UNIVERSITY OF BERGEN

Department of Informatics

MASTER'S THESIS

Communication in Turn Based Multiplayer Games Using Deep Reinforcement Learning

Author: John Isak Fjellvang Villanger

Supervisor: Troels Arnfred Bojesen

September 1, 2022

Abstract

This work investigates communication in cooperative settings of multi-agent reinforcement learning. We look at what conditions make it easier or harder for meaningful communication to arise between the agents. This includes introducing and showing the usefulness of learning biases in a discrete or continuous setting. In order to do this we extend the game of Negotiation to a continuous setting, introduce a new environment called Sequence Guess, and introduce a new learning bias that helps facilitate the emergence of communication in a continuous setting.

Acknowledgment

First and foremost I would like to thank my supervisor Troels Arnfred Bojesen for the invaluable feedback and encouragement throughout this thesis. I would like to thank my fellow students Johanna, Hans Martin, Knut, Mathias, Emir and Oda, my studies would be a lot less interesting without you. I would like to thank my friend Adriaan Ludl for letting me stay at his place for the final part of my thesis. And finally I would like to thank my family for their support and encouragement.

J.I.F.V

Contents

List of Figures

List of Tables

List of Abbreviations

RL	Reinforcement Learning
MARL	Multi Agent Reinforcement Learning
LSTM	Long Short-Term Memory
RNN	Recurrent Neural Network
ANN	Artificial Neural Network
MDP	Markov Decision Process
DTDE	Distributed Training, Decentralized Execution
CTCE	Centralized Training, Decentralized Execution
CTDE	Centralized Training, Decentralized Execution
ReLU	Rectified Linear Unit

Chapter 1

Introduction

1.1 Motivation

This thesis will look at communication in order to facilitate greater cooperation in a Multi-Agent Reinforcement Learning (MARL) setting. In MARL there are several decision makers also known as agents that interact with a shared environment. When an agent interacts with the environment, the state of the environment changes and the agent is given a reward. The goal for the agent is to maximize the cumulative reward. MARL can be useful as there are many potential applications where agents interact in a shared environment, for example in multiplayer games. In addition, in many MARL settings, communication between agents is desired, as this will grant the agents the ability to share information between each other. In this work there is no predefined language that the agents use, and instead we investigate how meaningful communication protocols can naturally arise between agents. This can help us better understand how simple communication protocols can arise in other settings as well, such as in nature, and how one could expect such communication protocols to behave. One example of how this could be of interest is that it could help us better understand the process that causes communication protocols to arise for organisms such as bacteria.

1.2 Objectives

The main goal of this thesis is to achieve "meaningful communication" between agents. "Meaningful communication" here entails that the communication helps the agents maximize their expected cumulative reward, with the communication itself being a part of the action space of the agents. In order to achieve meaningful communication, the effectiveness of learning biases that can help facilitate the emergence of communication is estimated.

Learning biases here means a bias towards certain communication protocols among all possible communication protocols. In addition, we also wish to find what conditions of the environment make it easier or harder for communication to arise between agents.

1.3 Approach and Limitations

This is an empirical study where simulations are done to investigate if the objectives are achieved or not. This is done by running a simulation with some initial conditions 10 or 20 times, and comparing them to a simulation run with different initial conditions the same number of times. The comparison is done by finding the mean and 95% confidence interval of the samples from the different initial conditions for each iteration. For many cases when comparing two different conditions there is no overlap of the confidence intervals which indicates that the null hypothesis should be rejected at the $\alpha = 0.05$ level. An example of the approach mentioned above is estimating the reward difference if a communication channel is open or not open, all else being equal.

Given more time and more computational resources one could increase the sample size and thus increase the accuracy of the results.

Another limitation is that little is said about the generalizability of the comparison results. This thesis shows how the different initial conditions affect the result in one environment, but does this also apply to other environments as well?

1.4 Contributions

This work introduces a new discrete environment that "encourages" the emergence of communication. We call this environment Sequence Guess and it is explained in section 3.2. This work also introduces a new learning bias that helps facilitate the emergence of communication in a continuous environment, this learning bias is explained in section 3.3.1.

1.5 Outline

The rest of the thesis is organized as follows:

- **Chapter 2 - Theoretical Background:** Provides the necessary domain specific information required to understand this thesis. This includes deep learning, reinforcement learning and multi agent reinforcement learning.

- **Chapter 3 - Method:** Explains in detail the environments, network architectures used, and the metric used for measuring the degree of communication.
- **Chapter 4 - Results and Discussion:** Shows and discusses the results of the experiments, and also discusses some more general topics when it comes to communication in MARL.
- **Chapter 4 - Conclusions and Recommendations for Further Work:** Concludes and and comes with recommendations for further work.
- **Bibliography**

Chapter 2

Theoretical Background

This chapter will provide relevant background knowledge for understanding this thesis. It will begin broadly with deep learning and reinforcement learning, and look at how these two can be combined to form deep reinforcement learning. Then it will look more closely at multi agent reinforcement learning before going into the specifics of cooperation and communication in such a setting.

2.1 Artificial Neural Networks

Artificial neural networks (ANNs) are a class of parameterized functions f_θ, that have been shown to have good function-approximation properties. This is shown by the function approximation theorem [10], where an ANN can under certain conditions approximate any function.

An example of an ANN can be seen in figure 2.1. This ANN can also be viewed as a series mathematical operations, where the weights connecting to a row of nodes are represented as a matrix $\boldsymbol{W}$. The input $\boldsymbol{x}$ and bias $\boldsymbol{b}$ are vectors.

$$\begin{aligned} f^1(\boldsymbol{x}) &= \boldsymbol{z} = \boldsymbol{W^{(1)}}^\top \boldsymbol{x} + \boldsymbol{b}^{(1)} \\ \boldsymbol{f}^2(\boldsymbol{z}) &= \boldsymbol{y} = \boldsymbol{W^{(2)}}^\top \boldsymbol{z} + \boldsymbol{b}^{(2)} \end{aligned} \tag{2.1}$$

f^1 and f^2 in (2.1) are referred to as layers, in this case f^1 and f^2 are specifically *fully connected layers.* The first layer of an ANN is called the input layer, the final layer of an ANN is called the output layer, any other layers are referred to as hidden layers. The number of nodes in a layer is referred to as the width of the layer, while the number of layers is known as the depth. Layers in an ANN is important to satisfy universal function approximation

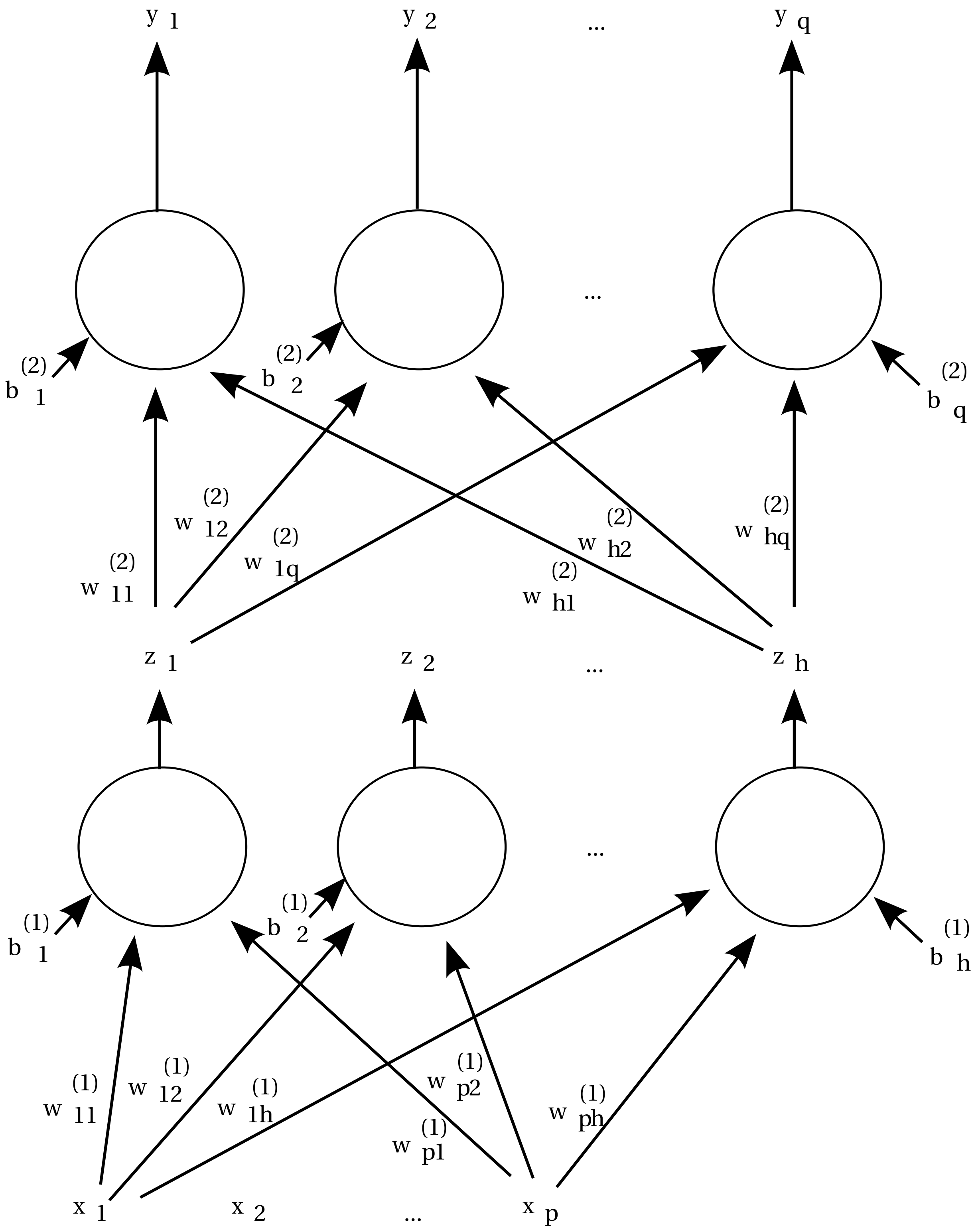

Figure 2.1: An example of a two layer ANN. $w_{x,y}^{(a)}$ are called weights while $b_x^{(a)}$ are called biases. The weights and biases are the parameters of the ANN (θ). The circles are called nodes. For each node the input is multiplied by a set of weights and then summed together with a bias term: $z_1 = w_{11}^{(1)} x_1 + w_{21}^{(1)} x_2 + ... + w_{p1}^{(1)} x_p + b_1^{(1)}$. Image taken from: [17]

properties. One of the conditions necessary for a network of arbitrary width to be able to approximate any function as shown by Hornik [10] is at least one hidden layer.

The purpose of the ANN is to learn a mapping $\hat{\boldsymbol{y}} = f_\theta(\boldsymbol{x})$ that approximates some target function f^*, this is done through the tuning of its parameters. One way to tune the parameters of an ANN is through gradient based optimization. This is possible because ANNs are partially differentiable with regards to their parameters θ:

$$\forall_i \frac{\partial}{\partial \theta_i} f_\theta(x_1, .., x_n), \quad \theta_i \in \theta \tag{2.2}$$

A more detailed look at gradient based optimization will be given in section 2.1.2.

2.1.1 Activation Functions

Activation functions are functions generally applied to the nodes of a layer, for example in the ANN represented equation (2.1) one could apply the activation function g twice:

$$\begin{aligned} \mathbf{a_1} &= \boldsymbol{W}^{\mathbf{(1)}^\top} \boldsymbol{x} + \boldsymbol{b}^{(1)} \\ \mathbf{z} &= g(\mathbf{a_1}) \\ \mathbf{a_2} &= \boldsymbol{W}^{\mathbf{(2)}^\top} \boldsymbol{z} + \boldsymbol{b}^{(2)} \\ \mathbf{y} &= g(\mathbf{a_2}) \end{aligned} \tag{2.3}$$

A linear model mapping from input to output can only represent linear functions. Thus one of the conditions necessary for an ANN of arbitrary width to be able to approximate any function is that the ANN contains non-linear activation functions [10]. Activation functions are also useful in the output layer in order to ensure that the outputs are within a desired range.

Here are the activation functions used in this work:

Hyperbolic Tangent

The hyperbolic tangent function (tanh):

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

The activation function ensures the output is within the range $(-1, 1)$, while $\tanh(0) = 0$. In this work the tanh activation function is used as a way to ensure that "communication" happens within a bounded interval.

Sigmoid

The sigmoid function (σ):

$$\sigma(x) = \frac{1}{1+e^{-x}} \tag{2.5}$$

The activation function ensures the output is within the range $(0,1)$, while $\sigma(0) = 0.5$. This can for example be useful for modelling the probability of picking between two categories.

Softmax

The softmax activation function differs from the ones previously mentioned as it is dependent upon x being a vector:

$$z_i = \frac{e^{x_i}}{\sum_{j=1}^{J} e^{x_j}} \tag{2.6}$$

for $i = 1, ..., J$. The softmax activation function ensures that $\sum_{i=1}^{J} z_i = 1$. This can for example be useful for policy modelling: let's say you have J different possible actions, then each z_i can be viewed as the probability of picking that action. The softmax activation function can be viewed as an extension of the sigmoid function to more than two categories.

Leaky Rectified Linear Unit

The Leaky Rectified Linear Unit is an extension of the Rectified Linear Unit (ReLU). ReLUs use the activation function:

$$g(x) = \max\{0, x\} \tag{2.7}$$

ReLUs are easy to optimize because they are similar to linear units, its derivative is 1 in its active domain. As will be shown in section 2.1.3, there are cases where optimization is easier when the derivative is close to one. The drawback of ReLU is that they can not learn via gradient based optimization when their activation is 0. This can lead to the network consisting of "dead nodes", where if the activation is always 0 it will always be 0 since the gradient will also always be 0. One proposed way to address this issue is the Leaky ReLU.

The Leaky ReLU is defined as:

$$g(x) = \max\{\alpha x, x\} \tag{2.8}$$

α is typically set to a small value like 0.001, this can help alleviate the issue of "dead nodes".

2.1.2 Gradient Based Optimization

Gradient based optimization refers to the process of minimizing or maximizing some function $f(\theta)$, by altering θ through gradient descent or gradient ascent [7]. One refers to $f(\theta)$ as the objective function. If the purpose is specifically to minimize its also referred to as the cost function or loss function. Maximization may be accomplished by a minimizing algorithm by using $-f(\theta)$ or vice versa. Considering θ as a single value, when employing gradient descent: if $\frac{\partial}{\partial\theta}f(\theta) < 0$, θ is increased and if $\frac{\partial}{\partial\theta}f(\theta) > 0$, θ is decreased. Similarly this also applies to when θ is a vector of values, we find $\frac{\partial}{\partial\theta_i}f(\theta)$ for each component in θ, this is known as the gradient.

Since to goal is to find a value for θ such that $f(\theta)$ is minimized, gradient descent proposes a new point by moving in the direction of the negative gradient [7]:

$$\theta' = \theta - \alpha\nabla_\theta f(\theta) \tag{2.9}$$

α is referred to as the learning rate or step size parameter and determines how much to change the value of each parameter with regards to the gradient. If α is too large one can easily overshoot a minimum point by moving too far, if α is too small learning may happen very slowly.

