Donders Graduate School for Cognitive Neuroscience
Master of Science Programme
MSc Thesis

Utilizing FORCE Learning to Model Adaptive Behavior

by

Jaap Buurman

Supervised by:

1. Marcel van Gervan

Radboud University Nijmegen

Date of final oral examination: 05-09-2016

Student name: Jaap Buurman
Student number: s0828122

1 Contents

2 Introduction 3

3 Methods 7

3.1 Optimal control theory 7

3.2 Reinforcement learning 8

3.3 Online learning 9

3.4 Varying degrees of complexity 9

3.5 Reservoir Networks 10

3.6 FORCE Learning 12

3.7 Tasks 13

3.7.1 Particle in a box problem 14

3.7.2 Pendulum problem 15

3.7.3 Mountain Car problem 17

3.8 FORCE Reinforcement Learning Framework 19

3.8.1 State Prediction 19

3.8.2 Policy Readout 22

3.8.3 Reward Readout 25

4 Results 28

4.1 Particle in a Box 29

4.2 Pendulum 33

4.3 Mountain Car 39

5 Discussion 45

6 References 48

Humans are able to learn from the environment and show a wide range of adaptive behaviors to solve the task at hand. They learn by trial and error. While reinforcement learning allows for artificial agents to learn via trial and error, they do so with algorithms that might not be the most biologically plausible. Recently, a new algorithm to train recurrent neural networks called FORCE learning has been proposed and this way of learning might be a lot more biologically plausible. We would like to research whether we can utilize this new algorithm to model adaptive behavior. Performance on a set of three toy problems was evaluated and it was shown that these agents were indeed able to learn to perform these tasks. Interestingly, this way of learning showed phenomena that are comparable to phenomena found in biological brains.

2 Introduction

Humans can show incredibly complex adaptive behavior in a sheer infinite number of situations. This behavior arises from an interaction between the environment, their nervous system and their bodies (Chiel & Beer, 1997). Their bodies create constraints on the control they can exert on their environment and generate feedback about their actions and the environment via sensory processing. In Tishby & Polani (2011) this is described as the perception-action cycle, where the current state influences our sensory perception, which in turn gets processed and saved to memory, and after this processing can generate actions which influence the state, thus closing the cycle. A graphical representation of this perception-action cycle is presented in Figure 1. Adaptive behavior uses this perception-action cycle to reach states which are closer to the current goal.

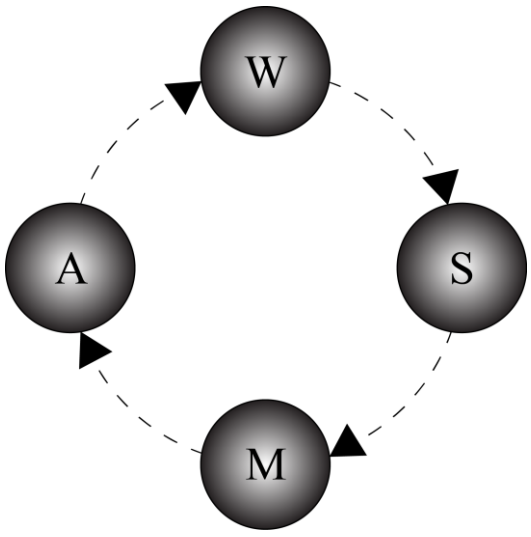

Figure 1. Graphical description of the perception-action cycle. The states W influence the sensors S of the agent, which gets processed and saved to memory M. This in turn generates actions A which influence the state and closes the cycle. Edited and reproduced from Tishby & Polani (2011).

Even when presented with a novel task, humans are capable of acceptable performance by means of close monitoring of the effects of their actions. This is done by generating a prediction about the consequences of one's action, and then comparing this prediction to the true effect. This prediction is called the efferent copy (Angel, 1976; Grüsser, 2010). Deviations from the desired outcome can then be corrected for. When a task has been repeated often enough it can become habitual and automatic (Bargh, 1994, 1996; Moors & Houwer, 2006; Wulf, McNevin, & Shea, 2001).

Optimal control theory has the goal of objectively maximizing the return from (or, equivalently, minimizing the cost of) a control process (Kirk, 2012). Optimal control theory uses a mathematical formalization of the control problem and then defines a reward or loss function related to this system. Reinforcement learning is learning what to do in certain situations in order to maximize the reward or minimize the loss (Sutton & Barto, 1998). This is achieved by mapping situations (states) to actions (control). In practice, reinforcement learning aims to approximate the optimal solution by exploration and exploitation (Sutton, Barto, & Williams, 1992). With exploitation the agent selects control values which it knows are good in order to bring itself closer to its goal. With exploration the agent selects actions that are not necessarily best according to its current knowledge, but allows him to visit unknown states in order to search for better strategies than it currently knows about.

Reinforcement learning has recently been applied to a number of Atari games with deep Q-learning (Hausknecht & Stone, 2015; Mnih et al., 2013, 2015). Because there is usually a long delay between actions and the resulting rewards, with Q-learning a function is learned that estimates future rewards conditioned on the action taken. In these cases deep and recurrent neural networks have been used to learn this function. Importantly, Q-learning traditionally does not work with continuous control values and can thus not be a good description of adaptive behavior as seen in organisms, which can show many behaviors with continuous control such as grasping motions. This issue has been resolved with actor-critic learning (Lillicrap et al., 2015) which does work with continuous control values.

For these algorithms they used online learning with stochastic gradient descent. Because of the random initialization of the weights the network starts off as a very poor approximation of the function of interest and then gradually improves during training. This very poor starting performance is in stark contrast to the acceptable performance with which human players start on new tasks such as these Atari games. This slow reduction of the error is inherent to gradient descent algorithms in neural networks.

In contrast to gradient descent based neural networks, echo state networks (ESNs) do not utilize gradient descent. An ESN is a recurrently connected neural network with random connection initialization, often called a reservoir. In contrast to classic recurrent neural networks, these recurrent connections are left unaltered during training. Output units are connected to the hidden units in the reservoir and it is on these connections between the hidden units and the readout units that learning takes place. The fact that learning only takes place on the readout unit weights makes it possible to use the exact same network for completely different tasks by using just a different readout unit. This resource sharing could be highly efficient in real brains and in fact a neural computation framework has been proposed suggesting the same thing (Maass, Natschläger, & Markram, 2002). The fact that ESNs use recurrently connected neurons also makes them biologically more plausible than feedforward models, as real brains are highly recurrent. We would like to use these biologically more plausible networks for reinforcement learning. However, training these echo state networks originally required offline-learning algorithms such as ridge regression (Marquardt, 1970).

In the work by (Sussillo & Abbott, 2009) a new algorithm for training reservoir networks was introduced, namely FORCE (first-order reduced and controlled error) learning. This algorithm works differently than traditional neural networks learning algorithms, in which modifications slowly reduce initial large errors. In contrast, in FORCE learning errors are forced to be small from the start and the goal of the learning algorithm is to reduce the amount of modification needed to keep the errors small. Moreover, this algorithm is an online-learning algorithm in which the weights are updated every single time-step rather than only once at the end of a trial (Bottou, 1998). Arguably, this different way of learning more closely mimics learning processes as seen in humans, in which performance is decent from the start and improved until the adaptive behavior becomes automatic.

Another key difference between standard ESNs and reservoirs trained by FORCE learning, is that the recurrent connections of the standard ESNs satisfy the so-called echo state property (Koryakin, Lohmann, & Butz, 2012; Lukoševičius, 2012). This means that the activity in these reservoirs decays as a function of time in the absence of any input. In contrast, the reservoirs most commonly used in FORCE learning are chaotic (Sussillo & Abbott, 2009). Hence, these reservoirs stay active even in the absence of input. Chaotic neural activity would be an important property to have in artificial neural networks that try to put forward a biologically plausible model, since biological brains also show patterns of chaotic

activity (Amit & Brunel, 1997; Brunel, 2000; Sompolinsky & Crisanti, 1988; van Vreeswijk & Sompolinsky, 1996).

Sussillo & Abbott (2009) have demonstrated the FORCE algorithm on a wide variety of tasks. They have used their algorithm to teach their reservoir to produce complex periodic and aperiodic signals and even used it to build a stable 4-bit memory. They have also linked their algorithm to biological tasks such as running and walking by training their network on motion capture data after which they could simulate both walking and running motions with the same network. This is of course a supervised learning task as the network learned to mimic the motion capture data. Our goal is to see if we can utilize FORCE learning in a reinforcement learning setting.

Thalmeier, Uhlmann, Kappen, & Memmesheimer (2015) used FORCE learning to solve a dynamic task in which a FORCE learning neural network had to exert a force on a pendulum to try and get the pendulum in an upright position by swinging it, and then balancing it. They achieved this by teaching the neural network how certain control values influence the system in a training phase in which random controls were used. During the testing phase, random control values were fed into the neural network which was then used to predict the resulting state of the system by means of simulating the effect of this control. The control values then got a reward value assigned by calculating how close it was to the goal state. The final control value that was used, was an exponentially weighted average of the control values, weighted by their assigned reward values.

This way of solving dynamic problems might not be the most computationally efficient, since simulating the effects of random controls is a rather brute force approach. At the same time, it seems closely related to certain phenomena in biological neural networks. It has been shown for example that certain hippocampal cellular assemblies show pre-play, a temporal sequence of hippocampal cell firing, which could be closely related to planning future events (Meer & Redish, 2010; Pfeiffer & Foster, 2013). One could view this as a way for the brain to simulate the future states and evaluate its possible consequences. Still, adaptive behavior as can be seen in animals and humans in real situations is unlikely to involve evaluation of random actions in a separate training phase. In this paper we aim to unify the training and testing phase, and devise an online learning method for solving dynamic problems. We will use the FORCE algorithm to model adaptive behavior and learning in dynamic situations and we will encounter phenomena that are likely to be of importance in the generation of adaptive behavior in biological organisms as well. In this

paper we specifically address the question whether FORCE learning can be effectively utilized to learn adaptive behavior in complex dynamic environments.

3 Methods

We will start by first formally defining a few key components of our implementation.

3.1 Optimal control theory

Formally, the objective of optimal control theory is to determine the control signals that will cause a process to satisfy the constraints and at the same time minimize or maximize some performance criterion (Kirk, 2012). To solve such a problem, it requires a mathematical description of the process that needs to be controlled, a statement of the constraints of the system and the specification of the performance criterion.

The processes of an optimal control problem are often modelled using simple ordinary differential equations that describe how the state changes as a function of the current states, current control values and the current time (Todorov, 2006). They contain the different state and control variables, often denoted by $x_n(t)$ and $u_n(t)$ respectively, where t is the current time and where n is a variable index. Rather than using multiple variables, the state of the system is defined with a state vector:

$$x(t)^T \equiv [x_1(t), x_2(t), \dots, x_n(t)]$$

The same is done for the control variables, resulting in the control vector:

$$u(t)^T \equiv [u_1(t), u_2(t), \dots, u_n(t)]$$

The differential equation can then be written as a function of these two vectors, which is referred to as the state equation:

$$\dot{x}(t) = f(x(t), u(t), t)$$

The constraints of the system can impose constraints on the possible control and/or state values. They also determine the starting and possibly the ending states of the system, called the boundary conditions. The starting condition is often denoted by $x(t_0)$ while the final state is denoted by $x(t_f)$.

Lastly, the optimal control problem needs an objective performance measure. This performance measure is an integral over time in the case of a continuous time system or a sum over time in the case of discrete time steps. This performance measure is often denoted by *J* and has the following general form:

$$J = h\big(x(t_f), t_f\big) + \int_{t_0}^{t_f} g(x(t), u(t), t)dt$$

where h and g are scalar functions. In the case of discrete time steps the integral is replaced by a sum.

Solving the optimal control problem is equivalent to finding an admissible control u^* which causes the system to follow an admissible trajectory x^* that minimizes *J* if it is a loss function or maximizes it in case of a reward function. In this case, u^* is called the optimal control and x^* is the optimal trajectory. The goal of optimal control is to find a function f called the optimal policy which can be used to calculate the optimal control:

$$u^*(t) = f(x(t), t)$$

3.2 Reinforcement learning

In reinforcement learning the goal is to learn to map situations to actions in order to maximize a numerical reward or to minimize a numerical loss (Sutton & Barto, 1998). This mapping needs to be learned without being explicitly being told what to do. Rather, this mapping is learned over time by trial and error. Actions are tried and the outcome is observed to learn this mapping. If the effect of one's actions on the cost or reward is instant, then learning this mapping is relatively easy. But in harder problems, actions might not only influence the immediate reward or loss, but all subsequent rewards and losses as well. This delayed reward/loss leads to a credit assignment problem (Sutton, 1984).

