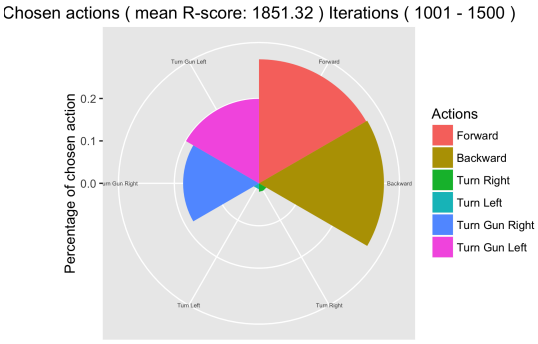
(a) Percentage of the chosen actions over the training iterations. Each iteration actually consist of 25 rounds times 300 turns. The network is trained with a random memory sample each turn.



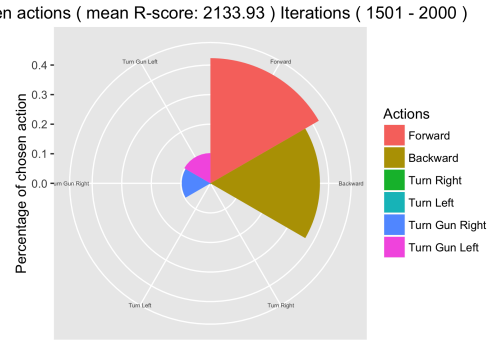
(b) The average percentage of the chosen actions over the first quarter of the training.



(c) The average percentage of the chosen actions over the second quarter of the training.



(d) The average percentage of the chosen actions over the third quarter of the training.



(e) The average percentage of the chosen actions over the fourth quarter of the training.

Figure 5.4: We can see that the Neural bot tries many different moves until it finds a good strategy. It does not evolve much after this event. Figures (c) and (e) show that it plays the same actions and gets around the same Robocode score. It most likely will not improve in this setting.

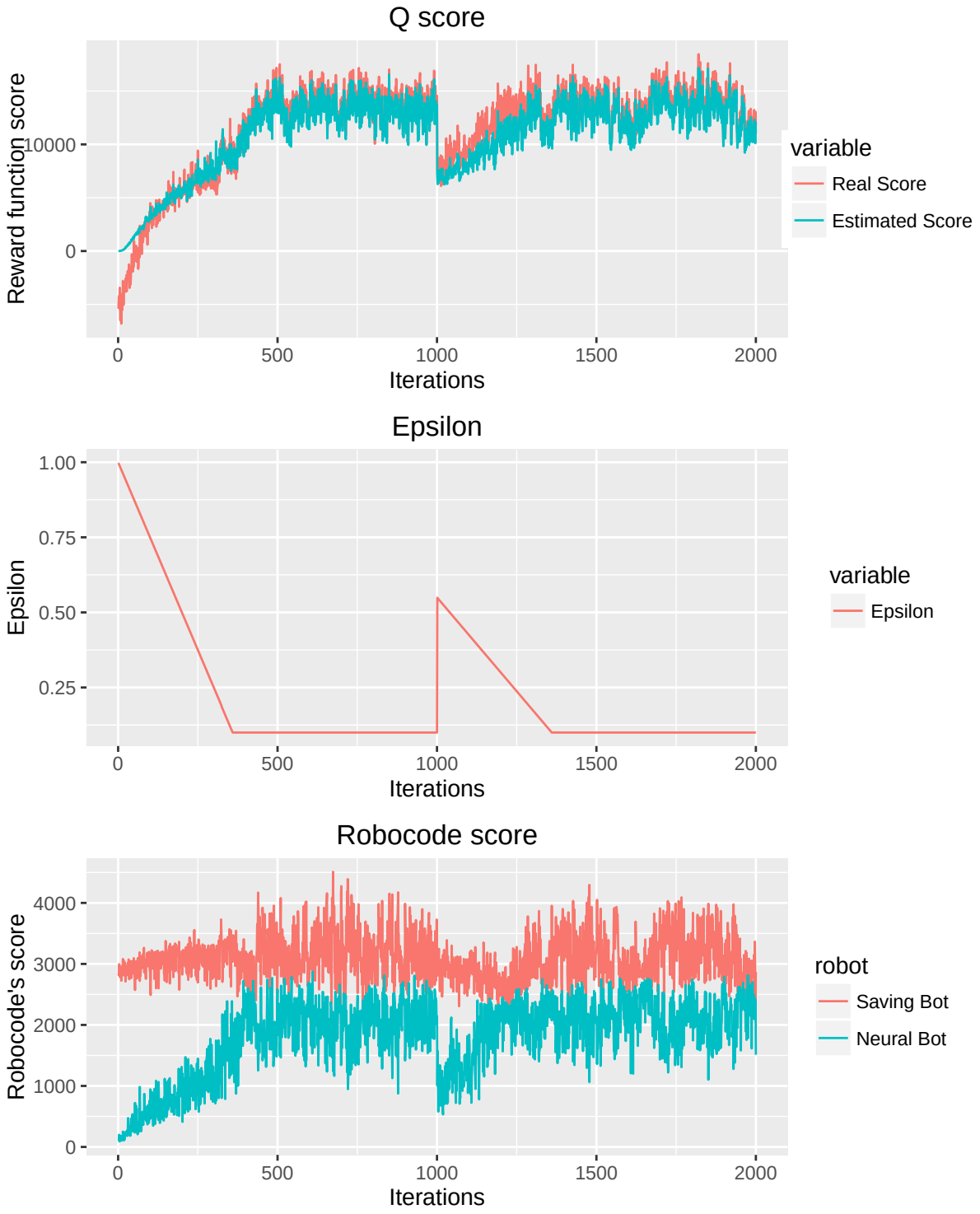


Figure 5.5: The Q score rose until iteration 500, then remained steady until iteration 1000. We observe a strong correlation between the Q score and the Robocode score, which was expected. The graphs reached the peaks when the epsilon started to be stable on the 0.1 value and this might be the reason of this stabilization. This is why epsilon has been reset to a higher value in order to check if more exploration would allow the score to continue to improve. It did not appear to be the case, since the lower Q score was due to the randomness of the actions, and went back to its highest value after the end of the epsilon decay.

5.3.3 Aiming offset information

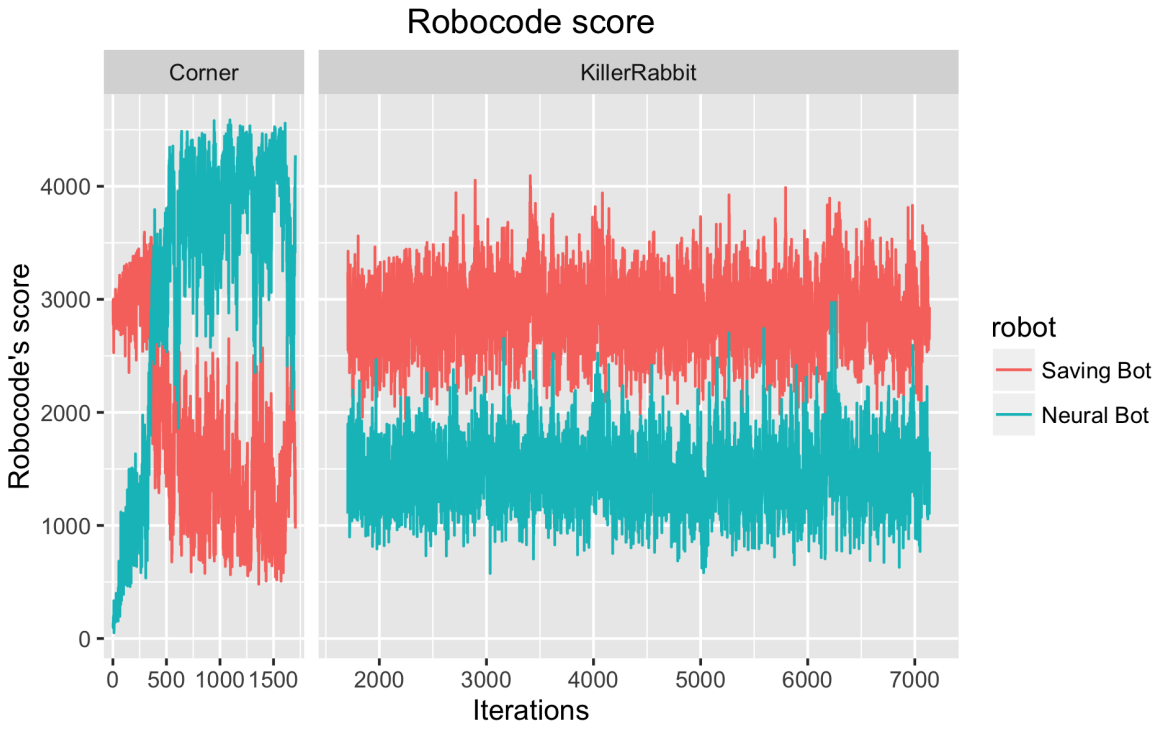
Goal of this model. Verify our hypothesis drawn from the previous results by testing with an offset for the aiming, and try to train our bot versus a competitive bot.

Settings. Now we will make simple modifications, and add the offset information to the input (figure 4.6). When the offset is equal to 0, if the bot shoots with the aligned scan data, the bullet will touch the center of a static enemy. Another change is that the weight of getting hit by a bullet and hitting an enemy by a bullet has been doubled since it is the main goal of this game. At this point we will also try different opponents: we will start with the basic *Corners* bot until iteration 1700 to learn the value of the offset information, then we will try to learn better behaviors on a more advanced bot (*KillerRabbit* - #512, randomly chosen in the top 1000) until iteration 7137 (around 53.527.500 samples).

Results. The scores (figure 5.6) stress the fact that, in this setting there will be no additional learning despite the use of more precise aim. The epsilon of the previous settings show us that the exploration might also be at its edge. Furthermore the result of some kind of cross validation is visible in figure 5.7. The statistic are generated using 4 iterations (100 rounds) against each enemy. The results are the averages of those 4 iterations to be able to compare the values with the other R-score graphs. Please note that the *Corners* bot is present in the cross evaluation since it is a sample bot. The *Bl4ck*, *net* and *Diamond* bots are from the top 1000 ranking. Of course, the bot is not learning during the cross evaluation. The first histogram is at iteration 700, which is the strong phase versus the *Corners* bot, 1700 is just before the enemy switch.

Conclusion. Since the Q score does not improve over time, we could start to think about changing the architecture of the network. The limited output of our network might also be the reason for the this constant score, since it might just be impossible to beat an enemy that is allowed to move more freely than our bot. As we can see in figure 5.7, our bot is performing worse at the end of the training than at the beginning, which may let us think that there is some overfitting of the strategy.

(a) The score of the reward function Q, true and estimated.



(b) The score of Robocode, for which the network has no information.

Figure 5.6: Our bot now beats the *Coners* bot. We can see that there is a jump down in both function, which was expected since the second bot is way stronger than the first one. We cannot observe any learning during the training session versus *KillerRabbit*. This might be an indicator that the exploitation (trying to improve the current behavior, opposed to exploration which means finding new behaviors) will not give better results with the current network architecture. The possible actions for our player might also be the reason.

