TÉCNICO LISBOA

Machine Learning techniques applied to the game Diplomacy

André Jardim Fernandes de Araújo

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisors: Prof. Pedro Alexandre Simões dos Santos
Prof. João Miguel de Sousa de Assis Dias

Examination Committee

Chairperson: Prof. Manuel Fernando Cabido Peres Lopes
Supervisor: Prof. Pedro Alexandre Simões dos Santos
Member of the Committee: Prof. Francisco António Chaves Saraiva de Melo

June 2023

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I'd like to thank my dissertation supervisors Prof. Pedro Santos and Prof. João Dias for helping me through this thesis with the weekly meetings, the fast responses and revisions and for pushing me to always learn more and go one step further right up until the end.

A special thanks to Philip Paquette, the creator of DipNet [Paquette et al., 2019], for sending me the human games dataset without which this work would not be possible and for helping me run Albert saving me countless debugging hours.

Agora em português para o meu pai conseguir ler. Obrigado aos meus pais por me apoiarem durante este trabalho, por estarem sempre lá quando as coisas corriam bem ou mal, por me ligarem a perguntar "Já acabaste" e pelo mais importante, o cartão refeição.

Obrigado à minha irmã por me aturar quando eu entrava no quarto dela a celebrar que estava qualquer coisa a funcionar e novamente para me queixar que afinal vi mal os resultados. E obrigado ao meu irmão que apesar de não o ver muito sei que aprecia muito o meu talento informático.

Obrigado aos meus amigos por me fazerem companhia quando precisava de uma pausa (ou muitas) da tese, por me contarem as histórias de terror da tese deles para eu não me sentir tão mal com a minha e por todo o apoio que me deram neste último ano e meio, honestamente não sei o que faria sem vocês. E claro um grande agradecimento à minha namorada por estar sempre lá para mim e por ser a minha maior fã, apesar de me distrair quando eu devia estar a trabalhar.

A todos vocês, demasiados para numerar, que me ajudaram durante este tempo - Obrigado.

To all of you - Thank you.

Abstract

Diplomacy is a 7-player game where agents compete for territories but need teamwork and eventual betrayal to get an edge over their opponents. This makes Diplomacy an excellent game to explore how Machine Learning (ML) can help an agent learn how to negotiate and how to take advantage of the results of that negotiation in its actions. In this thesis we developed two bots capable of playing Diplomacy, one that simply plays the game without communicating and another capable of negotiating at a basic level, making alliances, proposing peace and suggesting orders. For this purpose we used various ML techniques and architectures like Supervised Learning (SL), Transformers [Vaswani et al., 2017] and LSTMs [Graves and Graves, 2012].

Keywords

Cooperation; Competition; Diplomacy Game; Deep Learning; Supervised Learning.

Resumo

Diplomacy é um jogo de 7 jogadores onde agentes competem por territórios mas necessitam de cooperação, e eventualmente de traição, para superar os seus adversários. Isto faz do Diplomacy um jogo excelente para explorar como é que Machine Learning (ML) pode ajudar um agente a aprender como negociar e a tomar decisões baseadas nos resultados dessa negociação. Nesta tese desenvolvemos dois bots capazes de jogar o jogo Diplomacy, um que simplesmente joga o jogo sem comunicar e outro capaz de negociar a um nível básico, fazendo alianças, propondo paz e sugerindo ordens. Para este propósito usamos várias técnicas e arquiteturas de ML como Apredizagem Supervisionada (SL), Transformadores [Vaswani et al., 2017] e LSTMs.

Palavras Chave

Cooperação; Competição; Jogo Diplomacy; Aprendizagem Profunda; Aprendizagem Supervisionada.

Contents

List of Figures

List of Tables

Acronyms

SC	Supply Center
AI	Artificial Intelligence
DAIDE	Diplomacy Artificial Intelligence Development Environment
NN	Neural Network
RNN	Recurrent Neural Network
A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor Critic
MSE	Mean Squared Error
SL	Supervised Learning
RL	Reinforcement Learning
ML	Machine Learning
LSTM	Long Short-Term Memory
GCN	Graph Convolution Network
BRPI	Best Response Policy Iteration
PI	Policy Iteration
BR	Best Response
SBR	Sampled Best Response
IBR	Iterated Best Response
FPPI	Fictitious Play Policy Iteration
RM	Regret Matching
DORA	Double Oracle Reinforcement learning for Action exploration
DO	Double Oracle
NE	Nash Equilibrum

FAIR Meta Fundamental AI Research

1

Introduction

Contents

Cooperation and competition, as fundamental aspects of human society, have played a vital role in shaping our collective progress and achievements. Whether it is the collaborative efforts that drive scientific breakthroughs or the competitive spirit that makes nations outsmart one another, these forces have continuously pushed the boundaries of human potential. With the rise of Artificial Intelligence (AI) and Machine Learning (ML), it is no wonder that these principles of cooperation and competition would be explored by way of intelligent negotiation agents.

Diplomacy is a seven-player game that features both cooperation and competition, where through careful planning, teamwork and betrayals, a player can obtain an edge over its opponents. This provides a rich and complex environment for studying how artificial intelligence adapts to these kinds of social problems while still being contained and free from external variables. It is, however, more complex than similar games used for the testing of intelligent agents like Chess and Go, as each of the seven players controls multiple units with a large number of possible actions and all of these actions are played simultaneously.

The first attempts at creating Diplomacy bots have focused on rule-based agents, first only capable of playing the game without communication and then later appeared bots with limited messaging capabilities. Recently a wave of ML bots has emerged but again limited to the non-communication variant of the game. Our work focuses on studying how the recent techniques used by the ML agents can be applied to the communication aspect of the game by creating two bots, one with no messaging abilities and another capable of the limited negotiation of later rule-based bots.

1.1 Problem Statement and Goals

The goal of this thesis is to study the application of ML techniques, specifically intelligent agents, in an environment where negotiation and betrayal play a big part in the success of said agent. We wish to find what techniques would prove more useful in an environment like this and to what degree they would allow an agent to outclass handmade agents.

Previous attempts at this task were either rule-based agents [van Hal, 2013] or ML focused solely on the No Press variant of the game [Paquette et al., 2019, Anthony et al., 2020, Gray et al., 2020, Bakhtin et al., 2021]. Our goal with this work is to figure out how to build a ML agent that could learn how to communicate without sacrificing performance in the No Press variant of the game.

To explore this topic and analyze the performance of these techniques with and without communication, we developed two agents capable of playing the Diplomacy board game, both nicknamed Icarus. One of our agents, No Press Icarus, plays the non-communication (No Press) variant of Diplomacy, while the other, Press Icarus, plays the communication (Press) variant with some restrictions on the allowed messages to simplify the task. We gave special focus to the analysis of the Press agent as we intended

to introduce the first ML Press agent for Diplomacy.

To note that near the end of our work an agent was released that also explored this objective, Cicero [FAIR et al., 2022], but it was released too late to be considered in our development.

1.2 Organization of the Document

This document starts by describing the background needed to understand our work (Chapter 2), including the rules of Diplomacy (Section 2.1.1), the development environment we used (Section 2.1.2), the various models and algorithms used by our agents (Section 2.2) and the previous bots, both the rule-based bots (Section 2.1.3) and the ML state-of-the-art bots (Section 2.3).

After the background, we start by describing our agents (Chapter 3), followed by the details about how we trained our models (Chapter 4) and finally, the results obtained (Chapter 5). In the end we also include a summary of our work and the results obtained, as well as the conclusions derived from them (Chapter 6).

2

Background

Contents

This chapter explains some relevant information needed to understand our work. This includes the rules of Diplomacy, the environment used during development, some of the previous diplomacy bots that we used to train and test our agents and the most important techniques and algorithms we used.

2.1 Diplomacy

Diplomacy is a 7-player game where each player controls one of the seven “Great Powers of Europe” in the years before World War I. In this game the players must cooperate and clash to become master of the Continent.

2.1.1 Diplomacy’s rules

The seven European powers (Austria, England, France, Germany, Italy, Russia, and Turkey) fight over Supply Centers (SCs) in various provinces in Europe at the beginning of the 20th century. Besides the choice of the power a player controls there is no element of chance in the game.

There are a total of 34 SCs, scattered across 74 provinces which include land and water territories as seen in Figure 2.1. A player wins by owning 18 SCs. Alternatively the game can end earlier if all the remaining players agree to a draw. In scored games, for example in tournaments, the winning player wins 34 points for the 34 SCs in the map, while in a draw the points are either evenly split by all the remaining powers or split proportionally to the number of SCs a player controlled.

The way to interact with the game is by issuing orders to your army and fleet units. The game is split into years (starting in 1901) and each year has 5 phases: Spring Movement, Spring Retreat, Fall Movement, Fall Retreat, and Winter Adjustment. Before each movement phase there is a negotiation phase, usually with some time limit, where players can discuss alliances, plans, threats or anything else they wish.

There are four types of orders a player can place on each unit in a movement phase: Hold, Move, Support and Convoy, illustrated in Figure 2.2.

Hold: Hold is the default order. It consists of the unit standing its ground and defending the province it is occupying.

Move: The Move order means the unit will try to move into the desired adjacent province. If the province is defended the move is called an attack. Armies may only move through adjacent land provinces unless convoyed by a fleet. Fleets can move between water provinces, between coasts or between a coast and an adjacent water province.

A move is only successful if the attacking unit has more power than the defending or opposing powers. When an attack fails by having equal or less power than the opposition all units return to the province they moved from. If a unit is in the target province it stays in that province. When an attack

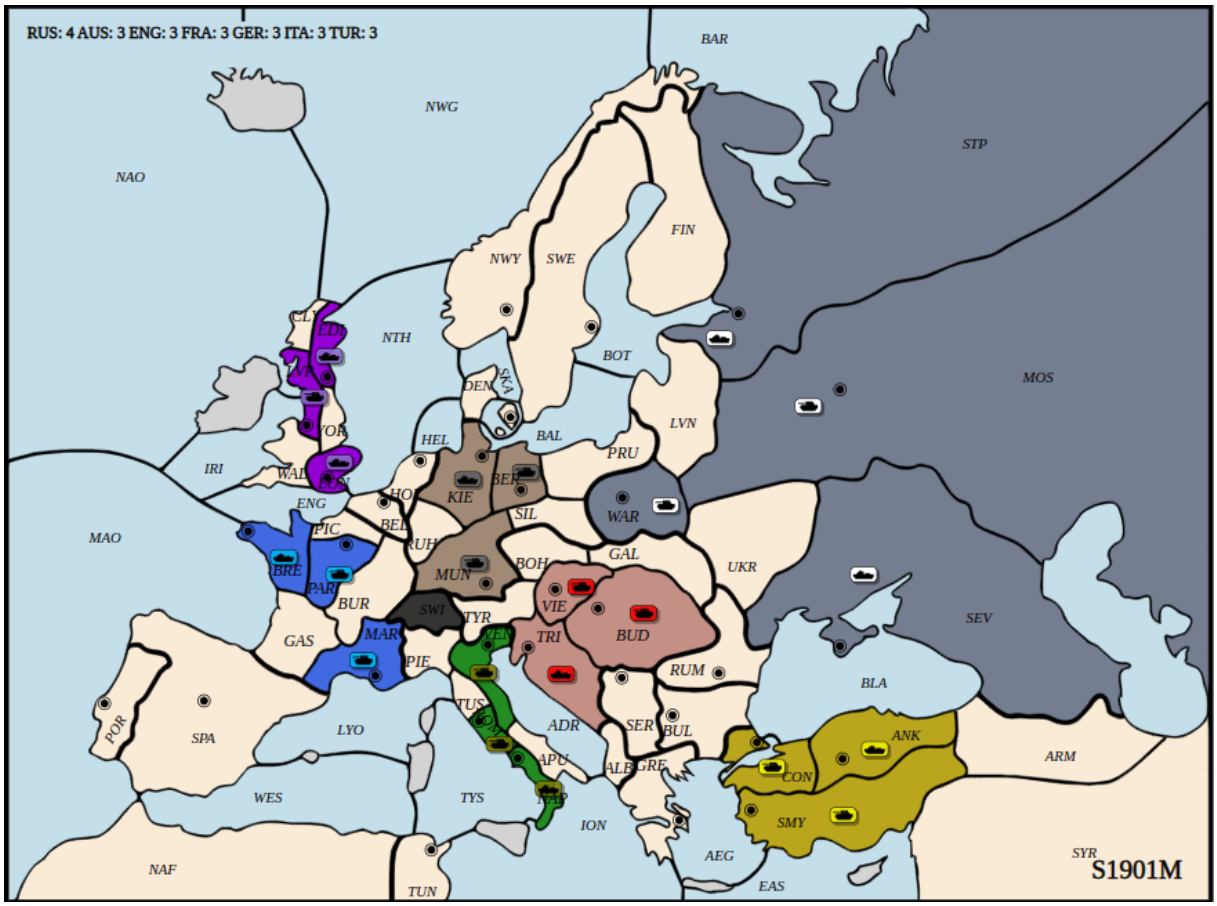


Figure 2.1: Starting map for a classic Diplomacy game. SCs are marked with a black dot. Image from the web interface of the Diplomacy game engine (Section 2.1.2.B).

succeeds, the attacking unit moves into the targeted province, and any unit in that province must retreat in the following retreat phase or be disbanded (removed from the board) if it has no available retreat provinces. A retreat works similarly to a move except it can not benefit from convoys and the target province has to be unoccupied and not a standoff location (a province left vacant by a failed attack).

Support: The Support order is the main cooperation method in the game. Units may support a move if they are adjacent to the target province (by indicating the origin and the target of the move). By doing this a unit adds one power to the move making it more likely for the attack to succeed. A unit can support not only units from the same power but also units from another power, giving rise to many cooperation opportunities. Armies can support adjacent land provinces and fleets can support both adjacent water and land provinces. If a support unit is attacked, the support is unsuccessful.

Convoy: Convoy orders can only be given to fleets. They allow a fleet to move an army through a water province from an adjacent land province to another. Convoys can also be chained by using multiple adjacent fleets. If a move fails, an army is sent back to its original province.

The adjustment phase consists of building or disbanding units to match the number of controlled Supply Centers (SCs). A power can only build a unit in one of the provinces it controlled at the beginning of the game, it must control it and it has to be unoccupied since there can only be one unit in each province. A player also has the option to waive some of its available builds if they so choose.

Communication in the normal game is done with natural language and depending on the variant it may allow private messages or only public messages and may allow communication throughout the game or only in negotiation phases taking place just before each movement phase. In No Press games some communication is still possible through the use of invalid orders or unit position, like a support from non-adjacent provinces to signal what provinces a power should invade or placing all your units on the border of a power's provinces indicating where you will attack.

For more details on the rules one can read the official rulebook [Calhamer, 2008].

2.1.2 Development Environment

This project was developed on a Diplomacy environment introduced in 2002 called Diplomacy Artificial Intelligence Development Environment (DAIDE) and to this day DAIDE is the standard for the development of Diplomacy AI. It was used for the development of many bots and some frameworks, namely it is the basis for all the bots mentioned in Section 2.1.3 and the Python Diplomacy game engine used in this project, described in Section 2.1.2.B.

In addition to these Diplomacy environments we used PyTorch [Paszke et al., 2019], a ML framework for Python, to develop our models.

2.1.2.A DAIDE

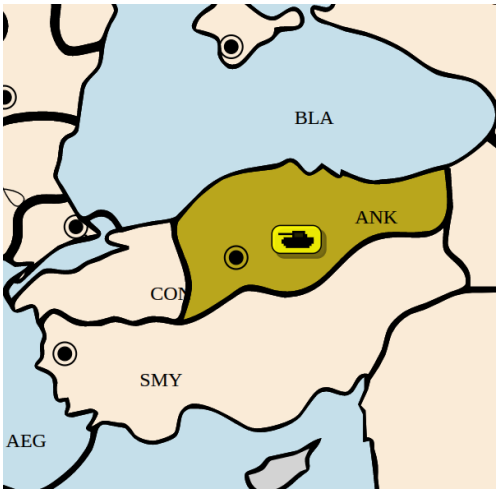
DAIDE [Norman, 2007a] is a research framework developed by the Diplomacy community that enables bots to compete against each other as well as against human players.

It includes a communication model and protocol, as well as a language in which diplomatic negotiations and instructions can be expressed. This environment allows the bots to play at different press levels, from No Press (Level 0) to Full Press as described in Table 2.1.

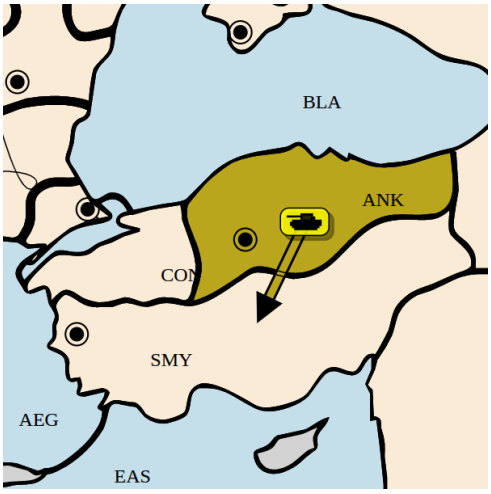
This project focuses on press levels 0 to 20, the messages allowed in those levels are explained in Appendix A. No matter the messages sent or agreements accepted, nothing is binding, a player can negotiate an alliance and immediately attack its supposed ally.

2.1.2.B Diplomacy - Python package

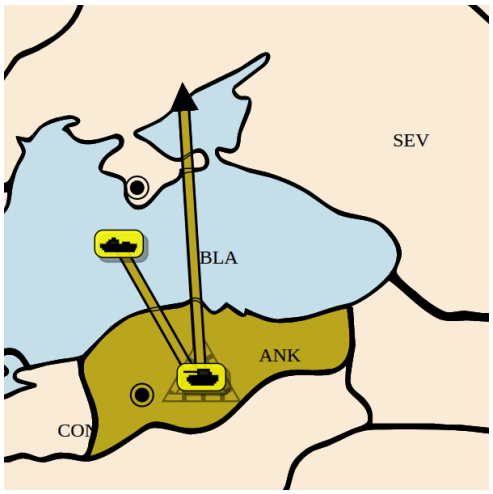
The Python package *Diplomacy* [Paquette, 2020] is an open-source Diplomacy game engine developed as part of the work in [Paquette et al., 2019], described in Section 2.3.1.



