Master Thesis

Learning Policies with Neural Ordinary Differential Equations

Lukas Schreiner

Thesis submitted in partial fulfillment
of the requirements for the degree of
Master of Science of Artificial Intelligence
at the Department of Advanced Computing Sciences
of the Maastricht University

Thesis Committee:

Prof. F. Thuijsman
Dr. P. Bonizzi
Elon Musk

Maastricht University
Faculty of Science and Engineering
Department of Advanced Computing Sciences

September 22, 2022

Contents

Table of abbreviations

RL Reinforcement learning

NN neural network

DRL deep reinforcement learning

nODE neural ordinary differential equation

ODE ordinary differential equation

MDP markov decision process

TD temporal difference

SARSA State–Action–Reward–State–Action

DQN deep Q neural networks

NFQ neural fitted Q iteration

Chapter 1

Introduction

Reinforcement learning (RL) is one of the major machine learning domains that experienced siginificant advances in recent years. An RL agent learns through interaction with the environment and it's corresponding reward signals. Through try and error, the agent is able to explore new strategies and come up with novel solutions to solve the given problem.

This fundamentally differentiates RL from supervised and unsupervised learning techniques that rely on learning patterns from data. Some authors classify RL as semi supervised. The key challenge is to learn a representation that gives guidance what the agent should do in each state it encounters. In early models this was done by a table that contained the action values for each state. This approach is obviously not feasible as the number of states and available actions increases since the table would grow exponentially and quickly exceed memory. A solution to this issue was to replace the table with a function that is represented by a neural network (NN). The advantage is that a NN can assign similar action values to states that have similar features, this technique is called deep reinforcement learning (DRL) [16].

Recently, a new kind of NN attracted a lot of attention. neural ordinary differential equation (nODE)s use the NN not for approximating function values directly but rather learn a differential equations for the inner state. The model output is then derived by solving an initial value problem with the learned ordinary differential equation (ODE). By discretising the nODE, the hidden state of the model can have an arbitrary number of layers, which comes with numerous advantageous properties. This work aims to given a throughout assessment whether nODEs are suited for being used in RL problems, in particular, whether they exceed performance of conventional NNs in a DRL context.

The thesis is structured the following way. Chapter 1 gives an general and formal introduction to reinforcement Learning (section 1.1), deep reinforcement learning (section 1.2) and neural ordinary differential equation (section 1.3). Chapter 2 introduces the model and necessary adjustments needed for RL. Chapter 2 describes the experimental set up before chapter 3 presents the empirical results from the experiments. Chapter 4 closes with a conclusion and provides an outlook and perspective on further research.

1.1 Reinforcement Learning

The Reinforcement Learning Problem RL is essentially a framework for learning how to interact with the environment from experience. The environment is thereby represented by a set of states $s \in S$ that the agents visits depending on the actions $a \in A$ played and optionally a random factor. In each state transition, the agent receives a reward signal $r(s, s', a)$ depending on the action played in current state and the resulting subsequent state. The agent now tries learn a policy $\pi(s, a)$ that maximises the total or average reward obtained. Since rewards occur in future states, the agent needs planning capabilities in order optimally solve the problem. In order to evaluate which action to choose the agent uses a state value function $V_\pi(s)$ that indicates what the discounted future rewards will be if the agent follows a given policy π. When learning, the agent faces the tradeoff whether to optimise the current strategy (on-policy) or try new random actions that might be suboptimal but lead to a higher reward in the future (off-policy). This is referred as the exploitation-exploration trade off in RL [15]. There exists a rich body of literature of methods that aim to solve the RL in various different ways. One of the main features that separates all is methods is whether the agent has a model of the environment available or not. If so, the agent can better asses the consequences of an action played yet coming up with a model of the environment is often not a trivial task for the researcher and so the recent literature mostly focuses on model-free approaches. See [27, 25] for an overview.