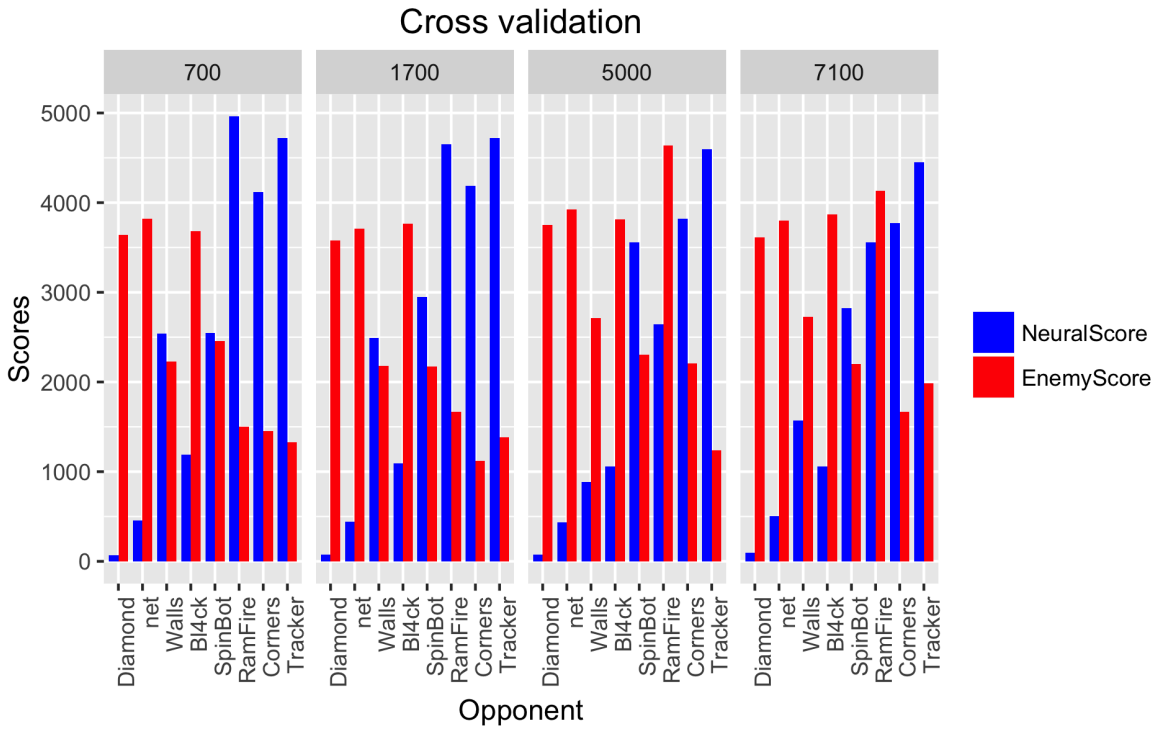


Figure 5.7: The four chosen iteration indices for this validation are in the middle (700) and the end (1700) of the training with *Corners*, and the middle (5000) and the end (7100) of the training with *KillerRabbit*. The opponent *Diamond*, *net* and *Bl4ck* are three very advanced bots, from the top 1000, and all the others are sample bots, some very basics, others a bit more challenging (*Walls*). The two scores are shown because they are not completely related. A good score means that a significant number of bullets touched the enemy, but a bot could win by dodging well and just touch once the opponent, which would reflect in both scores. We can see that *Diamond* has an advanced dodging strategy, but has around the same precision than the *Bl4ck* bot. The overall evolution shows some overfitting, with our bot getting worse versus other bots with time.

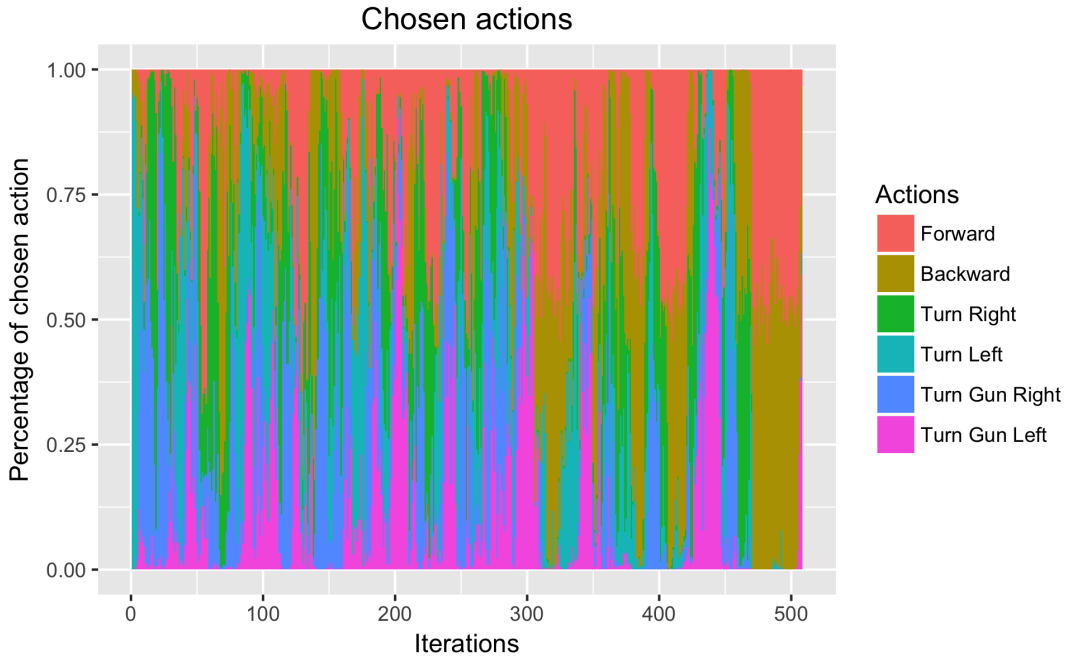
5.3.4 Aiming offset information with the updater Adam

Goal of this model. Evaluate the Adam updater.

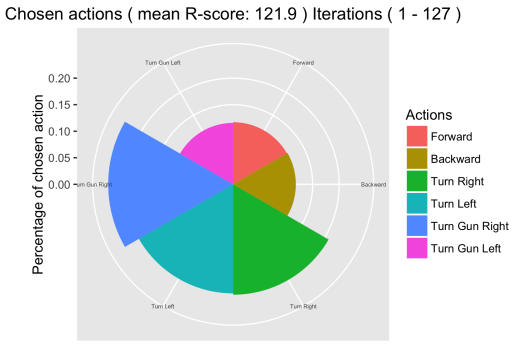
Settings. We will now try another updater, with the same settings as last model. The meta parameters of the Adam updater are the decay rate of 1st-order moment and the decay rate of the 2nd-order moment. They are set respectively at 0.9 and 0.999.

Results. We can see in figure 5.8 that he did not learn much after 500 iterations. Visual examination of the bot shows that he was still playing semi randomly, which also is seen in the Q score (figure 5.9), decreasing from the start.

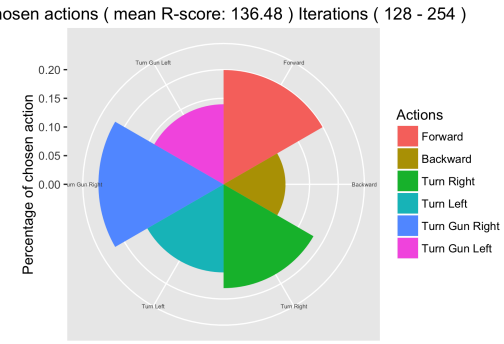
Conclusion. This is probably due to the increased speed of decay among the parameters since a large quantity of samples are required to learn something of significance. One advantage found in the Adam updater is that it is supposed to require very little fine tuning. This does not look to be true in our case, so we will go back to the Nesterov updater.



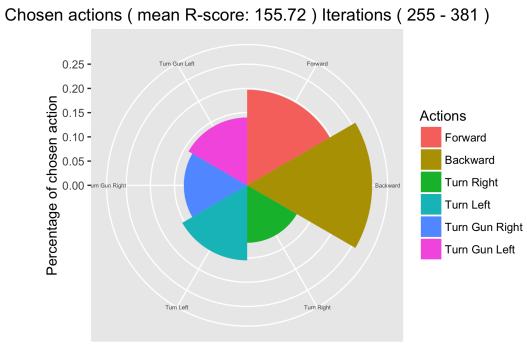
(a) Percentage of the chosen actions over the training iterations. Each iteration actually consist of 25 rounds times 300 turns. The network is trained with a random memory sample each turn.



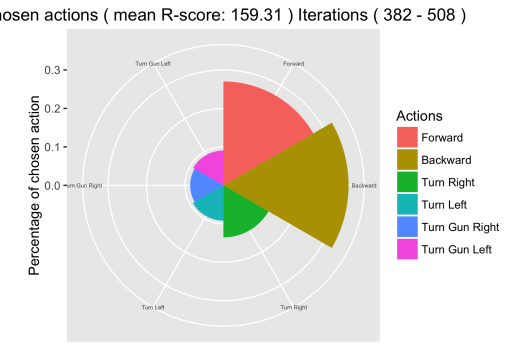
(b) The average percentage of the chosen actions over the first quarter of the training.



(c) The average percentage of the chosen actions over the second quarter of the training.

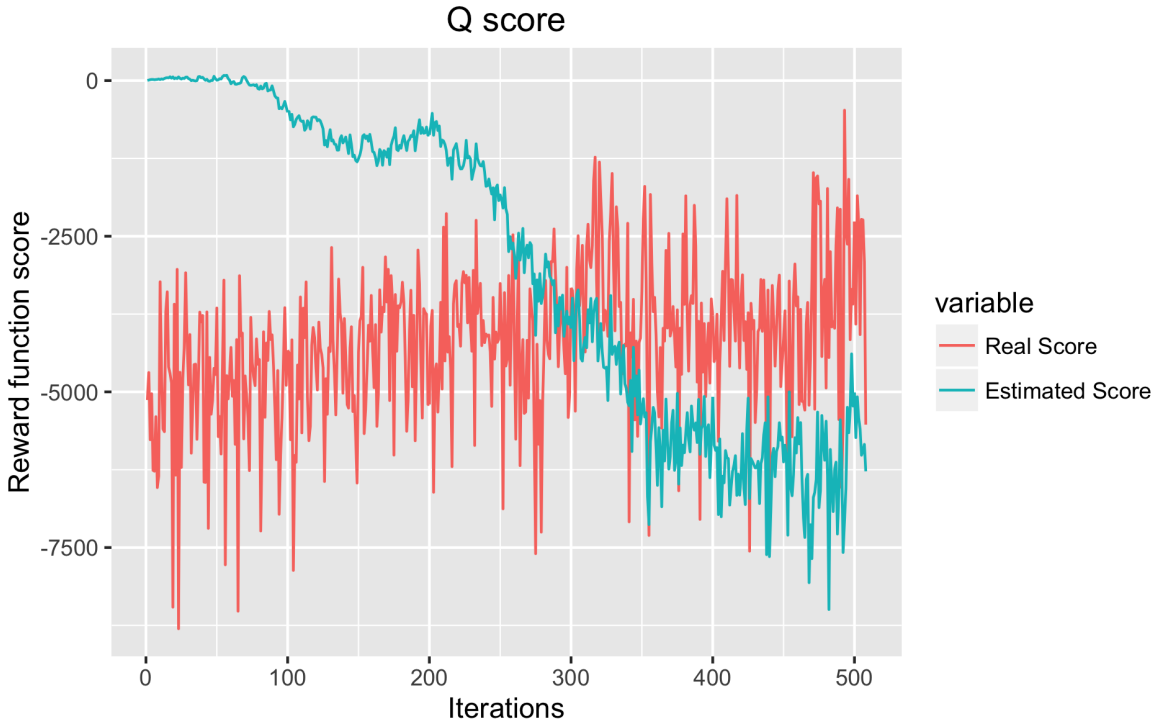


(d) The average percentage of the chosen actions over the third quarter of the training.

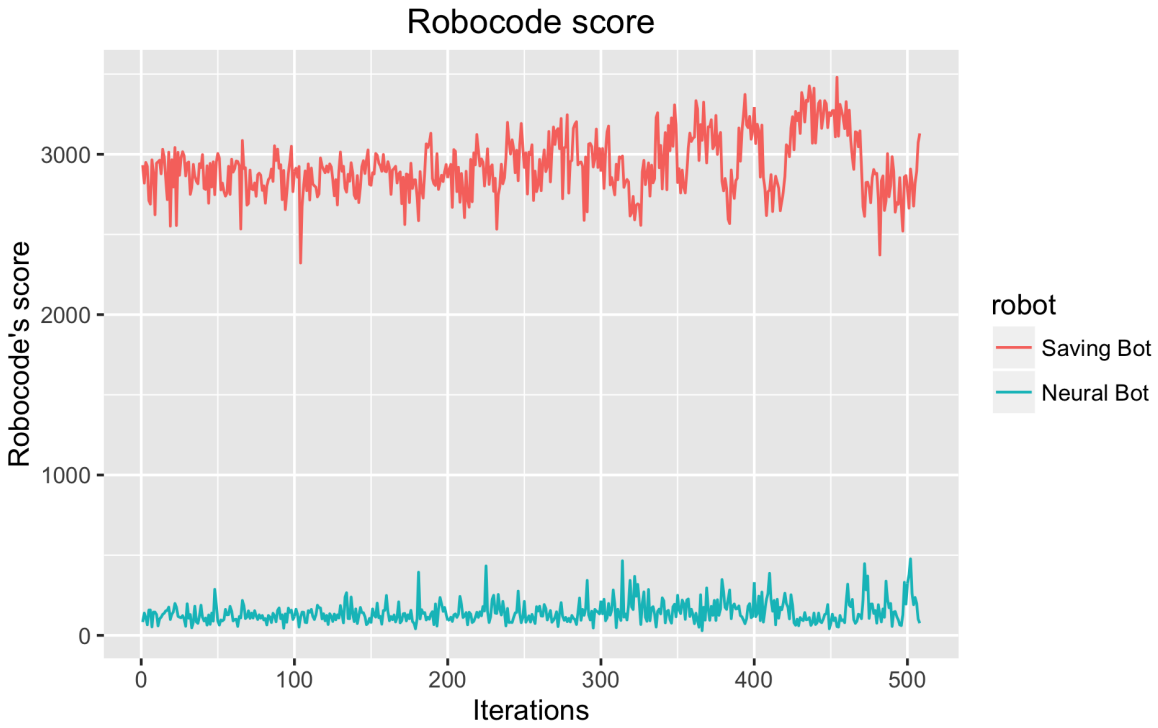


(e) The average percentage of the chosen actions over the fourth quarter of the training.

