Bachelor Thesis

Czech
Technical
University
in Prague

F3

Faculty of Electrical Engineering
Department of Cybernetics

Neural networks for humanoid robot control

Silvestr Stanko

Supervisor: Ing. Zdeněk Buk Ph.D.
Field of study: Cybernetics and Robotics
Subfield: Robotics
May 2016

Acknowledgements

I would like to thank my supervisor, Mr Buk for listening and help with server runs and my friend Tereza for help with language corrections.

Declaration

Silvestr Stanko

Prague, 24. May 2016

Abstract

In this thesis I explore several different neural network architectures, including recurrent neural networks, and test them on a bipedal robot walking simulation. I further explore model-free continuous deterministic reinforcement learning, suitable for physical control tasks.

I apply these methods on a simpler inverse pendulum balancing tasks and later on the bipedal robot walking task. Both the pendulum and the robot are able to maintain balance without a physical model and explicitly specifying correct or incorrect behaviors.

Keywords: neural networks, LSTM, reinforcement learning, Q-learning

Supervisor: Ing. Zdeněk Buk Ph.D.

Abstrakt

V této bakalářské práci nejprve popisuji několik odlišných druhů neuronových sítí, včetně rekurentních, a testuji je na simulaci chůze humanoidního robota. Dále popisuji spojité, deterministické algoritmy posilovaného učení, vhodné pro řízení ve fyzikálním prostředí.

Tyto metody aplikuji na úlohách balancujícího inverzního kyvadla a dále na simulaci robota. Jak kyvadlo, tak robot jsou schopni udržovat rovnováhu, a to bez fyzikálního modelu či explicitní specifikace špatného či vhodného chování.

Klíčová slova: neuronové sítě, LSTM, posilované učení, Q-učení

Překlad názvu: Řízení humanoidního robota pomocí neuronových sítí

Contents

Chapter 1

Introduction

The benefits of having robots capable of walking and performing physical tasks in any environment are endless, ranging from effective industry workers to servants in every household.

Teaching robots how to walk is a long-time goal of control theory, robotics and artificial intelligence research. This task is particularly hard, because it involves controlling legs with high degrees of freedom in nonlinear state spaces. No general solution has yet been found and the so-far implemented solutions, often utilizing control theory, are model-based and require a full dynamics model of the robot to work.

1.1 Used methods

Neural networks are a class of function approximators, applicable for many different tasks including walking. A subclass of neural networks, recurrent neural networks (RNNs) is a perfect fit for the walking task because they are able to capture time-dependent relations between the input and output signals.

Reinforcement learning is a general learning 'framework' suitable for (among other things) model-free physical control tasks. However, the developed methods were often applicable only on low dimensional state-spaces with discrete control. Recent advances in deep learning and continuous reinforcement learning have shown great promise in the direction of a general algorithm capable of learning any physical control task.

In this thesis, I focus on these new reinforcement learning methods and combine them with recurrent neural nets. I further apply them on a selected robot, simulated in three dimensions.

1.2 Overview of the Thesis

In chapter 2, I describe the selected robot, simulator and simulation settings used in most of the experiments.

Chapter 3 briefly introduces the reader to the core ideas of supervised learning and details simple and recurrent neural networks including a few learning algorithms. These methods are then applied on a small data set to test if the neural nets are indeed capable of walking.

Chapter 4 contains an introduction to reinforcement learning theory, including all information necessary for understanding deep Q-learning.

Chapter 5 details the DDPG algorithm together with prioritized experience replay. I further describe the implementation process and most of the relevant experiments.

Chapter 2

Robot, Simulator

The first steps in making a robot walk were to select a robot model and a robotics simulator.

In this chapter I will present the selected simulation settings and the reasons that led me to the choice of the simulator and the robot model.

2.1 Simulator Selection

A robotics simulator is used to create embedded applications for a robot without depending physically on the actual machine, thus saving costs and time. In some cases, these applications can be transferred on the real robot (or rebuilt) without modifications.

I have put together a list of requirements that the simulators would have to fulfill in order to enable me to run the simulation effectively.

- **Headless simulation support:** It was clear that it would be necessary to repeat the simulation (episodes) multiple times during the learning process. Headless simulation, meaning running the simulator without GUI, reduces computational costs by not rendering the graphics and also allows remote runs (for example on a server).

- **Joints accessible each step of the simulation:** Some simulators do not support this and the joint access is variable, meaning some time steps might be skipped. I opted for simulators with deterministic joint access.

- **Open-source license:** Open-source licensing (and available source code) are a must-have when dealing with complex tasks, since they often require at least some code modification of the original release.

- **Active community support, documentation:** Lacking thorough documentation and an active community support are often obstacles for efficient learning and later implementation of the simulation settings. This was a main reason I removed most of the available simulators from the possible candidate list.

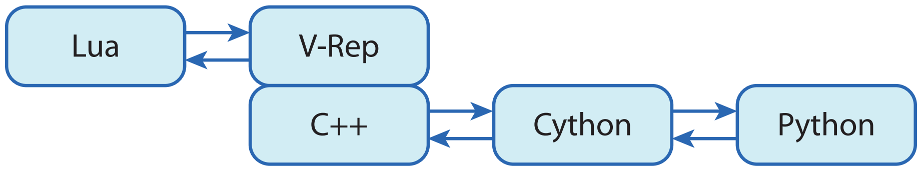

Figure 2.1: Main simulation loop

A good list of viable robotics simulators can be found at `https://en.wikipedia.org/wiki/Robotics_simulator`.

From here I chose two possible candidates: *Gazebo* and *V-Rep*.

After many unsuccessful attempts to make Nao model (I allready chose a model at that time) work under Gazebo (there are currently 3 main supported versions of Gazebo and it is not very clear under which the Nao model was supposed to work) I encountered an article [Nog] comparing Gazebo and V-Rep in the context of evolution algorithms (a similar setting to reinforcement learning) that recommended V-Rep in terms of user-friendliness and speed.

For these reasons I chose V-Rep as the simulation environment.

2.2 Simulation run structure

V-Rep supports 6 programming languages and I was hoping I could utilize Python machine learning libraries with which I had some experience. Unfortunately the latency between V-Rep's server and Python client turned out to be too large (~50 ms) for an effective and fast per-step communication.

V-Rep's plugin environment supports per-step communication without delays. Unfortunately, among other flaws (such as slightly confusing v_repMessage function), one could not send input arguments when running from a console, meaning only one running mode could be run at a time. Because of this, I chose to recompile V-Rep's client from source.

The run structure has changed over time and ended up looking like 2.1.

Running V-Rep from console starts a main C++ script that controls access to both the physics engine and the robot control scripts. The C++ script also processes input arguments and switches between different run-types (train, test, etc..).

Each time step, the main script calls some of the control functions that are all written in Cython[1]. Some simulation modes also use the V-Rep supported Lua scripts.

[1]Cython is a programming language that combines C's computational efficiency with Python's high level object-oriented programming. I use it as an interface between the C++ scripts that run the simulator and Python scripts that take care of the machine learning.

2.2.1 Simulation details

The simulator uses several different physics engines (Bullet, ODE, Vortex, Newton), I chose the default Bullet. The control loop is updated every *50*ms, the underlying physics engine loops ten times faster.

One problem across all physics engines is repeatability. In episodic setting the simulation behaves non-deterministically, meaning that repeating the same episode has slightly different outcomes every time. This is caused by the engine's float32 precision and fluctuations in the initial positions.

This problem could be addressed by fully restarting the engine each episodeepisode. However, each restart takes a few seconds which is too long considering that each episode takes most of the time less than a second to complete.

This is why I opted for not restarting fully each episode and instead relied on the learning algorithms to overcome this issue.

2.3 Robot

After a short research of the available humanoid robots I chose the NAO robot for the following reasons:

- it is one of few humanoid robots commercially available
- the Faculty of Information Technology, CTU has one model available for students, adding the possibility of future experiments on the physical model
- its simulation models were available for both V-Rep and Gazebo

The robot has 25 degrees of freedom and various sensors including a head camera, an accelerometer, a gyroscope, sonars and more. See figure 2.2.

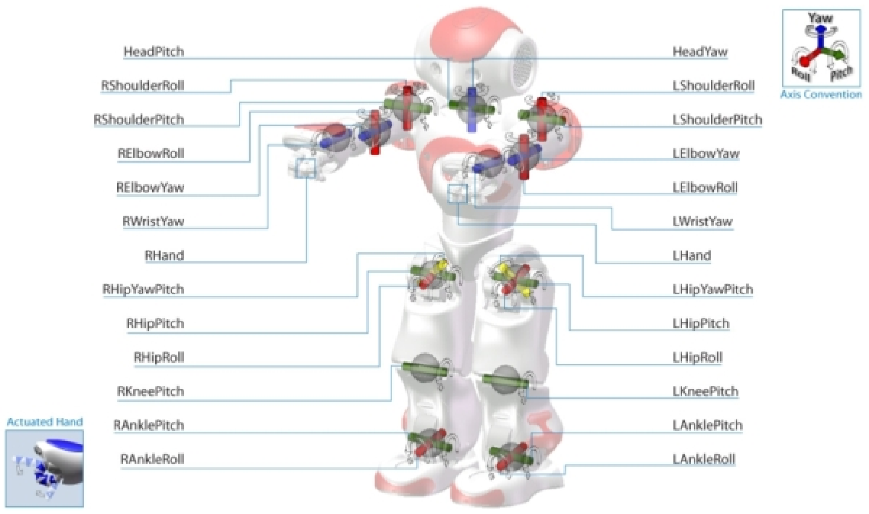

Figure 2.2: The NAO robot joint list[2]

2.3.1 State-Action space

Because of the nature of the walking task, it is unnecessary to control the head, arms or fingers. Therefore, the upper half of the body has been disabled in the simulator, leaving only the leg and torso joints controllable.