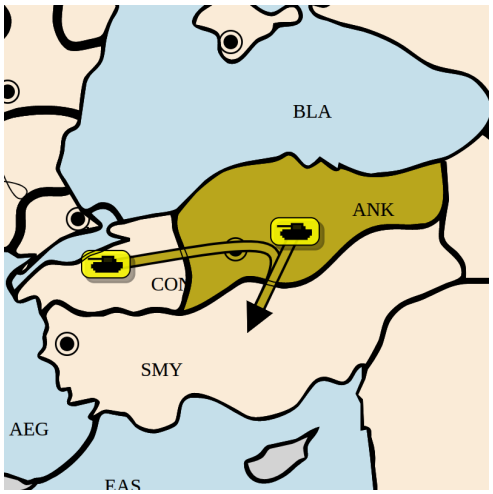
(a) This army is holding Ankara.



(b) This army is moving from Ankara to Smyrna.



(c) The fleet in the Black Sea is convoying an army moving from Ankara to Sevastopol.



(d) The army in Constantinople is supporting the move from Ankara to Smyrna. Even if Smyrna was defended by one unit the move would have been successful and the defending unit would have had to retreat.

Figure 2.2: Example of the possible moves in Diplomacy. Images from the web interface of the Diplomacy game engine (Section 2.1.2.B).

Level	Category
Level 0	**No Press**
Level 10	**Peace and Alliances**
Level 20	**Order proposals**
Level 30	Multipart Arrangements
Level 40	Sharing out the Supply Centers
Level 50	Nested Multipart Arrangements
Level 60	Queries and Insistences
Level 70	Requests for suggestions
Level 80	Accusations
Level 90	Future discussions
Level 100	Conditionals
Level 110	Puppets and Favours
Level 120	Forwarding Press
Level 130	Explanations
	Full Press

Table 2.1: DAIDE Press Levels. The relevant levels for our work are highlighted in bold.

We chose to use it since it presents a number of useful features that will facilitate the development of our agent. Those features are:

- it is compliant with DAT-C [Kruijswijk, 2004], a set of tests designed to determine if a Diplomacy framework follows all the rules correctly
- a Python server capable of handling multiple games
- a Python client allowing remote play with all the power and facilities of Python
- a web front-end with a human user-friendly interface
- a DAIDE server to play with DAIDE bots
- a webDiplomacy [Kuliukas and Zultar, 2004] API that allows play in this popular Diplomacy online platform

2.1.3 Rule-Based Bots

The first bots to be developed for Diplomacy were rule-based, meaning all the reasoning of the agents was directly implemented by the developer.

A lot of Diplomacy bots were developed over the years, some of the most famous can be found in [Norman, 2007b], but the ones we are interested in are the ones described in the following sections, namely RandBot and DumbBot, which were used as the minimum baseline for the No Press variant of Icarus, and Albert, the best rule-based bot, both in Press and No Press Diplomacy.

We will also talk about Tagus, the bot developed as part of the Thesis of Sancho Mascarenhas that is a precursor to this Thesis.

2.1.3.A RandBot

RandBot [Norman, 2007b] is a simple bot made by David Norman, who is also the creator of the DAIDE environment. It picks a random order for each unit from a subset of the possible orders. Given the simplicity of the bot, we developed our own RandBot using the Diplomacy Python package so the baseline games could be run locally without needing to connect to a DAIDE server.

2.1.3.B DumbBot

DumbBot [Norman, 2007b] is another simple bot made by David Norman that does not communicate. It calculates the value of each province based on if the location is a supply center and on the strength of the controlling power (if any) and then assigns orders for its units to attempt to move to the highest value adjacent provinces. This bot demonstrates a decent performance and is capable of beating many human players.

2.1.3.C Tagus

Tagus [de Mascarenhas, 2017] was built using DumbBot as a base with added logic to support Level 10 Press, Peace and Alliance Proposals. It calculates a Trust Factor and a Tension Factor between all players, based on cooperative and confrontational moves respectively and then uses those factors to choose who to send peace requests to and whether it should accept incoming peace requests. Tagus modifies the values DumbBot gives to each province by treating its allies' territories as its own. In addition it uses an Opening Library containing a list of some moves for each power to perform at the beginning of the game.

Unfortunately it was unable to outperform Albert so we did not use it as a benchmark.

2.1.3.D Albert

Albert [van Hal, 2013] is a rule-based bot made by Jason van Hal, last updated in March 2013, and was the state-of-the-art of Diplomacy bot in terms of communication prior to the release of Cicero (Section 2.3.6) in 2022.

It is capable of press up to Level 20 (Table 2.1). It can form peace agreements, alliances against other powers, and ask for demilitarised zones in specific provinces.

This bot was used to train the Press variant of Icarus via imitation learning and to measure the performance of both the Press and No Press variants of Icarus.

2.2 Technical Background

This work uses a variety of AI models and techniques, from simple feedforward Neural Networks (NNs) to more complex models like Transformers and techniques like Advantage Actor-Critic (A2C). This section aims to explain in some depth these models and techniques in order to make the inner workings of our agents clearer.

2.2.1 Neural Networks

NNs are ML models inspired by the way the human brain works in order to recognize underlying relationships in a set of data, essentially approximating a function that maps inputs to outputs. It does this by using a series of connected nodes called neurons. Each connection, also called an edge, can transmit information from one neuron to another, simulating the synapses in a biological brain.

Neurons are usually organized in layers that can perform different transformations on their inputs. In a typical feedforward network, signals travel from the first layer (input layer) to the last layer (output layer), usually going through a number of layers in-between (hidden layers). The incoming signals, represented as real numbers, are summed after being multiplied by their edge's weight and summed with a bias term, then they are transformed by a neuron using some non-linear function, usually called the activation function, before being sent to the nodes connected to the outgoing edges.

Some common activation functions include the sigmoid, softmax, tanh and ReLU functions.

The sigmoid activation function,

$$\text{sigmoid}(x) = \frac{1}{1 + e^{-x}}, \tag{2.1}$$

is used for models that predict probabilities since it normalizes values to be between 0 and 1.

The softmax function,

$$\text{softmax}(\mathbf{z})_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \quad \text{for } i = 1, \ldots, K, \tag{2.2}$$

is commonly used in the last layer of multiclass classification problems, since it takes as input a vector $\mathbf{z}$ of size K and returns a vector of the same size where all items are scaled to be between 0 and 1. These values can represent the probability of an input belonging to each one of the K classes, since unlike with the sigmoid function where the outputs are independent of each other, with the softmax function the outputs will sum up to one.

The tanh activation function,

$$\tanh(x) = \frac{e^{x} - e^{-x}}{e^{x} + e^{-x}}, \tag{2.3}$$

is used to normalize values between -1 an 1. This is useful for mapping strongly negative values to -1 and 0 values to near 0. It is a useful function for classification between two classes, one under zero and

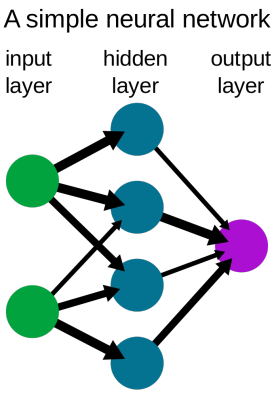


Figure 2.3: Simplified view of a feedforward artificial NN. Thicker arrows represent higher weights. Image from [User:Wiso, Public domain, via Wikimedia Commons, 2008].

another above.

ReLU (Rectified Linear Unit),

$$\mathrm{ReLU}(x) = \max(0, x), \tag{2.4}$$

is a very common activation function since it has been shown to be effective and computationally efficient. If the input of a neuron is less than zero the output will be zero, otherwise the value remains the same. It is typically used in the hidden layers of a NN.

The weights and biases of the edges and neurons determine how relevant a node or connection is to the modeling of the target function and are prone to change during the learning phase of the network. In order to learn the relevant weights, NNs use a variety of techniques, some of the most popular ones fall among the Supervised Learning (SL) and Reinforcement Learning (RL) paradigms, described below.

A common NN architecture is Recurrent Neural Networks (RNNs). These networks can process sequences of data instead of a single data point by having connections between network states. This works by having an information loop where part of the output of the network at one step is used as an input on the next step but one can also imagine this model as several copies of the same network passing information from one to the other as seen in Figure 2.4.

2.2.2 Long Short-Term Memory Network

A Long Short-Term Memory (LSTM) network [Graves and Graves, 2012] is a RNN architecture, this makes LSTMs suitable for sequence data, like text analysis, or time series data, like video analysis [Karpathy, 2015].

Normal RNNs have a problem with long-term dependencies, when a piece of information is further back in the data stream, the current input has difficulty accessing that data due to how activation functions transform data. In theory RNNs can learn how to use this far away data but in practice they seem

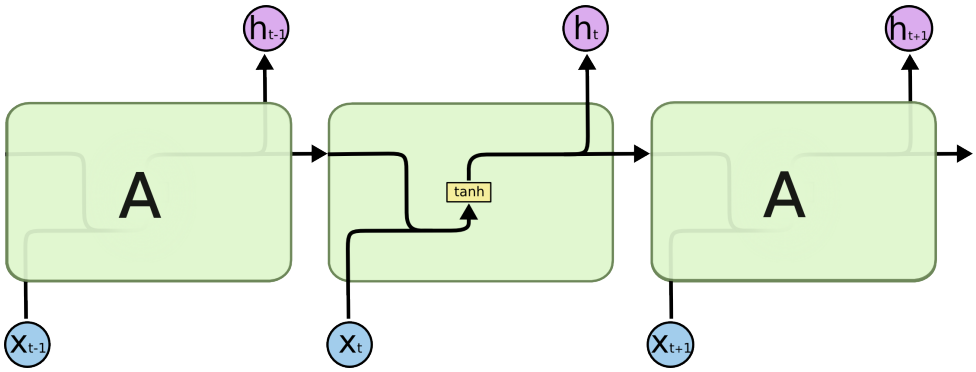


Figure 2.4: A simple RNN model using a tanh activation function. x_t and h_t are the inputs and outputs respectively at each time step t. Here h_{t-1} and x_t are concatenated before being fed to the tanh. Each module sends a hidden state to its successor. Image from [Olah, nd].

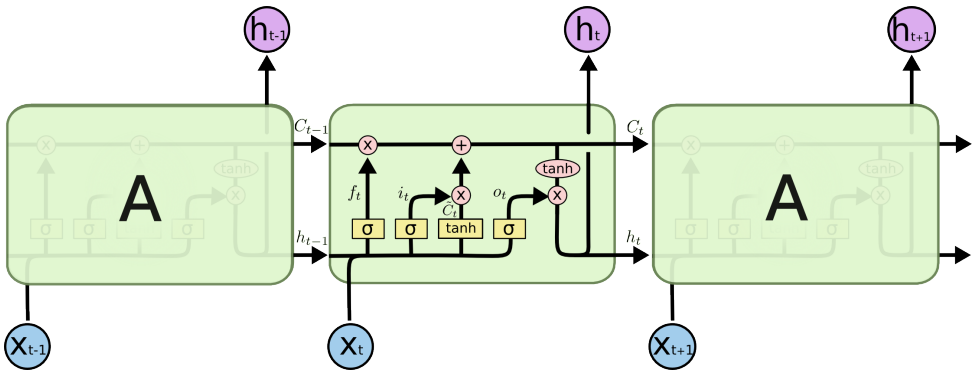


Figure 2.5: A LSTM with three gates, the forget gate, the input gate and the output gate. The yellow boxes represent a NN layer using the specified activation function and the pink circles a pointwise operation. The top line is the cell state that is altered by each gate. Image from [Olah, nd].

to have difficulty learning how to do so without careful problem-specific parameter tuning, which is undesirable. To solve this problem LSTMs were developed to have more control over what information to keep and what information to forget.

A common LSTM usually contains a cell, an input gate, an output gate and a forget gate. The cell remembers information for an arbitrary amount of time and the gates regulate the flow of information into and out of the cell as seen in Figure 2.5.

This LSTM receives a cell state C_{t-1} from the previous module. First the new input x_t and the output from the previous module h_{t-1} are processed by the forget gate f_t which consists of a NN layer with a sigmoid activation function, represented here by σ.

$$f_t = \sigma(W_f \cdot [h_{t-1}, x_t] + b_f) \tag{2.5}$$

Here W_f and b_f are respectively the weights and the bias for the forget gate. The forget gate will output a number between 0 and 1 which will be multiplied by the cell state C_{t+1}. The closer the number

is to 1 the more information from C_{t+1} is kept, with a 0 completing erasing all the previous information.

Next the same combination of x_t and h_{t-1} are passed to the input gate to update the cell state with the new information. A sigmoid layer i_t called the "input gate layer" decides what values to update and a tanh layer generates a vector of new candidate values $\tilde{C}_t$ that could be added to the cell state, these two combined will make the update to the cell state.

$$i_t = \sigma(W_i \cdot [h_{t-1}, x_t] + b_i) \tag{2.6}$$

$$\tilde{C}_t = \tanh(W_C \cdot [h_{t-1}, x_t] + b_C) \tag{2.7}$$

Now we can update the cell state C_{t-1} with the forget gate f_t and the input gate $i_t * \tilde{C}_t$:

$$C_t = f_t * C_{t-1} + i_t * \tilde{C}_t. \tag{2.8}$$

Finally we need to calculate the output h_t. First we pass the cell state through a tanh layer to push the values to be between -1 and 1 and multiply the result by the output of a sigmoid gate so we only output the parts we want to.

$$o_t = \sigma(W_o \cdot [h_{t-1}, x_t] + b_o) \tag{2.9}$$

$$h_t = o_t * \tanh(C_t) \tag{2.10}$$

LSTMs are commonly used in tasks like text recognition, for example with chatbots [Patil et al., 2020], where the input at each time step is an encoded word and so is the output, allowing the chatbots to communicate in natural language. The cell state, in conjunction with the various gates, allows for context to be maintained during the conversation, which is very important for correct sentence structure and meaningful conversations.

2.2.3 Transformers

Transformers [Vaswani et al., 2017] are a network architecture, commonly used in sequence-to-sequence tasks like translation. They outperform typical recurrent networks like LSTMs while allowing for parallelization, reducing training times.

These networks are composed of layers of encoders and decoders as shown in Figure 2.6. Each encoder layer has two sub-layers: a multi-head self-attention mechanism and a simple feedforward network. The decoder layers, in addition to having the same sub-layers as the encoder, add a third sub-layer that performs multi-head attention over the output of the encoder layers.

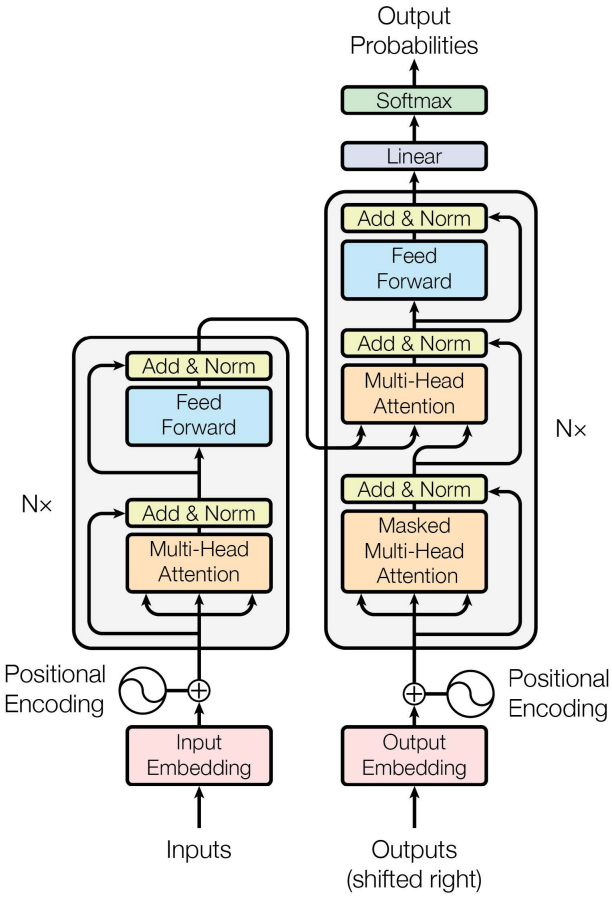


Figure 2.6: The Transformer model architecture. An encoder block on the left and a decoder block on the right. Image from [Vaswani et al., 2017].

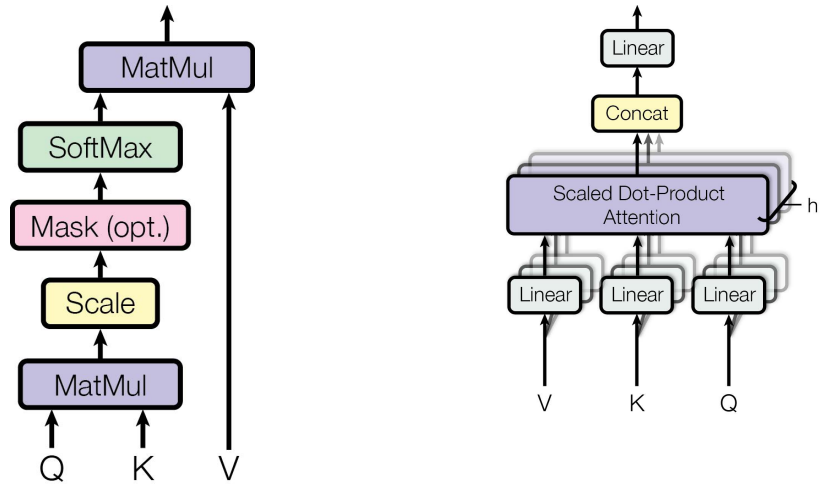
Attention is the process of mapping a query Q and a set of key-value pairs, represented respectively as K and V, to an output. The output is calculated using the inputted values, where the relevance of each value depends on how closely the corresponding key matches the query, calculated using

$$\text{Attention}(Q, K, V) = \text{softmax}(\frac{QK^T}{\sqrt{d_k}})V, \tag{2.11}$$

where d_k is the dimension of the keys and queries. This mechanism helps the network choose the information from the input that is relevant for the output, for example in translation tasks when trying to decide what word to output next, the network can look at the entire input phrase, while in simple RNNs it would only have access to the hidden state and the previous outputted word.

