Non-linear Algorithms for Reinforcement Learning

Victoria Manfredi

Submitted to the Department of Computer Science
of Smith College in partial fulfillment of the
requirements for the degree of
Bachelor of Arts with Honors

Judy Franklin, Faculty Advisor

Spring 2002

Abstract

We adapted two machine learning algorithms, temporal-difference learning (for prediction) and Sarsa (for action generation) to be nonlinear. We then used them for prediction and selection of notes in a musical sequence. We observed that both the linear and nonlinear temporal-difference algorithms combined with function approximation can learn to predict on a given note whether the future looks good or bad; that is, whether the context of the note is good or bad, given certain constraints. We found that the nonlinear algorithms can learn to predict more difficult constraint problems than can the linear. We determined that using an average trace as input to the network, to allow an alternate history besides the eligibility trace (which is used in calculating the prediction), improved the correctness of the nonlinear prediction. We then looked at the neuroanatomical framework for temporal-difference learning, which involves dopamine neurons. Finally, we found the combination of Sarsa with either a linear or a nonlinear network to be promising for choosing good actions or notes to play.

Contents

List of Figures

List of Tables

Acknowledgements

"This heart within me I can feel, and I judge that it exists. This world I can touch, and I likewise judge that it exists. There ends all my knowledge, and the rest is construction."

-Albert Camus, *The Myth of Sisyphus*

To my family: To my father who taught the value of knowledge. To my dear mother who taught me love and perseverance. To my brother who has always been there for me. Thank you and I love you.

To my professors: You have opened up to me an intellectual world that I never before knew existed and for this you have my utmost gratitude. To Prof. Streinu who introduced me to research and the beauty of computer science. To Prof. Thiébaut who was a superb professor. To Prof. Franklin who first introduced me to machine learning and who remained unfailingly patient and good-natured as I asked her innumerable questions, discovered bugs, and had ridiculous ideas during the course of this thesis. Perhaps most of all this thesis is due to her. Judy, you have been an amazing thesis advisor and it was a pleasure working for you.

To my friends: To my computer science friends and all denizens of M104 and M209: you have been with me through the good times and the bad times; and the finding of bugs and the crashing of programs. It was great to have you there. To Frosso: thank you for your friendship; you have been a true friend.

Chapter 1

Introduction

1.1 Background

Humans are capable of improvising jazz music. One way that humans improvise music is by playing certain notes over certain chords. This is because certain notes sound good with certain chords. For example, playing notes that are in the chord, along with playing the chord, sounds good. Another way that humans improvise jazz is to play a sequence of notes that is known to sound good. We were interested in getting a computer to simulate jazz improvisation on a basic level. In order to do this, the computer program, or agent, needed to learn to follow certain basic rules for jazz improvisation, similar to the ones that humans follow. We looked at several learning algorithms in the area of connectionist machine learning to achieve this end. First we used the temporal-difference reinforcement learning algorithm [Sut88] combined with a neural network to predict whether playing certain notes followed these rules. Then we used the Sarsa learning algorithm [SB98] combined with a neural network to enable the agent to choose which notes to play on the basis of which notes it predicted were best to play. We examined the different capabilities of these learning algorithms within the framework of both the linear and non-linear implementations.

1.2 Connectionist Machine Learning

Connectionist machine learning can be divided into three fields: supervised learning, reinforcement learning, and unsupervised learning [Hin87]. In supervised learning both the example to be learned and the correct output are available to the agent [Bis95], [SB98]. In reinforcement learning, the agent does not have access to the correct output, but does receive a reinforcement value based on the outcome of the example [SB98]. Via this reinforcement, it learns that certain actions are "good" actions because they result in positive reinforcement, while other actions are "bad" actions because they result in negative reinforcement [SB98]. In unsupervised learning, not only does the agent not have access to the correct output for a given example, it does not receive any reinforcement based on the actual outcome of the example [Bis95]. Rather, examples are clustered or categorized based on a similarity measure [Bis95]. Our focus is on supervised learning and reinforcement learning.

1.2.1 Supervised Learning

Our main focus in supervised learning was on adapting neural network-based supervised algorithms for use with reinforcement learning algorithms. Although there is a proof of convergence for temporal-difference learning when combined with linear function approximation, there is no proof of convergence for temporal-difference learning when combined with non-linear function approximation [TR97]. This issue is addressed in Section 1.2.3. A brief background to neural networks is given here.

Linear neural networks. A linear neural network, see Figure 1.1, consists of an input layer and an output layer. There are weighted edges going from the inputs to the output node. To get the output of the network the dot product of the input times the weight vector is calculated. The output is therefore the sum of inputs weighted by different values [Bis95].

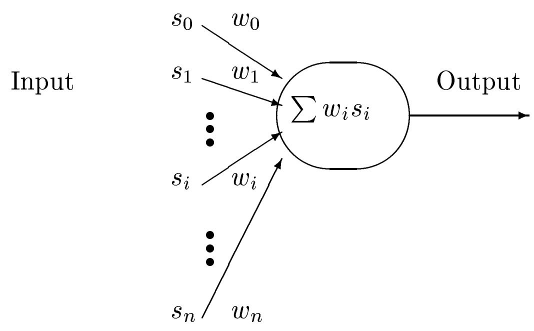

Figure 1.1: Linear neural network.

Non-linear neural networks. A non-linear neural network, see Figure 1.2, has an input layer and an output layer as well, but also a hidden layer. The hidden layer is an extra layer of computation between the input and output layers. Each hidden unit's output is computed by finding the linear sum of its inputs and the weights on the edges coming into it. This hidden unit sum is then passed through a non-linear function, f (giving the network its nonlinearity). The output unit then takes the linear sum of the hidden units' outputs and the weights on the edges coming into it. The output unit can now pass its output through either a nonlinear function like the hidden units or just output what it has (linear combination of non-linear (hidden unit) functions). We choose to use a linear output unit. In supervised learning, given the input vector, an error is formed between the correct output for that vector and the network's output. The weight values are adjusted using this error [Bis95].

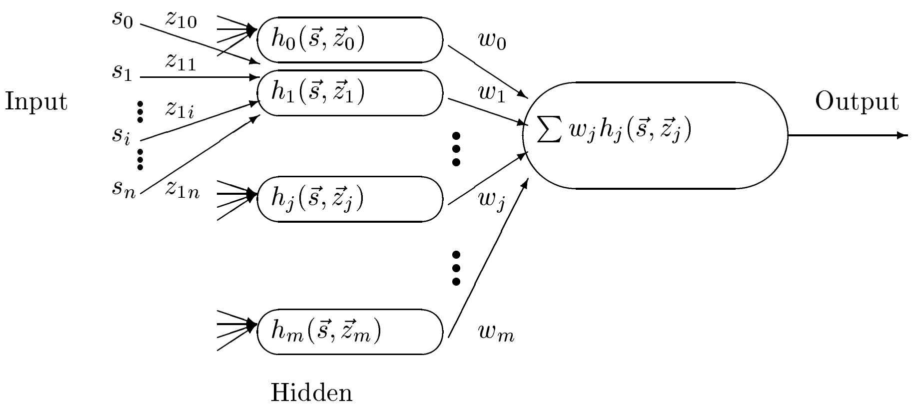

Figure 1.2: Non-linear neural network.

A linear neural network is one that approximates a linear function while a non-linear neural network approximates a non-linear function. A linear function is something like $y = ax + b$, and a non-linear function is something like $y = x^2$. Intuitively we can see that a linear function is pretty simple, and probably will not be able to learn as much as a non-linear function which is more complicated. This turns out to be true [Bis95], [HDB96].

A linear network will only be able to learn the OR function but not the XOR function, see Figure 1.3. This is because a linear function is linearly separable whereas a non-linear function is linearly inseparable: only one line is needed to divide the outputs for the linearly separable OR function but two lines are needed for the linearly inseparable XOR function [Bis95], [HDB96].

The actual function being approximated is unknown but it is represented in the neural network via the weights. The δ rule and backpropagation are two weight update algorithms for neural networks [Bis95], [HDB96].

See Section 2.3 for an explantion of function approximation. See Section 2.4.4 for an explanation of the δ rule and backpropagation.

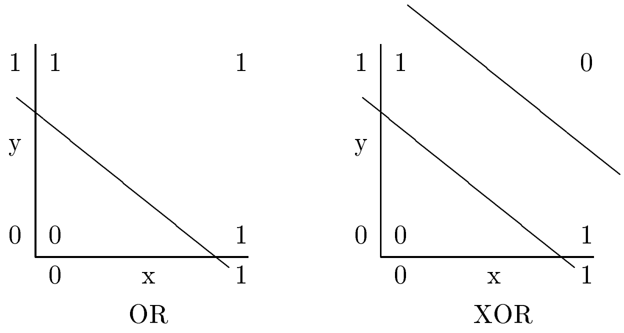

Figure 1.3: Linear versus non-linear separability. Source: [Uni]

1.2.2 Reinforcement Learning

In reinforcement learning, the agent is learning while doing: the agent takes an action in a state and receives some reinforcement for that action. Based on whether the reinforcement was positive or negative, the agent will learn whether it would be beneficial to take that action again when in the same state as before. What the agent must ultimately learn is how good each of its available actions will be in a given state; that is, what long term reinforcement it will eventually obtain as a result of taking each action. Thus, what the agent really needs to learn is the best action; in Q-learning and Sarsa for example, the agent "rates" each action while in other kinds of reinforcement learning, the agent just produces the best action [HH96], [KLM96], [Mit97], [SB98].

Our focus is on the Sarsa algorithm in which a value function is used to find out the value of being in some state. This value is based on the sum of two things: first, the immediate reward that the agent gets for being in the state, and second, an estimate of the future reward that the agent will get after being in this state and taking some action, arriving in another state, taking another action, and so on. A reward is therefore a reinforcement for a short-term event while a value is a reinforcement for a long-term event. This long-term event can be thought of as comprised of multiple short-term events. The action that an agent will take in a given state is dependent on the policy it uses. A policy usually gives the probability of taking some action when in a given state, although a deterministic policy is possible. The value of a state is therefore dependent on the policy being used since the policy stipulates the action that will be chosen in the state, and that consequently determines the subsequent reward [HH96], [KLM96], [Mit97], [SB98].

For reinforcement learning, there are two general types of algorithms that determine which action is best: policy iteration and value iteration. The basis behind value iteration is that if an agent visits enough different states often enough, it will be able to make a good estimate of the optimal value function. It can then use the value function to determine a policy. This policy, being based on the optimal value function (actually, a function close to the optimal value function) will indicate the best action to take in a given state to obtain the optimal value [HH96], [KLM96], [Mit97], [SB98].

Even before the value functions are absolutely correct for every state, a useful policy can still be achieved from the value functions. Policy iteration attempts to take advantage of this. In policy iteration, the agent uses an initally random policy, obtains some utility from the partially correct value function, and updates its policy based on that utility. It does this repeatedly until an acceptable policy is achieved, while in parallel repeatedly improving its value function estimate [HH96], [KLM96], [Mit97], [SB98].

1.2.3 Unification

Neural networks and reinforcement learning algorithms are each capable of learning individually. They are used in combination to attempt to obtain reinforcement learning based on non-linear function approximation. As mentioned, the non-linearity of a non-linear neural network is appealing due to the more complicated functions it can learn, whereas reinforcement learning techniques are appealing due to their ability to learn without needing to know the correct output. These two techniques are combined by using the reinforcement learning error to update the weights in the neural network. In particular, the ability to use a function rather than a look-up table in order to obtain some output allows, for example, larger state spaces [TS93].

There is however no theorem that proves that the weights in a non-linear network must converge when using reinforcement learning [BM95], [TS93], [TR97]. This lack of convergence is in part due to two things: first the erroneous estimation from noise as a result of the function approximator and second, the discounting of future reward [TS93]. This does not mean that when neural networks and reinforcemement learning algorithms are combined that the weights never converge, only that they are not guaranteed to converge [TS93]. Hence our justification for the non-linear experiments in this thesis. Our linear experiments are justified since convergence has been shown for linear function approximation combined with temporal-difference learning [TR97].

1.3 Definitions

For convenience, some definitions of some common terms used in this thesis are given below along with their context within our experiments.

1. **Neural network**. A linear neural network consists of an input layer and an output layer. A non-linear neural network consists of an input layer, hidden layer, and output layer, and also makes use of a non-linear activation function with the hidden units. There are then various types of neural networks that use a linear or a non-linear network as their bases [HDB96].

2. **State** at time t, s_t. The current situation [SB98]. This is the input $\vec{s}$ to the neural network. For us, s_t can be one of the 12 possible notes in the chromatic scale, or an averaged representation of some combination of these notes over time (see the definition for average trace). The state may also include a chord, where a chord is four notes played simultaneously.

3. **Action** at time t, a_t. What the agent chooses to do in a state at time t [SB98]. In our experiments, the agent must choose the next note to play. Hence a_t can also be one of the 12 notes in the chromatic scale.

4. **Reward**. A reward is a reinforcement that the agent receives for being in a state [SB98]. The reward will always be 0 unless it is the last state in the sequence. If it is the last state in the sequence, then if the sequence and chord were good, a reward of 1 is given. If the sequence and chord were bad, then a reward of 0 is given.

5. **Value**. A value refers to how good the current state is plus how good the next state will be after taking some action and so on [SB98].

6. **Value function**, $V(s_t)$. A value function takes a state and returns an estimated value for that state. Temporal-difference learning approximates this function with value iteration thereby allowing it to predict how good each state is expected to be [SB98].

7. **Q-value function**, $Q(s_t, a_t)$. A Q-value function takes a state and action pair and returns the estimated value for taking action a_t in state s_t [SB98].

8. **Value iteration**. Value iteration is a way of determining the value function by visiting each of the states many times and computing the reward and subsequent value to estimate the value for each state [SB98].

9. **Policy**. A policy is a set of rules telling the agent which action to choose in a state in order to maximize the reward and subsequent value that it receives [SB98].

10. **Eligibility trace**, $e(s_t)$. The eligibility trace is a history that keeps track of the most frequently and recently visited states. This is important for updating the weights. If the agent just got a reward of 1, it knows that something good just happened. The most frequently and recently visted states probably had something to do with that good thing that happened, so it makes sense to update the weights for those states more by way of the eligibility trace. Note that the eligibility trace is decayed by λ, $1 \leq \lambda \geq 0$. This is so the history is not permanent: it eventually fades away. Thus, if the current state is s_t, then $e(s_t)$ increases by one and all $e(s_i)$ are deccayed by λ. [SB98].

11. **Average trace**. The average trace is another way to give the neural network some history of the inputs it receives; this can be thought of as a self-recurrent unit. [Jor86] originated this idea and called the units context units. Rather than giving the agent simply a note in a sequence as input, it is given an averaged representation of the notes seen so far in the sequence, thereby giving the agent more information.

12. **Error change**, δ. The error change is used to modify the weights in the neural network so as to minimize the error [Bis95]

1.4 Outline

We looked at the reinforcement learning algorithm Sarsa. Sarsa uses value iteration to determine the value function and the subsequent policy that should be followed [SB98]. In order to do this, Sarsa uses temporal-difference learning [Sut88] to predict how good or bad the future looks from the current state [SB98]. Sarsa then uses this prediction to determine what actions should be played (the policy) to make the future look good rather than bad [SB98]. We began by first implementing the temporal-difference learning algorithm and then running experiments to see how well it could learn to critique different states. We then implemented Sarsa to see how well it was able to choose good actions to play. Chapter 2 explains temporal-difference learning generally, presents the algorithms actually implemented, and describes the data structures used. Chapter 3 contains the results for the temporal-difference learning experiments, Chapter 4 contains an explanation of Sarsa, Chapter 5 contains the results for the Sarsa experiments, and Chapter 6 contains an explanation of the neurobiological view of temporal-difference learning that involves dopamine neurons, as well as a mention of critiques that have been made concerning this model. Finally Chapter 7 summarizes our conclusions and presents ideas for future work.

Chapter 2

Temporal-Difference Learning

2.1 Introduction

The goal of temporal-difference (TD) learning is to approximate the value function. TD does this by updating the value function based on the difference between value predictions for successive states. This updating can be done in an incremental fashion (value iteration) as we shall see. An underlying assumption of TD is that the value function of the current state should be close to the value function of the next state. For instance, if we wish to learn to predict an event on Sunday and today is Monday, we can do it either by using as the error the difference between Monday's prediction and Sunday's prediction or by using as the error the difference between Monday's prediction and Tuesday's prediction, and then the difference between Tuesday's and Wednesday's prediction and so on, finally using as the error the difference between Saturday's prediction and Sunday's prediction (example from [SB98]). The reason behind this assumption is that, presumably, the next (or later) state has more information than the previous (or earlier) state, and therefore has a more accurate value function, and thus more correct information is propagated back to an earlier state. Thus, the error is the difference between the predicted value for the current state and the estimated value for the future from that state. If the next state has a high predicted value, then the preceding states will be updated to have higher values, and if the next state has a low predicted value, then the preceding states will be updated to have lower values. This propagates information from the future backwards [SB98], [Sut88].

We now present the tabular, linear and non-linear algorithms and the actual network model and data structures used.

2.2 Tabular TD(λ)

The general TD learning algorithm is tabular TD(λ). In tabular TD(λ), the value function works by table-lookup: a state is an index in the table of values. These values serve as predictions. A high value for a given state indicate that the future looks promising from that state while a low value indicates that the future looks calamitous. The λ indicates the use of an eligibility trace to change the predictions of more recently visited states by a greater amount than those states visited less recently. The eligibility trace reflects the "eligibility" of a state for updating, where the more frequently and recently the state is visited, the higher is its eligibility trace. This is done by incrementing by one the eligibility trace for a state whenever the state is visited, but always decaying the eligibility by λ at each time step, t, where $0 \leq \lambda \leq 1$. Therefore with time, if a state is not visited often enough then the eligibility for the state will be diminished. Since the eligibility trace is used in updating the prediction, a smaller eligibility trace will change the prediction $V(s_t)$ for a state s at time t by a smaller amount and a larger eligibility trace will change the prediction by a larger amount. The eligibility trace for a state at time t is given by $e(s_t)$ [SB98], [Sut88].

Besides λ, the TD(λ) algorithm uses two parameters to update the value function for a state, $V(s_t)$: a learning rate, α, and a discount rate, γ. The learning rate is used to determine how quickly the learning will be done; it does this by fixing what proportion of the error (to be described) should be used to update the value function. If too large a proportion is used, it is possible to "overshoot" and settle on a value for $V(s_t)$ that is far away from the optimal. If too small a proportion is used, overshooting will not happen, but it may take a long time to settle on the optimal value. The eligibility trace is also involved in determining by how much to change the prediction. For example, with error δ, where $\delta = V_{optimal}(s_t) - V_{estimate}(s_t)$, the value function for a state s will be changed by $\alpha\delta e(s_t)$ [SB98], [Sut88]:

$$V(s_t) = V(s_t) + \alpha\delta e(s_t)$$

Of course, $V_{optimal}(s_t)$ is unknown, but is estimated as $V_{optimal}(s_t) = r(s_t) + \gamma V(s_{t+1})$ where $r(s_t)$ is the reward for the current state, s_t, and s_{t+1} is the next state. We distinguish between reward and value. For our purposes, if the preceding states were sufficiently good, then the reward, $r(s_t)$ given in the final state will be 1, otherwise it will be 0. The value is the sum of many rewards; e.g., the prediction value, $V(s_t)$ is the sum of the predicted future rewards. The discount rate γ is used to discount future rewards, thereby allowing the prediction to converge to a finite sum. There are other ways to estimate $V_{optimal}(s_t)$. For further analysis see [SB98]. Note with this estimate $\delta = r(s_t) + \gamma V(s_{t+1}) - V(s_t)$ and the error is a function of successive differences in state values [SB98], [Sut88].

An outline of the TD(λ) algorithm (from [SB98]) is the following:

1. Set eligibility traces for all states to 0.

2. For each state,

   a. First, use the policy on the state to get the action (setting the policy is not part of TD).
   b. Then take the action, get the reward and reinforcement and determine the next state.

3. Find the error with the successive difference equation.

$$\delta = r(s_t) + \gamma V(s_{t+1}) - V(s_t)$$

4. Go through and update each state using the δ and eligibility trace. Notice the split of the TD(λ) equation into parts.

a. Update the value function of each state with the learning rate, error δ, and the eligibility trace.

$$V(s_i) = V(s_i) + \alpha\delta e(s_i)$$

b. Now update and decay the eligibility trace by λ. First update the eligibility trace for the current state.

$$e(s_t) = e(s_t) + 1$$

Then decay the eligibility traces for all states.

$$e(s_i) = \lambda e(s_i)$$

Note however, that if s_t is the current state then what $e(s_t)$ is really being set to is the following,

$$e(s_t) = \lambda[e(s_t) + 1]$$

This is because we previously incremented $e(s_t)$ by 1 when we visited state s at time t. Note as well that these eligibility traces must be reset to 0 each time the goal (final state) is reached.

Traditionally $e(s_i)$ is set to $\gamma\lambda e(s_i)$, see [SB98]. We choose to exclude the γ term, however, on the grounds that it will unnecessarily decay $e(s_i)$ further.

Thus it can be seen that the more frequently and recently a state is visited, the higher its eligibility trace will be. The eligibility trace ensures that those states that were visited more often are more affected by the update. Each time a state is visited its eligibility trace is incremented positively, otherwise it decays over time. Since the eligibility trace is used in the calculation to update the values of the states, then the larger the eligibility trace the more the corresponding values of the states are affected. The parameter λ is used to decay the eligibility trace.

5. Store the next state s_{t+1} as the current state s_t.

6. Continue (return to step 2) until goal state is reached.

Essentially the TD(λ) algorithm operates as follows. The change in current prediction, $V(s_t)$, is given by the difference between the current reinforcement and the prediction for the following states, and the prediction of current and discounted future reward for the current state. This is $(r + \gamma V(s_{t+1})) - V(s_t)$ in step 3. This is where "temporal-difference" comes into play. We are taking the difference between two temporally separated predictions. The discount rate γ is being used to prevent an infinite sum since the sum for $V(s_{t+1})$ would grow infinitely for any infinite sequence, and could grow very large for many finite sequences. Hence some way is needed to decrease these sums while still maintaining the information they represent; the solution is discounting. Furthermore it makes sense to discount future rewards, assuming cause and effect relationships are proximal in time. Therefore we are going to change the prediction for a state by the difference between the reward it receives for the current state, $r(s_t)$ plus the discounted future reward, $\gamma V(s_{t+1})$, and the current estimate of reward, $V(s_t)$. The prediction for a state, $V(s_t)$, is then changed by $\alpha\delta e(s_t)$. The eligibility trace for the state, given by $e(s_t)$ is saying, if we have visited this state frequently and recently, i.e., the eligibility trace is large, then let us update the prediction more. δ is most relevant for the most frequently and recently visited states, so let us update those states by a larger proportion of the error [SB98], [Sut88].

Since the error for the current state is the difference between predictions for two temporally separated states, then if the future state has a high prediction, the prediction for the current state will be increased, and if the future state has a low prediction then the prediction for the current state will be decreased.

There is an assumption here that allows this to happen: that we will go from the current state to the future state at some point again [SB98], [Sut88].

2.2.1 Tabular TD(0) and TD(1)

We can derive TD(0) and TD(1) from the TD(λ) algorithm. TD(0) is TD(λ) with $\lambda = 0$. Recall step 4a from the TD(λ) algorithm in which the eligibility traces are used to update $V(s_i)$, and step 4b in which the eligibility trace for the current state is updated $e(s_t) = e(s_t) + 1$, and in which all eligibility traces are decayed $e(s_i) = \lambda e(s_i)$. We first rearrange the order of those steps for TD(0).

4a. First update and decay the eligibility trace for the current state by λ.

$$e(s_t) = e(s_t) + 1$$

4b. Update the value function of each state with the learning rate, error δ, and the eligibility trace.

$$V(s_i) = V(s_i) + \alpha\delta e(s_i)$$

4c. Now decay all of the eligibility traces by λ.

$$e(s_i) = \lambda e(s_i)$$

Now if we set λ to zero, then in step 4a the only eligibility trace that will be one (or even non-zero) will be for the current state since after the eligibilities are used in step 4b all of the eligibilities for all of the states will be reset to zero in step 4c. This means then that there is no need to loop through all of the states updating the value function for each step with $V(s_i) = V(s_i) + \alpha\delta e(s_i)$, because unless $e(s_i)$ is nonzero, the value function for the state will not change. In fact, we can get rid of our eligibility trace altogether now as it no longer serves any purpose: it will either be 0 or 1. The value function for a state will not change when the eligibility trace is zero. The value function will only change now for the current state when the eligibility trace is updated to 1. Since there is no need to always be multiplying by one since it changes nothing, then the value function update rule for the TD(0) algorithm is simply given for the current state by [SB98], [Sut88],

$$V(s_t) = V(s_t) + \alpha\delta$$

We can simplify the TD(0) algorithm further by substituting the error $\delta = r(s_t) + \gamma V(s_{t+1}) - V(s_t)$ directly into the value function, rather than calculating it separately. Hence by substituting for δ we obtain [SB98], [Sut88],

$$V(s_t) = V(s_t) + \alpha[r(s_t) + \gamma V(s_{t+1}) - V(s_t)]$$

This then is the TD(0) algorithm and we need only update the current state using it. Note that since $r(s_t)$ is non-zero only on the last state then only the last state incorporates the final $r(s_t)$ directly into its value function [SB98], [Sut88].

TD(1) conversely is TD(λ) with $\lambda = 1$. The TD(1) algorithm can then be derived. Using $\lambda = 1$ gives $e(s_i) = e(s_i)$ for the eligibility trace. Now the eligibility traces for all states will not be discounted at all instead of discounted by λ [SB98], [Sut88].

2.3 Linear TD(λ)

Rather than using a tabular representation for states which can lead to combinatorial explosion, an alternative is to have a function that can approximate what the prediction should be for a given state. Using "function approximation" allows predictions to be made more compactly for large numbers of states, enables predictions to be made on states with continuous values (non-discrete states) and even permits predictions to be made for states not previously seen. Some method is needed to learn to approximate this function and we have chosen to use gradient descent. We start first with linear function approximation calling it linear TD(λ) [SB98], [Sut88]. The TD(λ) algorithm from [SB98] is:

1 $\vec{w} = random$, $\vec{e} = 0$
2 For each episode
3     $s_t = s_0$
4   For each state in episode

$\quad$ Get a

$\quad$ Get r and s_{t+1} using a

$\quad$ $\delta = r + \gamma V(s_{t+1}) - V(s_t)$

$\quad$ $\vec{e} = \lambda \vec{e} + \nabla_{\vec{w}} V(s_t)$

$\quad$ $\vec{w} = \vec{w} + \alpha \delta \vec{e}$

$\quad$ $s_t = s_{t+1}$

Continue until goal state is reached

This is the first algorithm that we implemented as linear TD(λ). Linear TD(λ) is capable of approximating (or learning) linear functions. A linear function is a function of the form $y = ax + b$. More relevantly a linear function is linearly separable: that is, given a set of points split into two classes (e.g., output of 1 or 0) plotted in the x,y plane then a line can be used to separate the two classes of points [Bis95].

For example, in the context of our experiments a linear function could learn that for any chord, playing note 2 is good. Thus we have two classes, good and bad, and the point is in good if it is note 2. However a linear function could not learn that playing note 2 is good only if chord 1 is played and that playing note 2 with any other chord is bad since this function is linearly inseparable. In order to learn a linearly inseparable function a non-linear function approximator is needed.

The linear network is a single node receiving the various inputs and outputting a linear combination of those inputs, see Figure 2.1. The actual updating or learning in the network occurs through the weight updating which is done using the linear TD(λ) algorithm. Thus δ is the error that gets distributed over the weights. Those weights that were most responsible for the error are the weights that are most modified by this error distribution [Bis95].

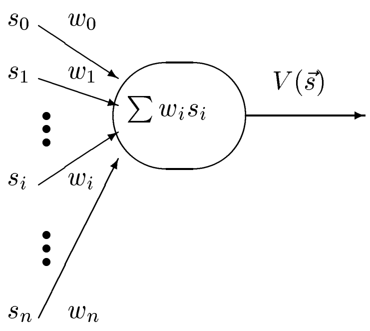

Figure 2.1: General linear TD(λ) network with $\vec{s}$ as input.

While in tabular TD(λ) the value function works by table-lookup, in linear TD(λ) the value function is given by a linear function that takes a state vector s and a weight vector w, e.g, $V(\vec{s}, \vec{w})$. We can treat the algorithm as a supervised learning algorithm and incrementally approximate the value function by using gradient descent to determine the weights to use that best approximate V. One way to minimize this error is to minimize the squared error. Hence we are minimizing the following, where δ represents the error and $\frac{1}{2}$ is used for convenience when taking the partial derivative [SB98].

$$E = \frac{1}{2}(\delta)^2$$

To minimize we take the derivative (partial derivative in this case because there is more than one variable) of the mean squared error with respect to each weight and make the ith weight change Δw_i proportional to $-\frac{\partial E}{\partial w_i}$ (for convenience we let $\Delta w = w_{t+1} - w_t$ be the weight change at time t). In doing this we will travel along the error surface in the direction that minimizes the error.

Using Figure 2.2 for illustration, consider the simple case of one weight. The error squared when plotted gives us a parabola. First note that w_{t+1} will normally be closer to the minimum point of the parabola than w_t, unless there is overshooting. With E as the y-axis, and w as the x-axis, if $\frac{\partial E}{\partial w} > 0$ then $\Delta w < 0$

because $\frac{\partial E}{\partial w}$ gives a tangent to the parabola, and this tangent has a positive slope (increasing). So w_{t+1} is less than w_t ($\Delta w < 0$). If $\frac{\partial E}{\partial w} < 0$ then $\Delta w > 0$ because the tangent to the parabola given by $\frac{\partial E}{\partial w}$ now has a negative slope (decreasing) and w_{t+1} is greater than w_t because Δw is positive (increasing).

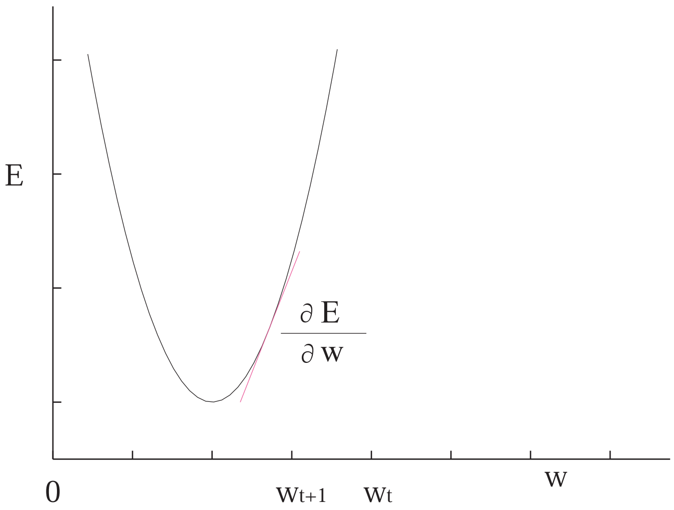

Figure 2.2: Gradient descent visualization.

