Clemson University
TigerPrints

All Theses | Theses

12-2011

A Comparison of the Performance of Neural Q-learning and Soar-RL on a Derivative of the Block Design (BD)/Block Design Multiple Choice (BDMC) Subtests on the WISC-IV Intelligence Test

Charreau Bell
Clemson University, charreau.s.bell@gmail.com

Follow this and additional works at: http://tigerprints.clemson.edu/all_theses

Part of the Computer Engineering Commons

Recommended Citation
Bell, Charreau, "A Comparison of the Performance of Neural Q-learning and Soar-RL on a Derivative of the Block Design (BD)/Block Design Multiple Choice (BDMC) Subtests on the WISC-IV Intelligence Test" (2011). *All Theses*. Paper 1279.

This Thesis is brought to you for free and open access by the Theses at TigerPrints. It has been accepted for inclusion in All Theses by an authorized administrator of TigerPrints. For more information, please contact awesole@clemson.edu.

A Comparison of the Performance of Neural Q-learning and Soar-RL on a Derivative of the Block Design (BD)/Block Design Multiple Choice (BDMC) Subtests on the WISC-IV Intelligence Test

---

A Thesis
Presented to
the Graduate School of
Clemson University

In Partial Fulfillment
of the Requirements for the Degree
Master's of Science
Computer Engineering

---

by
Charreau Sieanna Bell
December 2011

Accepted by:
Dr. Robert Schalkoff, Committee Chair
Dr. Ian Walker
Dr. Timothy Burg

Abstract

Teaching an autonomous agent to perform tasks that are simple to humans can be complex, especially when the task requires successive steps, has a low likelihood of successful completion with a brute force approach, and when the solution space is too large or too complex to be explicitly encoded. Reinforcement learning algorithms are particularly suited to such situations, and are based on rewards that help the agent to find the optimal action to execute given a certain state. The task investigated in this thesis is a modified form of the Block Design (BD) and Block Design Multiple Choice (BDMC) subtests, used by the Fourth Edition of the Wechsler Intelligence Scale for Children (WISC-IV) to partially assess childrens' learning abilities. This thesis investigates the implementation, training, and performance of two reinforcement learning architectures for this problem: Soar-RL, a production system capable of reinforcement learning, and a Q-learning neural network. The objective is to help define the advantages and disadvantages of solving problems using these architectures. This thesis will show that Soar is intuitive for implementation and is able to find an optimal policy, although it is limited by its execution of exploratory actions. The neural network is also able to find an optimal policy and outperforms Soar, but the convergence of the solution is highly dependent on the architecture of the neural network.

Dedication

I dedicate this thesis to my parents and my brothers. Since the beginning of my academic career, they have guided, supported, encouraged me towards excellence in all my endeavors. Without their constant love, advice, and resolute confidence in my abilities, I would not have been able to complete this thesis with the quality it deserves.

Acknowledgments

I would like to first and foremost thank God, who has been the source of my confidence and driving force behind all my undertakings.

I also especially thank my advisor, Dr. Robert Schalkoff, for patiently guiding me towards a feasible thesis topic and for all the advice and expertise he has provided me as I completed this work. I also thank Dr. Ian Walker and Dr. Timothy Burg for their recommendations and support through this process.

Finally, I'd like to thank my student colleagues who provided me with insight, ideas, and suggestions to complete this thesis.

Table of Contents

Title Page . . . . **i**

Abstract . . . . **ii**

Dedication . . . . **iii**

Acknowledgments . . . . **iv**

List of Tables . . . . **vi**

List of Figures . . . . **vii**

1 Introduction **1**
1.1 Automated Learning 1
1.2 Thesis Overview 5

2 Background **6**
2.1 Reinforcement Learning 6
2.2 Cognitive Architecture: Soar 10
2.3 Artificial Neural Networks 16
2.4 The Fourth Edition of the Wechsler's Intelligence Scale for Children (WISC-IV) . . . 23

3 The Problem **24**
3.1 Considerations of the Block Design (BD) and Block Design Multiple Choice (BDMC) Subtests 24
3.2 Block Design - Disembodied Formulation 24
3.3 Assumptions and Constraints 26

4 Methodology **27**
4.1 Soar 27
4.2 Neural Network 33

5 Results and Discussion **37**
5.1 Soar 37
5.2 Neural Networks 42
5.3 Quantitative Comparison of Soar and Neural Networks 55
5.4 Qualitative Comparison of Soar and Neural Networks 56

6 Conclusions **59**

Bibliography **61**

List of Tables

4.1 Soar's Representation of the Problem State 30
4.2 Neural Network Topologies Investigated 34

5.1 Numeric Preferences and Testing of Soar with Standard Functionality 38
(a) Numeric Preferences 38
(b) Test Performance 38
5.2 Numeric Preferences and Testing of Soar with Augmented Functionality 41
(a) Numeric Preferences 41
(b) Top Five Operators 41
(c) Test Performance 41
5.3 108x1x1 Multi-NN Test Performance 49
5.4 108x1x1 Multi-NN with Identical Initial Weights Test Performance 50
5.5 108x63x63 Single-NN Test Performance 50
5.6 Qualitative Comparison of Neural Network Architectures 54
5.7 Quantitative Comparison of Neural Network and Soar Performance 56
(a) Multi-NN Performance 56
(b) Single-NN Performance 56
(c) Soar Performance 56
5.8 Comparison of Soar and Neural Network Design 57

List of Figures

1.1 Difficulty of BD-DE Problem as a Function of the Puzzle Size and Number of Blocks 3
  (a) Soar Problem Complexity 3
  (b) NN Problem Complexity 3

2.1 Comparison of Supervised Learning and Reinforcement Learning Techniques 7
  (a) Supervised Learning 7
  (b) Reinforcement Learning 7
2.2 Soar's State Representation by Linked Working Memory Elements 12
2.3 Stages of the Soar's Decision Cycle 13
2.4 A Single Unit of an Artificial Neural Network 17
2.5 Structure of a Multilayer Feedforward Artificial Neural Network 18
2.6 Ideal Training Error As Training Proceeds for Learning Agent 20
2.7 Single-NN Q-learning Architecture 21
2.8 Multi-NN Q-learning Architecture 22

3.1 Desired Functionality of Automated Agent on BD-DE Subtest 25

4.1 Game Environment Relationship with Soar Kernel and Program Execution 28
4.2 Interpretation of Inputs to the Neural Networks 35
  (a) 2-input interpretation 35
  (b) 108-input interpretation 35

5.1 Soar Agent Training with Standard Functionality 38
5.2 Soar Training with Augmented Functionality 39
5.3 Comparison of Puzzle Solution Steps and Function Approximation Error 42
5.4 2x5x1 Multi-NN Training 43
  (a) 50 training patterns 43
  (b) 100 training patterns 43
  (c) 500 training patterns 43
5.5 12x13x1 Multi-NN Training 45
  (a) 50 training patterns 45
  (b) 100 training patterns 45
  (c) 500 training patterns 45
5.6 12x13x7 Single-NN Training 46
  (a) Logistic activation function 46
  (b) Linear activation function 46
5.7 Assessment of Error Measure Consistency for 12x13x7 Single-NN 47
5.8 108x1x1 Multi-NN Training 48
5.9 108x1x1 Multi-NN with Identical Initial Weights Training 49
5.10 108x63x63 Single-NN Training 51

Chapter 1

Introduction

1.1 Automated Learning

The increasing demand for responsive and interactive technologies such as unmanned vehicles, challenging video games, predictive software, and robotic assistants requires software able to accommodate a variety of requests made by the user. Predicting and encoding this much information for millions of possible states or situations is at least daunting and painstaking, if not practically impossible. Even worse, the software is fixed; there is no adaptation to different users, needs, or unexpected situations. The missing link between rigid and flexible software systems is something all humans are equipped with: the ability to learn. Endowed with the ability to find or determine the best action to take in a given situation based on its own experience, a software agent could provide greater functionality, more tailored responses to the user, and better performance of a task.

How can an automated agent be taught to learn a new task? The main ways in which agents learn are analogous to the methods by which humans learn. The most straightforward type of learning is supervised learning in which the agent is shown examples and informed of what the correct response should be, and eventually the agent learns to respond with the same answers. Another type of learning is unsupervised learning, in which the agent finds patterns in the dataset to model the data. The final type of learning is reinforcement learning, in which an agent interacts with the environment and figures out by trial-and-error the best action to take based on how the environment responds to its actions. This final type of learning is the focus of this thesis.

Reinforcement learning is a particularly powerful learning algorithm because a single feed-

back in the form of a reward can be administered at the end of its sequence of actions, and indicates whether the outcome was desirable or undesirable. From this reward, agents are able to learn the optimal action to take based on its current state. For problems with large state spaces, this requires a representation that is compact enough to meet the agent's memory requirements, allows fast recall of desired actions in states it has already seen, and enables informed decision-making for states it has not seen. The question then becomes - what is the best architecture for facilitating reinforcement learning for problems with a large state space?

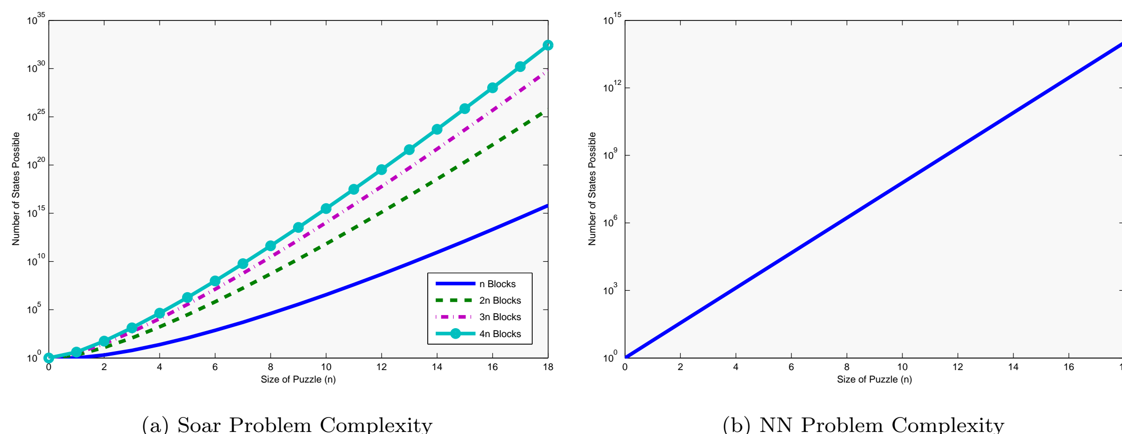
This thesis seeks to address these two questions by comparing two architectures, a production system and a neural network, to implement the well-known reinforcement learning algorithm Q-learning[22], and evaluates their performance on a test problem. The problem to be solved is the Block Design - Disembodied (BD-DE) test, a derivative the Block Design (BD) and Block Design - Multiple Choice (BDMC) subtests from the Fourth Edition of the Wechsler's Intelligence Scale for Children, an intelligence test that is used to assess the intelligence of school-age children. These tests on the WISC-IV help identifiy deficiencies in problem solving and sequential reasoning; an automated agent that solves this problem is therefore required to have both of these capabilities.

1.1.1 The BD-DE Problem and Complexity

The BD-DE problem involves reconstructing a goal image given a set of blocks that contain subpatterns of the goal puzzle, and each block is of 6 different types. This is described explicitly in Chapter 3, and shown in Figure 3.1. For most children, this is a simple task, and in fact, was a game in the early 1990s, but for others, it can be quite a challenge. According to [3], some children attempt to solve the puzzle by trial-and-error, whereas others add the pieces haphazardly and do not learn from their previous mistakes. Some children plan out their designs, while others are more impulsive. Children twist their bodies to gain more perspective on the puzzle, work from different directions to establish order, and also employ other strategies to fill in the puzzle correctly.

For automated agents, this task can be even more challenging. The agent is merely given a goal puzzle, an empty puzzle, and blocks with no further instructions (although some constraints are imposed as described in Chapter 3), and is expected to learn what to do with the blocks to maximize the reward given from the environment. Mathematically speaking, given a goal pattern that has been subdivided into 9 sections with 9 blocks available to reconstruct this pattern, this represents 9! or 362,000 possible solutions. The probability of randomly achieving a correct configuration is then

3×10^{-6}. This is the problem that will be solved by the production system, as well as variations on the puzzle size and the number of blocks provided. However, if all 6 block types are available at each step for each square, the solution space increases to 1.1×10^7, and the probability of randomly achieving the goal configuration or by a brute force approach is 9.9×10^{-8} for a single configuration. The neural network will be used to solve this task. The exponential increase in difficulty experienced by these agents to solve this problem is shown in Figure 1.1. The outcome of this work will aid in the determination of the best architectures to use for a given task, as well as demonstrate the effectiveness of automated agents using reinforcement learning to navigate problems that have a large solution space.



(a) Soar Problem Complexity

(b) NN Problem Complexity

Figure 1.1: Difficulty of BD-DE Problem as a Function of the Puzzle Size and Number of Blocks

1.1.2 Reinforcement Learning Using Soar and Artificial Neural Networks

Reinforcement learning with these architectures has been used to solve a number of puzzles in varying capacities. The Soar production system, together with its reinforcement learning module has been used to solve the missionaries and cannibals problem [11] which involves finding an optimal solution to a problem that humans solve with great difficulty; the Eaters problem [11], which requires an agent in a maze to maximize the food rewards it receives; and the taxicab problem, which requires the agent to find the optimal path to the destination with the constraint of gas consumption [1]. These implementations have achieved varying degrees of success.

Furthermore, neural networks have also been quite successfully used in concert with rein-

forcement learning algorithms. Gerald Tesauro's agent [19] plays Backgammon at a level very close to that of the best human players in the world [20] using a temporal difference (TD) reinforcement learning algorithm. TD-Gammon 3.0, the lastest iteration of the neural network, receives the raw state of the board as well as other heuristics as input to the network, and has a hidden layer with logistic activation functions for both the hidden and output layers. Training required 1,500,000 games to achieve expert play. The original research in 1992 found also found that although supervised learning neural networks suffer from overfitting when the number of units in the hidden layer is increased, TD-learning neural networks increase their performance with increased numbers of hidden units.

A Q-learning neural network was specifically used in the research done by Eck and Wezel for playing the game of Othello[21]. Their research contrasted two 3-layer feedforward architectures for the neural networks: a single neural network with Q-values as output for all actions (single-NN), and several distinct networks, each outputting the Q-value for one specific action (multi-NN). Each network had one hidden layer and hyperbolic tangent activation functions. The action selection method used was the Boltzmann distribution which eventually transitioned to greedy selection after 12,000,000 games. For each step, the reward was 0, except at the end of the game when the agent received -1 for a loss, and +1 for a win. The research concluded that both single-NN and multi-NN Q-learning agents perform better than agents with fixed strategies, although the single-NN was able to learn to play Othello faster than the multi-NN.

Q-learning neural networks have also been used for mobile robots to learn the tasks of obstacle avoidance and wall following. The work of Ganapathy and Liu in [5] explored a Q-learning neural network to implement a number of controllers. The architecture for both these controllers was a 3-layer feedforward network, with 8 linear input units, 16 tangent sigmoid hidden units, and 5 linear output units. The action selection method used the Boltzmann distribution, a set of intermediate rewards ranging from -0.1 to 0.3 for each of the 5 actions, and terminal rewards of -10 and +1 depending on the controller. Both these implementations were successful, and allowed the robot to learn the tasks of wall following and obstacle avoidance. The flexibility of the reward system can be shown by contrasting the sensor-based obstacle avoidance reward function in [5] with the reward system in [6]. Both systems use the same architecture and action selection mechanism, but differ in the magnitude of the rewards. In both cases, the system was able to learn to move in the environment without colliding with obstacles.

1.2 Thesis Overview

The rest of this thesis will provide an answer to the two research questions proposed. Chapter 2 will provide an overview of reinforcement learning concepts as well as Soar and neural networks. Chapter 3 will formally describe the BD-DE problem, and Chapter 4 will describe the details of the design of the agents used to solve it. Results of the tests used to assess architecture performance will be reported in Chapter 5, and this section will also provide insight on the implications of these results. Finally, Chapter 6 describes the conclusions of this research.

