ROYAL MILITARY ACADEMY

172th Promotion POL

Maxime Séverin

Academic year 2021 – 2022

2nd Master

A Variable-length Observation Encoder for Multi-Agent Reinforcement Learning

Second Lieutenant Officer Cadet

REMACLE Maximilien

Royal Military Academy

Master Thesis of the department CISS
presented to obtain the academic degree
of Master in Engineering Science
under the supervision of Senior Captain Koen BOECKX, ir.
Brussels, 2022

ROYAL MILITARY ACADEMY

Department CISS

Version 1.0

29/4/2022

A Variable-length Observation Encoder for Multi-Agent Reinforcement Learning

REMACLE Maximilien

RMA Brussels
Royal Military Academy

Preface

Contents

List of Figures

List of Tables

List of Abbreviations

α	Learning rate
γ	Discount Factor
π	Policy
π^*	Optimal Policy
G_t	Cumulative Discounted Reward
$J(\boldsymbol{\theta})$	cost function
$Q_*(s,a)$	Q-value of state-action pair (s,a) under optimal policy
$Q_\pi(s,a)$	Q-value of state-action pair (s,a) under policy π
$R(s,a,s')$	Reward function of an MDP
$T(s,a,s')$	Transition function of an MDP
$V_*(s)$	Value of state s under optimal policy
$v_\pi(s)$	Value of state s under policy π
AI	Artificial Intelligence
DL	Deep Learning
DQN	Deep Q-Network
DRL	Deep Reinforcement Learning
IQL	Independent Q-Learning
LSTM	Long Short-Term Memory
MARL	Multi-Agent Reinforcement Learning
MDP	Markov Decision Process
ML	Machine Learning
NN	Neural Network
RL	Reinforcement Learning
RNN	Recurrent Neural Network
VDN	Value-Decomposition Networks

Acknowledgments

1. Introduction

I recall seeing a package to make quotes

Snowball

1.1. Background

1.2. Literature Review

Part I.

Models and algorithms

2. Reinforcement learning

This chapter shortly explains the working and principles of reinforcement learning (RL) and Deep reinforcement learning (DRL).

RL is an area of machine learning (ML) which is a part of artificial intelligence (AI) . In DRL (section 3.4), RL is mixed with Deep Learning (DL). This allows to solve much more complex problems.

2.1. Markov Decision Processes

In RL, an agent performs in and interacts with an environment. This environment is called an Markov decision process (MDP)[1] . At every time step, the agent gets an observation of the environment which is a part of the environment's current state or all of it. The agent then chooses an action among a set of actions authorized by the environment and submits it to the environment. This action is chosen according to what is called the agent's policy. Finally, the environment reacts to this action by updating its state and providing the agent with a reward, depending on the previous state and the action performed by the agent. This process then repeats itself until the environment ends up in an end state. An illustration of the agent-environment interactions in an MDP is show on figure 2.1.

An MDP is defined by:

- A set of states
- A set of actions
- A transition function
- A reward function
- A start state
- Often one or several terminal state(s)

2.1.1. Transition function

The transition function $T(s, a, s')$ is the probability that taking action a in state s will lead to state s', i.e.:

[1] In this thesis, the terms "MDP" and "environment" are used interchangeably.

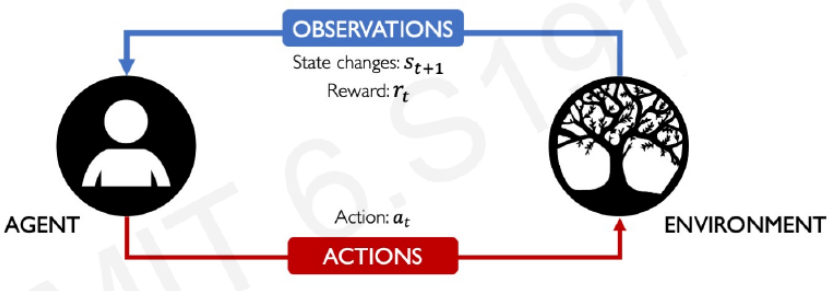

Figure 2.1. Agent-environment interaction in an MDP [AA22]

$$T(s, a, s') = P(S_{t+1} = s' | S_t = s, A_t = a) \tag{2.1}$$

Note: The lowercase notation refers to generic values while the uppercase notation refers to specific instances, made at certain time steps (indicated in subscript). For instance, S_t is the state observed at time step t.

2.1.2. Reward function

The reward function $R(s, a, s')$ is the reward you get by that taking action a in state s and end up in state s'.

In some MDPs, the reward only depends on the state he agent ends up in and is thus noted $R(s)$.

2.2. Concepts and definitions

2.2.1. Discounted factor and discounted cumulative reward

Very often, a discount factor γ ($0 \leq \gamma \leq 1$) is introduced. We then talk about the cumulative discounted reward G_t, which is expressed as follows:

$$G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2.2}$$

$$= R_{t+1} + \gamma G_{t+1} \tag{2.3}$$

The agent must then choose the actions that optimize G_t. introducing this γ models the uncertainty of the future rewards and helps our algorithm to converge.

2.2.2. Policy

The policy $\pi(s)$ is a function that returns an action a to take in state s. In a more general way, it can also give a probability distribution over all available actions in state s. If that is the case, it will be denoted $\pi(a|s)$. The optimal policy, i.e. the one that optimized (2.2) is $\pi^*(s)$.

2.2.3. Value and Q-value

The value of a state $V_\pi(s)$ is the expected (discounted) cumulative reward when starting from state s and choosing action with policy π.

$$V_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] \tag{2.4}$$

$V_{\pi^*}(s)$ or $V_*(s)$ is the value of state s under the optimal policy.

The Q-value of a state-action pair $Q_\pi(s, a)$ is the expected (discounted) cumulative reward when starting from state a and undertaking action a, then acting according to policy π.

$$Q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] \tag{2.5}$$

As for the state value, $Q_{\pi^*}(s, a)$ or $Q_*(s, a)$ is the Q-value of state-action pair (s,a) under the optimal policy. Sometimes, a state-action pair is also called a q-state.

The link between value and q-value is then:

$$V_\pi(s) = Q_\pi(s, \pi(s)) \tag{2.6}$$

Or, with the optimal policy:

$$V_*(s) = \max_a Q_*(s, a) \tag{2.7}$$

2.2.4. Episode and transition

An episode in an MDP is one specific sequence of states actions and reward that starts with the start state of the environment end stops with a terminal state.

In a MDP, moving from one state to another is called a transition. More specifically when we talk about a transition in RL, it can be (and will be for the rest of this thesis) a sample of one time step of a particular episode. The transition at time t contains the following data:

- S_t the state (or local observation) at time step t.
- A_t the action performed by the agent at time step t.
- R_t the reward received by the agent at time step t.
- S_{t+1} the state (or local observation) at time step t+1.
- *is_done* a Boolean value that is true only if S_t is terminal. S_{t+1} will then be None or 0.

2.2.5. Bellman equation

It can be shown that the Q-value function of the optimal policy $Q_*(s, a)$ obeys a recursive relationship that is called the Bellman equation:

$$Q_*(s, a) = \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma \max_{a'} Q(s', a')] \tag{2.8}$$

$$= \sum_{s'} T(s, a, s')[R(s, a, s') + \gamma V(s')] \tag{2.9}$$

The optimal policy can than easily be retrieved from $Q_*(s, a)$:

$$\pi_*(s) = \arg\max_a Q_*(s, a) \tag{2.10}$$

2.3. Tabular Q-learning

So, in RL, we want to find that optimal policy, that is, the one that maximizes (2.2). One easy algorithm to do so is called Q-learning. As the name suggests, the algorithm will try to learn the Q-values for every state and action for the MDP in which it is learning. The optimal policy will then be retrieved with (2.10).