The model uses attention in three different places:

- **Encoder layers**: Here the attention is self-attention, as in the queries, keys and values all come from the previous encoder layers, enabling the learning of relations between the words of a sentence



(a) Scaled Dot-Product Attention

(b) Multi-Head Attention

Figure 2.7: Scaled Dot-Product Attention (left). Multi-Head Attention consists of several attention layers running in parallel (right). Images from [Vaswani et al., 2017].

- **First decoder sub-layer:** Again self-attention is used, connected to the previous decoder layers, combined with a mask that prevents the decoder from looking at tokens farther ahead in the target sequence
- **Second decoder sub-layer:** Here the decoder receives the key-value pairs from the encoder and the queries from the previous decoder layer, allowing all position of the decoder to attend over all positions in the input sequence.

Multi-head attention simply means that multiple attentions are calculated using different learned projections of the inputs before concatenating them into the output, allowing the inputs to be explored in different representation subspaces.

Since the model processes the inputs all at once, a positional encoding is also added to maintain the position information of the inputs, this can be a learned parameter, or in the case of word vectors some representation of the position of a word in a sentence.

2.2.4 Supervised Learning

Supervised Learning (SL) is a family of ML algorithms used for training NNs. It consists of using a set of training data composed of input-output pairs and trying to learn the function that maps those inputs to their respective outputs.

To do this, the inputs are sent through the network to obtain a prediction. The error between the network prediction and the expected output is calculated using some loss function like Cross-Entropy

Loss or Mean Squared Error (MSE) and based on the error value and some learning algorithm, like backpropagation, the weights of the neurons and edges are adjusted. The successive adjustments will cause the predictions to be increasingly similar to the desired outputs until the training is stopped by some predetermined criteria.

Cross-Entropy Loss is used in classification problems, where every input can be classified as belonging to one of M classes. In multiclass classification problems, where $M > 2$, cross-entropy is calculated using,

$$-\sum_{c=1}^{M} y_{o,c} \log(p_{o,c}) \tag{2.12}$$

where $y_{o,c}$ denotes if a certain input o belongs to a certain class c, which could be either 0 or 1, 1 if o belongs to c and 0 otherwise. $p_{o,c}$ is the network's predicted probability of class c given input o.

MSE is one of the most commonly used loss functions. It calculates the difference between each predicted label and the respective expected label and then squares it,

$$\text{MSE} = \frac{1}{m} \sum_{i=1}^{m} (y^i - \hat{y}^i)^2. \tag{2.13}$$

Here m is the number of data points being processed, y^i is the observed value for each data point and $\hat{y}^i$ is the expected value for each data point. Squaring the difference in the labels makes it so large errors are more significant and thus less prevalent in the final trained model.

2.2.5 Reinforcement Learning

Reinforcement Learning (RL) [Sutton and Barto, 2018] is a ML paradigm consisting of releasing an agent in an environment and having the agent learn in a trial-and-error manner, by interacting with the environment and observing the results of its actions. It does this by observing the state of the environment s_t at each time step t and choosing an action a_t based on an internal policy π. The policy is a map that gives the probability of taking each action based on a given state $\pi(a, s) = \Pr(a_t = a \mid s_t = s)$.

After performing an action the agent receives the next state of the environment s_{t+1} and a reward r_t. This process continues until the agent reaches a terminal state at which point the process restarts. The return,

$$R_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}. \tag{2.14}$$

is the cumulative reward at time step t with discount factor $\gamma \in [0, 1)$:

The discount factor raised to the distance in time steps to a future reward γ^k has the effect that immediate rewards are more valuable than rewards from a distant future. The goal of the agent is to maximize the expected return R_t for each state s_t.

The simplest RL methods learn the optimal action for each state by exploring the entire state space

which is impractical for all but the smallest environments. For larger environments it is necessary to use function approximators, normally this is done by using NNs in Deep RL, for example by modeling the policy function π with a NN using the state as the input and the action probabilities as output.

2.2.6 Q-Learning

Q-Learning is a model-free RL algorithm that learns the action value $Q(s, a)$ for each pair of state s and action a. Given infinite time and a partly-random exploration policy Q-Learning can learn the optimal policy π^* for an environment. The algorithm has a function $Q : S \times A \to \mathbb{R}$ that evaluates each state-action pair using a real number.

Q is initiated to static or random values (chosen by the programmer). At each time step t the agent selects an action a_t, receives a reward r_t, enters state s_{t+1} and Q is updated by the update rule

$$Q_{t+1}(s_t, a_t) \leftarrow (1 - \alpha)Q_t(s_t, a_t) + \alpha[r_t + \gamma \max_a Q_t(s_{t+1}, a)], \tag{2.15}$$

with α being the learning rate ($0 < \alpha < 1$). The learning rate α weights the old and the new values for Q by $(1-\alpha)$ and α respectively, meaning the higher the learning rate, the bigger the change to the Q-value each step.

Besides the learning rate, $Q_{t+1}(s_t, a_t)$ depends on three factors:

- $Q_t(s_t, a_t)$: the old Q-value.
- r_t: the reward obtained from taking action a_t in state s_t.
- $\gamma \max_a Q(s_{t+1}, a)$: the maximum return that can be obtained in the next state s_{t+1}. This means that a higher discount factor γ will value more long-term rewards over the immediate reward r_t.

An episode of the algorithm ends when reaching a terminal state at which point the environment can be reset to the initial state to continue learning. Learning stops after a predetermined number of episodes or iterations, or when the changes in Q-values are under a specified value.

With Q-Learning the goal is to learn the optimal policy $\pi^*(s) = \arg\max_a Q(s, a)$, for every state s.

Q-Learning excels in solving problems with a small state space where it can visit each state multiple times and experiment with all the actions in each state, however for large state spaces there is the need to use function approximators like NNs. The family of algorithms that uses NNs to enhance Q-Learning is known as Deep Q-Learning.

2.2.7 Advantage Actor-Critic

A2C is a RL method, more specifically a policy gradient method, which means it learns a policy function using the returns from the environment. A2C is based on the Asynchronous Advantage Actor Critic

(A3C) method [Mnih et al., 2016] and for our explanation we will use [Vitay, nd] as a reference.

A2C uses an actor-critic architecture (Figure 2.8) which has two agents, both modeled using NNs:

- an actor that outputs the policy $\pi_\theta(s, a)$ for state s, where θ are the parameters of the policy network,
- a critic that outputs the value V_φ for state s, where φ are the parameters of the value network.

The algorithm consists of applying the following steps:

1. Acquire a batch of T transitions (s, a, r, s') using the current policy π_θ.
2. For each state compute the discounted sum of the next n rewards $\sum_{k=0}^{n} \gamma^k r_{t+k+1}$, using the rewards for the following states in the batch, and use the critic to estimate the value of the state encountered n steps later $V_\varphi(s_{t+n+1})$.

$$R_t = \sum_{k=0}^{n} \gamma^k r_{t+k+1} + \gamma^n V_\varphi(s_{t+n+1}) \tag{2.16}$$

3. Update the actor using the expected advantage of each action. This is done using the gradient of the expected return $J(\theta) = \mathbb{E}[R_t]$, which is approximated using the expression

$$\nabla_\theta J(\theta) = \sum_{t \in T} \nabla_\theta \log \pi_\theta(s_t, a_t)(R_t - V_\varphi(s_t)). \tag{2.17}$$

4. Update the critic to minimize the error between the estimated value of a state and its true value. This loss is calculated with:

$$\mathcal{L}(\varphi) = \sum_{t \in T} (R_t - V_\varphi(s_t))^2. \tag{2.18}$$

5. Repeat.

2.3 State-of-the-Art

In this section we describe the previous Diplomacy ML agents that constitute the current state-of-the-art.

2.3.1 DipNet

DipNet [Paquette et al., 2019] is the pioneer of the new wave of Diplomacy bots, the biggest breakthrough since Albert (Section 2.1.3.D) stopped being updated in 2013. It was released in 2019, by Philip Paquette et al. and it introduces ML as a way to develop No Press Diplomacy bots.

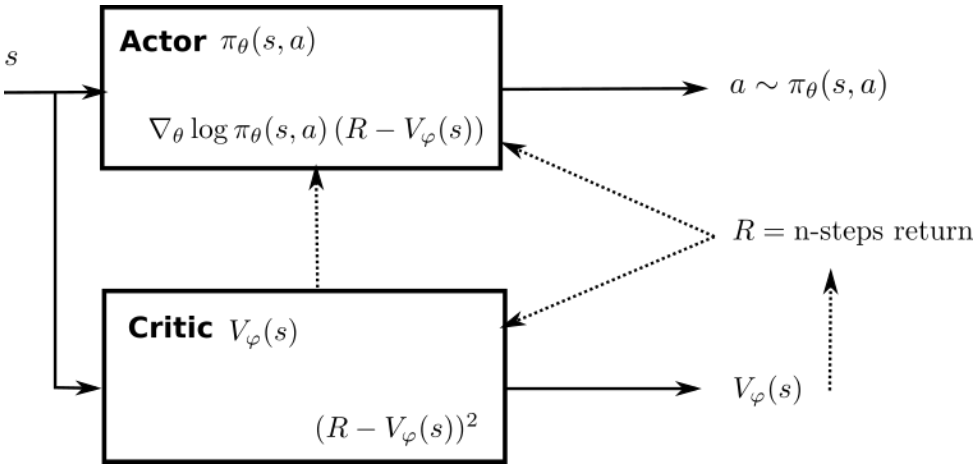


Figure 2.8: Advantage actor-critic architecture. Image from [Vitay, nd].

This bot uses SL to learn how to play Diplomacy by studying a database of human players and then a RL algorithm that improves upon the SL model through self-play.

First the board state and previous phase orders are encoded with relevant information using one-hot encoding, for example, if there is a unit in a province, what is the unit's controlling power etc. All the details of the encoding can be seen in Figure 2.9.

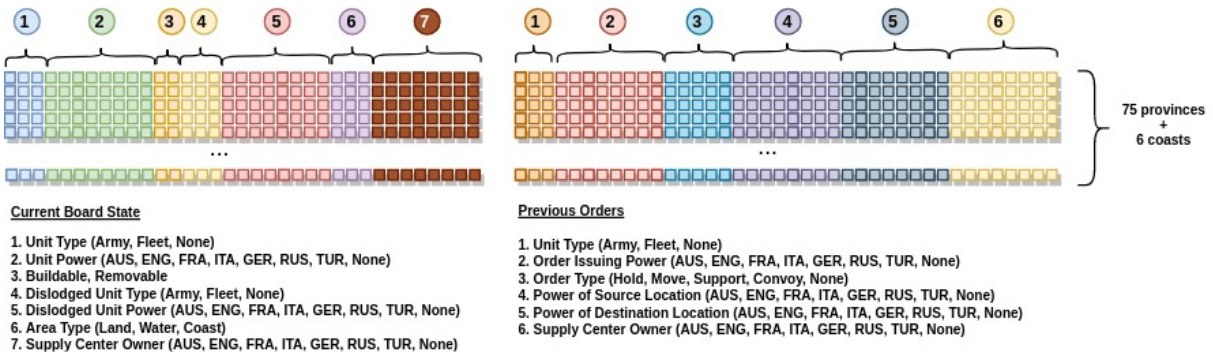


Figure 2.9: Encoding of the board state and previous orders in DipNet. Image from [Paquette et al., 2019].

Since Diplomacy has adjacency information (which provinces are connected to others), the DipNet model uses Graph Convolution Networks (GCNs) [Kipf and Welling, 2016] to take advantage of that information, that as the name indicates uses a graph structure to extract features for each node, in this case the provinces, based on the features of the surrounding nodes.

The input representations, encoded according to Figure 2.9, are x^0_{bo} and x^0_{po}, for the board state and the previous orders respectively. These inputs are then passed through L GCN layers, with each layer l having the board state $x^l_{bo} \in \mathbb{R}^{81 \times d^l_{bo}}$ and the previous orders $x^l_{po} \in \mathbb{R}^{81 \times d^l_{po}}$, with d^l_{bo} and d^l_{po} being the length of each encoding vector in layer l and 81 being the number of provinces (including coasts).

The board state and the previous orders are encoded by the same process so we will only describe

the board state encoding for simplicity. First the neighbor information is aggregated by:

$$y_{bo}^{l} = BatchNorm(Ax_{bo}^{l}W_{bo} + b_{bo}), \tag{2.19}$$

with $W_{bo} \in \mathbb{R}^{d_{bo}^{l} \times d_{bo}^{l+1}}$, $b_{bo} \in \mathbb{R}^{d_{bo}^{l+1}}$, $y_{bo}^{l} \in \mathbb{R}^{81 \times d_{bo}^{l+1}}$, A as the normalized map adjacency matrix of 81×81 and $BatchNorm$ being batch normalization [Ioffe and Szegedy, 2015], the process of scaling a subset (batch) of the input data so it has a standard variance and mean, which helps NNs train and converge faster.

Additional conditional batch normalization is performed using FiLM [Perez et al., 2018], a technique that adds learned values to the outputs of a NN layer based on certain inputs, in this case those learned values are conditioned on the player's power p and the current season s (Spring, Fall, Winter):

$$\theta_{bo}^{1}, \theta_{bo}^{2} = f_{bo}^{l}([p; s]) \qquad z_{bo}^{l} = y_{bo}^{l} \odot \theta_{bo}^{1} + \theta_{bo}^{2} \tag{2.20}$$

where f^{l} is a learned linear transformation, $\theta_{bo}^{1}, \theta_{bo}^{2} \in \mathbb{R}^{d_{bo}^{l+1}}$, and both addition and multiplication ($\odot$) are broadcast across provinces. Finally the activation function ReLU (explained in Section 2.2.1) and residual connections [He et al., 2016], connections that bypass one or more layers of the model, are used where possible:

$$x_{bo}^{l+1} = \begin{cases} \mathrm{ReLU}(z_{bo}^{l}) + x_{bo}^{l} & d_{bo}^{l} = d_{bo}^{l+1} \\ \mathrm{ReLU}(z_{bo}^{l}) & d_{bo}^{l} \neq d_{bo}^{l+1} \end{cases} \tag{2.21}$$

This process is repeated for each of the L layers of the GCN and there is no weight sharing. At the end, concatenation is performed to get $h_{enc} = [x_{bo}^{L}, x_{po}^{L}]$ where h_{enc}^{i} is the final embedding of the province with index i. DipNet uses $L = 16$. Thanks to the graph-based encoding, each province embedding has imbued knowledge about the surrounding provinces' state, which will be useful when decoding the provinces to choose an order.

The decoding is done sequentially to achieve coordination between units using a top-left to bottom-right order to process neighboring units together and to avoid jumps across the map.

Suppose i^{t} in the index of a province requiring an order at step t, the order o^{t} is chosen by a LSTM (Section 2.2.2):

$$h_{dec}^{t} = \mathrm{LSTM}(h_{dec}^{t-1}, [h_{enc}^{i^{t}}; o^{t-1}]), \tag{2.22}$$

where h_{dec}^{t-1}, $h_{enc}^{i^{t}}$ and o^{t-1}, the previous decoded embedding, the current encoded embedding and the previous obtained order, are used as inputs to the LSTM to get the current decoded embedding. This is followed by a mask that removes invalid orders for the target province:

$$o^{t} = \mathrm{MaskedSoftmax}(h_{dec}^{t}) \tag{2.23}$$

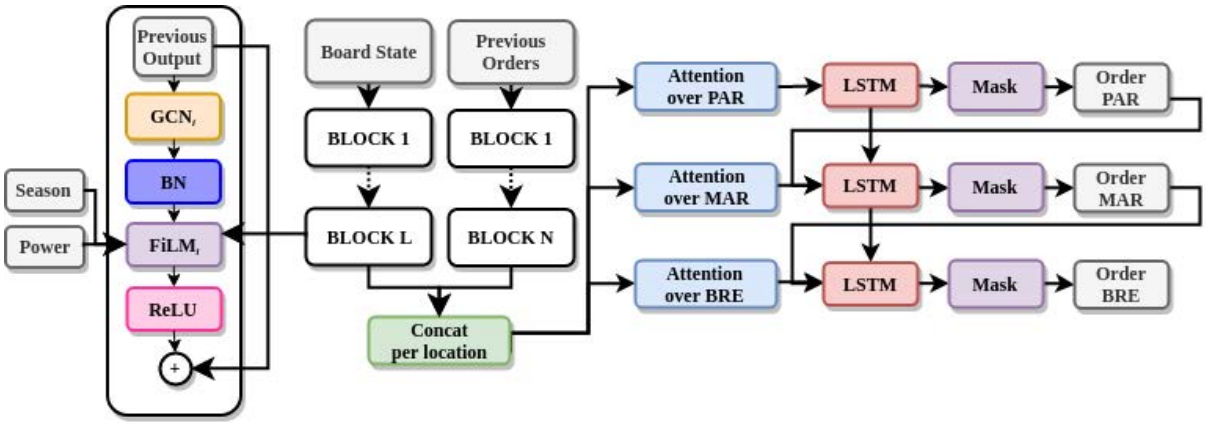


Figure 2.10: DipNet Architecture. Here PAR, MAR and BRE are provinces and attention over them means they require orders. Image from [Paquette et al., 2019].

The SL model is trained on a database of 156,468 human games. The RL agent uses the weights learned by the SL model and uses A2C (Section 2.2.7) to improve the agent by self-play, using the winning and losing of SCs as an intermediate reward of 1/-1, 34 as the reward for a solo win and, in case of a draw, the reward is proportional to the number of controlled SCs.

The paper concluded that while the RL agent slightly outperformed the SL agent, it was worse at cooperation, like executing successful cross-power support orders. Nonetheless both agents outperform all the previous rule-based bot, including the previous state-of-the-art bot, Albert (Section 2.1.3.D).

2.3.2 DeepMind's BRPI Agent

DeepMind's bot [Anthony et al., 2020], released in 2020, improves on DipNet's work described above, by slightly altering the SL method and using a Best Response Policy Iteration (BRPI) method for the RL bot instead of the A2C architecture.

