M2 Mathématiques, Vision et Apprentissage

École Normale Supérieure de Cachan

Deep Reinforcement Learning and Forward Modeling for StarCraft AI

Internship Report

Alex Auvolat

Under the supervision of:
Gabriel Synnaeve
Nicolas Usunier

May 2016 – September 2016

Facebook AI Research
Facebook France
6 rue Ménars, 75002 Paris

Contents

1 Introduction

As part of my Master 2 MVA degree (Mathematics, Computer Vision, and Machine Learning), I had to complete a four-month internship in a research lab on a machine learning-related topic. As I wanted to continue investigating Deep Learning following up on my previous internship [1] and also work on more recent topics combining Reinforcement Learning and Deep Learningi [15], the proposed internship on StarCraft AI at Facebook AI Research was a perfect fit for me. I was accepted for a four month internship that ran from mid-May to mid-September 2016, under the supervision of Gabriel Synnaeve (now at FAIR New York) and Nicolas Usunier.

During this internship, my main topic was forward modeling for StarCraft, i.e. building Deep Learning models that are able to predict the next state of a StarCraft game given the current state and player's actions. This work was done as a continuation of previous work at FAIR on Reinforcement Learning for simple StarCraft micro-management tasks, with the aim of improving the performances of previous RL models.

In Section 2, I will introduce the basis of neural networks construction and optimization. In Section 3, I will talk about the StarCraft micro-management task, and how game states can be featurized and fed to neural networks. In Section 4 I will talk about Deep Reinforcement Learning (DRL) and previous work on DRL for StarCraft AI. In Section 5 I will talk about my work on forward models, and on improving DRL performances using forward models.

2 Deep Learning Basics

Deep Learning is a family of machine learning models and architectures based on the principles of neural networks (sometimes abbreviated *neural nets*), which were first proposed in the 50's under the name of Perceptron [19]. Deep Learning is a field defined by the studying of artificial neural net models containing several layers of neurons, although the term now includes research on many practical applications of neural networks, not necessarily deep but that have a large number of neurons, and therefore have become practical only recently with the explosion of available computing power concentrated on a single system (typically, a GPU).

Deep neural networks are statistical models that can be adapted and applied to many *supervised learning* tasks, such as *classification* or *regression*. Deep neural networks can also be used as *generative* models, in which case they are trained to produce output such as sentences or images. They can also be used in various *unsupervised* settings, such as forward modeling, i.e. predicting the next state of a system given its current state. Finally, deep neural networks have recently found new applications in *reinforcement learning* via deep Q-learning and have been used to solve simple video games.

2.1 Neural Networks and Backpropagation

Artificial Neurons Artificial Neural Networks (ANNs) are originally inspired by the functioning of biological neurons, hence their name; however neural nets currently in application use a very simplified model of the computation realized by a biological neuron. In the ANN model, a neuron is a simple unit connected to several real-valued inputs (in particular, the inputs to a neuron may be the output of other neurons), and does a sum of the inputs multiplied by weights (sometimes called *synaptic weights*), followed by a bias and a nonlinearity. Mathematically, an artificial neuron is defined by the following equation:

$$f(x_1, \ldots, x_n) = a(w_1 x_1 + \cdots + w_n x_n + b)$$

where $(w_1, \ldots, w_n, b)$ are the parameters of the model, and a is a nonlinear function called an *activation function*. Traditional activation functions include the sigmoid function σ defined by $\sigma(x) = \frac{1}{1+e^{-x}}$, or the tanh function. More recently, the rectified linear unit (ReLU) defined by $a(x) = max(x, 0)$ has become a common choice as it has been shown to improve learning on certain type of deep or recurrent models [5]. The output value of a neuron is usually called its *activation value*, or simply its *activation*.

Artificial Neural Networks An ANN is a function that can be calculated by a directed acyclic graph of nodes, where each node is either an input node or a neuron, and one or several neuron nodes are identified as output nodes. An ANN is a parametric function, as the exact computation is defined by the set of weights and biases of all the neurons of the network. In this report, as often in literature, we will refer to the vector of these parameters as θ. The topology of the network, i.e. the number of neurons and their connections, is not part of the parameter vector, as it is not something we can optimize automatically by learning. The topology of the network, as well as other values such as the choice of activation function or the learning rule, are referred to as *hyperparameters*, and finding a good hyperparameter combination is usually done by hand search guided by intuition.

Neuron Layers Neurons are typically grouped in *layers*, meaning that the computation of the function defined by the neural net is defined as the composition of the functions defined by each layer. A neural net layer is typically fully-connected, meaning that each neuron of the layer is connected to each of the inputs of the layer. The computation done by the layer can then be expressed as a matrix multiplication, followed by the addition of a bias vector, and finally an element-wise application of the activation function. The definition in terms of matrix multiplication has been an important element in the acceleration of neural net computation, thanks to very efficient BLAS routines implemented on GPUs. The vector of outputs for a layer of neurons is usually called an *activation vector*, or sometimes a *state vector* in the case of recurrent neural networks. A layered architecture of this kind is usually called a *multi-layer perceptron* (MLP).

For image processing, another type of layers is commonly used: *convolutional neural networks*. Such layers are defined as a set of convolution filters applied on a 2D input. Highly optimized implementations of convolutions have also benefited Deep Learning research in these areas and has led to state of the art performances on image recognition tasks using these methods.

Loss Function To do supervised learning with deep neural networks, we must first define a cost function in relation to our training set. If our training set $\mathcal{X}$ is composed of training examples (x_i, y_i), where x_i is the input and y_i is the output we wish our model to predict, then the loss function (also called cost function) for a neural net f_θ is defined by:

$$\mathcal{L}(f_\theta) \quad = \quad \sum_{(x_i, y_i) \in \mathcal{X}} l(f_\theta(x_i), y_i)$$

where l is a loss function that can be defined in various manners depending on the type of task we want our model to solve. Typically for regression we will use an L_2 distance, whereas for classification we will use the negative log-likelihood of the prediction target.

The goal of machine learning is to find a set of parameters θ that minimizes the *expected loss* (or *risk*), i.e. the expectancy of $l(f_\theta(x), y)$ where (x, y) are drawn from a true-world distribution which is also supposed to have generated the training set $\mathcal{X}$.

Optimization In supervised learning, the optimization of the risk is done by optimizing the loss on the training set, which is considered to provide a good first approximation. In mathematical terms, it corresponds to finding θ^* defined by:

$$\theta^* \quad = \quad \text{argmin}_\theta \mathcal{L}(f_\theta)$$

As the parameter vector θ^* may easily be too specific to the training set, and provide a function that does not actually minimize the risk very well (a phenomenon called *overfitting*), several techniques called *regularization methods* have been developed. Such methods include the addition of a regularization term to the loss function, and the monitoring of the cost on a separate validation set which is not used during the training.

Backpropagation and Gradient Descent The method used predominantly in deep learning for searching for the optimal value of the parameters is gradient descent, where we improve the parameter vector θ by small steps following the gradient:

$$\theta_{t+1} \quad = \quad \theta_t - \lambda \nabla \mathcal{L}(f_{\theta_t})$$

where $\nabla \mathcal{L}(f_\theta)$ is the vector of the gradient of $\mathcal{L}(f_\theta)$ with respect to the parameter values θ. As we often work on very large datasets and calculating the gradient for the full loss function is very impractical, we usually proceed by *stochastic gradient descents*, where the gradient is approximated at each step by summing the loss over only a small batch of examples (called a *minibatch*). Practical minibatch sizes are of about 100 examples.

The gradient with respect to the parameters θ can be analytically derived from the computation graph of the neural network by using the chain rule and known analytical derivatives of simple functions. Its effective calculation involves propagating the gradient backwards from the output (where the gradient is proportional to the loss) to the parameters, hence the name of backpropagation [20].

