NTNU

Innovation and Creativity

Sequence learning in a model of the basal ganglia

Stian Søiland

Master of Science in Informatics

Submission date: June 2006

Supervisor: Keith Downing, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Abstract

This thesis presents a computational model of the basal ganglia that is able to learn sequences and perform action selection. The basal ganglia is a set of structures in the human brain involved in everything from action selection to reinforcement learning, inspiring research in psychology, neuroscience and computer science.

Two temporal difference models of the basal ganglia based on previous work have been reimplemented. Several experiments and analyses help understand and describe the original works. This uncovered flaws and problems that is addressed.

Preface

I started at NTNU in Trondheim in August 1998. I was initially studying physics, as I wanted to discover more about how our amazing world works. However, physics is full of differential equations and mathematics I couldn't handle.

I decided to change my studies to computer science, as I had great interest in computers, and particularly programming. After some years, when my MSc studies finally started, I was lucky to get appointed Keith Downing as my supervisor. After considering other projects, he suggested for me to work on a model of the basal ganglia. I had never heard about this brain structure before, and he quickly draw different boxes and pathways on the whiteboard with great enthusiasm.

This got me hooked. I started going to neuroscience classes and bought a neuroscience text book. I had forgot that one of the most amazing structures of the world is right there in our heads. It turned out in the end that I *was* going to discover something about how our world works. Unfortunately, the differential equations also came back, but I was not afraid of them anymore now. I knew how to attack them. With code.

I had never managed to do this work had it not been for the great support I've got from my friends, colleagues, family and girlfriend. I would like to thank specially my supervisor Keith, for his inspiring talks and emails; Diego, for his advice on "start writing as early as possible" (which I didn't); André, for his friendship and advises; Sverre, for always asking when I'll deliver; Magnus, for useful comments and help; Pernille, for helping me face trouble; Siv, for her motivation; Femke, for being helpful and understanding; my mum, for keeping me positive; Andrea, for his pasta and scientific discussions; Håvard, for his viewpoints and corrections; Magni, for our joint frustrations; John and University of Manchester, for letting me visit and have lots of fun; and least but most importantly, Gaby, for helping me stay focused (www) and for being amazingly kind and lovely

Stian Søiland, Manchester, UK, 2006-06-07

Contents

Abstract i

Preface iii

1 Introduction 1
- 1.1 Background . . . 1
- 1.2 Motivation . . . 1
- 1.3 Outline of this work . . . 2

2 Theory 3
- 2.1 Control theory . . . 3
  - 2.1.1 Actor-Critic architecture . . . 4
- 2.2 Reinforcement learning . . . 5
  - 2.2.1 Classical conditioning . . . 6
  - 2.2.2 Temporal difference (TD) learning . . . 6
- 2.3 Basal ganglia . . . 7
  - 2.3.1 Neurological description . . . 7
  - 2.3.2 Psychological research . . . 12
- 2.4 Continuous time recurrent neural networks . . . 14

3 Problem definition and methods 17
- 3.1 Modeling the basal ganglia . . . 17
- 3.2 Simulating basal ganglia in a robotic simulation . . . 18
- 3.3 Sequence learning . . . 19
- 3.4 Previous work . . . 20
  - 3.4.1 Temporal difference models . . . 20
  - 3.4.2 Berns and Sejnowski's model . . . 21
  - 3.4.3 Prescott et al. . . . 26

3.5 Experiment details 27
3.5.1 Implementation frameworks 28
3.5.2 Implementing the model of Berns and Sejnowski 30
3.5.3 Implementing the model of Prescott et al. 31

4 Experimental Results **33**
4.1 Reproducing the model of Berns and Sejnowski 33
4.1.1 Globus pallidus activity 33
4.1.2 Weight learning 35
4.1.3 Error values 37
4.1.4 Reward 39
4.1.5 Playback 39
4.2 Refinements 40
4.2.1 Noise 40
4.2.2 Leaky integrators with sigmoidal updates 43
4.3 Implementing a CTRNN 48
4.4 Reproducing the model of Prescott et al. 49
4.4.1 Transfer function 49
4.4.2 Performing action selection 50

5 Conclusion **53**
5.1 Summary 53
5.2 Discussion 53
5.2.1 Basal ganglia as a TD-learning actor-critic 53
5.2.2 Reproducing Berns and Sejnowski 54
5.2.3 Issues with reproducing experiments 56
5.3 Future work 57

A Source code **59**
A.1 Berns and Sejnowski in Python 60
A.2 CTRNN library for Python 68
A.2.1 Tests 78
A.3 Berns and Sejnowski using CTRNN 89
A.4 Prescott (2006) using CTRNN 95

Bibliography **103**

List of Figures

2.1 Negative feedback control system. 4
2.2 Actor-Critic architecture as a controller. 5
2.3 Medium spiny neurons in the striatum. 8
2.4 Coronal slices of the human brain highlighting the nuclei of the basal ganglia. 9
2.5 Pathways in the basal ganglia. 10
2.6 Firing rates in the direct pathway for a striatum at rest and being excited. 11
2.7 Monkey dopamine neuron firing when learning to predict a reward. . . . 13
2.8 Simple continuous-time recurrent neural network (CTRNN) network. . . 15
2.9 Potential development of a simplified continuous-time recurrent neural network (CTRNN) neuron. 15

3.1 Sequence learning in the basal ganglia. 19
3.2 Temporal difference models of basal ganglia. 20
3.3 Berns and Sejnowski (1998) computational model of the basal ganglia. . 22
3.4 Sequence learning in the Berns and Sejnowski's model. 23
3.5 Humphries and Gurney basal ganglia model. 27

4.1 Globus pallidus activities during learning in Berns and Sejnowski (1998). 34
4.2 Globus pallidus activities during learning in reimplementation. 34
4.3 Changes in connection strengths when learning sequence (original). . . . 36
4.4 Changes in connection strengths when learning sequence (reimplemented). 36
4.5 Firing of subthalamic nucleus unit 2. 38
4.6 Error value distribution (original). 38
4.7 Error value distribution (reimplemented). 39
4.8 Error values with three different instances of random noise. 42
4.9 Firing rate of globus pallidus unit with and without constant noise. . . . 44

4.10 Effect of sigmoidal update rule on continuous-time recurrent neural network (CTRNN) neurons. 46

Chapter 1

Introduction

1.1 Background

The composite brain structure *basal ganglia* inspires researchers in several sciences. First of all, neurologists are interested in how the diverse set of nuclei are involved in motor control, in particular how movement disorder diseases such as Parkinson's (Parkinson, 1817) and Huntington's (Huntington, 1872) are caused by degenerations within the basal ganglia.

Psychologists are researching how the basal ganglia is involved in planning and learning, including action selection and sequence learning, as covered in this work. Computer scientists find inspiration in the basal ganglia for reinforcement learning algorithms and designing artificial neural networks.

During the last 15 years, several approaches for modeling the basal ganglia on system level has been presented. As there is a diversity of functionality contributed to the basal ganglia, researchers have built models dealing with topics as classical conditioning, temporal difference learning, sequence learning and action selection.

1.2 Motivation

The original motivation for this work was to construct a robot simulation applying a computerized model of the basal ganglia, enabling the robot to learn sequenced movements, for instance moving in a maze towards a reward.

In order to do such a task, the basal ganglia model will have to perform sequence learning. This thesis investigates a model by Berns and Sejnowski (1998), which is reported to be able to learn and reproduce sequences. Several flaws in the original work

were identified and analyzed.

1.3 Outline of this work

Chapter 2 reviews the theories relevant for modeling basal ganglia, including actor-critic architecture and continuous time recurrent neural networks.

Chapter 3 describes the problem of doing sequence learning and modeling the basal ganglia. Methods applied for the implementations are detailed.

Chapter 4 presents the results from experimenting with the basal ganglia model.

Chapter 5 concludes the work and suggest future directions.

Chapter 2

Theory

This chapter introduces the theory relevant for this work. First, control theory is presented with an emphasis on actor-critic architectures, as the basal ganglia is often compared to actor-critics. Next, the section on reinforcement learning gives an overview of classical conditioning and temporal difference learning.

Following, the basal ganglia is described from a neurological and psychological point of view. Finally, this chapter gives a short theoretical introduction to continuous-time recurrent neural networks (CTRNN), a type of artificial neural networks often used in basal ganglia modeling.

2.1 Control theory

In classical control theory a system represents some unknown dynamics which is to be controlled. The system can represent real world things like a car, industrial plant or the environment of a room. A generic linear negative feedback control system is pictured in figure 2.1 on the following page.

The goal of the controller is to match the output of the system as closely as possible to the reference signal. To achieve this, a feedback loop is formed so that the *controller* is fed with the difference between the desired signal (the *set-point*) and the actual output. This difference can be called the *error*. In order to change the system output in the desired direction, the controller issues *control signals* to the system. The goal of the controller can therefore be said to minimise the error of the feedback loop.

A simple example of a control system is the temperature control of an office building. In this situation, the system is the environment of the building, where the output can be described by a temperature sensor, giving the room temperature. A thermostat setting

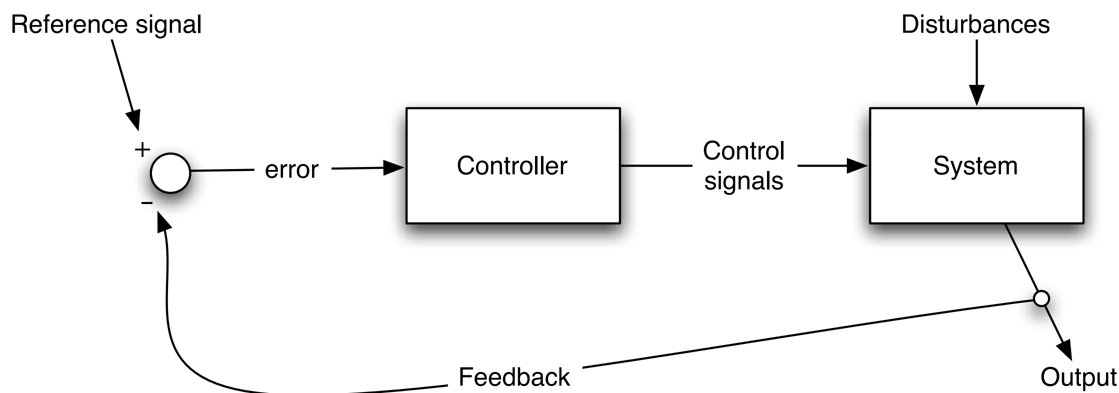

Figure 2.1: Negative feedback control system. The error is the difference between the reference signal and the output of the system. The difference is inputted to the controller which will issue control signals to modify the system as to minimise the error. Disturbances skew the system from the desired output. The desired output is fed to the controller as the reference signal.

provides the desirable state, thus forming the reference signal. The controller issues control signals to turn on or off heaters in order to increase or decrease temperature. However, there are also *disturbances* to the system, for instance an employee might leave a window open and push the room temperature out of the desired state.

2.1.1 Actor-Critic architecture

A feedback control system is considered a *closed-loop* system because the output of the controller will modify the environment, and thus also the inputs to the controller. On the other hand, an *open-loop* system does not have a feedback loop, and the output of such a system does not (directly or indirectly) influence its inputs.

Actor-Critic models are strongly related to control theory. Figure 2.2 on the next page pictures the Actor-Critic architecture from the viewpoint of control theory. The *actor* selects actions to be performed on the environment to match the feedback signals with the desired environment output, as determined by the context. The *critic* judges the actor's decisions, giving a positive or negative *reinforcement signal* depending on if the actor modified the environment in the right direction or not.

When comparing the actor-critic architecture to control systems, the actor can be considered the controller, while the environment is comparable to the system. The critic plays the role of a *temporal difference (TD) error* that gives an indication if the actor decision was better or worse than predicted (Barto, 1995). Note that the actor-critic

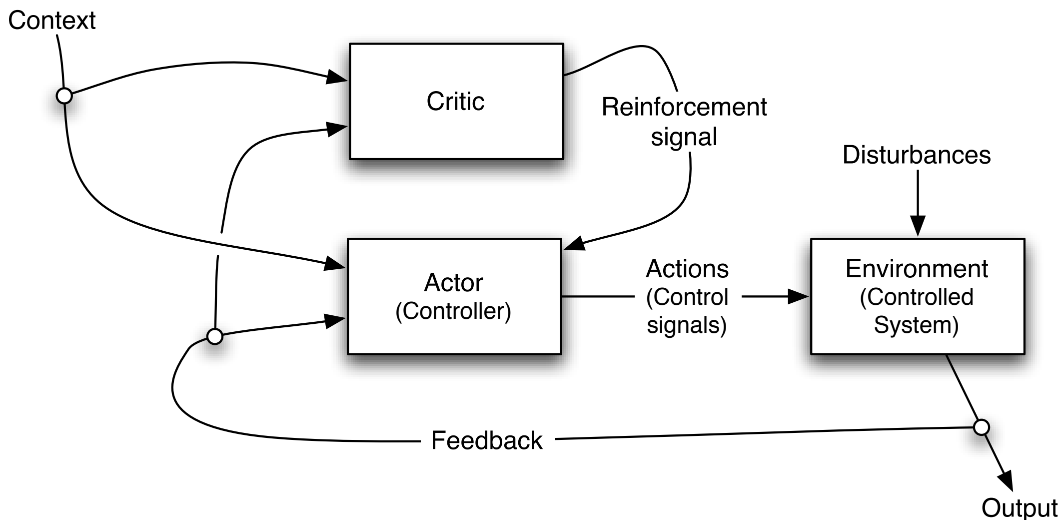

Figure 2.2: Actor-Critic architecture as a controller. The actor behaves as the controller in figure 2.1 on the facing page, acting on the environment to match the feedback with the contextual goal. The critic judges the actor by positive or negative reinforcement signals depending on the success of achieving the control objectives as provided by the context. Adapted from Barto (1995, fig. 1).

architecture set-point is a more generalised *context*, not necessarily directly comparable to the reference signal set-point of classical control systems.

The temporal difference (TD) error from the critic modifies the actor's *policy* to strengthen or weaken the selection of the preceding action. For instance, if the actor is implemented as an artificial neural network, a positive reinforcement signal combined with differential Hebbian learning (Sutton and Barto, 1981) would strengthen weights on connections that were active in selecting the action.

2.2 Reinforcement learning

Reinforcement learning (RL) is a broad field covering research in behavioral psychology, machine learning, neuroscience and statistics. Reinforcement learning can be described as *the problem faced by an agent that learns behavior through trial-and-error interactions with a dynamic environment* (Kaelbling et al., 1996).

Central to reinforcement learning is the concept of reward. As the agent visits different states, a reward is assigned, possibly negative representing punishments. The agent must perform actions to move from one state to another. The goal of an agent perform-

ing reinforcement learning is to maximise the cumulative reward (Wörgötter and Porr, 2005).

As reinforcement learning has its roots in psychology, this section will first describe classical conditioning before detailing a particular way to perform reinforcement learning called temporal difference (TD) learning. It is worth to note that several other RL heuristics exists, for reviews see Kaelbling et al. (1996) and Sutton and Barto (1998).

2.2.1 Classical conditioning

Classical conditioning is best illustrated by the example of Pavlov (1927), where his dogs started salivating when presented the ringing of a bell, predicting that food was to be served. The dogs learned that the bell stimulus always conditioned the reward. Thus the dogs were able to learn the temporal correlation of the bell predicting the reward. The conditioning signal, the bell, is called the conditioned stimulus (CS), while the reward signal is called the unconditioned stimulus (US). After learning, the CS will act as a substitution signal for the US .

As pointed out by Wörgötter and Porr (2005), this represents an *open-loop* condition. An open-loop condition is where the prediction does not influence the environment. The dog's salivation does not change the fact that the food is followed by the bell. The opposite is called *closed-loop* conditions, where the system output also influences the inputs. For instance with a robot, deciding to turn left will change the sensor readings, or for conditioning experiments, a monkey pushing a button at the right time will give him the food reward.

2.2.2 Temporal difference (TD) learning

In *delayed reinforcement learning*, the system observes a temporal sequence of input states, followed by the reinforcement signal, the reward. The task of such a system is to predict the expected reward given the input states (Tesauro, 1992). In a closed-loop environment, the system might also issue control signals to influence the sequence of states, where the goal is to maximise the reinforcement.

One of the challenges of delayed reinforcement learning is the temporal credit assignment problem. The reward signal can be received after performing a long sequence of action selections. The latest action should not be the only rewarded state, because the agent also had to choose the correct previous actions to end up in that state.

The temporal credit problem is therefore how to distribute reward or blame to the

different states leading to the rewarding state.

Instead of assigning credit based on the difference between predicted and actual outcome, in temporal difference (TD) learning (Sutton, 1988) the approach is to assign credit by the difference between temporally successive reward predictions. As such, the TD model uses a reward prediction, and one of the features of this is that there is no longer a need for a separate reward signal.

2.3 Basal ganglia

This chapter starts with a description of the *basal ganglia* from a neurological point of view. The focus of this work is on the functional aspects of the basal ganglia, so the text will not go in depth of the anatomical details.

Finally this chapter reviews how psychological research has shown that the basal gangia is linked to reinforcement learning.

2.3.1 Neurological description

Basal ganglia is a set of connected nuclei in mammal brains considered important for movement and cognition. In primates, the basal ganglia consist of the *striatum* (STR), *globus pallidus* (GP), *substantia nigra* (SN) and the *subthalamic nucleus* (STN). The *striatum* consists of the nuclei *caudate* and *putamen*, and the globus pallidus is split into *globus pallidus external* (GPe) and *globus pallidus internal* (GPi). The *substantia nigra* is also usually split into *substantia nigra pars compacta* (SNc) and *substantia nigra pars reticulata* (SNr).

The basal ganglia is a mirrored structure, as shown in figure 2.4 on page 9, so all nuclei are doubled for the left and right hemisphere. This text will, as a simplification, consider basal ganglia as a single set of nuclei. This is normal in literature on modeling basal ganglia.

Inputs

The separate nuclei caudate and putamen, called the *corpus striatum*, receive projections from large areas of the cerebral cortex and is considered the input zone of the basal ganglia.

The inputs to the caudate and putamen are however not equivalent. While the caudate primarily receives high level inputs from multimodal association cortices, the

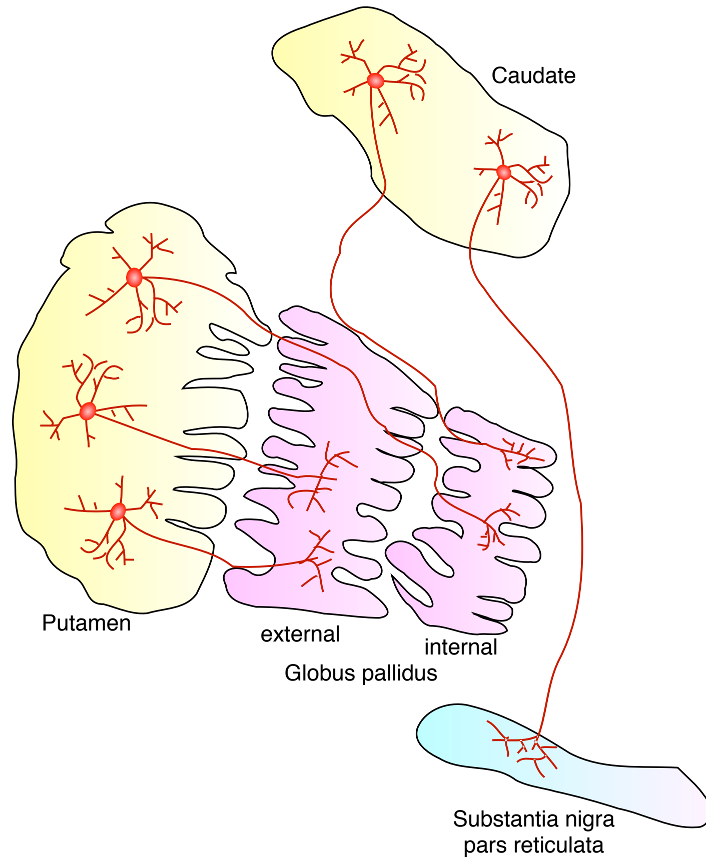

Figure 2.3: Medium spiny neurons in the striatum. Putamen and caudate, in this text referred to as the striatum, projects medium spiny neurons to globus pallidus and substantia nigra pars reticulata. The putamen is anatomically located with the split globus pallidus, but separated from the caudate and substantia nigra. Adopted from Purves et al. (2003, fig. 17.3)

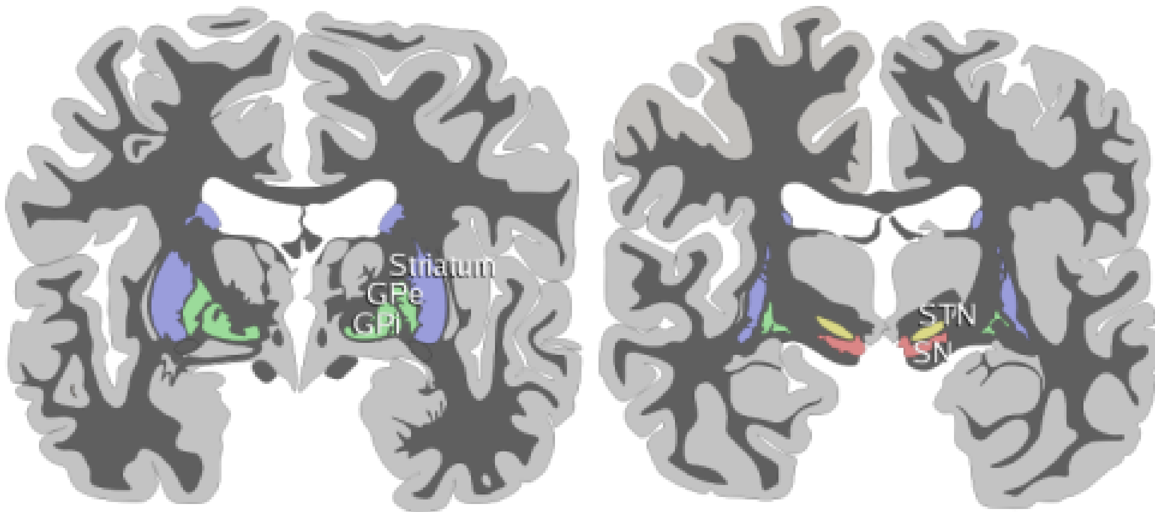

Figure 2.4: Coronal slices of the human brain highlighting the nuclei of the basal ganglia. The left slice features the *striatum* (STR), *globus pallidus external* (GPe) and *globus pallidus internal* (GPi). The right slice views the *subthalamic nucleus* (STN) and the *substantia nigra* (SN). Reproduced from Wikipedia [1].

inputs of the putamen are more lower level, including primary and secondary sensory cortical areas. In this text, focusing on the connectionism and learning aspect of the basal ganglia, the caudate and putamen are treated like one homogenous set of inputs and called *striatum*.

The striatum is mainly composed of *medium spiny neurons* characterized by their large dendritic trees which allows them to integrate inputs from a variety of cortical, thalamic and brainstem structures (Purves et al., 2003). Figure 2.3 on the preceding page shows how the spiny neurons of the striatum projects to the globus pallidus and substantia nigra.

Outputs

The primary output zones of the basal ganglia are the *globus pallidus internal* and *substantia nigra pars reticulata*. The internal globus pallidus projects to the VA/VL complex of the *thalamus*, where the outputs are relayed to the motor cortex. This completes the motor loop, beginning with the basal ganglia inputs from almost all areas from the cerebral cortex, and terminating on the motor and premotor areas of the frontal lobe and in the superior colliculus.