BRPI, as the name indicates, uses Policy Iteration (PI) with Best Response (BR) calculations as an improvement operator. Given a policy π^b defined for all players, BR selects the optimal policy π_i^* for the agent i that maximizes the return against the opponent policies π_{-i}^b.

Since the Diplomacy environment is too large to calculate the exact best response this agent uses Sampled Best Response (SBR). SBR makes three approximations:

1. It considers a single-turn improvement to the policy in each state instead of using multiple turns.

2. It only considers a small set of actions selected by a candidate policy.

3. It uses Monte-Carlo estimates over the opponent actions for candidate evaluation.

To calculate the value of an action a_i against an opponent policy π^b_{-i}, SBR uses:

$$Q_i^{\pi^b}(a_i, s) = \mathbb{E}_{a_i \sim \pi^b_{-i}} V_i^{\pi^b}(T(s, (a_i, a_{-i}))), \tag{2.24}$$

with $T(s, a)$ as the transition function of the game, which gives the next state given the current state s and the actions taken by the player a and the opponents a_{-i}, and $V_i^{\pi^b}(s)$ as the state value for policy π. This agent however uses a trained value network $\hat{V}$ to get an estimated action-value $\hat{Q}_i^{\pi^b}(a_i|s)$. Additionally it only selects actions from a set of candidate actions A_i chosen by a policy $\pi_i^c(s)$, and the SBR policy will heavily depend on selecting good candidate actions.

The last improvement is performing Monte-Carlo sampling on the opponent actions. This means that the best response is calculated for a random sampling of opponent moves instead of the whole opponent action space.

Algorithm 2.1: Sampled Best Response (SBR)

Require: Policies π^b, π^c and value function v
function SBR(s:state, i:player)
  for $j \leftarrow 1$ **to** B **do**
    $b_j \sim \pi^b_{-i}(s)$ `//Sample Base Profile`
  for $j \leftarrow 1$ **to** C **do**
    $c_j \sim \pi_i^c(s)$ `//Candidate Action`
    $\hat{Q}(c_j) \leftarrow \frac{1}{B} \sum_{k=1}^{B} v(T(s, (c_j, b_k)))$
  return $\arg\max_{c \in \{c_j\}_{j=1}^{C}} \hat{Q}(c)$

Algorithm 2.2: Best Response Policy Iteration (BRPI)

Require: Best Response Operator BR
function BRPI($\pi_0(\theta), v_0(\theta)$)
  for $t \leftarrow 1$ **to** N **do**
    $\pi^{imp} \leftarrow BR(\{\pi_j\}_{j=0}^{t-1}, \{v_j\}_{j=0}^{t-1})$
    $D \leftarrow$ Sample-Trajectories(π^{imp})
    $\pi_i(\theta) \leftarrow$ Learn-Policy(D)
    $v_i(\theta) \leftarrow$ Learn-Value(D)
  return π_N, v_N

The presented BRPI algorithms all use approximate BRs such as SBR, meaning they depend on π^b, π^c and v (base policy, candidate policy and value function). There are three proposed BRPIs: Iterated Best Response (IBR) and two variations of Fictitious Play Policy Iteration (FPPI), FPPI-1 and FPPI-2.

In IBR, at every time-step t, SBR is applied to the latest policy $\hat{\pi}^{t-1}$ and value $\hat{V}^{t-1}$ to obtain a new policy π' (i.e. $SBR(\pi^c = \hat{\pi}^{t-1}, \pi^b = \hat{\pi}^{t-1}, v = \hat{V}^{t-1})$). Then, using the policy π', a dataset of trajectories is created using self-play, to which a new policy $\hat{\pi}^t$ and value $\hat{V}^t$ are fitted using SL, as is used to imitate human play. This algorithm however may result in undesirable outcomes like cycling strategies that don't converge or deterministic play.

FPPI focuses instead on calculating the best response to the distribution of the opponent strategies calculated in previous iterations. The first strategy, FPPI-1, aims to train the policy and value networks, $\hat{\pi}^t$ and $\hat{V}^t$, at each time step t to approximate the time-average of BRs (rather than the latest BR). For the SBR algorithm the time-average policy is used as the latest policy. To train the network on the average of BRs so far, at the beginning of each game we sample one of the previous iterations $d \in 0, 1, \ldots, t-1$ and use the network saved from that iteration (checkpoint) to play the game, effectively producing that iteration's BR policy.

In FPPI-2 the policy network is trained to learn only the latest BR, and the strategy so far is given by the average of the previous checkpoints. The opponent strategy is given by $\mu := \frac{1}{t}\sum_{d<t} \pi^d_{-i}$ after sampling $d < t$, and all the players' actions are sampled from the same checkpoint. Player i's strategy at time t should be the approximate BR for this strategy and the next policy network π^t should imitate that BR. In SBR this means $\pi^b = \mu^t$.

The network architecture for this bot is largely the same as DipNet's (Section 2.3.1), but the changes made to the RL agent made this bot consistently better than DipNet, unfortunately the bot is still not capable of human-level play as it is still exploitable by competent human players.

2.3.3 SearchBot

SearchBot [Gray et al., 2020], developed by Meta Fundamental AI Research (FAIR), at the time Facebook AI Research, and released in 2020, uses SL similar to DipNet's, with a few changes, including some from DeepMind's bot (Section 2.3.2), to construct a blueprint policy but instead of using RL to improve the bot's performance it relies on equilibrium search.

Specifically, at each turn the agent calculates the equilibrium (best possible outcome for all players) for a subgame consisting of a few movement phases (usually 2 or 3) starting from the current board position and considering only the highest probability actions according to the blueprint policy π obtained from the SL model. The player receives a reward according to the value of the final state, taken from the blueprint's value network

It calculates the equilibrium for each agent using a sampled form of Regret Matching (RM) [Blackwell, 1956, Hart and Mas-Colell, 2000], an algorithm that tries to minimize the regret associated with past decisions by adjusting the probabilities of choosing different actions based on their past performance. The regret of a decision is the difference between the reward obtained by playing that decision and the reward that would have been obtained by playing the best decision in hindsight. Additionally SearchBot makes some modifications to the RM to improve its performance.

SearchBot was the first bot to achieve human-level play in No Press Diplomacy, ranking on the top 2% of human players in the popular webDiplomacy [Kuliukas and Zultar, 2004] website, outperforming the previous agents Albert, DipNet and DeepMind's BRPI agent and even beating human experts attempting

to exploit the bot. However the agent is still exploitable by a trained best responder, a NN specifically trained to beat it, although still less exploitable than previous works.

2.3.4 DORA

Double Oracle Reinforcement learning for Action exploration (DORA) [Bakhtin et al., 2021], released in 2021, was developed by FAIR, still under the name of Facebook AI Research, in an attempt to learn how to play No Press Diplomacy by learning from scratch, that is, without using a learned model from human games as a baseline.

This model uses Nash Q-learning [Hu and Wellman, 2003, Littman, 1994], a multi-agent variation of Q-learning that, when calculating an agent's expected future reward, assumes that after making a decision all agents follow a joint Nash Equilibrum (NE) policy, a policy where no agent can improve their outcome by unilaterally changing their strategy, given the strategies of the other players.

The Nash Q-learning update rule changes the Q-learning equation (Equation (2.15)) to this:

$$Q(s, \boldsymbol{a}) \leftarrow (1-\alpha)Q(s, \boldsymbol{a}) + \alpha\Big(r(s, \boldsymbol{a}) + \gamma \sum_{\boldsymbol{a'}} \sigma(\boldsymbol{a'})Q(s', \boldsymbol{a'})\Big). \tag{2.25}$$

Here $Q(s, \boldsymbol{a})$ represents the current Q-value for state s with joint action $\boldsymbol{a} = (a_i, ..., a_{\mathcal{N}})$, where $\mathcal{N}$ is the number of players, and $\sigma(\boldsymbol{a}) := \prod_i \sigma_i(a_i)$ is the probability of joint action $\boldsymbol{a}$, here a NE σ meaning the probability of the join action following a NE.

DORA simplifies this algorithm by using the Value function instead of the Q-function due to the large action space of Diplomacy, along with other adaptations for large action spaces like using function approximation.

Since in Diplomacy the reward $r(s, a)$ only depends on the next state $s' = f(s, a)$ and the transition policy is known (given all the actions the next state is always the same), r can be redefined as a function of just the next state s'. Given this, Equation (2.25) can be rewritten in terms of the state value function:

$$V(s) \leftarrow (1-\alpha)V(s) + \alpha(r(s) + \gamma \sum_{\boldsymbol{a'}} \sigma(\boldsymbol{a'})V(f(s, \boldsymbol{a'}))). \tag{2.26}$$

Given the large state space in Diplomacy, a deep NN $V(s; \theta_v)$ with parameters θ_v is used to approximate $V(s)$. Similarly, due to the large action space of the game, computing a NE σ for a stage game, a game with only one step and the Q-value for the action played as a reward, is infeasible so a policy proposal network $\pi(s; \theta_\pi)$ with parameters θ_π approximates the distribution of actions under the NE policy at state s.

First a candidate action set at state s is generated by sampling N_b actions from $\pi(s; \theta_\pi)$ for each player and selecting the $N_c \ll N_b$ actions with highest likelihood. Then a NE σ is approximated using

Sampled RM like SearchBot (Section 2.3.3), including the same improvements.

Data is generated via self-play, alternating between the optimal action given by the NE policy and a random action from the N_c proposals. Also, to improve training stability, slightly lagged versions of networks θ_v and θ_π, $\hat{\theta}_v$ and $\hat{\theta}_\pi$, are used and are only updated every few time steps. Then the value network is regressed towards the stage game value computed under the NE σ using MSE loss, whose gradient step is the update rule in Equation (2.26), and the policy proposal network is regressed towards σ using cross-entropy loss:

$$\begin{aligned} \text{ValueLoss}(\theta_v) &= \frac{1}{2}\Big(V(s;\theta_v) - r(s) - \gamma \sum_{\boldsymbol{a'}} \sigma(\boldsymbol{a'}) V\big(f(s,\boldsymbol{a'});\hat{\theta}_v\big)\Big)^2 \\ \text{PolicyLoss}(\theta_\pi) &= -\sum_i \sum_{a_i \in A_i} \sigma_i(a) \log \pi_i(s, a_i; \theta_\pi) \end{aligned} \tag{2.27}$$

Since this algorithm relies on the policy network to generate good candidate actions, DORA uses an algorithm based on the Double Oracle (DO), to introduce novel actions that may have been initially assigned a low probability and so may have never been sampled and reinforced.

DO [McMahan et al., 2003] is an iterative method for finding a NE in games with a large set of actions. It uses the fact that it is computationally easier to calculate a best response than a NE. Starting from a small set of candidate actions $A_i^0 \subseteq A_i$ for each player i (chosen for example by a trained policy), it computes a NE σ^t in the game, restricted to the candidate actions A^t. Then at each time step t, for each player i, it finds the best response action $a_i^{t+1} \in A_i$ to the restricted NE σ^t and creates $A_i^{t+1} := A_i^t \cup \{a_i^{t+1}\}$. Formally, a_i^{t+1} is an action in the full action set A_i that satisfies

$$a_i^{t+1} = \arg\max_{a' \in A_i} \sum_{\boldsymbol{a} \in \boldsymbol{A}^t} Q_i(s, a_1, ..., a_{i-1}, a', a_{i+1}, ...a_{\mathcal{N}}) \prod_{j \neq i} \sigma_j^t(a_j). \tag{2.28}$$

DO is however modified to only compute the expected value of a pool of $N_p \gg N_c$ actions since the action space is so large. The pool of N_p actions to consider are chosen by joining N_d $(d \leq c)$ of the most probable actions in that equilibrium with the best response actions chosen from a random pool of actions generated for the units adjacent to a random location. Intuitively this works due to the importance of locality in Diplomacy.

Similarly to the other ML agents DORA uses an encoder-decoder architecture based on the DipNet model (Section 2.3.1), but introducing its own modifications, like using a transformer-encoder (Section 2.2.3) instead of a graph-convolution encoder.

DORA was trained in both 2-player and 7-player Diplomacy, but we're only interested in the 7-player results. It seems that when training from scratch, DORA overwhelmingly outperforms all the previous agents in 6v1 matches. However, when playing one DORA agent vs six SearchBot agents, DORA is outperformed. This indicates that the DORA model achieves a different equilibrium than the human-

trained model and that Diplomacy has more than one optimal equilibrium. In fact, different DORA runs appear to have the same effect of being very effective when paired with each other but outperformed when playing against six agents from a different run.

However when training the DORA network first with imitation learning, by using classification learning on a database of Diplomacy games, like the remaining ML agents, and only then training the model through self-play, DORA outperforms SearchBot, both 1v6 and 6v1, meaning that the DORA model was still the No Press state-of-the-art in the human-compatible equilibrium at the time it was released.

2.3.5 Diplodocus

Diplodocus [Bakhtin et al., 2022] was released by the FAIR team in October 2022, near the end of this work. Given the release date, it was not a direct influence on our agents but it still merits a brief description.

Diplodocus is a No Press agent that improves on DORA (Section 2.3.4) by replacing the equilibrium finding algorithm with a variant of piKL [Jacob et al., 2022], an algorithm that allows a model to learn without deviating from a baseline policy, in this case used so the self-play RL agents can learn while still being compatible with a human playstyle.

This bot played in a tournament including human players as well as Deepmind's BR bot (Section 2.3.2) and DORA (Section 2.3.4) and it scored first and third place (with different configurations), well above the other agents. This makes it so Diplodocus is probably the current state-of-the-art in No Press Diplomacy.

2.3.6 Cicero

The paper detailing the workings of Cicero [FAIR et al., 2022] was published in November 2022 by FAIR, along with the code for the bot [FAIR et al., 2023]. While it was released in the final stages of this thesis, and as such did not influence the methods used, it is still worth mentioning as it is a groundbreaking work for Diplomacy AI, specifically Press Diplomacy as it is the current state-of-the-art in that game variant.

Cicero is capable of Full Press, that is, it can communicate in plain English with the other players without sacrificing its ability to play the game as well as its predecessors.

This bot consists of a Dialogue Module and a Strategic Reasoning Module, complete with a filter for low-quality messages, as seen on Figure 2.11.

The Dialogue Module used R2C2 [Shuster et al., 2022], a language model trained on text from the Internet, and then complemented that training with dialogue from diplomacy games, taking into account the dialogue history, the game state and the player ratings (from webDiplomacy [Kuliukas and Zultar, 2004]) of the player who sent each message, as well as some information about the message, like

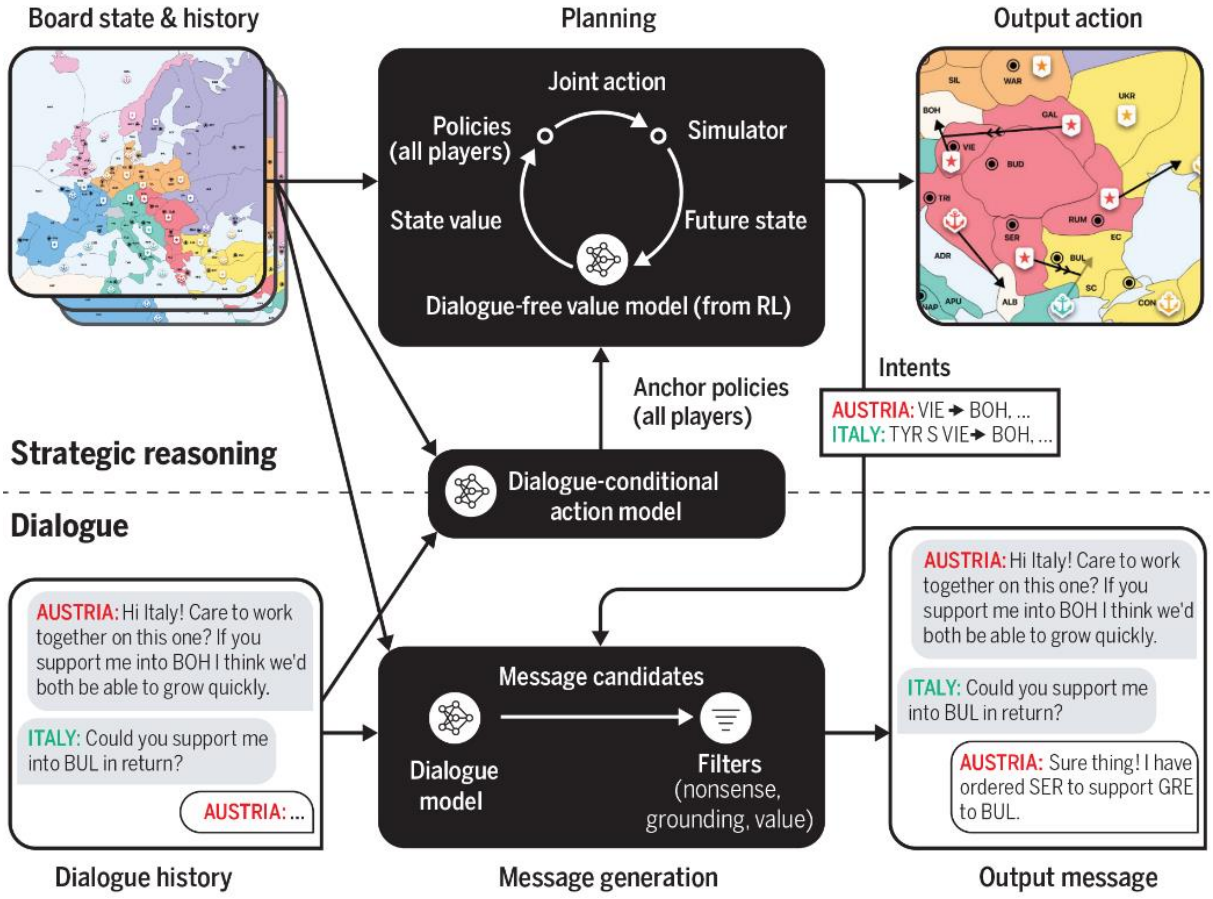


Figure 2.11: Arquitecture of Cicero. Image from [FAIR et al., 2022].

the time since the last message sent, but most importantly it uses intent. Intent consists of a pair of proposed orders for both of the players in a conversation that influences how the Dialogue Model predicts its messages. During training, this intent is derived from the conversations using a series of annotating strategies, but when running the model these intents are determined by the Strategic Reasoning Module that generates both an order that Cicero wants to play and an order that Cicero wants the other player to play, with the restriction that the opponent's order must be either beneficial for the other player or that it seems likely to be played based on past dialogue. Intents are also re-computed after every sent or received message to keep the results relevant. This intent mechanic is what allows the Dialogue Module to generate messages of a higher quality than the human messages it was trained on since it uses the orders generated by the Strategic Reasoning Module as a base, which have been proven by the No Press bots to be superior to the human orders.