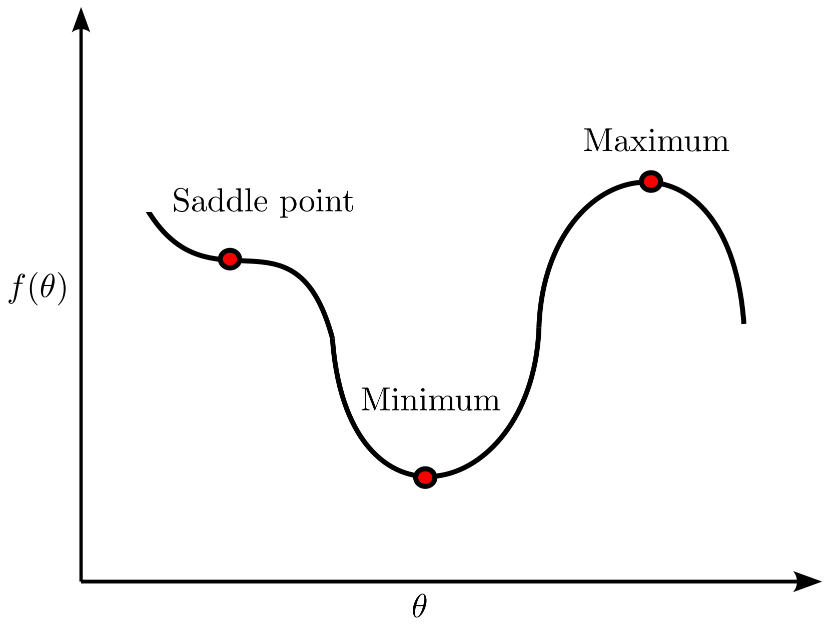

Figure 2.2: The three types of critical points in $f(\theta)$, critical points being points where the gradient is 0. θ here is a single value, in reality the gradient space usually consists of many more dimensions as θ will contain many parameters.

As one can see on figure 2.2 there are three different cases for when $\frac{\partial}{\partial \theta} f(\theta) = 0$, a local or global minimum, local or global maximum and a saddle point. For maximums and saddle points, random noise will usually be enough to allow us to continue the gradient descent towards a global minimum. In practice a local minimum that is not the global minimum will be rare when optimizing ANNs, since the parameter space of an ANN is large. In order to have a local minimum, every single parameter needs to have a derivative of 0 at that point. However one usually does not even find a local minimum, it has been shown by Goodfellow et al. [7], that in many cases the gradient norm increases in tandem with the loss function decreasing.

Tying Gradient Based Optimization Together With ANNs

The loss function is applied to the output of the ANN. Let g be the ANN and f the loss function:

$$L = f(g_\theta(\boldsymbol{x})) \tag{2.10}$$

Then to minimize this function with regards to θ one applies equation (2.9). In practice one does not try minimize for only one value of $\boldsymbol{x}$. One common strategy when minimizing for many different values of $\boldsymbol{x}$ is to utilize mini-batching [7], where one find the average gradient over a set of inputs. The size of this set is referred to as the batch size. When using mini-batching one only estimates the true gradient, as the true gradient would be the average gradient over all inputs. In practice using this estimate of the gradient instead of the true gradient often works better. There is a non-linear relationship between the number of samples and the accuracy of the estimate when estimating the standard error of the mean: $\frac{\sigma}{\sqrt{n}}$. This leads to using a mini-batch size of 10000 compared to one of 100 would only improve the accuracy of the estimate of the true gradient by a factor of 10. While the computational costs associated with using a mini-batch size of 10000 compared to 100 would be a factor of 100.

Gradient based optimization also naturally extends to ANNs that take several inputs and outputs several values. For several outputs one can utilize a loss function that takes the output of the ANN as input and returns a single loss value. Then one can find the gradient by finding the partial derivative with regards to each parameter on the final loss.

Adam Optimizer

The Adam Optimizer is a potential improvement upon the update rule in equation (2.9). It was introduced by Kingma and Ba [12]. When updating θ, instead of purely moving in the direction of the negative gradient with every step, one can in addition use information from previous updates to potentially find a more optimal direction. This can help in the case where one uses mini-batches (which the Adam optimizer is intended for) as the average gradient across several mini-batches will potentially be closer to the true gradient. Adam uses exponential moving averages of the gradient m and the squared gradient v. m is also called momentum, since it essentially adds a momentum term to the gradient descent. With the main idea being that this momentum can help update the parameters in the desired direction since "noise" in the gradient space of the current mini-batch can then partially be ignored. v can be viewed as an "adaptive learning rate" for each individual parameter, by decreasing the magnitude of the update for parameters that generally have a large gradient and vice versa.

Figure 2.3 shows the main idea behind the ADAM optimizer, and how it can help reach a minimum faster.

The moving averages uses the decay rates β_1 and β_2. $(\nabla_\theta f(\theta))^2$ indicates the elementwise square $(\nabla_\theta f(\theta)) \odot (\nabla_\theta f(\theta))$

The update rules for m and v are:

$$\begin{aligned} m' &= m\beta_1 + (1-\beta_1)\nabla_\theta f(\theta) \\ v' &= v\beta_2 + (1-\beta_2)(\nabla_\theta f(\theta))^2 \end{aligned} \tag{2.11}$$

Then one can use m' and v' to update the parameters:

$$\theta' = \theta - \alpha \frac{m'}{\sqrt{v'} + \epsilon} \tag{2.12}$$

where ϵ is some small value used to prevent division by 0. In the original implementation a bias correction is also applied to avoid the bias that comes from initializing m and v as some value.

From equation (2.11) and equation (2.12) one can see how m causes parameters that are generally updated in one direction to continue to update in that direction with every step. While v on the other hand will cause parameters that generally have a gradients with an absolute value larger than 1 to have smaller updates and vice versa.

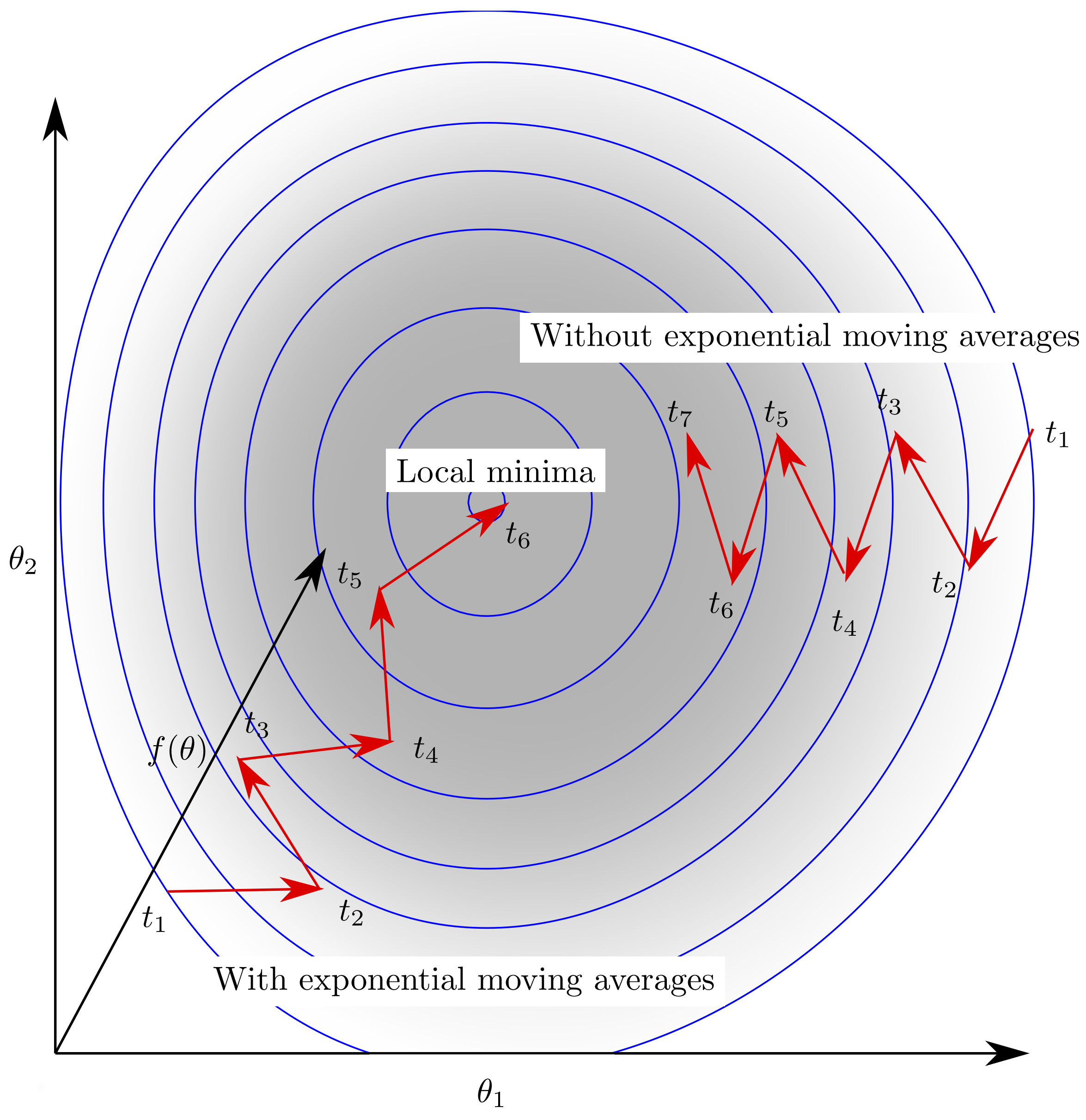

Figure 2.3: Showing gradient descent along a parameter space consisting of two parameters. With and without exponential moving averages over a series of timesteps t. The main idea being that using moving averages can help prevent oscillations along unwanted directions in the parameter space. The local parameter space around the arrows can be imagined as a canyon, and without exponential moving averages one ends up jumping between the canyon walls. Image adapted from: [23].

2.1.3 Recurrent Neural Networks

Recurrent neural networks (RNNs) are a family of networks primarily used for working on sequential data. Consider the the phrases "He was born in 1994" and "In 1994, he was born". Let's say one is assigned the task of extracting the year of birth in sentences. If one is going to use a traditional feed forward network to solve this as in figure 2.1. One would need a different set of parameters to detect "1994" and "born" in the first and second sentence, since they occur at different places. In contrast a RNN could go through the two sequences one word at a time and use the same parameters on each word to detect "born" and "1994", for the two different sentences. It can also "remember" having seen "born" or "1994" since it contains a recurrent hidden state.

Formalizing these thoughts one can define RNNs as ANNs that contain some sort of recurrence. In order to keep things within the scope of this work, we will only look at recurrent layers, but you could also have layers consisting of some recurrent units and some non-recurrent units. In general a recurrent layer can be explained by this equation:

$$\boldsymbol{h}_t = f_\theta(\boldsymbol{h}_{t-1}, \boldsymbol{x}_t) \tag{2.13}$$

$\boldsymbol{h}_t$ is the output of a recurrent layer at timestep t and is also called the hidden state. $\boldsymbol{h}_{t-1}$ is used together with the input $\boldsymbol{x}_t$ to compute $\boldsymbol{h}_t$. In practice this will cause the hidden state to work as some sort of summary of the previous inputs up to timestep t. This leads to the capability of detecting the date of birth in two different sentences while using the same parameters as explained above. In a RNN the recurrent layer(s) are often just one part of the network. For example h_t can be the input to some fully connected layers as in figure 2.1 that computes the final output of the network.

A recurrent layer can also be unfolded when t is finite, for example for $t = 3$:

$$\begin{aligned}\boldsymbol{h}_3 &= f_\theta(\boldsymbol{h}_2, \boldsymbol{x}_3) \\ &= f_\theta(f_\theta(\boldsymbol{h}_1), \boldsymbol{x}_2), \boldsymbol{x}_3) \\ &= f_\theta(f_\theta(f_\theta(\boldsymbol{h}_0, \boldsymbol{x}_1), \boldsymbol{x}_2), \boldsymbol{x}_3)\end{aligned} \tag{2.14}$$

The unfolded recurrent layer is mathematically identical to its folded counterpart. This helps illustrate how vanishing and exploding gradients can be a common problem when utilizing RNNs.

Vanishing and Exploding Gradients

As more layers and certain activation functions are added to a network, the chance of gradients approaching zero or infinity increases. This is further exacerbated in RNNs where one repeatedly multiplies by the same weight matrix. Goodfellow et al. [7] gives an example of this: Suppose you have a very simple RNN lacking a nonlinear activation function and inputs $\boldsymbol{x}$:

$$\boldsymbol{h}_t = \boldsymbol{W}^\top \boldsymbol{h}_{t-1} \tag{2.15}$$

After t steps this will be equivalent to multiplying by:

$$\boldsymbol{h}_t = (\boldsymbol{W}^t)^\top \boldsymbol{h}_0 \tag{2.16}$$

Then assume there is an eigendecomposition of $\boldsymbol{W}$ of the form:

$$\boldsymbol{W}^t = (\boldsymbol{V}\text{diag}(\boldsymbol{\lambda})\boldsymbol{V}^\top)^t = \boldsymbol{V}\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}^\top \tag{2.17}$$

Suppose $\boldsymbol{V}$ is orthogonal, then the recurrence relation may be simplified to:

$$\boldsymbol{h}_t = \boldsymbol{V}^\top\text{diag}(\boldsymbol{\lambda})^t\boldsymbol{V}\boldsymbol{h}_0 \tag{2.18}$$

With repeated multiplication of the same eigenvalues, the eigenvalues will approach zero or infinity if they are not close to 1. Goodfellow et al. [7] States that: "The **vanishing and exploding gradient problem** refers to the fact that the gradient through such a graph are also scaled according to $\text{diag}(\boldsymbol{\lambda})^t$." When the gradients vanish learning may happen very slowly as updates to the parameters will become vanishingly small. When the gradients explode learning may become very unstable as one can easily overshoot a minimum for some axis in the gradient space.

One architecture that attempts to adresses the problem of vanishing gradients in RNNs is the Long Short-Term Memory.

Long Short-Term Memory

Long Short-Term Memory (LSTM) is a form of gated recurrent layer introduced by Hochreiter and Schmidhuber [9]. The LSTM consists of two hidden states, $\boldsymbol{h}_t$ and $\boldsymbol{c}_t$. $\boldsymbol{h}_t$ is also the output of the LSTM layer. The LSTM layer also contain several "gates". These "gates" help

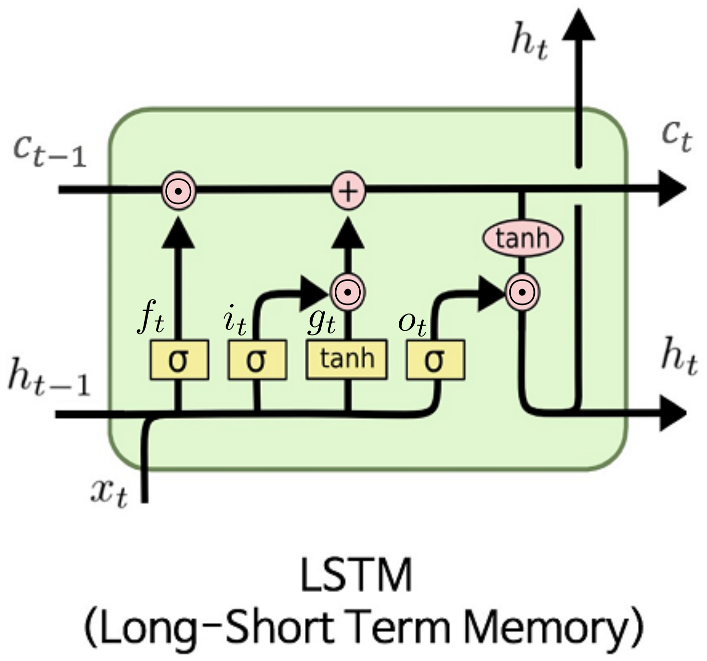

Figure 2.4: One LSTM layer visualized. $\boldsymbol{x}_t$ and $\boldsymbol{h}_{t-1}$ are used as input arguments to the gates(f_t, i_t and o_t), and to the input unit g_t. All of these functions generate vectors, $\odot$ indicates the Hadamard product of two vectors and + means the vector sum. Image adapted from: [18].

determine what information should be stored in the hidden states. For the sake of the simplicity of this explanation assume the gates are vectors consisting of zeroes and ones, a value of 1 would mean that the information is let through and vice versa. In reality a sigmoid activation function is used. The values will still be very close to one or zero in many cases when using a sigmoid activation function.

