

The Connect Four Artificial Player

Allen Brubaker, Jason Long
December 14, 2010

Abstract

Various reinforcement learning techniques were applied to the game of Connect Four to develop a viable artificial agent that was shown to perform well against human players. Utilizing a back-propagation neural network able to adjust momentum and learning rates on the fly, these techniques were taken from the temporal difference class of methods and applied to the game. Training was accomplished by a neural network engaged in self-play, where board states represented the input to the structure, and outputs computed were a prediction of future result of the game assuming perfect play thereafter. Testing was performed on a validation set composed of an optimal board database of all 8-ply positions and their theoretical results. The performance was measured using an L2 Error Norm, namely, root mean squared error (RMSE) to analyze the neural network's prediction power and how this differed from the true theoretical results. Additional techniques were explored in an effort to lower error rates. These included using a MiniMax algorithm and probability distributions to guide move selection and adding additional feature detectors in addition to using only the board state. The best root mean squared error found was .37324 compared to purely randomized game play, which yielded error rates of around .5 to .6. Though the error rate is only a modest .37324, it was shown to perform well enough to beat an intelligent human player with an average win loss ratio of 8:1. This led to the conclusion that using board state in concert with simple feature detectors would produce optimal results for any game playing agent.

Contents

Abstract	1
Introduction	3
The Game	3
Background	4
Reinforcement Learning	4
Solved Game	6
Learning System	7
Task, Performance Measure, and Training Experience	7
Target Function	8
Target Function Representation	8
Learning Mechanism	9
Method	10
Artificial Neural Network	10
K Nearest Neighbor	11
Learning Algorithm Applied to Connect Four	11
Input Value Transformation	12
Experiment	13
Overview	13
Qualitative Analysis	13
Appropriate Lambda for Human Play	14
Quantitative Analysis	14
What about K-nearest-neighbor	15
Additional Tests	15
Conclusion	16
Source Code	17
Appendix	17
Works Cited	24

Introduction

The game of Connect Four is simple to learn yet complex enough to allow those that devote time and effort to learning its various strategic nuances to attain champion status. Our goal was to embark on a journey to produce such a champion—one that was not human but artificial, playing at a level that would rival even the most seasoned players.

What follows is an introduction to the game of Connect Four along with various strategies entailed in successful game play. Background knowledge on the subject of reinforcement learning and how it relates to the current problem along with the status of Connect Four as a solved game and implications stemming therefrom is then provided. A discussion on the learning system commences thereafter, and finally implementation details and experimentation results are given.

The Game

Connect Four is a game rumored to have been invented by David Bowie, was first distributed under the famous Connect Four trademark by Milton Bradley in February 1974 (Wikipedia, 2010). During the length of the game two opponents, with an initial arsenal of 21 similarly colored checkers differing in color from their opponent's pieces, seated around an upright board of 7x6 grid, take turns dropping their pieces into the board from the top. The game is similar to Tic-Tac-Toe but differs in that placement in the board cannot be made arbitrarily. Instead it is guided by gravity to rest in one of 7 possible positions atop the previously placed pieces. The goal of the game is simple: Connect four pieces of one's checkers contiguously either vertically, horizontally, or diagonally before the opponent has a chance to do so.

While the game is simple enough to understand, deep tactical knowledge must be attained in order to excel at the game. Naively spotting a contiguous line of three pieces and block or exploit the position as needed is simple enough, but the true secret to excellent gameplay is forcing the opponent to take a move, which opens up a winning position for the current player. A few of these strategic nuances are outlined below (Schneider, Luis, & Rosa, 2002) (Wikipedia, 2010).

- ❖ **Column Placement** – Because the game has been mathematically solved, it is proven that the first player to go will force a win, given that this player performs optimally and first starts in the middle column. However, if the player drops a checker in the two adjacent positions, a draw will follow. Finally, placing in any of the remaining columns allows the second player to force a win given that this player plays optimally.
- ❖ **Offense or Defense** – The question of whether a player should play offensively or defensively is always of interest when playing a game. Interestingly enough, the first player that goes clearly has the advantage from the above column placement strategy. Thus, the first player must play offensively while the second player should play defensively.
- ❖ **Row Placement** – Here the precept, BERO, should be adhered to. This stands for *blue in even rows and red in odd rows*, where red is the first (attacking) player to go, and blue is the second (defending) player to counter. The concept states that the attacker should try to place his pieces in

odd rows starting from the bottom, whereas the defender should place his pieces in even rows. The structure of the game that emerges is optimal.

- ❖ **Middle Row** – Occupying the largest ratio of pieces in the middle row, subject to the BERO rule, has a greater chance of success.
- ❖ **Connect 3 and Triangle** – To win a game of connect 4, one must first win a game of connect 3. Having multiple contiguous lines of 3 elements will allow for easy winning. Furthermore, arranging 3 contiguous lines of 3 in a triangle pattern will guarantee victory.
- ❖ **Forced Moves** – The optimal way to play is to force an opponent to move in a certain position, thus creating a winning position for the current player. This can be observed in the following two scenarios. If the current player creates a contiguous line of three with an empty space on either side, the opposing player must block one of the openings, allowing the current player to complete a line of four on the other side. Secondly, forced open lines occurs when for instance two stacked adjacent horizontal lines of four of similar color, each missing an element on the same column, exists in the current board. The defending player places his checker to block the bottom line of four, but this opens up the second line of four for the current player, thereby winning the game by completing the line of four.

Background

Additional information should be expounded upon before delving into a discussion on the exact specifications of the Connect Four learning system and its implementation. What follows is a discussion on the learning paradigm used by the Connect Four learner, the status of Connect Four as a solved game and the implications this brings, and lastly a more in depth look at the optimal database used in the performance measure of the Connect Four learning system.

Reinforcement Learning

The fundamental concepts inherent in *Reinforcement Learning* can be understood best when contrasted to *Supervised Learning*. In supervised learning, a learner or agent walks hand in hand with an “expert” or supervisor in any given environment. Thus, the agent is directly taught by one already a master in how to best act or respond in a given situation in the environment. This is contrasted to reinforcement learning where the agent is dropped in the middle of an unknown environment and expected to learn without any explicit teacher guiding its steps. It must therefore “test the waters” by taking actions and receiving direct feedback from the environment itself, known as a reward. In this way the agent will learn to act in ways most agreeable to the environment to obtain far greater rewards. One can see a direct corollary to this style of learning in the human experience. We find ourselves far more commonly, even as infants when first plunged into this jungle of an environment, learning by experience rather than by a direct teacher. The reward is simply the pleasure or pain experienced by taking an action. Humans wish to maximize pleasure by taking positive actions that exploit the nature of the environment. In other words, we learn to develop and adhere to a policy, or a complex set of precepts that guide our movements in the environment. In summary, an agent takes an action in an unknown environment, receives a reward for its actions, and finds itself in a new state in the environment. Clearly reinforcement learning as it applies to artificial agents is advantageous in that specific knowledge about

the environment as well as direct feedback at each step by an expert teacher or mentor is not required. Thus this style of learning can be applied to any goal directed and decision-making problem where expert teachers might not exist due to the unknown nature of the current environment. (Ghory, 2004)

More formally reinforcement learning can be defined as follows. An agent and an environment continuously interact in a sequence of time steps. Time steps might be infinite or finite, this representing an example of an episodic task. A robotic vacuum cleaner never stops cleaning (infinite). A game is eventually won or lost (finite). The goal of an agent is to maximize expected return or the rewards of the current and subsequent actions, discounted by a value between 0 and 1 the more distant the action from the current. Through interaction in an environment the agent learns a policy π , or the rule-set for traversing the environment (maps states to actions). Secondly a value function of the policy, $V^\pi(s)$ is a value representing the expected return starting from state s and traversing the environment guided by policy π . The goal is to find the optimal policy and therefore value function to maximize expected return. (Lenz, 2003)

In the case of board games and more specifically Connect Four, the environment is defined by the game and its rules along with interaction with an opponent. Unfortunately, rewards are not given immediately for a given action. Only when the game terminates does the reward come in the form of victory, defeat, or a draw. The fundamental problem involves how to correctly reward intermediate steps for which no immediate rewards were given. How much did intermediary steps contribute to the end result of a victory? The problem is known as the *Temporal Credit Assignment Problem*. It is this area that has received heated research in recent years and the problem itself must be dealt with fully before any successful board game agent can be developed. (Ghory, 2004)

Various techniques have been designed to address these exact issues. One such technique is known as Temporal Difference Algorithm, or TD(λ). A TD(λ) based system consists of various methods to properly assign rewards to intermediary steps based on the end game result. Thus it labels specific boards as favorable or not favorable and locates various features of the boards that contribute to this result. These patterns are then captured in a knowledge base, whereby the system as a whole learns to play better. The more games with an opponent are played, the more features are extrapolated from the data, and the more proficiently the game can be played in the future. The knowledge base may sometimes be a rote matrix of previously seen game board states, or usually it is a neural network known for its ability to capture the essence of a function using a much smaller storage size (set of weights) and generalize to future unseen data (board states). (Ghory, 2004)

The technique used for our Connect Four agent is a simple element in the Temporal Difference class of methods, TD(λ) being another such element. The technique used simply assigns the score of the current board state to be the score of the successor board state, where it is again the current agent's turn. In this way the evaluation function, V , which outputs score $V(b) = \epsilon$ for each board state b , is incrementally refined based on successor board states. This is based on the idea that scores near the end of the game more accurately represent the true value of a board. "Under certain conditions this approach can be proven to converge toward perfect estimates of the true intermediate board score." (Mitchell, 1997)

Solved Game

A unique way to measure a game exists, which in fact is a way to measure the hardness or complexity of the game. This measure is the notion of *solved* or *unsolved*. A game is solved if its outcome can be predicted by an algorithm at any arbitrary state long before one naively can cite that the game is over. If a game is not solved it is known to be *unsolved*. In addition there are various measures as to how *solved* a game is. A game is known to be *ultra-weakly solved* if a mathematical proof exists that can prove that the first play will win, lose, or draw given that both players play optimally. Moreover, a game is known to be *weakly solved*, if an algorithm can be produced that will ensure that a particular player wins or draws regardless of how the opposing player performs. Finally, the quintessential solved game, or a *strongly solved* game, is one that not only can produce the above algorithm but further enforces that the algorithm be able to take as input any board state, even if the board contains egregious errors made by both players, and produce perfect play (Allis, 1994).