The Strategic Reasoning Module predicts policies for all players based on the dialogue history and the current game state while taking into account both the strength of the move and the likelihood of it being played in human games. This policy is then used to calculate the intents for the Dialogue Module,

as well as the actual order to play.

Like Diplodocus (Section 2.3.5) this model was trained using self-play RL with a punishment for straying too much from a human imitation policy, to avoid reaching a non-human equilibrium like DORA (Section 2.3.4). Again this was done using a variant of piKL [Jacob et al., 2022], an algorithm that computes the best action based on all player policies while reducing the divergence from an anchor policy, in this case an imitation policy from human games, but now taking into account the dialogue history, as that can alter the state of the game.

Cicero was shown to have human-level performance, having played in a few tournaments against some expert players and obtaining very good results, being able to convince players to follow Cicero's proposed plans, as well as being almost indistinguishable from a human player when sending messages. It does however have some flaws. In its current state it is almost always honest, although this did prove largely successful so it may not be so much a flaw as a valid strategy. It also sometimes made language mistakes, although this did not raise suspicions that it was a bot, as the tournament games had a time limit and humans are prone to similar mistakes. Also, it makes strategies only relating to the current turn of the game, not taking into account the effects that its actions may have in future turns. Finally, it lacks a more advanced strategy sense, like revealing information gotten from other players, asking questions or explaining its actions.

2.3.7 Icarus

Our work, described in detail in the next sections, started development after the release of DORA (Section 2.3.4) at a time when no ML agent was attempting to learn how to play Press Diplomacy.

We introduce a model to send messages to other powers and to reply to received messages, as well as use the diplomatic state of the game to influence the orders chosen at each phase. We do this using reduced press, namely Level 20 Press according to the DAIDE Press levels Table 2.1.

At the end of Icarus' development however Cicero (Section 2.3.6) was released, so although our work is not the Press state-of-the-art it introduces a novel method to manage Diplomacy messages.

3

Icarus

Contents

We present here Icarus' structure, starting with the various encodings that translate the game into NN readable features and then describing the two models we trained.

3.1 Input Representation

There are two types of game related information that needs to be encoded to be used by our model, the game state and the messages. Here we describe how we chose to tackle this encoding. Our approach was heavily influenced by DipNet's input representation (Section 2.3.1), just like all the other agents that followed it, as well as using some improvements from DORA (Section 2.3.4).

3.1.1 Game State Encoding

The game state is characterized by two components, the board state and the previous orders.

To encode the board state we used the same format as DipNet (Section 2.3.1), represented in Figure 3.1, this means we used a matrix where each line is a vector that represents a location using one-hot encoding for each variable, making it so the board state is a binary matrix.

Each location contains information about the stationed unit's type (if any), the power that controls the location, if the location is suitable for building or removing a unit, if there is any dislodged unit and if so which power owns it, the type of location (Land, Water, Coast) and if it contains a supply center.

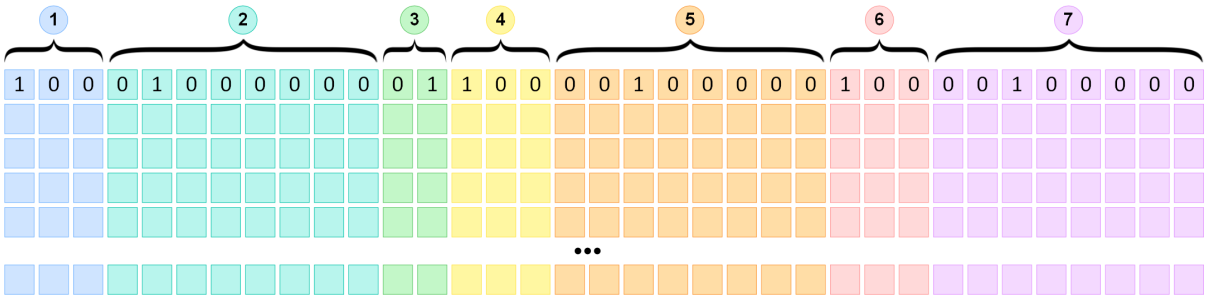


Figure 3.1: Board State Representation

Similarly the previous orders are represented by a matrix where each line represents a location and contains the encoding for the order played on that location last phase, however we decided to stray away from DipNet's one-hot encoding format for the orders and instead used a vector with 8 integer values for

each order, shown in Figure 3.2. We chose to use a different format because DipNet doesn't specify the locations involved in an action, only mentioning some factors like the controlling power of the location and relying on the province adjacency matrix and a GCN to deduce the provinces involved. We instead opted to use the more versatile transformers as explained below (Figure 3.3) so we decided to specify the locations on our embeddings.

Given this decision we strayed away from one-hot encoding and chose to use a vector of integers to represent all our variables. Specifically for the locations we used a constant list of provinces, like the one used by DipNet when decoding orders, that lists locations from the top-left to the bottom-right of the map and used the index of each province in that list plus one as its representation, with zero indicating an unused location variable.

The values that identify an order are:

Unit Type: This can be either an army or a fleet, 0 or 1.

Order Type: The possible order types are hold (1), move (2), support hold (3), support move (4), convoy (5), convoy to (6), retreat (7), disband (8), build army (9), build fleet (10), remove unit (11). These order types are more specific than the default actions described in Section 2.1.1 for two reasons, the orders have a different format depending on the context, for example a move to an adjacent province is the same order as a move using convoys, but the latter requires specifying what seas the unit is convoying through, and also the more specific actions can give a little more context to the model which may help it learn faster.

Order Location: This is the location where the order was issued. It is represented by the index of the location in the province vector explained above.

Target 'Location: This is used for the orders that affect a second location like moves, supports, convoys and retreats. It is also represented by its index in the province vector.

Extra Locations 1-4: These extra locations are used for different purposes depending on the action. The first extra location is used for the support move and the convoy orders, to specify the location the supported or convoyed unit is moving to.

The "convoy to" order however can potentially use all 4 extra locations to specify the seas it is moving through, this does limit our convoy orders to the ones affecting up to 4 seas, but we observed that orders with more than 4 convoying fleets are extremely rare so this is not a big limitation.

Each of the locations is represented by its index in the province vector.

After getting more familiar with the PyTorch framework [Paszke et al., 2019] we realized that there was a more effective way to represent the previous orders. That would be to use PyTorch's Embedding module to represent each order instead of manually calculating each embedding with the above method as we did for the messages described below (Section 3.1.2). This module takes a number of values, in this case the 13 244 unique orders (when ignoring convoys that are only generated during the game),

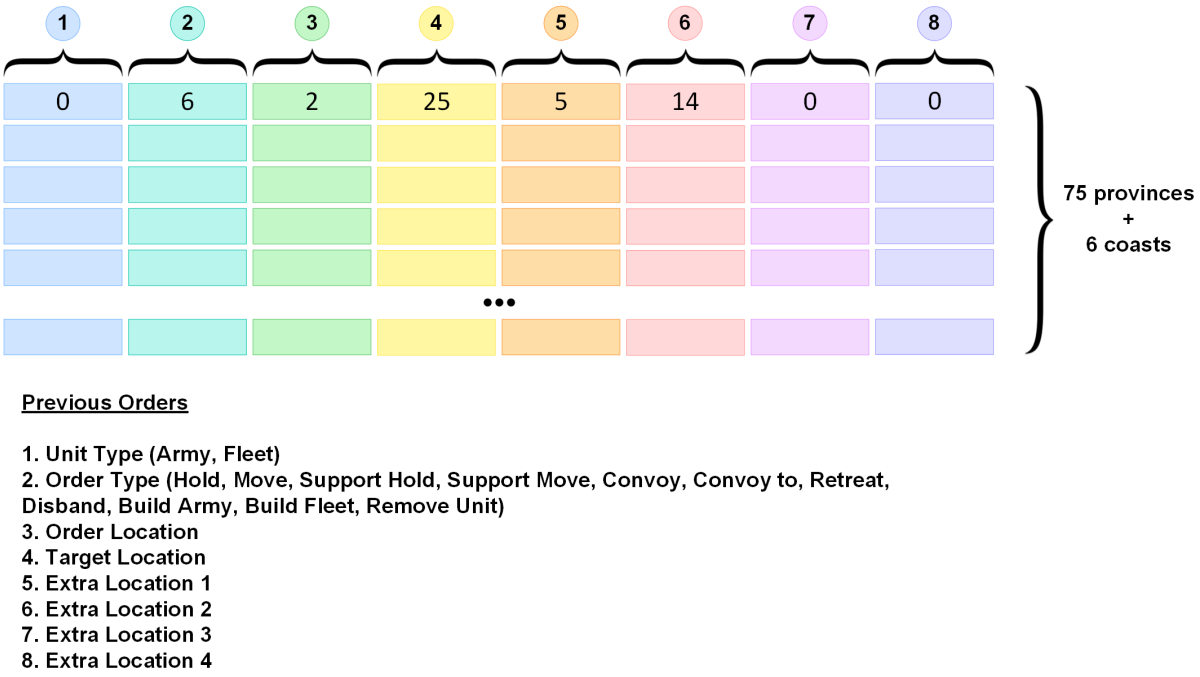


Figure 3.2: Previous Order Representation

and learns an embedding for each of them, which means the embeddings could have helped improve the network's performance. This would simplify the code as well as possibly improve performance, but by the time we realized this our model was already trained using the encoding described above and we didn't have enough time to retrain it, so this improvement can be left for a future work.

This combination of board state and previous orders is a good representation of the current game state, however by using these manual embeddings as is, each location would only have information about itself and not about the surrounding locations. DipNet (Section 2.3.1) solved this problem using GCNs [Kipf and Welling, 2016] but we decided instead to use DORA's approach (Section 2.3.4) of using transformers (Section 2.2.3), or more specifically the encoder segment of a transformer model. This transformer-encoder block, along with a linear layer to shape the inputs and a PyTorch Parameter used as a learnable positional encoding which is added to the linear layer's output makes up our Encoder Module, shown in Figure 3.3. For the transformer-encoder we experimented using a different number of layers, 3, 5 and 10 specifically and found that 5 layers considerably sped up training time without sacrificing too much performance. This model outputs a matrix of 81 locations where each location is represented by a vector of *embed_size*, that we set at 224, that in theory contains information about the location, as well as the relevant information from the surrounding provinces.

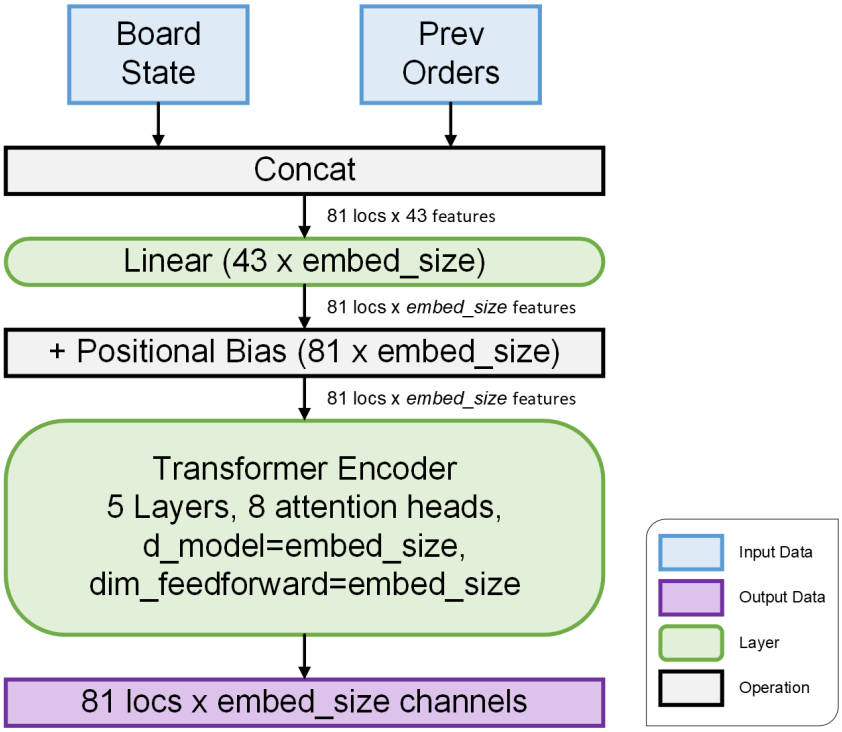


Figure 3.3: Architecture of the Encoder Module

3.1.2 Message Encoding

Icarus is capable of press up to Level 20 (Table 2.1) so it knows how to send PCE, ALY, XDO and DMZ messages, described in detail in Appendix A.

To encode the messages we started by generating every legal message. We defined PCE and ALY messages using all the possible combinations of powers and made one XDO for each legal order.

For the DMZ messages the product of all permutations of powers and locations would lead to an enormous number of possible messages, greatly increasing the size of our network. Given this we chose to only generate one DMZ for each location and up to two powers. To make it so that our bot would still be able to participate in games with players using the normal format, we split the DMZ messages during play if they contain more than one location. It is still only able to send and receive DMZ messages with up to two powers but we observed that Albert (Section 2.1.3.D) is also only capable of mentioning two powers so we believe that for now it is enough for the model to learn how to communicate and we leave the ability to generate the full scope of DMZ messages to a future work.

Given this list of messages, which contains 16 953 messages, we then used PyTorch's [Paszke et al., 2019] Embedding module to generate learned embeddings. This module takes the number of embeddings needed, in this case the number of messages plus the number of replies (YES, REJ, no reply) for each message bringing it to a total of $16\,953 + 16\,953 * 3 = 67\,812$ embeddings, and generates

a lookup table that attributes each of those values to an embedding of a specific size, that we set at *msg_embed_size*, 100 by default. That relation from a message to an embedding is trained along with the model in order to help the learning of the network. We also combine the embeddings into a message log that keeps only the ones relating to the latest message as shown in Figure 3.4. This message log is then used as input for the Press Model described below in Section 3.2.2.

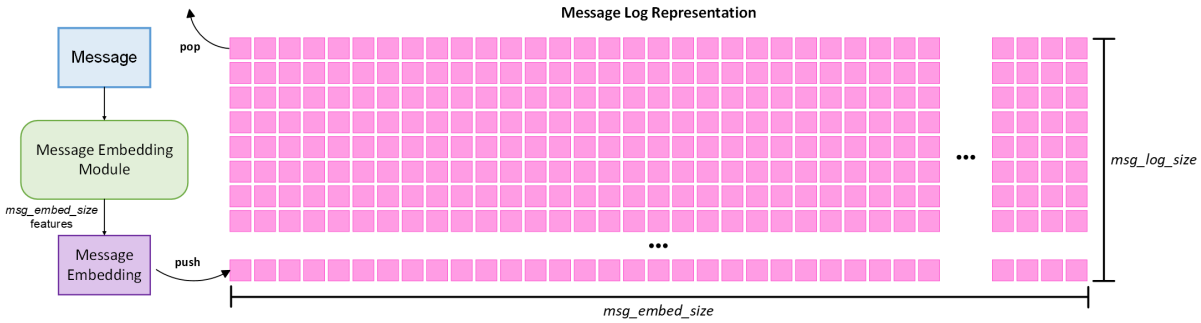


Figure 3.4: Message Encoding and Message Log Representation

3.2 Model Architecture

Given the encoded inputs we can now describe the models that process those inputs, the game state and the messages, and return the orders to play and the messages to send.

3.2.1 No Press Architecture

We started by creating a simple model capable of reading the game state and returning the appropriate order to play, and as such is not capable of sending or receiving messages. The general view of the model is shown in Figure 3.5.

The No Press Icarus model contains an Encoder Module, described in Section 3.1.1, a LSTM with *lstm_size* features in the hidden layer, default 200, for decoding the encodings into orders, similarly to DipNet (Section 2.3.1), and three linear layers. One of these layers is used to generate the policy vector by transforming the output of the LSTM into a vector of the same size as the number of orders, 13244, followed by a softmax activation function (Section 2.2.1) obtaining the probability of picking each order.

The other two linear layers are used to determine the value of the game state for each player. It starts by flattening the encodings of all locations into a single vector, then we feed this vector to the first linear layer, transforming it into a vector of size *embed_size* and apply a ReLU activation function (Section 2.2.1), then we use the second linear layer to generate a vector of size 7, having a value for each power, and finally, after using a softmax activation function, we get the expected probability

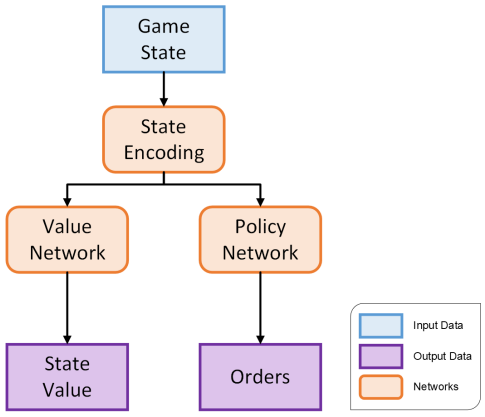


Figure 3.5: General view of the No Press Model

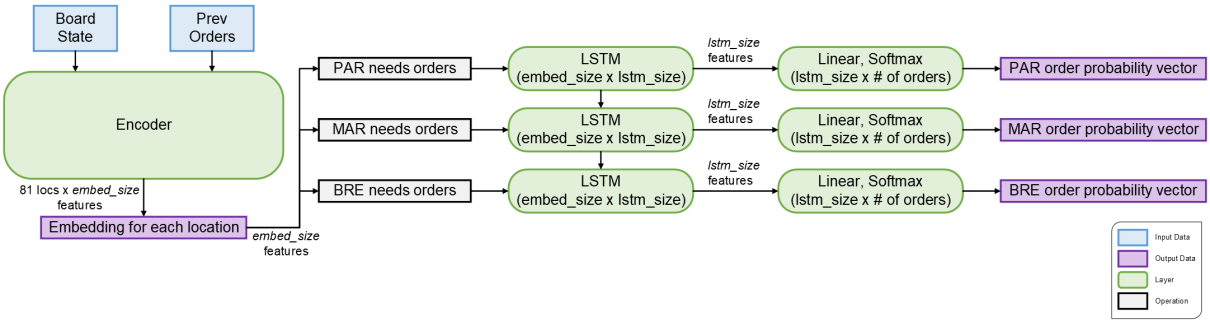


Figure 3.6: Architecture of the No Press Policy Network

of each power winning the game given the inputted game state. This value was originally for use in the A2C algorithm as the critic's value, as described in Section 4.1, but it can still be used to directly observe the model's predicted outcome of the game. It also affects the learning of the network, the value network learns by comparing the network's predicted outcome of a game to the real outcome and its loss is propagated through the network, affecting the Encoder's weights, which in turn affects the policy network's output.