Model-based reinforcement learning If there exists a probabilistic model of the environment, the transition probability $t(s, a, s')$ and its corresponding reward function $r(s, a, s')$ between states are known. In this case the problem can be modelled using a markov decision process (MDP) [2]. If the state and action space are reasonable in size, one can either use value iteration or policy iteration in order derive an optimal policy $\pi^*(s)$. Value iteration is an algorithm that visits every state and chooses thereby the action that yields the highest Bellman update consisting of the the current reward plus the discounted future state value. After sufficient iterations, the optimal policy is derived by choosing the action in state the yields the highest state values

$$V_{\pi,new}(s_t) = \max_a [r(s, a) + \gamma * \Sigma_s t(s, a, s') * V(s')], \text{ for } s \in S \tag{1.1}$$

After convergence, the optimal policy is given by

$$\pi(s)^* = \arg\max_a [r(s, a) + \gamma * \Sigma_s t(s, a, s') * V(s')], \text{ for } s \in S \tag{1.2}$$

Policy iteration, on the other hand, computes the utility values $u(s, \pi)$ for all the states following the current policy $\pi(s)$ thereby evaluating the policy.

$$u(s, \pi) = r(s, \pi(s)) + \gamma * \Sigma_s t(s, \pi(s), s') u(s', \pi)), \text{ for } s \in S \tag{1.3}$$

In a second step, the policy is improved until convergence $\pi_{new}(s) = \pi(s) = \pi(s)^*$

$$\pi_{new}(s) = \arg\max_{a'} [r(s, \pi(s)) + \gamma \Sigma_s t(s, a', s') u(s', \pi)], \text{ for } s \in S \tag{1.4}$$

When doing policy iteration, the agent plays the action that the current policy intends for the given state. If the the action does not chooses the state transition that leads to the highest state value in the future state, the policy is updated. While being a mathematically sound theory, this approach is not suited for high dimensional state spaces since visiting all states of the environment for one update is can become computationally very expensive.

Model-free reinforcement learning In many situations there does not exists a model of the underlying environment so the transition probabilities $t(s, a, s')$ are not known. In this case, the agent cannot compute an expected reward and needs to rely only on the reward signal of the actions played. In a naive monte carlo approach, the agent just plays through a whole episode of states and at the end of each episode the value function for visited states is updated with an average of the cumulated reward. While this approach is unbiased and will find an optimal policy at some point it is very sample inefficient.

A more efficient way is therefore to use temporal difference (TD) learning [24], which updates the state values after a certain number of states are played. Thereby a TD target estimate is computed by summing the current reward r_t with the γ discounted state value estimates ranging some states into the future. How those future states and actions are determined depends on the specific learning algorithm used. The difference between the TD target estimate and the old function estimate is called TD error and is used to update the function after being multiplied with the learning rate α. In TD(0) this means that only the next state is considered which gives an update rule for the value function

$$V_{new}(s_t) = V_{new}(s_t) + \alpha * \overbrace{(\underbrace{r_t + \gamma V_{old}}_{\text{TD estimate}} - V_{old})}^{\text{TD error}} \tag{1.5}$$

While the value function $V(s)$ gives the state value assuming the agent plays the best action, the Q-function $Q(s, a)$ gives back the state value for any action that the agent could play. This richer representation of state values is therefore well suited for explorative off policy algorithms. The optimal policy is then derived by choosing the action that yields the highest Q-value in a given state $\pi(s, a) = \arg\max_a Q(s, a)$. There are two main domains for learning the Q-function, on-policy algorithms like State–Action–Reward–State–Action (SARSA) [20] that learns by strictly following the policy $\pi(s_t) = a_t$.

$$Q_{new}(s_t, a_t) = Q_{old}(s_t, a_t) + \alpha * (r_t(s, a) + \gamma Q_{old}(s_{t+1}, a_{t+1}) - Q_{old}(s_t, a_t)) \tag{1.6}$$

Q-learning [30] is an off-policy learning algorithm because it mixes playing random actions (exploration), and following the reward maximising policy (exploitation). In both cases the Q-values are updated using a bellman equation

$$Q_{new}(s_t, a_t) = Q_{old}(s_t, a_t) + \alpha * (r_t(s, a) + \gamma \max_a Q_{new}(s_{t+1}, a) - Q_{old}(s_t, a_t)) \tag{1.7}$$

This allows the agent not only learn not only from imitation but also from experience replay [13]. Thereby the agent samples past state transitions in order to improve the Q-values. Q-learning tends to convergence faster than SARSA yet it incorporates higher variance. In practice, starting with a lot of off-policy exploration and then transitioning into on-policy learning works well for many problems. Usually an ϵ-greedy algorithm is used that represents a decreasing probability threshold that determines whether a random action is played or the agent should follow the policy. See [25] for an extensive overview.

1.2 Deep Reinforcement Learning

As already mentioned, storing value functions and policies in tables is only feasible when the action and state space is of limited size. For that reason the logical next step was to learn a function that represents the value function [26]. This way, not only RL problems with large state and action spaces could be solved, but it also could leverage the fact that some states are very similar in attributes and therefore have similar optimal actions.

