Deep Reinforcement Learning and Self Instructing Database

Debjyoti Paul
University of Utah
deb@cs.utah.edu

INTRODUCTION

Reinforcement learning (RL) is the area of machine learning that deals with *sequential decision making*. Sequential decision-making is a task of deciding, the sequence of actions to perform in an uncertain environment in order to achieve some goals. Sequential decision-making tasks covers a wide range of possible applications with the potential to impact many domains, such as robotics, healthcare, smart grids, finance, self-driving cars, and many more.

Inspired by behavioral psychology (e.g. Sutton [21]) reinforcement learning (RL) proposes a formal framework to this problem. The main idea is that an artificial agent may learn by interacting with its environment, similarly to a biological agent. I will dive more into the details of how a general RL problem is formally represented and what are the key components involved in the framework in Section 1.

1 PART A

In reinforcement learning there is no supervisor but a *reward* signal to guide the learning process. Even the feedback is delayed and not instantaneous which sets it as a different paradigm from other machine learning methods. In a sequential decision making process an/the *agent* gets to pick an action at every time step which tries to optimize the future cumulative *reward* signal. Traditional Reinforcement Learning requires explicit design of state space and action space, while the mapping from state space to action space is learned [22]. That makes the problem spaces and the possible states in an environment very limited, only with fully observable state of environment for agent. Neural networks enhances RL with the capability to make the agent learn a state abstraction and a policy approximation directly from its input data in a partially observable environment. Deep RL is the fruitful outcome of this attempt of RL enhancement.

1.1 Formal RL Framework:

The general RL problem is formalized as a discrete time stochastic control process where an agent interacts with its environment in the following way: the agent starts, in a given state within its environment $s_0 \in \mathcal{S}$, by gathering an initial observation $\omega_0 \in \Omega$. At each time step t, the agent has to take an action $a_t \in \mathcal{A}$. As illustrated in Figure 1 it follows three consequences: (i) the agent obtains a reward $r_t \in \mathcal{R}$, (ii) the state transitions to $s_{t+1} \in \mathcal{S}$, and (iii) the agent obtains an observation $\omega_{t+1} \in \Omega$. This control setting was first proposed by [3] and later extended to learning by Barto [2]. The goal of RL is to *select actions to maximise total future reward.*

Definition 1. Reward: A reward r_t is a scalar feedback signal that indicates how well *agent* is doing at step t.

University of Utah, Salt Lake City, USA.

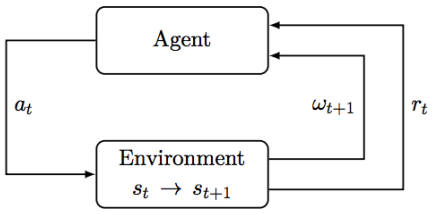

Figure 1: Agent-environment interaction in RL.

An example of *reward* in the case of flying toy robot helicopters are : (a) +ve reward for following desired trajectory, (b) fk??ve reward for crashing.

Definition 2. History: The history $\mathcal{H}_t$ is the sequence of observations, actions, rewards, i.e. all observable variables up to time t. Mathematically, $\mathcal{H}_t = \omega_1, r_1, a_1, \ldots, a_{t-1}, \omega_t, r_t$.

Formally, state $s_t \in \mathcal{S}$ is a function of the history $\mathcal{H}_t$.

$$s_t = f(\mathcal{H}_t)$$

Environment and agent both can have their own state which we define as s_t^e and s_t^a respectively. The environment state is usually not visible to the agent.

In a fully observable system, the agent directly observes environment state i.e. $\omega_t = s_t^a = s_t^e$. Formally this is called as Markov Decision Process (MDP). The Markov property means that the future of the process only depends on the current observation and the agent has no interest in looking at the full history.

Markov Decision Process (MDP):
An MDP is a 5-tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \gamma)$ where

$\mathcal{S}$ is the state space,
$\mathcal{A}$ is the action space,
$T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition function (set of conditional transition probabilities between states),
R: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$ is the reward function, where R is a continuous set of possible rewards in a range $R_{max} \in \mathbb{R}^+$ (e.g., $[0, R_{max}]$),
$\gamma \in [0, 1)$ is the discount factor.