We thus have an inverse relationship.

$$\Delta w \quad \propto \quad -\frac{\partial E}{\partial w}$$

Hence when we add Δw to the weight we adjust the weight so as to reduce E and therefore δ. For our purposes δ would ideally be the difference between the actual prediction $V^{\pi}(s_t)$ and the estimated prediction $V(s_t)$ for some state s at time t. Thus $\delta = V^{\pi}(s_t) - V(s_t)$ and the squared error we are minimizing becomes $E = \frac{1}{2}(V^{\pi}(s_t) - V(s_t))^2$. Therefore our weight change is given by,

$$\begin{aligned} w_{t+1} &= w_t - \frac{1}{2}\alpha\nabla_w\delta^2 \\ &= w_t - \frac{1}{2}\alpha\nabla_w[V^{\pi}(s_t) - V(s_t)]^2 \end{aligned}$$

$$
\begin{aligned}
&= w_t - \frac{2}{2}\alpha(V^\pi(s_t) - V(s_t))\nabla_w[V^\pi(s_t) - V(s_t)] \\
&= w_t - \alpha(V^\pi(s_t) - V(s_t))\nabla_w[V^\pi(s_t) - V(s_t)]
\end{aligned}
$$

The learning rate α is again used to determine how quickly learning should happen. Since $V^\pi(s_t)$ is independent of the weights we are estimating that its gradient is zero. Hence we get,

$$
\begin{aligned}
w_{t+1} &= w_t - \alpha(V^\pi(s_t) - V(s))\nabla_w[0 - V(s_t)] \\
&= w_t + \alpha(V^\pi(s_t) - V(s))\nabla_w V(s_t)
\end{aligned}
$$

We have no way of knowing the actual prediction $V^\pi(s_t)$, but there are several ways to estimate it (see [SB98]). We have chosen to estimate $V^\pi(s_t)$ as equivalent to $r + \gamma V(s_{t+1})$ where r is the current reward for being in state s_t and $\gamma V(s_{t+1})$ is the estimated discounted future reward for being in the next state s_{t+1}. Our estimation comes from [SB98], p.134, and is based on dynamic programming. Our final equation for changing the weights is therefore given by,

$$
w_{t+1} = w_t + \alpha(r + \gamma V(s_{t+1}) - V(s_t))\nabla_{w_t} V(s_t)
$$

Recall Figure 2.1. From this figure it can be seen that $V(s_t)$ is a linear function, $V(s_t) = \sum_{i=1}^{n} w_i s_i$ where s_i is the ith element of state vector s at time t and w_i is the weight on the edge connecting s_i to the output node. This means that line 8 of the linear TD(λ) algorithm becomes,

$$
\begin{aligned}
\vec{e} &= \lambda\vec{e} + \nabla_{\vec{w}} V(s_t) \\
=> e_i &= \lambda e_i + \frac{\partial V(s_t)}{\partial w_i} \\
&= \lambda e_i + \frac{\partial[\sum_{i=1}^{n} w_i s_i]}{\partial w_i} \\
&= \lambda e_i + \frac{\partial[w_0 s_0 + ... + w_i s_i + ... + w_n s_n]}{\partial w_i} \\
&= \gamma\lambda e_i + s_i
\end{aligned}
$$

Given the weight change calculated by gradient descent methods,

$$
\Delta w = \alpha(r + \gamma V(s_{t+1}) - V(s_t))\nabla_w V(s_t)
$$

$$= \alpha\delta\nabla_w V(s_t)$$

This then gives $\Delta w_i = \alpha\delta s_i$ and steps 8 and 9 become,

Given by Sutton and Barto [SB98]	Calculated Using Gradient-Descent
8 $\vec{e} = \gamma\lambda\vec{e} + s_i$	8 $\vec{e} = \gamma\lambda\vec{e} + s_i$
9 $\vec{w} = \vec{w} + \alpha\delta\vec{e}$	9 $\vec{w} = \vec{w} + \alpha\delta\nabla_w V(s_t)$

In the case of $\lambda = 0$ then $\Delta w_i = \alpha\delta s_i$ which is the same for both Sutton and Barto [SB98] and for the gradient-descent result. However if $\lambda \neq 0$ this does not hold true. Hence, it seems that there are some steps missing between 8 and 9. For comparison then we choose to make $V(s_t)$ a linear function of $e(s_t)$ where $V(s_t) = \sum_{i=1}^{n} w_i e(s_i)$ as in Figure 2.3 (while also maintaining step 8 from before) and $\Delta w = w_{t+1} - w_t$. See Sections 2.4.2, 3.2.1, and 3.3.1.

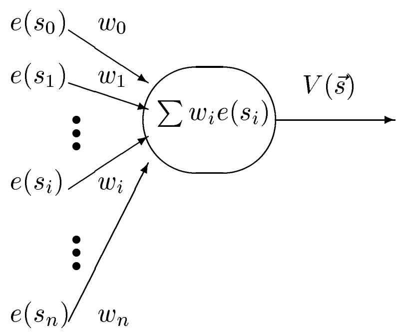

Figure 2.3: Linear TD(λ) network with $e(\vec{s})$ as input.

Note that $V(\vec{s}) = V(s_t)$ for our purposes. s_t represents the state vector at time t. Using $V(s) = \sum_{i=1}^{n} w_i e(s_i)$ we can now calculate the weight change (line 9 of the algorithm) for the weights leading into the output unit. Note that if a constant bias input of 1 is used, then $e(s_i)$ for the bias will also be 1.

$$
\begin{aligned}
\Delta w &= \alpha\delta\nabla_{w_t} V(s) \\
=> \Delta w_i &= \alpha\delta\frac{\partial V(s)}{\partial w_i} \\
\Delta w_i &= \alpha\delta\frac{\partial[\sum_{i=1}^{n} w_i e(s_i)]}{\partial w_i} \\
\Delta w_i &= \alpha\delta\frac{\partial[w_0 e(s_0) + \ldots + w_i e(s_i) + \ldots + w_n e(s_n)]}{\partial w_i} \\
\Delta w_i &= \alpha\delta e(s_i) \\
\Delta w_i &= \alpha(r + \gamma V(s_{i+1}) - V(s_i))e(s_i)
\end{aligned}
$$

This is in fact the same resulting weight change as the original linear TD(λ) algorithm from Sutton and Barto [SB98].

This then is the second algorithm implemented as linear TD(λ). A modified version of Figure 2.2 to correspond to our modified version of the linear TD(λ) algorithm can be found in Figure 2.3.

Therefore as a compromise we implemented the following in all of the algorithms we tried for steps 8 and 9,

Steps 8 and 9 for Linear TD(λ)
8 $\vec{e} = \gamma\lambda\vec{e} + \nabla_w V(s_t)$
9 $\vec{w} = \vec{w} + \alpha\delta\vec{e}$

We then had the option of using either the current state or the eligibility trace for the current state as the input. There are then two possible models we could use: letting $V(s_t)$ be a linear combination of weights and states, or letting $V(s_t)$ be a linear combination of weights and eligibility traces. Therefore we have:

1. Using the [SB98] linear TD(λ) algorithm (Linear Model $\vec{s}_{t\,Note}$).

2. Using a modified version of the [SB98] linear TD(λ) algorithm that uses the current chord and the eligibility trace of the note rather than the state as input to the network (Linear Model $e(\vec{s_t}_{Note})$).

2.4 Non-Linear TD(λ)

As mentioned linear TD(λ) allows at most a linear function to be learned. Introducing non-linearity with a non-linear TD(λ) algorithm allows both linear and non-linear functions to be learned. Essentially, the non-linear network permits a certain input to be considered good in some circumstances and bad in other circumstances. For instance, a non-linear network could learn that note 0 is good with chord 1 but bad with chord 2. Non-linearity is introduced to the network by way of a hidden layer and a non-linear activation function [Bis95].

Adding hidden units to our network for non-linearity, we must now recalculate the weight change for the weights going into the output unit, and now also calculate the weight change for the weights going into the hidden units. There are three ways that these weight updates can be done corresponding to three possible algorithms:

1. Using the general non-linear TD(λ) algorithm from [SB98] presented in Chapter 1 (Non-Linear Model $\vec{s_t}_{Note}$).
2. Using a modified version of the non-linear TD(λ) algorithm from [SB98] that uses the current chord and the eligibility traces for the current note as the inputs to the hidden units (Non-Linear Model $e(\vec{s_t}_{Note})$). (Note that it is also possible to use instead the eligibility traces for the hidden node ouputs rather than the eligibility traces for the notes; see Appendix A for an explanation of how weight updates could be done in this way.)
3. Using a modified version of the non-linear TD(λ) algorithm from [SB98] that uses the current chord and the average traces for the current note as the inputs to the hidden units (Non-Linear Model $a(\vec{s_t}_{Note})$).

2.4.1 Non-Linear Model $\vec{s}_{tNote}$

We first present here for reference the non-linear TD(λ) algorithm from [SB98].

1 $\vec{w} = random$, $\vec{e} = 0$

2 For each episode

3 $\quad s_t = s_0$

4 $\quad$ For each state in episode

5 $\quad\quad$ Get a

6 $\quad\quad$ Get r and s_{t+1} using a

7 $\quad\quad$ $\delta = r + \gamma V(s_{t+1}) - V(s_t)$

8a $\quad\quad$ $\vec{e}_{z_j} = \lambda \vec{e}_{z_j} + \nabla_{\vec{z}_j} V(s_t)$

8b $\quad\quad$ $\vec{e}_{w_j} = \lambda \vec{e}_{w_j} + \nabla_{\vec{w}} V(s_t)$

9a $\quad\quad$ $\vec{z}_j = \vec{z}_j + \alpha\delta\vec{e}_{z_j}$

9b $\quad\quad$ $\vec{w} = \vec{w} + \alpha\delta\vec{e}_w$

10 $\quad\quad$ $s_t = s_{t+1}$

11 Continue until goal state is reached

As in the linear case, we compromised and implemented for steps 8 and 9,

Steps 8 and 9 for Non-Linear TD(λ)
8a $\vec{e}_{hid} = \gamma\lambda\vec{e}_{hid} + \nabla_{z_j} V(s_t)$
8b $\vec{e}_{out} = \gamma\lambda\vec{e}_{out} + \nabla_w V(s_t)$
9a $\vec{z}_j = \vec{z}_j + \alpha\delta\vec{e}_{hid}$
9b $\vec{w} = \vec{w} + \alpha\delta\vec{e}_{out}$

The non-linear network consists of three layers: an input layer, a hidden layer, and an output layer. The input layer is composed of the inputs to the network, and consists of a note vector and a chord vector. Each of the hidden nodes in the hidden layer then receives as input a copy of the note vector and a copy of the chord vector, and each hidden unit produces as output the linear combination of the input vector times corresponding weight vectors. Since each of the hidden units will have different weight vectors, consequently each will weigh certain aspects of the input differently than other hidden units and each will produce a different output. It is hoped then that each hidden node will focus in on a different aspect of what is being learned For instance, perhaps several hidden nodes will focus in on the prediction value for one of the possible twelve notes with different chords. The output layer consists of one node and it takes the linear sum of the "squashed" outputs of the hidden nodes (i.e., the outputs after being passed through an activation function, see below) and the output node weights. Figure 2.4 presents a visual representation of the network.

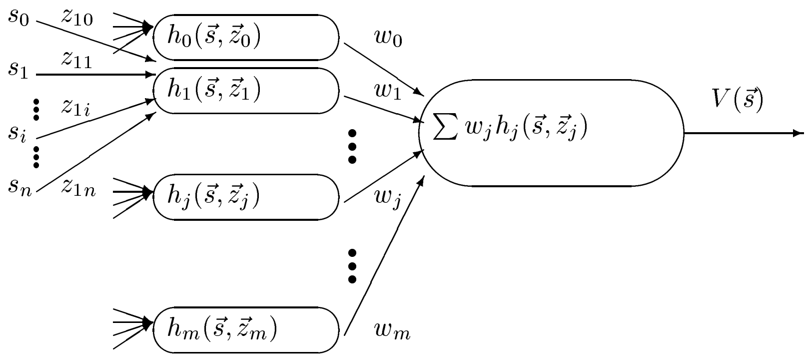

Figure 2.4: General non-linear TD(λ) network with $\vec{s}$ as input.

Backpropagation, see Section 2.3.4 for a summary of the algorithm, is used to update the weights. We derive the backpropagation algorithm indirectly in order to calculate the weight changes. The non-linear TD(λ) algorithm is used to obtain the error δ which the backpropagation algorithm needs in order to do this weight update.

The output of the jth hidden node is therefore given by $h_j(\vec{s}, \vec{z}_j) = f(\sum_{i=0}^{n} s_i z_{ji})$. Using this the output of the network, $V(s_t)$, becomes,

$$
\begin{aligned}
V(s_t) &= \sum_{j=0}^{m} w_j h_j(\vec{s}, \vec{z}_j) \\
&= \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} s_i z_{ji})
\end{aligned}
$$

The activation function being used, $f(\theta)$ is the logistic sigmoid function,

$$
f(\theta) = \frac{1}{1+e^{-\theta}}
$$

If we differentiate $f(\theta)$ we get $f'(\theta) = f(\theta)(1 - f(\theta))$ (see Appendix A). We now show how we modify the linear TD(λ) algorithm steps to include the non-linear network to get the non-linear algorithm.

We first calculate $\nabla_{\vec{z}_j} V(s_t)$ for step 8 and $\nabla_{\vec{w}} V(s_t)$ for step 9. For step 8 we find,

$$
\begin{aligned}
\vec{e}_{z_j} &= \lambda \vec{e}_{z_j} + \nabla_{\vec{z}_j} V(s_t) \\
=> e_{z_{ji}} &= \lambda e_{z_{ji}} + \frac{\partial V(s_t)}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (\sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} s_i z_{ji}))}{\partial z_{ij}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (w_0 f(\sum_{i=0}^{n} s_i z_{0i}) + \ldots + w_j f(\sum_{i=0}^{n} s_i z_{ji}) + \ldots + w_m f(\sum_{i=0}^{n} s_i z_{mi}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (w_j f(\sum_{i=0}^{n} s_i z_{ji}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j \frac{\partial (f(\sum_{i=0}^{n} s_i z_{ji}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} s_i z_{ji}) \frac{\partial (\sum_{i=0}^{n} s_i z_{ji})}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} s_i z_{ji}) \frac{\partial (s_0 z_{j0} + \ldots + s_i z_{ji} + \ldots + s_n z_{jn})}{\partial z_{ji}}
\end{aligned}
$$

$$
\begin{aligned}
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} s_i z_{ji}) s_i \\
&= \lambda e_{z_{ji}} + w_j f(\sum_{i=0}^{n} s_i z_{ji})(1 - f(\sum_{i=0}^{n} s_i z_{ji})) s_i \\
&= \lambda e_{z_{ji}} + w_j h_j(\vec{s}, \vec{z}_j)(1 - h_j(\vec{s}, \vec{z}_j)) s_i
\end{aligned}
$$

For step 9 we find,

$$
\begin{aligned}
\vec{e}_w &= \lambda e_w + \nabla_{\vec{w}} V(s_t) \\
=> e_{w_j} &= \lambda e_{w_j} + \frac{\partial V(s_t)}{\partial w_j} \\
&= \lambda e_{w_j} + \frac{\partial \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} s_i z_{ji})}{\partial w_j} \\
&= \lambda e_{w_j} + \frac{\partial (w_0 f(\sum_{i=0}^{n} s_i z_{0i}) + ... + w_j f(\sum_{i=0}^{n} s_i z_{ji}) + ... + w_m f(\sum_{i=0}^{n} s_i z_{mi}))}{\partial w_j} \\
&= \lambda e_{w_j} + f(\sum_{i=0}^{n} s_i z_{ji}) \\
&= \lambda e_{w_j} + h_j(\vec{s}, \vec{z}_j)
\end{aligned}
$$

Therefore the algorithm is,

1 $\vec{w} = random$, $\vec{e} = 0$

2 For each episode

3 $\quad s_t = s_0$

4 $\quad$ For each state in episode

5 $\qquad$ Get a

6 $\qquad$ Get r and s_{t+1} using a

7 $\qquad \delta = r + \gamma V(s_{t+1}) - V(s_t)$

8a $\qquad \vec{e}_{z_j} = \lambda e_{z_{ji}} + w_j h_j(\vec{s}, \vec{z}_j)(1 - h_j(\vec{s}, \vec{z}_j)) s_i$

8b $\qquad \vec{e}_{w_j} = \lambda e_{w_j} + h_j(\vec{s}, \vec{z}_j)$

9a $\qquad \vec{z}_j = \vec{z}_j + \alpha \delta \vec{e}_{z_j}$

9b $\qquad \vec{w} = \vec{w} + \alpha \delta \vec{e}_w$

$s_t = s_{t+1}$
Continue until goal state is reached

2.4.2 Non-Linear Model $e(\vec{s}_{tNote})$

In this model we use a modified version of the non-linear TD(λ) algorithm that uses the λ eligibility traces of the current notes as the inputs to the hidden units. We have two reasons for wanting to use the eligibility trace as the input. First it makes the math work correctly, as shown in Section 2.3. Second, it gives the neural network some information about the context of each note that it sees. For instance, rather than seeing a single note independent of the preceding notes, the eligibility trace contains information about the history of the note that it is seeing.

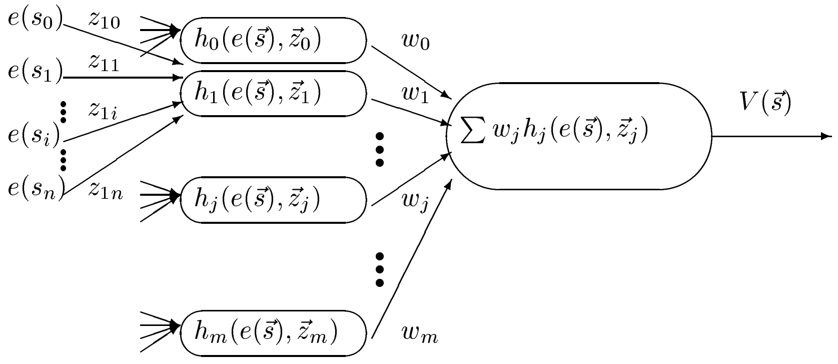

Figure 2.5: Non-linear TD(λ) network with $e(\vec{s})$ as input.

We have $h_j(e(\vec{s}), \vec{z}_j) = \sum_{i=0}^{n} e(s_i) z_{ji}$. Using this, $V(s_t)$ now becomes,

$$V(s_t) = \sum_{j=0}^{m} w_j h_j(e(\vec{s}), \vec{z}_j)$$

$$= \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} e(s_i) z_{ji})$$

Our algorithm outline is then,

1 $\vec{w} = 0$, $\vec{e} = 0$
2 For each episode
3 $\quad s_t = s_0$
4 $\quad$ For each state in episode
5 $\quad\quad$ Get a
6 $\quad\quad$ Get r and s_{t+1} using a
7 $\quad\quad$ $\delta = r + \gamma V(s_{t+1}) - V(s_t)$
8a $\quad\quad$ $\vec{e}_{z_j} = \lambda \vec{e}_{z_j} + \nabla_{\vec{z}_j} V(s_t)$
8b $\quad\quad$ $\vec{e}_{w_j} = \lambda \vec{e}_{w_j} + \nabla_{\vec{w}} V(s_t)$
9a $\quad\quad$ $\vec{z}_j = \vec{z}_j + \alpha \delta \vec{e}_{z_j}$
9b $\quad\quad$ $\vec{w} = \vec{w} + \alpha \delta \vec{e}_w$
10 $\quad\quad$ $s_t = s_{t+1}$
11 Continue until goal state is reached

We must now calculate $\nabla_{\vec{z}_j} V(s_t)$ for step 9 and $\nabla_{\vec{w}} V(s_t)$ for step 10. For step 9 we find,

$$\begin{aligned}
\vec{e}_{z_j} &= \lambda \vec{e}_{z_j} + \nabla_{\vec{z}_j} V(s_t) \\
=> e_{z_{ji}} &= \lambda e_{z_{ji}} + \frac{\partial V(s_t)}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (\sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} e(s_i) z_{ji}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (w_0 f(\sum_{i=0}^{n} e(s_i) z_{0i}) + \ldots + w_j f(\sum_{i=0}^{n} e(s_i) z_{ji}) + \ldots + w_m f(\sum_{i=0}^{n} e(s_i) z_{mi}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (w_j f(\sum_{i=0}^{n} e(s_i) z_{ji}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j \frac{\partial (f(\sum_{i=0}^{n} e(s_i) z_{ji}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} e(s_i) z_{ji}) \frac{\partial (\sum_{i=0}^{n} e(s_i) z_{ji})}{\partial z_{ji}}
\end{aligned}$$

$$
\begin{aligned}
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} e(s_i) z_{ji}) \frac{\partial(e(s_0)z_{j0} + ... + e(s_i)z_{ji} + ... + e(s_n)z_{ni})}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} e(s_i) z_{ji}) e(s_i) \\
&= \lambda e_{z_{ji}} + w_j f(\sum_{i=0}^{n} e(s_i) z_{ji})(1 - f(\sum_{i=0}^{n} e(s_i) z_{ji})) e(s_i) \\
&= \lambda e_{z_{ji}} + w_j h_j(e(\vec{s}), \vec{z_j})(1 - h_j(e(\vec{s}), \vec{z_j})) e(s_i)
\end{aligned}
$$

For step 10 we find,

$$
\begin{aligned}
\vec{e_w} &= \lambda \vec{e_w} + \nabla_{\vec{w}} V(s_t) \\
=> e_{w_j} &= \lambda e_{w_j} + \frac{\partial V(s_t)}{\partial w_j} \\
&= \lambda e_{w_j} + \frac{\partial(\sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} e(s_i) z_{ji}))}{\partial w_j} \\
&= \lambda e_{w_j} + \frac{\partial(w_0 f(\sum_{i=0}^{n} e(s_i) z_{0i}) + ... + w_j f(\sum_{i=0}^{n} e(s_i) z_{ji}) + ... + w_m f(\sum_{i=0}^{n} e(s_i) z_{mi}))}{\partial w_j} \\
&= \lambda e_{w_j} + f(\sum_{i=0}^{n} e(s_i) z_{ji}) \\
&= \lambda e_{w_j} + h_j(e(\vec{s}), \vec{z_j})
\end{aligned}
$$

Our modified algorithm is then:

1 $\vec{w} = random$, $\vec{e} = 0$
2 For each episode
3 $\quad s_t = s_0$
4 $\quad$ For each state in episode
5 $\quad\quad$ Get a
6 $\quad\quad$ Get r and s_{t+1} using a
7 $\quad\quad$ $\delta = r + \gamma V(s_{t+1}) - V(s_t)$
8a $\quad\quad$ $\vec{e}_{z_j} = \lambda \vec{e}_{z_j} + w_j h_j(e(\vec{s}), \vec{z_j})(1 - h_j(e(\vec{s}), \vec{z_j})) e(s_i)$
8b $\quad\quad$ $\vec{e}_{w_j} = \lambda \vec{e}_{w_j} + h_j(e(\vec{s}), \vec{z_j})$

9a $\quad \vec{z}_j = \vec{z}_j + \alpha\delta\vec{e}_{z_j}$

9b $\quad \vec{w} = \vec{w} + \alpha\delta\vec{e}_w$

10 $\quad s_t = s_{t+1}$

11 Continue until goal state is reached

From the algorithm it can be seen that in order to find the eligibility, it is necessary to actually know the eligibility, see steps 8a and 8b. In order to circumvent this we used s_t rather than $e(s_t)$ to calculate each eligibility.

2.4.3 Non-Linear Model $a(\vec{s}_{tNote})$

In this model we are using a modified version of the non-linear TD(λ) algorithm that uses an average trace on the current state. The average trace is then used as the input to the network. The reasoning behind incorporating another trace is the following. The eligibility trace only gives a history for each note, that is, it says how recently and frequently each note was seen. It does not give a history of the sequence, or where the note is situated within good sequences, or what note precedes a note in a good sequence. The average trace from [Jor86] attempts to provide a history that gives this information to the neural network.

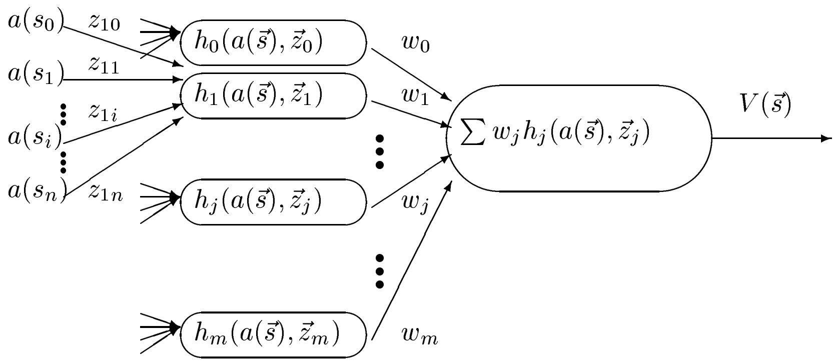

Figure 2.6: Non-linear TD(λ) network with $a(\vec{s})$ as input.

As mentioned the average trace is another mechanism for maintaining a decaying history of the inputs or current state. We choose to implement the average trace by maintaining a decaying history of the current state (the current note) using the equation,

$$a(s_t) \quad = \quad \beta a(s_{t-1}) + (1-\beta)s_{t-1}$$

Using the average trace gives $h_j(a(\vec{s}), \vec{z}_j) = \sum_{i=0}^{n} a(s_i)z_{ji}$. $V(s_t)$ then becomes,

$$\begin{aligned} V(s_t) &= \sum_{j=0}^{m} w_j h_j(e(\vec{s}), \vec{z}_j) \\ &= \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} a(s_i)z_{ji}) \end{aligned}$$

In order to incorporate the average trace, rather than pass s_t, instead $a(s_t)$ is passed as input to the network. Since the derivations for $a(s_t)$ will be the same as for $e(s_t)$, we substitute $a(s_t)$ for $e(s_t)$ in the algorithm given for $e(s_t)$. Therefore the algorithm is given by,

1 $\vec{w} = random$, $\vec{e} = 0$
2 For each episode
3 $\quad s_t = s_0$
4 $\quad$ For each state in episode
5 $\quad\quad$ Get a
6 $\quad\quad$ Get r and s_{t+1} using a
7 $\quad\quad$ $\delta = r + \gamma V(s_{t+1}) - V(s_t)$
8a $\quad\quad$ $\vec{e}_{z_j} = \lambda \vec{e}_{z_j} + w_j h_j(a(\vec{s}), \vec{z}_j)(1 - h_j(a(\vec{s}), \vec{z}_j))a(s_i)$
8b $\quad\quad$ $\vec{e}_{w_j} = \lambda \vec{e}_{w_j} + h_j(a(\vec{s}), \vec{z}_j)$
9a $\quad\quad$ $\vec{z}_j = \vec{z}_j + \alpha\delta\vec{e}_{z_j}$
9b $\quad\quad$ $\vec{w} = \vec{w} + \alpha\delta\vec{e}_w$
10 $\quad\quad$ $s_t = s_{t+1}$
11 Continue until goal state is reached

2.4.4 The δ Rule and Backpropagation

The weight update rule for a linear network is the δ Rule, where x_j is the jth input to the linear network [Pic00].

$$w_j = w_j + \alpha\delta_{Output}x_j$$

Using a non-linear network with a single output node and multiple hidden nodes, the backpropagation algorithm (from [Bis95]) is the following.

1. Compute the error for the output node. This is the difference between the correct output, c, and the actual output.

$$\delta_{Output} = c - a$$

2. Compute the error for each hidden node. This is given by the following, where h_j is used to represent $h_j(\vec{s}, \vec{z}_j)$.

$$\delta_{Hidden} = h_j(1 - h_j)w_j\delta_{Output}$$

3. Update the weights for the output node.

$$w_j = w_j + \alpha\delta_{Output}h_j$$

4. Update the weights for each hidden node.

$$z_{ji} = z_{ji} + \alpha\delta_{Hidden}s_i$$

We derived the backpropagation algorithm indirectly in the previous sections. We choose to replace δ_{Output} with the prediction error, $\delta = r + V(s_{t+1}) - V(s_t)$.

2.5 Eligibility and Average Traces

The purpose of both the eligibility trace and the average trace is to maintain some sort of history. The reason this history is so necessary for our experiments is because the reinforcement learning problem that we are looking at is not a Markov decision process. A Markov decision process defines a situation in which transitions are occuring from one state to another, and the information contained in the current state includes in some form or another the information of all of the previous states; this is the Markov property [SB98]. For instance for Gerald Tesauro's TD-Gammon [Tes95] in which a neural network using temporal-difference learning became proficient at the game of backgammon, the reinforcement learning problem was a Markov decision process. This is because each state was the current representation of the backgammon board, that contained all the previous states as well as the current state [Tes95].

In our reinforcement learning problem, a note is a state. A note in itself is context-dependent of the sequence that it is in. In order for the TD(λ) learning algorithm to have some sense of the context of the note, eligibility traces are used in the weight update. But since at the time the network actually receives some input, i.e., a note in the sequence, the network will be expected to produce some output for the note. In order for the network to receive some history along with the note, the possibility of using the eligibility trace as input seems promising. However, since in order to calculate the eligibility trace it is necessary to know the eligibility trace, we added another trace, the average trace. The sole purpose of the average trace is to keep a running average of the notes. Chapter 3 presents the results of the two types of linear TD(λ) experiments and the three types of non-linear TD(λ) experiments. It is there that the usefulness and limitations of the average trace can be seen most clearly. In short the average trace was found to be very useful with the non-linear counting experiments and moderately useful with the non-linear ordered sequence experiments. Due to lack of time, results with Sarsa using the average trace are inconclusive (there were no major differences when using the average trace as the input from simply using the state as the input).

2.6 Input and Network Representation

The main data structures used are to represent the input to the network and to represent the network itself. The input to the network consists of a note and a chord. The note is part of a larger sequence of notes but the network receives only one note at a time as input. A note is represented by 12 binary digits stored as a single-dimensional array of length 12. Each binary digit corresponds to a note type, hence there are 12 note types. Table 2.1 shows the twelve possible note types and their array representation. Each index of the array is then mapped to one of the note types. For instance, note type 0 is mapped to index 0 of the array by placing a one at index zero in the array and placing zeroes at all of the other indices.

Note Type	**Array Representation**
Note 0	1 0 0 0 0 0 0 0 0 0 0 0
Note 1	0 1 0 0 0 0 0 0 0 0 0 0
Note 2	0 0 1 0 0 0 0 0 0 0 0 0
...	...

Table 2.1: Array representation of a note.

A sequence is a two-dimensional array with the first dimension representing the index of the note type within the sequence and the second dimension representing the note type. For instance, a sequence of the notes 2, 5, 7, and 9 would be given by,

0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0

A chord is represented by a one-dimensional array of length 12 and is essentially four note representations overlaid on top of each other. For instance a chord consisting of notes 2, 5, 7, and 9 would be represented by,

0 0 1 0 0 1 0 1 0 1 0 0

There are two important "traces" that we have already mentioned that are stored as well: the eligibility trace and the average trace. Depending on the algorithm and/or the network being used for the experiment either trace or both may be used. The eligibility trace we have already seen as $e(s_t)$. The average trace we have already seen as $a(s_t)$. An eligibility trace and an average trace is maintained on each digit of the note. Therefore the eligibility traces and average traces are each represented by an array of length 12.

Lastly the current index of the note within the sequence is stored. Since the length of the sequence is known all that is stored is an integer. When it comes time to pass the note index to the network, the network creates a temporary array, with length the number of notes in the sequence, initialized to zero with the index corresponding to the current note index initialized to 1.

The linear network consists of an input layer comprised of 24 inputs or nodes (one input for each bit of a note or chord) and an output layer comprised of a single node. The only part of the linear network that needs to be stored are the inputs, the weight corresponding to each input, and the eligibility trace corresponding to each input. Hence there is also a weight chord array, a weight note array, an eligibility trace chord array, and an eligibility trace note array.

The non-linear network consists of an input layer comprised of 24 inputs, a hidden layer containing n_{Hid} hidden nodes, and an output layer consisting of a single node. What is actually stored is the input chord array, input note array, weight chord array for each hidden node, weight note array for each hidden node, weight array of length n_{Hid} for the output node, eligibility trace chord array, eligibility trace note array and/or average trace note array, and an eligibility trace array of length n_{Hid} for the output node.

Although biases are often used with neural networks [Bis95] we chose not to have a bias input on either the hidden or output nodes. Originally biases were used on these nodes but were found not to be helpful.

2.7 Goals

Our main goal for using TD(λ) was to have an agent learn that certain notes played in a certain order are good, and that certain notes played with certain chords are good.

Chapter 3

TD(λ) Experiments

3.1 Introduction

Using TD(λ) and a linear neural network allows a linear function to be approximated while using TD(λ) and a non-linear network allows a non-linear function to be approximated. Therefore, two sets of TD(λ) experiments were run: one set of experiments that used a linear function approximator and one set of experiments that used a nonlinear function approximator. Before presenting the results, some abbreviations are first given in Table 3.1.

Parameter	Abbreviation
Number of Rounds	n_{Rounds}
Number of Hidden Nodes	n_{Hid}
Number of Notes in a Sequence	n_{Notes}
Number of Chords	n_{Chords}
Number of Sequences	n_{Seqs}
Length of a Note	len_{Note}
Length of a Chord	len_{Chord}

Table 3.1: Abbreviations of different parameters.

Note that for all experiments $len_{Note} = 12$ and $len_{Chord} = 12$. This is to correspond to the 12 notes in the chromatic scale.

3.2 Linear Experiments

Here we present the experimental results for a linear network combined with TD(λ).

3.2.1 Ordered Sequence

The goal of the linear ordered sequence experiments was to determine whether TD(λ) could use a linear function approximator to learn to distinguish a good sequence from a bad one.

The experiment was run in the following way. On each of n_{Rounds} a sequence was randomly chosen from an existing data set. For each sequence chosen, one note in the sequence was passed at a time to the linear network. On the last note in the sequence, a reward was given to the network that was used in the calculation of δ. If the sequence had been good then this reward was 1, otherwise this reward was 0. The sequence that was arbitrarily decreed to be good was 0 1 2. Other sequences such as 2 1 0 that contained the same notes as the good sequence were considered to be bad, as were any other sequences besides 0 1 2. In the data set, the only sequences that contained 0 1 or 2 were the good sequences. This was a way of making it as clear as possible for the network what it needed to learn.

It was found that the linear network can learn very well that a certain ordered sequence, in this case 0 1 2, is a good sequence. The linear network propagates the reward back to the initial note of the sequence, 0, and learns to predict that the sequence is good as soon as it sees the note 0. Prediction values for notes 0, 1, and 2 are very close to 1 or are 1. Any other notes in any of the other sequences have prediction values, $V(s_t)$ values, that are very close to zero.

