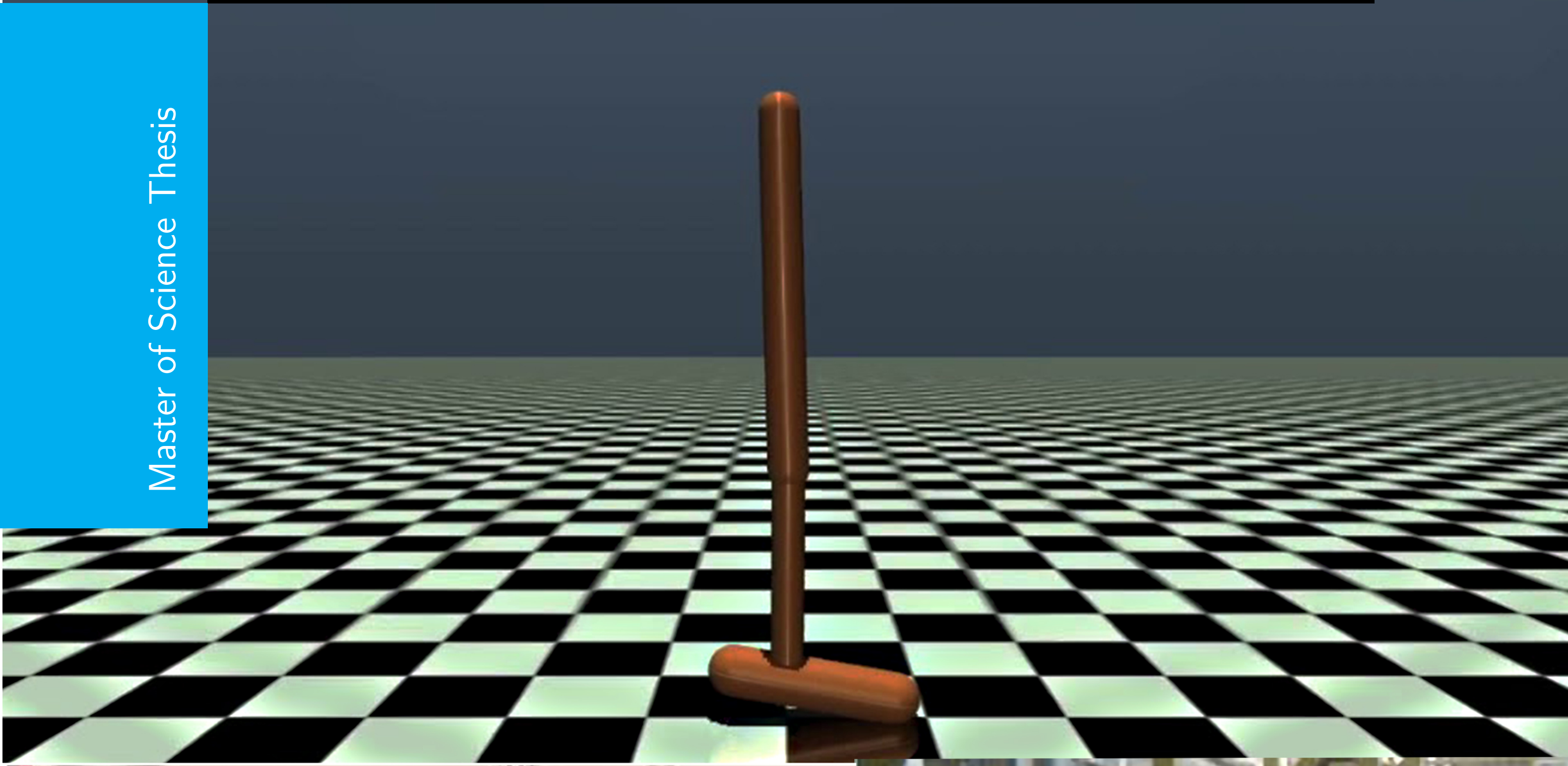
Deep vs Shallow Reinforcement Learning for low-dimensional continuous control tasks

V. Arnaoutis

Master of Science Thesis



Delft University of Technology

Delft Center for Systems and Control

Deep vs Shallow Reinforcement Learning for low-dimensional continuous control tasks

Master of Science Thesis

For the degree of Master of Science in Systems and Control at Delft University of Technology

V. Arnaoutis

June 28, 2019

Faculty of Mechanical, Maritime and Materials Engineering (3mE) · Delft University of Technology

Delft
University of
Technology

Summary

Deep Learning performance dependents on the application and methodology. Neural Networks with convolutional layers have been a great success in multiple tasks trained under Supervised Learning algorithms. For higher dimensional problems, the selection of a deep network architecture can significantly improve the accuracy of the network, however for low dimensional problems this might not be true. Shallow Neural Networks have successfully matched the performance of Deep Neural Networks in multiple tasks in the past and have been shown to be expressive enough to represent low dimensional continuous control problems. Through the thesis, the performance and expressiveness of Shallow and Deep networks is compared for low-dimensional continuous control tasks. The thesis begins by comparing the two network architectures in a Supervised Learning algorithm and progresses towards state-of-the-art Reinforcement Learning algorithms. The thesis provides an empirical approach towards comparison of neural networks and makes conclusions that can support the selection of a network architecture for continuous control applications using Deep Reinforcement Learning algorithms.

The selection of a shallow and deep neural network architecture that can learn the target function for one of the tested benchmarks (inverted pendulum) is achieved by training the networks in a vanilla Supervised Learning algorithm. The algorithm is then upgraded to incorporate a progressive tracking of the data, a process which was found capable for both the shallow and deep network. The complexity of the algorithm was increased by introducing bootstrapping of the data and removing any pre-collected data that assisted learning. From bootstrapping the target function, the networks were challenged to learn by themselves, a condition that motivated additional noise and poor data generation. Bootstrapping has been proven challenging for the deeper architecture which over-fit in itself. The final stage was the implementation of the state-of-the-art Reinforcement Learnign algorithm , Deep Deterministic Policy Gradient. The Reinforcement Learning algorithm completely removed the structured learning that was present in the previous methods as it forced the network to learn from simulating episodes. Once again, the deep architecture struggle to learn compared to the shallow one. At this point, two additional benchmarks were tested on the same algorithm using a shallow and a deep neural network. The performance for each benchmark varied, which suggested that both shallow and deep architectures could achieve sufficient results.

The thesis concluded that a shallow or deep architecture with two hidden layers, can be appropriate for training a low-dimensional continuous control task. The performance difference of shallow and deep architectures can vary between setups which can be the result of fine-tuning of hyper-parameters. An important setback that was presented was the repeatability and reliability of good learning for both architectures. It has been shown that indeed both network architectures can learn a specific task, however for one good training session, there could be many bad ones. For future implementations, it is proposed that a deep network architecture shall be favored, due to the additional expressiveness it can provide, however a shallow network shall also be tested as a benchmark-test for the deep network.

Table of Contents

Acknowledgements

I would like to thank my supervisor Prof.dr. Robert Babuska for the opportunity to be involved in this thesis and his consistent attention throughout the year. I would also like to thank my daily supervisor Ir. Tim de Bruin for his never-ending support and willingness to help me at any time.

Delft, University of Technology — V. Arnaoutis
June 28, 2019

Chapter 1

Introduction

Deep Learning has shown great success in image processing and time-series problems [1] [2] [3] [4] [5] [6]. Since then, Neural Networks have gained the interest of Reinforcement Learning algorithms, for approximation of complex high dimensional functions [7] [8]. Deep Reinforcement Learning has been developing in both high and low dimensional systems using both discrete and continuous control actions [9] [10] [11].

As any neural network is a universal approximator, a selection of an exact network architecture is hard to be supported. Nevertheless the performance of various depth networks can vary between tasks [12] [13]. The choice of a good performance architecture can be heavily dependent on the algorithm, environmental constrains and dimensions of the model [14] [15]. While depth can be theoretically proven to perform better or as good as a shallower network, this is not always true [16] [17]. In an attempt to minimize the variables and achieve an empirical study on the influence of the neural network architecture, we select to examine continuous control low-dimensional tasks.

Low-dimensional continuous control tasks have already been shown to learn with Neural Networks of very few layers [18] [19] [20]. For the thesis, the interesting transition is from shallow to deep networks, as defined in Chapter 2. Shallow Networks have already shown to be enough for such approximations, which makes the benefits of depth increase questionable [9] [21]. Thus, the study uses one architecture of each category, to compare performance.

The term "performance" of a neural network can vary based on the applied problem and the research goals. For this thesis the performance of a network is summarized in three properties. The most common performance measure of NNs is the *generalization error*, that represents how well the network can predict the output of new data samples with respect to the true output. This can be accomplished by comparing the NN with previously collected and unseen data or evaluate it during a simulation in a setup. The second performance criterion is the *computational time* and power required for the network to train on the data. The network's size and complexity can be a crucial factor of learning as some algorithms and networks could require multiple hours, even days to process the data. If for those cases, the approach could be replaced by a different network architecture that could achieve similar or slightly less performance in much shorter period then the faster version shall be favored. The third criterion is the *reliability* which is introduced by the control aspect of the problem, for simulations and real applications, where control models could be fragile and expensive. Reliability focuses on how well the same network can approximate a function given random initialization of the parameters. A high reliability of a neural network means that given a specific network architecture, the algorithm can learn a function consistently, with similar accuracy.

Supervised Learning can be proven consistent due to the possibly noise free data, which increase the robustness of the network during training [22]. Regression tasks can be learned by Neural Networks without the hustle of fine tuning multiple hyper-parameters that are involved with the Reinforcement Learning [23] [24]. The empirical approach followed in this thesis, is the study of the performance of Shallow and Deep Neural Networks for learning a function starting from vanilla-Supervised Learning regression task and progressing in increments towards state-of-the-art Reinforcement Learning. In Chapter 2 the basic properties, setups and tools used in the thesis will be introduced. In addition, the chapter will present the first step, the vanilla Supervised Learning, which will aim in identifying a shallow and deep neural network architecture that can approximate the state-cost function of the Inverted Pendulum model (Section 2-2). Chapter 3, will introduce tracking of the changes of the state-cost function by training the neural networks in evolving target data. The procedure will continue by challenging the neural networks to learn from themselves instead of learning through pre-collected data, by bootstrapping value iteration, as presented in Chapter 4. The final stage is to train the neural network by collecting sampled data from simulation by running episodes. In Chapter 5, a state-of-the-art Deep Reinforcement Learning algorithm [19] will be used to train multiple low dimensional continuous control benchmarks, including an Inverted Pendulum, a Magnetic Manipulator and a one legged robot. Throughout the thesis, both Shallow and Deep Neural Networks will be examined in similar conditions of learning and the learned functions will be evaluated under various scenarios to test the networks performance.

The conclusion will summarise the findings of each chapter of the thesis, covering a different stage of the transition towards Deep Reinforcement Learning. Using the results and empirical findings, an educated conclusion will be done on the comparison between shallow and deep network architectures. A suggestion for the ideal architecture will be presented which could be favored when considering low-dimensional continuous control systems.

Chapter 2

Preliminaries

This chapter will introduce the concepts and terminology used throughout this thesis. Part of the chapter is an introduction to parameters of Neural Networks, related to initializing and training the network. The three continuous control tasks, Inverted Pendulum, Magnetic Manipulator and Hopper-v3, will be introduced and used throughout the report as benchmarks for comparison of the networks. Additional techniques that involved visualization of the neurons and network parameters will be explained in this chapter and will be used in later chapters for comparison between NN architectures.

2-1 Introduction to Neural Networks

A basic previous understanding is expected from the reader to follow the remaining of the thesis. Terminologies that were not defined explicitily in the thesis follow their common definition, that in most cases can be found in [25] [24] [26]. Some unique or modified parameter explanations can be found in this section.

2-1-1 Definition of Shallow and Deep Neural Networks

The definition of shallow and deep networks can vary between literatures. With the success of convolutional layers in high-dimensional problems, deep network architectures exist in many sizes. To provide clear distinction between shallow and deep networks, each architecture is defined as shown below:

Shallow Neural Network

A shallow network is any network whose architecture includes only one Fully-Connected (FC) hidden layer, inbetween the input and output layer. The use of activation functions, dropout layer and other normalization techniques is not considered as depth increase but simply as additional characteristics to the existing hidden layer.

Deep Neural Network

A deep network is defined as any network with more basic hidden layers than a SNN (i.e. 2 hidden layers or more). As convolutional layers are not introduced in this research and the hidden layer type remained FC for all experiments, the DNN definition includes all FC-NN that have higher depth than the SNN. In practice, the DNNs used in the report do not exceed the depth of 4 FC hidden layers.

2-1-2 Epoch and batch size

Two terminilogies associated with learning in Neural Networks are the epoch and batch-size [27]. A single *epoch* represents one complete use of the whole data-set. When the data used for a parameter upgrade is smaller than the whole data-set then it is referred to as *mini-batch*. In general, the size of the data used for one gradient step is referred to as *batch-size*.

2-1-3 Experience Replay

The term Experience Replay (ER) is used to describe a memory buffer that is used to store data samples [28]. In both offline and online RL algorithms, where data is generated from running episodes, each step in an episode is stored in a large database out of which batches are randomly selected to train the network [29]. Each step in the memory buffer consists of a tuple that is constructed from the current state, next state, action taken, reward and state-cost. The use of memory replay assists in the de-correlation of data and train from multiple episodes. Improvements on the simple ER have been done in the past out of which one of the most popular is the Prioritised ER that chooses data samples that can increase the learning speed [30].

2-2 Continuous Control Benchmarks

This section will introduce the continous tasks that will be used for training the NNs. The section will provide information that will be referred later in the report and are foundamental to understand later chapters. The models include the Inverted Pendulum, a 1-DOF under-actuated system, the Magnetic Manipulator and the Hopper-v3, a popular low-dimensional continuous control benchmark for DRL from OpenAI-Gym [31].

2-2-1 Inverted Pendulum

The 1-DOF Inverted Pendulum system (IP) is a digital twin of an existing setup, created and owned by the Technological University of Delft. The dynamical model used to predict the

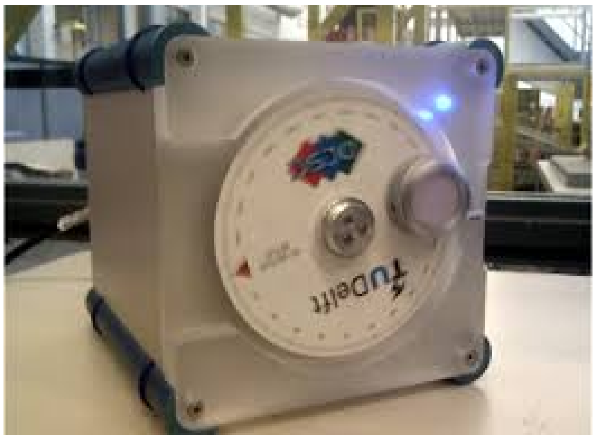


Figure 2-1: An image of the real Inverted Pendulum from which the dynamical model is extracted to be used in simulation.

response of the pendulum, corresponds to the exact dynamics of the pendulum (cite to be placed).

The Setup

The one degree of freedom underactuated pendulum is the main system of the thesis. Due to its underactuation characteristics and unstable equilibrium this low-dimensional system has a complex state-value function. With only two states required to control the model, the cost function of the IP can be easily plotted which makes this setup ideal for the thesis. The full motion of the pendulum can be described with two states, the position and velocity. The position of the Inverted Pendulum (IP) can be wrapped to a single loop with range 2π. Throughout the report the wrapping will be done either (a) from 0 to 2π radians, with 0 and 2π describing the angle of the lowest position or (b) from $-\pi$ to $+\pi$, without shifting the states (i.e. 0 describes the lowest position and $\pm\pi$ describe the upper position). The velocity range is set from -30 to $+30$ rad/s. The velocity range is greater than the maximum speed required to achieve swing up, which guarantees an extra margin for exploration. To actuate the pendulum a single input is used that is translated to the torque of the actuator with allowable range between $\pm 2V$. Figure 2-1 represents a picture of the real setup, whose properties were used to design the environment which the NN will be trained on.

Simulating a swing-up

One of the performance metrics that were used during the report is the success rate of the pendulum achieving a swing up and upper position stability. Throughout the report the initial position of the pendulum varied from the lowest position to multiple positions to cover uncertainty and noisy V-functions. To simulate the swing up, the dynamical model of the pendulum is used. Before introducing an actor network to provide continuous action, the input control used was selected from a discretised space. For each discrete action the next state was calculated, using the dynamic model of the IP. The next states were fed into the NN which predicted the state-value cost. Thereafter, the optimal action was selected using the equation $a^* = \text{argmax}V(s')$. The complete algorithm for the action selection can be seen in Algorithm 1. For the Reinforcement Learning implementation, where the actor-critic method was used, the action was predicted by the actor NN by providing the current states of the IP as inputs.

Algorithm 1: Selection aglorithm for optimal action a^* using discrete actions and the state-value function V

```
$sactions \rightarrow$ current state
$actions \rightarrow$ all discrete actions
$V \rightarrow$ empty list or vector with same length as actions
for $a$ in $actions$ do
    $s'$ = from_dynamic_model($s, a$)
    $V'$ = neural_network_predict($s'$)
    $V \leftarrow V'$ % appends predicted cost to the current list
end
$a^* = \text{argmax}(V)$
```

2-2-2 Magnetic Manipulator

The Magnetic Manipulator (MagMan) is the second low-dimensional continuous control benchmark used in the thesis. The magman uses coils and magnetic force to move a metallic ball to a specific position. The physical system, as shown in Figure 2-2, consists of four coils, however during the experiments in this thesis only the two will be used. Magman, similar to the IP, has only 2 observable states, the position and velocity of the ball and has two actions, the voltage input of the two controllable coils. The task of magman is to stabilize the metallic ball in-between the two coils starting from the same initial position. The reward is the absolute difference between the target and current position. During training, the states and actions are standardized and normalized between [-1, 1].

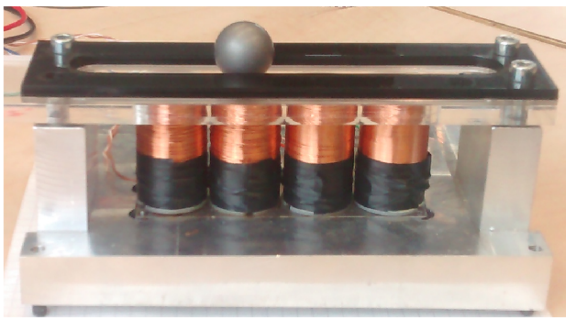
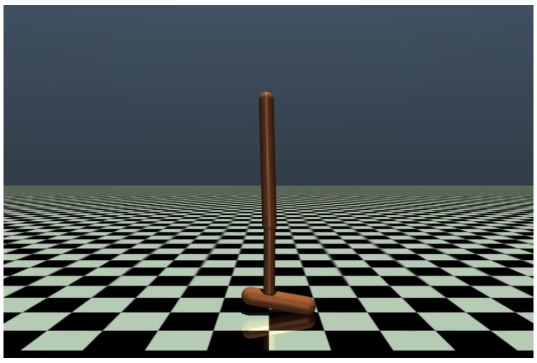


Figure 2-2: An image of the real Magnetic Manipulator from which the dynamical model is extracted to be used in simulation.

2-2-3 Hopper - v3

The Hopper-v3 (Hv3) environment is based on the MuJoCo physics engine and is a simulation of a 3 joint robot [31]. Hopper, shown in Figure 2-3 is a higher dimensional problem that is expected to learn hopping and moving forward. The Hopper has 12 observable states (6 positions and 6 velocities) and can be controlled by three joints. As the hopper is expected to learn to move forward with a similar and constant pace, the exact position on the map is not necessary for learning the correct policy. If the x-position is used as network input then it will introduce redudancy and increase trainable parameters to the network as the optimal action should be uncorrelated with the x-position. Such a network construction could reduce and make more difficult the NN training. Thus, for training a Neural Network, only 11 states are used.



position	#	translation
endpoint horizontal	0	linear
endpoint vertical	1	linear
joint 1 (endpoint)	2	angular
joint 2 (actuated)	3	angular
joint 3 (actuated)	4	angular
joint 4 (actuated)	5	angular

Figure 2-3: An image of the simulation of OpenAI Gym from the continuous control task; Hopper-v3. The three joint robot is trained to hop forwards while keeping its torso above a threshold value. The observable positions can be seen in the list, where joint 1 is not controllable however it is used to track the linear movement of the hopper. Position states 1-5 are used to train the neural network in addition to the corresponding 6 velocities of positions 0-6.