Even though the idea to use neural networks as function approximation is rather old [3], the breakthrough occurred with deep Q neural networks (DQN) [16] and AlphaGo [23]. Deep reinforcement learning is very powerful but it comes with new challenges regarding convergence [28]. The combination of non linear function approximation, experience replay and off-policy learning might cause the divergence in function approximation. In order to introduce stability into the function approximation setting, multiple approaches exist. All of them try to reduce the chance that suboptimal actions get overestimated and consequently selected too often which causes the model to diverge.

The neural fitted Q iteration (NFQ) algorithm [19] does not update network weights after each environment step but rather considers a the entire set of transition experiences. While this approach yields more stable results convergence is slow. DQNs find a suitable balance between stability and convergence by using a target and policy DQN network. While the weights of the policy net are updated after each enviroment step, the target network is used to select the actions. Every c steps the target net is updated with a copy of the weights of the policy net. Double DQNs [8] use two networks and the action is selected with a 50% chance by each of them. An updated version of this algorithm uses one network for action selection and the other one action evaluation. The weights of the former network are updated similar to an exponentially weighted moving average [29].

Another way of reducing the risk of overestimating is using prioritised experience replay [22], which samples state transitions according to their experienced TD error. That way, state transitions that yield a higher learning effect are sampled more often. Stochastic sampling ensures that probabilities are interpolated between a pure greedy approach and conventional uniform sampling. That allocates a nonzero probability to each transition and prevents overfitting.

Double Q Networks have since then contributed to advances in many fields, e.g. for automomous [31] as well as well as electric [6] vehicles but also nuclear [14], quantum [9] and IoT [11] applications.

1.3 Neural Ordinary Differential Equations

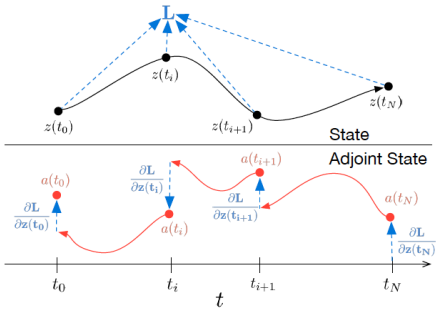

Figure 1.1: Continuous Backpropagation [5, Figure 2]

Neural ordinary differential equations [5] are a novel method for deep learning that follow a different approach. While conventional feed forward neural networks learn a function that maps the input vector x and model parameter θ to the model output $\tilde{y}$ ($f(x, \theta) = \tilde{y}$), nODEs use the neural network to learn a differential function that describes the hidden state $z(t)$.

$$f(z(t), t, \theta) = \frac{dz(t)}{dt} \tag{1.8}$$

The model output is then determined by solving an initial value problem using the model input as starting point at t_0 solving for the value at t_N. That way, the hidden state is continuous in the limit and discretised by the evaluation points at $t_0, t_i, t_{i+1}, \ldots, t_N$ by the ODE solver. The loss is then computed as the solution of the solver which defines the loss function L as follows

$$L(z(t_1)) = L(z(t_0) + \int_{t_0}^{t_1} f(z(t), t, \theta)) = L(ODESolve(z(t_0), f, t_0, t_1, \theta)) \tag{1.9}$$

Thereby $z(t_1)$ is the hidden state at t_1 and represents the model output. In order to learn the parameters θ of the neural network, the authors suggest to use the adjoint method [17] which defines an adjoint ODE that is solved backwards in time, see figure 1.1.

Continuous Backpropagation In order to compute the gradients with respect to the weights $\frac{dL}{d\theta}$, an adjoint state $a(t)$ is defined as the the gradient of the hidden state with respect to to the loss, see 1.10. That avoids backpropagating though the ODE solver. The dynamics of $a(t)$ are derived by making use of the definition of the neural network's hidden state through time $\frac{dz(t)}{dt}$ and the chain rule. The derivations follow [5, Appendix B.1].

$$a(t) = \frac{dL}{dz(t)} = \frac{dL}{dz(t+\epsilon)} \frac{dz(t+\epsilon)}{dz(t)} \tag{1.10}$$

$$a(t+\epsilon) = \frac{dL}{dz(t+\epsilon)} \tag{1.11}$$

The finite difference of the hidden state is then given by

$$z(t+\epsilon) = \int_t^{t+\epsilon} f(z(t), t, \theta)dt + z(t) = T_\epsilon(z(t), t) \tag{1.12}$$

which can be shown by transforming and integrating 1.8. Plugging 1.12 and 1.11 into 1.10 gives

$$a(t) = a(t+\epsilon)\frac{\partial T_\epsilon(z(t), t)}{\partial z(t)} \tag{1.13}$$

