university of groningen / faculty of science and engineering

Comparing the performance of Argumentation-Based Learning with tabular and approximation-based Q-Learning : A quantitative study

Bachelor's Project Thesis

Max Vincent Valk, s3246922, m.v.valk@student.rug.nl,
Supervisor: H. Ayoobi

Abstract: Argumentation-Based Learning (ABL) is a newly developed algorithm for on-line incremental learning which has been shown to have outperformed other algorithms in both learning speed and precision. To expand upon the comparisons made to other algorithms in the original paper, this study focused on tabular and Deep Q-Learning. A genetic algorithm was used to explore the parameters for Deep Q-learning. The ϵ-greedy and Boltzman exploration policies for the Deep Q-Learning algorithm were considered. It was found that ABL outperforms both tabular and Deep Q-Learning, both in terms of final precision and learning speed. However, it should be noted that the search for parameters for the Deep Q-Learning neural network and its exploration policy were by no means exhaustive, and further investigations are required for a more definitive conclusion.

1 Introduction

When Autonomous Systems are designed to operate in very dynamic environments such as the real world, it is impossible to account for each possible situation these systems may encounter. Therefore an agent operating in those environments needs to be able to make decisions about situations which may not have been encountered prior, and change their decision making strategies based on the outcomes of such actions. In order to achieve these goals the Argumentation-Based Learning algorithm was designed, which is rooted in argumentation theory (Ayoobi et al. (2019)). This method is promising and outperformed other algorithms such as a naïve Bayesian classifier, and the PART, PRISM, ID3, J48, ORF and ISVM algorithms on the same tasks. ABL performs better both in terms of learning speed and accuracy. In order to expand upon this set of comparisons to other applicable algorithms, this research will focus on comparing the performance of ABL with the performance of two other methods, tabular Q-learning and approximation-based Q-learning using a neural network, also known as Deep Q-learning.

1.1 Argumentation-Based Learning

Argumentation-Based Learning (ABL) is an algorithm that makes use of an Argumentation Framework (AF) and a Bipolar Argumentation Framework (BAF) (see Dung (1995) and Amgoud et al. (2008), respectively). It learns and chooses which behaviour to perform via interaction of two units: The hypothesis argumentation unit (using AF), and the hypothesis generation unit (using BAF). When a scenario is presented to an agent using ABL, the hypothesis argumentation unit generates the first action to perform in order to resolve the situation. If this fails, this information is passed on to the hypothesis generation unit. This unit is tasked with generating a second guess for the correct action to take, as well as updating the hypothesis argumentation unit. This architecture can be seen in figure 1.1.

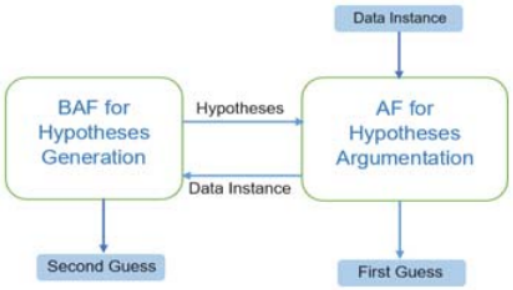

Figure 1.1: The high-level architecture of ABL, taken from Ayoobi et al. (2019).

1.1.1 Argumentation Frameworks

An argumentation framework is a pair $AF = (AR, R_{\text{att}})$, where AR is a set of abstract arguments and R_{att} a set of attack relations, formally $R_{\text{arr}} \subseteq AR \times AR$ (Dung (1995)). These arguments and their relations can be used to reason about which sets of arguments can be held without conflict.

1.1.2 Bipolar Argumentation Frameworks

A bipolar argumentation framework is an argumentation framework expanded with an additional supporting relation (Amgoud et al. (2008)). Formally, it is a triple $BAF =< AR, R_{\text{att}}, R_{\text{sup}} >$, where AR is a set of abstract arguments, R_{att} a set of attack relations $R_{\text{att}} \subseteq AR \times AR$ and R_{sup} is a set of support relations $R_{\text{sup}} \subseteq AR \times AR$. Support relations have a weight, which can be used to assign a numerical value to how well supported an argument is. This can be used to distinguish which argument is seen as defeating another in the case of a mutual attack relationships.

1.1.3 Argumentation theory in machine learning

Argumentation theory has been combined successfully with machine learning. An interesting application is using argumentation in order to clarify and improve classifications (Možina et al. (2007)). This is done by allowing an expert to provide the reasons for a classification with an example. Further, this system will not only produce classifications, but provide reasons alongside with it. This allows for the output to be understood by experts in the field that the classifier is performing in and allows them to correct erroneous reasonings. Other interesting uses have been proposed in the field of linguistics, where argumentation theory was used for out-of-domain sentiment classification (Carstens and Toni (2015) and Carstens and Toni (2017)), and argument mining (Carstens and Toni (2017)).