The three gates in order of use within an LSTM are:

- *The forget gate* (f_t) determines based upon $\boldsymbol{h}_{t-1}$ and the current input $\boldsymbol{x}_t$, what information should be forgotten in $\boldsymbol{c}_t$.
- *The update gate* (i_t) determines based upon $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$, what information should be added to $\boldsymbol{c}_t$.
- *The output gate* (o_t) determines based upon $\boldsymbol{h}_{t-1}$ and $\boldsymbol{x}_t$ what information from $\boldsymbol{c}_t$ should be used in order to generate $\boldsymbol{h}_t$. $\boldsymbol{h}_t$ is also the output of the recurrent layer at timestep t.

Figure 2.4 shows a LSTM layer with the gates marked. Based upon this figure and the explanation above showing the equations that govern the LSTM layer follows:

$$\begin{aligned}
\text{Update Gate: } i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{t-1} + b_{hi}) \\
\text{Forget Gate: } f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{t-1} + b_{hf}) \\
\text{Input Unit: } g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{t-1} + b_{hg}) \\
\text{Output Gate: } o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{t-1} + b_{ho}) \\
c_t &= f_t \odot c_{t-1} + i_t \odot g_t \\
h_t &= o_t \odot \tanh(c_t)
\end{aligned} \tag{2.19}$$

σ is the sigmoid activation function. $\odot$ is the Hadamard product. One reason for the tanh activation function is to keep values from exploding over several recurrent steps. Showing the range of the functions and dimensions of the matrices is useful in order to better understand the equations above:

Variables:
$x_t \in \mathbb{R}^d$
$f_t \in (0,1)^h$
$i_t \in (0,1)^h$
$o_t \in (0,1)^h$
$g_t \in (-1,1)^h$
$h_t \in (-1,1)^h$
$c_t \in \mathbb{R}^h$
$W_i \in \mathbb{R}^{h\times d}$, $W_h \in \mathbb{R}^{h\times h}$, $b \in \mathbb{R}^h$
d refers to the number of components in the input($\boldsymbol{x}_t$), h refers to the number of components in the hidden state ($\boldsymbol{h}_t$). It should be noted that the input weight matrices are of shape $h \times d$ in order for all vectors to have the same the number of components so that one can sum them together.

Encoder-Decoder RNN

There exists different RNN architectures depending on the "nature" of the function one desires to approximate. The "nature" can be defined in terms of: the input being a sequence, the output being a sequence or both. We employ RNNs for the case where both the input and output are sequences that are not necessarily of the same length. An example of this case is machine translation. One of the issues a simple RNN such as the one in equation (2.13) would have with machine translation is the fact that the length of the input sequence and the output sequence will not necessarily match, and that the order of words may change from one language to another.

Cho et al. [3] and Sutskever et al. [30] introduced the encoder-decoder architecture as a possible solution to this type of problem. They feed the input sequence to a RNN known as the encoder. The encoder generates hidden states based upon the input sequence. The hidden states can be used in many different ways to generate a context vector, one common and simple way is to set the final hidden state as the context vector. The context vector is then used in the decoder in one or more of several possible ways: One could for example initialize the hidden state of the decoder as the context vector, or one can use the context vector as the input to the decoder.

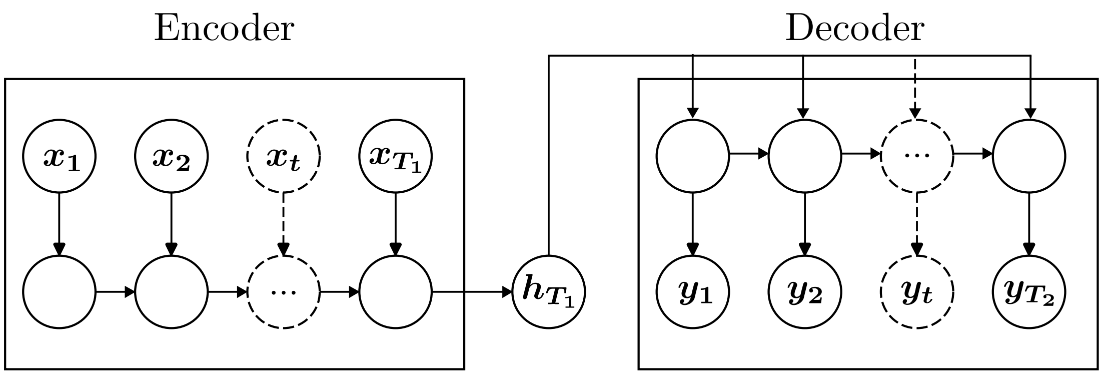

Figure 2.5: An example of an encoder-decoder architecture, an encoder RNN is fed some input sequence of length T_1. The final hidden state of the encoder is used as the context vector and is the input to the decoder RNN for every timestep in order to generate some sequence of length T_2. Both the encoder and decoder RNN could for example be LSTM layers.

Figure 2.5 shows one possible encoder decoder architecture. The architecture shown is the one used in this work.

2.2 Reinforcement Learning

Reinforcement learning (RL) is one of the three main fields of machine learning together with supervised learning and unsupervised learning. In reinforcement learning a decision maker, maps situations to actions, where the goal is to pick those actions that will maximize an expected accumulated reward signal. The reward is generally represented as a real number. For example in Tic-Tac-Toe the situation would be the board state and actions would be possible moves. The reward signal could be zero for all states, except for terminal states that is either a win or a loss, in that case the reward could be $+1$ for a win and -1 for a loss.

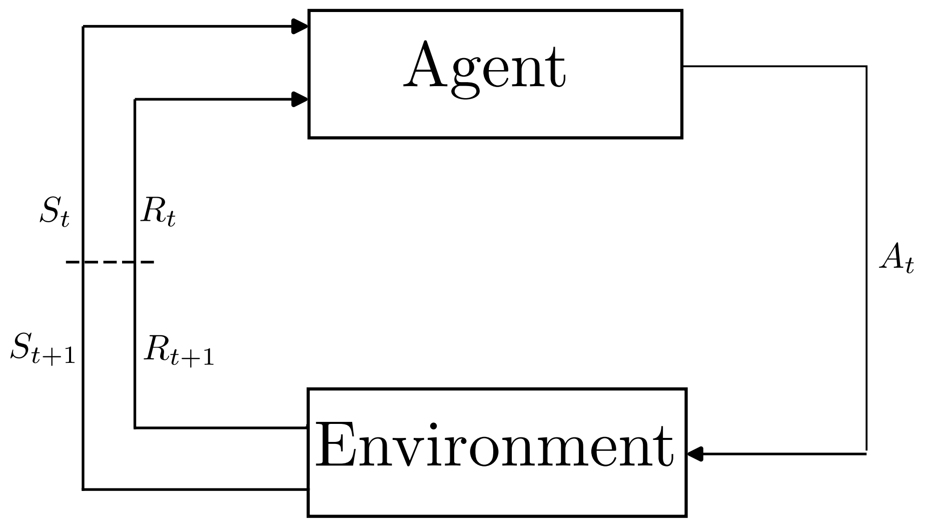

Figure 2.6: The Agent-Environment interaction in a MDP. The Agent is input some state S_t and reward R_t. From this the Agent outputs an action A_t. The action is input into the environment and the environment outputs some state S_{t+1} and reward R_{t+1}. This cycle repeats itself for each timestep t.

2.2.1 Markov Decision Process

Decision making is often defined as a the process of making choices among several possible alternative options. Sequential decision making would thus be a sequence of such choices. When one makes an action based upon a choice, the action will affect the state of the world. This also happens in Markov Decision Processes (MDPs), where an action affects what will be the resulting state. One property of sequential decision making is *intemporal choice*, where decisions made at one point in time affects possible decisions at a later point in time. Formalizing these thoughts gives rise to MDPs.

MDPs provide a mathematical framework for modelling sequential decision making. One of the assumptions of a MDP is that the environment can be split up into a series of discrete timesteps. In a MDP the learner/decision maker is called the agent. The environment encompasses everything that is not the agent. A state is denoted s, the set of all states S. An action a, the set of all actions A. A reward is some number $r \in \mathbb{R}$. The agent interacts with the environment in a series of discrete timesteps $t = 0, 1, 2, 3....$ This gives rise to a *trajectory*:

$$S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2... \tag{2.20}$$

As shown in figure 2.6 the agent is input some representation of the state, decides on an action receives a reward and a new representation of the state, from which the agent can decide on a new action. A given state action pair does not necessarily always result in the same new state, but there is instead a fixed probability for each $s \in S$:

$$p(s' = S_{t+1} | s = S_t, a = A_t) \in (0,1) \tag{2.21}$$

The fact that the state-transition probability in (2.21) is fully explained by the current state-action pair is known as the *Markov property.* The Markov property is useful and many algorithms in RL assume that the Markov property holds. Unfortunately as one shall see in section 2.3.2 this property does not necessarily hold in Multi Agent RL.

The sum of the sequence of rewards from a trajectory is called the return, denoted G.

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} ... + R_T \tag{2.22}$$

where T is the final time step. There being some final time step implies that there is some terminal state, a state from which no more actions can be made. The cases where there exist some terminal state are called episodic cases, when there is no terminal state it is a continuous case. This thesis focuses on the episodic case.

Policy

The agent in a MDP follows some policy. A policy, denoted π, is a mapping from states to probabilities of selecting each possible action. $\pi(a|s)$ denotes the probability that $A_t = a$ if $S_t = s$ at timestep t. The goal in reinforcement learning is to find the optimal policy (π_*), which is defined as the policy that will maximize expected return (G) from any given state.

Value Functions

Value functions are estimates of how "good" it is for an agent to be in state s, or how "good" it is for an agent to perform action a in state s. More formally: A state-value function v_π is a function that takes a state s as an argument and returns the expected return G from that state following policy π. For MDPs the value function for a given state is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} R_{t+k+1} | S_t = s\right] \tag{2.23}$$

where $\mathbb{E}$ is the expectation value under π, i.e the expected return given state s following π, An action-value function for policy π is denoted Q_π, it is the expected return in some state-action pair (s,a) following π. The action-value function for a given state action pair is defined as:

$$Q_\pi(s,a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a] = \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} R_{t+k+1} | S_t = s, A_t = a\right] \tag{2.24}$$

2.2.2 Policy Gradient Methods

The main goal of policy gradient methods is to model a policy as a parameterized function. Often denoted π_θ where θ refers to the parameters. The parameterized function can be anything as long as it is differentiable with regards to its parameters. In practice we also wish to ensure that the policy never becomes deterministic so that learning can happen i.e $\pi_\theta(a,s) \in (0,1)$. This work uses ANNs as π_θ.

Policy gradient methods differ from action-value methods in that instead of learning the value of actions taken, and then selecting actions based on their estimated action values, they select actions without consulting a value function.

Gradient based optimization is used in order to improve the policy. The objective for the episodic case now becomes to optimize the policy so that it maximizes the true value function in the initial state $v_{\pi_\theta}(s_0)$, this is done through gradient ascent:

$$\theta_{t+1} = \theta_t + \alpha\nabla_\theta v_{\pi_\theta}(s_0) \tag{2.25}$$

To tie this with gradient based optimization, section 2.1.2 $v_{\pi_\theta}(s_0)$ is an objective function. Note that we write θ as the arguments to the objective function in section 2.1.2, since in the context of optimizing, the parameters are the variables we change the value of while s_0 would contain fixed constants. In order to make $v_{\pi_\theta}(s_0)$ a loss function that can be optimized with gradient descent all that is needed is to multiply it by -1.

In practice one makes stochastic estimates whose resulting gradient approximates $\nabla v_{\pi_\theta}(s_0)$. This gradient can be found by using the policy gradient theorem.

2.2.3 Policy Gradient Theorem

The policy gradient theorem shows that θ can be updated without consulting a value function [31]. From the definition of a policy 2.2.1 and an action-value function (2.24):

$$\nabla v_\pi(s_0) = \nabla \left[\sum_a \pi(a|s_0) q_\pi(s_0, a) \right] \tag{2.26}$$

Here it is left implicit that π is a function with parameters θ and that the gradients ∇ are with respect to θ. When knowing the gradients of the policy parameters, one knows which way to update the policy parameters in order to maximize the expected return. The problem here, and the reason for the policy gradient theorem is that it can be difficult to compute the true value function. The true value function is dependent upon two things:

- Given a state, the effect of the policy on actions and thus on reward. This can be computed relatively straightforwardly.
- The effect of the policy on the state distribution. This is a function of the environment and is typically unknown. To illustrate this one could imagine a policy to play the "crosses" in tic-tac-toe, let there be another unknown policy to play "circles, by definition the policy of the "circles" player is considered part of the environment when viewed from the perspective of the "crosses" player. Since this policy of the "circles" player is unknown, one can't know the effect of this policy on the state distribution. Thus the effect of the "crosses" policy on the state distribution will also be unknown.

This illustrates the problem that the policy gradient theorem provides a solution to, it gives a metric one can use in order to maximize $v_\pi(s_0)$ without knowing the value of $v_\pi(s_0)$. In order to maximize $v_\pi(s_0)$ through gradient descent one only need to know $\nabla v_\pi(s_0)$.

The policy gradient theorem [31] derives from (2.26) that:

$$\begin{aligned} \nabla v_\pi(s_0) &= \sum_{s'} \eta(s') \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \\ &\propto \sum_s \mu(s) \sum_a \nabla \pi(a|s) q_\pi(s, a) \end{aligned} \tag{2.27}$$

$\eta(s')$ number of time steps on average spent in sate s'. Since only which direction to update θ matters, $\sum_{s'} \eta(s')$ can be removed if the equals is replaced with proportionality ($\propto$). $\sum_{s'} \eta(s')$ would be absorbed into the learning rate parameter of the stochastic gradient ascent anyway. $\mu(s)$ is the on policy distribution over states, how big a proportion of the states encountered is state s under some policy π.