This leaves 12 separate joints (6 for each leg) for control.

While it should be possible to control the robot using joint positions only, it is always beneficial to provide as much information about the state as possible. I have, for this reason, included the joint velocities and 3 accelerometer outputs to the state representation. V-Rep supports joint control in three different ways:

- **Position control** V-Rep's most common form of control. Uses an internal PID controller.
- **Velocity control** Mainly a tool used by the position controller, the physics simulator applies maximal torque on the joint, until the desired velocity is reached.
- **Torque control** Direct torque control is unfortunately not available, however this can be bypassed by setting the target velocity very high. By setting maximal torque on the joint, the internal physics simulator uses this torque to reach the high velocity, effectively applying torque control.

I have used both the position and torque control in the experiments.

[2] courtesy of `http://doc.aldebaran.com/2-1/family/nao_dcm/actuator_sensor_names.html`

Chapter 3
Supervised Learning

In this chapter I will describe the simple and recurrent neural networks and their learning algorithms in a supervised setting. This is done primarily to test these models if they are capable enough to generate the outputs necessary for walking.

3.1 Introduction

Supervised learning is the machine learning task of inferring a function from labeled training data. The training data consist of a set of training examples. Each example is a pair consisting of an input vector and a desired output vector. A supervised learning algorithm analyzes the training data and produces an inferred function, which can be used for mapping new examples. Optimal scenario will allow the algorithm to correctly determine the class labels for unseen instances. This requires the learning algorithm to generalize from the training data to unseen situations in a "reasonable" way.[Wikd]

In mathematical terms, the goal of supervised learning is optimizing a loss function in form

$$L(\theta) = \sum_{i=1}^{N} L_i(\theta) \tag{3.1}$$

where $L_i(\theta)$ is a function of parameters θ and is associated with the i-th observation in the data set.

Common choice of the loss function when optimizing regression is the mean squared error (MSE). MSE has statistical implications, since the Gauss–Markov theorem states that:

"In a linear model in which the errors have expectation zero conditional on the independent variables, are uncorrelated and have equal variances, the best linear unbiased estimator of any linear combination of the observations, is its least-squares estimator." [Wika]

The overall solution minimizes the sum of the squares of the errors made in the results of every single equation.

$$\min_{\theta} \frac{1}{N} \sum_{i=1}^{N} (\mathbf{\hat{y}_i} - \mathbf{y_i})^2 = \min_{\theta} \sum_{i=1}^{N} (\mathbf{\hat{y}_i} - \mathbf{y_i})^2 \tag{3.2}$$

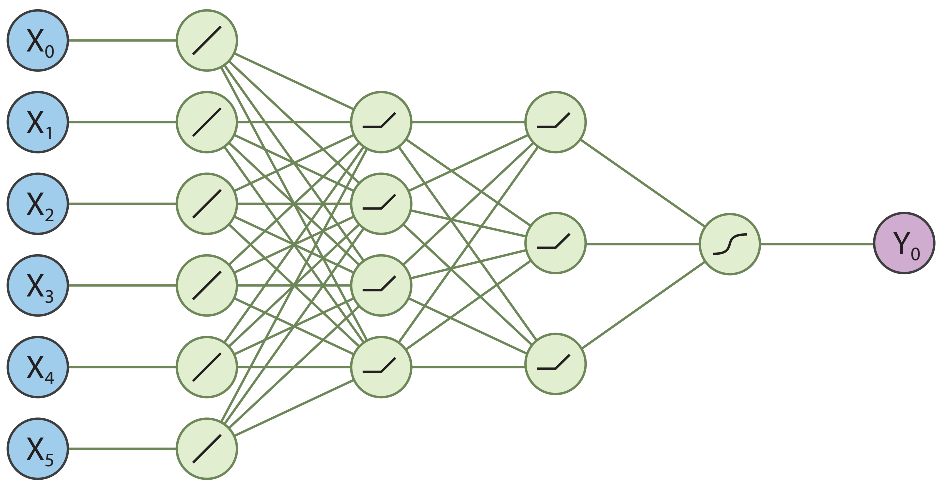

Figure 3.1: An example neural network with six inputs, two hidden layers and one output

$\mathbf{y_i}$ is the i-th observed output, $\hat{\mathbf{y}}$ is a function of parameters θ and i-th observed input $\mathbf{x_i}$.

$$\hat{\mathbf{y}}_\mathbf{i} = G(\theta, \mathbf{x_i}) \tag{3.3}$$

The inferred function G is usually a function approximator suitable for the task at hand.

3.2 Artificial Neural Networks

Neural networks (NN) are a family of function approximators inspired by biological processes in the brain. The network is generally presented as a system of interconnected 'neurons' which exchange messages with each other. The connections have numeric weights that can be tuned based on experience, making neural nets adaptive to inputs and capable of learning. A simple example can be seen in fig 3.1.

The basic feed-forward neural network comprises of several building blocks:

- **Layer:** Set of neurons in the same depth in the model. The common NN setup includes an input layer (the size of the input), output layer (the size of the output) and one or more hidden layers.
- **Activation function:** In each neuron the information propagates through an activation function. The function is often non-linear and it is this function that introduces non-linearity to the whole function approximation process.

  The function should meet a few requirements, most importantly it must be continuous and differentiable (If the learning is done using the function's gradient. This is not necessary in other settings, for example evolutionary learning).

Some examples of commonly used activation functions are: (fig 3.2)

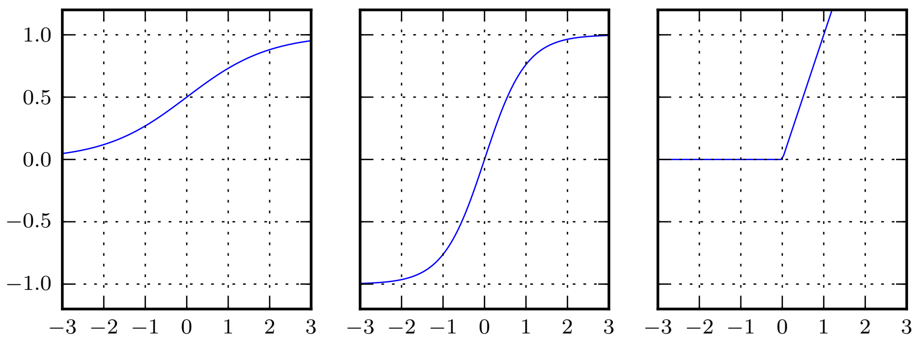

Figure 3.2: Commonly used activation functions in order: sigmoid, hyperbolic tangent, rectifier

- **Sigmoid function:** Due to its normalization properties, it is often used in the output layer when dealing with classification.

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

- **Hyperbolic tangent:** Another function used for its normalization properties, can be used in various settings, often in hidden layers.

$$\tanh(x) = \frac{e^{2x}-1}{e^{2x}+1}$$

- **Rectifier:** Because of the vanishing gradient problem appearing with previously mentioned activations in deep nets, the linear rectifier unit (ReLU) has gained on popularity in recent years. Efficient computation is another advantage of this activation.

$$\text{relu}(x) = \begin{cases} x & \text{for } x \geq 0 \\ 0 & \text{else} \end{cases}$$

The networks's wieghts are then used as parameters to be learned.
An example forward pass of the network on fig. 3.1 would look as follows:

$$h = \sigma\left(\underset{1\times 3}{W_3}\text{relu}\left(\underset{3\times 4}{W_2}\cdot\text{relu}\left(\underset{4\times 5}{W_1}\cdot x\right)\right)\right)$$

where W_i are matrices of parameters.

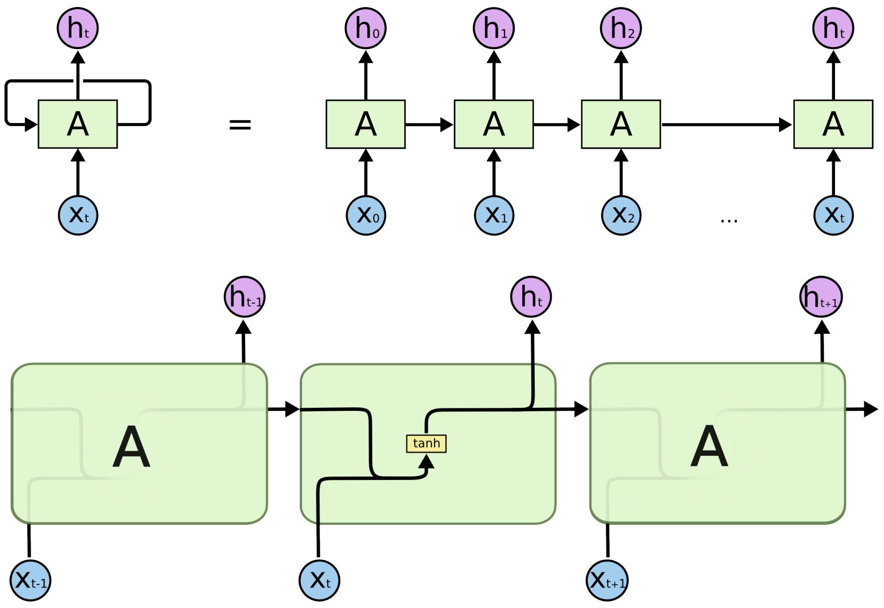

Figure 3.3: Visualization of unrolled fully recurrent network[1]

3.3 Recurrent Neural Networks

3.3.1 Motivation

An ideal function approximator should be able (with perfect information) to deal with the complex mechanism of bipedal (or indeed any) walking. In practice we should pick the methods such that the complexity of the desired solution is as low as possible.

While walking is surely a time-dependent task, the basic feedforward neural network does not take time into consideration (the information goes simply in→out). It would be beneficial to somehow implement a kind of memory that preserves the information from previous time steps. In the context of NNs, recurrent neural networks offer such time dependency features.