Figure 5.8: Actions analysis using the Adam updater. The actions exploration is positive, but it does not create any structured strategies (confirmed by visual check and the Q score).



(a) The score of the reward function Q, true and estimated.



(b) The score of Robocode, for which the network has no information.

Figure 5.9: The Q score is very poorly approximated, even after 500 iterations. It thus cannot increase, leading too a poor R score. Learning is either very slow, or not present at all.

5.4 Network architecture analysis

The next subsections will focus on the evolution of the architecture more than the choice of input or reward. We will start by deepening the network (subsection 5.4.1), with increase training time that led to an excessive amount of dying ReLU neurons. In the following evaluation, we let the bot play against himself, in order to avoid the previous problem (subsection 5.4.2). This model worked very well even if there was a risk of bad local maximum. The next model (subsection 5.4.3) will use the hybrid neural network architecture, combining the feed forward layers with the convolutional ones. We will then try appending a LSTM layer, removing the random memory replay algorithm (subsection 5.4.4). This led to a very poor result, due to the sequential memory training. The last model was trained on a server, using the precise aiming input and the convolutional hybrid architecture (subsection 5.4.5).

5.4.1 Deeper Net

Goal of this model. Evaluate the impact of a deeper neural network. We will also train the network using many different opponents.

Settings. We will now try a deeper architecture, as well as a more complete method to train the network. The new architecture has 3 more layers. The size of the layers are now, in the order: input size, 70, **60**, **50**, 40, 30, 20, **20**, output size (figure 4.9). Instead of very few enemy changes, we will try to change every 100 iterations. In order to promote new adaptations against the new enemies, we will also set the epsilon back to 0.2, with the usual linear decay (over 40 iterations) and let the 60 last iterations for the exploitation part of the search (figure 5.10). We will also do cross validation in order to determine if the bot forgets what it previously learned, or if he is generalizing better with the new experience. It remains a cross validation since every battle is new, and memorizing the previous battles would not increase the survival chances by itself.

Results. We can observe in figure 5.11 that the network crashes in iteration 2341. This means after 17.557.500 of samples. The network does not output numbers, but only NaN (Not a Number). We do not have access to the user interface to gather more information, but the structure of the network remains. This is thus probably an internal crash more than a saving/restoring the network issue. We can observe this strange behavior against the opponents using the Q-score in figure 5.12 or the Robocode score in figure 5.13. The cross validation shows the performances decay over the iterations in figure 5.14.

Conclusion. A possible explanation for the crash is the dying ReLU problem [26]. This happens when a large gradient causes the weights to update so that the neuron will never activate on any data point again. When this happens, the neuron "dies" and the gradient in this neuron will forever be 0. This happens more often with a high learning rate. We thus have two solutions: lower the learning rate, or use the leaky ReLU, which has a non-zero gradient in the negative x ($f(x) = \alpha x$ if $x < 0$ and x otherwise, with α a small constant). Sadly, the leaky ReLU are not always consistent, sometimes better than the ReLU but sometimes worst. There are other alternatives, like the noisy ReLU, but none outperforms the classical ReLU.

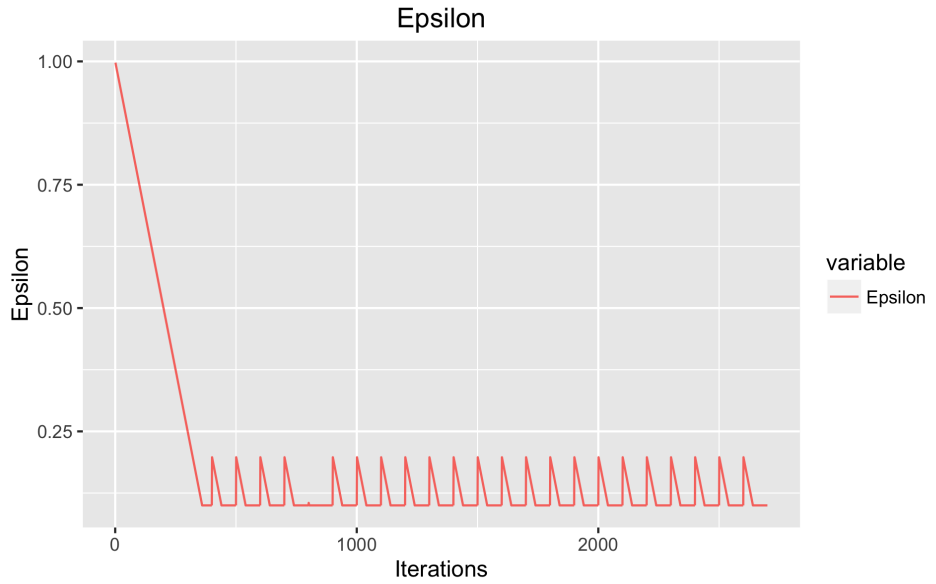


Figure 5.10: The epsilon strategy, with the usual decay at the start, and a new spike of exploration at the beginning of every enemy change. The missing spike is a mistake during the simulations. This does not really have a consequence since it does only change the ratio of randomness during the training against one opponent.

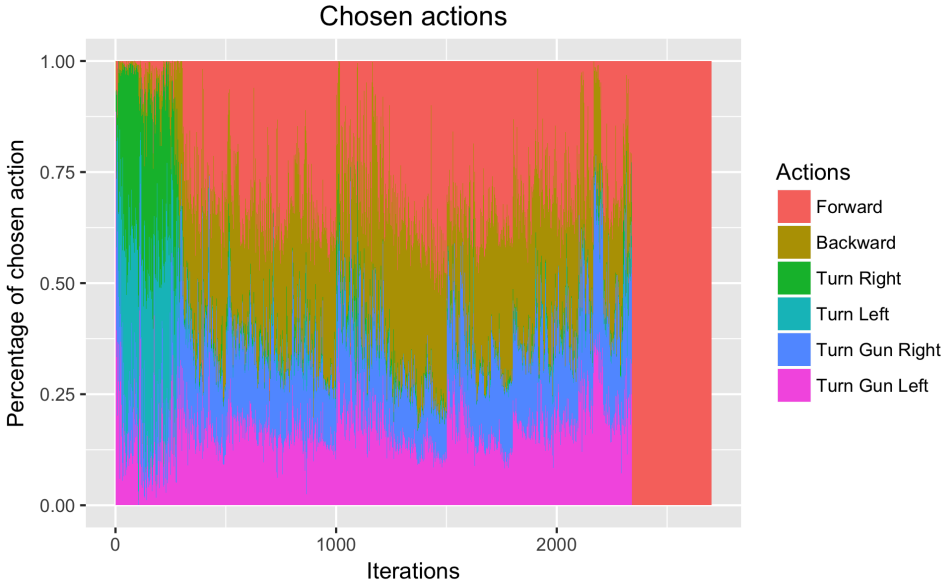


Figure 5.11: The percentage of the chosen actions during the training. We can see at the end that the network is always choosing the *Forward* action, but this is the default action. Actually, the network is dead and does not output anything.

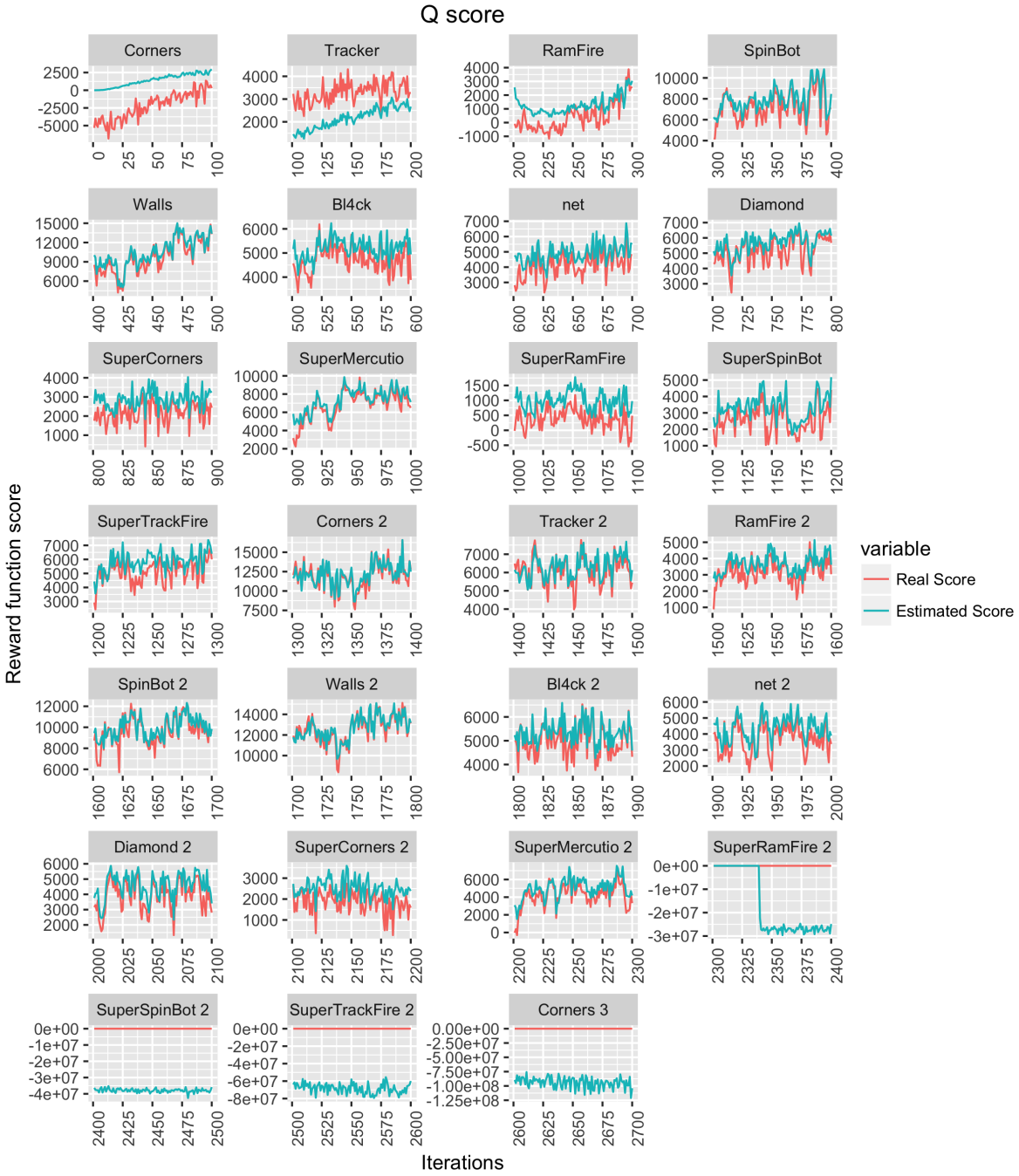


Figure 5.12: We can observe the positive progression during the early phase of the training, followed by stabilization and ultimately a crash at the end. Please note the scale of the score, which is due to the default value -10000 that is used to initialize the max function while evaluating the output of the network. Again, this does not represent the real score, but shows that the network does not output anything.