2.2.4 REINFORCE

From the policy gradient theorem (2.27) one can infer that:

$$\nabla v_\pi(s_0) \propto \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a|S_t, \theta) \right] \tag{2.28}$$

One can stop here and instantiate an all-actions algorithm. This would entail for each state encountered: looking at every possible action, this probably requires an estimate to the expected return from an action-state pair instead of stochastically sampling it. One way to estimate the expected return from an action-state pair could be to use a parameterized action-value function. For REINFORCE instead, one only looks at action A_t, the action actually taken at time t. Continuing from equation (2.28), one can multiply by $\frac{\pi(a|S_t,\theta)}{\pi(a|S_t,\theta)}$:

$$\nabla v_\pi(s_0) \propto \mathbb{E}_\pi \left[\sum_a \pi(a|S_t,\theta) q_\pi(S_t, a) \frac{\nabla \pi(a|S_t,\theta)}{\pi(a|S_t,\theta)} \right] \tag{2.29}$$

Replacing all actions with one action sampled from π:

$$= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right]$$

Because $E_\pi [G_t|S_t, A_t] = q_\pi(S_t, A_t)$:

$$= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t|S_t,\theta)}{\pi(A_t|S_t,\theta)} \right]$$

From this one arrives at the update rule of Reinforce:

$$\begin{aligned} \theta_{t+1} &= \theta_t + \alpha G_t \frac{\nabla \pi(A_t|S_t,\theta_t)}{\pi(A_t|S_t,\theta_t)} \\ &= \theta_t + \alpha G_t \nabla \ln(\pi(A_t|S_t,\theta_t)) \end{aligned} \tag{2.30}$$

As a policy gradient algorithm REINFORCE has some advantages compared to other algorithms in reinforcement learning. Methods based solely upon value functions often employ ϵ-greedy action selection, where ϵ-greedy action selection means selecting the action with highest expected return except for ϵ of the time where you select a random action instead. The reason for including ϵ is to ensure that the agent may learn that some action for a given state leads to a greater return than what the action-value function currently predicts is the optimal action. With REINFORCE on the other hand the policy is updated through gradient descent, and this causes the probabilities for selecting different actions also to change smoothly.

One reason for picking the REINFORCE algorithm is that it is more robust to violations of the Markov property. This is because it does not update its value estimates based on the value estimates of successor states [31]. Another reason for using the REINFORCE algorithm is that this work is mainly based upon Eccles et al. [4] and Cao et al. [2] where they employ the REINFORCE algorithm.

REINFORCE With a Baseline

A baseline for the reinforce algorithm $b(S_t)$ is a value subtracted from the return in the update rule of REINFORCE [31]:

$$\theta_{t+1} = \theta_t + \alpha(G_t - b(S_t))\nabla \ln(\pi(A_t|S_t,\theta)) \tag{2.31}$$

A baseline can help speed up learning by reducing variance. This generally happens when the baseline is close to the same value as the return. The introduction of a baseline is a generalization of the REINFORCE algorithm as the baseline can be 0.

We can show that a baseline will not affect the expected policy gradient as long as it is not dependent upon the current action and the policy gradient, by including a baseline in the policy gradient theorem:

$$\nabla v_\pi(s_0) \propto \sum_s \mu(s) \sum_a \nabla\pi(a|s)(q_\pi(s,a) - b(s)) \tag{2.32}$$

Based upon equation (2.32) one can show that the baseline will not affect the expected policy gradient:

$$\sum_a \nabla\pi(a|s)b(s) = b(s)\sum_a \nabla\pi(a|s) = b(s)\nabla 1 = 0 \tag{2.33}$$

2.3 Multi Agent Reinforcement Learning

Multi Agent Reinforcement Learning (MARL) is a sub-field of reinforcement learning. Single agent reinforcement learning involves a single agent interacting with some environment. With MARL on the other hand, you have several agents interacting with a shared environment. Depending on the task an interplay may arise between the agents; they can cooperate in order to achieve a shared goal, or they can compete and try to excel each other.

One can subdivide MARL into three categories. These categories are based upon the reward structure of the environment:

- **Full Cooperation**. All agents will receive the same reward for any state transition.
- **Full Competition**. The sum of rewards equal zero for any state transition, this is also known as a zero sum game.
- **Mixed**, does not satisfy any of the previously mentioned.

2.3.1 Formal Definitions in MARL

Since one now deals with several interacting policies it is useful to define a joint policy as the collection of all individual policies: $\boldsymbol{\pi} = \{\pi^1, \pi^2, \pi^3, ...\}$ with $\boldsymbol{\pi}^{-i}$ being the joint policy without π^i. $v^i_{\boldsymbol{\pi}}(s)$ is the expected return in state s for agent i given joint policy $\boldsymbol{\pi}$.

Definition 1. Best response [8]. Agent i's best response π^i_* to the joint policy $\boldsymbol{\pi}^{-i}$ is when:

$$v^i_{\pi^i_*, \boldsymbol{\pi}^{-i}}(s) \geq v^i_{\pi^i, \boldsymbol{\pi}^{-i}}(s) \tag{2.34}$$

for all states s and policies π^i. The best response is the optimal policy for some agent given a joint policy.

Definition 2. Nash Equilibrium [8]. A solution where each agent's policy π^i_* is the best response to the remaining joint policy $\boldsymbol{\pi}^{-i}_*$ such that the following inequality holds for all states s and all policies π^i:

$$v^i_{\pi^i_*, \boldsymbol{\pi}^{-i}_*}(s) \geq v^i_{\pi^i, \boldsymbol{\pi}^{-i}_*}(s) \tag{2.35}$$

One can imagine a Nash equilibrium as a "global" or "local" maximum as any updates to a single policy will only result in a lower or the same return for that policy. A Nash equilibrium is not necessarily a solution with the highest joint return, and a MARL setting may contain more than one Nash equilibrium.

Definition 3. Pareto optimality [8]. A joint policy $\boldsymbol{\pi}$ Pareto-dominates another joint policy $\hat{\boldsymbol{\pi}}$ iff:

$$v^i_{\boldsymbol{\pi}}(s) \geq v^i_{\hat{\boldsymbol{\pi}}}(s) \forall_i, \forall_s \in S \text{ and } v^i_{\boldsymbol{\pi}}(s) > v^i_{\hat{\boldsymbol{\pi}}}(s) \exists_i, \exists_s \in S \tag{2.36}$$

A policy $\boldsymbol{\pi}$ is regarded as Pareto-optimal if no policy $\hat{\boldsymbol{\pi}}$ can Pareto-dominate it. Note that a Nash equilibrium is not necessarily Pareto-optimal, but it follows from the definitions that a Pareto-optimal solution is also a Nash equilibrium in a fully cooperative setting.

2.3.2 Challenges in MARL

MARL is a nascent field, and is still undergoing several challenges. Some of the main challenges of MARL are:

Credit Assignment Problem

The credit assignment problem arises in a fully cooperative setting. In a fully cooperative setting the reward is the same for all agents. This leads to a sub optimal reward signal for each agent since the reward signal will include the contributions of every agent. This can make it harder to conclude what impact each individual action has. There has been done research into alleviating the credit assignment problem by finding various ways to factor out each agent's individual contribution [21].

Moving Target Problem

One of the core assumptions of a MDP, is that the transition probabilities remain fixed, i.e. $p(s'|s,a)$ does not change. Unfortunately when introducing several agents this is no longer the case. From the perspective of one agent the transition probabilities $p(s'|s,a)$ will change when the policies of the other agents change. This leads to MARL problems violating the Markov property. Because of this, MARL problems often don't have the same theoretical convergence guarantees that you would find in the single agent case.

Shadowed Equilibrium

Another challenge that has been shown to often arise in MARL is "relative over generalization". Here each agent has a tendency to converge towards a robust policy that pairs relatively well with wide range of other policies. A joint policy $\bar{\boldsymbol{\pi}}$ is shadowed by another joint policy $\hat{\boldsymbol{\pi}}$ in state s if and only if [6]:

$$\exists_{i,\bar{\pi}_i} \text{ such that } v_{\bar{\pi}^i,\bar{\boldsymbol{\pi}}^{-i}}(s) < \min_{j,\hat{\pi}_j} v_{\hat{\pi}_j,\hat{\boldsymbol{\pi}}^{-j}}(s). \tag{2.37}$$

What can often happen is that a more optimal joint policy will be shadowed by these more robust joint policies, if the more optimal joint policy can "collapse" when one of the agents change their policy.

2.3.3 Training and Execution Schemes

MARL can be divided into three combinations of two training and execution schemes. For training schemes you have centralized and distributed training, while for execution schemes you have centralized and decentralized execution.

Distributed Training, Decentralized Execution (DTDE)

Each agent has its own policy that maps a local observation to a distribution over individual actions. Information is not shared between agents. One of the biggest drawbacks of DTDE is the moving target problem. One example of DTDE is to have separate ANNs parameterize the policy of each agent while using the REINFORCE algorithm.

Centralized Training, Centralized Execution (CTCE)

Centralized training centralized execution describes a training scheme where a centralized executor learns the joint policy of all the agents. This allows the straightforward employment of single agent training methods, naturally this then also removes some of the issues in MARL, such as the moving target problem. One issue that gets exacerbated by this execution scheme is what is known as the *curse of dimensionality*; The total state action space becomes the product of the state action space of each individual agent. This implies that the computational costs associated with training a centralized executor are exponentially greater than it would be for a single agent.

Centralized Training, Decentralized Execution (CTDE)

Centralized training, decentralized execution is considered the state of the art practice in MARL. CTDE is similar to DTDE, the difference is that information of some type is shared between the agents during training that is not shared during execution. The type of information shared between the agents during training can be highly varied, in the case of homogeneous agents i.e. agents that are input equivalent local states and output equivalent individual actions. It can take the form of parameter sharing. Parameter sharing can for example entail using only one policy network for all the agents, it has been shown that parameter sharing will most likely speed up the learning process [8], what differentiates this method from CTCE is that the policy network will only receive the observations of a single agent at a time, and output the actions of a single agent at a time. Another way to centralize the training has been to use a centralized critic when employing Actor-Critic methods[1]. Here different techniques have also been tried, such as having the critic marginalizes out a single

[1]Actor-Critic methods in comparison to the REINFORCE algorithm learn during the episode, that is it bootstraps. In order to do this, it employs a value function $\hat{v}$ like the REINFORCE with baseline does. The value function is used to estimate the one step return. That is given A_t results in R_{t+1} and S_{t+1} the update rule for the policy parameters will be:

$$\theta_{t+1} = \theta_t + \alpha(R_{t+1} + \hat{v}(S_{t+1}) - \hat{v}(S_t))\nabla \ln(\pi(A_t|S_t,\theta)) \tag{2.38}$$

Note that $R_{t+1} + \hat{v}(S_{t+1})$ is the one-step return estimate and $-\hat{v}(S_t)$ is a baseline.

agents contribution from the global reward [5].

2.3.4 Cooperation in MARL

According to Lindenfors [15] a definition of cooperation within the field of biology is: *Cooperation is the collective functioning of some kind of units for the benefit of themselves and/or their component parts.* Cooperation within MARL also fit into this definition. Interestingly one can in addition also determine from the perspective of an agent "what is to the benefit of itself" by deciding on the reward function. From this, it naturally arises to define the collaborative nature of a MARL problem as the *degree to which the agents rewards are positively correlated.* From this one also arrives at the definition of a fully cooperative setting in section 2.3. The link between reward structure and cooperative behaviour has been confirmed by Leibo et al. [13], where they show that the degree to which the rewards are shared helps determine the degree of cooperation.

2.3.5 Communication in MARL

A lot of the recent research in MARL has been focused on Communication [8]. The main reasons for this is that many issues in MARL can be partially solved if communication is properly incorporated. For example partial observability can be loosened if agents are able to communicate information about their observed state to other agents. Partial observability refers to cases where each individual agent is not input the entire state but instead only a partial observation of it, this is often done intentionally when one desires to achieve communication in a MARL setting [4][2]. Another reason for intentional partial observability is due to the curse of dimensionality, and so learning can be easier if only the relevant information about the state is given to the agent. There are settings where better performance has been achieved by partial observability and communication rather than almost full observability [29].

The moving target problem can also be partially solved if agents communicate information about their policy to other agents.

One issue faced when trying to achieve communication is the shadowed equilibrium [8]. If some agents achieved meaningful communication and the actions of the agents are largely based upon the communication channel, then the joint policy becomes vulnerable to changes in the communication protocol. Some parameter update may cause the communication protocol to collapse and the joint reward to decrease drastically. Because of this what can happen is that the joint policy instead converges towards a more robust sub-optimal policy that ignores the communication channel.

This work focuses upon the case where communication is done through action selection in a MARL setting. There has however also been other approaches to communication in MARL, such as the sharing of the hidden states of the parameterized policies between networks [29][24][14]. There are also approaches that straddle the line between these two main approaches such as by Jiang and Lu [11] where the decision to communicate is treated as an action but the communication itself is the sharing of some hidden states. The reason this work uses the approach of having communication done through action selection is that communication as an action more closely models communication in other settings, where there often is some decision maker that decides what it wants to say among several possible options. It can be interesting to see what properties communication will have in such a setting and if parallels can be found to communication in other settings.

When there is a communication channel with no predefined meaning for messages we define such messages as *utterances*. If the messages are discrete then they are a sequence of symbols of some length, the set of all symbols is referred to as the *alphabet size*.

2.3.6 Positive Signalling

Positive signalling is one proposed learning bias to help facilitate communication in a MARL setting where communication is done through action selection. Positive signalling and positive listening was first introduced by Lowe et al. [16] in order to distinguish cases of real communication from pathological ones. If an agent exhibits positive signalling then its messages will be statistically dependent on its current action or its current observation. While if an agent exhibits positive listening then different messages to the agent will produce different actions. Eccles et al. [4] developed the idea of positive signalling and positive listening further and introduced them as loss functions in order to facilitate the emergence of communication in a discrete MARL setting. For positive signalling an agent that produces messages is encouraged to produce different messages in different situations. This is done by adding a loss term that incentivizes the speaker to produce messages that are uniformly random overall, but where each message is non-random when conditioned upon the current trajectory. They denote $\overline{\pi^i_M}$ the average message policy for agent i over all trajectories, weighted by frequency. $\overline{\pi^i_M}$ is estimated from the current mini-batch:

$$\overline{\pi^i_M} \approx \frac{1}{BT} \sum_b^I \sum_t^T \pi^i_M(\cdot|x^i_{b,t}) \tag{2.39}$$

B is the batch size and T is the length of the trajectory. $\pi^i_M(\cdot|x^i_{b,t})$ denotes the distribution over messages the policy outputs given $x^i_{b,t}$. In order to produce messages that are uniformly random overall they seek to maximize the entropy of the average message policy, $\mathcal{H}(\overline{m^i_t})$. While in order to produce messages that are non-random when conditioned upon the current trajectory they seek to minimize the entropy of the message policy when conditioned upon the current trajectory, $\mathcal{H}(m^i_t|x^i_t)$. This then gives us an equation with two terms. M^i is the set of all messages agent i can choose from. The positive signalling loss for agent i at timestep t, with regards to their current trajectory x^i_t is then:

$$\begin{aligned} L_{\text{ps}}(\pi^i_M, x^i_t) &= -\mathcal{H}(\overline{m^i_t}) + \mathcal{H}(m^i_t|x^i_t) \\ &= \sum_{m\in M_i} \overline{\pi^i_M}(m)\ln(\overline{\pi^i_M}(m)) - \sum_{m\in M_i} \pi^i_M(m|x^i_t)\ln(\pi^i_M(m|x^i_t)) \end{aligned} \tag{2.40}$$

In practice however, minimizing the entropy when it is conditioned upon the current trajectory does not work well. One reason for this may be according to Eccles et al. [4]: "for any $c < \log(2)$ the space of policies with entropy at most c is disconnected, in that there is no continuous path in policy space between some policies in this set".

Because of this they instead introduce a target entropy for $\mathcal{H}(m^i_t|x^i_t)$, $\mathcal{H}_{\text{target}}$. One can view $\mathcal{H}_{\text{target}}$ as an exploration parameter, with higher $\mathcal{H}_{\text{target}}$ meaning a higher degree of exploration. With the introduction of $\mathcal{H}_{\text{target}}$ the loss function becomes [4]:

$$L_{\text{ps}}(\pi^i_M, x^i_t) = -\mathcal{H}(\overline{m^i_t}) + \lambda(\mathcal{H}(m^i_t|x^i_t) - \mathcal{H}_{\text{target}})^2 \tag{2.41}$$

We assume λ here is a weighting for the two loss terms.