3.3.2 Theory

A recurrent neural network (RNN) is a class of artificial neural networks, where connections between units form a directed cycle. This creates an internal state of the network which allows it to exhibit dynamic temporal behavior. Unlike feedforward neural networks, RNNs can use their internal memory to process arbitrary sequences of inputs. This makes them applicable to temporally correlated tasks.

There are several classes of RNNs with different properties, I will mention two important ones.

[1] courtesy of `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

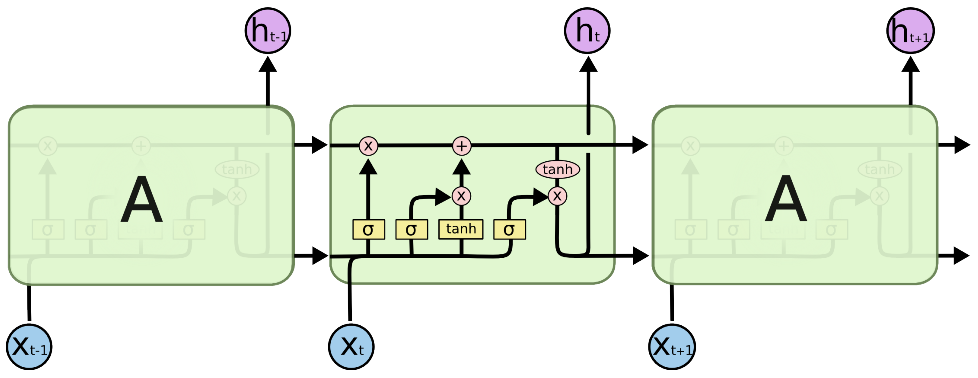

Figure 3.4: Visualization of unrolled LSTM[2]

3.3.3 Fully recurrent network

This is the basic recurrent architecture: a network of neurons, each with a directed connection to every other unit. Most architectures used nowadays are special cases.

The forward pass of one-layered fully recurrent network may look like this:

$$h_t = \sigma\left(\underset{n\times m}{W}\,x_t + \underset{n\times n}{U}\,h_{t-1} + \underset{n\times 1}{b}\right) \tag{3.4}$$

where h_t is the output of size $n \times 1$ and x_t is the input of size $m \times 1$ in time step t.

Vanishing gradient problem

The vanishing gradient problem is a difficulty found in training artificial neural networks with gradient-based learning methods and backpropagation.

In such methods, each of the neural network's weights receives an update proportional to the gradient of the error function with respect to the current weight in each iteration of training. The small size of these weights along with traditional activation functions cause the gradient updates to be small. In the recurrent setting this becomes a problem because with each time step the gradient becomes exponentially smaller. This causes the simple RNN to neglect the early updates the deeper we go into the backpropagation and effectively causes the network to 'forget' previous steps over time.

3.3.4 Long short term memory

Numerous researchers now use a deep learning RNN called the Long short term memory (LSTM)[HS97]. It is a deep learning system that unlike traditional RNNs doesn't have the vanishing gradient problem. LSTM introduces another hidden state and number of 'gates' - essentially special layers designed to promote networks memory capabilities.

The forward pass of one-layered fully recurrent network looks like this:

[2] courtesy of `http://colah.github.io/posts/2015-08-Understanding-LSTMs/`

$$\begin{array}{lcl}
f_t & = & \sigma\left(W_f x_t + U_f h_{t-1} + b_f\right) \\
i_t & = & \sigma\left(W_i x_t + U_i h_{t-1} + b_i\right) \\
c_t & = & \tanh\left(W_c x_t + U_c h_{t-1} + b_c\right) \\
C_t & = & f_t * C_{t-1} + i_t * c_t \\
o_t & = & \sigma\left(W_o x_t + U_o h_{t-1} + b_o\right) \\
h_t & = & o_t * \tanh(C_t)
\end{array} \tag{3.5}$$

where W, U, b are the network parameters, x_t is the input at time step t, C_t and h_t are the hidden states and h_t also acts as the network output. The main hidden state C_t is modified element-wise by a so-called forget gate layer f_t, input gate i_t and output gate o_t. This can be seen in fig 3.4. For more information about LSTMs, I recommend blogpost `http://colah.github.io/posts/2015-08-Understanding-LSTMs/` which contains clear and simple explanations.

3.4 Optimization

When dealing with real problems, it is often impossible to optimize a function analytically. Family of gradient descent (GD) methods uses the function's gradient to step in the direction of steepest descent (or ascent when finding maximum) and incrementally optimizes the function objective.

$$\theta_{t+1} = \theta_t - \alpha \nabla_\theta L \tag{3.6}$$

where α is the step size determining how far in the gradient direction we will step.

When optimizing a function approximator from sampled data points, common practice is to split data into batches and perform a gradient step on the batch. This is significantly more computationally effective then updating parameters for each sample separately (depending on the batch size).

3.4.1 Stochastic gradient descent

Stochastic gradient descent (SGD) is a gradient descent optimization method for minimizing an objective function that is written as a sum of differentiable functions. The core feature of SGD is picking the batches for GD at random. This change reduces the correlation in learning, introduced by taking the samples in order. SGD proved itself as an efficient and effective optimization method that was central in many machine learning successes.

3.4.2 Adam

The SGD itself can be quite limiting since it does not take into account the type of data being used and the change of the function gradient over time. Several SGD variants have been developed over the years using first and second order gradients and utilizing several types of momentum.

Algorithm 1 *Adam.* g_t^2 indicates the elementwise square $g_t \odot g_t$. Default settings are $\alpha = 0.001$, $\beta_1 = 0.9$, $\beta_2 = 0.999$ and $\epsilon = 10^{-8}$. All operations on vectors are element-wise. With β_1^t and β_2^t denotes β_1 and β_2 to the power t.

```
$m_0 \leftarrow 0$ (Initialize 1st moment vector)
$v_0 \leftarrow 0$ (Initialize 2nd moment vector)
$t \leftarrow 0$ (Initialize timestep)
while $\theta_t$ not converged do
    $t \leftarrow t + 1$
    $g_t \leftarrow \nabla_\theta f_t(\theta_{t-1})$ (Get gradients w.r.t. stochastic objective at timestep $t$)

    $m_t \leftarrow \beta_1 \cdot m_{t-1} + (1 - \beta_1) \cdot g_t$ (Update biased first moment estimate)
    $v_t \leftarrow \beta_2 \cdot v_{t-1} + (1 - \beta_2) \cdot g_t^2$ (Update biased second raw moment estimate)

    $\widehat{m}_t \leftarrow m_t / (1 - \beta_1^t)$ (Compute bias-corrected first moment estimate)
    $\widehat{v}_t \leftarrow v_t / (1 - \beta_2^t)$ (Compute bias-corrected second raw moment estimate)
    $\theta_t \leftarrow \theta_{t-1} - \alpha \cdot \widehat{m}_t / (\sqrt{\widehat{v}_t} + \epsilon)$ (Update parameters)
end while
return $\theta_t$ (Resulting parameters)
```

Adam is a method for efficient stochastic optimization that only requires first-order gradients with little memory requirement. The method computes individual adaptive learning rates for different parameters from estimates of first and second moments of the gradients [KB14]. See algorithm 1.

Adam is very useful when dealing with high dimensional spaces and on-line learning settings. This method has proved instrumental when optimizing the main algorithm of this thesis.

3.4.3 Overfitting, Regularization

Overfitting is a common problem that occurs when a statistical model (in our case NN) describes a random error or noise instead of the underlying relationship. Overfitting generally occurs when a model is excessively complex, such as having too many parameters relative to the number of observations. A model that has been overfit will generally have poor predictive performance, as it can exaggerate minor fluctuations in the data.

Regularization is a technique used for prevention of overfitting by introducing additional information to a loss function. Most common practice is to add a weighted *l2* norm of model's parameters to the original loss function in form

$$L_{\text{new}}(\theta) = L(\theta) + \lambda \|\theta\|_2^2 \tag{3.7}$$

This forces the optimization algorithm to keep the parameter's absolute value low and by doing so prevents extreme values from occurring. High parameter values are often the cause of overfitting.

3.5 Experiments

The main goal of the supervised experiments was to check which models were capable of learning pre-recorded sequences.

3.5.1 Data

The data used for learning were extracted from the available V-Rep Nao model. Part of the model is an example script with pre-recorded sequence of desired joint positions. This sequence was played for $N = 1000$ time steps in the simulator and the joint positions, joint velocities and accelerometer data were extracted each time-step.

This data served as the input at each step. For the output, the recorded joint positions were shifted by one time-step, so that the NN's goal is to predict the desired joint positions one step into the future.

3.5.2 Implementation, setup

I have implemented a fully recurrent neural network using only python's native *numpy* library used for matrix operations. The network was successfully able to mimic the training data, but the implementation itself was troublesome because every small change in the model (for example different activation) forced the whole backpropagation algorithm to change.

I rewrote the net using the *Theano* library which I picked for its complex symbolic differentiation capabilities. I constructed sort of a framework for neural networks, partially inspired by the *keras* library. The 'framework' allowed me to construct networks of several neural layers by just specifying the sizes and types of layers consecutively.

The supported layers are:

- Fully connected (dense) layer
- Fully recurrent layer
- LSTM layer

The Theano library (and a few others - like tensorflow) simplify things greatly. Instead of working out the backpropagation updates by hand (or other means), it is only necessary to input symbolic matrix equations of the forward run and compute the gradient using the library's *grad* function.

When training the recurrent networks, I had a few different choices:

- **Backpropagation through time:** a straightforward learning technique, that unfolds the recurrent network in time by stacking identical copies of the RNN, and redirecting connections within the network to obtain connections between subsequent copies. This process creates a simple (but deep) feedforward network that allows for the use of the basic backpropagation algorithm.