The dynamics of $a(t)$ is then derived using the definition of a derivative coming from finite differences in the limit.

$$
\begin{aligned}
\frac{da(t)}{dt} &= \lim_{x\to 0^+} \frac{a(t+\epsilon) - a(t)}{\epsilon} \\
&= \lim_{x\to 0^+} \frac{a(t+\epsilon) - a(t+\epsilon)\frac{\partial T_\epsilon(z(t),t)}{\partial z(t)}}{\epsilon} \\
&= \lim_{x\to 0^+} \frac{a(t+\epsilon) - a(t+\epsilon)\frac{1}{\partial z(t)}(z(t) + \epsilon f(z(t),t,\theta) + \mathcal{O}(\epsilon^2))}{\epsilon} \\
&= \lim_{x\to 0^+} \frac{a(t+\epsilon) - a(t+\epsilon)(I + \epsilon\frac{f(z(t),t,\theta)}{dz(t)} + \mathcal{O}(\epsilon^2))}{\epsilon} \\
&= \lim_{x\to 0^+} \frac{-a(t+\epsilon)\epsilon\frac{f(z(t),t,\theta)}{dz(t)} + \mathcal{O}(\epsilon^2)}{\epsilon} \\
&= \lim_{x\to 0^+} -a(t+\epsilon)\frac{f(z(t),t,\theta)}{dz(t)} + \mathcal{O}(\epsilon^1)
\end{aligned}
\tag{1.14}
$$

which in the limit gives

$$
\frac{da(t)}{dt} = -a(t)^T \frac{f(z(t),t,\theta)}{dz(t)} \tag{1.15}
$$

with 1.15 being one of the key results in the paper. The gradient at the final value t_N and model output is defined by

$$
a(t_N) = \frac{dL}{dz(t_N)} \tag{1.16}
$$

from which we can derive the the gradient with respect to to the hidden state at any point in time including the initial or input value.

$$
a(t_0) = \frac{dL}{dz(t_0)} + \int_{t_N}^{t_0} \frac{da(t)}{dt} dt = a(t_N) - \int_{t_N}^{t_0} a(t)^T \frac{\partial f(z(t),t,\theta)}{\partial z(t)} \tag{1.17}
$$

In 1.17, it is assumed that the loss L depends only on the last time point t_N. If the loss function also should depend on intermediate steps $t_1, t_2, t_3, ...$, the adjoint step is repeated backward for the intervals $[t_{N-1}, t_N], [t_{N-2}, t_{N-1}], ...$ with the gradients added together at the end.

Gradients for the model parameters In order to derive the gradients with respect to the parameters of the model, the authors generalise 1.15 by defining the adjoint state as a vector consisting of the hidden state z, the model parameters θ as well as the time dimension t, see [5, Appendix B.2]. Since the model parameters are a constant and t is constant with respect to itself, the time dynamics are given by

$$
\frac{\partial \theta(t)}{dt} = 0 \tag{1.18}
$$

$$
\frac{\partial t(t)}{dt} = 1 \tag{1.19}
$$

Using 1.8, 1.18 and 1.19 and the combined time dynamics is then given by

$$\frac{d}{dt}\begin{bmatrix} z \\ \theta \\ t \end{bmatrix} = f_{aug}([z, \theta, t]) := \begin{bmatrix} f([z, \theta, t]) \\ 0 \\ 1 \end{bmatrix} \tag{1.20}$$

The jacobian of f_{aug} is

$$\frac{\partial f_{aug}}{\partial [z, \theta, t]} = \begin{bmatrix} \frac{\partial f_{aug}}{\partial z} & \frac{\partial f_{aug}}{\partial \theta} & \frac{\partial f_{aug}}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix} \tag{1.21}$$

with $\mathbf{0}$ being matrices of zeros that have corresponding shapes depending on θ and t. The combined augmented state is defined as follows with the first element in the vector being similar to 1.10

$$a_{aug} := \begin{bmatrix} a \\ a_\theta \\ a_t \end{bmatrix} = \begin{bmatrix} \frac{dL}{dz(t)} \\ \frac{dL}{d\theta(t)} \\ \frac{dL}{dt(t)} \end{bmatrix} \tag{1.22}$$