1.2 Q-learning

Q-learning has shown successes in domains where learning must occur through interaction with the environment, which makes it a potentially suitable approach for the scenarios as discussed in Ayoobi et al. (2019). Examples of such successes are playing certain video games such as pong and breakout better than an expert human player (Mnih et al. (2015)). Q-learning in general, and the implementations chosen in this paper, will be discussed in section 2 and section 3 respectively.

2 Background

In this section, background information on a variety of topics is discussed. If the reader is interested in the actual implementation, this can be found in section 3 instead.

In subsection 2.1 active reinforcement learning is discussed. Subsection 2.2 up to subsection 2.5 discusses Q-learning. In subsection 2.6, neural networks are briefly discussed. Finally, genetic algorithms are discussed in subsection 2.7.

2.1 Active reinforcement learning

In active reinforcement learning problems, the goal is to learn the best action to take in any given scenario, under the restriction that the world is not fully observable to the agent(s) acting in the environment (Russel and Norvig (2016)). A function that tells such an agent which action to take in any given state is called a policy (π). The goal is thus to find the optimal policy, that is a policy that maximizes the possible rewards received. The problem can be described as a set of states S, a set of actions that the agent can take A, and a function

$R : S \times A \times S \mapsto \mathbb{R}$ which tells us the reward the agent receives for performing an action in a starting state and ending up in a subsequent state. It is known that applying an action in a state may result in a state transition, but it is not known to which states state-action pairs transition to. Moreover, these state transitions are stochastic. The set of possible states and actions that can be taken are known. This allows us to give a formal definition of a policy as: $\pi : s \mapsto a, \forall s \in S$. The best actions to take in the environment are learned through interaction, which is where the agent takes an action in the environment. This results in a sample, a tuple (s, s', a, r), with $s \in S$, $s' \in S$, $a \in A$ and $r \in \mathbb{R}$. This tuple contains the original state s, the action the agent took in that state a, the resulting state after performing said action s', and the reward observed r obtained with R. States may be terminating, which is to say that the simulation ends once a terminal state is reached. An example of a terminating state would be reaching the target location in a navigation problem.

2.2 Q-learning

In Q-learning (Watkins and Dayan (1992)) active reinforcement learning is performed by learning a Q-function, which maps state-action pairs onto a Q-score representing the average expected reward for taking that action. That is, the average expected reward that is received in total before reaching any terminating state. There is no need to learn the underlying transition model and rewards, as the algorithm is not interested in knowing the likelihood of all state-transitions, only the averaged best outcome. Learning of the Q-function is done via Q-updates, which are of the form:

$$Q(s,a) \leftarrow Q(s,a) + \alpha(R(s) + \gamma \max_{a'} Q(s',a') - Q(s,a)) \tag{2.1}$$

Where α represents the learning rate, or the speed of adjustment after receiving new information. γ represents the discount factor, which is used to devalue rewards that are obtained later. If a discount factor of $\gamma \in [0, 1)$ is specified, then the agent will favour more immediate reward over more distant rewards. $\max_{a'} Q(s', a')$ represents the maximal Q-value of any action in the resulting state s'. Through the gradual application of this Q-update with many observations, the Q-value will start to hold the approximate total expected value received for choosing any action in any state, typically adjusted so that more immediate rewards are favoured.

2.3 Exploration vs. Exploitation

An important problem encountered in Q-learning (and reinforcement learning in general) is the problem of exploration versus exploitation. As a Q-function is changed over time, it provides us with an approximation of the underlying actual Q-values. Since the algorithm does not have a model of the environment, it is not known when this Q-function cannot be improved further. This means that it may be the case that taking a sub-optimal action in one state may eventually yield a greater reward than performing the best action in this state. In order to address this, exploration (taking actions that may eventually lead to a better reward, but of which this is not known) and exploitation (using the learned knowledge to maximize the reward received) must be balanced. In order to solve this issue, a variety of exploration policies may be used, which balance these two needs of the system.

2.3.1 Exploration policy - Epsilon-greedy

The epsilon-greedy (Sutton and Barto (2018)) exploration policy selects an action at random with a probability of ϵ, and otherwise selects the action whose predicted Q-value is the highest. As initially the need to explore is higher, ϵ is typically reduced over time. If ϵ is never reduced entirely to zero, it guarantees the Greedy in the Limit of Infinite Exploration (GLIE) property. That is to say, all actions must be tried in all states without limiting the attempts to avoid missing an optimal solution, if given infinite time.

2.3.2 Exploration policy - Boltzman

The Boltzman exploration policy (Sutton and Barto (2018)) is a modified softmax function, and can be seen in Equation 2.2.

$$p(s,a,t) = \frac{e^{\frac{Q(s,a)}{t}}}{\Sigma_{i \in A} e^{\frac{Q(s,i)}{t}}} \tag{2.2}$$

Here, the probability of an action being selected in a state is determined by the Q-value of all ac-

tions in that state, combined with a temperature value. By adjusting the temperature value, the behaviour of this exploration policy changes. Lower temperature values result in a greedier behaviour, whereas higher temperatures result in random selection. As before with the ϵ-greedy exploration policy, the temperature is often changed over time. Another similarity is that selecting the right minimal temperature also guarantees the GLIE property.