Long-Term Dependencies and Vanishing Gradients In very deep neural networks, or in recurrent neural networks – which can be unrolled in time and seen as extremely deep neural networks – we are usually confronted to the problem of vanishing or exploding gradients [3, 17]. This problem consists of gradients that fail to properly assign credit over chains of many nonlinearities, due to getting mixed with gradients coming from irrelevant sources. Several techniques have been proposed to alleviate this problem, the most famous being Dropout [9], which consists in randomly removing a certain proportion of neurons during training time, making the network in effect *smaller* and gradients more focused, and Batch Normalization[11], which consists in renormalizing the values output by each neuron so that in average each neuron outputs values of mean 0 and variance 1. The use of Rectifier Linear Units [5] is also known to alleviate the vanishing gradient problem.

Tools Various frameworks exist to enable the rapid implementation of Deep Learning models that exploit the capability of modern GPUs for massively parallel computation. The codebase for the StarCraft AI project at FAIR is written in Torch [1], which is one

[1] `http://torch.ch`

of the big Deep Learning frameworks and uses Lua as a scripting language. As I had to extend the previous codebase, Torch and Lua were the primary tools I used during this internship.

Other frameworks for Deep Learning exist, such as Theano [2], TensorFlow [3] and MXNet [4]. These frameworks work with symbolic computation graphs and provide automatic differentiation at the operation level, which is not the case of Torch that only provides a set of predefined layers with backward pass implemented independently for each of them.

2.2 Convolutional Networks.

Architecture Convolutional neural networks are a specific neural net architecture which exploits the 2D structure of the input data. In a ConvNet, each neuron of a convolutional layer is only connected to input nodes that are in the neighbourhood of that neuron. Also, a convolutional layer is composed of several feature maps which each compute a single function at all the possible locations of the input. The function calculated by one feature map can therefore be expressed as a convolution of the input with a given filter.

In most ConvNets, convolutional layers are separated by downsampling layers, which reduce the size of the input and aggregate information spatially. The downsampling layers are commonly mean-pooling or max-pooling layers, but more recently simply skipping a certain number of convolutions on each dimension (convolution striding) has become a common choice.

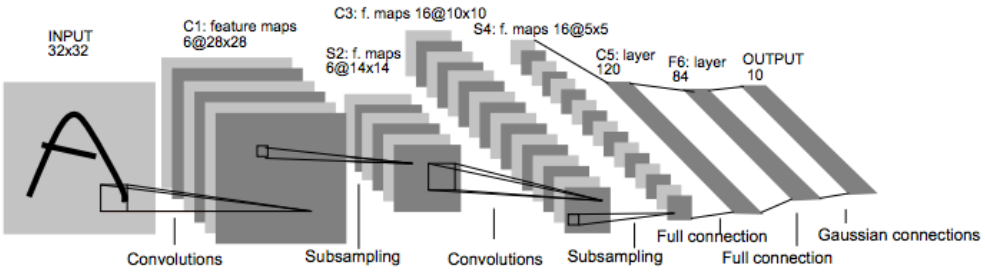

Figure 1: Architecture of LeNet-5, a Convolutional Neural Network for digit recognition, as shown in [13]. Each plane is a feature map, i.e. a set of units whose weights are constrained to be identical.

Usage Such convolutional neural networks are naturally adapted to image processing, where we want to build image features which are invariant by translation. They have remarkably performed state-of-the-art performance on image classification challenges such as ImageNet [12]. Figure 1 shows the architecture of LeNet-5, one of the first convolutional neural networks that was used for hand-written digit classification. Convolutional neural networks have also achieved very good performance on speech recognition tasks, and are an increasingly popular choice for character-level natural language processing.

[2] `http://deeplearning.net/software/theano/`
[3] `http://tensorflow.org`
[4] `http://mxnet.io/`

Figure 2: An epic Protoss vs. Zerg battle unravelling in *StarCraft: Brood War*.

3 StarCraft Problem and Featurization

3.1 Presentation of StarCraft

Task Definition StarCraft is a classic real-time strategy (RTS) video game, where two players confront one another and must build an army and command it in order to destroy the opponent's army. There are roughly two levels of strategy in StarCraft play: macro-management, which is the task of gathering resources and building a strong economy that will be able to sustain an army strong enough to defeat the opponent, and micro-management which corresponds to the management of individual units or groups of units (such as workers, soldiers, vessels, etc.) in order to win battles.

StarCraft as a Benchmark for AI StarCraft came out in 1998 and was for a long time the most played RTS game in pro competitions, before StarCraft II came out. It isn't played so much by pros anymore, however it has become a benchmark task for developing RTS AI. [16] is a relatively recent (2013) survey of approaches to hand-crafted AI for RTS video games such as StarCraft. As far as we know, there is no published work on using Deep Learning and Deep Q Networks for solving StarCraft. To this day, StarCraft remains an open problem in AI, as no computer program is currently able to beat the best human champions. It is commonly admitted that solving StarCraft would mark great progress in the field of artificial intelligence, as a good StarCraft player needs to combine many different skills and must be capable of strategic planning on both short-term and long-term timescales. Further difficulty is added by the fact that not all information about the opponent's play is revealed, and the player must be able to plan under high uncertainty and make guesses about the opponent's strategy.

Our Work In all the work done at FAIR on StarCraft, we only tackle a small portion of the problem: we restrict ourselves to single battle micro-management tasks. The simplest

scenario is 5 marines vs. 5 marines: in this scenario we are able to train a DRL model that consistently wins against StarCraft's built-in AI. Restricting to such a simple scenario vastly simplifies the problem: all the game information is known, all units have the exact same characteristics, and the action set contains only two basic actions (moving and attacking). Other small scenarios we are working on include more units (15v16) or different unit types (Vultures, Zealots, Dragoons, ...), however we weren't able to achieve consistent victory against the built-in AI on these harder scenarios, whereas a good human player would beat them easily.

3.2 Relevant Game Information

State Space The state s of a StarCraft game at time t consists of a set of units $(u_1, \ldots, u_{|s|})$, which each have the following properties:

- The team the unit is on (ally/enemy).
- The type of the unit, which defines its capabilities.
- The position of the unit. In StarCraft positions can be expressed in pixels, however in our work we always round positions to integer walktiles, with 1 walktile = 8 pixels. The StarCraft map is two-dimensional, therefore a unit position is a couple (x, y) of walktile coordinates.
- The velocity of the unit, which is subject to laws of acceleration for which we do not have an exact model.
- The unit's hit points (HP), which decreases when the unit is attacked. A unit has a given number of hit points when it spawns, and it dies when its hit points reaches 0. Some units have special "healing" abilities that can restore hit points to other units, however most of the time units do not recover lost hit points.
- The unit's weapon cooldown (CD), which is the time before the unit can use it's weapon again. The cooldown is reset to a high value (for instance, 15 frame for marines) when the unit attacks, and the unit may attack again when the cooldown reaches zero.
- The unit's attack value, i.e. the damage inflicted upon attacking an enemy unit.
- The unit's range, which is the maximum distance for a target it may attack.
- The unit's armor value, which reduces the amount of damage taken on attacks.

Units may be upgraded by spending resources on researching new technology. Upgraded units might have higher attack values, more armor, or more hit points. All these values are relevant for micro-management decision-making.