Historically, these algorithms known to have solved games share one important feature. They commonly produce an evaluation function that takes as input a board game position and ranks the board accordingly. This rank is then used to select subsequent moves which guides the algorithm or agent in producing optimal or perfect gameplay. One inherent flaw is such that game positions closer to the end of a game are more accurately evaluated as compared to those in the beginning. The first type of widely known type of algorithm is one that trivially uses brute force techniques in comprehensively searching every possible move sequence thereby obtaining its optimal theoretical result. This produces a lookup table that can be instantaneously queried in actual gameplay to provide optimal gameplay. Other more impressive algorithms have exploited ideas found in minimax search to produce some of the world's strongest players—improvements such as alpha-beta pruning and Buro's probabilistic Multi-ProbCut algorithm. Finally, algorithms using TD(λ) found in Tesauro's TD-Gammon working synergistically with a neural network have produced striking results. Only recently have additional feature detectors been added as input to a TD(λ), Neural Network hybrid in addition to the merely inputting the board state. This has produced truly extraordinary results (Ghory, 2004).

In reference to the current task of producing an agent to play Connect Four, it is indeed solved. It was first weakly solved mathematically by James D. Allen in October 1, 1998. Only 15 days later, the game was independently solved by Victor Allis in his thesis entitled: *A Knowledge Based Approach to Connect Four*. The discourse details VICTOR, an artificially intelligent agent, and how, with only a simple set of 9 simple strategic rules backed by a database of approximately half a million positions, could win every game starting with white by placing the initial piece in the middle column. Inspired by these results, John Tromp set out to strongly solve the game. He accomplished just this when in November, 1995 he published his results and database to the UCI Machine Learning Repository. The database took approximately 40,000 hours of computation on the Sun and SGI workstations at Centrum Wiskunde and Informatica (Tromp, 2005). As informed by the repository, "the database contains all legal 8-ply positions in the game of connect-4 in which neither player has won yet, and in which the next move is not forced" (UCI, 1995). It is interesting to note that the game is clearly not trivially solved. Using a brute force method took thousands of hours to analyze the 4.5 trillion unique positions in a Connect

Four game. Thus, the task at hand of creating an artificially intelligent Connect Four player without using Brute Force indeed remains a formidable task.

The database is presented as a text document of 67,557 instances. Each instance contains the 42 cells of a board state with exactly 4 x's and 4 O's and finally the theoretical result. If the instance was classified as a *win*, the next player that went was guaranteed a win if playing optimally regardless of the actions of the opposing player. A *draw*, on the other hand, meant both plays would at best draw if both play optimally; and lastly a *loss* meant the next player to go would lose if the opposing player performed optimally regardless of the next player's actions.

This database was vital in the decision to create a learner for the game of Connect Four. The reason for this was because it was easy to benchmark the performance of the any learner trying to learn to play Connect Four well. The current learner as detailed in this document solely used this optimal move information only as a means to benchmark the performance of the system. It was in no way used to train. This would have been considered cheating, since a purely reinforced method of learning was adopted and adhered to.

Learning System

Learning as it relates to machine learning is defined as follows: "A computer program is said to learn from experience E with respect to some class of tasks T and performance measure P, if its performance at tasks in T, as measured by P, improves with experience E" (Mitchell, 1997). Thus a well-posed learning problem is one that well defines these three features: The task, a measure of performance, and the source of experience. Moreover, such an artificial learner must properly cite its design components. This consists of the exact type of knowledge to be learned, or the target function; the underlying representation of its target knowledge, or the data structure and algorithms used; and the learning mechanism. The learning mechanism utilizes the target representation to learn the target function to perform well at a given task within its training experience, which should capture the probability distribution of real world examples (Mitchell, 1997). What follows is a detailing of the learning system that was created to learn the game of Connect Four.

Task, Performance Measure, and Training Experience

The learning task entailed aptly and intelligently playing Connect Four. Because the game has already been mathematically solved and multiple programs and algorithms developed, an online database of optimal moves was found. Thus to evaluate the current performance of the Connect Four learner, a simple evaluation of each board state was found and compared to the correct answer. The root mean squared error between the predicted score or evaluation and the actual theoretical result as obtained from the database would naturally encompass our error measure. As for the training experience or environment, it was desirable to select one that would be allowed to run interminably, producing better results as time increased and to reduce human interaction that would retard or impede learning process. Thus, a process of unsupervised reinforcement learning only utilizing information gathered from previously played games was selected. The training examples were generated by the learner

playing against itself in endless games, at each step producing a list of board states that represented the game from beginning to end.

Target Function

Specification of the target function is important, because it is this function that reduces the problem to one of emulating this function as closely as possible. If perfect emulation is achieved, a perfect learner emerges. The target function was one that would take as input a board state, or the board containing all the checkers present at a slice in time, and output a score of 1 if the previous player that went is guaranteed a win and the player plays optimally regardless of how the opposing player plays.

Conversely, the function would output a score of 0 if the previous player is guaranteed a loss and the opposing player plays optimally regardless of the performance of the previous player. Finally, a value of $1/2$ is output if both players are guaranteed a draw and both players play optimally.

Target Function Representation

The target function representation, or hypothesis used by the Connect Four learner must be one that produces a value or score that guides the learner in selecting the next best move. It is trivial to find the list of legal subsequent moves from the current state—recall that there are at most 7 of them, one for each column. However, selecting an optimal state to move to from the current state is not so trivial. If the learner already knew the target function, nothing would need to be learned and the best move could easily be ascertained. Unfortunately this was not the case, therefore the learner used its current “best guess” function, generated all possible next board states by dropping one checker in each column individually, and evaluated the board state. The board state with the highest score informed the learner that this was the next best state to actually move to.

Two well-known machine learning ideas were used as the target function representation or hypothesis for the Connect Four learner. These were an artificial neural network (ANN) using back-propagation and a k nearest neighbor. The notion of an artificial neural is one that tries to correctly learn a function that takes as input a set of values and outputs a set of output values. This is encompassed in a network of neurons, the first layer known as the input layer which accepts these input values, the last layer known as the output layer, and the middle layers known as hidden layers. The neurons are connected by means of edges each containing a weight. Learning takes place by adjusting weights based on a simple process of stochastic gradient descent through the error space as found by the current set of weights. The mathematical model was inspired by the structural and functional aspects of natural occurring biological neural networks. These mathematical models are well suited to classifying non-linear complex relationships between inputs and output values.

On the other hand, a k nearest neighbor (KNN) algorithm is an instance based learner that takes a set of training examples with labels and when given a new example or instance, classifies the instance using the majority vote of its k nearest neighbors. Closeness is determined by various distance metrics. Furthermore, KNN is a type of instance-based learning or lazy learning because the majority of the computation is deferred until classification time as opposed to training time, where examples are simply stored.

Learning Mechanism

Learning is accomplished by the specific algorithms used above, either KNN or ANN. Due to the nature of the training environment and the fact that delayed rewards are given for having taken a move, labels or scores at intermediate steps are unknown. Clearly at the end of the game the label of 1 can be given for winning, 0 for losing, and .5 for drawing, but how should intermediate steps be scored? Perhaps moves in the beginning were played optimally yet as the game progressed performance of moves degraded and the resulting game was lost. How can one score these intermediate board positions appropriately? All these decisions need to be appropriately addressed. The way the Connect Four learner labels intermediate states and learns are detailed below.

The Connect Four learner plays a game against itself with the current hypothesis V' . Recall that in the case of the current Connect Four learner, the hypothesis V' is either the current set of weights in the neural network or the example space of the k nearest neighbors. This hypothesis is used to guide the move or a board from state to state. It is simply the immediate successor board state that has the maximal score among all the other possible next states (up to seven of them). In other words, it is $\max\{V'(b')\}$, where $b' \in \text{NextStates}(b)$ and b is the current board. It is important to note that no learning takes place while playing a game—the hypothesis V' (ANN or KNN) remains constant. At the end of a game, the learner generates a sequence of training examples for that game along with a subsequent rating of the board, known as $V_{\text{train}}(b)$ for each board b (detailed below). These are the labels for each board in the current game trace (set of board in the game). Thus, each example is in the form $(b, V_{\text{train}}(b))$. Once examples are generated and properly labeled, the examples are fed into the ANN or KNN and learning, or training, takes place. After learning, V' is refined a little bit more and represents the target function V to a greater degree.

Given a set of examples without labels, the label or score $V_{\text{train}}(b)$ is estimated in the following way: $V_{\text{train}}(b) = V'(\text{successor}(b))$, or the score obtained from the next board state for the current player. This is precisely the board state following the current player's move and the opponent's response. This is based on the assumption that V' is more accurate towards the end of a game than the beginning. "Under certain conditions, this approach of iteratively estimating training values based on estimates of the successor state values can be proven to converge toward perfect estimates of V_{train} " (Mitchell, 1997).

One important facet of the above process is the following: As detailed above because $V_{\text{train}}(b) = V'(\text{successor}(b))$, in every one trace of boards in a game, there are actually two games occurring: One from the perspective of player 1 and another from player 2. Training values are calculated for player 1 every odd board, while for player 2 values are calculated every even board. Clearly training two different neural networks is counterproductive; thus, each board is normalized so that from the perspective of the current player he is always the blue checker, and the opposing player is always the green checker regardless of their true checker color. Thus two training sessions are gathered from every one game.

To summarize the learning system:

❖ **Task T:** Playing Connect Four intelligently

- ❖ **Performance measure P:** Root mean squared error computed from optimal board score database.
- ❖ **Training Experience E:** Games played against self
- ❖ **Target Function:** $V: \text{Board} \rightarrow \{0, .5, 1\}$
- ❖ **Target Function Representation:** $V'(b) = \text{ANN}(b)$ or $\text{KNN}(b)$.
- ❖ **Learning Mechanism:** $V'(b) = V'(< b, V'(\text{Successor}(b))>)$

Method

We will now describe how these learning algorithms were implemented, mentioning specific classes and methods used in the source code, which is submitted along with this paper.

Artificial Neural Network

The neural net was implemented in a generic way, so that the code we developed could be used for any machine learning problem. The constructor to the “Network” class accepts several parameters that controls the structure (number of layers and number of nodes in each layer) of the network. One can have the network predict a value by using the `PropagateInput` method. One can label an example and give it to the `LearnOneExample` method to have the neural network learn from that example. One can save a neural network to disk using the `Serialize.Serialize()` function, then load that same network (including all weights) on a future execution of one’s program using `Serialize.Deserialize()`.

The “Example” class is how instances are represented to the neural net. The instance itself is represented as a sequence of floating point values, called the “features” of the example. The Example class can optionally hold a “prediction” and a “label” as well, which are also sequences of floating point numbers. When a network is asked to predict a value with `PropagateInput`, it is given an instance of the Example class, and it will set the “prediction” of that example. Similarly, when a network is asked to learn from an example, it expects the example to have its “label” set.