2.4 Tabular Q-learning

In tabular Q-learning, the Q-value for each state-action pair is saved explicitly in a large table. These values in the table are initialised at random. The first time that the Q-value of a state-action pair is used to choose an action, the algorithm will thus perform a random guess. Furthermore, each scenario that is encountered is stored as an entry in the table, with its own Q-value, in its entirety. This has as effect that each time a Q-update is performed, it affects at most one entry in the table itself, and does not change the Q-values of similar states/actions. The consequence of this is that using this method accurate Q-values will only ever be obtained of the situations which have been encountered prior, and it has to rely on guessing if this is not the case until sufficient information has been obtained to make a more informed decision.

2.5 Approximation-based Q-learning

With an increase in states, there is an exponential increase in the state-space and therefore it is infeasible to keep track of all Q-values. This is known as the curse of dimensionality (Bellman (1957)). This can be addressed by approximating the Q-function, which removes the need for storing all Q-values separately. In this paper a neural network was used for this purpose. This network takes as its input a state, and has one output node per action giving the approximated Q-value for that action in the given state, as illustrated in Figure 2.1. This has as additional benefit that the different state-action pairs can influence all Q-values during training, meaning that information can be inferred from similar states. This is an improvement over the tabular method as that approach needs to encounter a particular state at least once in order to make a meaningful decision in it.

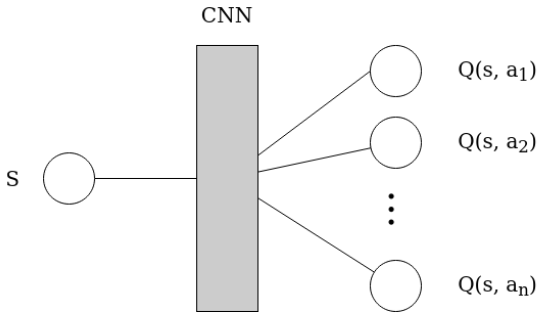

Figure 2.1: The structure of the neural network used for Q-value approximation. As input it takes the state, and as output it will produce the Q-value of each possible action in that state.

2.6 Neural Networks

A neural network consists of layers of connected artificial neurons. Each of these connections has a weight associated with it. The value of a neuron is calculated by multiplying the weights with the activations of the previous layer, and adding a bias:

$$a = \theta + \sum_{i=1}^{n} w_i x_i \tag{2.3}$$

Where θ represents the bias, w_i represents the weight of the connection i, and x_i represents the value of the output of the neuron in the previous layer associated with w_i. To calculate the output value of a neuron, its activation function is applied to the activation a.

2.6.1 Training a Neural Network to approximate a Q-function

When neural networks are trained on single examples, the network can unlearn previous information which nullifies the learning done prior. To mitigate this, one can collect enough samples to present the network with batches. This is done by saving previously encountered experiences and their outcomes. The batches used for training are also referred to as minibatches, and are smaller samples of this history of experiences. Re-using previous samples is a technique which is also known as experience replay (Lin (1993)).

2.7 Genetic Algorithm

A genetic algorithm is an algorithm that mimics aspects of evolution in order to find a solution to an optimisation problem (Kramer (2017)). It maintains a population of potential solutions, which it updates in iterations. At every iteration, each individual solution in the population is assigned a fitness score, after which a selection is made of individuals from the current generation to create the next.

2.7.1 Genotype

Each individual in the population has its features encoded in a genotype, which contains various parameters that describe it. As the genotype is an encoding, it may need to be interpreted to create the phenotype, or actual individual, which it represents.

2.7.2 Population & initialisation

The population has a size which remains constant over all generations. Before the first run an initial set of individuals is created at random, by setting each trait in each individual to a random value of a set of permissible values. Alternatively, a population can also be loaded in to resume training.

2.7.3 Fitness function

Each individual is evaluated by means of a fitness function. This function assigns a numerical score to each individual, allowing to make comparisons between them. Generally, a higher fitness indicates a better performing individual, and thus ideally a better solution. However, designing a fitness function is a matter in which great care is required. The fitness function is a heuristic used to guide the search for a desired performance, and is not precise. Individuals with a high fitness value may display unwanted behaviours. It can also be the case that the fitness function is not a heuristic for the goal that the designer of the function had in mind. For an overview of interesting examples of this behaviour, see Lehman et al. (2018).

2.7.4 Reproduction

Once the fitness score of each individual has been calculated, the next generation is constructed. Individuals are selected proportionate to their fitness, and several methods can be applied to create new individuals in the next generation. Fitness-proportionate selection is often done through roulette-wheel selection, a type of selection in which the probability of an individual i to be chosen is defined as:

$$p(i) = \frac{f(i)}{\Sigma_{j \in G} f(j)} \tag{2.4}$$

Where f is the fitness function, G the set of all individuals within the current generation, and $i \in G$. In this type of selection, given a properly designed fitness function, each individual within a population has a chance of being selected. The downside of using this selection method is that if there is one individual which scores very high, it can start to dominate the selection process, leading to a reduction of variety in a gene-pool.