Action Space The time is divided in discrete frames, which happen roughly at the speed of 25 frames per second (FPS). At each frame, a unit might take an action. Possible actions include:

- Moving to a target position
- Attacking an enemy unit

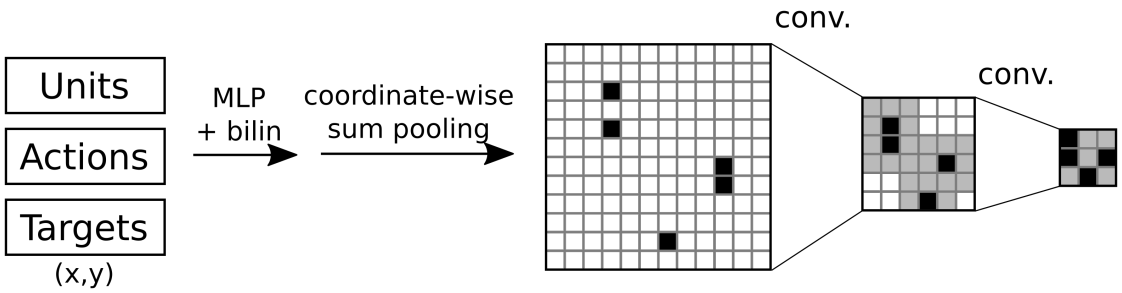

Figure 3: Featurization of StarCraft state for a convolutional neural network.

- Using defensive or offensive technologies, such as heals, shields, zone attacks, etc.

The current action of enemy units is also important for decision making, and can be considered part of the state. Since we want either to predict the future, which is a function of our current actions, or to give the value of $Q(s, a)$ where a is the set of our own actions, the model also needs a way of taking our own actions as inputs.

Orders vs. Commands In StarCraft, the low-level details of what units are doing is described in terms of orders. Orders may be moves, attacks, resource harvesting, standing by, etc. However these orders are much more precise than the actual commands a player can give units: a command may develop into several orders which are executed automatically by the game. A portion of my work has been to feed the detailed orders into the model so that it has a maximum of information about what is going on. This has required me to implement a partial conversion from commands to orders, as the action space for the RL task is in terms of commands and not orders.

3.3 Convolutional StarCraft Neural Network

Method Several ways of feeding all this information into a neural network can be conceived. Since the map has a 2D structure, it seems logical to use a convolutional neural network at some point in the model, which will be easily able to model zone effect, as well as collision dynamics between nearby units. Figure 3 shows the architecture of the neural network I have designed for this task. On the 2D map which is the input of the ConvNet, we want the following information to be present:

- On the walktile where a unit is, we need information about that unit.
- On the walktile where a unit is which is taking an action, we need information about that action.
- On the walktile which is the target of an action (e.g. the destination of a move or the unit which is the target of an attack), we need information about the action and the unit which is performing the action and is on another walktile.

For instance, if we have two ally units at positions $(4, 3)$ and $(10, 7)$, and both are attacking an enemy unit at position $(4, 5)$, the following information need to be fed into the network:

As several feature vectors might need to be combined on a single walktile, we cannot simply put these information on the map. In my architecture, the feature vectors are first

(x, y)	Type	Meaning
$(4, 3)$	Unit	Ally Terran Marine present here, 40 HP
$(4, 3)$	Action	Ally Terran Marine here attacking at $(+0, +2)$, 0 cooldown
$(10, 7)$	Unit	Ally Terran Marine present here, 12 HP
$(10, 7)$	Action	Ally Terran Marine here attacking at $(-6, -2)$, 5 frames cooldown
$(4, 5)$	Unit	Enemy Terran Marine present here, 25 HP
$(4, 5)$	Target	Ally Terran Marine attacking here from $(+0, -2)$, 0 cooldown
$(4, 5)$	Target	Ally Terran Marine attacking here from $(+6, +2)$, 5 frames cooldown

Figure 4: Feature vectors for a simple example state

transformed through a MLP independently of their position, which gives a representation in a features space that is learned by the network. Then, when several feature vectors must appear on the same walktile, the representations are simply summed by a pooling operation. In effect, this pooling operation takes as input the matrix of these intermediate representation as well as the associated 2D positions, and outputs a 2D image which contains zeroes where no information exists, and the sum of the intermediate representations on all cells where something is happening. Standard convolutions might then be applied on the 2D map, as shown in Figure 3.

The input of the MLP consists of some scalar features such as HP or cooldown, which are divided by a fixed value beforehand so that the typical value is of the order of 1, and some categorical values such as the unit type or action type, which are first embedded in a fixed-dimension vector space following the technique for word embeddings introduced in [2].

Benefits There are other ways of feeding the StarCraft state space into a neural network. Another such way is described below in Section 4.3 and has been successfully applied to reinforcement learning for micro-management tasks. There are many reasons to prefer convolutional networks over other methods, that all basically boil down to the fact that keeping the 2D structure of the game makes the featurization more straightforward and enables the neural net to exploit that structure optimally. In particular we do not need to explicitly encode the relative positions of units: their positions are directly visible on the 2D map. Having all the units appear on a 2D image also enables efficient handling of collisions between nearby units, as well as complex actions which have areas of effect (for instance a Protoss High Templar's Psi Storm).

4 Deep Reinforcement Learning for StarCraft

4.1 Presentation of Deep Q-Learning

Introduction Deep Q-Learning is a technique that has been recently introduced [14, 15] and which combines Q-learning, which is a classical form of reinforcement learning [24, 25], with deep neural networks. Since Q-learning and reinforcement learning was not the main topic of my internship, I will only do a quick introduction to Q-learning here.

Markov Decision Processes A Markov Decision Process is a way of formalizing the interaction between an agent and its environment, which provides the agent with a state transition function and rewards which are a function of its actions. It is defined as a tuple $(S, A_s, \mathbb{P}(s'|s, a), R(s, s'), \gamma)$, where:

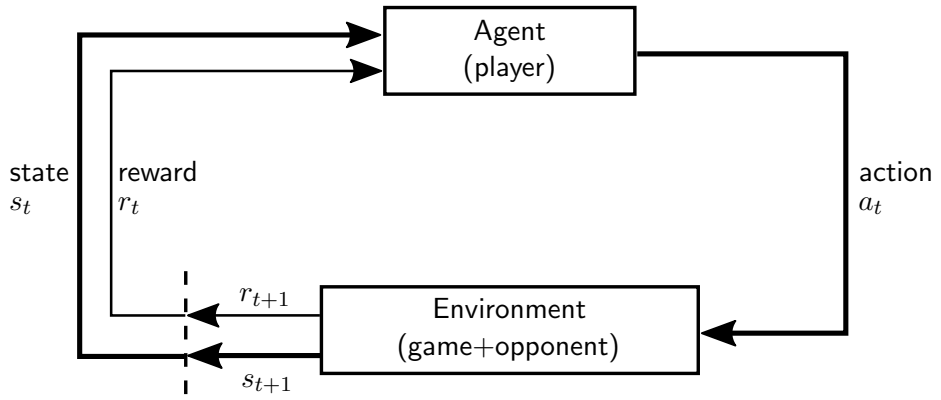

Figure 5: Illustration of a Markov Decision Process

- S is the space of all possible environment states
- A_s is the space of all actions the agent can possibly take when it is in state $s \in S$
- $\mathbb{P}(s'|s,a)$ is the probability of going into state s' when performing action a in state s.
- $R(s,s')$ is the reward provided to the agent by the environment upon transitioning from state s to state s'.
- γ is a discount factor, defining the *cumulative reward* of a series of states $(s_t)_{t=0}^{\infty}$:

$$R_{tot} = \sum_{t=0}^{\infty} \gamma^t R(s_t, s_{t+1})$$

In such an environment, the goal of the agent is to take at any time step the action that will maximize the expected cumulative reward. The state-action-reward loop is illustrated in Figure 5.