Chapter 2

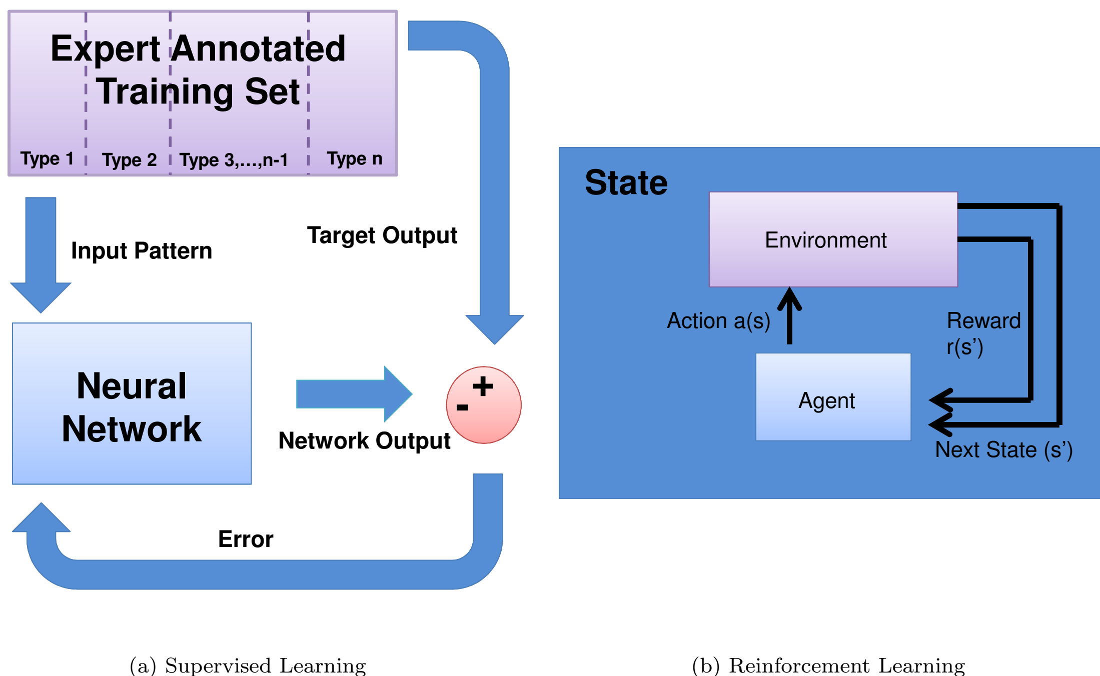
Background

2.1 Reinforcement Learning

Reinforcement learning is the method by which an agent learns to interact with the environment in order to maximize the rewards it expects to recieve in the future if it executes a certain set of actions. Unlike supervised learning, the agent is not provided the correct answer such that the agent can tailor its responses to match those given by the expert annotations in the training set; rather, the agent's learning strategy is based on a trial-and-error search to determine the best actions for maximized rewards. Figure 2.1 illustrates this difference approaches to learning between these two paradigms. In general, numerical rewards are returned to the system as a final assessment; that is, +1 for a favorable outcome, -1 for an unfavorable outcome, and 0 otherwise, for example. Intermediate rewards, that is, rewards that are given at non-terminal states, are also frequently given in order to guide the agent's actions towards a more favorable outcome and speed convergence as described in [9].

In addition to the large state space and related problems identified in Chapter 1, supervised learning techniques also frequently cannot be used because even the most seasoned experts do not always know or have the best strategies to maximize future rewards[19]. These considerations render explicit programming and even supervised learning infeasible techniques for solving the problem.

Thus, reinforcement learning algorithms are of great use in problems where an agent must navigate wisely through successive states. Because of their inherent sequential nature, these problems lend themselves to formulation within the Markov Decision Process (MDP) framework. The major



(a) Supervised Learning (b) Reinforcement Learning

Figure 2.1: Comparison of Supervised Learning and Reinforcement Learning Techniques

points of theory underlying MDPs will be outlined briefly in Section 2.1.1 and solutions of the MDP formulation will be described in 2.1.2. Information on reinforcement learning, deeper mathematical development of MDPs, and more analysis with relationship to reinforcement learning can be found in [18], [13], and [16].

2.1.1 Markov Decision Processes

MDPs are extensions of Markov chains. Markov chains are sequences of random variables s_0, s_1,..., s_n which for this framework are considered states of a system at time n to form the set of states S. For each state s, there is a fixed probability $P(s'|s)$ that the system will transition to state s'. MDPs have the additional characteristic that they have a finite set of actions $A = \{a_1, a_2, \ldots, a_n\}$ that represent all the possible actions that can be taken in the state space. Thus, given a state s at time n, the action a_n causes a transition to state s', the successor state, with probability $P(s'|s, a_n)$. Furthermore, in each state s, the agent receives a scalar, finite, bounded reward $r(s) \in \mathbb{R}$. The scalar discount factor γ defines the relative balance between the immediate reward and the future reward, and is used also to reduce difficulties in evaluating sequences of states[16]. Thus, an MDP that describes a sequential decision problem is a quintuple defined by S, A, $P(s'|s, a)$, R, and γ.

MDPs in reinforcement learning are also assumed to satisfy the Markov property,

$$P(s_n|s_{0:n-1}) = P(s_n|s_{n-1}),$$

which states that the current state depends only on the previous state and not on the sequence of states that preceded it. The significance of the Markov property for RL problems formulated through this framework is that the previous state encodes all the information necessary to make a decision to reach the successive state, and the history of all the states previously are implicitly represented by the previous state. This significantly reduces computational complexity.

As previously stated, the goal of the MDP formulation is to determine the best action to perform in each state such that $\mathbb{E}[\sum_{n=n_0}^{n=\infty} \gamma^n r_n]$, or the expected sum of discounted future rewards is maximized. The formulation of a value function $U(s)$ logically follows, and is defined for each state by Bellman's equation:

$$U(s) = R(s) + \gamma \max_{a \in A(s)} \sum_{s'} P(s'|s,a)U(s').$$

These value functions can be regarded as the utility, or "goodness" of that state. Solving Bellman's equations therefore produces the utility of each state which can be used to determine an optimal policy. Any set of actions defines a policy $\pi : S \to A$, though the optimal policy π^* defines the set of actions that maximize the future expected reward. If the utilities of each state are known, π^* is simply choosing the actions which leads to the successor state with the highest utility. Formally,

$$\pi^* = \arg\max_{\pi} U^{\pi}(s) \qquad \forall s \in S.$$

The non-linearity of Bellman's equations due to the max operator makes the solution of these equations difficult for non-trivial problems, and methods of determining the optimal policy will be described in the following section.

2.1.2 Algorithms for Calculating Optimal Policies for MDPs

There are two main approaches to solving Bellman's equations: dynamic programming and model-free approaches. Dynamic programming (DP) requires a complete and accurate model of the system, so R and $P(s'|s,a)$ must be known to calculate U for each state[18]. These algorithms solve for $U(s)$ iteratively and can reliably compute the optimal policies and value functions[21]. There

are two main DP algorithms: value iteration, which starts with arbitary values for the utilities of the states in Bellman's equations, and then iteratively recalculates the utilities to select an optimal action in each state; and policy iteration, which begins with an arbitary policy, executes the policy to calculate new state utilities, and then uses these utilities to identify an updated policy.

In the absense of a model, Watkins Q-learning[22] can also be used to calculate an optimal policy based on Q-values. Q-values differ from state utilities in that $Q : S \times A \rightarrow \mathbb{R}$ whereas $U : S \rightarrow \mathbb{R}$, and thus Q-values are defined as state-action pairs. Q-learning is similar to value iteration in that it is based on calculating state utilities to dictate the policy, but uses temporal difference (TD) learning to adjust $Q(s,a)$ towards $Q(s'|a)$. The Q-function is defined as the sum of immediate reward received having executed action a and the discounted reward garnered if an optimal policy is followed thereafter. This constraint equation is shown below:

$$Q(s,a) = R(s) + \gamma \max_{a'} Q(s',a'),$$

and holds when the correct Q-values have been reached[16]. Then, the optimal policy is defined as:

$$\pi^* = \arg\max_{a \in A} Q(s,a) \qquad \forall s \in S.$$

Employing this algorithm for one-step Q-learning requires iterative updates of the form

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

to update the utilities of the state-action pairs when an action a is taken in state s.

An important component of reinforcement learning algorithms is the balance between exploratory and exploitative actions. The above algorithms have been described in their *greedy* formulation; that is, the action that leads to the state with the highest utility is always selected to be taken. In practice, this action selection mechanism rarely converges to the optimal policy because of the limited portion of the state space the agent explores. However, executing a non-greedy action while the agent is training can provide information about states that could possibly lead to higher rewards in the future. There are several selection methods that allow for a balance between exploration and exploitation including using the Boltzmann distribution to probabalistically select actions, and ϵ-greedy approaches that involves selecting a random action with probability ϵ. These

action selection methods should be greedy in the limit of infinite exploration (GLIE); that is, the method should eventually become greedy so that the agent is able to choose the optimal actions to take in each state[16].

Q-learning is an off-policy control method, meaning that the state with the highest Q-value used to update the equation may not necessarily be taken in the next step of the algorithm. This is because of the necessity for exploration in training described above; exploration in training requires not executing the optimal action used in the Q-value update equation. This has a number of implications with respect to convergence when Q-values are represented by a function approximator even though Q-learning has the best convergence guarantees of all control methods[18].

Finally, the utilities or Q-values must be represented in such a way that the information can be recalled and applied to situations that were not present in the training set. One method to represent these utilities is through lookup tables, where each entry in the lookup table represents $U(s)$, or in the case of Q-learning, $Q(s, a)$. For problems that are particularly complex with many states and many actions that could be performed in a state, the size of these lookup tables quickly makes this representation infeasible because each possible entry must be known. In these cases, function approximators such as neural networks are extremely useful because they are able to approximate the underlying functional form at a set of data points and also provide generalization.

2.2 Cognitive Architecture: Soar

Automated learning can also be accomplished by using a production system, which uses rules to govern the agent's interaction with the environment such that goal-based behavior can be achieved. Soar is an example of such a learning agent. Soar an open-source software, and is often termed a cognitive architecture because its design seeks to utilize the mechanisms and structures that underly human cognition to imitate human problem solving and reasoning. Thus, the architecture also allows for learning and augments the traditional view of memory. In this context, learning encompasses figuring out the best course of action when the agent must decide which of two or more seemingly reasonable operators to execute.

In general, the Soar architecture solves problems by sequentially applying an operator to the current state in order to reach a successive state. Operators are essentially the actions that cause a state to transform into another state, and requires an understanding of the current state,

the productions that can be fired to propose an operator, and if many operators are proposed, the *best* choice. To this end, Soar has a number of memories to encompass these essential features, but also several forms of non-traditional memory including semantic memory, which attempts to make sense of structures and episodic memory, which allows the agent to remember situations that it has encountered in the past. Representation of information in memory, the method of rule application, learning, and practical implementation issues with Soar will be introduced in the following sections. More information can be found in [8].

2.2.1 Memory and Information Representation

There are three main types of memory in Soar: long-term memory (LTM), working memory (WM), and preference memory (PM). LTM is organized as productions, which are a set of rules based on logical implication that dictate what steps to take in light of the conditions of the current situation. From the perspective of traditional programming languages, productions can be considered as if-then statements. In Soar, there are four different purposes of productions: to fire operator proposals, to compare operators, to apply operators, or to provide elaborative information about the state. At least two productions are necessary to implement an operator: 1) to propose the operator, and 2) to apply the operator. Thus, in the case of block rotation for example, two productions could be formulated: "IF there is an available block X, then propose operator `rotate-block(X)`," and "IF `rotate-block(X)` operator is proposed, THEN execute block rotation action on the environment."

The second type of memory is working memory, which contains information about the current situation or current problem that needs to be solved, and is organized into sets of working memory elements (WMEs), or augmentations of the state. WMEs are identifier-attibute-value triples, where an identifier is Soar's internal representation of the object being described, attributes are qualities or descriptors of the object, and values are the values of the descriptor. All WMEs are required to be connected to the state to be a part of Soar's internal representation of the problem state. An example of Soar's representation of the state is shown in Figure 2.2. As illustrated here, all the WMEs are connected to the state directly or indirectly. As shown, the `input-link` and `output-link` elaborations are connected to the state through the `iol` identifier, which functions as both an identifier for these two WMEs and as a value of the input/output attribute of the state. An example of information present on the `input-link` for this problem is also shown; the goal map, state map, and the blocks are all attributes of the `input-link`.

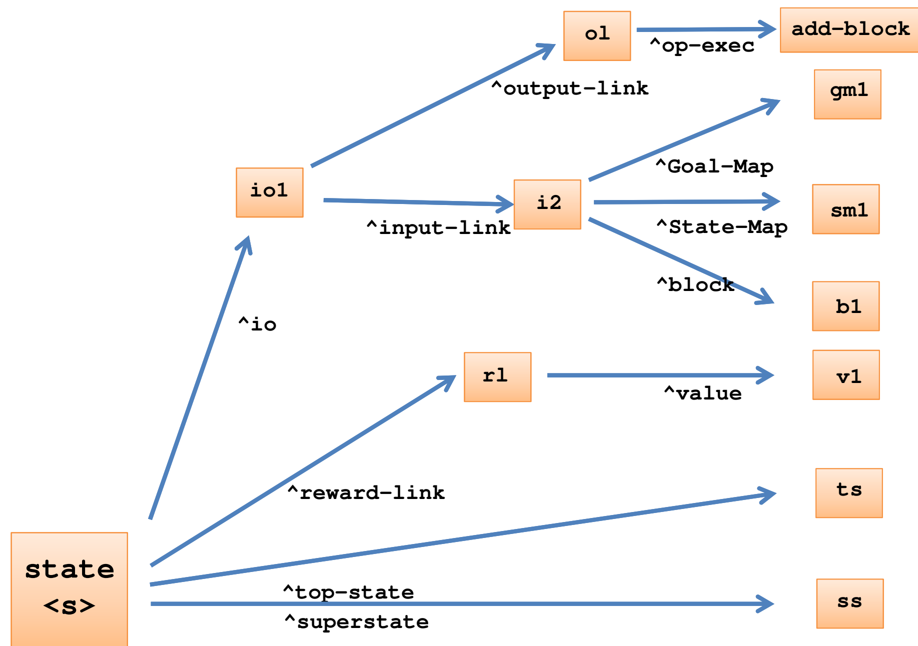


Figure 2.2: Soar's State Representation by Linked Working Memory Elements

The last type of memory is preference memory, which provides the framework by which operators are selected. PM contains the numerical utility of certain operators, or information about how well one operator compares to another. The impact of PM on reinforcement learning is described further in Section 2.2.3.2. Also, as stated above, semantic and episodic memories exist, which allow for the idea of structure and recalling events of the past. These most resemble WM, in that they are composed of WMEs which describe situations encountered in the past or how objects are structured. Together, these types of memory encode the knowledge necessary to make a choice about the best action to take based on the current situation.

2.2.2 Soar Decision Cycle

The Soar Decision Cycle is shown in Figure 2.3, and utilizes WM, LTM, PM, and the other types of learning in order to make an informed decision about which operators to select in certain states. The cycle has 5 steps - input, proposal, decision, application, and output - and is repeated until the current goal is reached. In finality, the mechanism by which Soar determines an operator to apply is this: the input stage accesses the `input-link` to obtain the state of the environment, which

is then placed in WM. Based on this current state, productions which are applicable based on their preconditions are proposed in the proposal stage. If one or more productions are applicable, they are fired. Productions are fired and retracted to interpret new data (state elaboration) or compare operators (operator comparison) as mentioned above. This occurs until *quiescence*, when there are no more complete matches or retractions of productions.

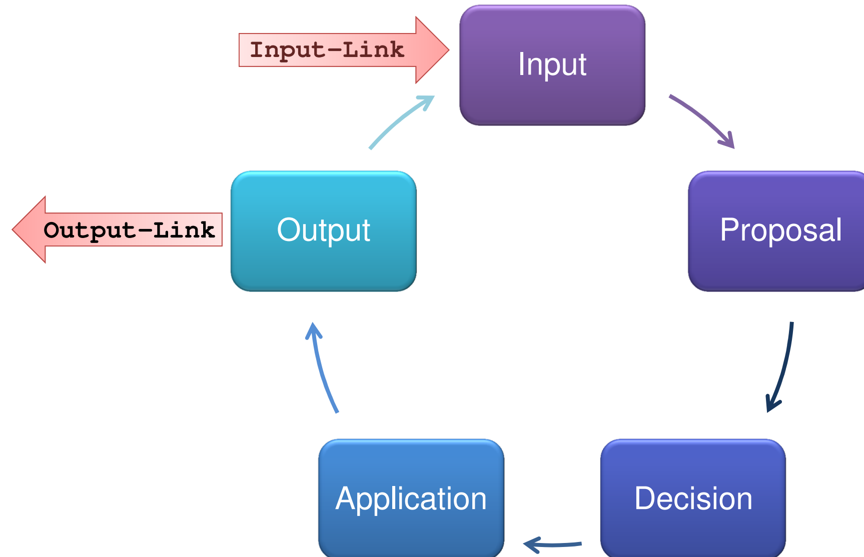


Figure 2.3: Stages of the Soar's Decision Cycle

The decision stage allows for an operator to be selected from those proposed. If a decision cannot be reached because no operator is proposed, there are several operators proposed, or other situations that impede decision making for the system, an impasse is generated. Otherwise, the system proceeds to the application phase, where productions fire to apply the operator (operator application) to change the state.

