Alfie Edwards

CM3203: One Semester Individual Project

Supervised by Dr Richard Booth

Final Report

Training Pairs of Communicating Machine Learning Agents to Complete Cooperative Tasks Requiring Information Exchange

May 11, 2018

Contents

1 Abstract

In this paper, I present my methods and findings after exploring the practicality of connecting the two neural networks with a communication link, and training the system as a single unit. I created two systems using different approaches, which I tested using games designed to require communication between the players to achieve success.

2 Introduction

A machine learning system can optimise a black box to approximate some function. Can this black box be extended further to cover multiple machine learning agents and the way they exchange information? In this paper, I explore the idea of connecting multiple machine learning agents using a general communication channel, and training them as if they were a single system. I do this by adapting the machine learning techniques of Q-Learning [2], NeuroEvolution of Augmenting Toplogies [4], and Cooperative Coevolutionary Genetic Algorithms [3] to work with the idea of multiple communicating agents.

3 Starting Point

I started with the idea of having machine learning agents develop their own protocol for information exchange within the context of a larger task. In other words, information exchange would be included within the black-box of the machine-learning algorithm, and would be optimised through standard training techniques. I wanted to test the viability of this idea, and any benefits it could bring.

3.1 Preliminary Research

Before beginning any planning work, I carried out initial research.

Existing Application of Multiple Agents

I first looked into the existing instances of utilising multiple agents to solve problems, looking for inspiration. I came across an interesting paper [5] describing multi-agent systems, or MAS for short.

Some problems can be split into multiple, much simpler, problems such that the combined complexity of the smaller problems is less than that of the original problem. In these cases, it can be faster to train multiple machine learning agents to complete the smaller sub-problems, then combining their outputs in a fixed way to form a complete solution. This is the basis of multi-agent systems. The paper explores these systems with a focus on machine learning, but the concept can be applied to any sort of system with multiple components.

MAS can also be applied when there is a forced division of information between agents, for example a distributed sensor network. In these domains, each agent only has immediate access to its own information, and as such, it is useful if each agent can work towards a meta-task by performing processing on only its subset of the total information.

Reinforcement Learning

One of the main potential benefits I was interested in was the ability to create a system that can be applied to a wide range of problems. To expand upon this I also decided to investigate reinforcement learning. It is common in machine learning to generate a model based upon a list of known inputs and outputs called the training data. You can take an existing model (for example an experienced human), then record the behaviour of them model, then use that data to train a machine learning agent that can emulate that model. Using this technique, the effectiveness of your machine learning agent is limited to the effectiveness of the model form which you generated your training data. Another downside is that an existing model is required at all.

Reinforcement learning entails a different approach. The training data is generated as you train your model. Effectively, your model learns from experience, focusing on or avoiding behaviours based on some metric to measure the quality of an outcome. This means you do not require any prior knowledge about how to achieve a desirable outcome from given inputs, and instead only require a way to evaluate outcomes. This general family of techniques is commonly seen in machine learning applications based around games, as typically it is not possible to generate perfect training data, and human-generated examples are both limited by human knowledge, and difficult to collect in large quantities.

Machine Learning Framework

Use of frameworks is very common in machine learning applications. Frameworks allow simple implementation of well-known techniques, and can facilitate both efficiency and code simplicity. A framework was perfect for this project as my focus was on the effectiveness of higher-level techniques. As part of my preliminary research, I investigated popular machine learning frameworks. I chose to use TensorFlow, mainly because of the wealth of related material available online, such as tutorials and questions. I also found it was capable of modelling unconventional neural network structures, a key requirement for the project.

3.2 Initial Ideas

I chose to stick closely to the idea posed as my starting point, and incorperate ideas from my research.. I planned to extend the idea of multi-agent systems to include communication links between the agents. I could then train the entire system as if it was one agent, using standard machine learning techniques.

Asymmetric Information

I planned to create design problems for the system such that they you not perform well without information exchange taking place. To do this I would need to intentionally withhold data from some agents, so they they must acquire it through a communication channel.

Neural Networks

I chose to focus specifically on using neural networks, as I understood how they worked, and they seem to be the dominant model in machine learning at the moment. This would also make it easy to implement a communication link between the two agents, as it could be modelled using standard neural network nodes and edges.

Games

I wanted to frame any problems I worked on as games. Games can serve as abstract, clinical environments which can be easily adjusted to suit your needs. I also believed that the game paradigm can make complex problems more immediately understandable, bringing with it intuitive concepts such as strategies and score. A multi-agent system with division of information could be framed as a cooperative game, where the players are given asymmetric information.

Two-Agents

I decided that I would focus on creating two agent systems. This put a good limit on the scope of the project, and helped ground my ideas.

Flexibility

Ideally I wanted to create systems that can be applied to a wide range of games with minimal need for domain-specific parameters. The minimum possible parameters would be the state size and the number of possible moves, since these can vary between games. These values are necessary for a neural-network based system to ensure that data can flow into and out-of the network.

3.3 Initial Goals

From my initial ideas, I created a set of goals. Some are concrete, while some are more qualitative. These goals represent the practical things I hoped to achieve, based on my initial ideas.

One-Player and Two-Player

I wanted my implementations to work with both One-Player and Two-Player games. This would allow me to make comparisons between the performance of each.

Graphical Displays

The game paradigm frames problems in an intuitive way. This can be amplified by the use of graphical displays to provide visual representations of systems. The more intuitive a system, the easier it is interpret and present results from experimenting with that system. Because of this, I decided I would create graphical interfaces for all non-trivial games I experimented with.

Successful Implementation

My biggest goal coming into this project is to create at least one successful implementation of my ideas. I want to successfully train a two-agent system to play a cooperative game that requires information exchange in order to succeed.

Multiple Implementations

I wanted to test more than one technique. A wide range of machine learning techniques exist, and I wanted to avoid focusing too heavily on a single one. This way I could create a simple system to test viability, then also experiment with a more unusual system.

Reinforcement Learning

I aimed to base my implementation on reinforcement learning, in order to achieve a good level of flexibility. This would eliminate the need pre-prepared training data as an input, and make it much easier to test my systems with multiple different games.

4 Games

I created a total for four games test my systems with. Each game has two versions, one-player and two-player.

4.1 Definitions

- **Games**

  I use the word game to refer to a system with a state which evolves over one or more time-steps, starting at some initial state (not necessarily the same each time). At each time-step, one action from a set of possible actions must be taken. The current state, and the action taken decide the state at the next time step. A score is presented at each time-step based on the state and the actions previously taken. A higher score represents a more favourable state for the player. Finally, some states are termination states. From these states, no other action can be taken, and so the score is finalised.

- **Two-Player**

  In this report, whenever the term two-player game is used, it is referring to something more specific than a game with two players. The two-player games in this project are cooperative games, in which information exchange is required to achieve an adequate score. In a two-player game, a single score is given to both players, meaning they are intrinsically dependant on each other.

4.2 XOR

A common basic test for a machine learning system is to learn the XOR function. XOR has a very small domain and range, but is an example of a non-linear function. Learning it should be trivial for any well-functioning machine learning system. I have framed the XOR function as a game where the state is a pair of 2 numbers, each 0 or 1. There are two available actions, 0, and 1. Performing the action corresponding to the XOR of the numbers in the state yields a score increase. Performing the other action yields no score increase. The game terminates after 1 turn. To perform well in the game, you must be able to perform an XOR operation on the two numbers in the state, so you can pick the action will increase your score.

Adapting for Two Players

To adapt this game for two players, I created a simple division of information. Each player plays its own separate game, however the second values in their states are swapped. This forces the layers to communicate this value to each other before they can reliably calculate the best action to take.

For example, say player 1 was given the state $(1, 0)$, and player 2 was given the state $(1, 1)$. For one-player XOR, the best actions would be 1, and 0 respectively. In two-player XOR however, the second values in their states have been swapped.

Player 1 needs to XOR the first value from their state with the second value from player 2's state to calculate the best action. This gives 1 XOR $1 = 0$. Player two must do the same with the *first* value from their state, and the *second* value from player 1's state, giving 1 XOR $0 = 1$.

4.3 Pathfinding

This is much more recognisable as a traditional game. I have created a graphical display for the game which displays the state and updates as the state changes. The game consists of a grid of tiles, called the board. Each tile is either a wall or a floor. The player starts on a floor tile and moves one grid tile each turn, in one of 4 cardinal directions $(\rightarrow, \uparrow, \leftarrow, \downarrow)$. The player can only move into floor tiles. To increase the score and terminate the game, the player must reach a goal position, which will be a floor tile to ensure it is reachable.

Scoring

The player starts with a score of zero. Every move the player makes decreases its score by 1 point. Trying to move into a wall decreases the players score by 5 points, in addition to the one point lost from making a move. Once the player reaches the goal position, their score is increased by 100 points, and the game terminates. Because the score does not increase until the game ends, machine players must reach the goal position before they can learn a strategy to achieve a good score.

Game Variants

The way the game board is generated, and the placement of the player and goal positions, can greatly affect the nature of the game. I have used two different configurations which require different types of strategy with varying levels of complexity. For both variants, I have fixed the goal position in a single location at the bottom right tile of the board.

- **Single Path**

  This is the more simple variant of the game, in which a single path twists randomly around the board from the player's start position, to the goal position. The path never touches or intersects its self.

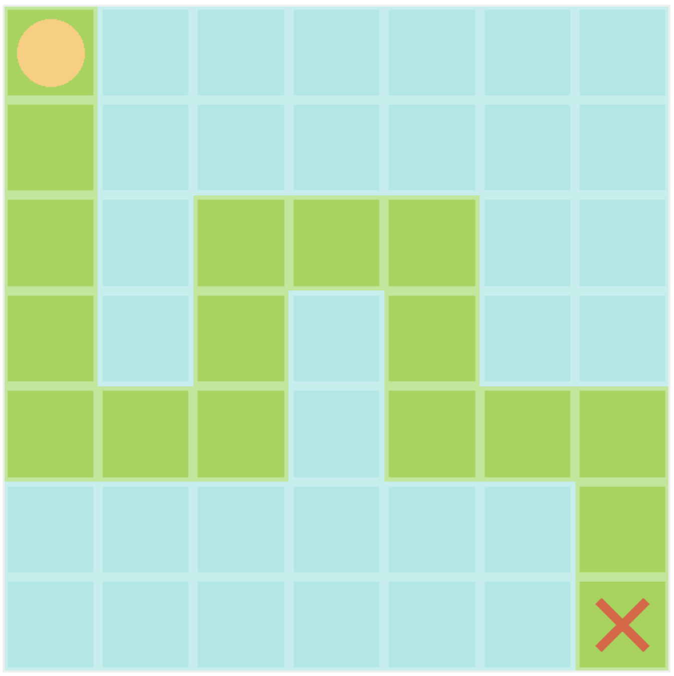

Figure 1: *Graphical interface for pathfinding game (single-path variant)*

- **Tree-Maze**

  In this variant, the map is generated as a complete maze with a tree-structure, meaning there is only one path between any two points in the maze.