With its dynamics given by 1.15

$$\frac{da_{aug}(t)}{dt} = -a_{aug}(t)^T \frac{\partial f_{aug}}{\partial [z, \theta, t]}(t) \tag{1.23}$$

Finally, 1.21 and 1.22 plugged into 1.23 yields

$$\frac{da_{aug}(t)}{dt} = -\begin{bmatrix} a & a_\theta & a_t \end{bmatrix} \begin{bmatrix} \frac{\partial f_{aug}}{\partial z} & \frac{\partial f_{aug}}{\partial \theta} & \frac{\partial f_{aug}}{\partial t} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} \end{bmatrix}(t) = -\begin{bmatrix} a\frac{\partial f_{aug}}{\partial z} & a\frac{\partial f_{aug}}{\partial \theta} & a\frac{\partial f_{aug}}{\partial t} \end{bmatrix}(t) \tag{1.24}$$

with the first element in the vector being equal to 1.15. Similar to 1.17, the gradients with respect to the model parameters θ can be obtained by integrating the second element of the vector in 1.24 over the full interval while setting $a_\theta(t_N) = \mathbf{0}$

$$\frac{\partial L}{d\theta} = a_\theta(t_0) = -\int_{t_N}^{t_0} a(t)^T \frac{\partial f(z(t), t, \theta)}{\partial \theta} dt \tag{1.25}$$

Characteristics and applications of NODEs

Claims from the authors

Normalising Flows [18]

Chapter 2

Reinforcement Learning with neural ODEs

This work is mainly concerned with applying neural nODEs to reinforcement learning problems and comparing their performance with regard to convergence, stability and overall reward achievements to a state of the art DQN model of similar size.

Like conventional ODEs, neural ODE can only map input x of similar dimensionality $\phi : \mathbb{R}^d \rightarrow \mathbb{R}^d$. That poses a major issue for their use DRL because the agent is required to learn a mapping from a d dimensional observation space to a n dimensional action space. Common approaches either augment the input vector with a number of zeros such that dimensions match [7] or add a linear layer at the end of the nODE [12]. It has been shown that the latter architecture for ResNet [10] is a universal approximator [12] and since ResNet are the contionous analogs of nODEs [21], this architecture is chosen in order to deal with dimensionality. See figure 2.1 for an example with a one dimensional action space.

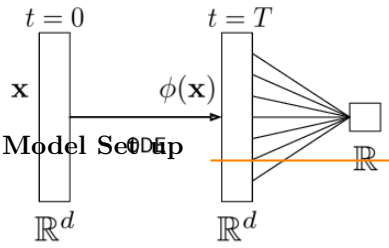

Figure 2.1: nODE architecture [7, Figure 2]

Hypothesis One hypothesis to be examined is that the continuous nature of neural nODEs might give them advantages in environments with action spaces that are continuous in nature.

Dual model, prioritised experience replay, epsilon greedy

Problem Setting An experimental set up is created that simulates a DRL problem, which is often referred to in the literature as the balancing pole [1] problem. The agent controls a cart that can go to the left or to the right in a limited space. The cart has a horizontal pole attached on a joint and the agent receives a reward of 1 for each time step where the pole is balances in a somewhat upright position on the cart. If the pole tilts or the maximum number of 500 time steps is reached, the episode ends. The implementation used in this work comes from OpenAI's Gym library [4]. A detailed description of the problem can be found in appendix A.1.

Set up In order to compare model performance, a conventional deep learning model is compared to neural ODE model with similar parameters are trained. In both set ups, are dual model set up is used with a policy and and target net [8] with prioritised experience replay [22]. First the models are trained evaluated under different conditions with different metrics. On the one hand, the speed of convergence is evaluated by measuring the number of episodes it takes the model to solve the environment for the first time while training. On the other hand, model robustness is tested by letting the model play the environment with different levels of distortions.

Description	Value
Learning rate	$l = 0.001$
Discount factor	$\gamma = 0.95$
Target net weight update	$\tau = 0.05$
Batch size	$b = 128$
Epsilon greedy start	$\epsilon_0 = 1.0$
Epsilon greedy end	$\epsilon_N = 0.0$
No. Episodes	$N = 2000$

Table 2.1: Model Hyperparameter

Chapter 3

Results

Chapter 4

Conclusion

4.1 Outlook

Bibliography

[1] Andrew G Barto, Richard S Sutton, and Charles W Anderson. Neuronlike adaptive elements that can solve difficult learning control problems. *IEEE transactions on systems, man, and cybernetics*, (5):834–846, 1983.