The final stage is the output stage, in which the products of the cycle can be output to the environment using the `output-link`. These five stages are applied repeatedly to successive states to achieve the goal-directed behavior of Soar.

2.2.3 Learning with Soar

Soar has several different mechanisms and approaches to learning. As stated above, under conditions in which a decision cannot be made or enough information does not exist, Soar reaches an *impasse*, which is considered by Soar to be an opportunity for learning. There are several types of impasses:

Tie impasse Impasse that occurs when two or more operators have the same preference

Conflict impasse Impasse that arises when two operators are mutually better or worse than each other (ex. O1 is better than O2; O2 is better than O1)

Constraint-failure impasse Impasse resulting from two or more operators with preferences indicating that they both *must* be chosen in the current cycle; can also be generated when an operator has preferences that state that it must both be chosen and not be chosen if the goal is to be achieved

No-change impasse Impasse generated when either a new operator cannot be chosen for a particular state (state no-change impasse) or an operator is selected, but no productions describe its application (operator no-change)

When an impasse is reached, a *substate*, a new state which contains the entire representation of the current state, is created within the state that generated the impasse. In this new state, the agent is able to perform a task called *subgoaling* by testing operators to determine which one should be selected. Impasses can be encountered in this substate, and therefore nested, as the same impasse resolution strategies will be employed in order to resolve these lower impasses. The result of these impasse resolution efforts is the creation of new WMEs that must be linked the state in which the impasse was generated.

2.2.3.1 Chunking

Although subgoaling resolves the impasse for the current state, what happens when the same state is encountered again in the future? The ideal situation is for the agent to repeat its past actions without again subgoaling to determine them. This can occur in Soar if learning is enabled, and the mechanism to do so is referred to as *chunking*. Chunks are new productions that are added to LTM, and are generated by the agent. After impasse resolution, the agent considers the actions

that led to the impasse, and sets them as the conditions of the production. It then sets the steps that resolved the impasse as the consequent of the production. Thus, the agent has learned how to solve the problem based on its experience from the past.

2.2.3.2 Reinforcement Learning in Soar

Other types of learning have also been incorporated into the Soar architecture, including reinforcement learning, episodic learning, and semantic learning. Although the latter two will not be addressed, the reinforcement learning mechanism allows for the incorporation of Q-function based RL methods into Soar (Soar-RL). In the Soar architecture, this directly impacts PM, as RL will change the preference values for the operators. This makes sense; the purpose of the Q-function in reinforcement learning is to learn the utility of performing certain actions in certain states in order to find the best actions to execute given the state. The reward is processed through the `reward-link` attribute of the state.

To create operators with Q-values as preferences, one additional production is necessary. This production dictates the portion of the state to be recalled for s in $Q(s, o)$ values. Then, PM contains the $Q(s, o)$ values for each state-operator pair.

Soar implements a a similiar approach for mapping state and actions to Q values, as seen by the notation above, and instead defines the utility of *operators* applied in certain states ($Q(s, o)$). $Q(s, o)$ calculations and updates are performed in the following manner: first, since the same operator can be proposed in different situations, Soar calculates the utility of executing a certain operator as a linear combination of all the proposed operator preferences for that specific operator. Second, when the $Q(s, o)$ values are updated after a reward is received, this update is divided among all of the operators of the type that was applied. More information concerning this implementation can be found in [11] and [8].

The system allows for designer control over parameters of the algorithms that can be chosen. Soar-RL allows users to choose between Q(λ) ($\lambda = 0$ for Watkins Q-learning) and State-Action-Reward-State-Action (SARSA) RL algorithms; from Boltzmann, ϵ-greedy, softmax, and deterministic action selections as well as the temperature T or ϵ parameters of these action selection mechanisms; and the value of the discount factor, learning rate, and other parameters. It also provides functionality for assessing the progress of the RL Q-values.

2.2.4 Communicating with the Soar Kernel

As briefly mentioned in Section 2.2.2, Soar can be used as the decision-making agent within an external environment by the `input-link` and `output-link` attributes of the state. In this instance, the Soar decision-making engine is referred to as the Soar kernel, and can be used to interact with simulated environments, robots, and any inputs from the external world. Both the `input-link` and the `output-link` have and accept the same WME structure as working memory. Communication between the kernel and an environment is most effectively mediated through the Soar Markup Language (SML), an interface implemented in C++.

2.3 Artificial Neural Networks

2.3.1 Architecture

Artificial neural networks (ANNs) are computative networks that are based on the structure of biological neurons, and are used to approximate functions. The most basic structure of an ANN is a perceptron which performs the mapping

$$o = f(\mathbf{w}^T \mathbf{x} + b),$$

where $\mathbf{w}$ is the vector of weights, $\mathbf{x}$ is the input vector, b is a scalar bias, and o is the output of the perceptron. This computation is performed by two processes shown in Figure 2.4 as the calculation of a net activation by a weighted combination of the inputs, and the computation of the output by use of a mapping function on the net activation. The mapping function between the net activation and output can be implemented by a number of functions, including sigmoid and logistic functions, hard delimiters, and linear functions.

The multilayer feedforward ANN (MLFF-ANN) is designed by assembling perceptrons into a network, which is shown in Figure 2.5. Each of the units represents one of the perceptrons in Figure 2.4. Each directed arrow in the figure represents a weight between the output of that unit and the input of the recipient unit. The MLFF-ANN architecture is characterized by each output o_i from a unit is connected as input to each of the units in the following layer. Additionally, there are no self-connections (the output of a unit connected as its own input) or look-ahead connections (output of one unit connected to the input of a non-adjacent layer). The layers of MLFF-ANNs include

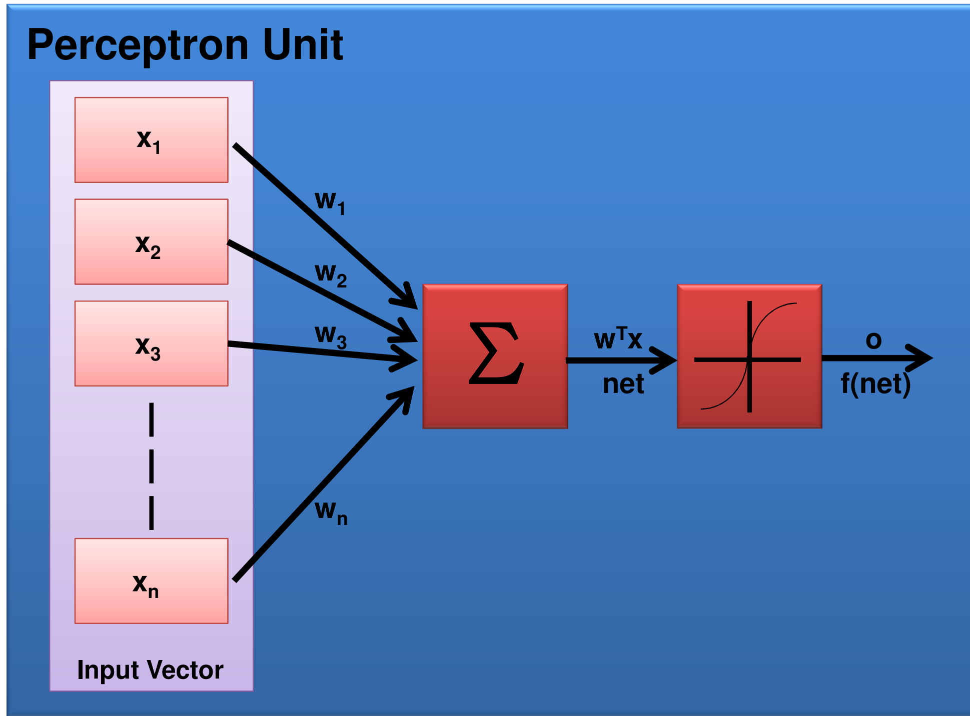


Figure 2.4: A Single Unit of an Artificial Neural Network

the input layer, which recieves an input vector $\mathbf{x_i}$ of features from the training set $\mathbf{X}$; one or more hidden layers, and an output layer. These latter two layers perform the activity of the perceptron on the vector of information output from the previous layer. The MLFF-ANN architecture shown above is a 3x4x1 neural network, and the number of units in each layer, as well as how many hidden layers should be used varies according to the purpose of the network.

Multilayer feedforward networks are able to implement the following mapping:

$$F(x) = \sum_{i=1}^{N} \alpha_i \phi(w_i^T x + b_i),$$

where $\phi(\cdot)$ represents the activation function. According to Cybenko's theorem, the universal approximation theorem, a standard MLFF-ANN with a single hidden layer and a finite number of hidden neurons and arbitrary activation function is a universal approximator, and therefore can be used to approximate continuous functions of n real variables with support in the unit hypercube, although convergence is conditional based on the number of units in the hidden layer and the

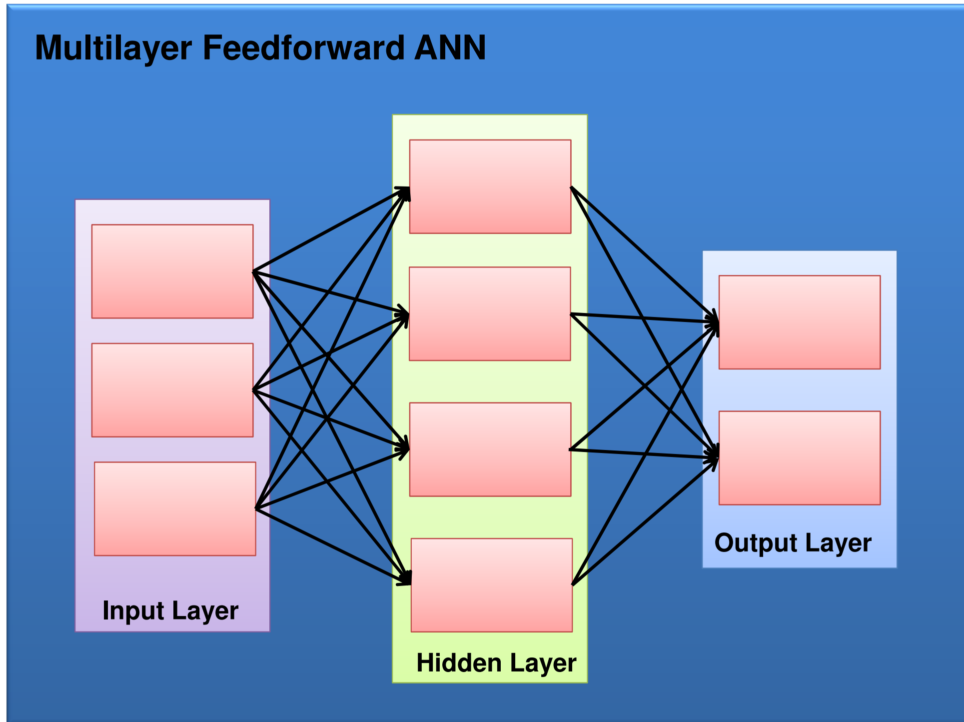


Figure 2.5: Structure of a Multilayer Feedforward Artificial Neural Network

properties of the function being approximated[2].

2.3.2 Network Training and the Generalized Delta Rule (GDR)

The network is trained by a method similar to the Widrow-Hoff learning rule[25] which involves repeatedly presenting the network with an example $\mathbf{x_i}$ from representative set of examples $\mathbf{X}$ from the problem space, and tuning its outputs to match the correct or expected response as dictated by the training set. Training the network is achieved by backpropagation of error by adjusting the weights of the neural network towards towards the desired output. After initializing the weights in the network, the following steps are taken for each $\mathbf{x_i}$ until an acceptable level of error has been reached:

1. Forward propagate $\mathbf{x_i}$ through the network to determine network output $\mathbf{o}$
2. Calcalate error $\mathbf{E}$ as a difference between output $\mathbf{o}$ and target $\mathbf{t_i}$ on norm N such that $E = ||\mathbf{t_i} - \mathbf{o}||_N$

3. Backpropagate error using gradient descent to change network weights to minimize allowable error

The first two steps in this process are straightforward; the third step requires more consideration. This minimization is achieved by first using gradient descent to determine the direction of descent that corresponds to minimizing the function error. Since a function is encoded by the weights of the neural network, these weights must be changed to reflect this change, which is achieved by the Generalized Delta Rule (GDR)[14]. GDR calculates the weight adjustment (Δw_{ji}) based on the error with respect to a specific weight ($-\partial E/\partial w_{ji}$), and is

$$\frac{\partial E}{\partial w_{ji}} = \frac{\partial E}{\partial o_i} \cdot \frac{\partial o_i}{\partial net_i} \cdot \frac{\partial net_i}{\partial w_{ji}}.$$

Since this differs for different units based on their layer location, the parameter δ is calculated as:

$$\delta_j = (t_j - o_j) f_j^{'}(net_j) \text{ for output units, and}$$

$$\delta_j = f_j^{'}(net_j) \sum_n \delta_n w_{nj} \text{ for internal units.}$$

Then, the weight correction for each weight in the network is:

$$\Delta w_{ij} = \epsilon \delta_j \tilde{o}_i,$$

where $\tilde{o}_i$ is o_i if this i^{th} input to the unit is the output of another neuron (in the case of output layers, for example), or i_i if the input to the unit is from the input layer. In addition, $\delta_n w_{nj}$ represents the δ_n from the next layer (L_{n+1}), w_{ji} represents the weight of the connection from unit i to unit j, and a single subscript (as in w_n) refers to a specific unit[17]. As the network learns, the error decreases between the network output and the function to be approximated. The ideal error function as the network trains is shown in Figure 2.6, and reflects the agent initially knowing nothing, and eventually settling on a policy as training proceeds.

It is also important to note that this formulation assumes training by pattern, which means that the network weights will be updated after every forward propagation of the inputs. Training by epoch can also be achieved by accumulating all the error over one pass of the entire training set, and then correcting the weights afterwards. This helps to eliminite oscillation to speed convergence.

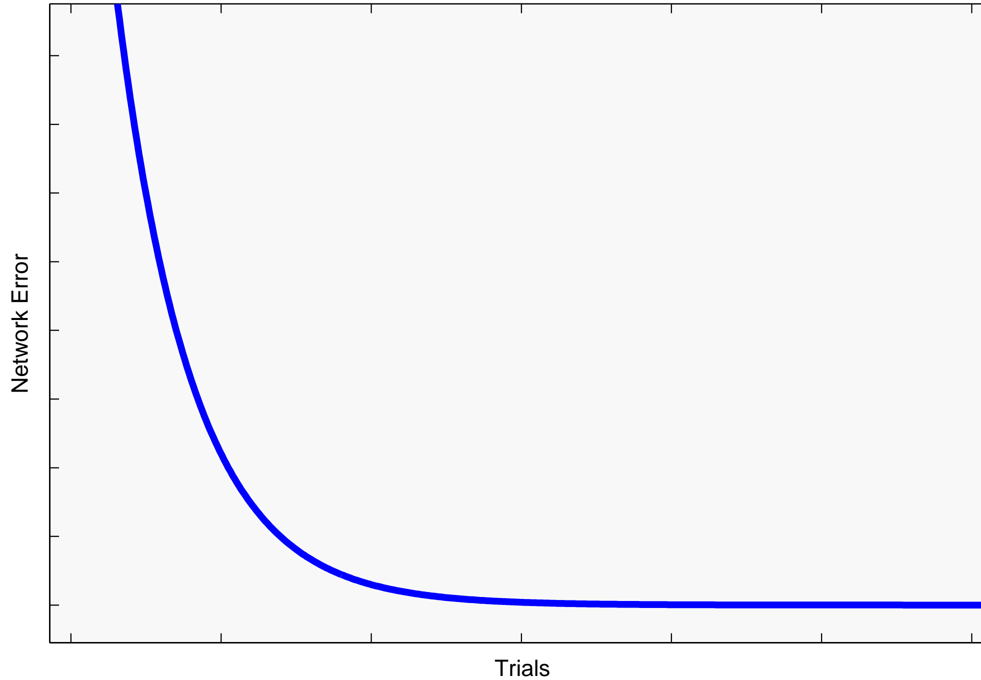


Figure 2.6: Ideal Training Error As Training Proceeds for Learning Agent

Other methods to improve the performance of the network include weight decay[7] and second-order techniques such as momentum. Lastly, minimizing the network error to zero can lead to "over-training" or "overfitting," in which the network essentially memorizes the noise present in the training set which can lead to poor generalization[17].

2.3.3 Q-learning Neural Networks

As stated in 2.3.1, MLFF-ANNs with at least one hidden layer are universal function approximators, and are therefore able to represent functions with complex mappings based on a training set composed of examples of the function the network is expected to learn, including Q^*, the optimal Q-function approximated by the Q-values. Q-learning neural networks have a slightly different training prescription from standard MLFF-ANNs in terms of network topology, training set generation, and feedback.