The nodes of the neural network are implemented using the “Neuron” class. Each neuron maintains a list of nodes it outputs to, and a list of nodes it receives input from. We used a fully connected neural net for Connect Four, i.e. all nodes of one layer connect to all nodes of the next layer. The neuron also has a list of “weights”, maintained along with the list of nodes it connects to. The weights are not stored directly, but they are actually objects themselves, of the “Weight” class. This allows the same weight object to be referenced by the downstream node and the upstream node.

Our neural network also has a constant node which is connected to every other node. The constant node always has a value of 1, but the weights on its connections to other nodes vary as needed by those nodes.

Finally, our neural net library has a sophisticated set of training algorithms, supporting things like automatic learning rate decay and momentum. These are controlled via the `NetworkParameters` class, and require use of the `TrainNetwork()` method to do training. The `TrainNetwork` method differs from the `LearnOneExample` method in that it learns from a set of examples rather than a single example, and that it implements learning rate decay, momentum, and termination conditions. Termination conditions

can specify when the network has been trained “enough”, and can be either a predefined number of training iterations, or a validation set of examples and an error threshold.

K Nearest Neighbor

We determined that a single learning algorithm may not give us sufficient data to work with, so we resolved to implement a completely different learning algorithm, but using the same interface so we could easily substitute a different learning algorithm for comparison. As a result, we implemented a simple K-nearest-neighbor algorithm.

Our K-nearest-neighbor algorithm was designed to predict a continuous variable (rather than choose from a set of classes), and allow a continuous stream of new examples to learn from (rather than a finite set of examples). The design is simple. As new examples are given (through the `LearnOneExample` method), they are added to the repository of known examples. Once a size limit has been reached, the `LearnOneExample` method will remove the oldest example to make room for the new example. This approach works well with our use of reinforcement learning, since when the algorithm starts the examples it is fed will not be very good labels, but as the machine learns the game, the examples it uses to learn should get better. Continually generating new good examples will eventually flush out the bad examples.

Learning Algorithm Applied to Connect Four

As described earlier in this paper, the basic concept is that the machine will learn, on its own, how to play Connect Four by generating its own examples through random play against itself, and then using the end result of the game to label the examples, from the end of the game back to the beginning moves in the game.

In our implementation, the Bot class has responsibility of choosing which move to play in a game. The learner is expected to evaluate how “good” a particular board position is, and assign a number to represent that goodness. A higher number means the bot is more likely to win than a lower number. The highest number would represent a board position that has already been won, and the lowest number represents a board position which has already been lost.

At any point in the game of Connect Four, a player has no more than seven choices of moves available to it. The bot chooses which move to make by considering the board position resulting from playing each of those seven moves. The learning algorithm is asked to predict a “goodness” value for each of those (up to) seven board positions. The “best” move would be the move whose resulting board position has the highest goodness.

When we train the network, we are careful to generate examples such that the board position is “normalized” so that the player who has just placed a checker is always the blue color, and the opponent is the green color. This makes our examples consistent, and ensures that if the learner outputs a large number, it always means the player who has just gone has the best chance to win. When it comes time to label an example, we use the predicted evaluation of the successor, i.e. the next move, the board state after the current player’s next move.

When the computer plays against itself we do not want the computer to always play the same way. This means we want the computer to be implemented such that it does not always pick its best move. But we want it to pick a good move nonetheless. (A good move ensures the successor move's evaluation will be an accurate label for the current move's evaluation.) We have implemented the "SelectMove" method, using our own invented algorithm, that picks a move with probability proportional to how good the move is. We have informally called our invention the "Lambda algorithm", since it uses Lambda as a tunable, though perhaps someone has invented it before us and given it a different name.

Here is how our Lambda algorithm works. Each move is initially given a score, the score corresponds to the "goodness" of the board position resulting from making that move. These scores have values from 0 to 1. The best score is determined. Then each move's score is subtracted from the best score so that all moves are nonnegative numbers, with the best move being 0. The probability of selecting a move is calculated as $1/L+s$ where L (Lambda) is a positive constant and s is the adjusted score for that move. The formula works out so that if one chooses a very small Lambda constant, the best move has a much higher probability than worse moves, and if one chooses a very large Lambda constant, the best move has roughly equal probability with the worse moves. This way, the amount of randomness a bot exhibits can be controlled by the choice of Lambda.

When learning, we want Lambda to be large enough that the bot will explore all moves, not just moves that it arbitrarily decides are good moves when it starts out. However, when playing against humans, we want the bot to choose the "best" move most of the time, so we use a smaller Lambda value.

Input Value Transformation

A Connect Four board state cannot be fed directly into a machine learning algorithm. Instead, the board state must be *transformed* into a vector of real-valued numbers for the learning algorithm to process. The choice in how to transform a board state into a vector can make or break a particular learning algorithm.

We started with a simple transformation that encodes all the information available, but keeps it in a pretty raw form. Since there are seven columns and six rows in the board state, we transform the board state into 42 numbers. For any empty cell, we use 0. For a cell containing the opponent's checker, we use -1. For a cell containing our own checker, we use 1. (As stated before, the board state is seen from the "perspective" of the player who just played.) These 42 numbers completely describe the board state, so therefore a neural network of sufficient neurons should theoretically be able to learn perfect play of Connect Four.

We were concerned that it would take many neurons and many training iterations for the network to draw out certain features from this board representation, so during our implementation we decided to try an alternate transform mechanism.

The alternate transform mechanism uses the same 42 numbers as above, and then adds 36 additional numbers. These 36 numbers encode information that can be determined from the first 42 numbers, but they are easy-to-calculate features that we speculated would be helpful for the learner. For each column, it is calculated whether there is a four-in-a-row containing three checkers of one color that just

needs a new checker in that column to complete the four-in-a-row. We do that for each column, for each player, for the first two empty rows in that column (that's 28 numbers). Furthermore, for each player we calculate, for all the potential four-in-a-row sequences on the board, how many have just one checker filled, two checkers filled, three checkers filled, and four checkers filled. That gives the other 8 numbers for the input vector.

Our assumption is that the neural network, if it finds these new numbers useful, will increase the weights on those numbers relatively quickly, and therefore learn faster than the neural network using the original 42 number input transformation. Furthermore, this new neural network has access to all the same information as the original neural network, so if the new numbers leave out important features of the board, those features are still represented in the first 42 numbers.

This alternate transform is called "Transform78" since it creates 78 numbers instead of 42. Note about the source code: selecting which transform is in effect must be done by changing the source code and recompiling.

Experiment

Overview

We created several learning data sets and had the computer train against itself for a certain amount of time. Some attempts we abandoned early because it looked like no progress was being made. We could measure progress in three ways: 1) Stop the training and play a game against the computer and evaluate qualitatively how well the computer was playing. 2) Look at what sort of predictions the computer was making near the end of each game (theoretically the predictions should be near 1 or 0 depending on win or loss). 3) Allow the neural net make predictions on our "validation set" and calculate the error rate of that.

Qualitative Analysis

When a neural network was initially created it would play pretty poorly. Each move would be effectively random, because the numbers outputted by the neural net did not represent goodness of the board, but just some arbitrary internal calculation it made using the numbers from the board state. As expected, the play of the computer was extremely poor. (It would sometimes help the human win!)

After a neural network is trained for an hour or more, it tended to play better. We found that the quality of a neural network's play was dependent on which transform method was being used.

The first analysis was done of a transform42-based neural network (Input42_Hidden100), after 526,500 iterations. There was clearly an improvement of the computer's play over the just-initialized network. However, it has a tendency to ignore threats (i.e. when the human has three checkers in a row and a spot available to play the fourth), which it should not do. The computer is not completely incompetent, but it is pretty easily beatable.

The second analysis was done of a transform78-based neural network (Input78_Hidden100), after only 6,500 iterations. Our assessment is that the computer will *usually* block threats and *usually* take advantage of opportunities. The human can still win most of the time, but on occasion the computer makes the right moves and finds a way to win. We found that the computer, when it was able to win, had setup a situation where it made a double threat (i.e. when there are two ways for it to win and only one can be blocked). This exhibits rather clever play. But at the same time, the computer still misses threats made against it, so it was both interesting and disappointing at the same time.

The third analysis was done on a transform78-based neural network after many more training iterations (Input78_Hidden100, after 100,000 iterations). We find that this neural network is a much more effective Connect Four player. In fact, it plays much better than the Input42 network, even though it has not been trained for as long. It will block virtually all threats made by the human, and tends to be pretty good at setting up a dual-threat that cannot be blocked. It was able to win against this paper's authors most of the time.

Appropriate Lambda for Human Play

The question of whether a lambda of zero should have been chosen when pitting a computer player versus a human player was inconclusive. The argument for setting Lambda to a nonzero value during training is so that the neural network will explore the instance space, but should Lambda be nonzero for play versus a human?

When Lambda is zero, the computer will play the same way to a particular sequence of moves. When the computer is weak, this behavior can be annoying because when one beats the computer and start over, one can just use the exact same sequence of moves to win again. It would be preferable to have the computer try a different sequence of moves so it does not keep losing. On the other hand, when the computer is consistently winning, then it is the human who is trying to create a different sequence of moves, so it is not an issue then.

Quantitative Analysis

In an effort to get an objective measurement at how well the neural net was learning, we validated it against the 8-ply database described above. The idea was to feed our neural nets a bunch of already-classified data and see how well it evaluated it. Unfortunately, we found that none of our trained neural nets could make much measureable progress using this metric. In other words, even though the neural network seemed to be playing better (against humans), it was still giving bad results to predicting labels in the 8-ply database.

We present a variety of neural networks, named according to how many input nodes and how many hidden nodes it had. Each graph shows the network's performance over the first 100,000 training iterations. Some of the graphs also include a "training error". These graphs were done after that part of the program was implemented. The training error is a running average of the error as measured for each new example fed into the training algorithm. All charts along with error rates can be found in the appendix. The lowest error rates obtained were using the transform78 neural network with 600 hidden

units. The lowest validation error was .37324. The graphs showing neural network performance can be found in the appendix to this paper.

What about K-nearest-neighbor

We described earlier our K-nearest-neighbor implementation. It is there in the source code, but switched off for Connect Four. It is switched off because we found it to make no headway at even predicting the end-of-game board states, which should theoretically be the easiest to predict. Upon reflection, we decided it was the nature of our input transformation, so we gave up on it.

The main problem with k-nearest-neighbor is the input representation. The k-nearest-neighbor learner operates under the assumption that two instances, when plotted in Cartesian space, if they are near each other will have similar values. However, in Connect Four, one could have two board states that are identical except for a single checker, but could have very different evaluations. That one checker that's different might make-or-break a four-in-a-row! Therefore, k-nearest-neighbor will only work if one has so many examples that not just nearest neighbors are found, but exact matches are found. In this case, one could expect to get good predictions, but finding the nearest neighbor is perhaps unnecessary and it would be enough to just find an exact match in the training set.