The following techniques can be used to obtain new individuals:

Elitism When elitism is used to obtain new individuals, a set number of the best performers in each generation is directly copied over into the next. This has as benefit that the best performance in a gene-pool should not drop over generations, at the cost of reducing genetic variety.

One parent crossover A single individual is selected from the generation. A mutation function is applied over all genes, which has a small probability to alter the value of any gene. This can be done by choosing a random (valid) value, a method which is referred to as random resetting. This method can be applied to genes encoding integers as well as categorical variables. Another method for mutation, often used with continuous variables, is to add a semi-random value to the current. Of these methods, the Gaussian mutation is sometimes used, which adds a value drawn from a Gaussian distribution as mutation. Typically, each gene has a set of permissible values (for instance, a pre-specified range), and the new value is clamped to fit within these bounds.

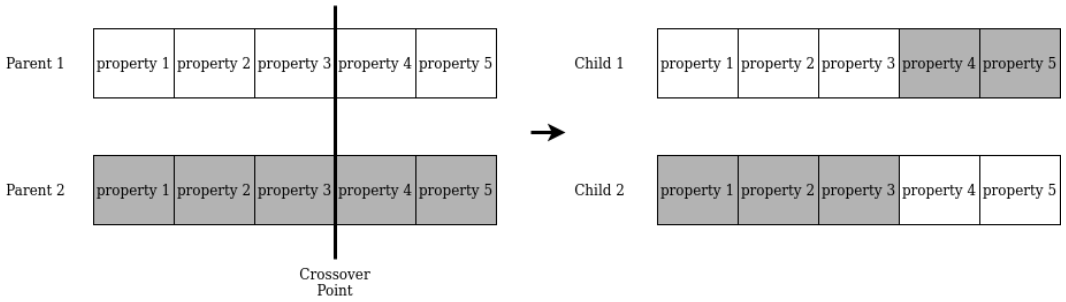

Figure 2.2: A visualisation of 1-point crossover: two parent genes are recombined to create 2 children by splitting both parents at 1 point.

Two parent crossover In two parent crossover, two individuals are selected from the current generation, and their genes are combined such that the result is two new individuals for the next generation. One method of accomplishing this is n-point crossover. In this method, n points are selected in the genome. The parents are split up in sections defined by these points, and are then recombined. An example of 1-point crossover can be seen in Figure 2.2.

It should be noted that there exist different crossover techniques, and it is possible to perform crossover using more than 2 parents. This is outside of the scope of this paper, however.

2.7.5 Parameter tuning in Genetic Algorithms

It was found by Jong (1975) that a genetic algorithm performed best if the population was between 50-100 individuals, with a crossover probability of 0.6 and mutation probability of 0.001. Furthermore, they found that elitism was viable for unimodal, but not multimodal functions. Schaffer et al. (1989) found through experimentation with a different set of functions that they achieved the best results using a population size between 20 and 30 individuals, with a crossover rate between 0.75 and 0.95, and a mutation rate in the range of 0.005 - 0.01. In the latter paper, there was no research conducted on the influence of elitism.

3 Methods

3.1 The scenario

This scenario is the same one as in Ayoobi et al. (2019). The context of the scenario shown to the agent is as follows: It was assumed that the agent has as goal to reach a specific location, as illustrated in Figure 3.1. Before reaching this goal, the agent is confronted with 6 different spaces which can hold an object, of differing colours, or no object at all. Only the fifth space is a relevant obstacle for reaching the specified goal. For everything that the system can encounter in the fifth field, there exists one correct action to perform. The agent has no initial knowledge on which action is the right action for a situation, nor does it know that only the fifth field is relevant. Any algorithm solving this problem has to learn these two properties.

The experiment was performed with 3 different types of objects (ball, box, person), which could have 4 possible colours (red, blue, yellow, green). Additionally, a field could be empty, making for 13 different possible combinations that can be encountered on each field. The object types and colours are stand-ins for meaningful observations an agent can make from the environment and may both play a role in the right solution to an obstacle. The agent has the ability to perform 4 different actions, each with an execution time associated with them. These are finding an alternative route (10s), pushing the obstacle (5s), asking (4s), and continuing (2s).

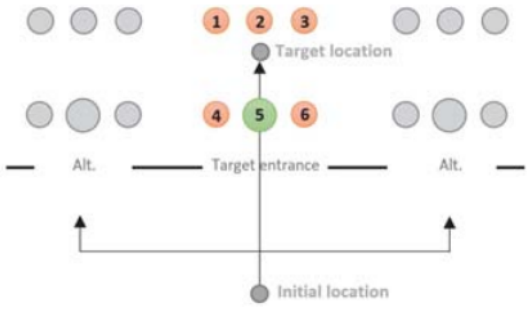

Figure 3.1: The environment in which the agent must act, taken from Ayoobi et al. (2019). The state-space observed by the agent consists of the 6 numbered fields, in which only the fifth field is relevant for solving the problem.