The *substantia nigra pars reticulata* on the other hand does not primary project

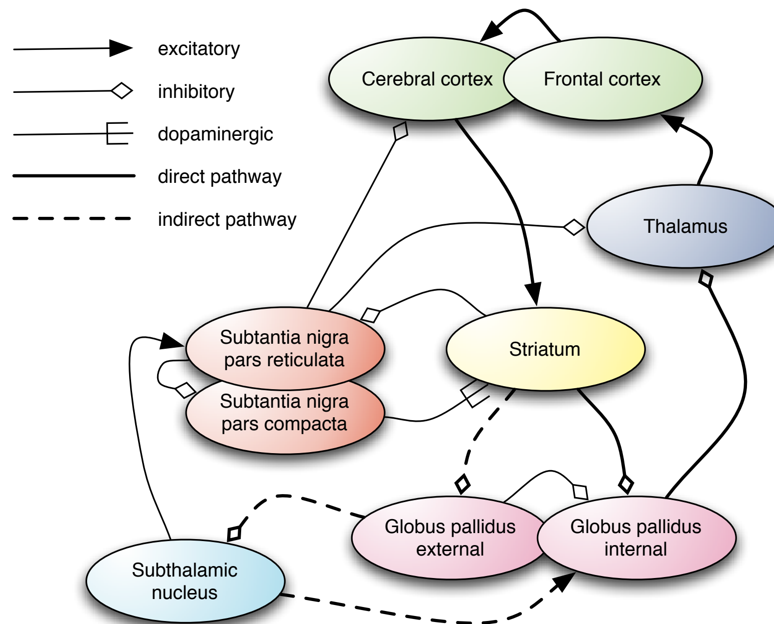

Figure 2.5: Pathways in the basal ganglia. The cortex projects to *striatum* (STR), which projects inhibitory through *globus pallidus internal* (GPi) and thalamus to complete the motor loop in the frontal cortex. The indirect pathway diverts from the striatum through *globus pallidus external* (GPe), *subthalamic nucleus* (STN) before modulating the direct pathway signals in the GPi .

through the thalamus, but directly to eye movement related areas in the superior colliculus.

In this text, the two nuclei are mostly considered equal and just mentioned as GPi. This assumption is based on their similar output functions, and the fact that developmental studies have shown that SNr was once a part of the globus pallidus, and that it is only in the most recent stages of evolutionary development that the nuclei have become separated (Purves et al., 2003).

Pathways

There can be considered to exist two primary pathways through the basal ganglia connecting the input zone to the output zone and providing the internal dynamics (Smith et al., 1998). The *direct pathway*, as shown in figure 2.5, is made up of the excitatory projections from the cortex to the striatum, inhibitory projections from striatum to the globus pallidus internal, with inhibitory projections further on to the thalamus where excitatory connections closes the loop back to the cortex.

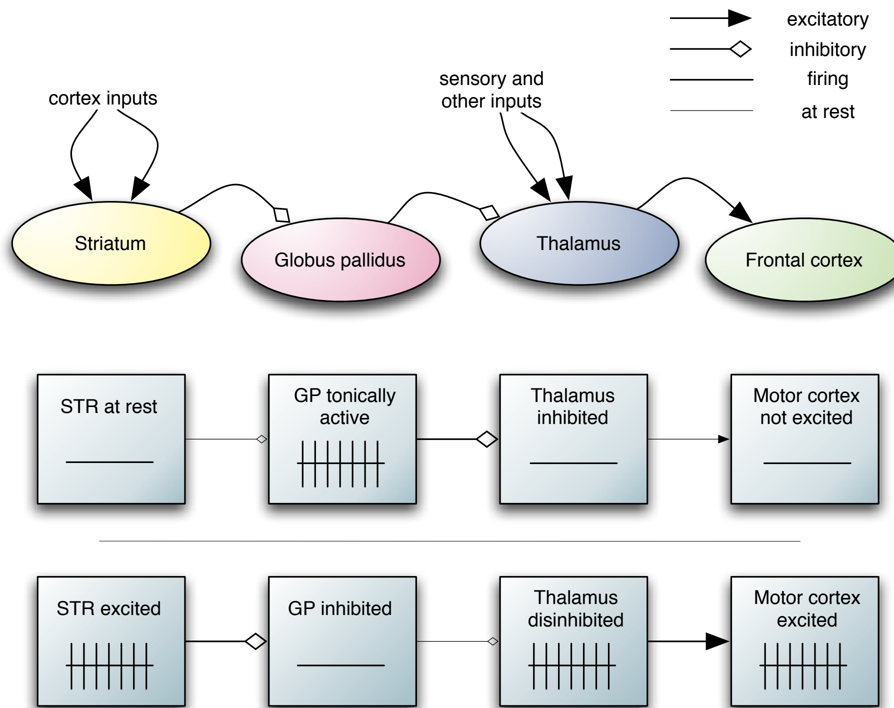

Figure 2.6: Firing rates in the direct pathway for a striatum at rest and being excited. The double inhibition of STR-GP and GP-thalamus enables a net excitation of the motor cortex due to other sensory inputs to the thalamus. Adapted from Purves et al. (2003, fig. 17.6).

The direct pathway can be said to be modulated by the *indirect pathway*. The indirect pathway also starts with the cortical projections to the striatum, but the striatum projects inhibititory through the globus pallidus external and further on through the *subthalamic nucleus*, where it excitatory affects the globus pallidus internal, merging with the direct pathway. In this way, the indirect pathway can be said to be modulating the signals of the direct pathway.

The GPi-thalamus pathway and the GPe-STN pathway are both inhibitory, but the globus pallidus is tonically active, which means that without inhibition by the *striatum* or the *subthalamic nucleus*, the globus pallidus neurons will always fire. This means that the GPi will inhibit the thalamus most of the times, except when stimulated by the striatum or STN, as shown in figure 2.6.

The net effect of this disinhibitory circuit of the direct pathway is that when the striatum is at rest, the globus pallidus is tonically active, which will inhibit the thalamus and thereby block thalamus from relaying sensory inputs to the motor cortex. On the

other hand, if the striatum is excited, the GP will be inhibited, which will no longer inhibit the thalamus that will then allow sensory signals to excite the motor cortex.

This view of the direct and indirect pathways has been extended by Gurney et al. (2001), their computational model of the basal ganglia suggests that the pathways can be considered as performing selection and control in the act of action selection. Bar-Gad et al. (2003) updates the classical box-arrow description as of figure 2.5 on page 10 with a summary of newer experimental data. *The direct and indirect pathways are not as segregated as once thought* (Bar-Gad et al., 2003). In addition, most paths are paired with a back-projection, for instance GPe to *striatum*. The cortex projects not only to the *striatum*, but also to the thalamus and STN .

For simplicity, in this text, only the classical pathways will be considered, as this is the general usage in basal ganglia modeling literature.

2.3.2 Psychological research

As reviewed in Schultz et al. (1997), experiments on dopamine neurons have revealed that dopamine in the basal ganglia is closely linked to reward prediction. In experiments with monkeys, as shown in figure 2.7 on the next page, short dopamine releases are initially triggered by the primary reward, such as drops of fruit juice released to the monkey's mouth. After repeated training of pairing visual and auditory cues (such as flashing a light) followed by the reward, the monkey is able to predict the reward at the time of the cue signals, because the dopamine is released at the time of the conditioned stimulus (CS), and no longer at the reward. However, if the reward fails to be delivered, the dopamine background signals will be depressed at the point in time where the monkey predicted it to be delivered.

This learning of the connection between an conditioned stimulus and a unconditioned stimulus, as initially described by Pavlov (1927), shows that the dopamine responses can be viewed as an error signal, noting the difference between the predicted and actual reward (Schultz et al., 1997). Before learning, the reward is not expected, and so the rise of dopamine can be said to signal a positive error. After learning the reward is received as expected, and so there is no longer an error signal at the delivery time, unless the reward is failing, in which a negative error signal (depression of the background dopamine firing) will occur.

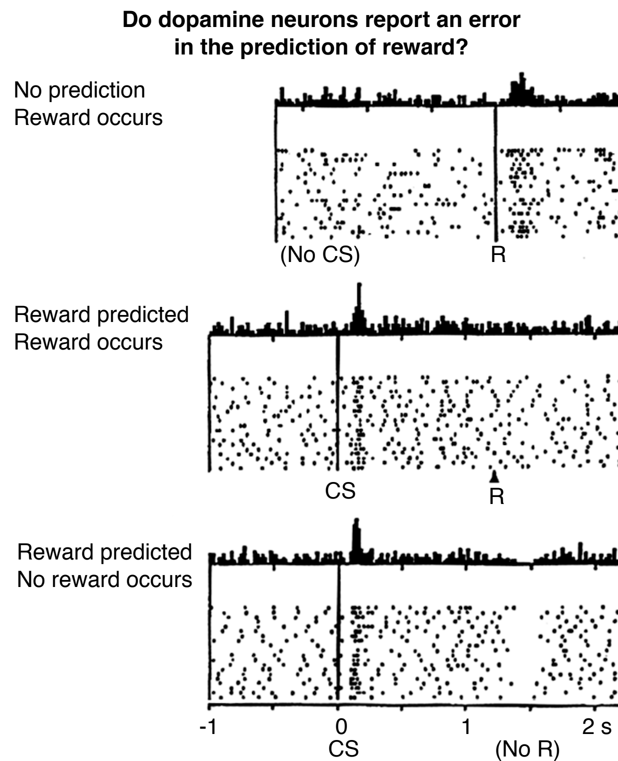

Figure 2.7: Monkey dopamine neuron firing when learning to predict a reward. Before learning (top graph), the conditioned stimulus (CS) is not detected, but the reward (R) gives a rise in dopamine firing. After learning (middle graph), the CS gives a sharp rise in dopamine firing, and predicts the reward so that no rise in firing occurs at reward. If the reward fails to be delivered after the CS (bottom graph), the prediction will depress the background dopamine firing at the predicted time of the award. Reproduced from Schultz et al. (1997, fig. 1).

2.4 Continuous time recurrent neural networks

Continuous-time recurrent neural networks (CTRNN) (Beer, 1995) is an artificial neural network applying leaky integrators as a countermeasure to the loss of the temporal dimension in discrete feed-forward networks. A neuron's activity is made dependant on both current inputs and previous activity. A time constant will generally specify how much historical activity should matter compared to new inputs, and such could be said to specify the temporal perspective of the neuron.

A large time constant means a neuron will average (integrate) inputs over a longer time, making it 'slow'. The output of such a neuron will be smoother than the outputs of a 'faster' neuron, because it will take a longer time to fully respond to a changed signal. A small time constant will place more importance on the history than the current inputs, and can introduce dampened oscillations because of over-compensation.

A generic CTRNNs can be described by the differential equation 2.1 (adapted from Beer, 1995):

$$\frac{\mathrm{d}y_i}{\mathrm{d}t} = \frac{1}{\tau_i}\left(-y_i + \sum_{j=1}^{N}\Big(w_{ji}\sigma\left(y_j + \theta_j\right) + I_i(t)\Big)\right) \qquad i = 1, 2, \ldots, N \tag{2.1}$$

where y is the current state of each of the N neurons, τ is the time constant, w_{ji} is the weight of the connection from neuron j to i, θ is the bias, and I is the external input. Figure 2.8 on the next page illustrates a simple CTRNN . σ is the activation function, usually as described by the sigmoidal function in equation 2.2.

$$\sigma(x) = \frac{1}{1 + e^{-x}} \tag{2.2}$$

What makes a CTRNN different from a discrete time recurrent network is that changes in the network are continuous, so that the output of a neuron at time t is dependent both on the inputs and on the previous activation level at $t - \Delta t$. This enables the CTRNN neurons to have some form of short-term memory and to get a grasp of time. Each neuron can have a different time constant, and thereby a different temporal view. A time constant equal to the time step Δt would effectively make the neuron behave like in a normal recurrent network, with no internal state or historical trace.

Equation 2.1 can be solved using the forward Euler method. As pointed out by Blynel and Floreano (2002); Hines and Carnevale (1998), the time step must never be

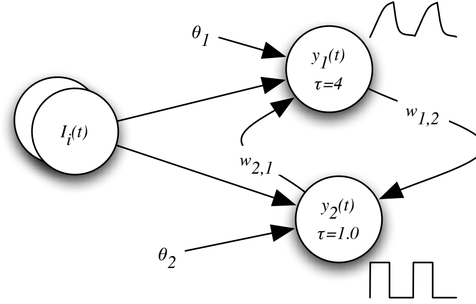

Figure 2.8: Simple continuous-time recurrent neural network (CTRNN) network. Inputs I_i feed the network, while neurons y_1 and y_2 have looping connections of weights $w_{1,2}$ and $w_{2,1}$. The neurons have biases θ_j and different time constants τ_j. The time constants modify the reaction time of the internal potential, and thus the shape of the output. Neuron y_1 has a large time constant $\tau = 4$, a slowly reacting potential, giving a curved output. The neuron y_2 has a fast time constant of $\tau = 1$, resulting in sharply edged outputs directly following the inputs.

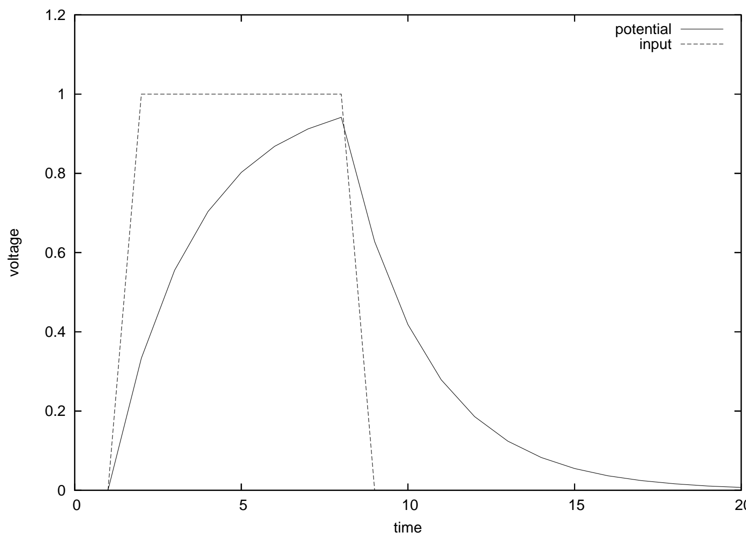

Figure 2.9: Potential development of a simplified continuous-time recurrent neural network (CTRNN) neuron. Calculated using the discrete equation 2.3 on the following page, with $\tau = 3$ and input active for $t = 2 \ldots 8$. The internal state integrates the input, so that when the input is active, the potential rises, while it decays in the lack of input. The potential develops negative exponentially towards the current input.

more than twice the smallest time constant in the system, or else the discrete integration will not be numerically stable. The discrete update rule for the CTRNN is:

$$y_i(t + \Delta t) = y_i(t) + \frac{\Delta t}{\tau_i}\left(-y_i(t) + \sum_{j=1}^{N}\Big(w_{ji}\sigma\left(y_j + \theta_j\right) + I_i(t)\Big)\right) \qquad i = 1, 2, \ldots, N \quad (2.3)$$

Figure 2.9 on the preceding page shows the internal state of a simple CTRNN neuron using equation 2.3 with $\Delta t = 1$, $\tau = 3$ and no recurrent connections. The potential develops in a negative exponential fashion towards the current input signal, like an RC circuit. The potential development of the CTRNN neuron also resembles an eligibility trace, as used in temporal difference (TD)-learning, see section 2.2.2 on page 6.

This kind of artificial neurons has also been called *leaky integrators* because they integrate their input over time, but the internal state also decays, or leaks, over time.

Chapter 3

Problem definition and methods

This chapter describes the problem of modeling the basal ganglia in computer simulations and reviews the categories of existing basal ganglia models. Next, the task of learning sequences in a basal ganglia context is specified. Previous work on this topic is reviewed, with a focus on the work by Berns and Sejnowski. Finally, implementation details related to this work are presented.

3.1 Modeling the basal ganglia

The functioning and internal workings of the basal ganglia are not yet fully understood, and as such the basal ganglia has been the basis of several computer modeling attempts, as reviewed in Joel et al. (2002); Bar-Gad et al. (2003); Gillies and Arbuthnott (2000); Gurney et al. (2004).

The models can be sorted into three categories, defined by Gillies and Arbuthnott (2000) as these functional aspects:

Reinforcement learning (RL) Models learning a sequence of actions using rewards signals delivered in the end of the sequence. The motivation for modeling basal ganglia using reinforcement learning comes from classical conditioning (Schultz et al., 1997), described in section 2.3.2 on page 12. By adopting the temporal difference (TD) learning algorithm (Sutton, 1988), basal ganglia has been successfully modeled as an actor-critic architecture (Barto, 1995; Suri and Schultz, 1999).

Serial processing Inspired by the looping features of the basal ganglia, serial processing models focuses on how motoric action sequences are learned and applied. The model of Suri and Schultz (1999) use the environmental context to 'playback' the

sequence and shows how an actor-critic can perform sequence learning. Beiser and Houk (1998) investigates how the basal ganglia could encode sequences as spatial patterns. The Berns and Sejnowski (1998) model learns by associating steps in the sequence with earlier steps, using different time constants to resolve disambiguations.

Action selection The tonically active outputs of the basal ganglia, *globus pallidus internal* (GPi) and *substantia nigra pars reticulata* (SNr) inhibits *thalamus* targets. This suggests that the basal ganglia can perform action selection of motoric actions. If the cortex projects conflicting motor signals, the basal ganglia allows only the appropriate actions to proceed, preferably in the correct order (Mink, 1996).

Gurney et al. (2001) shows how a model performing action selection can be analyzed as containing pathways for *selection* and *control*, merging the model with the actor-critic architecture and expanding on the classical view of the direct and indirect pathways. In Prescott et al. (2006), the model is embedded in a robot, performing action selection in a live environment.

3.2 Simulating basal ganglia in a robotic simulation

The original problem definition for this work was defined as:

> **Predicting temporal differences, a simulated model of the Basal Ganglia**
> Design a neural net-based model of the Basal Ganglia to be used in a simulated robot. This will enable the robot to predict the consequences of its actions based on its previous experiences, particularly those involving rewards or punishments. This realizes temporal difference learning in a neural net controller.

The idea behind this problem definition was to model the basal ganglia and apply it in a robotic simulation. As a manifestation of sequence learning, a typical task maze was considered, where the agent was to follow say left-left-right-left and finally receive a reward. If the model could be compared with temporal difference (TD) learning (as described in section 2.2.2 on page 6), on repeated occurrences of the reward, the agent should be able to learn the preceding actions leading to the reward.

The main challenge of this task was considered to be the sequence learning. The problem definition was therefore refined to the task of *learning sequences in a basal*

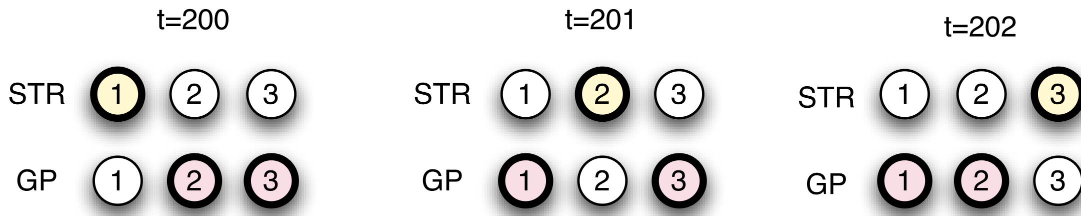

Figure 3.1: Sequence learning in the basal ganglia. While learning the sequence $1, 2, 3$, only a single *striatum* (STR) unit is made active at a time. The *globus pallidus* (GP) units should all be active, except the unit corresponding to the active STR unit.

ganglia model for action selection. Although not considering the robotic simulation, future robotic simulation continued to be a motivator. The following section will specify the task of sequence learning.

3.3 Sequence learning

The problem definition was defined as learning sequences in a basal ganglia model. In this context, a sequence is considered alternating inputs to the *striatum* (STR) matrix, represented by n artificial neurons (called units), one for each possible action of the sequence.

Each time step, as the sequence is unrolled, a single STR unit will be active, the unit corresponding to the current action in the sequence. Thus a sequence of say $1, 2, 3$ means that in the first time step, only the first STR unit is active, while in the next time step, the second STR is the only active input.

The model should reproduce those sequences at the output units called *globus pallidus* (GP), but inverted. As shown in figure 3.1, that means that if STR unit 2 is active, then all but the second GP units should be active. This performs action selection, and as will be explained, sequence learning.

The model is trained by looping the desired sequence until the network dynamics can be considered stabilised, which means until there is no considerable changes in the GP outputs compared to the previous learning cycle.

After training, the model is again presented with the first element of the sequence, called 'the hint', and should then reproduce the rest of the sequence at the GP outputs without further STR activation.

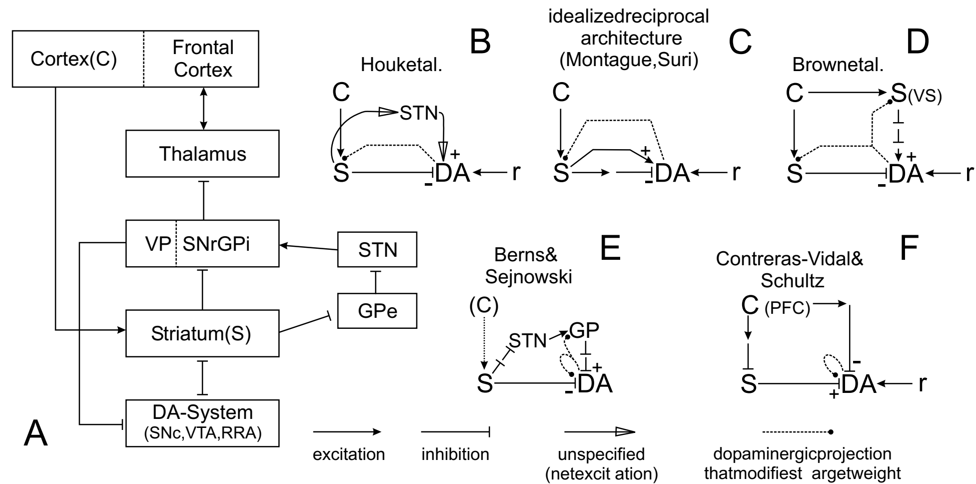

Figure 3.2: Temporal difference models of basal ganglia.
A: Box-and-arrows diagram of primary pathways. **B:** Houk et al. (1995) **C:** Montague et al. (1996); Suri and Schultz (2001) **D:** Brown et al. (1999) **E:** Berns and Sejnowski (1998) **F:** Contreras-Vidal and Schultz (1999)
Reproduced from Wörgötter and Porr (2005, fig. 8).

3.4 Previous work

Computational models of the basal ganglia vary in their architecture depending on the task that is examined. However, most models seems to be based on the classical view of the basal ganglia pathways, as explained in section 2.3.1 on page 10. This section will give an overview over some TD based models of basal ganglia, followed by details of the models by Berns and Sejnowski and Prescott et al..

3.4.1 Temporal difference models

Figure 3.2 show the diversity of the basal ganglia models based on TD learning, as reviewed by Wörgötter and Porr (2005). This figure simplifies the dopamine system (DA) for easier comparison, and show how researchers connect the cortex (C), striatum (S), globus pallidus (GP) and the subthalamic nucleus (STN). All models except Berns and Sejnowski (1998) also receive a reward (r).