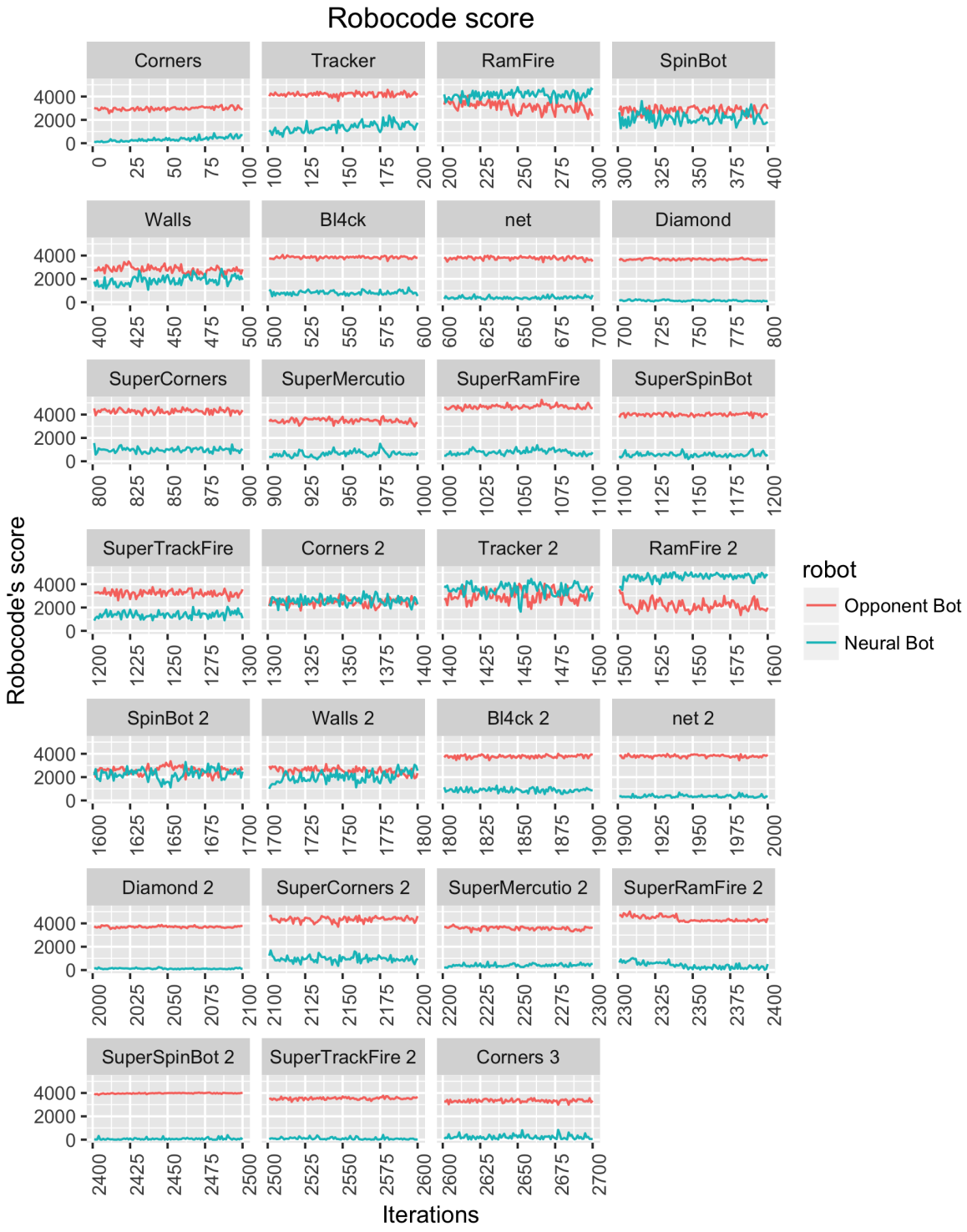


Figure 5.13: This score is less obvious regarding the crash of the neural net. We can see that the score vs the *Corners* 3 is significantly lower than the score on the *Corners* 2. Learning was correct during the first games, but it is hard to identify any real learning versus the competitive bots.

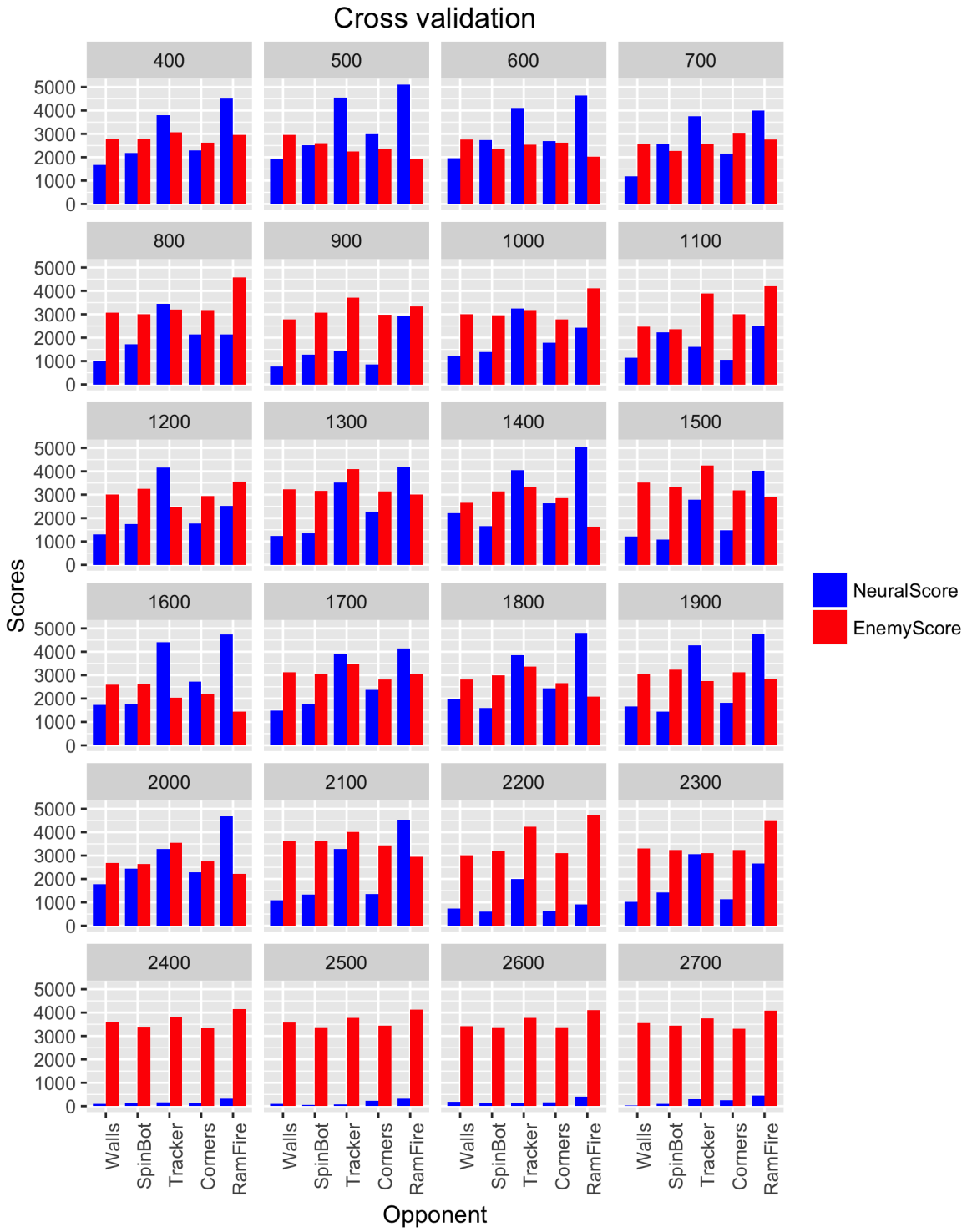


Figure 5.14: The crash of the neural net is obvious, especially in the last row. But this graph conveys much more information. We can see that the iterations 800 - 1100 are worst than the iterations 400 - 700. More precisely, it decays from iteration 500 which corresponds to the switch to very skilled opponents. The score increases again after the iteration 1200, which is the worst of the competitive bots, followed by the simpler ones. We can therefore assume that learning is at a low level if we play against an opponent which is too strong compared to our bot.

5.4.2 Training versus himself

Goal of this model. Train the bot against himself, and verify that it does not stay on a local maximum. We will also reduce the learning rate to avoid other crashes.

Settings. As we stressed in the previous subsection, training is not effective if the opponent is too strong. The optimal enemy should be a bot with exactly the same level than the neural bot. We will thus try the mirror match up, with a smaller learning rate (0.005 instead of the previous 0.01), still using the cross validation check to make sure that learning improves the result against multiple enemies, and not only against itself. The lower rate of learning value is not too significant since we made the hypothesis that the dying neurons were caused by both a high learning rate and a strong gradient which was induced by the difference of level between the two bots. The Q-score stays significant, but the R-score will not tell us much about the learning. We keep the same epsilon strategy, using the periodic spikes to explore more widely (figure 5.10).

We must note that there is a risk of a weak local maximum induced by the neural net playing against himself. Luckily for us, the AlphaGo article used this technique and it worked well for them, using a frozen previous state of the neural net playing against the new one. We will therefore consider this option in case of bad learning.

Results. After the learning, we achieved the best neural bot trained yet, so we will not use the frozen net as opponent. The problem of our actual bot is its inability to anticipate enemy movement. Even if the score (figure 5.15) does not clearly demonstrate an actual evolution after the iteration 1600, the visual verification allows us to see a better space management in the battlefield correlated with the iterations.

Conclusion. Let the neural bot play against itself is a great way to push the limits since we do not have the expected bad convergence. The complexity of the behaviors can increase and the opponent will always be as strong as the neural bot. Even better, when a new strategy is explored, if the strategy is good, we are sure that it will lead to a winning game. We also avoided the previous crash.

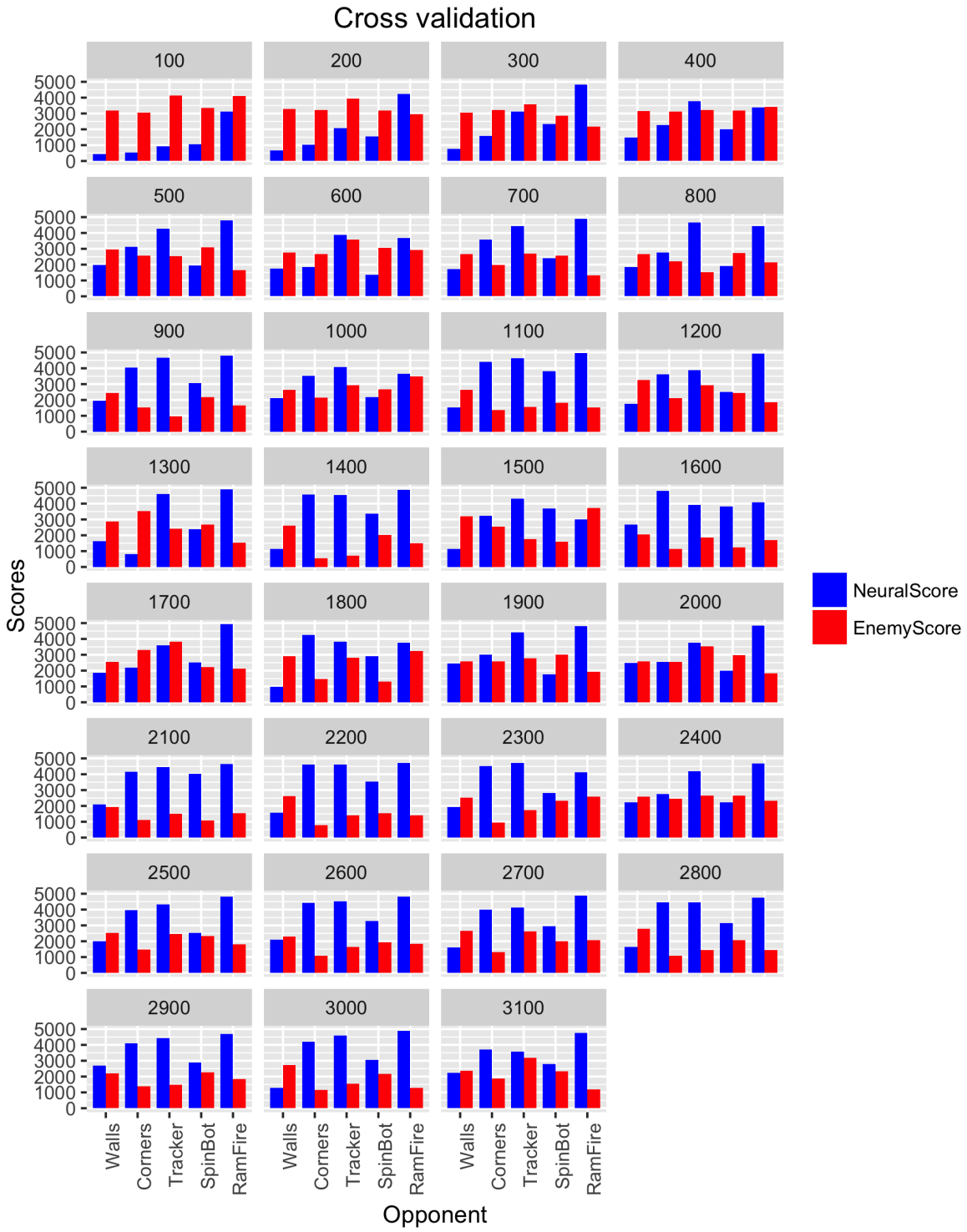


Figure 5.15: We clearly see an improvement during the first thousand iterations but it stays somehow steady afterwards. Visual verification indicates that the neural net is still learning, but more about space management in the battlefield than aiming precision. The neural bot demonstrates difficulty in beating the *Walls* sample bot, which is the only bot that actually needs movement prediction to kill. This is probably due to the lack of history in the input. Note that the neural bot still won sometimes against the *Walls* bot (iteration 2900, 2100, 1600)

5.4.3 Convolutional hybrid architecture

Goal of this model. Evaluate the hybrid architecture and the impact of using the log as additional input. We will also try to make it learn movement prediction.