- **Limited BPTT:** The previous method however forces the network to unfold once for each time step. This is computationally unsustainable for longer sequences. A simple and commonly used variant of BPTT uses a finite history and limits the number of unfolds.
- **Real-time recurrent learning:** RTRL is a gradient-descent method which computes the exact error gradient at every time step. The standard version of the algorithm has time complexity $O(n^4)$ [MLK95] (where n is number of processing units in the network), which is quite high for effective implementation.

I chose Limited BPTT, because it supports efficient repeated batch learning (unlike BPTT and RTRL), needed for the reinforcement learning experiments.

3.5.3 Network notation

I will use the following notation (mimicking notation of the python 'framework') in the remainder of this thesis:

- **{ }:** basic network container
- **{*i*, *j*, ...}:** each number represents a layer of stated size, the layers are connected in succession, the input and output layers are not explicitly mentioned
- **{lstm, rnn}:** virtual recurrent layers, the input and output sizes are specified by sizes of the previous and next layer

An example net from figure 3.1 would look like this: {4, 3}
Unless stated otherwise, the output layer has a tanh activation (to bound the output), hidden layers go through relu activation.

3.5.4 Experiments

The data itself are unfortunately quite limiting, because the recorded sequence is very short and after around 300 time steps it starts repeating itself. Because of this I couldn't use the usual train/validation/test splits and was unable to measure the model's performance by standard means. Instead I evaluated the learned nets on the robot itself, checking which models were able to walk.

I constructed several network architectures and retrained them repeatedly with different initial weights. Then I tested these networks on the robot itself, measuring the time and distance over several (10) runs. This was done to check the networks capability of overcoming the stochastic effects introduced by the simulator.

The nets used in the supervised experiments were the following: {}, {10}, {50}, {10, 10}, {50, 50}, {100, 100}, {lstm}, {lstm, 10}, {lstm, 100}, {rnn}, {rnn, 10}, {rnn, 100}

This evaluation process is not entirely correct, because in the supervised learning setting, we are only fitting the provided data and the model doesn't know anything about the real walking task. However this was done mainly to test if the nets were even capable of walking and not to learn them to walk in the full extent.

3.6 Conclusion

About a quarter of the networks were capable of walking, however often at the cost of stability. Perhaps the most stable net still capable of walking was {rnn, 10}. The results and comparisons can be seen in figures 3.5, 3.6 and also in video 4. For training details, see A.1.

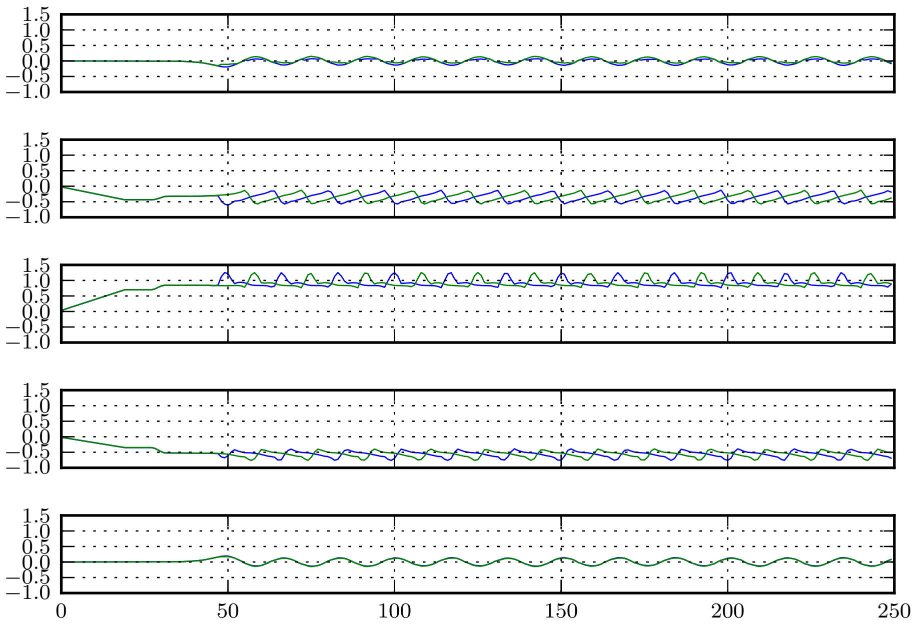

Figure 3.5: The recorded Nao joint angles dataset.

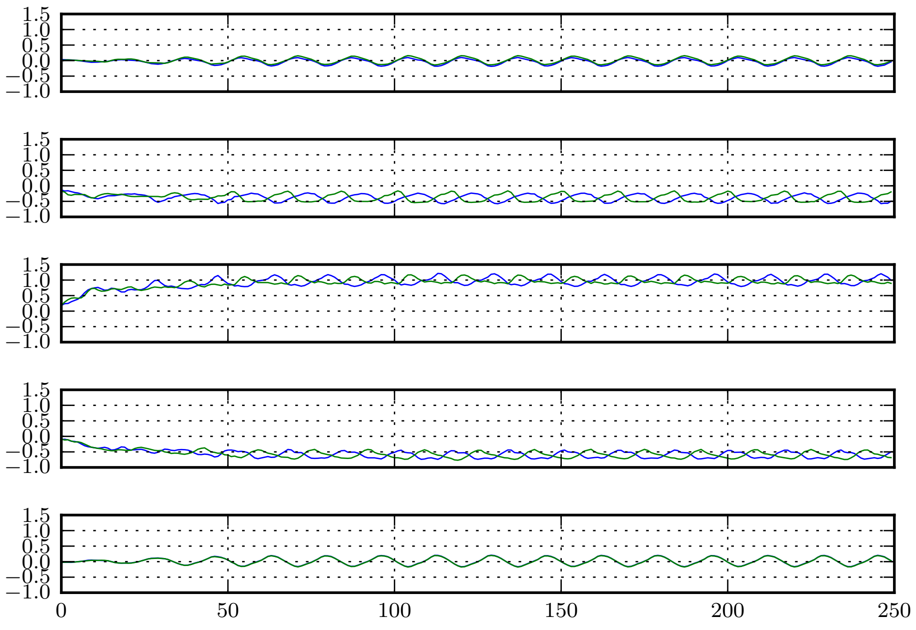

Figure 3.6: The recorded joint angles of a learned {rnn,10} net on the robot.

Chapter 4
Reinforcement Learning

Reinforcement learning (RL) is an area of machine learning inspired by behaviorist psychology, concerned with how software agents ought to take actions in an environment so as to maximize some notion of cumulative reward. The problem, due to its generality, is studied in many other disciplines, such as game theory, control theory, operations research, information theory, simulation-based optimization, multi-agent systems, swarm intelligence, statistics, and genetic algorithms.

Reinforcement learning differs from standard supervised learning in that correct input/output pairs are never presented, nor sub-optimal actions explicitly corrected. Further, there is a focus on on-line performance, which involves finding a balance between exploration (of uncharted territory) and exploitation (of current knowledge) [Wikc].

In machine learning, the environment is typically formulated as a Markov decision process (MDP). The main difference between classical techniques and reinforcement learning algorithms is that the latter do not need knowledge about the MDP and they target large MDPs where exact methods become infeasible.

4.1 Introduction

The basic RL model consists of:

- a state in a set of environment states $s \in \boldsymbol{S}$
- an action in a set of actions $a \in \boldsymbol{A}$
- rules of transitioning between states $s_t \to s_{t+1}$
- rules that determine the scalar immediate reward $r \in \mathbb{R}$ of a transition

The rules are often stochastic, especially the rules of state transitioning. Depending on the type of the problem, we divide the MDPs to fully observable (agent has full access to all information about the state) and partially observable (some information about the state is not available).

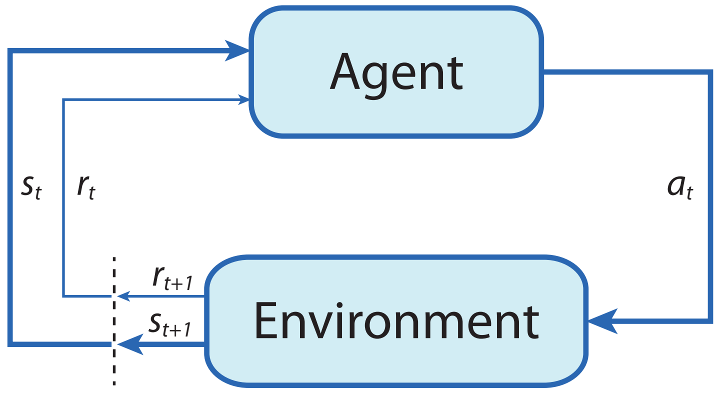

Figure 4.1: Interaction of a RL agent with the environment

A RL agent interacts with the environment at discrete time-steps, observing the state s_t by receiving an observation o_t[1] and typically a reward r_t. The agent then chooses an action a_t which is 'sent' to the environment. The environment moves to another state $s_t \to s_{t+1}$.

The goal of a RL agent is to collect as much reward as possible.

4.1.1 Policy

A policy π is used to pick actions in the MDP. In general, the policy is stochastic and denoted by $\pi_\theta : S \to P(A)$ where $\theta \in \mathbb{R}^n$ is the parameter vector and $\pi_\theta(a|s) = \mathbb{P}[a|s;\theta]$ selects an action according to some probability measure $P(A)$.

4.1.2 Reward

The reward in time-step t is a function $r_t(s_t, a_t)$ that returns the immediate reward of the transition $s_t \to s_{t+1}$. The return R_t^γ is the total discounted reward from time step t onward.

$$R_t^\gamma = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k) \tag{4.1}$$

$\gamma \in \langle 0, 1 \rangle$ is a coefficient capturing how much do we care about events in the far future[2].