First, Q-learning neural networks have three different possible forms: single-NNs, which

have one network that maps input states to the Q-value for |A| actions as in Figure 2.7; multi-NNs, which are composed of |A| networks that map the input state to a single Q-value representing a particular action as in Figure 2.8; or one neural network that maps the state and action as a pair to |A| Q-value outputs. Only the first two architectures are explored in this paper.

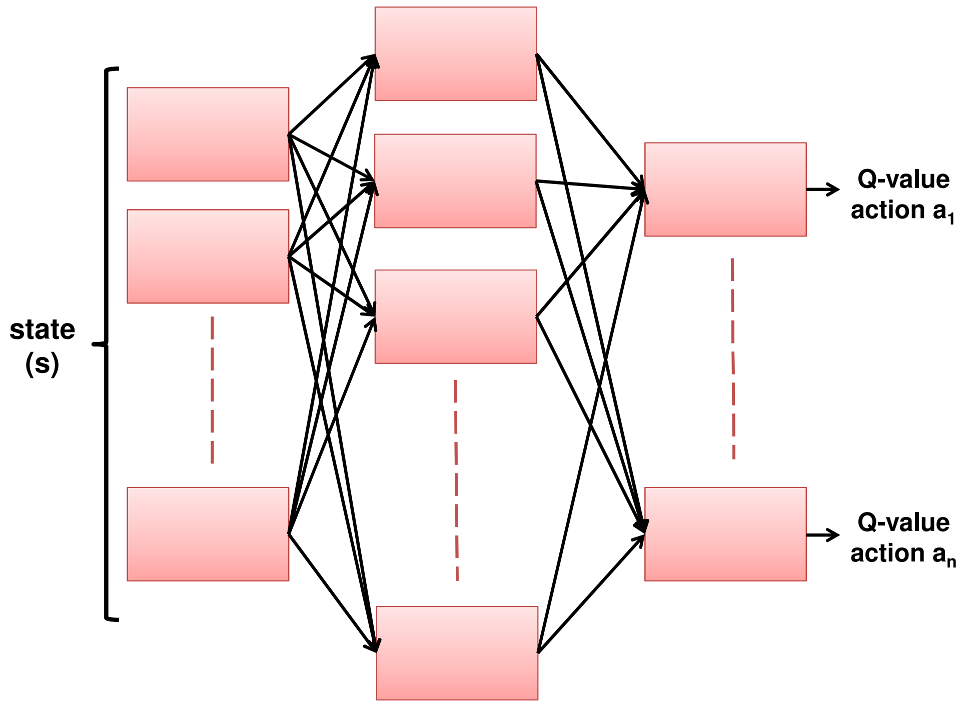


Figure 2.7: Single-NN Q-learning Architecture

Secondly, since the state changes as a response to the action taken, training can proceed along a specific trajectory, and although this is not necessarily required for convergence[23], this greatly simplifies training. Thus, the training set can be composed of a number of starting positions, and the rest of the training set is generated dynamically as the agent takes actions which result in arriving in the next state. This illustrates the importance of the Markov independence property; without this property, it is impossible to make an informed decision about the next action to take if the entire sequence history is not present.

Finally, although the weights of the network can be changed with the same use of gradient descent and GDR, there are no "correct" answers given as part of the training set; only immediate rewards and/or terminal rewards are applied as feedback to the system. These rewards are used to

update the Q-value approximation for state-action pairs. The full algorithm for neural Q-learning is given in Section 4.2.

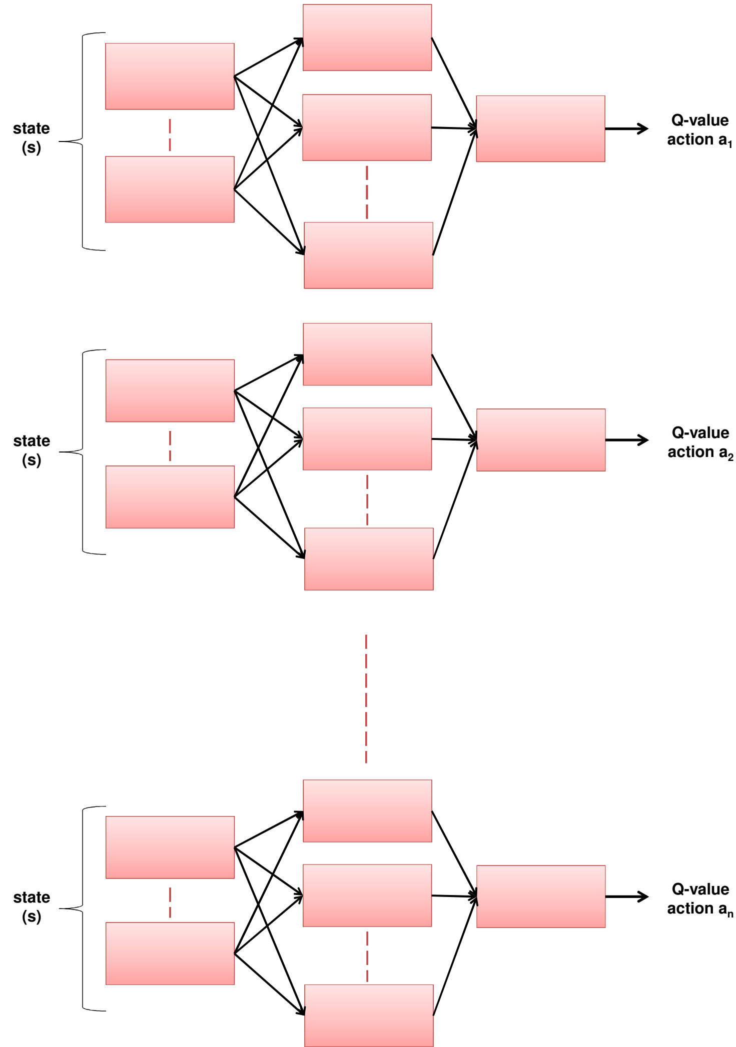


Figure 2.8: Multi-NN Q-learning Architecture

One well-known problem with implementing Q-learning with linear function approximators is their failure to converge to an optimal policy with probability 1 for all conditions, whereas storing Q-values with the tabular method, on the other hand, is guaranteed to always converge with probability 1[22]. This is because the error backpropagation produces non-local changes in the estimation of Q^* so that changing certain weights of the network can affect the Q-values of other state-action pairs. Since the convergence of Q-learning is based on performing local changes to decrease the

error, these non-local changes can cause divergence, and is referred to as interference[24].

Furthermore, off-policy bookstrapping methods including Q-learning are known to diverge because of the conflict between distributions used to update the Q-values and those encountered by the state trajectory[18]. [10] defines conditions for both the problem and the algorithm such that Q-learning implemented by linear function approximators is guaranteed to converge with probability 1. A recommendation in [18] is to use selection methods such as ϵ-greedy to satisfy at least one of these constraints, which aids in convergence. Nonetheless, Q-learning neural networks are not guaranteed to converge for all topologies, learning rates, and input representations of the neural network, and Q-learning with non-linear function approximators currently has no convergence proof or guarantee[21], [9], [15].

2.4 The Fourth Edition of the Wechsler's Intelligence Scale for Children (WISC-IV)

The WISC-IV is a test used to assess the intelligence quotient (IQ) for children between the ages of 6 and 16. The WISC-IV has fifteen subtests in order to provide a full assessment of a child's cognitive ability. These tests provide an overall assessment of the child's intelligence, as well as individual scores reflecting the child's propensity for verbal comprehension, perceptual reasoning, processing speed, and working memory[3].

The intelligence component explored in this thesis is perceptual reasoning which is partially tested by the Block Design (BD) subtest. The BD subtest is also a test of general intelligence. In this test, children are presented a pattern of red and white blocks, and asked to reconstruct this model image with a set of blocks. The students are assessed according to the speed in which they complete the puzzle as well as the difficulty of the puzzle completed[3]. The BD subtest was also chosen as the basis for the problem because it measures synthesizing ability, tests non-verbal organizational ability, and corresponds to problem-solving ability[4]. A related task is the Block Design Multiple Choice (BDMC) subtest, in which children do not physically move the blocks given to reconstruct the puzzle image; rather, they choose the correct constructed design from four alternatives[12].

WISC-IV is a well-known test used to assess and provide intervention for children with attention-deficit/hyperactivity disorder, gifted children, mentally challenged or intellectually disabled children, children with autism spectrum disorders, and emotionally disturbed children [12].

Chapter 3

The Problem

3.1 Considerations of the Block Design (BD) and Block Design Multiple Choice (BDMC) Subtests

Section 2.4 describes the WISC-IV BD and BDMC tests and their interpretation as an intelligence assessment. However, neither one of these subtests completely allows for testing the intelligence of an automated agent. The BD subtest measures non-verbal reasoning and problem solving through requiring the agent to construct the goal map from given blocks. However, the WISC-IV implementation of BD also requires that the agent physically move the blocks to recreate the goal map, and therefore visual-motor integration, motor planning and control, and visual-spatial processing is also a component of performance. None of these factors are of importance in this thesis.

On the other hand, the BDMC subtest does not require any physical embodiment, and thus allows the agent to match the correct design based on multiple already constructed alternatives. Although this eliminates the necessity for motor movement or visual processing, the reasoning and problem solving required for BD is lost.

3.2 Block Design - Disembodied Formulation

In order to preserve the benefits of the BD and BDMC subtests while removing the undesirable qualities, the amended Block Design - Disembodied (BD-DE) test was formulated. The

BD-DE test used in this thesis requires the agent to successfully construct the target goal map using the blocks available in the WISC-IV BD subtest, but without the physical requirement of visual or motor coordination.

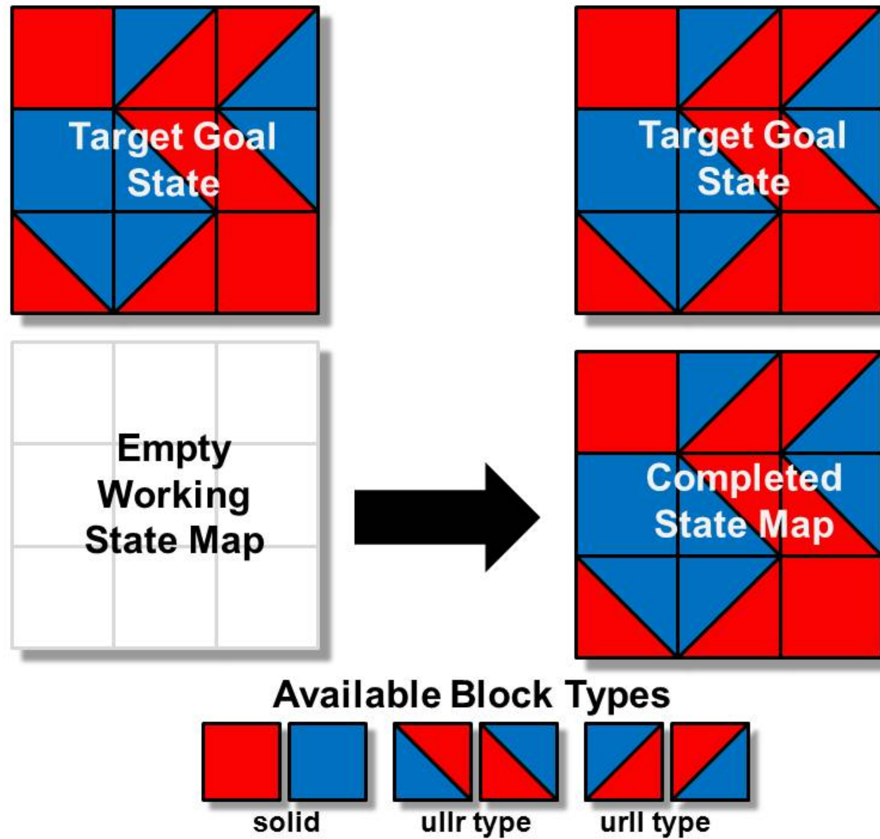


Figure 3.1: Desired Functionality of Automated Agent on BD-DE Subtest

Figure 3.1 shows an example of the desired functionality of the agent. The agent populates an empty state map of size n by selecting from b blocks, each of which has 6 possible types. This example depicts the 6 block types that the agent will select from to populate the working map such that it matches the goal map. The neural Q-learning agent will solve this problem, whereas Soar will be explicitly provided the 9 blocks needed to solve the puzzle. To achieve the goal state, the agent's actions must converge to an optimal policy π^*, described further in Chapter 2, that allows the agent to arrive in the corrrect terminal state s_T^* from initial state s_0 in the minimum number of steps at 100% accuracy. The constraints and assumptions on the problem for each of the architectures is described below.

3.3 Assumptions and Constraints

Both Q-learning implementations require a number of assumptions and constraints to effectively solve the problem. The common assumption is that the agent is aware of the constraints in solving the puzzle. These two constraints are:

1. Blocks may not be added to locations on the map that are currently occupied, and

2. Blocks may not be removed from locations on the map that are not currently occupied.

The effect of this is that the agent does not have to learn the physical rules surrounding the game, but only needs to learn the best practices of the game itself.

3.3.1 Additional Soar Constraints

In addition to the assumptions and constraints above, Soar also employs another constraint: only blocks that are not currently positioned on the map may be rotated.

3.3.2 Additional NN Constraints

The neural network additionally assumes that the agent must select from six blocks for each of the squares on the map, and does not represent each block individually. In fact, blocks are not explicitly modelled in the neural network, they are simply assumed to be present.

Chapter 4

Methodology

4.1 Soar

This thesis employs Soar 9.3.0 to solve the BD-DE problem described in Chapter 3. To this end, the Soar kernel is placed in a simulated BD-DE game environment to learn how to correctly solve the proposed puzzle based on rewards it gathers during the duration of play.

The game environment is programmed in C++, and interacts with Soar through the Soar Markup Language (SML) interface. The basic training of the agent is as follows: first, an arbitrary goal map and empty working state map is presented to Soar. Using the productions, operator preferences, and rewards it gathers, it adds, removes, and rotates blocks until it enters a terminal state s_T such that n blocks have been placed on the map. A reward from the environment is applied to the system based on the correctness of the puzzle, and the agent is then trained on a number of different puzzles and block sets. During this time, it employs Soar's reinforcement learning module to implement Q-learning with the intent that its sequence of actions will converge to the optimal policy.

After the agent completes the training set a predefined number of times, the agent's knowledge is then tested by disabling learning, setting all exploration probabilities to zero, and proposing puzzles that the agent has not yet seen. This defines the agent's performance. This functionality required interfacing with the Soar kernel through the `input-link` and `output-link`, identifying a WME structure representing the game state, encoding long-term knowledge through the use of productions, and identifying a useful reward system. These considerations will be described in the

following sections.

4.1.1 Interfacing with the Soar Kernel

The game environment provides the framework in which Soar operates; it is responsible for initializing and configuring Soar, creating and maintaining of the problem state, and instructing Soar about when to make its decisions. Figure 4.1 illustrates this relationship and the flow of the program. As shown in the figure, the environment first creates and initializes the kernel, loads all the productions encoded into long term memory, and registers with Soar to be informed of the addition of any information onto the `output-link`. One of the productions also contains configuration instructions, which enables reinforcement learning, sets the learning policy to Q-learning, and sets the action selection method to ϵ-greedy.

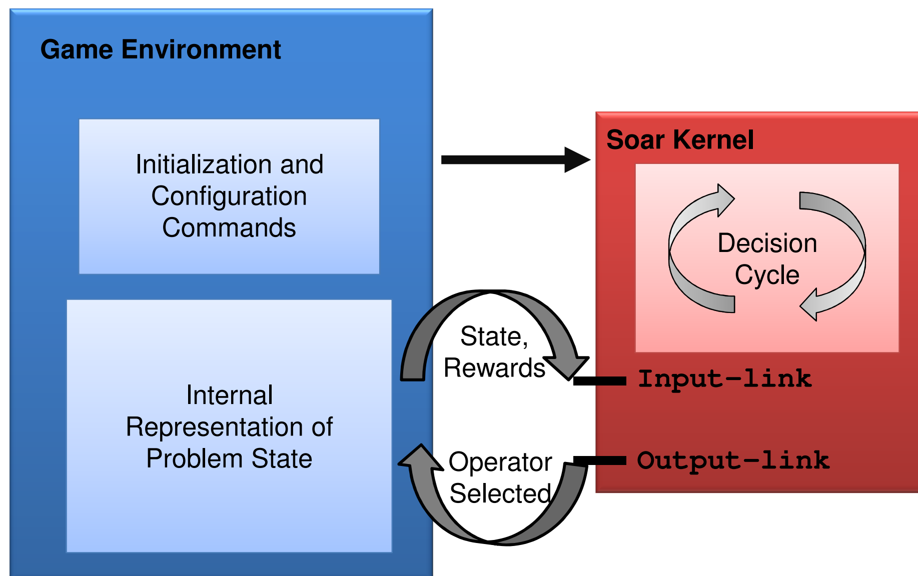


Figure 4.1: Game Environment Relationship with Soar Kernel and Program Execution

Then, training begins. The game environment is preprogrammed with a 11 training sets, the first of which is loaded onto the `input-link`. The game environment then instructs Soar to run until 9 blocks have been placed on the map, which is signalled by a production in Soar which fires an interrupt to halt kernel execution and return control to the game environment. Each operator

that Soar chooses as it solves the puzzle is placed on the `output-link`, which notifies the game environment to update the state of the puzzle based on Soar's decision. After this state update, the environment returns the reward on the `input-link`. One of the productions in long-term memory allows this reward on the `input-link` to be transferred to the reinforcement learning elaboration of the state WME. Soar then processes this reward by updating the preference of the selected operator based on this reward. This cycle proceeds until Soar is able to solve all the training sets in less than 1.5×the minimimum number of steps. When this condition is met, the action selection method is then changed to greedy (ϵ set to zero). This cycle proceeds for all the training sets until Soar is able to solve all the training sets in approximately the minimum number of steps. Then, the agent is tested on 5 test sets to assess its generalization capabilities.