Reward

The reward function is a composed by the horizontal velocity, the control action and a boolean parameter (i.e. 0 or 1) which determines healthiness, as shown in Equation 2-1. The healthiness is satisfied when the upper joint of the hopper is above a specific height (i.e. guarantee that the hopper is still standing on its leg and has not fallen).

$$\mathcal{R} = \underbrace{x_{velocity}}_{\text{forward velocity}} + \underbrace{h}_{\text{healthiness}} \underbrace{-1e^{-3}\sum a_i^2}_{\text{control cost}} \tag{2-1}$$

Termination

The termination criteria occur either when the healthiness is not satisfied (i.e $h = 0$) or when the model is outside of the defined space. This way the hopper never experiences bad states, which means that if it reaches a state out of its healthiness it will not learn to recover. In addition, episodes have different lenghts with maximum lenght the time it will need to escape from the whole space. To counter these disadvantages, in some algorithms it is suggested that the environment is run for a fixed number of steps. A negative effect that can be presented by the fixed number of steps is that the hopper might learn to dive before the end of the episode to cover more ground and receive higher reward. Terminating by a fixed number of steps is not ideal for the specific environment as the healthiness cost in the reward function is usually not enough to guarantee proper form of hopping [32]. During our experiments the task is always terminated when the healthiness or environment boundary conditions are not satisfied.

2-3 Pre-Generated Data for Inverted Pendulum

The project was initialized on existing research, which provided material that could be used to kick-start the thesis. A previously used Basis-function (BF) value iteration method was used on the same setup to construct the state value function of the pendulum [33]. The data was generated on a 51×51 grid as shown in Figure 2-4. The BF method was used throughout the project to generate data for comparison by modifying the grid, reward equation and gamma variables appropriately.

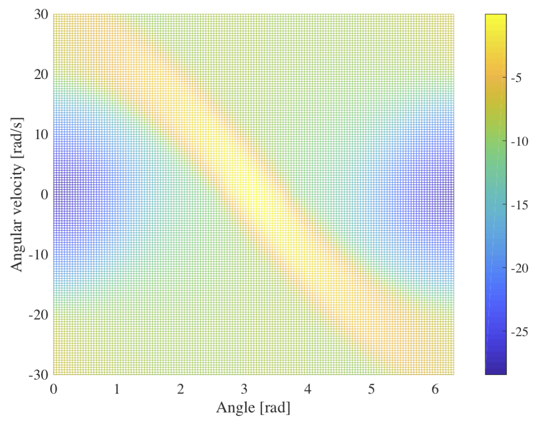


Figure 2-4: A representation of the BF value function of the 1-DOF pendulum. The graph is plotted on a grid of 51×51 points with respect to the angular velocity [-30,+30] and angular position $[0,+2\pi]$.

An example of the learned value-function can be seen in the simulation of a single episode where the pendulum attempts swing up from the lowest position. Figure 2-5 shows the states and reward during the swing up. The learned function is capable of achieving swing up within the given domain and can easily stabilise at the upper position with minimal control action.

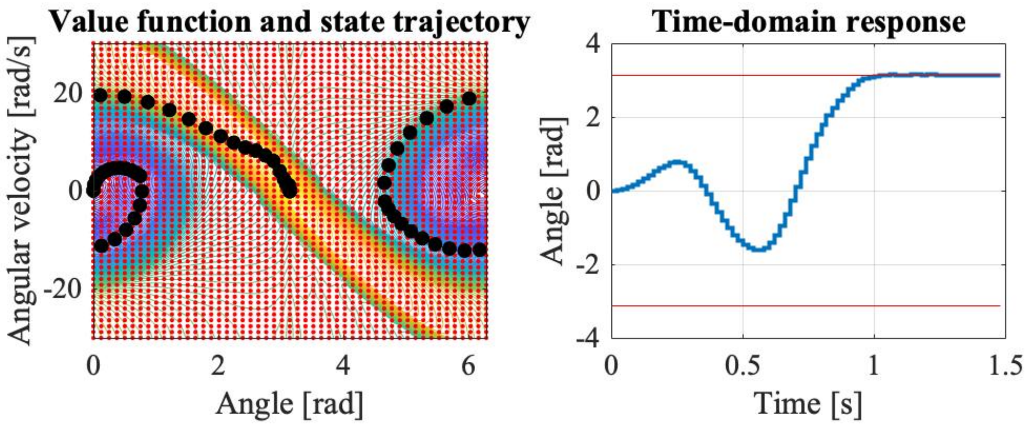


Figure 2-5: A representation of the trajectory of the pendulum with respect to the Value function and within a time domain. The pendulum using the BF Value function can achieve swing up within 1 second.

2-4 Visualization Techniques for Neural Networks

The section will introduce a series of concepts and performance tools used throughout the thesis in order to compare or collect insights of the Neural Networks (NN).

2-4-1 Neuron Visualisation

For a NN with two inputs, the predicted output can be plotted in a 3D graph [34]. The visualization of the approximated function can provide better insight to what the NN learned and where the learning failed [35]. An example is a NN that approximates the Value function of the IP with two states, position and velocity. The V-function, as seen in Figure 2-6, is the value of the output layer of the NN.

Similar to plotting the output function, the outputs of each hidden neuron can be plotted separately [34]. The possibility of visualizing neurons within the hidden layers can provide insight to the subfunctions and shapes learned by the NN during training. In the first hidden layer of either a shallow or deep NN, the output of invididual neurons are expected to be of the complexity of the activation function chosen for the layer. As seen in Figure 2-7 for the first hidden layer, a ReLU activation function will result in a piecewise linear function with clipped negative values at 0, while a tanh function will generate a smooth function between the values -1 and 1. The expressiveness of the function increases by going deeper to the hidden layers, with the neuron to be able to approximate more complex functions, those being either smooth or piecewise-linear dependending on the activation function. An example of a Deep Neural Network (DNN) with two hidden layers and a ReLU activation function can be seen in Figure 2-7 where each neuron is visualized invididually. The presented outputs are the results of the output of the neuron as a function of the network inputs, which takes into account the weights and biases of all previous layers. In fully-connected NNs, all the neurons of the previous layers are used to form the output of the new neuron based on the weights,

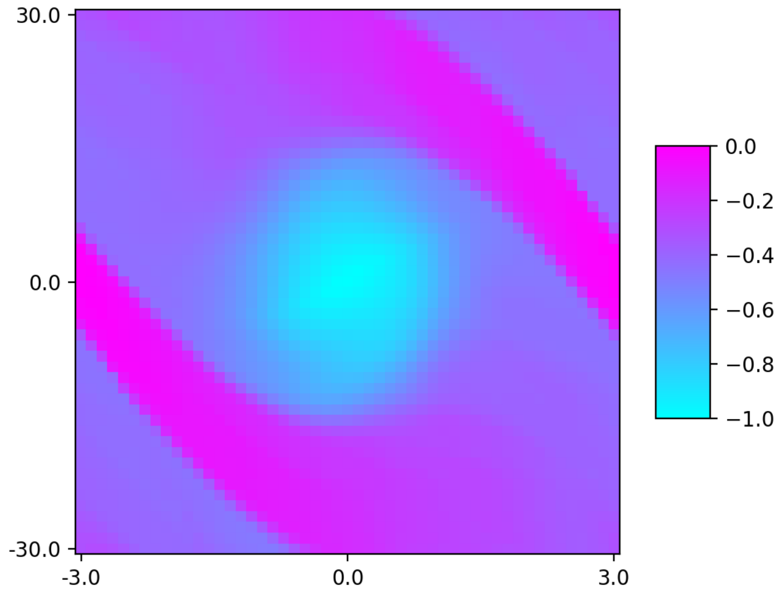


Figure 2-6: An example of the output function plotted by a NN with two inputs. Here, the Value function (i.e. output of the NN) is normalized between $[0, -1]$.

bias and activation functions. By visualizing each neuron as a function of the network inputs, all neurons can be compared and visualized side-by-side as they are expressed under the same inputs. Each neuron exact characteristics can be easily reflected to the neurons of the next layer and the output of the NN (which is in fact the visualization of the neuron at the output layer).

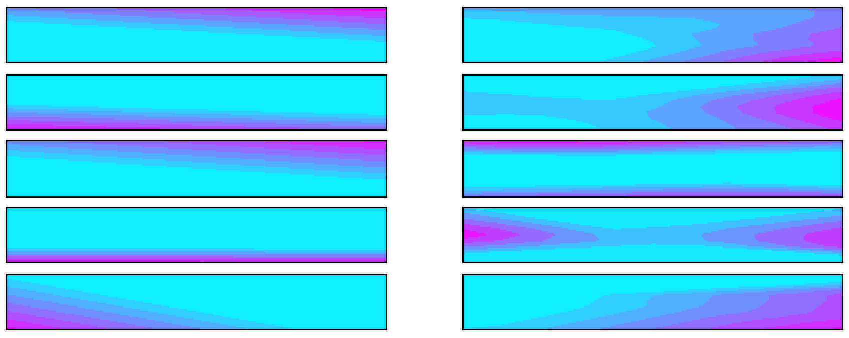


Figure 2-7: The plots represent 10 neurons from a two hidden layer NN architecture. The 5 neurons to the **left** are taken from the first hidden layer with a relu activation function, which result to a simple linear function with the negative part being clipped to zero. The 5 neurons to the **right** are extracted from the second hidden layer of the DNN and can be seen to have a more complex architecture, however given that the activation function remains relu, the function is also piece-wise linear. Depth increases the complexity the function that a neuron can approximate.

2-4-2 Activation Visualisation

Visualising the output of each neuron provides insight to the learned subfuctions of the NN and its expressiveness [35] [36]. Based on the width of the NN, it is impractical to visualize each neuron individually. An alternative method of visualization could be the visualisation of all the activation functions. By observation of the hidden layer neuron outputs, it can be noted that for low depth networks (1-3 hidden layers) with ReLU activations, the learned function can be captured by the activation transition (ActTran). The activation transition is defined as the neuron output function $f(x_1, x_2) = 0$ with x_1, x_2 being the inputs of the NN with a ReLU activation. The ActTran exist where the neuron output transitions from zero to positive values. In other terms, each neuron has an output function $f(x) = 0$ where a slight change in the states x would output values other than zero.

$$\begin{aligned} \text{ActTran@ } & f(x_1, x_2) = 0 \\ \text{for } x_1, x_2 \Rightarrow & f(x_1 + e, x_2 + e) = 0^+ \end{aligned} \tag{2-2}$$

For a neuron in the first hidden layer the ActTran corresponds to a linear function. The function is constructed by the inputs x_1, x_2, their weights and the neuron bias. The equation can be seen in 2-3.

$$f(x_1, x_2) = (w_{1_i}, w_{2_i}) \begin{pmatrix} x_1 \\ x_2 \end{pmatrix} + b_{n_i} \tag{2-3}$$

For a neuron after the first hidden layer the ActTran function becomes piecewise-linear and is composed by all the neuron outputs of the previous layer, the new weights and the current neuron bias. An example of the ActTran in the first and second hidden layer can be seen in Figure 2-8. For the second hidden layer, the ActTran changes direction when it interacts with an ActTran from the previous layer. This represents the importance of the first hidden layer and its effect in the additional hidden layers.

This representation concludes that additional hidden layers can represent more complex functions however they are always influenced by the neurons of the first hidden layer. The visualization technique can help us investigate under which conditions the NN learns meaningful representations and when the depth becomes a computational burden without influencing the NN capacity.

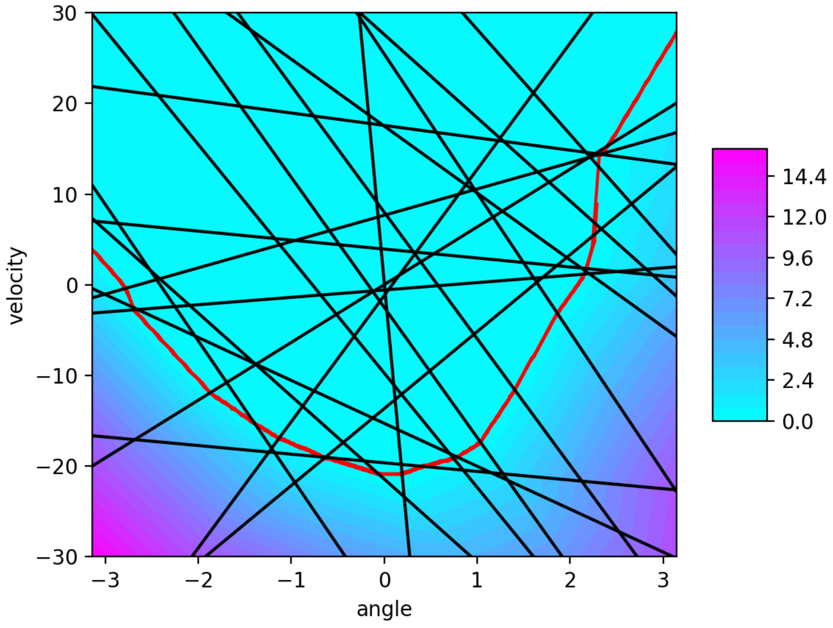


Figure 2-8: The graph displays the activation transitions of a single neuron from the second hidden layer. The black lines express the activation transitions of the previous layer, while the red line is the activation transition of the presented neuron. When the red lines meets a black line, it changes the trajectory of the red lines. This shows the impact the previous layer has on the next layers. By increasing the layers, a more expressive neuron can be generated, nevertheless the first layer is the foundation of the NN.

2-5 Shallow and Deep Architecture Selection

The section will introduce a Supervised Learning implementation on the IP. The goal of the section is to identify the architectures of NNs that can fit the Value function of the pendulum. The V-function data is already collected from a basis-function value iteration method and is used for training the NNs. The section will conclude on the optimal NN architecture for both a Shallow and Deep NN, with respect to trainable parameters, computational time and generalization error. The section is the first stage of the conversion from Supervised Learning to Reinforcement Learning.

The data used in this section was generated from the BF method, Section 2-3. The grid used was a 151×151 with approximately 23K samples. The data, including the position, velocity and state-cost value, was shuffled to reduce correlations between the samples. When less data was required, the data was randomly selected from the 23K sample pool.

2-5-1 Training the Neural Network

The NN was constructed to fit the regression problem. Following common standards, the data was divided into training and testing subsets using 80% and 20% samples respectively.

The training data was further split 75% and 25% for training and validation. The NNs were trained using mean-absolute-error (MAE) loss function and adaptive moment estimation (Adam) [37] [38]. An additional metric value of root-mean-squared-error (RMSE) was used for observation and comparison towards MAE. RMSE will penalize larger variance between the data, compared to the MAE. By observing both methods it can be concluded how evenly distributed the error is throughout the samples [39]. The network inputs were representing the angular velocity and angle of the pendulum. The output was representing the expected return of the state value function. The hidden layers had ReLU activation functions. The weights were initialized from a glorot distribution set which was reported to improve learning of ReLU functions [40] [41]. The activation function and optimization method selection was based on initial experiments which are summarised in Appendix A-1.

With only the depth and width being the undefined variables in the NN architecture, the NN was trained on the V-function regression problem. The hidden layers varied from 1 to 4 while varying the number of hidden units per layer. The training was stopped after the validation loss did not improve for 10 consecutive epochs. This method will be later referred to as patience stopping criteria with a patience of 10. The results from multiple runs can be seen in Table 2-1. The table shows the hidden layers and units alongside their final loss (MAE) and RMSE. The loss values seem to be low for all networks which suggests that the NN is capable of learning the required V-function. Nevertheless, the results point towards a problematic stopping criteria which is further discussed in the next section 2-5-2.

Table 2-1: Experimental results of various NN architectures trained to approximate the value function of the 1DOF pendulum. The results include both the loss value obtained using MAE and the metrics RMSE captured at the last trained epoch. The loss seem to be inconsistent with the increase of parameters which suggests a bad use of stopping criteria. Nevertheless, NNs with 2-4 hidden layers seem to have better performance compared to the shallow NNs.

Neural Network Architectures			Training		Testing	
H. Layers	H. Units/layer	Parameters	Loss	RMSE	Loss	RMSE
1	50	201	0.5497	0.7685	0.5454	0.7606
1	100	401	0.6781	0.9890	0.6923	0.9926
1	200	801	0.3463	0.5107	0.4580	0.6050
2	50	2751	**0.2181**	0.2908	0.2297	0.3102
2	100	10501	**0.2667**	0.3544	0.2424	0.3135
2	200	41001	0.1769	0.2379	0.1557	0.2042
3	200	81201	0.2250	0.3059	0.2419	0.3116
4	(200,100,50,50)	28351	0.2323	0.3157	0.2588	0.3721
4	(50,100,200,100)	45651	**0.1615**	0.2292	**0.1404**	0.1991
4	200	121401	0.1784	0.2511	0.1750	0.2525
4	1000	3007001	0.2475	0.3567	0.2601	0.4110

From Table 2-1 it can be observed that all deep NNs examined have loss less than 0.3 while the shallow NNs were unable to generalize as good. This could be explained by the increase of parameters with the addition of hidden layers. To investigate this theory, a comparison is done between a shallow and a two hidden layer NN with the same total parameters. A NN with two hidden layers of 200 units each is compared (= 41001 trainable parameters) with a shallow network of 10 thousand hidden units (= 40001 trainable parameters). Both networks are trained on 5000 epochs and tested on the whole data-set (2601 samples). The training loss after each epoch can be seen in Figure 2-9. This shows that increasing depth by 1 hidden layer can reduce the generalization error better than exponentially increasing the width. With the current experiment it can be concluded that the target function is complex enough to challenge the expressiveness of a shallow NN. However, it will be later shown that lower loss of the SNN could suggest worse fit on the approximated function but does not necessarily worsen the pendulum performance in simulations.

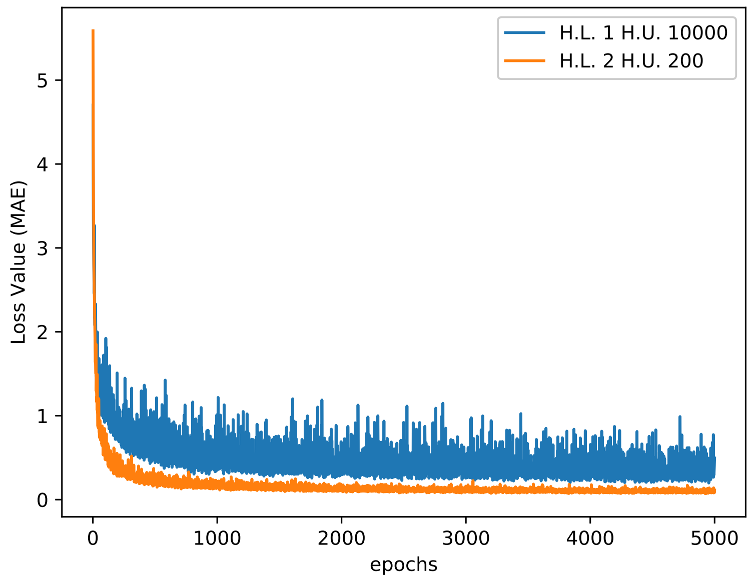


Figure 2-9: A comparison between different NN architectures with same trainable parameters. The shallow network performs worst with more fluctuations of the loss value compared to the deeper network, even if they have the same number of trainable parameters. The loss value represents the loss of the network when tested on a larger dataset of the V-function.

2-5-2 Stopping criteria

Bad stopping criteria can affect the learning of the NN. Before experimenting further with the NNs it is necessary to understand how the stopping parameters affect learning. Stopping the training too early can result in under-fitting while training for too long can result in over-fitting of the function. From section 2-5-1, the NNs trained with a patience of 10 on the validation loss showed inconsistency on results as bigger NNs performed worse than NNs with less parameters. Table 2-2 repeats the training while it observes the epoch which it was stopped. The premature stopping can be seen from the 3 hidden layer NN compared to the NN with 2 hidden layers.

Table 2-2: Experiments with varied hidden layers on a large sample data (22K samples) and early stopping.

Neural Network Architectures			Training		Testing		Stopped Epoch
H. Layers	H. Units/layer	Parameters	Loss	RMSE	Loss	RMSE	#
1	200	801	0.3771	0.5330	0.3814	0.5328	223
2	200	41001	0.2043	0.2755	0.2199	0.3351	49
3	200	81201	0.2650	0.3584	0.4243	0.5586	17
4	200	121401	0.2062	0.2871	0.3019	0.4535	34

The validation loss of the NNs from Table 2-2 is plotted in Figure 2-10 (Left). The stopping criteria evaluates the validation loss after each epoch and stops training when the validation loss does not improve for 10 consecutive epochs. Random weight initialization and the stochasticity of the optimizer can result in the criteria being met prematurely. Figure 2-10 (Right) shows one NN with 4 hidden layers and 200 units being trained 5 times with random weight initialization and patience stopping criteria.