In reinforcement learning the optimal policy needs to be learned by trial and error, and because of that a trade-off is created between so-called exploration and exploitation. In

exploration, new actions are being tried to try and discover good actions for producing reward or avoiding loss. At the same time, in order to perform well on the task at hand, the agent also has to use exploitation to pick actions that have been found to be beneficial in the past. In order to do well on a reinforcement task, these two tasks need to be carefully balanced for optimal performance (Ishii, Yoshida, & Yoshimoto, 2002; Thrun, 1992).

3.3 Online learning

In online learning, learning takes place as new samples come in sequentially (Bottou, 1998). Online learning is important for reinforcement learning, because beforehand we do not have access to a large number of labeled training samples. Rather, new samples (new states) arrive sequentially as actions are being evaluated by the agent. While there are online-learning based gradient descent algorithms such as stochastic gradient descent, performance typically starts off very low and then gradually improves as more and more samples come in.

3.4 Varying degrees of complexity

Optimal control problems can be categorized in terms of varying degrees of complexity. We would like to study how our implementation performs under these different levels of complexity since, arguably, if our approach is to model adaptive behavior in biological organisms, different problem classes must be handled well. First of all, the optimal control problem can be cast as a linear system or a nonlinear system (Todorov, 2006). Linear systems are constrained to linear relationships between variables and are thus easier to solve than their nonlinear counterparts. Second, problems can be fully observable of partially observable (Flokentin, 1962). When problems are fully observable, all the information about the problem can be observed at any time. This means that the current state contains all the information there is, and thus the next state only depends on this current state. The history of the previous states is irrelevant for the next state. In contrast, in a partially observable problem, the current state does not contain all the information to fully characterize a problem, and thus the history about past states is also important.

In order for our reinforcement learning framework to be a good description of adaptive behavior of real organisms, we should be able to use it to solve the more complex problems. For example, many real-world problems in adaptive behavior are highly nonlinear and thus our system should be able to deal with nonlinear problems. Furthermore, sensory input is

noisy by nature (Faisal, Selen, & Wolpert, 2008; Micheyl, McDermott, & Oxenham, 2009; Mortensen & Suhl, 1991) and thus real world problems are partially observable by definition. Because of this, we will focus our research on solving problems that are artificially corrupted by noise, such as to stay as close as possible to real-world problems. Since our current sensations are corrupted with noise, a better approximation of the ground truth state can be achieved by taking previous states into account. The recurrent connections in reservoir networks allow them to keep previous states in memory.

3.5 Reservoir Networks

Reservoir networks are a particular kind of recurrent neural network in which the recurrent connections are randomly initialized and left unaltered during training (Jaeger, 2004; Jaeger, 2010). These connections can be fully connected or sparsely connected. The neurons which are recurrently connected are called the hidden units and constitute the reservoir. For each output that needs to be produced, an output unit is linearly connected to all the hidden units via a weight vector. It is these weights that are trained in reservoir networks, usually as a simple regression task.

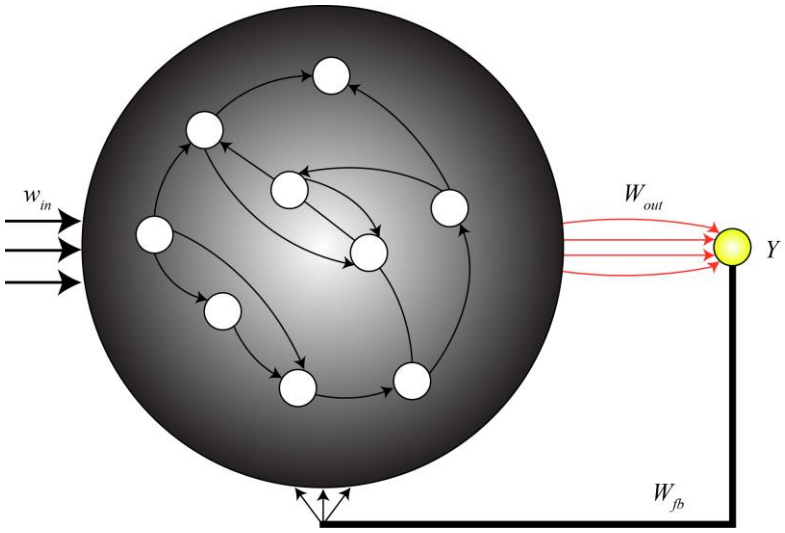

Figure 2. Graphical description of a reservoir network. The large disc is called the reservoir and contains the hidden units which are denoted by the smaller discs inside. The red weights labeled with W_{out} are the only connections that are trained. The black arrows W_{fb} are the feedback connections, while the white arrows W_{in} to the left are the input connections. Edited and reproduced from Sussillo & Abbott (2009).

Reservoir networks can be used to produce periodic or aperiodic signals without any inputs, but they can also be used for input-output mapping when inputs are provided. For a network with N hidden neurons, K input units and L output units several weight vectors and matrices are defined. First of all, we have a $N \times N$ matrix W_{rc} which contains the recurrent connection strengths. There are usually drawn from a uniform distribution, although normal distributions can also be used. Importantly, this matrix must have a spectral radius g of less than unity by scaling the matrix for the network to show the so-called echo state property (Lukoševičius, 2012). In other words, the largest absolute eigenvalue of this weight matrix should be less than one. This means that the activity of the reservoir dies out over time in the absence of input.

Secondly, we have a $N \times K$ matrix W_{in} that contains the strength of the input connections from the input units to the hidden units. Lastly, there is an optional $N \times L$ weight matrix W_{fb} that contains the connections from the output units back to the network and is used as a feedback signal. Given these matrices, the activation of the hidden neurons in the reservoir are updated as follows:

$$a(t+1) = \sigma(W_{rc}a(t) + W_{in}i(t+1) + W_{fb}y(t))$$

where $a(t)$ is the current state of the hidden units, $i(t+1)$ is the input and $y(t)$ is the previous output of the network. The function σ is the activation function of the hidden neurons and is usually a sigmoidal function such as the hyperbolic tangent. Given the current state of the network, the output of the network is calculated as follows:

$$y(t) = W_{out}a(t)$$

where W_{out} is a $L \times N$ matrix containing the output connection strengths. This is the only matrix that is trained in echo state networks. The remaining matrices are initialized randomly and left at these starting values. Training of the output weights usually consists of solving a simple linear regression task:

$$W_{out} = A^{-1}y$$

where A^{-1} is the pseudoinverse of the $N \times T$ state matrix that is collected over T time-steps. In order to calculate this pseudoinverse, the states of the network over the entire training time need to be collected first. It is for this reason, that this is an offline learning algorithm.

3.6 FORCE Learning

FORCE learning is a relatively new learning algorithm for reservoir networks (Sussillo & Abbott, 2009). It has two major advantages. First of all, learning W_{out} takes place at every time step, and thus it is an online learning algorithm. Second, in contrast to gradient descent-based algorithms in which initial large errors are slowly reduced, in FORCE learning errors are low from the start and the goal of the algorithm is to keep these errors small by rapid changes of the output weights.

In contrast to regular ESNs, FORCE learning usually uses a leaky update rule for the activation of the network. That is, the activation of the hidden units is not only dependent on the incoming activation, but also on the activation at previous time points. The change in activation is denoted by:

$$\tau \dot{a}(t) = -a + W_{rc} f(t) + W_{in} i(t) + W_{fb} y(t)$$

where τ is the time-constant of the network and f denotes the firing rate of the neurons. The update of the activation is approximated by means of the Euler method giving:

$$a(t+1) = (1-\tau)a + \tau \left(W_{rc} f(t) + W_{in} i(t) + W_{fb} y(t) \right)$$

Because the activation is now written as this differential equation, we use a separate equation for the resulting firing rate of the hidden neurons:

$$f(t) = \tanh\big(a(t)\big)$$

The output is again a simple linear readout as was the case in the echo state networks:

$$y(t) = W_{out} f(t)$$

Another change with FORCE learning compared to echo state networks is that the spectral radius can be larger than unity for the W_{rc} matrix. As a matter of fact, it was found (Sussillo & Abbott, 2009) that a spectral radius g of around 1.5 worked best for FORCE learning. In practice this means that in the absence of input, the reservoir will stay active forever, displaying chaotic activity.

Training in FORCE learning is again usually restricted to the output weight vector. There are multiple algorithms that can be used for updating these weights in FORCE learning, but the best performing one is the recursive least-squares algorithm (Haykin, 2008). In this algorithm the weights are updating during each time step by the following equation:

$$w_{out}(t) = W_{out}(t-1) - \delta(t)P(t)f(t)$$

where $\delta(t)$ is the error at time t before the weight update:

$$e(t) = y(t) - T(t)$$

with $T(t)$ the desired output at time t. In this algorithm, $P(t)$ is a running estimate of the inverse of the correlation matrix of the network's firing rates f plus a regularization term, which is updated at each time step by:

$$P(t) = P(t-1) - \frac{P(t-1)f(t)f^T(t)P(t-1)}{1 + f^T(t)P(t-1)f(t)}$$

The algorithm also requires an initialization of P, which is given by:

$$P(0) = \frac{I}{\alpha}$$

where I is the identity matrix and α is a constant, which controls the learning rate.

3.7 Tasks

Now that the general concepts have been introduced and defined, we will define the problems that we aimed to solve.

3.7.1 Particle in a box problem

We start by introducing an optimal control problem which we refer to as the particle in a box. For this problem, the mathematical optimal solution is known, which means we can easily compare the solution that our online reinforcement learning algorithm produces. The particle in a box problem simulates the movement of a particle through a box as a function of control values. It is a linear system and the version described below is fully observable. The original problem considers a two dimensional box but since the dynamics of the particle in both dimensions are completely independent, solving the problem in one dimension is sufficient. This same solution can then be applied to the other dimension as well. Therefore, for simplicity we will consider a one-dimensional case of the particle in a box simulation. The state equation of the particle in a box simulation is as follows:

$$\dot{x} = Ax + Bu = \begin{pmatrix} 0 & 0 \\ 1 & 0 \end{pmatrix} x + \begin{pmatrix} 1 \\ 0 \end{pmatrix} u$$

where the state vector x is defined as:

$$x^T(t) = \big(x_1(t), x_2(t)\big)$$

where $x_1(t)$ and $x_2(t)$ are the speed and position at time t respectively. In the one-dimensional case, $u(t)$ is just a scalar value and denotes the acceleration. This is the external control that can be applied to the system. The goal is to choose optimal values for u such that the following reward function is maximized:

$$J = -\sum_{t=0}^{t_f} (x(t)^T Q x(t) + u(t)^2)$$

$$= -\sum_{t=0}^{t_f} \left(x(t)^T \begin{pmatrix} 0 & 0 \\ 0 & 1 \end{pmatrix} x(t) + u(t)^2 \right)$$

$$= -\sum_{t=0}^{t_f} (x_2(t)^2 + u(t)^2)$$

This means that we have a quadratic cost associated with both position and acceleration. Because of the quadratic cost associated with position, this results in the goal of bringing our position to zero i.e. the middle of this one-dimensional box. This task can be optimally solved via a linear-quadratic regulator design for state space systems (Anderson & Moore, 2007). This algorithm can be used to calculate a gain matrix K such that the optimal control value that maximizes J over all the time-steps is given by:

$$u^*(t) = -Kx(t)$$

We can use these optimal control values to calculate the optimal trajectory x^* for each of our simulations and then compare our solution to this optimal solution. The constraints for this optimal control problem are given as follows:

$$x(t_0) = (0, \pm 0.8)$$

where the sign of the starting-position is chosen randomly each trial.

For our particle in a box, we will run each trial for 500 time-steps with a time-step size of $\frac{1}{60}s$. The successive states are updated by means of the Euler approximation (Süli & Mayers, 2003).

3.7.2 Pendulum problem

The second task that we would like to solve is the pendulum task. This task simulates the movement of a swinging pendulum with unit length (arbitrary units) as a function of control values. Since it is a non-linear problem, it is a more difficult problem than the particle in a box problem. The dynamics of the pendulum simulation are given by a second order differential equation and is as follows (Thalmeier et al., 2015):

$$\ddot{\phi}(t) + 0.1\dot{\phi}(t) + 10\sin\big(\phi(t)\big) = u(t)$$

This second order differential equation can be rewritten in vector form to yield the state equation:

$$\dot{x} = (x_2, u - 0.1x_2 - 10\sin(x_1))$$

where the state vector x is defined as:

$$x^T(t) = \big(x_1(t), x_2(t)\big)$$

where $x_1(t)$ and $x_2(t)$ are the angle and angular velocity at time t respectively. Again, $u(t)$ is just a scalar value and denotes the acceleration. This is the external control that can be applied to the system. In order to keep the numbers that are fed to the neural network in a certain range, we chose to transform the state into the x- and y-position of the tip of the pendulum rather than the current angle. The values that were used with the neural network are given by:

$$y^T = \left(\sin(x_1), -\cos(x_1), \frac{x_2}{2\pi}\right)$$

so that $y_1(t)$, $y_2(t)$ and $y_3(t)$ denote the x-position of the tip of the pendulum, y-position of the tip of the pendulum, and the rescaled angular velocity respectively.