3.2 Q-Learning

Two different types of Q-learning were performed for this paper, namely tabular and approximation-based Q-learning using a neural network. For the latter, two different versions were used, one using an ϵ-greedy exploration policy, and the other using Boltzman. The hyperparameters for the neural networks were found using a genetic algorithm, which will be discussed in subsection 3.5. Each version was run 1000 times. In each run, a random solution-key was generated, which for that run decided the correct behaviour to resolve each specific obstacle. The agent was then presented with 200 subsequent encounters, with each being a description of all objects appearing in slots 1 through 6. For each encounter, the agent attempted to find the correct solution and was allowed to retry until they eventually did.

Reward function The reward function was based on the time each action took to execute. If an action succeeded, the reward was 20 minus the time it took to complete said action. Otherwise, it was equal to the negative time needed to complete an action. This guarantees a positive reward for the right solution, while still keeping the execution times into account, as defined by Ayoobi et al. (2019).

3.3 Tabular-based Q-learning

In the experiment settings of a discount factor (γ) of 0 and a learning rate (α) of 1 were used. The algorithm used a greedy exploration policy, which always took the action with the highest Q-value. The Q-table was initialized at random, using uniformly drawn values from $[-2, 0)$.

3.4 Approximation-based Q-learning

For the implementation of the approximation-based Q-learning method, a neural network was used. The hyperparameters for the neural network were obtained using a genetic algorithm. Two different sets of hyperparameters were found, each using a different exploration policy. These values can be found in Table A.3. Furthermore, both exploration policies decayed their initial parameter (ϵ & temperature) using multiplication with their decay parameter.

The weights of each neural network were initialized by being randomly drawn from a normal distribution, with $\mu = 0$ and the standard deviation for each layer is calculated as:

$$\sigma = \sqrt{\frac{2}{f_i + f_o}} \tag{3.1}$$

Where f_i is the number of inputs for the layer, and f_o is the number of outputs. This method is known as the Glorot method, after one of the authors of the original paper describing it (Glorot and Bengio (2010)).

3.4.1 State-input encoding

In this paper, Exponential Linear Unit (ELU), as seen in Equation 3.2 (for the layers within the network) and linear, as seen in Equation 3.3 (for the final layer) activation functions were used.

$$R(a) = \begin{cases} a & a > 0 \\ \alpha(e^a - 1) & a \leq 0 \end{cases} \tag{3.2}$$

$$R(a) = a \tag{3.3}$$

The input states were presented to the network using one-hot encoding, for all 6 states. With this

encoding, each attribute of the state is encoded as a sequence of zeroes and ones. For instance, in this paper 4 different object types were possible (ball, box, person or none). This attribute is represented by three zeroes and a one, the order of which signifies what attribute is encoded, as demonstrated in Table 3.1. This encoding is applied to each attribute of the state and each individual digit is treated as a different input of the network.

Type	Encoding
Ball	1000
Box	0100
Person	0010
None	0001

Table 3.1: An example of 1-hot encoding, which encodes a property of a state onto a bitstring, with the position of the 1 indicating which property is encoded.

3.4.2 Training

Each sample obtained from interacting with the environment was stored in memory. Training began as soon as the amount of samples stored in memory was equal to a pre-defined minimum, performing 1 training per sample collected. A minibatch, a sampling from the recorded memories, was generated each training, whose size was also pre-specified. The most recently obtained sample was always included in this minibatch. The network then adjusted its weights using the Adam optimizer (Kingma and Ba (2014)).

3.5 Genetic algorithm

3.5.1 Design

Scenarios Before the simulation of each generation, a random mapping was made between each possible object on the fifth field and the correct action to perform. Further, all scenarios (states) that would be presented to the individuals were generated up front, so that each individual would train and be evaluated on the same data. Each individual was presented with 200 subsequent situations, in which they were allowed to select actions until they had found the correct one.

Genome All relevant parameters of the neural network were encoded into the genome, with pre-specified ranges, as can be seen in Table A.1. It should be noted that the variable for minibatch size should be equal to or smaller than the minimum size of the replay memory, and this was enforced at any time these variables changed or were instantiated. This was done by changing the minibatch size to be equal to the minimum size of the replay memory, when this constraint was violated. The gene "Starting exploration policy value" was used to determine the starting ϵ value for ϵ-greedy, and the starting temperature for Boltzman. For the latter, the value was first multiplied by 1000. Both exploration policies had a pre-specified lower limit, which were an epsilon of 0.01 and a temperature of 10 respectively. This value was decayed once per encountered scenario via multiplication with the value of the gene "exploration policy decay rate".