Bellman's Equation To solve the MDP, the agent must implement a value function $V(s)$ and a policy $\pi(s)$ which respect the following equations, known as Bellman's Equation:

$$\begin{aligned} \pi(s) &= \arg\max_a \left\{ \mathbb{E}_{s' \sim \mathbb{P}(\cdot|s,a)} \left[R(s,s') + \gamma V(s') \right] \right\} \\ V(s) &= \mathbb{E}_{s' \sim \mathbb{P}(\cdot|s,\pi(s))} \left[R(s,s') + \gamma V(s') \right] \end{aligned}$$

Bellman's Equation can be rewritten in the following form by introducing the Q function:

$$Q(s,a) = \mathbb{E}_{s' \sim \mathbb{P}(\cdot|s,a)} \left[R(s,s') + \gamma \max_{a'} Q(s',a') \right]$$

The policy $\pi(s)$ then has the much simpler form:

$$\pi(s) = \arg\max_a Q(s,a)$$

Q-learning In the case of a finite state space S and a finite action space A, we can apply simple tabular Q-learning, which consists in keeping a table of the values of $Q(s,a)$ for all states s and actions a. This table is initialized to a constant value. Then, traces are collected by some exploration method. At each timestep $s \xrightarrow{a} s'$, we can apply the following update to our Q function table:

$$Q(s,a) \quad \leftarrow \quad Q(s,a) + \alpha \left(R(s,s') + \gamma \max_{a'} Q(s',a') - Q(s,a) \right)$$

where α is the learning rate. This iterative improvement, called *temporal-difference learning*, updates $Q(s,a)$ by slightly moving it towards a more complete approximation using the current reward and observed next state.

ϵ-Greedy Exploration To collect traces $(s_1, s_2, \dots)$ used for Q-learning, we must chose an exploration policy. The most studied exploration policy is ϵ-greedy exploration, which consists in taking a random action with probability ϵ and the currently estimated best action $\arg\max_a Q(s,a)$ with probability $1-\epsilon$. This enables sufficient exploration, as all possible state-action pairs are tried with enough time, yet enables proper exploitation of the learned Q function to minimize the regret during learning. Common improvements to this simple technique include ϵ-annealing, i.e. decaying the value of ϵ over time so as to maximize exploitation once the Q function starts to converge to a good solution.

Deep Q Networks In the case of very large state spaces or action spaces, such as when the state is defined by an image (e.g. in Atari 2600 the state is what is displayed on screen), tabular Q learning becomes impossible. In such cases we define the $Q(s,a)$ function as a neural network Q_θ defined by a set of parameters θ that maps (s,a) to the corresponding Q value. The $Q(s,a)$ neural network can be trained by stochastic online gradient descent: suppose at time t we wish to update the parameters θ_{t-1} with the experience tuple (s_t, a_t, s_{t+1}) that was just collected, we will use the following loss function to optimize θ_t:

$$\begin{aligned} \mathcal{L}_t(\theta) &= (Q_\theta(s_t,a_t) - y_t)^2 \\ y_t &= R(s_t, s_{t+1}) + \gamma \max_a Q_{\theta_{t-1}}(s,a) \end{aligned}$$

For simple stochastic gradient, the update rule would be the following:

$$\theta_t \quad = \quad \theta_{t-1} + \alpha \cdot \nabla_\theta \mathcal{L}_t(\theta_{t-1})$$

where $\nabla_\theta \mathcal{L}_t(\theta_{t-1})$ is the gradient with respect to the parameter vector θ of the loss function $\mathcal{L}_t$ at the point θ_{t-1}. This update rule can be updated with a more complex update rule, such as Nesterov momentum or RMSProp (many others exists, but these are common choices).

Experience Replay The simple algorithm we just presented has a drawback that is that it requires new experience tuples to be collected by ϵ-greedy exploration for learning to occur. Moreover each experience tuple is used only once, which seems to be a waste. [14] introduced *experience replay*, which consists in collecting many experience tuples, and

learning on a subset of these tuples sampled uniformly, instead of learning only on the last collected tuple. This allows learning of the $Q(s, a)$ function to occur independently of the collecting of experience (even though collecting more experience will lead to a bigger dataset and therefore less overfitting on that experience. This also allows the constitution of minibatches of experience tuples containing information from different traces, therefore better approximating the average situation which we want our Q function to be able to handle. These methods have been shown to attain superhuman performance on many Atari 2600 video games, as shown in [15].

4.2 A Q Function for StarCraft

Greedy Action Selection In StarCraft, each unit must select an action at each frame. The number of units varies throughout the game as new units spawn and others die in battle. Most of the actions are durative, meaning that they will take several frames to execute. Many are also cyclic, i.e. they automatically repeat if no other action is given – a unit will keep attacking its target as long as we don't command it to do something else.

We cannot possibly explore all possible combination of actions for all the units, as the number of units and number of possible actions per units can be of several hundred in the most complex cases. We restrict ourselves to *greedy* action selection, where we iterate through units (in a random order) and select an action for each of them. The action selected by a unit may thus depend on the actions of the units that came before and whose action is already selected, but must be done independently of the action selection for the next units.

To implement this, we have a Q function that gives the Q value for the action of a single unit. It takes as an input: a state s which describes the state of the game and also contains the actions taken by the enemy units and by those of our units that have already selected an action, and an action a which is a legal action for the unit we are currently evaluating.

Frame Skipping Calculating a Q value for each unit for each order on each frame of the game is extremely costly, so we don't do the calculation on each frame and instead skip a constant amount of frames between each frame we select new actions. A common frame skip value is 9 frames (about 1/3 of a second), however smaller values perform better.

ConvNet StarCraft Model With the feturization and convolutional network described in Section 3.3, it is very easy to build a ConvNet made to evaluate the Q value of an action taken by a unit. The architecture I have designed is shown in Figure 6: the input to the model is the state and the actions of the units that have already chosen their action, plus a candidate action for the current unit. We then look at the pixels corresponding to the position of that unit in the convolutional layers, i.e. just after the 2D pooling step and in the output of each convolutional layer. By concatenating the representations present on all layers at this position, we obtain a *pixel column* which contains information about that unit (in the lower layers) as well as its surroundings (in the higher layers). We also extract the pixel column at the position of the target, if the action we are evaluating is a move or an attack action with a target at a different position. By concatenating these two pixel columns, we obtain a vector of a constant length, which we then feed into a simple multi-layer perceptron (MLP), which has a single output neuron giving the approximated Q-value.

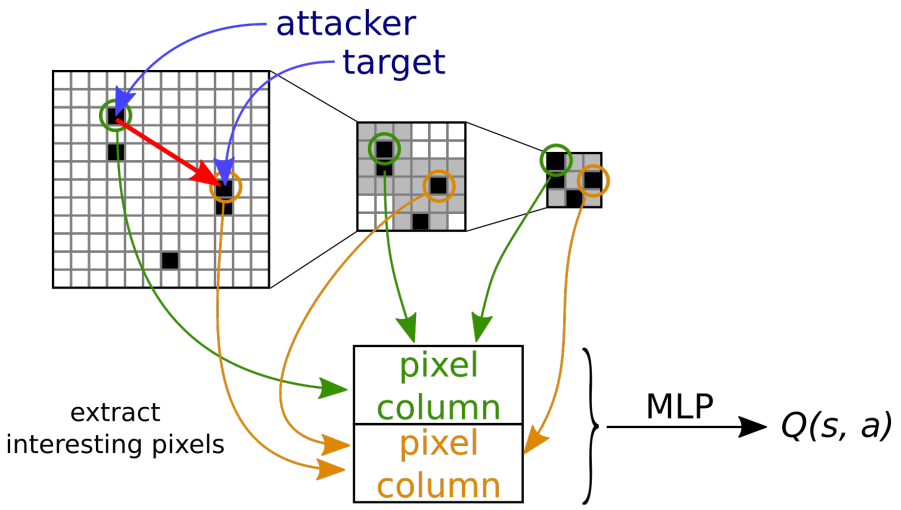

Figure 6: Evaluating $Q(s, a)$ for an attack action.