The policy network for this model, shown in Figure 3.6, receives the board state and the previous orders, encoded as described in Section 3.1.1 and feeds it into the Encoder, which gives us the embeddings for each location. The agent then selects the locations that need orders based on the power it controls and passes the encodings for those particular locations in order through the LSTM, based on a list of provinces, also used on DipNet, that lists locations from the top-left to the bottom-right of the map, to ensure locations base their orders on nearby ally units' orders, and then feeds them to the final linear layer, applies a softmax activation function, and uses a filter to remove illegal orders, obtaining the probabilities of playing each order. Similarly for the value network (Figure 3.7), the encoding for all locations is passed through the two value linear layers, with a ReLU function in between and a softmax function at the end to obtain a vector with the probability of each power winning the game from that state.

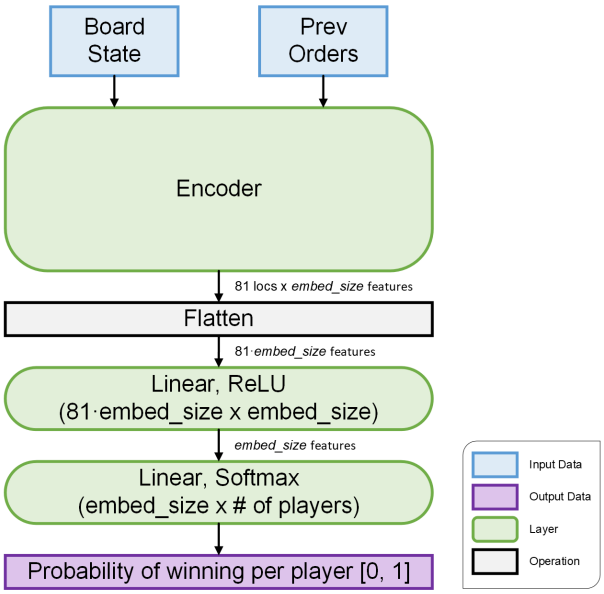


Figure 3.7: Architecture of the Value Network

3.2.2 Press Architecture

Icarus' full model, capable of press up to Level 20, is very similar to the No Press Model described above. The general view of the model is shown in Figure 3.8.

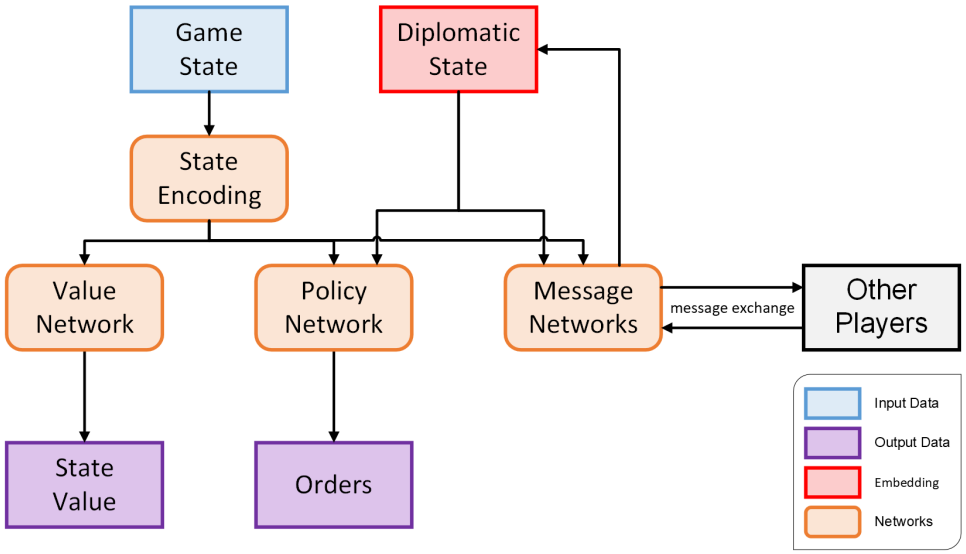


Figure 3.8: General view of the Press Model

This agent adds the message embedding module and the message log described in Section 3.1.2, three linear layers to process the messages and adds to the inputs of the policy network the diplomatic state, an embedding based on the message log. The policy network remains largely the same, adding

only the diplomatic state as an input to the LSTM, as shown in Figure 3.9. The value network remains the same as the No Press Model's shown in Figure 3.7, this does however mean that the value network does not depend on the exchanged messages which can affect the game state, we chose this simplification to avoid having to retrain the value network for the press model but this is a possible addition for a future work.

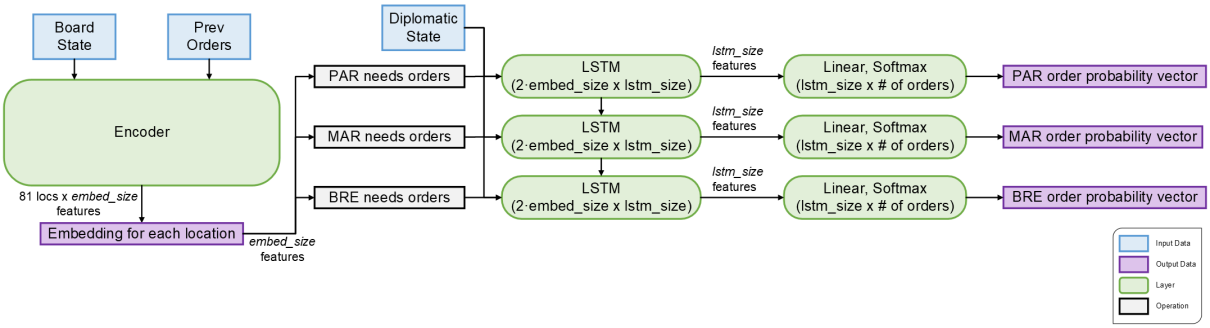


Figure 3.9: Architecture of the Press Policy Network

The main difference between the Press and No Press models comes of course from the messages, which are stored in the message log, composed of *msg_log_size* messages, 20 by default, each with an embedding of size *msg_embed_size*. The log is initialized with all zeros and every time a message is sent or received, its embedding is added to the log and the oldest message is removed if the log is full as shown previously on Figure 3.4 .

There are three situations where the embedding of the message log is used, as an input to the policy network when calculating what orders to play, when replying to a received message and when choosing what new messages to send, like at the beginning of the phase. The three new linear layers in this model address these situations, as seen in Figure 3.10, the first transforms the message log into an embedding with a length of *embed_size*, which after using a ReLU activation function (Section 2.2.1) represents the diplomatic state between the relevant power and the remaining players, and can then be used in the policy network for determining orders, or as input to the remaining linear layers.

The two other linear layers are used for sending and replying to messages. To generate the messages to send, the model takes the diplomatic state, joins it to the encoded game state and passes that through the Outgoing Messages linear layer which gives a vector with a value for each of the 16 953 possible messages, which does not include the replies. By then applying a sigmoid activation function (Section 2.2.1) we get the independent probability of sending each message. After filtering illegal messages we chose the messages with probability above a specific threshold to be sent to the respective targets, and after some testing we found that 20% was a good value for that threshold, allowing some but not all messages to go through.

The reply process is very similar, the diplomatic state is concatenated to the game state encoding,

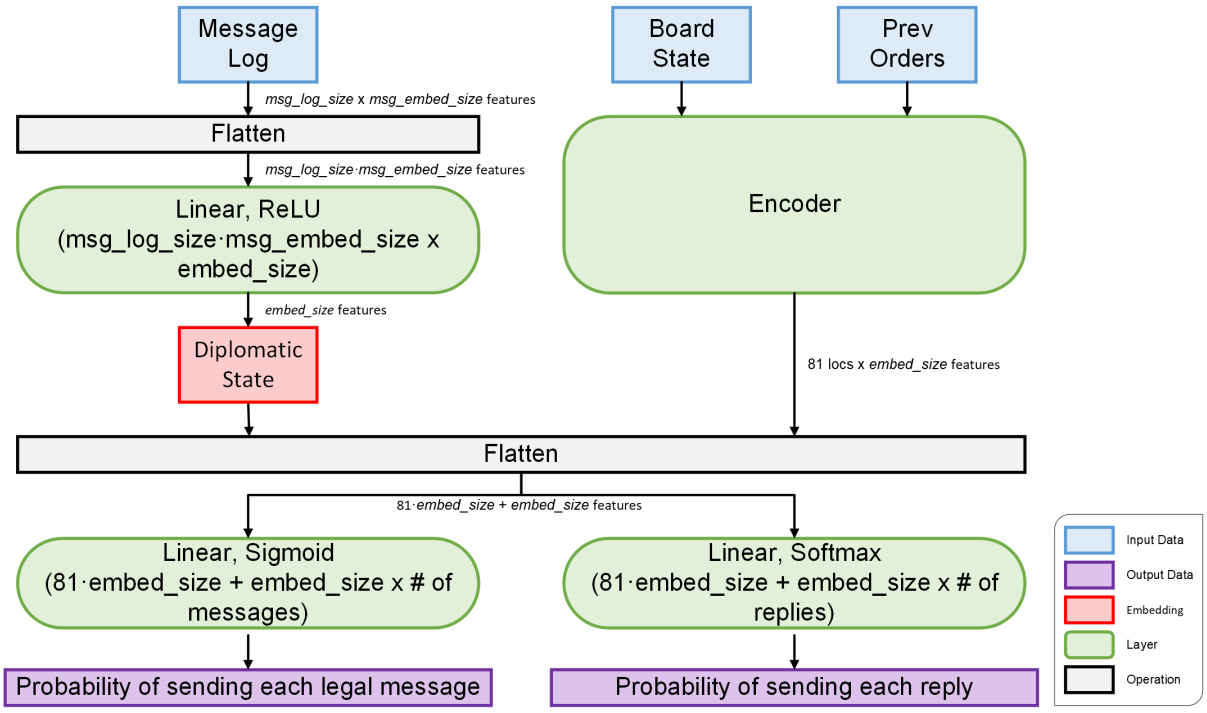


Figure 3.10: Architecture of the Press Model's Message Networks

and then fed to the Reply linear layer, except here we want the probability of sending each reply, so we use a softmax function instead of a sigmoid one (Section 2.2.1), and choose and send a reply (or don't in case the no reply option is chosen) based on these probabilities. After each reply is chosen and possibly sent, the outgoing messages are generated again since a message was received and another may have been sent, changing the message log and in turn the diplomatic state, which may warrant new messages. This loop of waiting for messages, calculating the reply to any received messages and generating possible new messages when the state changes is repeated until the defined press time for the model is over, and at that moment the orders to be played are calculated using the latest diplomatic state and the model waits for the next phase. This logic can be observed in Figure 3.11.

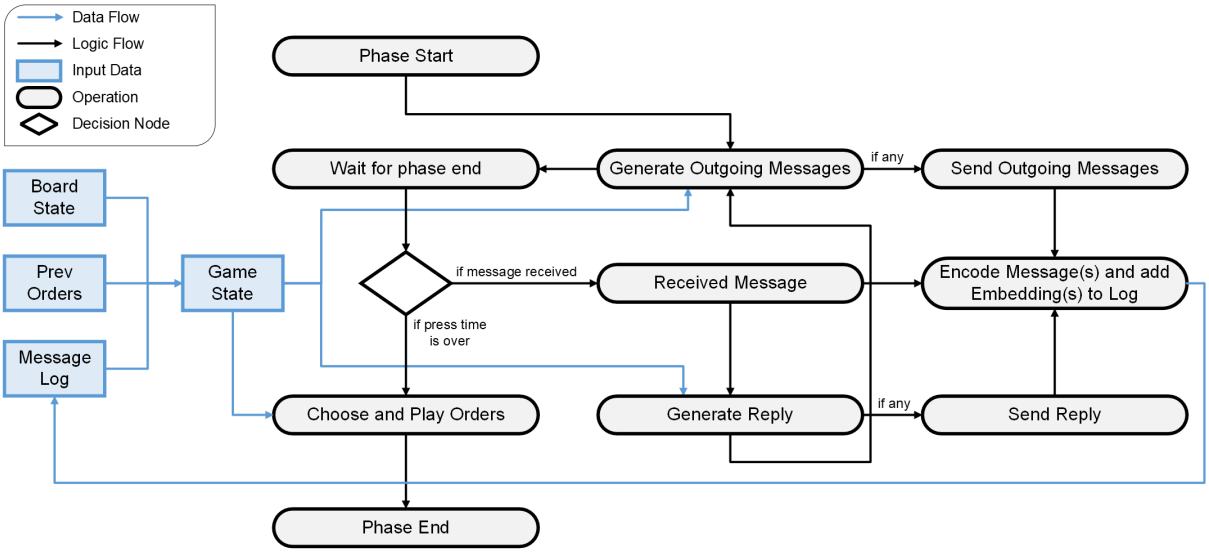


Figure 3.11: Data and State Flow of the Press Model

4

Training

Contents

Our implementation process went through a few stages to get to our completed bot. Our first idea was to try to train our No Press agent from scratch using RL but that proved unsuccessful for the reasons described below. We then tried SL and that showed better results and so we used the same technique on the Press variant.

We also wanted to apply RL on top of the SL agents to observe if the model improved in terms of gameplay and communication, but due to time and hardware constraints that wasn't possible, so we leave that experiment to a possible future work.

4.1 Reinforcement Learning

For our attempt at RL we tried to start from scratch, that is without a human imitation base, like DORA (Section 2.3.4). We decided to use A2C (Section 2.2.7), as DipNet did for its RL agent (Section 2.3.1), making use of the policy and value networks described in Section 3.2.1 as the actor and critic respectively.

Unfortunately we observed little to no progress in both the decrease of the loss and the quality of play. We deduced two reasons for this, the first one is that A2C was not a good algorithm for the game of Diplomacy, this seems like a good explanation since DipNet seems to observe minimal performance improvements with A2C, even when training on top of an imitation learning base, and DeepMind's bot (Section 2.3.2) outperformed it by changing the A2C algorithm for a BR based method. The second reason is that the available hardware for our work wasn't powerful enough to train a model without any baseline as discussed in Section 5.4.

4.2 No Press Supervised Learning

Our second, more successful attempt, at No Press Diplomacy used SL as a baseline as most of the previous works did. For this we requested the dataset used for DipNet from Philip Paquette, one of the authors. This dataset is described below in Section 4.2.1.

For the SL itself we used the standard technique of predicting the outputs using the inputs from an entry of the dataset, calculating the loss, in this case using Cross Entropy Loss, and propagating that loss backwards through the network.

For the learning rate we settled on $1e^{-4}$ for the policy network and $1e^{-6}$ for the value network after a few experiments. We used a lower learning rate for the value network as we observed that since it predicted only 7 values it oscillated a lot when using a higher learning rate, compared to the policy network which predicted a probability for each legal order.

We then kept track of the loss and accuracy of the predictions to observe the improvement of the model, these results are shown and discussed in Figure 5.1.

4.2.1 No Press Dataset

As explained above, we obtained a dataset of human games played on the webDiplomacy [Kuliukas and Zultar, 2004] from Philip Paquette. From the dataset provided we selected the No Press games as well as the Press games that didn't have any messages for a total of 139 735 games, of which we used 122 993. We removed 12 214 games that were not played on the standard 7-player map, 883 games that had the "BUILD_ANY" rule, which would allow powers to build units on any controlled province instead of only the home locations, which we thought strayed too much from the default game, and finally we removed 3 645 games that didn't have any power reaching at least 7 SCs. In Table 4.1 and Table 4.2 we can see the distribution of the maps and the rules for each game, keeping in mind that the rule distribution only contains the games on the standard map already.

Map	Number of Games
Standard	127 521
Standard France Austria	7 425
Standard Germany Italy	3 939
Standard Age of Empires	521
Standard Fleet Rome	282
Standard Age of Empires 2	47

Table 4.1: Number of games on each map on the No Press Dataset. Only the Standard games were used to train the model

Rule	Number of Games
POWER_CHOICE	127 521
NO_CHECK	73 290
NO_PRESS	21 915
PUBLIC_PRESS	1 240
BUILD_ANY	883

Table 4.2: Number of games that used each rule on the No Press Dataset after removing games on non-standard maps. Games with the "BUILD_ANY" rule were removed from the dataset.

We also found that from the final 122 993 games, 68 524 of them included at least one illegal order but since the number of games was so large we decided not to exclude the games, instead removing the orders. These illegal orders were mostly mistakes and orders we decided not to accept to reduce the number of possibilities for our model, like mentioning a non-existing coast, for example "BUL/WC", convoys longer than 4 that we deemed illegal for our bot, as described in Section 3.1.1, and unnecessarily long convoys that weren't part of our legal order list, for example, if you want to go from YOR to NWY, you would always have to pass through NTH since it's the only sea adjacent to YOR. But

since NTH is already connected to NWY, a convoy order only ever has to go from YOR to NTH to NWY, making any order passing through any other sea, like NWG, useless.

