Learning visuomotor robot control

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
PHYSICS

Author :	Frederik Hoekstra
Student ID :	-
Supervisor :	Prof.dr.ir. Tjerk Oosterkamp
2^{nd} corrector :	Dr. Nanda van der Stap

Leiden, The Netherlands, April 13, 2018

Learning visuomotor robot control

Frederik Hoekstra

Huygens-Kamerlingh Onnes Laboratory, Leiden University
P.O. Box 9500, 2300 RA Leiden, The Netherlands

April 13, 2018

Abstract

Deep reinforcement learning has solved the game of Go, along with all other board games. Can it also be applied to real-world use cases? This research combines a literature study and experimental evaluation, focusing on the case of automation for tele-operated robotics. This is necessary because tele-operation of robots is slow and cumbersome. Classical robotics solutions are expensive, and limited in precision, but deep reinforcement learning provides an opportunity for learning visuomotor skills using partial information.

Contents

Chapter 1

Introduction

The Problem

Sub-sea, a building that is on fire or at the verge of collapsing due to an earthquake, space. These are hazardous environments where remotely-operated robots are necessary. Robots have been doing manufacturing work for decades, and right now, the technology is there to make them more capable so they can be used to save lives in these strenuous and unknown environments. This requires a number of solutions which are studied under the i-botics program at TNO, where I did this internship.

The goal of the i-botics project is "...to provide the human operator with full perceptual and manipulation capabilities to intuitively perceive the remote environment and act as if being present at the remote site. " [1]

One of the problems is latency: actions of the operator arrive at the robot after some lag time T_1, and information about the environment of the robot takes another period T_2 to reach the operator. This makes complex interactions at large distances awkward and may cause instability of the system when using feedback control schemes. Automation of such interaction is non-trivial due to the variance of tasks.

The goal

The goal of this project is to investigate if and to which extent Deep Reinforcement Learning can be used to automate tasks in real-world robotic applications. What are its requirements? What are its strengths?

The state of Deep Robotic Learning

As an example of a task, let's say we want to automate turning a screw. A traditional robotics approach to this problem would be: write a screw-localizer module, then use an inverse dynamics module and trajectory planner to move the robot arm to the correct position, then use sensors to determine if the screw was grasped, and if not retry, until the screw is grasped, then turn it. All of these modules traditionally require a lot of programming and/or expensive sensors. This is called the robot control pipeline [2].

Deep Robotic Learning is the science of incorporating artificial neural networks in the robotic pipeline. Artificial neural networks have radically improved automatic image classification [3] [4] [5], and more recently revolutionized reinforcement learning [6]. The ability of a neural network to learn a generalizable function from training is exceptionally useful in robotics.

A task that has been widely studied in Deep Robotic Learning research is grasping [7] [8] [9]. Convolutional Neural Networks are very good at image classification, and can combine image input with other modalities. This has been leveraged by the robotics field, where the neural network is used as a grasp success predictor or GQ-CNN (Grasp Quality Convolutional Neural Network). These methods require an enormous amount of samples [8] [7], but the interest in grasping research has also led to the Dexnet databases with objects and optimal grasps[10][11][12], which can be used to train a GQ-CNN without needing thousands of hours of robot time (and wear and tear).

However, a GQ-CNN can only do that: predict a grasp quality based on input from a depth sensor. There are many more tasks that robots do that would benefit from automation, and just training a network from a database is relatively simple. So we looked for a different method for more general robotic automation.

Another prominent research topic in Deep Robotic Learning is imitation learning. Neural networks can be trained to accomplish all kinds of tasks if enough demonstrations are provided: neural networks are excellent function approximators. The main problem with this approach, however, is the multi-modality of demonstrations. A human might not use the exact same trajectory to demonstrate a task, but instead the distribution of trajectories (in joint and/or end effector space) may be multi-modal. In grasping a bottle, a human demonstrator may choose to approach the bottle from either the front or the side, which makes it impossible for the network to learn: a single function cannot represent both movements. A

solution to this problem is proposed in [13], where a neural network learns the entire distribution of actions for a given input. Their neural network also integrates vision input and a recurrent (memory) element, which allows it to remember what it has previously experienced during a rollout, so it can remember its choices (e.g. front or side approach). With demonstrations, the number of samples is relatively low compared to GQ-CNNs. It is hard to train a vision network with such limited samples. Their solution is to use an auto-encoder (see Section 3.2.4). This means, however, that only features that take up a significant portion of the pixels in the input, are 'sensed' by the network. Even though the sample count is relatively low for vision Deep Learning, it is still too high for an actual application: they used multi-task training on 5 tasks and gathered 15 hours of demonstration data for 5 tasks (success rates on all tasks in excess of 75%). Their performance for only 3 hours training on a single task are at best 44%. This means that 15 hours (or a comparable number) of training data is necessary to learn a robust network, and you cannot successfully train for a single task using only 3 hours of demonstrations. 15 hours of human demonstrations is not financially viable for most applications.

We will be using the approach proposed by Levine et al. [2]: to use an algorithm called Guided Policy Search to train a network to do all of the subtasks of the robotic pipeline end-to-end: from input (vision, joint states) to output (motorspeeds). An advantage of end-to-end training is that it allows the algorithm to learn 'shortcuts' or 'tricks' to accomplish a task. For example, imagine a human running to catch a ball. The classic robotic pipeline way of solving this challenge is modeling the forces on the ball to calculate the landing position, then using a planning algorithm to adjust the running pace. Humans have learned a much easier heuristic: watch the angle of the ball in the sky. If the ball looks like it is getting lower, run faster; if it looks like the ball is higher above you, slow down. End-to-end training allows robots to learn similar tactics. This same paper also contains a more versatile solution for the image training problem, using a soft-argmax function (see Section 4.1.2) to find the location of learned features in the image. The Guided Policy Search algorithm requires that the full state of the system is known at every timestep during training (for example, the exact position of certain objects that need to be moved), but can act based on limited observations at test time (such as images of the scene).

Research question and overview

The main research questions are *"Can Deep Reinforcement Learning be used to automate tasks in real-world robotic applications? To which extent? What are its requirements? What are its strengths and weaknesses?"*. To answer these questions, a literature study was conducted and experiments were performed on a subset of the algorithms. The methods of the literature study are described in Chapter 2. The methods used in the experiments are explained in Chapter 4

The questions answered with the literature study in Chapter 3 are *Which kinds of Deep Reinforcement Learning algorithms are available? What is their sample efficiency? Which algorithms have been used successfully in robotics and which functionality do they provide?*.

The experiments that were carried out answer the following questions: *Can a neural network learn a vision-based policy for moving a heavy 2-joint arm to 1 of 2 target positions by applying torques in a 2D environment that generalizes to different initial positions? How do choices in number of layers and pre-trained visual processing in the neural network architecture affect the performance of the policy network? Can a neural network learn a torque-based policy for 2D peg insertion? Can a neural network learn a policy for moving a 7-joint arm to a commanded target position in a 3D environment using joint velocity commands?*

Descriptions of the experiments can be found in Chapter 5, along with their results.

Chapter 6 contains a discussion of the results of the experiments and the literature study. The main research question is answered in Chapter 7, and an outlook is shared, indicating possible improvements on the current setup. The obvious question "Why does a physics master student investigate the applicability of Deep Reinforcement Learning?" is answered in Chapter 8.

Version of April 13, 2018– Created April 13, 2018 - 15:49

Chapter 2

Literature Methods

The questions I wanted to answer are: Which kinds of Deep Reinforcement Learning algorithms are available and what is their sample efficiency? Which algorithms have been used successfully in robotics and which functionality do they provide?

I started my search using the search terms in the left column of Table 2. I searched the Leiden Univsersity Library Catalogue and Google (Scholar). Another starting point of my search was the work of Levine's group in Berkeley, specifically [2]. Using the papers I found, I continued my search with the terms in the second column of Table 2.

Initial	**Derivative**
Deep Reinforcement Learning	Grasping
Deep Robotic Learning	Grasping Quality
Reinforcement Learning	GQ-CNN
Guided Policy Search	Q learning
	Value learning
	Reinforcement Learning
	Demonstrations

Chapter 3

Theory

Introduction

The goal of this theory section is twofold: on the one hand, it introduces the reader to Deep Learning and Reinforcement Learning. It also answers the research questions of the literature study.

3.1 Research questions

Which kinds of Deep Reinforcement Learning algorithms are available? What is their sample efficiency? Which algorithms have been used successfully in robotics and which functionality do they provide?

3.2 Introduction to Deep Learning

3.2.1 Short history

Neural networks were first invented in the 1950s in an effort to replicate the low-level functionality of the brain by mimicking a network of neurons. These were called multi-layer perceptrons, and are also called 'fully connected layers' when incorporated into a larger network architecture.

The first network that was applied to a real-world problem was MADALINE, a three-layer perceptron. It was used to filter echoes from phone signals, and implemented in hardware with vacuum tubes and memistors[14].

However, in 1969, Marvin Minsky and Seymour Papert published *Perceptrons*[15], a book in which they showed how limited perceptrons are,

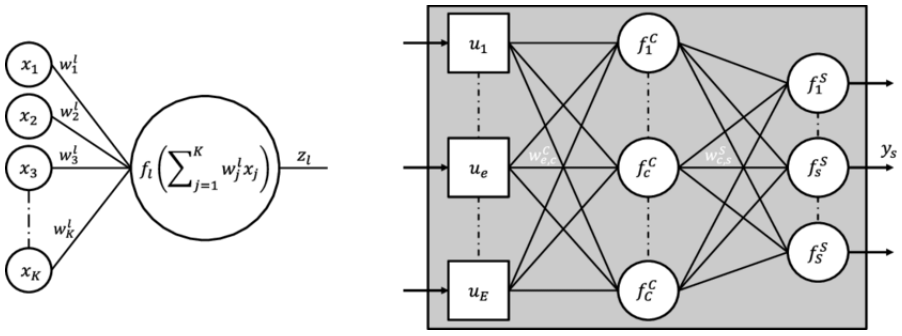

Figure 3.1: *Multi-layer perceptron network and neuron function diagrams.[18]*

by proving that a single-layer perceptron cannot learn an XOR-function. Multiple layers with non-linear activation functions can learn that, but this was not widely known, and there was no learning algorithm for such a network.

Thus the 'XOR argument' spread, and as a consequence, funding was withdrawn and the research ceased.

The second deep learning revolution happened in the 80s. A new learning rule for multi-layer perceptrons was popularized: backpropagation [16]. This allowed the error of the output neuron to be backpropagated through multiple layers of a network to update all weights accordingly. This made it possible to have a network learn the XOR-function. In a regular MLP, every input scalar would need its own unique associated weights: this is problematic when processing large images or other high-dimensional data. In 1989, Lecun et al. [17] invented a convolutional neural network: a special kind of architecture with only local connections and weight sharing which makes it very suitable for image processing. This is the building block that would later allow neural networks to process arbitrarily large images, or other locally correlated input such as speech and time series data. Throughout the 1990s and early 2000s, neural networks lost popularity, but in the last decade, deep learning has gained popularity due to the availability of big data and computing power in relatively cheap Graphical Processing Units, and a host of new architectures, some of which are explained in section 3.2.4.

3.2.2 Multi-layer perceptron

Each neuron receives n inputs from the previous layer, which are weighted according to the strength of the connection to the neuron. The weighted sum is passed through a non-linear function (the activation function) to

produce an output, which is passed along to all m neurons in the next layer [19]. The output of a single neuron is thus: $y = f(\sum_i w_i x_i + b)$

The inputs, summation and output represent the dendrites, soma and axon of biological neurons.

During training, the weights are updated according to the following update rule:

$$w_{new} = w + \eta(t - y) \cdot x \tag{3.1}$$

with x the input vector, w the weights vector, y output, t target (desired output).

Instead of using the error $(t - y)$ directly, a loss function is usually defined to map the output vector to some scalar error. For example, a regression task would typically use a mean square error (MSE) loss function.

3.2.3 Convolutional Neural Networks (CNNs)

For this explanation, I will assume that the input to a convolutional layer is an RGB image: this could of course also be a higher or lower dimensional input. Convolutional layers work by sharing the same weights all over the image, making them translation-invariant. The weights are stored in filters, or local receptive fields[19]. A filter has a dimension of $f * f * d$, where d is the depth of the input image (3 for an RGB image), and f is the filter size, a number usually much smaller than the size of the input image. Thus, the convolution is usually only applied in the spatial direction.

$$c_{m,n} = \sum_{k,l=-\widetilde{F},-\widetilde{F}}^{k,l=\widetilde{F},\widetilde{F}} w_{k,l} x_{m+k,n+l} \tag{3.2}$$

The output at the m,n position is the product of the filter with the input pixels in a small (filter size) region around the m,n position in the input.

Every filter in the layer convolves separately with the input to create a different (grayscale) feature map. These feature maps together form the input image to the next layer: the depth is now the number of filters in the previous (input) layer.