4.3 Previous Work

All my work is based on the previous work done at FAIR on the StarCraft problem, which is described in [22]. By plugging into that framework I have benefited from a lot of already written code, in particular the StarCraft to Torch bridge and the optimization methods. In this section I will outline what these contributions are, but please refer to [22] for the full details.

StarCraft to Torch Bridge A library called TorchCraft was developed at FAIR for interfacing StarCraft and Torch. It consists of a Windows DLL and of a Lua library. The DLL is injected in the StarCraft program and communicates with the Lua library through a network socket, making it possible to run StarCraft in its native Windows environment and the learning algorithm on a Linux system. The library allows inspection of the game state and action selection at each frame for each unit.

Previous Featurization and Model A previous model, which was designed before I came to FAIR, does not exploit the 2D structure with convolutional neural networks. Instead, it encodes each unit by is relative position to the unit whose action we are currently evaluating, to the potential target, and to the previous target of the current unit if there is one. This gives a constant-length feature vector for each unit which contains positional information. Each of these vectors is then passed in an MLP, and the obtained representations are then pooled (by a simple summation operation). The pooled value passes through a second MLP which outputs the approximated Q-value.

Standard Optimization Methods An implementation of simple deep Q-learning with experience replay is already available. There is also an implementation of policy gradient optimization, which is a simple gradient approximation method for online policy learning. A concise description of the policy gradient method can be found in [18]. Experiments have found that policy gradient performs similarly or worse than deep Q-learning.

Zero-Order Optimization Method An important contribution of [22] is the zero-order gradient approximation method, which has significantly improved the performances on the StarCraft micromanagement task. The basic motivation for this method is that during an exploration episode (i.e. a battle), we want to use a single determinstic policy. The reason for this is that if we use a stochastic exploration policy such as ϵ-greedy exploration, the actions taken by the different units at a single frame might be uncorrelated and inconsistent, never learning group strategies such as focus-firing. With the zero-order method we are able to explore the policy space instead of the action space, and backpropagate approximates of gradients with respect to the cumulative reward achieved by a single, consistent policy.

4.4 Results

Test Scenarios All the training and testing is done against StarCraft's built-in AI. Our performance measure is the win-rate againt the built-in AI. Training and testing has been done on the following scenarios:

- `m5v5`: a simple scenario where we control 5 Marines against an army of 5 Marines. The optimal strategy in this case is focus-firing, i.e. having all our units attack the same enemy unit. For example attacking the weakest enemy unit (with tie-breaking) yields good results.
- `m15v16`: this scenario is similar to `m5v5` except that we have 15 Marines and the enemy has 16. To win this scenario, units must focus-fire by groups of 6 or 7 units targeting a single enemy unit. Having all 15 units attacking the same enemy results in a waste of firing power ("overkill"), and ultimately a waste of time which often results in defeat.
- `w15v17`: in this map we control Wraiths instead of Marines. The essential difference is that Wraiths are flying units and therefore do not collide with one another (i.e. there may be several Wraiths on the same walktile, which is impossible for Marines).
- `dragoons_zealots`: we now have two kinds of units (Dragoons and Zealots), which require a different kind of management. Details can be found in [22].

Baseline Heuristics To check whether the models we build perform well, we compare them to hand-crafted baselines that apply a simple straight-forward rule to determine the action units must take. The different baselines are the following:

- *Attack closest* (`c`): all our units focus their fire on the enemy unit which is closest to the center of mass of our army.
- *Attack weakest and closest* (`wc`): attack the weakest enemy unit, and use distance to the center of mass of our army for tie-breaking.
- *Random no change* (`rand_nc`): each of our units chooses a random target unit and doesn't change target until either one dies.
- *No overkill, no change* (`nok_nc`): same as attack weakest and closest, but make sure that we don't overkill units, i.e. that we don't waste our firing power by having too many units targeting the same enemy.

	baselines				relative positions			convolutional
map	c	wc	rand_nc	nok_nc	Q	PG	ZO	ZO
m5v5	.85	.96	.55	.84	.99	.92	1.	.97
m15v16	.78	.10	.00	.68	.13	.19	.79	.00
w15v17	.11	.01	.15	.12	.16	.14	.49	.08
dragoons_zealots	.49	.82	.14	.50	.61	.69	.90	.87

Table 1: Results on the RL task. The values shown are win rates against the built-in AI, evaluated on 1000 battles. Q stands for Q-learning, PG for Policy Gradient and ZO for Zero-Order

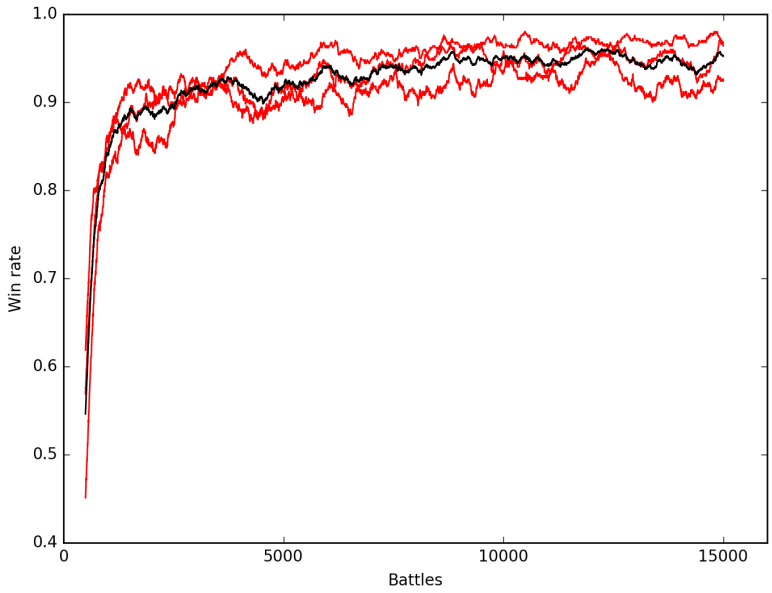

Figure 7: Training curve of our convolutional model on the m5v5 map. Each red curve corresponds to one training trajectory, the black curve is the mean. The plotted values are the win rate over the last 500 battles. The x-axis corresponds to the number of battles played by the algorithm.

Results Table 1 shows the results of the reinforcement learning experiments, in comparison with the baseline algorithms. Results for the baselines and with the relative positions model are taken from [22]. The results with the convolutional model (last column) are those that I have obtained during my internship.

Figure 7 shows the training curve for some of our convolutional models on m5v5, for three different models and with a learning rate of 0.001. The convolutional model works almost as well as the relative positions model on the simplest of all maps, m5v5, but doesn't attain a perfect score, where the relative positions model does. It also attains a honorable score on dragoons_zealots, which is not a very hard map. However, we observe that it fails to learn the strategies m15v16 and w15v17 which are more complex than simple focus-firing.