The goal is to choose optimal values for u such that the following reward function is maximized:

$$J = \sum_{t=0}^{t_f} y_2(t)$$

That is, we have a linear reward associated to the current height of the tip of the pendulum. In order to maximize the reward, the goal is to bring the pendulum in an upright position and then balance it.

The constraints for this control problem are given as follows:

$$x(t_0) = (0, 0)$$

For this pendulum swing problem, we will run each trial for 500 time-steps with a time-step size of $\frac{1}{100}s$. The successive states are updated by means of the Euler approximation (Süli & Mayers, 2003).

3.7.3 Mountain Car problem

The third and final task that we would like to solve is the mountain car problem. This task simulates the movement of a car as a function of control values. The car is stuck inside a valley and the goal is to drive the car to the top of the right hill, as can be seen in Figure 3.

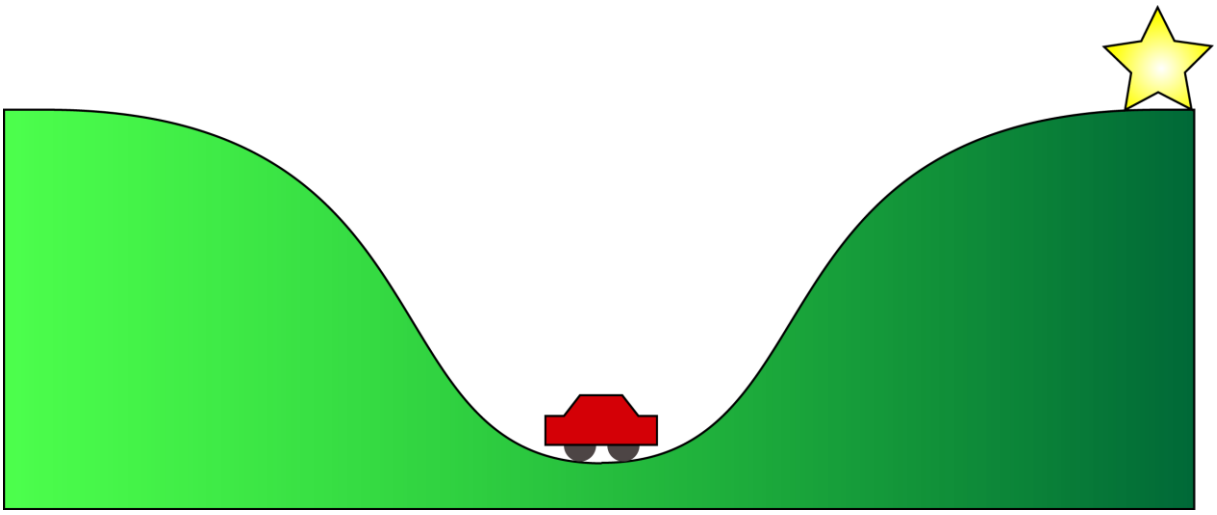

Figure 3. The mountain car problem. The car is stuck in a valley and the goal is to drive the car to the top of the hill on the right.

This is also a nonlinear problem. But what makes it especially difficult is the fact that the car's engine is underpowered and thus the car cannot simply drive to the right to reach the goal state. Rather, it must drive away from the goal state in order to build up momentum to be able to climb the hill on the right. There are many different versions of the mountain car problem, but we will use the version provided in Singh & Sutton (1996). The state equation written in vector form is given by:

$$\dot{x} = (0.001u - 0.0025\cos(3x_2), x_1)$$

where the state vector x is defined as:

$$x^T(t) = \left(x_1(t), x_2(t)\right)$$

where $x_1(t)$ and $x_2(t)$ are the speed and x-position at time t respectively. Again, $u(t)$ is just a scalar value and denotes the acceleration. This is the external control that can be applied to the system.

The goal is to choose optimal values for u such that the following reward function is maximized:

$$J = \sum_{t=0}^{t_f} x_2(t)$$

This means that we have a linear reward associated to the current x-position of the car. In order to maximize this, the goal is to bring the car as far to the right as possible. All the way on top of the mountain to the right.

The constraints for this control problem are given as follows:

$$x(t_0) = \left(0, -\frac{\cos^{-1} 0}{3}\right)$$

implying that the car is placed at the bottom of the valley at the beginning of every trail. In contrast to our previous two tasks, this one does have a final state upon which the simulation is terminated:

$$x_2\left(t_f\right) > 0.5$$

That is, the simulation is terminated once the x-position of the car is larger than 0.5. At that point, the car will be on the top of the hill on the right.

For the mountain car problem, we will run each trial with a maximum of 1000 time-steps with a time-step size of $1s$. If the agent reaches the goal state the trial is terminated. The successive states are updated by means of the Euler approximation (Süli & Mayers, 2003).

3.8 FORCE Reinforcement Learning Framework

We want to utilize FORCE learning to learn to solve the optimal control problem of the three tasks defined above and show behavior that is comparable to adaptive behavior in real organisms. Because we want to learn this in a reinforcement framework, similar to how real organisms learn new tasks, we cannot simply train a neural network on the optimal control values. We have to learn those by trial and error. In the following, we introduce increasingly sophisticated variants of our FORCE reinforcement learning (FORCE-RL) framework. Each of those variants will be validated using the introduced control problems.

3.8.1 State Prediction

Rather than learning the optimal control as the output directly, we use a network to learn to predict the next state given the current state plus a control value (Thalmeier et al., 2015). This way we can evaluate the effect of a number of controls, and then select the most optimal one. For this task we used a network with 300 neurons. We will use one input, namely the control value to use. Since the goal is to predict the next state, the network has to learn to produce the next state given this control input. The number of output units will thus be equal to the length of the state vector. In other words, this neural network has to learn the dynamics of the state equation of the task at hand yielding a state prediction. We will call this network the state network and the readout unit the state readout.

At each time-step of the task at hand, the neural network simulates the effect of c different control values, which we will call our control space. The simulated next state is then evaluated by calculating the reward value of that state. Because we are dealing with continuous control values, there are an infinite number of possible control values that can be used. We chose a similar approach as Thalmeier et al. (2015) by using an exponential weighted average of the control values, weighted by the reward value, as an approximation of the optimal control value.

Because all three tasks are dealing with delayed rewards, just simulating one state ahead is not enough for evaluation of the reward value. Because of this, our neural network needs to accurately plan the agent's actions. Therefore, we need to simulate not just one time-step into the future but multiple time-steps. To do that, we also need another control value for each successive step in the simulation. Trying the entire control space at every simulation step

for up to s simulated steps ahead would result in up to c^s different simulations, which is computationally infeasible for even moderately large s.

We can naively solve this issue by either selecting a control value of zero for every subsequent time-step after the first or, alternatively, by repeating the same control value over and over again. This results in c different simulations independent of s. These c simulated trajectories were then used to evaluate the reward and approximate the optimal control by applying an exponential weighting scheme. The exponential weighting of the control values that were tried is done in the following way. First we calculate the reward J over the c simulated trajectories. Because we calculated J for the c different simulations starting with the c different possible control values, J is a vector of length c. For numerical stability reasons, we then shifted our reward values such that the lowest reward was zero by:

$$J \leftarrow J - \max(J)$$

We then rescaled J so that these values would always be in the same expected range by scaling them such that all values were between zero and one:

$$J \leftarrow \frac{J}{\max(J)}$$

We then calculated our control by weighting our control space as follows:

$$u = \sum_{i=1}^{c} \frac{e^{\omega J_i}}{\sum_j e^{\omega J_j}} \cdot c_i$$

where ω is a weighting factor. This controls the influence of the suboptimal simulations on the weighting. Large values make the weighting of the best simulated trajectory dominate, while smaller values increase the relative importance of the other trajectories. Because our control space is very coarse we need the weighting of the other trajectories to approximate the optimal control.

Thus at each time-step the simulations will be used to calculate the control value that the agent intends to use. This final control value is then fed into the state network as the input

and the state readout is used to produce the network's output, which is the prediction of the next state $\tilde{x}(t+1)$:

$$y(t) = \tilde{x}(t+1)$$

The control value is then actually used in the task and the true next state $x(t+1)$ is evaluated, which forms the target for the FORCE learning algorithm. The prediction of the state readout is compared to the true next state, and the error between these two values is what drives the FORCE learning algorithm:

$$\delta(t) = y(t) - T(t) = \tilde{x}(t+1) - x(t+1)$$

Box 1 contains the algorithm for each trial in pseudo-code.

```
For all time-steps t
        For all values c in controlSpace
                simulateTrajectory(controlSpace(c))
                calculateReward()
        end
        control(t) = weightControlbyReward()                    % Exponential weighting scheme
        actualState(t+1) = calculateNextState(control(t))       % True system dynamics
        predictedState(t+1) = simulateStep(control(t))          % State Readout
        applyFORCE(predictedState(t+1), actualState(t+1)) % Train State Readout
end
```

Box 1. The neural network is used to simulate c different trajectories. Based on the reward associated with these simulations, a control value is calculated by the exponential weighting scheme. This calculated control value is then used to make a prediction about the next state, and at the same time used to evaluate the actual next state. The prediction is compared with the actual next state, and the error between these two values is what drives the FORCE learning algorithm.

For the state prediction network we have used the following hyper-parameters. The input, recurrent and feedback weights were initialized by drawing from an uniform distribution from -1 to $+1$. The recurrent reconnections had a sparsity of 10%, so that on average, only 10% of the connections were non-zero. The recurrent connections were then rescaled so that the maximum of the absolute of the eigenvalues was equal to g. Since a g-value of 1.5 provided the best results in previous research (Sussillo & Abbott, 2009), that is

the value that we used. The input connections were fully connected (non-sparse) and drawn from an uniform distribution from -1 to $+1$, while the feedback connections were also drawn from an uniform distribution but from -2 to $+2$. The output connections were initialized to zeros. The learning rate α was set to three. The time-constant of the neural network τ was set to 0.1. The weighting constant ω in the exponential weighting was set to 10. The states of all tasks were corrupted by Gaussian noise with a standard deviation of 0.03, so that the problems have the partial observability property. This was done to simulate the effects of sensory noise (Faisal et al., 2008; Micheyl et al., 2009; Mortensen & Suhl, 1991).

3.8.2 Policy Readout

The problem with the naive approach to selection of control values during simulation for the subsequent time-steps is that it is impossible to use control values with opposite sign at different time-steps in the same simulation. This is problematic for control problems that require this, such as the Mountain Car problem and the Pendulum problem. Both problems require control in one direction to build up momentum and then control in the other direction to reach the final goal.

In the current state-of-the-art AI playing the Asian game of Go, good moves are found by playing simulated games through Monte-Carlo tree search (MCTS) (Silver et al., 2016). Similar to how we cannot simulate all possible control values at each time step, the breadth and width of the tree encompassing all possible moves is extremely large (~250 and ~150 respectively), resulting in a search-space that is computationally infeasible to traverse completely. This was tackled by introducing a neural network that is referred to as the policy network. This network would output the probability distribution of all legal moves given the current state of the game. Only probable moves and thus probable games were then evaluated during the MCTS to evaluate the value of certain plays. This effectively constrained the breadth of the tree which was used in MCTS.

This policy network was pre-trained through supervised learning on expert games, and another version was pre-trained by reinforcement learning with self-play. We are going to use a similar trick for constraining the breadth of our simulations, but rather than pre-training a separate network, we want to use a separate readout that is trained in an online fashion. To evaluate the effects of different control values we will still consider the entire control space for the first time-step in our simulation, but the subsequent control values that are used will be

provided by a policy readout that is connected to the state reservoir. This readout was also trained with FORCE learning and was used to predict the next control value $u(t+1)$.