Fitness function The fitness function for selection was based on the two criteria on which Ayoobi et al. (2019) compared the algorithms: final accuracy, and speed of learning. As both learning speed and final performance are relevant, the resulting fitness score of the model was defined as the sum of the final accuracy and the learning speed. The final accuracy was obtained by presenting the model with 10,000 random scenarios at the end of a run, and assigning a score between 0 to 1 to obtain the fraction of correctly solved scenarios. These 10,000 scenarios were held constant for each generation. It should be noted that during this evaluation, the exploration policy that was used was temporarily replaced by a greedy policy, so only the prediction made by the final model was considered. The learning speed was approximated by averaging the accuracy of all 200 runs, where accuracy is the inverse of the needed guesses to successfully resolve an encounter. This resulted in a score of between 0 and 1. These two values are then summed, resulting in a final fitness ranging between 0 and 2.

Mutation The mutation operator was applied to individuals created using 1-parent crossover. For each gene, with a probability specified within the parameters for each run (the mutation rate), the gene would either be randomly reset, or scaled Gaussian noise would be added. This Gaussian

noise was drawn from a Gaussian distribution with $\mu = 0$ and $\sigma = 1$, which then was scaled by a multiplication with 0.1. Random resetting was applied to genes that were integers, while the noise was added to genes encoded as a floating point number.

3.5.2 Parameters

The genetic algorithm for each exploration policy was ran 4 different times, with different parameters as seen in Table A.1. These settings were inspired by the works of Jong (1975) and Schaffer et al. (1989), who performed quantitative studies of parameter settings for genetic algorithms. Each run consisted out of 25 independent simulations of the genetic algorithm. Each of these simulations was initialized with a generation of random genes of a size as indicated in Table A.1, after which 100 generations were simulated.

4 Results

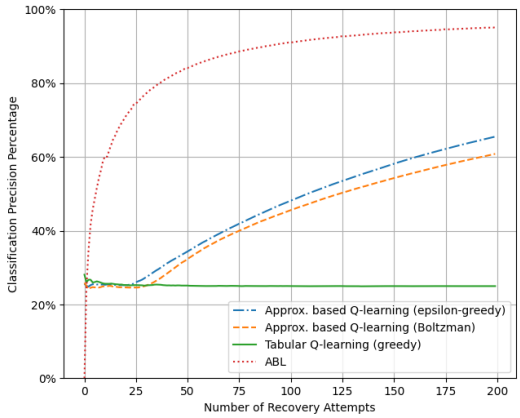

Figure 4.1: A comparison of various methods of Q-learning with ABL, averaged over 1000 individual runs.

The resulting guesses recorded were transformed into precision, as that is the metric that was used by Ayoobi et al. (2019). This was done by calculating the number of 1st correct solutions over the number of scenarios. All runs were then averaged, and the result can be seen in Figure 4.1. The results for ABL were taken from Ayoobi et al. (2019). The final precision for the tabular Q-learning and approximation-based Q-learning using Boltzman was 61%. The final precision for approximation-based Q-learning using ϵ-greedy was 66%. ABL outperformed all three in this regard, having obtained a final precision of 95%. Using permutation testing with 1 million repetitions, this difference in final accuracy was found to be significant between ABL and approximation-based Q-learning using ϵ-greedy ($p < 0.001$), ABL and approximation-based Q-learning using Boltzman ($p < 0.001$), and ABL and tabular Q-learning using a greedy exploration policy ($p < 0.001$).

After 30 attempts, ABL achieves a precision of 74%, whereas for the tabular Q-learning and approximation-based Q-learning using Boltzman, the precision was 25%. Lastly, the precision after 30 attempts for approximation-based Q-learning using an ϵ-greedy policy was 27%. This difference was found significant between ABL and approximation-based Q-learning using ϵ-greedy ($p < 0.001$), ABL and approximation-based Q-learning using Boltzman ($p < 0.001$), and ABL and tabular Q-learning using a greedy exploration policy ($p < 0.001$).

5 Discussion

5.1 Tabular Q-learning

The failure to learn by the tabular Q-learning algorithm is easily explained by examining how it stores the information it learns. In its table, each state is stored together with the approximated Q-values for each possible action in that state only. Therefore, tabular Q-learning can only obtain information about which action to take in a state by having encountered the exact same state before. No inferences can be drawn from related states. As the state-space is equal to $4,826,809$ different possible states (4 objects with 3 possible colours, as well as the option of an empty spot, in 6 spaces), only encountering 200 will lead to no noticeable increase in performance.

The Q-table has been randomly initialized, such that the values are negative, but smaller than the reward that would be received from choosing a

wrong option. Combined with a greedy exploration policy this results in the algorithm trying actions randomly that have not yet been tried in the situation until the correct one has been found. This is the best performance that can be expected from this algorithm, even if certain parameters or the exploration policy are changed. Using a discount factor of $\gamma = 0$ and learning rate of $\alpha = 1$ are the best possible settings. Only one action needs to be taken to resolve an encounter and hence there is no need to take into account a later, possible reward. Further, the environment is entirely deterministic, removing the need for gradual learning and allowing for a learning rate of 1. Lastly, any exploration policy can only either match the greedy exploration policy in performance, or perform worse. This is because any exploration policy does not solve the fundamental problem of lack of inference between similar states which hinders the generalisation of knowledge.