Analysis The poor results we obtain on this task are probably due to the difficulty for the model to disentangle important factors in the input data. The most important data

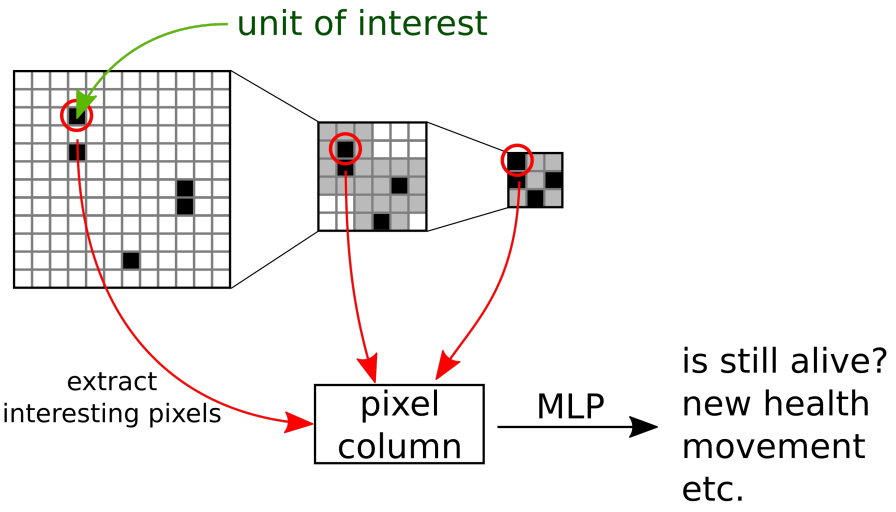

Figure 8: Predicting the next state from the convolutional neural network.

is that of the currently evaluated unit and of its target, which is input into the network in the same manner as the data of other units. Selecting the pixel columns of the unit and of the target may not be sufficient to precisely identify the features which might be useful for action selection. Also, a pure reinforcement learning does not provide us with much information as we observe a single reward signal at each timestep. Learning such a complex network from such a thin signal seems a complicated task to begin with.

5 Forward Modeling for StarCraft

5.1 Definition and Method

Predicted Values The forward model I implemented looks at the future at a fixed time horizon, which could correspond to the skip-frame value for reinforcement learning. Most of my experiments are for prediction 8 frames in the future, i.e. about one third of a second in the future. The values the model is trained to predict are the following:

- For each unit, is it still alive?
- For each unit which is still alive, how much hit points does it still have? Or alternately, how many hit points has it gained/loosed since previous frame?
- For each unit which is still alive, what is the new cooldown value for its weapon?
- For each unit which is still alive, to what position has it moved?

These values are enough to reconstitute the most important parts of the game in the future. If we are able to predict these values consistently, then that means we have a correct model of the short-term dynamics of the game.

Forward Model Network The model architecture I use for forward modeling is shown in Figure 8. It is based on the same principle than the Q-learning network: we feed the data in a convolutional network as explained in Section 3.3, and then we extract the pixel

columns at the position of all the units. For each of these pixel columns we predict the future state of the unit present at that position.

Baselines We compare the performance of our model to two simple baselines. The first baseline is considering that the future state is equal to the current state ($s' = s$). The second baseline interpolates future state given the current state and actions, taking only into account a limited subset of actions, including movements and basic attacks. The first baseline is referred to as the *constant baseline*, and the second as the *interpolate baseline*.

Loss Function The output of the network are of two kinds: a binary output for the *is alive* flag, and four real numbers for the HP, cooldown and movement values. The loss function for the binary output is binary cross-entropy, and for the real values we use mean squares error.

The binary cross-entropy is not a measure we can easily interpret, and it is not defined on the baselines, which always predict 0 or 1 probability of still being alive. Therefore for interpretation of the quality of the learned model we use the F1 score with a cross-validated threshold for predicting the *alive* class. For the real values, the MSE is defined on both the model output and the baselines so we can use that as a comparison.

Our model is in some sense generative, as it is required to generate a consistent state for the future of the game. Prof. Yann LeCun has proposed that I use adversarial training [6] for my model, so that in the case of various possible futures the model would better learn to choose one consistent future state and stick with it. However in our case, it seems that the future is very short-term (8 frames) and therefore quite deterministic, so the advantages of adversarial training are not obvious. Training the mean square error directly causes the model to predict the mean of all possible futures, which seems sensible. Even though I have chosen not to focus on implementing adversarial training for the forward modeling setting, I believe it could lead to interesting insights and performance improvements.

Dataset All the forward models are trained on a dataset of over 7500 pro games, which was build by Gabriel during his PhD thesis [21]. I have also built a synthetic dataset of games played by the baseline algorithms against the StarCraft built-in AI on `m5v5`. I have not trained the forward model on this dataset, however I have used it for evaluation purposes. It is divided by heuristic, which enables evaluating the performances of the model on different kinds of scenarios.

5.2 Motivations for Forward Models

Weights Initialization for Value Network As we have seen, the convolutional model does not succeed to learn complex strategies alone in a reinforcement learning setting. Since the forward model and the RL model share a large common part (the first MLP and the convolutional layers), a natural idea would be to use the forward modeling task as a pre-training step for these common layers. We can then keep these first layers and add a Q-value predictor on top of that, which would be able to learn the Q function based on existing feature extractors, potentially making learning easier and faster.

Value of Predicted State Another approach to using the forward model to play StarCraft would be to replace the $Q(s, a)$ function by first a prediction of the state s' that results of (s, a) by a forward model, followed by an estimation of the value $V(s')$ of that new state, that could serve as a value for $Q(s, a)$. In this setting, the forward model has

to be already very good as there is no way to train it end-to-end with the value function model, due to the binary thresholding of the *is alive* output feature of the forward model.

Tree Exploration Using the Forward Model A simple idea, which I have not implemented, would be to use the forward model as a simulator for our actions, and then to conduct a tree search to select actions that would yield to the potential best improvement over a certain time horizon. This is not quite equivalent to Monte-Carlo Tree Search [4], as we explore only one approximation of the future, which is non-deterministic, whereas MCTS is a framework for solving games which can be perfectly described by a deterministic tree. Also, the branching factor of StarCraft is much bigger than that of Go and Chess, with many actions leading to the same result, making it much harder to apply methods such as MCTS.

Learning Structure of Human Play There are other methods which could be applied to introduce more structure in the decision making process, which haven't been tried out yet but would likely yield substantial improvement. These two ideas both exploit patterns and regularities that appear in pro player games and could be learned by a neural network. The first idea is to predict which units will not take a new action at the current frame, thus limiting the number of Q function evaluations required. A further improvement would be to predict when groups of units take the same action together (for instance when units focus their firing power on a single enemy), which would enable the Q function to be evaluated only once for all the units of this group (or, more precisely, as many times as there are actions which can be taken by all these units). We can again use Gabriel's dataset [21] for this task.

5.3 Results

Forward Model Structure and Training Parameters I have trained several architectures of forward models. In this section I will only describe the results for the most successful one, which has the particularity of containing bilinear layers in the first MLP. Here is the full details of this architecture:

- The inputs to the network are composed of some real values and some categorical values such as order types and unit types. These categorical values are embedded with lookup tables into a 10-dimensional vector space each. The real valued inputs are then concatenated to the embeddings.
- The input MLP is composed of three layers of 50 linear neurons and 50 bilinear neurons each. More precisely, a layer of the input MLP is composed of 50 linear units that see the input to that layer, and 50 bilinear units which see the outputs of the 50 linear units. A bilinear unit is defined by a parameter matrix A and does the calculation $x^T Ax$. Each layer of the input MLP passes to the next layer the concatenation of the values of the 50 linear units and of the 50 bilinear units.
- The model has 3 convolution layers, each composed of 50 5×5 filters, with 3×3 convolution striding at each layer.
- The output MLP has two layers of 50 linear neurons each, followed by an output layer that has 5 units (one for dead/alive prediction and 4 for the hit point, cooldown and movement prediction).