Thus, an $i * i * d$ input, convolved with an $f * f * d$ filter produces an output that is $i - f + 1$ in both height and width. Of course, one could also choose to not calculate the output for every possible position, and instead move the filter by s steps for computing the next output value. This is called stride and will create a smaller output image. The output image can also be kept the same size by padding around the edges of the input with zeros. [20] is a guide containing all possible combinations of padding

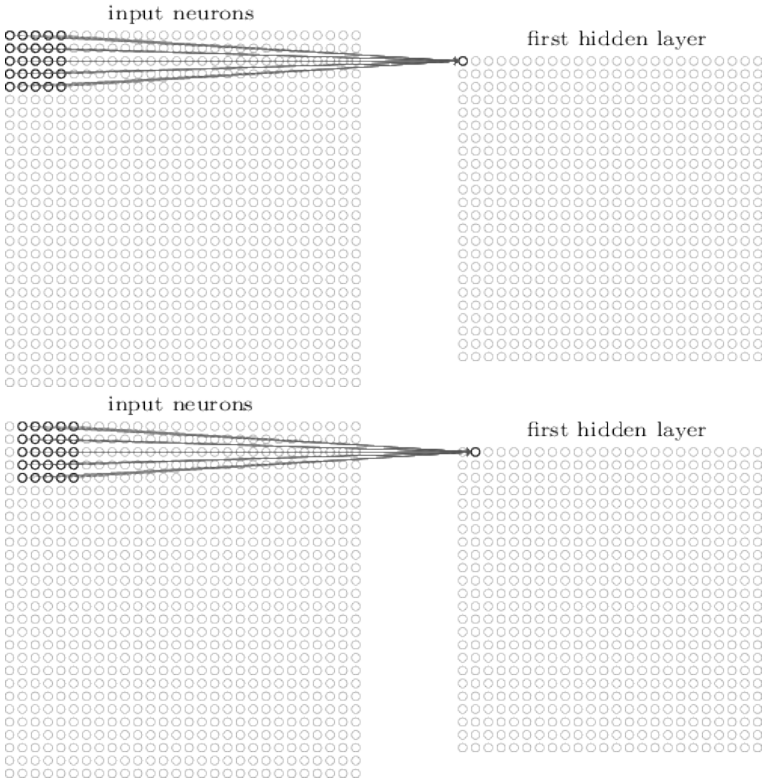

Figure 3.2: *Convolutional networks convolve input with a filter, then output a feature map which indicates where this feature is present[19]. The output at a point is calculated as the product of the filter with the input pixels around the same position.*

and strides, including the arithmetic for transposed convolution (a similar inverse-like operation which creates a larger output image).

Max-pooling is often used in-between convolutional layers to shrink the height and width of the feature maps and to reduce the number of parameters. Max-pooling of size n is an operation with an output half the height and width of the input. This is done by dividing the input into $n * n$ blocks; then for each block, the maximum value is set as output. This is done separately for each feature map, so depth is preserved. Thus, spatial information is lost, while depth (feature information) is preserved: this is desired for classifier networks, but not for robot control, where position information is more important, and less feature information is necessary than for classification. Section 4.1.2 contains information about convolutional neural network architectures used for robot control.

3.2.4 Unsupervised Learning

In the previous examples, training a neural network was done by presenting it with training examples consisting of input and the corresponding desired output. This is called *supervised learning*. However, most real data is unlabeled: when there is no label, we cannot do supervised learning.

There are a few things that one can do with unsupervised learning in combination with neural networks: learning a low-dimensional encoding of high-dimensional data such as images [22], training a generator for synthesizing real-looking data [23], using unlabeled data to aid in training a classifier [24].

For dimensionality reduction, an autoencoder is typically used: a convolutional network can serve as an encoder. It transforms the input data into a latent vector of reduced dimension. This latent vector is then fed into the decoder: a deconvolutional network, essentially the inverse of a convolutional network that tries to reconstruct the image. (the values of the weights are not forced to be related to those of the encoder though). The error is then the deviation from the input image. Due to the shape of the network diagram, an autoencoder is sometimes called a diabolo network [25].

The other popular deep unsupervised learning technique is called Generative Adversarial Networks (GANs) [23]. A GAN also typically consists of a convolutional and a deconvolutional network. In this context, the deconvolutional network is the *generator*: it generates high dimensional data using a random latent vector as input. The convolutional network is the *discriminator*: it tries to judge whether the presented input is from the real

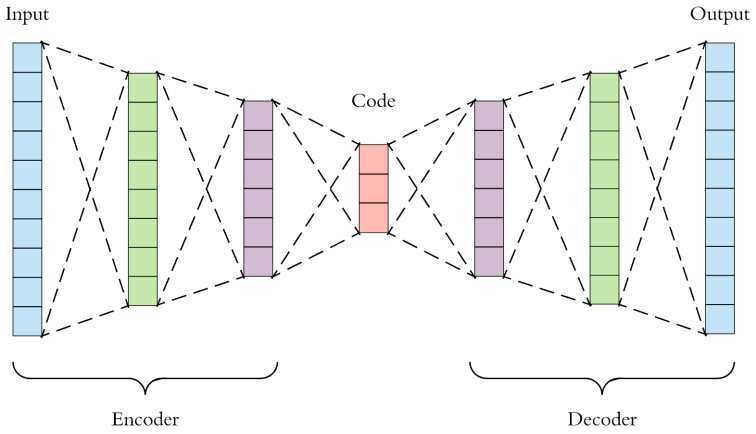

Figure 3.3: *Autoencoders encode and decode an input, and aim to minimize the reconstrution loss[21].*

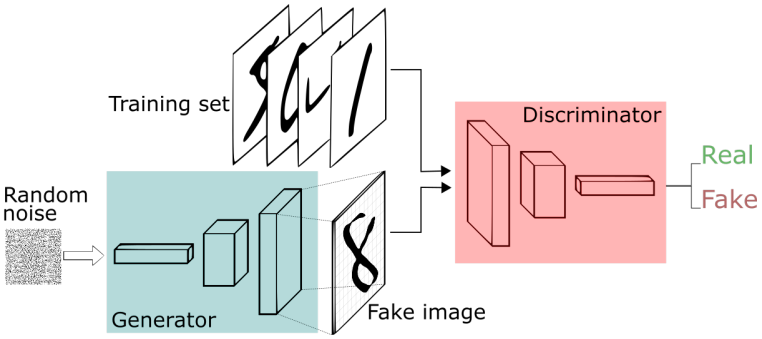

Figure 3.4: *A Generative Adversarial Network is built up from a generator, which generates images from noise, and a discriminator which tries to discriminate between real (training) data and generated images. [26] The generator is optimized to fool the discriminator.*

Figure 3.5: *Fake celebrity images generated by a GAN. This GAN was trained by gradually adding layers and scaling up the resolution during training [27].*

data or from the generator. In a GAN, the generator is rewarded for fooling the discriminator: thus it learns to generate output that the discriminator cannot distinguish from real.

This technique can be used to generate images and other high dimensional data, based on an unlabeled dataset. Notable examples include Image-to-Image translation [28] [29] and speech synthesis [30], a well as the celebrity face generator [27] (see Figure 3.5).

Both auto-encoders and GANs can be used to aid training of a classifier when using a big unlabeled and a small labeled dataset [24]. Learning an encoder is a similar task to learning a classifier: there is commonality in learning common patterns in the input data and encoding the differences between samples: this aids both reconstruction and classification. A discriminator is also similar to a classifier: the classes is separates are simply 'all the real classes' and the 'fake class'.

3.2.5 Overfitting

Neural networks typically have millions, or tens of millions of parameters. Thus, overfitting is a major problem, especially with larger networks and/or smaller datasets. There are multiple techniques which can be used to prevent overfitting.

Data augmentation is one way to prevent overfitting [31]: adding noise to data, rotating and distorting images, subtly changing color. Another is

using batch normalization between convolutional layers (see section 4.1.2). Other ways to prevent overfitting address the dependency of the network on a limited number of weights. Dropout [32] is one such method, where a fraction (usually 0.5) of the connections in the network are removed during training. Finally, a penalty can also be set on having some very strong connections by adding a regularization term to the loss function of the form $\sum_i w_i^2$ where w_i are the weights of the network.

3.3 Reinforcement Learning

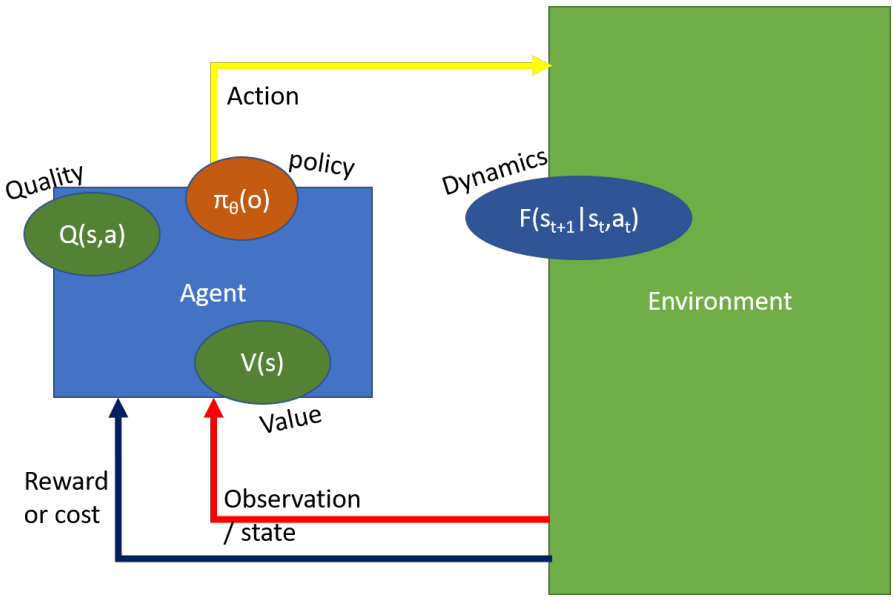

Figure 3.6: *The setup of Reinforcement Learning. An agent interacts with the environment, sending actions and receiving observations and rewards. The ovals are functions: the policy, which selectst an action based on an observation, the quality assigns a score (expected future reward) to an action in a certain state, and the value function specifies the expected future reward for the state. The dynamics, which determine the likelihood that a state will follow from a previous state and action, can be modeled by the agent to perform planning.*

Reinforcement Learning is a class of problems which revolves around an agent choosing actions to maximize the sum of received rewards from the environment: in short, sequential decision-making. The basic model in the theory of reinforcement learning is the Markov Decision Process (MDP): the probability to transition from state s_t to s'_{t+1} is only dependent on the current state s_t and the action a_t, and not on previous states. When

Version of April 13, 2018– Created April 13, 2018 - 15:49

the observation is not the full state, it is called a Partially Observed Markov Decision Process (POMDP).

At every timestep t the agent receives an observation o_t and reward r_t from the environment, and processes those to take an action a_t. The *objective* to maximize is the sum over all rewards $\sum_t(r_t)$. Sometimes the word cost function is used instead of reward, they are the same thing, but with opposite sign.

The key concept of reinforcement learning is sequential decision-making. Although I will mainly talk about Deep Reinforcement Learning in this section, reinforcement learning is a field that is much older than deep neural networks.

However, there are some functions in reinforcement learning that can be approximated by neural networks. Instead of listing a number of algorithms, I would like to focus on the relevant functions and explain how they are used in recent papers in Deep Reinforcement Learning, namely: the policy in section 3.3.1), the Q-function in section 3.3.2, the model in section 3.3.3, and the value function in section 3.3.4. After that, I will look at 2 specific solutions to the sample-efficiency problem in robotic deep learning: learning from human demonstrations in section 3.4 and meta-learning (from simulations) in section 3.5.

Note that reinforcement learning is a very old research area, and all of these concepts have been around since the 1950s [33]. The only thing that is new is the use of deep neural networks in this context.

3.3.1 Policy

$$\pi_\theta(a_t|o_t) \text{ or } \pi_\theta(a_t|s_t) \tag{3.3}$$

The policy π is the distribution over actions a_t, given an observation o_t (or the full state s_t). The policy is defined by the parameters θ. This is the function that represents what the agent is likely to do given an observation.

A variation on the above definition that is often used is to implement a recurrent network as a policy: a network that receives not only the input at each timestep, but also has a memory which can be written to and read from. The disadvantage is that a recurrent neural network is effectively an extremely deep network and usually requires more samples.

The simplest reinforcement learning algorithms are policy gradient methods. They sample from the (noisy) policy and then directly update the policy parameters θ by gradient descent on the objective. The first such method was REINFORCE [34].