Settings. For this training, we will try to use the convolutional layer to reduce the significant dimension of an input that includes the log which contains the five previous states. We will let the bot play against himself during the first 2500 iterations, then try to force him to learn movement prediction by letting it play the last 300 iterations against the *Walls* bot.

Results. The bot has a strong strategy, but still fails to aim preventively.

Conclusion. The hybrid model with the log are useful for the performances, the movement prediction is hard to learn.

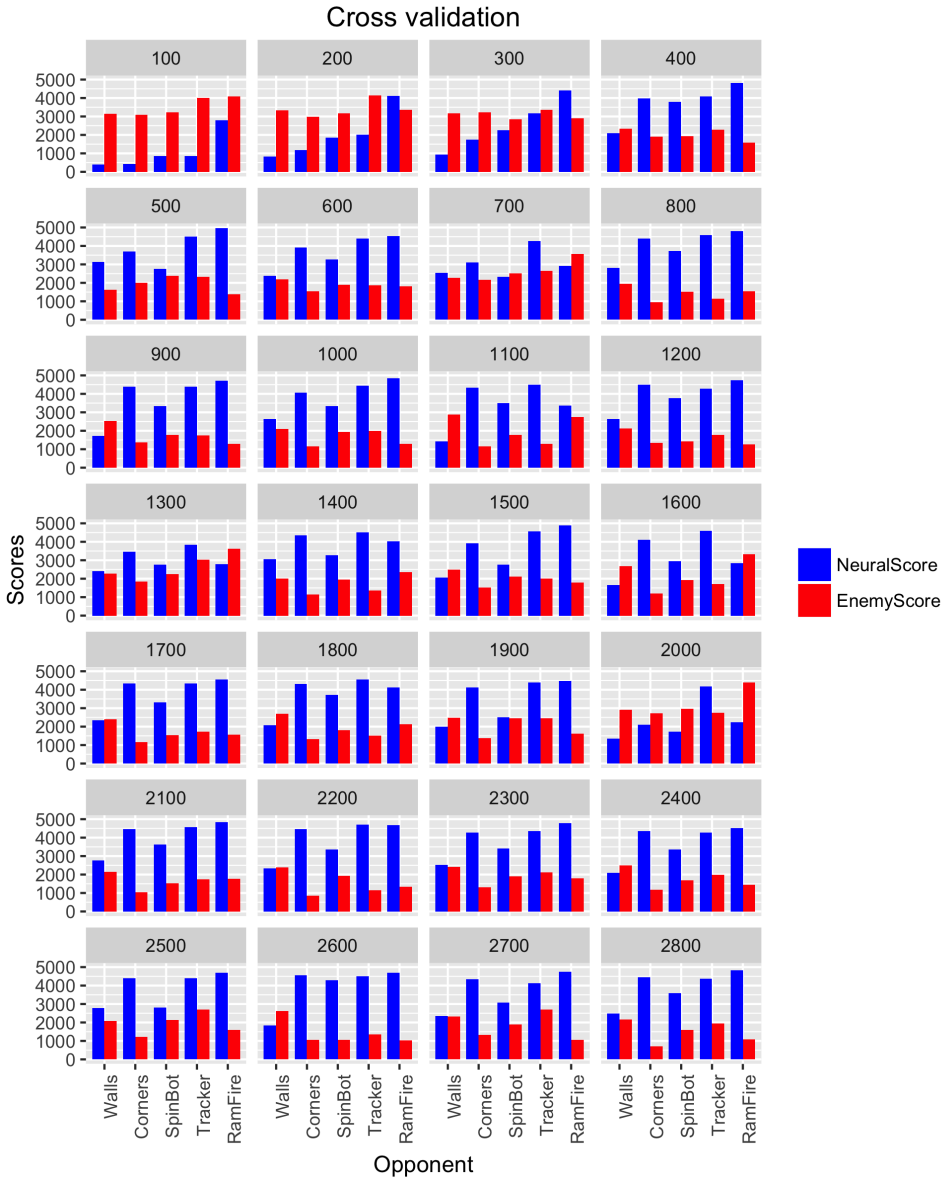


Figure 5.16: We can see that the neural bot beats all the sample bots as soon as iteration 500, even the *Walls* bot.

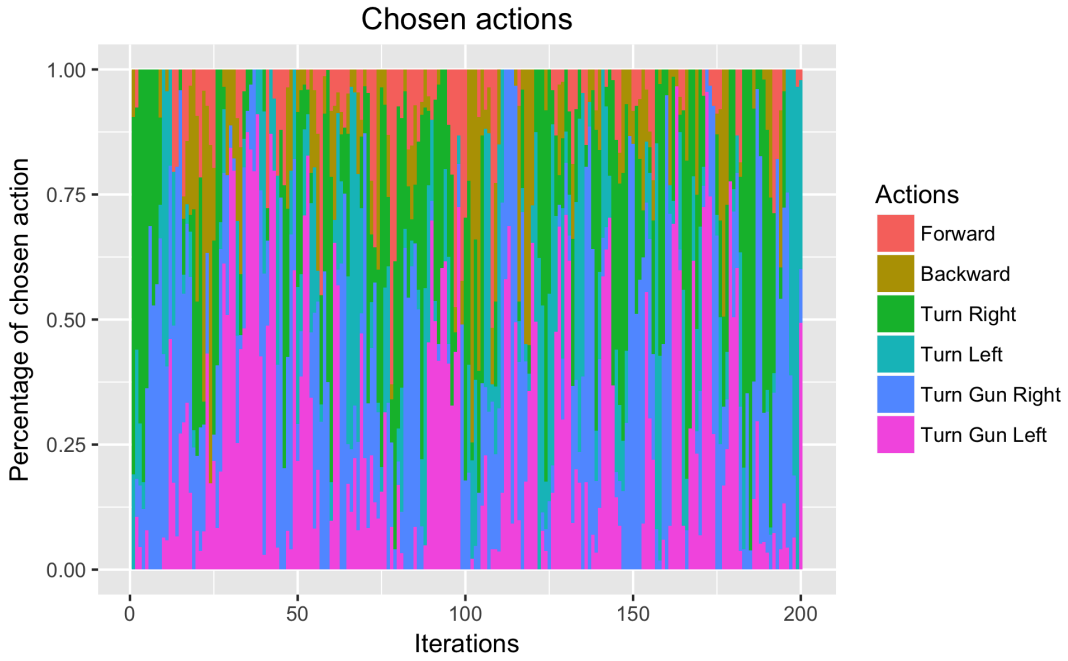
5.4.4 Convolutional - LSTM hybrid architecture

Goal of this model. Evaluate the impact of the LSTM layers added to our architecture.

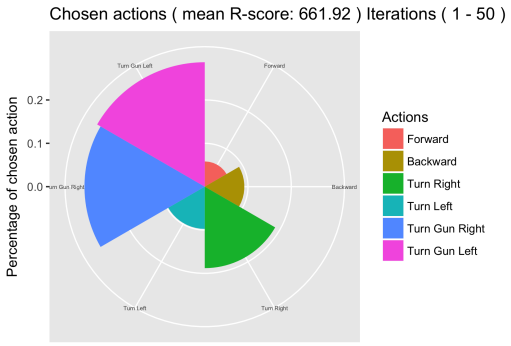
Settings. We will change the architecture of the network to try the impact of adding memory. This has the negative effect to force us to train the memories sequentially and thus stop the memory replay algorithm. This make the learning more unstable, and higher the risk to reach a bad local maximum. The LSTM layer has many parameters, and training of this kind of layer takes a lot of time since we need to update the weights in the previous time steps depending on the current mistake. We will first try against himself, copying the network to take care of the memory problem, then train him against the *Corners* bot in order to avoid the bad local maximum.

Results. Training LSTM layer is very time consuming since there are a lot of parameters to update, and we must go through the historic to update them. The 200 iterations took as long as the other previous training sessions (around 3 days and nights on my personal laptop). As shown in figure 5.17 there is no dominant behavior coming out of the exploration phase. A visual check on the neural bot stressed out that our bot was only playing one single action during all round long. This action changes after each round, which is also after each training of the network on a block of memory. The action remains independent of the current state of the game, which is what we feared while dropping the random memory replay algorithm. We can observe in figure 5.18 that the real Q score remains steady, and that the Robocode score is not better than the R score of a randomly playing bot.

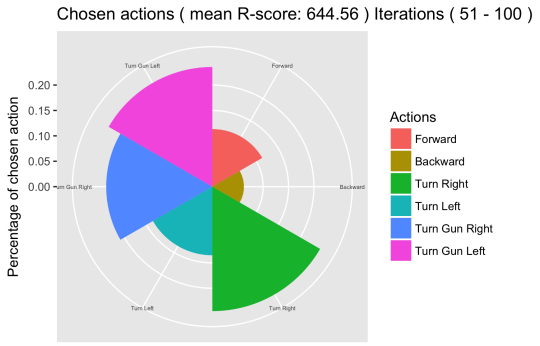
Conclusion. Dropping the random memory replay impact negatively the learning. We will thus not be able to train a model with memory. There is also a need for more computing power.



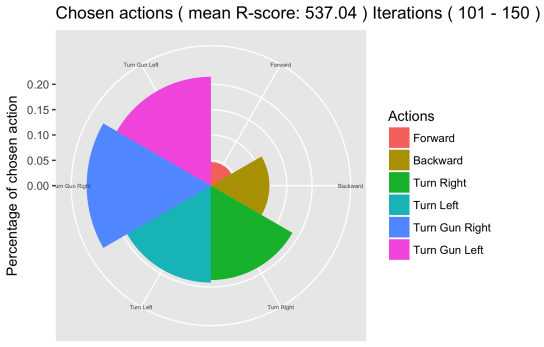
(a) Percentage of the chosen actions over the training iterations. Each iteration actually consist of 25 rounds times 300 turns. The network is trained with a random memory sample each turn.



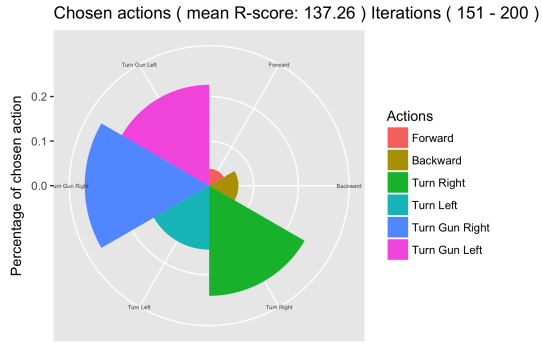
(b) The average percentage of the chosen actions over the first quarter of the training.



(c) The average percentage of the chosen actions over the second quarter of the training.