So, the appropriate solution would be to define an input transformation such that the instance points being near each other will actually mean something. We made several short-lived attempts at doing this, but each attempt was unfruitful and did not make it into our source code repository.

Additional Tests

In an effort to lower the error rate after it was found to be less than optimal, we designed and utilized additional techniques. Lambda logic was tweaked, moving based on scores at specified depths was explored, alternate means of labeling examples were also investigated, and alternate transform methods were utilized. Unfortunately, although more rigorous testing must be performed, initial testing produced only lackluster results no better than those already obtained.

Recall lambda previously used a probability distribution. Based on how good a score was compared to other scores out of the 7 total possible next moves, a move was taken. Instead of using lambda in this fashion, we simply took lambda to be a constant threshold from 0 to 1 in the following manner. If a random number generated fell below this lambda, then a random board position was taken regardless of its score. If the random number was greater than this lambda, the best move corresponding to the best score was taken.

A second approach taken in hopes of minimizing error was to investigate the possibility that perhaps looking at only the next set of 7 moves was not optimal. Instead, we implemented a recursive MiniMax algorithm, excluding alpha-beta pruning optimization, to locate the scores of all possible moves at a depth of k (or all k moves in the future). Once these scores were found, each successive level above level k were minimized (as would have been the goal of the opposing player), then maximized, and then minimized in an alternating fashion until level 1 was reached. Based on level 1's maximum score, a more informed move was taken based on not just data from the next immediate level of gameplay, but

instead based on data from all possible gameplay that could happen after k moves. Testing was done on a depth of $k=2$, and the only noteworthy results this gave us was to drastically slow down computation speed while producing no better results.

A third approach was to investigate alternate methods to labeling board states. Previously current board states were labeled with the prediction from the successor board state, or the board state when it was again the current player's turn. Instead, we tried utilizing a full-fledged $TD(\lambda)$ approach. Here we utilized information from all future successive board states instead of just the next immediate successor. Rewards were then discounted the more distant the successor board state was. Although λ , γ , and α values surely need additional fine tuning, initial results were no better than those already obtained.

Finally, different transforms were investigated. Recall that previously we had tested on simply taking the board state and feeding each cell of the board to an input node. Furthermore, we had taken the board state along with one feature detector and obtained our best results. We investigated with using an in depth set of feature detectors without the board state. The 34 features included the number of immediate potential contiguous lines of four, lines of three, and lines of two for the current player. Potential means if one more checker were added it would create a line of four. Other features were the number of checkers in each column, in each row, and across the entire board for the current player. These 17 features were extended to a set of 34 by counting checkers of the opposing player instead. These 34 features not including the board state acted as the means to transform a board to a set of values to feed to the neural net. Results are listed in the appendix section for 34 input units.

Conclusion

We have applied many different reinforcement learning techniques to the game of Connect Four, trying to find something that could effectively play the game as well as correctly predict the labels in the 8-ply database. We found the neural network, although more complicated to implement, learned the game more effectively than the k -nearest-neighbor algorithm. We determined this to be due to the input representation, and that k -nearest-neighbor assume that similar examples are close to each other in Cartesian space, but neural networks do not make that assumption.

We found that the input representation was an important determiner in how well the learning algorithm could learn the game. We started with a simplistic representation of the game board, then expanded it by adding features useful to Connect Four, and then tried using just the features with various results. The expanded representation seemed to work the best, with the most effective play against humans, and the lowest (by a small margin) error rate against the 8-ply database.

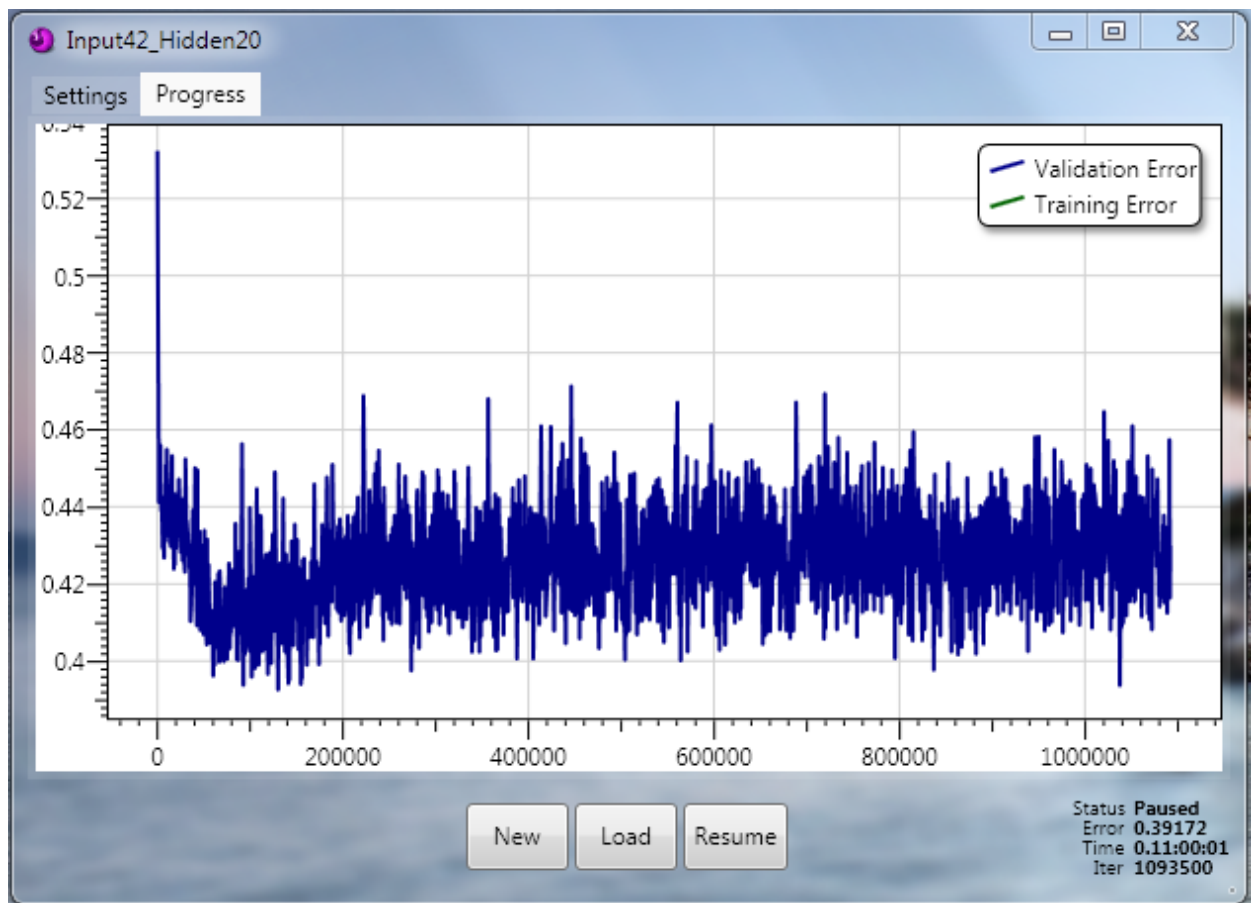
We found that even though our 8-ply database performance metric did not indicate much improvement by the neural network, its play against humans was much more impressive. We found that a neural network, with even a modest number of hidden layer neurons, could play well against humans, given enough training iterations and the right input representation.