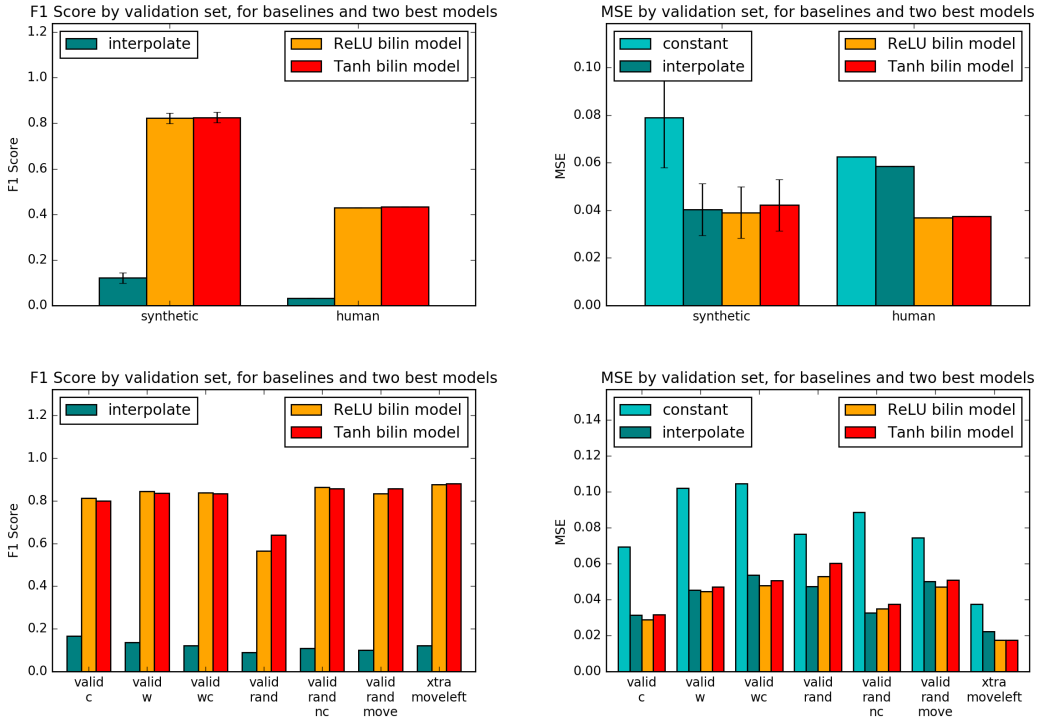

Figure 9: Evaluation of the error of the forward model (two models, same architecture but with different activation functions) against the two baselines. Top: evaluation on the synthetic dataset and the human dataset. Bottom: evaluation on the different scenarios composing the synthetic dataset. Left: F1 score for alive unit prediction (higher is better). Right: MSE error for predicting HP, cooldown and movement of units which are still alive (lower is better). The error bars on the top charts correspond to the variance between the different synthetic scenarios, i.e. between the values of the bottom charts.

The model is trained with adagrad and a learning rate of 0.01. I report results with both the ReLU and Tanh activation functions, which were found to perform similarly in our case.

Results on Forward Modeling Alone Figure 9 and Figure 10 show the error measures for this model for both the binary classification task of still alive prediction and the regression task for the real-valued features prediction. Figure 11 shows the F1 score as a function of the decision threshold for the is alive prediction task, with the corresponding precision/recall curves. We observe that the forward model is much better in dead/alive prediction, but does not beat the baselines for the real-valued features on the synthetic dataset. It does however beat the baseline on all domains on the human dataset. This can be explained by observing that the baseline was hand-crafted to handle precisely the cases that appear in the synthetic dataset, but does not take into account all the complex events (in particular the uses of various technologies) that can appear in the human play dataset.

Experiments on Transfer Learning Figure 12 shows the training curve of a ConvNet model on the `m5v5` scenario, initializing either from random parameters or from the parameters of a forward model. We observe that no progress is made on the `m5v5` scenario and we do not obtain 100% win rate. I have also tried to train a ConvNet model on bigger

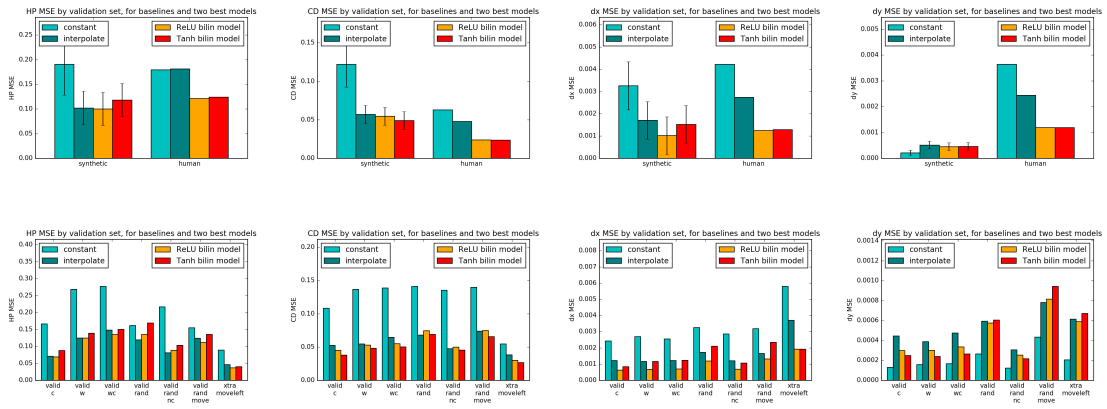

Figure 10: Breakdown of the mean square errors for different features. From left to right: MSE on HP, MSE on cooldown, MSE on x-axis movement, MSE on y-axis movement. Top: synthetic and human. Bottom: different synthetic scenarios.

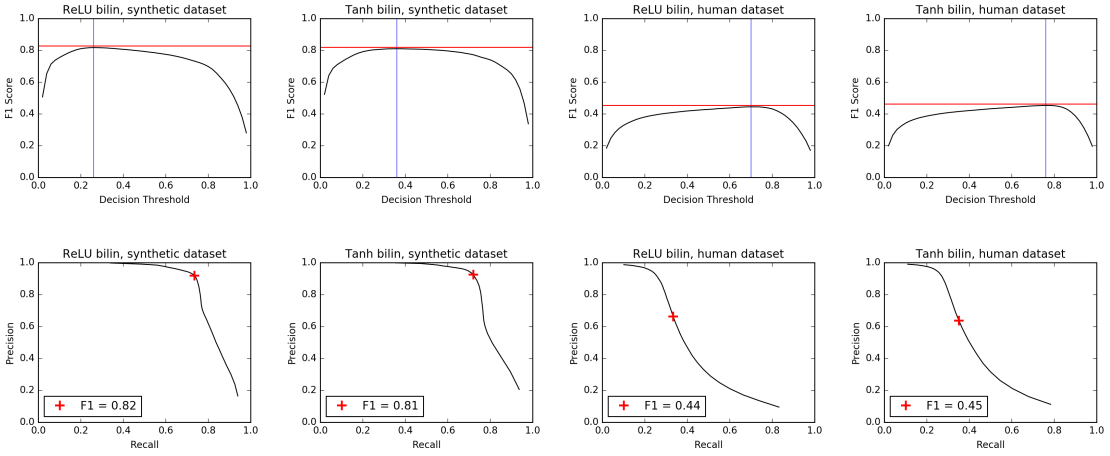

Figure 11: Top: F1 score curves as a function of the decision threshold. Bottom: precision/recall curves. Two left: synthetic dataset, ReLU model then Tanh model. Two right: human dataset, ReLU model then Tanh model.

maps such as `m15v16` but have obtained no positive results.

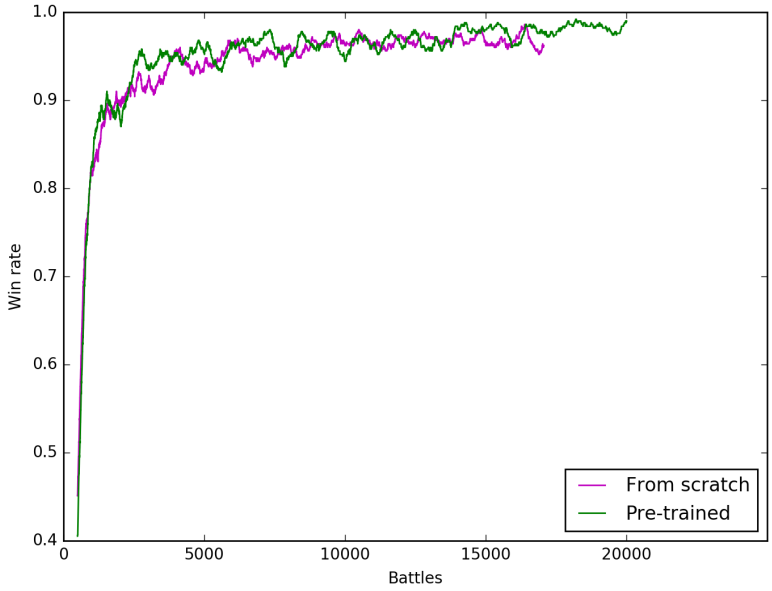

Figure 12: Training curve of our convolutional model on the `m5v5` map, with and without pre-training with the forward modeling task.