However, in the real world not many systems are fully observable, an *agent* indirectly observes the environment without knowing it's internal information state. In this scenario $s_t^a \neq s_t^e$.

Formally this is a partially observable Markov decision process (POMDP) illustrated in Figure 2.

Partially Observable Markov Decision Process (POMDP): A POMDP is a 7-tuple $(\mathcal{S}, \mathcal{A}, T, \mathcal{R}, \Omega, O, \gamma)$ where

$\mathcal{S}$ is the state space,
$\mathcal{A}$ is the action space,
$T : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow [0, 1]$ is the transition function (set of conditional transition probabilities between states),

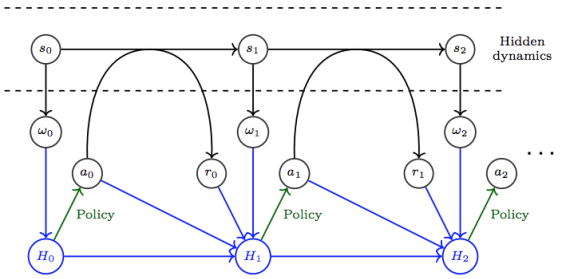

Figure 2: Partial Observed Markov Decision Process (POMDP).

R: $\mathcal{S} \times \mathcal{A} \times \mathcal{S} \rightarrow \mathcal{R}$ is the reward function, where R is a continuous set of possible rewards in a range $R_{max} \in \mathbb{R}^+$ (e.g., $[0, R_{max}]$),
Ω is a finite set of observations $1, ..., N_\Omega$,
$O : \mathcal{S} \times \Omega \rightarrow [0, 1]$ is a set of conditional observation probabilities,
$\gamma \in [0, 1)$ is the discount factor.

In a POMDP *agent* must construct its own state representation s_t^a. For example,
(i) **Complete history:** $s_t^a = \mathcal{H}_t$.
(ii) ***Beliefs* of environment state:** $s_t^a = \mathbb{P}[s_t^e = s^1], \ldots, \mathbb{P}[s_t^e = s^n]$ where s^k represents a state in environment.
(iii) **Recurrent Neural Network (RNN)**: $s_t^a = \sigma(s_{t-1}^a W_s + \omega_t W_o)$ where W_s and W_o represents weight vectors and σ some non linear function. [11, 12, 23].

This RNN approach to solving POMDPs is related to other problems using dynamical systems and state space models, where the true state can only be estimated [5].

1.2 RL Agent Components:

So far, we have introduced the key formalism used in RL, the MDP and the POMDP. Now I will talk about what is inside an RL agent and the components it uses for learning. An RL agent may include one or more of the following component for learning.

Value functions: Value function attempts to measure goodness/badness of each state and/or action. In other words, value function methods are based on estimating the value (expected return) of being in a given state i.e. future reward. The state-value function $V_\pi(s)$ is the expected return when starting in state s and following π henceforth:

$$V_\pi(s) = \mathbb{E}_\pi[R_t + \gamma R_{t+1} + \gamma^2 R_{t+2} + \ldots | s_t = s]$$

where $\gamma \in [0, 1]$ is a discount parameter, that governs the future sightedness of the model.

Policy Search: A policy is the *agent*'s behavior. It is a map from state to action. Examples of some policies are:
(i) *Deterministic Policy:* This is the simplest policy, here an action a is mapped to a state s i.e. $a = \pi(s)$.
(ii) *Stochastic Policy:* A policy can be stochastic based on probability of state observation, i.e. $\pi(a|s) = \mathbb{P}[\mathcal{A} = a|\mathcal{S} = s]$.
(iii) *Parameterized Policy:* A parameterised policy π_θ is chosen, whose parameters are updated to maximise the expected return $E[R|\theta]$ using either gradient-based or gradient-free optimisation [7].