We could also use a separate network for this task by providing it with the current state $x(t)$ and predicting the control value $u(t)$. With this network, we could then use simulated states during simulation as the input to calculate control values to use during the next step. But since the reservoir of the state network already contains a rich representation of the state, a separate network is not necessary. In fact, during testing we found that using two separate networks resulted in worse performance compared to using just one. This is probably due to the fact that each readout step introduces small errors. And thus a single readout step is more efficient compared to reading out the simulated state, feeding that information in a second network and then finally using another readout unit to calculate the control.

The state readout units and policy readout unit tried to solve the tasks by working together. The state network and readout were used to simulate c different future states based on the current state and a control value from our control space. Since the policy readout is trained to predict the control value one step into the future, this prediction can be used as the control input for simulating the next time-step. This process of simulating further and further ahead was used to simulate s number of time-steps into the future. The c simulated trajectories were then used to evaluate the reward and then used to approximate the optimal control by applying the same exponential weighting scheme.

So how was this policy readout unit trained? First, the policy readout was used to predict the control value that was calculated by the exponential weighting scheme. This predicted control value was compared to the actual found control value. The error between the prediction and found control value drove the FORCE learning algorithm for the policy readout:

$$\delta(t) = y(t) - T(t) = \tilde{u}(t+1) - u(t+1)$$

Since the policy readout is predicting the control value for the next time-step rather than the current time-step, we wait for the calculation of the control value at the next time-step before we apply the weight update by FORCE learning. This is necessary since it takes one time-step before we learn what the target should be for the control value. In Box 2 the algorithm is given in pseudo-code.

```
For all time-steps t
        For all values c in controlSpace
                simulateTrajectory(controlSpace(c))
                calculateReward()
        end
        control(t) = weightControlbyReward()        % Exponential weighting scheme
        if t ~= 1    % if t = 1 we don't have a prediction of u(t) yet which is made at t-1
                applyFORCE(predictedState(t), actualState(t))          % Train State Readout
                applyFORCE(predictedControl(t), control(t))           % Train Policy Readout
        end
        actualState(t+1) = calculateNextState(control(t))       % True system dynamics
        predictedState(t+1) = simulateStep(control(t))          % State Readout
        predictedControl(t+1) = predictControl()                % Policy Readout
end
```

Box 2. FORCE learning is now applied before the prediction of the next state, and the state update itself. This means that the prediction from one time-point earlier is still in memory. Since the state prediction is about the state one time-step ahead, this prediction is about the state at the current time-step. Since the neural network is then used to calculate the control value for the current time-step, we can then use FORCE learning to train the network to predict both the state and the control value at the next time-step.

Regardless whether the control values are any good, the state readout learns to predict the future state given the control value as input. These improved predictions can then be used to help select the best actions in any given state by simulating future states and evaluating the reward of these future states. Learning for the policy readout is not that simple. At the beginning, the policy readout is very bad at giving good suggestions about which control value to use. So the simulations are not representative of good trajectories. Still, they are good enough to evaluate to some extent the quality of the control values from our control space. And thus we get a better than random choice for our calculated control value, which is then used to learn the policy readout.

Since the policy readout learned a better than random control value, this improves the control values that this readout will suggest from that point on. This improves the quality of the simulations, and thus the calculated control value will also get closer to the optimal value. This improvement is then also learned by policy readout. This cycle of better control values resulting in better simulations resulting in even better control values continues until the system converges.

Even though the state readout units had feedback connections, no feedback connections were used for the policy readout unit as this impaired performance. The control values that get learned in the beginning are only better than random, but not even close to optimal, and thus over time better and better values get learned. This causes the distribution of

the feedback to change over time. Our hypothesis is that the learning is sensitive to this change in feedback distribution, which hinders performance, making it better to disable these feedback connections altogether.

3.8.3 Reward Readout

The total length in the number of time-steps of each simulation is an important parameter. It needs to be long enough so that the simulation can successfully be used for the agent to evaluate the expected reward based on the simulation. However, this means that simulations are computationally infeasible for very long tasks and are computationally impossible for infinite horizon problems (Bertsekas, 1995), where there is no end of the task by definition.

Another reason on top of computationally feasibility is accuracy. The longer the simulations are, the more the errors in the simulation from time-step to time-step will compound, resulting in decreasing accuracy as a function of simulation length. Both these reasons lead to the desire to constrain our simulation not only in breadth, as was done by the policy readout unit, but also in depth.

AlphaGo uses this same technique by introducing a value network (Silver et al., 2016). This network was trained to estimate the probability of either player winning given a certain state of the board. This way, rather than sampling the entire length of the tree during the tree search, the depth is constrained by terminating the current tree search and by evaluating the last sampled state by the value network. This way, the value of a certain board position could be evaluated without the simulation of the game all the way to the end.

We want to implement something similar by using temporal difference learning (Richard S. Sutton, 1988; Tesauro, 1995) to estimate the future discounted reward given the current state. With temporal difference learning the goal is to produce at each time-step an estimate of the following quantity:

$$R(t) = \sum_{i=0}^{\infty} \gamma^i r(t+1+i)$$

where $r(t)$ is the reward at time-step t, $R(t)$ is the estimated future discounted reward and γ is the so-called discounting factor with $0 \leq \gamma \leq 1$. Higher values of gamma increase the relative importance of the reward more into the future.

Rather than training a separate network to estimate the future discounted reward, we again opt to use an extra readout which we will call the reward readout unit. We tried to train this readout similar to how the previous readout units were trained by applying the recursive least squares algorithm, but this implementation diverged after the first trial. So we opted to use the gradient descent algorithm with eligibility traces (Sutton & Tanner, 2005; Sutton, 1984; Tanner & Sutton, 2005; Tesauro, 1995). For this, we also need a target value during each step similar to the recursive least squares algorithm. Since we want it to predict the future discounted reward, this will be the target. But how do we obtain the target? If we write out the above equation the target becomes:

$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \gamma^3 r_{t+4} + ..$$

As we can see, in order to obtain the target for the future discounted reward, we will have to wait until all the reward values come in. This would prevent online updating of the reward readout, and for infinite horizon problems you will never have all the future reward values since there are an infinite number of time-steps and thus infinite number of reward values. Luckily we can rewrite the equation in the following way:

$$R_t = r_{t+1} + \gamma(r_{t+2} + \gamma^1 r_{t+3} + \gamma^2 r_{t+4} + ..)$$

We now notice that the part of the equation between the brackets contains the future discounted reward of one time-step later, and thus we can write:

$$R_t = r_{t+1} + \gamma R_{t+1}$$

This equation is the so-called temporal difference target. It is this target that will be used to calculate the so-called temporal difference error, which is simply the difference between the current prediction and the temporal difference target:

$$\delta(t+1) = r_{t+1} + \gamma R_{t+1} - R_t$$

The vector w of size $N \times 1$ that holds the weights of the connections from all the neurons in the reservoir to the reward readout unit is classically updated in the following way:

$$w^i(t+1) = w^i(t) + \alpha * \delta(t+1) * f^i(t)$$

Where α is the learning rate, w^i is the weight from the i'th neuron to the reward readout unit and f^i is the fire rate of the i'th neuron. As mentioned before, we used eligibility traces to improve the performance of the temporal difference algorithm. In this case rather than using the firing rate in the update of the weights, we use the eligibility trace:

$$w^i(t+1) = w^i(t) + \alpha * \delta(t+1) * e^i(t)$$

where $e^i(t)$ is the eligibility trace of the firing rate of the i'th neuron, which is calculated as follows:

$$e(t+1) = \gamma * \lambda * e(t) + f(t+1)$$

where γ and λ are parameters. Similar to γ, we have $0 \leq \lambda \leq 1$. Eligibility traces help retain information about previous states of the regressors of the reward readout unit, in this case the firing rate of the neurons, making it easier to learn long-term dependencies resulting in faster learning and better performance (Singh & Sutton, 1996; Tanner & Sutton, 2005). Box 3 shows a summary of the algorithm that was used in our final architecture.

```
For all time-steps t
        For all values c in controlSpace
                simulateTrajectory(controlSpace(c))
                calculateReward()     % This is now done by the reward readout
        end
        control(t) = weightControlbyReward()        % Exponential weighting scheme

        if t ~= 1
                applyFORCE(predictedState(t), actualState(t))          % Train State Readout
                applyFORCE(predictedControl(t), control(t))           % Train Policy Readout
        end
        actualState(t+1) = calculateNextState(control(t))       % True system dynamics
        predictedState(t+1) = simulateStep(control(t))          % State Readout
        predictedControl(t+1) = predictControl()                % Policy Readout

        reward(t+1) = obtainReward(actualState(t+1))            % obtain the reward
        predictedReward(t+1) = calculateReward()                % make a reward prediction
        TDError(t+1) = calculateTDError()                               % calculate the TD error
        updateWeightsRewardReadout(TDError)                     % update the weights
end
```

Box 3. The same algorithm is used, but now a third readout unit, namely the reward readout which trained by temporal difference learning.

For this final architecture a few more parameters have been introduced. A value of 0.98 was used for both γ and λ. The learning rate α (not be confused with the learning rate for the recursive least squares algorithm that was used for the other two readout units) was set to 10^{-4}. Similar to the policy readout unit, this readout unit had no feedback connections as this impaired the performance. This is also because the reward predictions get better over time, which means that the distribution of the feedback signal would be different at the start of the trial than at the end.

4 Results

Because of the fact that for many of the results that we would to present the distributions are bimodal, standard statistical tests that compare the means would not be appropriate. This led to our decision to perform a qualitative analysis by visually interpreting the results.

4.1 Particle in a Box

We first evaluated the results by running 100 separate agents through 10 trials in which only the state prediction readout was used, using the same control value in every subsequent time-step in the simulation of the trajectories. Rather than plotting the reward J, we opted to plot the loss instead, which is simply the negative of the reward, or in other words $-J$. Since the optimal trajectory is known, we compared the loss of the trajectory produced by the agent to the loss of the optimal trajectory. These results can be found in Figure 4. As we can see, the loss quickly decreases as a function of trial number and then settles down in a steady state. The average loss is almost three times as high as the optimal control manages to produce.

The problem is that the usage of the same control value for every time step is not even close to optimal, and thus the simulations are not very representative of an optimal trajectory. This is due to the fact that there is a quadratic cost associated with control. This penalizes simulations which select high control values, since these high values are repeated for every time-step and thus result in a very high loss over the entire trajectory, which effectively biases the agent to select very small control values. In contrast, large control values at the start are actually very good because they quickly ramp up the speed, which results in a quicker trajectory towards a position of zero, which reduces the quadratic loss associated with position.

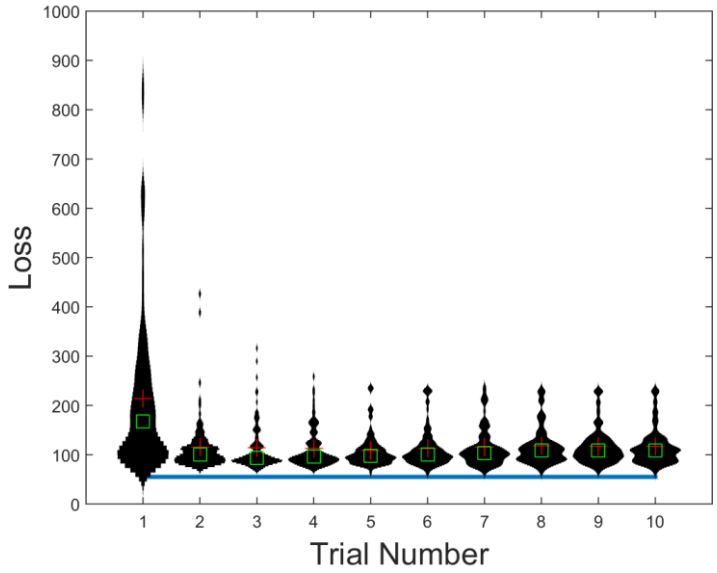

Figure 4. The distribution of the loss of 100 different agents on the particle in a box problem as a function of number of trials. In this test, the simulations were ran by repetition of the control values for

all subsequent time-steps in the simulation. The blue line denotes the loss of the mathematically available optimal trajectory, the red cross denotes the mean, while the green square denotes the median.

The second option we have is to use zero values for all subsequent time-steps in our simulations. The results of 100 agents with 10 trials each are presented in Figure 5. As we can see, the performance is similar to the usage of the same control value for all subsequent time-steps. Because now only the control value for the first step in the simulation is non-zero, the agent can be more aggressive in the beginning of a trial by selecting higher control values, such that its speed more quickly ramps up. This allows the agent to reach a position close to zero earlier, which reduces the quadratic cost associated with position.