We then looked at whether the linear network could distinguish the sequence 0 1 2 from the sequence 2 1 0. We wondered whether it could learn that only the ordered sequence 0 1 2 was good, if it had not been trained on sequences containg 0 1 or 2 in bad sequences. It was found that the linear network could not learn this, which was not surprising due to the limited capabilities of linear networks [HDB96]. The linear network had simply "memorized" that notes 0, 1, and 2 are good.

The training sequences consisted of ten sequences of 0 1 2 and ten sequences excluding notes 0, 1, and 2. Perhaps adding sequences of 2 1 0 that went unrewarded would improve the ability of the linear network to learn that 2 1 0 is bad, but it is more likely that the linear network will simply learn that 0 1 2 is not that good and is the same as 2 1 0 being somewhat good. This is because the linear network cannot give two different predictions for the same inputs, regardless of the order. That is, note 2 must either be good or bad to the linear network, it cannot be both. While using the average trace worked well for the non-linear network, it was never tried with the linear network, because given limited time, we thought that the non-linear case was more interesting.

1. **Linear Model $\vec{s}_{tNote}$**

   - In this model, recall that the input to the network is the array of bits composing a note, $\vec{s}_{tNote}$. The output is the prediction, $V(s_t) = \sum_{i=1}^{n} w_i \cdot s_i$. When tested on the sequence 0 1 2, predicted values were close to 1, while when tested on other sequences, predicted values were close to zero. When tested on the sequence 2 1 0 predicted values were again close to 1, with the same predictions given for each note as given for sequence 0 1 2. See Table 3.2.
   - These results follow from a linear network being capable of learning that x is good and y is bad, but being unable to learn that x is good given circumstances w but x is bad given circumstances z.

- Parameters were: $n_{Rounds} = 1000$, $n_{Hid} = 20$, $n_{Seqs} = 20$, $n_{Notes} = 3$, $\alpha = 0.01$, $\lambda = 0.7$, and $\gamma = 0.9$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2	Example Sequence 3
V_0 (0) = 0.9025	$V_0(6)$ = -9.96404e-22	$V_0(2) = 1$
V_1 (1) = 0.95	$V_1(9)$ = -3.44718e-21	$V_1(1) = 0.95$
V_2 (2) = 1	$V_2(5)$ = -2.83036e-22	$V_2(0) = 0.9025$

Table 3.2: Output for the linear ordered sequence experiments using the current state as the input.

2. **Linear Model** $e(\vec{s_t}_{Note})$

   - This model attempts to give the network some history of what it has seen by using as the input the eligibility trace rather than the note. The input to the network is therefore $e(\vec{s_t}_{Note})$, and $V(s_t) = \sum_{i=1}^{n} w_i \cdot e(s_i)$. When tested on the sequence 0 1 2 predicted values ranged from approximately 1.5 to 2.4, while predictions for other notes ranged from approximately 0.3 to 0.8. See Table 3.3.
   - Similarly for the preceding linear model with the state as the input, these results follow from what a linear network is capable of learning.

- Parameters were: $n_{Rounds} = 1000$, $n_{Hid} = 20$, $n_{Seqs} = 20$, $n_{Notes} = 3$, $\alpha = 0.01$, $\lambda = 0.7$, $\gamma = 0.9$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2
$V_0(0) = 1.58641$	$V_0(6) = 0.377576$
$V_1(1) = 2.00499$	$V_1(9) = 0.181708$
$V_2(2) = 2.32965$	$V_2(5) = 0.375965$

Table 3.3: Output for the linear ordered sequence experiments using the eligibility of the current state as the input.

3.3 Non-linear Experiments

Here we present the experimental results for a non-linear network combined with TD(λ). Various parameters were tried and these experiments represent the best results of those values.

3.3.1 Counting

The goal of the non-linear counting experiments was to determine whether a non-linear network could distinguish that, given enough "good" notes in a sequence that the sequence was good, while given too few "good" notes that the sequence was bad.

The note that was chosen to be good was note 2. Each sequence consisted of five notes in total. The training sequences were comprised of 10 sequences that consisted only of note 2 and 10 sequences that consisted of two notes of note 2 randomly distributed within the sequence and three other random notes.

On the last note of the sequence a reward of 1 was given if the sequence had consisted only of note 2, otherwise a reward of zero was given.

1. **Non-Linear Model** $\vec{s_t}_{Note}$

    - In this model the input to the network is again the array of bits composing a note in the current sequence, $\vec{s_t}_{Note}$, giving as the prediction, $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} s_i z_{ji})$. When tested on the sequence 2 2 2 2 2, the predicted values for 2 ranged from approximately 2.6 to 2.7. When tested on the sequence 1 2 6 2 4, the predicted values for 2 were even higher, nearing 2.8, while the predicted values for the other notes were lower, ranging from approximately 2.06 to 2.15. See Table 3.4.

    - As would be expected with the linear model but not with the non-linear, the network is unable to distinguish that note 2 is good only in certain circumstances (namely, when there is a sufficient number of note 2s). However, since this experiment was using a non-linear network, we must search for an explanation why we are seeing these results. We propose in explanation that in order for the network to see a note 2 and recognize whether it is good or bad, it needs some context for that note. More precisely, it needs information about whether the note 2 was preceded by other note 2s or by non-2 notes. Here, the non-linear network is being given no history since only the current state is the input. Therefore the network receives insufficent information. Thus, we next tried first the eligibility trace, and then the average trace as input to the non-linear network.

- Parameters were: $n_{Rounds} = 5000$, $n_{Hid} = 20$, $n_{Seqs} = 20$, $n_{Notes} = 5$, $\lambda = 0.95$, $\gamma = 0.9$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2
$V_0(2) = 2.68966$	$V_0(1) = 2.08545$
$V_1(2) = 2.66105$	$V_1(2) = 2.78826$
$V_2(2) = 2.65414$	$V_2(6) = 2.15045$
$V_3(2) = 2.64104$	$V_3(2) = 2.79593$
$V_4(2) = 2.62245$	$V_4(4) = 2.06616$

Table 3.4: Output for the non-linear counting experiments using the current state as the input.

2. **Non-Linear Model** $e(\vec{s_t}_{Note})$

   - In this model the input to the network is again the eligibility trace of the current note rather than just the current note. The input to the network is therefore $e(\vec{s_t}_{Note})$, and $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} e(s_i) z_{ji})$. When tested on the sequence 2 2 2 2 2, the predicted values for 2 were approximately 1.6 to 2.9. When tested on the sequence 1 2 6 2 4, the predicted values for note 2 were both higher and lower, ranging from about 1.7 to about 2.8, while the predicted values for the other notes were both lower and higher, ranging from approximately 1.4 to 2.9. One unexplained aberration is why the prediction for note 2 in the first position ($V_1(2)$), is so low for both Example Sequence 1 and Example Sequence 2. See Table 3.5.

   - Due to the nature of the updating needed for $e(\vec{s_t}_{Note})$, it is necessary to know the hidden units' outputs for the current state to find $e(\vec{s_t}_{Note})$ and $e(\vec{s_{t+1}}_{Note})$ and it is necessary to use $e(\vec{s_t}_{Note})$ and $e(\vec{s_{t+1}}_{Note})$ in order to calculate the hidden units' outputs for the current state. This is a result of the equations being used. In the actual implementation $\vec{s_{t+1}}_{Note}$ is used rather than $e(\vec{s_{t+1}}_{Note})$ in order to calculate $e(\vec{s_{t+1}}_{Note})$ This may account for the ineffectiveness of using $e(\vec{s_t}_{Note})$. Thus, the eligibility trace experiments for the non-linear network are not useful. However, in order for the network to make the distinctions that are being asked of it, such as note 2 being good in the context of one sequence, but not another, some sort of history is necessary. (Hence, the average trace shown ahead in counting experiment 3.)

- Parameters were: $n_{Rounds} = 5000$, $n_{Hid} = 20$, $n_{Seqs} = 20$, $n_{Notes} = 5$, $\alpha = 0.05$, $\lambda = 0.95$, $\gamma = 0.9$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2
$V_0(2) = 2.91674$	$V_0(1) = 2.85463$
$V_1(2) = 1.5932$	$V_1(2) = 1.71872$
$V_2(2) = 2.80643$	$V_2(6) = 1.41351$
$V_3(2) = 2.83752$	$V_3(2) = 2.77211$
$V_4(2) = 2.8331$	$V_4(4) = 2.66826$

Table 3.5: Output for the non-linear counting experiments using the eligibility of the current state as the input.

3. **Non-Linear Model** $a(\vec{s_t}_{Note})$

   - In this model the input to the network is now the average trace of the current note and past notes in the sequence. The input to the network is therefore $a(\vec{s_t}_{Note})$, and $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} a(s_i) z_{ji})$
   - When tested on the sequence 2 2 2 2 2, the predicted values for note 2 were approximately 0.87 to 0.92. When tested on the sequence 1 2 6 2 4, the predicted values for note 2 were lower, first about 0.4 then about -0.2, while the predicted values for the other notes were also low, ranging from about -0.3 to about 0.08. See Table 3.6.

- The average trace works well here. This is because the average trace will have a 1 for the note being counted and all other notes will be zero in the average trace when a reward occurs. So when fewer occurrences of the note being counted are seen, along with actually seeing other notes, then the prediction values for the note being counted is lower. It is seen with the counting experiments that in using $a(\vec{s}_{tNote})$, rather than getting bad values for seeing only two note 2s, it appears that instead the average trace delivers instead a value corresponding to how many consecutive note 2s there are (when the sequence does not consist only of note 2s) and how early on in the sequence they are seen.
- Parameters were: n_{Rounds} = 100000, n_{Hid} = 60, n_{Seqs} = 20, n_{Notes} = 5, α = 0.02, β = 0.6, λ = 0.6, γ = 0.95, Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2	Example Sequence 3
$V_0(2)$ = 0.915713 $V_1(2)$ = 0.901262 $V_2(2)$ = 0.894983 $V_3(2)$ = 0.885006 $V_4(2)$ = 0.87292	$V_0(1)$ = 0.0817891 $V_1(2)$ = 0.428296 $V_2(6)$ = -0.374205 $V_3(2)$ = -0.207129 $V_4(4)$ = 0.0728026	$V_0(1)$ = 0.0817891 $V_1(3)$ = -0.287966 $V_2(5)$ = -0.326363 $V_3(2)$ = -0.180622 $V_4(2)$ = 0.228415
Example Sequence 4	Example Sequence 5	
$V_0(2)$ = 0.915713 $V_1(2)$ = 0.901262 $V_2(5)$ = 0.109215 $V_3(4)$ = 0.0922673 $V_4(8)$ = 0.124575	$V_0(1)$ = 0.0817891 $V_1(5)$ = -0.12642 $V_2(6)$ = -0.458215 $V_3(7)$ = -0.404627 $V_4(4)$ = -0.561135	

Table 3.6: Output for the non-linear counting experiments using the average trace of the current and preceding states in the sequence as the input.

3.3.2 Ordered Sequence

These experiments are a repeat of the linear ordered sequence experiments, except now with a non-linear network. In the non-linear model, unlike in the linear model, we allow the network to learn on sequences that contain 0 1 or 2 in a random order (such as 2 1 0) as well as in the order 0 1 2. We found that there was little difference between the output of the non-linear network when it was allowed to also learn these sequences versus when it was given information about notes 0 1 and 2 only in the context of the sequence 0 1 2.

1. **Non-Linear Model** $\vec{s_t}_{Note}$

   - In this model the input to the network is again the array of bits composing a note in the current sequence, $\vec{s_t}_{Note}$ giving as the prediction, $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} s_i z_{ji})$. When tested on the sequence 0 1 2, predicted values were approximately 2.59 to 2.88. When tested on the sequence 2 1 0, the predicted values were lower, ranging from about 2.20 to 2.56. See Table 3.7.
   - The non-linear network has learned to some extent that the prediction for note 0 is best when it is in position 0 in the sequence and correspondingly for notes 1 and 2. The network has not learned to distinguish that note 1 is only good if preceded by note 0 and that note 2 is good only if preceded by 0 1. We see this from Example Sequence 3 in which the value of note 2 in the last position is even higher than when preceded by notes 0 1. We attempted to address this problem by giving the network the average trace as the input; see the second Non-Linear Model for ordered sequences.

- Parameters were: n_{Rounds} = 300000, n_{Hid} = 60, n_{Seqs} = 30, n_{Notes} = 3, $\alpha = 0.02$, $\beta = 0.6$, $\lambda = 0.6$, $\gamma = 0.95$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2
$V_0(0) = 3.1158$ $V_1(1) = 3.36427$ $V_2(2) = 3.52844$	$V_0(2) = 3.50465$ $V_1(1) = 3.15999$ $V_2(0) = 2.88441$
Example Sequence 3	Example Sequence 4
$V_0(3) = 2.42279$ $V_1(4) = 2.40006$ $V_2(2) = 3.66669$	$V_0(3) = 2.42279$ $V_1(4) = 2.40006$ $V_2(5) = 2.33351$

Table 3.7: Output for the non-linear ordered sequence experiments using the current state as the input.

2. **Non-Linear Model** $a(\vec{s}_{tNote})$

   - In this model the input to the network is now the average trace of the current note and past notes in the sequence. The average trace was, however, only used as input to calculate the current note prediction, $V(s_t)$, not the next note precdiction, $V(s'_t)$. (This decision was made based on a comparison of the output in Tables 3.8, 3.9 and 3.10.) The calculation for the next note prediction used only the next state. The input to the network is therefore $a(\vec{s}_{tNote})$, and $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} a(s_i) z_{ji})$
   - When tested on the sequence 0 1 2, the predicted values ranged from about 1.5 to about 2.1. When tested on the sequence 2 1 0, the predicted values ranged from 1.6 to 1.9. The value for each of the notes 0 1 and 2 is higher when in the appropriate position in the sequence 0 1 2 than when in the sequence 2 1 0. When note 2 is preceded by 3 4 rather than 0 1, the value for note 2 is even lower at

about 1.6. See Table 3.8.

- The average trace gives results that are slightly better than just using the note as input. Most noticeably, the prediction for note 2 drops to about 1.6 from about 2.16 when note 2 is preceded by notes 3 4 rather than 0 1. There are still limitations, however, to using either the average trace of the current note or simply the current note as input to the network. For instance, given the sequence 0 1 2 with a reward of 1 given on the last note, then the predictions for notes 0, 1 or 2 will be higher than for any other note. This even the linear network learns. What the linear network does not learn is that 0, 1 or 2 in other positions within the sequence is bad. The nonlinear network learns this to some extent. For instance, in the sequence 2 1 0, the value for 2 and 1 will be lower than when preceded by 0 and/or 1, but the value for 0 will be equally high. This is because the average trace being put through the network will have all 2 1 0 components, similar although not identical to nor in the same proportions as the 0 1 2 components that received the reward. It is hypothesized that there is enough overlap stored in the average trace here to allow the network to give higher predictions for these inputs. For the sequence 3 4 2, we find that the value for 2 is still lower than when preceded by 0 and 1. As a point of reference the predictions for all notes in the sequence 3 4 5 are lower than any of the predictions for 0 ,1 or 2 in the 0 1 2 sequence.

- Parameters were: $n_{Rounds} = 300000$, $n_{Hid} = 60$, $n_{Seqs} = 30$, $n_{Notes} = 3$, $\alpha = 0.02$, $\beta = 0.6$, $\lambda = 0.6$, $\gamma = 0.95$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2
$V_0(0) = 1.56168$	$V_0(2) = 1.93069$
$V_1(1) = 1.82923$	$V_1(1) = 1.64349$
$V_2(2) = 2.16036$	$V_2(0) = 1.84733$
Example Sequence 3	Example Sequence 4
$V_0(3) = 1.21671$	$V_0(3) = 1.21671$
$V_1(4) = 1.04792$	$V_1(4) = 1.04792$
$V_2(2) = 1.60931$	$V_2(5) = 1.01137$

Table 3.8: Output for the non-linear ordered sequence experiments using the average trace for the current and preceding states in the sequence as the input.

Note that the primary difference between Tables 3.9 and 3.10 is that in the experiments for Table 3.9, $n_{Rounds} = 100000$ while in Table 3.10, $n_{Rounds} = 300000$.

Example Sequence 1	Example Sequence 2
$V_0(0)$ = 0.926711	$V_0(2)$ = 0.626061
$V_1(1)$ = 4.46143	$V_1(1)$ = 4.93582
$V_2(2)$ = 5.0207	$V_2(0)$ = 4.95203
Example Sequence 3	Example Sequence 4
$V_0(3)$ = -0.104156	$V_0(3)$ = -0.104156
$V_1(4)$ = 0.73898	$V_1(4)$ = 0.73898
$V_2(2)$ = 2.67724	$V_2(5)$ = 1.76163

Table 3.9: Output for the non-linear ordered sequence experiments using the average trace as the input. The average trace was used as the input to calculate both the current note prediction, $V(s_t)$, and the next note prediction, $V(s_t')$. Parameters were $n_{Rounds} = 100000$, $n_{Hid} = 60$, $n_{Seqs} = 30$, $n_{Notes} = 3$, $\alpha = 0.02$, $\beta = 0.6$ $\lambda = 0.6$, $\gamma = 0.95$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2
$V_0(0)$ = -11.9739	$V_0(2)$ = -1.25587
$V_1(1)$ = -6.03075	$V_1(1)$ = -8.91193
$V_2(2)$ = -2.01644	$V_2(0)$ = -7.56653
Example Sequence 3	Example Sequence 4
$V_0(3)$ = -9.61921	$V_0(3)$ = -9.61921
$V_1(4)$ = 9.60173	$V_1(4)$ = 9.60173
$V_2(2)$ = 12.3745	$V_2(5)$ = 9.26105

Table 3.10: Output for the non-linear ordered sequence experiments using the average trace as the input. The average trace was used as the input to calculate both the current note prediction, $V(s_t)$, and the next note prediction, $V(s_t')$. Parameters were $n_{Rounds} = 300000$, $n_{Hid} = 60$, $n_{Seqs} = 30$, $n_{Notes} = 3$, $\alpha = 0.02$, $\beta = 0.6$ $\lambda = 0.6$, $\gamma = 0.95$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

3.3.3 Context Switch

The last experiment is context-switching in which the reinforcement is calculated based on characteristics of the previous sequence and the current chord. In these experiments, the first note of a sequence gets a reward of 1 if the last note of the previous sequence ended with a 0 and the current chord for the current sequence is given by C2, where the following notation is used,

$$\begin{aligned} \text{C0} &= 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 0\ 1\ 1\ 0 \\ \text{C1} &= 1\ 0\ 0\ 1\ 0\ 1\ 0\ 0\ 0\ 1\ 0\ 0 \\ \text{C2} &= 1\ 0\ 0\ 0\ 1\ 0\ 0\ 1\ 0\ 0\ 1\ 0 \end{aligned}$$

1. **Linear Model $\vec{s_t}_{Note}$**

   - In this model the input to the network is just the current note, $\vec{s_t}_{Note}$. Thus, $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} s_i z_{ji})$.
   - From Table 3.11 it can be seen that the program is unable to learn that note 0 is good or that notes played with C2 are good. We see this from the low values for all the notes regardless of whether a note is note 0 or is played when C2 is the current chord. So it does not learn that 0 as the last note in the sequence preceding a sequence played with C2 is good.
   - It can be seen that values for all notes are extremely low. We hypothesize that this is due to the network being unable to distinguish the importance of note 0, since it has no history of the previous sequence. Without the realization that note 0 as the last note of the preceding sequence is critical for the reward of 1, then that reward of 1 will end up being attributed to all notes without some other overreaching constraint; while it will be able to attribute the reward = 1 to chord 2, it is possible for chord 2 to show up without 0 as the last note of the preceding sequence, in which case chord 2 will not be rewarding, resulting in possible confusion for the network. So ultimately it will be unlikely that the network will be able to distinguish that chord 2 was important for receival of a reward = 1.

- Parameters were: n_{Rounds} = 60000, n_{Hid} = 60, len_{Chord} = 12, n_{Seqs} = 22, n_{Chords} = 3, n_{Notes} = 3, $\alpha = 0.02$, $\lambda = 0.6$, $\gamma = 0.95$. Finally, the weights were initialized to values ranging between 0.01 and 0.001.

Example 1
$V_0(2)$ = -9.6873e-05
$V_1(10)$ = 2.64126e-05
$V_2(0)$ = -0.000127041
C1, NC0
$V_0(5)$ = 0.000254639
$V_1(7)$ = 0.000213351
$V_2(0)$ = 9.67255e-05
C0, NC2
$V_0(6)$ = -7.93748e-05
$V_1(3)$ = 4.49982e-05
$V_2(10)$ = -2.55082e-06
C2, NC1
$V_0(2)$ = -0.000126258
$V_1(10)$ = -2.55621e-06
$V_2(3)$ = 4.39431e-05
C1, NC0

Table 3.11: Output for the linear context-switch experiments using the current state as the input. Note that C = chord, NC = next chord.

2. **Non-linear Model $a(\vec{s}_{tNote})$, $\vec{s}_{tChord}$; Overlapping Sequences.**

   - In this model the input to the network is again the average trace of the current note and past notes in the sequence. This input to the network is therefore $a(\vec{s}_{tNote})$. Therefore for this input, $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} a(s_i)z_{ji})$ Again the average trace was only used as input to calculate the current note prediction, $V(s_t)$, not the next note precdiction, $V(s'_t)$. The calculation for the next note prediction used only the next state. Other inputs to the network consisted of the chord. Therefore for these inputs, $V(s_t) = \sum_{j=0}^{m} w_j f(\sum_{i=0}^{n} s_i z_{ji})$. Finally, the first note of each sequence is given twice. The reasoning behind this is that if the note has already been seen then it is justifiable to be able to see it again.

- What we see happening is that the network learns very well that chord C2 is very good only if the preceding sequence ended with note 0. For instance, in Example Sequence 1, the last sequence (10 2 9), which has C2 as its current chord and 0 as the last note of the preceding sequence (11 1 0), has higher prediction values of about 1.48 than does the first sequence in the table (9 2 5) which does not meet these criteria and only has prediction values of about 1.37. However, the network has difficulty distinguishing that note 0 is only good when the next chord is C2. We see this in Example Sequence 2, where in the second sequence (1 11 0), 0 is the last note of the sequence and the next chord is C0 (indicated by NC0), but the prediction for note 0 is still very high at about 1.88. Interestingly, the first note of the last sequence, note 6, has a much higher prediction value (about 1.62) the first time it is seen than the second time it is seen (about 1.34). This is possibly due to the first note of the next sequence (on the first time it is seen) receiving positive reward as the reward of 1 is propagated back to note 0 when the next chord is C2. Finally, Example Sequence 3 shows the lower prediction values for notes when the current chord is C2 and the preceding sequence (not shown) does not end with note 0. See Table 3.12.

- Parameters were: $n_{Rounds} = 60000$, $n_{Hid} = 60$, $n_{Seqs} = 22$, $n_{Chords} = 3$, $n_{Notes} = 3$, $\alpha = 0.02$, $\beta = 0.6$, $\lambda = 0.6$, $\gamma = 0.95$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Example Sequence 1	Example Sequence 2	Example Sequence 3
$V_0(9) = 1.39215$	$V_0(11) = 1.44794$	$V_0(9) = 1.39215$
$V_1(2) = 1.37054$	$V_1(6) = 1.39954$	$V_1(2) = 1.37054$
$V_2(5) = 1.36032$	$V_2(4) = 1.36728$	$V_2(5) = 1.36032$
$V_0(11) = 1.32569$	$V_0(1) = 1.32545$	$V_0(11) = 1.32569$
C2, NC0	C0, N1	C2, NC0
$V_0(11) = 1.34449$	$V_0(1) = 1.48482$	
$V_1(1) = 1.32847$	$V_1(11) = 1.48415$	
$V_2(0) = 1.69711$	$V_2(0) = 1.87852$	
$V_0(10) = 1.51262$	$V_0(6) = 1.61588$	
C0, NC2	C1, NC0	
$V_0(10) = 1.49736$	$V_0(6) = 1.33504$	
$V_1(2) = 1.47925$	$V_1(6) = 1.32752$	
$V_2(9) = 1.47375$	$V_2(0) = 1.73067$	
$V_0(11) = 1.43669$	$V_0(6) = 1.45357$	
C2, NC0	C0, NC1	

Table 3.12: Output for the non-linear context-switch experiments using the average trace for the current and preceding states as the input. Note that C = chord, NC = next chord.

Chapter 4

Reinforcement Learning Models

4.1 Introduction

Sarsa(λ) is an algorithm for choosing actions [SB98]. Its name comes from the inputs it uses and the outputs it produces: s_t, a_t, r_{t+1}, s_{t+1}, and a_{t+1}. Sarsa(λ) takes the current state, s_t, calculates the best action to take in that state, a_t, finds the reward, r_{t+1} for taking a_t in s_t, and then uses a_t to find s_{t+1}, and then computes the next action a_{t+1}.

In order to choose the best action, both a policy and a value function are needed. The temporal-difference algorithm presented in Chapter 2 is integrated into Sarsa(λ) to provide the error needed to calculate the Q-values for a policy. Thus there are also tabular, linear, and nonlinear Sarsa(λ) algorithms. As with the TD(λ) algorithm we will look only at the linear and nonlinear Sarsa(λ) algorithms.

Sarsa(λ) uses the temporal-difference error found with TD(λ) to compute the best action to take. Rather than have a value $V(s_t)$ that predicts how good or bad the future looks from a state, instead a "Q-value", $Q(s_t, a_t)$, is learned and maintained that predicts how good each action will be when taken in a certain state. The Q-value then represents the combined policy and value functions. There is another model that separates the policy from the value function: the actor-critic model [SB98]. In this model, the policy function, as always, chooses actions and is therefore the actor, while the value function uses temporal-difference learning to give feedback about the actions chosen, that is "criticizes" them, and is therefore the critic. The critic gives feedback to the actor as the error. We chose not to use the actor-critic model.

4.2 Linear Sarsa(λ)

Presented here is the linear Sarsa(λ) algorithm from [SB98]. Let $A(s_t)$ be the set of actions available when in state s_t.

1 $\vec{w} = random$, $\vec{e} = 0$
2 For each episode
3     $s_t = s_0$, $a_t = a_0$
4   For each state in episode
5         $\vec{e} = \lambda\vec{e} + s_t$
6   Get r using a_t
7         $s_{t+1} = a_t$
8         $Q_a = \sum_{i=0}^{n} w_i s_i$
9         $a_{t+1} = argmax_a Q_a$
10   With probability ϵ, a_{t+1} = random $a_i \in A(s_t)$
11         $\delta = r + \gamma Q_a(s_{t+1}) - Q_a(s_t)$
12         $\vec{w} = \vec{w} + \alpha\delta\vec{e}$
13         $a_t = a_{t+1}$, $s_t = s_{t+1}$
14     $\vec{e} = 0$
15 Continue until goal state is reached

The linear Sarsa(λ) algorithm works in the following way. The algorithm loops, first updating the eligibilities for each state (step 5), then calculating r using the current a_t on the last note in the sequence (step 6). The next state is then updated to be the current action, since by definition the current action is used to choose the next state, which is in our case, the next note to play (step 7). The Q-values for each action are calculated in step 8. Based on these Q-values, the action corresponding to the highest Q-value is stored in step 9 as the next action. Using the ϵ-greedy policy in step 10, then with probability ϵ a random action will be stored as the next action, overwriting the previously stored action. The error δ is then calculated in step 11. Note that the Q-values used in step 10 are associated with the actual action and next action that were taken, not necessarily the actions that were found in step 9. This means that Q-values belonging to sub-optimal actions, from choosing a random action using the ϵ-greedy policy, can be used in the weight update. This is in contrast to Q-learning which uses the Q-value for the optimal action to do the weight update. The weights are then updated in step 12 . Finally the next state and action are stored as the current state and action respectively, in preparation for the coming steps 5 and 6.

Thus, most of the time Sarsa(λ) chooses the action that corresponds to the highest Q-value, that is, the state and action pair that gives the highest predicted value. But in order to learn which actions give the highest predicted value in a state, exploration is needed. Sub-optimal actions need to be explored in order to determine which actions are optimal. This is called the exploration/exploitation problem [SB98]: how much time should be spent exploring before the agent should start exploiting the information it has obtained Therefore, ϵ-greedy is a way of choosing actions, by choosing either the best action or some exploratory action with certain probabilities. Note that both Q-learning and Sarsa can use ϵ-greedy to choose actions [SB98]. ϵ-greedy says to choose an action at random with probability $1 - \epsilon$ and choose the best action with probability ϵ. If one wanted to decrease exploration with time, one could decay $1 - \epsilon$. The rate at which $1 - \epsilon$ was decayed would determine the rate at which exploration was decreased. We initialize the value of ϵ to be $Init_{Explore}$, and then set ϵ to be $Final_{Explore}$ after $Num_{Rounds} - 100$ rounds have passed. For all of the Sarsa experiments, $Init_{Explore}$ was set to 1. This was to allow as much exploration as possible, since a reward of 1 was only given on the sequence 0 1 2 or x 1 2 where x can be any of the 12 chromatic notes. We can see the necessity for exploration since the probability for randomly choosing the sequence 0 1 2 is 1/1000. This was also the reason that such a high exploration rate (1) was used until the last 100 rounds. The final exploration rate $Final_{Explore}$ was set to 0.1 for the linear Sarsa experiments but then changed to 0.4 for the non-linear Sarsa experiments to slightly increase the probability of seeing 0 1 2 randomly. We were interested in checking that the Q-values were being updated reasonably, so we chose these values for $Init_{Explore}$ and $Final_{Explore}$ for our initial experiments. Due to lack of time, other, perhaps more effective exploration parameters were not tried. Further experimentation would increase the number of rounds at a lower exploration rate, so that good actions would be chosen more frequently, and thus perhaps good sequences would be chosen more frequently which would result in the Q-values for those good actions being updated more often.

Sarsa(λ) is known as "an on-policy TD control method" [SB98], p.179, since as mentioned, it may use a bad action from ϵ-greedy and then use the bad action to update the weights. This is in contrast to Q-learning. Q-learning, although similar to Sarsa, updates the weights using the best action although it may use a bad action from ϵ-greedy. Since in Q-learning the actual policy learned is not (necessarily) the one used, then in order to choose notes using the learned policy, one would have to wait until the policy was learned, and then generate notes using the learned policy. In Sarsa(λ), notes can be chosen in real-time since the policy being learned is the one that is actually used to choose the notes.

4.3 Non-linear Sarsa(λ)

We chose to implement non-linear Sarsa(λ) by actually computing the predicted value for each of the 12 possible notes/states, and then choosing to play the note with the highest predicted value with some probability. Therefore, using a modified version of the Sarsa(λ) algorithm found in [SB98] our algorithm is then,

1 $\vec{w} = random$, $\vec{e} = 0$

2 For each episode

$s_t = s_0$, $a_t = a_0$
For each state in episode
    $\vec{e}_{z_j} = \lambda e_{z_{ji}} + w_j h_j(\vec{s}, \vec{z}_j)(1 - h_j(\vec{s}, \vec{z}_j))s_i$
    $\vec{e}_{w_j} = \lambda e_{w_j} + h_j(\vec{s}, \vec{z}_j)$
    Get r using a
    $s_{t+1} = a_t$
    Compute Q-values for each action using network
    $a_{t+1} = argmax_a Q_a$
    With probability ϵ, $a_{t+1} =$ random $a_i \in A(s_t)$
    $\delta = r + \gamma Q_a(s_{t+1}) - Q_a(s_t)$
    $\vec{z}_j = \vec{z}_j + \alpha\delta\vec{e}_{z_j}$
    $\vec{w} = \vec{w} + \alpha\delta\vec{e}_w$
    $a_t = a_{t+1}$, $s_t = s_{t+1}$
$\vec{e} = 0$
Continue until goal state is reached

The primary difference here between linear Sarsa(λ) and non-linear Sarsa(λ) is that the Q-values in the linear version are calculated using $Q_a = \sum_{i=0}^{n} w_i s_i$, while in the non-linear version the Q-values are calculated using the non-linear network.

4.4 Goals

Our main goal for using Sarsa(λ) was to have an agent choose to play certain good notes in the correct order. The notes and order should correspond to the types of sequences for which it received a reward of 1. Given the results of the TD(λ) experiments, we wanted to see to what extent the agent would be capable of choosing good actions that would correspond to notes in the sequence for which the agent received a reward of 1.

Chapter 5

Reinforcement Learning Experiments

5.1 Introduction

As with TD(λ), using Sarsa(λ) and a linear neural network allows a linear function to be approximated while using Sarsa(λ) and a non-linear network allows a non-linear function to approximated. Sarsa(λ) then bases the action it chooses to take on the function it has learned. Two sets of Sarsa(λ) experiments were run: linear experiments and non-linear experiments. Relevant abbreviations are given in Table 5.1.