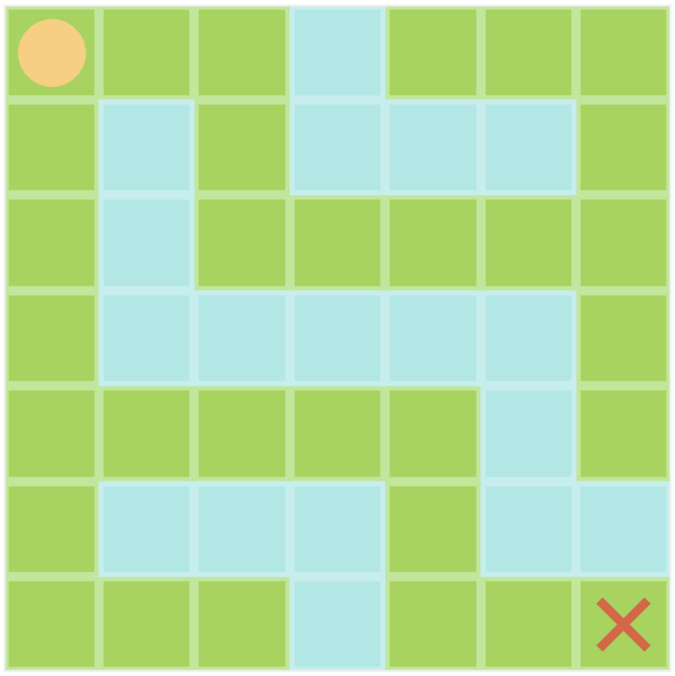

Figure 2: *Graphical interface for pathfinding game (tree-maze variant)*

The player starts in a random position in the maze. This variant requires a much more complex strategy than in the single path variant as the player needs to distinguish the single correct path out of potentially similar options. It also takes significantly longer to reach the goal by making random moves than in the single path variant. This slows down the training process.

Adapting for Two Players

I adapted this game for two players by assigning each agent its own game to play, but swapping their controls. This means the players are required to exchange information about each other's game states, or desirable moves. Note that swapping controls is exactly the same as swapping game states, since the players share a single score.

Configuration

For my experiments, I use a 7x7 board for both game variants. For the single-path variant, the player always starts in the top-left tile of the board, and the goal position is always in the opposite corner. The games terminate when the player reaches the goal position.

4.4 Number Matching

I designed this game specifically for two players. In this game, each player is presented with a set of numbers. Every turn, each player must toggle the activation state of one of these numbers (on/off). The best score can be achieved if the players reach a state where their active numbers sum to the same non-zero value. The numbers are generated such that there is only one state where this is possible.

I did not create a graphical interface with this game because of its numeric nature. A graphic interface would end up consisting mostly of numbers.

Scoring

The score is given out of 100, and is calculated using the binary representations of the sums of each players active numbers. The score represents the percentage of 1 bits in the two sums that are common to both. It is calculated by taking the number of bit positions where both sums contain a 1, divided by the average number of 1 bits between the two sums. This gives a value between zero and one, which is then multiplied by 100 to calculate the score. In the case where both sums are zero then the score is also zero.

Adapting for One Player

I adapted this game for one player by simply allowing the single player to take the roll of both players. The player can toggle a single number each turn. This number may be from either set. Scoring is calculated the same way as in the two-player version.

Configuration

For my experiments, I set the game to use 5 numbers for each player. The solution state always has 3 numbers on from each player. The games terminate when a score of 100 reached.

4.5 Alignment Game

This game was also designed specifically for two players. The setting of this game is an infinite track with multiple lanes. This forms a grid of tiles, where each lane is a row in the grid, and columns in the grid represent discrete steps along the infinite track. Walls are placed along the track at regular intervals. Each wall occupies and blocks an entire column except for one lane, where there is a hole in the wall. The position of the hole is random for each wall.

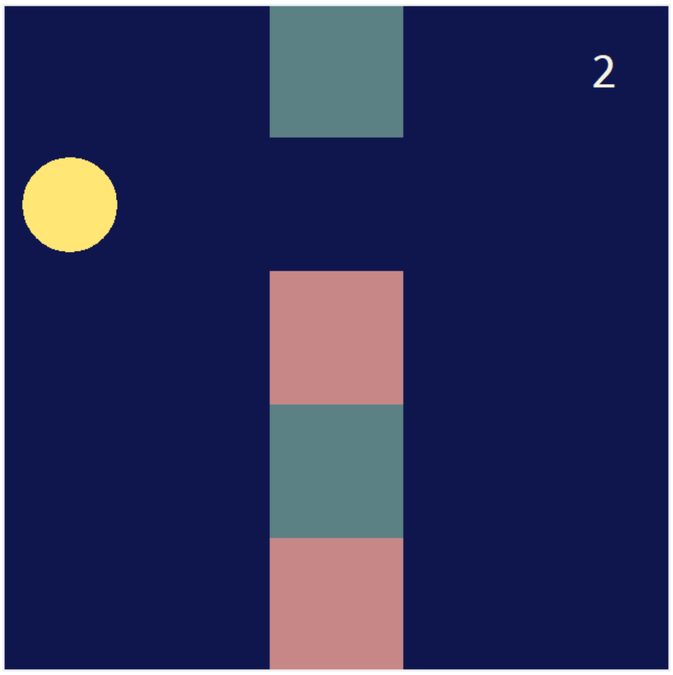

Figure 3: *Graphical interface for alignment game*

A single character is controlled by both players. The character advances one column along the track each turn, and additionally the players may move the character one lane up or down each turn. If the player advances into a wall tile, the game terminates, however they may pass freely through the hole in each wall.

Each wall tile can only be seen by one of the two players. This is chosen randomly for each wall tile. The players must combine their knowledge in order to identify where the real hole is located.

Each player has two possible actions they can take each turn. One player can move the character up into the lane above, or do nothing. The other player can move the character down into the lane below, or do nothing. If both players attempt to move the character, or if both players do nothing, the character will remain stationary. This means that both players must agree on where the character should move to ensure they both get the outcome they wanted.

Graphical interface

I create a graphical interface for this game, and a simple system for human control. The interface shows the game state, and updates each time the player makes an action. It also displays the current score in the top-right corner.

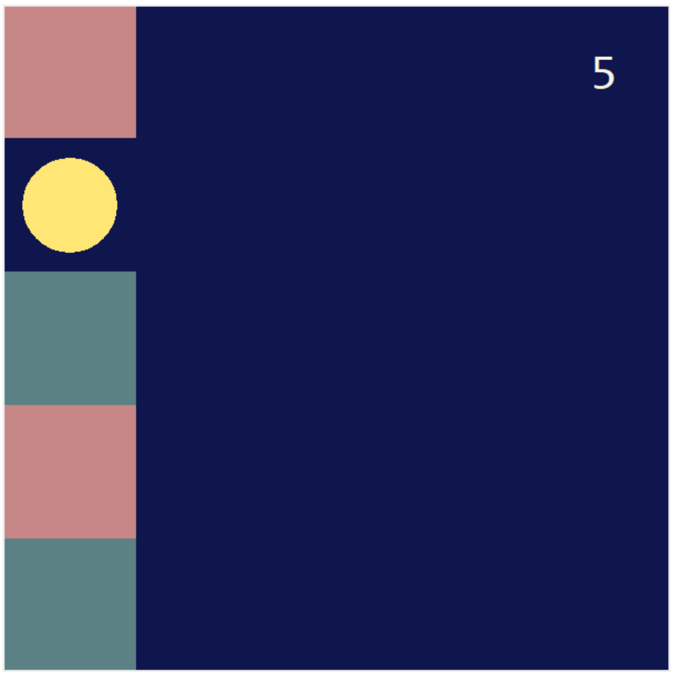

Figure 4: *Graphical interface for alignment game*

The red and blue wall tiles represent the sections of the wall seen by each player. The yellow circle is the character.

Scoring

The players start with a score of zero. Each time a wall is passed without dying, the players gain one point.

Adapting for One Player

I adapted this game for one player by giving the single player the information and actions of both players. The player has three actions to choose from instead of two. They can move up, move down, or do nothing. The player still needs to combine the two sets of information to determine the locations of holes.

Configuration

For my experiments, I set the game to use 5 tracks, and to place 6 spaces between each wall. The player can see the column they are in, and the 4 columns ahead of them in the track.

4.6 Implementation

For my implementation, I created a general-purpose interface for games. I implemented some games before I created this interface. For those games I created adapter classes which allow them to be interacted with through the interface, even though their underlying interface is different.

The interface is implemented in Python as an abstract class using the ABC standard-module, as Python does not support interfaces. Because Python is dynamically typed, any object matching the interface would work with my code, even if it does not explicitly extend the abstract class I have provided. The interface consists of 5 methods:

- **do_action(action)**

  Carry out the action specified in the action parameter. In two-player games, the action parameter should contain two values, representing the chosen actions of each player.

- **get_state()**

  Returns the current state of the game. The value returned is not necessarily the entire state, but rather a front-end, equivalent to a graphical display. Some parts of the state can be intentionally not exposed as part of a game. In two-player games, two states are returned, one for each player. The states could contain different information.

- **get_score()**

  Returns the current score. In two-player games, a single score is still used.

- **is_terminated()**

  Returns a boolean value representing whether the current state is a termination state. In two-player games, two values are returned.

- **reset()**

  This is a utility function, used to reset the game back to its initial state. This is often more efficient than creating a new game instance.

Additionally, all games can be cloned, using a method built-into the abstract class. Cloning creates an exact copy of it, with the same state and score.

5 Q-Learning

One of the most famous examples of reinforcement learning is the research carried out by DeepMind technologies into creating a general purpose algorithm which can learn to play Atari games, using only the score value and the pixels of the screen as inputs. In 2013, DeepMind technologies published the article [2] detailing their techniques. Their implementation was based off of a technique called Q-Learning.

Q-Learning systems maintain a table of $(state, action) \rightarrow result$ mappings called the Q-Matrix. This table is added to every time the system makes a move and observes its outcome. The outcomes are numbers representing the desirability of the resulting state relative to the previous state, based on some scoring system. The table is initially populated by taking random moves, then a model is trained to approximate outcomes based on random samples from the table. The system goes through cycles of randomly trying moves, and training the model with samples from the table. Once the model is accurate enough, it can be used to focus the random move choice onto favourable moves. This cycle is repeated until a desired prediction accuracy is reached for the model.

Rationale

One of the most notable things about the 2013 study is that a single algorithm was able to master a variety of different games, with the only game specific parameter being how the score was extracted. This sort of flexibility was one of my goals for this project. Furthermore the study already focused on games, so the techniques translated well into the context of the project.

5.1 Implementation Details

My implementation of Q-Learning comes in the form of a Python module. The module contains logic for playing through and learning one-player and two-player games, and facilitates communication in two-player games. The system interacts with games through a general-purpose interface [Implementation] meaning almost any game can be adapted to work with the system.

5.1.1 States and Actions

In the DeepMind technlologies study [2], the pixels of the screen from several frames are used to represent game states. Additionally, they used a fixed set of possible moves for all games, representing the inputs on an Atari 2600 controller. This meant that all games presented states in the same format, and all games accepted the same inputs.

In my implementation, there is no fixed format for game states. Instead, games use the most appropriate format. There is also no fixed set of inputs (actions), since the control schemes of the games are vary varied. Doing this smaller and generally faster models. Supporting different state-sizes and sets of actions makes the system more flexible.

5.1.2 Neural Networks

The underlying machine learning models in the module are neural networks.