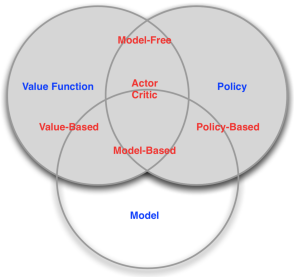

Figure 3: RL Agent Taxonomy.

Neural networks that encode policies have been successfully trained using both gradient-free [6, 9, 15] and gradient based [13, 16, 24] methods.

Model: A model tries to learn the behavior of the environment. First, it tries to predict what the environment will do next, that is predicting next state and is called Transitions ($\mathcal{P}$). Secondly, the model tries to predict the expected next reward and is called Rewards model ($\mathcal{R}$).

$$\mathcal{P}_{ss'}^a = \mathbb{P}[\mathcal{S}' = s'|\mathcal{S} = s, \mathcal{A} = a]$$

where s is the state prior state and s' is the resultant state on action a.

$$\mathcal{R}_s^a = \mathbb{E}[r|\mathcal{S} = s, \mathcal{A} = a]$$

where s is the state prior state and r is the reward for next step on action a.

1.3 RL Agent Taxonomy:

Model methods are optional in RL systems. In fact there are many model free based methods applied for real world problems. To understand a comprehensive taxonomies of categorizing RL Agent we present it in Figure 3.

RL focus on learning without access to the underlying model of the environment. However, interactions with the environment could be used to learn value functions, policies, and also a model. Model-free RL methods learn directly from interactions with the environment, but model-based RL methods can simulate transitions using the learned model, resulting in increased sample efficiency [1]. This is particularly important in domains where each interaction with the environment is expensive. However, learning a model introduces extra complexities, and there is always the danger of suffering from model errors, which in turn affects the learned policy; a common but partial solution in this latter scenario is to use model predictive control, where planning is repeated after small sequences of actions in the real environment [5]. Although deep neural networks can potentially produce a very complex and rich models [8, 18], sometimes simpler, more data efficient methods are preferable [10].

1.4 Deep Reinforcement Learning (DRL):

Traditional RL works was mainly on low dimensional problems (i.e. few state space, limited actions etc.). Integration of deep neural network with RL framework bolster the framework by converting higher dimensional problems into low dimensional representations. Initial DRL works was mainly involved on scaling up prior work

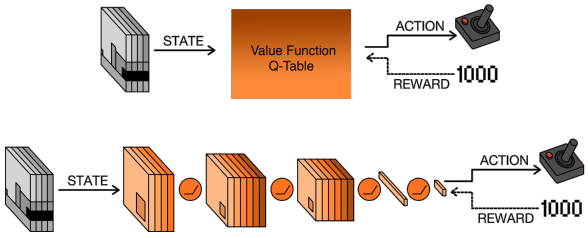

Figure 4: Q-learning RL (above) and DQN (below) [17].

in RL to high dimensional problems. DRL can deal efficiently with the curse of dimensionality unlike tabular and traditional non-parametric methods [4].

In DRL, deep neural network is trained to model/predict one or more of (a) the optimal value functions $V^{\pi}(s)$ (b) optimal policy $\pi(s)$ (c) optimal quality function Q^* (d) optimal actions $\mathcal{A}^*$.

Many works on DRL is based on gradient based backpropagation algorithm [13, 19, 20] which models the optimisation of the expected return as the optimisation of a stochastic function.

This stochastic function can have one or more models, policies and value functions combined in various ways. Each individual component might not directly optimise the expected reward but can incorporate useful information to optimize reward as a whole. For example, a DRL using a differentiable model and policy, it is possible to forward propagate and backpropagate through entire episodes; and policy component can learn the information over the history. Both can be summarized with a value functions for optimizing reward [13].

To present a concrete example of how a traditional RL network is extended to a DRL, I will present a value-function-based DRL algorithms with the DQN in Figure 4. Q-functions learns action-value function. In a traditional RL problem, a Q-learning function create and update a Q-table to find the maximum expected future reward of an action, given a current state. The model takes greyscale images as state from the video game; with the input current state Q-table returns with actions. It is a good strategy but it is not scalable.