Parameter	Abbreviation
Inital Exploration Rate	$Init_{Explore}$
Final Exploration Rate	$Final_{Explore}$

Table 5.1: Sarsa(λ) specific abbreviations of different parameters.

The Sarsa(λ) ordered sequence experiments correspond to the TD(λ) ordered sequence experiments found in Chapter 3. In Sarsa(λ), however, the sequence that is played is "composed" by the program, rather than given. Two rules were used to give a reward. As with TD(λ), a reward of 1 was given on the last note of the sequence if the sequence was 0 1 2. This is Rule #1. In Rule #2, a reward of 1 was given on the last note of the sequence if the sequence was x 1 2, where x can be any of the 12 possible notes. The primary reason for Rule #2 was to increase the probability of randomly seeing the sequence x 1 2, and thereby receiving a reward.

5.2 Linear Experiments

5.2.1 Ordered Sequence

- Results using Rule #1 show that linear Sarsa(λ) learns that notes 0 1 and 2 are good, but that note 0 is the best. Thus note 0 is the note that is always played. See Table 5.2. Results using Rule #2 show that linear Sarsa(λ) again learns that notes 0 1 and 2 are good, but that note 2 is the best. Linear Sarsa(λ) again therefore always plays note 2. See Table 5.3. As with the linear TD(λ) ordered sequence experiments, we chose only to use the average trace with the non-linear rather than the linear experiments since we felt the non-linear experiments to be more interesting.

- The lower Q-values when using Rule #2 may be a result of the ambiguity of the rule: a reward is given on sequences x 1 2 where x can be any note. We hypothesize that the linear network has difficulty coping with this ambiguity, hence the lower Q-values, and hence why the linear network is only able to learn that note 2 is good. While with Rule #1 the network is able to propagate the reward back to note 0, the network is unable to propagate the reward back to note 1 with Rule #2; this may again be due to the ambiguity of Rule #2.

- Parameters were: $n_{Rounds} = 10000000$, $\alpha = 0.01$, $\lambda = 0.6$, and $\gamma = 0.95$, $Init_{Explore} = 1$, $Final_{Explore} = 0.1$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Rule #1		
Q-values for First Note	Q-Values for Second Note	Q-values for Third Note
$Q_0(0) = 4.10561$	$Q_1(0) = 4.10212$	$Q_2(0) = 4.09545$
$Q_0(1) = 2.28904$	$Q_1(1) = 2.28904$	$Q_2(1) = 2.28904$
$Q_0(2) = 2.38758$	$Q_1(2) = 2.38758$	$Q_2(2) = 2.38758$
$Q_0(3) = 1.95384$	$Q_1(3) = 1.95384$	$Q_2(3) = 1.95384$
$Q_0(4) = 1.97974$	$Q_1(4) = 1.97974$	$Q_2(4) = 1.97974$
$Q_0(5) = 2.02906$	$Q_1(5) = 2.02906$	$Q_2(5) = 2.02906$
$Q_0(6) = 1.99363$	$Q_1(6) = 1.99363$	$Q_2(6) = 1.99363$
$Q_0(7) = 1.95338$	$Q_1(7) = 1.95338$	$Q_2(7) = 1.95338$
$Q_0(8) = 2.02421$	$Q_1(8) = 2.02421$	$Q_2(8) = 2.02421$
$Q_0(9) = 2.00021$	$Q_1(9) = 2.00021$	$Q_2(9) = 2.00021$
$Q_0(10) = 1.98155$	$Q_1(10) = 1.98155$	$Q_2(10) = 1.98155$
$Q_0(11) = 2.02111$	$Q_1(11) = 2.02111$	$Q_2(11) = 2.02111$
Note Played: $a(0) = 0$	Note Played: $a(1) = 0$	Note Played: $a(2) = 0$

Table 5.2: Output for the linear ordered sequence experiments for Sarsa(λ) using Rule #1.

Rule #2		
Q-values for First Note	Q-Values for Second Note	Q-values for Third Note
$Q_0(0) = 0.0637261$	$Q_1(0) = 0.0637261$	$Q_2(0) = 0.0637261$
$Q_0(1) = 0.0875549$	$Q_1(1) = 0.0875549$	$Q_2(1) = 0.0875549$
$Q_0(2) = 0.0950434$	$Q_1(2) = 0.0949445$	$Q_2(2) = 0.0947894$
$Q_0(3) = 0.0615986$	$Q_1(3) = 0.0615986$	$Q_2(3) = 0.0615986$
$Q_0(4) = 0.0585855$	$Q_1(4) = 0.0585855$	$Q_2(4) = 0.0585855$
$Q_0(5) = 0.0671858$	$Q_1(5) = 0.0671858$	$Q_2(5) = 0.0671858$
$Q_0(6) = 0.0597489$	$Q_1(6) = 0.0597489$	$Q_2(6) = 0.0597489$
$Q_0(7) = 0.0576425$	$Q_1(7) = 0.0576425$	$Q_2(7) = 0.0576425$
$Q_0(8) = 0.0668698$	$Q_1(8) = 0.0668698$	$Q_2(8) = 0.0668698$
$Q_0(9) = 0.0582388$	$Q_1(9) = 0.0582388$	$Q_2(9) = 0.0582388$
$Q_0(10) = 0.0594308$	$Q_1(10) = 0.0594308$	$Q_2(10) = 0.0594308$
$Q_0(11) = 0.0584055$	$Q_1(11) = 0.0584055$	$Q_2(11) = 0.0584055$
Note Played: $a(0) = 2$	Note Played: $a(1) = 2$	Note Played: $a(2) = 2$

Table 5.3: Output for the linear ordered sequence experiments for Sarsa(λ) using Rule #2.

5.3 Non-linear Experiments

5.3.1 Ordered Sequence

1. **Non-Linear Model** $\vec{s}_{t\,Note}$

   - In this model the current state is the input to the network that is used to calculate a prediction for each note and to choose the best note.
   - We can see from the output in Table 5.4 using Rule #1 that the network eventually learned to always play note 2. Note 0 also has a high Q-value, although not quite as high as note 2. Note 1, somewhat surprisingly, has a very low Q-value. Since note 2 always has the highest Q-value, regardless of the sequence position for the note currently being chosen, then it is not surprising that note 2 is the note that is always played (excluding random explorations). One possible explanation for the low Q-value for note 1 is that while note 0 in the zeroth position of the sequence is always good, note 1 in

the first postion of the sequence is good only if preceded by note 0. While note 2 is only good when preceded by notes 0 1, note 2 in the second position of the sequence also receives the full "effects" of the reward = 1, as compared with note 1 in the first position, thereby explaining the high Q-value for note 2.

- From the output in Table 5.5 using Rule #2, we can see that the non-linear network is better able to deal with the ambiguity of Rule #2; we see this with how it learns to always play note 1. In this case it makes sense that note 1 is good, unlike with Rule #1, since here, regardless of what precedes note 1, note 1 in the first position of the sequence is good. Because the first position and the second position of the sequence are closer together temporally than the zeroth position and the second position, the effects of the reward = 1 are less diminished for notes in the first position of the sequence as compared with notes in the zeroth position. Curiously, note 2 in the second position of the sequence has a low Q-value: the network has learned to propagate the reward back to the earliest predictor of the reward (note 1 in the first position of the sequence), and any following notes are low.

- Parameters were: $n_{Rounds} = 10000000$, $\alpha = 0.01$, $\lambda = 0.6$, $n_{Hid} = 60$, and $\gamma = 0.95$, $Init_{Explore} = 1$, $Final_{Explore} = 0.4$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Rule #1		
Q-values for First Note	Q-Values for Second Note	Q-values for Third Note
$Q_0(0) = 0.0427394$	$Q_1(0) = 0.0432572$	$Q_2(0) = 0.0433709$
$Q_0(1) = 0.0373587$	$Q_1(1) = 0.0378763$	$Q_2(1) = 0.0379899$
$Q_0(2) = 0.0441654$	$Q_1(2) = 0.0446831$	$Q_2(2) = 0.0447967$
$Q_0(3) = 0.0374412$	$Q_1(3) = 0.0379589$	$Q_2(3) = 0.0380725$
$Q_0(4) = 0.0374203$	$Q_1(4) = 0.0379382$	$Q_2(4) = 0.0380518$
$Q_0(5) = 0.0373167$	$Q_1(5) = 0.0378344$	$Q_2(5) = 0.037948$
$Q_0(6) = 0.037377$	$Q_1(6) = 0.0378948$	$Q_2(6) = 0.0380084$
$Q_0(7) = 0.0374188$	$Q_1(7) = 0.0379367$	$Q_2(7) = 0.0380503$
$Q_0(8) = 0.0373568$	$Q_1(8) = 0.0378748$	$Q_2(8) = 0.0379885$
$Q_0(9) = 0.037351$	$Q_1(9) = 0.0378685$	$Q_2(9) = 0.0379821$
$Q_0(10) = 0.0373994$	$Q_1(10) = 0.0379173$	$Q_2(10) = 0.038031$
$Q_0(11) = 0.0373375$	$Q_1(11) = 0.0378551$	$Q_2(11) = 0.0379686$
Note Played: $a(0) = 2$	Note Played: $a(1) = 2$	Note Played: $a(2) = 2$

Table 5.4: Output for the non-linear ordered sequence experiments for Sarsa(λ) using the state as input and using Rule #1.

2. **Non-Linear Model** $a(\vec{s}_{tNote})$

   - In this model the average trace of the current and preceding states is the input to the network that used to calculate a prediction for each note and to choose the best note.
   - We can see from the output in Table 5.6 using Rule #1 that the network eventually learned to always play note 2. The results are similar to the results for the state as the input. The Q-values are

Rule #2		
Q-values for First Note	Q-Values for Second Note	Q-values for Third Note
$Q_0(0) = 2.30339$	$Q_1(0) = 2.4098$	$Q_2(0) = 2.67439$
$Q_0(1) = 2.997$	$Q_1(1) = 3.13201$	$Q_2(1) = 3.43382$
$Q_0(2) = 2.2989$	$Q_1(2) = 2.40541$	$Q_2(2) = 2.67025$
$Q_0(3) = 2.31794$	$Q_1(3) = 2.42485$	$Q_2(3) = 2.69073$
$Q_0(4) = 2.2869$	$Q_1(4) = 2.39366$	$Q_2(4) = 2.6929$
$Q_0(5) = 2.21566$	$Q_1(5) = 2.3221$	$Q_2(5) = 2.58678$
$Q_0(6) = 2.23101$	$Q_1(6) = 2.33786$	$Q_2(6) = 2.60356$
$Q_0(7) = 2.38723$	$Q_1(7) = 2.49416$	$Q_2(7) = 2.76009$
$Q_0(8) = 2.18632$	$Q_1(8) = 2.2926$	$Q_2(8) = 2.55689$
$Q_0(9) = 2.33267$	$Q_1(9) = 2.43958$	$Q_2(9) = 2.70546$
$Q_0(10) = 2.30393$	$Q_1(10) = 2.41049$	$Q_2(10) = 2.67546$
$Q_0(11) = 2.24211$	$Q_1(11) = 2.34875$	$Q_2(11) = 2.61395$
Note Played: $a(0) = 4$ (Note: This was assumed to be a random exploration)	Note Played: $a(1) = 1$	Note Played: $a(2) = 1$

Table 5.5: Output for the non-linear ordered sequence experiments for Sarsa(λ) using the state as input and using Rule #2.

somewhat lower, probably since these experiments were run only for one million rounds, versus ten million rounds for the experiments with the state as input (this was due to time constraints).

- Again, From the output in Table 5.7 using Rule #2, we can see that the network eventually learned to always play note 1, similarly to the experiments with the state as the input.
- If we compare the the experiments using the state as the input with those using the average trace as the input, we see that there isn't much difference. They learn the same things. Since the average trace does contain more information, it would follow that under the right circumstances, the network would learn more given the average trace as input as compared with just the state. Therefore, further experiments would involve exploring the average trace with different parameters: more/fewer hidden nodes, higher/lower β, higher/lower α, and an increased number of rounds.
- Parameters were: $n_{Rounds} = 1000000$, $\alpha = 0.01$, $\lambda = 0.6$, $n_{Hid} = 60$, and $\gamma = 0.95$, $Init_{Explore} = 1$, $Final_{Explore} = 0.4$. Finally, the weights were initialized to values ranging in absolute value between 0.01 and 0.001.

Rule #1		
Q-values for First Note	Q-Values for Second Note	Q-values for Third Note
$Q_0(0) = 0.0707609$	$Q_1(0) = 0.0701092$	$Q_2(0) = 0.0693305$
$Q_0(1) = 0.0703177$	$Q_1(1) = 0.069666$	$Q_2(1) = 0.0688872$
$Q_0(2) = 0.0712483$	$Q_1(2) = 0.0705968$	$Q_2(2) = 0.0698183$
$Q_0(3) = 0.0701916$	$Q_1(3) = 0.0695398$	$Q_2(3) = 0.068761$
$Q_0(4) = 0.0702693$	$Q_1(4) = 0.0696173$	$Q_2(4) = 0.0688382$
$Q_0(5) = 0.070288$	$Q_1(5) = 0.0696363$	$Q_2(5) = 0.0688575$
$Q_0(6) = 0.0705109$	$Q_1(6) = 0.069859$	$Q_2(6) = 0.0690799$
$Q_0(7) = 0.0702393$	$Q_1(7) = 0.0695873$	$Q_2(7) = 0.0688081$
$Q_0(8) = 0.0702255$	$Q_1(8) = 0.0695733$	$Q_2(8) = 0.068794$
$Q_0(9) = 0.0702155$	$Q_1(9) = 0.0695639$	$Q_2(9) = 0.0687853$
$Q_0(10) = 0.0701178$	$Q_1(10) = 0.0694657$	$Q_2(10) = 0.0686866$
$Q_0(11) = 0.0702339$	$Q_1(11) = 0.0695824$	$Q_2(11) = 0.0688038$
Note Played: $a(0) = 1$ (Note: This was assumed to be a random exploration)	Note Played: $a(1) = 2$	Note Played: $a(2) = 2$

Table 5.6: Output for the non-linear ordered sequence experiments for Sarsa(λ) using the average trace of the current and preceding states in the sequence as input and using Rule #1.

Rule #2		
Q-values for First Note	Q-Values for Second Note	Q-values for Third Note
$Q_0(0) = 4.0437$	$Q_1(0) = 4.0144$	$Q_2(0) = 4.26295$
$Q_0(1) = 4.15844$	$Q_1(1) = 4.1289$	$Q_2(1) = 4.37994$
$Q_0(2) = 4.04393$	$Q_1(2) = 4.01463$	$Q_2(2) = 4.26315$
$Q_0(3) = 4.04434$	$Q_1(3) = 4.01503$	$Q_2(3) = 4.26364$
$Q_0(4) = 4.04696$	$Q_1(4) = 4.01765$	$Q_2(4) = 4.26634$
$Q_0(5) = 4.04403$	$Q_1(5) = 4.01473$	$Q_2(5) = 4.26332$
$Q_0(6) = 4.0463$	$Q_1(6) = 4.01698$	$Q_2(6) = 4.26567$
$Q_0(7) = 4.04495$	$Q_1(7) = 4.01564$	$Q_2(7) = 4.26432$
$Q_0(8) = 4.04702$	$Q_1(8) = 4.01769$	$Q_2(8) = 4.26646$
$Q_0(9) = 4.04439$	$Q_1(9) = 4.01509$	$Q_2(9) = 4.26364$
$Q_0(10) = 4.04679$	$Q_1(10) = 4.01747$	$Q_2(10) = 4.26621$
$Q_0(11) = 4.04375$	$Q_1(11) = 4.01445$	$Q_2(11) = 4.26298$
Note Played: $a(0) = 1$	Note Played: $a(1) = 1$	Note Played: $a(2) = 1$

Table 5.7: Output for the non-linear ordered sequence experiments for Sarsa(λ) using the average trace of the current and preceding states in the sequence as input and using Rule #2.

Chapter 6

Dopamine's Role in Learning

6.1 Introduction

Dopamine (DA) neurons in the nigrostriatal tract have shown learning capabilities [BR98], [Sch98], [LAS92]. Although these neurons are often classified as controllers of movement [Chi97], it has been found that the DA neurons in this pathway are activated by an unpredicted reward [LAS92], are unaffected by a predicted reward [LAS92], and are inhibited by the absence of a predicted reward (Ljungberg, 1991, as cited in [Sch98]). Thus, these DA neurons have learned to associate a conditioned stimulus with a subsequent reward, thereby enabling them to distinguish among unpredicted, predicted, and absent predicted rewards (Ljungberg, 1991, as cited in [Sch98]), [LAS92]. Several models have been presented to explain how DA neurons in the basal ganglia acquire this capability. While the underlying neuroanatomical and neurophysiological framework for each model is the same, the imposed hypothetical explanations for the observed dopamine neuron behavior differ. Two of these models will be examined in depth. The first model is attributed to Schultz [Sch98] and uses the computational model of temporal-difference learning by [Sut88] taken from computer science which was examined in Chapters 2 and 3. The second model is presented by Brown, Bullock, and Grossberg [BBG99] and challenges one particular aspect of the TD model. Background information about DA neurons, receptors, pathways, and behavior will first be presented, before focusing on the two models in detail.

6.2 Dopamine Anatomy and Receptors

Dopamine is a catecholamine neurotransmitter produced in the brain by dopamine neurons. In order for this production to occur, the amino acid tyrosine must first cross the blood-brain barrier. Then the enzyme tyrosine hydroxylase converts the tyrosine to L-hydroxyphenlalanine (also known as L-DOPA). Finally the amino acid decarboxylase converts L-DOPA into dopamine [CBR96].

There are three major dopamine pathways: the mesolimbic pathway from the ventral tegmental area to the ventral striatum, the mesostriatal pathway from the substantia nigra pars compacta to the striatum, and the mesocortical pathway from the ventral tegmental area to the frontal cortex. Generally, the mesolimbic pathway regulates reward, the mesostriatal pathway regulates movement, and the mesocortical pathway plays a role in cognition [Chi97].

The pathway the most is known about is the mesostriatal or nigrostriatal pathway; hence it is therefore used as a basis for the Schultz model [Sch98] and is also used by Brown *et al* [BBG99]. The nigrostriatal pathway is located within the basal ganglia since both the substantia nigra and striatum are components of it. The basal ganglia is comprised of input nuclei, output nuclei, and intrinsic nuclei. The input nuclei, also known as the striatum, consist of the caudate nucleus, putamen, and nucleus accumbens. The output nuclei consist of the globus pallidus internal segment, substantia nigra pars reticulata, and ventral pallidum. The intrinsic nuclei consist of the globus pallidus external segment, substantia nigra pars compacta, ventral tegmental area, and subthalamic nucleus [Mar96]. Figure 6.1 shows the components of the basal ganglia, their connections, and the neurotransmitters involved.

The striatum is then divided into the striatal matrix and the striosomes.

The distinguishing feature of the striosomes is a different chemical and protein composition than the striatal matrix. The ventral portion of the substantia nigra pars compacta along with the substantia nigra pars reticulata innervate the striosomes. The dorsal portion of the substantia nigra pars compacta along with the ventral tegmental area and retrorubral field innervate the striatal matrix [FMQ97].

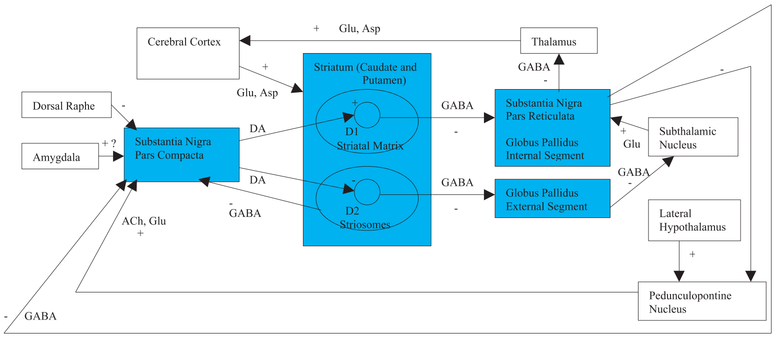

Figure 6.1: Nigrostriatal tract and connections. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

The nigrostriatal tract is the primary path of interest. In this pathway, there are dopaminergic projections from the substantia nigra to the striatum and there are projections from the striatum to the substantia nigra. There are GABAergic projections from the striosomes in the striatum to dopaminergic neurons in the substantia nigra pars compacta and to the globus pallidus. There are also GABAergic projections from the striatal matrix of the striatum to non-dopaminergic neurons in the substantia nigra pars reticulata. There are then projections from the non-dopaminergic neurons in the substantia nigra pars reticulata to dopaminergic neurons in the substantia nigra pars compacta. Finally, there are GABAergic projections from the ventral striatum to the substantia nigra pars compacta, substantia nigra pars reticulata, and the ventral tegmental area (see [Sch98]).

Since GABA is an inhibitory neurotransmitter, the GABAergic projections from the striosomes and ventral striatum will inhibit the dopamine neurons in the substantia nigra pars compacta. The GABAergic projections from the striatal matrix will inhibit non-dopaminergic neurons in the substantia nigra pars reticulata and will therefore indirectly excite the dopaminergic neurons in the substantia nigra pars compacta [Sch98].

The dopaminergic neurons in the substantia nigra pars compacta also receive input from other areas: the pedunculopontine nucleus gives excitatory input (acetylcholine, glutamate), the amygdala most likely provides excitatory input, and the dorsal raphe gives inhibitory input [Sch98].

There are two main divisions of dopamine receptors, D1 and D2, and these receptors are located in a variety of areas in the nervous system. Mentioned here are the important features of interest here for each type. In general dopamine exerts its effects after binding to the dopamine receptor through the use of G-proteins [Chi97].

Among other places, dopamine receptors of type D1 can be found postsynaptically on the dopamine receiving neurons in the striatum, nucleus accumbens, and the substantia nigra [CBR96] that synapse on neurons in the internal globus pallidum and substantia nigra pars reticulata (see [Sch98]). These receptors increase the likelihood of dopamine release in response to stimuli [CBR96]. D1 make up approximately 80% of the receptors and are adenylate cyclase-activating, with approximately 80% low-affinity and 20% high-affinity (Richfield *et al*, 1989, as cited in [Sch98]). Dopamine receptors of type D2 can be found presynaptically on the neurons releasing dopamine onto neurons in the striatum and nucleus accumbens [CBR96] which then synapse on neurons in the external globus pallidum (see [Sch98]). D2 receptors are autoreceptors and consequently they monitor the production, release, and reuptake of dopamine at the synapse [CBR96]. D2 receptors found on the dendrites of dopamine neurons in the substantia nigra and ventral tegmental area have an inhibitory effect when stimulated [CBR96]. D2 receptors found on interneurons in the striatum that release acetylcholine serve to inhibit this release [CBR96]. D2 make up approximately 20% of the receptors and are adenylate cyclase-inhibiting, with approximately 10% low-affinity, 90% high-affinity (Richfield *et al*, 1989, as cited in [Sch98]).

Comparing the affinity for each receptor type for dopamine, it is seen that D2 receptors have a greater affinity for dopamine than do D1 receptors. Because a bursting dopamine neuron increases so dramatically the dopamine concentration in the striatum (due to the dopamine reuptake transporter reaching its maximum capacity), activation of both D1 and D2 receptors would occur, with nearby D1 receptors activated and all D2 receptors activated [Sch98].

When stimulated, D1 receptors increase the depolarization of an already excited membrane and increase the hyperpolarization of an already inhibited membrane (see [Sch98]). D2 receptors, conversely, regardless of the membrane potential will increase the hyperpolarization of the membrane (Hernandez-Lopez *al*, 1997, as cited in [Sch98]). We know that D1 receptors are activated when there is a reward or a stimulus that predicts a reward. This means that information concerning a reward or a stimulus that predicts a reward will get propagated when either a D1 or D2 receptor is stimulated [Sch98].

There are a variety of ways to affect the plasticity of the dopamine synapse, where plasticity refers to changes at the synapse that affect the liklihood of the dopamine neuron firing [Sch98]. For example, the amount of neurotransmitter released at the dopamine synapse can be increased through the use of magnesium or GABA antagonists (see [Sch98]). Synaptic plasticity of the dopamine synapse is not limited to dopamine neurons projecting from the substantia nigra; for example, dopamine neurons synapsing in the hippocampus also show synaptic plasticity [Sch98].

6.3 Dopamine Neuron Behavior

A dopamine neuron responds to a reward or a predicted reward by being activated. Activation means that depolarization of the neuron has occurred and dopamine is released onto neurons in the striatum. The dopamine neurons in the nigrostriatal pathway, however, tonically release dopamine all the time [Sch98]. This means that there is a little dopamine always being released so activation causes dopamine neurons to release more dopamine than norrmal, rather than releasing the usual amount.

Certain conditions are necessary for a dopamine neuron to respond to the reward. First, the reward must not be generally predictable. For instance, the reward must not occur every sixty seconds. Rather the reward must be unpredictable. Second, the predicted reward must not be predictable either. This may seem contradictory, but it simply means that the neuron will activate if it receives some unexpected input and based on that input knows that a reward will follow. If the neuron receives some cyclically occurring or expected input, it will not activate even if that input predicts a reward [BMPM01], [LAS92]. The dopamine neuron's membrane potential remains unaffected when it receives the reward corresponding to some conditioned stimulus although the predicted reward is still received [LAS92]. Perhaps this lack of response with cyclic input is due to the conditioned stimulus becoming predicted or expected by the preceding conditioned stimulus and reward, and so forth. Dopamine neurons are inhibited, that is hyperpolarized when a predicted reward is absent following the stimulus that predicted it [Sch98]. When inhibited, for example by an inhibitory neurotransmitter such as GABA, dopamine neurons will release less dopamine than normally released tonically. Table 6.1 summarizes the possible dopamine responses.

Event	Consequence
1. Dopamine neuron receives a reward	1. Activation occurs
2a. Dopamine neuron receives a stimulus that predicts a reward 2b. Dopamine neuron then receives the predicted reward	2a. Activation occurs 2b. No activation occurs
3a. Dopamine neuron receives a stimulus that predicts a reward 3b. Dopamine neuron then does not receive the predicted reward	3a. Activation occurs 3b. Inhibition occurs
4. Dopamine neuron receives a cyclic reward	4. No activation occurs
5. Dopamine neuron receives a cyclic stimulus that predicts a reward	5. No activation occurs

Table 6.1: The possible events and subsequent responses of dopamine neurons. Source: [Sch98].

[Sch99], p. 249, says that "Recent experiments have shown that dopamine neurons code the discrepancy between the prediction and occurrence of rewards and in this way a crucial learning term for approach behavior [Sch99], p. 249." This is the behavior exhibited by dopamine neurons in Table 6.1. However, the mechanism by which dopamine neurons learn to do this must still be determined. [Sch98] suggests the temporal-difference algorithm which explicitly uses the difference in prediction of rewards while [BBG99] suggest that the temporal-difference algorithm may not be biologically feasible and propose another model. The dopamine models presented by Schultz [Sch98] and Brown *et al* [BBG99] will now be presented, and will be compared on the basis of their explanations of the phenomena presented in Table 6.1.

6.4 Schultz [Sch98] Model

Dopamine neurons in the nigrostriatal tract behave similarly to the TD algorithm when responding to rewards and learning to predict rewards. Just as the TD algorithm is part of the larger framework of reinforcement learning so too are nigrostriatal dopamine neurons part of a larger framework centered around the basal ganglia. In short, "The critic simulates dopamine neurons, the reward prediction enters from striosomal striatonigral projections, and the actor resembles striatal matrix neurons with dopamine-dependent plasticity [Sch98], p.26." In TD learning a reward is a number, generally 1, while 0 represents an absence of reward. With dopamine neurons, rewards can be an appetitive stimuli such as food or water. An absence of the appetitive stimuli represents an absence of the reward. Just as in TD learning where the network learns to predict a future reward, dopamine neurons also learn to predict future rewards. This means that dopamine neurons go from responding to a reward to responding to a stimulus that predicts the reward [SB98], [Sut88], [Sch98].

Similar to the TD algorithm, the dopamine neurons learn to predict reward effectively and respond to the predicted reward, rather than responding to the reward [Sch98]. The TD algorithm though, would predict highly for both the predicted reward (stimulus) and the reward and responds to both [Sut88], [SB98].

Dopamine neurons therefore respond to reward and to stimuli that predict a reward in the following manner [Sch98].

$$Response = Actual - Predicted$$

Actual refers to whether the reward or stimulus actually occured and *Predicted* refers to whether a reward or stimulus was predicted to occur.

The inputs to the dopamine neurons in the substantia nigra pars compacta were presented earlier. These inputs must now be related to the dopamine neuron response to rewards and predicted rewards [Sch98].

Returning to the five cases in Table 6.1 again, Schultz's [Sch98] proposal for how the dopamine neurons in the striatonigral tract learn to predict reward is the following.

1. Dopamine neuron receives a reward.

    - Activation occurs due to *i*) indirect excitation as a result of GABAergic projections from the striatal matrix inhibiting non-dopaminergic neurons in the substantia nigra pars reticulata, *ii*) direct excitation from the pedunculopontine nucleus, and *iii*) direct excitation from the amygdala. As a result of this excitation, more dopamine is released than tonically resulting in nearby D1 receptors being activated and all D2 receptors being activated. This means that neurons from the striatum to the internal globus pallidum are activated (D1); neurons from the striatum to substantia nigra pars reticulata are activated (D1); and neurons from the striatum to the external globus pallidum are activated (D2).

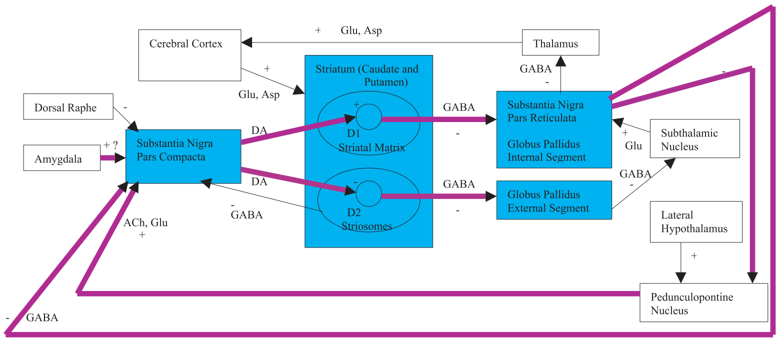

Figure 6.2: Excitation of dopamine neurons at time of reward in Schultz [Sch98] dopamine model. Purple highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

2. a. Dopamine neuron receives a stimulus that predicts a reward

   b. Dopamine neuron then receives the predicted reward.

   - a. As in 1, activation occurs due to *i*) indirect excitation as a result of GABAergic projections from the striatal matrix inhibiting non-dopaminergic neurons in the substantia nigra pars reticulata, *ii*) direct excitation from the pedunculopontine nucleus, and *iii*) direct excitation from the amygdala. As a result of this excitation, more dopamine is released than tonically resulting in nearby D1 receptors being activated and all D2 receptors being activated. This means that neurons from the striatum to the internal globus pallidum are activated (D1); neurons from the striatum to substantia nigra pars reticulata are activated (D1); and neurons from the striatum to the external globus pallidum are activated (D2).

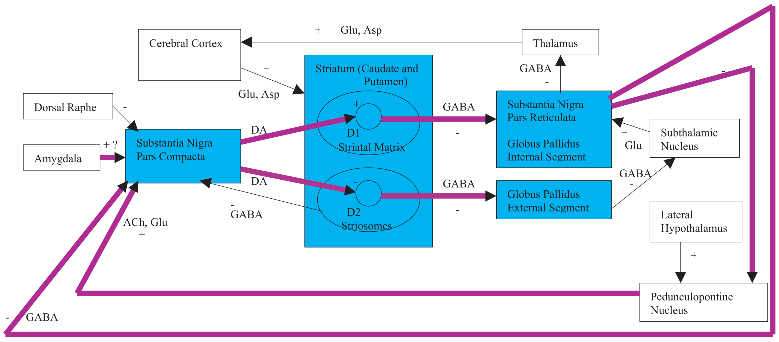

Figure 6.3: Excitation of dopamine neurons at time of conditioned stimulus in Schultz [Sch98] dopamine model. Purple highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

- b. No activation occurs due to pars reticulata and amygdala excitation being balanced by direct inhibition from the GABAergic projections from the striosomes and ventral striatum.

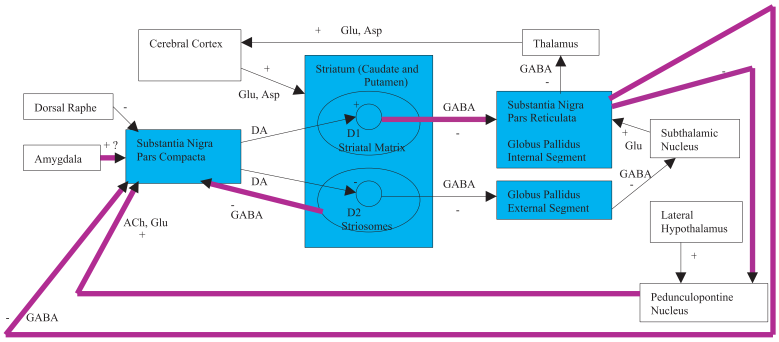

Figure 6.4: No activation in dopamine neurons at time of predicted reward in Schultz [Sch98] dopamine model. Purple highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [BBG99], [Sch98], [Mar96], [AHBR99a], [AHBR99b], [AHBR99c].