TensorFlow

For the implementation, I used the TensorFlow Python library to model the neural networks. I used built-in TensorFlow functionality to calculate loss for the models, and gradients to train the network[Cross Entropy].

TensorFlow is the main reason I created my implementation in Python. While it supports other languages, Python is considered the primary language of TensorFlow. Since TensorFlow handles all of the demanding computations, the performance of python was not an issue.

Predictions

Like in the DeepMind study [2], my model takes a state as input, and produces a predicted q-value for each possible action. The other architecture I considered was to use a (state, action) pair as input, and to output a single Q-Value for the given action in the given state.

This alternative architecture would have more flexibility for actions. Specifically, it would be much simpler to represent games in which the available actions vary depending on the game state, such as chess. It would however be less efficient to predict Q-Values for every available action in order to find the best. I suspect it would also be more difficult to train, as the internal model would need to perform drastically different calculations based on which move was input.

Cross Entropy

In my system, I use cross entropy to measure the loss during training. This value is minimised in order to maximise the accuracy of the system. Cross entropy is an unintuitive value in its raw form, however it makes a good loss value for classification problems [1].

It takes two sets of values, y and $\hat{y}$, where y represents actual values and $\hat{y}$ represents predicted values from a model. If both sets are considered as probability distributions for the same set of events, then cross entropy is the average number of bits needed to represent an event from y, if you use a coding strategy which is optimised for probabilities in $\hat{y}$.

For a given y, the maximum achievable cross entropy for $y, \hat{y}$ is achieved when the two distributions are equal. This maximum value is equal to the average number of bits needed to represent an event from in y, given the optimal encoding coding strategy. This means that in games with more possible actions, and where expected values for actions are closer together, the minimum achievable loss is higher than in simple games such as the XOR game.

The values in y usually reflect real-life uncertainty in classification training data, and so the minimum uncertainty achievable by the model is equal to that present in the original data. It actually functions similarly for q-values however. My training data comes from the q-matrix which is built from observations. This process is inherently random. The values in the q-matrix actually represent the estimated future-payoff of a given action. If two expected payoffs are similar in size, then there is uncertainty present in the system, regarding which is better.

If we have the prior knowledge about a game that there is no randomness, and the state space has already been fully explored, then we know that the highest q-value is actually 100% likely to correspond to the best action. The system however, cannot know that the state space is fully explored, or that there is no randomness present in the game. As a probability distribution, q-values represent the likelihood that a given action is the best available action.

Additionally, on a more basic level, cross-entropy uses the differences between numbers rather than their absolute values. This is beneficial, since the system only needs to predict the relative merits of different actions. Removing an unnecessary constraint from the desired output leads to faster training.

Another benefit is that it is fairly simple to train with partially explored states (where not all actions have been explored). In my system, I pass in a binary mask along with my training data, where each value represents whether an action has been explored. I multiply (element-wise) the mask with my training data and predictions before calculating the loss. Setting both values to zero increases the minimum possible cross entropy, but it does not change the values required for that minimum. This means the zeros are effectively invisible to the optimiser, and the system will still converge exactly the same way. This is also true for loss calculations such as RMS, but I chose to mention it because it is an important feature for my system.

I use TensorFlow's in-built implementation of the adam optimiser to optimise the networks. I set up the optimiser to minimise the cross-entropy loss value.

Architectures

I have encapsulated the underlying neural networks as Python classes called 'Agents'. The module defines two types of agent, FeedForward and Convolutional. These represent two different underlying neural network structures.

- **FeedForward**

  Feedforward agents use a simple layer-based architecture, where each layer is the result of a matrix multiplication with the previous layer. The number of layers and the size of each layer can be configured.

  Intermediate nodes use the Tanh activation function, while input and output nodes use linear activation. The Tanh function is commonly used in neural networks. It maps all values into the -1 to 1 range. Linear activation means each value is mapped to its self.

  For all games, I used 4 intermediate layers with $2x$, $3x$, $4x$, and $5x$ nodes respectively, where x is the number of nodes in the input layer, and also the number of values in the game state of the game being played.

- **Convolutional**

  Convolutional agents are based on convolutional neural networks, the type used in the DeepMind study [2]. The agents are comprised of a series of filter layers, followed by a series of fully-connected layers. These are all standard features of convolution neural networks. The size of filters, and the number and size of the fully connected layers can be configured.

  The fully connected layers use the RELU activation function, and the input and output nodes use linear activation. RELU (Rectified Linear Unit) is a simple function that maps values greater than zero to themselves, and all other values to zero.

  For the pathfinding game and the alignment game, a 5x5 filter layer is used, followed by 3x3 filter layer. The XOR game and number matching game have smaller game states, so they both use a single 2x1 filter layer. The outputs are then flattened to feed into the fully-connected layers.

  The alignment game and the number matching game use 4 fully connected layers with $2y$, $3y$, $4y$, and $5y$ nodes respectively, where y is the number of nodes output from the last filter layer. The last two layers are excluded in the XOR game, as in rare cases they dramatically slowed down training. For the pathfinding game, only 2 fully connected layers are used, with y, and $2y$ nodes respectively. This was to ensure the network would fit in the memory of my GPU.

- **Double Variants**

  Each agent type also has a double variant, for two-player games. These variants have two separate underlying neural networks with a communication link connecting them. On each network, a series of standard feedforward layers are added, branching off from the main structure. These layers are dedicated for communication, with the last layer representing the communication output to the other agent. This last layer uses linear activation, while the other layers use the same activation function as the intermediate nodes of the agent.

  The incoming communication nodes from the other agent are concatenated with the layer from which the communications branched off, and the network continues as normal from that point.

On each agent type, the number and size of the communication layers can be configured. For the feedforward agent, the layers before and after the communication link can be individually configured, meaning the communication layer can branch off from any desired layer in the network. For the convolutional agent, the communication layer is always located after the filter layers.

For double feedforward agents, 3 pre-communication layers are used, with x, $2x$, and $3x$ nodes. Next, 3 communication layers are used (excluding the communication output layer), with $3x$, $4x$, and $5x$ nodes. Finally, 3 post-communication layers are used with $3x$, $4x$, and $5x$ nodes. For the XOR game, the first pre-communication layer, the last communication layer, and the last post-communication layer were all excluded as they dramatically slowed down training in rare cases. The last communication layer and post communication layer are also excluded for the pathfinding game because they made the networks too big fit in my GPU memory.

Double convolutional agents use the same filter layers and fully-connected layers for each game as their single counterparts. For all games, 2 communication layers were used (excluding the communication output layer), with $2y$, and $3y$ nodes respectively. The pathfinding game was the exception to this, using only y and $2y$ nodes instead. Again, this was to limit the memory footprint of the network.

5.1.3 Communication Bandwidth

My module allows me to vary the number of values communicated between agents at each step. I call this the communication bandwidth. Different games may have different requirements for communication bandwidth. For example, the XOR game only requires a bandwidth of one value to achieve its peak accuracy. A bandwidth of zero would result in the system being unable to train past 50% accuracy, while increasing the bandwidth above 1 would result in no improvements.

A communication bandwidth of 4 was used for the pathfinding game, one for each possible move. 5 was used for the alignment game, as there are 5 tracks. 5 was also used for the number watching game, as each player has 5 number options. I picked these values, as I believed they would be sufficiently large for each game.

5.1.4 Exploration and Training

This part of the system has a major impact on training time. Because of this, I did a lot of experimenting to find a good balance between random and prediction-guided exploration.

Scheme

Exploration and training are two separate processes in my implementation. It is possible to interleave the two in a way that is tailored to a specific game. This is useful for testing, however to maximise flexibility I wanted to settle on a fixed pattern that worked for all games. I did not succeed at this goal. There are several exceptions required for specific games.

The system always rotates between playing through ten games, and then training on 100 samples from the Q-Matrix. One sample consists of every observed outcome for a single state. This is repeated until a desired prediction accuracy is reached. If there are not 100 samples in the Q-Matrix, then the program uses all of the available entries from the Q-Matrix to get as close to 100 samples as possible. This occurs in the first few rounds for most games. If a game has less than 100 possible states, for example the XOR game, then this will be the case for all rounds.

Simultaneous Games

Using TensorFlow allows me to run neural networks on my GPU. The highly parallel nature of GPUs meant that I could gain a lot of efficiency by passing multiple states into my network at the same time, and generating multiple predictions. The flipside of doing this is that the model gets updated less frequently. In general, the larger the branching factor for possible states, the more beneficial it is to run through multiple games before training. If the state space is small, then the games are less likely generate new and useful experience. In these cases, it is better to maximise training to improve the quality of predictions and unlock new possibilities.

Focusing Exploration

The biggest problem I ran into was caused by prediction-guided moves in the pathfinding game. To gain any useful experience playing through the pathfinding game, the agent must reach the goal location. To do this, the agent has to follow the single available path to the exit. If this path contains any states for which the agent does not predict the correct action, it can become unable to continue down the path, which halts the whole training session. The standard approach is to introduce a fixed chance to make a random move at every step. This makes it technically possible to escape these situations, however the expected time to overcome sequences of these states still increases exponentially with the length of the sequence.

The best solution I found was to start the exploration with some base chance to use a prediction, and the to decrease this chance by a factor every time a prediction is used. In games which are taking a large amount of turns, the exploration becomes almost entirely random. This means that more random moves are made in states that the model is not yet suited to.

To further this effect, I used the average accuracy of recent predictions to determine the decrease-factor. If recent predictions were 90% accurate, then the chance will decay by 10% for every move taken based on predictions. This brings about another benefit. In many games, the space of possible states increases exponentially with each move. The certainty of predictions can decrease over time as it is becomes likely that the player will encounter a new situation. The higher the accuracy of the model, the further into the game it should be able to make accurate predictions. Focusing on predictions in the earlier moves, means that exploration becomes focused in the areas where prediction accuracy is lowest. As the prediction accuracy increases, the focus shift towards later states, and away from earlier states that have already been fullt grasped.

I used this approach for the pathfinding, and the number matching game, but it was not suited to the other two games. For the XOR game, rules that span multiple turns were not applicable, so I stuck with fully random exploration. This was sufficient because the number of possible states is so small that they can all be explored with no issues. For the alignment game, the scheme was too random. Wrong moves have a high chance of causing the player to hit a wall, terminating the game. I could have decreased the decay rate, and the initial random chance, however I found that using a fixed 20% random move chance was sufficient. Additionally, I implemented a turn-limit into the learning system, specifically to prevent the alignment game from running on too long, when the model became sufficiently accurate.

5.2 Experiments and Evaluation

Accuracy Metric

As stated before, I use cross entropy to measure loss for my predictions [Cross Entropy]. Recall that cross entropy can be described using y as the actual expected values for each action, and $\hat{y}$ as the predicted values for each action. y and $\hat{y}$ can be thought of of probability distributions, where each value represents the probability that the corresponding action is the best available action in the current state. The minimum achievable loss represents the uncertainty in the y.

A more immediately understandable accuracy value can be derived from cross entropy, using the following formula:

$$accuracy = e^{-cross\_entropy}$$

This equation gives an accuracy value between 0 and 1. An accuracy of exactly 1 represents zero loss, which means y and $\hat{y}$ contain only one value, while values close to 1 can be achieved if y generally contains a single large value for each game state. An accuracy of close to zero means that loss is extremely high, and so either y as a probability distribution is very uncertain, or y is very badly approximated by $\hat{y}$. If a model outputs uniformly distributed random values, it would achieve an average accuracy of $\frac{1}{n}$ where n is the number of values in y (also the number of possible actions).