The goal of a RL agent is to find a policy that maximizes the expected cumulative reward from the start state, denoted by the objective function

$$J(\pi) = \mathbb{E}[R_0^\gamma | s_0; \pi] \tag{4.2}$$

[1]To simplify notation, we further consider $o_t \equiv s_t$

[2]$\gamma = 0$ means we only care about the immediate reward, $\gamma = 1$ means the value should depend on the future fully. This value is often close to 1 and is task-dependent.

We denote the probability density at state s_{t+k} after transitioning for k time steps from state s_t as $p(s_t \to s_{t+k}, k, \pi)$. We also denote the discounted state distribution as $\rho^\pi(s_{t+k}) = \int_S \sum_{t=1}^{\infty} \gamma^{k-1} p(s_t)$

4.1.3 Finite differences

I would like to mention one extremely simple RL approach. Let's assume we have an agent $\mu_t heta(s_t) = a_t$ (doesn't have to be deterministic).

By using finite gradient differences, we can approximate gradients of the objective function as:

$$\frac{\partial J}{\partial \theta} \approx \frac{J(\theta + \epsilon) - J(\theta)}{\epsilon} \tag{4.3}$$

This means we have to run a sample episode for each parameter we want to update. This makes this method highly inefficient, however when using agents with very few parameters, even this method has some practical uses (demonstrated for example on the AIBO robots, where it was used to speed up the walking process).

I have tried this method on the network learned in chapter 3. The robot was actually learning to move faster, until a certain point where it became unstable.

4.1.4 Action-Value function

A *value function* is a function defined as the expected total discounted reward from state s_t when picking actions according to the policy π.

$$V^\pi(s_t) = \mathbb{E}[R_t^\gamma | s_t; \pi] \tag{4.4}$$

An *action-value function* inputs a state and an action and outputs the expected return from state s_t after taking action a_t and afterwards picking actions according to the policy π.

$$Q^\pi(s_t, a_t) = \mathbb{E}[R_t^\gamma | s_t, a_t; \pi] \tag{4.5}$$

This difference may seem subtle, but is significant. It is common practice to pick actions that maximize the expected return - this would be impossible with the simpler value function.

4.2 Action-Value function approximation

The common approaches used for model-free value function approximation are:

- **Monte Carlo Methods:** Applicable only for episodic problems, the values of $V(s_t)$ are updated in the direction of the total reward seen when starting from state s_t. Given enough time, this procedure can construct a precise estimate of the V value.

$$V(s_t) \leftarrow V(s_t) + \alpha(R_t - V(s_{t+1})) \tag{4.6}$$

R_t is the actual return, α some learning constant.

MC methods however work effectively only for small MDPs and are overall slow.

- **Temporal Difference Methods:** TD methods deal with the limiting character of the MC methods by using it's own estimates to correct itself.

$$V(s_t) \leftarrow V(s_t) + \alpha(r_{t+1} + \gamma V(s_{t+1}) - V(s_t)) \tag{4.7}$$

$r_{t+1} + \gamma V(s_{t+1})$ is the estimated return.

4.2.1 Q-learning

Q-learning is an extension of TD learning that can be used for learning the action-value function. It simplifies the action policy dynamic by eliminating the policy from the equation and chooses actions that maximize the predicted reward. See 2.

Algorithm 2 Q-learning algorithm

```
Initialize $Q$-values arbitrarily for all state-action pairs.
for each step do
    Choose an action $a_t = \text{argmax}_a Q(s_t, a)$
    Execute action $a_t$, observe reward $r_t$ and observe new state $s_{t+1}$
    Update $Q(s_t, a_t) = Q(s_t, a_t) + \alpha[r_t + \text{argmax}_a Q(s_{t+1}, a) - Q(s_t, a_t)]$
end for
```

When dealing with small MDPs, it is possible to store the action-value values in a lookup table. This becomes unfeasible with larger MDPs and requires the use of a function approximator. Neural networks are a natural choice for this task, because they can capture highly non-linear relationships between inputs and outputs of the a-v function.

Directly implementing Q-learning with neural networks proved to be unstable in many environments. Since the network being updated is also used for calculating the target value, the Q updates are prone to divergence.

These methods, combined with actor-critic algorithms were used successfully on state-spaces with lower dimensions, however struggled when applied to high state space domains. This changed in 2015 when [MKS+15] successfully applied modified Q-learning algorithm on the Atari domain.

4.3 Deep Q-learning

The deep Q-learning algorithm (DQN) has been the first algorithm capable of solving array of diverse and challenging tasks. The tasks, without going to too much detail, were to maximize the game score on several different Atari games. The inputs to the systems were pixels a human would see when playing the games. The outputs were several different actions a human could take by using a controller (like up, down, etc..).

Convolutional neural networks followed by fully connected layers were used as the action-value function approximator. The algorithm therefore combines reinforcement learning with deep learning.

What made the algorithm stable compared to past attempts, were two main changes over the previously used Q-learning algorithm. The Q-learning used a replay buffer and a so-called target network.

- **Replay buffer:** The algorithm uses a large replay buffer from which it uniformly samples a minibatch each step of the learning process. This shift from online to batch learning has lowered the correlation of learned samples and as such has brought the most significant performance improvement.
- **Target network:** Another previously unutilized method is the addition of another neural network - a copy of the main one. The other network is not updated each learning step, it is instead updated less often with the original network weights. This brings the learning process closer to supervised learning for which robust solution exists.

See 3 for the complete algorithm.

Algorithm 3 Deep Q-learning algorithm

```
Randomly initialize action-value function $Q$ with weights $\theta^Q$.
Initialize target network $Q'$ with weights $\theta^{Q'} \leftarrow \theta^Q$
Initialize replay buffer $R$
for episode = 1, M do
    Receive initial observation state $s_1$
    for t = 1, T do
        With probability $\epsilon$ select a random action $a_t$,
        otherwise select $a_t = \text{argmax}_a Q(s_t, a_t)$
        Execute action $a_t$, observe reward $r_t$ and observe new state $s_{t+1}$
        Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
        Sample a random minibatch of $N$ transitions $(s_t, a_t, r_t, s_{t+1})$ from $R$
        Set $y_t = \begin{cases} r_t + \gamma Q'(s_{t+1}, a_{t+1}) & \text{for non terminal state} \\ r_t & \text{for terminal state} \end{cases}$
        Update $Q$ by minimizing the loss: $L = \frac{1}{N}\sum_i (y_t - Q(s_t, a_t|\theta^Q))^2$
        Every C steps update the target networks: $\theta^{Q'} \leftarrow \theta^Q$
    end for
end for
```

DQN was able to solve a set of complex problems in continuous state space domain, however the action domain consisted of only a few discrete actions.

The DDPG algorithm was able to combine deep Q-learning with actor-critic algorithms to overcome this limitation.

4.4 Actor-Critic Algorithms

Picking the actions that maximize the future rewards lays hard requirements on the AVF approximator. It is easy to imagine that even minor changes to the function's parameters will change the best action dramatically (because max only cares about the absolute values).

The *actor-critic* (see 4) is an extension of the ideas introduced by Q-learning and eases these requirements. Instead of picking the actions that maximize the future rewards (and effectively removing the actor from the equation), an actor network is introduced for picking the actions independently from the critic Q network.

It is a widely used architecture based on the policy gradient theorem used for stochastic policies

$$\begin{aligned}\nabla_\theta J(\pi_\theta) = \int_S \rho^\pi(s) \int_A \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \mathrm{d}a\mathrm{d}s = \\ \mathbb{E}_{s\sim\rho^\pi, a\sim\pi_\theta}[\nabla_\theta \log \pi_\theta(a|s) Q^\pi(s,a)]\end{aligned} \tag{4.8}$$

The algorithm consists of two crucial components.

The *actor* $\pi(s)$ is used for picking actions each time step and learns by gradient ascent of equation 4.8.

The *critic* $Q^\pi(s,a)$ is used for AVF approximation.

Algorithm 4 Actor-Critic algorithm

```
Initialize critic $Q(s,a|\theta^Q)$ and actor $\pi(s|\theta^\pi)$ with weights $\theta^Q$ and $\theta^\pi$.
Observe initial state $s_t$
for each step do
    Choose an action $a_t = \pi(s_t)$
    Execute action $a_t$, observe reward $r_t$ and observe new state $s_{t+1}$
    $y = r_t + Q(s_{t+1}, \pi(s_{t+1}))$
    $\theta^Q \leftarrow \theta^Q - \alpha\nabla_{\theta^Q}(y - Q(s_t,a_t))^2$
    $\theta^\pi \leftarrow \theta^\pi + \alpha\nabla_{\theta^\pi} J$
end for
```

4.4.1 Deterministic Policy Gradient

Formerly mentioned methods have been used for stochastic policies in the past. For some tasks however, specifically continuous control (like walking), it doesn't make much sense to consider stochastic action-picking.

Recently, Silver et al [SLH+14] proved that the deterministic policy gradient exists (it was previously thought it doesn't) and has the simple form of

$$\begin{aligned}\nabla_\theta J(\mu_\theta) = \int_S \rho^\mu(s) \nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)|_{a=\mu_\theta(s)} \mathrm{d}s = \\ \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)]\end{aligned} \tag{4.9}$$

Chapter 5

Deep Deterministic Policy Gradient

In this section I describe the DDPG algorithm, detailing the implementation process and the experiments.

5.1 The DDPG Algorithm

Lillicrap, Hunt et al [LHP+15] have taken the ideas underlying the success of Deep Q-Learning and combined them with the deterministic policy gradient.

The main difference between DQL and DDPG lies in the choice of the actor. In DQL, the actor is not explicitly represented and each action is chosen as the $\text{argmax}_a Q(s,a)$. This approach is feasible only in small action spaces and in larger spaces it would require demanding parameter search during each learning step. In DDPG, the actor is another function approximator that inputs state s_t and outputs action a_t, $\mu_\theta(s_t) = a_t$.