This programming decision to force the game environment to change and update the state instead of the action side of the operators in the productions is due to the implementation of WME ownership for Soar and the game environment. Both the game environment and Soar load information into working memory with the same structure, and so a distinction is made between what entity placed the WME into Soar's working memory - Soar, or the game environment. Because the only entity able to change or delete the WME is the entity that placed it in WM, the update strategy shown in Figure 4.1 was adopted.

4.1.2 State Representation and Working Memory

The state of the system is fully known if the configurations of the goal and working state puzzles are known, and the orientation and usage of each of the blocks present is known. This information is initially constructed in the game environment, and is placed on the `input-link` of Soar's state representation. The objects represented and attributes used to describe them are shown in Table 4.1. The reward and goal attributes of the state do not have additional elaborations, but are simply attributes of the `input-link` to describe these additional features of the state.

4.1.3 Production Memory

There are 15 productions that implemented the 5 main operators for addition, removal and rotation of blocks to solve the puzzle. Each operator requires a propose, apply, and RL feature assocation production corresponding to the proposal, decision, and application phases of the Soar

Object	**Attributes**	**Values**
Goal Map	Square	Soar Identifier ID
Square (of Goal Map)	Block Type Split type Solid color UR color UL color LR color LL color	{solid, mixed} {UR-LL, UL-LR, na} {red, white, na} {red, white, na} {red,white, na} {red, white, na} {red, white, na}
State Map	Square	Soar Identifier ID
Square (of State Map)	Name State Block	{Square-{1,2,..., n}} {open, closed} {Block-{0,1,...,b}}
Block	Name Type Name Type Split type Solid color UR color UL color LR color LL color Used	{Block-{0,1,..., b}} {Block-{0,1,2,3,4,5,6}} {solid, mixed} {UR-LL, UL-LR, na} {red, white, na} {red, white, na} {red,white, na} {red, white, na} {red, white, na} {true, false}
Reward	*	$\mathbb{R}$
Goal	*	{true, false}

Table 4.1: Soar's Representation of the Problem State

Decision Cycle. These operators and their descriptions are outlined below:

- `add-block-to-map-matching` and `add-block-to-map-not-matching`

  Propose To propose this operator, the current state map must have an open square, there must exist a block that has not yet been used, and the block characteristics must match (or not match for `add-block-to-map-not-matching`) the goal square characteristics.

  Apply The application of this operator results in a command being placed on the `output-link` stating the name of the operator, and which block should be moved to what location on the current map. The game environment performs this update to the state.

  RL The preference of this operator is set based on the rewards returned from the game environment. The RL rule corresponding to this operator recalls the attributes of block and the attributes of the goal-map location corresponding to the location at which the block was placed on the working state map. Thus, the agent learns the utility of adding

a block with certain attributes in a position corresponding to locations on the goal map that have (or do not have) the same attributes.

- `remove-block-from-map-matching` and `remove-block-from-map-not-matching`

  Propose This operator is proposed if there is a location on the current map that has been filled by a block and the block characteristics match (or do not match for `remove-block-from-map-not-matching`) those of the goal map characteristics.

  Apply The application of this operator requires a command to be placed on the `output-link` with the block to be removed from the map and the square from which it is to be removed. The game environment then performs this update to the state.

  RL The preference of this operator is set based on rewards garnered from the game environment. The RL rule corresponding to this operator recalls the attribute of the block and the attributes of the goal map location corresponding to the location at which the block was removed from the current map. Thus, the agent learns the utility of removing a block with certain attributes from a position corresponding to locations on the goal map that have (or do not have) the same attributes.

- `rotate-block-90-cw-urll`, `rotate-block-90-cw-ullr`, and `rotate-block-90-cw-solid`

  Propose These operators can be proposed for any of the blocks that have not been used to solve the current puzzle, and if the working map has an open square. Each operator is proposed based on the characteristics of the block types shown in Figure 3.1.

  Apply The application of these operators results in commands placed on `output-link` describing which block should be rotated.

  RL The preference of this operator is set based on rewards given by the game environment. The RL rule corresponding to this operator recalls the attributes of the block that was rotated, and the attribute of the goal square corresponding to the open square. The goal is for the agent to learn the utility of rotating blocks of a certain type based on the characteristics of the goal map corresponding to an open square on the working map.

- `end-task`

  Propose This operator is proposed when the goal attribute of the input-link has been updated to true. This means that n blocks have been placed on the puzzle.

Apply The application of this operator results in an interrupt being generated by Soar, returning control to the game environment.

RL There is no RL rule required for this operator; it is implemented only so that the preference of the operator applied in the previous state can be updated. The preference for this rule supercedes the preferences for the other operators so that the game is guaranteed to end after a terminal state is reached.

In addition, impasse resolution productions were written because of the `rotate-block-90-cw-solid` operator. Whenever a solid block is rotated, this leads to no measurable change of the state, and therefore, according to Soar, nothing has changed. This leads to a state no-change impasse. Impasse resolution was achieved by lowering the preference of that operator in that step such that other operators could be chosen such that they would not be chosen if other options existed.

4.1.4 Rewards, Parameters, and Test Sets

The experiment of this section consists of determining the efficacy this architecture by testing its ability to model individual blocks. Thus, the performance of Soar's add- and remove-block functionality is tested, and then augmented by the rotate operators to determine if additional functionality over the neural network can be implemented by Soar.

The parameters used to train and test the agent were the default settings - the learning rate was set to 0.3, and ϵ to 0.1. The reward function used was

$$r(s) = \begin{cases} -0.2 & \text{if block added to square matched} \\ -0.45 & \text{if a block was rotated} \\ -0.65 & \text{if a non-matching block was removed} \\ -0.75 & \text{if a matching block was removed or a non-matching block was added} \\ +2 & \text{if the puzzle was completed correctly} \\ -2 & \text{if the puzzle was completed incorrectly} \end{cases}$$

Each network was tested on 5 test sets. The test sets for the non-rotation implementation were 2 a standard 3x3 puzzles with 9 blocks, one 3x3 puzzle with 4 times as many blocks as squares, a 4x4 puzzle, and a 4x4 puzzle with 2 times as many blocks as squares. For the rotation task, this

was slightly amended, where the second and third test sets required rotation if the task was to be completed.

4.2 Neural Network

A number of different network topologies and architectures were investigated as solutions to the BD-DE subproblem because of the known convergence issues of Q-learning neural networks. The networks differed in overall architecture (single-NN versus multi-NN), input representation, portion of the map represented, activation function, and in error measurement. The algorithm below was used to train each of the investigated neural network topologies, listed in Table 4.2.

Algorithm 1 Training prescription for Q-learning neural networks

Initialize weights to random numbers ϵ [-0.5,0.5]
repeat
  Obtain new training example and initialize current puzzle state to empty (clear board)
  while $s \neq s_T^*$ (n squares correctly matched) **do**
    Forward propagate current state to network to get output Q-values
    Select and execute a based on ϵ-greedy to reach s'
    Receive r
    if $s' = s_T^*$ **then**
      Generate Q^{target}as $Q^{target} = r(s')$
    else if $s' = s_T$ **then**
      Generate Q^{target}as $Q^{target} = r(s')$
      $s' = s_0$
    else
      Generate Q^{target}as $Q^{target} = r(s') + \gamma \max_{a' \in A} Q(s', a')$
    end if
    Backpropagate $\partial \mathbf{E} / \partial \mathbf{w_{ji}} = \|\mathbf{Q^{target}} - \mathbf{Q(s, a)}\|_N$
    $s \leftarrow s'$
  end while
until network can solve each puzzle in n steps

4.2.1 State Representation

Chapter 3 described the assumptions and constraints placed on the neural network, including the disuse of explicit block representation in the neural network state. Thus, the input to the neural network is the target configuration of the goal map and the current configuration of the working map. This information is encoded by a 2-, 12-, or 108-input representation of the state. Figure 4.2 provides the interpretation of each signal to the network based on the number of inputs to the state.

The 12-input is not pictured here, as it mirrors that of the 108-input, except it represents 1 square of the 9-square map represented by the 108-input representation. Note that since these are binary inputs, each `1` represents that particular input as true, while a `0` reflects that an input is false.

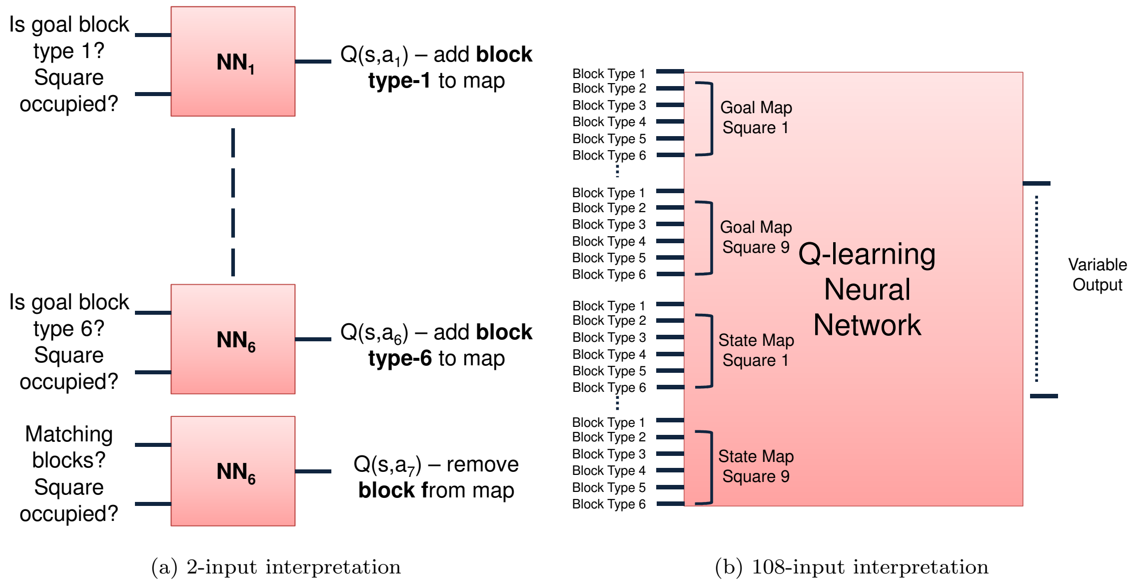
Inputs	**Input Interpretation**	**Multi-NN or Single-NN**	**Map Partition**	**Activation Function**
2	•Block Type Same or Match? •Square Occupied?	Multi-NN; 7 Actions	Single Square	Linear Logistic
12	•(6) Goal State Block Type •(6) Working Map Block Type	Multi-NN; 7 Actions	Single Square	Linear Logistic
12	•(6) Goal State Block Type •(6) Working Map Block Type	Single-NN	Single Square	Linear Logistic
108	•(54) Goal State Block Type for 9 Map Squares •(54) Working Map Block Type	Multi-NN; 63 Actions	9-Square Full Map	Linear
108	•(54) Goal State Block Type for 9 Map Squares •(54) Working Map Block Type	Single-NN	9-Square Full Map	Linear

Table 4.2: Neural Network Topologies Investigated

4.2.2 Experiments

Several experiments were performed as a comparison between the performance of the neural network topologies shown in Table 4.2 to determine their capacity to converge to an optimal solution, and the limitations or necessity for network structure and parameters in solving this problem. In addition, the number of patterns in the randomly generated training set was varied, and indicates the portion of the state space that must be seen for convergence. The error measure applied during backpropagation was varied in the networks to determine if a supremum rather than a Euclidean error measure results in better performance. This is because the supremum error measure performs local changes in the network, and should therefore lead to a higher likelihood of convergence. Lastly, a linear and logistic activation function were investigated, as the ability to converge is known to be dependent on this component of the neural network.

After training the networks by the algorithm described at the beginning of this chapter, the networks were assessed based on the time required for convergence to a policy, and the optimality of this policy. Policy optimality was assesed by testing network performance on 10 test sets. Parameters used for the learning rate, epsilon, and the decay rate were varied in order to allow convergence,



(a) 2-input interpretation (b) 108-input interpretation

Figure 4.2: Interpretation of Inputs to the Neural Networks

and were determined experimentally for each network. The reward function for the single-square representation is equivalent to that of the Soar implementation without rotation and is as follows:

$$r(s) = \begin{cases} -0.2 & \text{if block added to square matched} \\ -0.65 & \text{if a non-matching block was removed} \\ -0.75 & \text{if a matching block was removed or a non-matching block was added} \\ +2 & \text{if the puzzle was completed correctly} \\ -2 & \text{if the puzzle was completed incorrectly} \end{cases}$$

For the full-map representations, the reward function is as follows:

$$r(s) = \begin{cases} 0.5 & \text{if the correct block is added} \\ -0.2 & \text{for removing an incorrect block} \\ -0.45 & \text{for adding an incorrect block or removing a correct block} \end{cases}$$

No explicit terminal rewards are given for the full-map implementations; instead, the reward garnered for an action in the final step is equivalent to the reward administered for the action at any other time step.

Many of the network parameters were determined empirically based on the values allowing

network convergence. The training set was kept constant among the single-square implementations, and a separate training set was used and kept constant among the full-map representations. The determined learning rate, discount factor, and ϵ for random operator selection for the single-square implementations were 0.2, 0.9, and 0.25, respectively, and ϵ and the discount factor for the full-map representations were $0.05 \times 0.9975^{trial}$, and 0.83, respectively. The decaying exploration rate and decreased discount factor were employed in order to reach convergence. The learning rate for the 108x1x1 multi-NN was 0.05, but the learning rate for the 108x63x63 single-NN varied based on the number of patterns in the training set and required a decay rate. This will be discussed further in Chapter 5.

Chapter 5

Results and Discussion

5.1 Soar

5.1.1 Add- and Remove-Block Standard Functionality

Figure 5.1 shows the number of steps taken in each trial as Soar trains using its four add- and remove-block operators. The agent initially takes almost 300 steps to solve 11 10-step puzzles. Although the total number of steps decreases towards the minimum number of steps as the agent continues training, it still oscillates as training progresses, reflecting significant learning and unlearning of $Q(s, o)$ values. This oscillation becomes more pronounced rather than more constant as training continues, unlike the ideal training graph shown in Figure 2.6. However, the agent is able to complete the training set in approximately the minimum number of steps.

Table 5.1a describes the average value of the preferences for certain operator types after training, and Table 5.1b shows Soar's performance on the 5 test sets. The `add-block-to-map-matching` operator has the highest average value, whereas the other operator preferences lag by a factor of $\sim$6. Thus, Soar prefers to add a block to the map that matches the goal state more than any other operator. Furthermore, in each test case, Soar is able to solve the puzzle in the minimum number of actions in at least one of the test runs. An average was taken over several test runs as Soar will always execute a non-optimal operator with some small probability, regardless of whether learning is turned off or the parameters controlling exploration are set to zero. This is reflected in the variance of the average steps required to reach a terminal state s_T and the accuracy of the puzzle having

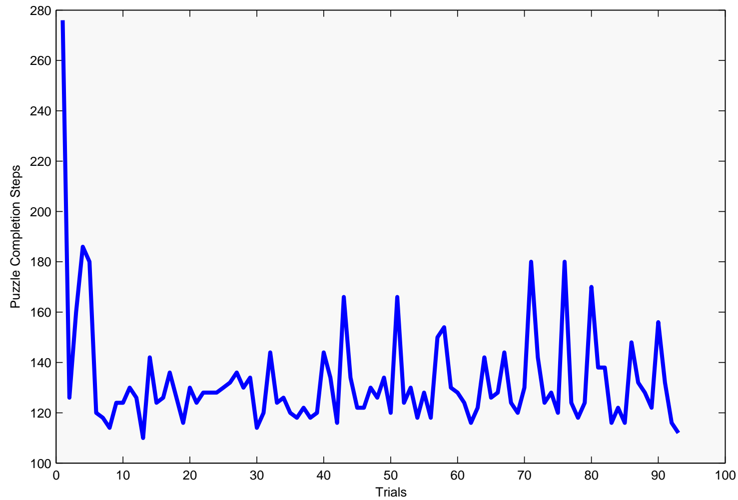


Figure 5.1: Soar Agent Training with Standard Functionality

reached that state, despite the preference values clearly favoring `add-block-to-map-matching`. Table 5.1b also indicates that the information learned from training on 3x3 puzzles with 9 or more blocks can be extended to 4x4 puzzles and 4x4 puzzles with an excess number of blocks.