Note that the maximum achievable accuracy is still limited, in the same way that the minimum achievable cross entropy is limited. This limit represents the certainty that the highest value in y is the best available action. The system inherently cannot be 100% sure about which is the best action, because it cannot know if games have a random element, or if all possibilities have been explored. Uncertainty is naturally always higher in games with a large number of possible actions, and games where many of the best actions have similar payoffs.

This means that often, the system will report a fairly low accuracy, when in-fact it has learned to make the correct move in 100% of cases. It is more important then, to look at *when* peak accuracy is reached, rather than the actual value of the peak accuracy. In two-player games, accuracy is averaged between the two agents, and each agent still has the same set of actions as in the one-player equivalent (excluding the alignment game). This means that the relative uncertainty of moves between the two versions of a game can be accurately judged by comparing the peak achievable accuracies.

Finally, the accuracy formula behaves unintuitively when it is used for two-player games. Since loss is averaged between the two agents *before* accuracy is calculated, the accuracy value appears at the logarithmic midpoint between the two accuracy values, rather than the true midpoint. This means that when both agents are not achieving similar accuracy levels, the overall reported accuracy is slightly lower than expected. This does not affect training, but it occasionally shows up on the graphs in this section. I have pointed it out wherever necessary.

5.2.1 XOR

The XOR game mainly functions as a control to compare other games to, as well as a sanity check to prove the system is properly functioning. As expected, the Q-Learning system quickly learns the XOR function. This means that the ideal models could be represented within all agent types, and that the training system effectively converged on those representations.

The graph below shows accuracy over the course of 5000 training rounds on one-player and two-player XOR games for both agent types.

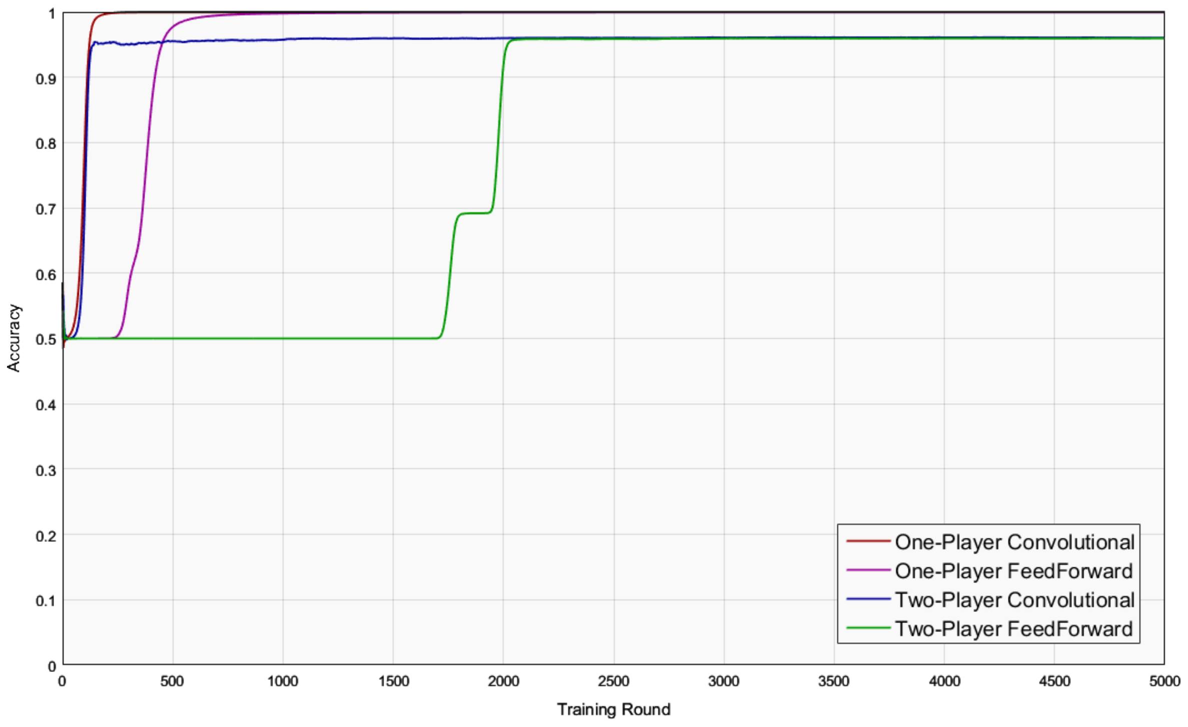

Figure 5: *Combined XOR results*

The first thing to note is that every agent started the session at 50% accuracy. This is because if you randomly guess actions for the XOR game, you have a 50% chance of guessing the best action each time, because there are only two options. Achieving below 50% would mean that the models were optimising in the wrong direction.

On the graph, you can see that in the one-player version of the game, both agent types achieved near-perfect accuracy. In the two-player version, both agent types plateaued at around 0.96. The reason for this is that in the two-player version, the expected values of the best actions are lower, because of uncertainty introduced by the other player's inputs. This increases the minimum uncertainty within the q-matrix, since it never assumes it has fully explored every possibility. Therefore, the maximum achievable accuracy is capped. All models still correctly predicted the best move for all possible states.

Both of the convolutional agents reached their peak accuracies at almost the same time. The change from one-player to two-player seems to have had little effect on the optimisation speed. The opposite is true for the feedforward agents. The two-player feedforward agent took significantly longer to reach its peak thank its one-player counterpart. I am not sure why the performance varied in this way between the two network types, but it is clear that convolutional networks are better suited for two-player games.

For both the one-player and the two-player versions, the feedforward agents took longer to reach peak accuracy than their convolutional counterparts. The convolutional agent has two benefits over the feedforward agent. First of all, filter layers can take advantage of correlation between spatially close inputs. This is not important for the XOR game, but it is for some other games. Second, the convolutional agents use much bigger networks than the feedforward agents, because filter layers result in a large amount of nodes. This could mean that some optima have larger neighbourhoods of possible similar representations, and therefor can be reached more directly through optimisation.

The final notable feature is the temporary plateau reached mid-way through training by the two-player feedforward agent. The underlying loss value of the step is approximately half way between the minimum and peak values, but this is adjusted into exponential space for accuracy so it does not appear exactly in the middle. At this point in the training, one agent had learned to correctly pick the best actions based on communications, but the other had not, so their true average accuracy was closer to 0.725. Once the other side of the communication was correctly learned, the agent overcame the plateau.

Communication Bandwidth

For the XOR game, I also tested the communication bandwidth. I did this to illustrate the importance of the communication link. For every game-type, I picked a bandwidth that I knew would be sufficient, based intuition about successful strategies for each game. I did not attempt to find minimum required communication bandwidth for any other game types, as the focus of the project was the viability of the agents, rather than identifying specific information about the games

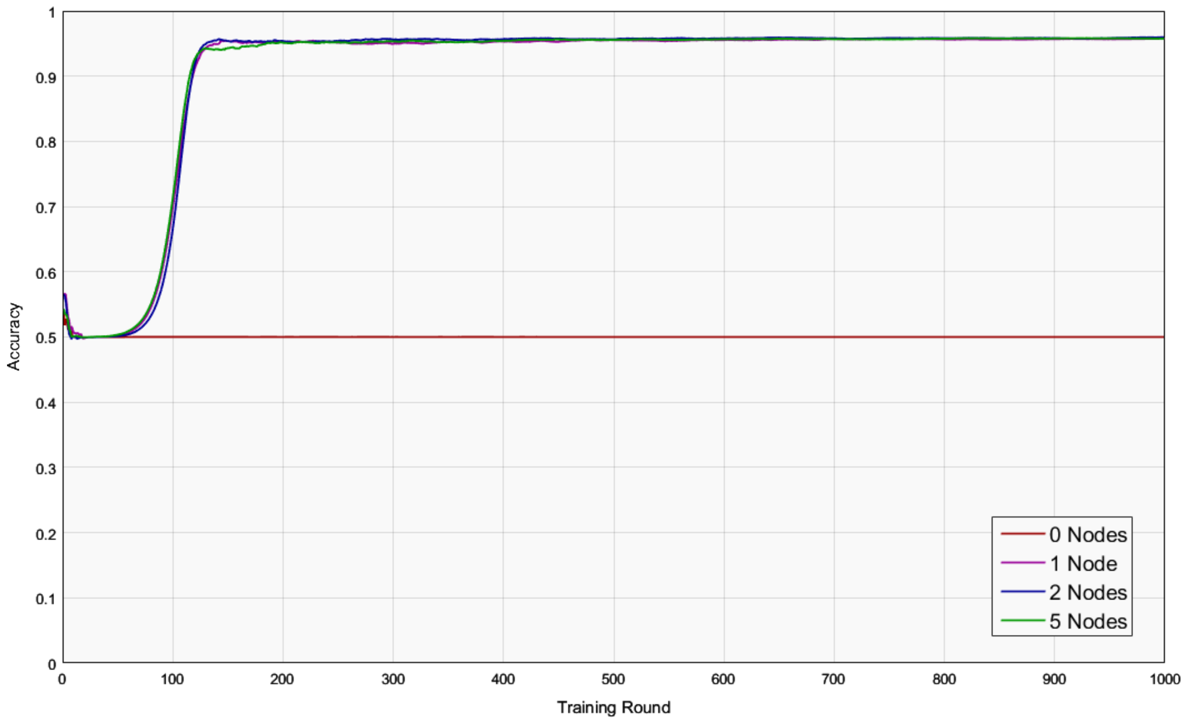

Figure 6: *Two-player XOR game using convolutional agent with varying communication bandwidth*

As expected, the agent remained stuck with effectively random guesses, when given a communication bandwidth of 0. A communication bandwidth of 1 was sufficient to communicate all critical information. It is clear from the graph that once this threshold is reached, performance remains constant. This differs between game types, and depends on the amount of critical information which needs to be exchanged each time step.

5.2.2 Pathfinding

The graph below shows accuracy over 5000 rounds of training for all four agent types, each playing the **single-path** variation of the pathfinding game. Both agent types were able to quickly reach near-perfect accuracy in the single-player version of the game, but similar to XOR, they were noticeably limited in the two-player version.