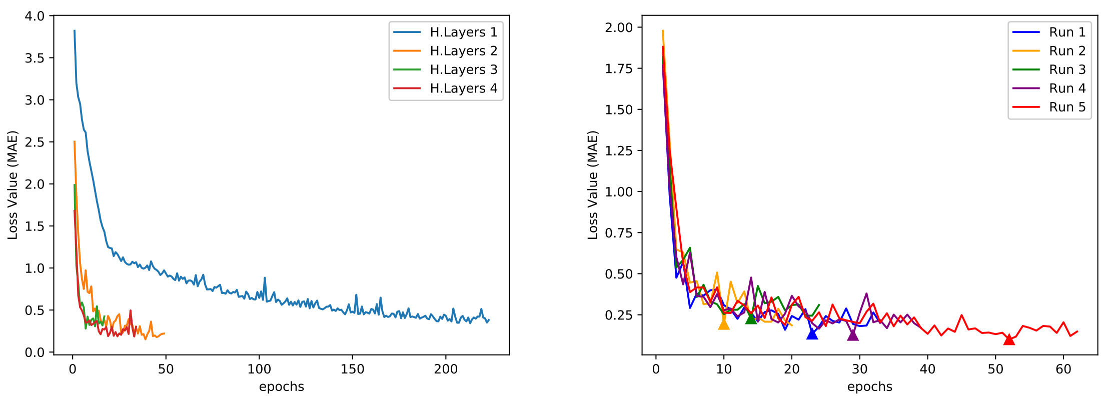


Figure 2-10: (Left) The validation loss values for each epoch of four neural networks with 200 hidden units per layer and varied number of hidden layer from 1 to 4. It can be seen that the shallow network takes the longest to stop however it achieves worse loss value than its deeper counterparts and has the slowest convergence. (Right) Multiple runs of the same 4 hidden layer NN with patience stopping criteria and random initialization. The NN terminates training at different instances. Longer training slowly converges to a lower loss value.

A premature termination of training will result in higher generalization error. This suggests the removal of the stopping criteria. The Figure 2-11 shows four NNs with 200 hidden units per layer and hidden layers from 1 to 4 trained for 1000 epochs. Non of the networks overfit on the data and all NNs are converging. The shallow NN (1 hidden layer) performs slightly worse than the deeper networks. The shallow NN converges (or slowly decreases) to a higher validation loss value than the deeper NNs. The results suggest that a NN with 2 hidden layers can fit faster than a SNN on the V-function given the selected training parameters. Deeper NNs can fit with similar accuracy to the 2 hidden layer DNN, thus their depth is considered redundant.

The patience stopping criteria shall not be included in the training of the NNs as they can

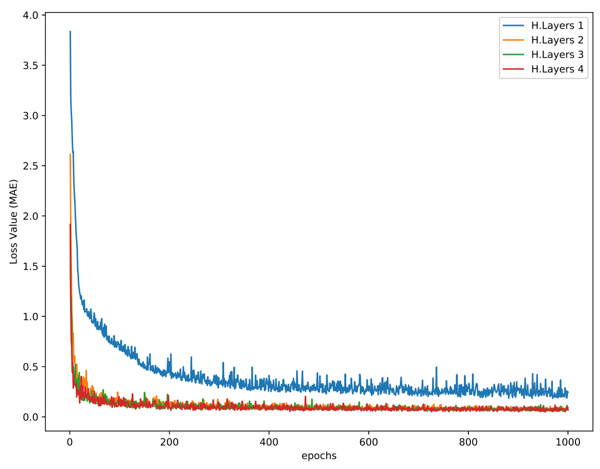
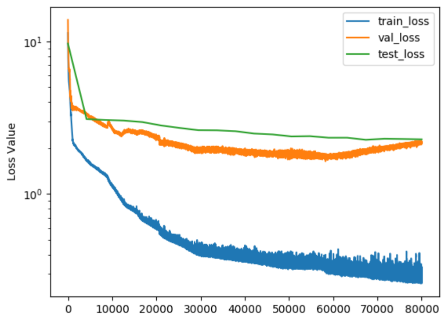


Figure 2-11: The validation loss value of different NNs with 200 hidden units per layer and varied depth. The training was run for 1000 epochs without early stopping option. The shallow network perform worse than the deeper networks with higher loss value and more fluctuations. The NNs with hidden layers 2 to 4 seem to have similar performance and convergence time.

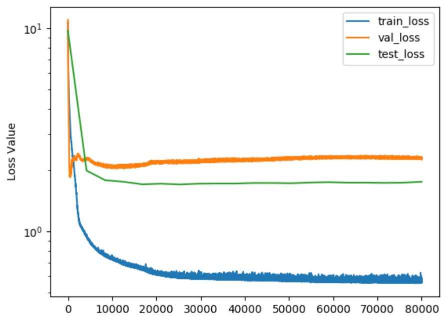
result in premature stopping. Alternatively, the NNs could be trained on a fixed number of epochs or until a threshold in validation loss is achieved. From the figures presented in this section, the over-fitting of the NN was not observable. The reasons why over-fitting is not visible in the loss graph and why the stopping criteria selected was a fixed epoch will be presented in Section 2-5-3

2-5-3 Overfit

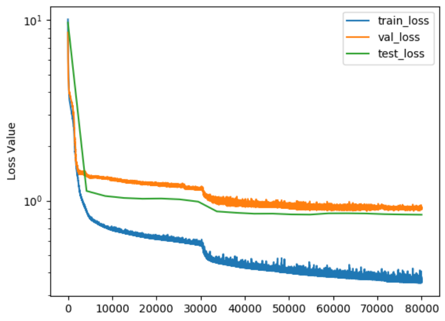
A well-trained network exists when it reaches its optimal capacity [42]. One way to visualize the optimal-fit is by over-training the network to observe the transition from under-fitting to over-fitting. From Figure 2-9 of the previous section, it can be seen that the NN does not overfit on the data. This could be explained by the complexity of the value-function and the high sampling data. In order to confirm the assumption, the NN was trained on much less samples to guarantee an overfit. Figure 2-12, shows the logarithmic loss of a NN trained on 50, 100 and 250 samples. With only 50 samples the value-function is not sufficiently expressed which explains the gap between validation and training loss. On epoch 60,000 the validation loss is increasing while training loss is decreasing and the NN is over-fitting on the training data. When the NN is trained on 100 samples the over-fitting is not as visible, but it could be seen after the 20,000 epochs. Training on 250 samples, causes the validation loss to follow the same curve with the training loss but with higher loss values. The results from this experiment suggest that the more sampling data used, the better the value-function can be represented.



(a) 50 samples



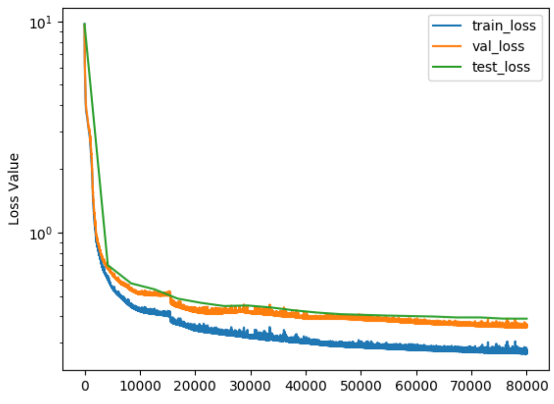
(b) 100 samples



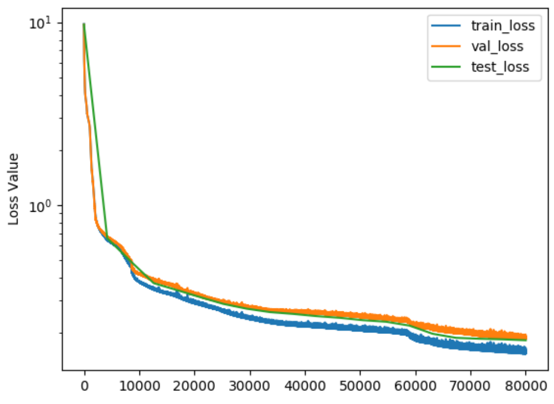
(c) 250 samples

Figure 2-12: Logarithmic loss of a NN with two hidden layers and 20 hidden units per layer. The NN is trained on (a),(b),(c) samples for 80,000 epochs where each epoch is one gradient update. Unlike the other experiments the mini-batches were removed from training due to the small sampling sizes used. From the loss curve, under-fitting and over-fitting can be clearly distinguished. However the loss value is not low enough, which suggests that the NN does not actually learn the function. The test loss is calculated on the remaining samples out of the 22 thousands samples and only on 20 epochs throughout training.

With enough sampling data and a complex function, the NN is less likely to overfit. As already shown increasing the sampling data reduces the gap between validation and training loss and minimizes over-fitting. Figure 2-12 confirmed that 250 samples were enough for the validation and training loss to have similar loss curves but the magnitude of loss was different. Under-fitting could exist when the training data is not enough. Figure 2-13 shows the loss when training the NN with larger data-sets. While some over-fitting can be still observed, the density of data does not allow the NN to learn something completely wrong. Training on 5000 samples reduces the loss between training and validation without visualizing over-fitting. Due to the functions complexity and gridded sampling, the NN is potentially learning the grid pattern thus why over-fitting could be taking place without being visualized.



(a) 1000 samples



(b) 5000 samples

Figure 2-13: Logarithmic loss of a NN with two hidden layers and 20 hidden units per layer. The NN is trained on (a),(b) samples for 80,000 epochs where each epoch is one gradient update.

Over-fitting a NN can be achieved by increasing the trainable parameters of the network with respect to the sampling data. In the latest experiment the data was 10 times higher than the trainable parameters without any over-fitting being observed (i.e. 5000 data on a NN with 501 parameters). In an attempt to visualize over-fitting we train a NN with 200 neurons per hidden layer (41,000 parameters) on the 5000 data. The NN achieves lower loss in less epochs without over-fitting, as shown in Figure 2-14. Thus the size of the network is not the reason for the lack of over-fitting. This concludes that the function is complex enough or that the grid is structured for the network to "visually" overfit. Another explanation can be the implicit regularization of the NN caused by gradient decent methods as suggested in [43].

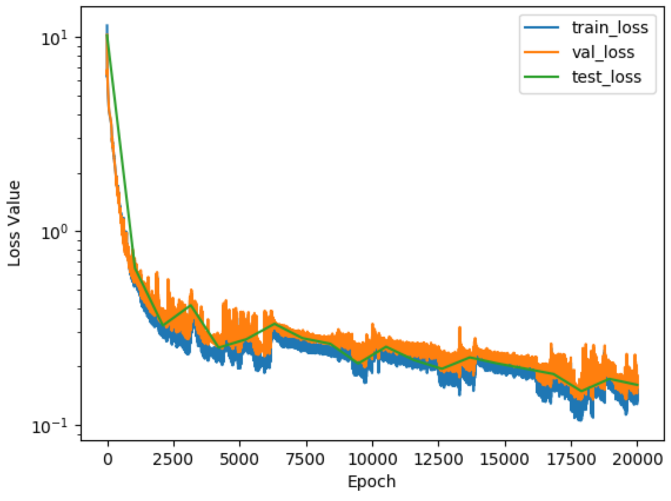


Figure 2-14: Logarithmic loss of a NN with two hidden layers and 200 hidden units per layer. The NN is trained on 5000 samples for 80,000 epochs where each epoch is one gradient update. The validation and training loss values are very close and no over-fitting is observed.

The training must be stopped on the perfect-fit. As discussed above, stopping criteria could dramatically change what the NN learns. Using a patience criteria has already been discarded due to the unpredicted fluctuations of the loss between epochs. An alternative method is to use a benchmark value before taking into account the patience criteria. From this section it can be concluded that, if the validation loss is lower than 0.2, then it is a good indication that the NN is actually learning. The constant 0.2 can thus be used as a benchmark before applying the patience criteria and introducing early stopping. In addition, as over-fitting is unobservable with large sampling data, a large number of epochs can also be considered as viable learning method as long as the sampling data-set is large.

2-5-4 Computational Time Comparison

In order to compare the computational time between neural networks of different depth, a simple experiment was run that investigates the time required to complete training. To allow fair comparison, the same trainable parameters have been set for NNs with hidden layers from 1 to 4. The comparison was done by training each NN on the same sampling data for increasing number of epochs from 1 until 100. The results can be seen in Figure 2-15. The computational time between shallow and deep networks does not vary which suggests that in such a small increments of depth, computational complexity is not easily visible and does not have a great impact.

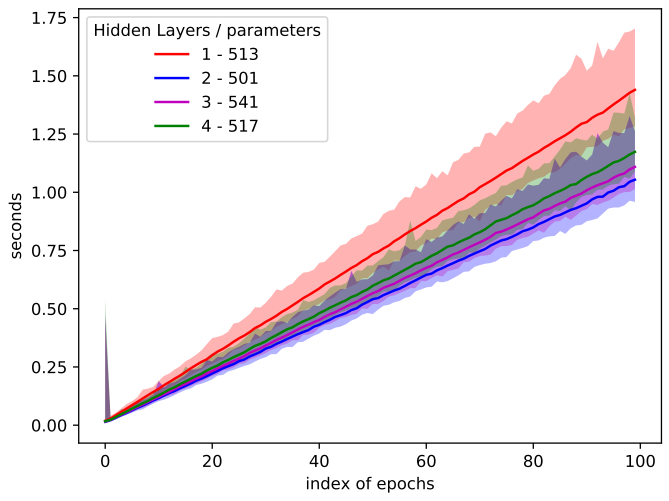
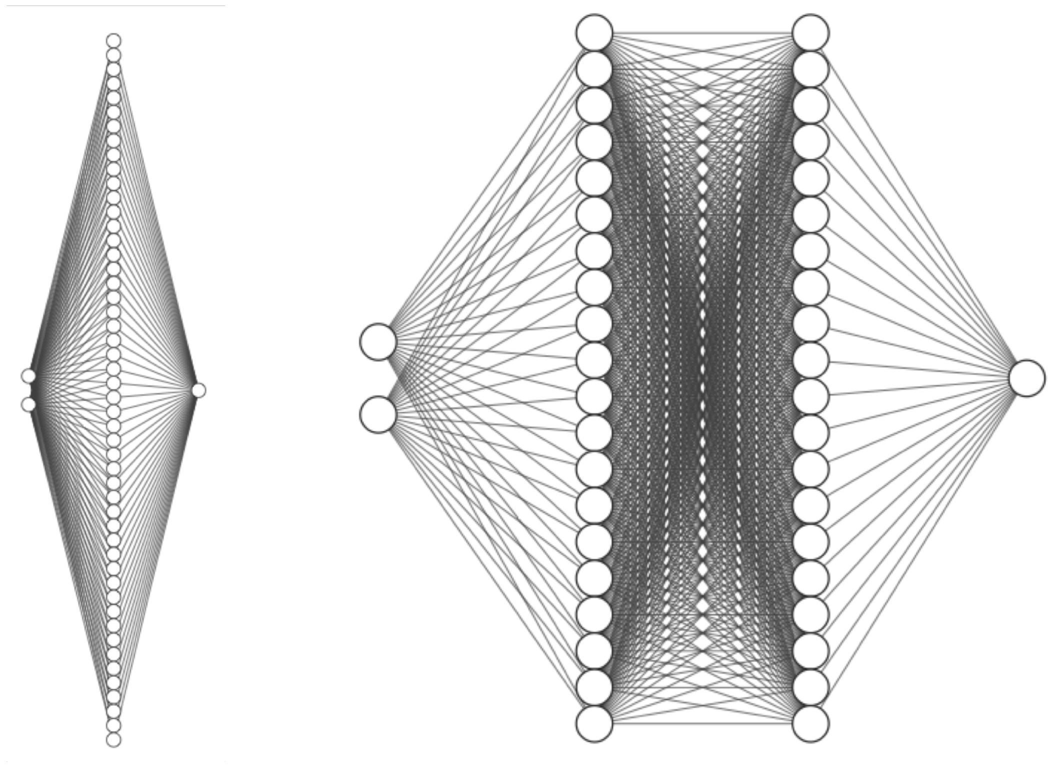


Figure 2-15: A comparison between the computational time of NNs with hidden layers from 1 to 4 and approximately the same parameters. The increase of depth does not affect the computational time using vanilla SL.

2-5-5 Architecture Selection

Having gained the necessary knowledge to train the NN, the final architectures were selected. For the representation of shallow NNs, the width of the network was minimized to 128 neurons (513 parameters). The number of trainable parameters was selected as the minimum width that can learn the target function. It was already shown that the SNNs were unable to reach as low loss values as the DNNs, however they were capable to learn the value-function with enough accuracy so that the pendulum would achieve swing up. For the representation of the

DNN, it has already been noted that increase of depth more than two layers does not reduce the generalization error and simply increase computational costs. Due to this conclusion, a two hidden layer NN was selected with 20 neurons per hidden layer (501 parameters). Similar to the SNN, the architecture was aiming at low loss, consistent performance and almost the same parameters with the SNN. Both NN designs can be seen in Figure 2-16.



(a) Shallow Neural Network **(b)** Deep Neural Network

Figure 2-16: The selected NN architecture for both Shallow (a) and Deep (b) Neural Networks. The SNN with 128 neurons on the hidden layer has a total of 513 trainable parameters. The DNN with 20 neurons per each hidden layer has a total of 501 trainable parameters.

2-6 Summary and Concluding Remarks

The chapter introduces the basis of the thesis. Some terminologies of Deep Learning are explained in order to establish common vocabulary. In addition, visualization tools and benchmarks are presented. An initial examination of the NN performance is shown, where using vanilla-SL, the deep and shallow NN architectures are evaluated and selected for minimal training parameters and successful learning on the IP benchmark.

Chapter 3

Tracking Value Iteration

The chapter introduces the concept of learning a target value function by tracking function changes from existing data. By changing the target function, the NN is challenged in the adaptability to new target data by moving from simpler to more complex functions. The goal of the Tracking Value Iteration method (TVI) is to observe any changes in capacity, expressiveness and overall performance of the NN compared to learning the target function directly (i.e. as shown in the SL Section 2-5). The chapter is the second stage of the conversion from Supervised Learning to Reinforcement Learning.

3-1 Tracking Value Iteration Dataset

The data used to train the NNs was generated by the BF method. The BF method converged within 115 iterations, whose sampling data was used to train the NN. The first iteration of the BF corresponds to the reward function while the last iteration (115) is the exact same data used to train the NN in Section 2-5, using SL. The evolution of the function can be seen in Figure 3-1. From the Figure it is possible to observe that the function transitions within the first 30 iterations and converges around iteration 50. From iteration 50 till iteration 115 there are only very small adjustments to the function.

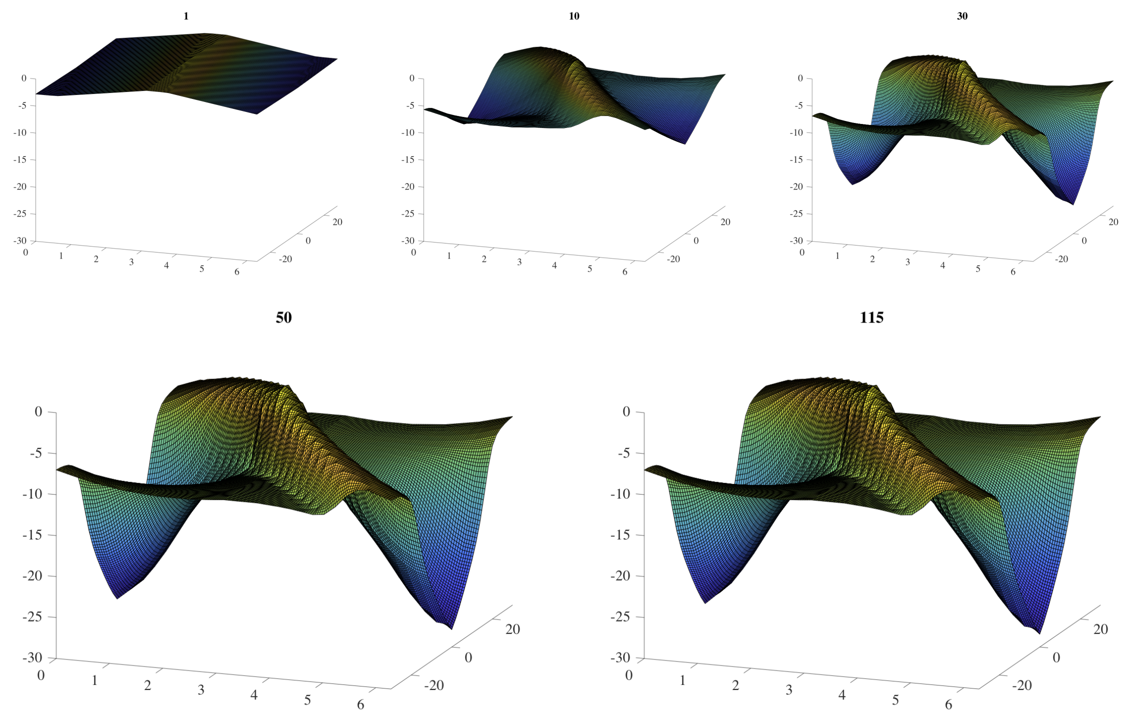


Figure 3-1: Value function through iterations using BF method. The value function has almost converged at iteration 50 while at iteration 30 the general shape of the function is distinguishable.