3. a. Dopamine neuron receives a stimulus that predicts a reward
   b. Dopamine neuron then does not receive the predicted reward

   - a. Again as in 1, activation occurs due to *i*) indirect excitation as a result of GABAergic projections from the striatal matrix inhibiting non-dopaminergic neurons in the substantia nigra pars reticulata, *ii*) direct excitation from the pedunculopontine nucleus, and *iii*) direct excitation from the amygdala. As a result of this excitation, more dopamine is released than tonically resulting in nearby D1 receptors being activated and all D2 receptors being activated. This means that neurons from the striatum to the internal globus pallidum are activated (D1); neurons from the striatum to substantia nigra pars reticulata are activated (D1); and neurons from the striatum to the external globus pallidum are activated (D2).

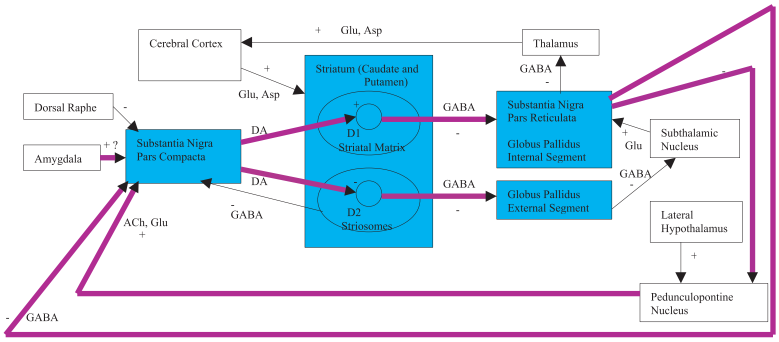

Figure 6.5: Excitation of dopamine neurons at time of conditioned stimulus in Schultz [Sch98] dopamine model. Purple highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

- b. Inhibition occurs due to direct inhibition as a result of GABAergic projections from the striosomes and ventral striatum. Less dopamine is released than tonically. Fewer D2 receptors are activated and some neurons from the striatum to the external globus pallidum are inhibited (D2).

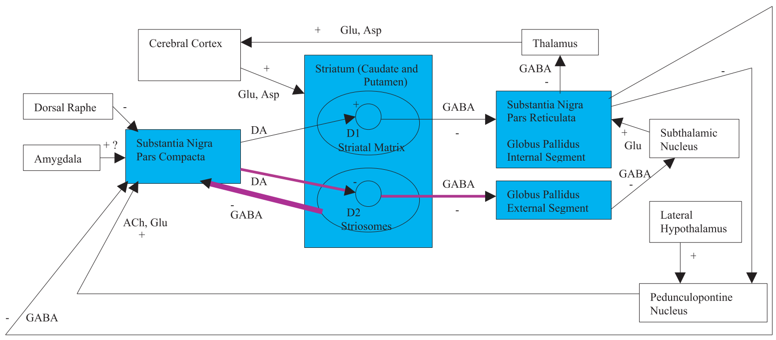

Figure 6.6: Inhibition at time of missing predicted reward in Schultz [Sch98] dopamine model. Purple highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

4. Dopamine neuron receives a cyclic reward.

    - No activation occurs due to some combination of the excitatory and inhibitory inputs previously mentioned. The same amount of dopamine is released as tonically. D2 receptors are activated and so are some neurons from the striatum to the external globus pallidum (D2). This situation is similar to 2b.

5. Dopamine neuron recieves a cyclic stimulus that predicts a reward.

    - Similarly as in 4, no activation occurs due to some combination of the excitatory and inhibitory inputs previously mentioned. The same amount of dopamine is released as tonically. D2 receptors are activated and so are some neurons from the striatum to the external globus pallidum (D2). This situation may also be similar to 2b, but now it is applied to both the reward and the conditioned stimulus.

The dopamine learning "algorithm" is similar to the TD learning algorithm since first a reward is obtained, then that reward is used to modify the weights of the network, increasing those weights that most contributed to a positive reward (1) and decreasing those weights that most contributed to a negative reward (0). In the case of the dopamine neurons the terminology of weights being modified would refer to neuronal changes that would incresase or decrease the likelihood of the cell being activated, for example due to an increased number of receptors. Weight adjustments in artificial neural networks appear to be the equivalent of strengthening a synapse in a biological neural network. The terminology of weight adjustment used in the area of artificial neural networks appears to stem from Donald Hebb's (Hebb, 1949, as cited in [Pic00]) comment that,

> *"When an axon of cell A is near enough to excite a cell B and repeatedly or persistently takes place in firing it, some growth process or metabolic change takes place in one or both cells such that A's efficiency, as one of the cells firing B, is increased (Hebb, 1949, as cited in [Pic00], p. 15)."*

The area of biological neural networks, such as the basal ganglia neural network examined here, appear to now also use the terminology of weights and weight changes, but still referring to the same phenomenon as was described by Hebb.

The strengthening of the weights both biologically and in the temporal-difference learning algorithm may be attributable to the following. Striatal neurons receive excitatory (glutamate, aspartate) input from the cortex coding for different attributes of the reward, and as mentioned, striatal neurons also receive dopaminergic input from the substantia nigra pars compacta. When a striatal neuron receives some general dopamine reward it will also be getting some input from the cortex giving some information about the reward (similar to the eligibility traces in TD). Since dopamine has the effect of further inhibiting a hyperpolarized membrane and exciting a depolarized membrane then only the cortical input coding for the attributes for the current reward would have an excitatory effect, and dopamine would cause further excitation and "...would specifically strengthen the synaptic weights (w) of Hebbian-type corticostriatal synapses that are active at the time of the reward-predicting stimulus, whereas the inactive corticostriatal synapses are left unchanged [Sch98], p.28."

Since the synapses are being strengthened then this is synaptic plasticity. This serves as a history for what has happened to get the reward and is similar to the eligibility trace in TD. The synaptic plasticity or eligibility trace then allows for the reward to be propagated backwards in time where "Potential physiological substrates of eligibility traces consist in prolonged changes in calcium concentration (Wickens and Kötter, 1995, as cited in [Sch98]), formation of calmodulin-dependent protein kinase II (Hout *et al*, 1995, as cited in [Sch98]), or sustained neuronal activity found frequently in the striatum (Schultz *et al*, 1995, as cited in [Sch98]), [Sch98], p.30."

The key comparison between dopamine neuron learning and TD learning is the following. The *Response* error for the reward is similar to $r - V(s_t)$, since r is the estimated actual reward while $V(s_t)$ is the current approximation of reward. The *Response* error for the predicted reward (stimulus) is similar to $\gamma V(s_{t+1}) - V(s_t)$, since $\gamma V(s_{t+1})$ is the discounted future reward while $V(s_t)$ is the current estimate of that reward. In both cases $V(s_t)$ represents the current estimate or prediction of both current and future reward while r is the actual current reward and $\gamma V(s_{t+1})$ is the predicted future reward. The general, global reward of the dopamine is similar to the general, global reward given by r, since neither specifies the specific action or event that is being rewarded rather just that a reward has occurred and it is up to the neurons or the network to determine how best to interpret and distribute the reward [Sch98]

The learning rule for the dopamine neurons is [Sch98],

$$\Delta w = \epsilon \hat{r} \; Input \; Output$$

This is similar to the one used for updating TD weights for the linear network [SB98],

$$\Delta w = \alpha \delta e(s)$$

Except now in [Sch98],

$$\epsilon = \alpha = \textit{Learning Rate}$$

$$\hat{r} = \delta = \textit{Error} = r + \gamma V(s_{t+1}) = V(s_t)$$

$$\textit{Input Output} = \textit{Eligibility} = \textit{Some History of What Just Happened}$$

This then is how the biological model of dopamine learning is unified with the computer science model temporal-difference-learning.

6.5 Brown, Bullock and Grossberg [BBG99] Model

Brown *et al* [BBG99] suggest that the excitatory limbic cortex - ventral striatum - pedunculopontine nucleus pathway serves to excite the dopamine neurons in the substantia nigra pars compacta at the time of the conditioned stimulus, that the lateral hypothalamus - pedunculopontine nucleus pathway serves to excite the dopamine neurons in the substantia nigra pars compacta when there is an unpredicted reward, and that the inhibitory limbic cortex - striosomes - substantia nigra pars compacta pathway provides inhibition at the time of a predicted reward, once the conditioned stimulus has been learned.

Note, that although it will not be discussed here, the Brown *et al* [BBG99] model also involves memory. From a computer science perspective, the role of memory in the [BBG99] model would support the necessity of recurrence in using artificial neural networks for certain types of learning. Brown *et al* [BBG99] propose the following sequence of events:

1. Dopamine neuron receives a reward.

    - Excitation of dopamine neurons in the substantia nigra pars compacta as a result of excitation from the lateral hypothalamus via the pedunculopontine nucleus.

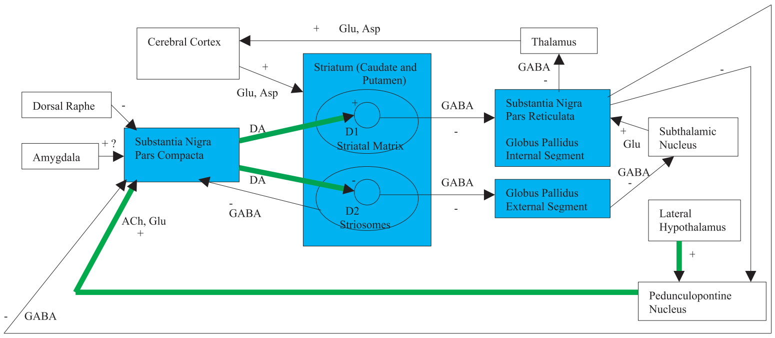

Figure 6.7: Excitation of dopamine neurons at time of reward in Brown *et al* [BBG99] dopamine model. Green highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

2. a. Dopamine neuron receives a stimulus that predicts a reward

   b. Dopamine neuron then receives the predicted reward.

   - a. Excitation from the limbic cortex activates the striatal matrix which inhibits the substantia nigra pars reticulata, and therefore excites the pedunculopontine nucleus and thefore excites the substantia nigra pars compacta, resulting in dopamine release.

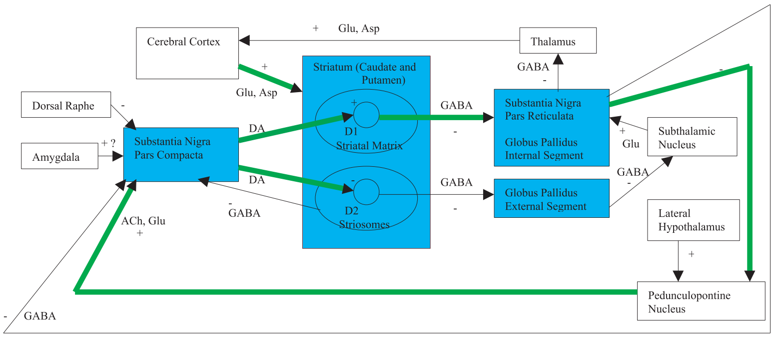

Figure 6.8: Excitation of dopamine neurons at time of conditioned stimulus in Brown *et al* [BBG99] dopamine model. Green highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

- b. Delayed inhibition from the striosomes to the substantia nigra pars compacta as a result of the following. Glutamate excites neurons in the striosomes which causes increased calcium concentrations within the neuron. There are then many calcium spikes occuring at different delays. When a dopamine burst (in response to the reward) also occurs with one of these spikes, then "that reward can be predicted at various times after CS onset, by strengthening the inhibitory effect of the striosomal cell with the appropriate delay [BBG99], p.7." This delayed inhibition is balanced by excitation of dopamine neurons in the substantia nigra pars compacta as a result of excitation from the lateral hypothalamus via the pedunculopontine nucleus, thereby resulting in no net change.

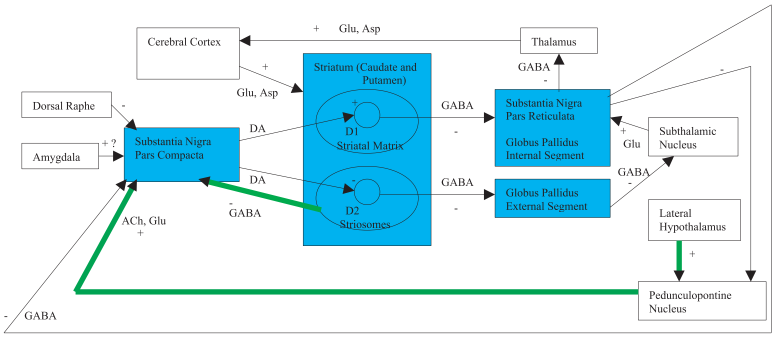

Figure 6.9: No activation in dopamine neurons at time of predicted reward in Brown *et al* [BBG99] dopamine model. Green highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [BBG99], [Sch98], [Mar96], [AHBR99a], [AHBR99b], [AHBR99c].

3. a. Dopamine neuron receives a stimulus that predicts a reward

   b. Dopamine neuron then does not receive the predicted reward

   - a. As in 2a excitation from the limbic cortex activates the striatal matrix which inhibits the substantia nigra pars reticulata, and therefore excites the pedunculopontine nucleus and thefore excites the substantia nigra pars compacta, resulting in dopamine release.

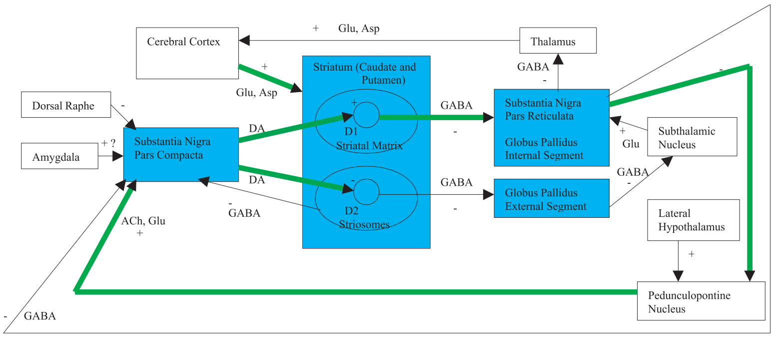

Figure 6.10: Excitation of dopamine neurons at time of conditioned stimulus in Brown *et al* [BBG99] dopamine model. Green highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

- b. Again there is delayed inhibition from the striosomes to the substantia nigra pars compacta as a result of the following. Glutamate excites neurons in the striosomes which causes increased calcium concentrations within the neuron. There are then many calcium spikes occuring at different delays. When a dopamine burst (in response to the reward) also occurs with one of these spikes, then "that reward can be predicted at various times after CS onset, by strengthening the inhibitory effect of the striosomal cell with the appropriate delay [BBG99], p.7." While again there is delayed inhibition, now that inhibition is no longer being balanced by excitation of dopamine neurons in the substantia nigra pars compacta as a result of excitation from the lateral hypothalamus via the pedunculopontine nucleus. The net change is therefore inhibition.

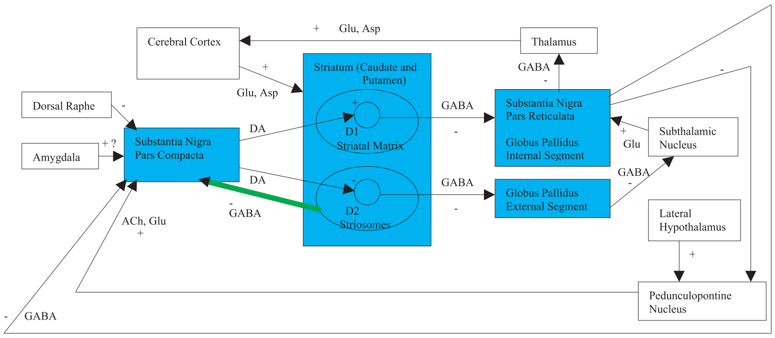

Figure 6.11: Inhibition at time of missing predicted reward in Brown *et al* [BBG99] dopamine model. Green highlighted arrows indicate the pathway involved. Blue labelled components belong to the basal ganglia. Sources: [AHBR99a], [AHBR99b], [AHBR99c], [BBG99], [Mar96], [Sch98].

6.6 Schultz [Sch98] Model vs Brown, Bullock, and Grossberg [BBG99] Model

Brown *et al*'s [BBG99] main contention point with Schultz's [Sch98] model appears to be that, while the final results of the dopamine neuron behavior can be explained by temporal-difference learning, the intermediate steps cannot be. In other words, using the temporal-difference error of the actual reward (or conditioned stimulus) minus the predicted reward (or conditioned stimulus) describes very well what is seen with dopamine neurons: if the reward is novel, then excitation; if a conditioned stimulus then the predicted reward, then excitation then no change; if a conditioned stimulus then the predicted reward is absent, then excitation then inhibition. However, the temporal-difference learning algorithm specifies that excitation in the time period between the stimulus and the reward should be seen, as the reward is propagated backwards through time. Brown *et al* [BBG99] point out that [LAS92] found that this is never realized, and that sometimes even excitation is seen both at the time of the conditioned stimulus and at the time of the predicted reward. Excitation at both time points, however, would actually lend support to the TD algorithm rather than challenge it.

It should be noted however, that both the Schultz [Sch98] model and the Brown *et al* [BBG99] are proposed explanations for observed phenomena. Each of the models has their own weaknesses and strengths.

The main strength of the Brown *et al* [BBG99] is that it allows for inhibition to occur precisely at the time the reward occurs [BBG99]. The main strength of the Schultz [Sch98] model is that the temporal-difference model objectively and correctly predicts whether there should be excitation or inhibition. However this strength is also a weakness since intermediate spikes are not observed during the interval between the stimulus and the reward (see [BBG99]). This means that the temporal-difference model does not seem capable of explaining the process by which the learning occurred. This doesn't necessarily negate the entire model, only this aspect of it.

The primary differences in the two models appear to be the following. *i*) First, the two models differ in the orgins of excitation for the dopamine neurons when there is a reward: the limbic cortex (for conditioned stimuli) and lateral hypothalamus (for reward) pathways [BBG99] versus direct excitation from the amygdala and the pedunculopontine nucleus and indirect excitation from the striatal matrix [Sch98]. Brown *et al* [BBG99], say that there are several ways that substantia nigra pars compacta dopamine neuron excitation can occur but that their model does not incorporate them. This may represent one of the difficulties in developing a model of these neurons: they receive so much input from such a variety of sources, that focusing on only one of the sources of input will be inadequate. *ii*) Second, the two models differ in the inhibitory mechanism at the time of an absent predicted reward. It is interesting to note, however, that the inhibitory pathways appear to be the same in both models, although the mechanisms may differ: striosomal inhibition of dopamine neurons in the substantia nigra pars compacta. Since the inhibitory mechanism of the [BBG99] model using timed calcium spikes is one of its strengths, one could hypothesize combining this inhibitory mechanism with the rest of the [Sch98] model to obtain yet another model.

In conclusion, more experimentation is needed to test each model to clarify the weak points, and to ultimately develop a new more comprehensive model that would likely include the different substantia nigra pars compacta inputs.

Chapter 7

Summary

7.1 Conclusions and Future Work

The experiments that we looked at would fall under pattern recognition and classification tasks. The usefulness of these experiments for jazz improvisation can, for example, be seen with the context-switching experiment: in jazz improvisation, it is sometimes nice to introduce the coming sequence of notes by playing certain notes that sound good in the coming sequence at the end of the current sequence.

We give a summary of the experimental results here. For ordered sequence linear TD(λ) experiments with the state as the input, the network learns well that the sequence 0 1 2 is good; it cannot, however, distinguish the sequence 0 1 2 from the sequence 2 1 0. For ordered sequence linear TD(λ) experiments with the eligibility trace as the input, the network again learns that the sequence 0 1 2 is good but is unable to learn that the sequence 2 1 0 is bad. While the overall results are similar, the clarity of the results, however, is diminished by the use of the eligibility trace as input instead of the state. Note that in these linear experiments, the only sequences on which the network was trained that contained notes 0, 1, or 2, was the sequence 0 1 2. These linear experiments were our initial, preliminary experiments, and at the time we had not yet decided to use the average trace as input to the network, nor to train it on sequences such as 2 1 0. These experiments did serve, however, as the basis for later ideas.

For the counting non-linear TD(λ) experiments with the state as input, the network learns that note 2 is always good, regardless of whether the sequence contains two note 2s or five note 2s. For the counting non-linear TD(λ) experiments with the eligibility trace as input, the network again learns that note 2 is always good regardless of whether the sequence contains two note 2s or five note 2s; there is however, an unexplained low value for 2 in the first position of the sequence ($V_1(2)$). For the counting non-linear TD(λ) experiments with the average trace as the input, the network learns as we would like it to: it learns that note 2 is good if there are a sufficient number of 2s in the sequence, otherwise it learns that note 2 is bad. For the ordered sequence non-linear TD(λ) experiments with the state as the input, it learns that the sequence 0 1 2 is good, and that the sequence 2 1 0 is less good but still better than the sequence 3 4 5; it is unable to learn that the sequence 3 4 2 is bad. For the ordered sequence non-linear TD(λ) experiments with the average trace as the input, it learns that the sequence 0 1 2 is good, that the sequence 2 1 0 is less good but still better than the sequence 3 4 5, and that the sequence 3 4 2 is less good although still better than 3 4 5. For the context-switching non-linear TD(λ) experiments with the state as the input it learns little; we hypothesized that it was too hard to learn without some kind of history that 0 as the last note of the preceding sequence was good in certain cases. For the context-switching non-linear TD(λ) experiments with the average trace as the input, the network learns that note 0 is always good as the last note of the sequence, and is able to distinguish that when 0 is the last note of the preceding sequence, then if the current chord is chord 2 then the current notes are good; it is unable to distinguish however, that note 0 is good only if the next chord is chord 2 (note that this should be feasible since information about the next chord is used in computing $V(s')$).

For the ordered sequence linear Sarsa(λ) experiments with the state as the input we found that the network learned to always play note 0 (when rewarded on the sequence 0 1 2), and the network learned to always play note 2 when rewarded on the sequence x 1 2, where x can be any note.

For the ordered sequence non-linear Sarsa(λ) experiments with the state as the input we found that the network learned to always play note 2 when rewarded on the sequence 0 1 2, and to always play note 1 when rewarded on the sequence x 1 2, where x can be any note. Similarly for the ordered sequence non-linear Sarsa(λ) experiments with the average trace as the input we found that the network learned to always play note 2 when rewarded on the sequence 0 1 2, and to always play note 1 when rewarded on the sequence x 1 2, where x can be any note. We hypothesize that adjusting the parameters might improve the usefulness of the average trace in the non-linear Sarsa(λ). While it appears that linear Sarsa(λ) learns more than non-linear with Rule #1 since it learns to play note 0, it should be noted, however, that it was found that the non-linear network with the state as input also learned to always play note 0 when run for one million rounds, but when run for ten million rounds it learned to always play note 2. Further exploration into these experiments is needed. The non-linear Sarsa(λ) was clearly better with Rule #2, however, since it learned to always play note 1, as compared with note 2 in the non-linear case.

What became apparent as the experiments progressed was that the network needed a history due to the non-Markovian nature of the problem. We chose to address the problem of the history using first the eligibility trace as the input and then the average trace from [Jor86].

The importance of the average trace in our problem is because the network has no context for the notes it sees. Providing a context for each note as it is given to the network is not the primary purpose of the eligibility trace. The eligibility trace is used in the weight update but when the network actually calculates a prediction for a note, it does not have access to the eligibility. So even if the eligibility is encoded in the network, when the network sees a note, it will have no information about the note's position in the sequence or what the other preceding notes in the sequence were; hence the network will output a similar value for the same note regardless of the context. This is why we chose to use the average trace from [Jor86]. We also attempted to alleviate this problem by giving as another input to the network the note index (the current index of the note within the sequence), but those results were not discussed here as the note index did not seem to provide substantial improvement. One possible reason for this lack of improvement may be that the note index was too much information for the network and it was too easy too generalize the information to other unrelated examples.

What we were attempting to do indirectly through the use of the eligibility and average traces as input to the network was to employ recurrence. The next step would therefore be to use a neural network that was in itself recurrent.

Returning to convergence issues, it was found that while the linear network showed good stability as expected, there were some fluctuations with the non-linear network (for example, cycling from all positive to all negative to all postive values). However, regardless of this fluctuation, the results appeared to be consistent. Addressing the fundamental issue of non-convergence when non-linear function approximation is combined with reinforcement learning, there are several possible directions to take. For instance, [BM95] have suggested an alternative algorithm to value iteration which they call the "Grow-Support" algorithm. Baird [Bai99] in his PhD thesis has suggested several methods for function approximation based reinforcement learning, in particular, "residual algorithms" and "Advantage learning."

In short, it was found that non-linear TD(λ) gives better predictions on harder problems than can linear TD(λ). It was also found that the average trace improved the accuracy of the non-linear TD(λ) prediction. Linear Sarsa(λ) gave better results with the unambiguous rule (Rule #1) but non-linear Sarsa(λ) was more adept at learning the more ambiguous rule (Rule #2). Given more experimentation with parameters, it is hypothesized that non-linear Sarsa(λ) would be at least as adept as linear Sarsa(λ) at learning the unambigous rule (Rule #1), and that for non-linear Sarsa(λ), the average trace could be developed to be more effective.

Finally, regarding the neurobiological view of temporal-difference learning, the pure temporal-difference learning view of Schultz [Sch98] may not be feasible [BBG99], but some combination of the Schultz [Sch98] and the Brown [BBG99] view may lead to a better anatomical model as well as to an alternative learning algorithm. One option for combining the two models would be to incorporate the mechanism for inhibition at the time of the predicted reward (based on weight strengthening from dopamine release occurring at the same time as a calcium spike) from Brown [BBG99] with the rest of the Schultz [Sch98] model.

Appendix A

Derivations

This appendix contains derivations.

A.1 Differentiation of Activation Function, $f(\theta)$

Differentiation of $f(\theta)$ to get $f'(\theta) = f(\theta)(1 - f(\theta))$.

$$
\begin{aligned}
f'(\theta) &= \frac{0(1+e^{-\theta}) - 1(e^{-\theta})(-1)}{(1+e^{-\theta})^2} \\
&= \frac{e^{-\theta}}{(1+e^{-\theta})^2} \\
&= \frac{1}{1+e^{-\theta}} \frac{e^{-\theta}}{1+e^{-\theta}} \\
&= \frac{1}{1+e^{-\theta}} \frac{1-1+e^{-\theta}}{1+e^{-\theta}} \\
&= \frac{1}{1+e^{-\theta}} \frac{(1+e^{-\theta})-1}{1+e^{-\theta}} \\
&= \frac{1}{1+e^{-\theta}} \{\frac{(1+e^{-\theta})}{1+e^{-\theta}} - \frac{1}{1+e^{-\theta}}\} \\
&= \frac{1}{1+e^{-\theta}} \{1 - \frac{1}{1+e^{-\theta}}\}
\end{aligned}
$$

A.2 Non-Linear Model $e(h_j(\vec{s_t}, \vec{z_j}))$

In this model we are using a modified version of the nonlinear TD(λ) algorithm [Sut88] that uses the λ eligibility traces for the hidden unit output as the input to the output unit.

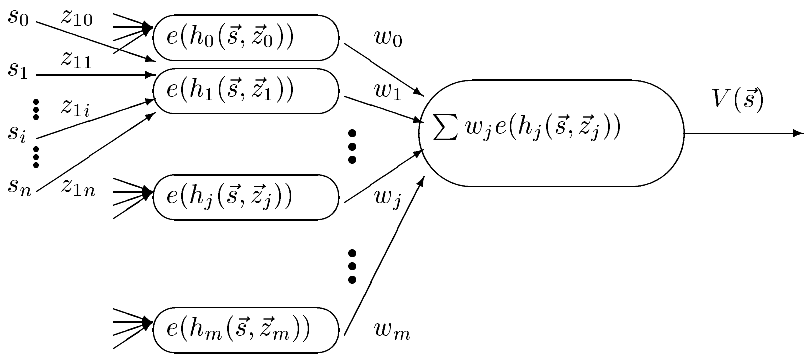

Figure A.1: Non-linear TD(λ) network with $e(h_j(\vec{s}_{tNote}, \vec{z}_j))$ as input.

We have $e(h_j(\vec{s}, \vec{z}_j)) = e(f(\sum_{i=0}^{n} s_i z_{ji}))$. Using this, $V(s_t)$ now becomes,

$$\begin{aligned} V(s_t) &= \sum_{j=0}^{m} w_j e(h_j(\vec{s}, \vec{z}_j)) \\ &= \sum_{j=0}^{m} w_j e(f(\sum_{i=0}^{n} s_i z_{ji})) \end{aligned}$$

Our algorithm outline is then,

1 $\vec{w} = 0$, $\vec{e} = 0$
2 For each episode
3 $\quad s_t = s_0$

4 For each state in episode

5 Get a

6 Get r and s_{t+1} using a

7 $\delta = r + \gamma V(s_{t+1}) - V(s_t)$

8a $\vec{e}_{z_j} = \lambda \vec{e}_{z_j} + \nabla_{\vec{z}_j} V(s_t)$

8b $\vec{e}_{w_j} = \lambda \vec{e}_{w_j} + \nabla_{\vec{w}} V(s_t)$

9a $\vec{z}_j = \vec{z}_j + \alpha\delta\vec{e}_{z_j}$

9b $\vec{w} = \vec{w} + \alpha\delta\vec{e}_w$

10 $s_t = s_{t+1}$

11 Continue until goal state is reached

We must now calculate $\nabla_{\vec{z}_j} V(s_t)$ for step 8a and $\nabla_{\vec{w}} V(s_t)$ for step 8b. For step 8a,

$$
\begin{aligned}
e_{\vec{z}_j} &= \lambda \vec{z}_j + \nabla_{\vec{z}_j} V(s_t) \\
=> e_{z_{ji}} &= \lambda e_{z_{ji}} + \frac{\partial V(s_t)}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial \sum_{j=0}^{m} w_j e(h_j(\vec{s}, \vec{z}_j))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (w_0 e(h_0(\vec{s}, \vec{z}_0)) + ... + w_j e(h_j(\vec{s}, \vec{z}_j)) + ... + w_m e(h_m(\vec{s}, \vec{z}_m))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (w_j e(h_j(\vec{s}, \vec{z}_j))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + \frac{\partial (w_j e(f(\sum_{i=0}^{n} s_i z_{ji}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j \frac{\partial (e(f(\sum_{i=0}^{n} s_i z_{ji}))}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j \frac{\partial (e(f(\sum_{i=0}^{n} s_i z_{ji}))}{\partial z_{ji}}
\end{aligned}
$$

We then adopt the truncated backpropagation method to approximate the partial derivative [HS97].

$$
e_{z_{ji}} = \lambda e_{z_{ji}} + w_j \frac{\partial (f(\sum_{i=0}^{n} s_i z_{ji}))}{\partial z_{ji}}
$$

$$
\begin{aligned}
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} s_i z_{ji}) \frac{\partial(s_0 z_{j0} + ... + s_i z_{ji} + ... + s_n z_{jn})}{\partial z_{ji}} \\
&= \lambda e_{z_{ji}} + w_j f'(\sum_{i=0}^{n} s_i z_{ji}) s_i \\
&= \lambda e_{z_{ji}} + w_j f(\sum_{i=0}^{n} s_i z_{ji})(1 - f(\sum_{i=0}^{n} s_i z_{ji})) s_i \\
&= \lambda e_{z_{ji}} + w_j h_j(\vec{s}, \vec{z}_j)(1 - h_j(\vec{s}, \vec{z}_j)) s_i
\end{aligned}
$$

For step 8b,

$$
\begin{aligned}
e_{\vec{w}} &= e_{\vec{w}} + \alpha \delta \nabla_{\vec{w}} V(s_t) \\
=> e_{w_j} &= e_{w_j} + \frac{\partial V(s_t)}{\partial w_j} \\
&= e_{w_j} + \frac{\partial \sum_{j=0}^{m} w_j e(h_j(\vec{s}, \vec{z}_j))}{\partial w_j} \\
&= e_{w_j} + \frac{\partial(w_0 e(h_0(\vec{s}, \vec{z}_0)) + ... + w_j e(h_j(\vec{s}, \vec{z}_j)) + ... + w_m e(h_m(\vec{s}, \vec{z}_m))}{\partial w_j} \\
&= e_{w_j} + e(h_j(\vec{s}, \vec{z}_j)) \\
&= e_{w_j} + e(f(\sum_{i=0}^{n} s_i z_{ji}))
\end{aligned}
$$

The algorithm outline then becomes,

1 $\vec{w} = 0$, $\vec{e} = 0$
2 For each episode
3 $s_t = s_0$
4 For each state in episode
5 Get a
6 Get r and s_{t+1} using a
7 $\delta = r + \gamma V(s_{t+1}) - V(s_t)$
8 For all $0 \leq j \geq n$ and $0 \leq i \geq m$
a $e_{z_j i} = \lambda e_{z_j} + w_j h_j(\vec{s}, \vec{z}_j)(1 - h_j(\vec{s}, \vec{z}_j)) s_i$
b $e_{w_j} = \lambda e_{w_j} + e(f(\sum_{i=0}^{n} s_i z_{ji}))$

9a $\vec{z}_j = \vec{z}_j + \alpha\delta\vec{e}_{z_j}$

9b $\vec{w} = \vec{w} + \alpha\delta\vec{e}_w$

10 $s_t = s_{t+1}$

11 Continue until goal state is reached

Appendix B

Code

This appendix contains code.