Note that this loss is only a learning bias, in order to produce the final loss for the message policy following trajectory x at timestep t one also needs to include the loss with regards to, for example the REINFORCE algorithm, the final loss term for such a message policy at timestep t with regards to trajectory x would then be:

$$L(\pi^i_M, x^i_t) = L_{\text{ps}}(\pi^i_M, x^i_t) - G_t \ln(\pi^i_M(m_t|x^i_t)) \tag{2.42}$$

Note that in general when using a learning bias loss, we find the loss with regards to for example the REINFORCE algorithm and add the loss with regards to the learning bias to produce a final loss which we use as a loss function in gradient based optimization.

Chapter 3

Method

This chapter will introduce the two environments used in this work, Negotiation and Sequence Guess. It will also provide the necessary details on the algorithms used for the experiments, including network architecture. And finally it will discuss the metric used in the experiments to measure degree of communication.

3.1 Negotiation

Negotiation is a game that consists of two agents, A and B, who try to distribute a set of n different types of beverages among themselves. Note that they agree on a division of each individual beverage, so the agents can for example split the first beverage evenly. Each agent is assigned a utility for each type of beverage, once a partition of the beverages has been agreed upon they are rewarded in part based upon their assigned utility for each beverage, $\boldsymbol{u} \in (0,1)^n$, the utility for each beverage is sampled from a uniform distribution, and each agent only receives its own utility as input.

As shown in figure 3.1 the agents have two communication channels; a proposal channel and a linguistic channel. The meaning of a proposal is predetermined, a proposal $\boldsymbol{p} \in (0,1)^n$, is a suggested partition of the beverages. The linguistic channel, on the other hand, does not contain such predetermined meanings. Utterances are sent in the linguistic channel; an utterance is a vector $\boldsymbol{z} \in (-1,1)^n$. The hope is for some communication protocol that helps the agents solve their task, to arise within this channel.

For each timestep, the input for an agent is: The current turn, the agents hidden utilities, the current utterance if the linguistic channel is open, and the current proposal if the proposal channel is open. If the proposal or linguistic channel is closed then the channels respective messages will be replaced by a zero vector as input to the agent. The output of an agent is:

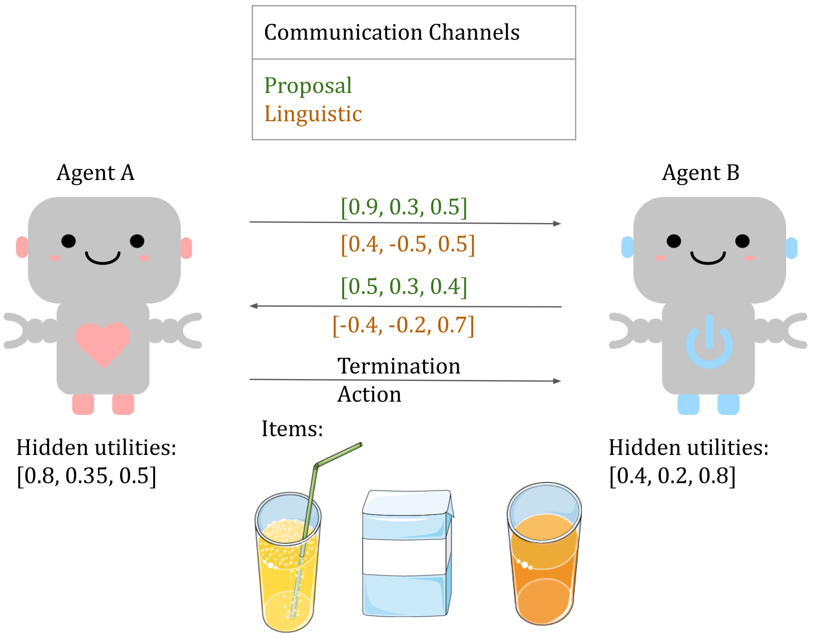

Figure 3.1: An example run of two agents negotiating over three different types of beverages. A proposal of $[0.9, 0.3, 0.5]$ from agent A would mean agent A receives their proposed fraction of each beverage (here: soda, milk and orange juice), while agent B receives the remainder. Agent B can either accept this proposal or come with a counter proposal. The linguistic channel has no predefined meaning. The hidden utilities indicates how each beverage is weighted when calculating the reward. The robots have been taken from [19][20]. The beverages have been taken from [26][25][27].

A proposal, an utterance, and agreement; whether or not an agent agrees to the previous proposal. Agreements are ignored on the first turn.

	Agent A	Agent B
Selfish	$0.5 \times 0.8 + 0.7 \times 0.35 + 0.6 \times 0.5 = 0.95$	$0.5 \times 0.4 + 0.3 \times 0.2 + 0.8 \times 0.4 = 0.58$
Selfish Normalized	$0.945 \div (0.8 + 0.35 + 0.5) = 0.57$	$0.58 \div (0.4 + 0.2 + 0.8) = 0.414$
Cooperative	$0.945 + 0.58 = 1.52$	$0.58 + 0.945 = 1.52$
Cooperative Normalized	$1.52 \div (0.8 + 0.35 + 0.8) = 0.78$	$1.52 \div (0.8 + 0.35 + 0.8) = 0.78$

Table 3.1: Rewards calculated from Figure 3.1 with the final proposal being $[0.5, 0.3, 0.4]$ from agent B. When normalizing the reward it is divided by the maximum possible reward. Selfish refers to strongly competitive setting where individual rewards are not shared. Cooperative refers to a fully cooperative setting where individual rewards are shared. Some values are rounded.

Table 3.1 shows how the rewards are calculated from the example run in figure 3.1, note that for Negotiation the reward and the return will always be the same value. For selfish agents, with a proposal $\boldsymbol{p}$ the proposing agent's (B) reward will be $r_B = \sum_i p_i u_{B_i}$, while the agreeing agent's (A) reward will be $r_A = \sum_i (1 - p_i) u_{A_i}$. If the rewards are shared, meaning that both agents receive the reward $r_A + r_B$ the setting is fully cooperative. The degree of potential cooperation can be tuned by a sharing parameter s_p, where $s_p = 1$ will mean full cooperation and $s_p = 0$ will mean "strong competition". Note that $s_p = 0$ is not full competition as defined in chapter 2.3, since the sum of rewards for any state transition will not equal 0. The reward for agent A with the sharing parameter will then be: $r_A + s_p r_B$. The rewards are normalized by dividing on the maximum possible reward, the maximum possible reward is found by finding the reward of the optimal proposal for each agent. The optimal proposal for agent A with $n = 3$ item categories will satisfy this formula:

$$\forall_{i \in \{1,2,3\}} : (u_{Ai} \geq s_p u_{Bi} \rightarrow p_i = 1) \wedge (u_{Ai} < s_p u_{Bi} \rightarrow p_i = 0) \tag{3.1}$$

I.e for each beverage i, if $u_{Ai} \geq s_p u_{Bi}$ then it is best for A to take everything, else B should take everything. Combing this together a general reward for agent A can be found. Let the reward calculated from the optimal proposal for agent A be $r_{\max}$. And let A's final reward be r_{final}.

$$r_{\text{final}} = \frac{r_A + s_p r_B}{r_{\max}} \tag{3.2}$$

The agents have a set number of discrete timesteps T to reach an agreement, if no agreement is reached within the allotted timesteps they both receive -1 as a reward.

3.1.1 Negotiation Through Communication

One of the main papers this negotiation environment is based on is a paper investigating emergent communication through negotiation by Cao et al. [2]. We have modified the setting from a discrete setting to a continuous one. In their case there are three types of items and between 0 and 5 items of each item type. Each agent is assigned a discrete hidden utility for an item between 0 and 10. An utterance is a sequence of length 5 with an alphabet size of 10. The main reason for extending this work to a continuous setting was curiosity, will the same results be observed in a continuous setting?

3.1.2 Algorithm for Negotiation

For Negotiation the REINFORCE algorithm is used with a parameterized baseline. The policy is parameterized by a single ANN. Consisting of a single LSTM layer and two fully connected layers. Each agent's individual policy is parameterized by its own ANN. The baseline is parameterized by a single LSTM layer and two fully connected layers. Figure 3.2 gives more details on the architecture.

The learning rate α begins at 0.001 for both the baseline and the policies. When an iteration

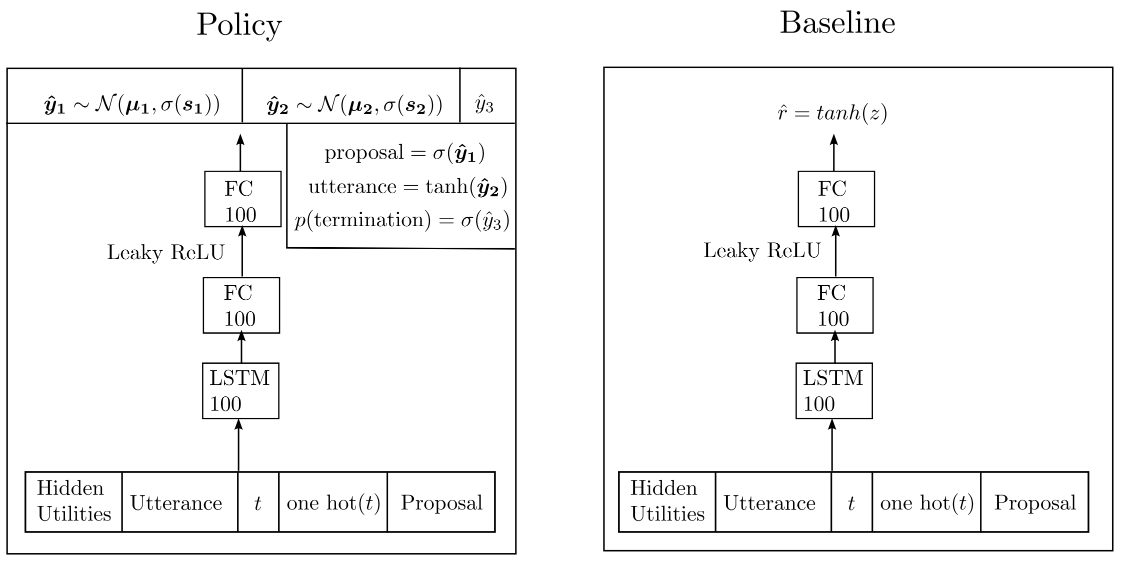

Figure 3.2: The network architecture used in Negotiation both for the policy and the parameterized baseline, t is the current turn. σ is the sigmoid activation function. The policy outputs means values (μs), while standard deviations are found from $\sigma(s)$. These values are used to instantiate normal distributions from which actions are sampled from. The policy also outputs $\hat{y}_3$ that is used to find the probability of termination. The numbers refer to the hidden layer size. FC refers to a fully connected layer. one hot(t) refers to the one-hot encoding of the current turn, i.e a zero vector except for component number t which is one.

Hyperparameter	Value
Batch Size	2048
Iterations	50 000
LSTM Layer Size	100
First Hidden Layer Size	100
Second Hidden Layer Size	100
Utterance Size	3
Max Turns	5
n beverages	3
s_p	1

Table 3.2: Hyperparameters used in Negotiation, the same layer sizes are used for the agents and the parameterized baseline

with $r \geq 0.9$ is reached, the learning rate is reduced to 0.0001 for both the baseline and the policies. $r \geq 0.9$ indicates a close to optimal joint policy if the agents only know their own hidden utilities. The reason for the learning rate adjustment is that the learning can very easily become unstable with a learning rate of 0.001. While learning would be very slow if it began with a learning rate of 0.0001. This approach made the learning more stable. Table 3.2 shows the hyperparameters used. The hidden layer sizes, utterance sizes and number of item categories are the same ones that Cao et al. [2] used for their implementation. A large batch size is picked since this helps make the positive signalling learning bias more accurate, and also because training is done on a GPU. A GPU can handle large batch sizes with little increase in training time. Jiang and Lu [11] also noted that a large batch size helped accelerate their learning process in a MARL setting.

Output of the Policy

Figure 3.2 includes the output of the policy used for Negotiation.

The proposal $\boldsymbol{p}$ and utterance $\boldsymbol{z}$ are sampled from normal distributions.

Each $p \in \boldsymbol{p}$ is sampled from a normal distribution whose mean and standard deviation are outputs from the policy network, a sigmoid activation is used on the output for standard deviations in order to avoid negative numbers and large values for the standard deviation. After sampling from the normal distribution the sigmoid activation function is used in order to ensure a legal proposal. Each $z \in \boldsymbol{z}$ is sampled in the same manner as p except that the tanh activation function is used instead of the sigmoid function. The tanh activation function is used in order to keep the communication channel bounded, if the communication channel isn't bounded then numerical stability will be an issue as the policy may increase or decrease the value of an utterance indefinitely.

The network also outputs the probability of termination from a sigmoid activation, and the termination action is sampled from this probability.

Details on the Baseline

The baseline is centralized, so the baseline always receives the same input as the agent whose turn it is. A negotiation trajectory can be described like this:

$$O_0^A, A_0^A, ..., O_{T-1}^A, A_{T-1}^A, O_T^B, A_T^B, R_T \tag{3.3}$$

Where T is the final timestep, O^A is the observation tensor for agent A. A^A Is the action for agent A. When calculating the baseline for agent A at timestep T, the sequence of states up to A's final timestep $O_0, ..., O_{T-1}$ is used, and when calculating the baseline for agent B $O_0, ..., O_T$ is used. When calculating the loss for the parameterized baseline Mean Squared Error is used:

$$L_A = \frac{1}{I} \sum_{i=1}^{I} (r_A^i - \hat{r}_A^i)^2 \tag{3.4}$$

Where $\hat{r}$ is the output of the network and r is the actual reward. i refers to the sample within a batch of size I. A is the agent the loss is calculated for, the final loss for the baseline will be the sum of the loss when predicting As reward and Bs reward: $L_A + L_B$.

Since learning has been observed to be unstable we have decided to use a parameterized baseline that is potentially able to accurately predict the expected reward from a given state, in order to reduce variance in the reward signal.

The inclusion of a centralized baseline does not affect the idea of the agent's only being allowed to communicate through a communication protocol, because during an episode no information is shared between the agents through the baseline.

3.2 Sequence Guess

We propose a new game inspired by the board game Mastermind. Sequence Guess is a two player game consisting of a mastermind and a guesser. The game consists of the guesser attempting to guess some target sequence, while the mastermind knows the target sequence. The goal for the mastermind is to provide information to the guesser through a communication channel about the target sequence, potentially with respects to the guesses made so far. The length and alphabet size of the target sequence are hyperparameters. The target sequence is sampled uniformly. After each guess the mastermind replies with an utterance. The length and alphabet size of an utterance are also hyperparameters. The game consists of T turns, where T is finite, a turn consists of one guess and one utterance. Figure 3.3 shows an excerpt of Sequence Guess.

The set of all target sequences is referred to as the *Target Language*, while the set of all utterance sequences is defined as the *Utterance language*. Furthermore we define the *Expressivity of a language* as the size of its respective set, so for example a sequence length of 4 and alphabet size of 3 will have an expressivity of $3^4 = 81$. This game differs from Negotiation in a few key aspects, in Negotiation each agent is given the same type of information, and the game is dialogue focused. Sequence Guess, on the other hand, is monologue focused. The mastermind needs to formulate some utterance and the guesser needs to understand how this utterance is related to the target sequence and previous guesses.