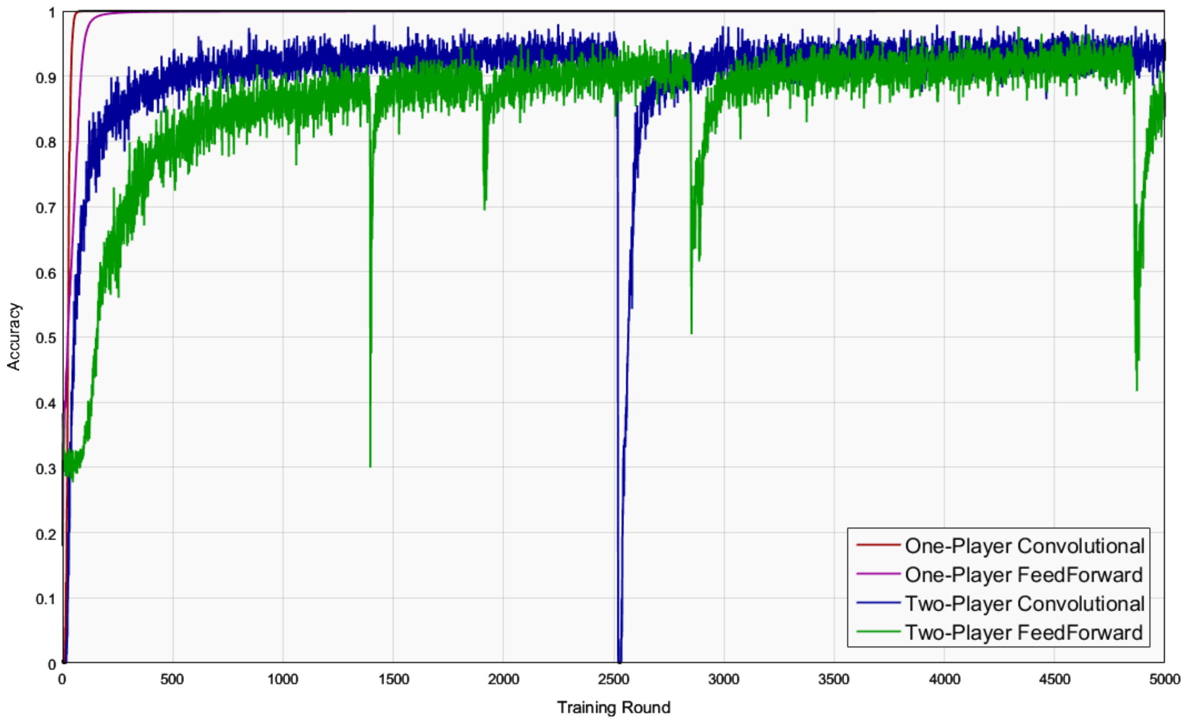

Figure 7: *Combined pathfingind game results (single-path variant)*

The next graph shows accuracy over 5000 rounds of training for all four agent types, each playing the **tree-maze** variation of the pathfinding game. Similar to the previous graph, both agent types excelled in the single-player version of the game, but had slightly limited accuracy in the two-player version.

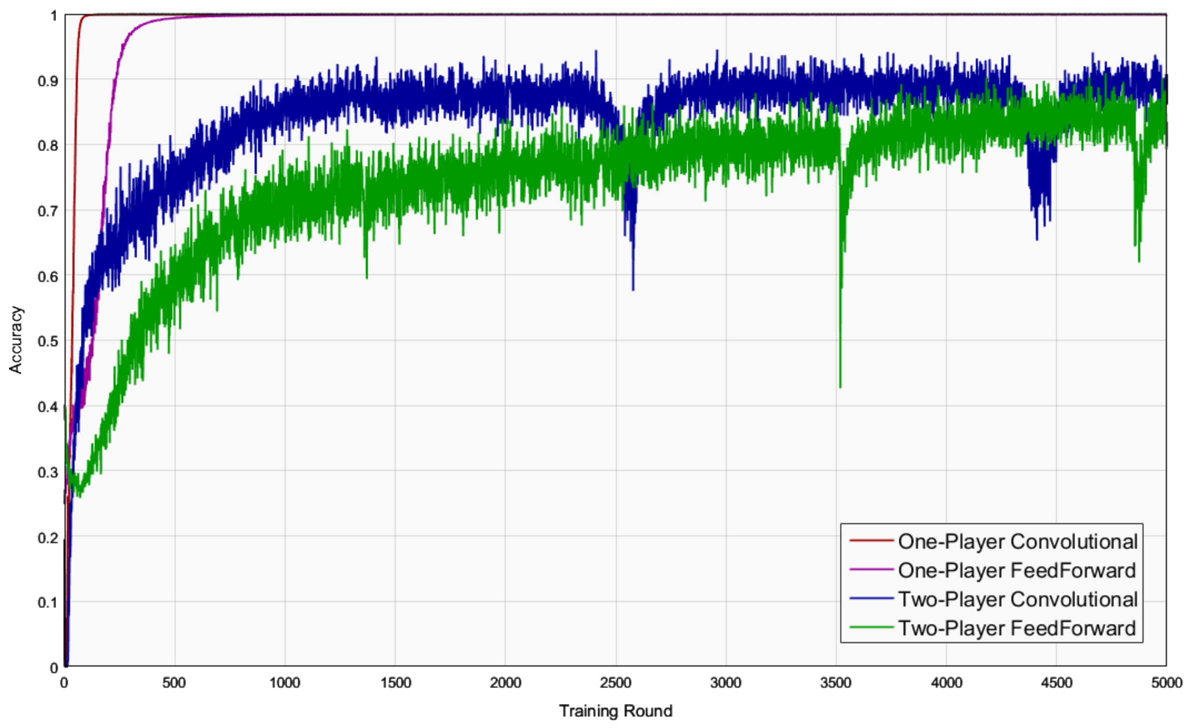

Figure 8: *Combined pathfinding game results (tree-maze variant)*

The two graphs are very similar. The most notable difference between them is that each agent-type took slightly more rounds to reach peak accuracy in the tree-maze variant, as if their lines had been stretched slightly along the x-axis. This is expected because the game-states are much more complex in the tree-maze variant, as they have additional paths that do not lead to the goal position.

Accuracy Troughs

The two-player versions in both graphs feature occasional accuracy troughs of significant size. Not only are they present in both graphs, but many of them appear similar in shape and location between the two graphs. The troughs are noticeably smaller for the tree-maze variant however. I cannot figure out exactly why these troughs occur, though I expect it is something to do with the adam optimiser used for the training. It is possible they represent discrete pivot points between strategies for the game, and the optimiser is naturally leading the networks between different the strategies.

Noise in Two-Player Games

The most striking feature of the graphs is that the accuracies for the two-player versions of the game were extremely noisy compared with the one-player versions. I expect that this is due to increased noise in the q-matrix because of the value of an action is affected by the action the other player takes.

Accuracy vs. Real-World Performance

The high accuracy of the one-player agents in both graphs is not completely reflected in their practical success rate. Due to the nature of the game, even small levels of inaccuracy can totally prevent the player from reaching the goal tile. In both variants of the game there is only a single path to the goal tile. This can be though of as a sequence of states the player must be in before they can reach the goal. If the player takes the wrong action in any of these states, then they are totally prevented from progressing further down the chain of states to the goal, as they will always make the same mistake again.

To frame it another way, say an agent reaches 99% accuracy, and the average distance from the starting point to the goal is 15 tiles. The average chance of the agent making the correct move every step is $0.99^{15} = 0.86$ That means the agent would still fail to reach the goal in 14% of games.

In reality, any mistakes by the agents are very minor in terms of the actual relative payoffs predicted for each action. I expect that introducing a small amount of noise into the inputs would allow them to overcome the majority of these mistakes after a few turns.

In contrast to the one-player results, the two-player agent's practical performance is more in line with their reported accuracy. I believe this is because the communication channel acts as a form of noise between the two players. If one becomes stuck along its path to the goal while the other is still making progress, the different inputs coming through the communication channel introduce a small random element to the predictions at each time step. This frequently allows the players to overcome obstacles they would otherwise be stuck on. You can see this in action when the agents play through a game. They alternate between having players stuck at points along their paths, but only rarely do they both get stuck at the same time.

5.2.3 Alignment Game

The graph below shows accuracy over 5000 training rounds on one-player and two-player versions of the alignment game.

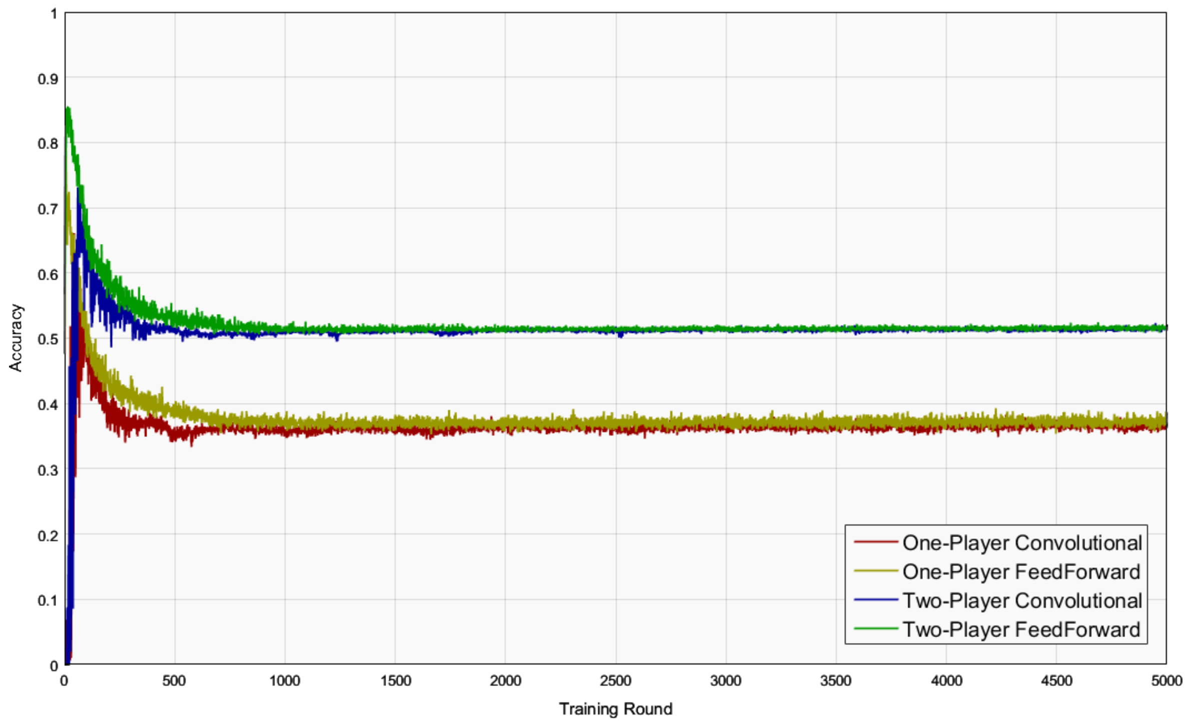

Figure 9: *Combined alignment game results*

It is immediately clear from the graphs that the Q-Learning system achieved a very poor accuracy in this game, for all agent types. In the one-player version, accuracies of around 37% were achieved, where the agents had three possible actions each turn. In the two-player version, an accuracy of around 52% was achieved, where each agent had only two possible actions to choose from. Interestingly however, the real-world performance of the agents is extremely good in all cases. This suggests that the game has a lot of innate uncertainty. This is likely true, as the game has a heavy random element in where the hole is generated in each wall. Additionally, in the configuration I used, there is some breathing room between walls, in which the agent's actions are relatively unimportant. This leads to a large number of actions with similar expected payoffs, and so a large uncertainty in which action is the best choice.

The biggest takeaway from this is that the innate uncertainty in the Q-Learning method is not a hindrance. Instead, the system is able to converge on correct values, even when random elements are present in the game. This graph, along with the models practical performance, illustrates that the system can observe noisy, uncertain data, and find underlying patterns to converge on an effective strategy.

5.2.4 Number Matching

The graph below shows accuracy over 5000 training rounds on one-player and two-player versions of the number matching game.