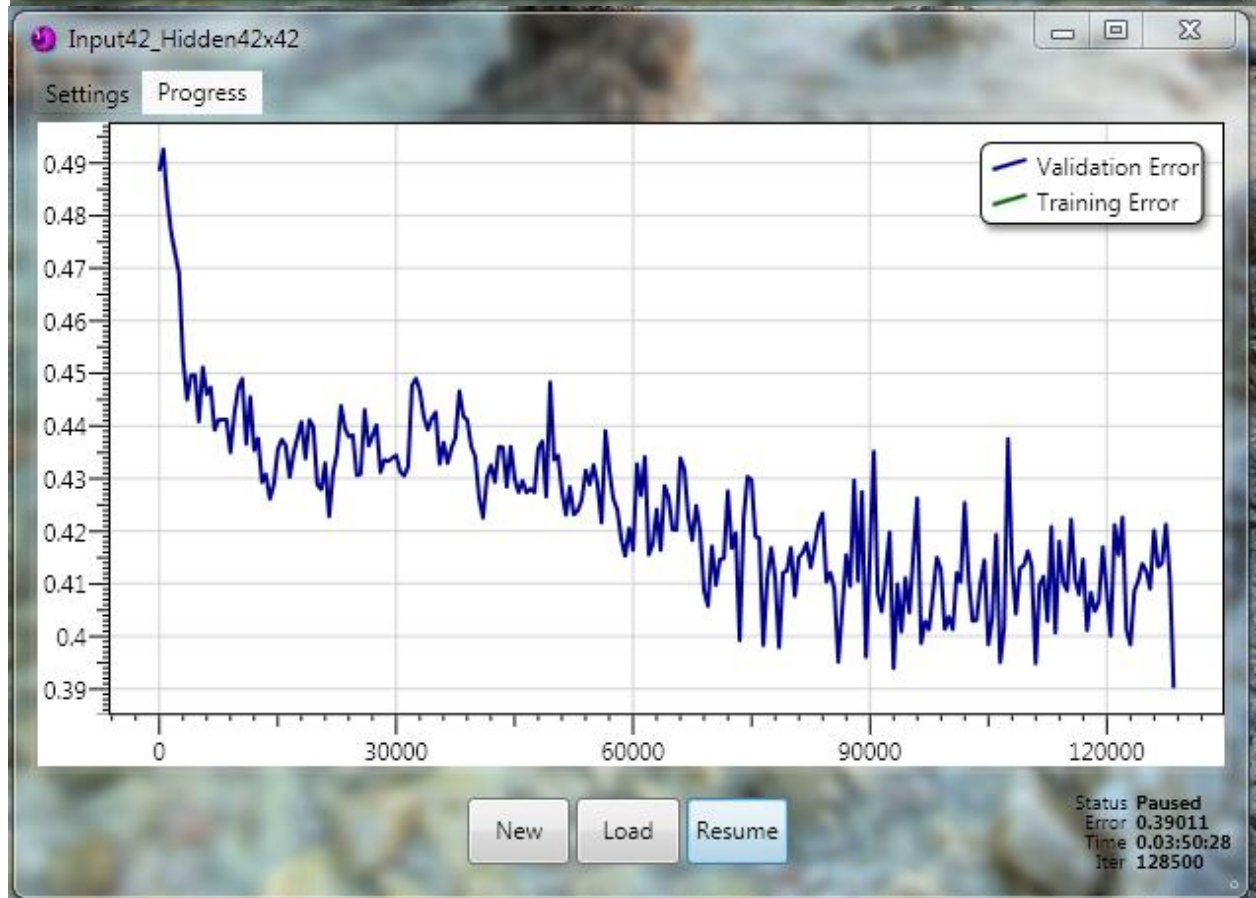
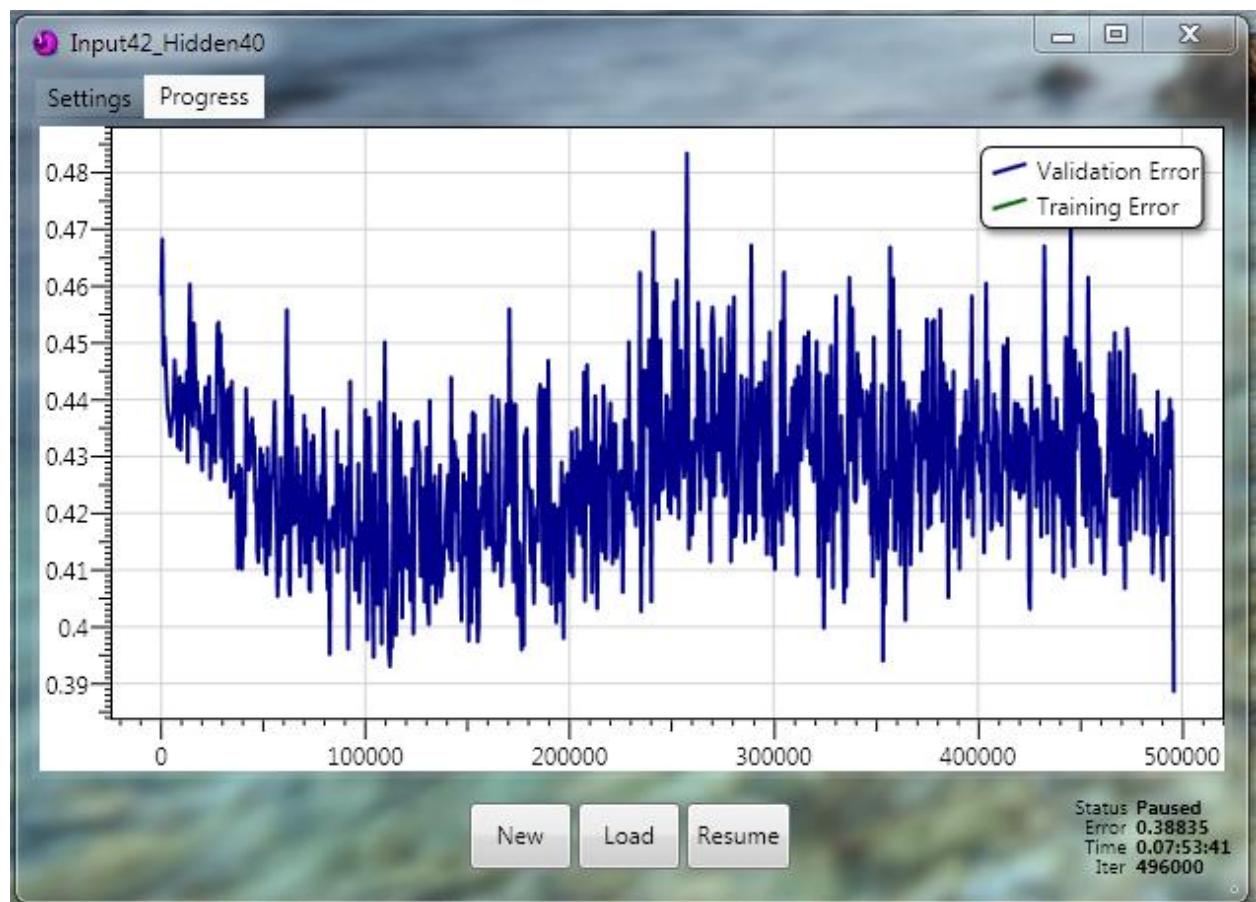
Source Code

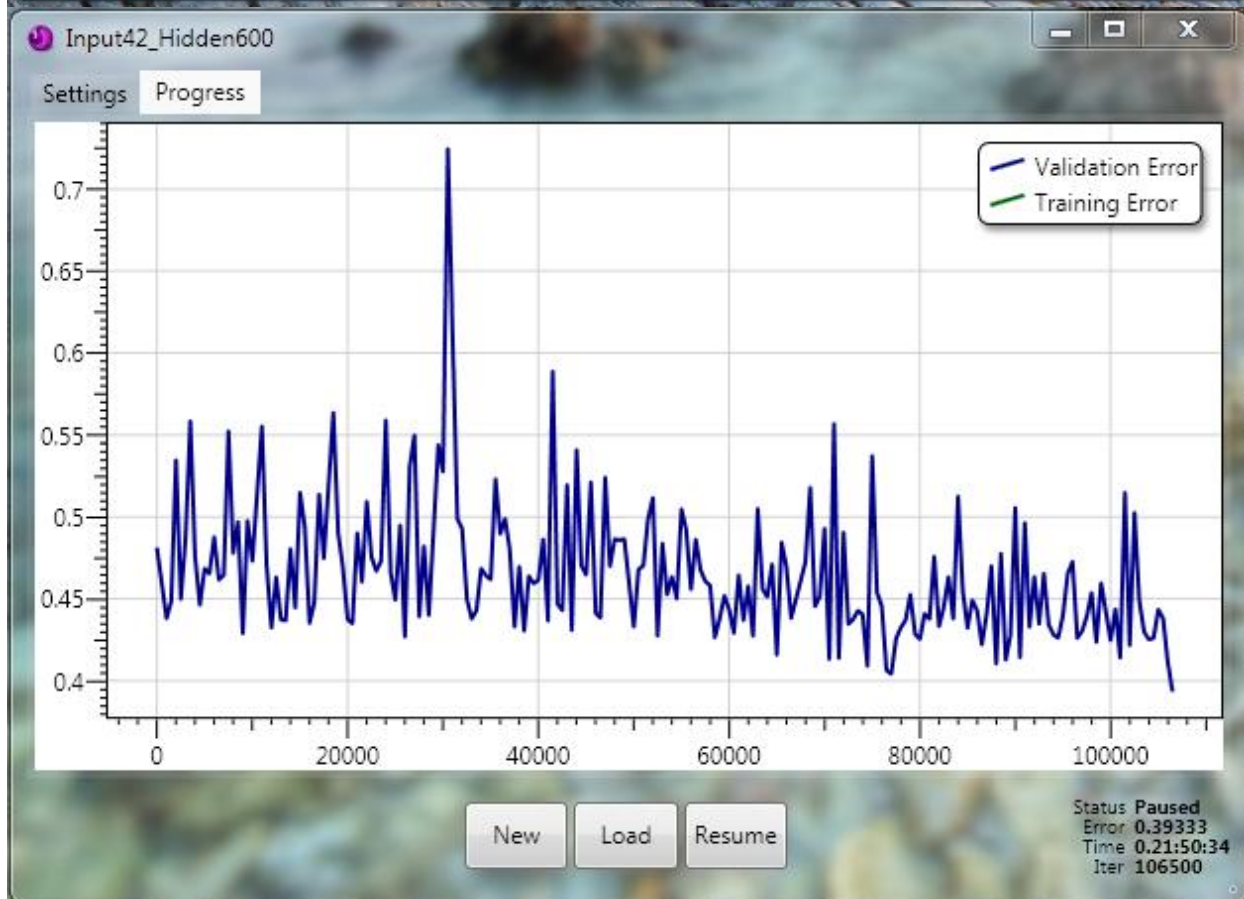
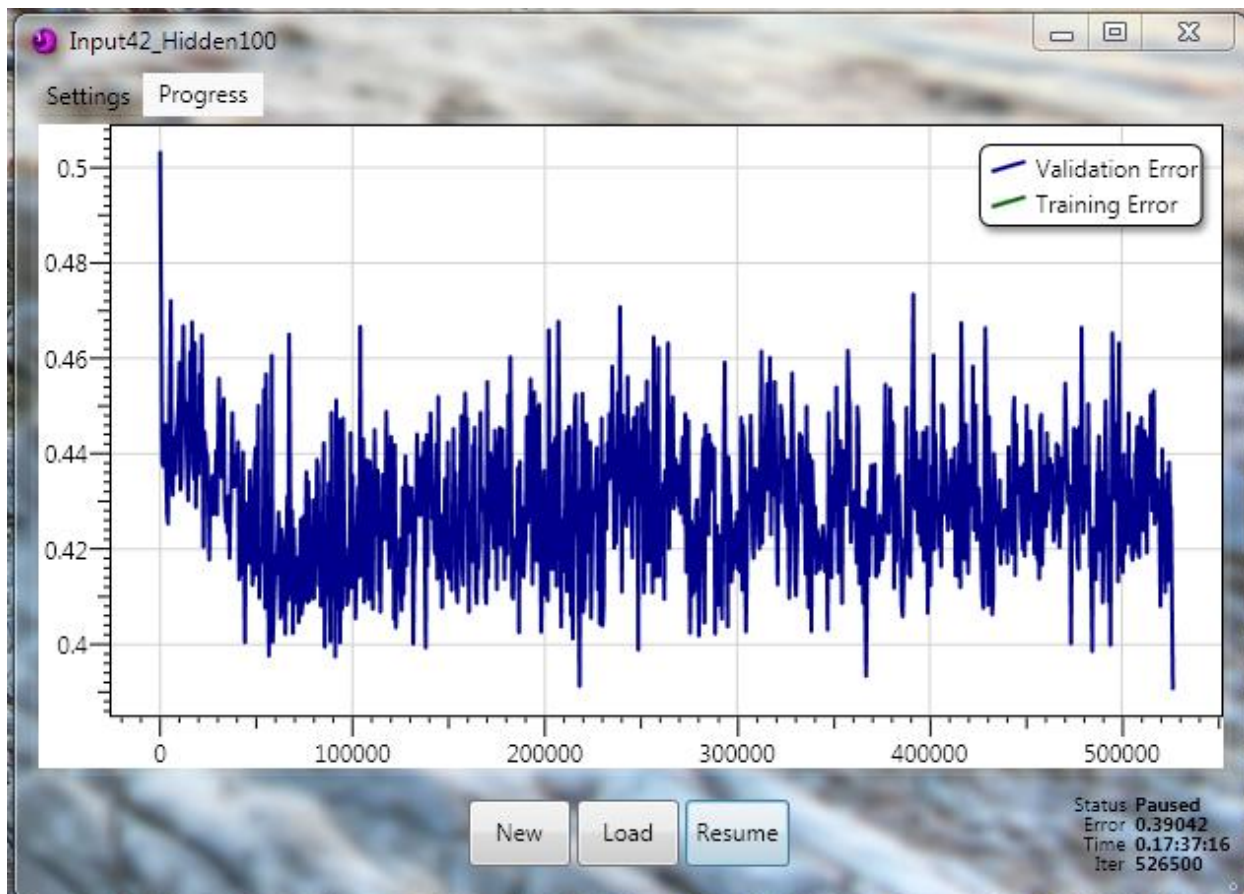
Source code related to this project is available online at <http://connect4neuralnetwork.googlecode.com>.