The model of Houk et al. (1995) was one of the first to follow the actor-critic architecture. Learning is essentially achieved by the TD rule, by calculating the temporal difference in the dopamine system, which performs as the critic.

Montague et al. (1996) models dopaminergic projections to the striatum as part of

the TD rule, but as pointed out by Wörgötter and Porr (2005), this model *assumes a direct excitatory pathway and an indirect inhibitory pathway*, but *in reality the situation, however, is reversed*.

Brown et al. (1999) do not use the assumption of direct and indirect pathways, but follows neurological evidence for a direct projection from the cortex to the dopamine system, in addition to projections from the ventral striatum. This model applies a set of band-pass filters to address timing issues that has been uncovered with other approaches.

Berns and Sejnowski (1998) resembles Houk et al. (1995), but *implements several pathways more accurately following the known anatomical structures than any of the other models* (Wörgötter and Porr, 2005). However, the model assumes weight learning is performed in the projections from *subthalamic nucleus* (STN) to GP , while the literature generally suggests that learning is performed in the striatum. This model calculates the error term internally and do not use a separate reward signal.

Contreras-Vidal and Schultz (1999) also modify weights outside the striatum, but has been criticised by Wörgötter and Porr for exciting instead of inhibiting the dopamine system, diverting from the approach of the other models. The model has been used in simulations for classical conditioning tasks.

3.4.2 Berns and Sejnowski's model

The model of Berns and Sejnowski (1998) was considered for further examination. The reason for this is that the original work reported sequence learning and reproduction, follows the known anatomy of globus pallidus to a great extent, and can be viewed as an actor-critic architecture.

Their proposed model incorporates many anatomical and physiological aspects of the *basal ganglia* shown by neurology. Berns and Sejnowski view the basal ganglia as an action selection device, a view supported by Barto (1995); Houk et al. (1995); Berns and Sejnowski (1996); Montague et al. (1996); Gurney et al. (1998).

Berns and Sejnowski models the basal ganglia, as shown in figure 3.3 on the following page, by including *striatum* (STR), *globus pallidus* (GP), *subthalamic nucleus* (STN) and *substantia nigra pars compacta* (SNc). The *striatum* is viewed as the input stage, while the GP represents the output stage. Each neuron in the model represents clusters of *in vivo* neurons, and is arranged as separate parallel paths, each STR-GP-STN combination representing a possible action to be selected.

In the direct pathway, the *striatum* layer directly determines which GP unit is to be inhibited, i.e. which action is to be selected. The GP neurons project inhibitory to the

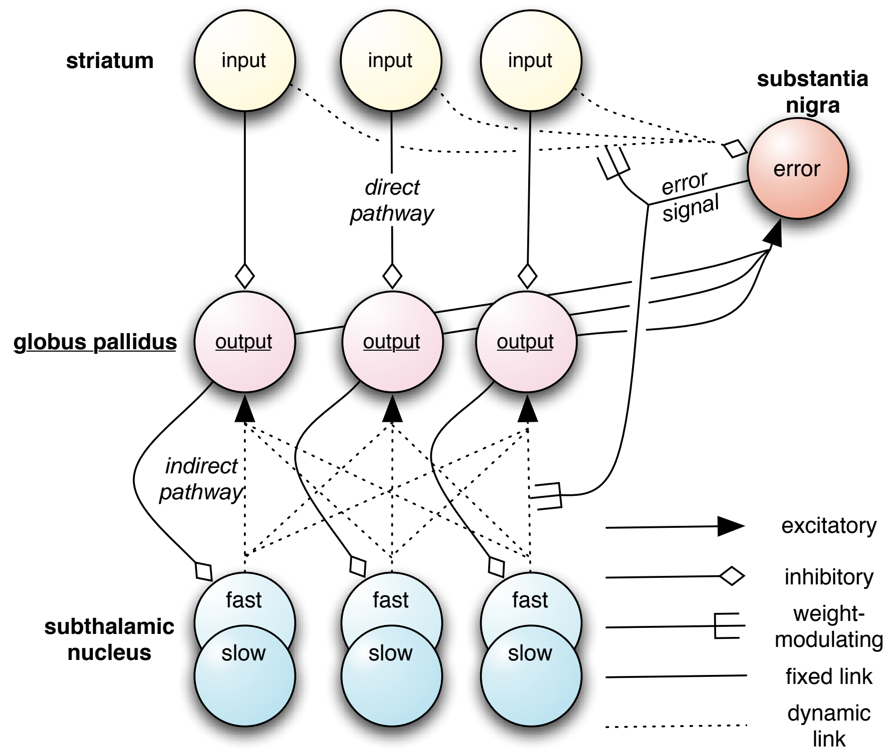

Figure 3.3: Berns and Sejnowski (1998) computational model of the basal ganglia. The *striatum* (STR) neurons represent the input layer of the structure, which have a direct pathway to the *globus pallidus* (GP) output layer. The *subthalamic nucleus* (STN) consist of both fast and slow neurons with independent projections, and can be viewed as the hidden layer. The indirect pathway GP -STN-GP should predict the next STR input. The *substantia nigra pars compacta* (SNc) monitors the difference between predicted and actual values, and applies the error signal as dopaminergic Hebbian learning (Hebb, 1949). This can be viewed as the critic in actor-critic models, see section 2.1.1 on page 4.

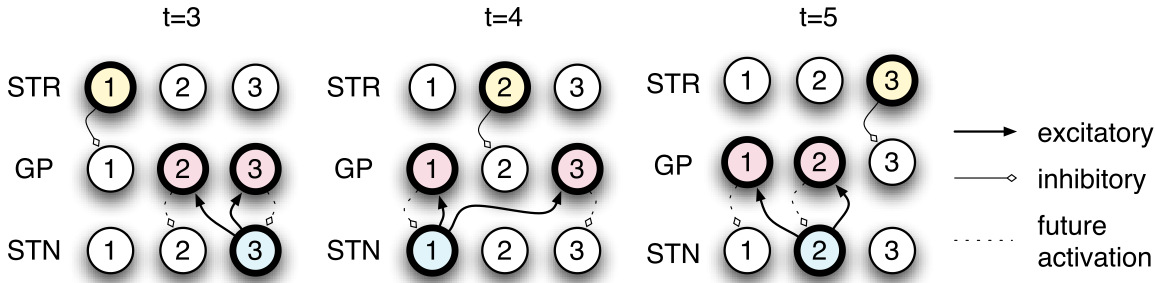

Figure 3.4: Sequence learning in the Berns and Sejnowski's model. The model learns the sequence 1, 2, 3, as in figure 3.1 on page 19. The *subthalamic nucleus* (STN) units are inhibited by the *globus pallidus* (GP) units active in the previous time step, otherwise tonically active. The excitatory STN-GP weights are learned to activate the non-selected GP units.

thalamus, and so inhibiting the selected GP unit will disable the corresponding thalamus inhibition, effectively enabling the corresponding action. Example: When *striatum* unit 2 is active, all but GP unit 2 should be active, inhibiting all but action 2 in the *thalamus*. See figure 3.4 and 2.6 on page 11.

The selected action is assumed to be the GP unit with the *lowest* firing rate, this can be called 'loser-take-all'. In this way, the basal ganglia can inhibit all thalamic areas except the selected action. See section 2.3.1 on page 10 for details.

The STN-GP connections in the indirect pathway are excitatory, and will activate the GP units that are not inhibited by *striatum*. All STN units project to all GP units, and the initial weights (zero) gives no preference to any of the links. The model is to learn these weights as to maximise the selection so that all but the selected GP unit will be as active as possible. As the signals are delayed, this means that effectively the model is to predict which GP units should be activated.

This model excludes the *thalamus* and the *cortex*, treats *globus pallidus external* (GPe) and *globus pallidus internal* (GPi) as a single nucleus, simplifies SNc and ignores *substantia nigra pars reticulata* (SNr), but it is still one of the computational models of the basal ganglia that is more accurate to the anatomical structures (Wörgötter and Porr, 2005).

The *subthalamic nucleus* (STN) is modeled as a *leaky integrator*, as in continuous-time recurrent neural networks (CTRNN) described in section 2.4 on page 14:

$$\frac{\mathrm{d}B_i(t)}{\mathrm{d}t} = \frac{1}{\tau}\big(- B_i(t) - \alpha G_i(t - \delta)\big) \tag{3.1}$$

where $B_i(t)$ is the activity (firing frequency) of the STN , $G_i(t)$ the activity of the matching GP unit (with an inhibitory synapse), τ is the time constant and δ the synaptic delay between the GP and STN units. α is a scaling factor for the magnitude of the inhibitory connection from GP .

Berns and Sejnowski discretise equation 3.1 on the previous page using the forward Euler method:

$$B_i(t) = \sigma\left(B_i(t-\Delta t) + \frac{\Delta t}{\tau}\Big(- B_i(t-\Delta t) - \alpha G_i(t-\delta)\Big)\right) \tag{3.2}$$

as a modified version of equation 2.3 on page 16, including the sigmoidal transfer function σ, which is an extended version of equation 2.2 on page 14 introducing a gain γ and a bias β:

$$\sigma(x) = \frac{1}{1+e^{-\gamma(x-\beta)}} \tag{3.3}$$

Berns and Sejnowski (1998) introduce a term λ representing the time constant scaled by the size of the time step:

$$\lambda = \frac{\tau}{\tau + \Delta t} \tag{3.4}$$

Substituting equation 3.4 into equation 3.2 yields the form used in the update rule of Berns and Sejnowski (1998):

$$B_i(s) = \sigma\big(\lambda B_i(s-1) - (1-\lambda)\alpha G_i(s-n)\big) \tag{3.5}$$

In this discrete version, s represents the discrete time step[1], $n = \frac{\delta}{\Delta t}$ the synaptic delay as a number of time steps and Δt is the length of a time step. The transfer function σ is described by equation 3.3 and normalises the STN output between 0 and 1 to represent the firing frequency.

What is special in this approach is that the transfer function also influences the internal state (the membrane potential) of the STN . As shown in section 4.2.2 on page 43 this reduces the effect of the time constants, and sets an implicit bias.

The system output is the firing frequency of the GP units, which is calculated

[1] Berns and Sejnowski (1998) uses t to represent both the time and the time step. To avoid this ambiguity, this text will use s for representing the time step in discrete time, while t represents the actual time in differential equations.

without using a leaky integrator:

$$G_i(s) = \sigma\left(\sum_j w_{ij}(s)B_j(s) - \alpha S_i(s) + \eta_i\right) \tag{3.6}$$

Here w_{ij} is the weight of the connection between STN unit j and GP unit i, which is a dynamic weight and therefore depending on the current time step s, and the output of the GP unit G_i is calculated by equation 3.5 on the facing page. S_i is the activity of the corresponding STR unit, i.e. the input to the basal ganglia model. η_i is described as the level of noise drawn from a uniform distribution. See section 4.2.1 on page 40 for a discussion on how this can be interpreted.

The SNc is modeled as an error unit, calculating the mismatch between the direct and indirect pathway. The GP output is compared to the weighted STR input:

$$e(s) = \sum_i \big(G_i(s) - v_i(s)S_i(s)\big) \tag{3.7}$$

where $v_i(s)$ is a dynamic weight representing the connection between STR unit i and the SNc .

This error signal is used for updating the weight of the connection between STN unit j and GP unit i. Every STN unit has connections to all GP units, and the connection strength w_{ij} is updated using a Hebbian learning rule (Sutton and Barto, 1981):

$$\Delta w_{ij}(s) = \rho_w\big(e(s)G_i(s) - S_i(s)\big)B_j(s) \tag{3.8}$$

where $\rho_w = 0.05$ is the learning rate, $e(s)$ the error signal as calculated by equation 3.7, G_i the GP output as equation 3.6, S_i the STR input, and B_j the STN output as by equation 3.5 on the facing page.

A simpler update rule is used for the connections from the STR to the error unit SNc :

$$\Delta v_i(s) = \rho_v e(s)S_i(s) \tag{3.9}$$

where the learning rate is $\rho_v = 0.1$ in the experiments described by Berns and Sejnowski (1998).

Experiments and results

Berns and Sejnowski trained their network to learn sequences. The sequence $1, 2, 3, 4, 2, 5$ was chosen because it is simple, yet the model would have to distinguish between 3 or 5 following 2. For each time step, the sequence was iterated, and the STN unit corresponding to the number from the sequence was made the only active STN unit.

During training, the activity of the non-selected GP units gradually raised as STN-GP weights were modified by equation 3.8 on the previous page. Figure 4.3 on page 36 shows how the model learned to increase the weights to GP unit 2 from STN units 2, 3 and 5, thus predicting that if STN unit 1 or 4 are active, GP unit 2 is the next to be selected.

After 200 time steps of training, the GP outputs were following almost exact the inverse of the STR inputs, such that the STN units increased the activation of non-selected GP units, but not the action that is inhibited by STR . This showed that the model was able to learn the sequence.

Berns and Sejnowski tested the network after learning the sequence, by cueing it by activating the first STR unit of the sequence. The network was able to reproduce the sequence without any further inputs from the *striatum*. The GP outputs were noisy, but the correct unit had minimal activity at each time step.

3.4.3 Prescott et al.

Prescott et al. (2006) presents a model of the basal ganglia embedded in a robot. The model is based on the authors' previous work Humphries and Gurney (2002), and performs effective action selection for a robot tested under conflicting situations.

The main idea of Prescott et al. is that *the basal ganglia acts as an* action selection *mechanism – resolving conflicts between functional units that are physically separated within the brain but are in competition for behavioral expression.*

The basal ganglia model is embedded in a robot and given the task to select actions depending on sensory and motivational conditions. The challenge is for the model to perform clean action selection even with multiple high-salience alternatives.

Figure **??** on page ?? show the pathways in the model suggested by Prescott et al. (2006). Note that authors have based their model of the rat basal ganglia instead of the human version, which main difference is that the GPi is replaced by the entopeduncular nucleus (EP). In addition, the two main types of dopaminergic receptors in striatum, D1 and D2 are treated separately, and used as parts of the functional *select* and *control* pathways, as presented in Gurney et al. (2001).

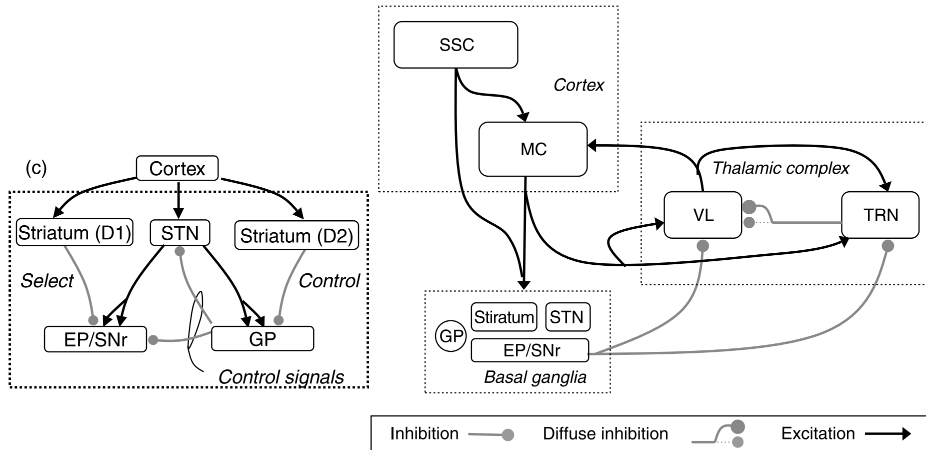

Figure 3.5: Humphries and Gurney basal ganglia model. Box c) to the left shows the internals of the basal ganglia model, while the rest of the figure shows how the basal ganglia is connected to the cortex and the thalamus. The pathways are considered performing *selection* and *control*. Reproduced from **?**.

This model also includes the GP -thalamic-cortex pathways and internals, thereby completing the motor loop and making the model far more compelex than that of Berns and Sejnowski (1998).

As the works of Prescott et al. has shown action selection embedded in a live robot, their model is also examined in this work and compared with the Berns and Sejnowski model.

3.5 Experiment details

This section describes and justifies technology choices done while performing this work. First, an attempt to model the basal ganglia using C++ is presented, followed by a discussion on why prototyping using Python turned out to be more convenient.

The rest of this chapter details how implementing the Berns and Sejnowski model was achieved.

3.5.1 Implementation frameworks

Prototyping using C++

Initial prototypes were developed using *C++* (iso, 2003), on the basis of the possible fast execution time of compiled code, in case the simulations were to be scaled up for a robot simulation or tuned by genetic algorithms, both requiring massive amounts of evaluations. Many existing robot simulation environments such as YAKS (Carlsson and Ziemke, 2001) are developed in C or C++, and could be easy to combine with a basal ganglia model in C++.

C++ is also an object-oriented language which was considered suitable for experimenting with different network layouts and techniques for modeling the *basal ganglia*. As a basis for building artificial neural networks in C++, the open source library *annie* was used and extended.

Annie (Shankar) is a library for building artificial neural networks, implemented in C++ with an object-oriented design. In *annie*, a network is built by instantiating classes like `RecurrentNeuron` within a `Layer` of a `Network`. Neurons can be connected to each other by method calls, and parameters such as the activation function can be set individually.

The following code extract shows how the neurons for the *globus pallidus* and *subthalamic nucleus* were created and connected in the basal ganglia prototype developed in C++.

```
/* addNeuron(layer) creates the neuron and adds it to the layer */
RecurrentNeuron *n_gp = addNeuron(gp);
RecurrentNeuron *n_stn = addNeuron(stn);

n_stn->setActivationFunction(ganglia::biased_sigmoid);
n_gp->setActivationFunction(ganglia::biased_sigmoid);

// direct pathway
n_gp->connect(n_str, STR_GP_WEIGHT);
// indirect pathway
n_stn->connect(n_gp, GP_STN_WEIGHT);
```

Being able to specialise each type and connect the network step by step was considered important for building a basal ganglia model, and *annie* proved to be the best tool for doing this in C++. Using the *annie* library simplified development and shifted focus from evaluating neural networks to the structure of the network.

C++ is designed to be used for industrial strength software development. As such,

the language utilises strong typing and generally making code rigid, for instance by distinguishing between private or public methods, or tagging methods and parameters as constant. These are valuable features when designing business software for stable, long term use. As C++ is based on C, the C++ developer occasionally also has to deal with pointers and memory allocations.

However, when designing scientific prototypes these features seems to be not as important as for the IT consultant. While building a functional neural network model of the basal ganglia, focus was kept at the C++ level, by changing the interfaces to allow access to private methods and attributes, casting and converting between different types, and debugging segmentation faults.

It was concluded that C++ was not the best choice of language for prototyping scientific challenges such as modeling the basal ganglia. It was desirable to use a higher level language resembling the mathematics, not the implementation.

Prototyping using Python

Initially as an experiment to verify outputs from the model, after two months of C++ coding, the current prototype was reimplemented in the programming language Python (van Rossum, 2005). The reimplementation, although much simplified compared to the C++ model at the time, was completed in less than a day. This gave a strong indication that C++ was not the right language for experiments which constantly demanded changing structures and constants, extensive statistical logging and debugging through inspection.

Further development were therefore performed using Python. Python is a high-level object-oriented dynamic programming language, which some of the main goals are that it should be easy to learn, write and *read*. Compared to many languages, Python code can be straight forward, it has been likened to *executable pseudocode*.

Compared to languages as C++ and Java, Python provides faster prototyping and makes it is easy to experiment with language features and your own code. However, as it is an interpreted and not a compiled language, execution time is not comparable to C++. It was considered that slightly larger execution time was not important compared to the goal of building a suitable model of the basal ganglia.

There are many open source third-party libraries available for Python, including an numeric library called *NumPy* (Oliphant). *NumPy* provides among other things highly optimised methods for matrix calculations.

NumPy was chosen because calculating artificial neural networks using linear algebra

gives cleaner code than using traditional for-loops. For instance, calculating a time step in a CTRNN (see section 2.4 on page 14) for all neurons, and even supporting unique time constants for each neuron can be expressed as 4 lines of Python code:

```
inputs = numpy.matrixmultiply(output, weight)
change = timestep/timeconst * (-potential + inputs + bias)
potential += change
output = map(transfer, potential)
```

In the code above, the variables `output`, `potential`, `bias`, `inputs` and `change` are all n-sized vectors, while `weight` is an $n \times n$ sized matrix. The code is easily compared to a vector version of equation 2.3 on page 16:

$$\vec{y}(t + \Delta t) = \vec{y}(t) + \frac{\Delta t}{\vec{\tau}}\Big(- \vec{y}(t) + \vec{u}(t) \times W + \vec{\theta}\Big) \tag{3.10}$$

$$\vec{u}(t) = \sigma\Big(\vec{y}(t)\Big) \tag{3.11}$$

In addition to making code clearer, the matrix operations of *NumPy* are implemented using static compiled languages as C and FORTRAN and exploit CPU vector features such as the Altivec engine (Diefendorff et al., 2000), which in informal tests on the basal ganglia model gave a considerable speed-up compared to pure Python code, in some cases by a factor 30.

In this work, experimenting with network shapes and layout was essential, and so keeping a high-level view of the calculations seemed like a reasonable approach, justifying the choice of using *NumPy*.

3.5.2 Implementing the model of Berns and Sejnowski

In the attempt of reproducing the results of Berns and Sejnowski (1998), a model directly mirroring the equations as described in section 3.4.2 on page 21 was implemented in Python. Each equation was implemented as a function. For instance, equation 3.6 on page 25 was implemented as:

```
def calc_GP(self, i):
    sum = 0.0
    for j in range(self.inputs*2):
        sum += self.w[i,j]*self.STN[j]
    noise = random.uniform(-0.25, 0.25)
    result = sum - self.effect * self.STR[i] + noise
    return self.sigmoid(result)
```

Some implementation details were not immediately evident from the paper, for instance the use of noise, as detailed in section 4.2.1 on page 40. After experimenting with the effects of the different possible interpretations, it was concluded that for *noise drawn from a uniform distribution* with a *magnitude* of 0.5 (Berns and Sejnowski, 1998) should be implemented as a random value drawn from the uniform distribution between -0.25 and 0.25, redrawn at each time step and for each input.

3.5.3 Implementing the model of Prescott et al.

The model by Prescott et al. (2006) was implemented using the Python CTRNN library developed. The reported task of action selection ..

Tests shows it works. Most notably, the Prescott model does not include the concept of learning. OK!

Chapter 4

Experimental Results

This chapter presents the results of reproducing the Berns and Sejnowski model. Results from the reimplementation are compared to results in the original work.

Finally, this chapter describes how details of the implementation was refined to clarify details from the original work.

4.1 Reproducing the model of Berns and Sejnowski

As described in section 3.5.2 on page 30, the exact equations from Berns and Sejnowski (1998) was implemented to reproduce the results of their basal ganglia model. The reimplementation results matched the original results only partially. This section details the expected and achieved results of the direct reimplementation.

4.1.1 Globus pallidus activity

Figure 4.1 on the following page shows the expected *globus pallidus* (GP) activity while learning a sequence as reported by Berns and Sejnowski (1998). While the GP units initially do not fire very organised, after learning they should approach the inverse of the *striatum* (STR) inputs. When action n is selected in STR , all GP units except unit n should be fully activated, while unit n should be fully inhibited. As the GP projects to the *thalamus*, this should inhibit all actions except action n in the *thalamus*.