[2] Richard Bellman. A markovian decision process. *Journal of mathematics and mechanics*, pages 679–684, 1957.

[3] Dimitri Bertsekas and John N Tsitsiklis. *Neuro-dynamic programming*. Athena Scientific, 1996.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[5] Ricky TQ Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. *Advances in neural information processing systems*, 31, 2018.

[6] Zheng Chen, Hongji Gu, Shiquan Shen, and Jiangwei Shen. Energy management strategy for power-split plug-in hybrid electric vehicle based on mpc and double q-learning. *Energy*, 245:123182, 2022.

[7] Emilien Dupont, Arnaud Doucet, and Yee Whye Teh. Augmented neural odes. *Advances in neural information processing systems*, 32, 2019.

[8] Hado Hasselt. Double q-learning. *Advances in neural information processing systems*, 23, 2010.

[9] Zhimin He, Lvzhou Li, Shenggen Zheng, Yongyao Li, and Haozhen Situ. Variational quantum compiling with double q-learning. *New Journal of Physics*, 23(3):033002, 2021.

[10] S Jian, H Kaiming, R Shaoqing, and Z Xiangyu. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision & Pattern Recognition*, pages 770–778, 2016.

[11] Kai Li, Wei Ni, Bo Wei, and Eduardo Tovar. Onboard double q-learning for airborne data capture in wireless powered iot networks. *IEEE Networking Letters*, 2(2):71–75, 2020.

[12] Hongzhou Lin and Stefanie Jegelka. Resnet with one-neuron hidden layers is a universal approximator. *Advances in neural information processing systems*, 31, 2018.

[13] Long-Ji Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8:293–321, 1992.

[14] Zheng Liu and Shiva Abbaszadeh. Double q-learning for radiation source detection. *Sensors*, 19(4):960, 2019.

[15] James G March. Exploration and exploitation in organizational learning. *Organization science*, 2(1):71–87, 1991.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[17] Lev Semenovich Pontryagin. *Mathematical theory of optimal processes*. CRC press, 1987.

[18] Danilo Rezende and Shakir Mohamed. Variational inference with normalizing flows. In *International conference on machine learning*, pages 1530–1538. PMLR, 2015.

[19] Martin Riedmiller. Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method. In *Machine Learning: ECML 2005: 16th European Conference on Machine Learning, Porto, Portugal, October 3-7, 2005. Proceedings 16*, pages 317–328. Springer, 2005.

[20] Gavin A Rummery and Mahesan Niranjan. *On-line Q-learning using connectionist systems*, volume 37. University of Cambridge, Department of Engineering Cambridge, UK, 1994.

[21] Michael Sander, Pierre Ablin, and Gabriel Peyré. Do residual neural networks discretize neural ordinary differential equations? *Advances in Neural Information Processing Systems*, 35:36520–36532, 2022.

[22] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[23] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016.

[24] Richard S Sutton. Learning to predict by the methods of temporal differences. *Machine learning*, 3:9–44, 1988.

[25] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[26] Richard S Sutton, David McAllester, Satinder Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. *Advances in neural information processing systems*, 12, 1999.

[27] Csaba Szepesvári. Reinforcement learning algorithms for mdps. 2009.

[28] John Tsitsiklis and Benjamin Van Roy. Analysis of temporal-diffference learning with function approximation. *Advances in neural information processing systems*, 9, 1996.

[29] Hado Van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. In *Proceedings of the AAAI conference on artificial intelligence*, 2016.

[30] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8:279–292, 1992.

[31] Yi Zhang, Ping Sun, Yuhan Yin, Lin Lin, and Xuesong Wang. Human-like autonomous vehicle speed control by deep reinforcement learning with double q-learning. In *2018 IEEE intelligent vehicles symposium (IV)*, pages 1251–1256. IEEE, 2018.

Appendix A

Experimental Set Up

A.1 Balancing Pole

The action space of the balancing pole problem is discrete and consists of 2 possible actions.

Num	Action
0	Push cart to the left
1	Push cart to the right

The observation space is four dimensional and covers position and velocity of the cart as well as the pole.

Observation	Min	Max
Cart Position	−4.8	4.8
Cart Velocity	− inf	inf
Pole Angle	∼ -0.418 rad (-24°)	∼0.418 rad (24°)
Pole Angular Velocity	− inf	inf

The environment yields a reward of one for each time step, where the cart is still balancing the pole and it's position is still within the observation space. After a maximum of 500 time steps, the game stops resulting in a maximum reward of 500 per episode.