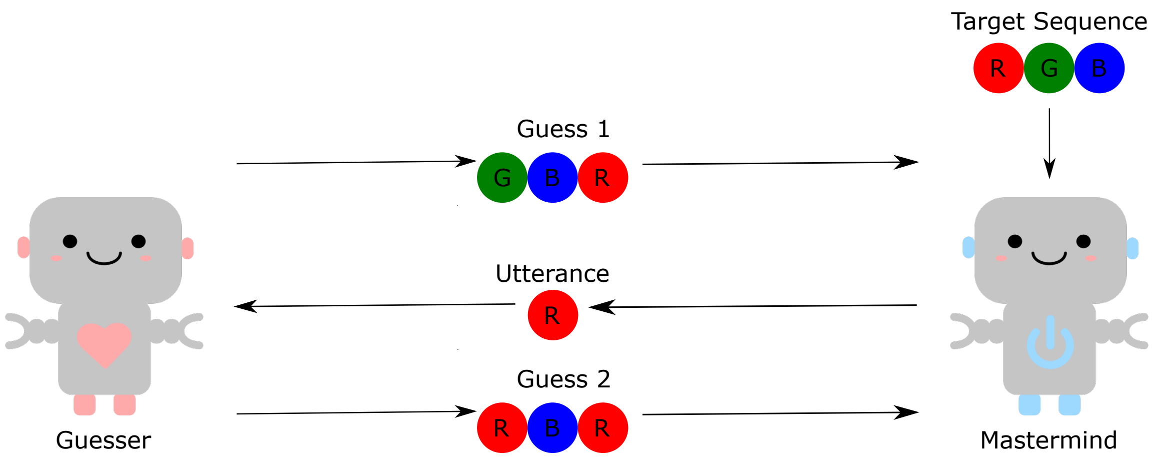

Figure 3.3: An excerpt of Sequence Guess, the guesser attempts to guess some target sequence, while the mastermind tries to provide information about the target sequence to the guesser. The target alphabet size is 3 and target sequence length is 3. Utterance alphabet size is 3 and utterance sequence length is 1. Robots taken from [19][20]

3.2.1 Reward Structure

The meaning of a guess is already predefined and rewards are determined based upon guesses. The guesser receives the latest utterance and current turn number and makes a guess of the target sequence. If it is the correct sequence the game terminates and both agents receive a reward of one, else the game continues, the mastermind receives the correct sequence, the current guess, and current turn number as input and outputs an utterance. On the masterminds turn both agents receive -0.1 in reward, as an incentive to guess the correct sequence as fast as possible. On the final turn they are rewarded according to how good a guess the final guess was. Let J be the target sequence length, for every correct letter in its correct place in the guess, $\frac{1}{J}$ will be added to the reward.

3.2.2 Algorithm

The REINFORCE algorithm with a baseline is used. A moving average is used as the baseline. This is how the baseline is calculated at iteration $t+1$:

$$b_{t+1} = 0.7b_t + 0.3G_t \tag{3.5}$$

Hyperparameter	Value
Batch Size	2048
Learning Rate	0.001
Iterations	100 000
Encoder Size	100
Decoder Size	100 + n turns
Hidden Layer Size	100

Table 3.3: Hyperparameters used in Sequence Guess. The same hyperparameters are used for the guesser and the mastermind. The reasoning behind the choice of hyperparameters is similar to that of Negotiation.

Where G_t is the mean return over the entire mini-batch for the current iteration. This is the same baseline that Cao et al. [2] used. Since the stability issues observed in Negotiation has not been observed in this game, a more simple baseline should suffice.

3.2.3 Network Architecture

DTDE is employed for this setting. Both agents are parameterized by an encoder-decoder architecture using LSTMs. Where a vector is generated from the input sequence, and some output sequence is generated by using this vector as input to the decoder together with a one-hot encoding of the current turn. The reason for the one-hot encoding of the current turn is so that different turns can produce different guesses. This can also be achieved by having the hidden state of the decoder persist across turns as has been done for experiments where the expressivity of the utterance language is smaller than the target language. Figure 3.4 illustrates the encoder-decoder architecture in more detail. The decoder output is used in a fully connected layer. Guesses and utterances are sampled according to the distribution from the softmax activation function in the final output layer.

Table 3.3 shows the hyperparameters used in Sequence Guess.

3.3 Positive Signalling

For Negotiation and Sequence Guess, learning biases are employed to encourage different messages to be produced from different trajectories. The term trajectories is used here since the policies are parameterized by an LSTM so the current action will not only depend on the current observation, but also previous observations. The exception to this is Sequence Guess if the hidden state of the decoder does not persist across turns.

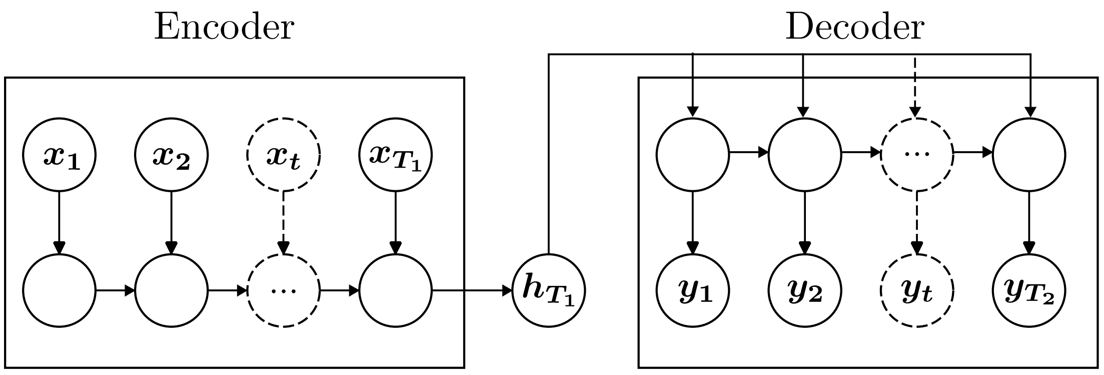

Figure 3.4: The encoder-decoder architecture used for Sequence Guess. *In the case of the mastermind* input x_t will contain symbol number t from the guess sequence and target sequence, T_1 will be the length of the target sequence and T_2 will be the length of the utterance sequence. The output y_t is used in a fully connected layer with a Softmax activation function in order to find utterance symbol number t. *In the case of the guesser* input x_t will contain symbol number t from the utterance, T_1 will be the length of the utterance sequence and T_2 will be the length of the target sequence. y_t is used in a fully connected layer with a Softmax activation function in order to find guess symbol number t. In both cases a one-hot encoding of the current turn is appended to the final hidden state of the Encoder in order to produce the context vector.

3.3.1 Positive Signalling in Negotiation

For Negotiation, positive signalling is not implemented in the same manner as shown by Eccles et al. [4], as they assume a set of discrete symbols is used for communication, but here continuous variables are used instead. Because of this we introduce a new learning bias.

In the case of Negotiation, an utterance consists of three numbers each in the range $(-1, 1)$. Thus an utterance can be viewed as a 3-dimensional vector. Every turn an agent outputs some utterance $\boldsymbol{Z}^{I\times J}$, where I is the size of the current mini-batch and J is the sequence length. In order to produce different utterances for different trajectories, we seek a more uniform distribution of utterances. we do this by trying to maximize the distance between utterances in vector space.

$\boldsymbol{Z}$ is multiplied by $\frac{I}{2}$ in order to account for varying batch sizes (as some games within a mini-batch can terminate before others): $\boldsymbol{Z}_{\text{normalized}} = \boldsymbol{Z}\frac{I}{2}$. $\boldsymbol{Z}_{\text{normalized}}$ is randomly split into two parts of equal size along the mini-batch dimension, $\boldsymbol{Z_1}$ and $\boldsymbol{Z_2}$.

$\boldsymbol{Z_1}$ and $\boldsymbol{Z_2}$ is used to find the mean distance between two utterances:

$$\mu_{\text{dist}} = \frac{2}{I}\sum_{i}^{\frac{I}{2}}\sqrt{\sum_{j}^{J}(z_{1\,i,j} - z_{2\,i,j})^2} \tag{3.6}$$

We then define a loss penalty as:

$$L_1 = \frac{1}{\mu_{\text{dist}} + \epsilon} \tag{3.7}$$

Where ϵ is some small value to prevent division by 0. One issue that arose with this loss function was that several utterances could end up getting "pushed" to -1 or 1 causing the loss to explode. Thus a second loss term is introduced that punishes values that are close to -1 or 1:

$$L_2 = \frac{1}{IJ}\sum_{i}^{I}\sum_{j}^{J}(\frac{1}{1 - z_{i,j} + \epsilon} + \frac{1}{1 + z_{i,j} + \epsilon}) \tag{3.8}$$

Then these two terms are added together to produce a final learning bias loss for the current turn:

$$L_{\text{ps}} = L_1 + \lambda L_2 \tag{3.9}$$

λ is a weighting for the second loss term. We used $\lambda = 0.001$. The L_{ps}s from different turns are summed together in order to produce the final learning bias loss for an agent.

3.3.2 Positive Signalling in Sequence Guess

The mastermind outputs some utterance $\boldsymbol{Z}^{I\times J\times K}$ on his turn. I is the batch size. J is the sequence length and K is the alphabet size. There is a softmax activation function applied over the third dimension. Each value $z_{i,j,k}$ can thus be viewed as the probability of picking letter k in position j of the utterance sequence, with this being game number i within some mini-batch.

Recall the loss function for positive signalling 2.41:

$$L_{\text{ps}}(\pi_M, x_t) = -\mathscr{H}(m_t) + \lambda(\mathscr{H}(m_t|x_t) + \mathscr{H}_{\text{target}})^2 \tag{3.10}$$

Where $\mathscr{H}(m_t)$ is the entropy of an estimate of the average message policy $\overline{\pi}$. $\mathscr{H}(m_t|x_t)$ will in our case be the average entropy of the message policy when conditioned upon a trajectory. $\mathscr{H}_{\text{target}}$ is some target entropy. We estimate the average message policy from the current mini-batch:

$$\bar{\pi} \approx \sum_{i}^{I} \frac{\boldsymbol{U}_i}{I} = \boldsymbol{A} \tag{3.11}$$

The entropy of the approximation of the average policy is then found:

$$\mathscr{H}(m_t) \approx -\sum_{j}^{J}\sum_{k}^{K} \frac{a_{j,k}\ln(a_{j,k})}{J} \tag{3.12}$$

Intuitively one would like to maximize the entropy of the average policy, as target sequences in sequence guess are sampled from a uniform distribution, one would thus also expect the utterances to follow a similar distribution.

Next we estimate the average entropy of the message policy when conditioned upon an input trajectory. The average conditioned entropy is found like this:

$$\mathscr{H}(m_t|\overline{x}_t) \approx -\sum_{i}^{I}\sum_{j}^{J}\sum_{k}^{K} \frac{z_{i,j,k}\ln(z_{i,j,k})}{JI} \tag{3.13}$$

Intuitively one would like to minimize $\mathscr{H}(m_t|\overline{x}_t)$ to ensure that the same input states produce the same messages every time. Yet a target entropy $\mathscr{H}_{\text{target}}$ is still included. What seems to happen is that without $\mathscr{H}_{\text{target}}$ the agent converges quickly upon a sub optimal message policy and gets stuck in this local optimum. $\mathscr{H}_{\text{target}}$ is then included as sort of a exploration parameter increasing the chances of converging towards an optimal policy.

$\mathscr{H}(m_t|\overline{x}_t)$ and $\mathscr{H}(m_t)$ is then used in equation (3.10) in order to produce the learning bias loss for the current turn for the entire mini-batch. The value used for $\mathscr{H}_{\text{target}}$ is $0.1\ln(K)$ which is $\frac{1}{10}$ of the maximum possible entropy. The learning bias loss for each turn in a mini-batch is summed together in order to produce the final learning bias loss.

3.4 How to Measure Communication

In order to measure communication in Sequence Guess and Negotiation the *Return difference*, ΔG is used. Where ΔG is the return difference when running the same experiment with or without communication. Running the experiment without communication means always setting the communication channel to zeroes. As shown by Lowe et al. [16] ΔG is sufficient indicator of communication, however it may not be necessary. In other words if ΔG is big then there is communication happening, but there may also be communication happening without ΔG being big.

Chapter 4

Experiments and discussion

Here we will look at experiments and their results with regards to the objectives. We estimate the effect of positive signalling in Negotiation and Sequence Guess. With regards to finding out more about what conditions of the environment makes it easier or harder for communication to arise between agents we will:

- Discuss similarities and differences between the two games.
- Look at the effects of different levels of expressivity in the target and utterance language in Sequence Guess.
- Discuss the risk of falling into a sub-optimal Nash equilibrium in a strongly competitive setting of Negotiation.

4.1 Similarities and Differences Between the Two Games

One of the main differences between the two games is that Negotiation is symmetric and continuous while sequence guess is asymmetric and discrete.

When it comes to achieving communication Negotiation seems to be the more challenging game. There are several possible reasons for the increased challenge in Negotiation:

1. The difference in reward for optimal play with or without communication is very small. As can be seen in table 4.1, expected reward when not using any communication channel is very close to one. This also applies to the continuous setting of Negotiation. One theory is that since the expected gain from using the communication channel is so small it is easy for the agents to fall into a shadowed equilibrium where they ignore the communication channel.

2. In Negotiation, each agent needs to learn to communicate their hidden utilities, and to understand how other agents communicate their hidden utilities. In sequence guess it is a monologue, one agent needs to communicate the target sequence, while the other agent needs to understand the message of the agent attempting to communicate the target sequence, but not vice versa.

3. It seems easier to converge on a stable communication policy when the communication is done with discrete symbols rather than continuous variables. When using continuous variables every parameter update will either increase or decrease the output value for some input. This seems to result in a less stable communication protocol than the discrete case, due to the "moving" nature of the communication policy. In a discrete case on the other hand every parameter update either increases or decreases the probability of picking some symbol. This results in there being one direction you can update the parameters for greater stability. In the continuous case you can't achieve this kind of stability.

4.2 Results of Cao et al. in a Discrete Cooperative Setting of Negotiation

Cao et al. seem to have a working communication policy for the fully cooperative setting [2], as can bee seen in table 4.1. One interesting thing to note is that how the final turn is decided upon seems to affect how well the algorithm performs with respect to the communication channels. When having random termination between 4 and 10 turns using only the linguistic channel seem to outperform all other cases. The same can not be said when having a termination at turn 10, then, Proposal, Linguistic and Both seem to perform equally well.

Termination turn		Proposal	Linguistic	Both	None
Random 4-10	Fraction of JR	0.93 ± 0.10	0.99 ± 0.02	0.92 ± 0.11	0.95 ± 0.11
	Turns Taken	3.10 ± 0.99	3.71 ± 0.58	2.98 ± 0.97	2.27 ± 0.69
10	Fraction of JR	0.96 ± 0.07	0.95 ± 0.10	0.96 ± 0.07	0.85 ± 0.31
	Turns Taken	9.06 ± 2.50	4.56 ± 2.83	9.06 ± 2.41	3.30 ± 2.50

Table 4.1: Results from Cao et al. [2] for a fully cooperative setting of Negotiation. The authors show joint reward (JR) and turns taken in a fully cooperative setting under different conditions averaged over 20 runs, ± one standard deviation. Termination is either: Random, between turn 4 and 10 at each round, according to a truncated Poisson distribution with mean 7. Or at turn 10

4.3 Positive Signalling

4.3.1 Results in Sequence Guess

For Sequence Guess the findings by Eccles et al. [4] extend to this new game, positive signalling results in an improved performance, As can be seen by the increase in ΔG in Figure 4.1. With positive signalling you can expect convergence to better communication policies than what you would expect without positive signalling. However it does not necessarily converge to an optimal communication policy if the alphabet size and sequence length of an utterance is the same as the target.