The reimplementation results for the GP activation, shown in figure 4.2 on the next page, shows that the initial activities are somewhat diffuse, although the selected actions are inhibited. Since the noise is drawn from a random distribution, the activities of non-inhibited units does not match those of figure 4.1 on the following page. After learning,

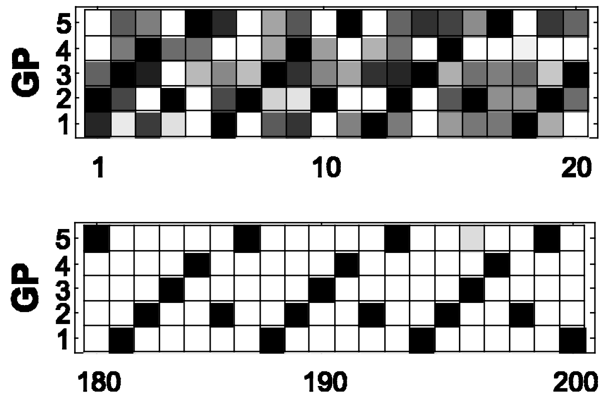

Figure 4.1: Globus pallidus activities during learning in Berns and Sejnowski (1998). Shading indicates the *globus pallidus* (GP) activity while learning the sequence $1, 2, 3, 4, 2, 5$, grading from from black (0) to white (1). In the initial time steps, GP unit firing is disorganised, but after 200 time steps of training, the activities almost exactly mirrors the inverse of the input sequence, where the selected unit is fully inhibited and the other units fully activated. Reproduced from Berns and Sejnowski (1998, fig. 4a).

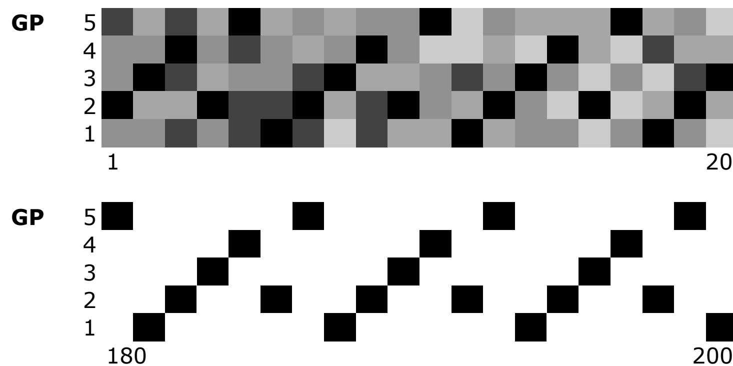

Figure 4.2: Globus pallidus activities during learning in reimplementation. Compare with figure 4.1. The reimplementation activities resembles the results of Berns and Sejnowski (1998), after 200 time steps the units are either fully inhibited (selected) or fully activated.

the non-selected units are fully activated due to the learning of the weights from the *subthalamic nucleus*.

4.1.2 Weight learning

The basal ganglia model is to learn to predict a sequence. In the implementation of Berns and Sejnowski this prediction is shown by exciting the GP units that are *not* going to be selected in the current time step.

The direct pathway directly inhibits the selected GP unit corresponding to the active STR unit. The indirect pathway through *subthalamic nucleus* (STN) excites all the other GP units. As the indirect pathway in this model is delayed and has to react according to previous GP activities, the model will have to learn to predict which GP units are to be excited.

As an example, the GP unit 2 is never inhibited two time steps at a row while learning $1, 2, 3, 4, 2, 5$. The time step after GP unit 2 has been inhibited (selected), the STN unit 2 (assuming short time constants[1]) will be tonically active, since it will no longer be under inhibition from GP unit 2. Since the GP unit is never selected twice in a row, an active STN unit 2 is an indicator that GP unit 2 should be excited. The same is true for active STN units 3 and 5, as in the sequence 2 does not follow 3 or 5.

The model of Berns and Sejnowski (1998) correctly learns these weights, strengthening the connections from STN units 2, 3 and 5, and weakening the others. Figure 4.3 on the next page shows how the weights of connections from the short time constant STN unit is learned.

The connections to GP unit 2 from STN units 3, 5, and particularly 2 shows the direct relations between firing of STN units and GP units. Since an STN unit is tonically active unless inhibited by the corresponding GP unit, the mapping will also show the sequencing.

However, STN units 1 and 4 are both active at the same time as the GP unit 2, because in the sequence, 2 follows both 1 and 4. Thus there is not any strong activation from these units, in fact the learning approaches these weights to zero.

In the reimplementation of the Berns and Sejnowski model, the weight changed in a similar way, as shown in figure 4.4 on the following page. The weights from GP unit 1 and 4 do not increase as much as the predicting units, and the weight from the GP

[1]For longer time constants, the STN units will be active for more than 1 time step, thus blurring the activity levels over time, but the peak activity will still be in the time step after the corresponding GP selection.

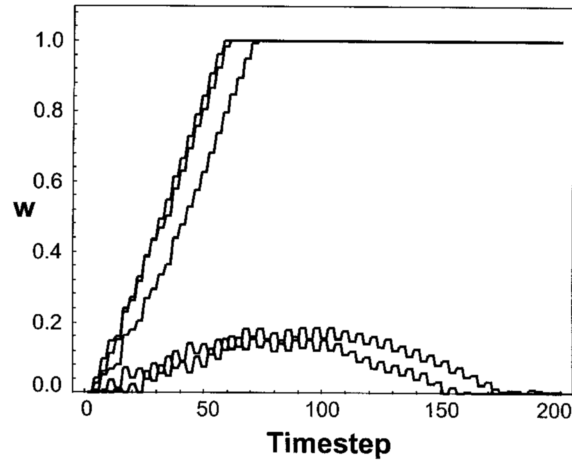

Figure 4.3: Changes in connection strengths when learning sequence (original). Weights from the five *subthalamic nucleus* (STN) units with short time constants to *globus pallidus* (GP) unit 2 are shown while learning the sequence $1, 2, 3, 4, 2, 5$. The three weights that increased to saturation are from STN unit 2, 3 and 5, units that were not active prior to GP unit 2 firing. When STN units 1 and 4 were active, GP unit 2 was inhibited by the *striatum* (STR), and therefore the weights for these connections were not significantly increased. Reproduced from Berns and Sejnowski (1998, fig. 5).

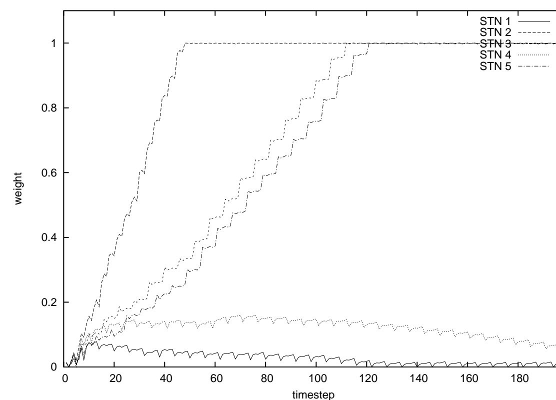

Figure 4.4: Changes in connection strengths when learning sequence (reimplemented). Weights from the five *subthalamic nucleus* (STN) units with *long* time constants to *globus pallidus* (GP) unit 2 while learning the sequence $1, 2, 3, 4, 2, 5$. The weight changes resembles those of figure 4.3, except that the connection from STN unit 2 grows faster.

unit 2 grows three times as fast as the weights from units 3 and 5. There is an possible explanation for this, even though the three units equally predicts that GP unit 2 is not to fire in the next time step. The STN unit 2 fires twice as often as the others, and the weight is therefore strengthened twice as often. In addition, the pre-synaptic activity part of the weight change equation 3.8 on page 25 makes the weight grow proportionally to the STN output, which in average will stay higher in unit 2, both due to rapid activation and long time constants.

Note that the reimplementation figure shows the development of the connections from the *long* time constant STN units. The weights from the short time constants (not shown) also developed strong connections from STN units 2, 3 and 5, but did not feature the "rise and fall" of weights from unit 1 and 4. With short time constants, the STN 1 and STN 4 weights were learned to be zero after just a few time steps. This seems reasonable, for short time constants the STN units will only be active immediately after the activation of the corresponding GP unit, as shown by the sawtooth shape in figure 4.5 on the following page.

4.1.3 Error values

The error values as produced by equation 3.7 on page 25 did not behave as expected in the reimplementation. According to figure 6b) in Berns and Sejnowski (1998), the error should start out quite stochastic with a maximum at about 3, and then approach an area near zero, due to the learned STN to GP connections and the increasing STR to GP inhibiting connections. Figure 4.6 on the following page shows the expected error value distribution, while figure 4.7 on page 39 shows the error value distribution of the reimplementation.

There is at least one possible reason to why the error function 3.7 on page 25 does not perform as expected. During learning, all but one STR units outputs 0, while the active unit outputs 1. The goal of the learning is for all GP units except one to output 1, while the selected unit is totally inhibited.

Since the given error equation sums the GP activity and subtract a weighted sum of STR activity, it should approach the $n-1$ fully active GP units minus one fully active STR unit. So for a sequence with $n = 5$ different values, the error should approach 3, which matches figure 4.7 on page 39.

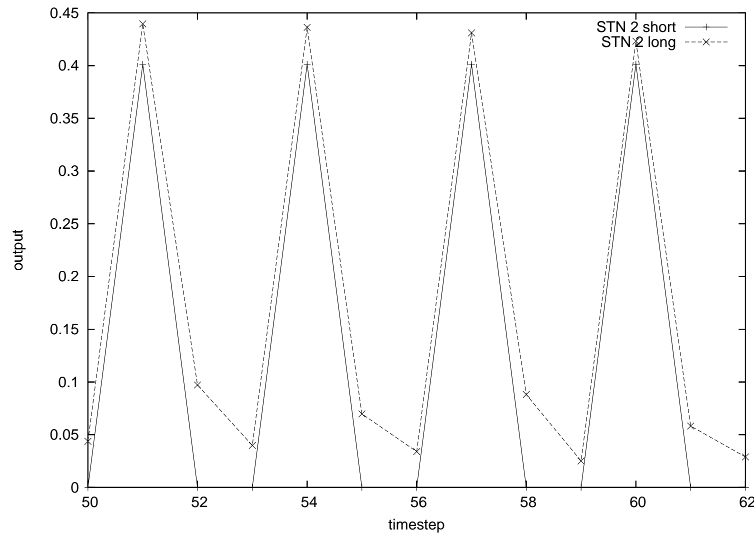

Figure 4.5: Firing of subthalamic nucleus unit 2. The *subthalamic nucleus* (STN) unit with a short time constant gives a sawtooth shaped output, reacting solely to the current signal. The unit with a long time constant decays after the input signal, but not fast enough to reach zero before the next input. Note that the inputting *globus pallidus* (GP) unit 2 will fire twice as often as the other units.

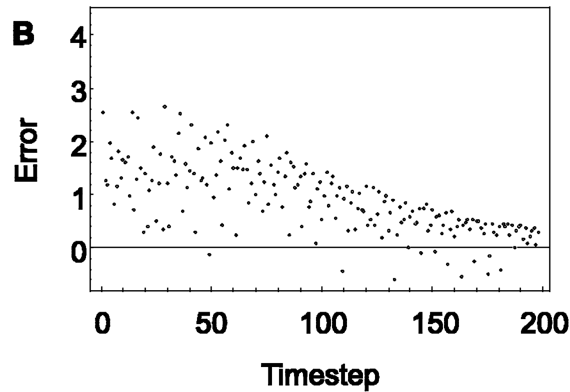

Figure 4.6: Error value distribution (original). The error signal is calculated by equation 3.7 on page 25, the difference between the weighted sum of the *striatum* (STR) activity and the sum of the *globus pallidus* (GP) activity. The error signal converges to zero. Reproduced from Berns and Sejnowski (1998) figure 6b.

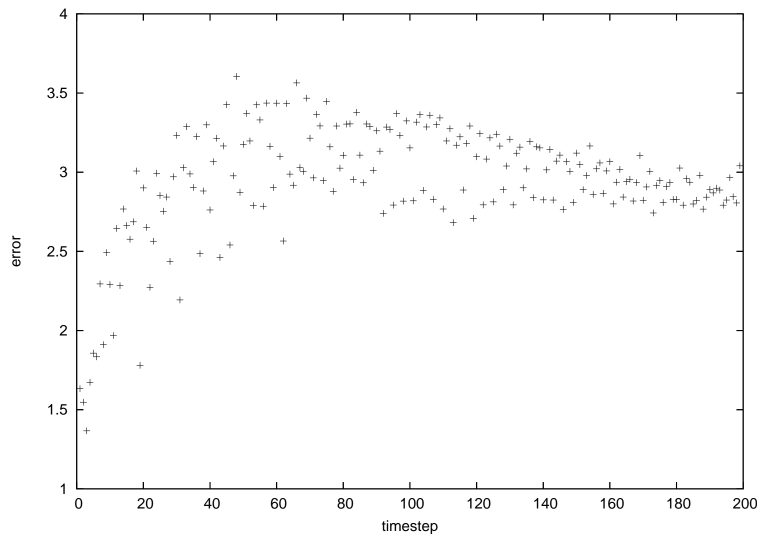

Figure 4.7: Error value distribution (reimplemented). The error signal is calculated by equation 3.7 on page 25 as in figure 4.6 on the preceding page. The error signal does not converge to 0, but to 3.

4.1.4 Reward

Berns and Sejnowski (1998) defines the reward for the model as the sum of the GP activities, being proportional to how well the GP matched the inverse of the STR activity. The reward can be seen as the saturation level of the non-selected GP units. As figure 4.2 on page 34 showed, after learning the non-selected GP units were fully active, so that the total reward converges to 4. This is true both for the original reward plot (Berns and Sejnowski, 1998, fig. 6a) and in the reimplementation, and so the reward plot for the reimplementation is not included here.

4.1.5 Playback

The original work reports that the model is able to playback the sequence learned. This playback is 'hinted' by inputting a single item of the sequence into the striatum, and then the network plays back the rest of the sequence. This is achieved because of the GP -STN -GP loop.

In the reimplementation, this features was not reproducable. While the model was

to reproduce the looping sequence $0, 1, 2, 3, 1, 4, 0, 1, 2, 3, 1, 4,$ the reimplementation returned for example $0, 1, 1, 0, 2, 4, 0, 1, 1,$ or $0, 1, 0, 1, 0, 1, 2, 3,$ but not even looping. The outcome seems to rely on the random noise and not the learned weights. Without noise, the output didn't change much between timesteps.

Several attempts were made to try to reproduce the playback, but with no success. It was concluded that due to several mechanism being wrongly described by the original work, such as the error function or the leaky integrator, it was difficult to fix the model.

One reason for why the reproducability does not work is that the network is trained with a STN inhibiting the GP neurons with a factor 10. During playback, this inhibition is lacking. As active GP units inhibit corresponding STN units, the 'selected' STN unit will not be active enough compared to the conditions under training.

4.2 Refinements

While attempting to reproduce the results of Berns and Sejnowski (1998), and for testing the robustness of the model, several parameters and techniques were varied in experiments. This section covers the details of how the implementation was refined and clarified.

4.2.1 Noise

In Berns and Sejnowski (1998), pre-synaptic *noise drawn from a uniform distribution* is added when calculating the GP potential. It is not clear from the paper what this means or what is its purpose. The *magnitude of random synaptic noise* η is said to be 0.5.

Several questions can be raised about Berns and Sejnowski's use of noise:

1. What is the effect and reason for adding noise to the system?

2. Is the noise time-variant or fixed at initialization?

3. What does it mean that the "magnitude" is 0.5? From which uniform distribution are the noise values drawn?

In robotic simulations, it is common to add noise to inputs to simulate the noise added by real sensors, to make the model robust for transferring to a real robot. This view is not explicitly explained in Berns and Sejnowski (1998), and the original work

only applies noise to the GP calculations. One can assume that it would be equally important to add noise to the STR inputs, which is binary 0.0 or 1.0.

By including the noise in only the GP calculations, the original works does not add noise to the error calculation or weight changes (both depend on STR inputs), but these calcuations do include the GP output, and therefore indirectly include noise.

It has been observed that adding noise can help search algorithms escape local minimas (Selman et al., 1994). This is also a valid reason for including noise, as the model is to find the correct weights, and the Hebbian learning network can be said to perform a local search.

Berns and Sejnowski use the notation η_i, the noise is made dependent on the GP unit i. This suggest that the noise is fixed by initialization per GP unit. In the reproduced experiments, this was achieved by generating a list of randomly chosen numbers at the time of initialization, one number for each GP unit.

However, this made results from the playback testing to be highly dependent on the initial chosen noise. For instance, when trying to playback of the sequence $0, 1, 2, 3, 1, 4$ (a zero-based version of the Berns and Sejnowski sequence), some runs reproduced $0, 1, 2, 3, 3, 3, \ldots$, while other runs got stuck as early as $0, 1, 1, 1, \ldots$

In figure 4.8 on the next page the error value as calculated by equation 3.7 on page 25 is plotted for three different instances. Each instance is started with different randomly chosen noise levels that are kept constant through all time steps, but unique per GP unit. Noise values are chosen randomly from a uniform distribution $(0.0, 0.5)$. The model seems to be able to compensate for the error introduced by this noise after about 40 time steps, after that point there is not any noticeable difference between the error values of the different instances.

The reason for applying noise in this fixed manner is not clear. It is interesting to note that the model is able to compensate for the constant noise, but the relevance of this can be discussed. It is possible that adding this constant noise can make it easier for the model to differentiate between different GP units so to avoid fluctuations in the Hebbian learning.

To test this, the experiments was repeated without any noise. The tests revealed the fact that having only positive noise levels uniformly distributed in the interval $(0.0, 0.5)$ effectively is a positive bias to the GP units with an average of 0.25. This makes it relatively easy for the model to achieve the desired GP state of approaching 1.0 for inhibited passive actions.

However, the floor at ~ 0.0 for selected (disinhibited) actions seems unaffected by the bias during learning, due to the *striatum*-GP connection being enhanced by a factor

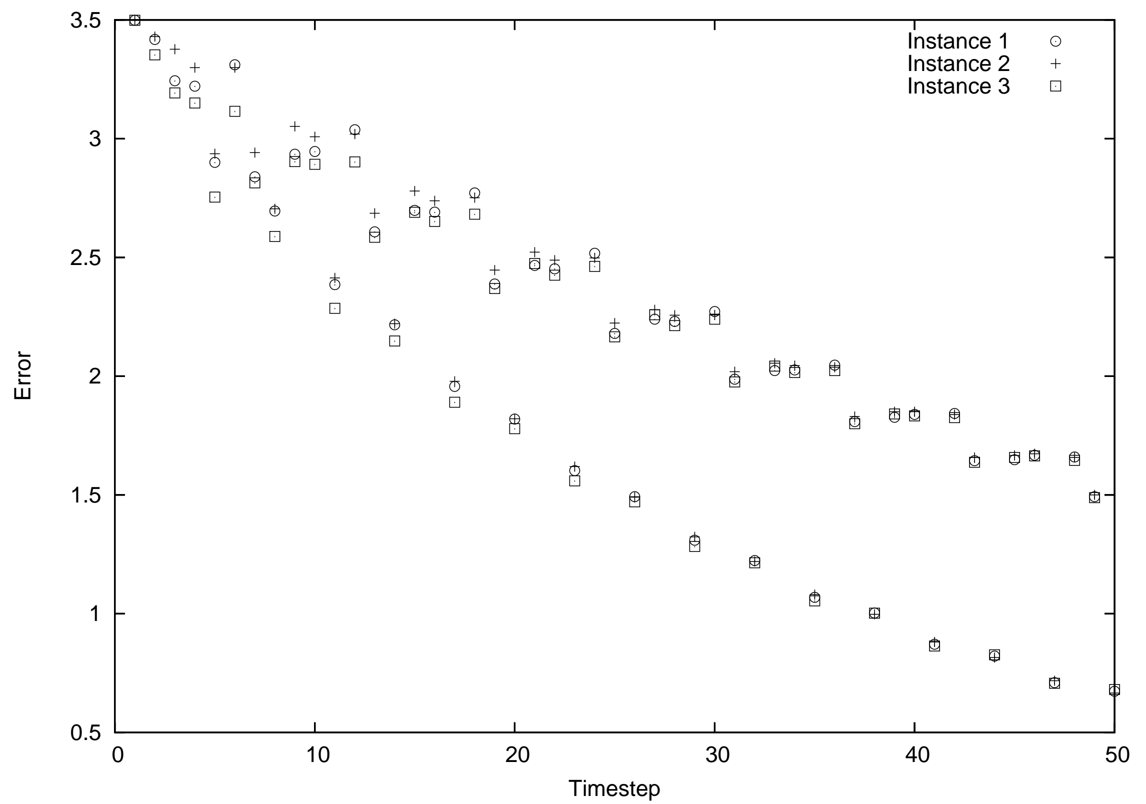

Figure 4.8: Error values with three different instances of random noise. Each instance of the model is initialised with random noise per *globus pallidus* (GP) unit that are constant throughout the simulation. The error of the the different instances is at first notably different, but after about 40 time steps, the differences in error levels are not noticeable.

$\alpha = 10$, as shown in equation 3.6 on page 25. This scaling factor, named *relative effect of an inhibitory synapse to an excitatory one*, is left unexplained in Berns and Sejnowski (1998), but in effect it will make an active STR unit always inhibit the corresponding GP unit. Weights are constrained to a maximum of 1.0, and there are 10 STN units in the experiments with a sequence of 5 different actions, the STN units will never alone be able to activate a GP unit under inhibition by *striatum*.

Figure 4.9 on the following page shows the firing rate of a GP unit without noise, two instances with randomly chosen constant noise, and an instance where the bias is fixed at 1.0. This figure shows that instances with a fixed positive noise climbs much faster to the desired inhibiting GP state than the noiseless instance. However, this effect is not due to some hidden features of the 'random' noise, but simply due to the positive bias, as proven by the instantly near perfect results of the GP unit with a fixed bias of 1.0. In fact, if experiments are repeated with constant noise chosen from the interval $(-0.25, 0.25)$ instead, GP units with negative noise will perform worse than the noiseless version.

However, such noise is random throughout the simulation and not fixed at the point of initialization. Thus, the noise is dependent on both the time and GP unit, which can be denoted as $\eta_i(t)$.

From the experimental results described above, it was deduced that if noise is to be of any use in the model, it has to be random over time, and it should be both negative and positive. The effect of noise that is only positive is equivalent to GP neurons having a positive bias. Although the use of a positive bias in this context does seem to give some interesting effects, it is hardly needed to use noise to achieve this.

It is similarly interesting to note that the model is able to compensate for the constant noise over ~ 40 time steps, but the constant noise does not seem to add any other valuable effects for the system.

When observing weight changes and comparing noiseless instances with the constant random noise instances, there are no noticeable differences except for changes imposed by the bias.