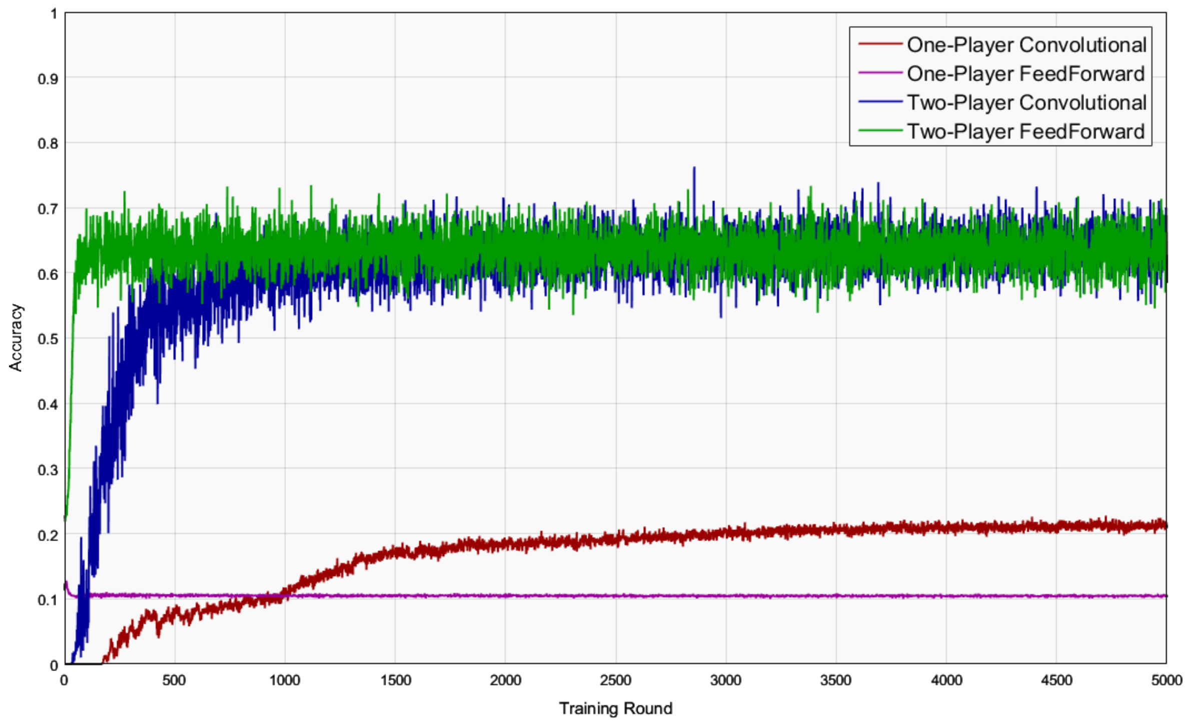

Figure 10: *Combined number matching game results*

It is clear from the graph that results in this game were by far-the worst out of all of the games.

Highly Connected State Space

All versions of the game suffer from the opposite problem as the pathgame. The state space is too highly connected. There are many different paths to reach the optimal state, with many having no overlap in intermediate states at all. Furthermore, there is a second local optimum present in the state space, in the exact opposite state (all options inverted) to the global optimum. All of this amounts to too much uncertainty for any sort of reliable prediction. This problem could potentially be alleviated by training many separate agents, then taking a consensus opinion on the best action.

The openness of the state space also causes the majority of agents to become stuck in cycles of sub-optimal states. Furthermore, this also occurs in the two-player game, unlike in the pathfinding game, where players are less likely to get stuck in the two-player version. Normally, I would suggest introducing some noise into the system, however when the uncertainty is so high already, this could push the system towards total randomness. Technically, this still would be more effective strategy than getting stuck in cycles, as the agents currently do, because you do eventually reach the optimum state by random chance.

One-Player Performance

This is the only game where agents performed worse at the one-player version. I think this is mainly due to the way I adapted the game. I gave the roles of both players to a single player, however I only allowed the player to toggle one number option each turn. In the two-player version, two options get toggled each turn, one by each player. This means a lot of new possible states are introduced in the one-player version of the the game. This also means the chain of actions required to reach a perfect score is twice as long. Additionally, the player has twice as many actions to choose from each turn, because they can toggle both sets of number options. This all adds up to significantly higher uncertainty than in the two-player version.

The peak accuracy for the one-player convolutional agent is more indicative of the minimum uncertainty level than the line of the one-player feedforward agent. It is worth noting that the one-player convolutional agent consistently place a high value on the best actions, but the amount of noise was too high to achieve reliable predictions. Additionally, it appears that its accuracy was still gradually improving, even in the final training rounds.

It appears that the one-player feedforward agent was not able to find any patterns in the noisy data. It failed to reach an accuracy far above 0.1, which is the random-strategy accuracy for this game, since the agents had 10 actions to choose from each round.

Two-Player Performance
Another notable result is that the two-player feedforward agent actually performed better than its convolutional counterpart. I expect this is because the convolutional agent assumes some correlation between spatially adjacent values in the game state. There is no such correlation in this game. It may be that the game has high enough uncertainty, that the additional noise added by this false assumption has a visible impact on the gradient calculation.

Both agents for the two-player version of the game achieved their peak accuracies relatively quickly, however both suffered from extremely poor real-world performance, as every agent did. I expect that with enough rounds of exploration, the signal-to-noise ratio in the Q-Matrix would increase, and the agents would find valid gradients. A training scheme more focused on exploration would have been much better suited for this all versions of this game. I can only conclude that using the same training scheme for such a variety of games is counterproductive, despite my want for a one size fits all system.

5.3 Conclusions

It is clear that joining two agents via a communication link, and training them as one unit, is a viable technique. It could potentially remove unnecessary complexity in systems by removing the need for hand-picked exchange protocols or data coding schemes, and centralising training into one place.

Two-agent systems pair well with Q-Learning, however some system specific considerations should be made to avoid performance issues. Combining agents increases uncertainty in classification problems, however Q-Learning still manages to offer good real-world performance.

6 Genetic Algorithm

Rationale

For my second implementation, I decided to try an approach based on genetic algorithms. One reason for this is that evolution-based optimisation is significantly different from the gradient descent optimisation used in my Q-Learning implementation. One of the benefits of genetic algorithms is that you do not need to use any gradient calculations to determine how to tune your models.

For this system, I wanted to focus on flexibility. In general, genetic algorithms are known for being effective at solving complex problems, but being computationally intensive compared to other optimisation techniques.

6.1 NEAT

I chose to base my implementation on a genetic-algorithm system called NEAT described in this paper Evolving Neural Networks through Augmenting Topologies [4] from 2002. The title feature of NEAT is growing neural network structures over time, starting from a set of minimal networks. The network structures are randomly augmented each generation, and the best augmentations are kept by a process of fitness-based selection. Each network is represented as an individual in the population, and each generation, the new population is generated by applying a variety of genetic operators [Genetic Operators] to members of the previous population. My implementation closely follows the system described in the NEAT paper.

6.1.1 Networks

Networks consist of nodes and edges. Every network contains the same number of input nodes, and the same number of output nodes, specified while initialising the system. Additionally, every network has a single fixed input node with the value of 1. This replaces the bias value on edges in a typical neural network, as edges from this node will add a fixed amount to the value of their end-node, based on the weight of the edge. Every network in the initial population has the same fully connected structure, with edges from every input node to every output node.

6.1.2 Edges

Edges are represented globally by an index. Each networks holds two values for each edge it contains: the weight of the edge; and whether the edge is disabled (called the disabled value).

If an edge is disabled, it is excluded from the network, but it is still remembered (including its weight value). A disabled edge can no longer have its weight mutated, and is no longer used for fitness evaluation. It can still however affect speciation, and offspring created with the cross operator.

6.1.3 Nodes

Nodes are represented globally with their own index system, separate from edge indices. No network-specific information is tied to nodes, as it is for edges. The nodes contained in a network can be inferred from the edges, however for practically each network still maintains a list of the nodes it contains.

Nodes are arranged into a global list called the dependency order. This is the order in which the values of nodes are resolved when data is being fed through networks. Naturally, the input nodes are always at the start of the list, and the output nodes are always at the end. A node may not have incoming edges from any nodes after it in the dependency order. This feature is not an integral part of NEAT, and is more of an implementation detail to prevent cycles in the graphs.

6.1.4 Mutation Indexing

As previously mentioned, each time a new network feature (a node or an edge) is randomly mutated, it is assigned a global numerical index. In the NEAT paper, these are referred to as innovation numbers. If the same feature is independently mutated multiple times, the same index will be assigned. This is achieved by keeping a record containing the structural definition for each index.

Keeping track of mutations makes it easy to find common features between two networks. This makes it easy to perform speciation, and enables the cross genetic operator [Cross].

6.1.5 Genetic Operators

Genetic operators are applied to individuals or groups in a population, in order to generate offspring. The goal of genetic operators is to create individuals similar to the parents, but with minor differences. All of the specific values in this section were taken from the NEAT paper [4], with the exception of the random normal values which I determined experimentally.

- **Carry Forward**

  This is the simplest operator. It copies the individual without applying any other genetic operators. Any species with more than 5 members has its fittest member carried forward into the next generation. This protects the best graph features from being randomly discarded, and is a common feature of genetic algorithm systems.

- **Weight Mutation**

  This operator randomly mutates the weight values of edges. For each edge, a random normal value is generated with a mean of zero and a standard deviation of 0.5. For 90% of edges, their weight is offset by their corresponding value. For the other 10%, the value replaces their weight.

  The magnitude and chance of weight mutations are chosen as a compromise between two effects. The higher the values, the more quickly weights globally converge on optima, but the more erratic individual weight values become. More erratic weight values makes it less likely for optimal combinations of values across multiple edges to occur, as less of the edges which are at optimum values, maintain those values.

  Offspring have an 80% chance to be affected by the weight mutation operator. Weights are only mutated for non-disabled edges.

- **Add Edge**

  This operator adds a new edge to the offspring individual between two nodes that were not previously connected. The end node of a new edge is always later in the dependency order than the start node, to prevent cycles in the networks. The weight of a new edge is a random normal value with a mean of zero and a standard deviation of 0.5.

  Offspring have a 5% chance to be affected by the new edge operator.

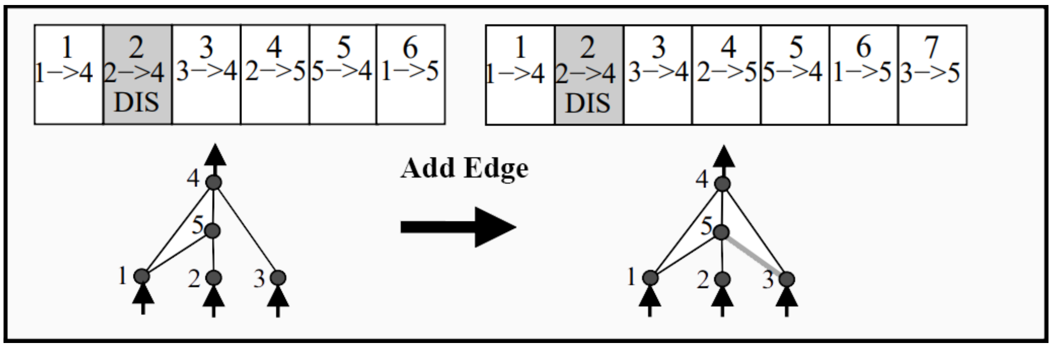

Figure 11: *Add edge genetic operator (original from NEAT paper [4])*

- **Add Node**

  This operator adds a new node to the offspring network. A new node is inserted by choosing an edge in the network, and replacing it with two new edges and a node, as shown in the diagram below. The weights of the new edges are random normal values with a mean of zero and a standard deviation of 0.5. The edge being replaced is not removed from the graph. Instead, its disabled value [Edges] is set to true.