3-2 Analytical comparison between SL and TVI

The deep and shallow NN architectures, chosen in Section 2-5-5, were trained using the TVI data. To compare the TVI method with the vanilla SL method, the NNs were trained on the same amount of sampling data (i.e. 2601 samples randomly selected from a 151×151 grid for each BF iteration).

3-2-1 Deep Neural Network

The DNN was trained for 30 epochs per iteration in order to achieve good function approximation at each iteration. To compare the vanilla SL method with the TVI method, the vanilla SL NN was trained on the same target data 30 epochs per time, after which training was paused to evaluate the network. The test loss for both SL and TVI can be seen in Figure 3-2, where the NNs were tested on the whole dataset of the last iteration. With the TVI being trained on different target functions, the higher initial loss in the first iterations is expected. With enough epochs for the DNN to learn effectively each function per iteration, is able to track the changes of the function and achieve lower loss than the SL method. The TVI method converges to the lower loss value than SL by iteration 50 which shows good tracking of the function.

From the test loss, it is clear that SL method is faster as it directly learns on the required function while the TVI is guided through a changing target function. In order to observe

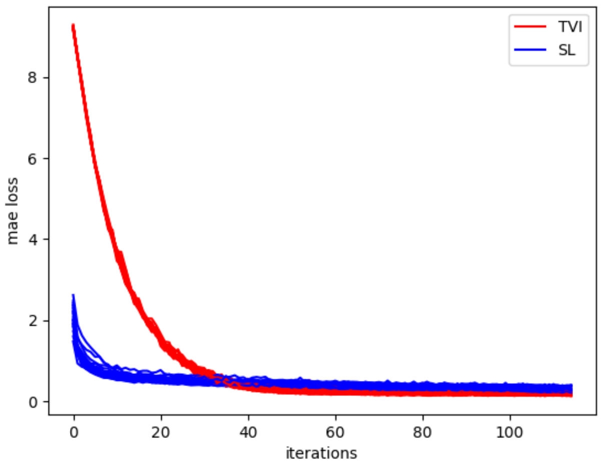


Figure 3-2: Test MAE loss per iteration. TVI and SL DNNs are tested on the last iteration of the V-function. The experiment was run 20 times and the loss for each was recorded. TVI converges consistently at iteration 50-60 (1500-1800 epochs). This shows that the TVI was able to follow the transitions of the V-function per iteration. SL converges around iteration 10 (300 epochs), as it is trained directly on the final function. By tracking the changes of the function, the TVI NN is able to achieve lower loss value than the SL NN.

the learned differences of the two networks, the value difference between the expected and predicted V-function is plotted on Figure 3-3. From the difference surface it is safe to conclude that both NNs learn similar V-function and their critical failing point is on the steep surface of the V-function. Therefore, the limitation of learning comes from the NN architecture and not from the methodology.

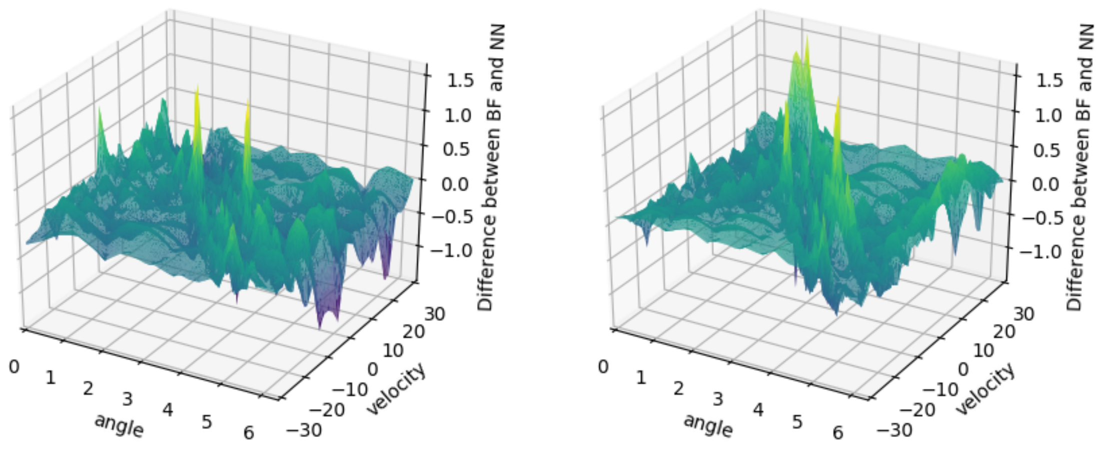


Figure 3-3: Left: TVI Right: SL. The difference of V-function approximated by the NNs and target function ($V_{diff} = V_{predicted} - V_{true}$). The error is oscillating at 0 with some high picks around the area where the value function is very steep.

An additional performance and comparison measure can be achieved from a swing up simulation as described in Section 2-2-1. For a simple comparison the pendulum was initiated from the lowest position without initial velocity. Figure 3-4 shows the states and the control input (action) generated using the V-function of the BF method, SL-NN and TVI-NN. The NN methods achieve swing up at similar time and are slighty slower than the BF method. The pendulum is stabilized with minimum oscillations and small control inputs. The direction of the swing up is based on the accuracy of approximation near the lower states while the stabilization at the top is depended on the accuracy near the peak at $[\pm\pi, 0]$. The difference in performance between the BF and the NNs suggests that the NN does not learn the optimal V-function but a slightly sub-optimal one.

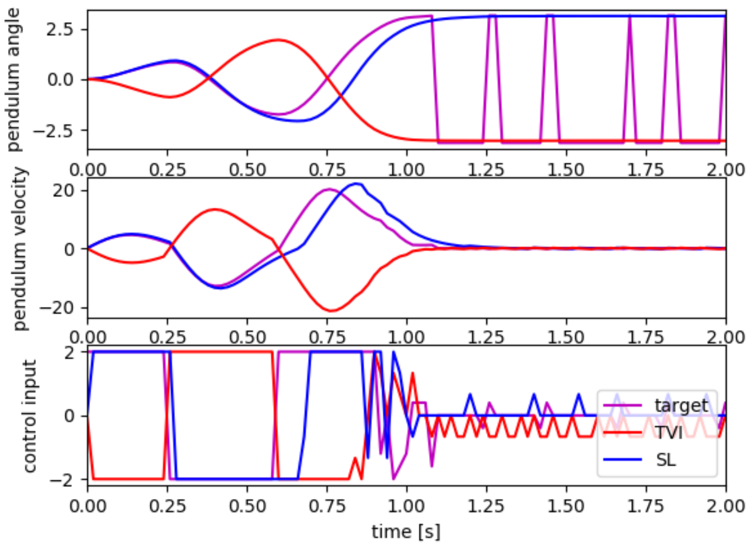


Figure 3-4: Time-response and control input of the IP using the target V-function, TVI and vanilla SL methods. With state initialization at the lowest position both NN methods could accomplish a successful swing up. Comparing the NNs to the BF response, it can be seen that the actions followed from the pendulum were not the optimal.

3-2-2 Shallow Neural Network

To allow direct comparison with vanilla-SL, the SNN was chosen to train for a total of 30 epochs per iteration. This sums to a total of 3450 epochs equally distributed along the 115 iterations. The test loss for both SL and TVI can be seen in Figure 3-5, where the NNs were tested on the whole data of the last iteration. With the TVI being trained on different target functions, the higher initial loss in the first iterations is expected. With enough epochs for the SNN to learn effectively each function per iteration the network converges to the same loss value by iteration 50. As the target function for the SL was unchanged, the network converged faster on the desired function with higher loss than the TVI.

Even with a slightly difference in the loss value between the vanilla-SL and TVI method, both methods train the NN well enough to achieve a swing up from the lower position. The

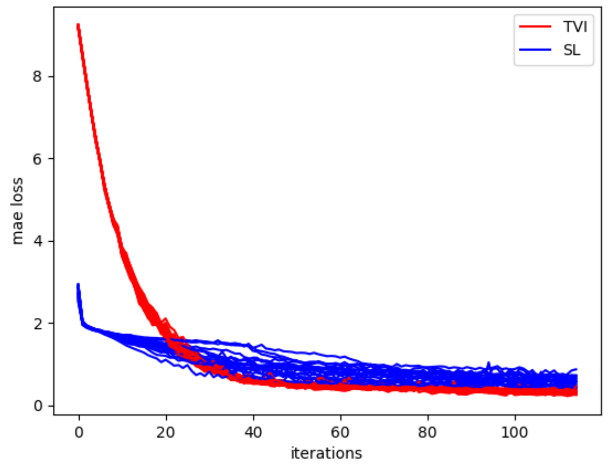


Figure 3-5: Test MAE loss per iteration. TVI and vanilla-SL SNNs are tested on the last iteration of the V-function. The experiment was run 20 times and the loss for each was recorded. TVI converges consistently at iteration 50-60 (1500-1800 epochs). This shows that the TVI was able to follow the transitions of the V-function per iteration. SL converges around iteration 10 (300 epochs), as it is trained directly on the final function. By tracking the changes of the function, the TVI NN is able to achieve lower loss value than the SL NN.

simulation for both methods in comparison to the simulation of the target function from the BF method can be seen in Figure 3-6.

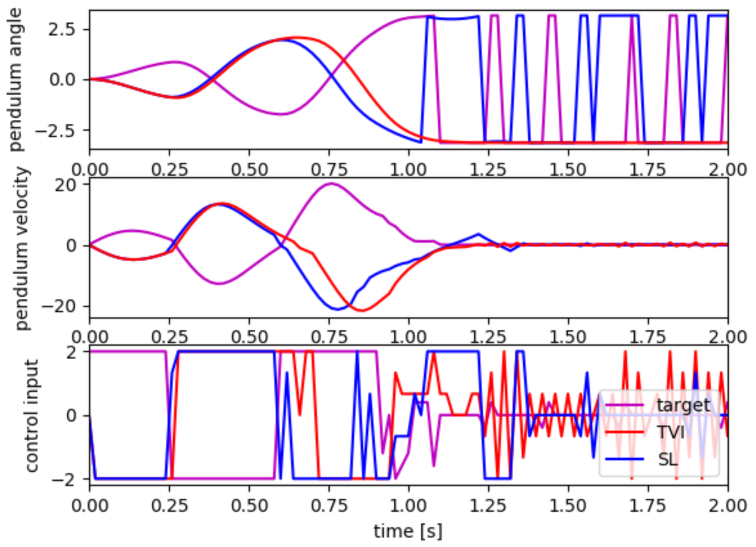


Figure 3-6: Time-response and control input of the IP using the target V-function, TVI-SNN and vanilla SL-SNN. With state initialization at the lowest position both NN methods could accomplish a successful swing up. Comparing the NNs to the BF response, it can be seen that the actions followed from the pendulum were not the optimal.

3-2-3 Increasing the trainable parameters/width

As discussed in Section 2-5, the current NN architectures are designed to minimize the parameters while maintaining good approximation. In theory, increasing the parameters of the NNs can increase the approximation accuracy even further. In the previous sections, the simulation comparison between BF and NNs concluded in the NNs being sub-optimal. The goal is to increase the parameters of the NNs in an attempt to improve the swing up speed. The new NN increases the width of the hidden layers to 64 neurons per layer (4353 parameters) and 1024 neurons per layer (4097 parameters) for the deep and shallow network respectively.

Deep Neural Network

The TVI-DNN was trained for 30 epoch per iteration for a total of 115 iterations as in the previous experiment .The test loss (MAE) towards the last iteration can be seen in Figure 3-7. The loss of the wider NNs is similar to the loss of the thinner NNs in Figure 3-2. By looking into the time-response of a swing up simulation, in Figure 3-8, both NNs achieve sub-optimal performance from the target method.

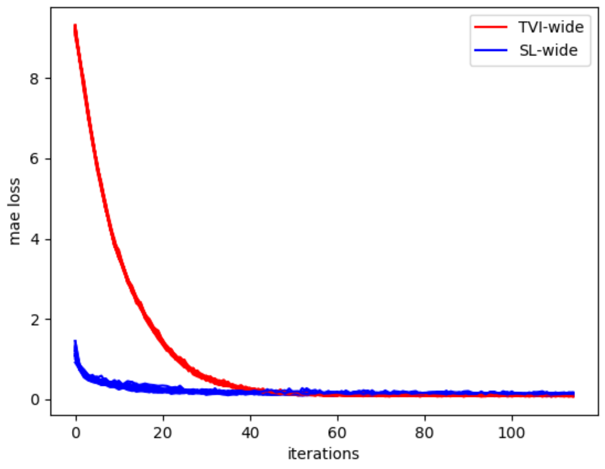


Figure 3-7: Test loss per epoch/iteration. SL-DNN-wide and TVI-DNN-wide are tested on the last iteration of the V-function. TVI converges at epoch 40-50 (one epoch per iteration). This shows that the TVI was able to follow the transitions of the V-function per iteration. Vanilla SL converges around epoch 30, as it only learns on the last iteration. The learning curves are similar with those from the thinner NN in Figure 3-2.

Shallow Neural Network

The TVI-SNN was trained for 30 epoch per iteration for a total of 115 iterations as in the previous experiment .The test loss (MAE) towards the last iteration can be seen in Figure 3-9. The loss of the wider NNs is similar to the loss of the thinner NNs in Figure 3-5. By looking into the time-response of a swing up simulation, in Figure 3-10, both NNs achieve sub-optimal performance from the target method.

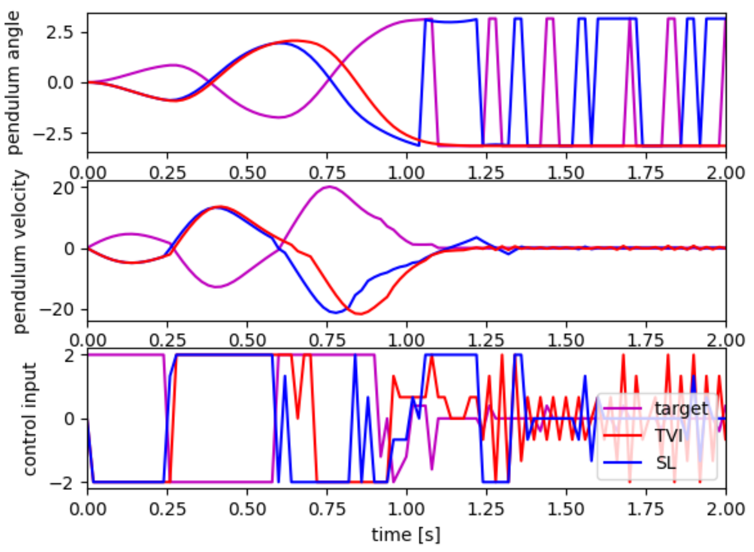


Figure 3-8: Time-response and control input of the IP using the target V-function, TVI and vanilla SL methods with wider NNs. With state initialization at the lowest position both NN methods could accomplish a successful swing up. The performance of the NNs does not improve with the increase of parameters.

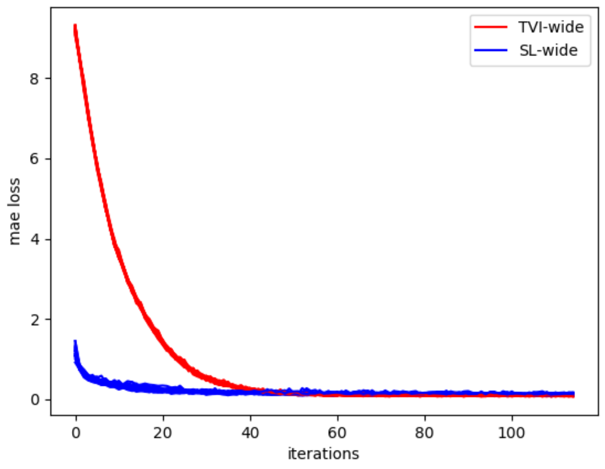


Figure 3-9: Test loss per epoch/iteration. SL-SNN and TVI- SNN are tested on the last iteration of the V-function. TVI-NN converges at epoch 40-50 (one epoch per iteration). This shows that the TVI-NN was able to follow the transitions of the V-function per iteration. SL-NN converges around epoch 30, as it only learns on the last iteration. The learning curves are similar with those from the thinner NN in Figure 3-5.

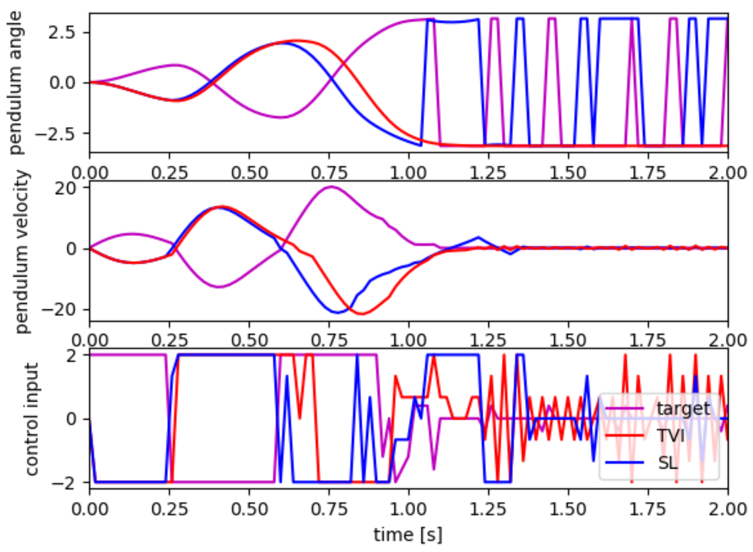


Figure 3-10: Time-response and control input of the IP using the target V-function, TVI and vanilla SL methods with wider SNNs. With state initialization at the lowest position both NN methods could accomplish a successful swing up. The performance of the NNs does not improve with the increase of parameters.

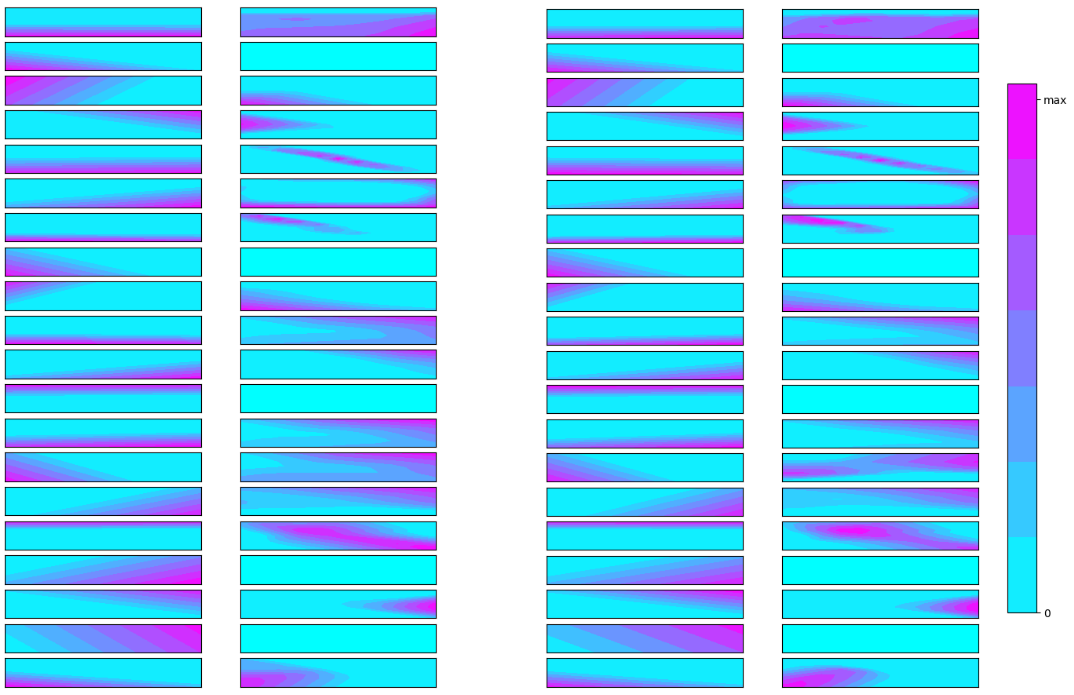
3-3 Visual comparison of parameters between SL and TVI

Identifying internal difference between NNs with similar approximation accuracy requires visualization of the neurons. As shown in the previous chapter, comparing different training methods on the same NN architecture can result in similar performance and approximation accuracy. However the question remains what exactly does each NN learn, as different internal structures (parameters) can result in the same output. Each neuron of a NN approximates a different subsection of the final output. To identify the differences between NNs, this section visualizes the weights and outputs per hidden unit. The goal of this section is to identify if there is a visible difference between the SL and TVI methodology.