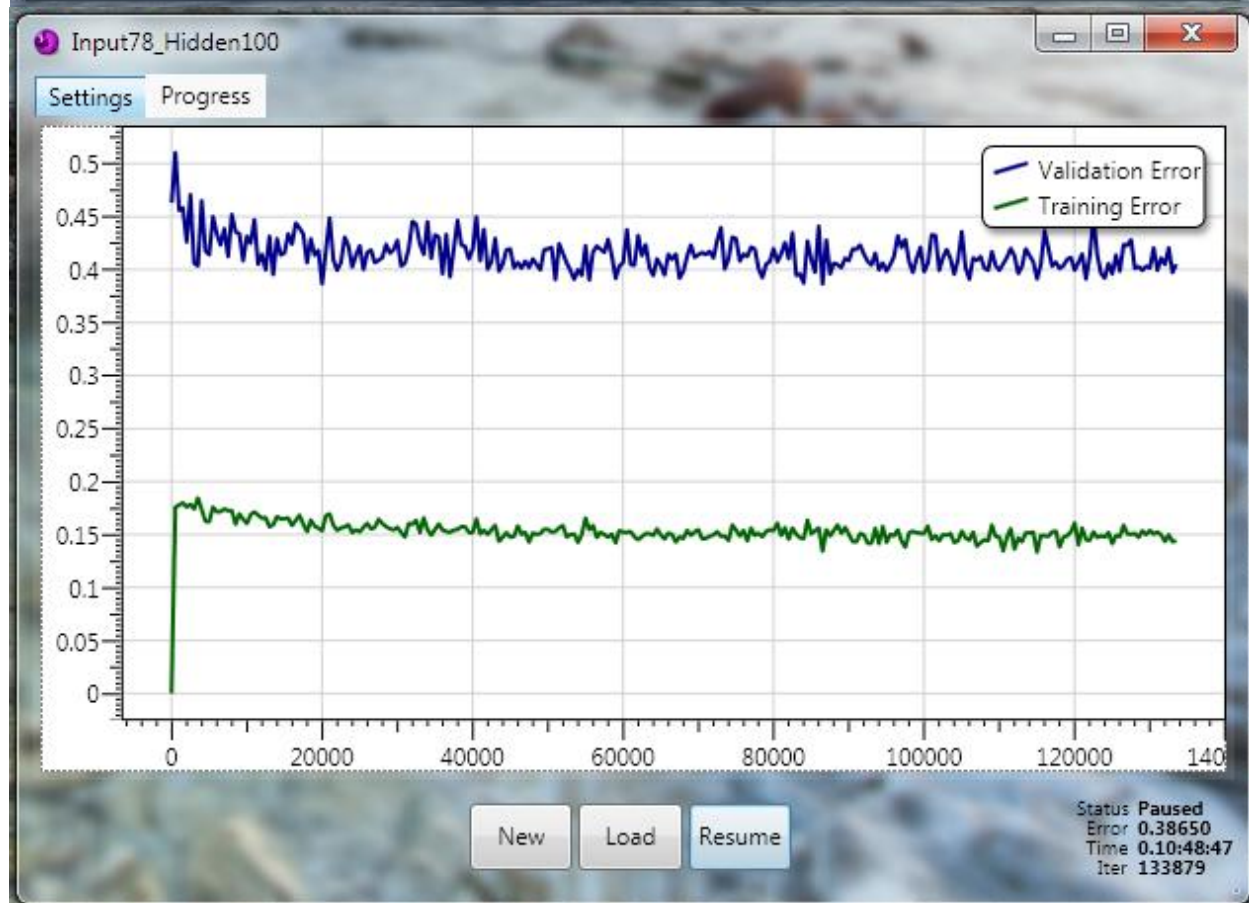
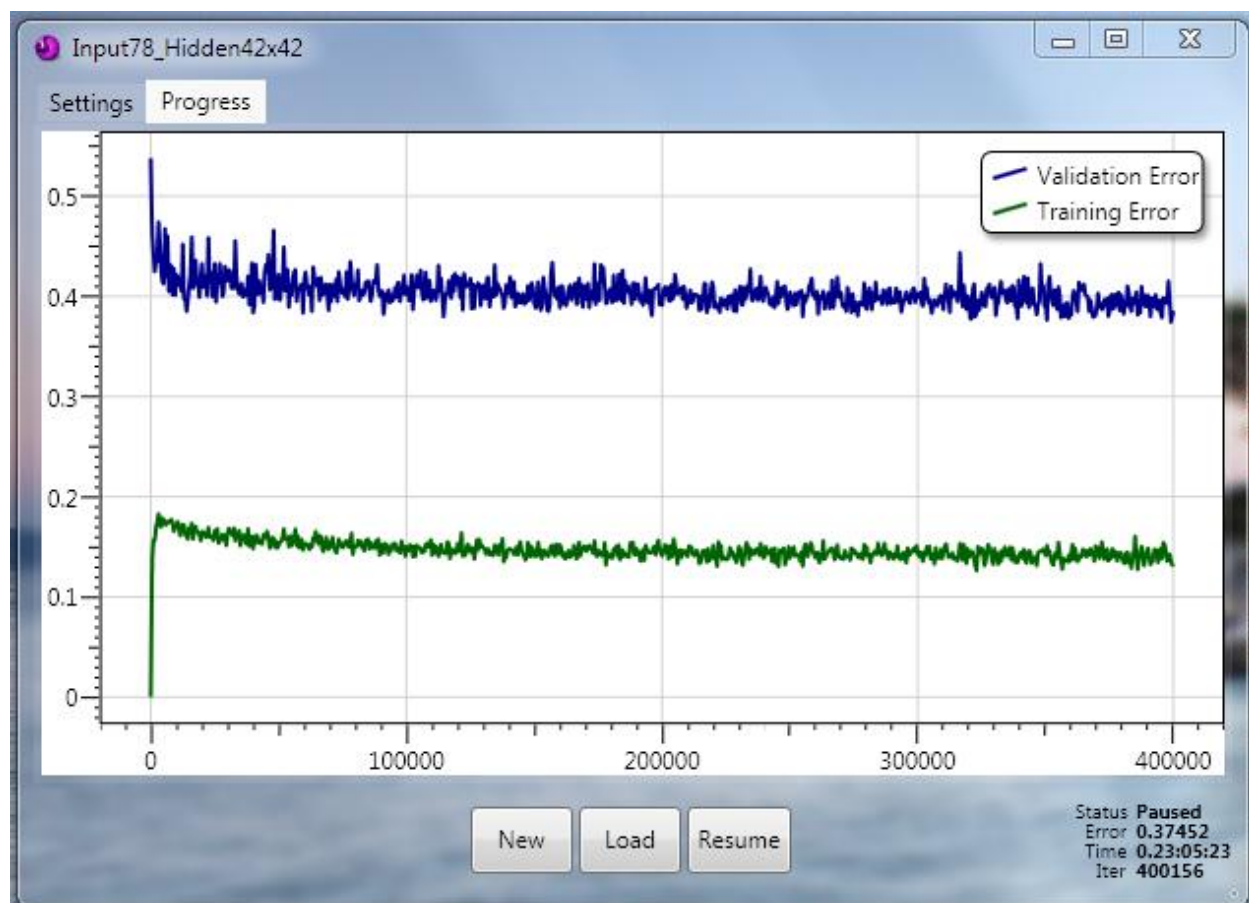
Appendix