2.3.1. The Algorithm

First, a table is created. This table contains the estimations of the Q-values for every possible state and action in the environment. It can be initiated with random values or zeros. This table is called the Q-table. An example of such a table for an environment with N states and M actions is show on figure

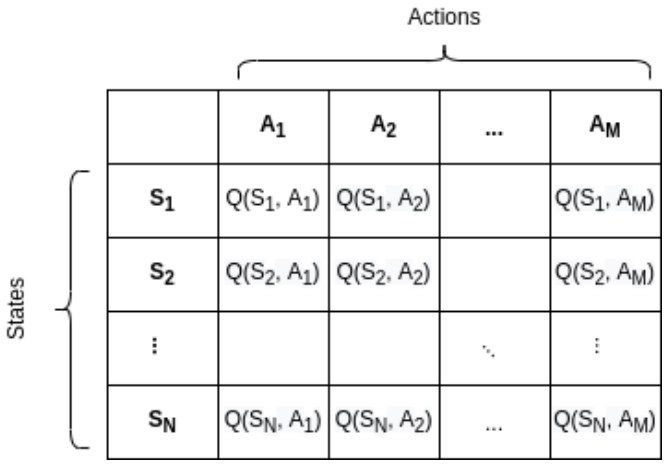

Figure 2.2. Example of a Q-table in Q-learning [Q-l21]

2.2.

Then, the agent will perform in the environment, collecting data in the form of transitions. For each transition, the agent "learns" by updating its Q-table using (2.11), inspired by (2.8).

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{2.11}$$

where:

- α $(0 < \alpha < 1)$ is the learning rate.
- $[R_t + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$ is the temporal difference. It is actually the difference between what $Q(S_t, A_t)$ should be (i.e. $R_t + \gamma \max_a Q(S_{t+1}, a)$) and what it really is.

At every learning step, that is, at every transition, $Q(S_t, A_t)$ is incremented by the temporal difference multiplied by the learning rate. If $\alpha = 1$, the agent ignores prior knowledge and $Q(S_t, A_t)$ is replaced by the temporal difference at every learning step. If $\alpha = 0$, the agent is not learning at all. The learning rate actually determines how fast the Q-values in the Q-table will change. A higher learning rate might lead to faster convergence of our Q-values towards the optimal values, but a too high learning rate might also lead to no convergence at all.

If S_t is a terminal state, 2.11 becomes:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t - Q(S_t, A_t)] \tag{2.12}$$

2.3.2. The epsilon-greedy policy

As explained in previous section, the agent learns by performing in its environment. While it does this, it updates its q-table with the data obtain from the environment at every transition. Thus, the different states visited by the agent have a really important impact on the quality of the learning. Those visited states depend on the actions the agent decides to perform, which are determined by its policy. Hence, besides the learning process, the agent's policy is also really important in the learning process.

It might seem a good option to use (2.10), but using the agent's Q-table instead of the optimal Q-values function $Q_*(s, a)$, which is unknown. However, following that sub-optimal policy might lead to an agent that explores only certain states. If the environment gives some high reward when going to certain states, but the agent never goes to those states, the Q-values for those states will never be updated

to higher values and the agent will never learn the optimal policy.

Thus, the agent shall explore every state during the learning. A solution to that is the epsilon-greedy policy. At every transition, the probability that the agent will choose an action using (2.10) with its own Q-table is $1-\epsilon$. And there is an ϵ probability that the agent will take a random action from all available actions in the environment. It is obvious that ϵ is a really important parameter in the epsilon-greedy policy and that $0 \leq \epsilon \leq 1$.

2.3.3. The exploration/exploitation dilemma

The exploration/exploitation dilemma is a problem that is often encountered in RL. As already stated in section 2.3.2, we have to add some randomness in the chosen actions in order to allow the agent to explore all of the possible states. That is what is called exploration.

However, in some more complex environment, there are states in which the agent is very unlikely to find itself by performing only random actions. Let's take the example of a video game where you have several levels: if you play randomly, you could get some little rewards if you start to progress in the correct direction. But, if the game is complex, it is very unlikely that the agent will finish the whole level and access the next ones by playing randomly. Hence, the agent will never observe the more advanced states, further in the game. So, at a certain moment, the agent has to exploit what it has already learnt in order to access some other states and explore the results of action undertaken in those states. This is called the exploitation.

The dilemma is thus to find a good equilibrium between exploration and exploitation, in order to be sure to explore every possibility, but still advanced to the states that are more difficult to access. One solution to this problem is to implement an epsilon decay. The idea is to decrease the value of the epsilon parameter of the agent's epsilon-greedy policy as the episodes progress. This way, the agent will act more randomly in the start of the training, when it hasn't learnt much, but will act more and more according to its q-table, as the values of that table converge to the optimal solution.

3. Deep Learning and neural networks

This chapter presents the basic concepts of deep learning and neural networks as well as how they can be used in reinforcement learning .

3.1. Definition

A neural network or NN is a function that takes a tensor (a multi-dimensional vector) as input and then computes an output, which is another tensor of values. This computation involves a lot of parameters that are used to compute the output from the input. It can be noted as follows:

$$\mathbf{Y} = NN(\mathbf{X}; \boldsymbol{\theta}) \tag{3.1}$$

where $\mathbf{X}$ and $\mathbf{Y}$ are respectively the input and output tensors, NN the neural network function and $\boldsymbol{\theta}$ the neural network parameters.

The parameters of a neural network can be tuned or "learned" in such a way that the neural network approximates any given function. The parameters are learned using some data composed of some inputs and the outputs that we want our neural network to produce when fed with those inputs.

For instance, if we want our neural network to be able to make the distinction between an apple and a banana, we will give him some data with photographs of apples and bananas (the inputs) as well as the corresponding labels for each photograph (the outputs). The network will then learn from this data by tuning its parameters. Then, the neural network should be able to classify some new images of apples and bananas that he has never been presented with during its training. In this example, the input tensors of the network are the values of all the pixels in the image, and the output will only have two values: the probability that the network thinks the input is a picture of an apple and the probability that it thinks it is a banana.

Other examples are:

- Image up-scaling (resolution increase)
- Machine translation
- Image generation
- Approximation of the Q-values in function of an agent's observation (see section 3.4)

3.2. Working Principle

3.2.1. Single Neuron

The basic building block of an artificial neural network is the artificial neuron. The name comes from the real neurons in our brains, because the technology has been inspired by it. Figure 3.1 shows a schematic view of a biological and an artificial neuron. An artificial neuron is a mathematical function that takes several values as input an computes one output. The output is a weighted sum of the inputs to which is also added a fixed value, called the bias. For a n inputs neuron, the output y is thus calculated as follows:

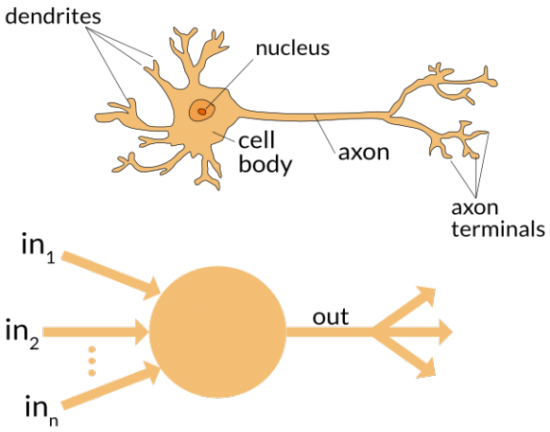

Figure 3.1. Schematic view of a biological and an artificial neuron [neu20]

$$y = b + \sum_{i=1}^{n} w_i x_i \tag{3.2}$$

where x_i is the i-th input, w_i its corresponding weight and b the neuron's bias. This can also be written in matrix form:

$$y = b + WX \tag{3.3}$$

where W is a horizontal vector containing all the weights and X is a vertical vector containing all the inputs.