If the noise is to be both positive and negative, centered about 0, and with a magnitude of 0.5, the uniform distribution must be of the interval $(-0.25, 0.25)$

4.2.2 Leaky integrators with sigmoidal updates

Berns and Sejnowski (1998) apply the sigmoidal activation function 3.3 on page 24. What is special is the way the activation function is applied compared to normal usage of

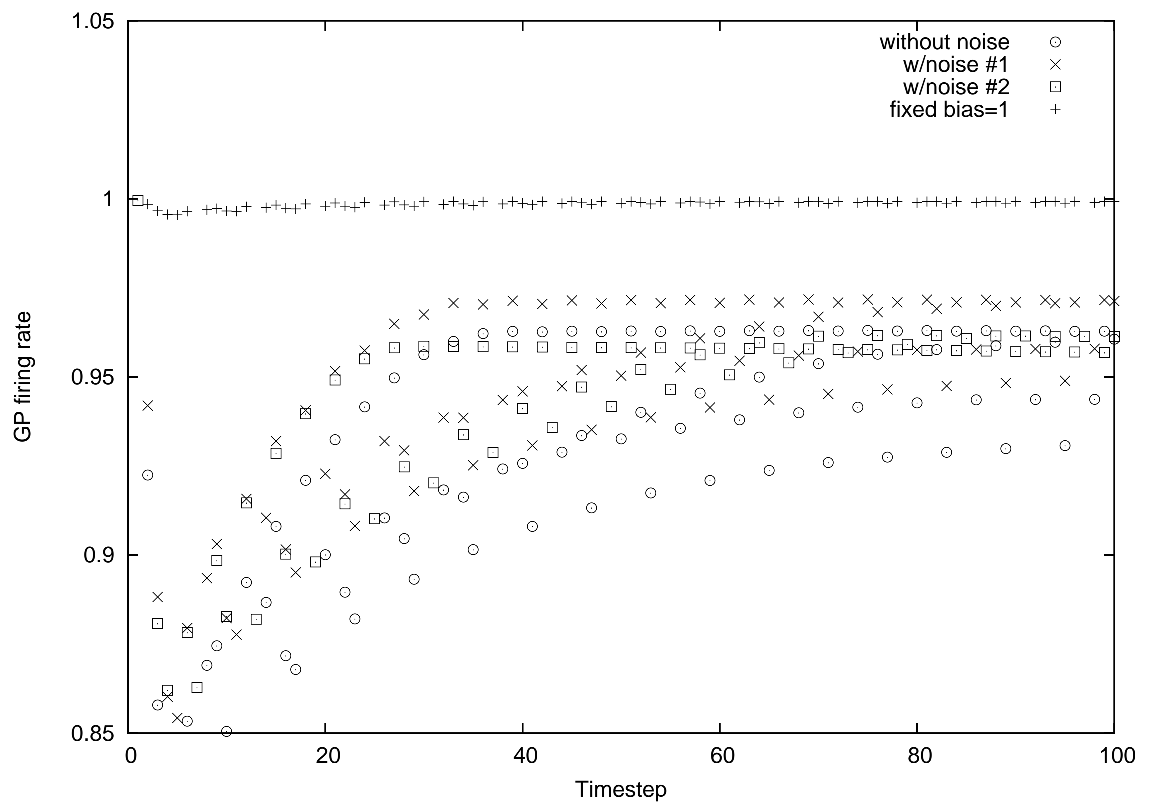

Figure 4.9: Firing rate of globus pallidus unit with and without constant noise. *globus pallidus* (GP) units with constant random positive noise achieve better results than a unit without noise. However, this is due to the positive bias, as shown by the better performing unit with a fixed bias of 1.0. Not shown: firing rates near 0.0 when inhibited by *striatum* (STR).

continuous-time recurrent neural networks (CTRNN) (Beer, 1995; Blynel and Floreano, 2002; Di Paolo, 2003). As pointed out in section 3.4.2 on page 21, Berns and Sejnowski use the result of the transfer function also to update the internal state of the STN neurons. This internal state, usually a representation of the neuron membrane potential, is therefore said to be equal to the output of the neuron, the firing rate.

As shown in the discrete update rule in equation 3.5 on page 24, the sigmoid function is applied several times, as simplified in:

$$B(s) = \sigma\left(\alpha B(s-1) + \beta\right) \tag{4.1}$$

$$B(s+1) = \sigma\left(\alpha\sigma\left(\alpha B(s-1) + \beta\right) + \beta\right) \tag{4.2}$$

To investigate the effect of using an iterative sigmoidal update rule, two different rules were applied in a Octave (Murphy, 1997) simulation, a function $u(t)$ representing the normal leaky integrator potential, while function $v(t)$ represents the sigmoidal update rule as in Berns and Sejnowski:

$$u(t) = u(t-1) + \frac{1}{\tau}\Big(-u(t-1) + I(t)\Big) \tag{4.3}$$

$$v(t) = \sigma\left(v(t-1) + \frac{1}{\tau}\Big(-v(t-1) + I(t)\Big)\right) \tag{4.4}$$

where τ is the time constant, σ the sigmoid function 3.3 on page 24 with parameters gain $\gamma = 4$ and bias $\beta = 0.1$ as in Berns and Sejnowski (1998). The implicit time step is set to 1 second, and so the time constant is set to $\tau = 9$, as a scaled up version of the slow STN suggested by Berns and Sejnowski where $\tau = 90$ms and $\Delta t = 10$ms. The input $I(t)$ is zero except in time step 10 and $17 \ldots 22$, thus representing one short and one long signal.

As shown in figure 4.10 on the next page, and discussed in section 2.4 on page 14, the normal potential $u(t)$ develops in a negative exponential way towards the current input. Thus, the longer the input, the closer the potential gets to the input voltage.

The sigmoidal transfer function applied as $\sigma(u(t))$ normalises the neural output to be in the range $(0, 1)$, but the bias $\beta = 0.1$ shifts the neutral output to 0.4 instead of 0.5. Other than that, the sigmoid output almost exactly traces the membrane potential. In fact, if $u(t)$ is shifted with $+0.4$, the two graphs will overlap except for the top around $t = 23$.

However, for the case of the incrementally applied sigmoidal rule $v(t)$, as described

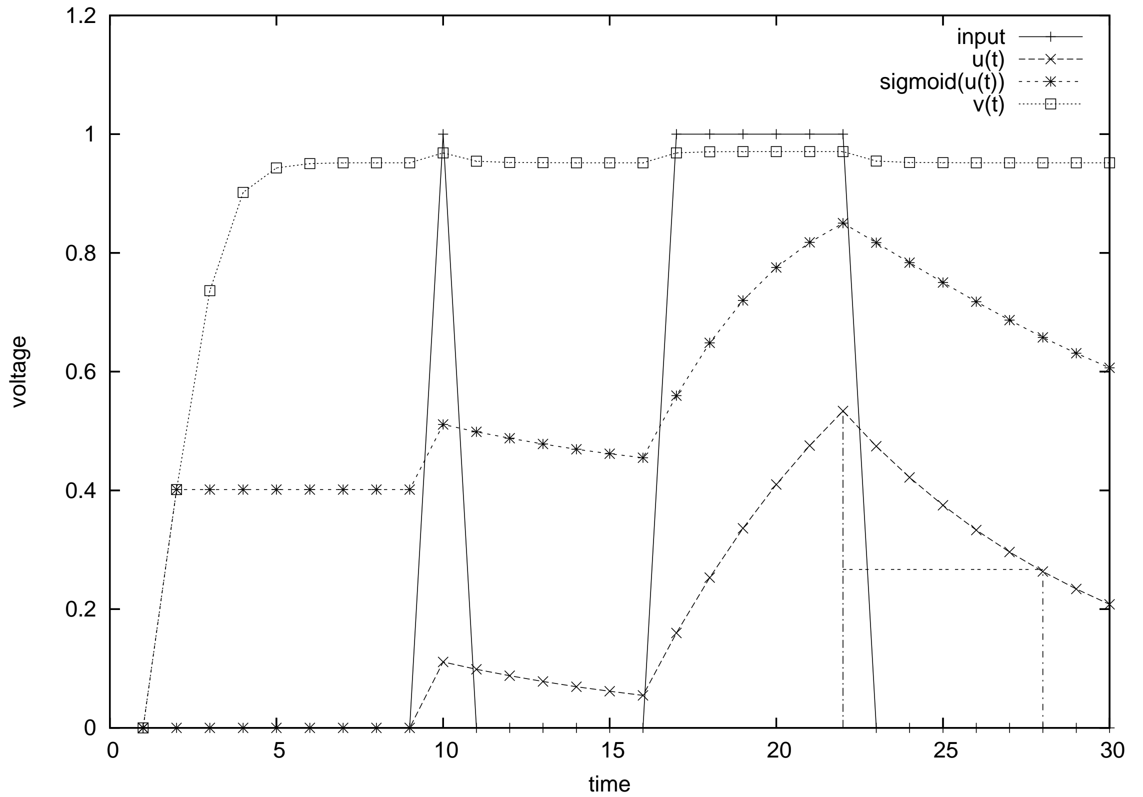

Figure 4.10: Effect of sigmoidal update rule on continuous-time recurrent neural network (CTRNN) neurons. $u(t)$ represents the membrane potential reacting to the input, as described by equation 4.3 on the preceding page. $sigmoid(u(t))$ shows the neural output after applying the sigmoid transfer function 3.3 on page 24 to $u(t)$. $v(t)$ pictures the membrane potential and neuron output as described by equation 4.4 on the preceding page, where the sigmoid function is part of the update rule.

by equation 4.4 on page 45, is very different. As shown for $t = 2$, the new value is calculated as $\sigma(0) \approx 0.4$, and will in further time steps increase towards ~ 0.97, even before any inputs are present.

As shown in the close-up of $v(t)$ in figure **??** on page ??, the potential is not changed significantly by an input of 1.0, and the sigmoidal update rule seems to not make much difference on the input duration. The first input signal lasting a single time step at $t = 10$ is not able to raise the potential as high as the longer signal at $t = 17 \ldots 22$. However, the potential charge flats out after about 3 time steps. The same is true for the decays, after 3 time steps the decay is not noticeable, this is true for both the short and long signal.

When comparing this to the output of update rule 4.3 on page 45, the decay rate is considerably faster for the incrementally sigmoid update rule 4.4 on page 45. The decay of the potential $u(t)$ is shown to be reduced by half from $t = 22$ to $t = 28$ in figure 4.10 on the preceding page. For the activation function $sigmoid(u(t))$ the output is decayed by half from $t = 22$ to $t = 28$ if we assume ~ 0.4 as the floor. Thus, the rate of halving is about 6 time steps. However, the incremental update rule decays by over 80% in just one time step, when assuming the floor of about ~ 0.952. In effect the incremental sigmoidal update rule shifts the time constant closer to 1.

The effect of the time constant can be said to delay the decaying of the signal and smooth the signal over time, thus providing an eligibility trace. When applying the incremental sigmoidal update rule as done in Berns and Sejnowski (1998), this delay effect is largely reduced, and makes it questionable why Berns and Sejnowski uses the *leaky integrator* with short and long time constants, but destroy the delay effect by applying the sigmoidal transfer function to the membrane potential.

Repeating the experiment with a lower time constant $\tau = 0.7$ revealed dampened oscillations as the input changed state. This is expected, as the time constant is lower than the time step, but not as low as half the time step, where the system becomes numerically unstable (Blynel and Floreano, 2002). The integrator tends to over-compensate the changed input, and this creates the oscillations. However, with time constants less than 1, $v(t)$ seems to be more stable than $sigmoid(u(t))$, as the integrated sigmoid method gives the effect of a time constant closer to 1 and dampens the oscillations.

4.3 Implementing a CTRNN

For supporting modelling basal ganglia using leaky integrators, a CTRNN library was implemented in Python. This library supports complex networks through a simplified API. A simple network is constructed by specifying the number of neurons, and various features such as the bias, time constants, transfer function and weights can be modified per neuron by using array indexing:

```
net = ctrnn.CTRNN(5)
net.bias[0] = 0.4
net.timeconst[3] = 1.4
net.transfer[1] = ctrnn.step
```

Network evaluations are performed using matrix operations of the library *NumPy* (Oliphant). The network can be considered as a set of vectors and matrices, and so setting the weights for all connections originating from neuron 4 can be achieved as:

```
for n in range(5):
    net.weight[4,n] = -1.5
```

The library supports a concept called layers. Unlike in feed-forward network, calculations on fully recurrent networks are not performed layer by layer, as any neuron can be connected to any other neuron, but for the whole network at once. The concept of a layer is still useful for addressing common properties.

In the developed library, layers are supported as a way of constructing a network. Instead of specifying the total number of layers, a network is constructed by building layers of different sizes. The underlying CTRNN network is built as normal, and the layers are presented to the programmer as slices of the network.

The code below shows how to use the library for a simple feed-forward network:

```
layers = ctrnn.Layers()
layers.add_input_layer("input", 3)
layers.add_layer("hidden", 6)
layers.add_layer("output", 2)
layers.build_net()
net = layers.net
# Connect input->hidden->output
for hidden in layers.hidden:
    for input in layers.input:
        net.weight[input,hidden] = random.random()
    for output in layers.output:
        net.weight[hidden,output] = random.random()
layers.input.set_inputs([0.4, 0.3, 0.9])
```

```
net.calc_timestep()
net.calc_timestep()
print layers.output.output
# array([ 0.89475212, 0.88843912])
```

Although a CTRNN with all time constants set to 1 and no recurrent connection can be used as a feed forward network, the user will have to calculate as many time steps as the depth of the network for the signals to propagate.

The library was used to reimplement the models of Berns and Sejnowski and Prescott et al.. However, as the simple reimplementation of Berns and Sejnowski (1998) failed to reproduce key features of the model, the full CTRNN model is not examined further in this text.

4.4 Reproducing the model of Prescott et al.

Prescott et al. (2006), as presented in section 3.4.3 on page 26, show a rather complex basal ganglia model, embodied in a robot to perform live action selection.

The developed CTRNN library, as described in 4.3 on the preceding page, was applied to attempt an reimplemention of the model described in the original work. The full network was successfully deducted from the equations provided.

4.4.1 Transfer function

A piercewise, linear transfer function is applied in the original work and is expressed as:

$$y = L(a,\theta) = \begin{cases} 0, & a > \theta \\ (a-\theta), & \theta \leq a \leq 1/(1+\theta) \\ 1, & a > (1+\theta) \end{cases}$$

When expressed as code, an bug is immediately evident:

```
def piecewise(a, theta):
    if a < theta:
        return 0.0
    if a <= 1.0/(1.0+theta):
        return a-theta
    if a > (1+theta):
        return 1
    logging.warning("Unknown piecewise value %s", a)
    return 1
```

The conditions set up do not cover all possible a values, because there could be a gap between $1/(1+\theta)$ and $1+\theta$. In addition, there is no guarantee that the value is bounded to $(0, 1)$ as expected by transfer functions.

```
>>> print prescott.piecewise(0.45, 0.5)
0.0
>>> print prescott.piecewise(0.59, 0.5)
0.09
>>> print prescott.piecewise(0.66, 0.5)
0.16
>>> print prescott.piecewise(0.67, 0.5)
WARNING:root:Unknown piecewise value 0.67
1
```

In addition, notice how the piecewise transfer function in this case will immediately jump from 0.16 to 1.0. When examining results for a negative $\theta = -0.9$ as used in the original paper, the transfer function will output values up to 10.9 before going for the 'upper' bound of 1.0:

```
>>> print prescott.piecewise(-0.85, -0.9)
0.05
>>> print prescott.piecewise(10, -0.9)
10.9
>>> print prescott.piecewise(10.1, -0.9)
1
```

It is evident that the authors did not use this function. The function they used has probably returned $a - \theta$, but restricted to the range $(0, 1)$. If the authors had presented the function using programming code this could have be done as:

```
def piecewise(a, theta):
    return max(1.0, min(0.0, a-theta))
```

4.4.2 Performing action selection

In the embedded version of the model, the robot runs in a sense-act cycle at 7 Hz. The authors did not want to run the basal ganglia with such large time steps. Instead of selecting a specific mapping between the two time steps, the original work runs the basal ganglia model to convergence at each robot time step.

This means that the network is evaluated until the *smallest* Δa *on two consecutive time-steps was less than 0.0001* (Prescott et al., 2006). Δa_i is the change in the leaky integration update rule. In the reimplementation, this statement was considered to

mean the *largest* Δa, as a network can be unstable even if a single output is stable. This functionallity was provided by the developed CTRNN library.

The original work tested the model's ability to perform action selection offline from the robot. This was achieved by using 5 different action channels, meaning 5 units at each layer STN , GP , etc. Salience inputs were provided to the somatosensory cortex (SSC) as vectors $(s_1, s_2, 0, 0, 0)$, where s_1 and s_2 were varied in each run.

Then the model was tested with increasing s_1 values from 0.0 to 1.0, and for each s_1 value testing with increasing s_2 values from 0.0 to 1.0. Both values were incremented in steps of 0.01. This algorithm can be expressed as:

```
for s1 in xrange(100):
    s1 /= 100.0
    self.net.output[:] = [0.0]*len(self.net.output)
    for s2 in xrange(100):
        s2 /= 100.0
        inputs = [s1, s2, 0.0, 0.0, 0.0]
        self.ssc.set_inputs(inputs)
        self.step()
```

What is to be achieved is that s_1 should be the only selected action in the output of the system, even as s_2 increases. In the original work, the model manages to do this even as s_2 is slightly stronger than s_1. At a point, s_2 becomes the new selected action, and s_1 is fully depressed.

The reimplementation using the CTRNN library fully confirmed this behaviour.

Chapter 5

Conclusion

This chapter reviews what has been achieved in this work. A discussion first describes how the basal ganglia can be compared to an actor-critic performing temporal difference learning. Following that, the implications of reproducing the original work are detailed.

A note on implementing scientific prototypes describes how different programming languages affect the outcome and how using mathematics instead of program code in scientific publications affects reproducibility.

The chapter closes with a summary of possible future work.

5.1 Summary

The computational model of basal ganglia as described by Berns and Sejnowski (1998) was reimplemented. The model was trained with a looping, simple sequence; it learned the weights predicting the sequence, and produced outputs as expected during training. However, it was not possible to reproduce the playback behaviour as described by Berns and Sejnowski. Several possible flaws in the original work were identified and analyzed.

5.2 Discussion

5.2.1 Basal ganglia as a TD-learning actor-critic

The human brain nuclei *basal ganglia*, as detailed in section 2.3 on page 7, can be described as an actor-critic architecture. Although the functional role of basal ganglia *in vivo* is not yet fully understood, it has been credited with motor control and action selection. The primary basal ganglia output layer, *globus pallidus* (GP), projects to the

thalamus, inhibiting motor control signals that would otherwise have been sent out from thalamus.

The basal ganglia receives inputs from the *cortex* to the *striatum*, and thus can be said to monitor the overall picture of the current context and state. Based on the current and earlier contexts, the basal ganglia inhibits selected GP units, effectively allowing selected motor control signals to pass through the thalamus. Thus the basal ganglia can be viewed as performing action selection.

The actor-critic architecture as described in section 2.1.1 on page 4 consists of an actor that performs actions on the environment, based on the context and reinforcement signals from the critic, who judges the actor's decision.

The model of Berns and Sejnowski mimics pathways in basal ganglia as described by neurology literature, but simplifies some structures. Their model can be compared to the actor-critic, assigning the actor role to the structures GP and *subthalamic nucleus* (STN), while the *substantia nigra pars compacta* (SNc) plays the role of the critic. The reinforcement signal modifies the weights of the STN-GP connections so that the basal ganglia output (the selected action in the GP) more closely matches the *striatum* (STR) context.

The critic can be said to provide the TD error signal as in temporal difference learning, described in section 2.2.2 on page 6. In TD-learning, the solution to the temporal credit assignment problem is to observe the difference between the system's own predictions over time. This can be compared to the 'internal' reward as observed in basal ganglia and classical conditioning experiments showing increased dopamine levels at the learned conditioned stimulus (CS).

In view of the Berns and Sejnowski (1998) model, this internal reward is given by the SNc calculated error, and used by the Hebbian learning rule to adjust weights. If we assume the *in vivo* basal ganglia does something similar, the observed dopamine release is an indicator that the basal ganglia performs temporal difference learning.

5.2.2 Reproducing Berns and Sejnowski

This work shows experiments with reimplementing the work of Berns and Sejnowski to be able to reproduce sequences in a basal ganglia inspired artificial neural network. The reimplementation was not as straight-forward as expected, as the model turned out to be inaccurately or possibly incorrectly described in Berns and Sejnowski (1998). The original work describes playback features that were not reproducible, as shown in section 4.1.5 on page 39.

Experiments revealed that Berns and Sejnowski have incorrectly described the error function as calculated by SNc . However, the reimplementation still managed to learn the weights predicting the sequence in a similar way to the original work. This suggests that the error term might not have been as important for Hebbian learning as the pre- and post-synaptic activity.

In the original work, noise was added to the system. In reproducing, the noise was first assumed to be positive and static in time. Results show that applying noise in this way is equivalent to fixed biases. A fixed bias of 1.0 to the GP units puts the output close to the desired full saturation when not inhibited by the *striatum*, and one could argue that the system is really taught to always activate all GP units, but that by design, it would not be possible to override the $\alpha = 10$ scaled inhibitions to the selected GP unit.

Berns and Sejnowski use a *leaky integrator*, as in CTRNN architectures. However, their version is slightly different than the usage normally reported in CTRNN literature. Experiments show that by setting the neuron potential to the firing rate, as in the update rule of the original work, the neuron behaves as if it is tonically firing, achieving an output of almost full saturation even without any inputs.

It has been shown in literature that inhibiting neurons in basal ganglia *are* tonically active, however this would apply to the STR and GP units, and not the excitatory STN units. Berns and Sejnowski mention that *globus pallidus internal* (GPi) is tonically active, but this is not reflected in their implementation unless considering the positive noise.

Neurons implementing the sigmoidal update rule, applied in the original work to the STN units, are hardly affected by inputs, except by applying large negative signals, such as with the -10 signal from STR to GP . This gives a reason for why Berns and Sejnowski have applied this relative strong inhibiting scaling factor, to compensate for their sigmoidal leaky integrators' small response.

Experiments also showed that using a sigmoidal leaky integrator will work as a dampener on the effect of the time constant. This means that the eligibility traces of the "slow" STN units are reduced to only a few time steps. However, for learning a very short sequence, it is reasonable that the STN units should not have short-term memory lasting much longer, because otherwise it would overlap with the looping of the sequence. A simpler way to achieve this with normal continuous-time recurrent neural network (CTRNN) neurons would be to use smaller time constants.

A reduced time constant could explain the lack of reproducable playback, as reported in section 4.1.5 on page 39.

5.2.3 Issues with reproducing experiments

It was concluded that prototyping scientific experiments in a strongly typed and low-level language as C++ was inefficient. In a prototyping context, the importance of designing proper software interfaces is fairly low compared to the ease of changing implementations, logging and debugging.

In comparison with C++, the programming language Python proved to be a far more productive choice, giving freedom to experiment with the prototypes and moving focus to the model to be implemented instead of the interfaces to design. However, it might still be feasible to do a back-port from Python to C or C++ for a high-performance version of the most critical parts of the model. This is in fact a technique often used by Python developers. By using the matrix operational library *annie*, most heavy calculations will already be optimised.

It was realised that even though a paper provides equations for all parts of their model, the equations might not be correct. In Berns and Sejnowski (1998), this is most likely due to faults introduced in “back-porting” from the programming language version of the model (that produced the results) to equations to be included in the paper.

When again porting from those equations to code in attempts to reproduce the original work, new errors can be introduced, and faults in the equations themselves are revealed. Although it is possible, to a great extent, to analyze the different errors, as in this work, this does not guarantee that the original results can be reproduced.

It is the view of this author that the computer science community has an tendency for preferring mathematics in papers even when a source code extract could provide a better explanation, be more understandable and known to be correct. The lack of included source code leads to incomplete publications that can be hard or impossible to reproduce. This problem is, to a great extent, more present in computer science than in other sciences.

For instance in biomedical research, it is much more common to include technical details such as apparatuses used and amounts of different chemicals applied. These details are invaluable when reproducing results.

In computer science, the norm is to transcribe everything as abstract mathematics, and never include details such as constants or ‘hacks’ that were invaluable in making the experiments work. This is probably partly a legacy from computer science breaking out of the mathematical sciences, and is described as ‘generalising’. It is this author’s belief that in many computer science papers, the general mathematics is often derived from the experimental source code, but the reverse process is seldom tested, and so there is

no guarantee that the 'generalised' equations even covers the source used in the reported experiments.