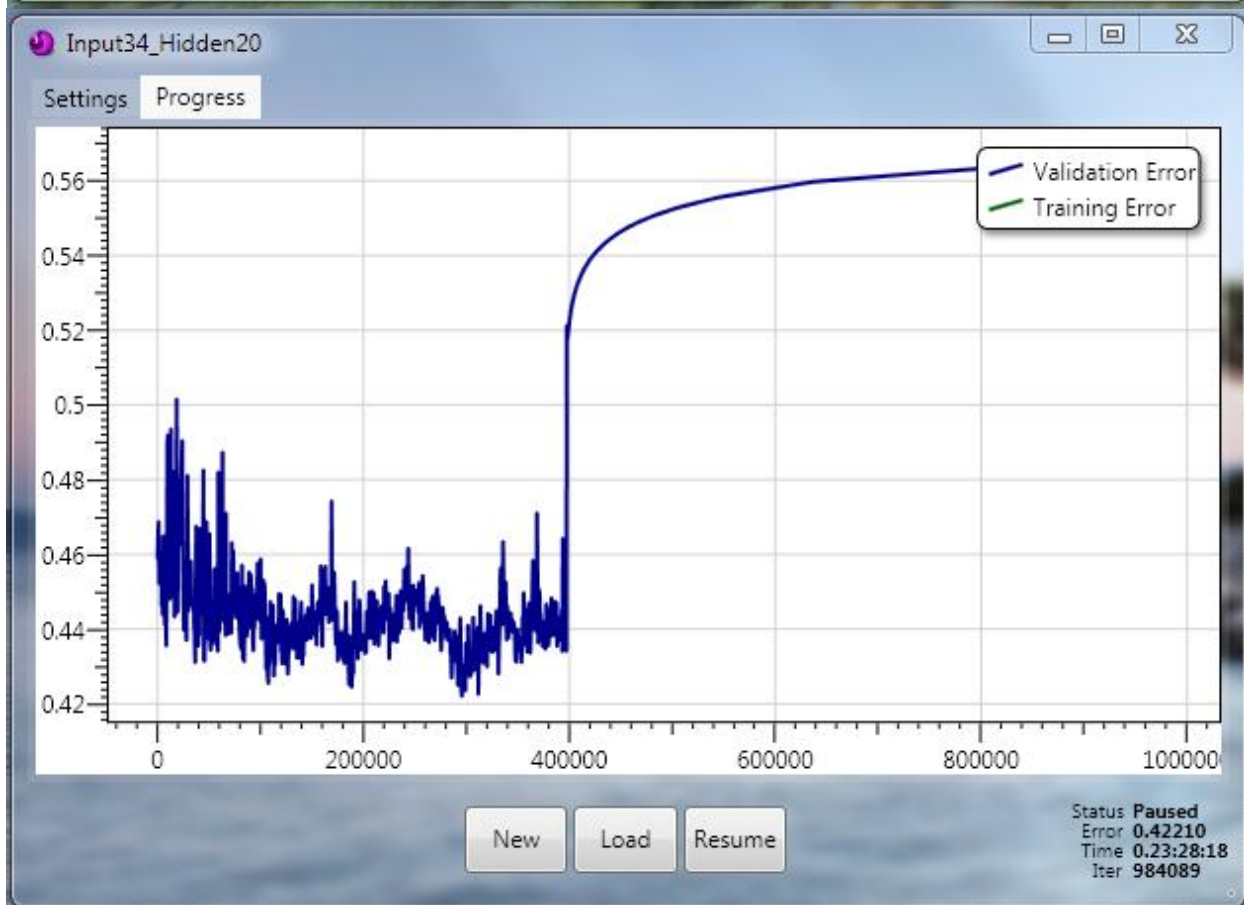
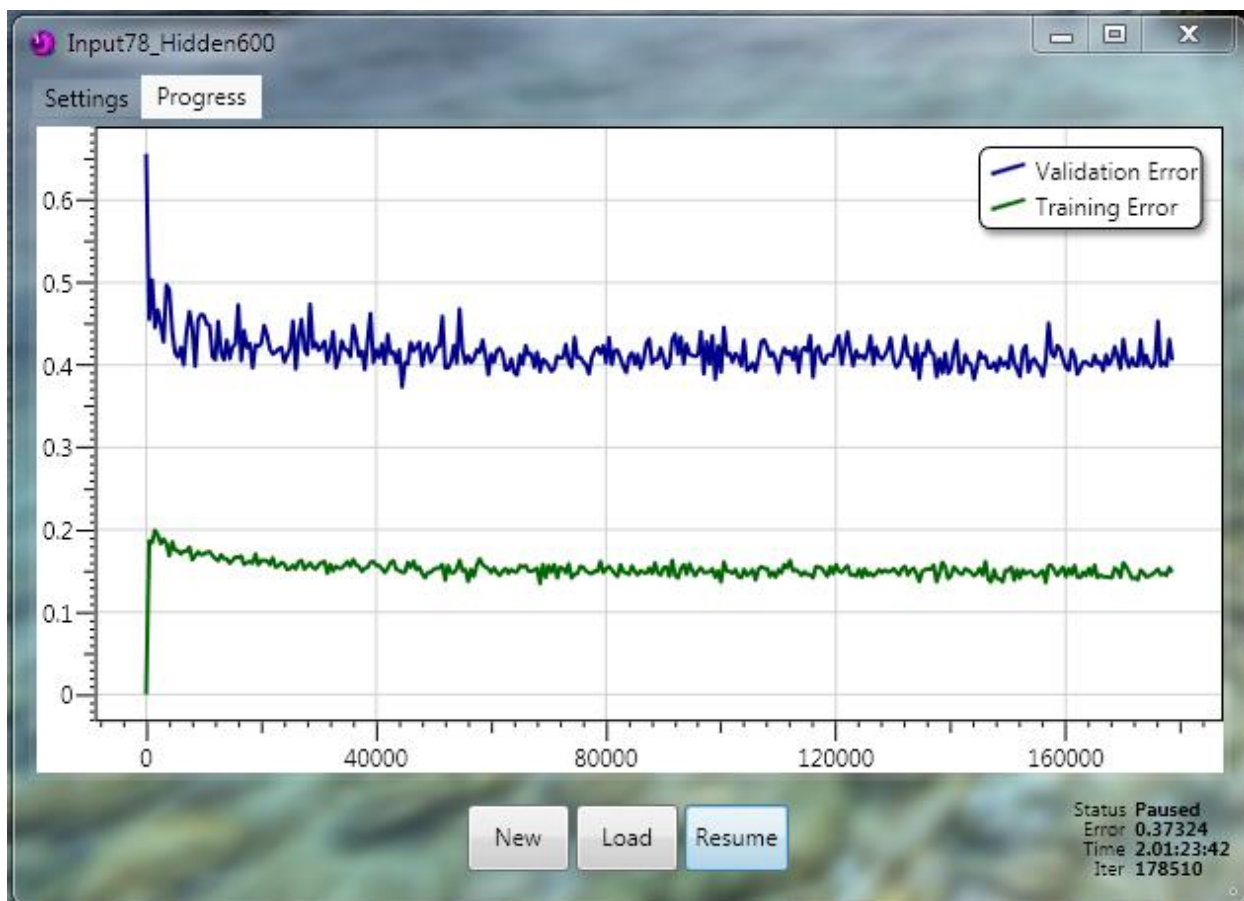
In this appendix we have included screenshots showing the 8-ply database prediction rates for variously sized neural networks. The name of the neural network, given in the upper-left of each image, indicates the number of input nodes and the number and layout of hidden-layer nodes, respectively. *Input42* is the simplistic board representation, *Input 78* is the expanded input representation using the board state plus additional features, and *Input34* is the features-only representation.

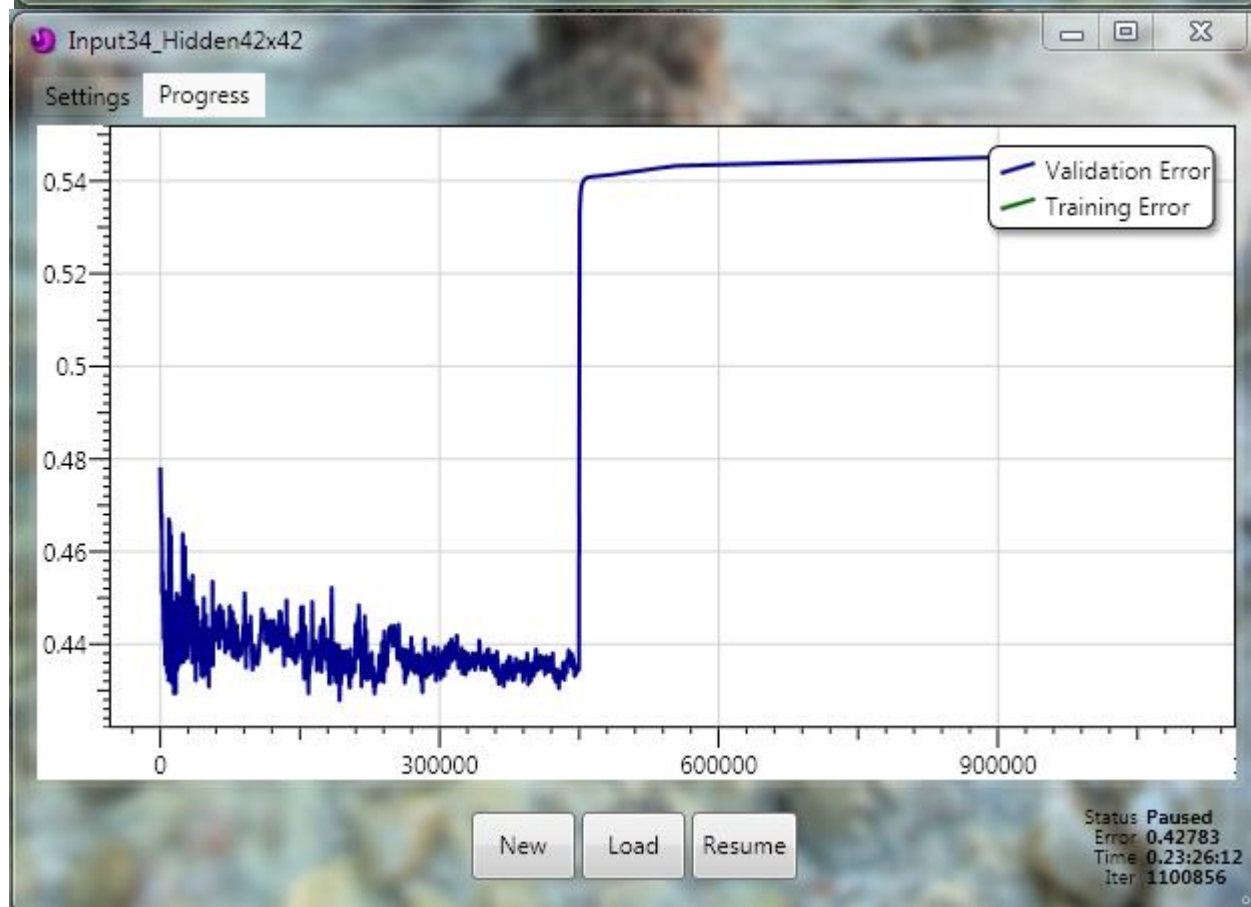
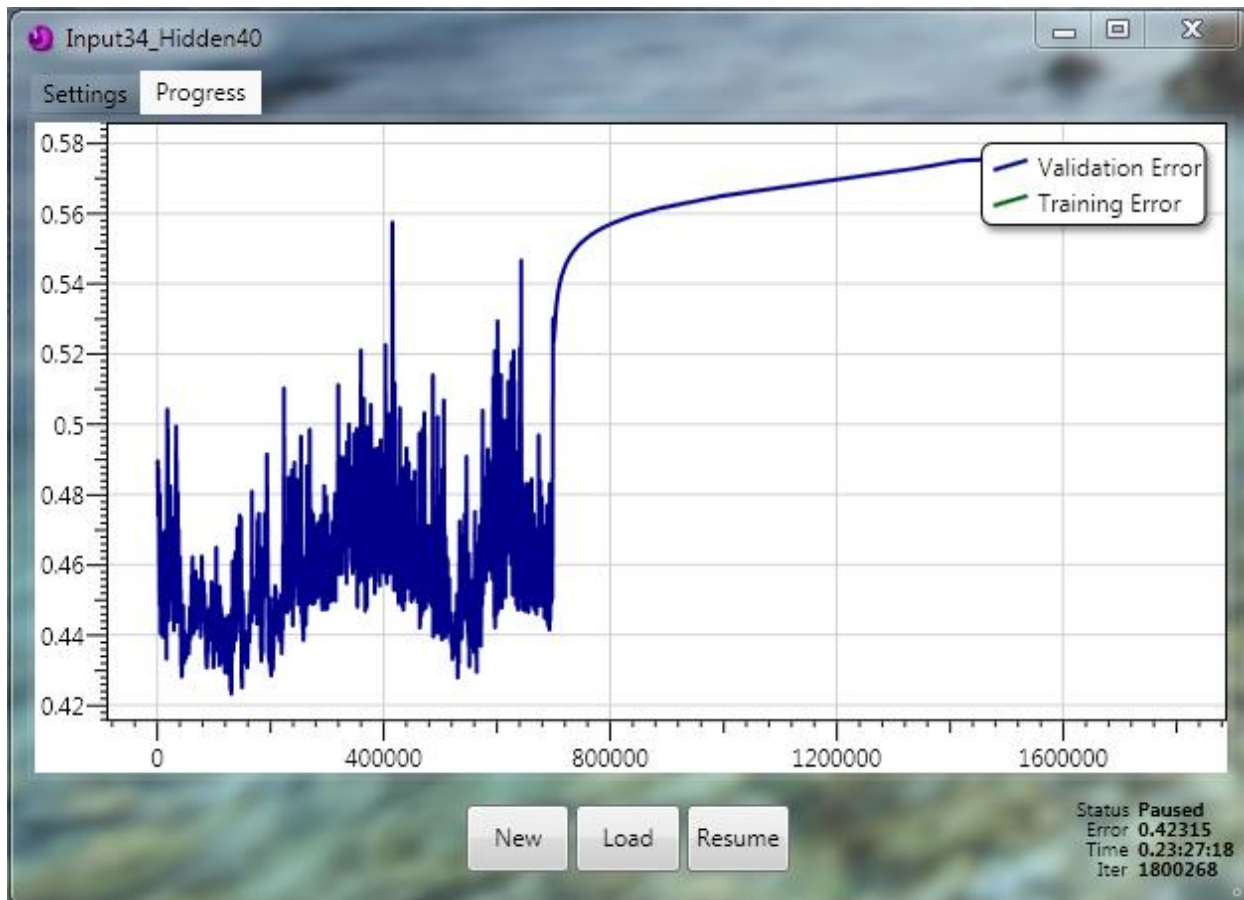