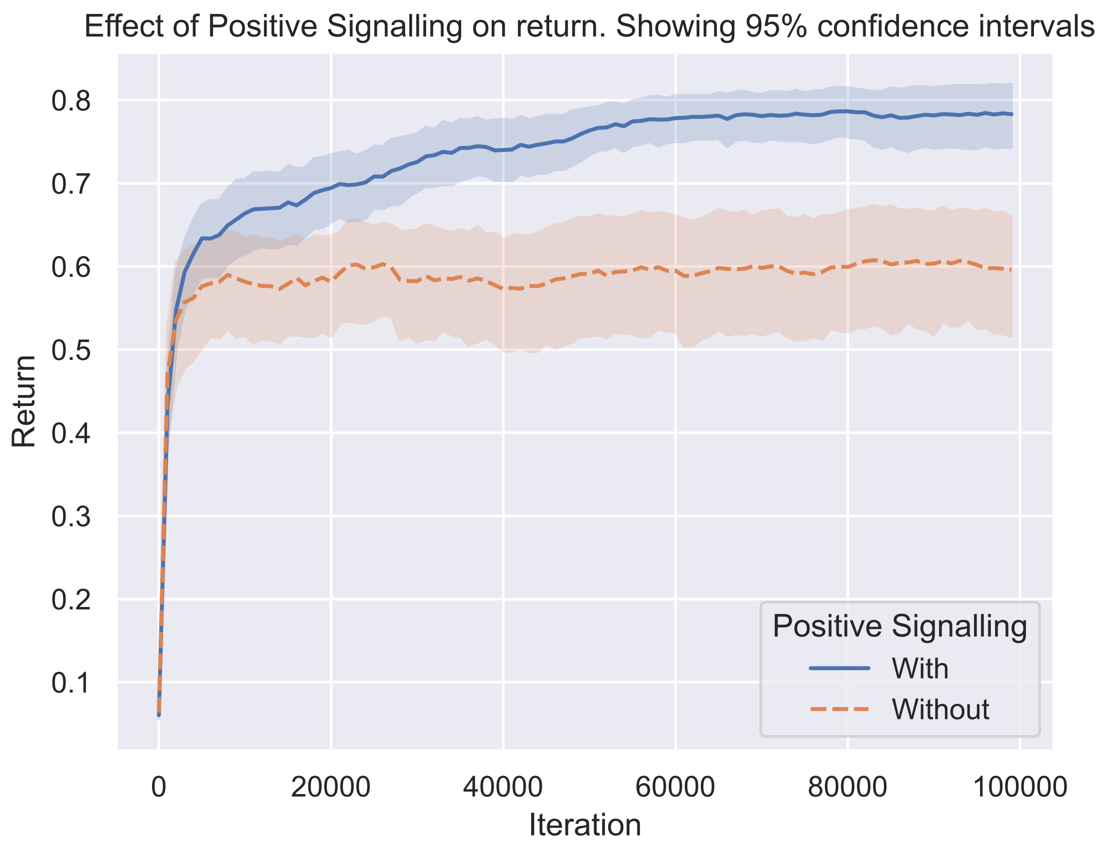

Figure 4.1: This graph shows the average return in sequence guess over 20 runs with and without positive signalling. The figure illustrates how positive signalling causes the agents to converge on better policies. Theoretical optimal return with an incorrect initial guess is 0.9. The maximum number of turns is 5. Both the target and utterance languages have sequence length 3 and alphabet size 3.

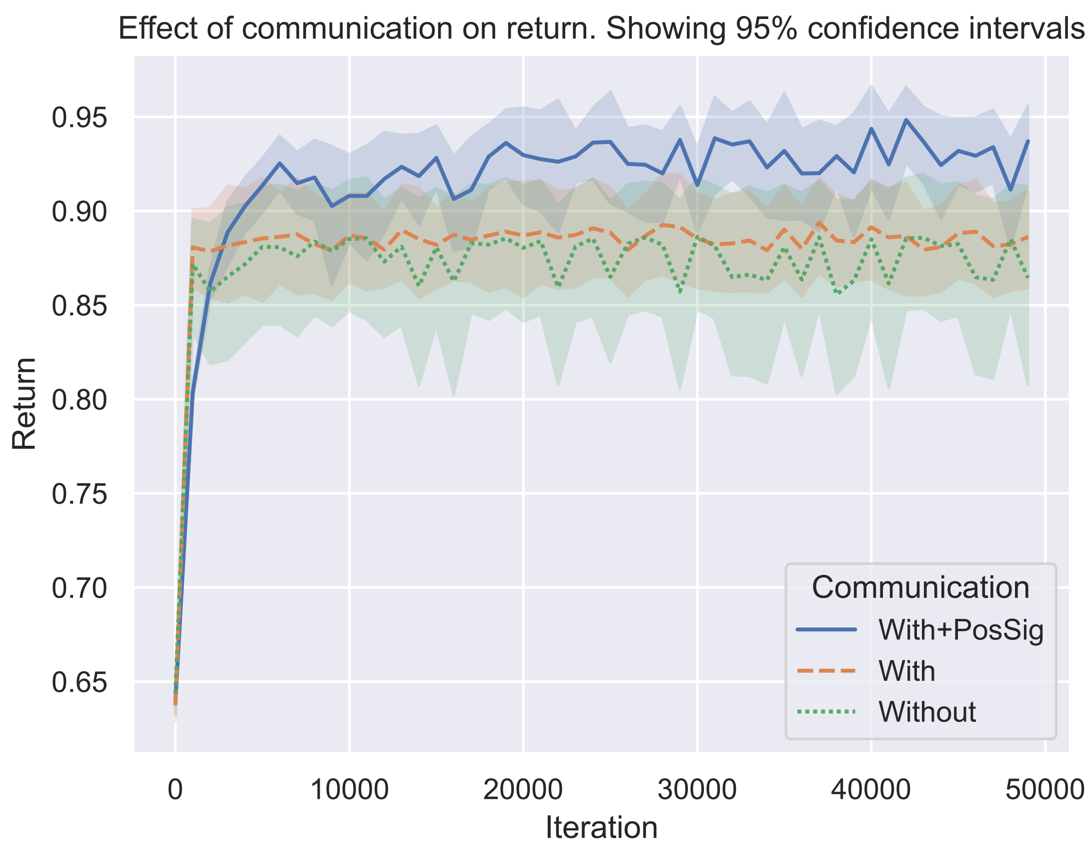

Figure 4.2: This graph shows the average return in negotiation, with 10 runs for each communication setting. The proposal channel is closed to the agents. The maximum number of turns is 5. The figure illustrates how meaningful communication fails to arise without using a learning bias to facilitate communication this setting

4.3.2 Results in Negotiation

For Negotiation the new positive signalling learning bias introduced also improves performance. As can be seen by the increase in ΔG in Figure 4.2. Using communication without positive signalling does not result in any meaningfully improved performance compared to the no communication baseline. What has been observed with positive signalling in Negotiation is that the average return from each iteration oscillates greatly between around 0.985 and around 0.925, where an average return of around 0.985 indicates a working communication policy while an average return of 0.925 indicates very little use of a communication policy. This is most likely due to point 3 in section 4.1.

More generally however there are still some issues with the stability of the policies, this applies to every communication setting in figure 4.2. As can be seen by how wide the confidence intervals are and oscillations of the mean values. Several approaches have been tried such as decreasing learning rate, changing how the baseline is calculated and so on. But it seems that still more work is required to solve this issue. It does however look promising and considering how the main idea of positive signalling applied for a discrete and for a continuous setting, it seems reasonable to assume that this new learning bias specifically will also apply to other continuous settings.

4.4 Differing Sequence Lengths and Alphabet Sizes in Sequence Guess

Here we will look at differing sequence lengths and alphabet sizes in Sequence Guess in order to find out more about what conditions make it easier or harder for communication to arise. Positive signalling is used for all experiments in this section.

4.4.1 Greater Alphabet size and Sequence Length on Target Sequence

One case where the curse of dimensionality does really apply is on the alphabet size and length of the target sequences, as illustrated in figure 4.3, where one can see a clear decrease in expected return as the alphabet size and sequence length increases.

The expressivity of a language increases exponentially with a linear increase in sequence length. For example an alphabet size and sequence length of 4, the language has expressivity $4^4 = 256$. While an alphabet size and sequence length of 5 will lead to an expressivity of $5^5 = 3125$. There are cases where an encoder decoder architecture is not the best architecture, for example if the sequence lengths and sizes are the same the problem can be trivialized to just correlating one letter in each alphabet with each other. We do wish however, to through communication in a RL setting find more general solutions that can apply to varying sizes and sequence lengths of the two languages.

4.4.2 Higher Expressivity for Utterances than the Target

One of main problems that arise during Sequence Guess that positive signalling helps alleviate, is that the the agents pretty quickly converge on some solution to the problem but that this solution is not necessarily close to an optimal solution. When the agents converge on a sub-optimal solution; The mastermind will in many cases produce the same utterance for two different target sequences, but divergence from this policy will still lead to a lower expected return. However if the utterance sequence is longer than the target sequence length, and/or the alphabet size is greater than target alphabet size. There is a greater chance there may be some letter in the utterance that can be changed in order to differentiate the two

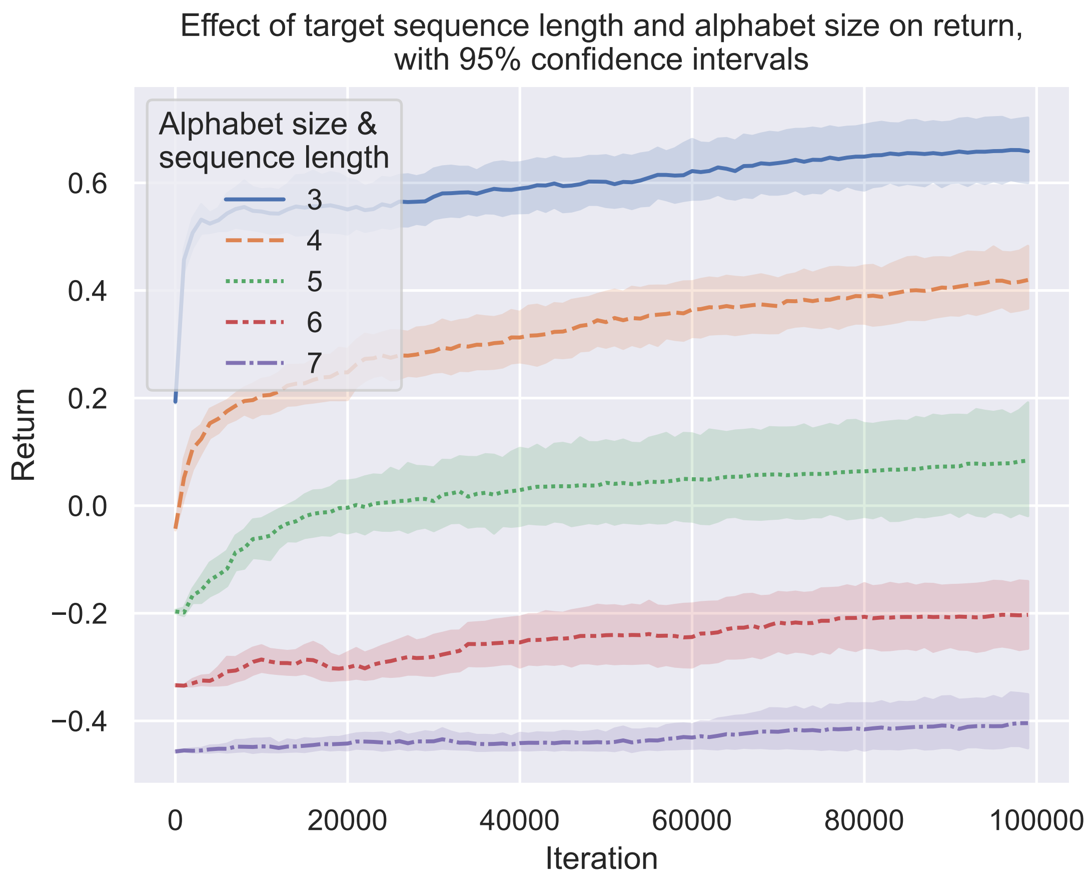

Figure 4.3: This graph shows the average return over 10 runs. The maximum number of turns is 3. The utterance alphabet size and length is also the same as the target alphabet size and length. The figure illustrates how the curse of dimensionality applies and how further investigation is needed to find protocols that can handle target sequences of high expressivity.

target sequences. Comparing the results in figure 4.4 with those in figure 4.3 illustrates this point. In conclusion, the curse of dimensionality does not necessarily apply to the communication policy and big action spaces can be helpful.

4.4.3 Lower Expressivity for Utterances than the Target

Figure 4.5 shows cases for where the mastermind can't theoretically provide all the information about the target sequence in just one turn. Interestingly, with a target size of 3 and utterance size of 1 a close to optimal solution is observed, however this is not the case for the other language lengths. This may be due to the fact that in these other cases the utterance size is greater than 1, this indicates that the algorithm currently seems more optimized for utterances of a single symbol rather than a sequence. This indicates that the use of an encoder may not be an optimal solution for this type of problem.

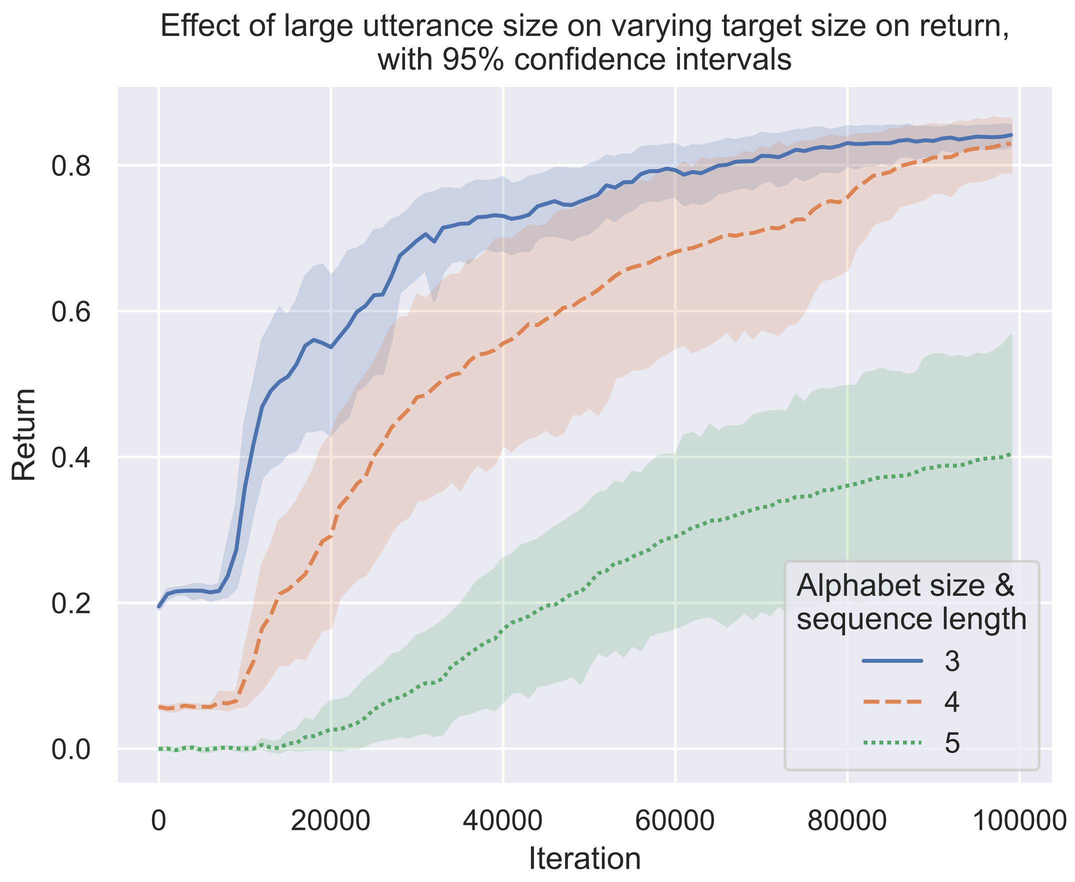

Figure 4.4: This graph shows the average return over 10 runs. The maximum number of turns is 3. The utterance alphabet size is 100, the utterance length is the same as the target alphabet length. The figure illustrates how the curse of dimensionality does not necessarily apply for the utterance policy. A large alphabet size makes it easier to create different correlations for different inputs, when some correlation is found it can be reinforced through RL. For a target alphabet size and sequence length of 6 the algorithm failed to produce any convergence.