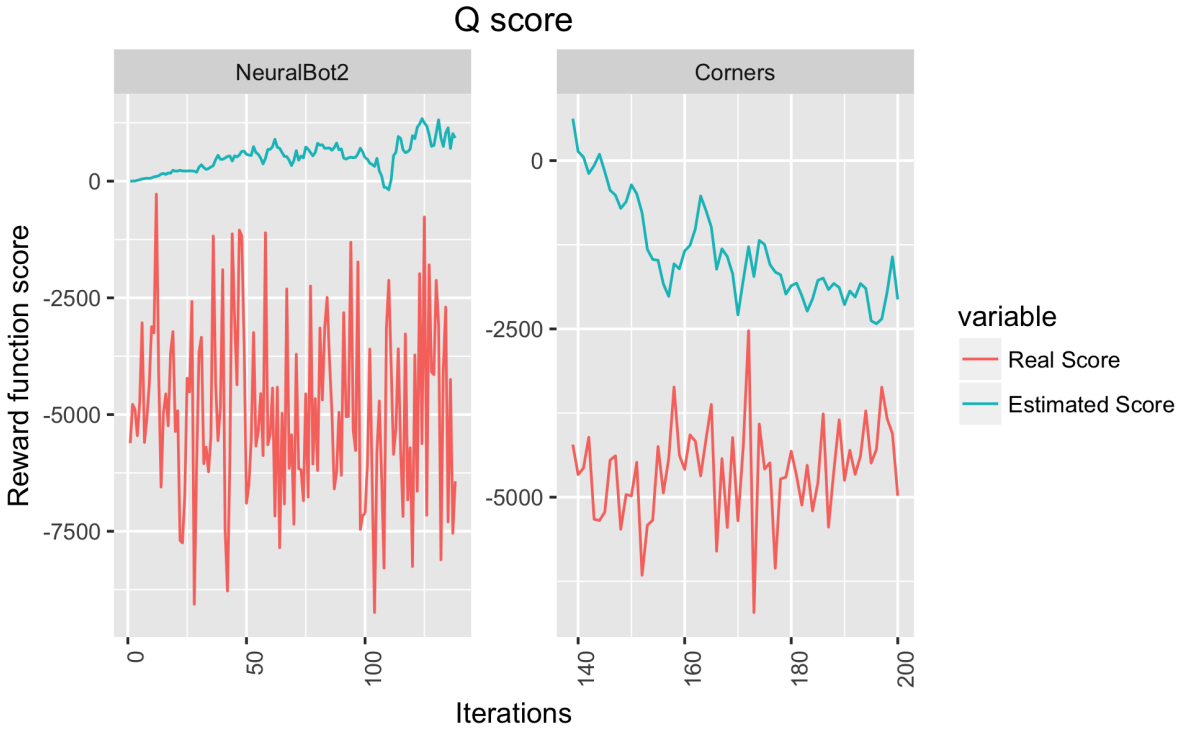


(d) The average percentage of the chosen actions over the third quarter of the training.

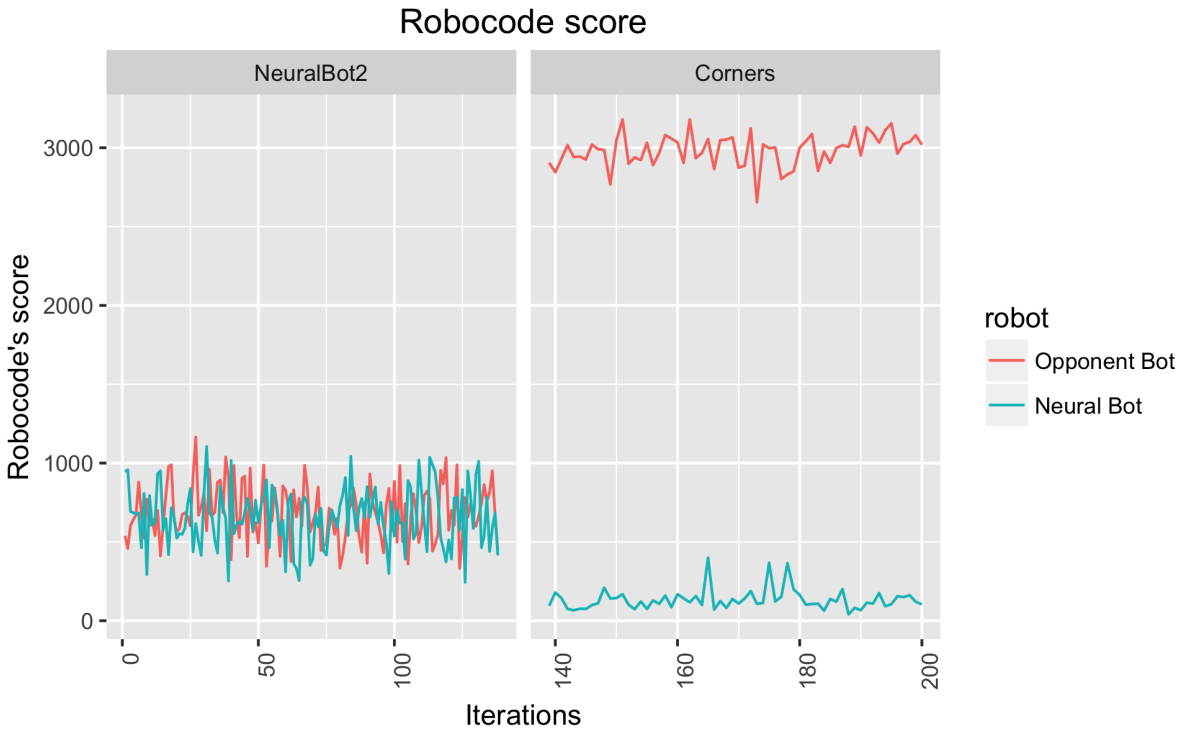


(e) The average percentage of the chosen actions over the fourth quarter of the training.

Figure 5.17: The actions are not well explored, and appears as if strategies the neural bot is actually performing is a single action during the whole round phase. The action may change from one round to another, but it remains a trivial behavior.



(a) The score of the reward function Q, true and estimated.



(b) The score of Robocode, for which the network has no information.

Figure 5.18: The real Q score does not increase over time, showing that no improvement was gained from the neural network point of view. The Robocode score shows a very poor performance versus the *Corners* bot, similar to a random player.

5.4.5 Convolutional hybrid with precise aiming

Goal of this model. Train the best model so far, using a computing server (32 cores).

Settings. We will now evaluate the convolutional hybrid network with the precise aiming actions, which raise the input size from 136 to 571 neurons, and thus the log input size will be 2855 neurons. The number of actions are also bigger, going from 6 to 10 since we let the bot turn the gun with three different angles steps. This of course slows learning down since there are many more parameters to update at each iteration. The need of more computing power came out from the previous simulation. We will thus now use the server instead of my laptop which has 2 cores. Computation parallelism has limits, and it is not worth to fully use the 32 cores since launching threads takes time. This training session took 137 hours on 8 cores that were computing at full power all the time. We will not use the GPU (Graphical processing units) since the operating system requirements of the library are not met.

Results. Figure 5.19 shows the difference between the Robocode score during the cross validation. It is a more condense way to represent the previous histograms. If the score is above 0, it means that the Neural Bot has beaten the enemy. This figure is special: learning is very slow at the beginning, then we can see that the bot has a mind switch, and actually starts to gain skill. It does not improve much against the three competitive bots (Diamond, net and Bl4ck) and still struggles to beat the *Walls* bot as well as the *SpinBot*. We can see in figure 5.20 that the bot has learn to dodge by always moving forward and backward, but it does not turn much (subfigure (c,d,e)) which make the bot poor at dodging when he is facing the opponent. It actually uses random actions to sometimes turn, but it is clearly not optimized. Another interesting fact is that the multiple angle steps are well used, not throwing useless steps away. We can see that the bot has found the main actions of the usual strategy by the iteration 300 even if the score increases at the 1500 iteration.

Conclusion. We probably arrived to the best possible score using these settings. There are many possible things to change, or more abstraction to do in order to continue to improve but this will be left for future works.

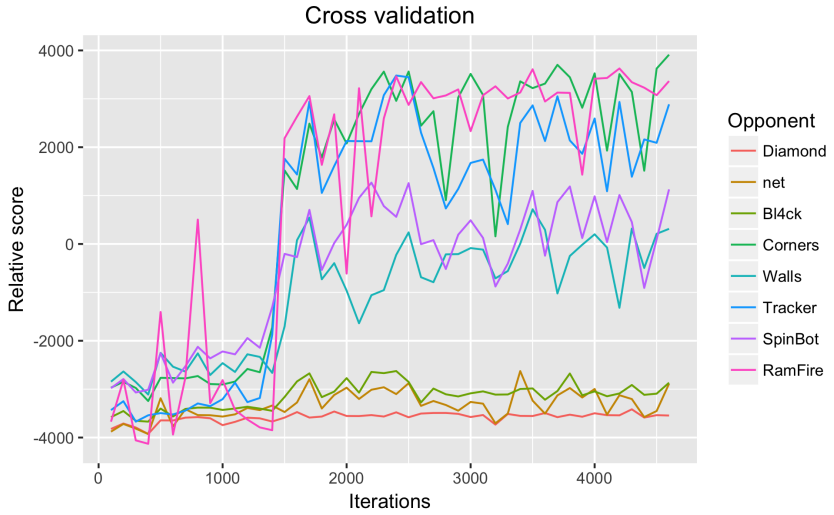
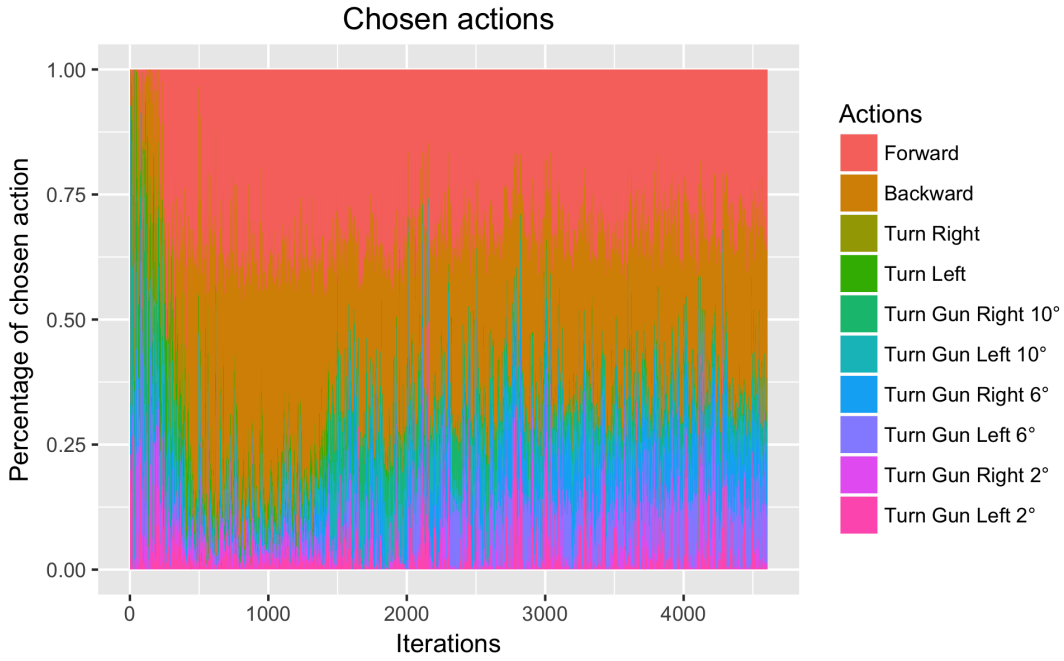
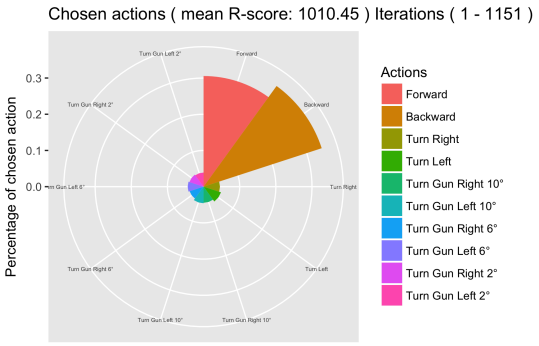


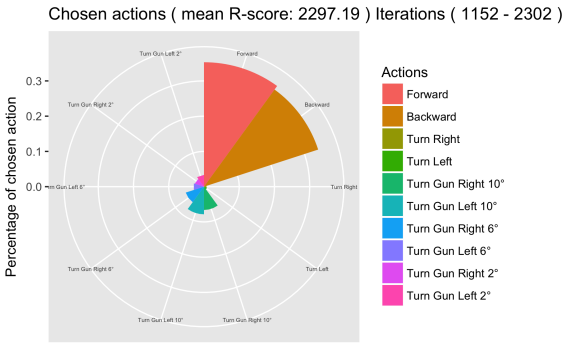
Figure 5.19: This figure shows the relative score between the the neural bot and different opponents. A score above 0 means that the neural bot won the battles. We can observe a mind switch around iteration 1500. The neural bot still struggles against the three competitive bots, and has difficulties against the *Walls* and *SpinBot* but consistently beats the other bots.