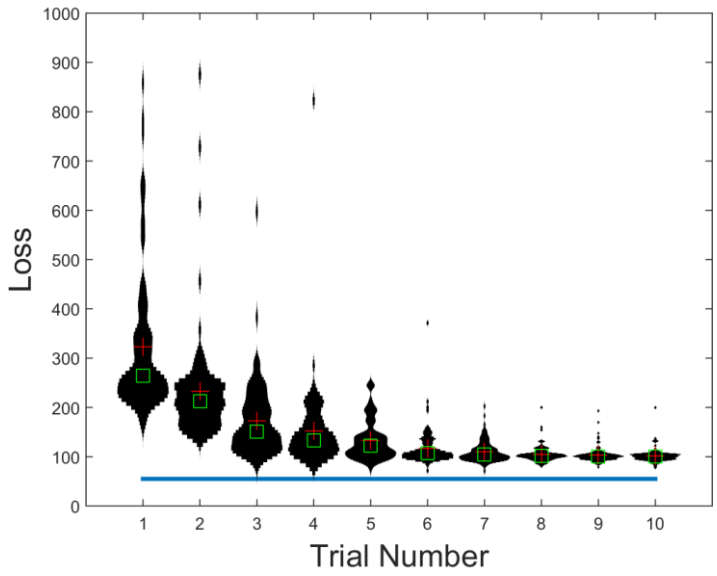

Figure 5. The distribution of the loss of 100 different agents on the particle in a box problem as a function of number of trials. In this test, the simulations were ran by using zeros for the control values for all subsequent time-steps in the simulation. The blue line denotes the loss of the mathematically available optimal trajectory, the red cross denotes the mean, while the green square denotes the median.

By introducing the policy readout unit we are no longer limited to a naive approach of repetition of the same control value or zero control value for all subsequent time-steps in the simulations. The results that this implementation produced are given in Figure 6. We can see a nice learning curve and compared to the previous solutions without the policy readout, we obtain a smaller loss. There still appears to be quite a bit of variability, which we believe is caused due to the random initialization of the reservoir. Some reservoirs are better suited to the task at hand purely by chance.

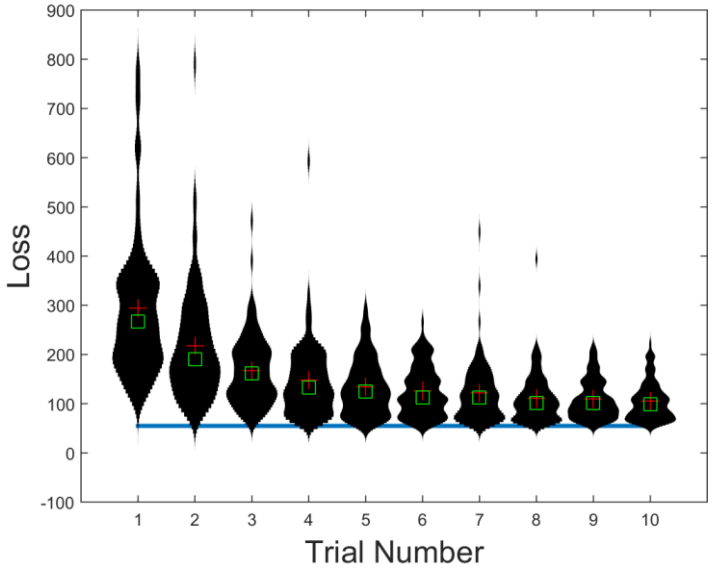

Figure 6. The distribution of the loss of 100 different agents on the particle in a box problem as a function of number of trials. In this test, the simulations were ran by using the control values suggested by the policy readout for all subsequent time-steps in the simulation. The blue line denotes the loss of the mathematically available optimal trajectory, the red cross denotes the mean, while the green square denotes the median.

Finally, we tried to use the reward readout to evaluate the reward of the simulations. Rather than fully simulating the entire trial all the way to the end as before, only 30 time-steps into the future are considered in the simulations. At that point, the reward readout is used to evaluate which simulation yields the highest reward. No clear learning curve can be seen and there was a huge variability in the performance where many agents diverged. Further analysis showed that this had to do with the cost function that was used for this task. In some agents, some of the trials would diverge from the target position of zero, rather than converging on this position. This resulted in large negative or positive values for the position. Due to the quadratic cost associated with position, this would result in the agent incurring huge losses. These losses would make the temporal difference learning algorithm diverge, which made the reward evaluations very poor, which in turn led to very poor overall performance. The usage of error clipping in our temporal difference algorithm, which means clipping the error value to make sure it stays within pre-specified bounds helped a little, but did not eliminate the problem completely.

We will now have a look at the tenth trial for the different implementations to see how they compare performance-wise. Because the performance on the particle in a box problem was so variable and many agents diverged for the architecture that included a reward readout,

we have decided to omit these results as to make the comparison between the other architectures more clear. Including the results of this architecture would results in a less than ideal y-axis scale for the results of the other three architectures.

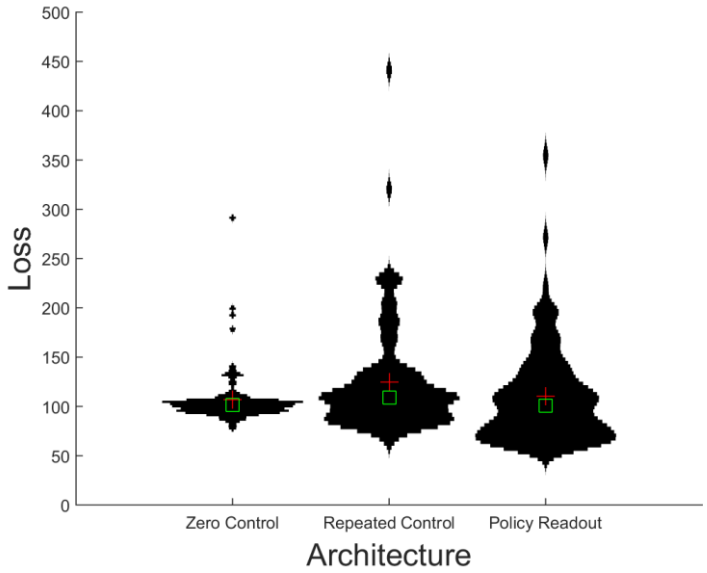

Figure 7. The distribution of the loss of 100 different agents on the particle in a box problem for the different architectures on the tenth trial. 'Zero Control' is the architecture that only did state prediction while using zero values for the control in the subsequent time-steps. 'Repeated Control' is similar as the previous architecture, but used repetition of the same control value through all time-steps. 'Policy Readout' is the architecture that included a policy readout that made suggestion for the control values at the subsequent time-steps in the simulation. The red cross denotes the mean, while the green square denotes the median.

The results are presented in Figure 7. As argued before, the fact that the architecture that only did state prediction and used a repetition of the same control values for all time-steps performed the worst makes sense. Since there is a quadratic cost associated, picking high control values in the simulations is very bad, since these are repeated for all subsequent time-steps, which incurs a heavy cost. This biases the agent to use very small control values, even though high control values at the beginning of a trial are very good. This is because high control values quickly ramp up the speed of the agent, which makes sure that the distance to a position of zero is reduced as quickly as possible, which is advantageous due to the quadratic cost associated with position.

The architecture that only did state prediction, but used zero value for the subsequent time-steps performs similar. The simulations are no longer punished with a high cost associated with high control values, since these values are only used once in the simulation.

This allows the agent to ramp up the speed as mentioned earlier. While it will incur a larger cost at the beginning of the trial, the agent will be at a position of zero earlier, and thus the total cost of the trial will be lower. While it did not bias towards a small control value as the previous architecture did, this second architecture is also not able to evaluate the choices of control values at time-steps in the simulation other than the first. This means it cannot take into account the increase in speed through the control at the other time-steps. Because of the quadratic cost associated with the control value, it is advantageous to accelerate with moderate control values over multiple time-steps rather than one big control value at just one time-step. But because there is also a quadratic cost associated with position, and the fact it cannot accurately plan the control values over multiple time-steps, this architecture tends to have a bias towards higher control values; The selection of high control values ensures the highest acceleration and thus speed, which minimizes the quadratic cost associated with position, but also leads to higher costs associated with the control.

The last architecture in this comparison is the architecture that also utilizes a policy readout. Because the policy readout is able to supply decent control values for the subsequent time-steps, it can more accurately simulate trajectories that are closer to an actual trajectory the agent should take as to minimize the cost. This makes for a more accurate comparison of the different control values, and thus a better control value gets selected. In Figure 7 we can see that this architecture performs similar to the previous two architectures on average. However, there is a larger variability in the performance due to the random different initializations of the reservoir. Importantly however, is that this architecture results in some of the best solutions as can be seen from the distributions.

4.2 Pendulum

We again first look at the results of 100 separate agents over 10 trials in which the same control value was repeated in every subsequent time-step in the simulation of the trajectories. These results are presented in Figure 8.

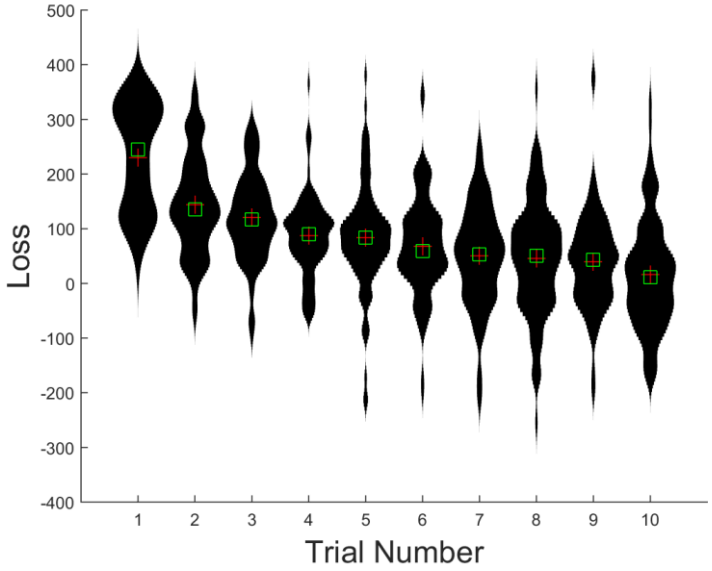

Figure 8. The distribution of the loss of 100 different agents on the pendulum problem as a function of number of trials. In this test, the simulations were ran by repetition of the control values for all subsequent time-steps in the simulation. The red cross denotes the mean, while the green square denotes the median.

Because the optimal solution is not known for this problem, we cannot compare the loss of the trajectory produced by the agent with the theoretical optimal trajectory. But we can see a learning curve. The loss starts off high and then drops as a function of trial. Interestingly, the results are highly variable even for later trials and we notice that the agent is often unable to keep the pendulum balanced at its inversion point. We believe this is due to the added Gaussian noise of the state of the pendulum, which makes the highly sensitive task of balancing the pendulum in the upright position extremely difficult. We noted far less variability and better performance in a separate simulation with no added noise. However, performance was still not as good as found in the literature (Thalmeier et al., 2015) and the balancing in an upright position still failed for some trials.

There are multiple possible reasons for this discrepancy. First off, in the aforementioned paper they have used a spiking implementation of a neural network. It could be possible that the extra information that such a model contains, such as spike timing (Memmesheimer, Rubin, Ölveczky, & Sompolinsky, 2014; Thalmeier et al., 2015) is crucial for this performance level. Second, our task setup might be different. Thalmeier et al (2015) did not provide which control values the agent was able to use. In this paper, the control values that were possible were not strong enough to get the pendulum in an upright position by just pushing it up. Rather, it had to be swung up through motions in both directions in

order to obtain enough velocity. While this should not make the balancing at the top more difficult, this fact does introduce the need for longer simulations. Short simulations would not chose to push the pendulum down, because it would have no way of knowing that this trajectory would result in a greater height once the pendulum swung up again at the other side. These necessary longer simulations in turn make the simulations less accurate, which could hamper the ability of the agent to balance the pole at its inversion point.

Figure 9 shows the performance when we use zeros in all the subsequent time steps rather than repeating the same control value over and over again. Since applying a force for just one time-step does not get the pendulum up in a meaningful way, the simulations are not very useful in determining what control value to use. This causes the simulations to be very similar to each other, which makes it difficult to select the correct control value to use. We see a weird learning curve where the loss first goes down and then back up, converging to a loss that is higher than in the previous solution.