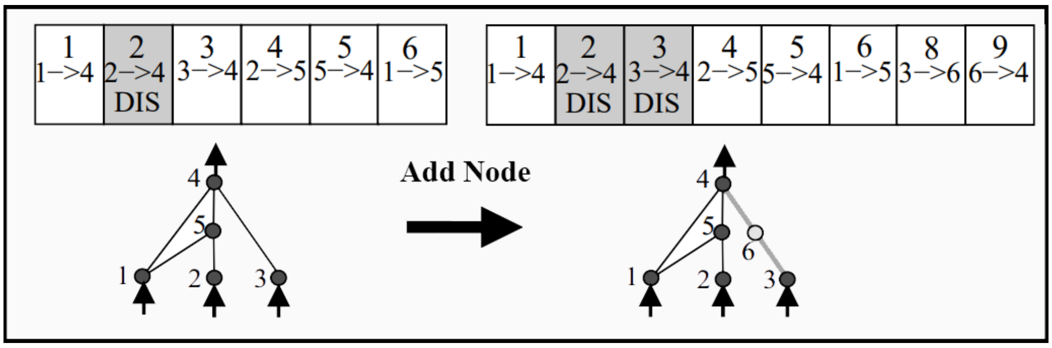

Figure 12: *Add node genetic operator (original from NEAT paper [4])*

The new node is placed into the dependency order at the midpoint between the start and end nodes of the original edge. This gives a roughly equal chance on average for nodes to be inserted before or after a new node. Other placements would cause new nodes to tend towards one end of the graph.

Offspring have a 3% chance to be affected by the new node operator.

- **Cross**

  Genome crossing is a common operator used in genetic algorithms where you combine two individuals to create one or more offspring individuals, each of which is some mixture of the two parents.

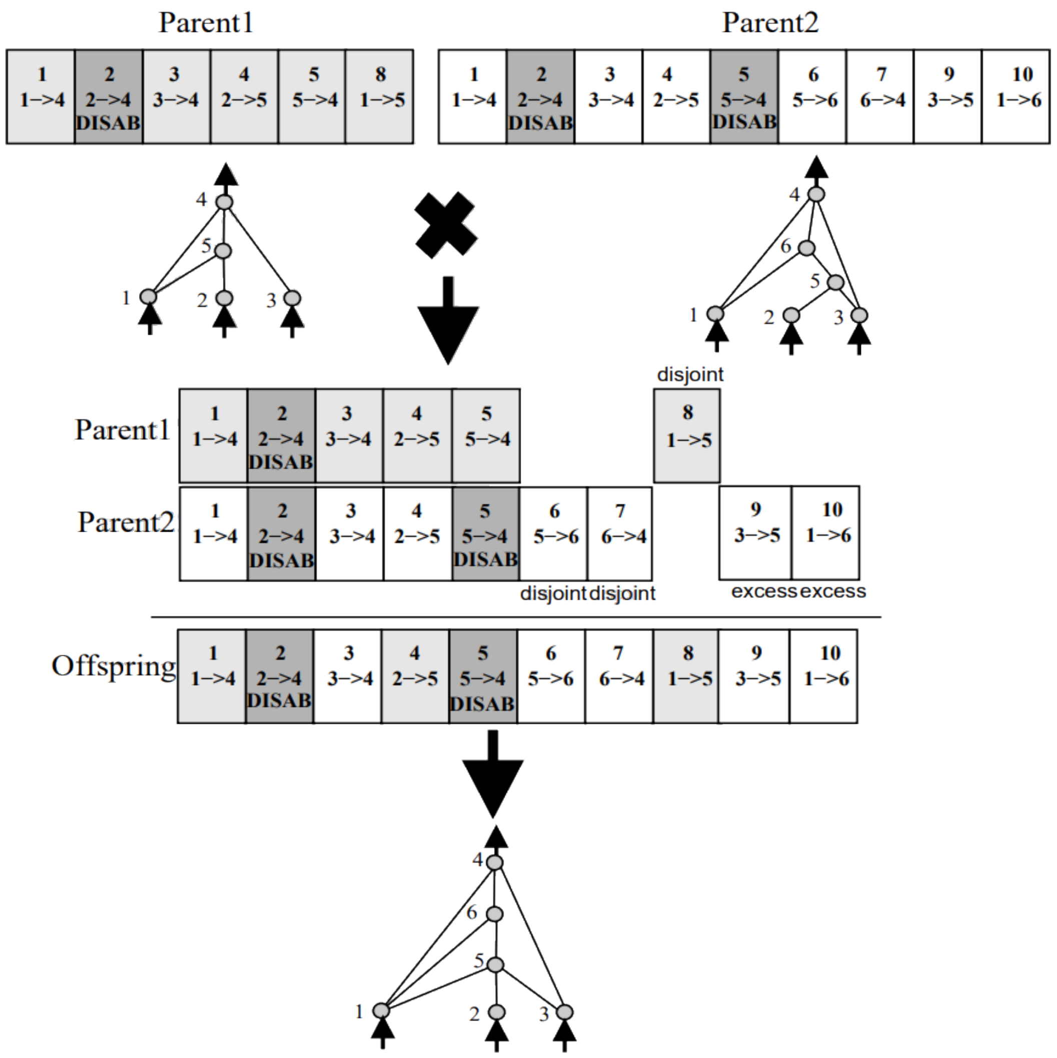

Figure 13: *Cross node genetic operator (original from NEAT paper [4])*

In NEAT, the two parents are almost always chosen from the same species. In 0.1% of cases, the parents are chosen from different species. The fitter parent is used as a baseline, then each edge common to both parents has a 50% chance to inherit its weight from the other parent. Next, all edges in the offspring have a 75% change to be disabled if they are disabled in either parent. This means that edges that do not positively affect fitness are likely to become disabled over time.

Offspring have a 75% of being created using the cross operator. The offspring created using the cross operator are still subject to other operators.

6.1.6 Speciation

Speciation is used in neat to separate groups of significantly different networks within a population. If networks are too different, crossing them will not yield any benefit as only common features can be exchanged. These features could have different purposes in individuals with significantly different structures. If speciation is too fine however, features which could be beneficially combined may kept separate, and species could suffer from having too few members to reliably improve each generation.

A species is defined by a single representative individual called the holotype. All members of a species have a compatibility distance [Compatibility Distance] below a certain value with the holotype of that species. To determine which species a new individual is in, it is compared with the holotypes of all existing species, in age order (oldest species first). It is placed in the first species for which the compatibility distance fall below the threshold value. If the compatibility distance is too high for all existing species, a new species is created with the individual as the holotype. Finally, each generation, a new holotype is picked for each surviving species. It is chosen at randomly from previous generation members of that species. This ensures that species are always roughly represented by their holotype. If this was not done, species would periodically be re-labelled as they outgrew their generations-old holotypes.

Species are also used to artificially favour new graph structures. The fitness of each individual is divided by the size of its species, meaning species sizes tend towards being proportional to their relative fitnesses. If fitness values were not adjusted like this, new species with a small population and a below-average fitness would have all of their members culled. This is not desirable because new species represent significantly different network structures. It is better to give new structures more individuals and time to optimise and reach their potential before comparing them to existing long-lived species.

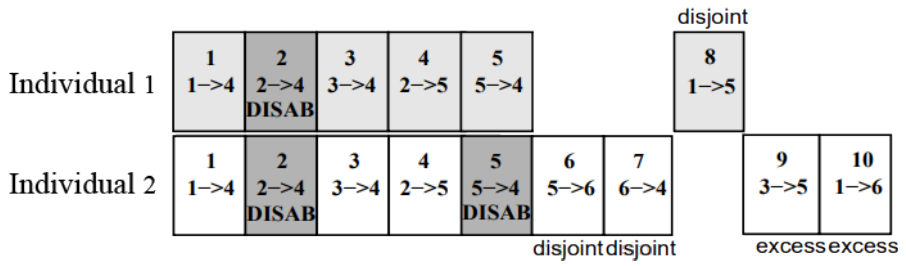

Figure 14: Edge lineup (original from NEAT paper [4])

6.1.7 Compatibility Distance

Compatibility distance can be calculated between any two individuals to determine how similar they are. The edges of the two individuals can be lined up, as shown in the figure above. From this arrangement, we split the edges into three classes: common, disjoint, and excess.

$$compatibility\_distance = \frac{c_1 E}{N} + \frac{c_2 D}{N} + c_3 \cdot \overline{W}$$ [4]

In the above equation, E represents the number of excess edges, D represents the number of disjoint edges, and $\overline{W}$ represents the mean absolute difference between the weights of edges common to both individuals. This includes disabled edges. c_1, c_2, and c_3 are constants used to control the priority of each term. I use the values 1, 1, and 0.4 respectively in my implementation. I used a compatibility distance threshold of 3 to determine speciation. These values were all taken from the NEAT paper [4].

6.2 CCGA

On top of the neat evolution system, I built a layer to support two-player games. I used an approach called coevolution, an extension to standard genetic algorithms to support two inter-dependant populations. The 1994 paper, A Cooperative Coevolutionary Approach to Function Optimization [3], by Potter and De Jong, describes a coevolution system for function optimisation called CCGA. In CCGA, the function being learned is split into multiple sub-functions which are handled by separate populations. Members from all populations are combined in-order to evaluate fitness, meaning no additional fitness metrics need to be developed for the sub-tasks.

The system works by evaluating fitness and creating offspring, one sub-population at a time, in a round-robin fashion. Note that the paper refers to these sub-populations as species, however I will continue using the NEAT definition of species, meaning groups of similar individuals within a population. I adapted this for two-player games by assigning a sub-population for each player, and using NEAT for the underlying genetic operators.

6.2.1 CCGA-2

I specifically chose to utilise the CCGA-2 variant of CCGA in which the fitness of individuals within one sub-population is calculated using a combination of the fittest individuals from the other sub-populations, along with a random selection of individuals. The best resulting fitness value is taken for each individual.

Standard CCGA-1 only uses the fittest individual from each population, and does not include random selection. Using CCGA-2 makes it less likely that a high-fitness combination of networks will be culled because they never get evaluated together. As stated in the CCGA paper [3], CCGA-2 improves performance when sub-functions have inter-dependant variables, with the Rosenbrock function given as an example.

6.2.2 Communication

I took a different approach to communication than in my Q-Learning implementation. Maintaining communication nodes mid-way through the networks would have required a lot of additional logic, especially because only networks for two-player games would require them. Instead, I took inspiration from recurrent neural networks and persisted the values outside of the networks.

Each network generated for two-player game s has additional input and output nodes for communication. Between each game turn, the values from the communication output nodes are copied into the communication input nodes of the other player. This means there is a 1-turn latency on communications, however the method was simple and elegant to implement, as it required no additional logic in lower layers of the system.