3.2.2. Neuron Layer

Furthermore, to obtain several outputs, several neurons can be stacked an form what is called a layer. Figure 3.2[1] shows a two neurons layer with five inputs. The output of such a neurons layer may be calculated as follows:

$$Y = \mathbf{W}X + B \tag{3.4}$$

This is almost the same as (3.3), but now we have Y a vertical vector containing all outputs, B a vertical vector containing the biases of every neuron and $\mathbf{W}$ a two-dimensional matrix containing every weight of every neuron, that is, all the W horizontal vectors from (3.3) of all the neurons, stacked on top of each other.

3.2.3. Adding Non-linearity

Earlier in this chapter, we said that the aim of a neural network is to approximate a given function. However, (3.4) clearly shows that the transformation between output and output is linear. Thus, only linear functions may be approximated, which is very limiting. To overcome this problem, some non-linearity is added into the neural networks. This is done by applying a non-linear function to the output of each neuron. In the field of deep learning, those functions are called activation functions. Figure 3.3 shows four activation functions that are widely used in DL: sigmoid, Rectified Linear Unit (ReLU), hyperbolic tangent (tanh) and Exponential Linear Unit (ELU). Equations (3.5) to (3.8) show the formula of those activation functions.

[1]Figure generated with NN-SVG [ale22]

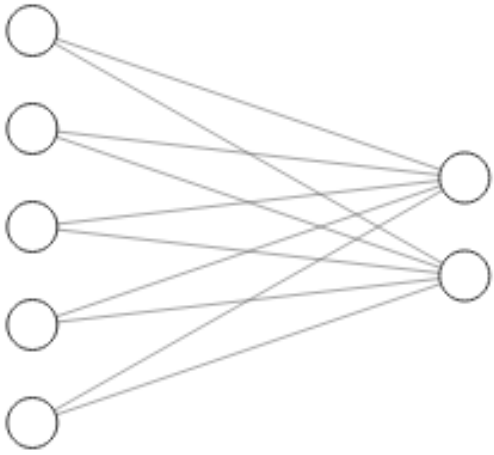

Figure 3.2. Schematic view of a two neurons layer with five inputs

$$sigmoid(x) = \frac{1}{1 - e^{-x}} \tag{3.5}$$

$$ReLU(x) = max(0, x) = \begin{cases} 0 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.6}$$

$$tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}} \tag{3.7}$$

$$ELU(x) = \begin{cases} e^{x} - 1 & \text{if } x < 0 \\ x & \text{otherwise} \end{cases} \tag{3.8}$$

3.2.4. Deep Neural Networks

In order to approximate more complex functions, deep neural networks are often used. They consist of several neurons layers stacked horizontally, where the output of each layer is fed as input to the next layer. Of course, a non-linear function is applied at the output of every layer. such a neural network is said to be fully connected. Figure 3.4[2] shows a neural network with five inputs and two outputs. The data goes through three layers of 10 neurons before (called hidden layers). The output of the last hidden layer is fed as input to the output layer. The activation function is not shown on the figure, because it is considered to be part of each neuron, i.e. each neuron performs (3.3) then applies an activation function.

3.2.5. Supervised Learning

In a neural network, the parameters ($\boldsymbol{\theta}$ in equation 3.1) are the weights and biases of all the neurons in the network. Learning the weights and biases using a database composed of both inputs and their corresponding outputs is called supervised learning. This is opposed to another field in machine learning called unsupervised learning, where the training data is only composed of inputs. The objective is then to detect patterns in the data and classify it into categories.

[2]Figure generated with NN-SVG [ale22]

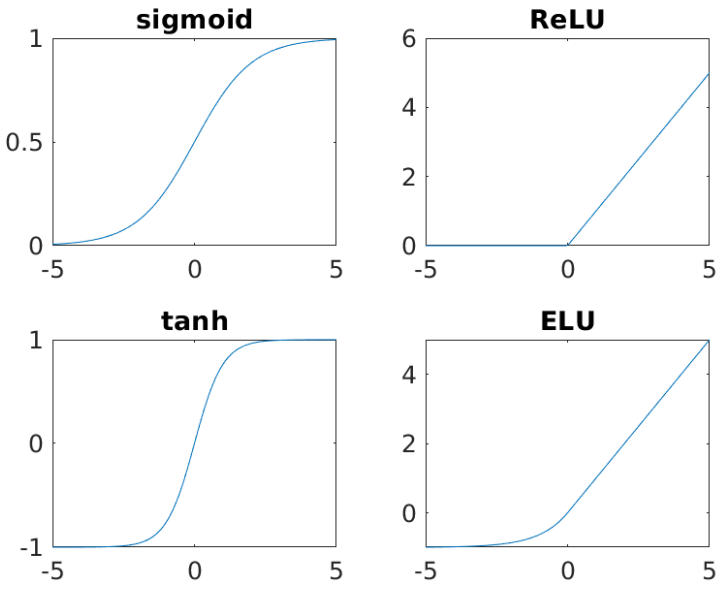

Figure 3.3. Widely used activation functions

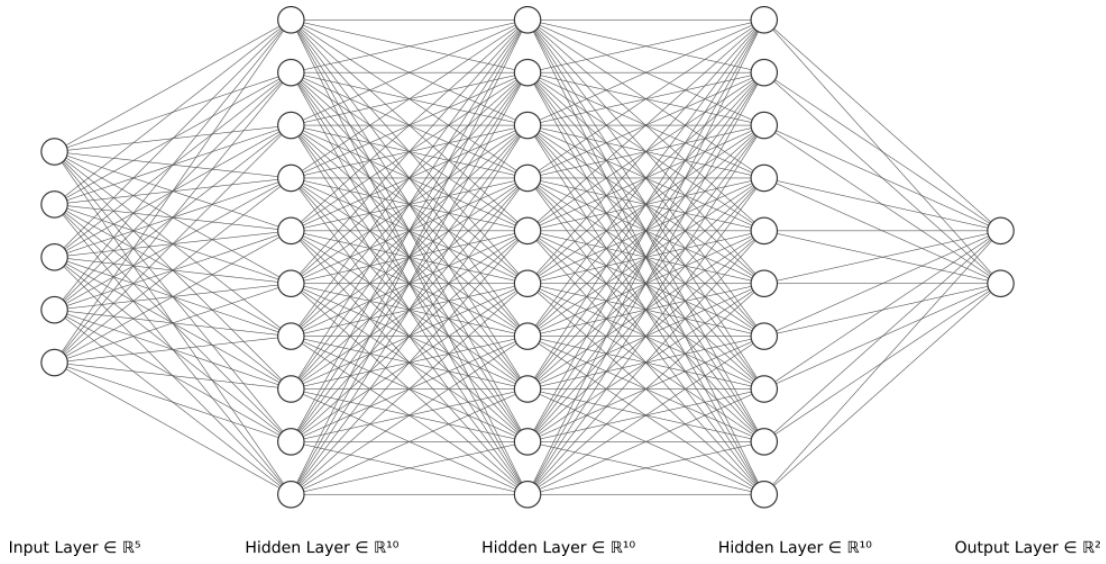

Figure 3.4. Deep neural network

Loss function

In supervised learning, the parameters are optimized by minimizing a cost function $J(\boldsymbol{\theta})$:

$$J(\boldsymbol{\theta}) = \mathcal{L}(\hat{Y}, Y) \tag{3.9}$$

where $\hat{Y} = NN(X;\boldsymbol{\theta})$ and $L(\hat{Y}, Y)$ is call the loss function and quantifies the difference between the expected output from the learning data and the output produced by the neural network. $\hat{Y}$ is often called the prediction and Y the target.