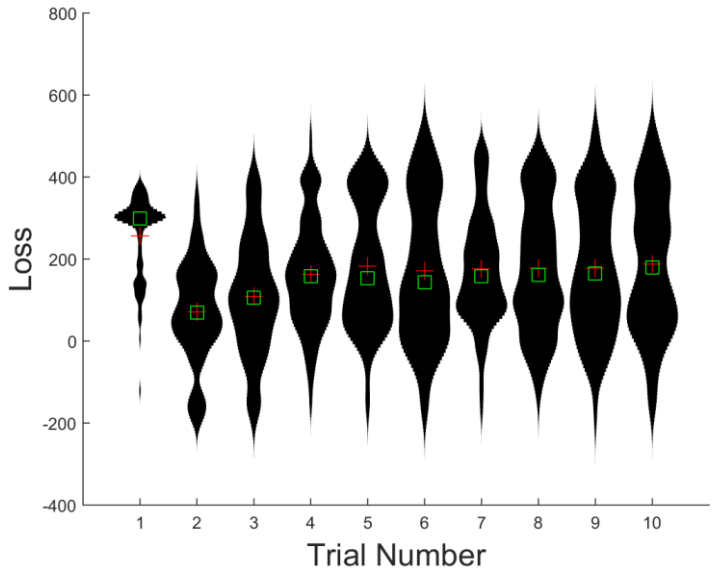

Figure 9. The distribution of the loss of 100 different agents on the pendulum problem as a function of number of trials. In this test, the simulations were ran by using zeros for the control values for all subsequent time-steps in the simulation. The red cross denotes the mean, while the green square denotes the median.

Figure 10 shows the performance of the architecture that has the added policy readout unit. There is a huge variability in the performance. Some of the agents perform very well with very low losses. However, the second thing that we notice is that the average

performance is actually worse compared to the original implementation without a policy readout (Figure 8). Theoretically this makes sense. It is true that the pendulum first has to be moved to one side, and then pushed to the other side to build up enough velocity to be able to reach the inversion point. But the first movement to one side is accompanied by an increasing height of the tip of the pendulum, and thus an increasing reward value. Only when pushing it further up is no longer possible, the movement has to be reversed to swing it up on the other side. But as long as the simulation length is long enough to see past the initial decrease of the reward by going down and to see the big increase towards the inversion point, this move will be made. The original implementation without the policy readout is thus able to solve the problem as is. Adding the policy readout adds another layer of complexity that is not needed for solving the task, and this added complexity deteriorates the performance.

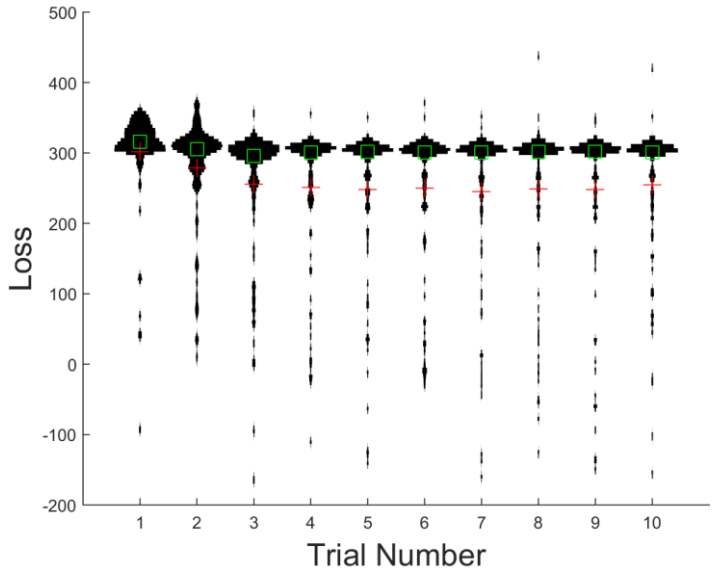

Figure 10. The distribution of the loss of 100 different agents on the pendulum problem as a function of number of trials. In this test, the simulations were ran by using the control values suggested by the policy readout for all subsequent time-steps in the simulation. The red cross denotes the mean, while the green square denotes the median.

Figure 11 shows the result of the architecture with the added reward readout and a simulation length of 30 time-steps compared to the 100 time-steps in the previous architectures. This architecture performs much better at this problem than it did on the particle in a box problem. We can clearly see a learning curve, as the loss goes down as a function of the number of trials. Compared to the previous working architectures on the pendulum problem without the reward readout, we do see that these results have a higher variability. We

had hoped that decreasing the number of time-steps that had to be simulated would improve the accuracy, and thus the performance. However, this turned out not to be the case. The increased complexity of this architecture results in similar performance, but higher variability.

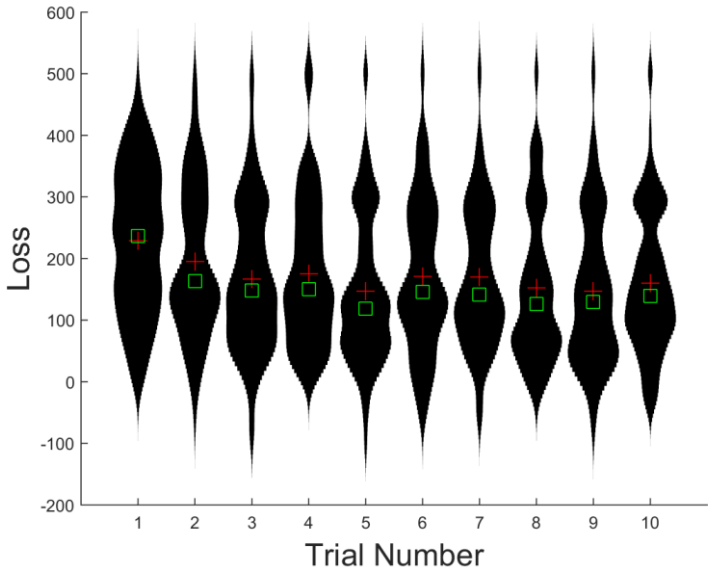

Figure 11. The distribution of the loss of 100 different agents on the pendulum problem as a function of number of trials. In this test, the simulations were ran by using the control values suggested by the policy readout for all subsequent time-steps in the simulation. Rather than calculating the reward over an entire simulation, the reward was approximated by the reward readout after 30 time-steps. The red cross denotes the mean, while the green square denotes the median.

The important observation however, is that we can see relatively decent performance while the simulations are just 30 time-steps compared to the 100 time-steps that were used with the previous architecture. Clearly this change in architecture is advantageous for the computation complexity.

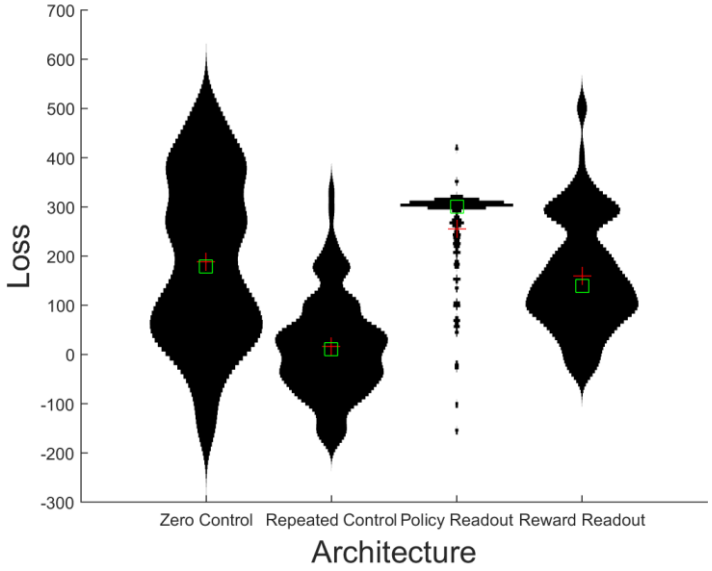

Figure 12. The distribution of the loss of 100 different agents on the pendulum problem for the different architectures on the tenth trial. 'Zero Control' is the architecture that only did state prediction while using zero values for the control in the subsequent time-steps. 'Repeated Control' is similar as the previous architecture, but used repetition of the same control value through all time-steps. 'Policy Readout' is the architecture that included a policy readout that made suggestion for the control values at the subsequent time-steps in the simulation. 'Reward Readout' is the architecture that only simulated for 30 time-steps and estimated the reward through the reward readout unit. The red cross denotes the mean, while the green square denotes the median.

In Figure 12 we have again presented a comparison of the different architectures on the tenth trials of the pendulum problem. The architecture that only did state prediction and used zero values for the control in the subsequent time-steps in the simulation performed worse than the one that used repetition of control values. This was due to the fact that a control value for just one time-step was not sufficient to push the pendulum up in any meaningful way. This caused all simulations to be very similar, making it difficult to select the best control value.

The architecture that did state prediction while using repetition of control values for the subsequent time-steps did much better. This implementation was similar to the one found in the literature (Thalmeier et al., 2015), except for the difference that we used online learning rather than a separate training phase. A simple repetition of the same control values throughout the entire simulation did not hamper the simulation, since this is actually needed in order to build up enough momentum to reach an upright position.

The architecture that included a policy readout performed worse than the previous simple architecture. This has to do with the fact that the repetition of the same control values

is not a very bad strategy at all. In order to be able to push the pendulum up, repetition of the same control value is actually needed, since a push for just a single time-step is not sufficient to get the pendulum in an upright position. This means that the addition of a policy readout makes the architecture more complex without a clear advantage in this case. This increase of complexity is probably the reason for the regression of the performance.

The final architecture is the architecture that also included a reward readout. Performance is better than the third, although it was still worse than the second architecture. The shorter simulation length allowed for a higher simulation accuracy than the third architecture, which improved the performance. On top of this, this shorter simulation length of just 30 time-steps rather than the 100 that the other three architectures needed is advantageous for the computation complexity.

4.3 Mountain Car

We first evaluated the results of 100 agents running through 10 trials in which the same control value was repeated in every subsequent time-step in the simulation of the trajectories. As we can see in Figure 13 the performance is around 500 steps, which means it is actually able to solve the task, although at a large number of steps. At first these results puzzled us, however a separate simulation showed that using a control value of 1 (acceleration forward) at every single time-step resulted in increasingly large oscillations of the x-position as the car until the car escaped. The architecture is unable to come up with a better solution, as its simulations cannot simulate a trajectory in which the car first backs off, and then drives forward; exactly the kind of trajectory that is needed to reach the top of the mountain in a small number of steps.

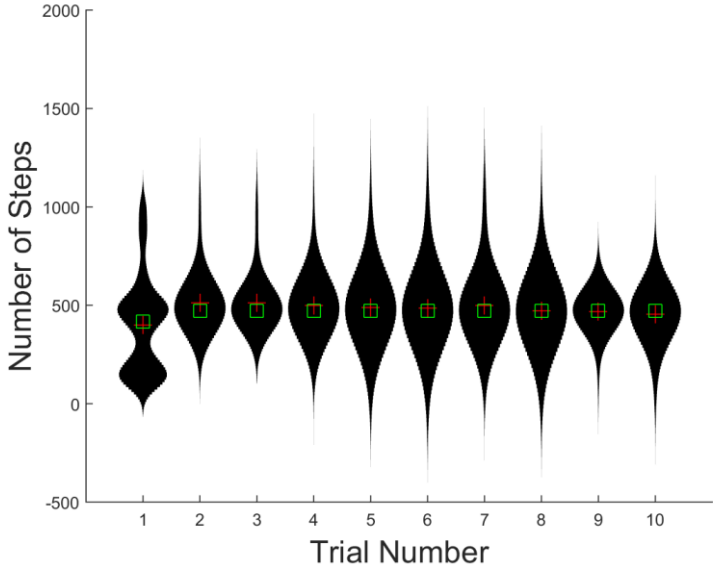

Figure 13. The distribution of the number of steps required to complete the mountain car problem for 100 different agents as a function of number of trials. A maximum of 1000 time-steps were given, such that failed trials were counted as 1000 steps. In this test, the simulations were ran by repetition of the control values for all subsequent time-steps in the simulation. The red cross denotes the mean, while the green square denotes the median.

We will now show the results when the subsequent time-steps used zero control values in the simulations. As mentioned before, we did not expect this implementation to perform well given the fact that this also will not allow any simulations that use backing off and then driving forward. Figure 15 confirms our hypothesis and we again see very poor performance. The agent is unable to escape the valley most of the trials given a median equal to the maximum number of steps. Since only one time-step utilizes a non-zero control value, the simulations are very similar to each other which makes it difficult for the agent to select the best control value. However, since it now only uses a control value at the first time-step and then uses control values of zero for the subsequent time-steps, it can now select control values of -1. This implementation also sees oscillations of the x-position as the previous one did, but now it can accelerate the oscillations in the reverse direction as well, by selecting a control value of -1. This causes some trials to perform remarkably well, although it is not very consistent.