The Deep Deterministic Policy Gradient (DDPG, see algorithm 5 at the end of this chapter) has been successfully applied on an array of diverse continuous control tasks including cartpole swingup, 2D legged locomotion or dexterous manipulation. Some of the tasks were also solved directly from visual inputs. However, it has not been used for locomotion in three dimensions.

5.2 Implementation

Following the implementation of the supervised methods, I continued with the Theano library. Making the code work has proved to be quite a challenge due to several important details that slipped the current (at the time of implementation the paper was just submitted to the ICLR 2016 conference) version of the DDPG paper.

5.2.1 DPG Simplification

The deterministic policy gradient, as presented in both DPG and DDPG papers is in the form of

$$\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s\sim\rho^\mu}[\nabla_\theta \mu_\theta(s) \nabla_a Q^\mu(s,a)]$$

To add confusion, in the DPG paper the expression is presented in reverse order ($\nabla_\theta J(\mu_\theta) = \mathbb{E}_{s\sim\rho^\mu}[\nabla_a Q^\mu(s,a)\nabla_\theta\mu_\theta(s)]$).

The equation consists of two terms:

- $\nabla_a Q^\mu(s,a)$: this is a row vector containing the expression

$$\nabla_a Q^\mu(s,a) = \left[\frac{\partial Q}{\partial a_1}, \dots, \frac{\partial Q}{\partial a_n}\right]$$

- $\nabla_\theta \mu_\theta(s)$: this is a matrix of size $n \times l$[1]

$$\nabla_\theta\mu_\theta(s) = \begin{bmatrix} \frac{\partial a_1}{\partial \theta_1} & \cdots & \frac{\partial a_1}{\partial \theta_l} \\ \vdots & \ddots & \vdots \\ \frac{\partial a_n}{\partial \theta_1} & \cdots & \frac{\partial a_n}{\partial \theta_l} \end{bmatrix}$$

When comparing the dimensions, the order became clear ($1\times n \cdot n \times l = 1 \times l$). What's more important, after multiplying these terms we arrive at the following expression

$$\nabla_a Q^\mu(s,a)\nabla_\theta\mu_\theta(s) = \left[\sum_{i=1}^{n} \frac{\partial Q}{\partial a_i}\frac{\partial a_i}{\partial \theta_1} \quad \cdots \quad \sum_{i=1}^{n} \frac{\partial Q}{\partial a_i}\frac{\partial a_i}{\partial \theta_l}\right]$$

After examining the equations, it became clear that each term represents the multidimensional chain rule of

$$\sum_{i=1}^{n} \frac{\partial Q}{\partial a_i}\frac{\partial a_i}{\partial \theta_j} = \frac{\partial Q}{\partial \theta_j}$$

This simplifies the original expression as

$$\nabla_{\theta^\mu} J = \nabla_a Q^\mu(s,a)\nabla_\theta\mu_\theta(s) = \nabla_{\theta^\mu} Q(s,\mu(s)) \tag{5.1}$$

This rewritten formula is very intuitive, since it says that following the gradient of the reward function with respect to actor's parameters is the same as following the gradient of the critic with respect to the actor's parameters. We are trying to maximize the critic's output!

While in the end I found the equations mentioned in the DPG paper, they are not represented as the final result for some reason.

This small change (basically just a rewrite) has shed light for me on the problematic and allowed the implementation of a much simpler and cleaner (and faster) version of the DDPG algorithm.

[1] m = size of the state feature vector, n = size of the action, l = number of actor parameters

5.2.2 Critic updates

Another fact missing from the DDPG paper was the complete computation of the y term used for learning the critic.

In the paper, the y-term is computed as $y_i = r_i + \gamma Q'(s_{i+1}, \mu(s_{i+1}))$. This unfortunately doesn't tell us what to do with the terminating state. I followed the same formula used in DQN, where it is necessary to ground the critic in the terminating state as $y_i = r_i$.

When conducting experiments, I tried several different settings and it turned out that even when the terminating state was not bounded, the algorithm was able to converge on simpler tasks. This is most probably due to the fact that, compared to DQN, DDPG doesn't care about the absolute values of the critic and only uses it's gradient.

In the end I opted for the bounded version on all of the experiments.

5.2.3 Adam

In my experience, the difference between using SGD and its variants (AdaGrad, AdaMax, RMSProp, Adam) is usually a few percent points in the loss function value after training. This is why I skipped the implementation of Adam (using simple SGD) and focused on different parts of the code.

This has turned out to be a crucial mistake, because the algorithm without Adam was unable to converge. It is unclear if the authors knew about this issue, since they didn't mention it in the paper. See 5.1 for the comparison.

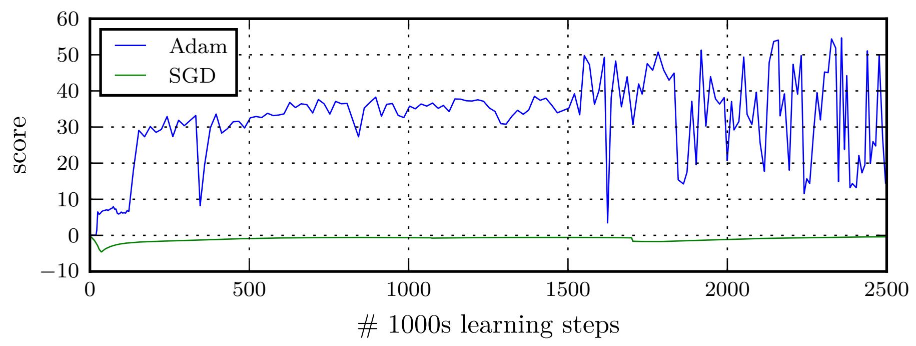

Figure 5.1: Comparison of Adam and SGD algorithms on the swingup task. Actor={50,50}, Critic={100,100}

5.3 Exploration vs Exploitation

Exploration vs exploitation is a dilemma that a RL agent faces when optimizing for some loss function that is a function of the environment.

At any given point in time the agent has to choose an action from the action space. The exploitation approach would be to choose the action that he knows will produce the most reward. However it is possible (and often the case) that taking another action will lead to a grater reward down the path.

The exploration in the DDPG paper is done by adding time-correlated noise to the actions. The used noise is generated with Ornstein-Uhlenbeck process.

5.3.1 Ornstein-Uhlenbeck Process

The Ornstein–Uhlenbeck (OU) process is a stochastic process that, roughly speaking, describes the velocity of a massive Brownian particle under the influence of friction. The process is stationary, Gaussian, and Markovian, and it is the only nontrivial process that satisfies these three conditions, up to allowing linear transformations of the space and time variables. The process can be considered to be a modification of the random walk in continuous time, or Wiener process, in which the properties of the process have been changed so that there is a tendency of the walk to move back towards a central location, with a greater attraction when the process is further away from the center. [Wikb]

An OU process satisfies the following stochastic differential equation

$$\mathrm{d}x_t = \theta(\mu - x_t)\mathrm{d}t + \sigma \mathrm{d}W_t \tag{5.2}$$

where $\theta > 0, \mu, \sigma > 0$ are the parameters and W_t denotes the Wiener process. The incremental solution can be then written as

$$x_t = x_0 e^{-\theta t} + \mu(1 - e^{-\theta t}) + \frac{\sigma}{\sqrt{2\theta}} e^{-\theta t} W_{e^{2\theta t}-1} \tag{5.3}$$

This incremental solution is used in the final implementation. See 5.3 for example runs.

The noise is an instrumental part of the DDPG algorithm and without it, there is a little chance the model will find any good solutions and most likely gets stuck in a local extreme. See 5.2 for comparisons with and without using noise.

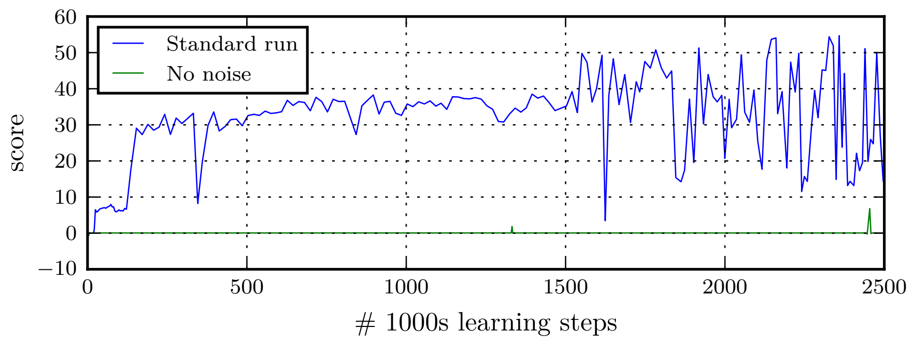

Figure 5.2: Comparison runs on the swingup task. Actor={50,50}, Critic={100,100}

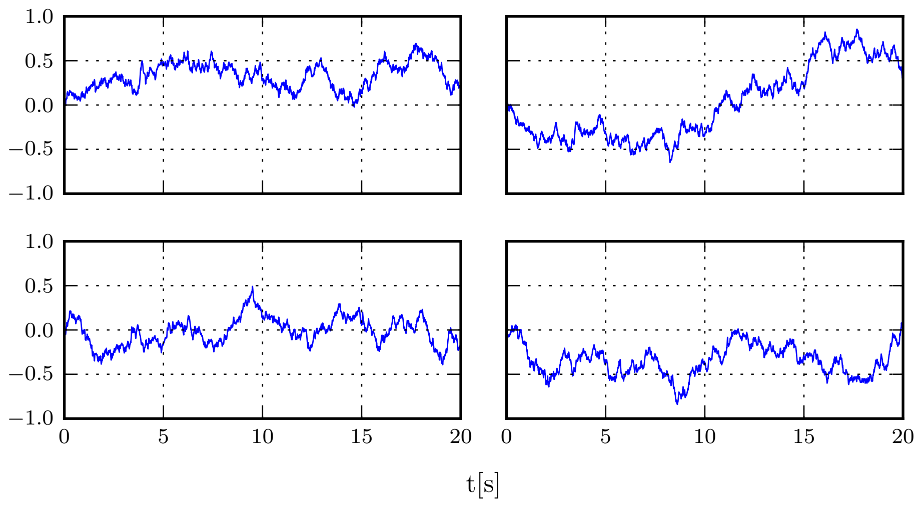

Figure 5.3: Example runs of OU process with parameters: $\mu = 0$, $\theta = 0.15$, $\sigma = 0.2$

5.4 Prioritized Experience Replay

Experience replay lets online reinforcement learning agents remember and reuse experiences from the past. In the DDPG paper (and prior work), training examples were uniformly sampled from the replay memory. This approach unfortunately doesn't differentiate significance of the training examples. This leads to slow learning, because newly discovered rewarding subpolicies are trained at the same rate as the unrewarding ones.