3-3-1 Visualizing the neurons

A method of understanding the sub-functions and weight selection done by the NN is by visualizing the output of each neuron. The existing approach [34] suggests that visualizing the sub-functions of a NN can provide further inside into what the NN actually learns. The methodology is adopted and used on the 20 unit per layer TVI-DNN and SL-DNN. Figure 3-11 (b) shows the TVI-DNN neurons learned at the end of training. The first layer consists of different linear functions with the ReLU activation constraining the function above 0. The second layer can represent functions in greater complexity and is able to generate some characteristics of the final V-function. Out of the 20 neurons, the 5 neurons suffer from the dying ReLU problem. Figure 3-11 (a) shows the same neurons after only trained on the first 50 iterations. As shown previously, the V-function converges around that iteration and the

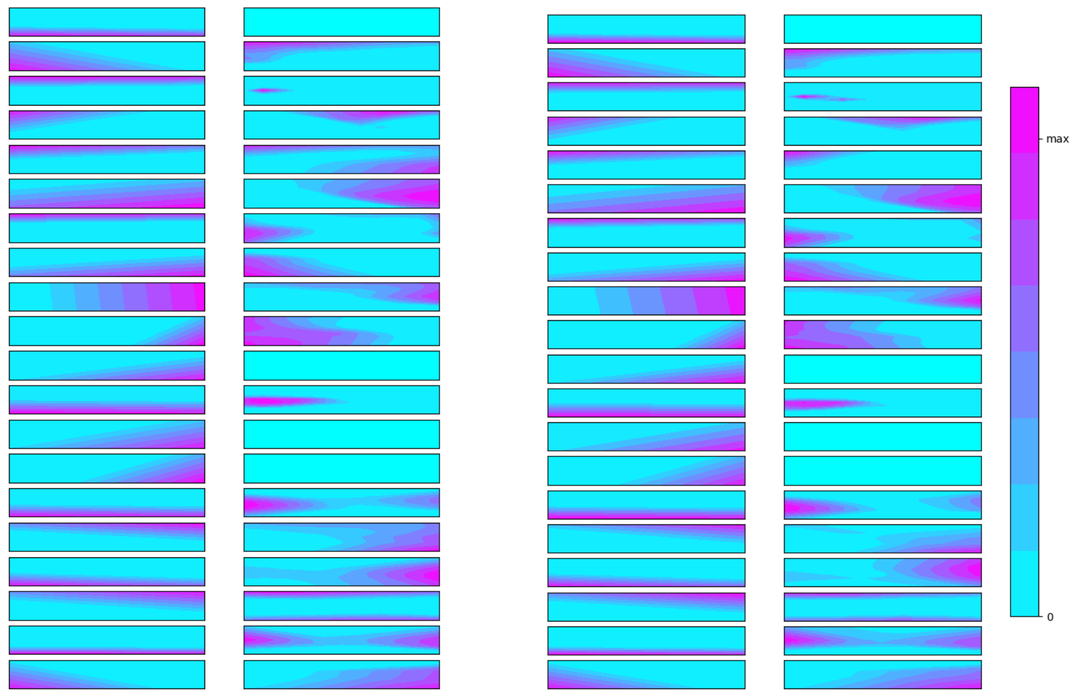
TVI-DNN is able to iterate longer on the same function. Interestingly, only slight changes can be seen from iteration 50 till iteration 115. While the NN is always learning and adopting the weights and biases for every epoch it seems that is less prone to extreme changes even if it trains longer. This would suggest that the TVI-DNN converges to its optimal value and can learn meaningful subfunction characteristics.



(a) Convergence Iteration (50) **(b)** Last Iteration (115)

Figure 3-11: Representation of the neurons output of the TVI-DNN. The horizontal axis is the angle between $[0, 2\pi]$ and the vertical axis is the velocity from $[-30, 30]$. First column presents the 20 neurons of the first hidden layer while the second column represents the 20 neurons of the second hidden layer. The depth is normalized for each neuron for visualization purposes. When the TVI-DNN reaches the convergence iteration (a) the NN has indeed almost fully converge when compared with the last iteration (b). After iteration 50 there exist 5 dead neurons on the second layer of the NN.

Training the NN through iterations can assist the learning of specific characteristics. To confirm this finding we look further into the vanilla SL-DNN. From Figure 3-12 (b) the neurons of the SL-DNN can be seen. The second layer of the DNN shows less symmetry and exact characteristics of the complete NN, however its performance is similar to the TVI-DNN. By visualizing again the SL-DNN during training, on the corresponding iteration 50 of the TVI method (epoch 1500), the neurons look very similar to those of the complete training. As shown previously in Figure 3-2, the SL-DNN converges much earlier than epoch 1500, however the epoch was selected for direct comparison with the TVI method. The visualizations verify the resistance of the network to achieve dramatic changes which suggests the benefits by learning through iterations. The SL-DNN completes the learning with 4 neurons being dead similarly to TVI-DNN.



(a) 1500th Epoch (Relative TVI - iteration 50) **(b)** Last Epoch (Relative TVI - iteration 115)

Figure 3-12: Representation of the neurons output of the SL-DNN. The horizontal axis is the angle between $[0, 2\pi]$ and the vertical axis is the velocity from $[-30, 30]$. First column presents the 20 neurons of the first hidden layer while the second column represents the 20 neurons of the second hidden layer. The depth is normalized for each neuron for visualization purposes. Comparing from iteration 50 (a) to iteration 115 (b) , there are only minor changes between the output of the neurons. Note that the SL-DNN converges before iteration 50 as shown in Figure 3-2

3-4 Summary and Concluding Remarks

The chapter applies the tracking value iteration data algorithm in order to evaluate the performance of the NNs in evolving target data. Shallow and Deep architectures succeed in approximating effectively the value function. By simulating an episode in the IP benchmark, it is possible to see that the learned function does not differ drastically between the two networks but the performance is sub-optimal to the target function. Finally, by visualizing the neurons of the NNs, it was concluded that learning through iterations could be beneficial for the NN as the sub-functions learned by each neuron were more descriptive of characteristics of the final function. So far, shallow and deep neural networks have performed equally as good.

Chapter 4

Bootstrapping Value Iteration

The chapter presents the results for the 3rd transition step of the thesis from SL to RL that is self-learned Value Iteration. The goal of the method is for the NN to learn the Value function using the Bellman Equation and converge to the optimal V-function. By removing the target data used in the TVI, the Bootstrapping Value Iteration (BVI) method is expected to learn from itself by predicting the target V-function. This method can amplify the approximation errors of the NN as for each iteration the NN is trying to fit on its own predicted data. The aim is to use the same NN architectures that have already succeed in vanilla SL and TVI and examine their performance when introduced to bootstrapping.

4-1 BVI methodology

The goal of the BVI algorithm is to prove that the NN architecture can learn through bootstrapping [44]. To maintain as many similarities with the TVI method, it was decided that a grid structure will be used to construct the V-function at each iteration. The grid will allow the BVI to learn in parallel on the whole observation space instead of learning through episodes. The V-function was generated by a grid structure of 51×51 states. For each state the next state was calculated from the dynamic model of the IP by choosing 7 uniformly distributed action from the action range. The tensor of $51 \times 51 \times 7$ stores all the possible next states from the grid. The reward for the IP was set using the pendulum angle, $R(s) = |s(1)| - \pi$. The Bellman equation can be seen in Equation 4-1, where γ is a constant, s' is the next state after taking the action $\alpha = \text{argmax}(V(s))$. The BVI algorithm is trying to learn the Bellman equation by choosing the maximum cost-value of the next state predicted by the NN. The complete algorithm of BVI can be seen in Algorithm 2.

$$V(s) = R(s') + \gamma V(s') \tag{4-1}$$

Algorithm 2: Bootstrapping Value iteration using a NN and a grid structure on the whole observation space.

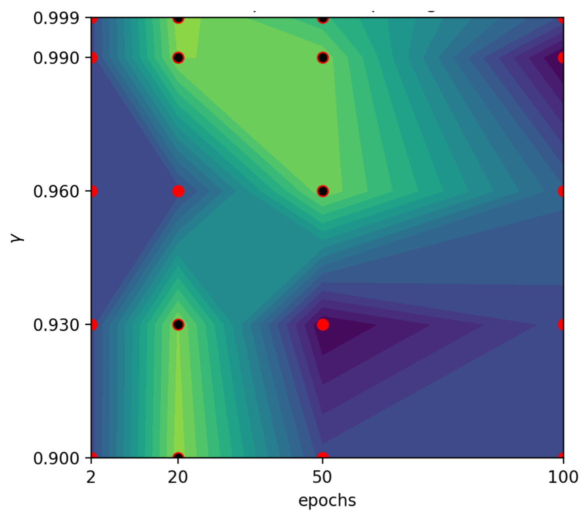
```
$\gamma \rightarrow$ discount factor
$actions \rightarrow$ set of discrete actions
grid_states $\rightarrow$ grid of pos, vel dimensions: $\{$pos $\times$ vel$\}$
next_states $\rightarrow$ tensor of next states dimensions: $\{$pos $\times$ vel $\times$ action$\}$
$R \rightarrow$ tensor of rewards using $s'$ dimensions: $\{$pos $\times$ vel $\times$ action$\}$
**Initiate** Neural Network
**for** *total iterations* **do**
    $s'$ = argmax(Neural_Network_predict(next_states))
    $V = R(s') + \gamma$ Neural_Network_predict($s'$)
    Neural_Network_learn(grid_states, V)
**end**
```

4-2 Discount factor and Epochs

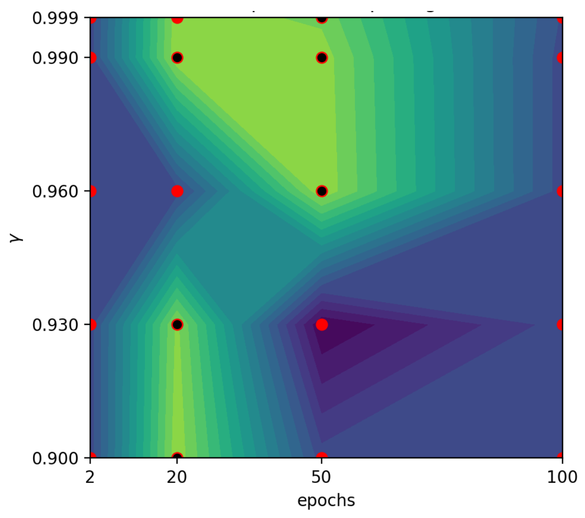
A new parameter that comes into play with the BVI algorithm is the discount factor. The discount factor γ can affect the performance of the IP as it can drastically change the value-function. Bootstrapping a NN with a very high γ (e.g. 0,999) can be harmful for the learning as it can amplify the approximation errors of the network. Lowering the disount factor (e.g. $\gamma = 0,9$) can prevent the pendulum from swinging up as the immediate reward will be priortized and the pendulum will not learn to "swing" in order to gain momentum and reach the top.

With the goal to observe any harmful phenomena caused by too high or too low discount factor introduced in the BVI method, the IP is trained on various values of γ and for different epochs at each iteration. Too many or too few epochs used for training the NN per iteration can result in over-fitting or under-fitting of the data which in theory could counteract some of the side-effects of the discount factor. By varying both the epochs and discount factor it is expected that the learning will be worst near the extremely low and high values with the NN being unable to learn the correct value function.

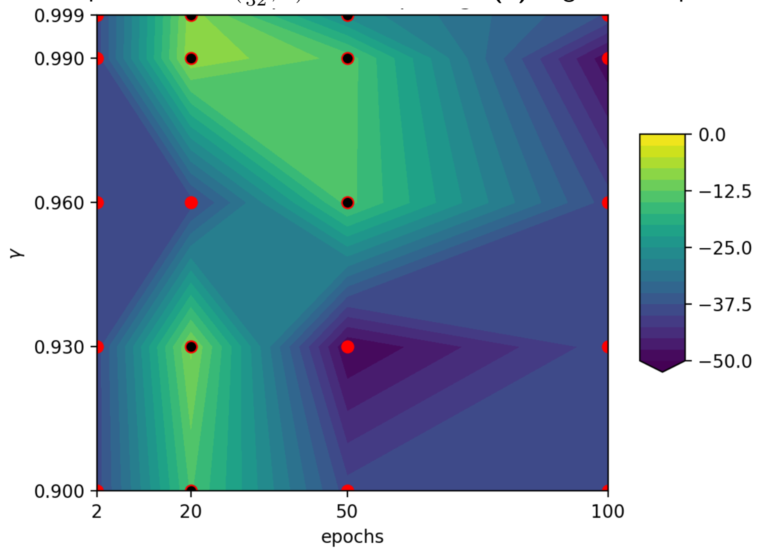
For initial experimentation the DNN with 501 parameters was trained on varied epochs and discount factors. To evaluate the results, the return of one episode with respect to epochs and γ is plotted in Figure 4-1. As the simulation of swing up from lowest position could sometimes receive zero input and not initialize movement, the episode is repeated two more times. The additional episode have initial state with slight offset of the pendulum, left and right. While the return value for each initialization seems very consistent, some NNs randomly fail to achieve swing up.



(a) Left initial position $s = (\frac{\pi}{32}, 0)$



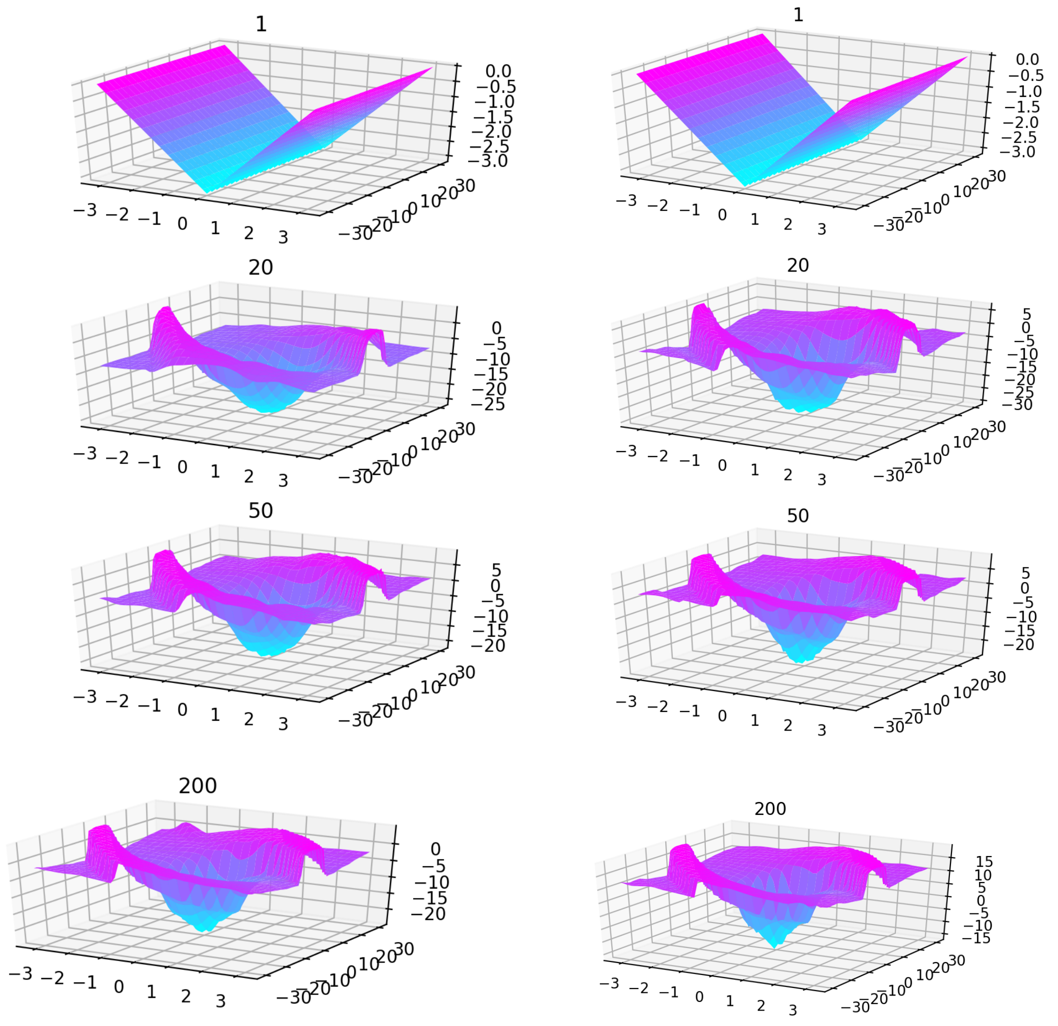
(b) Right initial position $s = (\frac{61\pi}{32}, 0)$



(c) Zero initial position s=(0,0)

Figure 4-1: VI for varied γ and epochs using a NN with **tanh** activation function. The red dots correspond to the values that the VI was trained on, while the black dots correspond to the simulations where the pendulum achieved swing up.

To gain more insight to where the NN fails, we examine directly the learned V-fuction by tracking its changes through the bootstrapping iterations. Two NNs are visualised one with a successful swing up and the other one without. Figure 4-2 shows the learned V-functions through various iterations of learning for the runs of epoch/γ being $20, 0.96$ and $50, 0.99$. As observed by the V-function, both approximations look very similar, thus it is expected that very small errors affect the learning.



(a) $\gamma = 0.96$, epochs $= 20$ **(b)** $\gamma = 0.99$, epochs $= 50$

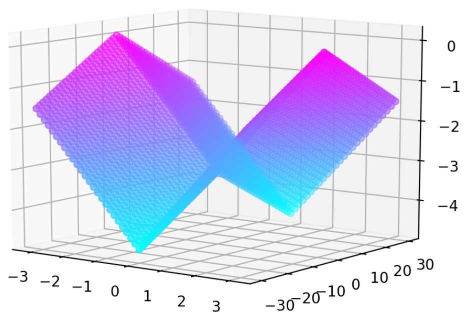
Figure 4-2: BVI of two NN with different epoch and discount factor. The plots consist of the learned V-function throughout bootstrapping, at iterations 1, 20, 50 and 200 (final iteration). By comparing side by side the two runs, they both seem to follow close path of learning, and the general shape of the V-function is captured by both approximators.

4-3 Upgrade Reward Function

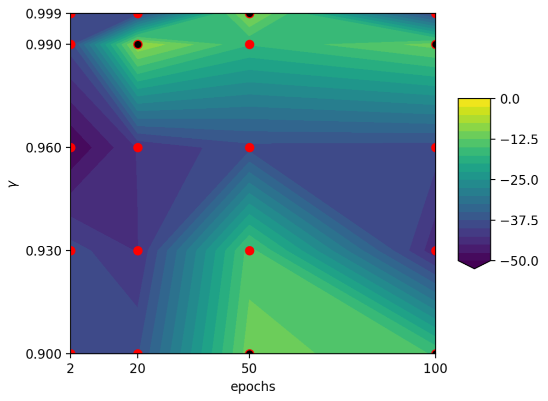
In the previous section it was shown that small errors and inaccuracies can result in the pendulum to fail swing up. In an attempt to capture the whole domain and improve learning, we change the reward function to include all the states of the model [45] [46]. The upgraded reward function is

$$R(s) = |s(1)| - \pi - 0.01 \tag{4-2}$$

and can be shown in Figure 4-3 (a) . The updated epoch/γ diagram can be seen in Figure 4-3 (b) . The updated diagram seems to still show inconsistencies towards achieving swing up. This suggests that the testing method used to verify the learning of NN could be easily influenced by small errors in the V-function and the inclusion of both states in the reward function did not solve the problem.



(a) Updated Reward

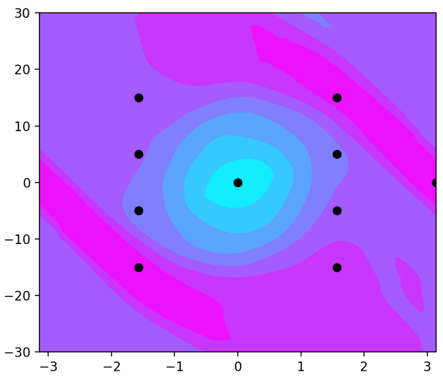


(b) Updated Epoch/γ Diagram

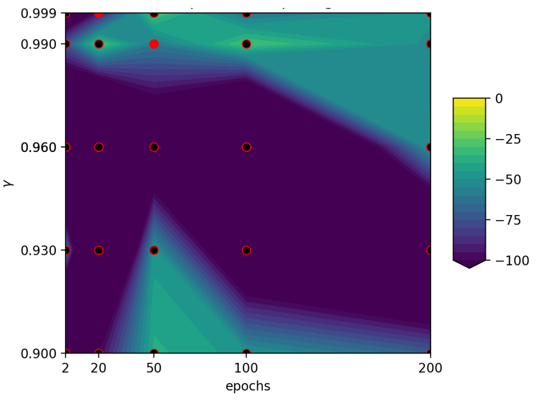
Figure 4-3: (a) Upgraded reward function that takes into account the position and velocity of the pendulum. The updated reward has only one max point at $[\pi, 0]$. (b) The return of one episode with initial position $[0, 0]$. The BVI-DNN is trained on various epochs and different γ values. The results seems to be random and unstable.