Experiments with Evaluating the Value of Predicted States I have also experimented with replacing the $Q(s, a)$ by a $V(s')$ network, where s' is the new state as predicted by the forward model. The $V(s')$ function is implemented by a convolutional network with a similar structure. I have had no positive results with this experiment, probably because the forward model is not able to model the consequences of a single action very precisely.

5.4 Analysis and Possible Improvements

There are several potential problems in the current model structure that make it hard to use it for reinforcement learning.

Model Structure The model is probably limited by the fact that it cannot identify which unit we are currently looking at before the last MLP, which means that it cannot very well identify what information is relevant and what information is not. This was not the case in the relative positions model: in that model, all the game information was encoded relatively to the current unit, so the relevant perspective was known to the network from the start. We need to make changes to the model structure so that the network has easier access to the relevant information; it is not yet clear to me how this should be done.

Computation Time Another limitation we have is that of training time and computing capability: running a single battle takes a lot of time (up to a minute on the biggest scenarios), which greatly limits the number of training samples we can obtain. There are

many optimizations to the model that could be done, and scaling to multi-GPU systems would definitely help.

Task Complexity The biggest limitation is due to the fact that the network is expected to learn a complex policy (such as focus-firing without overkill) by pure exploration with no prior knowledge of what a good strategy would look like. Using examples of human play as a basis for action selection, as well as for discovery of useful structure in the game would certainly be useful, and might enable us to build models that are able to learn more complex strategies.

Proposal Architecture for a StarCraft DNN AI Here is a proposal architecture that would combine all the ideas above:

- Network 1: Given a state, predict which units will have taken a new action in the next n frames. This network could be trained with adversarial training on the human replay dataset.
- Network 2: Given a state, the set of units which will take an action in the next n frames (i.e. an output of network 1), and one of such units, predict which units will take the same action as that unit. This network could also be trained with adversarial training on the human replay dataset.
- Network 3: Given a state and a set of selected units (i.e. an output of network 2), gives the Q value of each of the actions which could be taken by all the units of the set. The Q function could first be trained with Q learning using the human replays as experience. Then, once it has a good Q function approximation, we could fine-tune it by reinforcement learning.

The advantage of such a network is that it would directly be trained on whole game data, meaning that it could potentially learn to do macro-management as well as micro-management. Of course, this is a very high level view, and there are many details to figure out and a lot of work to be done before getting such a program to work.

6 Conclusion

During this internship, I have shown how it is possible to build a model that predicts future StarCraft states using current state and action. This forward model performs significantly better than a hand-crafted rule-based baseline, showing that it is useful to have a model which is able to learn. However in the current state the forward model's predictions are not accurate enough to enable efficient play uniquely based on the predicted outcome of an action.

I have also shown that it is possible to train a convolutional neural network to play StarCraft using pure reinforcement learning. However it has for the moment been limited to small scenarios and more work is required to make it scale.

This internship only lasted 4 months, which is a very short period of time to tackle such a big problem. Still, I have been able to contribute quite a lot to the codebase at FAIR, and I hope it will be useful for the team's future work.

Acknowledgements

I would like to thank Gabriel Synnaeve and Nicolas Usunier for taking me in for this internship. They have been very supportive of my work, and have spent a lot of time accompanying my work and answering my questions. They have always provided valuable feedback and allowed me to progress quickly on this project. I would like to thank all the members of the FAIR team who have helped me out with technical issues or with more theoretical questions, in particular Timothée Lacroix, Soumith Chintala, Alexis Conneau, Étienne Simon. I would like to thank Prof. Yann LeCun for playing ping-pong with me and for providing insightful remarks on my work. I would like to thank Vinh Francis Guyait for helping me set up my Linux box. More than all, I would like to thank Pini Tevet for the everyday extraordinary culinary experiences. Finally, I would like to thank all of the Facebook Paris people for the joyful spirit and the epic foosball matches.

I am grateful to Facebook for providing me the opportunity to work in such a great environment. Without the FAIR infrastructure I would not have been able to run as many experiments and to obtain such results.

References

[1] Alex Auvolat. Deep learning for natural language processing with large vocabularies. 2015.

[2] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Janvin. A neural probabilistic language model. *The Journal of Machine Learning Research*, 3:1137–1155, 2003.

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

[4] Cameron B Browne, Edward Powley, Daniel Whitehouse, Simon M Lucas, Peter I Cowling, Philipp Rohlfshagen, Stephen Tavener, Diego Perez, Spyridon Samothrakis, and Simon Colton. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in Games*, 4(1):1–43, 2012.

[5] Xavier Glorot, Antoine Bordes, and Yoshua Bengio. Deep sparse rectifier neural networks. In *International Conference on Artificial Intelligence and Statistics*, pages 315–323, 2011.

[6] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in Neural Information Processing Systems*, pages 2672–2680, 2014.

[7] Alan Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 6645–6649. IEEE, 2013.

[8] Alex Graves, Marcus Liwicki, Horst Bunke, Jürgen Schmidhuber, and Santiago Fernández. Unconstrained on-line handwriting recognition with recurrent neural networks. In *Advances in Neural Information Processing Systems*, pages 577–584, 2008.

[9] Geoffrey E Hinton, Nitish Srivastava, Alex Krizhevsky, Ilya Sutskever, and Ruslan R Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

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

[11] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*, 2015.

[12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105, 2012.

[13] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[14] 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.

[15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[16] Santiago Ontanón, Gabriel Synnaeve, Alberto Uriarte, Florian Richoux, David Churchill, and Mike Preuss. A survey of real-time strategy game ai research and competition in starcraft. *IEEE Transactions on Computational Intelligence and AI in games*, 5(4):293–311, 2013.

[17] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks. *arXiv preprint arXiv:1211.5063*, 2012.

[18] Jan Peters and J Andrew Bagnell. Policy gradient methods. In *Encyclopedia of Machine Learning*, pages 774–776. Springer, 2011.

[19] Frank Rosenblatt. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review*, 65(6):386, 1958.

[20] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Cognitive modeling*, 5:3, 1988.

[21] Gabriel Synnaeve and Pierre Bessiere. A dataset for starcraft ai\ & an example of armies clustering. *arXiv preprint arXiv:1211.4552*, 2012.

[22] N. Usunier, G. Synnaeve, Z. Lin, and S. Chintala. Episodic Exploration for Deep Deterministic Policies: An Application to StarCraft Micromanagement Tasks. *ArXiv e-prints*, September 2016.

[23] Sudheendra Vijayanarasimhan, Jonathon Shlens, Rajat Monga, and Jay Yagnik. Deep networks with large output spaces. arxiv:1412.7479, 2014.

[24] Christopher JCH Watkins. *Learning from delayed rewards*. PhD thesis, University of Cambridge England, 1989.

[25] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[26] Kelvin Xu, Jimmy Ba, Ryan Kiros, Aaron Courville, Ruslan Salakhutdinov, Richard Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. *arXiv preprint arXiv:1502.03044*, 2015.