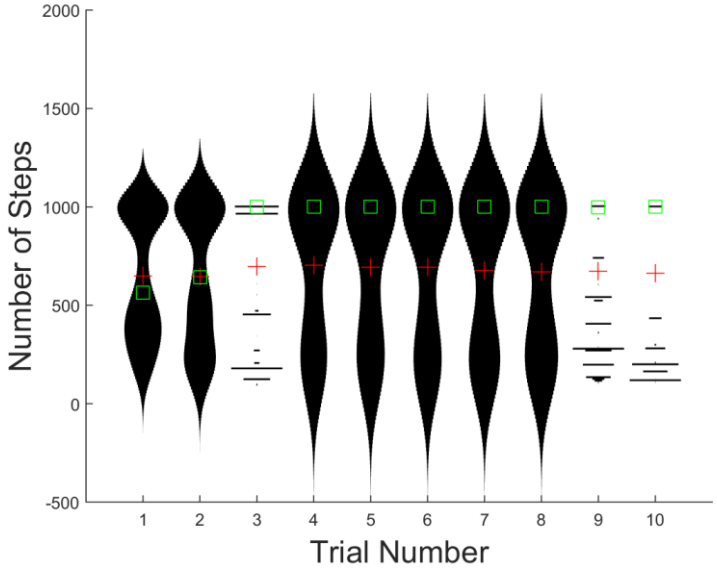

Figure 14. The distribution of the number of steps required to complete the mountain car problem for 100 different agents as a function of number of trials. A maximum of 1000 time-steps were given, such that failed trials were counted as 1000 steps. In this test, the simulations were ran by using zeros for the control values for all subsequent time-steps in the simulation. The red cross denotes the mean, while the green square denotes the median.

As we can see in Figure 15, the addition of a policy readout allows the agent to plan a backward and then forward trajectory in its simulation and thus we now finally see a learning curve in the mountain car problem. However, the variability of the performance is very high even for later trials. While some initializations of the reservoir are able to successfully complete the mountain car task, some reservoirs do very poorly and are unable to learn the task correctly.

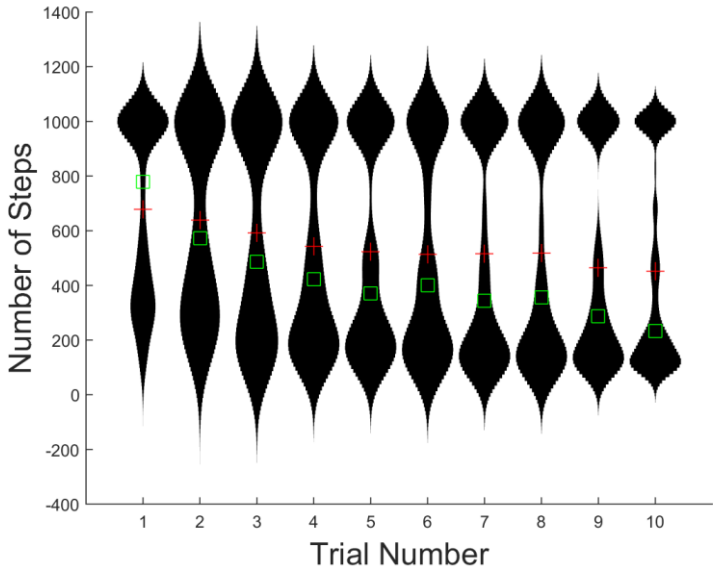

Figure 15. The distribution of the number of steps required to complete the mountain car problem for 100 different agents as a function of number of trials. A maximum of 1000 time-steps were given, such that failed trials were counted as 1000 steps. In this test, the simulations were ran by using the control values suggested by the policy readout for all subsequent time-steps in the simulation. The red cross denotes the mean, while the green square denotes the median.

Adding the reward readout results in a large increase in variability, as can be seen in Figure 17. The important observation is that while only 30 time-steps are simulated, which is insufficient to simulate the entire trajectory, the agent still manages to reach the goal. This is again advantageous for the computation complexity.

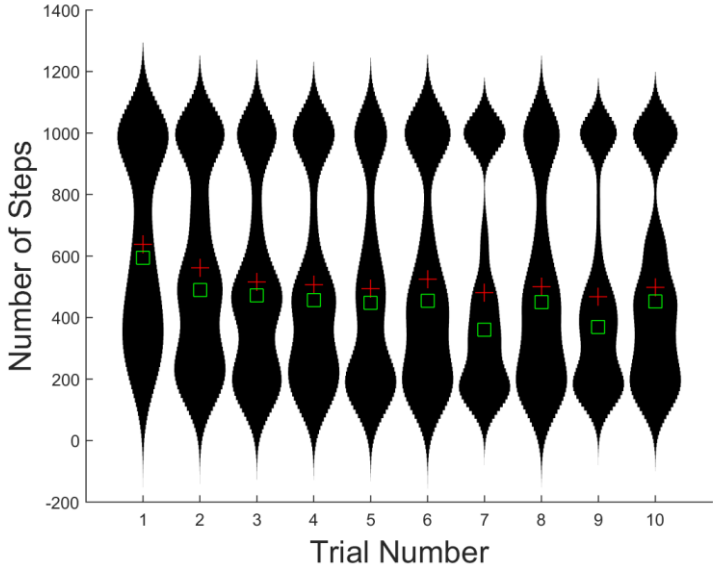

Figure 16. The distribution of the number of steps required to complete the mountain car problem for 100 different agents as a function of number of trials. A maximum of 1000 time-steps were given, such that failed trials were counted as 1000 steps. In this test, the simulations were ran by using the control values suggested by the policy readout for all subsequent time-steps in the simulation. Rather than calculating the reward over an entire simulation, the reward was approximated by the reward readout after 30 time-steps. The red cross denotes the mean, while the green square denotes the median.

In Figure 17 we have again presented a comparison between the four architectures on the mountain car problem on the tenth trial. As we can see, both architectures that only incorporate state prediction perform very poorly. The simulations cannot perform the backward and then forward movement that is needed to be able to escape the valley, because this requires control values of opposite sign at the different stages of the trajectory simulation. Both using zero values for the subsequent time-steps and repeating the same control values are thus not appropriate to generate a simulation with a good solution. The average performance of both is similar to simply using a control value of one for all time-steps.

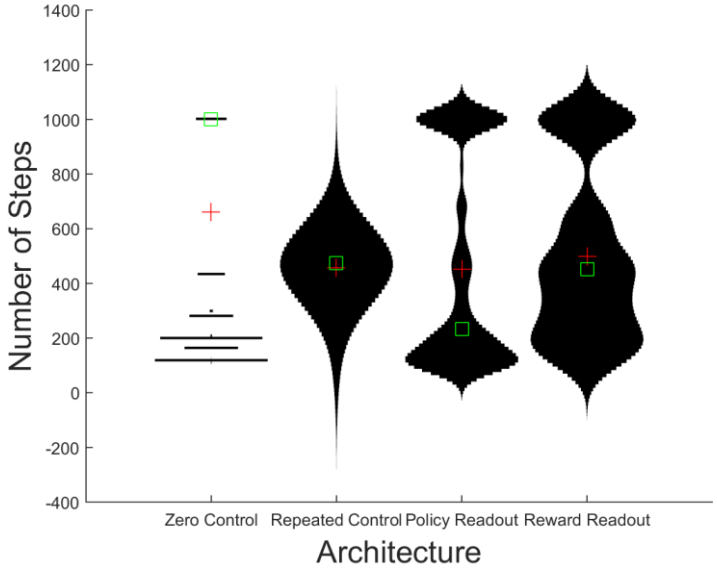

Figure 17. The distribution of the number of steps required to complete the mountain car problem for 100 different agents for the different architectures on the tenth trial. 'Zero Control' is the architecture that only did state prediction while using zero values for the control in the subsequent time-steps. 'Repeated Control' is similar as the previous architecture, but used repetition of the same control value through all time-steps. 'Policy Readout' is the architecture that included a policy readout that made suggestion for the control values at the subsequent time-steps in the simulation. 'Reward Readout' is the architecture that only simulated for 30 time-steps and estimated the reward through the reward readout unit. The red cross denotes the mean, while the green square denotes the median.

This third architecture performs much better in this regard. Since we now have a policy readout supplying control values for the subsequent time-steps in the simulation, we are no longer bound to the simple zero or repetition scheme. This allows it to simulate complex trajectories that can build velocity by first going to the left and then using this velocity to escape the valley on the right. These much better simulations can then successfully used to guide the agent to the most appropriate control value to use at a particular time-step. The most interesting observation is the very clear bimodal distribution that we have. Some of the reservoirs manage to perform this task with a very little amount of steps, while others fail and are at the maximum of 1000 steps. We hypothesize that this has to do with the fact that some random initializations of the reservoirs are better at performing a certain tasks than others.

The fourth architecture shows worse performance to the third architecture, but rather than simulating 100 time-steps, which are needed to be able to simulate the full back and forward motion, it can now perform the same task with just 30 time-steps. This provides a clear advantage to this architecture from a computational standpoint. While the average

performance is again very close to the 500 steps that a simple control value of one for all time-steps implementation produces, we see a very large variability. We believe this is due to the complexity of this architecture. However, some agents perform very well and we see that the distribution extends to a very low number of steps. The best agents utilizing the reward readout perform as well as the best performing agents that only use the policy readout.

5 Discussion

In this paper we tried to answer the question: "Can we utilize FORCE learning to learn adaptive behavior in a dynamic environment?" We saw that this is indeed possible for the toy problems that were presented here. In contrast to traditional reinforcement learning algorithms (Sutton & Barto, 1998), performance is already decent even from the very first trial and is improved on, and quickly converges. In the case of the particle in a box, where the mathematically optimal solution is available, we can see the best version of our FORCE-RL framework to converge to a solution close to this. This decent performance on the very first trial which his then improved on is much more similar to how real organisms learn. The balance between exploration and exploitation that traditional reinforcement learning algorithms need to find (Ishii et al., 2002; Sutton & Barto, 1998; Thrun, 1992) is also sidestepped in this implementation since exploration can be done in the mental simulations, similar to the pre-play that is found in rats (Meer & Redish, 2010; Pfeiffer & Foster, 2013).

State simulation alone, however, is not always sufficient to produce an agent that can learn to solve these toy problems. Our earlier implementation of our FORCE-RL framework that only used state prediction was unable to find a solution in the mountain car problem and suboptimal solution for the particle in a box problem. For the mountain car problem, the repetition of the same control values or using zeros in all subsequent time-steps did not allow for simulations which had the required first moving to a position with a lower reward, and then using the buildup velocity to reach the goal state with a high reward. The introduction of our policy readout unit alleviated this problem by providing control values at all time-steps in the simulation. This policy readout unit was also trained in an online fashion similar to how animals and humans would learn, preventing the need of separate training and test trials.

Research in simulation of future states has been conducted by Oh, Guo, Lee, Lewis, & Singh (2015). Here, simulations were used to evaluate which actions lead to the most novel state as compared to the memories of previous states. This information is then used to guide

the exploration process by visiting these novel states and then learning from these states. In contrast, we evaluate these simulated states on their reward/loss value immediately and then chose an action that resulted in a simulation with the highest reward.

That also means our learning is on policy. On policy learning means that the policy that is used to select which actions to take is also used to learn from the environment (Sutton & Barto, 1998), which is in contrast to a separate exploration process. This means that our reinforcement learning algorithm does not satisfy the condition for the so-called deadly triad in reinforcement learning (Rich Sutton, 2015). The deadly triad states that a reinforcement algorithm might experience divergence rather than convergence when using function approximation, bootstrapping and off-policy learning at the same time. While our algorithm uses function approximation (the neural networks) and bootstrapping (the policy readout learns control estimates from estimates by the simulations), it does not use off-policy learning and thus avoids this deadly triad.

Our final architecture in our FORCE-RL framework is also very similar to the architecture that was used to produce an agent that was able to beat the world champion at a game of Go (Silver et al., 2016). The state prediction readout was used to simulate future states, similar to how AlphaGo used a tree-search algorithm to simulate future states conditioned on an action. The policy readout was used to constrain to search breadth in our simulations, similar to AlphaGo's policy network. And finally, both implementations used a value network in the case of AlphaGo and a reward readout in our case, to constrain the depth of the simulation. Of course in both cases, simulation of future states would not be needed if the value prediction were perfect. But AlphaGo showed that this is not the case, and that performance is improved by using simulation in tandem with value evaluation. Furthermore, this reward readout allows our FORCE-RL framework to be used on infinite horizon problems (Bertsekas, 1995) and can be helpful in limiting the computational complexity of the simulations by constraining the depth of the simulations.