5.3 Future work

Several issues could be a topic for future work. Generally, the usage of non-tested equations in computer science literature should be analysed and addressed.

Basal ganglia models are diverse and don't seem to have a common goal. The problem is to approach the neuroscience literature, but computer scientists also seem to want to include all techniques learned from machine learning. Although this inspiration has proven fruitful and has given valuable insight, it is clear that at some point one should let go of the computer science and make the model approximate real data.

In the basal ganglia literature, there are even suggested models that have never been evaluated. Researchers in computer science should take reproducibility as serious as other sciences, and this requires published work to include the neccessary technical details.

For the models investigated in this work, one could research futher if the peculiarities of Berns and Sejnowski (1998) unintentionally did provide some insight into basal ganglia behaviour or sequence learning. As shown in this work, methods described and results reported in the original work don't match up, so further work could try to solve the ambiguities.

The model of Prescott et al. (2006) seems promising, and should be a subject for futher research. Implementing learning and the dopamine system in this model could yield great results.

Appendix A

Source code

This appendix includes source code used in this work. This code is also available on request[1], and is released under the GPL license[2].

The source code included is:

Berns and Sejnowski in Python The direct implementation of Berns and Sejnowski (1998) producing most of the results discussed in 4.1 on page 33

CTRNN library for Python A general library for continuous-time recurrent neural network (CTRNN) developed

Berns and Sejnowski using CTRNN Applying the CTRNN library for implementing Berns and Sejnowski (1998)

Prescott et al. (2006) using CTRNN Applying the CTRNN library for implementing Prescott et al. (2006)

This source code is not included:

- The early C++ model, as described in 3.5.1 on page 28, as it is incomplete and did not produce any valuable results.
- Variations of Berns and Sejnowski implementation for testing noise, error functions, etc
- Octave simulations producing figures 2.9 on page 15 and 4.10 on page 46
- Octave source for producing the other result graphs

[1] http://soiland.no/master
[2] http://www.gnu.org/copyleft/gpl.html

A.1 Berns and Sejnowski in Python

This program directly implements the equations of Berns and Sejnowski (1998) and tests the basal ganglia model on the sequence $0, 1, 2, 3, 1, 4$. The main flow of the program is:

- Train the network by looping over the sequence for a total of 200 time steps:
  - Advance the sequence for each time step, and activate only the striatum inputs corresponding to the current sequence number
  - Calculate one time step, layer by layer:
    1. *subthalamic nucleus* (STN)
    2. *globus pallidus* (GP)
    3. *substantia nigra pars compacta* (SNc) (error)
    4. Modify weights from STN to GP
    5. Modify weights from striatum to SNc
    6. Log all weights and neuron outputs
  - Pick the lowest outputting *globus pallidus* (GP) unit as the model's guess and log
- Reset the network potentials
- Hint the network by stepping through the first item of the sequence to be played back.
- Step through the network for 20 trial steps with no inputs

```
#!/usr/bin/env python
# *-* encoding: utf8
#
# Copyright (c) 2005-2006 Stian Soiland
#
# Permission is hereby granted, free of charge, to any person obtaining
# a copy of this software and associated documentation files (the
# "Software"), to deal in the Software without restriction, including
# without limitation the rights to use, copy, modify, merge, publish,
# distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to
# the following conditions:
#
```

```
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
# CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
# SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
#
# Author: Stian Soiland <stian@soiland.no>
# URL: http://soiland.no/i/src/
# License: MIT
#
"""Basal ganglia model as of Berns & Sejnowski 1998.

This model (re)implements the simulations as described by [1] by almost
directly mapping the equations to functions.

[1] Berns, G & Sejnowski, T - A computational model of how the basal
ganglia produce sequences - Journal of cognitive neuroscience 10:1 pp.
108-121
"""

import sys
import math
import random
import itertools

# Write stuff to screen
DEBUG=2
# Log stuff to *txt
LOG=1


def constrain(x, min, max):
    """Ensure x is within limits"""
    if x > max: return max
    elif x < min: return min
    return x

def _print_nums(file, nums):
```

```
    """Print list of numbers space-seperated to a file object"""
    nums = map(str, nums)
    print >>file, " ".join(nums)

class Berns:
    def __init__(self):
        # "the sequence (1,2,3,4,2,5)"
        self.seq = [0,1,2,3,1,4]
        # "was learned using 5 units"
        self.inputs = max(self.seq)+1
        # As suggested by figure 4, learning is achieved by 200
        # timesteps
        self.trainsteps = 200
        # As suggested by figure 7, only hint 1 timestep
        self.hintsteps = 1
        # timesteps for trials after hint
        self.trialsteps = 20
        # From p119 (methods):
        # "All weights were constrained to the range 0 to 1"
        self.min_v = 0.0
        self.min_w = 0.0
        self.max_v = 1.0
        self.max_w = 1.0
        # lambdas, converted STN timeconsts as by eq 3
        self.long = 0.9 # 90 msec
        self.short = 0.4 # 7 msec
        # As by table 2
        self.gain = 4.0
        self.bias = 0.1
        # alpha: "relative effect of an inhibitory synapse to an
        # exitatory one"
        self.effect = 10
        # learning rates
        self.w_learning = 0.05
        self.v_learning = 0.01
        # "magnitude of random synaptic noise"
        self.noiselevel = 0.5
        # /table2
        # "Beginning at a state in which the weights were all
        # zero" (and calc_v_change() gives positive changes, so v
        # should grow)
        self.v = [0.0 for x in range(self.inputs)]
        self.w = {}
```

```
        for i in range(self.inputs):
            for j in range(self.inputs*2):
                # Initially 0, as suggested by figure 5, and by p118:
                # "Beginning at a state in which the weights were all
                # zero"
                self.w[i,j] = 0.0
        # Set initial outputs to 0 as by Figure 7 STN Short t=1
        self.reset()
        if LOG:
            # Each file has numbers seperated by space, one line for
            # each timestep
            # Neural outputs
            self.f_STR = open("STR.txt", "w")
            self.f_GP = open("GP.txt", "w")
            self.f_STN_s = open("STN_s.txt", "w")
            self.f_STN_l = open("STN_l.txt", "w")
            # Weights
            self.f_W = open("W.txt", "w")
            self.f_V = open("V.txt", "w")
            # trained and guessed value
            self.f_ACTION = open("ACTION.txt", "w")
            # Error
            self.f_ERROR = open("E.txt", "w")


    def reset(self):
        # Reset before testing.. current outputs are set to zero.
        self.STR = [0.0] * self.inputs
        self.GP = [0.0] * self.inputs
        self.STN = [0.0] * self.inputs*2
        self.error = 0.0


    # eq 2 subthalamic
    def calc_STN(self, i):
        # Determine time constant by odd or even index
        if i % 2:
            lambd = self.long # slow / odd index (n*2 + 1)
        else:
            lambd = self.short # fast / even index (n*2)
        # leaky integrator in discrete time
        # Corresponding GP unit is i/2.. so that
        # STN 0, STN 1 <-- GP 0
```

```
        # STN 2, STN 3 <-- GP 1
        # etc.
        # Both STN[i] and GP[i/2] will be from the previous timestep (ie
        # G(t-n) where n=1) because calc_STN() is called before
        # calc_GP(), and the results are not pushed back to STN before
        # after calc-ing.
        return self.sigmoid(lambd * self.STN[i] -
                    (1-lambd) * self.effect * self.GP[i/2])

    # eq 3 (not used)
    # time(0.007) -> 0.4 (7 msec)
    # time(0.09) -> 0.9 (90 msec)
    #def time(timeconst, length_timestep=0.01):
    #    # 10 ms timesteps
    #    return timeconst / (timeconst+length_timestep)
    # self.long == self.time(0.09) # 0.9 == 90 msec
    # self.short == self.time(0.007) # 0.4 == 7 msec

    # eq 4 transfer function
    def sigmoid(self, x):
        return 1.0 / (1.0 + math.exp(-self.gain * (x-self.bias)))

    # eq 5 globus pallidus
    def calc_GP(self, i):
        sum = 0.0
        for j in range(self.inputs*2):
            # Note: w[i,j] means from j to i!
            sum += self.w[i,j]*self.STN[j]
        # "Noise drawn from a uniform distribution"
        # (We'll assume this means a distribution centered around 0 with
        # width 0.5, thereby a uniformly drawn random number between
        # -0.25 and 0.25)
        noise = random.uniform(-self.noiselevel/2, self.noiselevel/2)
        result = sum - self.effect * self.STR[i] + noise
        return self.sigmoid(result)

    # eq 6
    def calc_w_change(self, i,j):
        """Calculate weight change for STN(j) to GP(i)"""
        r = self.w_learning * (self.error * self.GP[i] - self.STR[i]) * \
            self.STN[j]
        #print r,
        return r
```

```
    # eq 7 error / SNc
    def calc_error(self):
        """Calculate value for SNc, the error"""
        sum = 0.0
        for i in range(self.inputs):
            sum += self.GP[i]
            sum -= self.v[i]* self.STR[i]
        if LOG:
            print >>self.f_ERROR, sum,
        return sum

    # eq 8
    def calc_v_change(self, i):
        """Calculate weight change for STR(i) to SNc"""
        return self.v_learning * self.error * self.STR[i]

    def step(self, input=None):
        # 1. Set STR input vector (if provided)
        if input:
            assert len(input) == self.inputs
            self.STR = input
        # 2. Calculate STN (using STN and GP values from prev. timestep)
        temp_STN = []
        for j in range(self.inputs*2):
            temp_STN.append(self.calc_STN(j))
        self.STN = temp_STN
        # 3. Calculate GP (using current STN and STR values)
        for i in range(self.inputs):
            self.GP[i] = self.calc_GP(i)
        # 4. Calculate error (using current STR and GP values)
        self.error = self.calc_error()
        for i in range(self.inputs):
            # 5. Change weights STN(j)->GP(i)
            for j in range(self.inputs*2):
                # Note: w[i,j] means from j to i!
                self.w[i,j] += self.calc_w_change(i,j)
                self.w[i,j] = constrain(self.w[i,j], self.min_w, self.max_w)
            # 6. Change weights GP(i)->SNc(i)
            self.v[i] += self.calc_v_change(i)
            self.v[i] = constrain(self.v[i], self.min_v, self.max_v)
        # 7. Log all outputs and weights
        self.log()
```

```
def train(self):
    for x in range(self.trainsteps/len(self.seq)):
        for number in self.seq:
            # Make input vector
            input=[0.0]*self.inputs
            input[number] = 1.0
            self.step(input)
            # The network's guess is the lowest firing GP
            guess = self.GP.index(min(self.GP))
            if DEBUG>1:
                print number, guess, self.GP, "e=%s" % self.error
            if LOG:
                print >>self.f_ACTION, number, guess

def log(self):
    if not LOG:
        return
    _print_nums(self.f_STR, self.STR)
    _print_nums(self.f_GP, self.GP)
    _print_nums(self.f_STN_l, [elem for n,elem in enumerate(self.STN) if n%2])
    _print_nums(self.f_STN_s, [elem for n,elem in enumerate(self.STN) if not n
        %2])
    _print_nums(self.f_V, self.v)
    W = self.w.items()
    W.sort()
    _print_nums(self.f_W, [weight for _,weight in W])


def test(self):
    self.reset()
    seq = itertools.cycle(self.seq)
    for hint in itertools.islice(seq, self.hintsteps):
        input=[0.0]*self.inputs
        input[hint] = 1.0
        self.step(input)
        guess = self.GP.index(min(self.GP))
        if DEBUG:
            print hint, guess, self.GP, "e=%s" % self.error
        if LOG:
            print >>self.f_ACTION, hint, guess
    if DEBUG:
        print "--finished hints"
```

```
        # As suggested by Figure 7, STR is silent after hint
        self.STR = [0.0]*self.inputs
        correct = 0
        answers = []
        for answer in itertools.islice(seq, self.trialsteps):
            self.step()
            guess = self.GP.index(min(self.GP))
            answers.append(guess)
            if DEBUG:
                print answer, guess, self.GP, "e=%s" % self.error
            if LOG:
                print >>self.f_ACTION, answer, guess
            correct += guess==answer
        return correct, answers


if __name__ == "__main__":
    if sys.argv[1:]:
        random.seed(sys.argv[1])
    berns = Berns()
    berns.train()
    print "---- Testing ----"
    berns.test()
```

A.2 CTRNN library for Python

A library for constructing and evaluating continuous-time recurrent neural network (CTRNN) networks. Using the library *NumPy* (Oliphant), networks are evaluated using matrix calculations, as described in section 3.5.1 on page 30. The network is represented by the class `CTRNN` and is evaluated with the method `calc_timestep()`. The method `stabilize` runs the network until it is considered stable, that is until the outputs do not change considerably.

As CTRNN networks do not necessarily have the same configuration for every neuron, a wrapper class `Layers` is included that allows accessing slices of the network. This makes it easy to set time constants for parts of the network representing say the *subthalamic nucleus*.

```
#!/usr/bin/env python2.4
# *-* encoding: utf8
#
# Copyright (c) 2006 Stian Soiland
#
# Permission is hereby granted, free of charge, to any person obtaining
# a copy of this software and associated documentation files (the
# "Software"), to deal in the Software without restriction, including
# without limitation the rights to use, copy, modify, merge, publish,
# distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to
# the following conditions:
#
# The above copyright notice and this permission notice shall be
# included in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND,
# EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
# CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
# SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
#
# Author: Stian Soiland <stian@soiland.no>
# URL: http://soiland.no/i/src/
# License: MIT
#
"""Continuous time recurrent neural network.
```

```
"""


from math import exp, sqrt, pi, tanh
import operator
import logging

try:
    import numpy
except ImportError:
    # Try to go old-fashioned
    import Numeric as numpy


# Hackish, generate IEEE 754 special values
#inf = 1e300 * 1e300
#nan = inf - inf

def identity(x):
    """Identity transfer function"""
    return x

def step(x, limit=0.4):
    """Step transfer function"""
    if x < limit:
        return 0.0
    return 1.0

def sigmoid(x, gain=1.0, bias=0.0):
    """Sigmoidal transfer function."""
    try:
        return 1.0 / (1.0 + exp(-gain * (x-bias)))
    except OverflowError:
        return 0.0

def dsigmoid(x):
    """Derivative of sigmoid()"""
    f = sigmoid(x)
    return f * (1-f)

def gaussian(x, mu=0.0, sigma=1.0):
    """Gaussian transfer function"""
```

```
    return (1 / (sqrt(2*pi) * sigma)) * exp( -0.5 * ( (x-mu)/sigma ) ** 2 )

def dgaussian(x, *args, **kwargs):
    """Derivative of gaussian(), parameters as for gaussian()"""
    return -2 * gaussian(x, *args, **kwargs) * x

def signum(x):
    if x < 0.0:
        return -1.0
    return 1.0

# The same as math.tanh
#def tansig(x):
#    return ( 2/ (1+exp(-2*x)) ) -1

def dtansig(x):
    """Derivative of tanh()"""
    f = tanh(x)
    return 1-(f**2)

def tanh_pos(x):
    """tanh with lower limit of 0.0"""
    if x < 0.0:
        return 0.0
    return tanh(x)


class Layers(object):
    """Layers for constructing and accessing a CTRNN.

    A layer is a set of neurons. A network is made of several layers.
    Note that compared to feed-forward networks, there are no rules that
    neurons in a layer cannot be interconnected. In addition,
    timestep calculation is done globally, not per layer.

    These layers can be used to group neurons into different layers for
    easier access to their parameters. For instance, a network can
    consist of an input layer, a hidden layer, and an output layer,
    which can have different weights and time constants. Instead of
    remembering that the hidden neurons are in the range 15 to 26 and
    using these offsets in all code, the layer can be accessed as the
    attribute "hidden".
```

```
    Construction work by adding layers using add_layers(). When all the
    layers are ready, build_net() is called. After this, no more layers
    can be added, the complete CTRNN network is available as attribute
    .net, and the different layers as attributes by .their_name.

    Example:
        layers = Layers()
        layers.add_input_layer("input", 3)
        layers.add_layer("hidden", 5)
        layers.add_layer("output", 2)
        layers.build_net()
        assert len(layers.net.potential) == 3+5+2

        layers.input.set_inputs((10, -5, 15))
        layers.hidden.timeconst[3] = 5
        layers.hidden.weight[0, 0] = 13
    """

    def __init__(self):
        self.layers = []
        self.input_layers = set()
        self.neurons = 0
        self.net = None

    def add_input_layer(self, name, neurons):
        """Add the input layer with the given name.

        Like add_layer(), but input layers will have their transfer
        function set to identity(), and their timeconstant will be 1.
        """
        self.add_layer(name, neurons)
        self.input_layers.add(name)

    def add_layer(self, name, neurons):
        if self.net:
            raise "Cannot add layers after build_net()"
        if hasattr(self, name):
            raise AttributeError, "already exists: %s" % name
        # Placeholder until build_net
        setattr(self, name, None)
        min = self.neurons # Inclusive
        self.neurons += neurons
        max = min + neurons # exclusive
```

```
        self.layers.append((name, slice(min,max)))

    def build_net(self, timeconst=1.0):
        self.net = CTRNN(self.neurons, timeconst)
        for (name, slice) in self.layers:
            if name in self.input_layers:
                layer = _InputLayer(self.net, slice)
            else:
                layer = _Layer(self.net, slice)
            setattr(self, name, layer)


class _Layer(object):
    """Proxy access to network parameters for the given layer.

    Properties such as bias, output, potential and timeconst are sliced
    out to refer to the current Layer only. Note that due to the use of
    numpy.array, these slices are also assignable, and changes are
    reflected in the grand network.

    The property ''weight'' is the part of the weight matrix for
    connections going *to* this layer, from *all* neurons. As thus, the
    matrix is sized with n rows and m columns, where n is the number of
    neurons in the whole network, and m is the number of neurons in this
    layer.

    Examples:

        # Set the weight from network neuron number 14 (global indexes)
        # to neuron 2 in this layer.
        layer.weight[14,3] = 0.7

        # Get global index for usage in another layer.weight
        index = layer[2]

        # Check the output of the layer
        print layer.output

        # Set all the biases at once (assumed layer size 4)
        layer.bias[:] = [0,1,2,3]

    """
    def __init__(self, net, slice):
```

```
        self.net = net
        self.slice = slice

    def __len__(self):
        step = self.slice.step or 1
        return (self.slice.stop - self.slice.start) / step

    def __getitem__(self, item):
        step = self.slice.step or 1
        index = item * step
        if index < 0:
            index = self.slice.stop + index
        else:
            index = self.slice.start + index
        if index < self.slice.start:
            raise IndexError, item
        if index >= self.slice.stop:
            raise IndexError, item
        return index

    def calc_timestep(self):
        self.net.calc_timestep(self.slice)

    def _slicer(self, array):
        # Note: This will only work for assignments if array is of
        # numpy.arraytype, where slices work as pointers instead of
        # making copied arrays
        assert isinstance(array, numpy.arraytype)
        return array[self.slice]

    @property
    def bias(self):
        return self._slicer(self.net.bias)

    @property
    def output(self):
        return self._slicer(self.net.output)

    @property
    def potential(self):
        return self._slicer(self.net.potential)

    @property
```

```
    def timeconst(self):
        return self._slicer(self.net.timeconst)

    @property
    def transfer(self):
        return self._slicer(self.net.transfer)

    def set_transfer(self, transfer):
        """Set the transfer function for the whole layer.
        """
        self.transfer[:] = [transfer] * len(self)

    @property
    def weight(self):
        # We return the weights pointing TO this neuron,
        # ie. where second dimension is our slice
        return self.net.weight[:,self.slice]

class _InputLayer(_Layer):
    def __init__(self, net, slice):
        super(_InputLayer, self).__init__(net, slice)
        self.fix()

    # FIXME: Separate the input layer from the "real" network
    def fix(self):
        """Make sure there is no monkey business going on"""
        self.set_transfer(identity)
        self.timeconst[:] = [1.0] * len(self)
        self.weight[:] = [0.0] * len(self.weight)

    def set_inputs(self, inputs):
        self.bias[:] = inputs
        self.calc_timestep()
        # If this fails, you have messed up the weights, timeconstants
        # or transfers of this input layer.
        assert (self.output == inputs).all()


class CTRNN(object):
    """Continuous time recurrent neural network.
    """

    def __init__(self, neurons, timeconst=1.0, transfer=sigmoid,
```

```
                timestep=1.0):
        """Construct a new CTRNN of the given number of neurons.

        Optional parameter timeconst is the time constant for neurons,
        by default 1.0. Individual time constants can later be modified
        through self.timeconst.

        Optional parameter transfer is the default transfer function.
        Transfer functions can be set individually for each neuron by
        the list self.transfer.

        Optional parameter timestep is the size of the timestep
        (delta T), by default 1. Note that for the network to be stable,
        the timestep size must be less than double the smallest
        timeconstant in the network, ie. if the smallest timeconstant is
        2, the timestep must be less than 4.
        """
        # Prepare logging
        self.log = logging.getLogger("ctrnn")
        if not self.log.handlers or logging.root.handlers:
            logging.basicConfig()

        self.num_neurons = neurons

        # NOTE: All arrays are numpy.array-s - which enables
        # by-reference slicing

        # internal state, membrane potential
        self.potential = numpy.zeros(neurons, numpy.Float)
        # Our timestep \Delta t must be smaller than twice the smallest
        # timeconst
        self.timestep = timestep
        assert 2*timeconst > timestep
        # By default, no bias
        self.bias = numpy.zeros(neurons, numpy.Float)
        self.timeconst = numpy.array([float(timeconst)] * neurons)
        self.weight = numpy.array(numpy.zeros((neurons, neurons)),
                                  numpy.Float)
        self.transfer = numpy.array([transfer] * neurons)
        # Output values as calulated by calc_timestep()
        self.output = numpy.zeros(neurons, numpy.Float)

    def connect_all(self, weight=None, func=None, ref_self=False):
```

```
        """Connect all neurons using the specified weight or function.

        Either weight or func must be specified, but not both.

        If weight is specified, it is the scalar weight assigned to all
        connections.

        If func is specified, it is assumed to be a function taking no
        arguments. The function will be called for each connection,
        and the result is assigned to the connection. No order can be
        assumed, so this function is normally something like
        random.random.

        If ref_self is set to True, self-looping weights will also be
        assigned between neuron n and neuron n, otherwise they will be
        assigned 0.0.
        """
        assert ((weight is None and func) or
                (weight is not None and not func))
        for n in range(self.num_neurons):
            for m in range(self.num_neurons):
                if n == m and not ref_self:
                    w = 0.0
                elif func:
                    w = func()
                else:
                    w = weight
                self.weight[n,m] = w

    def set_transfer(self, transfer):
        """Set the transfer function for the whole network.
        """
        self.transfer[:] = [transfer] * len(self.transfer)

    def calc_timestep(self, slicing=None):
        """Calculate the next timestep.

        If slicing is given, it is assumed a slice object from which
        neurons are to be updated. Otherwise, all neurons are updated.
        """
        # We do this as nice matrix operations
        if not slicing:
            # ie. [:] - everything FIXME: Undocumented in Python
```

```
            slicing = slice(None)
        inputs = numpy.matrixmultiply(self.output,
                                      self.weight[:,slicing])
        change = self.timestep/self.timeconst[slicing] * (-self.potential[slicing]
            +
                                              inputs + self.bias[slicing])
        self.potential[slicing] += change
        self.output[slicing] = [f(x) for f,x in
                zip(self.transfer[slicing], self.potential[slicing])]

    def stabilize(self, max_steps=200, precision=None):
        """Run the network until it stabilizes.

        A network is considered stable if the output does not change
        from one timestep to the next. (Note that depending on the
        transfer function, potentials can still change by this
        definition)

        If parameter max_steps is supplied, it is the maximum number of
        timesteps to run, by default 200.

        If parameter precision is supplied, it specifies the maximum
        difference between the highest and lowest output for the network
        to be considered stabilized. By default, full stabilizing
        would apply, requiring IEE754 double precision substraction to
        return 0.0

        Return the number of time steps used to stabilize, or None
        if the network did not stabilize within the maxiumum steps.
        """

        for x in xrange(max_steps):
            # Make a copy that is not changed by calc_timestep
            prev = list(self.output)
            self.calc_timestep()
            diff = numpy.subtract(prev, self.output)
            if precision is None:
                # All must be 0
                if not numpy.sometrue(diff):
                    self.log.info("Stabilized in %s timesteps", x)
                    return x
            else:
                # The maximal difference must be less than precision
```

```
            span = numpy.absolute(diff).max()
            if span <= precision:
                return x


        return None
```