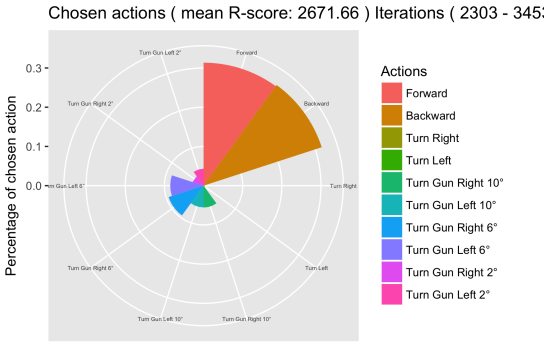
(a) Percentage of the chosen actions over the training iterations. Each iteration actually consist of 25 rounds times 300 turns. The network is trained with a random memory sample each turn.



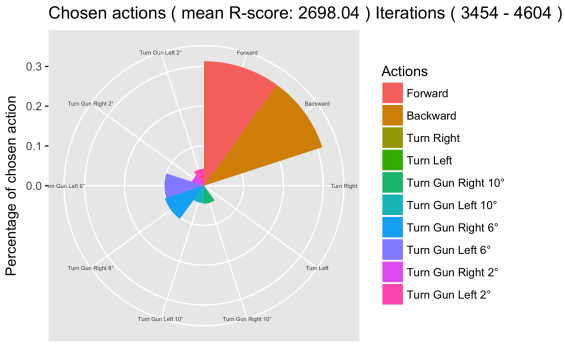
(b) The average percentage of the chosen actions over the first quarter of the training.



(c) The average percentage of the chosen actions over the second quarter of the training.



(d) The average percentage of the chosen actions over the third quarter of the training.



(e) The average percentage of the chosen actions over the fourth quarter of the training.

Figure 5.20: We can see that the bot has found the main actions of the usual strategy by the iteration 300. The bot still turns in the beginning of the training, even if those actions are not chosen later on. This makes a poor dodging strategy when the bot is facing the other bot (confirmed with a visual check).

5.5 Comparisons

We will now compare the model to each other. Figure 5.21 represent the evolution of the different models over the iterations. The geometric mean was used to compare the scores as described in the paper "How not to lie with statistics: the correct way to summarize benchmark results" [14]. We can observe that even if the deeper network crashes, it outperform some models until the iteration 1500. We could not have taken the relative score instead of the neural score because it would sometimes be negative, which is not suitable with the geometric mean.

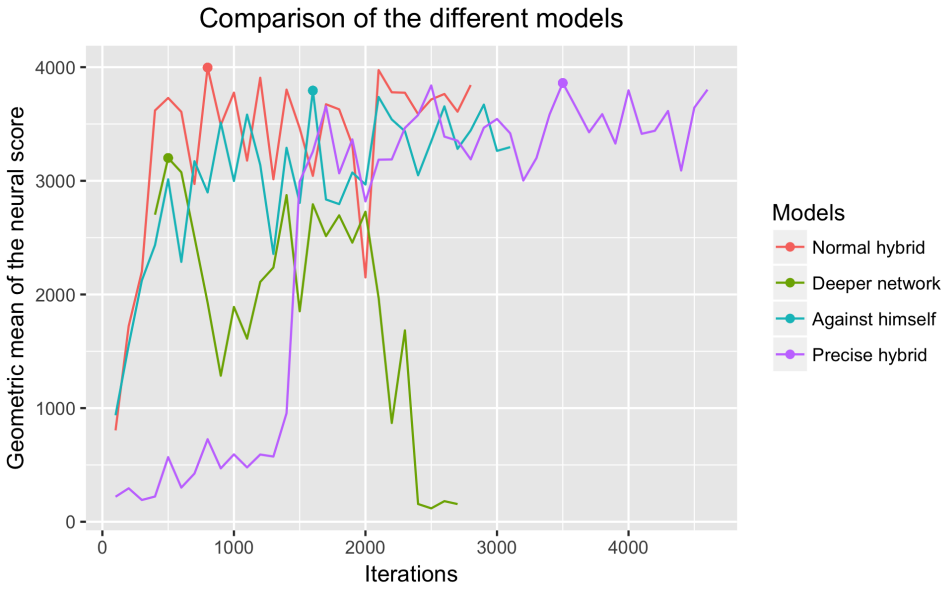


Figure 5.21: The convolutional-LSTM hybrid architecture was omitted because of its poor performances and short training. The y axis is the geometric mean of the neural score over the five sample bots. The dots correspond the the best iteration for each model.

Figure 5.22 focuses on the best iteration of each model. As we can see, the best results are not always made by the same model. The color of the histogram is an indicator of the time needed to train the network. The darker models have the advantage to be rapidly trained in comparison to the others.

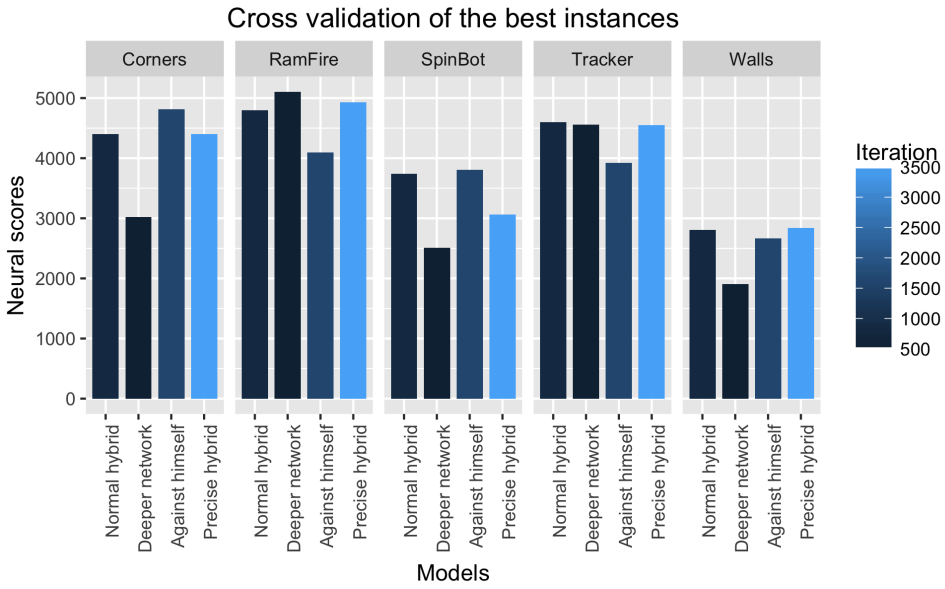


Figure 5.22: The neural score against the five sample bots for the best iteration of each model. We can see that there is not an obvious winner against every bot.

In order to find the best model, we need to compare each performance as a ratio over the best performance. Figure 5.23 stress out that the best model is not the precise hybrid, which took the most computing resources, but the first convolutional hybrid architecture.

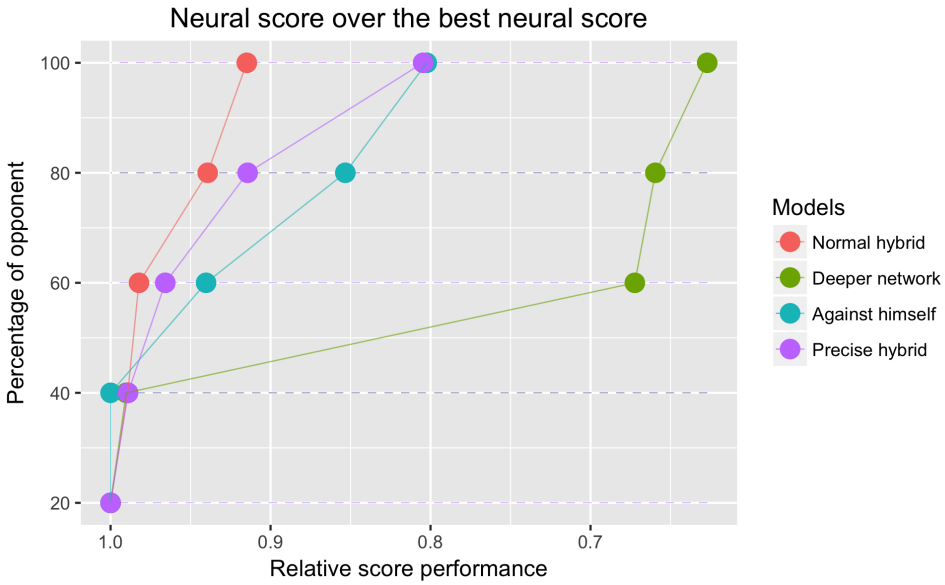


Figure 5.23: The relative score performance ordered by descending order is a good summarize of benchmarks. We see that all the models are the best in at least one instance, and the first mirror model is the only one to be the best in two instances. But the first hybrid architecture is the most relevant model since it remains consistent in the other instances.

5.6 Some statistics

In total, including the training sessions that were not discussed in the result chapter, 30632 iterations ran on my computer which equals approximately $30632 * 25 * 300 = 229.740.000$ training samples. As reference, the Atari article used 10 million frames for its training. The LSTM network had 64867 parameters that learn something at each training sample. The total number of parameter updates will use the half of that number to get a rough average with the other networks: $\frac{64867}{2} * 229.740.000 = 7.4512723 * 10^{12}$ which is 7 trillion or 7 million million parameter update. It takes around 10 minutes to compute and train an iteration of the LSTM neural network. This gives $\frac{(10*60*60)s}{25*300*64867} = 0.7399 \frac{ms}{parameter\_update}$. This makes the rough global computation time to $7.4512723 * 10^{12} parameter\_update * 0.7399 \frac{ms}{parameter\_update} = 551376000.74s = 106.36 days$ of consecutive computation time. This is of course not completely accurate, but it gives an order of magnitude.

6 | Conclusion

We know thanks to the Universal Approximation theorem that a feed forward network with only one layer containing a finite number of neurons can approximate any continuous function on compact subsets. This theorem was not mentioning the learning of the parameters, but now that we have the computational power on our side, the neural networks become an accessible and powerful tool. The deep Q-learning that allows to learn behaviors using this technology is fascinating by its performances. The fact that we start the training with a randomly initialized brain, and that by playing enough battle using rewards, it converges toward a well constructed neural network that is able to have strategy, is still surprising. The state of the art contains networks that are able to learn music, drawing, facial recognition, and so much more applications.

The problem is that we must have a good environment for the network in order to ease the learning. And in some cases, we can directly feed the image as input of the neural network as it has been done for the Atari games. But in other cases, like in Robocode, the image contains too much information; the enemy bullets must be invisible for the tank. Another problem is the reward function, when the action score dependence is not trivial, there is a real need of specific evaluation. The third problem is the action space. For the Atari games, the limited actions made their choice easy, propose all the actions to the network. But again, this does not work for Robocode since the vast action space makes the exploration hard to handle. The last problem is the question about the architecture of the network. How can we find the best network to approximate our complex Q-function?

In this master's thesis, we proposed a possible solution for all of those problems. We tried multiple input representations, and made changes in response to the mistakes that we could observe in the neural bot behavior. The reward function must try to contain information for any actions, in order to let the network learn directly from its exploration. It is not required thanks to the Bellman's equation, but it greatly improves the learning speed. The action space need restrictions in order to let the bot learn all the implications of each action. We had two choices here; implement meta actions that contains trivial behaviors (target + shoot + move around), or let the neural bot build itself the behaviors with the basic action (forward, backward and others). We developed the second choice because it does not limit the possibilities of play, but suffers from longer training time. We also evaluated the impact of the different neural network architectures; feed forward, deeper feed forward, convolutional hybrid, convolutional - LSTM hybrid.