Version of April 13, 2018– Created April 13, 2018 - 15:49

A new, very sample-efficient policy search method is called Guided Policy Search [35] [2] [36] [37] [38]. This method uses optimization in action space of very simple local linear-Gaussian controllers. These local controllers consist of a linear feedback and feedforward term (plus Gaussian noise) at every timestep and can thus only learn a fixed sequence from point A to point B. However, for Guided Policy Search, the network tries to learn a policy function that behaves similarly to these trajectories, and then the local controllers adapt to and improve upon the network's behaviour. A fit of the dynamics can be used for faster optimization of the local controllers. This is the most sample-efficient end-to-end learning method available for deep reinforcement learning. For more information on Guided Policy Search, see section 4.1.

3.3.2 Q-function

$$Q(s_t, a_t) \tag{3.4}$$

The Q-function assigns a 'quality' to state-action pairs and can be used to decide which action is most likely to eventually lead to a high cumulative reward.

The basic Q-learning update rule is:

$$Q(s_t, a_t) = (1 - \alpha)Q_{old}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot max_a Q(s_{t+1}, a)) \tag{3.5}$$

Here, the learning rate (step size) is α, s_t, a_t, r_t are state, action and reward respectively, at timestep t, and γ is the discount factor. This is a factor by which future rewards are discounted: this is the same factor that is used in finance when calculating future expected profits. Future rewards are uncertain and thus are worth less than the same reward now.

This update rule, however, assumes that the next action will be the one with the highest Q-value. However, a policy that would always do that would never explore other actions. This is at the heart of the *exploration-exploitation* problem: a balance needs to be struck between exploiting what the agent knows will work and exploring new actions. Other words used in this context are *variance* for an algorithm with a lot of exploration and *bias* for an algorithm that does more exploitation. Q-learning is usually done with an epsilon-greedy policy: take the optimal action with probability $(1 - \epsilon)$, take a random action with probability ϵ.

If we change the update rule by taking into account the actual next action that the agent has taken, instead of assuming it will always take the optimal action, it will lead to:

$$Q(s_t, a_t) = (1 - \alpha)Q_{old}(s_t, a_t) + \alpha \cdot (r_t + \gamma \cdot Q(s_{t+1}, a_{t+1})) \tag{3.6}$$

Version of April 13, 2018– Created April 13, 2018 - 15:49

This update rule is called SARSA [39], because it takes those 5 parameters as input: $S_t, A_t, R_t, S_{t+1}, A_{t+1}$.

A recent Deep Learning example where the Q-function was represented by a network can be found in [40], where a neural network learns to play Atari games. The input is the state of the pixels, and the outputs are the Q-values for all the possible actions. This network achieved superhuman performance on Breakout, Enduro and Pong. But not on Q*bert, Seaquest and Space Invaders.

In Robotic Deep Learning, the classifying power of convolutional neural networks is used to classify potential grasps on depth image data [8] [10] [11] [12]. This is called a grasping quality convolutional neural network (GQ-CNN). In an application, this network is presented with several hundred grasp candidates, it assigns a grasp quality to each of them, then the robot executes the grasp with the highest quality. This is an example of a Q-function for a discrete action space of grasp candidates.

3.3.3 Model (transition function and reward function)

$$\text{dynamics/transition: } p(s_{t+1}|s_t, a_t) \text{ or } s_{t+1} = f(s_t, a_t) \tag{3.7}$$

$$\text{reward: } r_{t+1} = g(s_t, a_t) \tag{3.8}$$

A model approximates the environment: based on the current state and action, it returns (the probability distribution over) the next state, and sometimes the reward function is also modeled.

The advantage of fitting a model is increased sample efficiency: less samples are needed if you use the gathered information to infer a model. The disadvantage is that the performance of the learned policy is limited by the accuracy of the model. This is very important in Robotic Learning, as taking samples with an actual robot is time-consuming, and learning in simulation usually does not transfer easily to real environments [41]. When tens of thousands of samples can be taken using simulation, model-free is the way to go, as it allows for more complex tasks to be learned [37].

3.3.4 Value function

The value function represents the expected cumulative reward from a given state, using the policy and an estimate of the dynamics. Given the Q-function, we can say that:

$$V(s_t) = max_a(Q(s_t, a_t) \tag{3.9}$$

Version of April 13, 2018– Created April 13, 2018 - 15:49

The foundation of Value learning is the Value Iteration algorithm as developed by Bellman[33]. This algorithm requires a transition function $p(s_{t+1}|s_t, a_t)$ and a reward function $r(s_t, a_t, s_{t+1}$ and is thus model-based. The algorithm is described in pseudocode in Algorithm 1.

Algorithm 1 The Value Iteration algorithm

Require: S set of all states; A set of all actions; λ threshold
Require: $P(s'|s, a)$ transition function and $R(s, a, s')$ reward function
Assign $V_0(S)$ arbitrarily
$k \leftarrow 0$
repeat
  $k \leftarrow k + 1$
  for all states $s \in S$ **do**
    $V_k(s) = max_a \sum_{s'} P(s'|s, a)(R(s, a, s') + \gamma V_{k-1}(s'))$
  end for
until $|V_k(s) - V_{k-1}(s)| < \lambda \forall s$
return V_k

Tamar et al.[42] developed a neural network implementation of this Value Iteration algorithm to learn to plan (see figure 3.8). To do this, they implemented a neural network architecture that includes the reward and transition function (see Section 3.3.3), as well as an iteration module that refines the value function, and a policy. They applied this algorithm to a grid-world, a crater landscape, web search and a simple continuous control obstacle avoidance task. Note however, that this method assumes *fully observable* MDPs, and does not cover the case of partial observability.

Another common family of methods that uses a neural network to learn a value function is (advantage) actor-critic based methods. This entails separately learning a value network (the critic) and an advantage network (the actor). Advantage is defined as $A(s, a) = Q(s, a) - V(s)$. By optimising for advantage instead of reward (like REINFORCE, see 3.3.1), the actor can learn optimal actions in bad situations (low value states): from a state with very low value, a reward that is not very bad is actually a very good move. REINFORCE cannot learn this, since it has no information about the (estimated) value of a state. Another advantage of this method is that it allows the critic to learn from mini-batches: value iteration can only learn after it has reached the final state. The critic can learn from small mini-batches using its current estimate of the value function. This introduces some bias into the system: less exploration, more sample efficiency. Combining this with asynchronous updates from multiple

Version of April 13, 2018– Created April 13, 2018 - 15:49

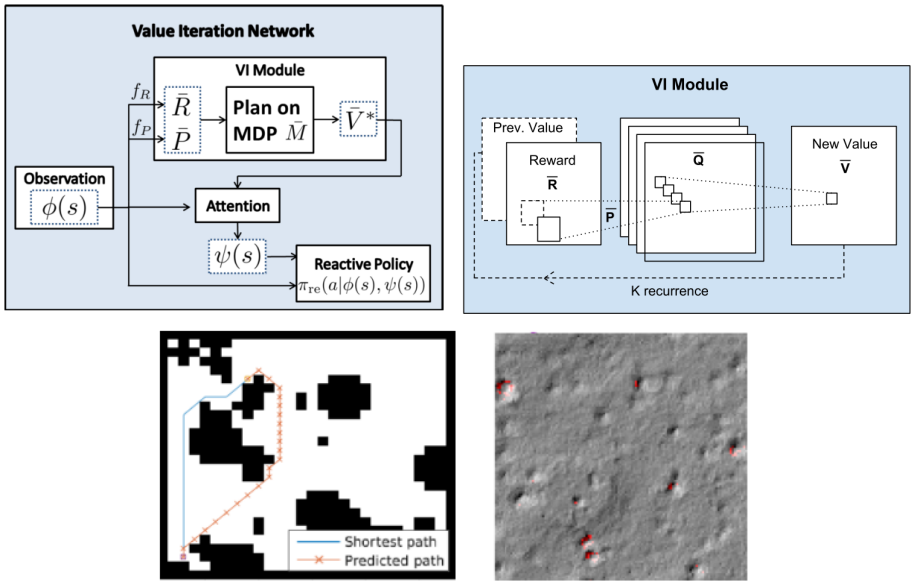

Figure 3.7: *The Value Iteration Network. All functions are implemented as neural networks: reward f_R, dynamics f_P, Q-function, Value function and policy. Top left: the full network. Top right: The Value Iteration module. The iterative nature of this algorithm is implemented using a recurrent neural network. Bottom: the environments that this method was applied to: grid world and Mars. Note that this method assumes full observability.*

agents introduces enough variance to counter the inherent bias of actor-critic methods [43]. Sample efficiency is thus better than most value and Q-learning methods, but still orders of magnitude behind Guided Policy Search.

The difference between the definitions of the Q and Value functions may seem small and trivial. The difference between Q-learning and Value-learning algorithms, however, is not small and trivial: Q-learning algorithms work by trial and error and are *model-free*. Value learning algorithms are generally *model-based*.

3.4 Learning from demonstrations

There is a number of ways in which demonstrations can help a neural network policy learn a robotic task. The simplest is to train the neural network through regression on the dataset (observation, action) that was gathered through demonstrations. This is done in [13], using a recurrent neural network. A recurrent neural network has a memory: it receives

Version of April 13, 2018– Created April 13, 2018 - 15:49

an additional input that is dependent on the state of the network in the previous timestep. They used a one-hot vector to do multi-task training: a vector of zeros with a 1 in a single dimension to indicate which task is being performed. See section 1 for more about this paper.

A standard improvement on this method for reactive (non-memory) policies is DAgger [44]. This involves doing a roll-out of the policy, then annotating the new inputs that are received with appropriate outputs. This is useful, since a trained reactive policy may not exactly follow the demonstrated trajectory, and instead encounter inputs that are outside the distribution of demonstrations.

Another way in which demonstrations can help is by providing a starting point for Guided Policy Search. This is especially useful when using a model-free optimizer [37]. For more information on Guided Policy Search, see section 4.1.

The most sample-efficient way of doing demonstration learning is Guided Cost Learning [45]. This Inverse Optimal Control method uses Guided Policy Search, and optimizes a cost function that is represented by a neural network at each iteration. In this way, it adapts the cost function so that it not only encodes the right information for a successful policy, but is also easily learnable at each iteration of policy optimization.

3.5 Meta-learning and transfer learning

Gathering a lot of samples is easy in simulation. This is an important reason why deep reinforcement learning is so successful in games, but not so much in robotics: the real world is different from simulations in ways that are important for robotic task execution. There is a number of ways to improve the transfer of simulation learning to the real world.

The CAD2RL algorithm [46] uses a number of simulation environments with different colours and rendering settings. A policy is trained for collision avoidance of a drone in these very different looking environments. The trained policy can then be deployed to a real drone, since the real world looks like 'just another simulation'. The diversity in the synthetic environments is apparently good enough so the policy can generalize to the real world.

This idea of training in multiple environments to generalize to a new one is also present in meta-learning or 'one-shot learning'. The goal in these fields is to train a network that can very easily adapt to a new task (or environment). The most successful algorithms are Model-Agnostic Meta-Learning (MAML) [47] and its derivative Reptile [48]. The basic idea is that

Version of April 13, 2018– Created April 13, 2018 - 15:49

instead of searching for the parameters θ that deliver optimal average task performance, we are looking for the parameters $\widetilde{\theta}$ that deliver optimal task performance after 1 gradient step update for that specific task. As such, it finds a point in parameter space from where it can quickly learn all of the trained tasks with just 1 (or a few) steps. This allows it to generalize to new tasks.

As such, MAML could be used to train a policy network on a lot of different simulation environments, which could then quickly learn a similar policy for a real robot.

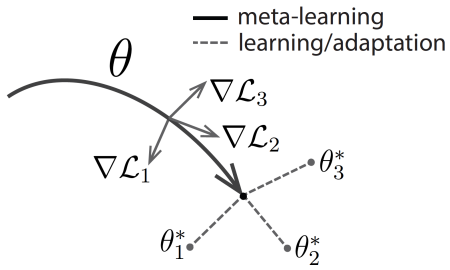

Figure 3.8: *The MAML [47] algorithm optimizes for performance after 1 gradient step. As such, it finds a point in parameter space from where it can quickly learn all of the trained tasks with just 1 (or a few) steps. This allows it to generalize to new tasks. Figure from [47].*

3.6 Conclusion

Learning from demonstrations is possible, using an LSTM that learns the entire distribution of human actions, however this requires 15 hours of human demonstrations and has not been shown to work with a sophisticated vision architecture, only with an autoencoder (which can only learn to see large objects)[13]. Model-free Deep Reinforcement Learning algorithms perform best, but they need lots of samples and are thus only suitable for use in simulated environments.

The easiest robotics application of Deep Learning is Grasping Quality Convolutional Neural Networks (GQ-CNNs), which rate grasp candidates based on RGB-D image data. Training can be performed using synthetic data, thus sample-efficiency is not an issue. Implementation of such a method on a grasping robot arm would be straight-forward.

The sample problem can be solved by using meta-learning, allowing pre-training of a policy network that can then easily adapt to new tasks or environments. However, this has only been shown to work with very

Version of April 13, 2018– Created April 13, 2018 - 15:49

similar and/or simple tasks. Another option is pre-training in simulation: this works best with meta-learning, since the simulation is likely different form reality: training a network to adapt easily to different simulations could allow it to work well in the real world too.