We did notice that there were 8 035 mentions of the SWI province, that is Switzerland, which in the standard map is a neutral province that cannot be occupied. This is used in No Press games to communicate between players, but since it would mean adding a lot of new orders we decided to keep it illegal for our bot.

Finally we did an analysis of the win rate of each power on the final dataset, shown in Table 4.3.

Power	Number of Solo Wins	Percentage of Solo Wins	Number of Survived Draws	Percentage of Survived Draws
Austria	5 625	11.18%	40 746	56.07%
England	6 148	12.22%	54 589	75.12%
France	8 253	16.40%	55 680	76.62%
Germany	7 008	13.93%	44 477	61.20%
Italy	4 696	9.33%	44 418	61.12%
Russia	8 864	17.62%	44 249	60.89%
Turkey	9 726	19.32%	55 267	76.05%
Total	50 320	40.91%	72 673	59.09%

Table 4.3: Win rate per power on the No Press training dataset

4.3 Press Supervised Learning

For the Press variant of our SL bot we used the same method as the No Press variant (Section 4.2) but now using a dataset with messages. For the message networks we used the same learning rate as the policy network, $1e^{-4}$. We also froze the value network and imported its weights from the No Press model since it did not depend on the diplomatic state. Again we kept track of the policy's loss and accuracy, adding now the loss for the messages, as well as their F1-Score [PyTorch-Metrics, nd]. We use F1-Score instead of regular accuracy as the accuracy would always be very high due to the large number of messages not sent.

For the dataset we needed games played with up to Level 20 Press (Table 2.1), meaning using only PCE, ALY, XDO and DMZ messages. As there are no human games played with such limited vocabulary, and producing one would take a long amount of time and expert Diplomacy players, we decided to generate our own dataset using Albert (Section 2.1.3.D). As with the No Press bot, RL can be later used to surpass the base dataset performance, in this case to beat Albert.

Although we only generated a relatively small number of games it was enough to see an increase in performance as seen in Figure 5.2, proving this method would be successful with more time and data.

4.3.1 Press Dataset

The press dataset used both Albert and RandBot (Section 2.1.3.A) to generate games. RandBot was needed to avoid early draws which Albert has a tendency to do, additionally RandBot was used in place of other bots since it ran locally (unlike for example DumbBot), not needing to connect to the DAIDE server which was prone to errors, reducing the time it took to generate games, and was weak enough so that Albert could play for a while and generate a good amount of messages in different states, since with DipNet the game would be over very quickly. Additionally we randomized the amount of RandBots each game, from 1 to 5, so we could generate Albert's messages in different situations, which explains why RandBot managed to beat Albert in a few games. These games were played on the standard map using no extra rules, so the powers were attributed randomly.

Due to limited hardware and time, and the slow speed of Albert, we were only able to generate 324 games, of which 60 were removed since none of the Albert powers reached at least 7 SCs, leaving us with 264 learnable games, the power distribution of which is presented in Table 4.4.

	Albert		RandBot	
Power	Solo Win	Survived Draw	Solo Win	Survived Draw
Austria	9	79	4	26
England	2	92	1	32
France	16	92	6	32
Germany	7	72	1	20
Italy	14	89	6	23
Russia	9	71	1	35
Turkey	3	88	3	37
Total	60	182	22	149

Table 4.4: Number of wins and survived draws per power on the Press training dataset

5

Evaluation

Contents

In this chapter we evaluate both Icarus models, No Press and Press. We examine their training results, specifically the loss and the accuracy of the model, evaluate their performance directly by comparing them to some of the bots described in Sections 2.1.3 and 2.3 as well as looking at some specific cooperation metrics that we find important for a good diplomacy bot.

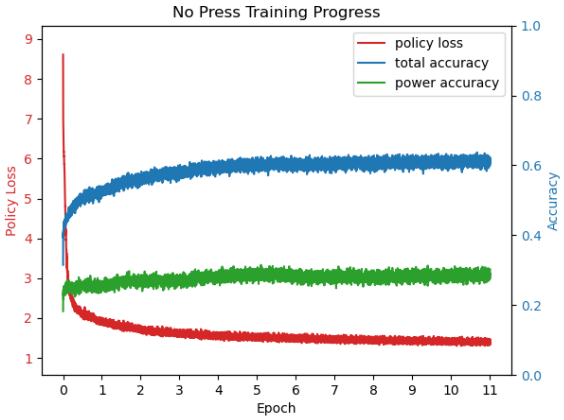
5.1 Training Results

Since both models use SL we can check the training progress every run through the dataset, or epoch, by checking the loss of the model, which should get smaller with more training, and the accuracy of the predictions, which should grow. Additionally we used validation sets to check if and when the models overfit to the dataset, with validation size of 200 games for No Press Icarus and 20 games for Press Icarus.

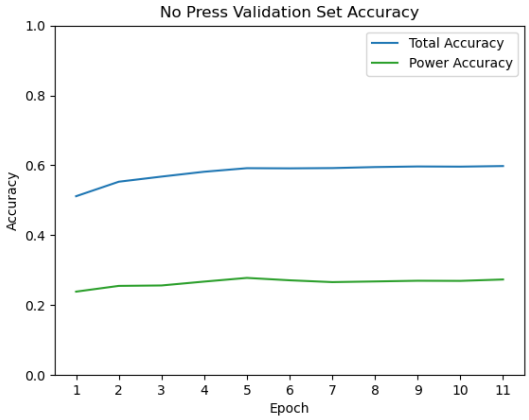
5.1.1 No Press Training

We trained the No Press model using SL with the human dataset described in Section 4.2.1. To evaluate the training progress of the model we used three metrics, policy loss, total accuracy and power accuracy. Policy loss is the Cross-Entropy loss of the predictions as explained in Section 4.2. Total accuracy and power accuracy measure the accuracy of the predicted orders, total accuracy measures the accuracy of all the predicted orders by dividing the correctly predicted orders by the total number of orders, and power accuracy measures the number of powers who correctly predicted all their orders, so it is calculated by dividing the number of powers who got all their orders right over the number of powers still playing.

As seen in Figure 5.1 during training the loss keeps getting smaller, showing us the model is learning, and the accuracy also keeps increasing although it seems to have diminishing returns from the fifth epoch forward. In fact if we also look at the validation accuracy the fifth epoch seems to be the peek accuracy, with the following epochs showing some degree of overfitting. For this reason we decided to use the fifth epoch model for our performance tests against other bots in Section 5.2.1.



(a) Policy loss, total accuracy and power accuracy of predictions per epoch for the No Press model

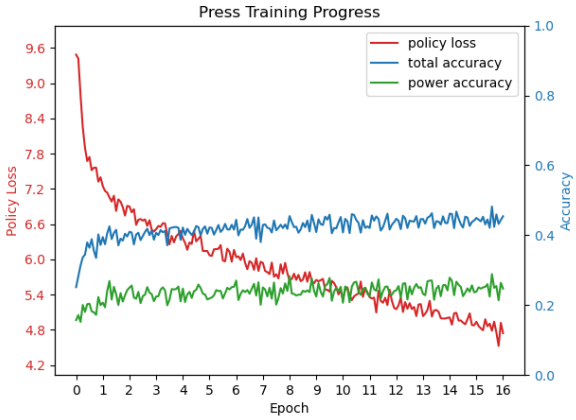


(b) Validation set accuracy per epoch for the No Press model

Figure 5.1: Training results for the No Press model

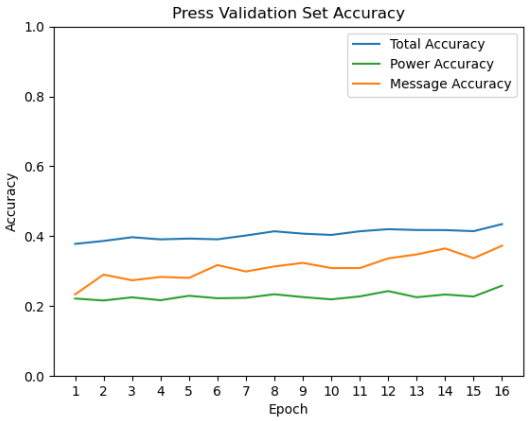
5.1.2 Press Training

For the Press model we also used SL but with the Albert-generated press dataset described in Section 4.2.1. To evaluate its training progress we used the same metrics as with the No Press model, policy loss, total accuracy and power accuracy, and we added the message loss and the message f1-score as explained in Section 4.3. Again in Figure 5.2 we see the loss decrease and the accuracies and f1-score increase for the policy and the messages respectively, in this case however there is no plateau or overfitting, probably due to the small dataset and the consequently short training time, so we used the model generated by the last completed epoch, epoch 16, for the bot games described below. These results are however still enough to prove that our model can learn and would probably obtain good results in press games given a bigger dataset and more training time.



(a) Policy loss, total accuracy and power accuracy of predictions per epoch for the Press model

(b) Message loss and f1-score per epoch for the Press model



(c) Validation set accuracy per epoch for the Press model

Figure 5.2: Training results for the Press model

5.2 Performance against other bots

To measure the performance of our bots we mostly used games against the previous bots, namely DumbBot and Albert from the rule bots (Section 2.1.3) and DipNet SL and DipNet RL from the ML agents (Section 2.3.1). We chose not to use the more advanced bots like SearchBot (Section 2.3.3) and DORA (Section 2.3.4) since our bot unfortunately could not beat DipNet so it was very unlikely to have been able to beat stronger opponents.

We played 100 games for each match, with 2 matches for each relevant opponent bot for No Press Icarus, one where we paired 1 of our agents against 6 other bots and another with 6 of our bots and 1 opponent, and for Press Icarus we added some relevant matches with and without press so we could see

how it affects the win rate. These configurations allow us to check if our bot is capable of communicating with the opposing agents when playing alone and if it is strong enough to beat powerful agents if it has a majority of agents. We also planned to play 4 vs 3 matches in case some of these bot pairs seemed to be of similar strength, however we observed that there was always a clear winner regardless of the number of agents so we decided these balanced matches were unnecessary.

5.2.1 No Press Icarus

The results of the No Press Icarus can be seen in Table 5.1. We can see that the model consistently wins against RandBot both when playing 6vs1 and 1vs6. Unfortunately it fails to beat DumbBot or the No Press Albert variant (Section 2.1.3) in the 1vs6 matches and has very poor results in the 6vs1 matches.

Interestingly enough it performs better against Albert in 6vs1 matches than it does against DumbBot which is supposedly worse (although the 24 game win rate is far from the 85.71 expected from a balanced game), we attribute this increased win rate to the fact No Press Icarus is trained on a dataset of human games and Albert uses rules manually created with a human playstyle in mind so Icarus would be better prepared to answer Albert's plays, while DumbBot logic is more mathematical so its action may differ from the human meta Icarus is trained for.

Finally as expected the bot is unable to beat any of the more advanced DipNet models (Section 2.3.1) so we didn't test against the later ML agents as the result would surely be the same.

Match	Wins	Survived Draws
6 No Press Icarus vs 1 RandBot	99	1
1 No Press Icarus vs 6 RandBot	97	1
6 No Press Icarus vs 1 DumbBot	10	0
1 No Press Icarus vs 6 DumbBot	0	4
6 No Press Icarus vs 1 Albert	24	0
1 No Press Icarus vs 6 Albert	0	0
6 No Press Icarus vs 1 DipNet SL	4	0
1 No Press Icarus vs 6 DipNet SL	0	1
6 No Press Icarus vs 1 DipNet RL	1	0
1 No Press Icarus vs 6 DipNet RL	0	0

Table 5.1: Icarus No Press win rate against the various other bots in 1 vs 6 games, based on 100 games per match. In a balanced match the bot with one agent should win around 14.29 games and the six agents 85.71.

5.2.2 Press Icarus

We ran similar tests for the Press variant of our bot as seen in Table 5.2, with the difference that we ran some matches with and without press so we could see the difference in performance caused by the messages.

Again our agent easily beats RandBot, meaning the message networks added don't interfere too

much with the basic order logic. It does perform slightly worse, but still with an overwhelming win rate, in 1vs6 games against RandBot but that is to be expected given the smaller dataset and train time.

Just like No Press Icarus it loses to the DipNet agents, as expected and unfortunately the addition of press between the agents seems to have no effect against DumbBot agents, it does however have a slight effect on the games against Albert although this could be due to random chance. We explore the messages exchanged in those games in more detail in Section 5.3 to determine how the addition of messages really affects our agent.

Match	Press On	Press Icarus Wins	Press Icarus Survived Draws
6 Press Icarus vs 1 RandBot	Yes	99	1
6 Press Icarus vs 1 RandBot	No	99	1
1 Press Icarus vs 6 RandBot	No	86	12
6 Press Icarus vs 1 DumbBot	Yes	0	0
6 Press Icarus vs 1 DumbBot	No	0	0
1 Press Icarus vs 6 DumbBot	No	0	2
6 Press Icarus vs 1 (No Press) Albert	Yes	4	0
6 Press Icarus vs 1 Albert	Yes	5	0
6 Press Icarus vs 1 Albert	No	1	0
1 Press Icarus vs 6 Albert	Yes	0	0
1 Press Icarus vs 6 Albert	No	0	0
6 Press Icarus vs 1 DipNet SL	Yes	1	0
6 Press Icarus vs 1 DipNet SL	No	0	0
1 Press Icarus vs 6 DipNet SL	No	0	0
6 Press Icarus vs 1 DipNet RL	Yes	0	0
6 Press Icarus vs 1 DipNet RL	No	0	0
1 Press Icarus vs 6 DipNet RL	No	0	0

Table 5.2: Press Icarus win rate against the various other bots in 1 vs 6 games, some with press and some without, based on 100 games per match. In a balanced match the bot with one agent should win around 14.29 games and the six agents 85.71.

5.3 Cooperation Analysis

We will now examine more closely the messages sent and received by Press Icarus and how these affect how the agent interacts with other powers, like sticking to agreed XDO and DMZ proposals and how an alliance or peace affects supports to allies. For these statistics we used the 100 games played between 6 Press Icarus and 1 Albert shown in the table above (Table 5.2), since this gives us 6 Press Icarus agents to examine the messages and orders while still having an Albert agent to send messages that Icarus might not have learned to use. In this section we will use mostly the acronyms for the messages, these are described in more detail in Appendix A.

5.3.1 General Message Statistics

We started by analyzing the message patterns of both Press Icarus and Albert. First we graphed the average number of messages per phase for both bots in Figure 5.3. As we can see most messages are sent in the first phase, this consists mostly of PCE and ALY messages to establish the initial peaces and alliances.

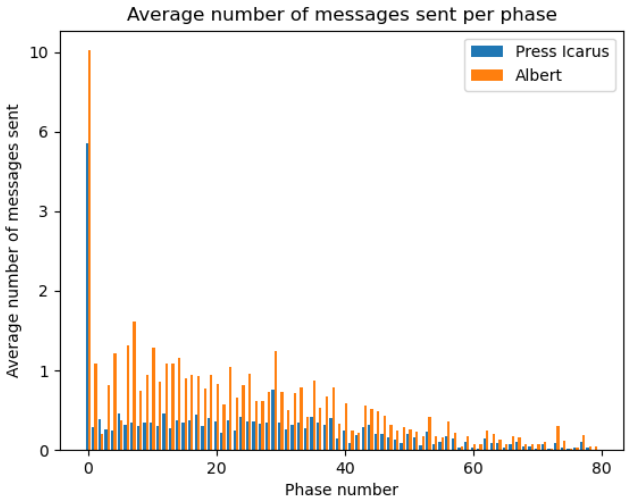


Figure 5.3: Average number of messages sent per phase over the course of the 100 games of Press Icarus vs Albert. Note that the top half of the y-axis is compressed for better visualization of the smaller values.

Next we analyzed the type of propositions (PRP) each bot sent per game as well as how many of those sent propositions were accepted by the other powers. Additionally we also examined how many propositions each agent received and accepted. This information is displayed in Table 5.3.

From this information we can extract a few significant conclusions:

- Press Icarus learned how to reply to certain messages since he accepted 51,72% messages and if it was random it would be closer to 33,33% since there are 3 reply options (accept, reject and ignore).
- Press Icarus didn't learn how to use DMZ and XDO orders, sending less than one DMZ per game and no XDO orders across the 100 games. This is probably due to the large number of variations these orders have, DMZ having variations for each location and power combination and XDO having one for each of the 13 244 unique orders. This combined with the small size of the training dataset, which likely had a very low percentage of the possible variations of these orders, made it unlikely for our agent to learn how to properly use these messages.
- Press Icarus is much more likely to accept ALY and DMZ propositions than Albert, this may be due to an imbalanced dataset preferring accepting propositions, or specifically ALY and DMZ proposi-

tions. A future work could explore how balancing techniques applied to the dataset can improve the performance of the model.

Player	Message	Average Sent PRP	% of Sent PRP Accepted	Average Received PRP	% of Accepted PRP
Press Icarus	PCE	3,68	76,10%	3,20	76,90%
	ALY	6,95	35,98%	6,01	40,00%
	DMZ	0,97	45,38%	1,89	53,49%
	XDO	0,00	-	2,41	45,86%
	Total	11,60	49,50%	13,51	51,72%
Albert	PCE	2,80	75,00%	5,67	72,84%
	ALY	8,36	58,13%	1,37	7,30%
	DMZ	6,84	51,17%	14,01	38,62%
	XDO	14,50	45,86%	0,00	-
	Total	32,50	52,65%	21,05	45,80%

Table 5.3: Statistics on the average amount of sent and received propositions in the 100 games with 6 Press Icarus vs 1 Albert

5.3.2 Message Impact on Orders

After getting a general view of the message distribution we moved to analyze how the agreements made affected the play of the agents, namely if agents followed the agreed DMZs and XDOs and if the peaces and alliances affected cross-power supports.

First we need to define how alliances work. For this we considered an alliance started between two powers when one of the powers sent a proposition for PCE or ALY and the other accepted, the alliance was then broken when one of the powers moved or supported a move into a supply center controlled by the allied power. This doesn't take into account the strategies of "bouncing" units purposefully (when two powers attack the same province with only one unit so neither power moves into the province) or of "borrowing" a supply center for one phase, both seen in higher level play, but given the limited communication ability of the bots we felt this wouldn't be an issue.