4.5 Nash Equilibrium in Negotiation Without Reward Sharing

If some fraction of the rewards are not shared it does not seem reasonable for communication to arise. Assume a setting of negotiation with no reward sharing:

Assume agent A uses policy π_A that comes with some utterance providing some form of information about its hidden utilities, then it is highly likely there will exist some adversarial agent B using policy π_B that uses this information to its advantage. Since π_B acts partially upon information from the communication channel, it is highly likely there exists some adversarial policy π_C that gains from the utterances it provides to B. This illustrates how the use of a communication channel is inherently unstable in a strongly competitive setting. However there does exist a stable solution.

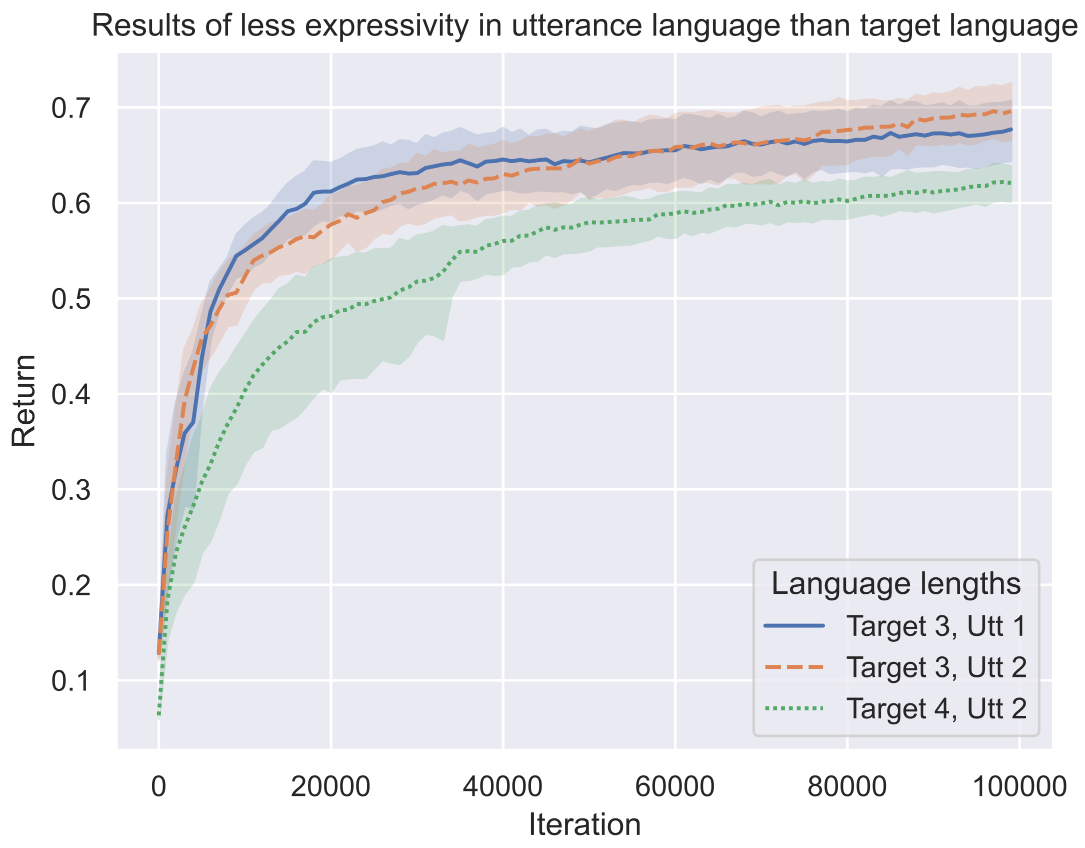

Figure 4.5: This graph shows the average return over 10 runs for different language lengths. The maximum number of turns is 4. The utterance alphabet size and target alphabet size is 3. The hidden state of the decoder persists across turns for both the guesser and the mastermind. Since the reward is -0.1 on the masterminds turn, with a target length of 3 and utterance length of 1 a return of 0.7 is to be expected with an optimal communication policy given incorrect initial guesses. In the two other cases however, a return of 0.8 is to be expected with an optimal communication policy given incorrect initial guesses.

Agent A can avoid the existence an adversarial policy π_B by using a policy providing no useful information in the communication channel. Agent B can avoid the existence of an adversarial policy π_C by using a policy that does not act upon information in the communication channel.

When the use of a communication channel will be inherently unstable, it seems reasonable to assume that through RL one will eventually converge towards more stable solutions that don't use the communication channel.

Recall the definition of a Nash equilibrium 2. Assume you have two communication policies π_D and π_E that both ignore inputs to the communication channel and don't output anything meaningful to the communication channel. These two policies will form a Nash equilibrium with respects to the communication channel as neither policy will increase its reward by deviating from the current communication policy. Note that this Nash equilibrium will exist regardless of the degree to which the rewards are shared. The main argument here is that it is easier to fall into such a Nash equilibrium if the rewards are not shared. In a fully cooperative setting the "adversarial" policies mentioned couldn't exist as they would only increase the joint reward and thus not be adversarial.

As has been shown in the MARL literature, there is a strong tendency in MARL to converge towards a Nash equilibrium rather than a Pareto-optimal solution [8]. This was also confirmed specifically for negotiation in a discrete setting by Cao et al. [2]. A fully cooperative setting naturally helps solve this issue, in a fully cooperative setting by definition a Pareto-optimal solution will also be a Nash equilibrium since all agents will receive the same reward. This argument and previous findings is the reason we are not showing results of experiments where there is no reward sharing in negotiation.

Chapter 5

Conclusions and Recommendations for Further Work

5.1 Summary and Conclusions

This work found that for cooperative settings meaningful communication can naturally arise between agents. It confirmed the findings of Eccles et al. [4] that positive signalling as a learning bias helps facilitate the emergence of communication. This was done by introducing a new environment where one player has to learn to communicate a target sequence to a guesser.

This work also introduced a new learning bias based upon the idea of positive signalling that facilitates the emergence of communication in a continuous setting. This was shown by adapting the work of Cao et al. [2] to a continuous setting.

It was found that in Sequence Guess, a higher alphabet size of the utterance language than the target language made it easier to converge towards a more optimal solution. It was also found that a high expressivity of the target language made it harder to converge towards a more optimal solution.

A theoretical argument was also made with regards to Negotiation, that some degree of reward sharing makes it easier for meaningful communication to arise.

5.2 Recommendations for Further Work

5.2.1 Short Term

It was observed that learning in the continuous case of Negotiation was not very stable. One could take steps to try improve the stability of the learning. This could for example be testing out weight decay [7], dropout [28], stochastic weight averaging [22] or layer normalization within the LSTM-unit [1].

One could also take steps to prevent the oscillations observed in the return for Negotiation when using positive signalling. One possible solution to this problem could be to separate the message policy from the remaining action policy when a certain reward threshold is reached, for example 0.99, and then freeze the parameters of the message policy while continuing training for some iterations. This would then remove the "moving" nature of the message policy as mentioned in section 4.1 since the message policy would be static, but it would still allow for improvement of the remaining joint policy with regards to the static joint message policy.

The new learning bias of positive signalling in a continuous setting should be explored further in order to find out more about its general effectiveness. One way to to this would be to measure the effectiveness of this bias in other continuous settings.

For Negotiation experiments could be run with different degrees of reward sharing in order to find out if there is some threshold value for the amount of reward sharing that causes the agents to converge on a stable communication policy, or maybe the the agents will oscillate between a working communication policy and no communication policy, and the degree of reward sharing will predict how long the agents will stay at each end of the oscillation.

5.2.2 Long Term

As stated in section 2.3.5. One can subdivide the communication protocols in MARL into two categories, communication protocols that are are updated as if they are actions in a RL setting, and the sharing of hidden states within networks. As has been done by Sukhbaatar et al. [29], Peng et al. [24], and Lin et al. [14]. A proper comparison between these two approaches to communication in MARL should be done. So that one can get a deeper understanding of what strengths and weaknesses of these two approaches have compared to each other.

For Negotiation, but also MARL in general, it would be interesting to see what dynamics arise when you use more than two agents in some sort of negotiation game. Especially with regards to a reward sharing parameter, will the agents form "alliances" and "bully" the other agents? One could also extend this to training populations of agents and then pit groups of agents against each other, is it possible for some "shared language" within a group of agents to arise if they communicate through action selection? Note that in order to extend things in this direction a very simple negotiation game is probably needed since the environment itself will still have a high degree of complexity when each agent's individual policy is taken into account.

In the case of Sequence Guess finding a protocol that can solve for target sequences of high expressivity and utterance sequences of arbitrary expressivity also remains an open question. The goal could be to find a general method of breaking the sequences down into smaller parts of a recurrent nature that adapts itself to arbitrary lengths and utterance sizes of the two alphabets, much like how words form sentences in human language.

References

[1] Ba, J. L., J. R. Kiros, and G. E. Hinton (2016). Layer normalization. *stat 1050*, 21.

[2] Cao, K., A. Lazaridou, M. Lanctot, J. Z. Leibo, K. Tuyls, and S. Clark (2018). Emergent communication through negotiation. *arXiv preprint arXiv:1804.03980*.

[3] Cho, K., B. van Merriënboer, D. Bahdanau, and Y. Bengio (2014). On the properties of neural machine translation: Encoder–decoder approaches. *Syntax, Semantics and Structure in Statistical Translation*, 103.

[4] Eccles, T., Y. Bachrach, G. Lever, A. Lazaridou, and T. Graepel (2019). Biases for emergent communication in multi-agent reinforcement learning. *Advances in neural information processing systems 32*.

[5] Foerster, J., G. Farquhar, T. Afouras, N. Nardelli, and S. Whiteson (2018). Counterfactual multi-agent policy gradients. In *Proceedings of the AAAI conference on artificial intelligence*, Volume 32.

[6] Fulda, N. and D. Ventura (2007). Predicting and preventing coordination problems in cooperative q-learning systems. In *IJCAI*, Volume 2007, pp. 780–785.

[7] Goodfellow, I., Y. Bengio, and A. Courville (2016). *Deep Learning*, pp. 82, 85, 282–283, 402–403, 290, 231. MIT Press. http://www.deeplearningbook.org.

[8] Gronauer, S. and K. Diepold (2022). Multi-agent deep reinforcement learning: a survey. *Artificial Intelligence Review 55*(2), 895–943.

[9] Hochreiter, S. and J. Schmidhuber (1997). Long short-term memory. *Neural computation 9*(8), 1735–1780.

[10] Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks 4*(2), 251–257.

[11] Jiang, J. and Z. Lu (2018). Learning attentional communication for multi-agent cooperation. *Advances in neural information processing systems 31*.

[12] Kingma, D. P. and J. Ba (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

[13] Leibo, J. Z., V. Zambaldi, M. Lanctot, J. Marecki, and T. Graepel (2017). Multi-agent reinforcement learning in sequential social dilemmas. In *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*, AAMAS '17, Richland, SC, pp. 464–473. International Foundation for Autonomous Agents and Multiagent Systems.

[14] Lin, T., J. Huh, C. Stauffer, S. N. Lim, and P. Isola (2021). Learning to ground multi-agent communication with autoencoders. *Advances in Neural Information Processing Systems 34*, 15230–15242.

[15] Lindenfors, P. (2017). *For Whose Benefit?: The Biological and Cultural Evolution of Human Cooperation*. Springer.

[16] Lowe, R., J. Foerster, Y.-L. Boureau, J. Pineau, and Y. Dauphin (2019). On the pitfalls of measuring emergent communication. *arXiv preprint arXiv:1903.05168*.

[17] Mcstrother (2010). Two layer ann. https://en.wikipedia.org/wiki/File:Two_layer_ann.svg. Seen: 05/08/2022. License: Creative Commons Attribution 3.0 Unported.

[18] MingxianLin (2018). Lstm. https://commons.wikimedia.org/wiki/File:LSTM.png. Seen: 08/08/2022. License: Creative Commons Attribution-Share Alike 4.0 International.

[19] Mvolz (2019a). Kawaii robot power clipart. https://commons.wikimedia.org/wiki/File:Kawaii_robot_power_clipart.svg. Seen: 05/08/2022. License: Creative Commons CC0 1.0 Universal Public Domain Dedication.

[20] Mvolz (2019b). Kawaii robot with heart clipart. https://commons.wikimedia.org/wiki/File:Kawaii_robot_power_clipart.svg. Seen: 05/08/2022. License: Creative Commons CC0 1.0 Universal Public Domain Dedication.

[21] Nguyen, D. T., A. Kumar, and H. C. Lau (2018). Credit assignment for collective multiagent rl with global rewards. *Advances in neural information processing systems 31*.

[22] Nikishin, E., P. Izmailov, B. Athiwaratkun, D. Podoprikhin, T. Garipov, P. Shvechikov, D. Vetrov, and A. G. Wilson (2018). Improving stability in deep reinforcement learning with weight averaging. In *Uncertainty in artificial intelligence workshop on uncertainty in Deep learning*.

[23] Olegalexandrov, Z. (2012). Gradient descent. https://commons.wikimedia.org/wiki/File:Gradient_descent.svg. Seen: 05/08/2022. License: Public Domain.

[24] Peng, P., Y. Wen, Y. Yang, Q. Yuan, Z. Tang, H. Long, and J. Wang (2017). Multiagent bidirectionally-coordinated nets: Emergence of human-level coordination in learning to play starcraft combat games. *arXiv preprint arXiv:1703.10069*.

[25] smart.servier.com (2016a). Milk. https://smart.servier.com/smart_image/milk/. Seen: 05/08/2022. License: Creative Commons Attribution 3.0 Unported.

[26] smart.servier.com (2016b). Orange juice. https://smart.servier.com/smart_image/orange-juice/. Seen: 05/08/2022. License: Creative Commons Attribution 3.0 Unported.

[27] smart.servier.com (2016c). Soda. https://smart.servier.com/smart_image/soda-2/. Seen: 05/08/2022. License: Creative Commons Attribution 3.0 Unported.

[28] Srivastava, N., G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research 15*(1), 1929–1958.

[29] Sukhbaatar, S., R. Fergus, et al. (2016). Learning multiagent communication with backpropagation. *Advances in neural information processing systems 29*.

[30] Sutskever, I., O. Vinyals, and Q. V. Le (2014). Sequence to sequence learning with neural networks. *CoRR abs/1409.3215*.

[31] Sutton, R. S. and A. G. Barto (2018). *Reinforcement Learning: An Introduction* (Second ed.)., pp. 324–332, 115. The MIT Press.