The only sample-efficient end-to-end training method which has been shown to work in a complex (7 joints) continuous action space environment with a sophisticated vision network is Guided Policy Search[35]. This method requires the full state (such as positions of relevant objects) to be known at training time to train a reactive policy network that can act based on limited information. The Inverse Optimal Control method Guided Cost Learning is also based on Guided Policy Search, and other expansions exist for tasks that require memory and for using camera images instead of position trackers (see sections 4.1.2 and **??**) that make it an attractive framework for Deep Robotic Learning.

Version of April 13, 2018– Created April 13, 2018 - 15:49

Chapter 4

Experimental Methods

This chapter consists of an explanation of the Guided Policy Search algorithm and some of its extensions in Section 4.1, and an exposition of the methods and materials in Section **??**, including software and hardware.

4.1 Guided Policy Search

Guided Policy Search is a family of algorithms that use Linear Gaussian state-feedback controllers (which are in this context referred to as *trajectories*) to guide a neural network in learning an optimal control policy.

Overview

There are different local controllers for different *conditions*: different conditions are typically defined by different initial and/or target positions. Thus each controller learns a sequence of actions and feedback terms to reach a different target state.

A single neural network is then trained to imitate these local controllers: for each timestep, the neural network input and the local controller's output have been stored, and the network is trained through regression to return an output similar the actions that the controllers.

So far, this is just imitation learning: however, the network may not be able to learn the same policy as the trajectories: the local controllers are no function approximators and are not dependent on the input, they only have a mapping from a timestep to a fixed command. Thus the network is rolled out: samples are taken of the network controlling the robot for each condition, and the trajectory optimization is then constrained to

this roll-out by limiting the KL-divergence between the trajectories and the network's global policy.

If no network is used, the trajectories are constrained to their previous roll-out.

The parameters of a trajectory include a feedback term and a forward action term at each timestep.

The parameters of the trajectory can be optimized in a variety of ways, with model-free updates, model-based updates, or combining both kinds of updates. [38] This is discussed in Section 4.1.1

4.1.1 Trajectory Optimization

The standard trajectory optimizer is the iterative Linear Quadratic Regulator[49]. This method is based on a linear fit of the local dynamics of the system, and a quadratic expansion of the cost function. Then, the cost is minimized, constrained to a certain KL-divergence, since the first and second-order approximations are only valid close to the trajectory. For details, see [35]. Another optimizer is the path integral-based PI2 optimizer [37], which is stochastic. The lack of a dynamics fit means this optimizer is slower, but can learn more complex actions because it is not constrained by any assumptions about the dynamics of the system being linear and smooth. This can be useful for tasks like door opening.

A third option is to combine the model-based LQR update with the PI2 model-free update, which is done in the PILQR optimizer [38].

4.1.2 Neural network architecture

The neural network architecture used is a fairly straight-forward convolutional neural network that receives the image as input. Its output is then concatenated with robot state information (joint angles and speeds), and passed through 2 fully connected layers with 40 hidden units. The output layer consists of 7 nodes, the number of joints of the robot arm. All activations are ReLu (rectified linear unit $f(x) = x$ if $x > 0$, else 0) except for the output layer, which is linear.

Batch normalization

The convolutional layers also use batch normalization [50] after each activation function. This is a layer that learns the distribution of the activations and normalizes each batch, thus preventing the distribution to change radically. This is especially important for the deeper layers: if the

input to those layers changes drastically due to weight updates of earlier layers, their own weights will have to be adjusted to handle this different input. It also reduces sensitivity to initialization.

Soft-argmax

Most convolutional neural networks transform an input image with large spatial dimensions and low depth to a feature vector where all spatial information is lost and only feature information is left.

However, for control tasks, it is important to keep the spatial information. It is still desirable to reduce the size of the output of the convolutional layer, to reduce the number of weights in the fully connected layers. This is why the feature maps of the last convolutional layer are passed through a soft-argmax layer to extract the positions of the highest activation for each feature. First, a spatial softmax is applied to the input a_{cij}, then the expected position (f_{cx}, f_{cy}) is calculated as the mean of this softmax distribution:

$$\begin{aligned} s_{cij} &= e^{a_{cij}} / \Sigma_{i'j'} e^{a_{ci'j'}} \\ f_{cx} &= \Sigma_{ij} s_{cij} x_{ij} \\ f_{cy} &= \Sigma_{ij} s_{cij} y_{ij} \end{aligned} \tag{4.1}$$

Supervised and unsupervised pre-training

In order for the policy network to learn a controller, the vision layers must be pre-trained to learn the features of relevant objects. There are 2 ways to do this: supervised or unsupervised.

For supervised pre-training, a dataset of images with position labels is used: position labels can be positions of parts of the robot and/or target marker/object. Add 1 fully connected layer behind the soft-argmax for rescaling.[2]

For unsupervised pre-training, no labels are used. Instead the network is set up like an autoencoder, where the target output is a down-sized grayscale version of the input image. The network learns to 'reconstruct' this image by applying 1 fully connected layer to the output of the soft-argmax layer. In order to make sure that the network learns the positions that are useful for predicting the dynamics, a smoothness penalty is added to the loss function: $g_{\text{slow}}(\mathbf{f}_t)$, the L2-norm of the second time derivative of the feature points, to disencourage learning features whose position changes in a non-smooth (noisy) way.

Version of April 13, 2018– Created April 13, 2018 - 15:49

$$g_{\text{slow}}(\mathbf{f}_t) = ||(\mathbf{f}_{t+1} - \mathbf{f}_t) - (\mathbf{f}_t - \mathbf{f}_{t-1})||^2$$
$$L = \sum_{t,k} ||I_{\text{downsamp},k,t} - h_{\text{out}}(f_{k,t})||^2 + g_{\text{slow}}(f_{k,t}) \tag{4.2}$$

The unlabeled dataset can be generated simply by training a trajectory on a task with constant target and initial position. The trained feature extractor can then also be used to translate images into feature coordinates, which can be used as state variables for cost functions and trajectories.[36]

In other words, it allows trajectory training on tasks that require visual position information. An image can then be used to define a target state.

4.1.3 Memory States in Guided Policy Search

Guided Policy Search has mostly been used to train a reactive policy: a function from observation to action, independent from previous timesteps. Some tasks however, require memory. These are often trained using a recurrent network: a network with layers that receive inputs from the state of the network at the previous timestep. However, this effectively makes the network as deep as the number of timesteps and makes a lot more difficult to train than a reactive policy.

In [51], an extension of Guided Policy Search is proposed and proven to work using memory as part of the state. In this work, a number of state (and observation) dimensions have been added, with numbers that can be directly modified by the policy using additional action dimensions. By separating the memory from the policy, the policy is effectively still reactive. The only difference is that it now also learns to write to and read from the memory.

4.2 Experimental methods and materials

The experiments were done by applying variations of the Guided Policy Search [52] algorithm to different experiments in 2D (using Box2D [53]) and 3D (using Gazebo [54]) simulation environments.

Experiments were designed to test whether the system could learn policies that need to estimate:

- (local) inverse dynamics for 2D
- (local) inverse kinematics for 3D

- strategies for 2D peg insertion (non-linear dynamics)

Experiments consist of a number of conditions, which typically have different initial and/or target positions. Each sample is 5 seconds, each timestep 0.05 seconds for a total of 100 timesteps per sample.

For every experiment, the MDGPS algorithm[55] is used, the network receives the joint angles, joint velocities and end effector position as input. Comparisons are made between the performance of the network as a function of additional input. In these comparisons, **tgt** stands for direct target position input; **img** means image input from which the target position can be inferred; **blind** networks have no target information, only pose information.

The 2D environment was used to compare the performance of different network architectures.

For all vision experiments, the convolutional layers of the network are pre-trained on pose and target position estimation from images. All network optimization uses the Adam [56] optimizer.

The software I used consists of the Guided Policy Search repository [52], which is implemented in Python 2.7 [57] and uses the Tensorflow [58] library for deep learning. This was run on Ubuntu 16.04 LTS.

Among the modifications I made to the Guided Policy Search repository are: image input for Box2D; efficient storage of image data in sample objects; an upgrade from tensorflow 0.8 to tensorflow 1.3, which adds new functionality and compatibility; extensive neural network architecture options through *hyperparams.py*; improvements to interfacing with Gazebo.

m

Hardware used:	CPU	Intel(R) Core(TM) i7-6700K CPU @ 4.00GHz
	Motherboard	ASUS MAXIMUS VIII RANGER
	RAM	2x8GB 2133MHz DIMM
	GPU	Nvidia GeForce GTX 970 4GB

The specific setup, research question and conclusion per experiment can

be found in Chapter 5.

Version of April 13, 2018– Created April 13, 2018 - 15:49

Chapter 5

Experiments

In this chapter, the performed experiments are described for the 2D and 3D simulation envrionments. Sections 4.1-4.4 are built up as follows: each section starts with the respective research questions, then the hypothesis is stated, followed by the specific methods used in the experiment. The results are then presented, and the conclusion answers the research question.

5.1 Box2D inverse dynamics

Research Questions

Can a neural network learn a vision-based policy for moving a heavy 2-joint arm to 1 of 2 target positions by applying torques in a 2D environment that generalizes to different initial positions? How does a vision-based network compare to a blind network, and a policy network that receives the exact target position directly?

Hypothesis

By pre-training a vision network on estimating the position of the target in images, the vision-based network will surpass the blind network, which does not have any information to indicate which of the 2 target positions it should move to. It is expected that the policy network with direct target position input will perform better than the policy network that receives an approximate target position from the vision network.

Methods

We use the Box2D [53] environment with a heavy 2-joint arm and weak motors. The target is marked by a blue star figure at a distance (x, y) of $(10, 5)$ or $(10, -5)$ from the base of the arm. Both arm segments are length 10.

The vision network has 3 convolutional layers followed by a soft-argmax layer and then 3 fully connected layers and is pre-trained on images of random target and arm positions to estimate the target position. The network's square filters have sizes 7,5,5, the first conv layer has a stride of 2, and the activations are all ReLu, excrept for the last layer which is linear.

The policy network that receives the joint state and end effector position, along with the optional target input, consists of 2 fully connected hidden layers of 20 units with ReLu activations, the final layer has a linear activation function.

Initial positions of the robot arm are shown in figure 5.1, figure 5.2 and in table 5.1 as defined by the joint angles in radians: angle 1 is 0 if the first segment of the arm points straight up, and the positive direction is anti-clockwise. angle 2 is 0 if the second segment is aligned with the first, also anti-clockwise.

The trajectories (Linear Gaussian Policies) for each training condition are pre-trained with 5 iterations of LQR, taking 3 samples per condition per iteration. Guided Policy Search training was done using the Mirror-Descent Guided Policy Search algorithm for 4 iterations with 3 samples per condition per iteration.

The cost function used consists of 3 terms: a small cost on the square magnitude of the action vector to punish unnecessary and high forces, an L1L2 norm on the end effector distance of the following form: $L = 0.5 l_2 d^2 + l_1 \sqrt{\alpha + d^2}$ whose weight increases quadratically with the time since the start of the trial, and a binary cost term for the end effector distance at the last timestep.

During training, the policy and trajectory controllers are trained on both target positions for every initial position. For testing, 2 of the initial positions are changed and testing is again done for both target positions. This means training is done on a total of 6 conditions, testing on 4.