B.1 Linear TD(λ)

```
//-------------------------------------------------------------------------
//  Simulator.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear TD:  Header file for Simulator.cpp
//
//-------------------------------------------------------------------------
#include "TD.h"
#include "Constants.h"
#include "Notes3NSeq.h"
#include "NotesCharlieParker.h"

class Simulator {

  public:
  private:

};



//-------------------------------------------------------------------------
//  Simulator.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear TD:  Main file that runs the linear temporal-difference
//            learning algorithm.  Sequences to be learned are
//            randomly chosen in this file.
//
//-------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include <string.h>
#include "Simulator.h"

//================================================================
//
//  Main
```

```
//
//=================================================================
int main()
{
  int i = 0, j = 0, k = 0;

  int* seqs[NUM_SEQS] =
  {C_CS_D_Seq1,   C_CS_D_Seq2,   C_CS_D_Seq3,   C_CS_D_Seq4,   C_CS_D_Seq5,
   C_CS_D_Seq6,   C_CS_D_Seq7,   C_CS_D_Seq8,   C_CS_D_Seq9,   C_CS_D_Seq10,
   C_CS_D_Seq11,  C_CS_D_Seq12,  C_CS_D_Seq13,  C_CS_D_Seq14,  C_CS_D_Seq15,
   C_CS_D_Seq16 , C_CS_D_Seq17,  C_CS_D_Seq18,  C_CS_D_Seq19,  C_CS_D_Seq20};

  int* chords[NUM_CHORDS] = {BFLAT,F7};
  TD*  tdLambda  = new TD();
  int* chord     = chords[0];
  int* seq       = seqs[0];
  int* note      = seq;
  int  chooseSeq = lrand48() % NUM_SEQS;

  cout << "# Number of rounds is                : " << NUM_ROUNDS    << endl;
  cout << "# Length of a sequence is            : " << LEN_SEQ       << endl;
  cout << "# Length of a note is                : " << LEN_NOTE      << endl;
  cout << "# Length of a chord is               : " << LEN_CHORD     << endl;
  cout << "# Number of sequences is             : " << NUM_SEQS      << endl;
  cout << "# Number of notes in a sequence is   : " << NUM_NOTES     << endl;
  cout << "# Number of chords is                : " << NUM_CHORDS    << endl;
  cout << "# Maximum reward is                  : " << RMAX          << endl;
  cout << "# ALPHA output node learning rate is : " << ALPHA_O       << endl;
  cout << "# GAMMA discount factor is           : " << GAMMA         << endl;
  cout << "# LAMBDA output node is              : " << LAMBDA_O      << endl;

  //---------------------------------------------------------------------------
  //  Training
  //---------------------------------------------------------------------------
  for (i = 0; i < NUM_ROUNDS; i++)
    {
      chooseSeq = lrand48() % NUM_SEQS;
      seq       = seqs[chooseSeq];
      chord     = chords[lrand48() % NUM_CHORDS];

      //---  Learn note and chord ---
      for (j = 0; j < NUM_NOTES; j++)
        {
          note = &seq[j*LEN_NOTE];
          tdLambda->TDAlgorithm(seq, note, chord, j, i);
        }
```

```
      //---  Print chord ---
      if (i > NUM_ROUNDS - 2)
        {
          cout << "chord : " << endl;
          for (k = 0; k < LEN_CHORD; k++)
            cout << chord[k] << " ";
          cout << "i = " << i << " and chooseSeq = "
               << chooseSeq + 1 << endl;
          cout << "---------------------------------------" << endl;
        }

      //--- Replace the eligibility traces ---
      tdLambda->ReplaceElig();

  }// end for i

  //---------------------------------------------------------------------------
  //  Test on untrained sequence
  //---------------------------------------------------------------------------
  seq   = test_C_CS_D_Seq1;
  chord = chords[0];

  //---  Learn note and chord ---
  for (i = 0; i < NUM_NOTES; i++)
    {
      note = &seq[i*LEN_NOTE];
      tdLambda->TDAlgorithm(seq, note, chord, i, NUM_ROUNDS);
    }

  //---  Print chord ---
  cout << "Test" << endl;
  cout << "chord : " << endl;
  for (k = 0; k < LEN_CHORD; k++)
    cout << chord[k] << " ";
  cout << "---------------------------------------" << endl;

}// end main



//------------------------------------------------------------------------
//  TD.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
```

```
//  Linear TD:  Header file for TD.cpp
//
//---------------------------------------------------------------------------

#include <math.h>
#include "Constants.h"

class TD {

  public:
       TD();
       void   TDAlgorithm(int*, int*, int*, int noteIndex, int round);
       int    GetState(int*);
       double GetValue(int*, int*);
       double Critique(int*, int*);
       void   ReplaceElig();
       double InitWeight();
       double ActFunc(double tempSum);

  private:
       double alpha;            // Learning rate
       double gamma;            // Discount rate
       double lambda;           // Eligibility decay rate
       double delta;            // Error
       double wN[LEN_NOTE];     // Note/State weights
       double wC[LEN_NOTE];     // Chord weights
       double eN[LEN_NOTE];     // Note/State eligibility
       double eC[LEN_NOTE];     // Chord eligibility

       double r;                // Reward
};



//---------------------------------------------------------------------------
//  TD.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear TD:  This file implements the linear temporal-difference
//            learning algorithm of Richard Sutton (1988).
//
//---------------------------------------------------------------------------

#include <iostream.h>
```

```
#include <stdio.h>
#include <stdlib.h>
#include "TD.h"

//=================================
//
//  Constructor
//
//=================================
TD::TD()
{
  int i = 0;
  delta  = 0;
  alpha  = ALPHA_0;
  gamma  = GAMMA;
  lambda = LAMBDA_0;
  r      = 0;

  for (i = 0; i < LEN_NOTE; i++) {
      wN[i] = InitWeight();
      wC[i] = InitWeight();
      eN[i] = 0;
      eC[i] = 0;
    }
}

//=================================================
//
//  The TD() algorithm - with hidden units
//
//=================================================
void TD::TDAlgorithm(int* seq, int* note, int* chord, int noteIndex, int round)
{

  //------------------
  // Find the state
  //------------------
  int state = GetState(note);

  //-------------------------
  // Find the next state
  //-------------------------
  int* nextNote;
  int  zeroArray[LEN_NOTE] = {0};
  if (noteIndex < NUM_NOTES - 1)
    nextNote = &note[LEN_NOTE];
  else
```

```
    nextNote = zeroArray;

  //---------------------------------------
  // Find the predicted value for the state
  //---------------------------------------
  double V = GetValue(note, chord);

  //---------------------------------------------
  // Find the predicted value for the next state
  //---------------------------------------------
  double VPrime = GetValue(nextNote, chord);

  //--------------
  // Get reward
  //--------------
  if (noteIndex >= NUM_NOTES - 1)
    r = Critique(seq, chord);

  //------------------------------
  // Find delta for output node
  //------------------------------
  delta = r + gamma*VPrime - V;

  //-------------------------------------------------------------------
  // Update eligibilities of network inputs for use by hidden nodes
  //-------------------------------------------------------------------
  for (int i = 0; i < LEN_NOTE; i++) {
    eN[i] = gamma * lambda * eN[i] + note[i];
    eC[i] = gamma * lambda * eC[i] + chord[i];
  }

  //--------------------------------
  // Update output node weights
  //--------------------------------
  for (int i = 0; i < LEN_NOTE; i++) {
    wN[i] = wN[i] + alpha * delta * eN[i];
    wC[i] = wC[i] + alpha * delta * eC[i];
  }

  if (round > NUM_ROUNDS - 2)
    cout << state << "\tV = " << V << "  reinforcement = "
         << r << "   noteIndex = " << noteIndex << endl;

  r = 0;

}
```

```
//==================================================================
//
//  GetState
//
//==================================================================
int TD::GetState(int* note)
{
  int i = 0;

  for (i = 0; i < LEN_NOTE; i++)
    if (note[i] == 1)
      break;

  return i;
}

//==================================================================
//
//  GetValue
//
//==================================================================
double TD::GetValue(int* note, int* chord)
{
  double dotProd = 0;

  //-------------------------------------
  // Sum from (i = 1 to N) of wi*e(s)i
  //-------------------------------------
  for (int i = 0; i < LEN_NOTE; i++)
    dotProd += wN[i]*note[i];

  return dotProd;
}

//==================================================================
//
//  Critique:  the reward function
//
//==================================================================
double TD::Critique(int* seq, int* chord)
{
  double reward = 0;

  //-----------------------------------------------------------------
  //  Rule 1:  If there's a note in sequence that is a chord note then
  //        reward = RMAX, otherwise reward = 0;
  //-----------------------------------------------------------------
```

```
  /*  for (int i = 0; i < (NUM_NOTES * LEN_NOTE); i++) {
    for (int j = 0; j < LEN_CHORD; j++) {
      if ((seq[i] == 1) && (chord[j] == 1) && ((i % LEN_NOTE) == j))
        reward++;
    }
  }
  */

  //--------------------------------------------------------------------
  //  Rule 2:  If there exists within the sequence the ordered notes,
  //         0 1 2, then reward = RMAX, otherwise reward = 0;
  //--------------------------------------------------------------------
  // If the first digit of the first note is 1 and
  //    the second digit of the second note is 1 and
  //    the third digit of the third note is 1
  if (seq[0]==1 && seq[LEN_NOTE+1]==1 && (seq[2*LEN_NOTE+2]==1))
    reward = 1;
  else
    reward = 0;


  //--------------------------------------------------------------------
  //  Rule 3:  If there exists within the sequence the ordered notes,
  //         2 2 2, then reward = RMAX, otherwise reward = 0;
  //--------------------------------------------------------------------
  /*if (seq[2]==1 && seq[LEN_NOTE+2]==1 && (seq[2*LEN_NOTE+2]==1))
    reward = 1;
  else
    reward = 0;
  */

  return reward;
}

//=================================================================
//
//  ReplaceElig: resets the eligibility traces for all of
//             the states/notes to zero.
//
//=================================================================
void TD::ReplaceElig()
{
  for (int i = 0; i < LEN_NOTE; i++) {
    eN[i] = 0;
    eC[i] = 0;
  }
}
```

```
//====================================================
//
//  ActFunc
//
//  The sigmoid activation function as given by,
//
//      f(sum) = 1 / (1 + e^(-sum))
//
//====================================================
double TD::ActFunc(double tempSum)
{
  return (1 / (1 + exp(-tempSum)));
}

//====================================================
//
//  InitWeight
//
//====================================================
double TD::InitWeight()
{
  double newW = 10;

  if (drand48() > 0.5)
    while (newW < 0.001 || newW > 0.01)
      newW = drand48() / DENOM;
  else
    while (newW > -0.001 || newW < -0.01)
      newW = -drand48() / DENOM;

  return newW;
}
```

```
#-------------------------------------------------------------
# Makefile
#
# V. Manfredi and J. Franklin        Dept of Computer Science
# Honors Thesis                      Smith College
# fall, spring 2002                  Northampton, MA 01063
#
# Makefile for linear and non-linear temporal-difference
# learning code.
#
#-------------------------------------------------------------
```

```
all: Simulate

Simulator.o: Simulator.cpp Simulator.h
	g++ -g -Wall -c Simulator.cpp

TD.o: TD.cpp TD.h
	g++ -g -Wall -c TD.cpp

Simulate: Simulator.o TD.o
	g++ -g -o Simulate Simulator.o TD.o
```

```
//---------------------------------------------------------------------------
//  Constants.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear TD:  Constants file for linear temporal-difference learning
//            code.
//
//---------------------------------------------------------------------------

#define NUM_ROUNDS      10000
#define LEN_SEQ            36
#define LEN_CHORD          12
#define LEN_NOTE           12
#define NUM_SEQS           20
#define NUM_CHORDS          2
#define NUM_NOTES           3
#define RMAX                1
#define ALPHA_O          0.01
#define GAMMA            0.95
#define LAMBDA_O          0.7
#define DENOM             100
```

```
//---------------------------------------------------------------------------
//  Notes3NSeq.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
```

```
//
//  Example and test sequences for linear temporal-difference learning.
//
//------------------------------------------------------------------------

//------------------------------------------------------------------------
// C_CS_D_Seqi[LEN_NOTE*NUM_NOTES] makes up a state
//------------------------------------------------------------------------
// C  = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
// CS = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
// D  = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
//------------------------------------------------------------------------
int C_CS_D_Seq1[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq2[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq3[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq4[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq5[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq6[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq7[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq8[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq9[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
//---------------------------------------------------------------------------
int C_CS_D_Seq10[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                        0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0};

int C_CS_D_Seq11[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq12[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                        0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq13[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};

int C_CS_D_Seq14[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};

int C_CS_D_Seq15[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                        0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq16[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int C_CS_D_Seq17[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int C_CS_D_Seq18[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int C_CS_D_Seq19[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                        0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0};

int C_CS_D_Seq20[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                        0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

//-------------------------------------------------------------------------------
```

```
int test_C_CS_D_Seq1[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                            0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                            1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int test_C_CS_D_Seq2[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                            0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                            0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int test_C_CS_D_Seq3[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                            0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                            0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0};

int test_C_CS_D_Seq4[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                            0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                            0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int test_C_CS_D_Seq5[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                            0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                            0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};


//------------------------------------------------------------------
//  NotesCharlieParker.h
//
//  V. Manfredi and J. Franklin        Dept of Computer Science
//  Honors Thesis                      Smith College
//  fall, spring 2002                  Northampton, MA 01063
//
//  Example chords for linear and non-linear temporal-difference
//  learning.
//
//------------------------------------------------------------------

#ifndef SONGS_H
#define SONGS_H

int AFLAT7[12]     = {1,0,0,1,0,0,1,0,1,0,0,0};
int BFLAT[12]      = {0,0,1,0,0,1,0,0,0,1,1,0};
int BFLAT7[12]     = {0,0,1,0,0,1,0,0,1,0,1,0};
int BFLATMINOR[12] = {0,1,0,0,0,1,0,0,1,0,1,0};
int C7[12]         = {1,0,0,0,1,0,0,1,0,0,1,0};
int CMINOR[12]     = {1,0,0,1,0,0,0,1,0,0,1,0};
int D7[12]         = {1,0,1,0,0,0,1,0,0,1,0,0};
int DFLAT7[12]     = {0,1,0,0,0,1,0,0,1,0,0,1};
int DFLATMINOR[12] = {0,1,0,0,1,0,0,0,1,0,0,1};
int EFLAT7[12]     = {0,1,0,1,0,0,0,1,0,0,1,0};
```

```
int G7PLUS9[12]     = {0,0,0,1,0,1,0,1,1,0,1,1};
int G7[12]          = {0,0,1,0,0,1,0,1,0,0,0,1};
int GMINOR[12]      = {0,0,1,0,0,1,0,1,0,0,1,0};
int F7[12]          = {1,0,0,1,0,1,0,0,0,1,0,0};
int F7PLUS9[12]     = {0,1,0,1,0,1,1,0,1,1,0,0};

enum note_index
{
  C=1,Dflat=2,D=3,Eflat=4,E=5,F=6,Gflat=7,G=8,Aflat=9,A=10,Bflat=11,B=12
};

#endif
```

B.2 Non-linear TD(λ) for Ordered Sequence and Counting

```
//-----------------------------------------------------------------------
//  Simulator.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Header file for Simulator.cpp
//
//-----------------------------------------------------------------------

#include "TD.h"
#include "Constants.h"

#if ORD_SEQ
#include "NotesOrdSeq.h"
#endif

#if COUNT
#include "NotesCount.h"
#endif

#include "NotesCharlieParker.h"

class Simulator {
  public:
  private:
};



//-----------------------------------------------------------------------
//  Simulator.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Main file that runs the non-linear temporal-difference
//            algorithm for the non-linear ordered sequence and counting
//            experiments. Sequences to be learned are randomly chosen
//            in this file.
//
//-----------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
```

```
#include <iostream.h>
#include <math.h>
#include <string.h>
#include "Simulator.h"

//================================================================
//
//  Main
//
//================================================================
int main()
{
  int i = 0, j = 0;
  TD* tdLambda = new TD();

  //------------------------------------------------------------------
  // Initialize test sequences
  //------------------------------------------------------------------
  #if COUNT
  int* seqs[NUM_SEQS] =
  {CountSeq1,   CountSeq2,   CountSeq3,  CountSeq4,   CountSeq5,
   CountSeq6,   CountSeq7,   CountSeq8,  CountSeq9,   CountSeq10,
   CountSeq11,  CountSeq12,  CountSeq13, CountSeq14,  CountSeq15,
   CountSeq16 , CountSeq17,  CountSeq18, CountSeq19,  CountSeq20};
  int* chords[NUM_CHORDS] = {BFLAT,F7,C7};
  #endif

  #if ORD_SEQ
  int* seqs[NUM_SEQS] =
  {OrdSeq1,   OrdSeq2,   OrdSeq3,  OrdSeq4,   OrdSeq5,   OrdSeq6,
   OrdSeq7,   OrdSeq8,   OrdSeq9,  OrdSeq10,  OrdSeq11,  OrdSeq12,
   OrdSeq13,  OrdSeq14,  OrdSeq15, OrdSeq16 , OrdSeq17,  OrdSeq18,
   OrdSeq19,  OrdSeq20,  OrdSeq21, OrdSeq22,  OrdSeq23,  OrdSeq24,
   OrdSeq25,  OrdSeq26,  OrdSeq27, OrdSeq28,  OrdSeq29,  OrdSeq30};
  int* chords[NUM_CHORDS] = {BFLAT,F7,C7};
  #endif

  //------------------------------------------------------------------
  // Initialize chords, sequences, notes, and average note traces
  //------------------------------------------------------------------
  //--- Chords ---
  int chord     = lrand48() % NUM_CHORDS;
  int nextChord = lrand48() % NUM_CHORDS;
  while (chord == nextChord)
    nextChord = lrand48() % NUM_CHORDS;
  int* chordArray     = chords[chord];
  int* nextChordArray = chords[nextChord];
```

```
//--- Sequences ---
int seq     = lrand48() % NUM_SEQS;
int nextSeq = lrand48() % NUM_SEQS;
int* seqArray       = seqs[seq];
int* nextSeqArray   = seqs[nextSeq];

//--- Notes ---
int* noteArray      = seqArray;
int* nextNoteArray  = &seqArray[LEN_NOTE];

//--- Average note trace ---
double noteBar[LEN_NOTE]     = {0};
double nextNoteBar[LEN_NOTE] = {0};
for (i = 0; i < LEN_NOTE; i++) {
  noteBar[i] = noteArray[i];
  nextNoteBar[i] = nextNoteArray[i];
}

//---------------------------------------------------------------------
// Print out information about the constants
//---------------------------------------------------------------------
cout << "# Number of rounds is                 : " << NUM_ROUNDS    << endl;
cout << "# Number of hidden nodes is           : " << NUM_HID_NODES << endl;
cout << "# Length of a sequence is             : " << LEN_SEQ       << endl;
cout << "# Length of a note is                 : " << LEN_NOTE      << endl;
cout << "# Length of a chord is                : " << LEN_CHORD     << endl;
cout << "# Number of sequences is              : " << NUM_SEQS      << endl;
cout << "# Number of notes in a sequence is    : " << NUM_NOTES     << endl;
cout << "# Number of chords is                 : " << NUM_CHORDS    << endl;
cout << "# Maximum reward is                   : " << RMAX          << endl;
cout << "# ALPHA output node learning rate is : " << ALPHA_O       << endl;
cout << "# ALPHA hidden node learning rate is : " << ALPHA_H       << endl;
cout << "# GAMMA discount factor is            : " << GAMMA         << endl;
cout << "# LAMBDA is                           : " << LAMBDA        << endl;
cout << "# BIAS is                             : " << BIAS          << endl;

//-----------------------------------------------------------------------------
//  Training/Learning
//-----------------------------------------------------------------------------
for (int round = 0; round < NUM_ROUNDS; round++)
  {
    //--- Store the current and next chord ---
    chord = nextChord;
    chordArray = chords[chord];
    nextChord = lrand48() % NUM_CHORDS;
    while (chord == nextChord)
```

```
  nextChord = lrand48() % NUM_CHORDS;
nextChordArray = chords[nextChord];

//--- Store the current and next sequence ---
seq = nextSeq;
seqArray = seqs[seq];
nextSeq = lrand48() % NUM_SEQS;
#if CON_SWI
while (nextSeq < 11 && nextChord == 2)
  nextSeq = lrand48() % NUM_SEQS;
#endif
nextSeqArray = seqs[nextSeq];

//--- Find the current note and the next note ---
noteArray     = seqArray;
nextNoteArray = &seqArray[LEN_NOTE];

//--- Reset the average traces ---
for (j = 0; j < LEN_NOTE; j++) {
  noteBar[j]     = noteArray[j];
  nextNoteBar[j] = nextNoteArray[j];
}

//--------------------------------------------------------------------------
//  Learn note and chord
//--------------------------------------------------------------------------
for (i = 0; i < NUM_NOTES; i++)
  {
    //--- Find the current note ---
    noteArray = &seqArray[i*LEN_NOTE];

    //--- Find the next note ---
    if (i == (NUM_NOTES - 1))
      nextNoteArray = &nextSeqArray[0];
    else
      nextNoteArray = &seqArray[(i+1)*LEN_NOTE];

    #if BETA_TRACE
    //--- Find noteBar and nextNoteBar ---
    for (j = 0; j < LEN_NOTE; j++) {
      noteBar[j]     = BETA * noteBar[j] + (1-BETA) * noteArray[j];
      nextNoteBar[j] = BETA * nextNoteBar[j] + (1-BETA) * nextNoteArray[j];
    }
    #endif

    #if SOME_BETA_TRACE
    //--- Find noteBar and nextNoteBar ---
```

```
      for (j = 0; j < LEN_NOTE; j++) {
        noteBar[j]     = BETA * noteBar[j] + (1-BETA) * noteArray[j];
        nextNoteBar[j] = nextNoteArray[j];
      }
      #endif

      #if NO_BETA_TRACE
      //--- Find noteBar and nextNoteBar ---
      for (j = 0; j < LEN_NOTE; j++) {
        noteBar[j]     = noteArray[j];
        nextNoteBar[j] = nextNoteArray[j];
      }
      #endif

      //--- Get the output from the neural net and learn using TD ---
      tdLambda->TDAlgorithm(chordArray, nextChordArray, noteArray, nextNoteArray,
                            seqArray, noteBar, nextNoteBar, i, round);

    }// end for i

    //----------------------------------------------------------------------------
    // Print the chord and round information
    //----------------------------------------------------------------------------
    if (round >= (NUM_ROUNDS - 100)) {
      cout << "chord : " << endl;
      for (i = 0; i < LEN_CHORD; i++)
        cout << chordArray[i] << " ";
      cout << "\nnext chord : " << endl;
      for (i = 0; i < LEN_CHORD; i++)
        cout << nextChordArray[i] << " ";
      cout << "round = " << round << " and seq = "
           << seq + 1 << endl;
      cout << "---------------------------------------" << endl;
    }

  }// end for round

//----------------------------------------------------------------------------
// Test on an untrained sequence
//----------------------------------------------------------------------------
for (int round = 0; round < 1; round++)
  {
    //--- Store the current and next sequence ---
    seqArray = test_OrdSeq3;
    nextSeqArray = test_OrdSeq4;

    //--- Store the current and next chord ---
```

```
chordArray = chords[0];
nextChordArray = chords[1];

//--- Find the current note and the next note ---
noteArray     = seqArray;
nextNoteArray = &seqArray[LEN_NOTE];

//--- Reset the average traces ---
for (j = 0; j < LEN_NOTE; j++) {
  noteBar[j]     = noteArray[j];
  nextNoteBar[j] = nextNoteArray[j];
}

//-----------------------------------------------------------------------
//  Learn note and chord
//-----------------------------------------------------------------------
for (i = 0; i < NUM_NOTES; i++)
  {
    //--- Find the current note ---
    noteArray = &seqArray[i*LEN_NOTE];

    //--- Find the next note ---
    if (i == (NUM_NOTES - 1))
      nextNoteArray = &nextSeqArray[0];
    else
      nextNoteArray = &seqArray[(i+1)*LEN_NOTE];

    #if BETA_TRACE
    //--- Find noteBar and nextNoteBar ---
    for (j = 0; j < LEN_NOTE; j++) {
      noteBar[j]     = BETA * noteBar[j]     + (1-BETA) * noteArray[j];
      nextNoteBar[j] = BETA * nextNoteBar[j] + (1-BETA) * nextNoteArray[j];
    }
    #endif

    #if SOME_BETA_TRACE
    //--- Find noteBar and nextNoteBar ---
    for (j = 0; j < LEN_NOTE; j++) {
      noteBar[j]     = BETA * noteBar[j] + (1-BETA) * noteArray[j];
      nextNoteBar[j] = nextNoteArray[j];
    }
    #endif

    #if NO_BETA_TRACE
    //--- Find noteBar and nextNoteBar ---
    for (j = 0; j < LEN_NOTE; j++) {
      noteBar[j]     = noteArray[j];
```

```
        nextNoteBar[j] = nextNoteArray[j];
      }
      #endif

      //--- Get the output from the neural net and learn using TD ---
      tdLambda->TDAlgorithm(chordArray, nextChordArray, noteArray, nextNoteArray,
                            seqArray, noteBar, nextNoteBar, i, NUM_ROUNDS);
    }// end for i

   //-----------------------------------------------------------------------------
   // Print the chord and round information
   //-----------------------------------------------------------------------------
   cout << "Test Sequence" << endl;
   cout << "chord : " << endl;
   for (i = 0; i < LEN_CHORD; i++)
     cout << chordArray[i] << " ";
   cout << "\nnext chord : " << endl;
   for (i = 0; i < LEN_CHORD; i++)
     cout << nextChordArray[i] << " ";
   cout << "round = " << round << " and seq = "
        << seq + 1 << endl;
   cout << "---------------------------------------" << endl;

  }// end for round

}// end main



//---------------------------------------------------------------------------
//  TD.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Header file for TD.cpp
//
//---------------------------------------------------------------------------

#include <math.h>
#include "Constants.h"

class TD {

 public:
     TD();
```

```
      void   TDAlgorithm(int*, int*, int*, int*, int*, double*,
                         double*, int noteIndex, int round);
      int    GetState(int*);
      double GetValue(double*, double*, int*, double*, int*, int noteIndex);
      void   ComputeElig(int*, double*, int*, int noteIndex);
      double Critique(int*, int*, int*);
      void   ReplaceElig();
      void   Backprop();
      double ActFunc(double tempSum);
      double InitWeight();

  private:
      double alphaO;                       // Learning rate for output
      double alphaH;                       // Learning rate for hidden
      double gamma;                        // Discount rate
      double lambda;                       // Eligibility decay rate
      double delta;                        // Error

      double wB;                           // Output bias weight
      double w[NUM_HID_NODES];             // Output node weights
      double e[NUM_HID_NODES];             // Eligibility output

      double wNCB[NUM_HID_NODES];          // Hidden bias weight
      double wN[NUM_HID_NODES*LEN_NOTE];   // Note/State weights
      double wC[NUM_HID_NODES*LEN_NOTE];   // Chord weights
      double eN[NUM_HID_NODES][LEN_NOTE];  // Note eligibility
      double eC[NUM_HID_NODES][LEN_NOTE];  // Chord eligibility

      double wNIndex[NUM_HID_NODES][NUM_NOTES+1]; // Note Index weight
      double eNIndex[NUM_HID_NODES][NUM_NOTES+1]; // Note Index elig

      double hid[NUM_HID_NODES];           // Note hidden output
      double hidPrime[NUM_HID_NODES];      // Next note hidden output
      double fHid[NUM_HID_NODES];          // Squashed note hidden output
      double fHidPrime[NUM_HID_NODES];     // Next note squashed hidden
                                           //   output

      double r;                            // Reward
      int    freqNote[LEN_NOTE];           // How often note appears
};


//---------------------------------------------------------------------------
//  TD.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
```

```
//  Honors Thesis                         Smith College
//  fall, spring 2002                     Northampton, MA 01063
//
//  Non-linear TD:  This file implements the non-linear temporal-difference
//            learning algorithm of Richard Sutton (1988) for the
//            non-linear ordered sequence and counting experiments.
//
//---------------------------------------------------------------------------

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include "TD.h"

//=================================
//
//  Constructor
//
//=================================
TD::TD()
{
  int i = 0, j = 0;
  alphaO   = ALPHA_O;
  alphaH   = ALPHA_H;
  gamma    = GAMMA;
  lambda   = LAMBDA;
  delta    = 0;
  r        = 0;

  for (i = 0; i < LEN_NOTE; i++)
    freqNote[i] = 0;

  wB = InitWeight();

  for (j = 0; j < NUM_HID_NODES; j++)
    {
      for (i = 0; i < NUM_NOTES; i++)
        wNIndex[j][i] = InitWeight();

      for (i = 0; i < LEN_NOTE; i++)
        {
          wN[j*i]  = InitWeight();
          wC[j*i]  = InitWeight();
          eN[j][i] = 0;
          eC[j][i] = 0;
        }
```

```
      wNCB[j] = InitWeight();
      w[j]    = InitWeight();
      e[j]        = 0;
      hid[j]      = 0;
      hidPrime[j]  = 0;
      fHid[j]     = 0;
      fHidPrime[j] = 0;

    }// end j


}// end constructor



//==================================================
//
//  The TD() algorithm - with hidden units
//
//==================================================
void TD::TDAlgorithm(int* chord, int* nextChord, int* note, int* nextNote,
                     int* seq, double* noteBar, double* nextNoteBar,
                     int noteIndex, int round)
{
  int state = GetState(note);
  freqNote[state]++;

  //--- Compute V(state) ---
  double V = GetValue(hid, fHid, note, noteBar, chord, noteIndex);

  //--- Compute e(state) ---
  ComputeElig(note, noteBar, chord, noteIndex);

  //--- Compute V(statePrime) ---
  int* tempChord;
  if (noteIndex >= (NUM_NOTES-1))
    tempChord = nextChord;
  else
    tempChord = chord;

  double VPrime = GetValue(hidPrime, fHidPrime, nextNote, nextNoteBar,
                           tempChord, (noteIndex+1) % NUM_NOTES);

  //--- Get reward ---
  if (noteIndex >= NUM_NOTES-1)
    r = Critique(seq, chord, nextChord);
```

```
  //--- Find error delta ---
  delta = r + gamma * VPrime - V;

  //--- Backpropagation ---
  Backprop();

  //--- Replace eligibility traces ---
  if (noteIndex == 0)
    ReplaceElig();

  //--- Print ---
  if (round >= (NUM_ROUNDS - 100))
    cout << state << "\tV = " << V << "  reinforcement = "
         << r << "   noteIndex = " << noteIndex << endl;

  r = 0;

}// end TDAlgorithm

//==================================================================
//
//  GetState
//
//==================================================================
int TD::GetState(int* note)
{
  int i = 0;

  for (i = 0; i < LEN_NOTE; i++)
    if (note[i] == 1)
      break;

  return i;

}// end GetState

//==================================================================
//
//  GetValue
//
//==================================================================
double TD::GetValue(double* tempHid, double* tempSquashHid,
                    int* note, double* noteBar, int* chord,
                    int noteIndex)
{
  int NIndex[NUM_NOTES] = {0};
  NIndex[noteIndex] = 1;
```

```
  int i = 0, j = 0;

  for (j = 0; j < NUM_HID_NODES; j++) {

    //--- Sum from (i = 1 to n) of wi*si ---
    tempHid[j] = 0;

    #if NOTE
    for (i = 0; i < LEN_NOTE; i++)
      tempHid[j] += note[i] * wN[(j*LEN_NOTE)+i];
    #endif

    #if NOTE_BAR
    for (i = 0; i < LEN_NOTE; i++)
      tempHid[j] += noteBar[i] * wN[(j*LEN_NOTE)+i];
    #endif

    #if CHORD
    for (i = 0; i < LEN_NOTE; i++)
      tempHid[j] += chord[i] * wC[(j*LEN_NOTE)+i];
    #endif

    #if NINDEX
    for (i = 0; i < NUM_NOTES; i++)
      tempHid[j] += NIndex[i] * wNIndex[j][i];
    #endif

    tempSquashHid[j] = ActFunc(tempHid[j]);
  }

  //--- Calculate linear comb of Vs and HidPrimes for output node ---
  double value = 0;
  for (int j = 0; j < NUM_HID_NODES; j++)
    value += w[j] * tempSquashHid[j];

  return value;

}// end GetValue

//==================================================================
//
//  ComputeElig
//
//==================================================================
void TD::ComputeElig(int* note, double* noteBar, int* chord, int noteIndex)
{
  int j = 0, i = 0;
```

```
  int NIndex[NUM_NOTES] = {0};
  NIndex[noteIndex] = 1;
  for (j = 0; j < NUM_HID_NODES; j++) {
    for (i = 0; i < LEN_NOTE; i++) {
      #if NOTE
      eN[j][i] = lambda * eN[j][i] + w[j] * fHid[j] * (1-fHid[j]) * note[i];
      #endif

      #if NOTE_BAR
      eN[j][i] = lambda * eN[j][i] + w[j] * fHid[j] * (1-fHid[j]) * noteBar[i];
      #endif

      eC[j][i] = lambda * eC[j][i] + w[j] * fHid[j] * (1-fHid[j]) * chord[i];
    }
      for (i = 0; i < NUM_NOTES; i++)
        eNIndex[j][i] = lambda * eNIndex[j][i] + w[j] * fHid[j] * (1-fHid[j])
          * NIndex[i];

      e[j] = lambda * e[j] + fHid[j];
  }

}

//====================================================================
//
//  Critique:  the reward function
//
//====================================================================
double TD::Critique(int* seq, int* chord, int* nextChord)
{
  double tempReward = 0;

#if ORD_SEQ
  //--------------------------------------------------------------------
  //  Rule 1:  If there exists within the sequence the ordered notes,
  //         C,C#,D, then tempReward = RMAX, otherwise tempReward = 0;
  //--------------------------------------------------------------------
  // If the first digit of the first note is 1 and
  //    the second digit of the second note is 1 and
  //    the third digit of the third note is 1
  if (seq[0]==1 && seq[LEN_NOTE+1]==1 && seq[2*LEN_NOTE+2]==1)
    tempReward = 1;
#endif

#if COUNT
  //--------------------------------------------------------------------
  //  Rule 2:
```

```
  //-------------------------------------------------------------------
  if (seq[2]==1 && seq[LEN_NOTE+2]==1 && seq[2*LEN_NOTE+2]==1
      && seq[3*LEN_NOTE+2]==1 && seq[4*LEN_NOTE+2]==1)
    tempReward = 1;

#endif

  return tempReward;

}// end Critique

//=================================================================
//
//  Backprop
//
//=================================================================
void TD::Backprop()
{
  int j = 0, i = 0;

  for (j = 0; j < NUM_HID_NODES; j++) {
    for (i = 0; i < LEN_NOTE; i++) {
      wN[j*i] += alphaH * delta * eN[j][i];
      wC[j*i] += alphaH * delta * eC[j][i];
    }
    for (i = 0; i < NUM_NOTES; i++)
      wNIndex[j][i] += alphaH * delta * eNIndex[j][i];

    w[j]    += alphaO * delta * e[j];
    wNCB[j] += alphaH * delta * BIAS;
  }
  wB += alphaO * delta * BIAS;


}// end Backprop

//=================================================================
//
//  ReplaceElig: resets the eligibility traces for all of
//               the states/notes to zero.
//
//=================================================================
void TD::ReplaceElig()
{
  for (int j = 0; j < NUM_HID_NODES; j++) {
    for (int i = 0; i < LEN_NOTE; i++) {
      eN[j][i] = 0;
```

```
      eC[j][i] = 0;
    }

    for (int i = 0; i < NUM_NOTES; i++)
      eNIndex[j][i] = 0;

    e[j] = 0;

  }// end j

}// end ReplaceElig

//=====================================================
//
//  ActFunc
//
//  The sigmoid activation function as given by,
//
//      f(sum) = 1 / (1 + e^(-sum))
//
//=====================================================
double TD::ActFunc(double tempSum)
{
  return (1 / (1 + exp(-tempSum)));

}// end ActFunc

//=====================================================
//
//  InitWeight
//
//=====================================================
double TD::InitWeight()
{
  double newW = 10;

  if (drand48() > 0.5)
    while (newW < 0.001 || newW > 0.01)
      newW = drand48() / DENOM;
  else
    while (newW > -0.001 || newW < -0.01)
      newW = -drand48() / DENOM;

  return newW;

}// end InitWeight
```

```
//----------------------------------------------------------------------------
//  Constants.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Constants file for non-linear temporal-difference
//                 learning code for the non-linear ordered sequence
//                 and counting experiments.
//
//----------------------------------------------------------------------------

// Ordered Sequence
#define NUM_ROUNDS     300000
#define NUM_HID_NODES      60
#define LEN_SEQ            36
#define LEN_CHORD          12
#define LEN_NOTE           12
#define NUM_SEQS           30
#define NUM_CHORDS          3
#define NUM_NOTES           3
#define RMAX                1
#define ALPHA_O          0.02
#define ALPHA_H          0.02
#define BETA              0.6
#define GAMMA            0.95
#define LAMBDA            0.6
#define DENOM              20
#define BIAS                1

//----- Choose one of these -----
//#define BETA_TRACE         -1
//#define SOME_BETA_TRACE    -2
#define NO_BETA_TRACE      -3

//----- Choose one of these -----
#define ORD_SEQ          -4
//#define COUNT            -5

//----- Choose one of these -----
#define NOTE             -6
//#define NOTE_BAR         -7

//----- Choose some of these -----
```

```
//#define CHORD            -8
//#define NINDEX           -9

/*
// Counting
#define NUM_ROUNDS     100000
#define NUM_HID_NODES      60
#define LEN_SEQ            60
#define LEN_CHORD          12
#define LEN_NOTE           12
#define NUM_SEQS           20
#define NUM_CHORDS          3
#define NUM_NOTES           5
#define RMAX                1
#define ALPHA_O          0.01
#define ALPHA_H          0.01
#define BETA              0.6
#define GAMMA            0.95
#define LAMBDA            0.6
#define DENOM              3
#define BIAS                1

//----- Choose one of these -----
#define BETA_TRACE         -1
//#define SOME_BETA_TRACE    -2
//#define NO_BETA_TRACE      -3

//----- Choose one of these -----
//#define ORD_SEQ           -4
#define COUNT             -5

//----- Choose one of these -----
//#define NOTE              -6
#define NOTE_BAR          -7

//----- Choose some of these -----
//#define CHORD            -8
//#define NINDEX           -9
*/


//---------------------------------------------------------------------------
//  NotesOrdSeq.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
```

```
//  fall, spring 2002                  Northampton, MA 01063
//
//  Example and test sequences for non-linear temporal-difference learning.
//  Ordered sequence examples.
//---------------------------------------------------------------------------

//---------------------------------------------------------------------------
// OrdSeqi[LEN_NOTE*NUM_NOTES] makes up a state
//---------------------------------------------------------------------------
// C  = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
// CS = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
// D  = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
//---------------------------------------------------------------------------
int OrdSeq1[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq2[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq3[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq4[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq5[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq6[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq7[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq8[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq9[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq10[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

//---------------------------------------------------------------------
int OrdSeq11[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq12[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq13[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int OrdSeq14[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int OrdSeq15[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0};

int OrdSeq16[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};

int OrdSeq17[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int OrdSeq18[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int OrdSeq19[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq20[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0};
```

```
//--------------------------------------------------------------------------
int OrdSeq21[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq22[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq23[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                    0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int OrdSeq24[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0};

int OrdSeq25[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                    0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int OrdSeq26[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1};

int OrdSeq27[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1};

int OrdSeq28[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};

int OrdSeq29[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int OrdSeq30[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                    0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                    0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};

//--------------------------------------------------------------------------
int test_OrdSeq1[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int test_OrdSeq2[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
                                   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int test_OrdSeq3[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int test_OrdSeq4[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};

int test_OrdSeq5[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};



//-----------------------------------------------------------------------
//  NotesCount.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Example and test sequences for non-linear temporal-difference learning.
//  Counting sequence examples.
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
// CountSeqi[LEN_NOTE*NUM_NOTES] makes up a state
//-----------------------------------------------------------------------
int CountSeq1[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq2[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq3[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
int CountSeq4[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq5[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq6[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq7[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq8[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq9[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq10[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

//--------------------------------------------------------------------------------
int CountSeq11[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int CountSeq12[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq13[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int CountSeq14[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int CountSeq15[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq16[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0};

int CountSeq17[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int CountSeq18[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                   0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq19[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                   0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0};

int CountSeq20[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

//----------------------------------------------------------------------------
int CountSeq21[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int CountSeq22[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                     0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq23[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int CountSeq24[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0};

int CountSeq25[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                     0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq26[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                     0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                     0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};
```

```
int CountSeq27[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                      0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int CountSeq28[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                      0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int CountSeq29[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int CountSeq30[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                      0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                      0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                      0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0};

//---------------------------------------------------------------------------
// Test sequences
int testCountSeq1[LEN_NOTE*NUM_NOTES] = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                         0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                         0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                         0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                         0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int testCountSeq2[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                         0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                         0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                         0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                         0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

B.3 Non-linear TD(λ) for Context-switching