The input choice resulted in a very complete representation, which took care of the rotational symmetry. The specific reward made the learning easy, but it might also be the reason of a lost of performances once the network has learned the basics. Since the reward gives feed back on certain conditions, the neural bot is train to meet those conditions instead of trained to win the game. Of course, the Q-score was correlated with the Robocode score, but not perfectly. The action space resulted in flaws for part of the behaviors. There was an aiming problem that could have been avoided using the meta actions option. But the basic actions were a good choice for the complexity of the resulting behaviors. The neural networks architecture analysis stressed out that the most complex model is not always the best. The best neural network was the convolutional hybrid model, which learned fast and stayed consistent against all the bots. The memory of the LSTM neurons forces us to train the network sequentially instead of using the random memory

replay algorithm.The convolutional - LSTM was not suited for the Robocode game since this lack of random memory replay made the neural bot fall in a negative local maximum.

This works demonstrate that complex behaviors with partially observable environment can be learned using the deep Q-learning algorithm. The architecture choice remains challenging since we do not have a heuristic for the number of layers, or number of neurons. The type of the layers can be easier to find if we have an idea of the patterns that we want to extract. We cannot compare our bot with the other public bot since we do not have access to their code, but all of them chose the meta action instead of the basic actions, with only one shooting option; toward the center of the enemy. This makes them incapable of aiming preventively, and thus makes them less dangerous against a dodging bot. This is why the neural bot probably outperforms them.

We sadly cannot evaluate our bot with the top 1000 to get a real rank since the loading of the neural network library and its dependent libraries takes more time that the allowed time per bot. The training is also done off time restriction, but it makes our bot illegal for tournaments since we are not allowed to have a long initialization. The last architecture using LSTM also need time at the end of each round to clear the memory. So we can argue on the real capabilities of the Neural bot since it still gets beaten by most of the top 1000 bots, but he has to play in a very limited action space compared to the huge freedom that they have.

It clearly was an option to have selected to use larger, deeper, more complex neural networks with longer training time and lower learning rates, but everything still is a function of time and computation power. As well as opening the action space, which was still small after all. It was appropriate within the imposed time constraints. The Adam updater was not completely tested by lack of reward compared to the invested time, but it could have been more developed. Most of the computation has been done on my personal computer, but there is a spark section in the DL4J library which allows to distribute the calculation on the cloud. There are two reasons why it was not done. It takes significant resources of both time and funding. A appropriate compromise was to use the computation server Ada, which has 32 cores instead of the 2 cores of my personal computer. It was used to train the final network. Training multiple small networks specialized in one behavior, and train a bigger one telling which small neural network to use given the situation could have been a good extension of this thesis.

Glossary

battle In Robocode, a battle defines multiple rounds, and is used by the game to make a better comparison between two bots. 5–7, 25–27, 29, 54

bot Software agent that performs intelligently. 3, 4, 6, 23–28, 33, 39, 41, 46, 47, 49–51, 54, 59–61, 71

CNN Convolutional neural network. Network adapted for large input processing. 11, 13

ϵ Epsilon: percentage of randomly chosen actions. 17, 21

FNN Forward neural network. Network without cycle. 10, 11, 13

γ Gamma: discount factor used in the evaluation of future rewards. 8, 15

LSTM Long short-term memory. Network with internal memory, used for short or long term memory applications. 9, 14, 15, 17–19, 26, 36, 37, 54, 62, 69–71

policy Rule that the agent follows in selecting actions, given the state it is in. 1, 8, 9, 21–23, 72

Q-function Function that takes the state and the action as input, and output the Q-value. 3, 16, 70, 72

Q-learning Algorithm used to collect experience data by playing a game following some policy. 1, 3, 5, 8, 9, 15, 16, 20, 21, 32, 39, 70, 71

Q-network Network used to evaluate the Q-function. 1, 3, 8, 28, 31

Q-value Quality of an action. 9, 16, 17, 23, 28, 32, 72

ReLU Rectified linear unit activation function. The image of this activation function $\in [0, \rightarrow [$ and it does not suffer of the vanishing gradient problem. 9, 10, 33, 35–37, 54

RNN Recurrent neural network. Network with cycle, used for shot term memory applications. 13–15, 17, 18, 20

round In Robocode, a round defines a fight until one of the two tank dies. 6, 7, 24, 26, 42, 44, 46, 47, 49, 52, 63, 66, 72

σ Sigmoïd activation function. The image of this activation function $\in [0, 1]$ and it suffers of the vanishing gradient problem. 9, 15

tanh Hyperbolic tangent activation function. The image of this activation function $\in [-1, 1]$ and it suffers of the vanishing gradient problem. 9, 10, 14, 15

turn In Robocode, a turn defines the time during which the robot has to choose his next action. 6, 26, 42, 44, 47, 52, 63, 66

Bibliography

[1] Cs231n convolutional neural networks for visual recognition. http://cs231n.github.io/neural-networks-3/#sgd. (Accessed on 04/13/2017).

[2] Game physics - robowiki. http://robowiki.net/wiki/Robocode/Game_Physics. (Accessed on 03/21/2017).

[3] Robocode home. http://robocode.sourceforge.net/. (Accessed on 03/21/2017).

[4] Jimmy Ba, Volodymyr Mnih, and Koray Kavukcuoglu. Multiple object recognition with visual attention. *arXiv preprint arXiv:1412.7755*, 2014.

[5] Yoshua Bengio, Nicolas Boulanger-Lewandowski, and Razvan Pascanu. Advances in optimizing recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8624–8628. IEEE, 2013.

[6] Yoshua Bengio, Patrice Simard, and Paolo Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[7] Seth Bling. Mari/o - machine learning for video games. https://www.youtube.com/watch?v=qv6UVOQ0F44. (Accessed on 04/27/2017).

[8] ArmoredSandwich's channel. Genetic algorithm and robocode - youtube. https://www.youtube.com/watch?v=Hp6bhARBGc4. (Accessed on 04/27/2017).

[9] decaynoctu. Robocode - scouting for target with q-learning - youtube. https://www.youtube.com/watch?v=RjkK4ODjMT0. (Accessed on 04/27/2017).

[10] Dl4j. Convolutional networks in java - deeplearning4j: Open-source, distributed deep learning for the jvm. https://deeplearning4j.org/convolutionalnets.html. (Accessed on 03/10/2017).

[11] DL4J. How to visualize, monitor and debug neural network learning. https://deeplearning4j.org/visualization. (Accessed on 05/16/2017).

[12] DL4J. How to visualize, monitor and debug neural network learning - deeplearning4j: Open-source, distributed deep learning for the jvm. https://deeplearning4j.org/visualization. (Accessed on 04/01/2017).

[13] Ruben Fiszel. Reinforcement learning and dqn, learning to play from pixels - ruben fiszel's website. https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html. (Accessed on 03/06/2017).

[14] Philip J Fleming and John J Wallace. How not to lie with statistics: the correct way to summarize benchmark results. *Communications of the ACM*, 29(3):218–221, 1986.

[15] Morten Gade, Michael Knudsen, Rasmus Aslak Kjær, Thomas Christensen, Christian Planck Larsen, Michael David Pedersen, and Jens Kristian Søgaard Andersen. Machine learning to robocode. (Accessed on 04/21/2017).

[16] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation by google deepmind. `https://www.youtube.com/watch?v=Zt-7MI9eKEo`. (Accessed on 05/04/2017).

[17] Karol Gregor, Ivo Danihelka, Alex Graves, Danilo Jimenez Rezende, and Daan Wierstra. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*, 2015.

[18] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[19] Andrej Karpathy. Convnetjs deep q learning reinforcement learning with neural network demo. `https://cs.stanford.edu/people/karpathy/convnetjs/demo/rldemo.html`. (Accessed on 03/27/2017).

[20] Andrej Karpathy. Cs231n convolutional neural networks for visual recognition. `https://cs231n.github.io/`. (Accessed on 03/10/2017).

[21] Andrej Karpathy. Deep reinforcement learning: Pong from pixels. `http://karpathy.github.io/2016/05/31/rl/`. (Accessed on 03/27/2017).

[22] Andrej Karpathy. The unreasonable effectiveness of recurrent neural networks. `http://karpathy.github.io/2015/05/21/rnn-effectiveness/`. (Accessed on 03/10/2017).

[23] Diederik Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[24] Flemming N. Larsen. Readme of robocode. `http://robocode.sourceforge.net/docs/ReadMe.html`. (Accessed on 03/21/2017).

[25] Steve Lawrence, C Lee Giles, Ah Chung Tsoi, and Andrew D Back. Face recognition: A convolutional neural-network approach. *IEEE transactions on neural networks*, 8(1):98–113, 1997.

[26] Feifei Li. Convolutional neural networks for visual recognition. `http://ibcs1-wd.wikispaces.com/file/view/cs231n-full.pdf`. (Accessed on 04/20/2017).

[27] Tambet Matiisen. Guest post (part i): Demystifying deep reinforcement learning - nervana. `https://www.nervanasys.com/demystifying-deep-reinforcement-learning/`. (Accessed on 03/10/2017).

[28] Avi Mishayev. Genetic programming in robocode - youtube. `https://www.youtube.com/watch?v=HGL_J6c0Nj8&index=2&list=PLuMAL1QPhstufXzEtwG8Xj8Mso6oAUawn`. (Accessed on 04/27/2017).

[29] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[30] Gade Morten, Knudsen Michael, Rasmus Aslak Kjær, Christensen Thomas, Planck Larsen Christian, Pedersen Michael David, Kristian Jens, and Andersen Søgaard. Applying machine learning to robocode. `http://www.dinbedstemedarbejder.dk/Dat3.pdf`, decembre 2003. (Accessed on 03/21/2017).

[31] Jon Nielsen. Robocode - evolving the perfect tank with artificial intelligens - youtube. https://www.youtube.com/watch?v=Ru9W-9CxdQ8. (Accessed on 04/27/2017).

[32] Christopher Olah. Understanding lstm networks – colah's blog. http://colah.github.io/posts/2015-08-Understanding-LSTMs/. (Accessed on 03/10/2017).

[33] Andrew Oustimov and Vincent Vu. Artificial neural networks in the cancer genomics frontier. http://tcr.amegroups.com/article/view/2647/html. (Accessed on 04/30/2017).

[34] Mengye Ren, Ryan Kiros, and Richard Zemel. Exploring models and data for image question answering. In *Advances in Neural Information Processing Systems*, pages 2953–2961, 2015.

[35] David Silver. Deep reinforcement learning - videolectures.net. http://videolectures.net/rldm2015_silver_reinforcement_learning/. (Accessed on 03/10/2017).

[36] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[37] Steven Spielberg. Github - stevenpjg/qlearningrobocodenn: Trains a robot in a robocode using q-learning and neural networks. https://github.com/stevenpjg/QlearningRobocodeNN. (Accessed on 04/21/2017).

[38] Steven Spielberg. Reinforcement learning of a robot with functional approximation using a neural network - youtube. https://www.youtube.com/watch?v=qZd7ptXNIkI. (Accessed on 04/21/2017).

[39] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. http://nn.cs.utexas.edu/downloads/papers/stanley.ec02.pdf. (Accessed on 04/27/2017).

[40] Ilya Sutskever, James Martens, George E Dahl, and Geoffrey E Hinton. On the importance of initialization and momentum in deep learning. *ICML (3)*, 28:1139–1147, 2013.

[41] Gerald Tesauro. Td-gammon, a self-teaching backgammon program, achieves master-level play. *Neural computation*, 6(2):215–219, 1994.

[42] theberkeleyview. Adam: A method for stochastic optimization. https://theberkeleyview.wordpress.com/2015/11/19/berkeleyview-for-adam-a-method-for-stochastic-optimization/. (Accessed on 04/13/2017).

[43] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *AAAI*, pages 2094–2100, 2016.

[44] Wikipedia. Feedforward neural network - wikipedia. https://en.wikipedia.org/wiki/Feedforward_neural_network. (Accessed on 03/10/2017).

[45] Wikipedia. Markov decision process - wikipedia. https://en.wikipedia.org/wiki/Markov_decision_process. (Accessed on 03/10/2017).

Rue Archimède, 1 bte L6.11.01, 1348 Louvain-la-Neuve **www.uclouvain.be/epl**