A.2.1 Tests

Unit tests for the CTRNN library confirms basic functionality.

```
#!/usr/bin/env python2.4
# *-* encoding: utf8
#
# Copyright (c) 2006 Stian Soiland
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the
# "Software"), to deal in the Software without restriction, including
# without limitation the rights to use, copy, modify, merge, publish,
# distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to
# the following conditions:
#
# The above copyright notice and this permission notice shall be included
# in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
# OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
# CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
# SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
#
# Author: Stian Soiland <stian@soiland.no>
# URL: http://soiland.no/i/src/
# License: MIT
#
"""Tests for ctrnn.py
"""

import unittest
```

```
import random

import ctrnn
import numpy

class TestTransfers(unittest.TestCase):
    def testIdentity(self):
        self.assertEqual(ctrnn.identity(1.5), 1.5)
        self.assertEqual(ctrnn.identity(-0.9), -0.9)
        self.assertEqual(ctrnn.identity(0.0), 0.0)

    def testStep(self):
        self.assertEqual(ctrnn.step(-0.3), 0.0)
        self.assertEqual(ctrnn.step(0.2), 0.0)
        self.assertEqual(ctrnn.step(0.4), 1.0)
        self.assertEqual(ctrnn.step(0.8), 1.0)
        self.assertEqual(ctrnn.step(1.8), 1.0)

    def testSigmoid(self):
        self.assertEqual(ctrnn.sigmoid(1000), 1.0)
        self.assertEqual(ctrnn.sigmoid(-1000), 0.0) # Not OverflowError
        self.assertEqual(ctrnn.sigmoid(0), 0.5)
        self.assertEqual(ctrnn.sigmoid(0, gain=4), 0.5)
        # 0.401312339887548
        assert 0.401 < ctrnn.sigmoid(0, gain=4, bias=0.1) < 0.402
        # 2.6503965530043108e-261
        assert 0.0 < ctrnn.sigmoid(-600) < 2.66e-261

    def testSignum(self):
        self.assertEqual(ctrnn.signum(-0.3), -1.0)
        self.assertEqual(ctrnn.signum(-4.3), -1.0)
        self.assertEqual(ctrnn.signum(0.3), 1.0)
        self.assertEqual(ctrnn.signum(2.3), 1.0)
        self.assertEqual(ctrnn.signum(0.0), 1.0)

    def testTanhPos(self):
        assert 0.995 < ctrnn.tanh_pos(3) < 0.996
        assert 0.761 < ctrnn.tanh_pos(1) < 0.762
        assert 0.0099 < ctrnn.tanh_pos(0.01) < 0.01
        self.assertEqual(ctrnn.tanh_pos(0), 0)
        self.assertEqual(ctrnn.tanh_pos(-0.5), 0)
```

```
class TestCTRNN(unittest.TestCase):
    def testConstruction(self):
        neurons = 5
        net = ctrnn.CTRNN(neurons)
        self.assertEqual(net.num_neurons, neurons)
        self.assertEqual(len(net.potential), neurons)
        self.assertEqual(len(net.bias), neurons)
        self.assertEqual(len(net.timeconst), neurons)
        self.assertEqual(len(net.transfer), neurons)
        self.assertEqual(len(net.output), neurons)
        self.assertEqual(net.weight.shape, (neurons, neurons))

        self.assertEqual(list(net.potential), [0.0]*neurons)
        self.assertEqual(list(net.output), [0.0]*neurons)
        self.assertEqual(list(net.bias), [0.0]*neurons)
        for row in net.weight:
            self.assertEqual(list(row), [0.0]*neurons)

    def testConstructionDefault(self):
        neurons = 5
        net = ctrnn.CTRNN(neurons)
        # Default timeconst should be 1.0
        self.assertEqual(list(net.timeconst), [1.0]*neurons)
        # Default transfer should be sigmoid()
        self.assertEqual(list(net.transfer), [ctrnn.sigmoid]*neurons)

    def testConstructionParameters(self):
        neurons = 5
        timeconst = 1.5
        transfer = ctrnn.step
        net = ctrnn.CTRNN(neurons, timeconst, transfer=transfer)
        self.assertEqual(list(net.timeconst), [timeconst]*neurons)
        self.assertEqual(list(net.transfer), [transfer]*neurons)

    def testConstructionTooLowTimeconst(self):
        neurons = 5
        invalid_timeconst = 0.499
        valid_timeconst = 0.501
        self.assertRaises(AssertionError, ctrnn.CTRNN, neurons,
                          invalid_timeconst)
        ctrnn.CTRNN(neurons, valid_timeconst)

    def testConnectAll(self):
```

```
        neurons = 3
        net = ctrnn.CTRNN(neurons)
        # Should not be allowed to call with too few or too many
        # parameters
        self.assertRaises(AssertionError, net.connect_all)
        self.assertRaises(AssertionError, net.connect_all, 1.5, random.random)

        weight = 0.5
        # All rows should be equal to weight if ref_self
        net.connect_all(weight, ref_self=True)
        for row in net.weight:
            self.assertEqual(list(row), [weight]*neurons)

        # All except the x==y cells should now be equal
        net.connect_all(weight)
        for x,row in enumerate(net.weight):
            for y,w in enumerate(row):
                if x == y:
                    self.assertEqual(w, 0.0)
                else:
                    self.assertEqual(w, weight)

        # And test by random()
        net.connect_all(func=random.random, ref_self=True)
        unique = set()
        for row in net.weight:
            for w in row:
                # All should be unique
                assert w not in unique
                unique.add(w)

    def testSetTransfer(self):
        neurons = 5
        net = ctrnn.CTRNN(neurons)
        # Default is sigmoid
        matches = net.transfer == [ctrnn.sigmoid]*neurons
        self.assert_(matches.all())

        net.set_transfer(ctrnn.identity)
        matches = net.transfer == [ctrnn.identity]*neurons
        self.assert_(matches.all())

class TestCalcTimestep(unittest.TestCase):
```

```
    def setUp(self):
        self.neurons = 3
        self.net = ctrnn.CTRNN(self.neurons)

    def testNeutral(self):
        self.assertEqual(list(self.net.output), [0.0]*self.neurons)
        self.net.calc_timestep()
        self.assertEqual(list(self.net.output), [0.5]*self.neurons)
        self.assertEqual(list(self.net.potential), [0.0]*self.neurons)

    def testBias(self):
        self.net.bias[0] = 1.0
        self.net.calc_timestep()
        self.assertEqual(list(self.net.potential), [1.0, 0.0, 0.0])
        #0.7310585786300049
        output0 = self.net.output[0]
        assert 0.730 < output0 < 0.732
        self.assertEqual(self.net.output[1], 0.5)
        self.assertEqual(self.net.output[2], 0.5)
        self.net.calc_timestep()
        # Should not change (timeconst=1)
        self.assertEqual(output0, self.net.output[0])

    def testTimeconst(self):
        self.net.bias[0] = 1.0
        self.net.calc_timestep()
        output0 = self.net.output[0]
        self.net.timeconst[0] = 1.5
        self.net.calc_timestep()
        # Should not change as we have reached the bias
        self.assertEqual(output0, self.net.output[0])
        self.net.bias[0] = 0.0
        # should now drop gradually towards 0.5
        self.net.calc_timestep()
        # 0.58257020646231472
        assert 0.582 < self.net.output[0] < 0.583
        self.net.calc_timestep()
        # 0.527749235055
        assert 0.527 < self.net.output[0] < 0.528

    def testWeights(self):
        self.net.bias[0] = 1.0
        # from 0 to 1
```

```
        self.net.weight[0,1] = 1.0
        self.net.calc_timestep()
        # Does not reach it on this timestep
        self.assertEqual(self.net.potential[1], 0.0)
        self.assertEqual(self.net.output[1], 0.5)
        self.net.calc_timestep()
        # Should be the only input, and therefore set the potential
        self.assertEqual(self.net.potential[1], self.net.output[0])
        # 0.67503752737682365
        assert 0.675 < self.net.output[1] < 0.676

    def testSlice(self):
        self.net.bias[0] = 1.0
        self.net.bias[2] = 1.0
        self.net.calc_timestep(slice(0,1)) # ie. only 0
        # Updated
        self.assertEqual(self.net.potential[0], 1.0)
        # 0.7310585786300049
        assert 0.730 < self.net.output[0] < 0.732
        # Unchanged
        self.assertEqual(self.net.potential[1], 0.0)
        self.assertEqual(self.net.potential[2], 0.0)
        self.assertEqual(self.net.output[1], 0.0)
        self.assertEqual(self.net.output[2], 0.0)

        # Update it all
        self.net.calc_timestep()
        self.assertEqual(self.net.output[1], 0.5)
        assert 0.730 < self.net.output[0] < 0.732
        self.net.weight[1,0] = 1.0
        self.net.calc_timestep(slice(0,1)) # ie. only 0
        # Now, even though we only updated neuron 0, he should
        # include the input from the previous calculation of 1.
        #0.817574476194
        assert 0.817 < self.net.output[0] < 0.818

class TestStabilize(unittest.TestCase):
    def setUp(self):
        self.neurons = 3
        self.net = ctrnn.CTRNN(self.neurons)

    def testWeights(self):
        # Should be stable in 1 step with no connections
```

```
        self.assertEqual(self.net.stabilize(), 1)
        self.net.weight[0,1] = 1.0
        self.net.weight[1,2] = 1.0
        # Should be stable in 2 steps to propagate
        # 0->1 and 1->2
        self.assertEqual(self.net.stabilize(), 2)
        # Let's introduce some fun
        self.net.weight[2,0] = 1.0
        steps = self.net.stabilize()
        # Should be about 24
        assert 10 < steps < 100
        # And now, let's also check that it IS stable
        output = list(self.net.output)
        self.net.calc_timestep()
        output_2 = list(self.net.output)
        self.assertEqual(output, output_2)
        # And actually, all values should be the same since our weights
        # are the same
        self.assertEqual(output[0], output[1])
        self.assertEqual(output[1], output[2])

    def testTimeConstant(self):
        self.net.bias[0] = 1.0
        self.net.timeconst[0] = 1.5
        steps = self.net.stabilize()
        # Should be about 33
        assert 10 < steps < 100
        # Almost almost almost 1.0
        assert 0.99999 < self.net.potential[0] < 1.00001

    def testMaxSteps(self):
        self.net.bias[0] = 1.0
        # NOTE: Illegal timeconstant < 0.5 -> unstable network
        self.net.timeconst[0] = 0.4
        steps = self.net.stabilize()
        self.assertEqual(steps, None)

        self.net.timeconst[0] = 0.95 # stable again, in about 40 steps
        steps = self.net.stabilize(max_steps=5) # but 5 is not enough
        self.assertEqual(steps, None)
        steps = self.net.stabilize()
        # about 35
        assert 2 < steps < 100
```

```
    def testPrecision(self):
        precision = 0.01
        self.net.bias[0] = 1.0
        self.net.timeconst[0] = 0.6
        #steps = self.net.stabilize(precision=precision)
        steps = self.net.stabilize(precision=precision)
        # Would take about 88 steps with precision=None
        assert 7 < steps < 11

        # And check that we actually are within precision
        prev = self.net.output
        self.net.calc_timestep()
        now = self.net.output
        diff = now - prev
        assert numpy.absolute(diff).max() < precision



class TestLayers(unittest.TestCase):
    def testAdd(self):
        layers = ctrnn.Layers()
        hid_neurons = 5
        out_neurons = 2
        layers.add_layer("hidden", hid_neurons)
        layers.add_layer("output", out_neurons)
        self.assertEqual(layers.neurons, hid_neurons+out_neurons)

    def testAddInput(self):
        layers = ctrnn.Layers()
        in_neurons = 4
        layers.add_input_layer("input", in_neurons)
        self.assertEqual(layers.neurons, in_neurons)
        assert "input" in layers.input_layers

    def testBuild(self):
        hid_neurons = 5
        out_neurons = 2
        in_neurons = 4
        neurons = hid_neurons+out_neurons+in_neurons
        layers = ctrnn.Layers()
        layers.add_input_layer("input", in_neurons)
        layers.add_layer("hidden", hid_neurons)
```

```
        layers.add_layer("output", out_neurons)
        # Name already in use
        self.assertRaises(Exception, layers.add_layer, "hidden",
                          hid_neurons)
        self.assertEqual(layers.neurons, neurons)
        layers.build_net()
        self.assertEqual(layers.net.num_neurons, neurons)
        assert isinstance(layers.input, ctrnn._Layer)
        assert isinstance(layers.hidden, ctrnn._Layer)
        assert isinstance(layers.output, ctrnn._Layer)
        assert isinstance(layers.input, ctrnn._InputLayer)
        assert not isinstance(layers.hidden, ctrnn._InputLayer)
        assert not isinstance(layers.output, ctrnn._InputLayer)

        # Make sure they are separate
        layers.input.bias[:] = [0,1,2,3]
        layers.hidden.bias[:] = [4,5,6,7,8]
        layers.output.bias[:] = [9,10]
        self.assertEqual(list(layers.input.bias), [0,1,2,3])
        self.assertEqual(list(layers.hidden.bias), [4,5,6,7,8])
        self.assertEqual(list(layers.output.bias), [9,10])
        # And that they are added in order, and that changes to layers
        # are reflected back in the actual net
        self.assertEqual(list(layers.net.bias), range(11))

class TestLayer(unittest.TestCase):
    def setUp(self):
        self.in_neurons = 4
        self.hid_neurons = 5
        self.out_neurons = 2
        self.layers = ctrnn.Layers()
        self.layers.add_input_layer("input", self.in_neurons)
        self.layers.add_layer("hidden", self.hid_neurons)
        self.layers.add_layer("output", self.out_neurons)
        self.layers.build_net()

    def testLength(self):
        hidden = self.layers.hidden
        self.assertEqual(len(hidden), self.hid_neurons)

    def testGetItem(self):
        hidden = self.layers.hidden
        self.assertEqual(hidden[0], self.in_neurons)
```

```
        self.assertEqual(hidden[-1], self.in_neurons+self.hid_neurons-1)
        # Check that boundaries are enforced (This would be valid
        # indexes because they would cross into "input" or "output", but
        # are not supposed to be returned from the "hidden" layer.
        self.assertRaises(IndexError, lambda: hidden[self.hid_neurons])
        self.assertRaises(IndexError, lambda: hidden[-self.hid_neurons-1])
        self.assertEqual(self.layers.input[-1]+1, hidden[0])
        self.assertEqual(hidden[-1], self.layers.output[0] - 1)

    def testProperties(self):
        # Set weird values before and after hidden
        self.layers.input.bias[:] = [4.0]*self.in_neurons
        self.layers.output.bias[:] = [7.0]*self.out_neurons
        hidden = self.layers.hidden

        self.assertEqual(list(hidden.bias), [0.0]*self.hid_neurons)
        self.assertEqual(list(hidden.potential), [0.0]*self.hid_neurons)
        self.assertEqual(list(hidden.output), [0.0]*self.hid_neurons)
        self.assertEqual(list(hidden.timeconst), [1.0]*self.hid_neurons)
        self.assertEqual(list(hidden.transfer), [ctrnn.sigmoid]*self.hid_neurons)
    def testSetTransfer(self):
        hidden = self.layers.hidden
        neurons = len(hidden)
        # Default is sigmoid
        matches = hidden.transfer == [ctrnn.sigmoid]*neurons
        self.assert_(matches.all())

        hidden.set_transfer(ctrnn.signum)
        matches = hidden.transfer == [ctrnn.signum]*neurons
        self.assert_(matches.all())

        # And that the other are untouched
        # (Note how the input layer has identity by default)
        self.assertEqual(list(self.layers.input.transfer),
                         [ctrnn.identity]*self.in_neurons)
        self.assertEqual(list(self.layers.output.transfer),
                         [ctrnn.sigmoid]*self.out_neurons)

    def testTimestep(self):
        hidden = self.layers.hidden
        hidden.calc_timestep()
        self.assertEqual(list(hidden.output), [0.5]*self.hid_neurons)
        self.assertEqual(list(hidden.potential), [0.0]*self.hid_neurons)
```

```
        hidden.bias[:] = [1.0] * self.hid_neurons
        hidden.calc_timestep()
        self.assertEqual(list(hidden.potential), [1.0]*self.hid_neurons)

        # Make sure nothing else changed
        self.assertEqual(list(self.layers.input.bias), [0.0]*self.in_neurons)
        self.assertEqual(list(self.layers.input.potential), [0.0]*self.in_neurons)
        self.assertEqual(list(self.layers.input.output), [0.0]*self.in_neurons)
        self.assertEqual(list(self.layers.output.bias), [0.0]*self.out_neurons)
        self.assertEqual(list(self.layers.output.potential), [0.0]*self.out_neurons
            )
        self.assertEqual(list(self.layers.output.output), [0.0]*self.out_neurons)

    def testInput(self):
        input = self.layers.input
        values = [-0.5, -0.1, 0.1, 0.5]
        input.set_inputs(values)
        self.assertEqual(list(input.bias), values)
        self.assertEqual(list(input.potential), values)
        self.assertEqual(list(input.output), values)
        self.assertEqual(list(input.transfer),
                         [ctrnn.identity]*self.in_neurons)



if __name__ == "__main__":
    unittest.main()
```

A.3 Berns and Sejnowski using CTRNN

Reimplementation of the Berns and Sejnowski (1998) model using the CTRNN library described in section A.2 on page 68. Compared to the implementation A.1 on page 60, this implementation does not use the sigmoidal update rule, but applies leaky integrators as used in the CTRNN literature.

```
#!/usr/bin/env python2.4
# *-* encoding: utf8
#
# Copyright (c) 2006 Stian Soiland
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the
# "Software"), to deal in the Software without restriction, including
# without limitation the rights to use, copy, modify, merge, publish,
# distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to
# the following conditions:
#
# The above copyright notice and this permission notice shall be included
# in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
# OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
# CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
# SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
#
# Author: Stian Soiland <stian@soiland.no>
# URL: http://soiland.no/i/src/
# License: MIT
#
"""CTRNN basal ganglia, approaching Berns & Sejnowski 1998.
"""

import logging
import itertools
import ctrnn
```

```
def sigmoid_bias(x):
    """Sigmoid with gain and bias set according to Berns1998."""
    return ctrnn.sigmoid(x, gain=4, bias=0.0)


DEBUG=1
STATS=1


class Berns:
    def __init__(self):
        # Logger
        self.log = logging.getLogger("berns")
        if not self.log.handlers or logging.root.handlers:
            logging.basicConfig()
        # constants and initial weights/values
        self.seq = [0,1,2,3,1,4]
        self.inputs = max(self.seq)+1
        self.neurons = self.inputs * 4 # str, gp, stn_l, stn_s
        self.trainsteps = 300 # timesteps to train
        self.hintsteps = 1   # timesteps to hint in testing
        self.trialsteps = 30 # timesteps for trial
        self.long = 4 # 20 msec
        self.short = 0.7 # 7 msec
        self.effect = 10
        self.w_learning = 0.5
        self.v_learning = 0.1
        self.v = [0.0 for x in range(self.inputs)]
        # By default, timeconst=1 (no ctrnn) and no weights
        self.net = ctrnn.CTRNN(self.neurons)
        self.net.transfer[:] = [sigmoid_bias for x in range(self.inputs)]
        for n in range(self.inputs):
            # 1-1 mappings
            self.net.weight[self.str(n), self.gp(n)] = -100.0 # inhib
            self.net.weight[self.gp(n), self.stn_l(n)] = self.effect
            self.net.weight[self.gp(n), self.stn_s(n)] = self.effect
            for m in range(self.inputs):
                # And STNs connect to all GPs
                self.net.weight[self.stn_l(n), self.gp(m)] = 0.0
                self.net.weight[self.stn_s(n), self.gp(m)] = 0.0

            # Only STN has timeconsts != 1
            self.net.timeconst[self.stn_l(n)] = self.long
```

```
            self.net.timeconst[self.stn_s(n)] = self.short
            #self.net.bias[self.stn_l(n)] = -0.9
        self.set_input()
        self.net.stabilize()


    def str(self, n):
        return n

    def gp(self, n):
        return n + self.inputs

    def stn_l(self, n):
        return n + self.inputs*2

    def stn_s(self, n):
        return n + self.inputs*3

    def set_input(self, input=-1):
        # input is 0 or 1
        for n in range(self.inputs):
            self.net.bias[self.str(n)] = (n == input) and 1000 or -1000
        # Update only those neurons
        self.net.calc_timestep(slice(self.str(0), self.str(self.inputs)))
        self.log.debug("set_input(%s) %s", input, [self.net.output[self.str(n)] for
             n in range(self.inputs)])

    def get_output(self):
        return [self.net.output[self.gp(n)] for n in range(self.inputs)]

    def winner(self):
        output = self.get_output()
        return output.index(min(output))

    def calc_error(self):
        sum = 0.0
        for n in range(self.inputs):
            sum += (1-self.net.output[self.gp(n)])
            sum -= self.v[n] * self.net.output[self.str(n)]
        self.error = sum

    def calc_v_change(self, i):
        return self.v_learning * self.error * self.net.output[self.str(i)]
```

```
def calc_w_change(self, i, stn):
    # Calculates the change between input i and STN unit stn
    r = (self.w_learning * (self.error * self.net.output[self.gp(i)] -
                         self.net.output[self.str(i)]) *
         self.net.output[stn])
    return r

def calc_STN(self):
    neurons = slice(self.stn_l(0), self.stn_l(self.inputs))
    self.net.calc_timestep(neurons)
    neurons = slice(self.stn_s(0), self.stn_s(self.inputs))
    self.net.calc_timestep(neurons)

def calc_GP(self):
    neurons = slice(self.gp(0), self.gp(self.inputs))
    self.net.calc_timestep(neurons)

def step(self, input=None):
    if input is not None:
        self.set_input(input)
    # Calculated from previous GP values
    self.calc_STN()
    # Calculated from new STNs, and new STRs
    self.calc_GP()
    # Compares new GPs with new STRs
    self.calc_error()

    # Update weights
    for n in range(self.inputs):
        gp = self.gp(n)
        for m in range(self.inputs):
            stn_l = self.stn_l(m)
            stn_s = self.stn_s(m)
            self.net.weight[stn_l, gp] += self.calc_w_change(n, stn_l)
            self.net.weight[stn_s, gp] += self.calc_w_change(n, stn_s)
        self.v[n] += self.calc_v_change(n)
        # FIXME: Add weight constraints
    self.log_stats()

def train(self):
    for x in range(self.trainsteps/len(self.seq)):
        for number in self.seq:
```

```
                self.step(number)
                guess = self.winner()
                self.log.info("Train h=%s g=%s %s", number, guess, self.get_output()
                    )

    def reset(self):
        self.net.potential[:] = [0] * self.neurons

    def log_stats(self):
        if not STATS:
            return
        if not hasattr(self, "_f"):
            self._f = dict()
            for out in ("str", "gp", "stn_l_p", "stn_l", "stn_s", "e", "f", "v", "
                w_stn_l", "w_stn_s"):
                self._f[out] = open("%s.txt" % out, "w")

        layers = ("str", "gp", "stn_l", "stn_s")
        for n in range(self.inputs):
            for layer in layers:
                mapper = getattr(self, layer)
                self._f[layer].write("%s " % self.net.output[mapper(n)])
            self._f["stn_l_p"].write("%s " % self.net.potential[self.stn_l(n)])
        self._f["stn_l_p"].write("\n")

        # finished all log-lines
        for layer in layers:
            self._f[layer].write("\n")

        print >>self._f["e"], self.error
        print >>self._f["v"], " ".join(map(str, self.v))

        # And the GP weights
        for n in range(self.inputs):
            gp = self.gp(n)
            for m in range(self.inputs):
                stn_s = self.stn_s(m)
                self._f["w_stn_s"].write("%s " % self.net.weight[stn_s][gp])
            for m in range(self.inputs):
                stn_l = self.stn_l(m)
                self._f["w_stn_l"].write("%s " % self.net.weight[stn_l][gp])
        self._f["w_stn_s"].write("\n")
        self._f["w_stn_l"].write("\n")
```

```
    def test(self):
        self.reset()
        seq = itertools.cycle(self.seq)
        for hint in itertools.islice(seq, self.hintsteps):
            self.step(hint)
            guess = self.winner()
            self.log.info("Hint h=%s g=%s %s e=%s", hint, guess,
                          self.get_output(), self.error)

        # Set no input
        self.set_input(-1)
        correct = 0
        answers = []
        for answer in itertools.islice(seq, self.trialsteps):
            self.step()
            guess = self.winner()
            answers.append(guess)
            self.log.info("Test a=%s g=%s %s e=%s", answer, guess,
                          self.get_output(), self.error)
            correct += guess==answer
        return correct, answers


if __name__ == "__main__":
    logging.getLogger().setLevel(logging.INFO)
    berns = Berns()
    berns.train()
    berns.log
    berns.test()
```