Operator Type	**Average Preference**
add-block-to-map-matching	-0.25
add-block-to-map-not-matching	-1.47
remove-block-from-map-matching	-1.48
remove-block-from-map-not-matching	-1.42

(a) Numeric Preferences

Test Set	**Minimum Steps**	**Minimum Steps Achieved?**	**Average Steps**	**Average Accuracy**
1	10	√	16	100%
2	10	√	11.2	95.4%
3	10	√	10	88.4%
4	17	√	18.2	97.5%
5	17	√	17.4	91.25%

(b) Test Performance

Table 5.1: Numeric Preferences and Testing of Soar with Standard Functionality

5.1.2 Standard Functionality Augmented by Rotate Block Operators

The rotate-block operators augment Soar's capabilities for solving puzzles so that puzzle situations that require a block to be rotated in order to reach a goal state can be achieved since the exact type of the block is not present. This of course excludes the solid blocks; the total number of solid blocks required to solve the puzzle must be provided. Soar's learning process for this task is illustrated in Figure 5.2. This figure also demonstrates some amount of learning because the agent reduces the total number of steps from ∼700 to an average value of ∼140. Again, Soar continues to oscillate in the number of steps to solve the puzzle until it achieves approximately the minimum number of steps over the entire training set.

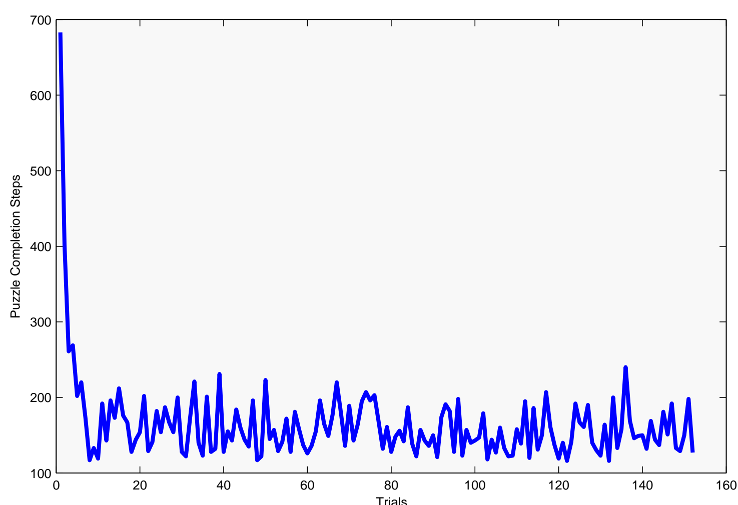


Figure 5.2: Soar Training with Augmented Functionality

The average operator preferences in Table 5.2a and the top five preferred operators, shown in Table 5.2b explain the agent's poor performance, shown in Table 5.2c. According to Table 5.2a, the agent should prefer to add a matching block to the map all the time. In the test sets that require rotation to solve, the agent will eventually reach a state in which the `add-block-to-map-matching` operator can no longer be applied, which ideally signals that a block must be rotated. This should theoretically correspond to the secondmost preferred operator, a block rotation, being selected. However, in reality, the preferences for adding a non-matching block and rotating one of the split-type blocks are very close and have overlapping ranges. Thus, the action chosen depends on the

actual values of each of the operators that can be applied, which could lead to the addition of incorrect blocks. This is confirmed by Table 5.2b. As shown here, after adding all of the possible matching blocks, the agent's next preferred operator is to add an unmatching block to the map. Its third choice for an operator is correct; rotating this block type transforms it into another block type that can be then used for completing the map. The last two operators are both poor choices.

This poor policy is seen by the agent's performance on the test sets. The agent performs the first test, the standard block matching test for a 3x3 puzzle solvable by the original configurations of 9 blocks at 100% accuracy. However, as the agent enters to the next two tests, a 3x3 with 9 blocks and a 3x3 with 36 blocks, both of which require sequential block rotation and addition, the agent falters. The performance is the worst for the third test set that requires 9 rotations to achieve the correct goal configuration, and additionally has an excess number of blocks. In this test, the agent finishes in on average 11 steps, which means that it consistently adds incorrect blocks rather than rotating them to reach the correct orientation before adding them. However, the agent is able to solve the standard 4x4 in the minimum number of steps at 100%, although when given an excess number of blocks, it can no longer solve them with 100% accuracy in the minimum number of steps.

As shown here, the agent generally finishes the puzzle in near the optimal number of steps it should take; however, the accuracy of the puzzle submitted is extremely poor, especially in the case of the third training set. It is possible that the agent has not yet converged to an optimal policy, although the agent is able to complete the training set within ± 4 steps of the minimum number of steps. However, since 100% accuracy is not required to move to the next pattern in training (such an implementation did not converge in a reasonable amount of time), this could explain the undertraining. In addition, undertraining is suggested because the preferred operator is the `add-block-to-map-matching` operator and a correct rotate-block operator is nearly the second choice. A longer training time may have allowed the agent to find the correct relative preference for this second operator.

5.1.3 An Assessment of Soar

The previous two sections have demonstrated that Soar is able to determine an optimal policy π^* that enables it to execute an optimal action in every state and reach the final, correct goal state in the minimum number of steps at 100% accuracy. However, due to the execution of non-optimal actions even during greedy action selection while training, it is difficult to assess when

Operator	Average Preference
add-block-to-map-matching	-1.953±0.26
add-block-to-map-not-matching	-2.701±0.08
remove-block-from-map-matching	-2.843±0.03
remove-block-from-map-not-matching	-2.849±0.04
rotate-block-90-cw-urll	-2.708±0.056
rotate-block-90-cw-ullr	-2.738±0.061
rotate-block-90-cw-solid	-3.147±0.126

(a) Numeric Preferences

Operator	Preference
add-block-to-map-matching	-1.953±0.26
add-block-to-map-not-matching •Goal: solid white block •Block: urll white/red	-2.29
rotate-block-90-cw-urll •Goal: ullr white\red •Block: urll red/white	-2.45
rotate-block-90-cw-ullr •Goal: solid white •Block: ullr red\white	-2.46
add-block-to-map-not-matching •Goal: urll red/white •Block: ullr red\white	-2.5

(b) Top Five Operators

Test Set	Minimum Steps	Minimum Steps Achieved?	Average Steps	Average Accuracy
1	10	$\surd$	11.2	100%
2	19	$\times$	54.4	63.8%
3	24	$\times$	11.4	28.6%
4	17	$\surd$	18.8	86.25%
5	17	$\times$	18.8	78.75%

(c) Test Performance

Table 5.2: Numeric Preferences and Testing of Soar with Augmented Functionality

the agent has to converged to the optimal value to stop training. On the other hand, though the number of steps oscillates and the magnitude of these oscillations rise as training proceeds, convergence to an optimal policy can be assessed through testing, and has been shown to converge in the no-rotation case. This is not true for the agent with the rotate operators, and this added functionality corresponds to a loss in learning and generalization capabilities.

Lastly, it is interesting to note that Soar's implementation of $Q(s, o)$ values is midway between a lookup table and a function approximator. In general, without having executed a certain operator during training, the preference for that operator is zero during testing. Thus, Soar is susceptible to poor performance when the training set is not representative of problem space or

it is not trained long enough, although this is true for many learning agents. However, if that operator has been executed in any situation, although it may not have ever been executed in a particular situation, the preference for the operator can still be calculated due to Soar's $Q(s, o)$ linear combination update strategy.

5.2 Neural Networks

5.2.1 2x5x1 Multi-NN

Figure 5.3 shows the correspondence between the evolution of the steps required to complete the training set and the error in function approximation for the network. As shown, the two graphs are similar in shape, although their magnitudes differ. This indicates that the number of steps required for correctly solving the puzzle decreases as the error decreases and vice versa. In addition, this allows for regarding one quantity as indicative of the other. Thus, the number of steps required for training set completion is presented in the following results as these numbers are more intuitive for understanding.

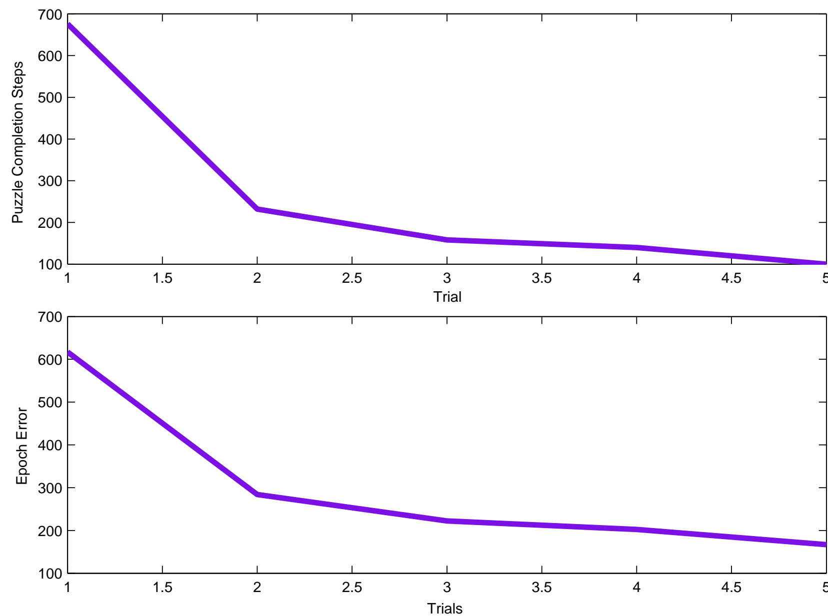
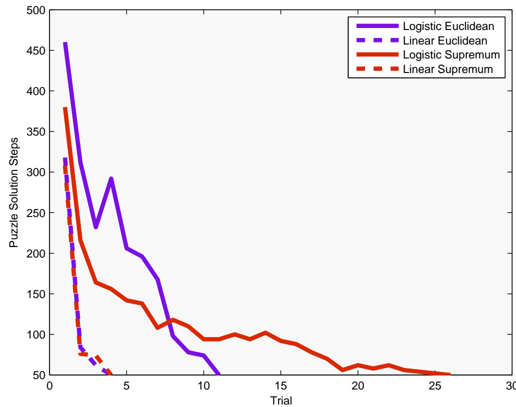


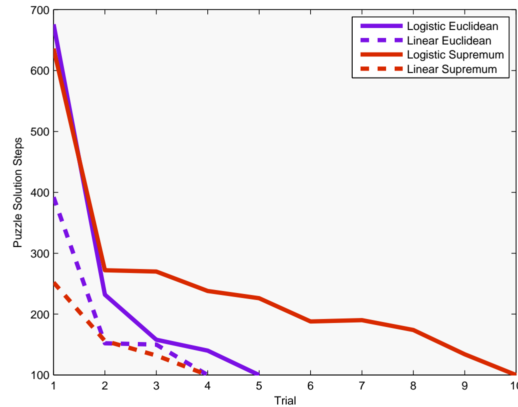
Figure 5.3: Comparison of Puzzle Solution Steps and Function Approximation Error

Figure 5.4 compares the training and performance of a 2x5x1 multi-NN with different error

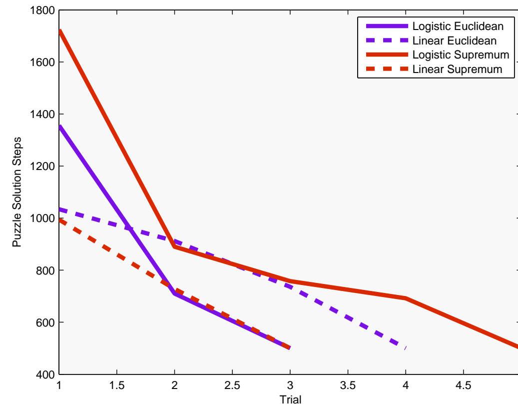
measures and activation functions. As shown, regardless of the size of the training set, all implementations were able to converge, although the 50-pattern set converged fastest based on the total number of training examples seen. The main difference between training set sizes is their impact on the smoothness of the error function E. The 50-pattern training set in Figure 5.4a has the most oscillative error function which is most pronounced in the logistic Euclidean and logistic supremum graphs. On the other hand, Figures 5.4b and 5.4c of the 100- and 500-training pattern sets have increasingly smoother graphs of E. Nonetheless, the oscillation or smoothness of the decreasing E does not impact the convergence of the algorithm for this network.



(a) 50 training patterns



(b) 100 training patterns



(c) 500 training patterns

Figure 5.4: 2x5x1 Multi-NN Training

As shown by these figures, this network trains best with linear activation functions and

Euclidean error measures. Employing the logistic activation function results in the worst convergence time for most training set sizes and error measures with no increase in generalization. The 500-pattern training set differs slightly in that the logistic/Euclidean architecture performs better than the linear/Euclidean architecture. Since the number of trials required for convergence for each type are so similar, it is possible that the difference could be due to the random action selections from ϵ-greedy formulation. Lastly, because of the composition of the training sets, if the algorithm is able to converge, this means that the exact mapping for each state-action pair is known, and no explicit test for network generalization is necessary. The same is true for all single-square implementations.

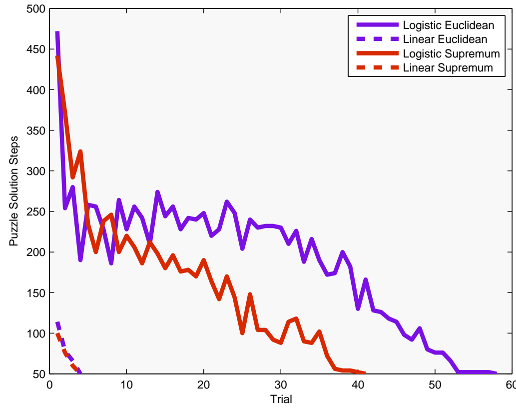
5.2.2 12x13x1 Multi-NN

Figure 5.5 illustrates the training process for the 12x13x1 multi-NN. The effect of the difference in training patterns on E is shown, and again, the 500-pattern training set does the least learning and unlearning. It also requires the fewest training epochs to converge, although the 50-pattern training set requires the fewest total trials for convergence. In addition, the linear activation functions perform better than that of the logistic activation function, and for the linear activation function, the Euclidean error measure converges at a rate equal or faster than the supremum.

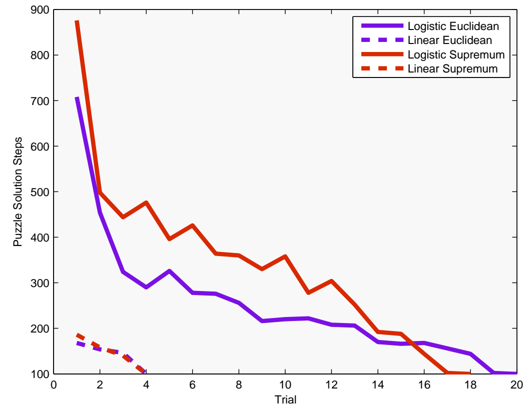
However, this multi-NN then differs from the 2x5x1 multi-NN in the previous section in the logistic activation function. Here, the supremum outperforms the Euclidean error measure. Since the training set and parameters are the same as the previous implementation, the architecture accounts for this difference in convergence times for architectures trained with the same number of patterns in the training set. It is again possible that the random actions selected during training may have had an impact on these convergence times, although the disparity in the number of trials for convergence and the consistency of ϵ throughout the test sets makes this unlikely. Lastly, because this network and 2x5x1 multi-NN were tested on the same training patterns, it can be concluded that the 2x5x1 multi-NN outperforms the 12x13x1 multi-NN for the logistic activation functions; however the performance of the 2x5x1 is relatively equivalent or slightly better for the linear implementations.