[SQAS15] presented a few prioritization processes, that speed up the learning process and even lead to better found policies. These processes, together with DQN variant called Double DQN are the current state-of-the-art on the Atari task.

The TD error

$$\delta = (r_t + \gamma Q(s_{t+1}, a_{t+1})) - Q(s_t, a_t) \tag{5.4}$$

can be viewed as a sort of an indicator of how 'surprising' a single sample is.

The technique presented in the paper uses proportional prioritization depending on the TD error of each sample. The probability of sampling a transition is therefore

$$P(i) = \frac{p_i^\alpha}{\sum_k p_k^\alpha}, \quad p_i = |\delta_i| + \epsilon \tag{5.5}$$

where α determines how much prioritization is used, with $\alpha = 0$ corresponding to the uniform case.

Prioritized replay introduces bias because the new update distribution does not correspond with the expectation distribution (uniform). This can be corrected by using importance-sampling weights

$$w_i = \frac{P(i)^{-1}}{\max_i w_i} \tag{5.6}$$

that fully compensates for the non-uniform probabilities.The weights are normalized to prevent large values from occurring. These weights can be folded into the Q-learning update by multiplying the gradient of each sample by w_i.

5.4.1 Implementation

The algorithm proposed in the paper is suited for Q-learning (using critic only), however there is no reason why the same algorithm shouldn't work in the case of actor-critic variants. To compensate the importance-sampling distribution bias with the actor, I used the same batches and w weights as in the critic.

After implementing the algorithm with a simple array, the complexity is $O(nN)$ where N is the size of the replay memory, n is batch size. This has caused that most time spent learning was wasted on the priority sampling. Further optimization was required.

I implemented a prioritized queue (heap queue) and used it for maintaining a greedy version of the algorithm, lowering the complexity to $O(n \log_2 N)$. This version (also proposed in the paper) chose n of the most 'surprising' candidates. As predicted in the paper, the algorithm was unable to learn anything. This is because the expectation of the deterministicaly picked values did not respect the probability distribution $\mathbb{E}_{s\sim\rho}$.

Sum trees

To lower the complexity and preserve the probability sampling, it was necessary to construct a binary sum tree. Sum tree is a tree where each parent node contains the sum of its children values. In this implementation, only the deepest nodes contained the replay memories.

To simulate the priority sampling, the value of the root node was evenly split into n ranges and a random value was sampled from each range.

This value was then traced in the tree (to trace a value means to find the most right node, where sum of the nodes on the left is less then this value) by traversing from top to bottom, see 5.4.

Implementation was done in a heap-like array, where index i is parent to nodes $2i+1, 2i+2$. The complexity of adding, replacing and updating a node is $O(\log_2 N)$.

For comparison between uniform and proportional distributions, see 5.5. The proportional version is able to find better solutions faster.

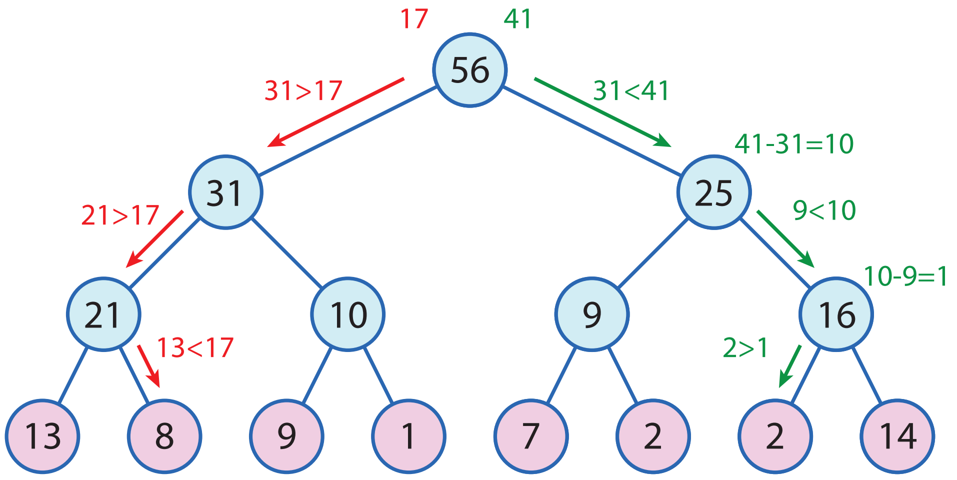

Figure 5.4: The sumtree structure used in prioritization. Example traces are shown in red and green.

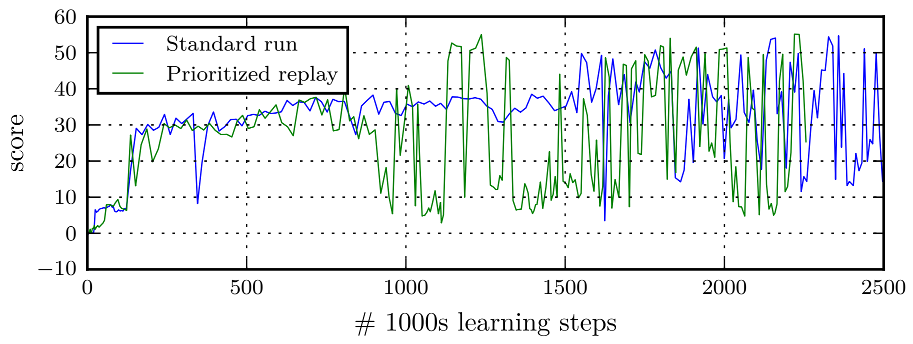

Figure 5.5: Comparison runs on the swingup task. Actor={50,50}, Critic={100,100}, Small Actor={} , Small Critic={}

5.5 Experiments

The experiments were conducted on several different networks. I experimented with feedforward and recurrent networks in critic, actor or both.

The training time suffered when both actor and critic used recurrent nets, because the inputs and the internal states had to have three dimensions instead of four. So I did not use the fully recurrent setup and instead tested on fully feedforward or recurrent actor only.

5.5.1 Cartpole simulation

Because it was uncertain if the algorithm would be able to make the robot walk and because the simulator is quite slow, I constructed (using publicly available code[2]) a simpler environment on which I debugged and tested the algorithm.

I created two tasks: cartpole balance and cartpole swingup.

Balance

The agent tries to balance an inverse pendulum by applying a force to the cartpole. This task is one of the easiest tasks presented in the DDPG paper and was used for the debugging process, since it should converge very quickly.

The episode is terminated after 20 seconds or after the pendulum falls down.

Swingup

One of the harder problems solved in the DDPG paper, the pendulum starts low and should try to swing up and then balance. This task was used mainly for testing NN sizes and training constants.

The episode is terminated after 20 seconds or if the cartpole gets too far from center.

The state in both tasks has 4 dimensions, namely $d, \dot{d}, \theta, \dot{\theta}$[3] - see 5.6. The action has one dimension and it's the force applied on the cart horizontally. The used time step was $dt = 0.01$.

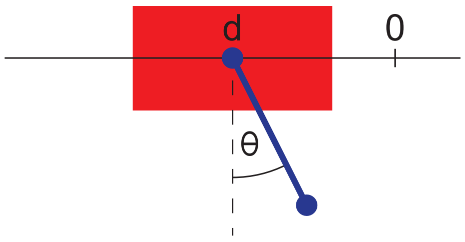

Figure 5.6: Pendulum state representation

The reward was transferred to the agent as follows:

$$r_t = dt \cdot (|\phi| - 0.1 \cdot |a_t| - 0.1 \cdot |d|)$$

The coefficients 0.1 were picked arbitrarily, their purpose was to keep the agent from outputting too large values too often (a_t) and to stop the cartpole drifting too far from the center (d). If the episode ended before the time limit, a negative reward of -0.1 was added to the terminating state.

Video 4 shows a successfully learned pendulum swingup. See A.2 for training details.

[2] `http://www.moorepants.info/blog/npendulum.html`

[3] θ was normalized between $-\pi, \pi$ to ease the reward calculation

Network size dependency

During my experimentation I noticed a correlation between sizes of critic and actor networks and the convergence ability.

When using a very small actor and a sufficiently large critic, the network was still able to learn to the extent of the actor's ability.

When using a small critic, the algorithm was unable to converge to good results. This is because the critics outputs must be accurate for the actor to learn anything. I also noticed that if the critic was too small to learn, the critic's output slowly diverged to extremely high values. This is a good indicator for checking if the critic is too small.

See 5.7 for comparisons.