4-4 State multi-initialization

To overcome the uncertainty of one bad initial position, the pendulum is initialized from multiple states and the return of all episodes is summed. Figure 4-4 (a) shows the various states that the pendulum is initialized. Figure 4-4 (b) presents the updated epoch/γ diagram. With multiple episodes the BVI- DNN continuous to fail to achieve consistent swing up suggesting that the V-function learned is partially wrong. Given the multiple adaptations done to accomodate the NN learning, it can be concluded that the current DNN architecture fails to learn the proper V-function.



(a) All initial states



(b) Updated Epoch/γ Diagram

Figure 4-4: (a) Representation of all initial states before swing up. The 10 positions try to describe different locations around the value function that usually would not be visited from the lowest position (b) The sum of the return of multiple episodes with initial states as shown in (a). The NN is trained on various epochs and different γ values. The learning seems to be fluctuating and inconsistent with the epochs and γ parameters.

Small inaccuracies in the Value-function are shown to have large impact on the performance of the DNN irrelevant to epochs or γ. This would suggest that a SNN could outperform the DNN by learning a simpler and more correct V-function. To verify this, the BVI-SNN with 128 neurons (513 parameters) is trained under the same conditions. The sum return of multiple episodes with different initial states can be seen in Figure 4-5. As suggested, the SNN is capable of approximating the V-function and achieve consistent swing up across all γ values. Additionally, it consistenly fails to learn when it is trained for 2 epochs per iteration. This is expected as the NN is not trained long enough at each iteration and possibly longer maximum iterations would be needed for learning. With the SNN being capable of learning the V-function through BVI, it is possible to investigate and identify why the DNN failed to learn.

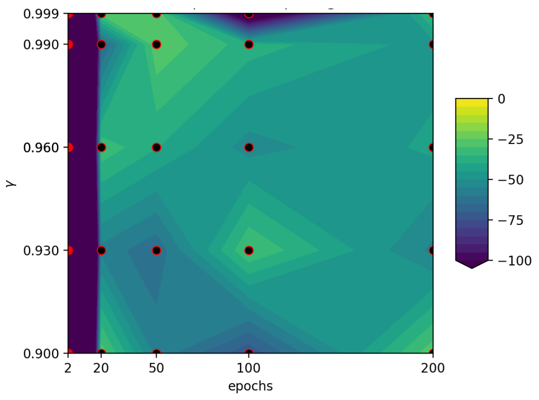


Figure 4-5: The sum of the return of multiple episodes with various initial states. The SNN is trained on different epochs and γ values. The sum return is consistent for the NNs trained on more than 20 epochs for all γ. Small variations in the valley are cause from the lower initial position where swing up is not always achieved, similar to the DNN.

4-5 Difference between Shallow and Deep

Training both SNN and DNN using BVI, results in the SNN to be able to learn and perform on various episodes while the DNN performance is random and incosistent. The same NNs were previously trained in a vanilla-SL and TVI environment where both performed equally good. It is therefore important to understand why the performance of DNN drops dramatically when it learns through BVI.

A direct comparison could be done by visualizing the output of the SNN and DNN or by comparing the training loss of the networks on a specific epoch and γ value. In addition, by normalizing the V-function between $[0, 1]$ it is possible to look at the difference in between the learned functions from the SNN, DNN and BF. The above comparisons are captured in Figure 4-6 where the NNs were trained on 100 epochs and $\gamma = 0.96$. By comparing the difference of V-function between SNN and DNN or by observing the output of the DNN, a "twister" effect can be noticed in the steep area around the origin. The twister effect cannot be observed on either the SNN or BF which could be the reason behind the poor performance of the DNN. On the other hand, the training loss of the DNN converges to a lower value than the loss of the SNN which would suggest that the deep network should outperform the shallow one.

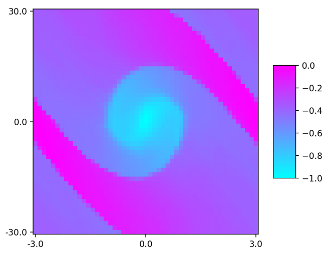
The poor performance of the DNN could be affected by the grid data used on the BVI method. Even if the same grid format was used on the vanilla-SL and TVI without introducing the effect, bootstrapping with the additional expressiveness of the second hidden layer could have resulted in the DNN to learn a characteristic that was harmful for the pendulum performance.

4-5-1 Visualizing the expressiveness

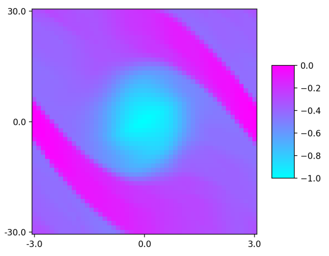
Using the BVI algorithm with DNN and SNN, the SNN seem to generalize better on the V-function. From previous experiments using vanilla SL and TVI, the DNN was able to perform at least as good as the SNN. In order to identify why the SNN outperforms the DNN and specifically why it is more consistent in learning to swing up from the lowest position, we visualize the activation transitions (described in Section 2-4-2) of both networks for specific epoch/γ combinations.

Shallow NN

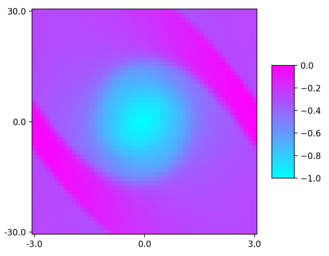
With the SNN outperforming the DNN architecture it is important to understand where the one fails and the other one prevails. For more insight to the learning of the SNN we compare the learned parameters of the network after trained with TVI and BVI. From the figure 4-7, the activation lines are more symmetrical and well distributed in the TVI network than the BVI network. Introducing the self learning component into the BVI algorithm the SNN performance also drops. Nevertheless, it is still capable of achieving swing up from multiple initialized states which suggests that the reduced complexity of the SNN results in a more robust learning through BVI.



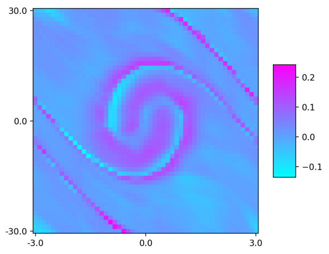
(a) DNN



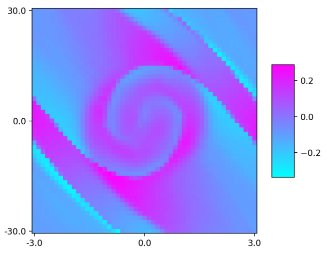
(b) SNN



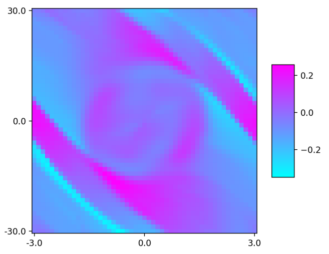
(c) BF



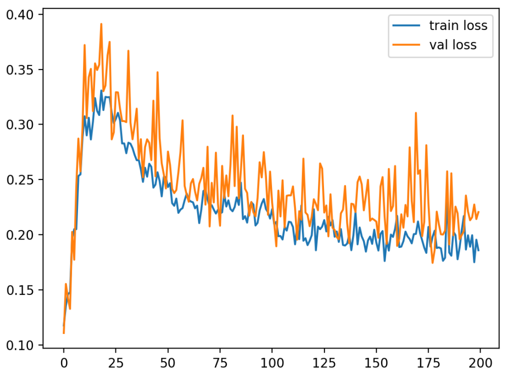
(d) Deep - Shallow



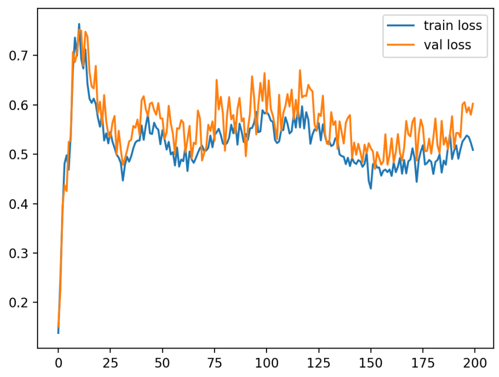
(e) BF - Deep



(f) BF - Shallow



(g) DNN Training Loss

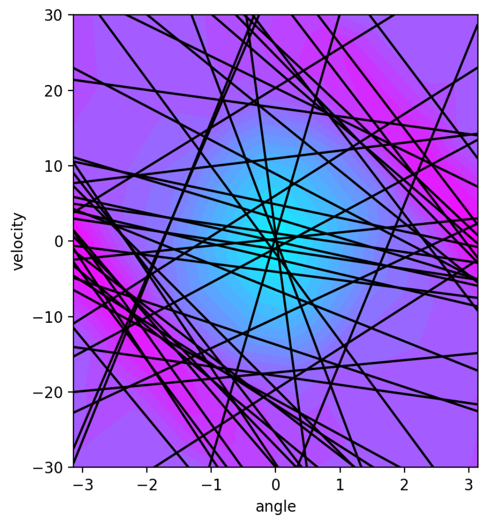


(h) SNN Training Loss

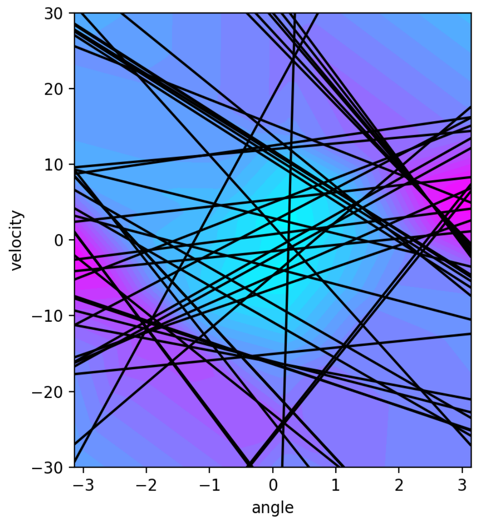
Figure 4-6: A visualization of the learned V-function of the DNN (a) and SNN (b). By normalizing the learned functions we can directly subtract them. The difference between DNN and SNN (c) and BF and DNN (d) creates the twister effect. The effect is not visible between BF and SNN (e). In contrast, the training loss of the DNN (f) is lower than the loss of the SNN (g) which would suggest better learning.

Deep NN

The DNN failed to properly learn the V-function through BVI when trained under the same conditions with TVI. By visualizing the learned V-function and the activations for both first and second hidden layer, as shown in Figure 4-8, the activation lines from the first hidden layer seem to be unevenly distirbuted. Similar observation was made for the SNN however with the addition of a second hidden layer the performance drop seems to be more drastical.

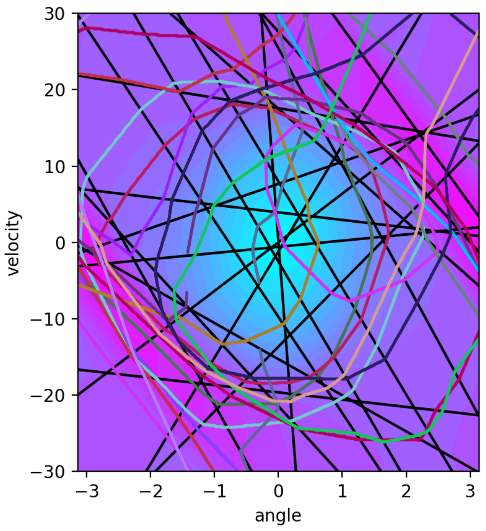


(a) Tracking Value Iteration

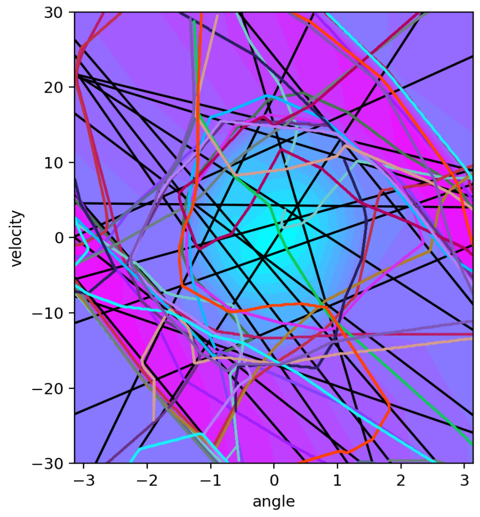


(b) Bootstrapping Value Iteration

Figure 4-7: V-function learned from a SNN by (a) TVI and (b) BVI. The TVI-SNN has a more symmetric structure as shown by the activation transitions of the hidden layer (black lines). The BVI-SNN is still capable to achieve swing up and perform well from multiple initial states however the activation lines seem more usymmetrical which can be expected due to bootstrapping.



(a) Tracking Iteration Learning



(b) Bootstrapping Value Iteration

Figure 4-8: V-function learned from a DNN by (a) TVI and (b) BVI. The TVI-DNN has a more symmetric structure comparing the activations of the first hidden layer (black lines), which results in better approximations for the second hidden layer piece-wise linear activations (coloured lines). In this case, the increase of expressiveness by depth, is shown to be harmful for the learning.

4-6 Summary and Concluding Remarks

The chapter is presenting the concept of bootstrapping on value iteration, using the NN itself. Bootstrapping has been proven challenging for the Neural Network, especially for the deeper architecture. When comparing the results with the TVI algorithm, from the previous section, the learned sub-functions of the NN, learned by each neuron were more chaotic using the bootstrapping algorithm. Where the shallow NN managed to overcome the extra complexity, the deep network struggled as its additional expressiveness introduced unwanted patterns to the learned function. Possible bottlenecks and failure points for the deep network could have been the grid structure used to generate the value function, in addition to over-fitting in each iteration. Shallow NNs outperformed the Deep NNs as their simpler architecture proven more robust to the pitfalls of the algorithm used.

Chapter 5

Reinforcement Learning

The chapter introduces the final convergence step from supervised to reinforcement learning. The NN is expected to learn the value function and the policy function, without pretrained data, by following the reward function provided [25] [47]. The chapter will implement a state-of-the-art Reinforcement Learning algorithm on three continuous control tasks, the IP, Magman and OpenAI Gym Hopper-v3. An additional investigation will be done on the input type of the NNs by comparing global and local inputs for the IP and Magman tasks.

5-1 Deep Deterministic Policy Gradient

This section will introduce the state-of-the-art RL algorithm called Deep Deterministic Policy Gradient (DDPG) [19]. The algorithm will be used as a testing algorithms for the shallow and deep networks architectures on various low-dimensional continuous control tasks.

5-1-1 Action Noise

A very popular method to introduce exploration in a continuous space is by Ornstein Uhlenbeck action noise (OUnoise) [48]. The stochastic process has a temporalily correlated noise which can improve the exploration in systems that require momentum. The equation of OUnoise designed to approximate the brownian motion with friction, by generating temporarily correlated actions around 0 can be discretely written as

$$X_{n+1} = X_n + \theta(\mu - X_n)\Delta t + \sigma \Delta W_n \tag{5-1}$$

where the mean μ is zero, standard deviation σ is 0.5, θ is 0.15 and time step is 0.01. The parameter X_n presents the noise at the current discrete step with X_0 being the initial noise which is equal to 0. The ΔW_n is the Wiener process (a.k.a. Brownie motion process) that presents normal variance with mean equal to 0 and variance Δt. By modifying the Brownie motion process the mathematical solution can be seen in Equation 5-2, where RN is a normally distributed random number.

$$X_{n+1} = X_n + \theta(\mu - X_n)\Delta t + \sigma\sqrt{\Delta t}RN \tag{5-2}$$

5-1-2 The Algorithm

DDPG is an off-policy RL algorithm, used in continuous applications [19]. DDPG is an actor-critic method that uses target actor and critic networks (Q' and μ'), in addition to the actor and critic networks (Q and μ) to achieve deterministic learning. The actions during training are predicted from the actor network with the addition of OUnoise (Equation 5-3). The state transitions with the reward are stored in a memory buffer as shown in Section 2-1-3. The Bellman equation is updated using the reward of critic network and the Q value from the target critic and actor (Equation 5-4). The critic network is trained on mini-batches using a Mean-Squared-Error loss function as shown in Equation 5-5 where i is the index of each sample in the batch. After the actor and critic networks are trained their counterparts target networks are updated by interpolating the weights between the previous target network iteration and the new train network. The interpolation is done with the constant τ which is usually set very small. The target network parameter update equation can be seen in Equation 5-6.

$$a = \mu(s_t) + OUnoise \tag{5-3}$$

$$y(s_t) = r(s_{t+1}, a) + \gamma Q'(s_{t+1}, \mu'(s_{t+1})) \tag{5-4}$$

$$\mathcal{L} = \frac{1}{N}\sum_i (y(s_i) - Q(s_i, a_i))^2 \tag{5-5}$$

$$\theta^{Q'} \leftarrow \tau\theta^{Q} + (1-\tau)\theta^{Q'} \tag{5-6}$$

5-1-3 Hyper-parameters

The training was run for 5M *timesteps* (later referred to as steps) with the DDPG algorithm being updated in every step from an experience memory buffer. Each step reflected an actual step of the simulation that was found as a more appropriate measure for all the environments, including those with premature episode termination such as Hopper-v3. The buffer stored 50,000 samples in total out of which a batch of 128 was selected for one update. The discount factor γ was set at 0.99.

5-1-4 Neural Network

Two identical Neural Networks were used for DDPG, one to be used as an actor and one for critic. The hidden layer activations of both networks were ReLU while for the output layer were linear and tanh for the critic and actor networks respectively. For the actor and critic networks the Adam optimizer was used with $1e-4$ and $1e-3$ learning rates respectively. The target networks were updated with the parameter τ set at 0.001.

5-1-5 Action Scaling - Normalization

In order to maintain the same neural network architecture for all continuous tasks the actor output had to be scaled [20] [49]. The actor output activation function was set to tanh that would output values between $[-1, 1]$. After predicting the action, the algorithm would scale the action to the action space of the environment (i.e. for the pendulum it will scale it by 2 to range between $[-2, 2]$ and for magman it would normalize it between $[0, 0.6]$).

5-2 Inverted Pendulum

The IP was trained on DDPG for both shallow and deep network architectures. The observations were normalized between $[-1, 1]$ [49]. The output activation function of the actor was set to tanh and was scaled before being applied to the IP.

5-2-1 Global Inputs

The task was trained on the same NN architecture on 10 different runs where the mean return per step for all runs can be seen in Figure 5-1 (Left). For each network and each run, the return of 10 episodes from predefined initial states can be seen in Figure 5-1 (Right). The initial states were the exact same ones used in the BVI Chapter 4 when the multi-initialization was introduced (Figure 4-4).

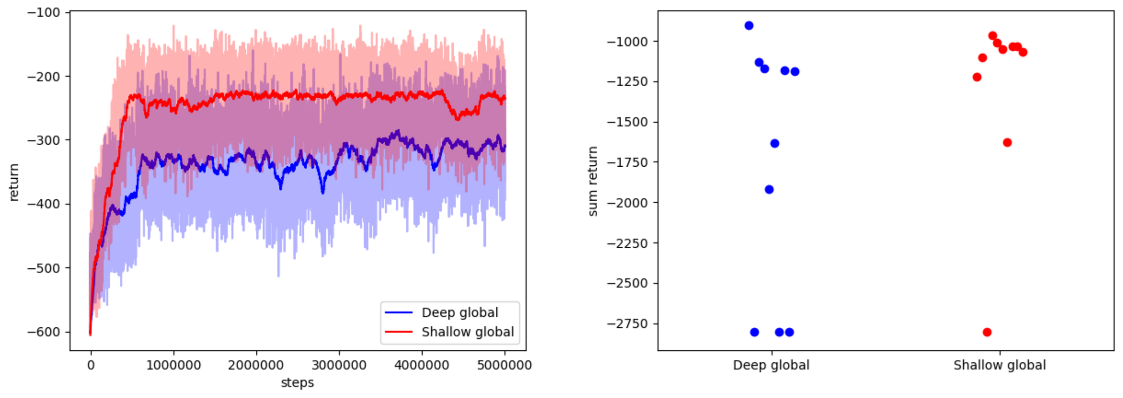


Figure 5-1: (Left) The mean return of 10 runs per NN architecture. The bold line is the smoothed return of the mean. (Right) The sum return after simulating each NN from multiple initial states.

Both Shallow and Deep networks achieve sufficient performance with similar results. The DNN was less consistent in learning than the SNN which can be shown by the difference of the mean return. Out of the 10 runs, only 1 SNN fails to learn anything in comparison to the

3 DNN. This cannot conclude anything as the runs were limited, however it seems that SNN have been slightly more consistent. Figure 5-2 shows the separate runs individually for SNN (Left) and DNN (Right). The DNN is shown to have slightly varied learning point, while shallow is more consistent in learning at around the same step. Learning for SNN is more robust compared to the DNN random unlearning characteristics.