5.2.3 12x13x7 Single-NN

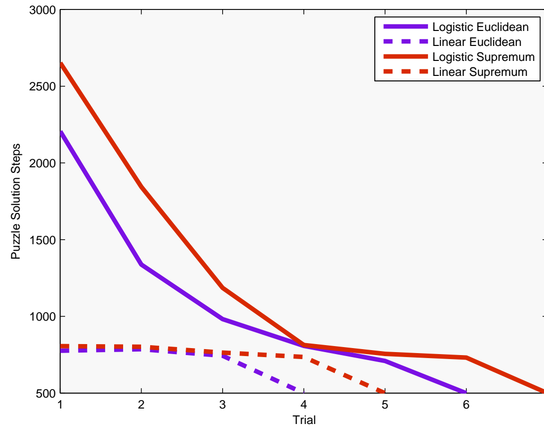
The 12x13x7 single-NN represents the first architecture presented that is not able to converge for some combinations of error measure and activation function. This is shown in Figure 5.6, which



(a) 50 training patterns



(b) 100 training patterns



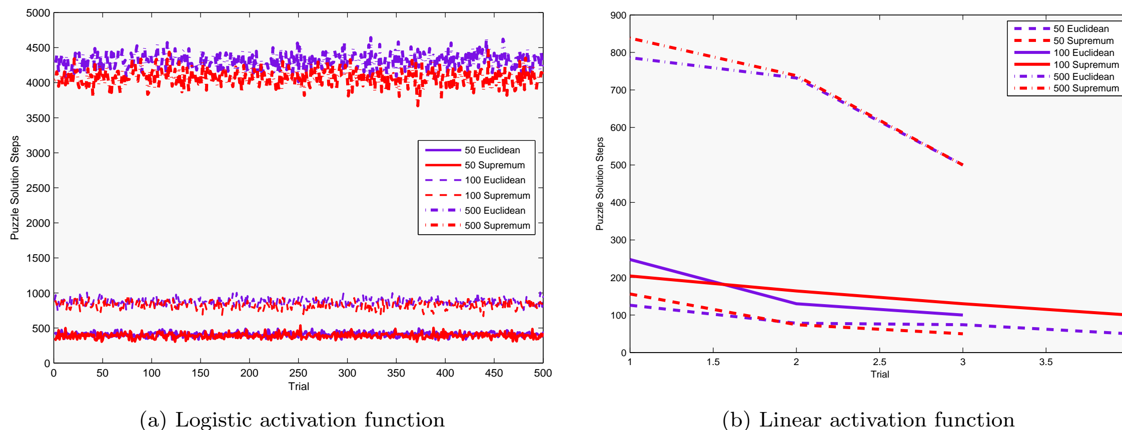
(c) 500 training patterns

Figure 5.5: 12x13x1 Multi-NN Training

separates the implementations by activation function. Figure 5.6a shows the error during training for an architecture with units having logistic activation functions. It is readily noticed that the error does not have an overall downward trend for any of these training sets. Instead, the error merely oscillates around a central value, which suggests that the network is not learning and is not approaching convergence. For all of these non-converging tests, it can be noticed that the supremum error measure consistently performs slightly better than its Euclidean counterpart for each number of training patterns, as it requires slightly fewer iterations to converge for each puzzle.

However, this inability to converge for the MLFF-ANN architecture with non-linearities is somewhat unexpected; since neural networks with sigmoidal activation functions are universal ap-

proximators, it is counterintuitive that the network would be unable to learn the $Q^*(s, a)$ represented by the network. It is especially counterintuitive because Backgammon[19], Othello[21], and mobile robot navigation[5], [6] all use logistic, sigmoidal, or hyperbolic tangent functions in MLFF-ANNs to approximate the value function for the reinforcement learning algorithms. However, no convergence proof exists for using non-linear function approximators with Q-learning, and therefore it not guaranteed to converge.



(a) Logistic activation function

(b) Linear activation function

Figure 5.6: 12x13x7 Single-NN Training

On the contrary, Figure 5.6b demonstrates that both error measures for the linear activation functions were able to converge in 4 or less iterations. However, there is no conclusion that can be made about the supremum versus the Euclidean error measure from these results; both the Euclidean and supremum error measures for the 500 pattern training set led to the same number of trials for convergence, whereas the 50- and 100- pattern training sets had better performance with the supremum and Euclidean error measures, respectively. The reason for this discrepancy is most likely due to the exploratory choices from the ϵ-greedy action selection method.

This is shown in Figure 5.7, where the solid line represents the supremum and the dashed line represents the Euclidean error measure. This figure shows several trials for the 100 pattern training set with constant starting weights and training sets. Despite these consistencies, neither the supremum nor the Euclidean error measures consistently outperformed the other in terms of iterations required for convergence. However, the Euclidean error measure generally took a few less steps in some puzzles in the training set, and based on these trials, took on average fewer trials to converge. Then, this difference in performance may be due to the probabalistic choice of operators

due to ϵ-greedy combined with the small number of cycles required for convergence. Practically, the difference between the trials required for convergence with these two error measures is trivial.

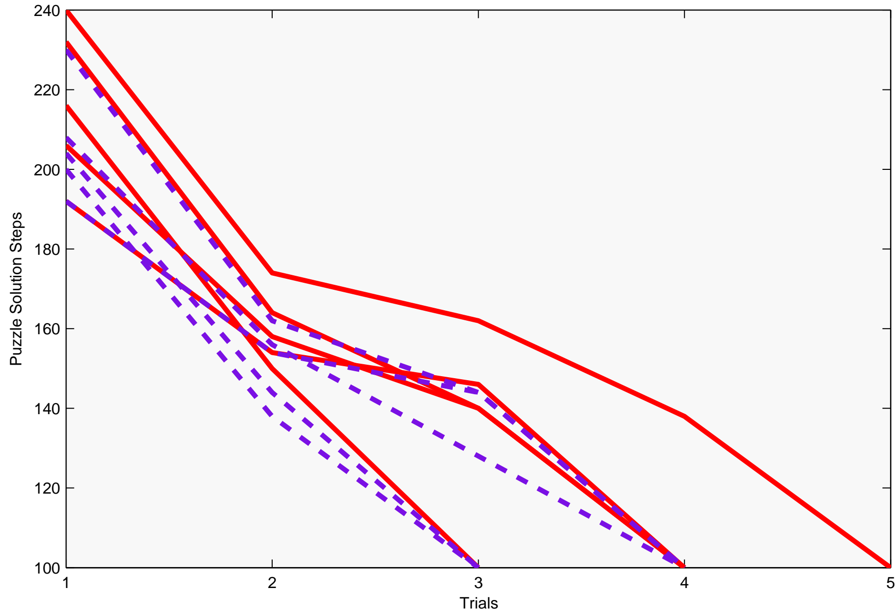


Figure 5.7: Assessment of Error Measure Consistency for 12x13x7 Single-NN

5.2.4 108x1x1 Multi-NN

Like the 12x13x7 single-NN in the previous section, the 108x1x1 multi-NN was also unable to converge for some combinations of error functions and activation functions. The algorithm could not converge in less than 5,000 iterations for any logistic activation function. Furthermore, the supremum error measure for the 500-pattern dataset was unable to converge in less than 5,000 trials. The results for the linear activation function are shown in Figure 5.8. Here, the 500-pattern training set does not reflect the smoothest E; in fact, it increases for 5 iterations before approaching convergence. The 50- and 100- pattern training sets are much smoother and show the expected learning curve. In addition, the supremum error measures for the 50- and 100-pattern data sets take *much* longer for convergence than that of the Euclidean; for the 50-pattern training set, supremum convergence requires twice as many trials for convergence, and the 100-pattern training set requires almost 200 times more trials than its Euclidean counterpart.

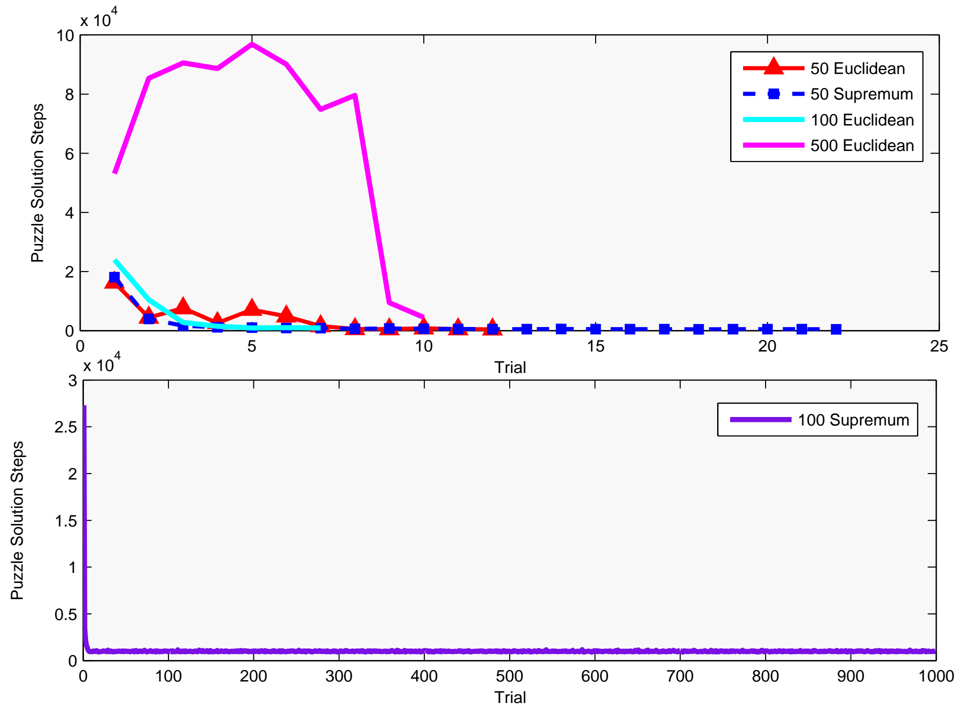


Figure 5.8: 108x1x1 Multi-NN Training

These longer convergence times do not, however, correspond to better generalization by the network. As shown in Table 5.3, the architecture with the best performance is the 500-pattern linear multi-NN as it solves all puzzles in the test set with 100% accuracy. Then, this architecture is able to closely approximate Q^* and has converged to π^*. Although the supremum error measure could not be performed for the 500-pattern data set, the 50- and 100- pattern datasets demonstrate that the supremum error measure consistently performs at a lower accuracy on test sets that it has not seen. This suggests that a supremum error measure would not have increased the performance of the 500-pattern dataset. Lastly, this experiment suggests that for a multi-NN architecture for problems of this type, more training data increases the performance of the agent and increases the likelihood of the agent to converge to an optimal policy rather than an incorrect policy. In the cases with the smaller training sets, the neural networks were able to find the optimal Q-functions for the function represented by the patterns in the dataset. However, this did not represent the global Q^* to be approximated over the entire problem space.

An interesting question to address for this architecture is if starting each of the networks with the same weights would be effective in decreasing the training time and increasing the perfor-

Table 5.3: 108x1x1 Multi-NN Test Performance

Architecture	Training Epochs	Average Correct Squares	Average Accuracy
50 Euclidean	12	6.7	74.4%
50 Supremum	22	6.6	73.3%
100 Euclidean	7	8.8	97.8%
100 Supremum	999	7.5	83.3%
500 Euclidean	10	9	100.0%

mance of the supremum error measure. Conceptually, this implementation suggests that the winning network's weights are adjusted slightly above or slightly below the outputs of another network based on the error feedback. This experiment was conducted on the 100-pattern architecture to maximize performance but also preserve the ability of the supremum error measure to converge. Figure 5.9 and Table 5.4 discredit this approach, as the supremum error measure still takes 10 times longer to converge to a policy. Furthermore, the Euclidean network still outperforms the supremum network. However, no conclusions can be made with respect to the previous experiment because the difference in network weights could have caused this shorter convergence time.

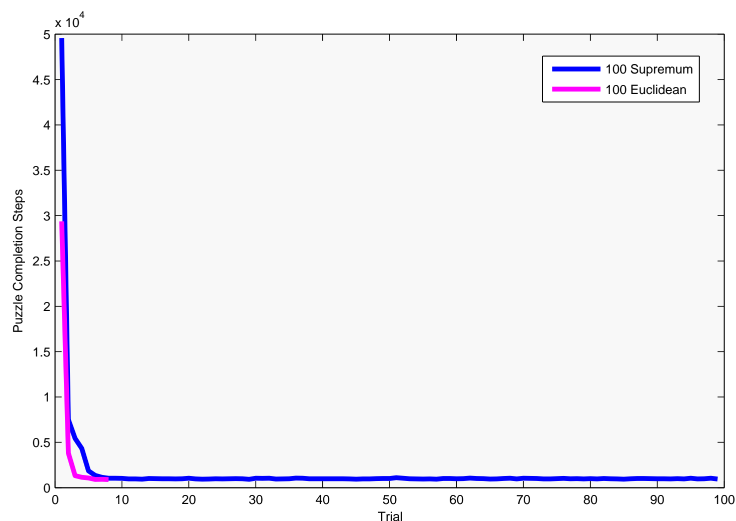


Figure 5.9: 108x1x1 Multi-NN with Identical Initial Weights Training

The experiments in this section therefore suggest that the networks that lead to the best generalization are those that can be trained using larger training sets and implement linear activation functions with Euclidean error measures. Lastly, no significant advantage is gained by initializing

Table 5.4: 108x1x1 Multi-NN with Identical Initial Weights Test Performance

Architecture	Training Epochs	Average Correct Squares	Average Accuracy
Supremum	99	7.3	81.1%
Euclidean	8	8.6	95.6%

the weights of all the networks to the same values.

5.2.5 108x63x63 Single-NN

This neural network representing $Q(s, a)$ in one network requires an additional learning rate decay parameter to converge. These decay rates were based on the learning rates used for the 108x1x1 multi-NN, though the actual decay parameter was determined experimentally. The decay formula was of the form $\eta_n = \eta_0 \lambda^n$. η_0, the initial learning rate was experimentally determined and set to 0.05. The decay parameter λ was also experimentally determined to be 0.9975 for the 50- and 100-pattern training sets, and 0.99985 for the 500-pattern training set. For the 100- and 500-pattern training sets, deviating very far from these values resulted the training error increasing steeply after several training epochs. As in the above section, the network was unable to converge for logistic activation functions, and additionally, none of the training sets were able to converge for the supremum error measure.

The results of training and testing of the 108x63x63 neural network is shown in Figure 5.10 and Table 5.5. As shown, there is not a significant difference in the training times of the 50-, 100-, and 500-pattern training sets, although the 500-pattern training set does converge one epoch before the other two. However, although the agent is able to converge in a similar number of epochs, the generalization capabilities vary greatly from the 50- to the 500- pattern training set. Again, the 500-pattern training set is the best performer with 100% accuracy on each of the tests. This demonstrates that this network is able to closely approximate $Q^*(s, a)$, and has determined the optimal policy π^* for reaching the goal state starting from an initially empty map.

Table 5.5: 108x63x63 Single-NN Test Performance

Patterns In Training Set	Training Epochs	Average Correct Squares	Average Accuracy
50	9	6.2	68.9%
100	9	8.3	92.2%
500	8	9	100.0%

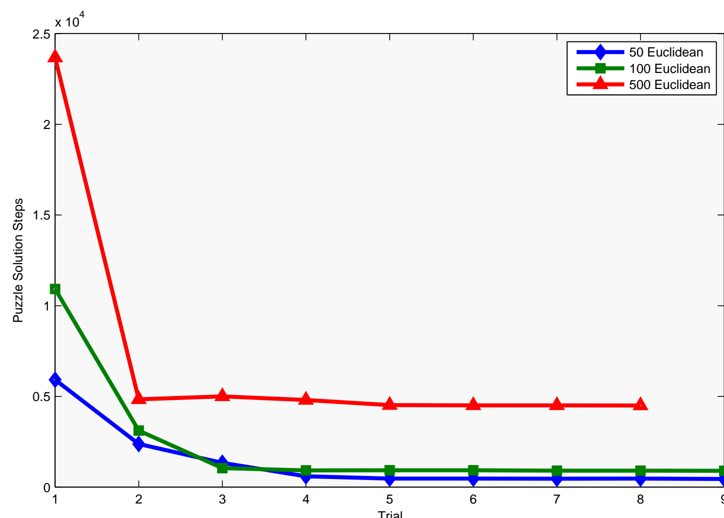


Figure 5.10: 108x63x63 Single-NN Training

This experiment demonstrates that with the addition of a learning decay parameter, the network is indeed able to converge to an optimal policy. Again, the training set should be large so the agent has a good representation of the problem space to accurately estimate $Q^*(s, a)$.

5.2.6 Comparing Network Architecture Components

5.2.6.1 Comparison of Training Set Sizes

The number of patterns in each training set affected both the training and performance of the networks. The single-square representations all experienced smoother E functions in the 500-pattern training set, reflecting that the agent learns without unlearning from epoch to epoch. The impact of the training set size on generalization capabilities of the network was explicitly seen in the full-map representations; more patterns in the training set correspond to better generalization. This is intuitive; in training, the agent is able to see more of the problem space and experience rewards from the environment on this augmented space. Thus, the agent has a training set that is most likely more representative of the problem space. Then, the solution to this problem suggests that a larger training set corresponds to a better policy for generalization.

5.2.6.2 Comparison of Input Representations

As postulated, the input representation affects the convergence of the network. This is most clearly seen in the 2x5x1 and 12x13x1 multi-NNs, which have the same Q-learning NN structure, training set, and algorithm parameters. However, for the logistic activation function, the Euclidean error measure resulted in shorter convergence times on the 2x5x1, whereas the 12x13x1 had better convergence with the supremum error measure, except in the 500-pattern training set case. The sole difference (neglecting the ϵ-greedy selection mechanism as described previously) is the input representation since both have $2n+1$ hidden layer units for n units in the input layer. The 108x1x1 multi-NN cannot be compared because it could not converge with a logistic activation function.