On the other hand, the DQN takes the state-a stack of greyscale frames from the video game-and processes it with convolutional and fully connected layers, with ReLU nonlinearities in between each layer. At the final layer, the network outputs a discrete action, which corresponds to one of the possible control inputs for the game. Given the current state and chosen action, the game returns a new score. The DQN uses the reward-the difference between the new score and the previous one-to learn from its decision. More precisely, the reward is used to update its estimate of Q and the error between its previous estimate and its new estimate is backpropagated through the network.

1.5 Current Research Challenges:

To end with the short review on DRL, I present some of the research challenges in Deep Reinforcement Learning process.

Exploration vs Exploitation: Online decision making involves a fundamental choice *(i) Exploitation* or *(ii) Exploration*. Exploitation refers to making the best decision with the current information gathered by the agent. Whereas exploration involves attempting new action for more information. This is also known as *exploitation-exploration* dilemma. The best long-term strategy may involve lot of exploration and short term sacrifices. On the other hand if the agent has already found best strategy exploration, it may reduce the rewards.

Transfer Learning: Transfer learning is about efficiently using previous knowledge from a source environment to achieve new (slightly) different tasks in a target environment. To achieve this, agent must develop generalization capabilities such as *(i) feature selection (ii) removing asymptotic bias (iii) reduce overfitting and function approximator (iv) Optimizing horizon* (length of observations history involved in decision making process) etc.

Learning without explicit reward function: In reinforcement learning, the reward function defines the goals to be achieved by the agent. Due to the complexity of the environments in practical applications, defining a reward function can turn out to be rather complicated. Approaches like *imitation learning* (supervised learning) and *inverse reinforcement learning* where agent determines possible reward functions given observations of optimal behavior, are research challenges for the next decade.

2 PART B. SELF INSTRUCTING DATABASE:

2.1 Overview:

Any standard database has considerable large number of configuration knobs. Databases needs to be tweaked with proper configuration for running efficiently with different workloads and hardware resources. Tweaking database configuration knobs needs a great level of expertise from database administrator (DBA) because optimal configuration varies with the type of workloads. Beside that, even finding optimal knob configurations might involve a trial and error process for a DBA (which restricts the search spaces for knobs). An auto-configuring database system or self instructing database is a desirable feature to demand from database companies.

Human database administrators rely on experience and intuition to configure it. DRL process mimick same learning strategies i.e. learn from mistakes and correctness to maximize future rewards. Keeping this in mind we can explore how DRL can provide a solution to automatic database tuning.

2.2 Problem Formulation:

Given a workload $\mathcal{W}$ (which is a serialized query profile DAG of execution task), with a knobs setting $C = \{c_1, c_2, \ldots, c_{|C|}\}$ a typical database outputs a log of metrics $\mathcal{M} = \{m_1, m_2, m_3, \ldots, m_{|\mathcal{M}|}\}$ as shown in Figure 5. The database keeps receiving workloads at some discrete interval of time and it runs with some configuration and outputs a metric log, which is also called a discrete time stochastic control process.

We can apply deep reinforcement learning to train a neural network in taking over the database tuning process by optimizing configuration for observable database workload. Essentially, we have to define a RL problem environment consisting of the four following components to perform the learning as shown in Figure 6:

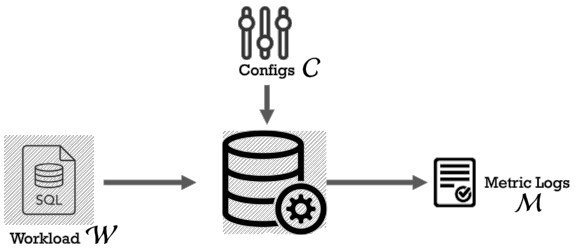

Figure 5: A typical database system.