A.4 Prescott (2006) using CTRNN

Implementation of Prescott et al. (2006) using the CTRNN library. The network is run to stabilization for each sequence of the input, as described in the original work. The method `salience` tests the salience by gradually increasing the salient input $s2$ to compete with the salient input $s1$.

```
#!/usr/bin/env python2.4
# *-* encoding: utf8
#
# Copyright (c) 2006 Stian Soiland
#
# Permission is hereby granted, free of charge, to any person obtaining a
# copy of this software and associated documentation files (the
# "Software"), to deal in the Software without restriction, including
# without limitation the rights to use, copy, modify, merge, publish,
# distribute, sublicense, and/or sell copies of the Software, and to
# permit persons to whom the Software is furnished to do so, subject to
# the following conditions:
#
# The above copyright notice and this permission notice shall be included
# in all copies or substantial portions of the Software.
#
# THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS
# OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF
# MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT.
# IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY
# CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT,
# TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
# SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
#
# Author: Stian Soiland <stian@soiland.no>
# URL: http://soiland.no/i/src/
# License: MIT
#
"""CTRNN basal ganglia, as of Prescott 2006
"""

import logging
import itertools
import Numeric
import ctrnn
from isplit import isplit
```

```
from sets import Set
import operator
import sys



DEBUG=1
STATS=1

def piecewise(a, theta=0.0):
    """Piecewise lineaer transfer as by Prescott et al. 2006"""
    # NOTE: Assumes we really want bounded between 0..1 and fixes
    # a serious bug in the original equation
    return min(1.0, max(0.0, a-theta))

class Prescott(ctrnn.Layers):
    def __init__(self):
        # Logger
        self.log = logging.getLogger("prescott")
        if not self.log.handlers or logging.root.handlers:
            logging.basicConfig()
        # constants and initial weights/values
        self.seq = [0,1,2,3,1,4]
        self.inputs = max(self.seq)+1
        self.trainsteps = 20 # timesteps to train
        self.hintsteps = 1   # timesteps to hint in testing
        self.trialsteps = 0  # timesteps for trial
        self.striatum_delta = 0.2
        self.timeconst = 3.3333
        # "The model was considered to have converged
        #  whenever the smallest delta a on two consecutive
        #  timesteps was less than 0.0001"
        # (We'll assume Prescott meant the LARGEST delta)
        self.stable_limit = 0.0001
        self.w = {
            # Unless otherwise noted, connections are 1-1 by channel
            ('ssc', 'd1'): (1+self.striatum_delta)*0.5,
            ('ssc', 'd2'): (1-self.striatum_delta)*0.5,
            ('ssc', 'mc'): 1,
            ('ssc', 'stn'): 0.5,
            ('mc', 'd1'): (1+self.striatum_delta)*0.5,
            ('mc', 'd2'): (1-self.striatum_delta)*0.5,
            ('mc', 'vl'): 1,
            ('mc', 'trn'): 1,
```

```
        ('mc', 'stn'): 0.5,
        ('d1', 'snr'): -1,
        ('d2', 'gp'): -1,
        ('stn*', 'snr'): 0.9, # all-to-all
        ('stn*', 'gp'): 0.9, # all-to-all
        ('gp', 'stn'): -1,
        ('gp', 'snr'): -0.3,
        ('snr', 'trn'): -0.2,
        ('snr', 'vl'): -1,
        ('trn', 'vl'): -0.125,
        ('trn*', 'vl'): -0.4, # all-to-all
        ('vl', 'mc'): 1,
        ('vl', 'trn'): 1,
    }

    # Build layers
    ctrnn.Layers.__init__(self)
    layers = reduce(Set.union, map(Set, isplit(self.w)))
    for layer in layers:
        if "*" in layer: continue
        if layer == "ssc":
            self.add_input_layer(layer, self.inputs)
        else:
            self.add_layer(layer, self.inputs)
    self.build_net(timeconst=self.timeconst)
    self.net.set_transfer(piecewise)
    # Otherwise ssc will also have piecewise
    self.ssc.fix()
    for n in range(self.inputs):
        for ((src_l,dest_l), weight) in self.w.items():
            dest = getattr(self, dest_l)[n]
            if src_l[-1] == "*":
                # All of them are inputs
                for m in range(self.inputs):
                    src = getattr(self, src_l.rstrip("*"))[m]
                    self.net.weight[src, dest] = weight
            else:
                src = getattr(self, src_l)[n]
                self.net.weight[src, dest] = weight
        # Just make sure this weight is set after trn* -> vl
        self.vl.weight[self.trn[n], n] = self.w["trn", "vl"]
        # And the biases
        self.d1.bias[n] = -0.2
```

```
            self.d2.bias[n] = -0.2
            self.stn.bias[n] = 0.25
            self.gp.bias[n] = 0.2
            self.snr.bias[n] = 0.2
        self.set_input()
        self.net.stabilize()

    def set_input(self, input=-1):
        # input is 0 or 1, depending on the selected channel
        inputs = [n == input for n in range(self.inputs)]
        self.ssc.set_inputs(inputs)

    def get_output(self, timestep=False):
        if timestep:
            self.net.calc_timestep()
        return self.gp.output

    def winner(self):
        output = self.get_output()
        return list(output).index(min(output))


    def step(self, input=None):
        if input is not None:
            self.set_input(input)
        self.net.stabilize(precision=self.stable_limit)
        self.log_stats()

    def train(self):
        for x in range(self.trainsteps/len(self.seq)):
            for number in self.seq:
                self.step(number)
                for x in range(8):
                    self.step()
                guess = self.winner()
                self.log.info("Train h=%s g=%s %s", number, guess, self.get_output()
                    )

    def reset(self):
        self.net.potential[:] = [0.0] * self.neurons

    def test(self):
        self.reset()
```

```
        seq = itertools.cycle(self.seq)
        for hint in itertools.islice(seq, self.hintsteps):
            self.step(hint)
            for x in range(8):
                self.step()
            guess = self.winner()
            self.log.info("Hint h=%s g=%s %s", hint, guess,
                          self.get_output())

        # Set no input
        self.set_input(-1)
        correct = 0
        answers = []
        for answer in itertools.islice(seq, self.trialsteps):
            self.step()
            for x in range(8):
                self.step()
            guess = self.winner()
            answers.append(guess)
            self.log.info("Test a=%s g=%s %s", answer, guess,
                          self.get_output())
            correct += guess==answer
        return correct, answers

    def log_stats(self):
        if not STATS:
            return
        if not hasattr(self, "_f"):
            # Open files for writing
            self._f = dict()
            for out,_ in self.layers:
                # Outputs
                self._f[out] = open("%s.txt" % out, "w")
                # Weights
                out = "w_" + out
                self._f[out] = open("%s.txt" % out, "w")

        # Log the stats
        for name,_ in self.layers:
            # outputs
            out = self._f[name]
            layer = getattr(self, name)
            out.write(" ".join(map(str, layer.output)))
```

```
            out.write("\n")
            # weights
            out = self._f["w_" + name]
            for n in xrange(len(layer)):
                out.write(" ".join(map(str, layer.weight[:,n])))
                out.write(" ")
            out.write("\n")

        #print >>self._f["E"], self.error

    def salience(self):
        """Test salience space"""
        for s1 in xrange(100):
            s1 /= 100.0
            self.net.output[:] = [0.0]*len(self.net.output)
            for s2 in xrange(100):
                s2 /= 100.0
                inputs = [s1, s2, 0.0, 0.0, 0.0]
                self.ssc.set_inputs(inputs)
                self.step()
                sys.stdout.write(".")
                sys.stdout.flush()
            sys.stdout.write("\n")
            sys.stdout.flush()

if __name__ == "__main__":
    logging.getLogger().setLevel(logging.INFO)
    prescott = Prescott()
    #prescott.train()
    #prescott.test()
    prescott.salience()
```

Glossary

actor-critic

An architecture similar to *control systems*. The *actor* selects the action to perform on the *environment*, while the *critic* provides the actor with a *reinforcement signal* by judging the effect of the actions. 4

anterior

in the front, opposite of posterior 12

closed-loop

A system where its output will influence the system's input. Examples are *feedback controllers*, animals and robots, because they all interact with their environment and thereby change their own (sensory) inputs. 4

CS: conditioned stimulus

sensory cue that preceeds the reward (US) 6, 12, 13, 54

CTRNN: continuous-time recurrent neural network

artificial neural network using continuous time and recurrent connections, as opposed to discrete time feed-forward networks vii, viii, 3, 14–16, 23, 43, 46–49, 55, 59, 68, 78, 89

GP: *globus pallidus*

relays information from the striatum to the thalamus. 7, 12, 19, 21–27, 33–44, 51, 53–55, 60

GPe: *globus pallidus external*

part of the indirect pathway 7, 9, 10, 23

GPi: *globus pallidus internal*

part of the output zone of the basal ganglia, projects to the motor cortex through the thalamus 7, 9–11, 18, 23, 26, 55

in vivo

taking place in a living organism 21, 53, 54

inferior

lower, opposite of superior 12

interior

inside, opposite of exterior 12

lateral

on the side, opposite of medial 12

motor cortex

The region of the cerebral cortex concerned with motor behaviour 9

nuclei

plural of *nucleus*, collection of nerve cells in the brain that are anatomically discrete, and which typically serve a particular function. 7

open-loop

A system where the input is not influenced by the system output. Opposite of *closed-loop*. 4

premotor cortex

Motor association areas in the frontal lobe anterior to primary motor cortex, thought to be involved in planning or programming of voluntary movements 9

SN: *substantia nigra*

composed of *substantia nigra pars reticulata* and *substantia nigra pars compacta* 7, 9

SNc: *substantia nigra pars compacta*

a densely packed part of the substantia nigra 7, 21–23, 25, 54, 55, 60

SNr: *substantia nigra pars reticulata*

part of the output zone of the basal ganglia, projects mainly to eye movement controls. 7, 10, 18, 23

STN: *subthalamic nucleus*

part of the indirect pathway 7, 9, 10, 12, 21–26, 35–40, 43, 45, 51, 54, 55, 60

STR: *striatum*

the inputs to the basal ganglia 7, 9, 10, 19, 21, 22, 25, 26, 33, 35–41, 43, 44, 54, 55

superior

upper or outside, opposite of inferior 9

superior colliculus

part of the roof of the midbrain, plays an important role in orienting movements of the head and eyes 9

TD: temporal difference

a delayed reinforcement learning method based on the temporal differences in predictions 4–7, 16–18, 20

tonically active neuron

a neuron who always fires unless inhibited by dendrite signals. This can be compared to an artifical neuron having a positive bias. 11

US: unconditioned stimulus

follows the CS, usually the actual reward 6

VA/VL complex

The *ventral anterior* and *ventral lateral* nuclei of the *thalamus* that relay signals to the premotor and motor cortex. 9

ventral

on the underside, opposite of dorsal 12

Bibliography

ISO/IEC 14882:2003: Programming languages: C++. 2003. URL http://www.iso.org/iso/en/CatalogueDetailPage.CatalogueDetail?CSNUMBER=38110.

I. Bar-Gad, G. Morris, and H. Bergman. Information processing, dimensionality reduction and reinforcement learning in the basal ganglia. *Progress in Neurobiology*, 71(6):439–473, December 2003. doi: 10.1016/j.pneurobio.2003.12.001. URL http://www.sciencedirect.com/science/article/B6T0R-4BH62K9-1/2/bee825f6b204eaf551839d08ab16732d.

A. G. Barto. Adaptive critics and the basal ganglia. In J. C. Houk, J. L. Davis, and D. G. Beiser, editors, *Models of Information Processing in the Basal Ganglia*, pages 215–232. MIT Press, Cambridge, MA, 1995. URL http://mitpress.mit.edu/catalog/item/default.asp?sid=01D398C6-F7C5-47FD-A518-E241F26DF9EA\&ttype=2\&tid=8362.

R. D. Beer. On the dynamics of small continuous-time recurrent neural networks. *Adaptive Behavior*, 3(4):469–509, 1995. ISSN 1059-7123. URL http://portal.acm.org/citation.cfm?id=218539.

D. G. Beiser and J. C. Houk. Model of cortical-basal ganglionic processing: Encoding the serial order of sensory events. *Journal of Neurophysiology*, 79(6):3168–3188, June 1998. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=9636117.

G. S. Berns and T. J. Sejnowski. How the basal ganglia make decisions. In A. Damasio, H. Damasio, and Y. Christen, editors, *The Neurobiology of Decision Making*, pages 101–113. Springer-Verlag, 1996.

G. S. Berns and T. J. Sejnowski. A computational model of how the basal ganglia produce sequences. *Journal of Cognitive Neuroscience*, 10(1):108–121, January

1998. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=9526086.

J. Blynel and D. Floreano. Levels of dynamics and adaptive behavior in evolutionary neural controllers. In *ICSAB: Proceedings of the seventh international conference on simulation of adaptive behavior on From animals to animats*, pages 272–281, Cambridge, MA, USA, 2002. MIT Press.

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, December 1999. URL http://citeseer.ist.psu.edu/231717.html;ftp://cns-ftp.bu.edu/pub/diana/BroBulGro99.ps.gz.

J. Carlsson and T. Ziemke. Yaks - yet another khepera simulator. In U. Rückert, J. Sitte, and U. Witkowski, editors, *Autonomous Minirobots for Research and Entertainment - Proceedings of the 5th International Heinz Nixdorf Symposium*, pages 235–241, Paderborn, Germany, 2001. HNI-Verlagsschriftenreihe.

J. L. Contreras-Vidal and W. Schultz. A predictive reinforcement model of dopamine neurons for learning approach behavior. *Journal of Computational Neuroscience*, 6 (3):191–214, 1999.

E. A. Di Paolo. Evolving spike-timing-dependent plasticity for single-trial learning in robots. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 361(1811):2299–2319, October 2003. ISSN 1364-503X. doi: 10.1098/rsta.2003.1256. URL http://dx.doi.org/10.1098/rsta.2003.1256.

K. Diefendorff, P. K. Dubey, R. Hochsprung, and H. Scales. Altivec extension to powerpc accelerates media processing. *IEEE Micro*, 20(2):85–95, March 2000. ISSN 0272-1732. doi: 10.1109/40.848475. URL http://dx.doi.org/10.1109/40.848475.

A. Gillies and G. Arbuthnott. Computational models of the basal ganglia. *Movement Disorders*, 15(5):762–770, September 2000. ISSN 0885-3185. doi: 10.1002/1531-8257(200009)15:5⟨762::AID-MDS1002⟩3.0.CO;2-2. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=11009178.

K. Gurney, T. J. Prescott, and P. Redgrave. A computational model of action selection in the basal ganglia. i. a new functional anatomy. *Biological Cybernetics*, 84(6):

401–410, May 2001. doi: 10.1007/PL00007984. URL http://dx.doi.org/10.1007/PL00007984.

K. Gurney, T. J. Prescott, J. R. Wickens, and P. Redgrave. Computational models of the basal ganglia: from robots to membranes. *Trends in Neurosciences*, 27(8):453–459, August 2004. doi: 10.1016/j.tins.2004.06.003. URL http://www.sciencedirect.com/science/article/B6T0V-4CP6B64-1/2/f2737553e13eb034fe862e6b5033cd68.

K. N. Gurney, T. J. Prescott, and P. Redgrave. The basal ganglia viewed as an action selection device. In *Eighth International Conference on Artificial Neural Networks*, pages 1033–1038, Skövde, Sweden, September 1998. URL http://citeseer.ist.psu.edu/context/1043794/0.

D. O. Hebb. *The Organization of Behavior: A Neuropsychological Theory.* Wiley, New York, June 1949. ISBN 0805843000. URL http://www.amazon.fr/exec/obidos/ASIN/0805843000/citeulike04-21.

M. Hines and N. T. Carnevale. Computer modeling methods for neurons. In M. A. Arbib, editor, *The handbook of brain theory and neural networks*, pages 226–230. MIT Press, Cambridge, MA, USA, 1998. ISBN 0262511029. URL http://portal.acm.org/citation.cfm?id=303697.

J. C. Houk, J. L. Adams, and A. G. Barto. A model of how the basal ganglia generate and use neural signals that predict reinforcement. In J. C. Houk, J. L. Davis, and D. G. Beiser, editors, *Models of Information Processing in the Basal Ganglia*, pages 249–270. MIT Press, Cambridge, MA, 1995.

M. D. Humphries and K. N. Gurney. The role of intra-thalamic and thalamocortical circuits in action selection. *Network: Computation in Neural Systems*, 13(1):131–156, 2002. URL http://stacks.iop.org/0954-898X/13/131.

G. Huntington. On chorea. *The Medical and Surgival Reporter*, 26(15):317–321, April 1872. URL http://www.neuro.psychiatryonline.org/cgi/content/full/15/1/109.

D. Joel, Y. Niv, and E. Ruppin. Actor-critic models of the basal ganglia: new anatomical and computational perspectives. *Neural Netw.*, 15(4):535–547, June 2002. ISSN 0893-6080. doi: 10.1016/S0893-6080(02)00047-3. URL http://dx.doi.org/10.1016/S0893-6080(02)00047-3.

L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey, May 1996. URL http://arxiv.org/abs/cs.AI/9605103.

J. W. Mink. The basal ganglia: focused selection and inhibition of competing motor programs. *Prog Neurobiol*, 50(4):381–425, November 1996. ISSN 0301-0082. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=97158072.

P. Montague, P. Dayan, and T. Sejnowski. A framework for mesencephalic dopamine systems based on predictive hebbian learning. *J. Neurosci.*, 16(5):1936–1947, March 1996. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=8774460.

M. Murphy. Octave: A free, high-level language for mathematics. *Linux Journal*, 1997 (39es), July 1997. ISSN 1075-3583. URL http://portal.acm.org/citation.cfm?id=326884.

T. Oliphant. Numerical python home page. URL http://numeric.scipy.org/.

J. Parkinson. An essay on the shaking palsy. Published as a monograph, London, May 1817. doi: 10.1176/appi.neuropsych.14.2.223. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=11983801.

I. P. Pavlov. *Conditioned reflexes*. Routledge & Kegan Paul, London, 1927. URL #.

T. J. Prescott, , K. Gurney, M. D. Humphries, and P. Redgrave. A robot model of the basal ganglia: behavior and intrinsic processing. *Neural Networks*, 19(1):31–61, January 2006. ISSN 0893-6080. doi: 10.1016/j.neunet.2005.06.049. URL http://dx.doi.org/10.1016/j.neunet.2005.06.049.

D. Purves, G. J. Augustine, D. Fitzpatrick, W. C. Hall, A. S. Lamantia, J. O. Mcnamara, and M. S. Williams. *Neuroscience*. Sinauer Associates, Sunderland, Massachusetts, USA, 2003.

W. Schultz, P. Dayan, and P. R. Montague. A neural substrate of prediction and reward. *Science*, 275(5306):1593–1599, March 1997. ISSN 0036-8075. URL http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=9054347.

B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the national conference on Artificial intelligence*, volume 12, pages 337+, Seattle, WA, July 1994. John Wiley & Sons Ltd.

A. Shankar. Annie - artifical neural network library. URL `http://annie.sourceforge.net/`.

Y. Smith, M. D. Bevan, E. Shink, and J. P. Bolam. Microcircuitry of the direct and indirect pathways of the basal ganglia. *Neuroscience*, 86(2):353–387, September 1998. ISSN 0306-4522. URL `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=9881853`.

E. Suri and W. Schultz. Internal model reproduces anticipatory neural activity, 1999. URL `http://citeseer.ist.psu.edu/690846.html;http://www.cnl.salk.edu/~suri/choice.pdf`.

R. E. Suri and W. Schultz. Temporal difference model reproduces anticipatory neural activity. *Neural Computation*, 13(4):841–862, 2001.

R. S. Sutton. Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44, August 1988. ISSN 0885-6125. doi: 10.1023/A:1022633531479. URL `http://dx.doi.org/10.1023/A:1022633531479`.

R. S. Sutton and A. G. Barto. Toward a modern theory of adaptive networks: expectation and prediction. *Psychological Review*, 88(2):135–170, March 1981. ISSN 0033-295X. URL `http://www.ncbi.nlm.nih.gov/entrez/query.fcgi?cmd=Retrieve\&db=pubmed\&dopt=Abstract\&list_uids=7291377`.

R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning)*. The MIT Press, March 1998. ISBN 0262193981. URL `http://www.amazon.com/exec/obidos/redirect?tag=citeulike-20\&path=ASIN/0262193981`.

G. Tesauro. Practical issues in temporal difference learning. *Machine Learning*, 8(3-4):257–277, May 1992. ISSN 0885-6125. doi: 10.1007/BF00992697. URL `http://dx.doi.org/10.1007/BF00992697`.

G. van Rossum. *Python Reference Manual*, 2.4.2 edition, September 2005. URL `http://docs.python.org/ref/ref.html`.

F. Wörgötter and B. Porr. Temporal sequence learning, prediction, and control: a review of different models and their relation to biological mechanisms. *Neural Comput*, 17(2):245–319, February 2005. ISSN 0899-7667. doi: 10.1162/0899766053011555. URL `http://www.ingentaconnect.com/content/mitpress/neco/2005/00000017/00000002/art00001`.