5.2 Approximation-based Q-learning

5.2.1 Approximation-based Q-learning using ϵ-greedy

The comparatively worse performance of the approximation-based Q-learning method (using ϵ-greedy) as compared to ABL can be partially explained by looking at the shape of the precision in Figure 4.1 and the parameters in Table A.3. The initial chance level performance is the result of the high value of ϵ at that time, combined with the minimum size of the replay memory, which governs when training can occur. The performance starts increasing when ϵ has decayed further, and the network has collected enough experiences to start training. This, combined with the slower improvement rate as compared to ABL, can be attributed to the need of a neural network to train using a lot of data. It learns using correlation only, and when there are many inputs which are not related to the desired output, this increases the amount of samples that are needed. As ABL forms hypotheses about causation, it quickly find evidence against, and for, causal links. This results in a faster increase in classification precision.

5.2.2 Approximation-based Q-learning using Boltzman

The performance of the approximation-based Q-learning method (using Boltzman) can be explained similarly to the one of the algorithm using ϵ-greedy. The initial value of the temperature is decayed to the minimum of 10 within the first 4 scenarios that are presented to the agent. However, the untrained network provides Q-values that are relatively small (due to how it is initialized), and even with this minimal temperature the chances of selection for each action are nearly identical, essentially acting like the ϵ-greedy exploration policy. There usually is a very slight preference for the highest scoring action, which can be seen in the results as the performance of the untrained network with this policy is a little lower (around 0.5%) than with truly picking an action at random. The difference between this method and ABL can also be explained by the need of the neural network to collect many samples prior to training.

Perhaps more interesting is how the different exploration policies have a very different value for the minimum size of the replay memory, but still seem to be improving their precision at a point that is close in time. With the ϵ-greedy policy, enough data to begin training was usually collected at around 10 scenarios encountered. However, the value of ϵ decays slowly and at this point, the agent is still taking a random action 69% of the time. When it reaches the 25th scenario, this is still 42%. Therefore, learning may start earlier, but the ability of the network to act according to obtained information is limited by its exploration policy. As the temperature for Boltzman is already at the lowest value possible by the time sufficient data has been collected in order to start training, it immediately starts showing strong preferences for actions.

5.3 Exhaustivity of parameter search

Regarding the exhaustivity of the parameter search, a few remarks can be made. Certain behaviour and parameters could have been a part of the search performed by the genetic algorithm, but were not. Only one type of schedule for the

temperature and ϵ was evaluated. A search over certain parameters needs to be considered over a wider range, such as the minimum temperature for the Boltzman exploration policy, as it was found that an agent assuming this minimum value within few scenarios performed well.

Further, different exploration policies may be used. While this might result in a different approach with regards to exploration, however, the fundamental learning is done by the neural network itself. In the neural network, only one method of initialisation, and specific activation functions were considered. It may be interesting to explore if differently selected parameters would make a difference in performance. Nevertheless, the issues that were touched upon in the difficulties of learning in a neural network, given the task at hand, still hold regardless of the choice of parameters.

6 Conclusion

This study aimed to compare the performance of Argumentation-Based Learning (ABL) to tabular and approximate Q-learning using a neural network, also known as Deep Q-Learning. It was shown that ABL performed better than tabular Q-learning in a deterministic scenario in which a single action was required to reach a terminating state. ABL also outperformed approximation-based Q-learning, but further research is required to conclude that no possible implementation can outperform ABL.

7 Acknowledgements

I would like to thank the following people for their contributions:

- My supervisor, Hamed Ayoobi, for his ongoing guidance, support and patience.
- Noemy Vergara Vera, for bringing out the best version of me, as well as being an excellent rubber ducky.
- My family, without whom I would not be able to get this far.
- Harrison Kinsley, who has kindly provided many helpful code snippets on the website `http://www.pythonprogramming.net`, some of which were used for this project.

References

Amgoud, L., Cayrol, C., Lagasquie-Schiex, M., and Livet, P. (2008). On bipolarity in argumentation frameworks. *International Journal of Intelligent Systems*, 23(10):1062–1093.

Ayoobi, H., Cao, M., Verbrugge, R., and Verheij, B. (2019). Handling unforeseen failures using argumentation-based learning. In *2019 IEEE 15th International Conference on Automation Science and Engineering (CASE)*, pages 1699–1704.

Bellman, R. (1957). *Dynamic Programming*. Princeton University Press, 1st edition.

Carstens, L. and Toni, F. (2015). Improving out-of-domain sentiment polarity classification using argumentation. In *2015 IEEE International Conference on Data Mining Workshop (ICDMW)*, pages 1294–1301.

Carstens, L. and Toni, F. (2017). Using argumentation to improve classification in natural language problems. *ACM Transactions on Internet Technology*, v17 n3 (20170712):1–23.