Given this definition we calculated Press Icarus was part of about 3,59 alliances each game, breaking 2,47 of them after roughly 3,57 movement phases. In contrast Albert made 8,12 alliances and broke 2,55 after 5,27 turns, having only a 31.4% break ratio to Press Icarus' 68,80%, telling us that while Icarus might have learned how to make alliances it hasn't fully grasped their implication in-game.

Next we looked at supports, specifically how many support orders each agent played, how many were self supports, how many were cross-power supports, of those how many were to allied powers and of all those supports which ones were successful, meaning if it was a support hold there was a unit there to be supported and if it was a support move the move being supported was actually played.

Starting with the self supports (Table 5.4) we can see that unsurprisingly Albert makes great use of self supports, making more than 40 supports per game, all of them being successful of course since it

controls both the unit supporting and being supported. Press Icarus however seems to lack the coordination to both play supports and the related order.

	Self Supports			
	Support Hold		Support Move	
Power	Attempts	% of Success	Attempts	% of Success
Press Icarus	1,81	99,75%	2,94	17,27%
Albert	21,62	100,00%	29,67	100,00%

Table 5.4: Statistics on the average amount of self supports played in the 100 games with 6 Press Icarus vs 1 Albert

Looking at cross-power supports (Table 5.5) it's not a big surprise that they are largely unsuccessful and that an alliance has little impact on the outcome given that alliances last only a few phases and that Press Icarus doesn't understand how to best make use of support orders, in fact there are almost no allied cross-power supports in the 100 games. Unlike self supports this affects Albert as well since these games are played between 6 Press Icarus and only 1 Albert, so Albert's supports are always directed at Icarus' controlled powers making them largely unsuccessful.

	Cross-Power Allied Supports			
	Support Hold		Support Move	
Power	Attempts	% of Success	Attempts	% of Success
Press Icarus	0,008	100,00%	0,015	11,11%
Albert	0,01	100,00%	0,11	27,27%

	Cross-Power Non-Allied Supports			
	Support Hold		Support Move	
Power	Attempts	% of Success	Attempts	% of Success
Press Icarus	3,60	99,62%	5,83	16,65%
Albert	3,40	16,76%	0,62	93,55%

Table 5.5: Statistics of the cross-power supports played in the 100 games with 6 Press Icarus vs 1 Albert

Taking a look at the remaining messages, starting with the DMZ, or demilitarised zones, we saw in Table 5.3 that Press Icarus rarely ever sent DMZs but when it did it often chose the locations BOH and SIL, which again likely comes from an imbalance in the dataset. Across the 100 games Press Icarus agreed to 496 DMZs of which he broke 343 after about 4,35 turns, with a 69,15% break ratio it seems to lack the understanding behind the agreement. Albert on the other hand agreed to 338 DMZs and broke only 88 after 6,5 turns, a 26,04% break ratio.

Finally from the XDOs, or suggested orders, as shown above (Table 5.3) Press Icarus sent no XDOs so Albert received none, but of the 653 XDOs Icarus accepted across the 100 games only 12 were actually played, showing that Icarus also doesn't understand the meaning behind XDOs, as expected from our small dataset.

5.4 Discussion

Starting with the No Press Icarus model, looking at both the training results and the performance against other bots (Section 5.1.1 and 5.2.1) we see that it was capable of learning the game to a certain degree, beating out RandBot and winning some games against Albert and DumbBot, but still being considerably weaker than the state-of-the-art ML bots. We attribute this to a few factors, first our model has some flaws as discussed in the previous chapters, for example our lackluster encoding of the previous orders (Section 3.1.1), next we lacked the time and computational power (discussed below) to experiment with different models and to tune the network's parameters to increase performance, and finally, while our dataset was the same as the one used for DipNet (Section 2.3.1) it could have used some of the improvements introduced in the later ML bots, like filtering games based on player rating.

Our Press model suffered similar issues, it also obtained a basic grasp of the game, again beating RandBot in both 6v1 and 1v6 matches, and managed to work with the press side of the game to an extent, being able to send and agree to peace and alliances, but it still lacked the understanding for both applying those agreements to the orders played and to process the more complex proposals like the demilitarised zones and the suggested orders.

Press Icarus had the same model flaws as the No Press model with the addition that we chose a pretty simple network to process the messages consisting mostly of linear layers, that while quick to train, which was necessary for the available hardware, made it so the message understanding of our model was also pretty basic. The dataset problem however was greatly amplified, consisting of only 264 games generated by Albert, making it so not only the model would likely never be able to beat Albert, given that it was learning directly from its games without improvement, but also the limited amount of data made it so the model only saw a very small percentage of the thousands of possible messages, making it practically impossible to obtain a good grasp on the press system.

Again Press Icarus, like No Press Icarus, had the problem of limited time and hardware. To train the No Press model we used a GeForce RTX 3090 24GB and for the Press model a GeForce RTX 2080 Ti 11GB, which while good graphics cards had nowhere near the processing power needed to complete these tasks in a timely manner. In fact it took us 48 days to train the 5 epoch model we ended up using for No Press Icarus, although we ran it for over 3 months to obtain the 11 epochs of data in Section 5.1.1.

Given the significantly smaller dataset the Press model didn't suffer as much on the training side, taking us only 2 days to train it and allowing some experimentation with different techniques and parameters. The lack of computational power did however cause the dataset to be this small, as it was composed of games using the Albert bot that we generated ourselves, and the slow speed at which Albert played its orders and the error-prone DAIDE servers needed for Albert to connect made the process very slow, especially since we were using only one machine to play those games.

For comparison the FAIR team used 192 Nvidia V100 GPUs, "the most advanced data center GPU

ever built" [Nvidia, nd], when training DORA, 4 used for the actual training and the rest for data generation, and it still took a week to train.

6

Conclusion

Contents

6.1 Conclusions

In this work we developed two Diplomacy agents, with and without press capabilities, which was the main goal of this thesis. We trained these agents with supervised learning and employed some state-of-the-art network architectures like Transformer [Vaswani et al., 2017].

Although our agents weren't able to surpass the more advanced Diplomacy bots the fact that they both consistently beat RandBot means not only that they acquired some basic grasp on the game but that the press capabilities of the Press bot didn't impair its No Press performance.

Our implementation of the messaging system also had a positive result as we managed to learn how to send and agree to simple peace and alliance propositions, even if it lacked the ability to apply that knowledge in-game or process more complex messages it shows the system is a good starting point for more advanced messaging bots. The code for our agents is available on GitHub [Araújo, 2023].

6.2 Future Work

This work has multiple areas that could be improved. Starting with the dataset, the No Press dataset should be filtered based on player rating, learning from only the best players, and there is work to be done studying how the learning process could be affected by balancing the dataset, that is making sure there is an even number of each type of order and message. The Press dataset is where the biggest improvement could be made, we used Albert to generate our games and while that is a valid strategy if one were able to generate a large number of them it would be better to use an expert human dataset for better results, of course to generate that dataset with the reduced press applied in this work would be difficult, an alternative is to upgrade the Press model to Full Press and use already available human datasets, similar to what Cicero [FAIR et al., 2022] did.

Speaking of upgrades to the model we suggest a few, first if using our exact same model the easiest improvement would be to use a PyTorch Embedding module [Paszke et al., 2019] for the previous orders like we used for the message encoding (Section 3.1.2) since the system we used assigns arbitrary values to each order type. Some improvements could also be made to the inputs of the LSTM, namely including the previous orders in the input instead of relying solely on the hidden state so the model can have a better context of the orders already played.

The Press model also has some possible improvements, first by fixing the limitations we imposed on the possible messages, namely DMZs only being able to mention one location and up to two powers at a time, when the DAIDE protocol allows for any number of locations and powers to be mentioned. The model could also be improved in several ways, for example the diplomatic state in our model is being used as an input to the LSTM but it probably should be added as input to the Encoder so the location embeddings are imbued with context from the diplomatic state. Another improvement would be adding

the diplomatic state as an input to the value network, either directly as an input or indirectly by adding the diplomatic state to the Encoder as mentioned, this would make the state value dependent on the messages exchanged making for a better assessment of the game state.

Our last suggested improvement is to use RL on the models after training with SL, as we intended to do but failed due to the time and hardware limitations, this would make it so the model could play better than the dataset it was trained on, making a full Albert press dataset that much more viable, although there should be extra care to make sure the messages retain their meaning as without a dataset to follow, the model could gravitate to using messages for different purposes than intended making it incompatible with human play, for example by punishing large deviations from a human imitation policy like Cicero [FAIR et al., 2022].

Bibliography

[Anthony et al., 2020] Anthony, T., Eccles, T., Tacchetti, A., Kramár, J., Gemp, I., Hudson, T., Porcel, N., Lanctot, M., Pérolat, J., Everett, R., et al. (2020). Learning to play no-press diplomacy with best response policy iteration. *Advances in Neural Information Processing Systems*, 33:17987–18003.

[Araújo, 2023] Araújo, A. (2023). Coder266/icarus.

[Bakhtin et al., 2021] Bakhtin, A., Wu, D., Lerer, A., and Brown, N. (2021). No-press diplomacy from scratch. *Advances in Neural Information Processing Systems*, 34:18063–18074.

[Bakhtin et al., 2022] Bakhtin, A., Wu, D. J., Lerer, A., Gray, J., Jacob, A. P., Farina, G., Miller, A. H., and Brown, N. (2022). Mastering the game of no-press diplomacy via human-regularized reinforcement learning and planning. *arXiv preprint arXiv:2210.05492*.

[Blackwell, 1956] Blackwell, D. (1956). An analog of the minimax theorem for vector payoffs.

[Calhamer, 2008] Calhamer, A. B. (2008). *The Rules of Diplomacy*. The Avalon Hill Game Co.

[de Mascarenhas, 2017] de Mascarenhas, S. F. (2017). Ai player for board game diplomacy. Master's thesis, Instituto Superior Técnico.

[FAIR et al., 2022] FAIR, M. F. A. R. D. T., Bakhtin, A., Brown, N., Dinan, E., Farina, G., Flaherty, C., Fried, D., Goff, A., Gray, J., Hu, H., et al. (2022). Human-level play in the game of diplomacy by combining language models with strategic reasoning. *Science*, 378(6624):1067–1074.

[FAIR et al., 2023] FAIR, M. F. A. R. D. T., Bakhtin, A., Brown, N., Dinan, E., Farina, G., Flaherty, C., Fried, D., Goff, A., Gray, J., Hu, H., et al. (2023). Diplomacy cicero and diplodocus.

[Graves and Graves, 2012] Graves, A. and Graves, A. (2012). Long short-term memory. *Supervised sequence labelling with recurrent neural networks*, pages 37–45.

[Gray et al., 2020] Gray, J., Lerer, A., Bakhtin, A., and Brown, N. (2020). Human-level performance in no-press diplomacy via equilibrium search. *arXiv preprint arXiv:2010.02923*.

[Hart and Mas-Colell, 2000] Hart, S. and Mas-Colell, A. (2000). A simple adaptive procedure leading to correlated equilibrium. *Econometrica*, 68(5):1127–1150.

[He et al., 2016] He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

[Hu and Wellman, 2003] Hu, J. and Wellman, M. P. (2003). Nash q-learning for general-sum stochastic games. *Journal of machine learning research*, 4(Nov):1039–1069.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. In *International conference on machine learning*, pages 448–456. pmlr.

[Jacob et al., 2022] Jacob, A. P., Wu, D. J., Farina, G., Lerer, A., Hu, H., Bakhtin, A., Andreas, J., and Brown, N. (2022). Modeling strong and human-like gameplay with kl-regularized search. In *International Conference on Machine Learning*, pages 9695–9728. PMLR.

[Karpathy, 2015] Karpathy, A. (2015). The unreasonable effectiveness of recurrent neural networks.

[Kipf and Welling, 2016] Kipf, T. N. and Welling, M. (2016). Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*.

[Kruijswijk, 2004] Kruijswijk, L. B. (2004). Datc - diplomacy adjudicator test cases.

[Kuliukas and Zultar, 2004] Kuliukas, K. and Zultar (2004). Webdiplomacy.

[Littman, 1994] Littman, M. L. (1994). Markov games as a framework for multi-agent reinforcement learning. In *Machine learning proceedings 1994*, pages 157–163. Elsevier.

[McMahan et al., 2003] McMahan, H. B., Gordon, G. J., and Blum, A. (2003). Planning in the presence of cost functions controlled by an adversary. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, pages 536–543.

[Mnih et al., 2016] Mnih, V., Badia, A. P., Mirza, M., Graves, A., Lillicrap, T., Harley, T., Silver, D., and Kavukcuoglu, K. (2016). Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR.

[Norman, 2007a] Norman, D. (2007a). Daide - diplomacy artificial intelligence development environment.

[Norman, 2007b] Norman, D. (2007b). Daide client downloads.

[Norman, 2010] Norman, D. (2010). Diplomacy ai development environment message syntax.

[Nvidia, nd] Nvidia (n.d.). Nvidia v100.

[Olah, nd] Olah, C. (n.d.). Understanding LSTM networks.

[Paquette, 2020] Paquette, P. (2020). Diplomacy: Datc-compliant game engine with web interface.

[Paquette et al., 2019] Paquette, P., Lu, Y., Bocco, S. S., Smith, M., O-G, S., Kummerfeld, J. K., Pineau, J., Singh, S., and Courville, A. C. (2019). No-press diplomacy: Modeling multi-agent gameplay. *Advances in Neural Information Processing Systems*, 32.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[Patil et al., 2020] Patil, S., Mudaliar, V. M., Kamat, P., and Gite, S. (2020). Lstm based ensemble network to enhance the learning of long-term dependencies in chatbot. *International Journal for Simulation and Multidisciplinary Design Optimization*, 11:25.

[Perez et al., 2018] Perez, E., Strub, F., De Vries, H., Dumoulin, V., and Courville, A. (2018). FiLM: Visual reasoning with a general conditioning layer. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 32.

[PyTorch-Metrics, nd] PyTorch-Metrics (n.d.). F-1 score - pytorch-metrics.

[Shuster et al., 2022] Shuster, K., Komeili, M., Adolphs, L., Roller, S., Szlam, A., and Weston, J. (2022). Language models that seek for knowledge: Modular search & generation for dialogue and prompt completion. *arXiv preprint arXiv:2203.13224*.

[Sutton and Barto, 2018] Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.

[User:Wiso, Public domain, via Wikimedia Commons, 2008] User:Wiso, Public domain, via Wikimedia Commons (2008). Neural network example.

[van Hal, 2013] van Hal, J. (2013). Diplomacy ai - albert.

[Vaswani et al., 2017] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. *Advances in neural information processing systems*, 30.

[Vitay, nd] Vitay, J. (n.d.). Advantage actor-critic methods.

A

DAIDE Press Messages

Here are detailed the messages used for reaching agreements between powers for Level 10 and Level 20 Press in the DAIDE framework. For simplicity we will focus on the negotiation messages represented by the keyword **press_message** and ignore the ones pertaining to communicating with the server and order submission, so there are no relevant messages in Level 0 (No Press) since there are no direct messages between players. For a more detailed description of all messages, one can consult the DAIDE syntax document [Norman, 2010].

- **Level 10**
  - Proposing an Arrangement:
    - **press_message = PRP (arrangement)**: The sending power is proposing an arrangement. Unless otherwise specified an agreement, formed by accepting an arrangement, lasts for one turn. For the agreement to be valid, all receiving powers have to reply using the **YES** token. An **arrangement** can be in the forms described below.
    - **arrangement = PCE (power power power . . .)**: Arrange Peace between the listed powers. The arrangement is continuous, that is, it lasts for more than the current turn.

* **arrangement = ALY (power power . . .) VSS (power power . . .)**: Arrange an Alliance between the powers in the first list. The second list is the powers to ally against. The arrangement is continuous.
* **arrangement = DRW**: Arrange a Draw. In the case of a game that accepts partial draws this may also be DRW (power power power . . .). The list of powers must include at least two powers. This does not immediately draw the game if accepted, it is only a proposal, to actually request a draw one would use **DRW** while this command would be **PRP(DRW)**.
* **arrangement = SLO (power)**: Propose a Solo (solo win by capturing 18 SCs) to the specified power. Note that the client can't actually order a solo – but may be able to order its units in a way that causes one to occur.
* **arrangement = NOT (arrangement)**: All of the available arrangements can have **NOT** placed in front of them to mean the opposite.
* **arrangement = NAR (arrangement)**: All of the available arrangements can also have **NAR** placed in front of them. This represents the lack of an arrangement, so the arrangement can or not be followed. **NAR** is most commonly used in order to undo a previous arrangement.

– Replying to a message:
  * **reply = YES (press_message)**: Accept the arrangement
  * **reply = REJ (press_message)**: Reject the arrangement
  * **reply = BWX (press_message)**: Refuse to answer. None of the sending power's business.

– Canceling a proposal:
  * **press_message = CCL (press_message)**: A press message which proposes an arrangement can be canceled before the arrangement has been agreed to by using a **CCL** message. This can also be used to cancel a **YES** reply to a proposed arrangement before all parties involved in the proposal have replied. A **CCL** message which is sent after the arrangement has been agreed has no effect, and **CCL** may not be used to cancel a message which is not attempting to form an agreement.

– Making a statement:
  * **press_message = FCT (arrangement)**: This message states that **arrangement** is true. This is generally used either to pass information or to unilaterally override a previously made agreement. There are no defined replies to **FCT**, although, of course, a **FCT** message may cause other statements or proposals to follow.

- **Level 20**
  - **arrangement = XDO (order)**: This is an arrangement for the given order to be ordered. **order** uses the same format as the normal order submissions. An **XDO** arrangement applies to the next turn in which the order type is valid, Hold, Move, Supply and Convoy for the Movement Phase, Retreat or Disband for the Retreat Phase, and Build, Remove or Waive for the Adjustment Phase. Replies use the format described in Level 10.
  - **arrangement = DMZ (power power . . .) (province province . . .)**: This is an arrangement for a demilitarized zone, meaning the listed powers should remove all units from the listed provinces and not order to move, support, convoy to, retreat to, or build any units in any of these provinces. The arrangement is continuous. Replies use the format described in Level 10.