6.3 Evaluating Fitness

To evaluate fitness, I created algorithms to take a network, feed values through it, and return the outputs. I created models on top of this to pass data between game instances and the networks. Additionally, these models handle the persistence and transfer of communications between players for two-player games, and can enforce turn-limits on games.

For each fitness evaluation, multiple game instances can be played through before calculating a fitness score. This improves the consistency of fitness scores in games with random elements, and also reduces the requirement for smooth score-gradients in games. In my implementation, 10 games are played each round.

For one-player games, the average score is taken across all game instances played. For two-player games, the average scores are taken across all game for each partner from the other population, and the highest value is used as the fitness, in accordance with CCGA-2.

6.4 Experiments and Evaluation

My standard NEAT implementation for a single population worked reasonably well. My coevolution implementation on the other hand suffered from catastrophically bad performance. This meant that for standard NEAT, I used a population of 150, as suggested in the NEAT paper [4], however, I was forced down to a population size of only 25 for my coevolution implementation.

6.4.1 XOR

Again, I have used the XOR game as a benchmark, and sanity check. In this case however, the check failed. My coevolution implementation failed to reach any reasonable performance level.

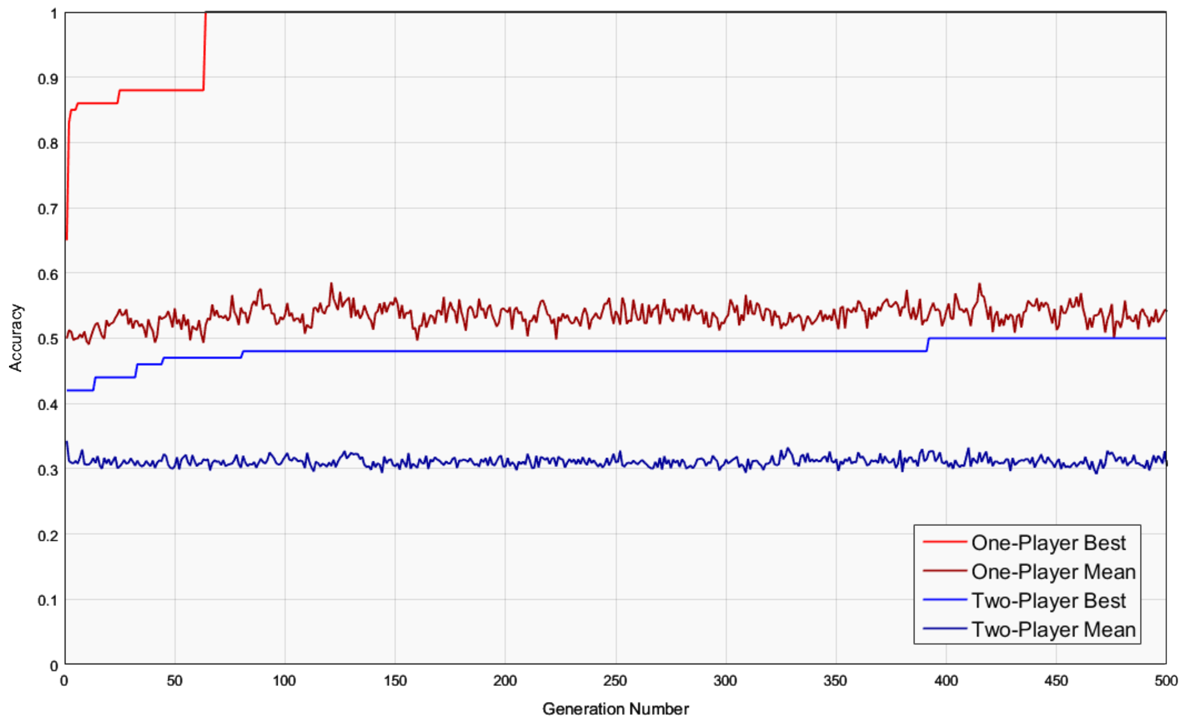

Figure 15: *Accuracy in NEAT learning the XOR game*

The one-player XOR game was learned by single population NEAT in 65 generations. The mean accuracy of the population does not appear to improve significantly past generation 100. My implementation closely matches that used in the NEAT paper [4]. In the paper, it is stated that their implementation takes an average of 32 generations to learn the XOR function. I am not sure why my implementation performs worse.

The two-player performance was very bad, considering XOR is such a simple function. One thing to note about two-player version of the game is that both players must choose the correct answer to achieve any score. This was not a problem for Q-Learning because it deals with expected values, but fitness here was based upon raw score. This is why the one-player mean value hovers around 50%, while the two-player mean sits at only 30%.

6.4.2 Other Games

I was not able to reasonably test coevolution with any other games. I was unable with the pathdfinding game because of the innate large size of the game states. In the alignment game, and the number matching game, the accuracy never significantly improved. The single-population NEAT implementation is able to perform adequately at the alignment game, but performs poorly in the pathfinding game, and number matching game. I have not included measurements, because the purpose of this project was to test systems of multiple communicating agents. Data for standalone single-agent performance is not relevant to the project.

6.5 Conclusions

It is hard for me to draw conclusions about the viability of genetic algorithms for communicating multi-agent systems. The methods I employed definitely proved to be unsustainable. That said, I do not think this entirely rules out the combination of the two ideas. Coevolution has already be shown to be a viable technique [3]. The issues I encountered were mainly related to efficiency, though my ability to evaluate other parts of the system was limited because of those issues. If efficiency could be improved, such a system may become more practical.

Efficiency

The efficiency issues I encountered mainly stemmed from the fact that each individual required many evaluations to judge its efficiency. It could instead be more viable to treat pairs as individuals, and use more traditional genetic algorithms with a single population. Alternatively, it could be more stick to single-round tasks, and implement some sort of mid-network communication, rather than the delayed persistent communication I used.

The tools I used in my approach were poorly suited to the problem. Using a library other than TensorFlow could possibly yield a huge performance increase. Since TensorFlow does not have good support for mutability, it was a very bad match for NEAT. Ideally, the genetic model and the execution model would be merged, such that each individual would exist only as a functioning neural network model, with no other representations to manage.

The inadequacy of TensorFlow meant it performed similarly to standard python maths, eventually leading me to switch to a python implementation to model the neural networks. Because of this, just implementing the system in a lower level language could have made a lot of difference.

Challenges for Evolution

The main road-block for the evolution process seemed to be score gradient. Many games offer score increases in discrete chunks. Sometimes, a lot of complex behaviour needs to be randomly developed before any increase in fitness can be seen. This was successfully alleviated to some degree by averaging the score over multiple plays, this is not a perfect solution however. Another way to alleviate this would be to increase the population size. This would be a sustainable solution, if there are sufficient computational resources available. In general, I think my system suffered most because of the small population size forced by poor performance.

7 Final Thoughts

7.1 Conclusions

To recap, I first implemented four games, each with a two-player version requiring communication between the players for success. I implemented the Q-Learning system, and tested it's ability for all game types, contrasting one-player and two-player performance. Finally, I implemented the NEAT genetic algorithm system, and a CCGA layer on top of it to support two-player games. I tested this system on the XOR game as a test, and the CCGA implementation failed to perform adequately.

I have successfully shown that it is feasible to link up two machine learning agents via a communication channel, and train them as a single system. I have demonstrated Q-Learning as an effective partner for such a system. This is because it is particularly effective at revealing gradient in systems with a lot of noise, and with random elements.

There are a lot more pitfalls to be overcome when using a genetic-algorithm based approach with communicating multi-agent systems, however I still believe that this approach has a lot of promise, if attacked carefully.

7.2 Suggestions for Further Exploration

In this project, I have demonstrated the feasibility of communicating multi-agent systems, but I have not practically explored any of the possible benefits or applications of such systems. I would be interested to be a physical implementation, using the techniques explored in this project. I would especially like to see an implementation where the agents exist on physically separated machines.

Communication Protocol Optimisation

In a system of multiple neural networks which must exchange information, it could be possible to automatically optimise the data exchange for desirable properties. For example, you could train a system to minimise communication bandwidth requirements, without needing to manually designing data-specific compression schemes. You could also optimise for data integrity on a communication channel prone to data-loss. This could allow for data integrity schemes tailored to specific communication channels, or even dynamically adapting to new conditions.

Encoding and Exchanging Knowledge

The same principles of black-boxing data transfer could also be applied to data storage. An agent could be trained to store information, and use it later to complete tasks. The data storage scheme could be completely determined by the agent. You could even have agents exchange knowledge using black-boxed communication as explored in this project. This would also allow for multi-step communication where information communicated at a previous time-step could be used in a later calculation. This could allow for much more complex communication behaviour to develop.

Indirect Communication

In this project, all communication between agents has taken place through channels hard-coded into the structure of the agents. It would be interesting to instead experiment with less direct forms of communication. For example, agents could communicate by making observable changes to some world state such as a game board. This behaviour could come about naturally, so long as the agents benefit in some way by exchanging information.

Further Investigation of Genetic Algorithms

My genetic algorithm implementation was largely lacking in efficiency. A similar system could be built including functionality to freely mutate neural networks. This would mean genetic operators could be applied directly to neural networks. An implementation like this may be able to support a larger populations for coevolution.

Systems of More than Two Players

In this project, I only explored the feasibility of two-agent systems. It would be interesting to investigate more general models for n-agent systems, and how efficiency of would be impacted by the number of agents.

I may even be possible to create a network of individual devices which can be optimised as a single system. The network could even feature a multitude of device types, with different roles in the meta-system.

Communication Bandwidth Requirements

It would be possible to test how well a communicating agent system can compress information by testing minimum communication bandwidth requirements against existing compression methods.

7.3 Reflection

Overall I view this project as a success. I gained a lot of experience with machine learning tool and techniques. That was one of my main personal goals for the project. Additionally, this is the longest-term project of this nature I have ever undertaken. It has given me a much better sense of the kind of work involved in undertaking a large personal code project. Finally, the project was very research heavy. I spent more time reading and researching for this project then I ever have before. Part of the reason was that I started the project with very little machine learning experience.

Implementation-wise, my Q-Learning implementation is everything I planned for it to be. It effectively allowed me to meet my goals for the project, and provided some interesting data. I wish that I could have started again with my genetic algorithm implementation. I came into it with the tools which I chose for Q-Learning, only to find them not suited to the job. I still learned a lot from experimenting with the system, but overall it did not provide much towards the project.

References

[1] A friendly introduction to cross-entropy loss. `https://rdipietro.github.io/friendly-intro-to-cross-entropy-loss/`. Accessed: 2018-05-10.

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. cite arxiv:1312.5602Comment: NIPS Deep Learning Workshop 2013.

[3] Mitchell A. Potter and Kenneth A. De Jong. A cooperative coevolutionary approach to function optimization. In *Proceedings of the International Conference on Evolutionary Computation. The Third Conference on Parallel Problem Solving from Nature: Parallel Problem Solving from Nature*, PPSN III, pages 249–257, London, UK, UK, 1994. Springer-Verlag.

[4] Kenneth O. Stanley and Risto Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary Computation*, 10(2):99–127, 2002.

[5] Peter Stone and Manuela Veloso. Multiagent systems: A survey from a machine learning perspective. *Autonomous Robots*, 8(3):345–383, Jun 2000.