Version of April 13, 2018– Created April 13, 2018 - 15:49

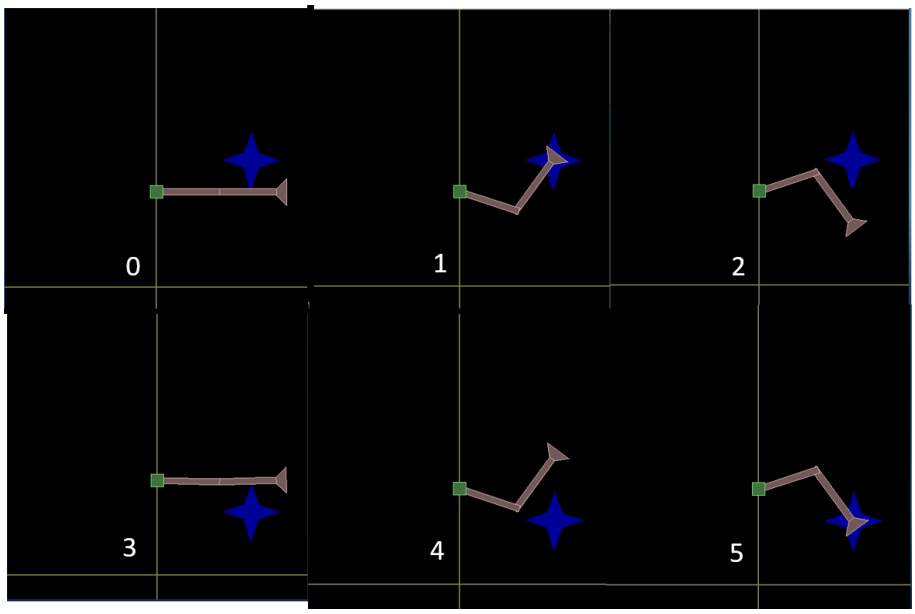

Figure 5.1: *Train conditions for the 2-joint arm Box2D environment. The arm is shown in the initial pose, the target position is marked by the blue star and the condition number (index) is shown in white. The yellow lines intersect at (0,0), the green base of the arm is at (0,15).*

Condition number for (high, low) target positions	**Training** joint angles (a1, a2) in π radians	**Test**
0, 3	-0.5, 0	
1, 4	-0.6, 0.4	-0.6, 0
2, 5	-0.4, -0.4	-0.4, 0

Version of April 13, 2018– Created April 13, 2018 - 15:49

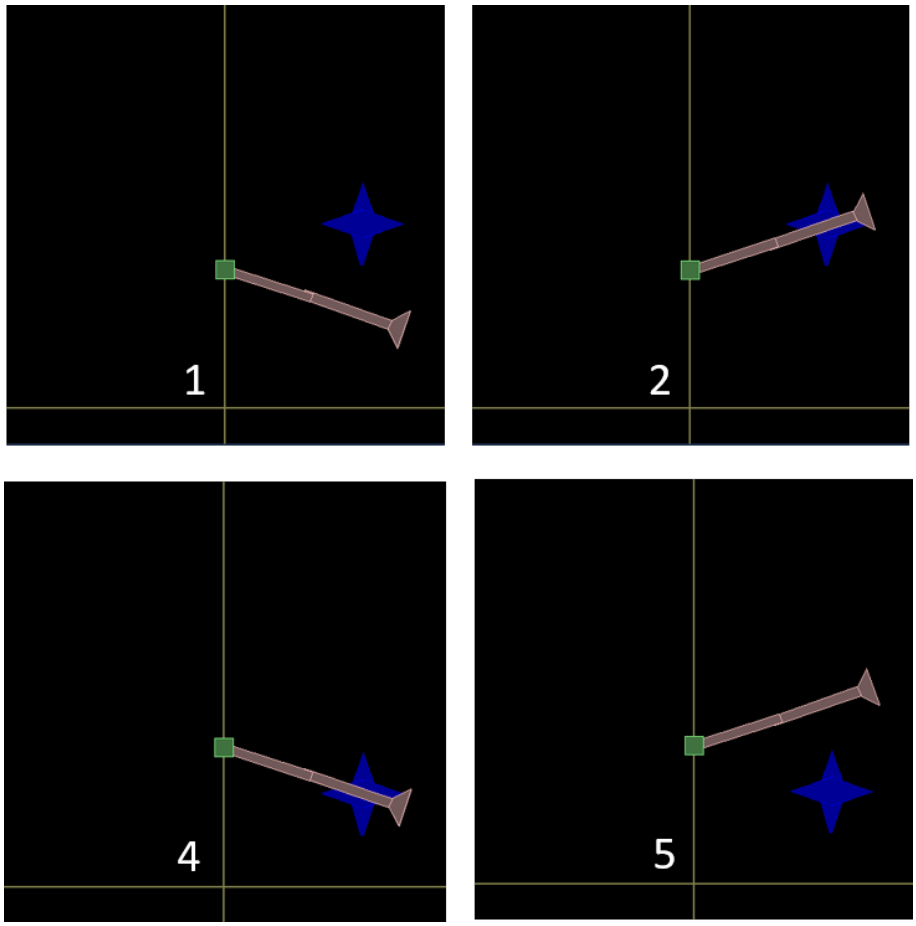

Figure 5.2: *Test conditions for the 2-joint arm Box2D environment. The arm is shown in the initial pose, the target position is marked by the blue star.*

Version of April 13, 2018– Created April 13, 2018 - 15:49

Results

Because the value of the cost function for an optimal policy is ill-defined, we decided to evaluate the performance of the policy in terms of the end effector distance from the target at the final timestep.

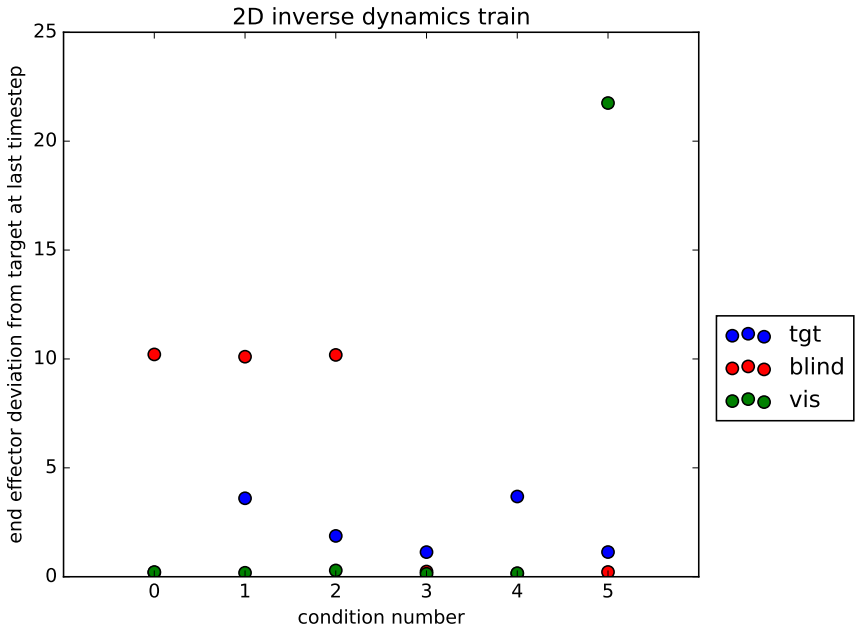

Figure 5.3: *Performance of the policy on the trained conditions in terms of the end effector distance from the target position, for all 3 inputs to the network.*

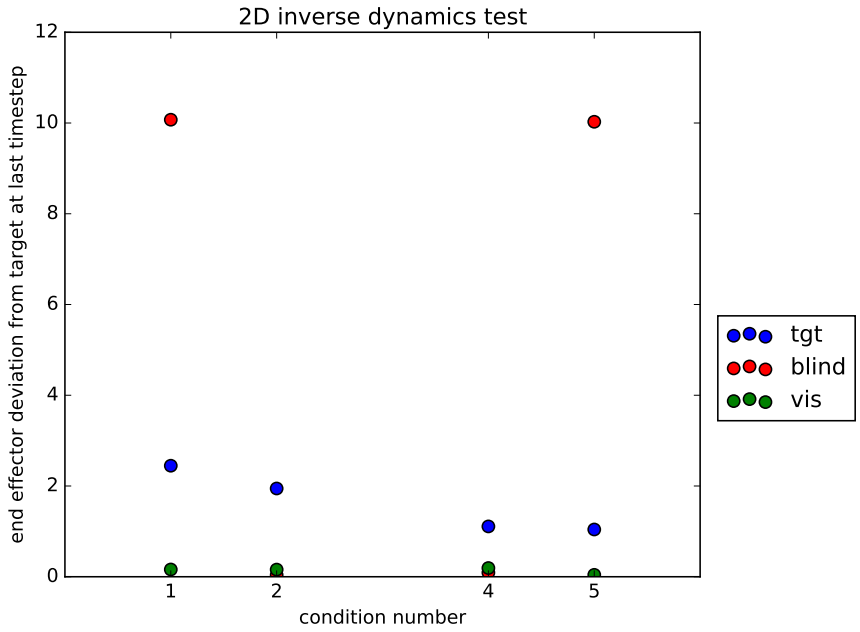

Figure 5.4: *Performance of the policy on the test conditions in terms of the end effector distance from the target position, for all 3 inputs to the network.*

Conclusion

As expected, the vision-based policy network's performance exceeds that of the blind network, which has learned a mapping from initial to target position that, given our train and test conditions, can only be right for half the conditions: there are 2 different target positions but the blind policy receives the exact same input regardless of target position so it can, at best, learn to move toward one target position.

The vision-based policy has a very low end effector deviation for all train and test conditions, except for train condition number 6. In this condition, the arm entered a part of joint space for which it was not trained, which just so happened to lead to a positive feedback.

Surprisingly, the vision-based policy outperforms the policy that receives exact target information. This is probably due to the slight movement of the estimated target position as the end effector covers part of the target. Instead of degrading the performance of the policy, this 'extra information' combined with the joint state apparently makes it easier for the network to position the arm more accurately.

Version of April 13, 2018– Created April 13, 2018 - 15:49

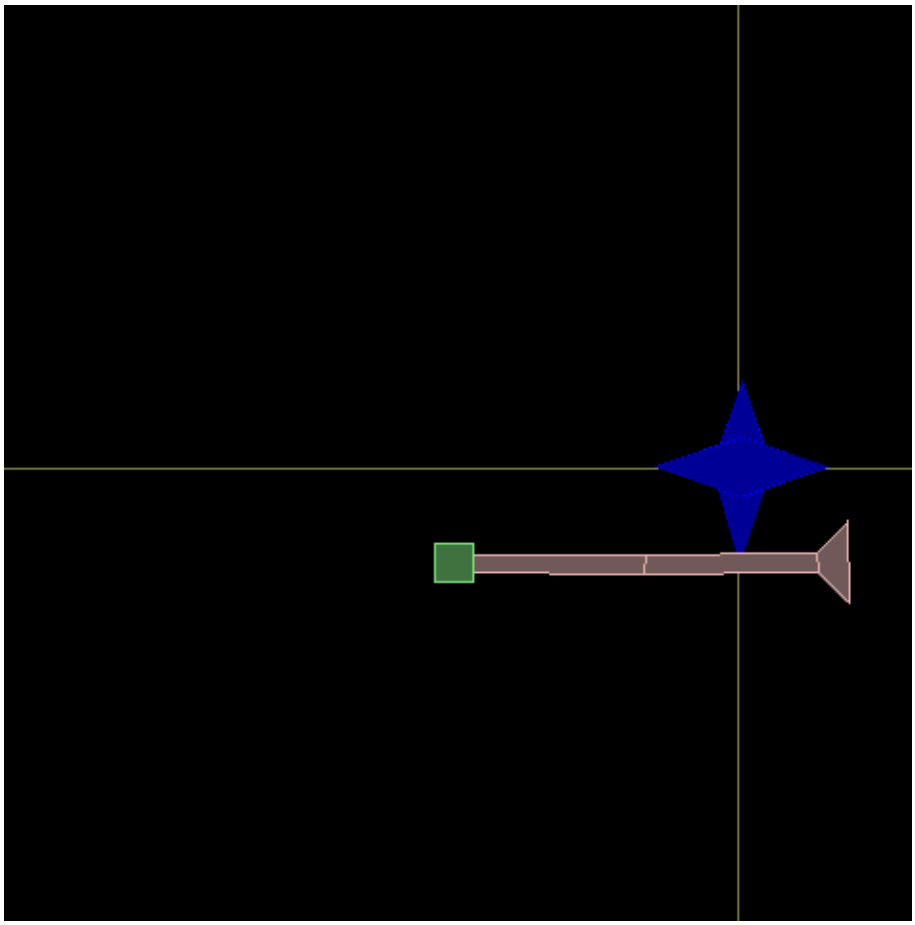

Figure 5.5: *The yellow lines intersect at the estimated position of the target, which corresponds perfectly to the actual target position when the arm does not overlap the target.*

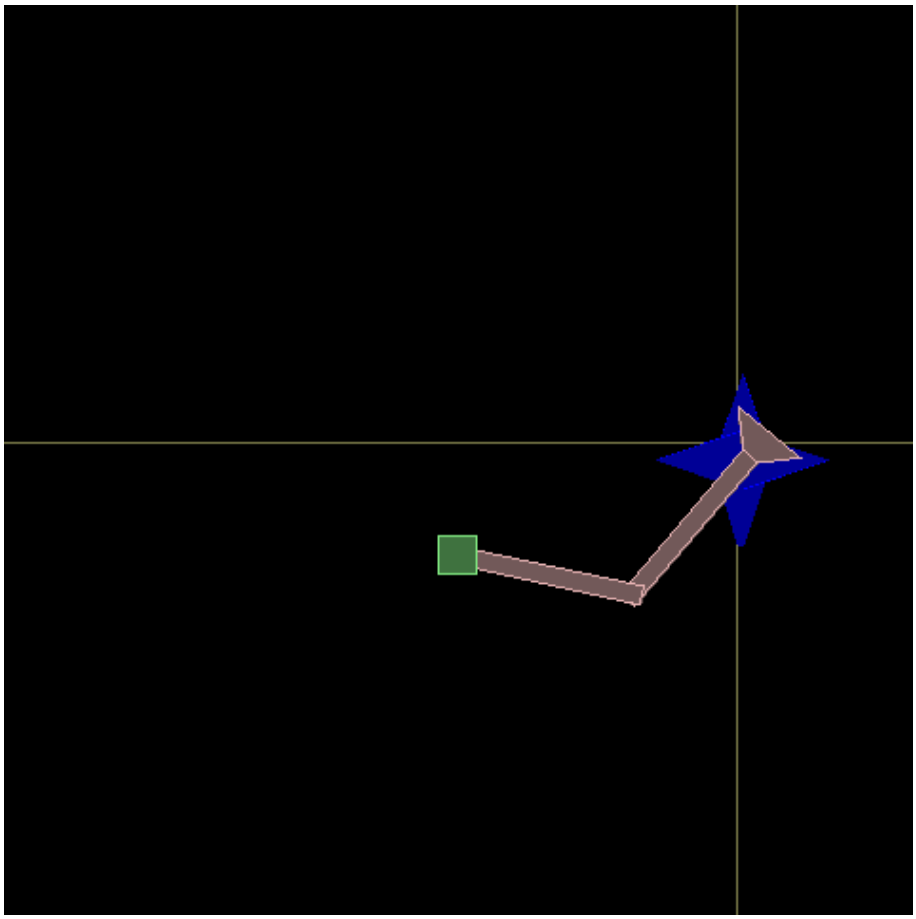

Figure 5.6: *The yellow lines intersect at the estimated position of the target. As shown here, this estimation deviates slightly when the arm overlaps the target. Remarkably, the policy network that received this input actually performed better than the one that received the actual target position.*

This remarkable result already shows the power of end-to-end learning: a conventional robotic pipeline would have chased the erroneous estimated target position, perhaps causing an offset from or oscillation around the actual target position, but a neural network can learn to use the information that this 'erroneous sensor' provides to outperform the perfect baseline. Strictly speaking, this is a form of overfitting: the policy network will only avoid this kind of overfitting if it harms its performance in one of the trained conditions. If it does not, it is a valid strategy.