```
//---------------------------------------------------------------------------
//  Simulator.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Header file for Simulator.cpp
//
//---------------------------------------------------------------------------

#include "TD.h"
#include "Constants.h"
#include "NotesContextSwitch.h"
#include "NotesCharlieParker.h"

class Simulator {
  public:
  private:
};
```

```
//----------------------------------------------------------------------------
//  Simulator.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Main file that runs the non-linear temporal-difference
//            algorithm for the non-linear context-switching experiments.
//            Sequences to be learned are randomly chosen in this file.
//
//----------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include <string.h>
#include "Simulator.h"

//=================================================================
//
//  Main
//
```

```
//==================================================================
int main()
{
  int i = 0, j = 0;
  TD* tdLambda = new TD();
  int* seqs[NUM_SEQS] = {ConSwiSeq1,   ConSwiSeq2,   ConSwiSeq3,
                         ConSwiSeq4,   ConSwiSeq5,   ConSwiSeq6,
                         ConSwiSeq7,   ConSwiSeq8,   ConSwiSeq9,
                         ConSwiSeq10,  ConSwiSeq11,  ConSwiSeq12,
                         ConSwiSeq13,  ConSwiSeq14,  ConSwiSeq15,
                         ConSwiSeq16 , ConSwiSeq17,  ConSwiSeq18,
                         ConSwiSeq19,  ConSwiSeq20,  ConSwiSeq21,
                         ConSwiSeq22};
  int* chords[NUM_CHORDS] = {C8,C9,C7};

  //--- Chords ---
  int chord     = lrand48() % NUM_CHORDS;
  int nextChord = lrand48() % NUM_CHORDS;
  while (chord == nextChord)
    nextChord = lrand48() % NUM_CHORDS;
  int* chordArray     = chords[chord];
  int* nextChordArray = chords[nextChord];

  //--- Sequences ---
  int seq       = lrand48() % NUM_SEQS;
  int nextSeq   = lrand48() % NUM_SEQS;
  while (seq < 11 && chord == 2)
     seq = lrand48() % NUM_SEQS;
  while (nextSeq < 11 && nextChord == 2)
    nextSeq = lrand48() % NUM_SEQS;
  int* seqArray      = seqs[seq];
  int* nextSeqArray   = seqs[nextSeq];

  //--- Notes ---
  int* noteArray      = seqArray;
  int* nextNoteArray  = &seqArray[LEN_NOTE];

  //--- Average note trace ---
  double noteBar[LEN_NOTE]     = {0};
  double nextNoteBar[LEN_NOTE] = {0};
  for (i = 0; i < LEN_NOTE; i++) {
    noteBar[i] = noteArray[i];
    nextNoteBar[i] = nextNoteArray[i];
  }

  //--- Constant information ---
  cout << "# Number of rounds is                : " << NUM_ROUNDS    << endl;
```

```
cout << "# Number of hidden nodes is          : " << NUM_HID_NODES << endl;
cout << "# Length of a sequence is            : " << LEN_SEQ       << endl;
cout << "# Length of a note is                : " << LEN_NOTE      << endl;
cout << "# Length of a chord is               : " << LEN_CHORD     << endl;
cout << "# Number of sequences is             : " << NUM_SEQS      << endl;
cout << "# Number of notes in a sequence is   : " << NUM_NOTES     << endl;
cout << "# Number of chords is                : " << NUM_CHORDS    << endl;
cout << "# Maximum reward is                  : " << RMAX          << endl;
cout << "# ALPHA output node learning rate is : " << ALPHA_O       << endl;
cout << "# ALPHA hidden node learning rate is : " << ALPHA_H       << endl;
cout << "# GAMMA discount factor is           : " << GAMMA         << endl;
cout << "# LAMBDA is                          : " << LAMBDA        << endl;
cout << "# BIAS is                            : " << BIAS          << endl;

//-----------------------------------------------
//  Training/Learning
//-----------------------------------------------
for (int round = 0; round < NUM_ROUNDS; round++) {

  //--- Store the current and next chord ---
  chord = nextChord;
  chordArray = chords[chord];
  nextChord = lrand48() % NUM_CHORDS;
  while (chord == nextChord)
    nextChord = lrand48() % NUM_CHORDS;
  nextChordArray = chords[nextChord];

  //---- Store the current and next sequence ---
  seq = nextSeq;
  seqArray = seqs[seq];
  nextSeq = lrand48() % NUM_SEQS;
  while (nextSeq < 11 && nextChord == 2)
    nextSeq = lrand48() % NUM_SEQS;
  nextSeqArray = seqs[nextSeq];

  //--- Find the current note and the next note ---
  noteArray     = seqArray;
  nextNoteArray = &seqArray[LEN_NOTE];

  //--- Reset the average traces ---
  for (j = 0; j < LEN_NOTE; j++) {
    noteBar[j]     = noteArray[j];
    nextNoteBar[j] = nextNoteArray[j];
  }

  //-----------------------------------------------
  //  Learn note and chord
```

```
//--------------------------------------------
for (i = 0; i < NUM_NOTES; i++) {

  //--- Find the current note ---
  noteArray = &seqArray[i*LEN_NOTE];

  //--- Find the next note ---
  if (i == (NUM_NOTES - 1))
    nextNoteArray = &nextSeqArray[0];
  else
    nextNoteArray = &seqArray[(i+1)*LEN_NOTE];

  //--- Find noteBar and nextNoteBar ---
  for (j = 0; j < LEN_NOTE; j++) {
    noteBar[j]     = BETA * noteBar[j]     + (1-BETA) * noteArray[j];
    //noteBar[j]     = noteArray[j];
    //nextNoteBar[j] = BETA * nextNoteBar[j] + (1-BETA) * nextNoteArray[j];
    nextNoteBar[j] = nextNoteArray[j];
  }

  //--- Get the output from the neural net and learn using TD ---
  tdLambda->TDAlgorithm(chordArray, nextChordArray, noteArray, nextNoteArray,
                        seqArray, noteBar, nextNoteBar, i, round);

  //--- This next section of code implements giving the first note a 2nd time ---
  if (i == (NUM_NOTES - 1)) {

    //--- Find the current note and the next note ---
    noteArray = &nextSeqArray[0];
    int tempNext[LEN_NOTE];
    for (j = 0; j < LEN_NOTE; j++)
      tempNext[j] = 0;
    tempNext[lrand48() % LEN_NOTE] = 1;
    nextNoteArray = tempNext; //&nextSeqArray[LEN_NOTE];

    //--- Find noteBar and nextNoteBar ---
    for (j = 0; j < LEN_NOTE; j++) {
      noteBar[j]     = BETA * noteBar[j]     + (1-BETA) * noteArray[j];
      //noteBar[j]     = noteArray[j];
      //nextNoteBar[j] = BETA * nextNoteBar[j] + (1-BETA) * nextNoteArray[j];
      nextNoteBar[j] = nextNoteArray[j];
    }

    //--- Get the output from the neural net and learn using TD ---
    tdLambda->TDAlgorithm(nextChordArray, nextChordArray, noteArray, nextNoteArray,
                          seqArray, noteBar, nextNoteBar, i+1, round);
  }// end if i
```

```
    }// end for i

    //--------------------------------------------
    // Print the chord and round information
    //--------------------------------------------
    if (round >= (NUM_ROUNDS - 100)) {
      cout << "chord : " << endl;
      for (i = 0; i < LEN_CHORD; i++)
        cout << chordArray[i] << " ";
      cout << "\nnext chord : " << endl;
      for (i = 0; i < LEN_CHORD; i++)
        cout << nextChordArray[i] << " ";
      cout << "round = " << round << " and seq = "
           << seq + 1 << endl;
      cout << "-------------------------" << endl;
    }

  }// end for round
}// end main



//----------------------------------------------------------------------------
//  TD.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Header file for TD.cpp
//
//----------------------------------------------------------------------------

#include <math.h>
#include "Constants.h"

class TD {

  public:
       TD();
       void   TDAlgorithm(int*, int*, int*, int*, int*, double*, double*,
                          int noteIndex, int round);
       int    GetState(int*);
       double GetValue(double*, double*, double*, int*, int noteIndex);
       void   ComputeElig(double*, int*, int noteIndex);
       double Critique(int*, int*, int*);
       void   ReplaceElig();
```

```
      void   Backprop();
      double ActFunc(double tempSum);
      double InitWeight();

  private:
      double alphaO;                      // Learning rate for output
      double alphaH;                      // Learning rate for hidden
      double gamma;                       // Discount rate
      double lambda;                      // Eligibility decay rate
      double delta;                       // Error

      double wB;                          // Output bias weight
      double w[NUM_HID_NODES];            // Output node weights
      double e[NUM_HID_NODES];            // Output eligibility

      double wNCB[NUM_HID_NODES];         // Hidden bias weight
      double wN[NUM_HID_NODES*LEN_NOTE];  // Note/State weights
      double wC[NUM_HID_NODES*LEN_NOTE];  // Chord weights
      double eN[NUM_HID_NODES][LEN_NOTE]; // Note eligibility
      double eC[NUM_HID_NODES][LEN_NOTE]; // Chord eligibility

      double wNIndex[NUM_HID_NODES][NUM_NOTES+1]; // Note Index weight
      double eNIndex[NUM_HID_NODES][NUM_NOTES+1]; // Note Index elig

      double hid[NUM_HID_NODES];          // Note hidden output
      double hidPrime[NUM_HID_NODES];     // Next note hidden output
      double fHid[NUM_HID_NODES];         // Squashed note hidden output
      double fHidPrime[NUM_HID_NODES];    // Next note squashed hidden
                                          //   output

      double r;                           // Reward
      double stored_r;                    // Stored reward for use later

      int    freqNote[LEN_NOTE];          // How often note appears
};


//----------------------------------------------------------------------------
//  TD.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  This file implements the non-linear temporal-difference
//            learning algorithm of Richard Sutton (1988) for the
```

```
//            non-linear ordered sequence and counting experiments.
//
//-----------------------------------------------------------------------------

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include "TD.h"

//==================================
//
//  Constructor
//
//==================================
TD::TD()
{
  int i = 0, j = 0;
  alphaO   = ALPHA_O;
  alphaH   = ALPHA_H;
  gamma    = GAMMA;
  lambda   = LAMBDA;
  delta    = 0;
  r        = 0;
  stored_r = 0;

  for (i = 0; i < LEN_NOTE; i++)
    freqNote[i] = 0;

  wB = InitWeight();
  for (j = 0; j < NUM_HID_NODES; j++) {
    for (i = 0; i < NUM_NOTES; i++)
      wNIndex[j][i] = InitWeight();
    for (i = 0; i < LEN_NOTE; i++) {
      wN[j*i]  = InitWeight();
      wC[j*i]  = InitWeight();
      eN[j][i] = 0;
      eC[j][i] = 0;
    }
    wNCB[j] = InitWeight();
    w[j]    = InitWeight();
    e[j]         = 0;
    hid[j]       = 0;
    hidPrime[j]  = 0;
    fHid[j]      = 0;
    fHidPrime[j] = 0;
  }// end j
```

```
}// end constructor

//==================================================
//
//  The TD() algorithm - with hidden units
//
//==================================================
void TD::TDAlgorithm(int* chord, int* nextChord, int* note, int* nextNote,
                     int* seq, double* noteBar, double* nextNoteBar,
                     int noteIndex, int round)
{
  int state = GetState(note);
  freqNote[state]++;

  //--- Compute V(state) ---
  double V = GetValue(hid, fHid, noteBar, chord, noteIndex);

  //--- Compute e(state) ---
  ComputeElig(noteBar, chord, noteIndex);

  //--- Compute V(statePrime) ---
  int* tempChord;
  if (noteIndex >= (NUM_NOTES-1))
    tempChord = nextChord;
  else
    tempChord = chord;

  double VPrime = GetValue(hidPrime, fHidPrime, nextNoteBar, tempChord,
                           (noteIndex+1) % (NUM_NOTES+1)); // for a(s)
                           //(noteIndex+1) % (NUM_NOTES));   // for s

  //--- Get reward ---
  if (noteIndex >= NUM_NOTES-1)
    stored_r = Critique(seq, chord, nextChord);

  //--- Use reward ---
  if (noteIndex == 3)
    r = Critique(seq, chord, nextChord); //stored_r;

  //--- Find error delta ---
  delta = r + gamma * VPrime - V;

  //--- Backpropagation ---
  Backprop();

  //--- Replace eligibility traces ---
  //if (noteIndex == 2) // for s
```

```
  if (noteIndex == 3) // for a(s)
    ReplaceElig();

  //--- Print ---
  if (round >= (NUM_ROUNDS - 100))
    cout << state << "\tV = " << V << "  reinforcement = "
         << r << "   noteIndex = " << noteIndex << endl;

  r = 0;

}// end TDAlgorithm

//=====================================================================
//
//  GetState
//
//=====================================================================
int TD::GetState(int* note)
{
  int i = 0;

  for (i = 0; i < LEN_NOTE; i++)
    if (note[i] == 1)
      break;

  return i;

}// end GetState

//=====================================================================
//
//  GetValue
//
//=====================================================================
double TD::GetValue(double* tempHid, double* tempSquashHid,
                    double* noteBar, int* chord, int noteIndex)
{
  int NIndex[NUM_NOTES] = {0};
  NIndex[noteIndex] = 1;
  int i = 0, j = 0;

  for (j = 0; j < NUM_HID_NODES; j++) {

    //--- Sum from (i = 1 to n) of wi*si ---
    tempHid[j] = 0; //BIAS * tempWeightBias;
    for (i = 0; i < LEN_NOTE; i++)
      tempHid[j] += noteBar[i] * wN[(j*LEN_NOTE)+i]
```

```
        + chord[i] * wC[(j*LEN_NOTE)+i];
    tempSquashHid[j] = ActFunc(tempHid[j]);
  }

  //--- Calculate linear comb of weights and hids for output
  double value = 0;
  for (int j = 0; j < NUM_HID_NODES; j++)
    value += w[j] * tempSquashHid[j];

  return value;
}// end GetValue


//===================================================================
//
//  ComputeElig
//
//===================================================================
void TD::ComputeElig(double* noteBar, int* chord, int noteIndex)
{
  int j = 0, i = 0;
  int NIndex[NUM_NOTES] = {0};
  NIndex[noteIndex] = 1;
  for (j = 0; j < NUM_HID_NODES; j++) {
    for (i = 0; i < LEN_NOTE; i++) {
      eN[j][i] = lambda * eN[j][i] + w[j] * fHid[j] * (1-fHid[j]) * noteBar[i];
      eC[j][i] = lambda * eC[j][i] + w[j] * fHid[j] * (1-fHid[j]) * chord[i];
    }
      for (i = 0; i < NUM_NOTES; i++)
        eNIndex[j][i] = lambda * eNIndex[j][i] + w[j] * fHid[j] * (1-fHid[j])
          * NIndex[i];

      e[j] = lambda * e[j] + fHid[j];
  }
}

//===================================================================
//
//  Critique:  the reward function
//
//===================================================================
double TD::Critique(int* seq, int* chord, int* nextChord)
{
  double tempReward = 0;

  //----------------------------------------------------------------------
  // Rule #1
```

```
  //
  // If 0 is the last note in the sequence or the second to last note in
  //   the sequence and the next chord is a C7, then reward of 1,
  //   otherwise reward of 0
  //
  // C7 chord is 1,0,0,0,1,0,0,1,0,0,1,0
  //
  //-----------------------------------------------------------------------
  if (nextChord[0]==1 && nextChord[4]==1 && nextChord[7]==1 && nextChord[10]==1)
    if (seq[(NUM_NOTES-1)*LEN_NOTE]==1)
      tempReward = 1;

  return tempReward;

}// end Critique

//=================================================================
//
//  Backprop
//
//=================================================================
void TD::Backprop()
{
  int j = 0, i = 0;

  for (j = 0; j < NUM_HID_NODES; j++) {
    for (i = 0; i < LEN_NOTE; i++) {
      wN[j*i] += alphaH * delta * eN[j][i];
      wC[j*i] += alphaH * delta * eC[j][i];
    }
    for (i = 0; i < NUM_NOTES; i++)
      wNIndex[j][i] += alphaH * delta * eNIndex[j][i];

    w[j]    += alphaO * delta * e[j];
    wNCB[j] += alphaH * delta * BIAS;
  }
  wB += alphaO * delta * BIAS;

}// end Backprop

//=================================================================
//
//  ReplaceElig: resets the eligibility traces for all of
//              the states/notes to zero.
//
//=================================================================
void TD::ReplaceElig()
```

```
{
  for (int j = 0; j < NUM_HID_NODES; j++) {
    for (int i = 0; i < LEN_NOTE; i++) {
      eN[j][i] = 0;
      eC[j][i] = 0;
    }
    for (int i = 0; i < NUM_NOTES; i++)
      eNIndex[j][i] = 0;

    e[j] = 0;

  }// end j

}// end ReplaceElig

//=====================================================
//
//  ActFunc
//
//  The sigmoid activation function as given by,
//
//      f(sum) = 1 / (1 + e^(-sum))
//
//=====================================================
double TD::ActFunc(double tempSum)
{
  return (1 / (1 + exp(-tempSum)));

}// end ActFunc

//=====================================================
//
//  InitWeight
//
//=====================================================
double TD::InitWeight()
{
  double newW = 10;

  if (drand48() > 0.5)
    while (newW < 0.001 || newW > 0.01)
      newW = drand48() / DENOM;
  else
    while (newW > -0.001 || newW < -0.01)
      newW = -drand48() / DENOM;

  return newW;
```

```
}// end InitWeight


//-------------------------------------------------------------------------
//  Constants.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear TD:  Constants file for non-linear temporal-difference
//                 learning code for the context-switching experiments.
//
//-------------------------------------------------------------------------

#define NUM_ROUNDS      60000
#define NUM_HID_NODES      60
#define LEN_SEQ            36
#define LEN_CHORD          12
#define LEN_NOTE           12
#define NUM_SEQS           22
#define NUM_CHORDS          3
#define NUM_NOTES           3
#define RMAX                1
#define ALPHA_O          0.02
#define ALPHA_H          0.02
#define BETA              0.6
#define GAMMA            0.95
#define LAMBDA            0.6
#define DENOM              20
#define BIAS                1



//-------------------------------------------------------------------------
//  NotesContextSwitch.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Example and test sequences for non-linear temporal-difference learning.
//  Countext switching sequence examples.
//-----------------------------------------------------------------------
//-----------------------------------------------------------------------
```

```
// ConSwiSeqi[LEN_NOTE*NUM_NOTES] makes up a state
//-------------------------------------------------------------------
// C  = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
// CS = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
// D  = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0}
//-------------------------------------------------------------------
int ConSwiSeq1[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                       0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq2[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                       0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq3[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                       0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq4[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                       0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq5[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                       0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq6[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                       0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq7[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                       0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq8[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                       0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq9[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                       1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq10[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                        1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq11[LEN_NOTE*NUM_NOTES] = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
```

```
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                        1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

//-----------------------------------------------------------------------
int ConSwiSeq12[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq13[LEN_NOTE*NUM_NOTES] = {0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                        0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq14[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                        0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq15[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0};

int ConSwiSeq16[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0,
                                        0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq17[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                        0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq18[LEN_NOTE*NUM_NOTES] = {0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0};

int ConSwiSeq19[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int ConSwiSeq20[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                        0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0};

int ConSwiSeq21[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int ConSwiSeq22[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0,
                                        0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                        0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

```
//-----------------------------------------------------------------------------
int test_ConSwiSeq1[LEN_NOTE*NUM_NOTES] = {1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                           0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                           0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0};

int test_ConSwiSeq2[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                           0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0,
                                           0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int test_ConSwiSeq3[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0,
                                           1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
                                           0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0};

int test_ConSwiSeq4[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
                                           0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0,
                                           0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0};

int test_ConSwiSeq5[LEN_NOTE*NUM_NOTES] = {0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                           0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0,
                                           0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0};
```