1) Observable State: This is also input to the neural network. This is typically the current workload in form of query characteristics, for which the system should be optimised as well as the current state of the configuration. Figure 6 illustrates workload $\mathcal{W}_t$ is mapped to observation/state $\omega_t \in \Omega$ or $s_t \in \mathcal{S}$ for time stamp t. (Note: $\omega_t = s_t$ in MPD)
*(*Note: These notations are defined in Section 1.1).

2) Actions: An action is a bounded set of configuration C_t where each knobs can have a range of permissible values. Changing the size of a database buffer is an example of action. Now we can represent $C_t \rightarrow a_t \in \mathcal{A}$.

3) Reward: We map the metric $\mathcal{M}_t$ to rewards $r_t \in \mathcal{R}$. Since reward is a scalar function and metric $\mathcal{M}_t$ is a set of values representing the goodness and badness of database execution on workload $\mathcal{W}_t$, we need to apply some function $r_t = f(\mathcal{M}_t)$ to keep it simple. Later we will see an alternative approach where function f is not needed with a multitask-agent DRL.

4) Hyperparameters for Neural Network: This includes properties of the neural network (e.g. number of hidden layers, number of nodes per layer) as well as properties of the learning process like the number of iterations.

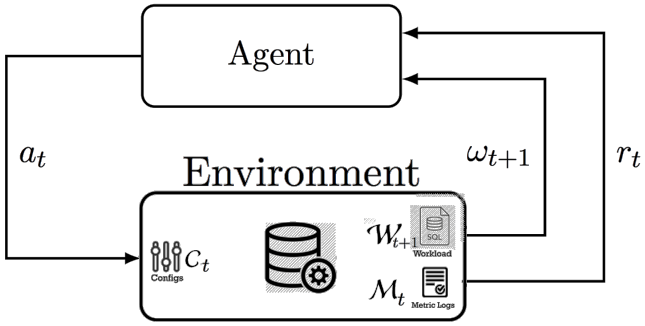

Figure 6: RL Agent-Database Environment Interaction.

Training and Challenges:

With the high level components defined, now we will go through the workflow of learner. Assuming the *neural network agent (NNA)* is configured with the required hyperparameters, the learning process starts with time $t = 0$. First, a workload $\mathcal{W}_t = s_t$ is fed into *neural network agent* (NNA). Then, the NNA explore action set $\mathcal{A}$ to produce an action (configuration) $a_t = C_t$. The database environment on receiving action configuration C_t executes workload $\mathcal{W}_t$ and returns with metrics $\mathcal{M}_t$. Some function $f : \mathcal{M}_t \rightarrow r_t \in \mathbb{R}$ converts metric to scalar reward. This process is repeated many times and agents either explore action set or exploit learned optimal action to predict next best configuration. The desired goal is to optimize maximum cumulative positive rewards.

An intuitive solution for choosing function $f : \mathcal{M}_t \rightarrow r_t \in \mathbb{R}$ is through the following steps:

- negate all the metrics whose desired objective is to minimize, such that we now optimize, for maximization.
- normalize each metric $m_i \in \mathcal{M}_t$ with their satisfied range of operation.
- rewards is sum of normalized metrics.

To avoid the problem of choosing a function f, we can transform the problem to a *Multi-task Deep RL* problem. Some hierarchical RL techniques also decompose tasks into subtasks, these methods then solve the subtasks in a locally optimal way and then global optimality can be achieved by aggregating back together. Another approach is to try *Linear Temporal Logic* specification that enables an interleaving of subtasks to support global optimization [14]. These techniques can help in avoiding selection of f by considering a specific set of metrics and optimize actions for it.

Model based/Model-free Agent: Model-free and model based both type of agent can be fruitful in this type of scenario. However model based approach needs more effort in design. The advantage of model based approach is that it can learn strategies to trade-off exploration and exploitation to learn quickly. But it is non-arguably plausible to choose gradient based methods because configurations knobs tend to have convex properties. Also selecting RL agent components such as *value functions, policy functions, actor-critic or model* is subjected to further experimentation with various hyperparameters (such as number of hidden layers, ReLU layers etc.).