All input representations are able to converge with the linear activation function, and the Euclidean error measure overall results in faster convergence times. However, this is not true for the 2x5x1, which in the 500-pattern training set case sees faster convergence times for the supremum error measure. Since the 12x13x1 multi-NN differs only in input representation, again, this must be the source of the difference. Since the relationship between input representation and convergence is still unclear, no general conclusions can be drawn from these results.

5.2.6.3 Comparison of Error Measures

The supremum and Euclidean error measures also significantly affected the convergence of the networks. Although the single-square representations were able to converge with a supremum error measure, the full-map representations either took comparatively more trials to converge, or was unable to converge at all. In general, the Euclidean error measure was the most reliable, as it was always able to converge if the activation function was able to converge, and resulted in fewer iterations required for convergence in all except the 2x5x1 case.

5.2.6.4 Comparison of Activation Functions

The logistic and linear activation functions also had an impact on convergence for all the networks. Except for the 500-training pattern case of the 2x5x1 multi-NN, the linear activation function resulted in much shorter convergence times and was always able to converge. The same cannot be said for the logistic activation function, which could not converge for 3 out of the 5 networks tested here. In order to set parameters allowing convergence for the full-map implementation,

the learning rates were decreased, and decaying exploration rates and learning rates were tested, although convergence was never reached. Since there is no convergence proof or constraints outlined for Q-learning with non-linear function approximators, additional solutions could not be employed. Nonetheless, this problem suggests that the best or most reliable mapping function for solving this problem is a linear activation function.

5.2.7 Qualitative Comparison of Network Architectures

Table 5.6 provides a summary of the results of implementing the BD-DE intelligence problem with different neural network architectures. Each architecture has a number of qualitative considerations that must be weighed before assessing the best architecture for solving the problem.

The single-square implementations have comparatively fewer weights than the full-map representations, and are not restricted by the size of the map. These implementations assume independence of squares on the map, and therefore can correctly solve any arbitrary BD-DE puzzle of size n that begins with an empty map as an initial state. In addition, as long as the network converges, 100% accuracy on test sets are ensured. This is because the agent has seen every possible situation that it will encounter in testing through the training set.

Thus, these architectures are exceptional if a problem can be decomposed into a set of subproblems that can be solved as the continual reapplication of one function. The implementation of this BD-DE intelligence test with intermdiate rewards is an example of such a situation, and thus, these architectures successfully solve the problem.

Full-map representations have the advantage in that if there is some dependence or extra information provided by knowledge of the surrounding positions on the map, these representations can use this information to their advantage. This can be seen in the NNs designed for playing Othello[21] and Backgammon[19], since knowledge of the state of only one of the positions on the board is nearly useless. However, these implementations suffer from architectural constraints because they are not as free to employ any error measure, learning rate, decay rate, activation function, or error backpropagation measure if they are expected to converge in a reasonable amount of time. However, having found appropriate parameter values and functions, with the appropriate-sized training set, these architectures were able to learn the optimal policy for solving the BD-DE problem-solving task for a goal map with 9 positions and displayed exceptional generalization capabilities. Lastly, unlike the square map representations, there is not a straightforward way that

Table 5.6: Qualitative Comparison of Neural Network Architectures

Architecture	Type	State Represented	Advantages	Disadvantages
2x5x1	Multi-NN	Single Square	•Few network weights •Optimal policy inherent •Any map size solvable	•No relationship among squares on the map •No generalization
12x13x1	Multi-NN	Single Square	•Direct representation of one square •Few network weights •Fast training •Optimal policy inherent •Any map size solvable	•No relationship with other squares on map •No generalization •More weights than Single-NN counterpart
12x13x7	Single-NN	Single Square	•Direct representation of one square •Fewer network weights than Multi-NN •Fast training •Optimal policy inherent •Any map size solvable	•No relationship with other squares on map •No generalization •More limited architecture
108x1x1	Multi-NN	Full Map	•Representation of entire state	•Solves limited puzzle sizes •Many weights •More limited architecture •Slower training than Single-Square
108x63x63	Single-NN	Full Map	•Representation of entire state •Fewer weights than Multi-NN counterpart	•Solves limited puzzle sizes •More limited architecture •Additional constraints •Slower training than Single-Square

these architectures can be used to solve puzzles of an arbitrary size.

With these considerations, for sequential problems that have sparse rewards and cannot be decomposed into one reapplicable function, the single-NN full-map representation is the most useful. It is able to model the entire state, take into account the entire map and tailor its behavior accordingly, and has the same performance as the multi-NN but has fewer weights. If the problem can be decomposed, the single-NN single-square representations are best, as the networks are smaller, converge faster, and have a smaller set of architectural constraints.

5.3 Quantitative Comparison of Soar and Neural Networks

The above results have shown that all the architectures tested - Soar and the 5 neural network topologies are able to converge to an optimal policy for the BD-DE puzzle. Which one is the best for solving the problem in terms of performance? Table 5.7 shows the performance of Soar and the neural networks. As shown, each of the architectures is able to achieve 100% accuracy on the test set, and therefore the evaluative criteria is the total training iterations required for the agents to converge to the optimal policy. As shown, the multi-NNs in general require more iterations to converge than their single-NN counterparts, although the 12x13x1 multi-NN and the 12x13x7 single-NN are both able to converge in 150 trials. In addition, these tables again show that the single-square representations converge much faster than the full-map representations.

The best neural networks, however, are able to converge in fewer iterations than Soar. However, Soar converges faster than either of the full-map representations of the neural networks. Thus, in the case that a problem can be subdivided into reapplicable functions, these results suggest that a neural network approximating that function should be chosen over a full-state representation or a production system. However, in the case that this is impossible, Soar should instead be used rather than a full-map neural network as long as the solution space is sufficiently small. Otherwise, a full-state neural network should be used.

Multi-NN	**Map Representation**	**Performance**	**Training Patterns**	**Trials Required**	**Total Training Time**
2x5x1	Single Square	100%	50	4	200
12x13x1	Single Square	100%	50	3	150
108x1x1	Full Map	100%	500	10	5000

(a) Multi-NN Performance

Single-NN	**Map Represented**	**Performance**	**Training Patterns**	**Trials Required**	**Total Training Time**
12x13x7	Single Square	100%	50	3	150
108x63x63	Full Map	100%	500	9	4500

(b) Single-NN Performance

Soar	**Map Represented**	**Performance**	**Training Patterns**	**Trials Required**	**Total Training**
Standard	Full Map	100%	11	110	1210
Rotation	Full Map	100%	11	152	1672

(c) Soar Performance

Table 5.7: Quantitative Comparison of Neural Network and Soar Performance

5.4 Qualitative Comparison of Soar and Neural Networks

In addition to the performance of the architectures, the implementation and usage of both architectures must be assessed to determine the best solution for the problem at hand. As a production system, Soar caters to the logical thought processes of human beings; the productions are straightforward to formulate, and the results in terms of numeric preferences are intuitive to interpret. Better yet, if Soar is able to converge to the optimal policy, Soar is able to solve a puzzle of any arbitrary size n, because the preference values for the operators are not dependent on the size of the network. However, as previously mentioned, Soar implements lookup table-like functionality for its operators; thus, if an operator is not used during training, the preference for that operator becomes zero during testing, which may not correspond to an optimal policy. Even further, these lookup tables can become large depending on the actions available, and can begin to defeat the purpose of why this architecture was initially employed in terms of function representation and speed.

Table 5.8: Comparison of Soar and Neural Network Design

	Advantages	**Disadvantages**
Soar	•Logical implementation •Easier interpretation of preferences •Solves puzzle of arbitrary sizes •Similar to lookup table	•Unable to control exploration in testing •Error graphs not helpful •C++ Interface •Software manual
Neural Network	•Easily interpretable error graphs •Converges to optimal policy quickly •Best performance	•No useful guidance on parameter or architecture determination •Not guaranteed to converge for all architectures •Training not easily interpretable •Difficult in implementation without a toolbox

Soar also exhibits unusual and altogether unwanted behavior in terms of explorative policies. As shown by the results in the preceeding sections, there are unsolvable issues with Soar in that even during testing when learning has been disabled and the probability for executing random actions has been set to zero, Soar still executes suboptimal operators. This activity is highly undesirable, especially in the case of high-risk technology whose faulty operation could end up in user fatalities. In addition, the error graphs are somewhat unreliable in determining when the algorithm has converged; therefore, it is difficult to identify when to end training. Interfacing with Soar also requires proficiency in C++, which can be challenging based on the experience of the programmer. Lastly, from a practical standpoint, Soar is an open-source, constantly evolving software, and therefore, helpful and informative software documentation can significantly lag begind product development; this can present a struggle in the attempt to implement problem solving with certain learning capabilities in Soar.

On the other hand, the activity of neural networks is difficult to interpret due to their numerical nature, and without the use of a toolbox, implementation can be challenging depending on the level of expertise of the programmer. In addition, the full-map representations cannot solve a puzzle of arbitrary size, although the single-square representations can perform this function. It has also been shown that the network performance is based on the number of patterns in the training set; without enough patterns, the network is unable to converge to the optimal policy. Furthermore, as

demonstrated by the latter implementations of the neural network, they frequently must be finessed into convergence with the careful choice of learning and weight decay rates, and architecture in terms of input representation, error measure, and activation functions.

Even further, they produce unexpected and sometimes conflicting results; the research conducted by Eck [21] finds that a single-NN architecture performs the best, [9] suggests that the multi-NN has the best performance because it is not hindered by conflicting weight updates, whereas this research would conclude both architectures perform with no appreciable difference. Furthermore, the mobile robot obstacle avoidance research in [5] and [6] utilize intermediate and final rewards, whereas none of the full-map architectures in this thesis are able to converge with both types of rewards. For the 108x1x1 multi-NN and the 108x63x63 single-NN, the final reward must be equivalent the reward for the last action taken.

Despite these drawbacks, the neural networks display intuitive error curves that allow the user to easily determine if the agent is learning, which Soar fails to produce. In addition, as shown by this thesis, the full-map representation converges very quickly in 9 iterations, which takes less physical time than that required by Soar, although Soar implements full-map functionality and is able to converge in fewer trials. Thus, both qualitative and quantitative considerations must be weighed in order to reach a final decision about which architecture to employ to solve a Q-learning reinforcement learning problem.

Chapter 6

Conclusions

This thesis proves that an an automated agent endowed with the ability to learn can find an optimal policy for solving a portion of an intelligence test that measures problem solving and synthesis abilities. In doing this, the agent has demonstrated greater learning ability than some children.

This thesis investigated two architectures for solving this problem - Soar, a production system equipped with a reinforcement learning module, and a multilayer feedforward neural network to solve the Block Design - Disembodied problem, a test derived from the Block Design and Block Design Multiple Choice subtests on the Fourth Edition of the Wechsler Intelligence Scale for Children. This is a non-trivial problem; the expectation of the agent matching a block of a certain type to the goal state square of the same type is not inherent in the agent's programming or design. Thus, the agent must learn through trial-and-error interaction with the environment that this is the correct functionality. Both architectures employed Q-learning, a reinforcement learning algorithm that enables an automated agent to determine the best actions to take in a particular situation based on the way the environment responds to its actions.

Soar was able to converge to the optimal policy for the BD-DE problem, although it could not converge to an optimal policy for the additional desired functionality of block rotation to solve a larger set of problems. Many topologies of the neural network were attempted and varied based on the size of the training set, the input representation, the error measure employed, and the activation function used. This research suggests that larger training sets, a linear activation function, and Euclidean error measure aid in affecting convergence to an optimal policy in the fewest training trials.

Applicability of these results to problems with different characteristics still remains to be determined, and future research should identify qualities of this problem and instances where these results will be applicable. Nonetheless, each of the neural networks was able to converge in a reasonable amount of time with the correctly selected components, although the network performance was dependent on the number of patterns in the training set.

The Soar implementation suffered from undesired execution of suboptimal operators under greedy action selection, and the implementation of the operator preferences can result in an a suboptimal policy if an operator is not applied during training. The neural network required empirical search for parameters and functions that allowed convergence, but when this information was identified, some networks were able to converge in fewer trials than that required by Soar.

In finality, the deciding factor is this: is the application able to sustain suboptimal actions taken during testing or usage? According to this research, if the application cannot take actions that result in suboptimal actions, out of these two architectures, a neural network should be used. However, if the application is able to allow suboptimal actions to be taken, then, the ability to decompose the problem into one reapplicable function should be assessed. If this is possible, a neural network should be employed, as this leads to shorter convergence times and more reliable performance.

However, if the full state is necessary and a training set representative of the problem space is available, Soar should be used, as it converges to an optimal solution faster than full-state neural networks. On the other hand, if the number of actions or problem space is very large, Soar will experience the slow down and limitations of a lookup table, and a neural network may provide a faster implementation in physical time. In the case that the qualitative aspects of Soar do not outweigh the qualitative and quantitative aspects of the neural network, a full-state neural network implementation can therefore also be employed, although it may take longer to converge, and there are no definitive guidelines for the selection of algorithm or architecture parameters.

Bibliography

[1] M.K. Bloch. Hierarchical reinforcement learning in the taxicab domain. Technical Report CCA-TR-2009-02, Center for Cognitive Architecture, University of Michigan, 2009.

[2] G. Cybenko. Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals, and Systems*, 2:303–314, 1989.

[3] Dawn P. Flanagan and Alan S. Kaufman. *Essentials of WISC-IV assessment*. Essentials of psychological assessment series. Wiley, 2009.

[4] G. Frank. *The Wechsler enterprise: an assessment of the development, structure, and use of the Wechsler tests of intelligence*. International series in experimental psychology. Pergamon Press, 1983.

[5] V. Ganapathy, Soh Chin Yun, and W.L.D. Lui. Utilization of Webots and Khepera II as a platform for neural Q-learning controllers. In *Industrial Electronics Applications, 2009. ISIEA 2009. IEEE Symposium on*, volume 2, pages 783–788, Oct 2009.

[6] Bing-qiang Huang, Guang-yi Cao, and Min Guo. Reinforcement learning neural network to the problem of autonomous mobile robot obstacle avoidance. In *Proceedings of the Fourth International Conference on Machine Learning and Cybernetics*, pages 85–89, August 2005.

[7] Anders Krogh and John A. Hertz. A simple weight decay can improve generalization. In *Advances in Neural Information Processing Systems 4*, pages 950–957. San Francisco, CA: Morgan Kaufmann, 1992.

[8] John E. Laird and Clare Bates Congdon. *The Soar User's Manual Version 9.3.1*, June 2011.

[9] Long-ji Lin. Self-improving reactive agents based on reinforcement learning, planning, and teaching. In *Machine Learning*, pages 293–321, 1992.

[10] Francisco S. Melo and M. Isabel Ribeiro. Q-learning with linear function approximation. In *Proceedings of the 20th Annual Conference on Learning Theory*, pages 308–322. Springer-Verlag, 2007.

[11] Shelley Nason and John E. Laird. Soar-RL: Integrating reinforcement learning with Soar. In *Cognitive Systems Research*, pages 51–59, 2004.

[12] Aurelio Prifitera, Donald H. Saklofske, and Lawrence G. Weiss, editors. *WISC-IV Clinical Assessment and Intervention*. Elsevier, 2nd edition, 2008.

[13] Sheldon M. Ross. *Introduction to Probability Models*. Elsevier, 10th edition, 2010.

[14] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. *Learning representations by back-propagating errors*, pages 696–699. MIT Press, 1988.

[15] G.A. Rummery and M. Niranjan. On-line Q-learning using connectionist systems. Technical Report CUED/F-INFENG/TR 166, Cambridge University Engineering Department, September 1994.

[16] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 3rd edition, 2010.

[17] Robert J. Schalkoff. *Artificial Neural Networks*. McGraw-Hill, 1997.

[18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[19] Gerald Tesauro. Practical issues in temporal difference learning. In *Machine Learning*, pages 257–277, 1992.

[20] Gerald Tesauro. Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3):58–68, March 1995.

[21] N.J. van Eck and M. van Wezel. Reinforcement learning and its appplication to Othello. *Computers and Operations Research*, 35(6):1999–2017, 2008.

[22] C.J.C.H Watkins. *Learning from Delayed Rewards*. PhD thesis, University of Cambridge, England, 1989.

[23] C.J.C.H Watkins and P. Dayan. Technical note: Q-learning. *Machine Learning*, 9(10):279–292, 1992.

[24] Scott Weaver, Leemon Baird, and Marios Polycarpou. An analytical framework for local feedforward networks. *IEEE Transactions on Neural Networks*, 9(3):473–482, 1998.

[25] Bernard Widrow and Marcian E. Hoff. *Adaptive Switching Circuits*, pages 123–124. MIT Press, 1988.