Version of April 13, 2018– Created April 13, 2018 - 15:49

5.2 Network architecture comparison

Research question

How do choices in number of layers and pre-trained visual processing in the neural network architecture affect the performance of the policy network?

Hypothesis

The convolutional layers have to be pre-trained, but it is best to immediately feed the output of the convolutional layers (the soft-argmax features) into the policy network, which can then learn during Guided Policy Search (on the job) how to use that information. This is expected to work better than using the fully connected layers from the pre-trained model to feed only the estimated target position, since a lot of visual information is then lost. Due to the complexity of the task, one hidden layer is probably not enough: differences between distances must first be calculated, then a control decision must be made, so there is likely some optimum between 2 and 4 hidden layers.

Methods

The experiment from section 5.1 is used, with the exception that the on-axis alignment cost term is not used, and only 3 samples are taken per iteration per condition. A comparison is made using the lowest cost achieved on the training conditions only: we want to see which network can learn to best use the visual information to follow the guiding trajectories and mimimize cost, and are not interested in the generalization for this experiment. The compared architectures are:

tgt, n A target estimation network as in section 5.1: 2 fully connected hidden layers after the feature position layer are trained to predict a target position (x,y) which is then passed into a policy network with n hidden layers.

fp, n After training the target estimation network, only the convolutional layers are used: the (x,y) positions of all 10 features of the last convolutional layer are then passed into a policy network with n hidden layers.

Version of April 13, 2018– Created April 13, 2018 - 15:49

Results

network type, n	Cost
tgt, 2	463
fp, 1	286
fp, 2	261
fp, 3	682

Conclusion

No pre-trained processing of the visual features is better than using a fully pre-trained target position estimator. The optimal number of hidden layers is 2, which aligns with the expectations.

5.3 Box2D peg insertion

Research questions

Can a neural network learn a torque-based policy for 2D peg insertion, a task with non-linear, non-smooth dynamics, using Guided Policy Search with an LQR trajectory optimizer? Can a neural network learn to use visual information to generalize to different target positions, and how does this compare to a blind network and one that receives exact target information?

Hypothesis

A neural network peg insertion policy can be learned using GPS with an LQR trajectory optimizer. The blind network may perform well for a specific (range of) target positions, but will have lower performance when averaging across multiple target positions when compared to the vision and target network.

Methods

The simple Box2D environment is modified with an extra joint and a peg-shaped segment consisting of 2 triangles. The widest part of the 'peg' triangle is the tip, which has a width of 1 . The length of the peg from the tip to the broadside of the 'wrist' part of the end effector is 3 . In addition,

Version of April 13, 2018– Created April 13, 2018 - 15:49

2 static collidable blocks are placed just below the target with a small hole (width 1.6). The target position corresponds to the peg position when it is fully inserted.

The network architecture is slightly different for this one: the fully connected layers of the vision network have been removed, such that the policy network receives the full output of the soft-argmax layer (x,y positions of the highest activation of 5 filters in the final layer: 10 numbers in total), as this was found to improve performance (see section 5.2).

In this experiment, the coordinates of 2 points on the end effector, at the base and the tip, are used as end effector points data, so that the cost takes into account both the position and orientation of the end effector. For evaluation, only the distance of the tip from the goal was considered.

The binary cost term is now set to a distance of 2.5, so that it rewards only poses where the peg is almost fully inserted. An additional cost term is introduced when training the neural network using Guided Policy Search (but not during pre-training of the trajectories) using the same L1L2 norm ($L = 0.5 l_2 d^2 + l_1 \sqrt{\alpha + d^2}$) as the quadratically increasing cost, but with a weight of zero on the y-coordinates of both end effector points: thus it rewards alignment with the axis of the hole. 8 samples per iteration per condition were used during Guided Policy Search, for 4 iterations. 15 samples per condition per iteration were used during LQR trajectory pre-training, for 24 iterations.

For training conditions, the arm uses the same initial position (joint angles: $(-0.5\pi, 0)$), and 4 different target positions, see figure 5.8. For testing conditions, we tried 3 different target positions that are within the trapezium formed by the 4 training conditions. To evaluate the policy, 20 samples were taken per condition, with Gaussian noise added to the policy output.

Version of April 13, 2018– Created April 13, 2018 - 15:49

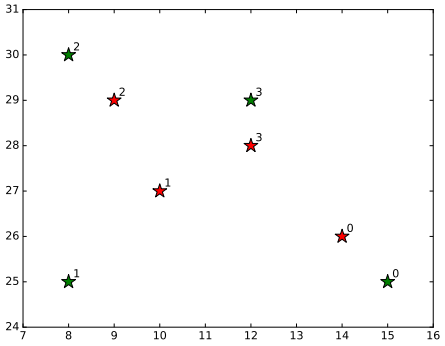

Figure 5.7: *Target positions for train and test conditions for the 3-joint peg insertion experiment. The base of the arm is at (0,15).*

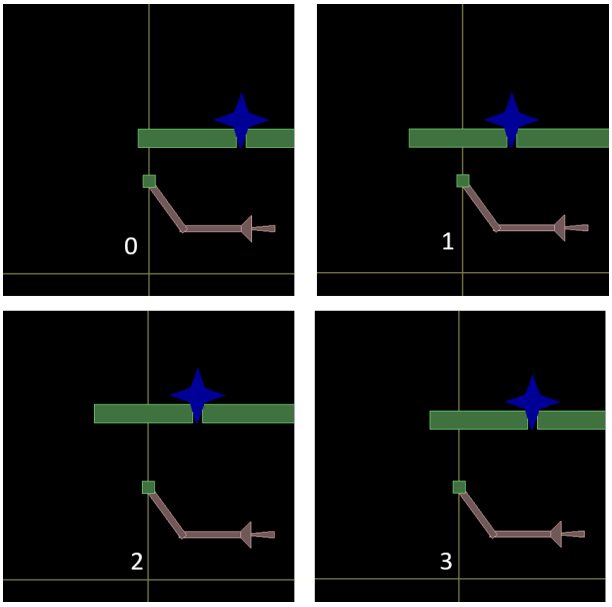

Figure 5.8: *Train conditions for the 3-joint peg insertion experiment. The arm is shown in the initial pose, the target position is marked by the blue star and the condition number (index) is shown in white. The yellow lines intersect at (0,0), the green base of the arm is at (0,15).*

Results

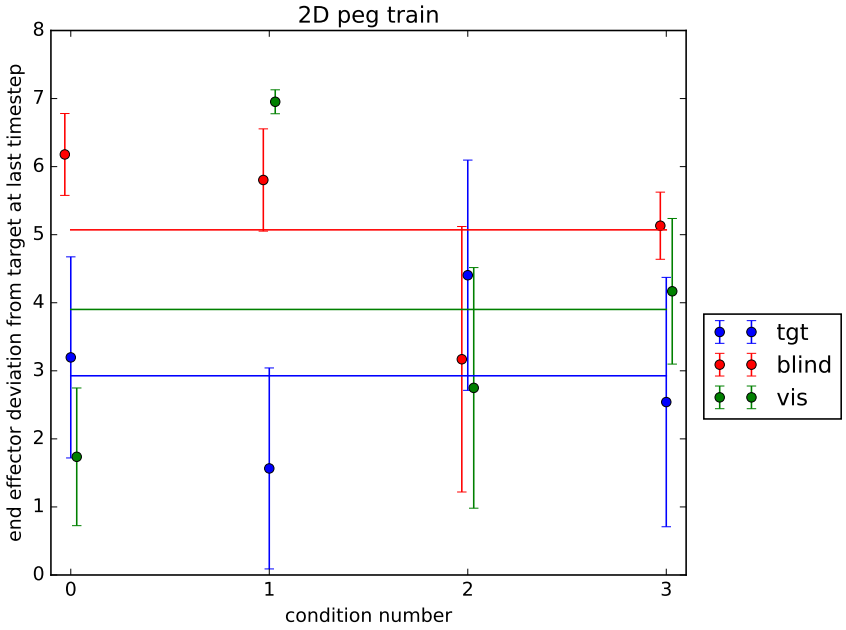

Figure 5.9: *Performance of the policy on the trained conditions in terms of the end effector distance from the target position, for all 3 inputs to the network. The horizontal lines are the average score across all conditions, the vertical lines indicate standard deviation per condition. As expected, the performance of the vision network is between the tgt and blind performance.*

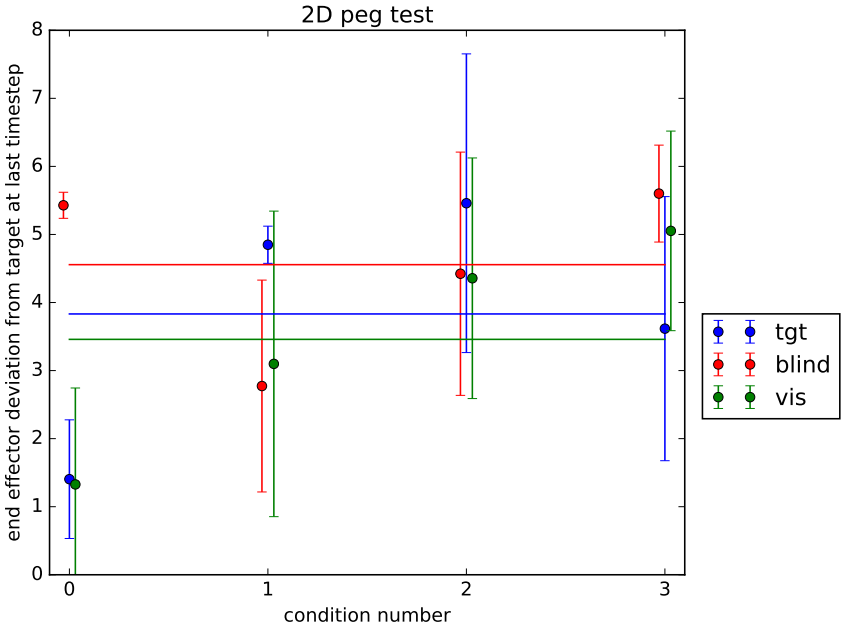

Figure 5.10: *Performance of the policy on the test conditions in terms of the end effector distance from the target position, for all 3 inputs to the network. The lines are the average score. The horizontal lines are the average score across all conditions, the vertical lines indicate standard deviation per condition. The performance of the blind policy is comparatively better than on the training set, because the test target positions are all closer to the center of the target area than under training conditions.*

Discussion and conclusion

This experiment is significantly more difficult. There are 2 reasons for this. The first is the non-linear, non-smooth nature of the peg insertion task. Theoretically, there is no proof that LQR trajectory optimization should work for this task[49], yet we still found it to work more reliably than the model-free trajectory update. The sudden movements it learned caused divergence during policy optimization. Secondly, there is an artificial difficulty due to the extreme inertia of the robot arm: no real robot arm would need as much torque to stop a rotation as this one, which was originally designed for the under-actuated arm balancing experiment.

Due to these difficulties, none of the learned policies can insert the peg consistently at each of the target positions. Still, the blind policy has a clearly worse score than the other policies. The difference is comparatively small during testing conditions due to the target positions being

Version of April 13, 2018– Created April 13, 2018 - 15:49

close there, but still the target and vision policies outperform the blind network on all but one condition. Condition (test, 1) is placed almost perfectly in the center, so it is to be expected that the blind policy has learned to move to that position.