Transfer Learning: It is essential for this type of system that a learned agent adapt to a new environment or a new database. Transfer learning is only possible when the agent can obtain generalization in learning procedure. When the quality of the dataset increases, the risk of overfitting is lower and the learning algorithm can trust more the maximum likelihood model by looking into a larger policy class and less approximations. On the other hand, when the quality of the dataset is low, the learning algorithm should be cautious on being too confident about the maximum likelihood model and should favor more robust policies. Hence, the sampling technique to obtain a good coverage of sample space is important.

2.3 Conclusion:

In Section 2 I presented an overview of the DRL model for autonomous or self-instructing databases. The model proposed is the starting point which can be extended as required for experimentation. Challenges remaining have also been address in this small review. A detailed deep dive with workload characteristics/representation is needed independently for future work on autonomous databases.

REFERENCES

[1] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath. A brief survey of deep reinforcement learning. *arXiv preprint arXiv:1708.05866*, 2017.
[2] A. G. Barto, R. S. Sutton, and C. W. Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[3] R. Bellman. Dynamic programming. *Princeton, USA: Princeton University Press*, 1(2):3, 1957.

[4] Y. Bengio, A. Courville, and P. Vincent. Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828, 2013.

[5] D. P. Bertsekas. Dynamic programming and suboptimal control: A survey from adp to mpc. *European Journal of Control*, 11(4-5):310–334, 2005.

[6] G. Cuccu, M. Luciw, J. Schmidhuber, and F. Gomez. Intrinsically motivated neuroevolution for vision-based reinforcement learning. In *Development and Learning (ICDL), 2011 IEEE International Conference on*, volume 2, pages 1–7. IEEE, 2011.

[7] M. P. Deisenroth, G. Neumann, J. Peters, et al. A survey on policy search for robotics. *Foundations and Trends® in Robotics*, 2(1–2):1–142, 2013.

[8] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel. Deep spatial autoencoders for visuomotor learning. In *2016 IEEE International Conference on Robotics and Automation (ICRA)*, pages 512–519. IEEE, 2016.

[9] F. Gomez and J. Schmidhuber. Evolving modular fast-weight networks for control. In *International Conference on Artificial Neural Networks*, pages 383–389. Springer, 2005.

[10] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine. Continuous deep q-learning with model-based acceleration. In *International Conference on Machine Learning*, pages 2829–2838, 2016.

[11] M. Hausknecht and P. Stone. Deep recurrent q-learning for partially observable mdps. *CoRR, abs/1507.06527*, 7(1), 2015.

[12] N. Heess, J. J. Hunt, T. P. Lillicrap, and D. Silver. Memory-based control with recurrent neural networks. *arXiv preprint arXiv:1512.04455*, 2015.

[13] N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa. Learning continuous control policies by stochastic value gradients. In *Advances in Neural Information Processing Systems*, pages 2944–2952, 2015.

[14] R. T. Icarte, T. Q. Klassen, R. Valenzano, and S. A. McIlraith. Teaching multiple tasks to an rl agent using ltl. 2018.

[15] J. Koutník, G. Cuccu, J. Schmidhuber, and F. Gomez. Evolving large-scale neural networks for vision-based reinforcement learning. In *Proceedings of the 15th annual conference on Genetic and evolutionary computation*, pages 1061–1068. ACM, 2013.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[17] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[18] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh. Action-conditional video prediction using deep networks in atari games. In *Advances in neural information processing systems*, pages 2863–2871, 2015.

[19] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[20] J. Schulman, N. Heess, T. Weber, and P. Abbeel. Gradient estimation using stochastic computation graphs. In *Advances in Neural Information Processing Systems*, pages 3528–3536, 2015.

[21] R. S. Sutton. Temporal credit assignment in reinforcement learning. 1984.

[22] R. S. Sutton and A. G. Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[23] D. Wierstra, A. Förster, J. Peters, and J. Schmidhuber. Recurrent policy gradients. *Logic Journal of the IGPL*, 18(5):620–634, 2010.

[24] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992.