While the comparison of this simulation with pre-play is one good example of similarities between our reinforcement learning model and real life adaptive behavior, we also see a lot of other phenomenon arise that are also found in real organisms. The prediction by the State Network about the effect of the agent's actions could be interpreted as the efferent copy (Angel, 1976; Grüsser, 2010). And where most traditional reinforcement learning algorithms do not work with continuous control values (Mnih et al., 2013, 2015; Sutton & Barto, 1998) (but, see (Lillicrap et al., 2015) for a counterexample), real world behaviors such

as grasping motions are continuous in control. Continuous control values do not pose a problem for this reinforcement learning framework with the FORCE algorithm (Sussillo & Abbott, 2009b).

ESNs deliver a simple and effective way for training recurrent neural networks (Herbert Jaeger, 2010; Lukoševičius, 2012; Millea, 2014) and since the reservoir is left untrained, the same circuits can be used for different tasks by utilizing a different readout unit. This is an efficient way to reuse existing neuronal circuits and this concept might also be used in real brains (Hoerzer, Legenstein, & Maass, 2014; Maass et al., 2002). FORCE learning gives us an interesting extension of this idea by making learning online compared to the regular way of offline training of echo state networks by linear regression. It also more closely mimics the way real organisms learn by keeping the errors small, even from the start (Sussillo & Abbott, 2009).

The way these networks learn from and interact with the environment also adheres to the perception-action cycle (Tishby & Polani, 2011). The environment affects the sensors of the agent with the added noise (the input for both networks). This information is then stored in memory, which is provided by the recurrent connections. This information is processed in the network by simulations of trajectory, which eventually leads to a control value, which is the action that is used. This action then changes the state of the agent, which is again picked up by the agent's sensors, closing the cycle.

In these toy problems we have only used the state descriptions of the current state. Organisms however have to deal with high-dimensional input, such as visual input. For our model to be an accurate description, it should be able to deal with high dimensional inputs as well. Future research should look into the possibility for these ESNs to deal with high dimensional input such as renderings of the environment (Lillicrap et al., 2015; Todorov, Erez, & Tassa, 2012).

Because traditional reinforcement learning algorithms learn to map states to actions (Sutton & Barto, 1998), this forces them to restart training when the goal is changed. But since our algorithm learns to predict future states, which is goal independent, it should generalize to other goals as well. Only the policy and reward readouts need to be retrained, but with help of the state readout the agent should show acceptable performance even from the beginning. In fact, separate reward and policy readout units could be used for different tasks, so that switching to another known tasks only requires the usage of different readout units rather than retraining. This hypothesized ability to generalize could be an interesting

future research goal, because it is closer to the way humans show adaptive behavior as humans are excellent in generalizing to novel tasks.

Lastly, since we noticed a very large variability in performance of the agents trying to perform our tasks, it would be interesting to research if this variability can be reduced by making sure that the reservoir that is used is suited for the task at hand. While back-propagation through time is not biologically very plausible, training the recurrent connections through algorithms such as (anti-)Hebbian learning might be a nice goal to pursuit. Furthermore, while feedforward networks or recurrent networks that are trained via back-propagation through time are not biologically very plausible, they are arguably more powerful than these simpler reservoir models. It would be interesting to see whether this way of on-line learning be also be used with different function approximators to obtain state-of-the-art performance.

Given the fact that our learning is similar to how it is observed in the real adaptive behavior, and combined with the observed phenomena that emerged that closely mimic phenomena also found in real organisms, we believe our model to be a plausible description of adaptive behavior as seen in real organisms. Future research could help strengthen this model as an accurate description of adaptive behavior.

6 References

Amit, D. J., & Brunel, N. (1997). Model of global spontaneous activity and local structured activity during delay periods in the cerebral cortex. *Cerebral Cortex*, *7*(3), 237–252. http://doi.org/10.1093/cercor/7.3.237

Anderson, B. D. O., & Moore, J. B. (2007). *Optimal Control: Linear Quadratic Methods*. Courier Corporation.

Angel, R. W. (1976). Efference copy in the control of movement. *Neurology*, *26*(12), 1164–1164. http://doi.org/10.1212/WNL.26.12.1164

Bargh, J. A. (1994). The four horsemen of automaticity: Intention, awareness, efficiency, and control as separate issues.

Bargh, J. A. (1996). Principles of automaticity.

Bertsekas, D. P. (1995). *Dynamic programming and optimal control* (Vol. 1, No. 2). Belmont, MA: Athena Scientific.

Bottou, L. (1998). Online learning and stochastic approximations. *On-Line Learning in*

Neural Networks, 17(9), 142.

Brunel, N. (2000). Dynamics of Networks of Randomly Connected Excitatory and Inhibitory Spiking Neurons. *Journal of Physiology-Paris*, 94(5), 445-463.

Chiel, H. J., & Beer, R. D. (1997). The Brain Has a Body: Adaptive Behavior Emerges from Interactions of Nervous System, Body and Environment. *Trends in Neurosciences*, *20*(12), 553–557. http://doi.org/10.1016/S0166-2236(97)01149-1

Faisal, A. A., Selen, L. P. J., & Wolpert, D. M. (2008). Noise in the Nervous System. *Nature Reviews. Neuroscience*, *9*(4), 292–303. http://doi.org/10.1038/nrn2258

Flokentin, J. J. (1962). Partial Observability and Optimal Control. *International Journal of Electronics and Control*, *13*(3), 263–279. http://doi.org/10.1080/00207216208937438

Grüsser, O.-J. (2010). Early concepts on efference copy and reafference. *Behavioral and Brain Sciences*, *17*(2), 262-265. http://doi.org/10.1017/S0140525X00034415

Hausknecht, M., & Stone, P. (2015). Deep Recurrent Q-Learning for Partially Observable MDPs. *arXiv*.

Haykin, S. (2008). Adaptive filter theory. *Pearson Education India.*

Hoerzer, G. M., Legenstein, R., & Maass, W. (2014). Emergence of Complex Computational Structures From Chaotic Neural Networks Through Reward-Modulated Hebbian Learning. *Cerebral Cortex*, *24*(3), 677–690. http://doi.org/10.1093/cercor/bhs348

Ishii, S., Yoshida, W., & Yoshimoto, J. (2002). Control of exploitation–exploration meta-parameter in reinforcement learning. *Neural Networks*, 15(4), 665-687.

Jaeger, H. (2004). Harnessing Nonlinearity: Predicting Chaotic Systems and Saving Energy in Wireless Communication. *Science*, *304*(5667), 78–80. http://doi.org/10.1126/science.1091277

Jaeger, H. (2010). The " echo state " approach to analysing and training recurrent neural networks – with an Erratum note. *GMD Report*, 148, 34.

Kirk, D. E. (2012). *Optimal Control Theory: An Introduction*. Courier Corporation.

Koryakin, D., Lohmann, J., & Butz, M. V. (2012). Balanced echo state networks. *Neural Networks*, *36*, 35–45. http://doi.org/10.1016/j.neunet.2012.08.008

Lillicrap, T. P., Hunt, J. J., Pritzel, A., Heess, N., Erez, T., Tassa, Y., … Wierstra, D. (2015). Continuous control with deep reinforcement learning. *arXiv*.

Lukoševičius, M. (2012). A practical guide to applying echo state networks. *Neural Networks: Tricks of the Trade* (pp. 659-686).

Maass, W., Natschläger, T., & Markram, H. (2002). Real-time computing without stable states: a new framework for neural computation based on perturbations. *Neural*

Computation, *14*(11), 2531–2560. http://doi.org/10.1162/089976602760407955

Marquardt, D. W. (1970). Generalized inverses, ridge regression, biased linear estimation, and nonlinear estimation. *Technometrics*, *12*(3), 591–612. http://doi.org/10.1080/00401706.1970.10488699

Meer, M. Van Der, & Redish, A. (2010). Expectancies in decision making, reinforcement learning, and ventral striatum. *Frontiers in Neuroscience*, 3, 6.

Memmesheimer, R. M., Rubin, R., Ölveczky, B. P., & Sompolinsky, H. (2014). Learning Precisely Timed Spikes. *Neuron*, *82*(4), 925–938. http://doi.org/10.1016/j.neuron.2014.03.026

Micheyl, C., McDermott, J. H., & Oxenham, A. J. (2009). Sensory noise explains auditory frequency discrimination learning induced by training with identical stimuli. *Attention, Perception & Psychophysics*, *71*(1), 5–7. http://doi.org/10.3758/APP.71.1.5

Millea, A. (2014). Explorations in Echo State Networks.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., & Riedmiller, M. (2013). Playing Atari with Deep Reinforcement Learning. *arXiv*, 1–9.

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A. a, Veness, J., Bellemare, M. G., … Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, *518*(7540), 529–533. http://doi.org/10.1038/nature14236

Moors, A., & Houwer, J. De. (2006). Automaticity: a theoretical and conceptual analysis. *Psychological Bulletin*, 132(2), 297.

Mortensen, U., & Suhl, U. (1991). An evaluation of sensory noise in the human visual system. *Biological Cybernetics*, *66*(1), 37–47. http://doi.org/10.1007/BF00196451

Oh, J., Guo, X., Lee, H., Lewis, R. L., & Singh, S. (2015). Action-Conditional Video Prediction using Deep Networks in Atari Games. In *Advances in Neural Information Processing Systems* (pp. 2845–2853).

Pfeiffer, B., & Foster, D. (2013). Hippocampal place-cell sequences depict future paths to remembered goals. *Nature*, 497(7447), 74-79.

Silver, D., Huang, A., Maddison, C. J., Guez, A., Sifre, L., van den Driessche, G., … Hassabis, D. (2016). Mastering the game of Go with deep neural networks and tree search. *Nature*, *529*(7587), 484–489. http://doi.org/10.1038/nature16961

Singh, S., & Sutton, R. S(1996). Reinforcement learning with replacing eligibility traces. *Machine Learning*, 22(1-3), 123-158.

Sompolinsky, H., & Crisanti, A. (1988). Chaos in Random Neural Networks. *Physical Review Letters*, 61(3), 259.

Süli, E., & Mayers, D. (2003). *An introduction to numerical analysis*. Cambridge University Press.

Sussillo, D., & Abbott, L. F. (2009). Generating Coherent Patterns of Activity from Chaotic Neural Networks. *Neuron*, *63*(4), 544–557. http://doi.org/10.1016/j.neuron.2009.07.018

Sutton, R. S. (1984). Temporal credit assignment in reinforcement learning.

Sutton, R. S. (2015). Introduction to Reinforcement Learning with Function Approximation. *NIPS 2015 Conference.*

Sutton, R. S., & Barto, A. (1998). *Reinforcement learning: An introduction* (Vol. 1, No. 1). Cambridge: MIT Press.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, *3*(1), 9–44. http://doi.org/10.1007/BF00115009

Sutton, R. S., Barto, a G., & Williams, R. J. (1992). Reinforcement learning is direct adaptive optimal control. *IEEE Control Systems*, 12(2), 19-22, http://doi.org/10.1109/37.126844

Sutton, R. S., & Tanner, B. (2005). Temporal-Difference Networks. *arXiv.*

Tanner, B., & Sutton, R. S. (2005). TD(λ) networks: Temporal-difference networks with eligibility traces. *Proceedings of the 22nd International Conference on Machine Learning (ICML)*, 889–896. http://doi.org/10.1145/1102351.1102463

Tesauro, G. (1995). Temporal difference learning and TD-Gammon. *Communications of the ACM*, 38(3), 58-68.

Thalmeier, D., Uhlmann, M., Kappen, H. J., & Memmesheimer, R.-M. (2015). Learning universal computations with spikes. *arXiv*.

Thrun, S. (1992). Efficient exploration in reinforcement learning.

Tishby, N., & Polani, D. (2011). The Information Theory of Decision and Action. In *Perception Action Cycle* (pp. 601-636), Springer New York.

Todorov, E. (2006). Optimal Control Theory. *Bayesian Brain: Probabilistic Approaches to Neural Coding*, 269-298.

Todorov, E., Erez, T., & Tassa, Y. (2012). MuJoCo: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems* (p. 5026-5033). IEEE.

van Vreeswijk, C., & Sompolinsky, H. (1996). Chaos in Neuronal Networks with Balanced Excitatory and Inhibitory Activity. *Science*, 274(5293), 1724.

Wulf, G., McNevin, N., & Shea, C. (2001). The automaticity of complex motor skill learning as a function of attentional focus. *The Quarterly Journal of Experimental Psychology*: *Section A*, 54(4), 1143-1154.