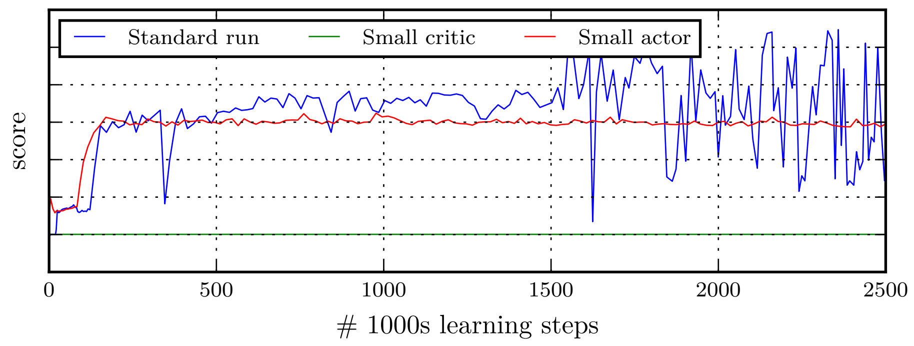

Figure 5.7: Comparison runs on the swingup task. Actor={50,50}, Critic={100,100}, Small Actor={} , Small Critic={}

5.5.2 Robot

As previously mentioned, the robot's state representation has 27 inputs (12 positions, 12 velocities, 3 accelerometers) and 12 outputs.

I did a few experiments using position control, however the exploration noise makes the learning harder since there is a big difference between adding a constant to the applied force (where it has the potential to start moving the joint at different speeds) and adding a constant to desired position (the consequences are much more drastic).

For this reason I used torque control in the remainder of the experiments.

Maintaining balance

The first task trained on the robot from scratch was maintaining a balance. While this task would be trivial in position control setting, when using torque control it becomes more challenging.

The reward was transferred to the agent as follows:

$$r_t = dt \cdot (1 - 0.1 \cdot |a_t|)$$

If the episode ended before the time limit, a negative reward of -0.1 was added to the terminating state.

The robot was able to successfully balance after ~600k episodes. See video 3, A.2 for training details.

Walking

A serious problem encountered when training the robot was the training time. While 2.5 million steps of training took about 10 hours[4], the same amount took more than 4 days on the simulator[5]. I was therefore unable to conduct a full parameter search and all the mentioned tests were done on the best supervised actor { rnn, 10}.

The reward was transferred to the agent as follows:

$$r_t = dt \cdot (0.1 \cdot (1. - \frac{1}{N}\sum_t |a_t|) + 100 \cdot \Delta x)$$

where Δx is the change in position of the robot's torso during step t.

If the episode ended before the time limit, a negative reward of -0.1 was added to the terminating state.

Unfortunately due to the mentioned time requirements, I was unable to perform enough tests (or to test long enough) to prove or disprove if the algorithm is capable of walking. The best result so far was a policy that made the robot fall forward on the end of the 15 second episode time limit. It's interesting that this was possible, considering the actor has access to only a limited time frame.

[4]on a 4-core 3.3 GHz CPU

[5]The simulator core - physics engine, unfortunately runs on a single thread and has a significant overhead. This made training on a server unfeasible - the training process did not benefit from more CPUs and the training process of 2.5 million steps would take 2 weeks on available server CPUs.

Algorithm 5 DDPG algorithm

```
Randomly initialize critic network $Q(s, a|\theta^Q)$ and actor $\mu(s|\theta^\mu)$ with weights $\theta^Q$ and $\theta^\mu$.
Initialize target network $Q'$ and $\mu'$ with weights $\theta^{Q'} \leftarrow \theta^Q$, $\theta^{\mu'} \leftarrow \theta^\mu$
Initialize replay buffer $R$
for episode = 1, M do
  Initialize a random process $\mathcal{N}$ for action exploration
  Receive initial observation state $s_1$
  for t = 1, T do
    Select action $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$ according to the current policy and exploration noise
    Execute action $a_t$ and observe reward $r_t$ and observe new state $s_{t+1}$
    Store transition $(s_t, a_t, r_t, s_{t+1})$ in $R$
    Sample a random minibatch of $N$ transitions $(s_i, a_i, r_i, s_{i+1})$ from $R$
    Set $y_i = \begin{cases} r_i + \gamma Q'(s_{i+1}, \mu(s_{i+1})) & \text{for non terminal state} \\ r_i & \text{for terminal state} \end{cases}$
    Update critic by minimizing the loss: $L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$
    Update the actor policy using the sampled policy gradient:
        $\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum_i \nabla_a Q(s, a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu} \mu(s|\theta^\mu)|_{s_i}$
    Update the target networks:
        $\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$
        $\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$
  end for
end for
```

Chapter 6

Conclusion

In this bachelor thesis, I explored several different neural networks architectures suitable for a bipedal walking task and trained and tested them in the supervised setting. I also conducted a small research of available robotics simulators and robot models.

I continued by explaining the core ideas of reinforcement learning, deep Q-Learning and deterministic policies. I further explored the Deep Deterministic Policy Gradient - a model-free general learning algorithm capable of learning different control tasks.

I combined DDPG with recurrent neural networks and prioritized experience replay and was able to successfully apply these methods on a cartpole swingup task and bipedal balancing task using the same model.

6.1 Future Work

There is still a lot of work to be done before the used reinforcement learning methods could be applied to real world applications. The deterministic actor-critic provides a good basis for continuous control reinforcement learning, while deep learning allows the approximation of complicated and even visual policies.

However, the main problem is the stability of learning. I would recommend focusing on these topics in future research:

- **Batch normalization:** recent advancement of deep learning, this technique reduces internal covariate shift by normalizing inputs of each layer [IS15]. This could improve generalizing over different tasks and also ease learning of the critic.

- **Double Q-Learning:** the Q-Learning critic approximation is often a cause of overestimating the output values. The Double Q-Learning algorithm [vHGS15] fights this by introducing another critic, used exclusively for $Q(s_{t+1}, a)$ approximation.

- **Picky replay memory:** instead of making use of different sample priorities in replay buffer, it would be best to store only the important and most rewarding states. This is a focus of ongoing research and these methods are sure to improve the stability of the learning process.

- **Better exploration policy:** The noise policy plays a crucial role in DDPG and other algorithms and the Ornstein-Uhlenbeck process is probably too naive to explore demanding multi-dimensional tasks on its own. The noise should also be able to change its size and focus over time.

As for the robot walking task, I would recommend a faster simulation framework than a full-fledged robotics simulator to allow for a proper parameter search without such time demands.

Appendix A

Training Details

A.1 Supervised Learning

Parameter	Value	Short Description
β	1e-3	actor training constant
γ	0.99	TD update constant
Critic regularization	1e-2	value is divided between critics parameters
Recurrent steps	10	size of the BPTT limit
Batch size	250	•
Training epochs	1e3	•

Table A.1: Supervised training details

A.2 DDPG

Parameter	Value	Short Description
α	1e-3	critic training constant
β	1e-4	actor training constant
γ	0.99	TD update constant
τ	1e-3	shadow networks training constant
Critic regularization	1e-2	value is divided between critics parameters
Recurrent steps	10	size of the BPTT limit
Batch size	64	•
Replay size	1e6	•
Noise constants	0.15, 0.2	θ, σ constants used in OU process
Critic learning step	1e4	critic learning starts after n steps
Actor learning step	3e4	actor learning starts after n steps
Training epochs	>1e6	•

Table A.2: DDPG training details

Appendix B

CD structure

- `/dlbw`

  Contains all the code necessary for running the learning algorithms, for usage see `README.md`

- `/software`

  Contains the V-Rep simulator and necessary python libraries.

- `/thesis`

  Contains the thesis (`thesis.pdf`) and the TEXsource files

- `/videos`

  1. nao_recorded_sequence.avi
  2. nao_supervised.avi
  3. nao_ddpg_standing.avi
  4. pendulum_swingup.mp4

Appendix C

Bibliography

[HS97] Sepp Hochreiter and Jurgen Schmidhuber. Long short-term memory. 1997.

[IS15] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[KB14] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *CoRR*, abs/1412.6980, 2014.

[LHP$^+$15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *CoRR*, abs/1509.02971, 2015.

[MKS$^+$15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb 2015. Letter.

[MLK95] M.W. Mak, Y. L. Lu, and K. W. Ku. Improved real time recurrent learning algorithms: a review and some new approaches. *Neurocomputing*, 24:13–36, 1995.

[Nog] Lucas Nogueira. Comparative analysis between gazebo and v-rep robotic simulators.

[SLH$^+$14] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. 2014.

[SQAS15] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015.

[vHGS15] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *CoRR*, abs/1509.06461, 2015.

[Wika] Wikipedia. Mean squared error.

[Wikb] Wikipedia. Ornstein–uhlenbeck process.

[Wikc] Wikipedia. Reinforcement learning.

[Wikd] Wikipedia. Supervised learning.

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

BACHELOR PROJECT ASSIGNMENT

Student:	Silvestr S t a n k o
Study programme:	Cybernetics and Robotics
Specialisation:	Robotics
Title of Bachelor Project:	Neural Networks for Humanoid Robot Control

Guidelines:

Explore available systems for physical simulations suitable for humanoid (bipedal) robots experiments and find the appropriate model. Explore various types of neural networks for the robot control. Focus mainly on recurrent neural networks. Propose and implement neural network including the training algorithm. Check the methods on selected experimental scenarios**.**

Bibliography/Sources:

[1] R. J. Williams and D. Zipser: A Learning Algorithm for Continually Running Fully Recurrent Neural Networks, 1989.
[2] D. Silver, et al.: Deterministic Policy Gradient Algorithms, Proceedings of the 31th International Conference on Machine Learning, ICML 2014, Beijing, China, 21-26 June 2014.

Bachelor Project Supervisor: Ing. Zdeněk Buk, Ph.D.

Valid until: the end of the summer semester of academic year 2016/2017

L.S.

prof. Dr. Ing. Jan Kybic	prof. Ing. Pavel Ripka, CSc.
Head of Department	**Dean**

Prague, December 18, 2015