B.4 Linear Sarsa(λ)

```
//-------------------------------------------------------------------------
//  Simulator.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear Sarsa:  Header file for Simulator.cpp
//
//-------------------------------------------------------------------------

#include "Sarsa.h"
#include "Constants.h"

class Simulator {
  public:
  private:
};
```

```
//-------------------------------------------------------------------------
//  Simulator.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear Sarsa:  Main file that runs the linear Sarsa algorithm
//
//-------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include <string.h>
#include "Simulator.h"

//================================================================
//
//  Main
//
//================================================================
int main()
{
  Sarsa*  tdLambda  = new Sarsa();
```

```
  cout << "# Number of rounds is                : " << NUM_ROUNDS    << endl;
  cout << "# Length of a sequence is            : " << LEN_SEQ       << endl;
  cout << "# Length of a note is                : " << LEN_NOTE      << endl;
  cout << "# Length of a chord is               : " << LEN_CHORD     << endl;
  cout << "# Number of sequences is             : " << NUM_SEQS      << endl;
  cout << "# Number of notes in a sequence is   : " << NUM_NOTES     << endl;
  cout << "# Number of chords is                : " << NUM_CHORDS    << endl;
  cout << "# Maximum reward is                  : " << RMAX          << endl;
  cout << "# Initial exploration rate is        : " << INIT_EXPLORE  << endl;
  cout << "# Ending exploration rate is         : " << END_EXPLORE   << endl;
  cout << "# ALPHA learning rate is             : " << ALPHA         << endl;
  cout << "# GAMMA discount factor is           : " << GAMMA         << endl;
  cout << "# LAMBDA is                          : " << LAMBDA        << endl;

  //----------------------------------------------------------------------------
  //  Training/Learning/Testing
  //----------------------------------------------------------------------------
  tdLambda->SarsaAlgorithm();

}// end main



//-------------------------------------------------------------------------
//  Sarsa.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear Sarsa:  Header file for Sarsa.cpp
//
//-------------------------------------------------------------------------

#include <math.h>
#include "Constants.h"

class Sarsa {

  public:
       Sarsa();
       void   SarsaAlgorithm();
       int    GetState(int*);
       void   FindQAction(double*, int*, int*);
       double FindMaxAction(int*, int*, int round);
       double Critique(int*, int*);
```

```
        void   ReplaceElig();
        void   UpdateElig(int*, int*);
        double ActFunc(double sum);
        double InitWeight();
        void   UpdateWeights();

   private:
        double r;                 // Reward
        double delta;             // Error
        double alpha;             // Learning rate
        double gamma;             // Discount rate
        double lambda;            // Eligibility decay rate

        double wN[LEN_NOTE];     // Note/State weights
        double wA[LEN_NOTE];     // Action weights
        double eN[LEN_NOTE];     // Note eligibility
        double eA[LEN_NOTE];     // Action eligibility
        double Qa[LEN_NOTE];     // Q-value for each action

        int    s[LEN_NOTE];      // State
        int    a[LEN_NOTE];      // Action
        int    sPrime[LEN_NOTE]; // Next state
        int    aPrime[LEN_NOTE]; // Next action

        double maxQa;            // Max Q-value for state
        double maxQaPrime;       // Max Q-value for next state
        int    aSeq[NUM_NOTES];  // Actions played to form seq
};


//-------------------------------------------------------------------------
//  Sarsa.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear Sarsa:  This file implements the linear Sarsa
//               algorithm found in Sutton and Barto (1998).
//
//-------------------------------------------------------------------------

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include "Sarsa.h"
```

```
//=================================
//
//  Constructor
//
//=================================
Sarsa::Sarsa()
{
  int i  = 0;
  r      = 0;
  delta  = 0;
  alpha  = ALPHA;
  gamma  = GAMMA;
  lambda = LAMBDA;

  maxQa      = 0;
  maxQaPrime = 0;

  for (i = 0; i < LEN_NOTE; i++)
    {
      wN[i] = InitWeight();
      wA[i] = InitWeight();
      eN[i] = 0;
      eA[i] = 0;
      Qa[i] = 0;
      s[i]  = 0;
      a[i]  = 0;
      aSeq[i]   = 0;
      sPrime[i] = 0;
      aPrime[i] = 0;
    }

  s[lrand48()%LEN_NOTE] = 1;
  a[lrand48()%LEN_NOTE] = 1;

}

//==================================================
//
//  The Sarsa(lambda) algorithm
//
//==================================================
void Sarsa::SarsaAlgorithm()
{
  int i = 0, j = 0, k = 0;

  //--- For each a, find Q_a ---
```

```
FindQAction(Qa, s, a);

//--- Find a corresponding to the max Q_a ---
maxQa = FindMaxAction(a, aSeq, 0);

for (k = 0; k < NUM_ROUNDS; k++) {
  for (j = 0; j < NUM_NOTES; j++)
    {
      //--- Update eligibilities ---
      UpdateElig(s, a);

      //--- Take action a, observe reward r, and next state s' ---
      if (j == (NUM_NOTES - 1))
        r = Critique(s, aSeq);
      else
        r = 0;

      //--- Next state is current action ---
      for (i = 0; i < LEN_NOTE; i++)
        sPrime[i] = a[i];

      //--- For each a, find Q_a ---
      FindQAction(Qa, sPrime, a);

      //--- Find a corresponding to the max Q_a ---
      maxQaPrime = FindMaxAction(aPrime, &aSeq[j], k);

      //--- Calculate error delta ---
      delta = r + gamma * maxQaPrime - maxQa;

      if (k > (NUM_ROUNDS - 100)) {
        cout << "round = " << k << endl;
        cout << "r = " << r << endl;
        cout << "gamma * maxQaPrime = " << gamma * maxQaPrime << endl;
        cout << "maxQa = " << maxQa << endl;
        cout << "delta = " << delta << endl;
      }

      //--- Update weights ---
      UpdateWeights();

      //--- Update current action to be next action ---
      for (i = 0; i < LEN_NOTE; i++) {
        a[i] = aPrime[i];
        s[i] = sPrime[i];
      }
```

```
        //--- Update maximum Q-value for state ---
        maxQa = maxQaPrime;
      }

    if (k > (NUM_ROUNDS - 100))
      {
        cout << "round = " << k << endl;
        for (i = 0; i < NUM_NOTES; i++)
          cout << "On note index " << i << " the note/action played was: "
               << aSeq[i] << endl;
      }
    ReplaceElig();

  }// end for k

}

//=====================================================================
//
//  GetState
//
//=====================================================================
int Sarsa::GetState(int* note)
{
  int i = 0;

  for (i = 0; i < LEN_NOTE; i++)
    if (note[i] == 1)
      break;

  return i;
}

//=====================================================================
//
//  FindQAction
//
//=====================================================================
void Sarsa::FindQAction(double* tempQa, int* tempN, int* tempA)
{
  for (int i = 0; i < LEN_NOTE; i++)
    tempQa[i] = wN[i] + wA[i];
}

//=====================================================================
//
//  FindMaxQAction
```

```
//
//=====================================================================
double Sarsa::FindMaxAction(int* tempa, int* tempaSeq, int round)
{
  double tempQa = -1000000;
  int indexQa = lrand48() % LEN_NOTE;

  //--- Print the Q-values ---
  if (round > (NUM_ROUNDS - 100)) {
    for (int k = 0; k < LEN_NOTE; k++)
      cout << Qa[k] << "  ";
    cout << endl;
  }

  //--- Find best action ---
  for (int i = 0; i < LEN_NOTE; i++) {
    tempa[i] = 0;
    if (tempQa < Qa[i]) {
      tempQa = Qa[i];
      indexQa = i;
     }
  }

  //--- Exploration vs Exploitation
  double explore = INIT_EXPLORE;
  if (round > (NUM_ROUNDS - 100))
    explore = END_EXPLORE;
  if (drand48() < explore)
    indexQa = lrand48()%LEN_NOTE;
  tempa[indexQa] = 1;

  //--- Store chosen action ---
  tempaSeq[0] = indexQa;

  return Qa[indexQa];
}

//=====================================================================
//
//  Critique:  the reward function
//
//=====================================================================
double Sarsa::Critique(int* seq, int* aSeq)
{
  //---------------------------------------------------------
  // Rule #1
  //            If  0 1 2  then  r = 1
```

```
  //------------------------------------------------------
  /* if (seq[0]==1 || seq[LEN_NOTE+1]==1 || seq[2*LEN_NOTE+2]==1)
    r = 1;
  else
    r = 0;
  */

  //------------------------------------------------------
  // Rule #2
  //            If  1 2  then  r = 1
  //------------------------------------------------------
  if (seq[LEN_NOTE+1]==1 && seq[2*LEN_NOTE+2]==1)
    r = 1;
  else
    r = 0;

  return r;
}

//================================================================
//
//  ReplaceElig: resets the eligibility traces for all of
//             the states/notes to zero.
//
//================================================================
void Sarsa::ReplaceElig()
{

  for (int i = 0; i < LEN_NOTE; i++) {
    eN[i] = 0;
    eA[i] = 0;
  }

  r = 0;
}

//================================================================
//
//  UpdateElig
//
//================================================================
void Sarsa::UpdateElig(int* stemp, int* atemp)
{
  for (int i = 0; i < LEN_NOTE; i++) {
      eN[i] = lambda * eN[i] + stemp[i];
      eA[i] = lambda * eA[i] + atemp[i];
  }
```

```
}

//=====================================================
//
//  ActFunc
//
//  The sigmoid activation function as given by,
//
//      f(sum) = 1 / (1 + e^(-sum))
//
//=====================================================
double Sarsa::ActFunc(double sum)
{
  return (1 / (1 + exp(-sum)));
}

//=====================================================
//
//  InitWeight
//
//=====================================================
double Sarsa::InitWeight()
{
  double newWeight = 10;

  if (drand48() > 0.5)
    while (newWeight < 0.01 || newWeight > 0.1)
      newWeight = drand48()/DENOM;
  else
    while (newWeight > -0.01 || newWeight < -0.1)
      newWeight = -drand48()/DENOM;

  return newWeight;
}

//=====================================================
//
//  UpdateWeight
//
//=====================================================
void Sarsa::UpdateWeights()
{
  for (int i = 0; i < LEN_NOTE; i++) {
    wN[i] = wN[i] + alpha * delta * eN[i];
    wA[i] = wA[i] + alpha * delta * eA[i];
  }
}
```

```
//-----------------------------------------------------------------------
//  Constants.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Linear Sarsa:  Constants file for linear Sarsa code.
//
//-----------------------------------------------------------------------

#define NUM_ROUNDS   10000000
#define LEN_SEQ            36
#define LEN_CHORD          12
#define LEN_NOTE           12
#define NUM_SEQS           20
#define NUM_CHORDS          1
#define NUM_NOTES           3
#define RMAX                1
#define ALPHA            0.01
#define GAMMA            0.95
#define LAMBDA            0.6
#define DENOM               1
#define INIT_EXPLORE        1
#define END_EXPLORE       0.1
```

```
#-------------------------------------------------------------
# Makefile
#
# V. Manfredi and J. Franklin        Dept of Computer Science
# Honors Thesis                      Smith College
# fall, spring 2002                  Northampton, MA 01063
#
# Makefile for linear and non-linear Sarsa code
#
#-------------------------------------------------------------
all: Simulate

Simulator.o: Simulator.cpp Simulator.h
	g++ -g -Wall -c Simulator.cpp

Sarsa.o: Sarsa.cpp Sarsa.h
```

```
	g++ -g -Wall -c Sarsa.cpp

Simulate: Simulator.o Sarsa.o
	g++ -g -o Simulate Simulator.o Sarsa.o
```

B.5 Non-linear Sarsa(λ)

```
//-----------------------------------------------------------------------------
//  Simulator.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear Sarsa:  Header file for Simulator.cpp
//
//-----------------------------------------------------------------------------

#include "Sarsa.h"
#include "Constants.h"

class Simulator {
  public:
  private:
};



//-----------------------------------------------------------------------------
//  Simulator.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear Sarsa:  Main file that runs the non-linear Sarsa algorithm
//
//-----------------------------------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <iostream.h>
#include <math.h>
#include <string.h>
#include "Simulator.h"

//==================================================================
//
//  Main
//
//==================================================================
int main()
{
  Sarsa*  tdLambda  = new Sarsa();
```

```
  cout << "# Number of rounds is               : " << NUM_ROUNDS    << endl;
  cout << "# Number of hidden nodes is         : " << NUM_HID_NODES << endl;
  cout << "# Length of a sequence is           : " << LEN_SEQ       << endl;
  cout << "# Length of a note is               : " << LEN_NOTE      << endl;
  cout << "# Length of a chord is              : " << LEN_CHORD     << endl;
  cout << "# Number of sequences is            : " << NUM_SEQS      << endl;
  cout << "# Number of notes in a sequence is  : " << NUM_NOTES     << endl;
  cout << "# Number of chords is               : " << NUM_CHORDS    << endl;
  cout << "# Maximum reward is                 : " << RMAX          << endl;
  cout << "# ALPHA output node learning rate is : " << ALPHA_O      << endl;
  cout << "# ALPHA hidden node learning rate is : " << ALPHA_H      << endl;
  cout << "# GAMMA discount factor is          : " << GAMMA         << endl;
  cout << "# LAMBDA is                         : " << LAMBDA        << endl;
  cout << "# BIAS is                           : " << BIAS          << endl;

  //-----------------------------------------------------------------------
  //  Training/Learning
  //-----------------------------------------------------------------------
  tdLambda->SarsaAlgorithm();

}// end main



//-----------------------------------------------------------------------
//  Sarsa.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear Sarsa:  Header file for Sarsa.cpp
//
//-----------------------------------------------------------------------

#include <math.h>
#include "Constants.h"

class Sarsa {

  public:
       Sarsa();
       void   SarsaAlgorithm();
       int    GetState(int*);
       double GetValue(double*, double*, double*, int*, int noteIndex);
       double FindMaxA(int*, int*, int round, int noteIndex);
```

```
    void   ComputeElig(double*, int*, int*, int noteIndex);
    double Critique(double*, int*, int*);
    void   ReplaceElig();
    void   UpdateElig(double*, int*, int*, int noteIndex);
    void   Backprop();
    double InitWeight();
    double ActFunc(double tempSum);

private:
    double alphaO;                       // Output Learning rate
    double alphaH;                       // Hidden Learning rate
    double gamma;                        // Discount rate
    double lambda;                       // Eligibility decay rate
    double delta;                        // Error

    //--- Output nodes ---
    double wBO;                          // Output bias weight
    double w[NUM_HID_NODES];             // Output node weights
    double e[NUM_HID_NODES];             // Output eligibility

    //--- Hidden nodes ---
    double wBH[NUM_HID_NODES];           // Hidden bias weight

    double wN[NUM_HID_NODES][LEN_NOTE];  // Note/State weights
    double eN[NUM_HID_NODES][LEN_NOTE];  // Note eligibility
    double sN[LEN_NOTE];                 // State
    double sNPrime[LEN_NOTE];            // Next state

    double wC[NUM_HID_NODES][LEN_NOTE];  // Chord weights
    double eC[NUM_HID_NODES][LEN_NOTE];  // Chord eligibility
    int    sC[LEN_NOTE];                 // State
    int    sCPrime[LEN_NOTE];            // Next state

    double wA[NUM_HID_NODES][LEN_NOTE];  // Action weights
    double eA[NUM_HID_NODES][LEN_NOTE];  // Action eligibility
    int    a[LEN_NOTE];                  // Action
    int    aPrime[LEN_NOTE];             // Next action

    double wI[NUM_HID_NODES][NUM_NOTES]; // Note Index weights
    double eI[NUM_HID_NODES][NUM_NOTES]; // Note Index elig

    double hid[NUM_HID_NODES];          // Note hidden output
    double hidPrime[NUM_HID_NODES];     // Next note hidden output
    double fHid[NUM_HID_NODES];         // Squashed note hidden output
    double fHidPrime[NUM_HID_NODES];    // Next note squashed hidden
                                        //   output
```

```
       double Qa[LEN_NOTE];                // Q-value for each action
       int    aSeq[NUM_NOTES];             // Actions played to form seq

       double r;                           // Reward
       double stored_r;                    // Stored reward for use later
};



//--------------------------------------------------------------------------
//  Sarsa.cpp
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear Sarsa:  This file implements a modified version of the
//                   Sarsa algorithm found in Sutton and Barto (1998).
//
//--------------------------------------------------------------------------

#include <iostream.h>
#include <stdio.h>
#include <stdlib.h>
#include "Sarsa.h"

//==================================
//
//  Constructor
//
//==================================
Sarsa::Sarsa()
{
  int i = 0, j = 0;
  r        = 0;
  stored_r = 0;
  delta    = 0;
  alphaO   = ALPHA_O;
  alphaH   = ALPHA_H;
  gamma    = GAMMA;
  lambda   = LAMBDA;

  wBO = InitWeight();                  // Output bias

  for (j = 0; j < NUM_HID_NODES; j++)
    {
      for (i = 0; i < LEN_NOTE; i++)
```

```
        {
          wN[j][i] = InitWeight();    // Hidden weight - note (state)
          eN[j][i] = 0;               // Hidden elig - note
          wC[j][i] = InitWeight();    // Hidden weight - chord (state)
          eC[j][i] = 0;               // Hidden elig - chord
          wA[j][i] = InitWeight();    // Hidden weight - action
          eA[j][i] = 0;               // Hidden elig - action
        }
      for (i = 0; i < NUM_NOTES; i++)
        {
          wI[j][i] = InitWeight();    // Hidden weight - note index (state)
          eI[j][i] = 0;               // Hidden elig - note index
        }
      wBH[j]  = InitWeight();         // Hidden bias
      w[j]    = InitWeight();         // Output weight
      e[j]         = 0;               // Output elig
      hid[j]       = 0;               // Hidden output - current state
      hidPrime[j]  = 0;               // Hidden output - next state
      fHid[j]      = 0;               // Squashed hidden output - current state
      fHidPrime[j] = 0;               // Squashed hidden output - next state
    }// end j

  for (i = 0; i < LEN_NOTE; i++)
    {
      Qa[i]        = 0;               // Q values for state action pairs
      sN[i]        = 0;               // Hidden input - note
      sNPrime[i]   = 0;               // Hidden input - next note
      sC[i]        = 0;               // Hidden input - chord
      sCPrime[i]   = 0;               // Hidden input - next chord
      a[i]         = 0;               // Hidden input - action
      aPrime[i]    = 0;               // Hidden input - next action
    }

  for (i = 0; i < NUM_NOTES; i++)     //
    aSeq[i] = 0;                      // Action sequence:  what notes did Sarsa
                                      //   choose to play?

  sN[lrand48()%LEN_NOTE] = 1;         //
  sC[lrand48()%LEN_NOTE] = 1;         //  Initial state and action
  a[lrand48()%LEN_NOTE]  = 1;         //
}

//=====================================================
//
//  The Sarsa(lambda) algorithm - with hidden units
//
//=====================================================
```

```
void Sarsa::SarsaAlgorithm()
{
  int    i = 0, j = 0, k = 0;
  double testValue = 0;
  double testhid[NUM_HID_NODES]  = {0};
  double testfhid[NUM_HID_NODES] = {0};
  double maxQa = 0, maxQaPrime = 0;

  //--- Find a corresponding to the max Q_a ---
  maxQa = FindMaxA(a, aSeq, 0, 0);

  //------------------------------------------------------------
  // Choose notes for many sequences
  //------------------------------------------------------------
  for (k = 0; k < NUM_ROUNDS; k++)
    {
      //--------------------------------------------------------
      // Choose notes for one sequence
      //--------------------------------------------------------
      for (j = 0; j < NUM_NOTES; j++)
        {
          //--- Update eligibilities ---
          UpdateElig(sN, sC, a, j);

          //--- Take action a, observe reward r, and next state, s' ---
          if (j == (NUM_NOTES - 1))
            r = Critique(sN, sC, aSeq);
          else
            r = 0;

          //--- Make next state the current action ---
          for (i = 0; i < LEN_NOTE; i++)
            sNPrime[i] = a[i];

          //--- Find a corresponding to the max Q_a ---
          maxQaPrime = FindMaxA(aPrime, &aSeq[j], k, j);

          //--- Set hidden nodes' outputs and squashed outputs for ---
          //       use in backpropagation weight updating
          GetValue(hid, fHid, sN, sC, j);

          //--- Calculate error delta ---
          delta = r + gamma*maxQaPrime - maxQa;

          //--- Print ---
          if (k > (NUM_ROUNDS - SOME_ROUND)) {
            cout << "----------------------------------------------" << endl;
```

```
  cout << "index = " << j << endl;
  cout << "maxQaPrime = " << maxQaPrime << endl;
  cout << "maxQa = " << maxQa << endl;
  cout << "delta = " << delta << endl;
  cout << "round = " << k << endl;
  cout << "r = " << r << endl;
  cout << "gamma * maxQaPrime = " << gamma * maxQaPrime << endl;
}

//--- Backpropagate error ---
Backprop();

//--- Update current action/state to be next action/state ---
for (i = 0; i < LEN_NOTE; i++)
  {
    a[i] = aPrime[i];

    #if STATE_INPUT
    sN[i] = sNPrime[i];
    #endif

    #if AVG_INPUT
    sN[i] = BETA * sN[i] + (1-BETA) * sNPrime[i];
    #endif
  }

#if STATE_INPUT
//--- Update current Q_a to be nextt Q_a ---
maxQa = maxQaPrime;
#endif

#if AVG_INPUT
//--- Update current Q_a to be nextt Q_a ---
for (int m = 0; m < NUM_HID_NODES; m++)
  {
    testhid[m] = 0;
    for (int n = 0; n < LEN_NOTE; n++)
      testhid[m] += sN[n]*wN[m][n];
    testfhid[m] = ActFunc(testhid[m]);
  }
testValue = 0;
for (int p = 0; p < NUM_HID_NODES; p++)
  testValue += w[p]*testfhid[p];
maxQa = testValue;
#endif

    }// end for j
```

```
      #if AVG_INPUT
      for (i = 0; i < LEN_NOTE; i++)
        sN[i] = 0;
      #endif


      //--- Print notes chosen to be played ---
      if (k > (NUM_ROUNDS - SOME_ROUND))
        {
          cout << "round = " << k << endl;
          for (i = 0; i < NUM_NOTES; i++)
            cout << "On note index " << i << " the note/action played was: "
                 << aSeq[i] << endl;
        }
      ReplaceElig();

    }// end for k
}

//==================================================================
//
//  GetState
//
//==================================================================
int Sarsa::GetState(int* note)
{
  int i = 0;

  for (i = 0; i < LEN_NOTE; i++)
    if (note[i] == 1)
      break;

  return i;
}

//==================================================================
//
//  GetValue
//
//==================================================================
double Sarsa::GetValue(double* tempHid, double* tempfHid,
                       double* tempsN, int* tempsC, int noteIndex)
{
  int i = 0, j = 0;
  int NIndex[NUM_NOTES] = {0};
  NIndex[noteIndex] = 1;
```

```
  for (j = 0; j < NUM_HID_NODES; j++) {
    hid[j] = 0;

    for (i = 0; i < LEN_NOTE; i++)
      hid[j] += tempsN[i]*wN[j][i];

    //for (int n = 0; n < NUM_NOTES; n++)
    // hid[j] += NIndex[i]*wI[j][i];

    fHid[j] = ActFunc(hid[j]);
  }

  //--- Calculate linear comb of weights and hid outputs
  double value = 0;
  for (int j = 0; j < NUM_HID_NODES; j++)
    value += w[j] * fHid[j];

  return value;

}// end GetValue



//===================================================================
//
//  FindMaxA
//
//===================================================================
double Sarsa::FindMaxA(int* tempa, int* tempaSeq, int round, int noteIndex)
{
  int i = 0, j = 0, k = 0;
  int    testAction = -1;
  double testValue = 0;
  double temp =  -100000000;
  double testhid[NUM_HID_NODES]  = {0};
  double testfhid[NUM_HID_NODES] = {0};
  int    testA[LEN_NOTE]         = {0};
  int    NIndex[NUM_NOTES] = {0};
  NIndex[noteIndex] = 1;

  for (i = 0; i < 12; i++)
    tempa[i] = 0;

  //-------------------------------------------------------------------------
  // Test each of the 12 possible actions to find which gives highest value
  //-------------------------------------------------------------------------
```

```
for (i = 0; i < LEN_NOTE; i++ )
  {
    testA[i] = 1;
    if (i > 0)
      testA[i-1] = 0;

    //------------------------------------------------------
    // Calculate value for one of the 12 possible actions
    //------------------------------------------------------
    for (j = 0; j < NUM_HID_NODES; j++)
      {
        testhid[j] = 0;
        for (k = 0; k < LEN_NOTE; k++)
          testhid[j] += testA[k]*wN[j][k];

        //for (k = 0; k < NUM_NOTES; k++)
        // testhid[j] += NIndex[k]*wI[j][k];

        testfhid[j] = ActFunc(testhid[j]);
      }

    testValue = 0;
    for (j = 0; j < NUM_HID_NODES; j++)
      testValue += w[j]*testfhid[j];

    if (round > (NUM_ROUNDS - SOME_ROUND))
      cout << "Q = " << testValue << endl;

    if (testValue > temp) {
      temp = testValue;
      testAction = i;
    }
}

//--------------------------------------
// Exploration vs. Exploitation
//--------------------------------------
double explore = INIT_EXPLORE;
if (round > (NUM_ROUNDS - 100))
  explore = 0.4;
if (drand48() < explore)
  testAction = lrand48()%LEN_NOTE;
tempa[testAction] = 1;
tempaSeq[0] = testAction;

//------------------------------------------------------
// Calculate value for one of the 12 possible actions
```

```
  //------------------------------------------------------
  for (j = 0; j < NUM_HID_NODES; j++)
    {
      testhid[j] = 0;
      for (k = 0; k < LEN_NOTE; k++)
        testhid[j] += tempa[k]*wN[j][k];
      testfhid[j] = ActFunc(testhid[j]);
    }
  testValue = 0;
  for (j = 0; j < NUM_HID_NODES; j++)
    testValue += w[j]*testfhid[j];

  return testValue;
}

//====================================================================
//
//  Critique:  the reward function
//
//====================================================================
double Sarsa::Critique(double* note, int* chord, int* tempaSeq)
{
  double r = 0;
  //-------------------------------------------------------
  // Rule #1
  //            If  0 1 2  then  r = 1
  //-------------------------------------------------------
  if (tempaSeq[0]==0 && tempaSeq[1]==1 && tempaSeq[2]==2)
    r = 1;
  //    r = 3;
  //else if (tempaSeq[0]==0 || tempaSeq[1]==1 || tempaSeq[2]==2)
  //  r = 2;
  //else if(tempaSeq[0]==1 || tempaSeq[0]==2 || tempaSeq[1]==0 ||
  //          tempaSeq[1]==2 || tempaSeq[2]==0 || tempaSeq[2]==1)
  //  r = -2;
  //else
  //  r = -1;
  else
    r = 0;


  //-------------------------------------------------------
  // Rule #2
  //            If  1 2  then  r = 1
  //-------------------------------------------------------
  /*if (tempaSeq[1]==1 && tempaSeq[2]==1)
    r = 1;
```

```
  else
    r = 0;
  */

  return r;
}

//=================================================================
//
//  ReplaceElig:  resets the eligibility traces for all of
//              the states/notes to zero.
//
//=================================================================
void Sarsa::ReplaceElig()
{
  for (int j = 0; j < NUM_HID_NODES; j++) {
    for (int i = 0; i < LEN_NOTE; i++) {
      eN[j][i] = 0;
      eC[j][i] = 0;
      eA[j][i] = 0;
    }
    for (int i = 0; i < NUM_NOTES; i++)
      eI[j][i] = 0;

    e[j] = 0;

  }// end j

}

//====================================================================
//
//  UpdateElig
//
//====================================================================
void Sarsa::UpdateElig(double* noteBar, int* chord, int* action, int noteIndex)
{
  int j = 0, i = 0;
  int NIndex[NUM_NOTES] = {0};
  NIndex[noteIndex] = 1;

  for (j = 0; j < NUM_HID_NODES; j++) {
    for (i = 0; i < LEN_NOTE; i++) {
      eN[j][i] = lambda * eN[j][i] + w[j]*fHid[j]*(1-fHid[j]) * noteBar[i];
      eC[j][i] = lambda * eC[j][i] + w[j]*fHid[j]*(1-fHid[j]) * chord[i];
      eA[j][i] = lambda * eA[j][i] + w[j]*fHid[j]*(1-fHid[j]) * action[i];
    }
```

```
    for (i = 0; i < NUM_NOTES; i++)
      eI[j][i] = lambda * eI[j][i] + w[j]*fHid[j]*(1-fHid[j]) * NIndex[i];

    e[j] = lambda * e[j] + fHid[j];
  }
}

//=====================================================
//
//  ActFunc
//
//  The sigmoid activation function as given by,
//
//      f(sum) = 1 / (1 + e^(-sum))
//
//=====================================================
double Sarsa::ActFunc(double sum)
{
  return (1 / (1 + exp(-sum)));
}

//=====================================================
//
//  InitWeight
//
//=====================================================
double Sarsa::InitWeight()
{
  double newW = 10;

  if (drand48() > 0.5)
    while (newW < 0.001 || newW > 0.01)
      newW = drand48()/DENOM;
  else
    while (newW > -0.001 || newW < -0.01)
      newW = -drand48()/DENOM;

  return newW;
}

//=================================================================
//
//  Backprop
//
//=================================================================
void Sarsa::Backprop()
{
```

```
  int j = 0, i = 0;

  for (j = 0; j < NUM_HID_NODES; j++) {
    for (i = 0; i < LEN_NOTE; i++) {
      wN[j][i] += alphaH * delta * eN[j][i];
      wC[j][i] += alphaH * delta * eC[j][i];
      wA[j][i] += alphaH * delta * eA[j][i];
    }
    for (i = 0; i < NUM_NOTES; i++)
      wI[j][i] += alphaH * delta * eI[j][i];

    w[j]   += alphaO * delta * e[j];
    wBH[j] += alphaH * delta * BIAS;
  }
  wBO += alphaO * delta * BIAS;

}// end Backpropagation



//---------------------------------------------------------------------------
//  Constants.h
//
//  V. Manfredi and J. Franklin         Dept of Computer Science
//  Honors Thesis                       Smith College
//  fall, spring 2002                   Northampton, MA 01063
//
//  Non-linear Sarsa:  Constants file for non-linear Sarsa code.
//
//---------------------------------------------------------------------------

/*
// Average trace as input
#define NUM_ROUNDS     100000
#define SOME_ROUND      10000
#define NUM_HID_NODES      60
#define LEN_SEQ            36
#define LEN_CHORD          12
#define LEN_NOTE           12
#define NUM_SEQS           30
#define NUM_CHORDS          3
#define NUM_NOTES           3
#define RMAX                1
#define ALPHA_O          0.02
#define ALPHA_H          0.02
#define BETA              0.6
#define GAMMA            0.95
```

```
#define LAMBDA            0.6
#define DENOM               1
#define BIAS                1
#define INIT_EXPLORE        1
#define FINAL_EXPLORE     0.4
//#define STATE_INPUT        -1
#define AVG_INPUT          -2
*/

// State as input
#define NUM_ROUNDS    1000000
#define SOME_ROUND      10000
#define NUM_HID_NODES      60
#define LEN_SEQ            36
#define LEN_CHORD          12
#define LEN_NOTE           12
#define NUM_SEQS           30
#define NUM_CHORDS          3
#define NUM_NOTES           3
#define RMAX                1
#define ALPHA_O          0.01
#define ALPHA_H          0.01
#define BETA              0.6
#define GAMMA            0.95
#define LAMBDA            0.6
#define DENOM              20
#define BIAS                1
#define INIT_EXPLORE        1
#define FINAL_EXPLORE     0.4
#define STATE_INPUT        -1
//#define AVG_INPUT          -2
```

Bibliography

[AHBR99a] A. K. Afifi, G. Van Hoesen, A. Bechara, and R. Rodnitzky. Functional anatomy of basal ganglia. *http://www.vh.org/Providers/Textbooks/BasalGanglia/Figure2.html*, 1999.

[AHBR99b] A. K. Afifi, G. Van Hoesen, A. Bechara, and R. Rodnitzky. Functional anatomy of basal ganglia. *http://www.vh.org/Providers/Textbooks/BasalGanglia/Figure3.html*, 1999.

[AHBR99c] A. K. Afifi, G. Van Hoesen, A. Bechara, and R. Rodnitzky. Functional anatomy of basal ganglia. *http://www.vh.org/Providers/Textbooks/BasalGanglia/Figure4.html*, 1999.

[Bai99] L. C. BairdIII. *Reinforcement Learning Through Gradient Descent.* PhD thesis, School of Computer Science, Carnegie Mellon University, 1999.

[BBG99] J. Brown, D. Bullock, and S. Grossberg. How the basal ganglia use parallel excitatory and inhibitory learning pathways to selectively respond to unexpected rewarding cues. *Journal of Neuroscience*, 19(23):10502–10511, December 1999.

[Bis95] C. Bishop. *Neural Networks for Pattern Recognition.* Clarendon Press, New York, 1995.

[BM95] J. A. Boyan and A. W. Moore. Generalization in reinforcement learning: Safely approximating the value function. *Advances in Neural Information Processing Systems*, 7, 1995.

[BMPM01] G. S. Berns, S. M. McClure, G. Pagnoni, and P. R. Montague. Predictability modulates human response to reward. *Journal of Neuroscience*, 21(8):2793–2798, April 2001.

[BR98] K. C. Berridge and T. E. Robinson. What is the role of dopamine in reward: hedonic impact, reward learning, or incentive salience? *Brain Research Reviews*, 28:309–369, 1998.

[CBR96] J. R. Cooper, F. E. Bloom, and R. H. Roth. *The Biochemical Basis of Neuropharmacology, 7th ed.* Oxford University Press, New York, 1996.

[Chi97] G. Di Chiara. Alcohol and dopamine. *Alcohol Health and Research World*, 21(2):108–113, 1997.

[FMQ97] R. S. Feldman, J. S. Meyer, and L. F. Quenzer. *Principles of Neuropsychopharmacology.* Sinauer Associates Inc., Sunderland, Massachusetts, 1997.

[HDB96] M. T. Hagan, H. B. Demuth, and M. Beale. *Neural Network Design.* PWS Publishing Company, Boston, 1996.

[HH96] M. E. Harmon and S. S. Harmon. Reinforcement learning: A tutorial. *http://eureka1.aa.wpafb.af.mil/rltutorial*, December 1996.

[Hin87] G. E. Hinton. Connectionist learning procedures. Technical report, CMU-CS-87-115, Computer Science Department, Carnegie-Mellon University, 1987.

[HS97] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997.

[Jor86] M. Jordan. Attractor dynamics and parallelism in a connectionist sequential machine. In *Proceedings of the Eighth Annual Conference of the Cognitive Science Society*, 1986.

[KLM96] L. P. Kaelbling, M. L. Littman, and A.W. Moore. Reinforcement learning: A survey. *Journal of Artificial Intelligence Research*, 4:237–285, May 1996.

[LAS92] T. Ljungberg, P. Apicella, and W. Schultz. Responses of monkey dopamine neurons during learning of behavioral reactions. *Journal of Neurophysiology*, 67(1):145–163, January 1992.

[Mar96] J. H. Martin. *Neuroanatomy Text and Atlas 2nd ed.* Appleton and Lange, Stamford, Connecticut, 1996.

[Mit97] T. Mitchell. *Machine Learning*. McGraw-Hill, Boston, Massachusetts, 1997.

[Pic00] P. Picton. *Neural Networks*. Palgrave, New York, 2000.

[SB98] R. Sutton and A. Barto. *Reinforcement Learning*. MIT Press, Cambridge, Massachusetts, 1998.

[Sch98] W. Schultz. Predictive reward signal of dopamine neurons. *Journal of Neurophysiology*, 80(1):1–27, July 1998.

[Sch99] W. Schultz. The reward signal of midbrain dopamine neurons. *News in Physiological Sciences*, 14(6):249–255, December 1999.

[Sut88] R. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3:9–44, 1988.

[Tes95] G. Tesauro. Temporal difference learning and td-gammon. *Communications of the ACM*, 38(3):58–67, March 1995.

[TR97] J. N. Tsitsiklis and B. Van Roy. An analysis of temporal-difference learning with function approximation. *IEEE Transactions on Automatic Control*, 42(5):674–690, May 1997.

[TS93] S. Thrun and A. Schwartz. Issues in using function approximation for reinforcement learning. *Proceedings of Fourth Connectionist Models Summer School*, December 1993.

[Uni] The Robert Gordon University. A summary of basic - artificial neural networks. *http://www.scms.rgu.ac.uk/staff/nc/Files/Internal/SummaryBANN.htm*.