5.4 3D 7-joint robot arm inverse kinematics

This may seem like a big step, but the algorithm is the very same: the only thing that changes is the number of dimensions. We also switch from torque commands to velocity commands: in the literature, the PR2 robot is used, which receives commands that are interpreted as the current through the electromotors. Due to the low inertia of the arms and the counter-weights, this is closer to joint velocity commands than to the commands in the previous experiments.

Research Questions

Can a neural network learn a policy for moving a 7-joint arm to a commanded target position in a 3D environment using joint velocity commands?

Hypothesis

A neural network policy for inverse kinematics of a 7-joint arm can be trained using Guided Policy Search.

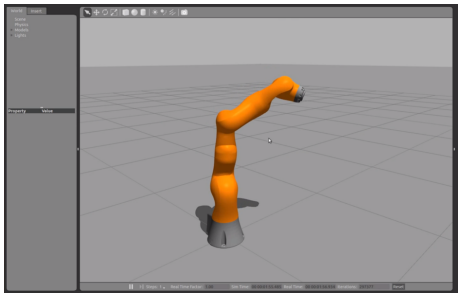

Figure 5.11: *Screenshot of the KUKA robot model in the Gazebo environment.*

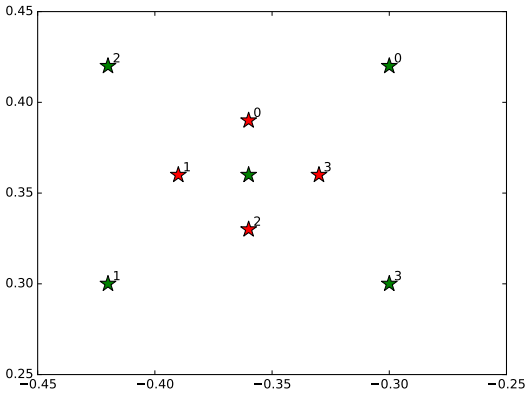

Figure 5.12: *Target positions for train and test conditions for the 7-joint 3D experiment. The z-coordinate for all target points is 0.4 meters.*

Methods

We use the KUKA IIWA 7-joint robot arm both in the real world and in the Gazebo 3D simulation environment. A joint velocity controller was implemented in Python to translate the joint velocity commands from the robot to joint position commands for the robot controller. The network now has 40 units per hidden layer, to account for the increased dimensionality of the system.

The end effector distance cost terms are now slighty modified: the L1 term is replaced by a logarithm to reward precise placement of the end effector: this presumably works better in 3D. The action term is still there, now the distance cost terms are $L = 0.5 l_2 d^2 + l_{\log} \log(\alpha + d^2)$

The target positions for train and test sets are shown in Figure 5.12. The policy is evaluated on end effector distance at final timestep, using 20 rollouts per condition for both train and test set.

Results

Version of April 13, 2018– Created April 13, 2018 - 15:49

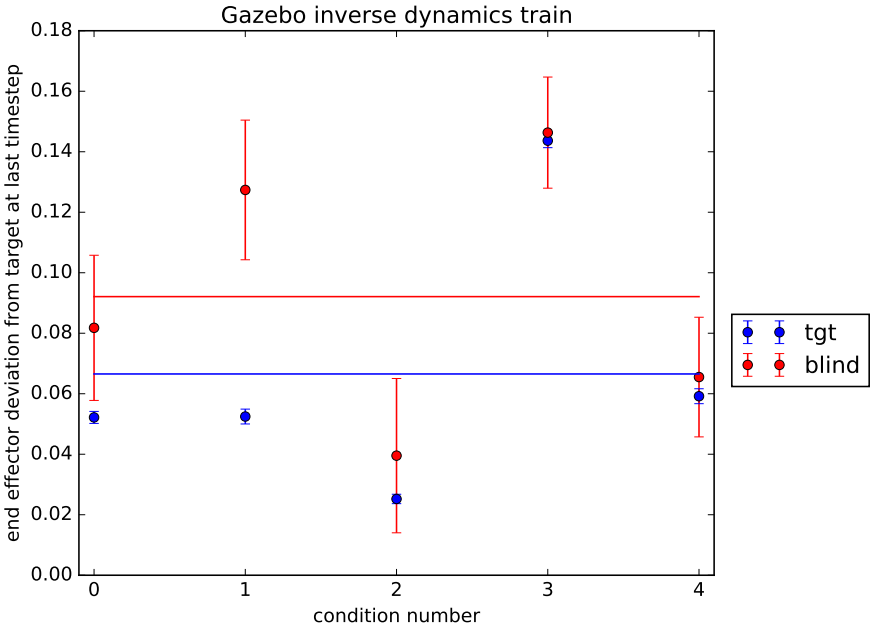

Figure 5.13: *Performance of the policy on the trained conditions in terms of the end effector distance from the target position, for both networks at different KL step sizes. The horizontal lines are the average score across all conditions, the vertical lines indicate standard deviation per condition. Note the much smaller standard deviations of the tgt policy: this indicates stronger feedback terms, allowing less exploration: the tgt policy is surer of where to go than the blind policy. As expected, performance on target position 4 (center) is comparable for the 2 networks. Oddly enough, the tgt policy has failed to learn a good policy for condition 3.*

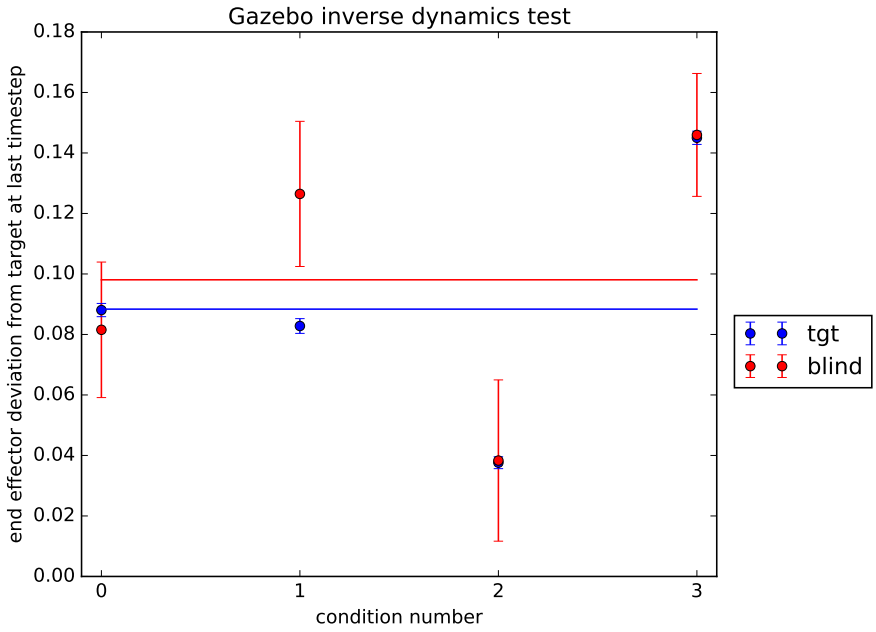

Figure 5.14: *Performance of the policy on the test conditions in terms of the end effector distance from the target position, for both networks. The horizontal lines are the average score across all conditions, the vertical lines indicate standard deviation per condition. As the test conditions are rather close to the center, there is not much difference in performance between both networks. The tgt policy only substantially outperforms the blind policy in condition 1.*

Discussion and conclusion

The experiment is not successful yet: the policy only performs slightly better than a blind policy: it has learned how to move the arm to the approximate target area, but is only barely using the specific information about the target position that it has been given. This could be due to the dynamics fit: a lot of time has been put into tuning the fitting hyperparameters for the 2D environment. Also, the dynamics GMM fit was designed for (simple, low inertia) torque control: it requires that there are at least twice as many state as action dimensions. The state would be given by the joint angles and velocities in torque control, but for velocity control the previous joint velocity is not really relevant. Since it is provided anyway, the dynamics fit is fitting to irrelevant values.

As in previous experiments, we also see severe limitations in the generalization of the learned policy. There are a number of ways to improve generalization in the policy network: add more conditions, add noise to policy inputs during training and/or use a non-adaptive optimizer [59] such as SGD (with (Nesterov) momentum).

Chapter 6

Discussion

Theory

Most Deep Learning methods need a lot of data and a lot of training time. However, this mainly due to the complexity of the function the network needs to learn. In order to learn a more complex function, a deeper network with more parameters is needed: this also increases the need for training data and training time.

Reinforcement Learning methods with little bias (thus: a lot of exploration) (such as Q-learning methods) need a lot of samples (data) because the Q-function is often quite complicated: it is only fully specified once every possible sequence of states and actions has been visited.

In robotics, we need a very sample-efficient method: combining the above statements seems to indicate that Deep Learning and Reinforcement Learning together are not a good fit: they both need a lot of data, which we cannot provide.

This is why Deep Learning in Robotics initially focused on Grasp Quality Convolutional Neural Networks: using synthetic data means there is a lot of data available, and the problem of grasp quality is not a reinforcement learning problem, it's just classification. [11] [12] [10]

Learning from demonstrations with a regular recurrent network is only feasible with multi-task learning, and even then it requires on the order of 2 full workdays of demonstrations [13].

The solution to the problem of reinforcement learning in robotics, as used in Guided Policy Search, is in the cost (or reward) function: using a very rich reward with a non-zero gradient in large parts of the state-space, shapes the cost landscape to guide the policy network to a locally optimal solution. Additionally, such a cost function allows application of the rela-

tively simple time-varying LQR optimizer: using a second-order approximation of the cost and a linear approximation of the dynamics, the action can be optimized per timestep. Optimizing in action-space like this is a lot more sample-efficient than optimizing the neural network parameters directly using the cost function. The resulting time-varying linear-gaussian controllers can be used to guide the neural network to a locally optimal solution.

But doesn't the network need a lot of samples to learn to imitate these controllers? The answer is that the policy network does not need to learn a very complicated function: a non-linear controller based on positions and joint angles. Unlike a classifier for classifying images into thousands of categories, this is a kind of function that could almost be handcrafted. As a result, the network is smaller and does not need as many samples. Because this is a reactive policy, and it does not need to take into account observations from previous timesteps, every single timestep of the sampled linear-gaussian controller is a datapoint.

This makes Guided Policy Search a very sample-efficient way of Deep Reinforcement Learning. [8] [2] [36] [37] [38] [55] [51] [45]

Experiments

Sections 5.1 and 5.3 show that with Guided Policy Search, the policy network can learn to use incomplete information to its advantage.

We found that using pre-trained trajectories was necessary due to the inertia of the arm, which complicated the dynamics: undesirable local optima were easily reached where the arm would be spun around by a neural network which only provided positive feedback. This was probably in part due to the dynamics fit: a Gaussian Mixture Model is fit to the global dynamics (every timestep), then linear dynamics are found by linear regression starting from the global prior. This is optimized for systems with a high number of dimensions compared to the number of samples (7 joint robot arm: at least 14 dimensions (joint angles and joint velocities)), and the hyperparameters had to be tuned significantly to work for the 2-joint arm. The best way to evade this behaviour is starting guided policy search from trajectories that overshoot: this demonstrates the negative feedback to the network, so it can learn a proper feedback policy. When shown samples that do not change movement direction, this may lead to the network learning a bias.

When reading about the trajectory optimizers in Section 4.1.1, it would

Version of April 13, 2018– Created April 13, 2018 - 15:49

appear that the PILQR optimizer that combines model-based and model-free updates is best suited to tasks like peg insertion. In our experiments however, the model-free update would cause divergence when optimizing the policy, unless the covariance damping parameter was set so high that the model-free update did not change the trajectory noticeably, and performance was not improved over the standard LQR optimizer.

Overfitting shoud be mitigated by use of noise on inputs during policy optimization, and/or non-adaptive optimizers[59].

Training time was not much: at 5 seconds per sample, the first experiment, including pre-training, clocks in at 6x9x3 = 162 seconds (without any demonstration). For peg insertion, many more samples were used, but even the total sample time for that experiment is 4x(8x4 + 15x24) = 1568 seconds, less than half an hour. This is a factor 30 less robot time than the demonstration experiment, and can be done without any human involvement: the robot controls itself. So how about the network optimization? Training times for state-of-the-art classifier or generator networks can be weeks or months, but in our case, 16.000 optimization passes take only a few minutes (2 minutes if there are no convolutional layers). Multiply this by the number of iterations and we are looking at about an hour of calculation time on a 3 year old GTX970 GPU. This could be sped up further by doing the network optimization on a cloud service using multiple high-end GPUs if necessary.

Chapter 7

Conclusion

Can Deep Reinforcement Learning be used to automate tasks in real-world robotic applications? To which extent? What are its requirements? What are its strengths and weaknesses?