A lot of different loss functions are used in Deep Learning, depending on the application. (3.10) shows the Binary Cross Entropy (BCE) loss, which is used a lot when the outputs of the networks should be either 0 or 1. (3.11) shows the Mean Square Error (MSE) loss.

$$\mathcal{L}_{BCE}(\hat{Y}, Y) = -\frac{1}{N}\sum_{i=1}^{N} Y_i \, ln(\hat{Y}_i) + (1 - Y_i) ln(1 - \hat{Y}_i) \tag{3.10}$$

$$\mathcal{L}_{MSE}(\hat{Y}, Y) = \frac{1}{N}\sum_{i=1}^{N} \left(\hat{Y}_i - Y_i\right)^2 \tag{3.11}$$

This cost is rather easy to compute for given parameters and training data. However, it might be really heavy computation wise for big neural networks and/or data sets. Furthermore, this can be parallelized a lot and the computation time can be reduced very significantly by making this computation on GPUs (Graphics Processing Units), which have a very large number of cores that can work in parallel (up to tens of thousands) whereas classical CPUs (Central Processing Units) have at most a couple of tens of cores.

Gradient descent

The way the cost function is minimized is by applying the gradient descent algorithm, which is an iterative method. $\boldsymbol{\theta}_t$ will thus denote the value of $\boldsymbol{\theta}$ at iteration t.

First, the gradient of J with respect to $\boldsymbol{\theta}$ (i.e. $\nabla_{\boldsymbol{\theta}} J$) is calculated. This is done using the backpropagation algorithm. The explanation of its working is out of the scope of this thesis.

Then, a small step in the negative direction of the gradient is taken, that is:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - \alpha \nabla_{\boldsymbol{\theta}} J \tag{3.12}$$

where α is called the learning rate. The discussion about this learning rate in RL (section 2.3.1) is also valid here.

Those two steps are repeated until some stop criterion is achieved (Number of iterations, threshold value for the loss, accuracy of the predictions, ...).

Furthermore, the gradient is almost never calculated based on one single sample. Instead, the average over several (or all of the) samples of the training data is used. This sample is called a batch. The batch size is a very important hyperparameter in neural network training. A higher batch size often results in faster convergence and better results. However, beyond a certain point, increasing the batch size

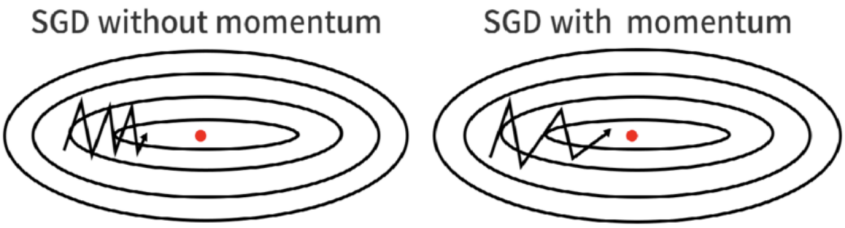

Figure 3.5. Stochastic Gradient Descent with and without momentum [Mus22]

even more won't hardly increase the performance, but will increase the computation time.

Using only on or a few sample from the training data is called stochastic gradient descent (SGD). The computation time for one step in much smaller than for much bigger batches, but the direction of the steps is not optimal. Hence, with SGD, much more smaller steps are taken instead of bigger, but slower steps.

Such a method that calculates the size and direction of the steps to take at every iteration is called an optimizer.

Momentum

Other optimizers than the gradient descent exist. In order to increase the efficiency, a momentum term might be added. The momentum term takes into account the direction of the previous steps to determine the size and direction of the current step. It works as follows:

$$m_{t+1} = \alpha \nabla_{\boldsymbol{\theta}} J + \mu m_t \tag{3.13}$$

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t - m_{t+1} \tag{3.14}$$

where μ is a parameter that determines how much the previous steps influence the current one.

An illustration of momentum applied to a two parameters optimization problem is show on figure 3.5. The image can be visualize as a plane where the position (in (x,y) coordinates) represents the value of the two parameters. The black elliptical curves are iso-loss curves and the red dot is the optimal values of the two parameters for a minimal loss.

Other optimizers

Other more sophisticated optimizers have been developed in the field of Deep Learning. The goal of every optimizer is to achieve faster convergence and/or to convergence towards better results (lower loss values). Adam [KB14] is maybe the most widely used optimizer in Deep Learning. It estimates the first and second momentum and computes several adaptive learning rates.

3.3. Recurrent Neural Networks

3.3.1. Basic RNN

Besides the classical Fully Connected (FC) neural networks, there exist a lot of other types of neurons that offer some advantages, depending on the application. Recurrent Neural Networks (RNN) are used to process sequences of data, i.e. a sequence of inputs that have the same dimensions. In most of the

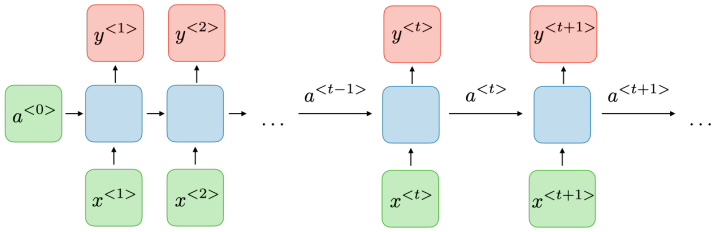

Figure 3.6. Recurrent Neural Network[rnn22]

cases, the sequence is a time series.

In a FC NN, if two inputs are fed one after the other to the network, both are processed independently. Whereas in RNN, some information from the (processing of the) first input is kept in the network and used to process the second input. The information that is kept is actually all the hidden states, that is the values of the outputs of the neurons in the hidden layers of the network. For the first input of the sequence, this hidden state must be initialized to some value. The most common practice is to initialize it at zero or random values.

Figure 3.6 shows an illustration of a RNN. x, a and y are respectively the inputs, hidden states and outputs. The exponent expresses the position in the sequence, i.e. the time step.

3.3.2. Long Short-Term Memory

The regular RNNs allow to insert some kind of memory in the network, where the networks "remembers" the previous inputs it has received. There is, however, a limitation to this: if the sequence is too long, it is difficult for the network to carry information from early time steps to late ones. Thus, RNNs have memory, but it is short-term. Long Short-Term Memory (LSTM) cells [HS97] are a solution to this.