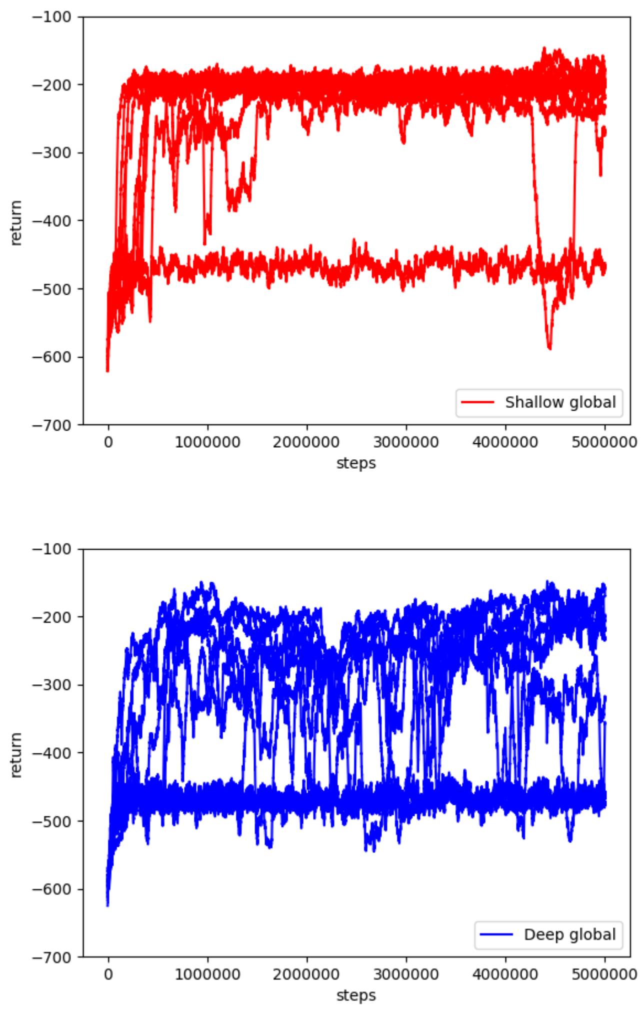


Figure 5-2: Representation of individual runs for shallow and deep NN on the IP using DDPG and global inputs. SNN has more constistent performance with better learning and convergence than the DNN.

5-2-2 Local Inputs

In addition to the experiments done using the standard inputs for the pendulum, a new set of experiments were conducted with pre-processing of the input data. The position and velocity inputs were converted to 441 basis functions. As the NN architecture used for the IP was already minimized to challenge the capacity of the network, trying to match the parameters with such an increase of input size would bottleneck the networks. Instead the network architecture remained the same (i.e. 20 neuros per layer for DNN and 128 neuros for SNN) which caused the total trainable parameters of each network to vary. The goal of these experiments is to investigate if a more expressive input formatting can improve the learning using the same number of neurons for each NN. The set-backs of this method were the computational time that was much higher for the SNN, due to increase of parameters. Similar to global inputs, 10 NNs were trained for each architecture. The mean return per architecture can be seen in Figure 5-3 in addition to the sum return from multiple initial states for each individual NN.

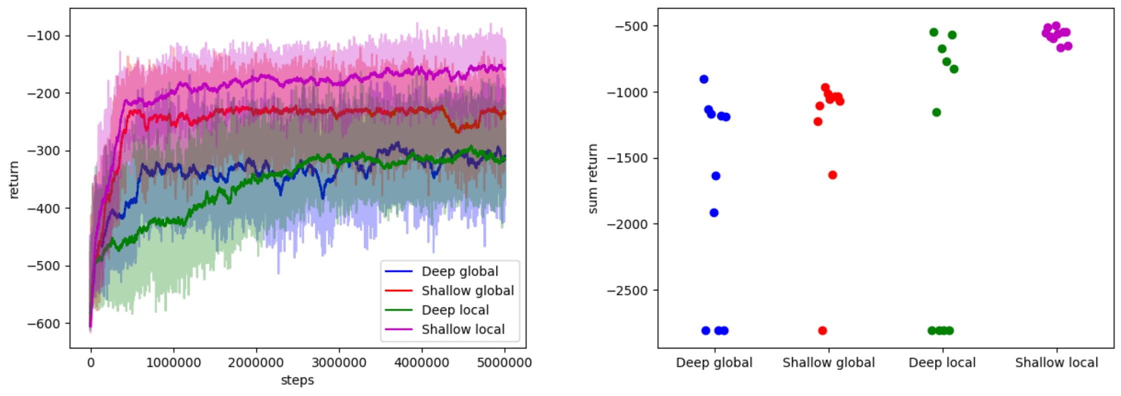


Figure 5-3: (Left) The mean return of 10 runs per NN architecture. The bold line is the smoothed return of the mean. (Right) The sum return after simulating each NN from multiple initial states.

By comparing the results of the global and local inputs, it is possible to see that the local inputs improve the performance of both NN architectures. This could have been expected due to the increase of trainable parameters, nevertheless, by breaking down the inputs the complexity of the problem increases. For the DNN, the local inputs improve the performance of the networks that learn, while those that do not learn still exist. Comparing the SNN-global and SNN-local, there is a similar distinction between learning and unlearning, however for SNN-local all 10 runs have successfully learned. SNN maintains better performance to the DNN, for both global and local inputs.

5-2-3 Computational Time

A final observation on the setup is done by comparing the computational time between architectures. The goal is to identify if the increase of depth or width, dramatically affects the training time of the network. The total computational time for all runs using DNNs and SNNs can be seen in Figure 5-4. The global input NNs consist of approximately the same training parameters and have very fimilar computational time. The local input NNs are constructed with more parameters than the global however the computational time does not dramatically increase. The choice of SNN or DNN with the same trainable parameters does not affect the total training duration using DDPG.

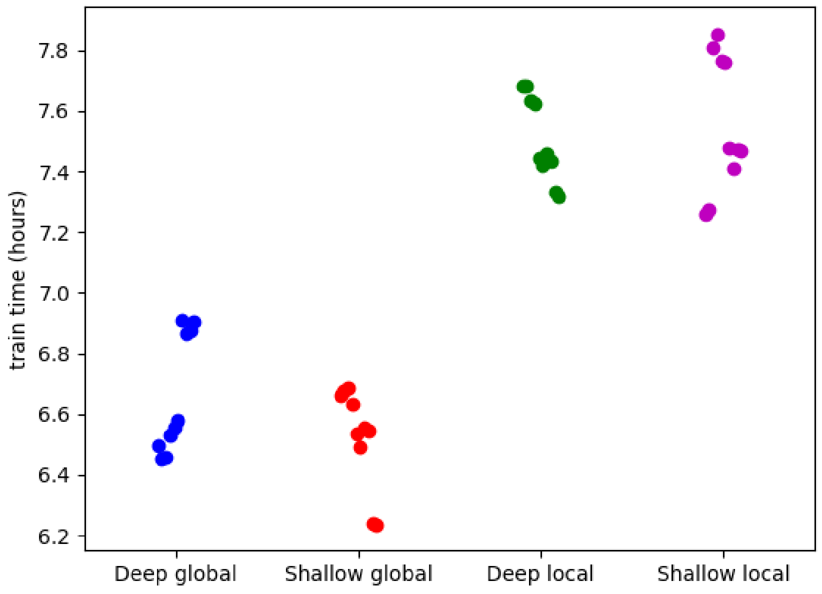


Figure 5-4: A representation of the computational time for all the trained NNs. The NNs were trained in sets of four, due to the computer capacity, something that can reflect in the plot. Comparing the deep with the shallow architectures, it can be seen that the computational time is approximately the same.

5-3 Magman

The Magman environment was also train the using DDPG algorithm. Unlike the IP, the Magman was always reset at the same initial position (0,0) and had the same target position (0.035,0) in between the two coils. The observations were normalized between $[-1, 1]$ and the output of the actor (tanh activation) was scaled to $[0, 0.6]$ [49].

5-3-1 Global Inputs

As there was no previous experimentation with the Magman setup, using the same NN architecture with the IP could result in poor learning for one or both NNs. Therefore, the width for the SNN and DNN was increased to guarantee that there are enough parameters for the NN to learn. The return per step, collected from 10 runs per NN architecture, can be seen in Figure 5-5-Left. From the graph, it is possible to see that the SNN had more failures in learning that the DNN, while some extreme unlearning (steep dives of the return throughout learning) can be seen in both architectures. To observe the final performance of each network, Figure 5-5-Right shows the return of a single episode in the magman environment for each individual NN. More failures in learning can be observed by the SNN, which has 4 runs out of 10 getting stuck in some local minima (above one coil), while only 1 run of the DNN fails without learning anything.

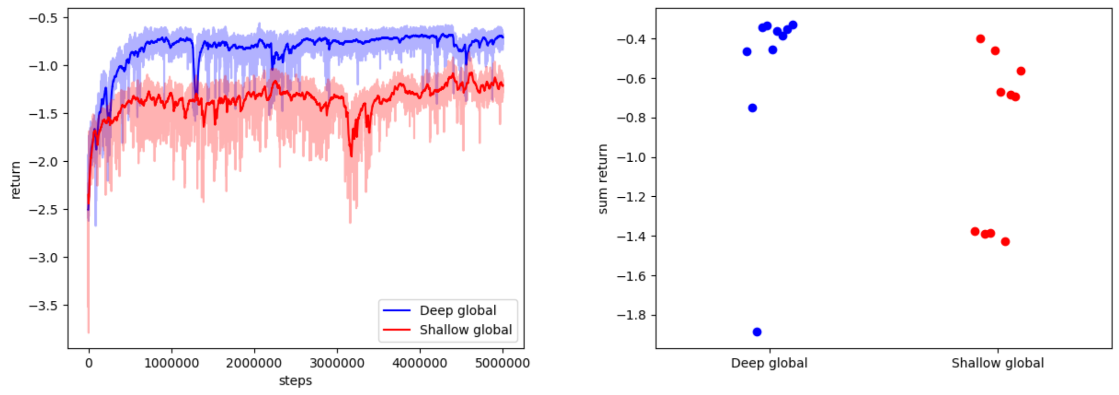


Figure 5-5: (Left) Mean return per step on the magman setup using SNN and DNN. The mean was collected from 10 randomly initialized runs using the same NN architecture and hyper-parameters. **(Right)** Return of a single run using the trained networks on the magman simulation. By comparing the two graphs it is easy to see that the mean and convergence value per step is affected by the NNs that failed to learn.

To gain more insight to the return per step of each run, the individual runs are shown in the Figure 5-6. From the plot, it can be seen that the 1 run of DNN that failed to learn can be easily spotted while for the SNN the learning is very noisy. DNNs achieve the top reward and converge on it without too much noise, a phenomenon not observable in the SNN.

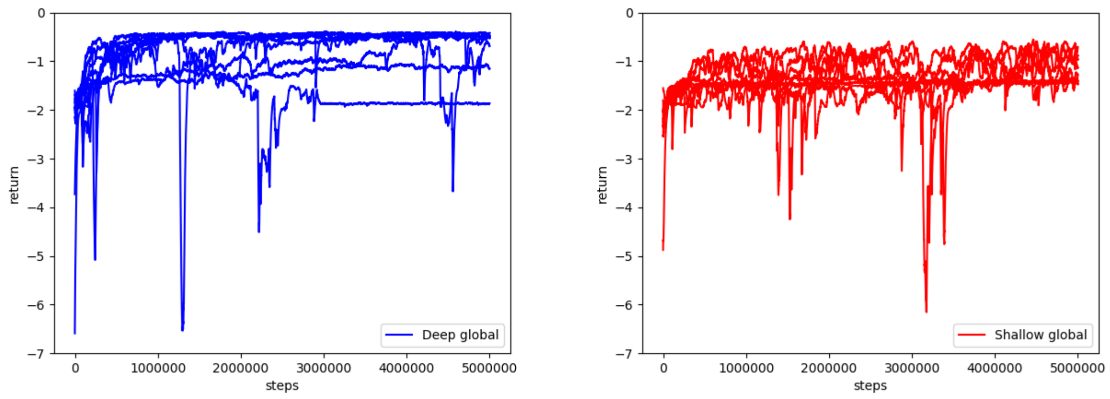


Figure 5-6: Representation of individual runs for shallow and deep NN on the magman using DDPG and global inputs. DNN has more constistent performance with better convergence than the SNN.

5-3-2 Local Inputs

Similar to the IP benchmark task, the observable states (position and velocity) were pre-processed to a basis function grid of 21×21 resulting to a total of 441 parameters from $[0, 1]$. To purely compare the effect of the change of input format, the NN architecture remained unchanged with 64 neurons per layer for the DNN and 512 neurons for the SNN. The results of 10 runs per architecture can be seen in Figure 5-7 in addition to the return of one episode for each trained NN. For the magman benchmark, pre-processing the input did not improve the performance of the simulation. As the initial state and target state are the same, the DNN-global and SNN-global already achieve near perfect simulation. Nevertheless, it can be noted that the DNNs, both global and local, are more consistent in reaching the optimal return. As the magman simulation task is deterministic and the initial states are always the same, it is believed that the better performance of the DNN can be explained from over-fitting.

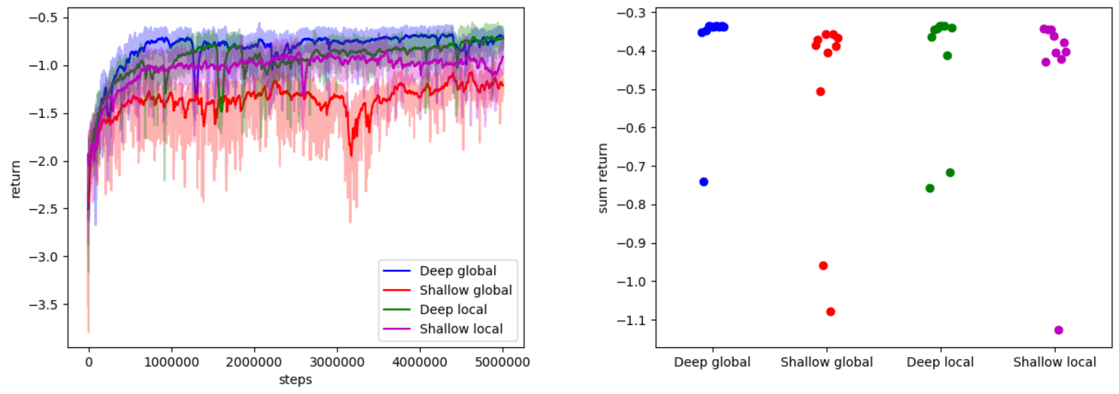


Figure 5-7: (Left) The mean return of 10 runs per NN architecture. The bold line is the smoothed return of the mean. (Right) The sum return after simulating each NN from multiple initial states.

5-3-3 Random Initialization

From the experiments done in using DDPG on both pendulum and magman, it was observed that shallow network outperformed the deep network when there was more randomness in the environment. For example, the pendulum was randomly initialized in the whole domain while the magman was always initialized from the same state. Repeating the same sequence every time could be beneficial for the DNN, that is why we have observed an increase in performance for the magman setup. To confirm or discard this disclaimer, the NNs are retrained on the magman setup however the states are now randomly initialized in every episode. The attempt to increase the complexity and spread the sampled points to a broader domain was expected to increase the performance of the SNN. As shown in Figure 5-8, the DNN consistently outperforms the SNN with the new setup parameters. In contrast, the SNN barely learns to compensate for the random state initialization which can conclude that the difference between the architectures does not necessarily lie in the state initialization condition. Figure 5-8 also presents the return of one episode from the same initial state for each NN. The distinction between the learning of DNN and SNN can be clearly seen from the return gap of the two architectures.

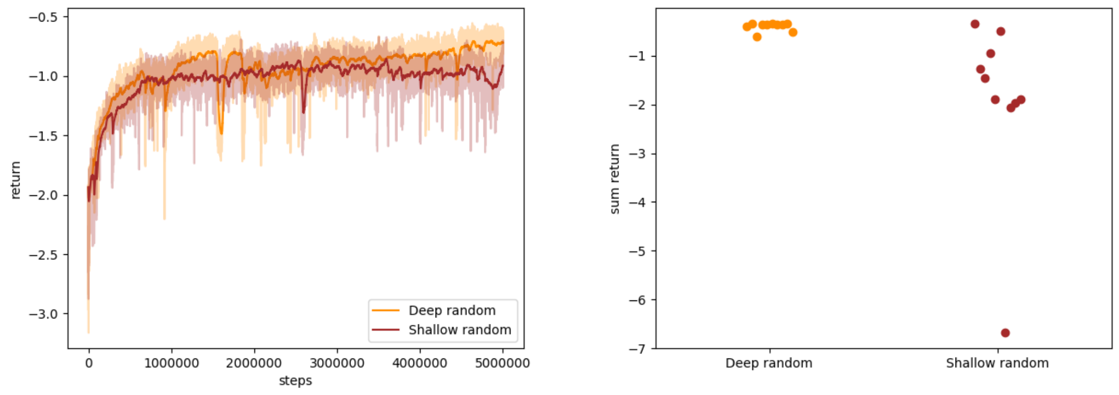


Figure 5-8: (Left) The mean return of 10 runs per NN architecture. The bold line is the smoothed return of the mean. (Right) The return after simulating each NN from the same initial state. Where from the return it can be expected that both NN architectures learn, by observing the simulation it is clear that the SNN does not perform as good as the DNN.

5-4 Hopper

The Hopper-v3, described in Section 2-2 was trained using the DDPG algorithm. As there was an important increase in observation states and actions, compared to the IP task, the NN architectures were increased in width. For the SNN, the hidden layer was increased to 512 units, 6657 parameters. For the DNN, each hidden layer was increased to 64 units for a total of 4929 parameters [20].

5-4-1 Training

Both DNN and SNN were trained using DDPG on the Hopper-v3 environment. For each network architecture there were a total of 10 runs from which the mean return per step can be seen in Figure 5-9 - Left (For the SNN only 5 runs were captured as the MuJoCo licence had expired before the end of the experiments. The mean was taken over those 5 runs). Comparing the smoothened return of the SNN and DNN, both neural networks have very similar performance on the specific setup. An additional Figure 5-9-Right, presents the maximum return recorded in each run. From the maximum return of the SNN, it can be seen that throughout the learning, it achieved significantly higher return over the DNN. Such high return could result by either learning a faulty movement that would exploit a loopwhole in the environment dynamics or by learning to Hop for very long time.

DNN and SNN have very similar performance in learning the continuous control task. While from simulation it can be observed that even the best rewarded runs could have inconvenient methods to collect the reward, the issue of the task not learning perfect forward hopping is not in the interest of the thesis. An additional set-back was the unlearning as providing a total number of steps for the algorithm without any condition for earlier stopping resulted in some of the runs to learn early and unlearn the good policy before the end of training.

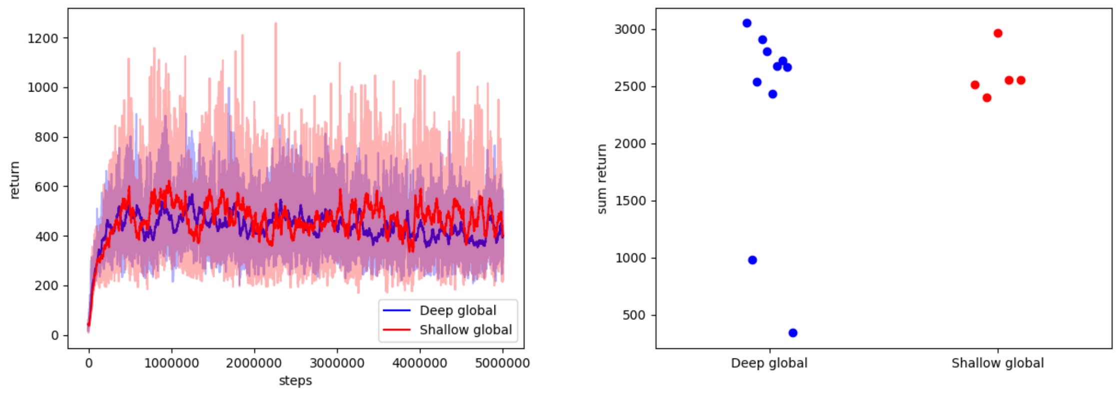


Figure 5-9: (Left) Mean return of SNN and DNN runs on the Hopper-v3 environment. **(Right)** Maximum return recorded per episode for individual runs of the SNN and DNN. Learning seems consistent for SNN and DNN with the SNN having less total runs to compare from. Some of the learned policy functions of both SNN and DNN when simulated were not the "hopping" expected from the task but rather a weird movement that managed to collect equally as much or higher reward.

5-5 Summary and Concluding Remarks

In this chapter the state-of-the-art DRL algorithm, DDPG, is applied in all three benchmarks. The evaluation consists of comparison of the return during training and the performance of the NN during simulation after training. The performance of shallow and deep NNs vary between benchmarks, which suggests that a critical difference exists between them. By observing the performance of the NN in two input structures (i.e. global and local), and multiple low-dimensional environments it was concluded that the deep network performed better in the magman setup, the shallow network performed better in the pendulum and equal performance was observed in the Hopper-v3 environment. As the Hopper-v3 being the highest dimension system of all three, it can be noted that the difference between IP and MagMan was not in the dimensions. Both control setups have similar characteristics, which makes the different performance of the networks interesting. The performance gap can be either due to the function, that is to be approximated or from the selection of hyper-parameters that can vary between setups (e.g. a batch size of 128 samples can improve learning of a SNN for the pendulum, while it can harm the learning for the magman task).