Deep Reinforcement Learning works best when exploration and learning can be done in simulation. For real-world applications that require a high sample efficiency, Guided Policy Search allows training of a neural network that generalizes and can act based on partial observations. It requires that the full state of the system is known during training, unless a clever encoding can be trained. This has been done on image data and might be adapted to other data with spatial correlations.

The main strength of a network trained with Guided Policy Search is that it can learn to use incomplete or erroneous data. The main weakness is that it only works if a proper cost function has been designed. Cost function design can be replaced with human demonstrations using Guided Cost Learning [45]. This is a more sample-efficient method (10 minutes of demonstrations) than training a recurrent network directly on demonstrations (15 hours of demonstrations) and involves using sample-based Inverse Optimal Control to train a neural network to learn a non-linear cost function that can be optimized effectively.

Specifying a cost function that is both easily learnable and rewards successful policies is often difficult. Then, demonstrations can be provided and a neural network can then be used to learn an effective cost function in the inner loop of Guided Policy Search: this is called Guided Cost Learning [45].

Yes, real-world reinforcement learning tasks can be automated using Guided Policy Search, but generalization remains difficult and limited, and a lot of human effort is till required to implement such a method.

Version of April 13, 2018– Created April 13, 2018 - 15:49

Chapter 8

Connection with the physics master

Deep Learning is not usually the subject of a physics master's project. In this section, I will answer two common questions: Why did I choose for this subject? What is the value of this project for LION?

Motivation

Deep Learning has revolutionized pattern recognition and reconstruction in numerous modalities: image [3] [4] [5], speech [60] [61], radar [62].

More importantly, a Deep Learning method has recently been invented which mastered all board games [6].

This suggests that Deep Learning can do more than just pattern recognition: artificial intelligence can now learn complex tasks.

If artificial intelligence can learn complicated tasks like playing Go, then an application in the physics laboratory is not far off.

I wanted to know what artificial intelligence can and can't learn, and how to train a network for such tasks.

Value for physics department

This internship within the center of Deep Learning in TNO has taught me a lot about Deep Learning methods and their strong and weak points. This will allow me to recognize problems tha can benefit from a Deep Learning approach and help inform colleagues about which Deep Learning solutions are likely to work for a given problem.

This kind of inter-disciplinary knowledge within the institute is needed for a new and unknown area like Deep Learning outside the pattern recognition domain.

Deep Learning has solved all board games, can it solve the problems in our physics laboratory?

One example of a physicist who uses deep neural networks to solve a real problem in the laboratory is Baireuther [63], who trained a recurrent neural network to function as a decoder in Quantum Error Correction, surpassing the performance of all previous decoders.

Version of April 13, 2018– Created April 13, 2018 - 15:49

Bibliography

[1] *i-botics (http://i-botics.com/)*.

[2] S. Levine, C. Finn, T. Darrell, and P. Abbeel, *End-to-End Training of Deep Visuomotor Policies*, Journal of Machine Learning Research **17**, 1 (2015).

[3] A. Krizhevsky, I. Sutskever, and G. E. Hinton, *ImageNet Classification with Deep Convolutional Neural Networks*, Advances In Neural Information Processing Systems , 1 (2012).

[4] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, *Going deeper with convolutions*, in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, volume 07-12-June, pages 1–9, 2015.

[5] K. He, X. Zhang, S. Ren, and J. Sun, *Delving deep into rectifiers: Surpassing human-level performance on imagenet classification*, in *Proceedings of the IEEE International Conference on Computer Vision*, volume 2015 Inter, pages 1026–1034, 2015.

[6] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, *Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm*, arXiv e-prints (2017).

[7] L. Pinto and A. Gupta, *Supersizing self-supervision: Learning to grasp from 50K tries and 700 robot hours*, in *Proceedings - IEEE International Conference on Robotics and Automation*, volume 2016-June, pages 3406–3413, 2016.

[8] S. Levine, P. Pastor, A. Krizhevsky, and D. Quillen, *Learning Hand-Eye Coordination for Robotic Grasping with Deep Learning and Large-Scale Data Collection*, The International Journal of Robotics Research **1** (2016).

[9] E. Jang, S. Vijayanarasimhan, P. Pastor, J. Ibarz, and S. Levine, *End-to-End Learning of Semantic Grasping*, arXiv e-prints (2017).

[10] J. Mahler, M. Matl, X. Liu, A. Li, D. Gealy, and K. Goldberg, *Dex-Net 3.0: Computing Robust Robot Suction Grasp Targets in Point Clouds using a New Analytic Model and Deep Learning*, arXiv e-prints (2017).

[11] J. Mahler, J. Liang, S. Niyaz, M. Laskey, R. Doan, X. Liu, J. A. Ojea, and K. Goldberg, *Dex-Net 2.0: Deep Learning to Plan Robust Grasps with Synthetic Point Clouds and Analytic Grasp Metrics*, arXiv e-prints (2017).

[12] J. Mahler, M. Matl, X. Liu, A. Li, D. Gealy, and K. Goldberg, *Dex-Net 3.0: Computing Robust Robot Suction Grasp Targets in Point Clouds using a New Analytic Model and Deep Learning*, arXiv e-prints (2017).

[13] R. Rahmatizadeh, P. Abolghasemi, L. Bölöni, and S. Levine, *Vision-Based Multi-Task Manipulation for Inexpensive Robots Using End-To-End Learning from Demonstration*, arXiv e-prints (2017).

[14] B. Widrow, *An Adaptive Adaline Neuron Using Chemical Memistors*, Technical report, 1960.

[15] M. Minsky and S. Papert, *Perceptrons.*, M.I.T. Press, 1969.

[16] J. Schmidhuber, *Deep Learning in neural networks: An overview*, 2015.

[17] Y. LeCun and Y. Bengio, *Convolutional networks for images, speech, and time series*, The handbook of brain theory and neural networks **3361**, 255 (1995).

[18] A. Lazrak, A. Leconte, D. Chèze, G. Fraisse, P. Papillon, and B. Souyri, *Numerical and experimental results of a novel and generic methodology for energy performance evaluation of thermal systems using renewable energies*, Applied Energy **158**, 142 (2015).

[19] S. Theodoridis, *Neural Networks and Deep Learning*, in *Machine Learning*, pages 875–936, Determination Press, 2015.

[20] V. Dumoulin and F. Visin, *A guide to convolution arithmetic for deep learning*, (2016).

[21] A. Dertat, *Applied Deep Learning*, 2017.

[22] A. Makhzani, J. Shlens, N. Jaitly, I. Goodfellow, and B. Frey, *Adversarial Autoencoders*, (2015).

[23] I. Goodfellow, J. Pouget-Abadie, and M. Mirza, *Generative Adversarial Networks*, arXiv preprint arXiv: ... , 1 (2014).

[24] A. Rasmus, H. Valpola, M. Honkala, M. Berglund, and T. Raiko, *Semi-Supervised Learning with Ladder Networks*, (2015).

[25] Y. Bengio, *Learning Deep Architectures for AI*, Foundations and Trends® in Machine Learning **2**, 1 (2009).

[26] T. Silva, *A Short Introduction to Generative Adversarial Networks - Thalles' blog*.

[27] T. Karras, T. Aila, S. Laine, and J. Lehtinen, *Progressive Growing of GANs for Improved Quality, Stability, and Variation*, arXiv e-prints (2017).

[28] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, *Image-to-Image Translation with Conditional Adversarial Networks*, (2016).

[29] C. Hesse, *Image-to-Image web demo*, 2017.

[30] S. Yang, L. Xie, X. Chen, X. Lou, X. Zhu, D. Huang, and H. Li, *Statistical Parametric Speech Synthesis Using Generative Adversarial Networks Under A Multi-task Learning Framework*, arXiv e-prints (2017).

[31] J. Wang and L. Perez, *The Effectiveness of Data Augmentation in Image Classification using Deep Learning*, Unpublished (2017).

[32] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, *Dropout: A Simple Way to Prevent Neural Networks from Overfitting*, Journal of Machine Learning Research **15**, 1929 (2014).

[33] R. Bellman, *Dynamic Programming*, Princeton University Press, 1957.

[34] R. J. Williams, *Simple statistical gradient following algorithms for connectionist reinforcement learning*, Machine Learning **8**, 229 (1992).

[35] S. Levine and P. Abbeel, *Learning Dynamic Manipulation Skills under Unknown Dynamics with Guided Policy Search*, Advances in Neural Information Processing Systems 27 , 1 (2014).

[36] C. Finn, X. Y. Tan, Y. Duan, T. Darrell, S. Levine, and P. Abbeel, *Deep spatial autoencoders for visuomotor learning*, in *Proceedings - IEEE International Conference on Robotics and Automation*, volume 2016-June, pages 512–519, 2016.

[37] Y. Chebotar, M. Kalakrishnan, A. Yahya, A. Li, S. Schaal, and S. Levine, *Path integral guided policy search*, in *Proceedings - IEEE International Conference on Robotics and Automation*, pages 3381–3388, 2017.

[38] Y. Chebotar, K. Hausman, M. Zhang, G. Sukhatme, S. Schaal, and S. Levine, *Combining Model-Based and Model-Free Updates for Trajectory-Centric Reinforcement Learning*, Proceedings of Machine Learning Research **70** (2017).

[39] G. A. Rummery and M. Niranjan, *On-line Q-learning using connectionist systems*, page Technical Report CUED/F/INFENG/TR 166 (1994).

[40] V. Mnih, D. Silver, and M. Riedmiller, *Playing Atari with Deep Reinforcement Learning*, Nips , 1.

[41] J.-B. Mouret, S. Koos, and S. Doncieux, *Crossing the Reality Gap: a Short Introduction to the Transferability Approach*, (2013).

[42] A. Tamar, Y. Wu, G. Thomas, S. Levine, and P. Abbeel, *Value iteration networks*, in *IJCAI International Joint Conference on Artificial Intelligence*, pages 4949–4953, 2017.

[43] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. P. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, *Asynchronous Methods for Deep Reinforcement Learning*, arXiv e-prints (2016).

[44] S. Ross, G. Gordon, and A. Bagnell, *A reduction of imitation learning and structured prediction to noregret online learning*, Journal of Machine Learning Research **15**, 627 (2011).

[45] C. Finn, S. Levine, and P. Abbeel, *Guided Cost Learning: Deep Inverse Optimal Control via Policy Optimization*, (2016).

[46] F. Sadeghi and S. Levine, *RL: Real Single-Image Flight Without a Single Real Image Training entirely in simulation Test in real world*, arXiv e-prints (2017).

[47] C. Finn, P. Abbeel, and S. Levine, *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*, arXiv e-prints (2017).

[48] A. Nichol and J. Schulman, *Reptile: a Scalable Metalearning Algorithm*, (2018).

[49] W. Li and E. Todorov, *Iterative linear quadratic regulator design for nonlinear biologoical movement systems*, 1st International Conference on Informatics in Control, Automation and Robotics , 1 (2004).

[50] S. Ioffe and C. Szegedy, *Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift*, Proceedings of Machine Learning Research **37** (2015).

[51] M. Zhang, Z. Mccarthy, C. Finn, S. Levine, and P. Abbeel, *Learning Deep Neural Network Policies with Continuous Memory States*, ICRA (2016).

[52] C. Finn, M. Zhang, J. Fu, X. Y. Tan, Z. McCarthy, E. Scharff, and S. Levine, *Guided Policy Search Code Implementation*, 2016.

[53] E. Catto, *Box2D*, 2013.

[54] *Gazebo*.

[55] W. Montgomery and S. Levine, *Guided Policy Search as Approximate Mirror Descent*, (2016).

[56] D. P. Kingma and J. Ba, *Adam: A Method for Stochastic Optimization*, arXiv e-prints (2014).

[57] G. Van Rossum, *Python 2.7*.

[58] M. Abadi et al., *TensorFlow: A System for Large-Scale Machine Learning TensorFlow: A system for large-scale machine learning*, in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI '16)*, pages 265–284, 2016.

[59] A. C. Wilson, R. Roelofs, M. Stern, N. Srebro, and B. Recht, *The Marginal Value of Adaptive Gradient Methods in Machine Learning*, NIPS (2017).

[60] Z. Zhang, J. Geiger, J. Pohjalainen, A. E.-D. Mousa, W. Jin, and B. Schuller, *Deep Learning for Environmentally Robust Speech Recognition: An Overview of Recent Developments*, arXiv e-prints (2017).

[61] W. Xiong, J. Droppo, X. Huang, F. Seide, M. Seltzer, A. Stolcke, D. Yu, and G. Zweig, *Achieving Human Parity in Conversational Speech Recognition*, arXiv e-prints (2016).

[62] B. Yonel, E. Mason, and B. Yazici, *Deep Learning for Passive Synthetic Aperture Radar*, arXiv e-prints (2017).

[63] P. Baireuther, T. E. O'Brien, B. Tarasinski, and C. W. J. Beenakker, *Machine-learning-assisted correction of correlated qubit errors in a topological code*, Quantum (2017).