In LSTMs not only is the hidden state passed to the next time step, but also what is called the cell state. This cell state is a kind of memory of the neuron that can hold information during a lot of time steps. The working of a LSTM cell is a bit more complicated than a simple recurrent neuron or classical neuron. Actually, a LSTM cell is composed of four classical neurons, as described in 3.2.1. Those neurons are called gates.

The first gate is the forget gate. Its activation function is a sigmoid, so that its output value is always between 0 and 1. This gate determines whether or not the cell state of the previous time step will be carried on to the next step. An forget value of 1 means the previous cell state is kept, while a forget value of 0 means the previous cell state is forgotten.

The second gate is the cell gate or candidate. This gate has a tanh activation function, which constraints its output value between -1 and 1. This gate computes a candidate value to replace the cell state of the previous time step.

The third gate is called the input gate, it has a sigmoid activation function. It determines if the candidate will actually be added to the current cell state or discarded. It works the same way as the forget gate.

The three previous gates determine the value of the cell state:

$$c_t = f_t c_{t-1} + i_t g_t \tag{3.15}$$

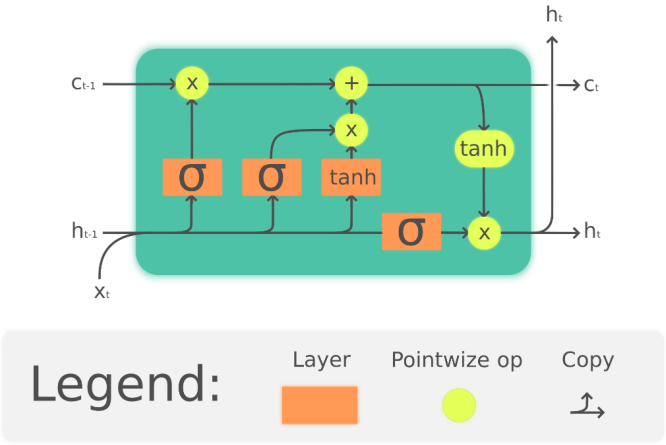

Figure 3.7. Long Short-Term Memory layer [Che18]

where c_t is the cell state at time step t and f_t, i_t and g_t are respectively the output of the forget, input and cell gates at time step t.

The last gate is called the output gate, and also has a sigmoid activation function. It computes what information should be in the hidden state, in the same way as the forget and input gates. The next hidden state is then computed as follows:

$$h_t = o_t tanh(c_t) \tag{3.16}$$

where o_t is the output of the output gate at time step t.

A visualisation of a LSTM cell is show on figure 3.7. Note that:

- The input for all of the gates is the input of the LSTM cell concatenated with the hidden state of the previous time step.
- The figure shows a LSTM layer. So, every gate is a classical neuron layer and h_t and c_t are vectors.

3.3.3. Gated Recurrent Unit

Gated Recurrent Units (GRUs) [CGCB14] are a newer solution this alleviates the short-term aspect of the memory in classical RNN cells. GRUs are pretty similar to LSTMs, but a lighter version, hence faster. There is no cell state in GRUs, so only the hidden state is passed to the next time step. They also only have two gates: a reset and an update gate.

The reset gate determines how much of the information from the previous time step is used to calculate the new candidate for the next hidden state. The update gate acts like the input and forget gates of an LSTM. It determines to what extent the previous hidden state and the new candidate contribute to the next hidden state. An illustration of a GRU cell is shown on figure 3.8.

GRUs are a bit faster than LSTMs, but don't have a real memory (the cell state in LSTMs). None of them has a clear advantage over the other. Depending on the application, one of the two might offer slightly better results for a specific use. The best solution is always to try both and determine which one works the best.

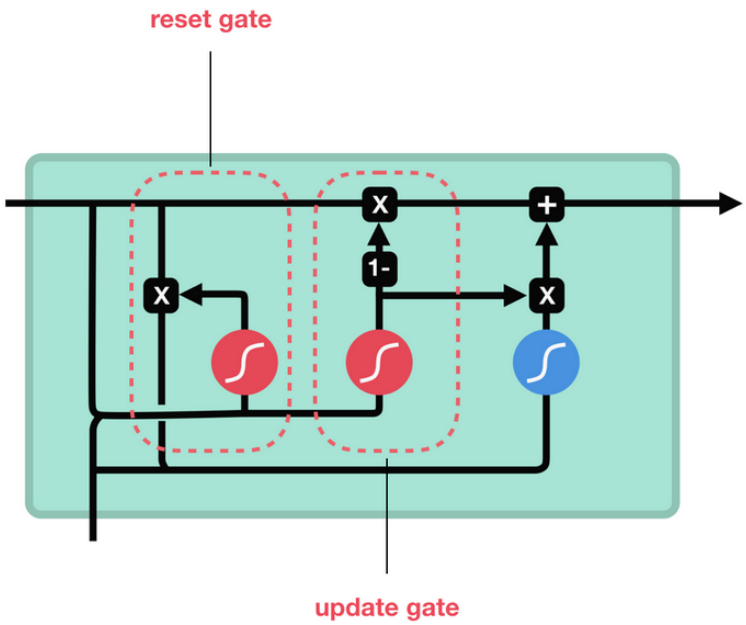

Figure 3.8. Gated Recurrent Unit[Phi20]

3.4. Deep Reinforcement Learning

3.4.1. Limitations of tabular learning

As seen in sec 2.3.1, a Markov Decision Process can be optimised using Q-learning, by creating a table that has as many rows as there are different possible states (i.e., the size of the state space) in the MDP, and as many columns as there are possible actions (i.e. the size of the action space). Even though this algorithms might be efficient with simple MDPs, when the complexity increases, it is limited by some important factors.

First, in some MDPs, the state space (and action space) is so large that the Q-table would be really to big. Some MDPs are games where the agent's observation are the pixel of the screen of the game. For every pixel the red, blue and green colours are coded with one byte each, which means there are over 16.5 millions different colours (255^3). Multiplying that by the amount of pixels in an image would yield to at least several tens of billions of different possible states, which means the same amount of rows in the Q-table.

Second, there are a lot of different possible states that are very similar, like two adjacent positions or a small different of colour in an image. Hence, they should have similar (or identical) Q-values. In tabular learning, the agent must observe all of those different states several times in order to approximate correctly the corresponding Q-values. Furthermore, there could also be a lot of possible state that the agent will actually never meet. For example, when the agent's observations are images.

3.4.2. Deep Q-Learning

A solution to this is to use a neural network to approximate the Q-values. The input of the network is then the observation of the agent and the outputs are the estimated Q-values for every possible action. This solution works really well with a lot of complex MDPs. It has been used to train an agent that can play better that humans at several Atari® games [MKS+13].

The idea in Deep Q-learning, also called Deep Q-Network (DQN), is to let an agent evolve in an MDP and save the transitions in a buffer, called the experience buffer. Each transition contains the observed state (S), the action taken (A), the reward received (R) and the state observed after the action's been undertaken (the "next observed state", S_{next}).

At each learning step, a batch is created by sampling from the experience buffer. Then, the loss is computed using some loss function. The first argument of this loss function is the current approximated Q-value, that is the output of the Q-Network when fed with the observed state, for the undertaken action. The second argument is the target value:

$$R + \gamma \max_a Q(S_{next}, a) \tag{3.17}$$

where R is the reward, Q(S) the outputs of the Q-Network for observation S, i.e. the approximation of all Q-values, and Q(S,a) is the output of the Q-Network for a specific action a.

The Q-Network is then trained using a gradient descent or another optimizer with the values of the computed loss. This might be done at every transition (every step in the MDP) or after every episode.