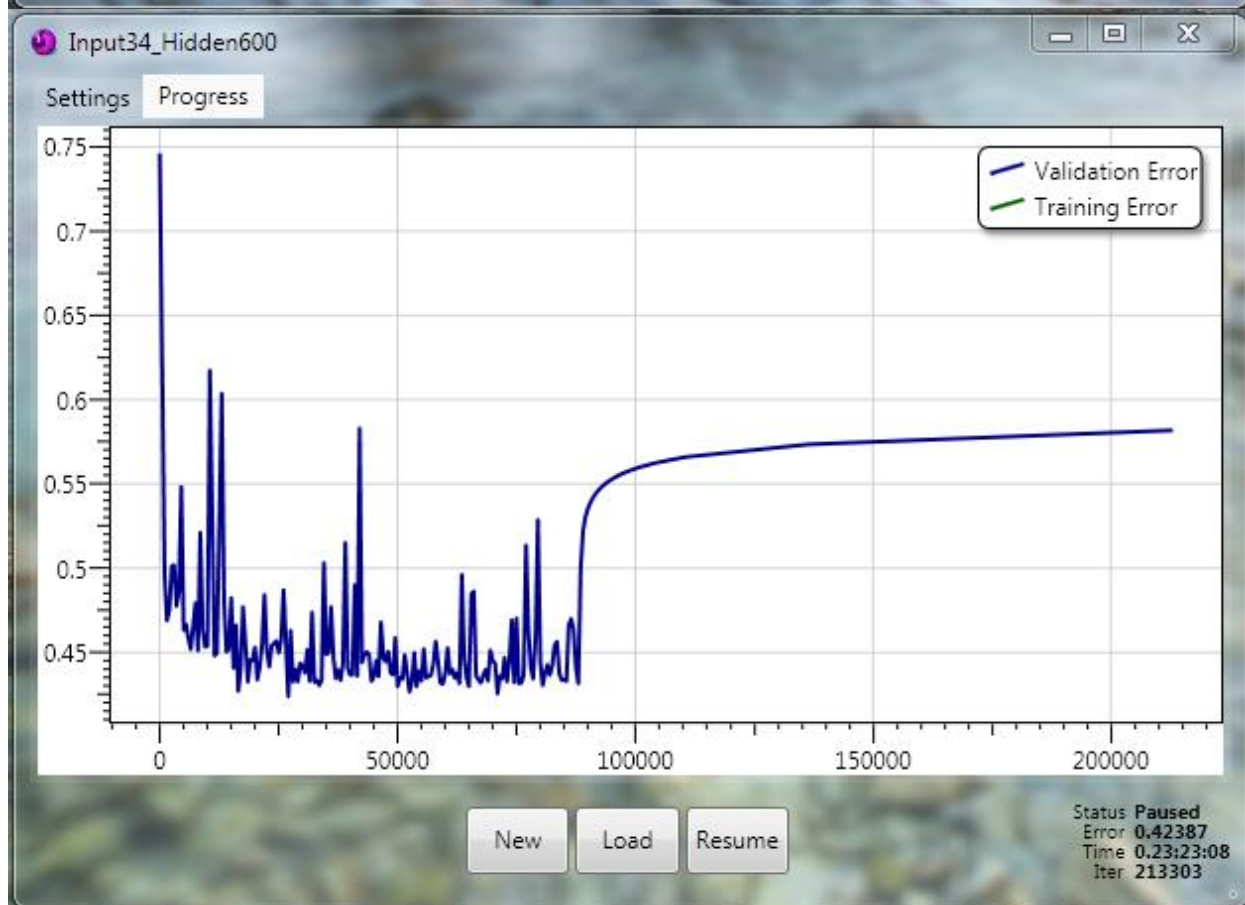
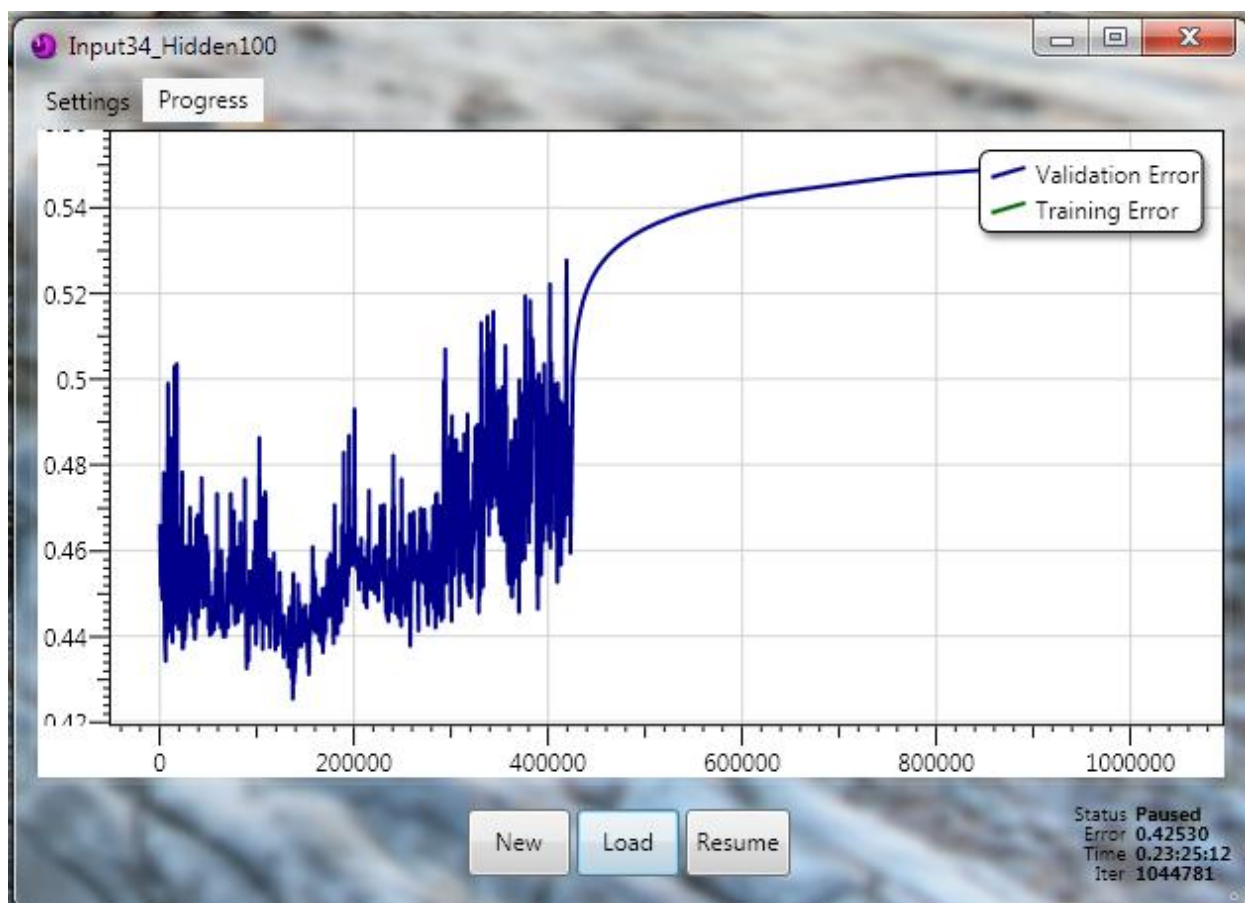












Works Cited

- Allis, V. (1994). *Searching for Solutions in Games and Artificial Intelligence*. Retrieved from Department of Computer Science, University of Limburg: <http://fragrieu.free.fr/SearchingForSolutions.pdf>
- Ghory, I. (2004, May 4). *Reinforcement Learning in Board Games*. Retrieved from University of Bristol: <http://www.cs.bris.ac.uk/Publications/Papers/2000100.pdf>
- Lenz, J. (2003). *Reinforcement Learning and the Temporal Difference Algorithm*. Retrieved from Computing and the College of Engineering, University of Wisconsin-Madison: <http://homepages.cae.wisc.edu/~ece539/project/f03/lenz.pdf>
- Mitchell, T. (1997). *Machine Learning*. McGraw Hill.
- Schneider, M., Luis, J., & Rosa, G. (2002). *Neural Connect 4 - A Connectionist Approach to the Game*. Retrieved from CiteSeerX: <http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.12.8621>
- Tromp, J. (2005, November). *John's Connect Four Playground*. Retrieved from Centrum Wiskunde and Informatica: <http://homepages.cwi.nl/~tromp/c4/c4.html>
- UCI. (1995, February 4). *Connect-4 Data Set*. Retrieved from UCI Machine Learning Repository: <http://archive.ics.uci.edu/ml/datasets/Connect-4>
- Wikipedia. (2010, December 3). *Connect Four*. Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Connect_Four