Dung, P. (1995). On the acceptability of arguments and its fundamental role in nonmonotonic reasoning, logic programming, and n-person games. *Artificial Intelligence*, 77(2):321–357.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. *Journal of Machine Learning Research*, v9 (2010 12 01):249–256.

Jong, de, K. (1975). *An analysis of the behavior of a class of genetic adaptive systems*. PhD thesis, University of Michigan.

Kingma, D. and Ba, J. (2014). Adam: A method for stochastic optimization. *International Conference on Learning Representations*.

Kramer, O. (2017). *Genetic Algorithm Essentials*. Springer, 1st edition.

Lehman, J., Clune, J., Misevic, D., Adami, C., Altenberg, L., Beaulieu, J., Bentley, P. J., Bernard, S., Beslon, G., Bryson, D. M., Chrabaszcz, P., Cheney, N., Cully, A., Doncieux, S., Dyer, F. C., Ellefsen, K. O., Feldt, R., Fischer, S., Forrest, S., Frénoy, A., Gagné, C., Goff, L. L., Grabowski, L. M., Hodjat, B., Hutter, F., Keller, L., Knibbe, C., Krcah, P., Lenski, R. E., Lipson, H., MacCurdy, R., Maestre, C., Miikkulainen, R., Mitri, S., Moriarty, D. E., Mouret, J.-B., Nguyen, A., Ofria, C., Parizeau, M., Parsons, D., Pennock, R. T., Punch, W. F., Ray, T. S., Schoenauer, M., Shulte, E., Sims, K., Stanley, K. O., Taddei, F., Tarapore, D., Thibault, S., Weimer, W., Watson, R., and Yosinski, J. (2018). The surprising creativity of digital evolution: A collection of anecdotes from the evolutionary computation and artificial life research communities.

Lin, L. (1993). *Reinforcement Learning for Robots Using Neural Networks*. PhD thesis, Carnegie Mellon University.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. A., Veness, J., Bellemare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wiestra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 2015 Feb 26; 518(7540):529–533.

Možina, M., Žabkar, J., and Bratko, I. (2007). Argument based machine learning. *Artificial Intelligence*, v171 n10-15 (200707):922–937.

Russel, S. and Norvig, P. (2016). *Artificial Intelligence: A Modern Approach*. Pearson, 3rd edition.

Schaffer, J., Caruana, R., Eshelman, L., and Das, R. (1989). A study of control parameters affecting online performance of genetic algorithms for function optimization. In *Proceedings of the Third International Conference on Genetic Algorithms*, pages 51–60.

Sutton, R. and Barto, A. (2018). *Reinforcement Learning - An Introduction*. The MIT Press, 2nd edition.

Watkins, J. and Dayan, P. (1992). Q-learning. *Machine Learning*, v8 n3-4 (199205):279–292.

A Appendix

A.1 Genetic algorithm

Parameter	De Jong I	De Jong II	Schaffer et al. I	Schaffer et al. II
Pool size	64	64	24	24
Crossover	40	40	18	18
Elitism	0	4	0	2
Mutation rate	0.001	0.001	0.01	0.01

Table A.1: The settings used for the genetic algorithm, as inspired by the research done by Jong (1975) and Schaffer et al. (1989). The crossover represents how many individuals were computed via crossover of two parents. The remaining individuals that were not created using crossover or elitism (if applicable), were copied over directly into the next generation after a mutation function was applied.

Parameter	Minimum value	Maximum value
Learning rate	0.000001	1.0
Minimum size of replay memory	1	200
Minibatch size	1	200
Layers in Neural Network	1	16
Neurons per layer	1	32
Starting exploration policy value	0.01	1.0
Exploration policy decay rate	0.01	1.0

Table A.2: The valid ranges of each parameter encoded in a gene in the neural network. Note that all parameter ranges specified with a decimal point indicate floating point values, whereas the other ranges indicate integers. For the Boltzman exploration policy, the encoded starting exploration policy value was first multiplied by 1000 when constructing the phenotype.

A.2 Neural Network hyperparameters

Parameter	Value (ϵ-greedy)	Value (Boltzman)
Learning rate	0.9138424977829493	0.7895546693120532
Minimum size of replay memory	42	174
Minibatch size	42	53
Layers	2	3
Neurons per layer	3	11
Starting exploration policy value	0.9768238402411479	0.3870137999961333
Exploration policy decay rate	0.9663420261989969	0.3976583785573679

Table A.3: The hyperparameters found for the best performing neural network of all genetic algorithm settings tried. The starting exploration policy value was multiplied by 1000 when the phenotype was created from the genotype.

B Appendix

B.1 Software

Software / Library	Version
Python 3	3.7.5
Tensorflow	2.2.0
Tensorflow_gpu	1.14.0
Keras	2.3.1
Numpy	1.16.2
Matplotlib	3.0.2
Tqdm	4.43.0

Table B.1: A list of software and libraries which have been used in this study. The source code can be found at `https://github.com/MaxVinValk/Bachelor_Project`