3.4.3. Target network

One issue of this method is that the Q-Network is evaluated in S and in S_{next}. There is only one step between those two states, which means they are very similar. A neural network will have a hard time to tell the difference between those states. Thus, when the network will be updated for S, the values for nearby states (including S_{next}) will also change. Hence, by improving the estimation for S, the estimation for S_{next} might get worse.

To fix this stability issue, a copy of the Q-Network is made. This copy is called the target network. This network is used to compute the target value (3.17). The trick is not to update the weights of the target at every training step, but either copy the weights of the Q-Network every given period, expressed in number of episodes.

3.4.4. (Hyper) Parameters

4. Multi-Agent Reinforcement Learning

Up to now, there was only one agent performing in the environment. However, the environment that's implemented in this thesis (see chapter 6. *tanksEnv*) is a Multi-Agent environment, where several tanks perform at the same time.

This chapter describes some Multi-Agent Reinforcement Learning (MARL) algorithms that are used to train multiple agents in a Multi-Agent MDP.

4.1. Independent Q-Learning

One of the simplest MARL algorithms is Independent Q-Learning (IQL) [Tan93]. This simple method consists of implementing the basic DQN algorithm for every agent. Then, all agents perform simultaneously in the environment and they all train with their own observation and rewards that they get individually from the environment.

4.2. Value-Decomposition Networks

Although IQL works well for a lot of environments, faster convergence and/or better performance of the agents can be obtained with more advanced methods that combine the information of the different agents during training. This is called centralised learning. However, for the execution, each agent has still its own Q-Network that approximates the Q-values for its own observe states. This is called decentralised execution.

Value-Decomposition Networks (VDN) [SLG+17] is such a centralised learning decentralised execution MARL algorithm. Instead of using individual Q-values, a joint Q-value function is learnt: $Q_{tot}(\mathbf{S}, \mathbf{A})$, where $\mathbf{S}$ is the joint observation of the state and $\mathbf{A}$ is the joint action vector, containing all action of every agent. In VDN, the joint Q_{tot} is simply the sum of all independent Q-values[1]. Q_{tot} may be expressed as follows:

$$Q_{tot}(\mathbf{s}, \mathbf{a}; \boldsymbol{\theta}) = \sum_{a=1}^{n} Q_a(s_a, a_a; \theta_a) \tag{4.1}$$

where n is the number of agents, Q_a, S_a and A_a respectively the individual Q-Network, observation and actions of agent a and θ_a the parameters (i.e. the weights and biases) of Q_a.

$$\mathcal{L}(\theta) = \sum_{i=1}^{b} \left(y_i^{tot} - Q_{tot}(\mathbf{S}_i, \mathbf{A_i}; \boldsymbol{\theta}) \right)^2 \tag{4.2}$$

[1] In this case, term "Q-value" refers to the output of the individual Q-Networks of the agents. However, since only the Q_{tot} is learnt, those values are not necessarily the proper Q-values for the agents.

where b is the batch size, $\boldsymbol{\theta}$ contains all parameters of all Q-Networks and the subscript i refers to the specific values of the i-th transition of the batch and

$$y_i^{tot} = R_i + \gamma \max_{\mathbf{a}} Q_{tot}(\mathbf{S}_{next,i}, \mathbf{a}; \boldsymbol{\theta}^-) \tag{4.3}$$

where $\boldsymbol{\theta}^-$ are the parameters of the target Q-Networks, as explained in section 3.4.3.

The reward R_i is also a joint reward. This reward might be directly given by the environment or be the sum or mean of the rewards that the agent receive individually from the environment.

Even though the Qa functions do not estimate the expected return, performing greedy action selection with the outputs of the Qa-Networks for each agent will directly an individual policy that can be used for decentralised execution.

4.3. QMix

QMix [RSS+18] is another approach that is also a centralised learning decentralised execution method. The difficulty is to derive decentralised agent's policies that are fully consistent with the centralised policy that has been learnt. In VDN, this is guaranteed by the fact that Q_{tot} is the sum of the Qa values of every agents. However, this condition is not necessary to assure the consistency between the centralised and decentralised policies. Instead, we could only ensure that a global argmax applied on Q_{tot} yields the same result as a serie of argmax operations applied on the individual Qa values:

$$\arg\max_{\mathbf{a}} Q_{tot}(\mathbf{s}, \mathbf{a}) = \begin{pmatrix} \arg\max_{a_1} Q_1(s_1, a_1) \\ \vdots \\ \arg\max_{a_n} Q_n(s_n, a_n) \end{pmatrix} \tag{4.4}$$

If (4.4) is satisfied, choosing greedy actions from Q_{tot} or individually from the Qa functions would yield the same result. This is the case for VDN.

However, the idea of QMix is that the relation between Q_{tot} and Q_a can be extended to a larger set of functions, that is all the monotonic functions. This can be guaranteed by imposing the following constraint:

$$\frac{\partial Q_{tot}}{\partial Q_a} \geq 0, \forall a \in [1, 2, ..., n] \tag{4.5}$$

Imposing (4.5) is a sufficient condition to ensure (4.4). To do this, QMix uses Q-Networks to compute the Q_a values, a mixer network that computes Q_{tot} with the outputs of the Q-Networks and a bunch of neural networks that compute the weights of the mixer network. Such networks are call hypernetworks [HDL16]. The trick to ensure (4.5) is to limit the weights to positive values. This is done by taking the absolute value of the outputs of the hypernetworks that compute the weights.

Figure 4.1 shows the overall architecture of QMix, as well as the mixer and agent networks. The figure comes from [RSS+18], where the notation is different. However, Q_a and Q_{tot} refer to the same value as explained in this thesis. It can be observed that the Q_a networks have only access to o_t^a the local observation of agent a at time step t, while the hypernetworks of the mixing network have access to S_t the full state of the environment at time step t. Moreover, the exact architecture of the Q-networks (Q_a) might be adapted to fit the best the specific application. The use of a GRU layer (or any other RNN layer) is not strictly necessary.

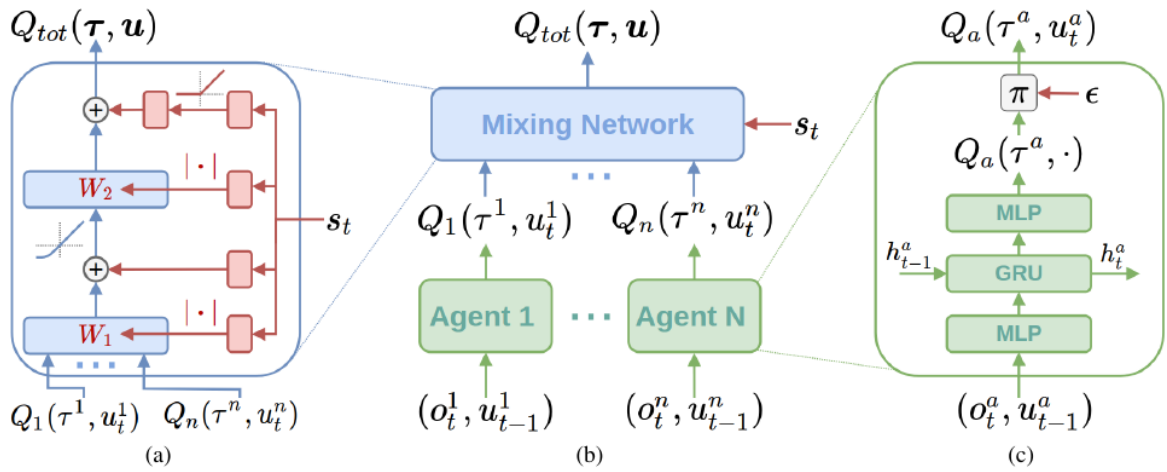

Figure 4.1. (a) Mixing network (red: hypernetworks, blue: mixing network layers). (b) Global QMix architecture. (c) Agent Q-Network. [RSS+18]

By implementing a more complex joint Q-value factorisation function, QMix is able to learn more complex joint Q-value functions, performs better and learns faster (in less episodes) than IQL or VDN, in a lot of different MDPs. QMix has been developed and compared with VDN and IQL in [RSS+18]. Figure 4.2 shows some of the obtained results.