Chapter 6

Conclusion

The chapter will summarise the general conclusions derived from the experimental research presented in the project. By aiming to answer the research question between which network architecture can be best applied in continuous control applications, the conclusions will concentrate on an experimental decision between the two architectures with respect to performance, current benchmarks, state-of-the-art algorithms and computational power.

6-1 Performance

The accuracy of function approximation in both SNN and DNN has been recorded from vanilla-SL to state-of-the-art RL. The generalization error and general learning characteristics of the NN were more profound in the SL methods, as the hyper-parameters to be tuned were less than the RL methods. Implementing the state-of-the-art RL methods required setting up multiple hyper-parameters, which could either directly or indirectly affect the NN performance which made a confident experimental conclusion impossible. In general it was observed that the SNN was under-fitting the function (i.e. as seen in the initial SL methodology) while the DNN could badly over-fit the data (i.e. as seen in the VI methodology). The poor-fitting was based on either the number of sampling data, the NN width or the optimization parameters. From the literature it has already been shown many times that with proper conditions both NN architectures can achieve similar results [12] [11] [16] [17]. From our experiments we confirm that both NNs if trained under the correct conditions could partially learn given multiple runs. For the tasks that had very strict domain conditions or were initialized from the same position (i.e. Hopper-v3 and MagMan) the DNN seemed to perform better, as the data was clustered in a smaller space. However, due to the multiple hyper-parameters, it is impossible to know if a slight change in those values would affect the outcome. Thus the conclusion can be made that in low-dimensional continuous control applications, successful learning is not guaranteed from either SNNs or DNNs, whereas the choice lies on the formulation of the task including environment, observation space, termination criteria and algorithm.

6-2 State-of-the-art algorithms

As stated in the previous section, the NN architecture selection can be heavily influenced by the choice of RL algorithm. In current practices, most if not all state-of-the-art RL algorithms are focusing on over-fitting the NN by high sampling data either online or offline. The algorithms are usually associated with a procedure that will directly or indirectly regularize the NN, a characteristic that benefits the performance of the DNNs. A slight modification of the RL algorithm hyper-parameters or simple randomization can cause the same NN architecture to produce different results [23]. In tasks such as the IP, where the initialization of the episode was random, DNNs were more prone to forgetting than the SNNs. Thus the selection of the NN architecture can potentially come down to the algorithm implementation including batch size, memory buffer, learning rate and other hyper-parameters.

6-3 Environment complexity

Nevertheless, the state-of-the-art algorithms are designed based on the current benchmarks that have already been proven to lack the criteria of a well rounded test-bed [9]. Looking into OpenAI gym and similar upgraded versions of those environments (i.e. DeepMind Control Suite [50]), most high dimensional continuous control tasks consist of episode termination criteria or reward penalties that prevent observation of the whole continuous space. In reality, the environments have been set so strictly that force the algorithm to either learn the proper function or not at all. By forcing the control space within tight limits, the environment guarantees that most of the observed data will be clustered around the points of interest. The high increase of data can benefit DNN architectures that tend to over-fit by indirectly being regularized. For example, looking into the simpler pendulum environment learning the whole value function using DRL method has been proven more challenging for the DNN, compared to learning a higher dimensional problem such as the Hopper-v3, where the task is constrained from visiting the whole observation space. Understanding the conditions in which the task is required to be trained in, could influence the choice between SNNs and DNNs.

6-4 Computational consideration

While performance requirements are the main factor into the comparison of the architectures, practical implementation in terms of computational power should also be taken into account when deciding on the ideal NN architecture. It could be a common misconception that DNNs require more computational time than SNNs. The *computational time* of a SNN and DNN on a single epoch, using Tensorflow, are very similar (Figure 6-1). This can be expressed by the improvements done in both computer software and hardware in terms of tensors computation. Even if the network complexity does increase by the addition of extra hidden layers, the computational time does not.

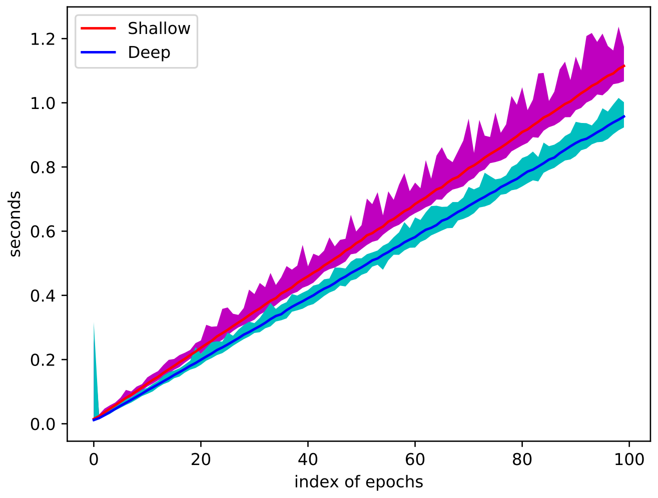


Figure 6-1: A comparison between the computational time of a shallow and deep NN with the same number of parameters on the same data. The x-axis is the number of epochs for which the network was trained on. The DNN training algorithm of Keras with Tensorflow as Backend is slightly faster than the SNN. The results are run on a GPU, however the results are consistent for CPU too.

As the computational time is not affected by the complexity of the network architecture, and both architectures perform very similarly, it is possible to also look at the overall *computational cost* of learning. By comparing the training steps required to achieve sufficient performance for both NNs, the SNN was always slower in convergence than the DNN for the vanilla-SL and TVI algorithms while it was very similar for BVI and RL algorithms. This is due to the additional expressiveness of the DNN which allows the network to fit faster on the same data.

In general, taking into account that the number of trainable parameters does not exponentially increase from shallow to deep NNs, as SNNs tend to also be wider, there is not a real computational setback for using a two hidden layer neural network over a shallow one. Given that the number of parameters does not pose a physical limitation in DRL control applications, equal learning time can be a pro-factor for the DNNs. In additon, when considering a pre-processing of the input data that would exponentially increase the number of inputs (e.g. converting position and velocity to a basis function grid) then using a DNN could reduce the total trainable parameters and speed up training.

6-5 Shallow vs. Deep

To conclude, Deep NNs come with many complexities and uncertainties during learning [40], compared to the limitations of SNNs. The implementation of many DRL algorithms introduces regularization and data collection methods that benefit over-fitting. In addition, some low-dimensional continuous control benchmarks favor clustered data and repeatability over complete randomization. While it has been shown that in a Supervised Learning environment, a SNN can approximate equally as good the target function, the additional expressiveness of DNN can be used as a catalyst for overfitting and learning on poor data. Nevertheless, DNNs have been proven more inconsistent during learning, where the SNNs were sometimes unable to fully express the functions. Thus, it is suggested that before attempting training with a DNN, a SNN could be used as a benchmark network for comparison of multiple runs of the DNN. In theory, any DNN with equally as many trainable parameters with a SNN, can achieve as good of a performance as the SNN. In practice, it is wise to investigate if the SNN could suffice in learning the target function rather than applying a DNN directly.

The conclusion is heavily focused on the current benchmarks and DRL algorithms that are considered state-of-the-art and are only directed towards low-dimensional continuous control tasks. In the future, developments and implementation of new algorithms could exist that motivate the use of shallower and thinner NNs. For the present, a common rule of thumb that is used for constructing a DNN is the use of two hidden layers with hidden neurons per layer $10\times$ the observation states (i.e. hidden neurons = $10\times$ network inputs). From that rule of thumb, a SNN can be also constructed with the aim of maintaining equal total trainable parameters. The expressiveness and capacity of such a shallow and deep NN, shall be sufficient for learning using the standard DRL algorithms.

6-6 Future Work and Recommendations

The thesis provided an answer and an empirical conclusion on the learning of two different network architectures for Reinforcement Learning in low dimensional continuous control application. While it was shown that bootstrapping was harmful for the DNN on the IP setup which was the reason that the SNN performed better on the final stage of RL, the results did not hold the same for the magman setup. A continuation of the research could repeat similar research on the magman setup, as it was done on the pendulum and investigate the differences. An early assumption can be done that the SNN will also fail in the bootstrapping step, which will assist in pinpointing the critical difference of learning between Supervised and Reinforcement Learning. Additional research can be done on purely the Reinforcement Learning stage, as it was the one with the most varied and unexpected results. As it was stated in Chapter 5, the NN learning can be affected by various hyper-parameters. In an attempt to identify the bottleneck of learning, experiments could be done on the pendulum and magman setups by modifying the hyper-parameters of the current algorithm so that the network that previously failed can consistently learn. Furthermore, it has been shown that both architectures can perform well in different environments. In order to bridge the gap between the two architectures, a hybrid network could be designed that will include both shallow and deep connections. With the use of skipping connections a DNN can have the shallow layer linked directly to the output, which may benefit and improve the robustness of learning [51] [52] [53] [54]. For such a hybrid network, it would be interesting to see which of the two elements of the architecture is more dominant in terms of conservation of learning (e.g. using the hybrid-NN the performance can be matched with that of the SNN for the pendulum setup, but it will be also as poor on the magman setup showing that the shallow connections were more dominant or vice versa). The hybrid network could combine the benefits of both basic architectures and outperform them in all the environments.

Appendix A

Selection of NN architectures

A-1 Optimizers and Activation functions

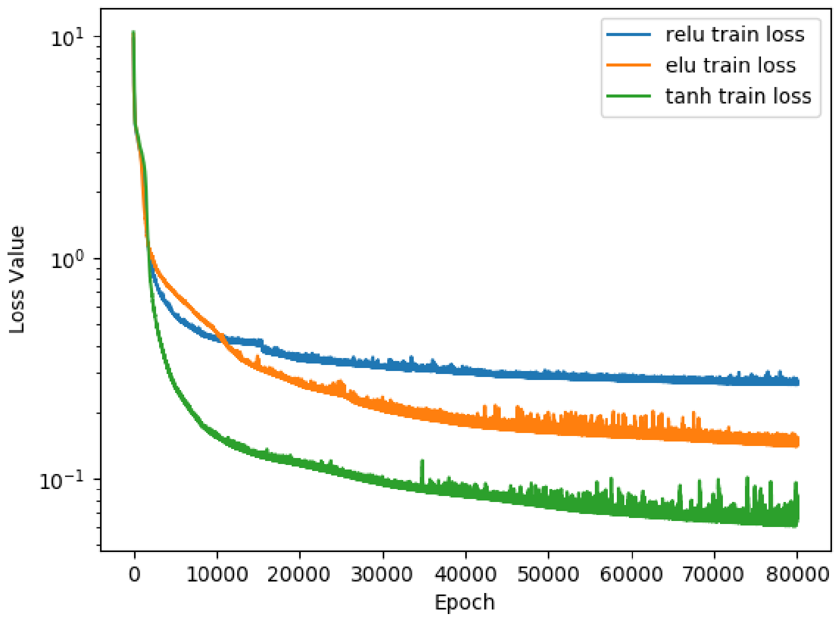


Figure A-1: Example of the same NN, with 2 hidden layers and 20 hidden units per layer, trained on different activation functions. Tanh seems to perform better than the other methods. ReLU converges faster than ELU however ELU has lower final loss value. For initial testing the ReLU activation function was selected as it is commonly favoured by the community for DRL problems. The tanh activation function was also tested later in the report to compare results with ReLU.

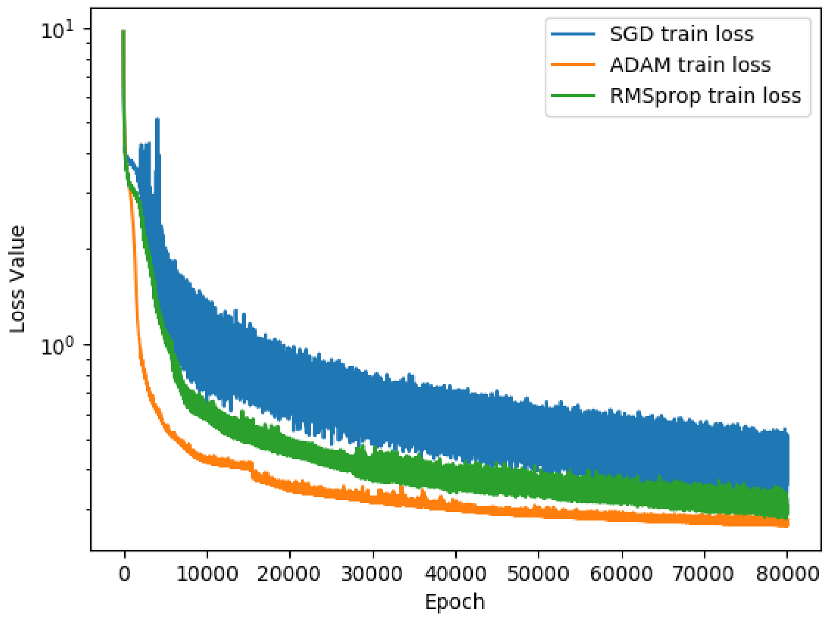


Figure A-2: Example of the same NN, with 2 hidden layers and 20 hidden units per layer, trained on different optimization algorithms. Adam is comparably the best algorithm in terms of stability and lower loss. The insensitivity of Adam to its learning parameters provides a robust optimizer for different NN architectures. Vanilla SGD is highly fluctuating which shows the effect that learning rate and the use of momentum can have on the optimizer. Due to the need of comparison between different architectures and parameters, SGD was found very sensitive an optimizer. RMSprop provides better results that SGD however it achieves higher loss than Adam and is commonly considered inferior to Adam by the community.

References

[1] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," 2015.

[2] Y. Lecun and Y. Bengio, "Convolutional networks for images, speech, and time-series," 01 1995.

[3] D. C. Ciresan, U. Meier, and J. Schmidhuber, "Multi-column deep neural networks for image classification," *CoRR*, vol. abs/1202.2745, 2012.

[4] A. Dosovitskiy, J. Tobias Springenberg, and T. Brox, "Learning to generate chairs with convolutional neural networks," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[5] M. Hessel, W. Dabney, J. Modayil, D. Horgan, H. van Hasselt, B. Piot, T. Schaul, M. Azar, G. Ostrovski, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," 2017.

[6] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification," *arXiv:1502.01852*, 2015.

[7] X. Guo, S. Singh, H. Lee, R. L. Lewis, and X. Wang, "Deep learning for real-time atari game play using offline monte-carlo tree search planning," in *Advances in Neural Information Processing Systems 27* (Z. Ghahramani, M. Welling, C. Cortes, N. D. Lawrence, and K. Q. Weinberger, eds.), pp. 3338–3346, Curran Associates, Inc., 2014.

[8] C. Clark and A. Storkey, "Training deep convolutional neural networks to play go," 2015.

[9] A. Rajeswaran, K. Lowrey, E. Todorov, and S. Kakade, "Towards generalization and simplicity in continuous control," 2018.

[10] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Harley, T. P. Lillicrap, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," 2016.

[11] J. Schulman, P. Moritz, S. Levine, M. I. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," 2018.

[12] L. Busoniu, T. de Bruin, D. Tolic, J. Kober, and I. Palunko, "Reinforcement learning for control: Performance, stability, and deep approximators," 2018.

[13] S. Levine, "Exploring deep and recurrent architectures for optimal control," 2013.

[14] A. Irpan, "Deep reinforcement learning doesn't work yet." `https://www.alexirpan.com/2018/02/14/rl-hard.html`, 2018.

[15] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," 2016.

[16] H. Mhaskar, Q. Liao, and T. A. Poggio, "When and why are deep networks better than shallow ones?," in *AAAI*, 2017.

[17] C.-H. Chang, "Deep and shallow architecture of multilayer neural networks," 2015.

[18] Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, and N. de Freitas, "Sample efficient actor-critic with experience replay," 2017.

[19] 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," 2016.

[20] R. Islam, P. Henderson, M. Gomrokchi, and D. Precup, "Reproducibility of benchmarked deep reinforcement learning tasks for continuous control," 2017.

[21] K. Pasupa and W. Sunhem, "A comparison between shallow and deep architecture classifiers on small dataset," 2016.

[22] J. Brownlee, "Difference between classification and regression in machine learning," 2017.

[23] P. Henderson, R. Islam, P. Bachman, J. Pineau, D. Precup, and D. Meger, "Deep reinforcement learning that matters," 2019.

[24] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[25] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning.* The MIT Press Cambridge, Massachusetts, London, England, 1998.

[26] J. Schmidhuber, "Deep learning in neural networks: An overview," *Neural Networks*, vol. 61, pp. 85–117, 2015. Published online 2014; based on TR arXiv:1404.7828 [cs.NE].

[27] F. Chollet *et al.*, "Keras." `https://keras.io`, 2015.

[28] S. Zhang and R. S. Sutton, "A deeper look at experience replay," 2018.

[29] L. ji Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," in *Machine Learning*, pp. 293–321, 1992.

[30] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," 2016.

[31] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[32] F. Pardo, A. Tavakoli, V. Levdik, and P. Kormushev, "Time limits in reinforcement learning," Proceedings of Machine Learning Research, PMLR, 2018.

[33] L. Buşoniu, D. Ernst, B. D. Schutter, and R. Babuška, "Approximate dynamic programming with a fuzzy parameterization," ., 2010.

[34] D. Smilkov and S. Carter, "A Neural Network Playground kernel description." https://playground.tensorflow.org/#activation=tanh&batchSize=10&dataset=circle&regDataset=reg-plane&learningRate=0.03&regularizationRate=0&noise=0&networkShape=4,2&seed=0.08158&showTestData=false&discretize=false&percTrainData=50&x=true&y=true&xTimesY=false&xSquared=false&ySquared=false&cosX=false&sinX=false&cosY=false&sinY=false&collectStats=false&problem=classification&initZero=false&hideText=false, 2015.

[35] G. Montúfar, R. Pascanu, K. Cho, and Y. Bengio, "On the number of linear regions of deep neural networks," 2014.

[36] M. Raghu, B. Poole, J. Kleinberg, S. Ganguli, and J. S. Dickstein, "On the expressive power of deep neural networks," ., 2017.

[37] S. Ruder, "An overview of gradient descent optimization algorithms," 2016.

[38] D. P. Kingma and J. L. Ba, "Adam: A method for stochastic optimization," 2017.

[39] T. Chai and R. R. Draxler, "Root mean square error (rmse) or mean absolute error (mae)? – arguments against avoiding rmse in the literature," *Geosci. Model Dev*, 2014.

[40] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," ., 2010.

[41] I. Sutskever, J. Martens, G. Dahl, and G. Hinton, "On the importance of initialization and momentum in deep learning," 2013.

[42] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016. http://www.deeplearningbook.org.

[43] T. Poggio, K. Kawaguchi, Q. Liao, B. Miranda, L. Rosasco, X. Boix, J. Hidary, and H. Mhaskar, "Theory of Deep Learning III: explaining the non-overfitting puzzle," *Center for Brains, Minds and Machines*, 2018.

[44] Y. Li, "Deep reinforcement learning: An overview.," *CoRR*, vol. abs/1701.07274, 2017.

[45] J. Fu, K. Luo, and S. Levine, "Learning robust rewards with adverserial inverse reinforcement learning," *CoRR*, vol. abs/1710.11248, 2018.

[46] L. Matignon, G. J. Laurent, and N. L. Fort-Piat, "Reward function and initial values: Better choices for accelerated goal-directed reinforcement learning," in *ICANN*, 2006.

[47] A. J. Champandard, "Reinforcement learning," 2001.

[48] G. E. Uhlenbeck and L. S. Ornstein, "On the theory of the brownian motion," *Phys. Rev.*, vol. 36, pp. 823–841, Sep 1930.

[49] H. Anysz, A. Zbiciak, and N. Ibadov, “The influence of input data standardization method on prediction accuracy of artificial neural networks,” *Theoretical Foundation of Civil Engineering*, 2016.

[50] Y. Tassa, Y. Doron, A. M. T. Erez, Y. Li, D. de Las Casas, D. Budden, A. Abdolmaleki, J. Merel, A. Lefrancq, T. Lillicrap, and M. Riedmiller, “Deepmind control suite,” 2018.

[51] A. E. Orhan and X. Pitkow, “Skip connections eliminate singularities,” 2018.

[52] S. Zagoruyko and N. Komodakis, “Diracnets: Training very deep neural net- works without skip-connections,” 2018.

[53] G. Huang, Z. Liu, K. Q. Weinberger, and L. van der Maaten, “Densely connected convolutional networks,” 2016.

[54] F. Lagzi, T. Ball, and J. Boedecker, “Compact representations and pruning in residual networks,” 2018.