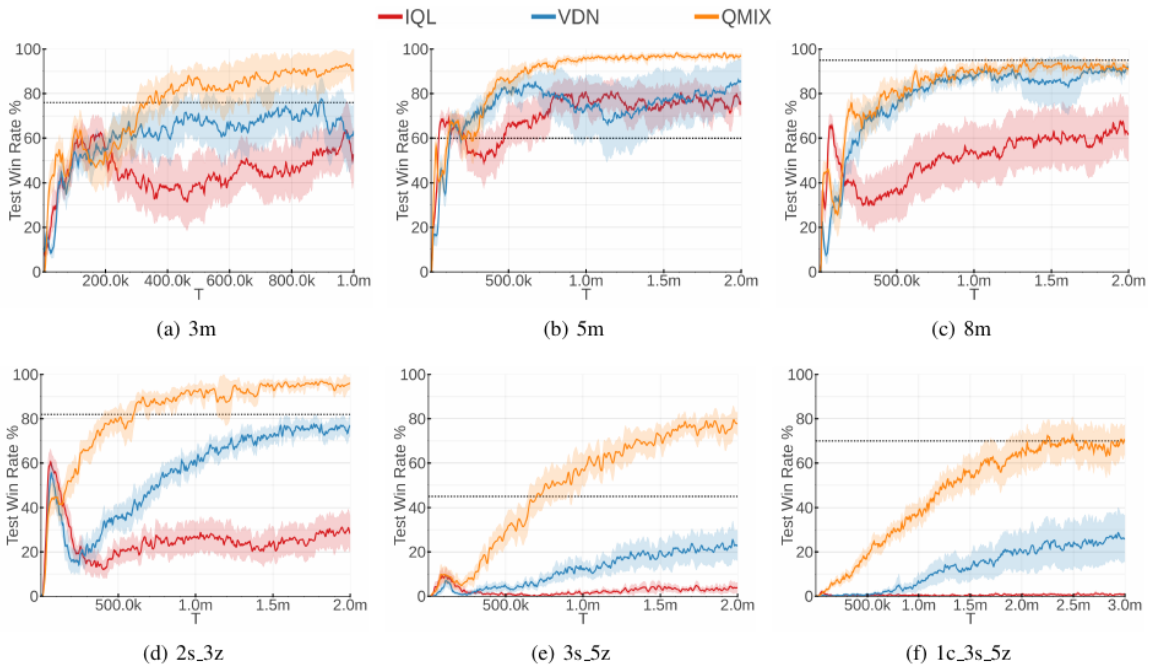

Figure 4.2. Win rates for IQL, VDN and QMix, learning to play the StarCraft II video game on six different maps.

5. POMDP and variable length observation spaces

5.1. Partially observable Markov Decision Processes

5.2. Variable length observation spaces

5.3. RNN-based encoder

Part II.

The environment

6. *tanksEnv*

This chapter presents *tanksEnv*, a tanks battle ground environment. It is based on the environment developed in [BOE20].*tanksEnv* is a grid-like environment of finite size. Two teams are playing against each other: team ”red” and team ”blue”.

6.1. Tanks

Each team is composed of one or several players (let’s call them tanks) that are defined by their states, which contain the same information for each tank, that is:

- The position of the tank in (x,y) coordinates
- An id that is different from all tanks (of both teams)
- Its team
- How much ammo it has left
- Which other tank it’s aiming at
- Whether it is still in the game or has been hit and can no longer move or shoot

6.2. Actions

The different available actions are:

- *nothing*: Do nothing.
- *north*, *south*, *east*, *west*: Move in one of the cardinal directions, if there is nothing preventing the tank from doing so (obstacle, other tank, edge of the grid).
- *aim-i*: Aim at tank i (tank which id is i), if this tank is visible by the tank that tries to execute the action.
- *shoot*: Shoot at the tank previously aimed at, if it is visible and action *aim-i* has already been carried out.

When taking action *shoot*, there is a certain probability to kill the target, that depends on the range to the target. This is shown on figure **??**. Explanations about how it was determined are to be found in appendix A.1.

6.3. Parameters

The environment has a lot of parameters that allow to modify its characteristics and working. Those parameters, description and default values are to be found in table 6.1.
Note: One cell = the distance between two adjacent cells.

6.4. Rendering

Two rendering are possible: A rendering of the full environment, as

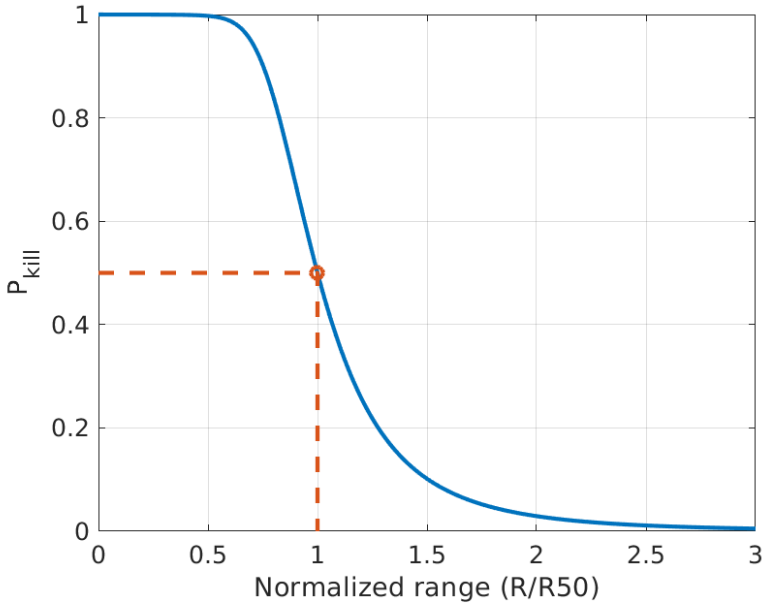

Figure 6.1. P_k as a function of $\frac{R}{R50}$

Parameter	Description	Default value
size	Size of the grid	5×5
players_description	Start position and team of each tank	One tank per team Random start position
visibility	Maximum observable range (euclidean distance)	4 cells
R50	Range at which $P_h = 50\%$	3 cells
obstacles	List of coordinates of all obstacles in the grid	No obstacles
borders_in_obstacles	Edges of the grid added to obstacles	False
max_cycles	Maximum number of transitions (-1 = no limit)	-1
max_ammo	Maximum number of shots per tank (-1 = no limit)	-1
im_size	Number of pixels in height of the render image	480

Table 6.1. Parameters of the *tanksEnv* environment

A. Supplementary Information

A.1. Probability of kill as a function of range

Table 9 from [CRG19] reports the results of an experiment where the ballistic dispersion is measured as a function of range. This is done for one particular laboratory weapon. Table A.1 and figure A.1 show this experimental data as well as a fourth degree polynomial approximation. This approximation is:

$$\hat{\sigma}(R) = 9.53 \times 10^{-13} R^4 - 9.28 \times 10^{-10} R^3 + 3.92 \times 10^{-7} R^2 + 2.42 \times 10^{-5} R + .243 \tag{A.1}$$

Let's take some assumptions:

- The shape of the function that links ballistics dispersion and range is the same for this experiment and for the tanks in the *tanksEnv* environment.
- The aim point (i.e. the mean point of the impacts) of a tank is exactly positioned at the center of the target.
- The target is perfectly round.
- The error E (or ballistic deviation) of one shot is measured as the distance between the center of the target and the actual impact point.
- The error E follows a half normal probability distribution. The CFD (cumulative density function of such a distribution is $f_c(e) = P(E < e) = erf\left(\frac{e}{\sqrt{2}\sigma}\right)$
- The target is always killed when hit.

Range [m]	std [mils]
50	0.25
100	0.25
150	0.25
200	0.26
250	0.26
300	0.27
350	0.27
400	0.28
450	0.29
500	0.3
550	0.31
600	0.32
700	0.36
800	0.43

Table A.1. Standard deviation of the ballistic dispersion as a function of range

According to the previously stated assumptions, the probability to hit the target is:

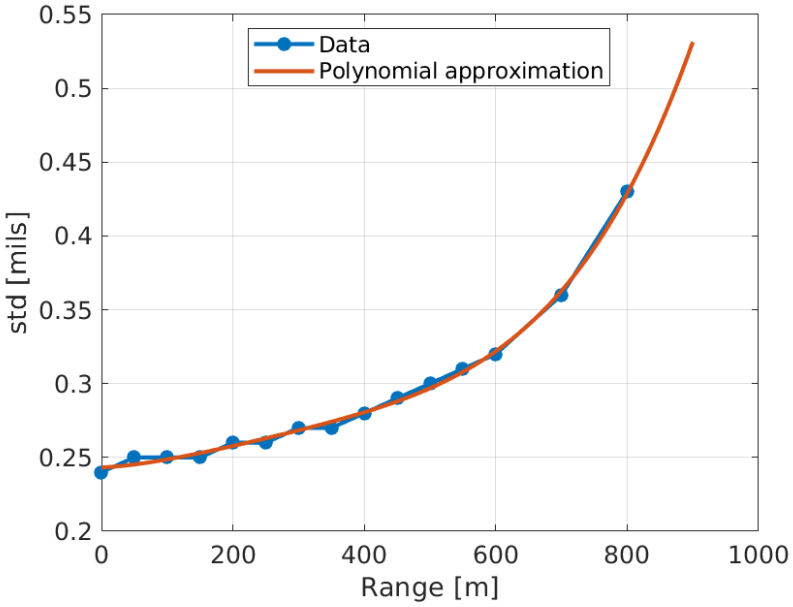

Figure A.1. Standard deviation of the ballistic dispersion as a function of range

$$P_h(R) = P(E < d/2)\Big|_{\sigma=\hat{\sigma}(R)} = erf\left(\frac{d/2}{\sqrt{2}\hat{\sigma}(R)}\right) \tag{A.2}$$

where d is the diameter of the target. This parameter doesn't influence the final result since it will be removed by the normalization and replaced by the R50 parameter defined in the next parameter.

Let's now to express the function $P_k(R)$ (probability of kill in function of the range) that will be used in the *tanksEnv* environment. One of the parameters of the environment is $R50$, the range at which the probability to kill is 0.5. Moreover, for $d/2 = 1$, $P_h(\text{R=1269m}) = 0.5$. Hence (given that the target is always killed when hit):

$$P_k(R; R50) = P_h\left(\frac{1269R}{R50}\right) = erf\left(\frac{1}{\sqrt{2}\hat{\sigma}(\frac{1269R}{R50})}\right) \tag{A.3}$$

Figure 6.1 shows P_k as a function of the normalized range. This final result resemblances a lot to the curves of single shot hit probability in [Var14] (slide 13) which are based on experimental data from APFSDS (armor piercing discarding sabot) shots.

B. (Speed up RNN)

Bibliography

[AA22] Ava Soleimany Alexander Amini. 6s191 - introduction to deep learning. http://introtodeeplearning.com, 2022.

[ale22] alexlenail. Nn-svg. https://github.com/alexlenail/NN-SVG, Mar 2022. [Online; accessed 19. Mar. 2022].

[BOE20] Cmdt Ir Koen BOECKX. *Developing Strategies with Reinforcement Learning*. PhD thesis, Royal Military Academy, 2020. https://github.com/koenboeckx/VKHO.

[CGCB14] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical Evaluation of Gated Recurrent Neural Networks on Sequence Modeling. *arXiv*, December 2014. https://arxiv.org/abs/1412.3555.

[Che18] Guillaume Chevalier. Larnn: Linear attention recurrent neural network. *arXiv*, Aug 2018. https://arxiv.org/abs/1808.05578v1.

[CRG19] D. Corriveau, C.A. Rabbath, and A. Goudreau. Effect of the firing position on aiming error and probability of hit. *Defence Technology*, 15(5):713–720, 2019. https://www.sciencedirect.com/science/article/pii/S2214914719301965.

[HDL16] David Ha, Andrew Dai, and Quoc V Le. Hypernetworks. *arXiv preprint arXiv:1609.09106*, 2016. https://arxiv.org/abs/1609.09106.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv*, Dec 2014. https://arxiv.org/abs/1412.6980.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing Atari with Deep Reinforcement Learning. *arXiv*, December 2013. https://arxiv.org/abs/1312.5602.

[Mus22] Musstafa. Optimizers in deep learning - mlearning.ai - medium. *Medium*, Feb 2022. https://medium.com/mlearning-ai/optimizers-in-deep-learning-7bf81fed78a0.

[neu20] Activation functions | fundamentals of deep learning. https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them, Jul 2020. [Online; accessed 16. Mar. 2022].

[Phi20] Michael Phi. Illustrated Guide to LSTMs and GRUs: A step by step explanation. *Medium*, June 2020. https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21.

[Q-l21] Epsilon-Greedy Q-learning. https://www.baeldung.com/cs/epsilon-greedy-q-learning, Jan 2021. [Online; accessed 16. Mar. 2022].

[rnn22] Cs 230 - recurrent neural networks cheatsheet. https://stanford.edu/~shervine/teaching/cs-230/cheatsheet-recurrent-neural-networks, Apr 2022. [Online; accessed 6. Apr. 2022].

[RSS+18] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: Monotonic value function factorisation for deep multi-agent reinforcement learning. In *International Conference on Machine Learning*, pages 4295–4304. PMLR, 2018.

[SLG+17] Peter Sunehag, Guy Lever, Audrunas Gruslys, Wojciech Marian Czarnecki, Vinicius Zambaldi, Max Jaderberg, Marc Lanctot, Nicolas Sonnerat, Joel Z. Leibo, Karl Tuyls, and Thore

Graepel. Value-Decomposition Networks For Cooperative Multi-Agent Learning. *arXiv*, June 2017. https://arxiv.org/abs/1706.05296.

[Tan93] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *In Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337. Morgan Kaufmann, 1993.

[Var14] Prof James K Varkey. Chance of hit. Dec 2014. https://cupdf.com/document/chance-of-hitpptx.html.