Deep Reinforcement Learning with Direct Policy Search and Regularized Convolutional Neural Fitted Q Iteration

by
Cosmo Harrigan

A thesis presented for the degree of **Bachelor of Science in Applied and Computational Mathematical Sciences: Discrete Math and Algorithms**

at the University of Washington
March 2016

Thesis Supervisor: Dr. Dieter Fox

Abstract

We review the deep reinforcement learning setting, in which an agent receiving high-dimensional input from an environment learns a control policy without supervision using multilayer neural networks. We investigate two different approaches: neuroevolution for direct policy search, and Neural Fitted Q Iteration for learning an action-value function. We reproduce some results from an experiment by (Koutnik et al) where a policy is learned from pixels with the *TORCS* race-car simulator. We then extend the Neural Fitted Q Iteration algorithm (Riedmiller et al) by introducing a novel variation called *Regularized Convolutional Neural Fitted Q Iteration* (RC-NFQ) that incorporates convolutional neural networks similarly to the Deep Q Network algorithm (Mnih et al) and dropout regularization to improve generalization performance. Finally, we present preliminary results indicating learning progress when applied to a robot in a real-world environment.

Contents

Chapter 1

Neural Networks

1.1 Introduction

Neural networks are non-linear parametric statistical models. There are multiple supervised and unsupervised learning algorithms associated with them. Learning multilayer neural network models, also known as *deep learning*, involves choosing a model type and an architecture for the model, and then learning suitable model parameters. While all such models consist of directed graphs of artificial neurons, many specific higher-level regularities have been proposed as building blocks. Examples include convolutional neural networks and recurrent neural networks, which are covered in a subsequent section. They are proposed in order to counteract the combinatorial explosion of possible network configurations based on the following observations about the domains operated upon in the natural world:

- The world contains many statistical regularities
- Similar patterns tend to appear in multiple regions of space and time
- Natural scenes have a hierarchical, compositional structure

The most common learning algorithm for neural networks is the backpropagation algorithm [1] [2], which is a supervised learning method that uses labeled examples, and computes the difference between the network's output and the target output, and iteratively makes small adjustments to the network until its predictions better match the labeled examples.

An excellent survey of the field is available in [3] containing an overview of each subfield, along with historical developments and 888 references. A textbook, *Deep Learning* [4], is also available.

1.2 Convolutional Neural Networks

Convolutional neural networks [2] [5] [6] [4, Chapter 9] are multi-layer perceptrons with particular constraints on their weights. They have demonstrated significant successes in recent years in the field of computer vision [7]. They demonstrate some similarities to the human visual cortex, although they also exhibit many differences [8] [9].

Within each layer there exists a set of *feature detectors*, each of which responds to the presence of a particular pattern in an input tensor. Each feature detector is applied at multiple locations in the input tensor. This application of identical feature detectors across the input is referred to as *weight sharing*. At each location, the extent to which the feature is present is calculated by the *discrete convolution* operator. To represent the information more compactly, *dimensionality reduction* techniques which aim to extract the most relevant components are utilized between layers; usually *max pooling* layers [2], which only consider the most prominent feature within a particular location, or, more recently, *strided convolutions* [10], which spatially separate the application of the feature detectors in such a way as to reduce the dimensionality of the output by reducing the amount by which they overlap.

1.3 Recurrent Neural Networks

Recurrent neural networks [1] [4, Chapter 10] are neural network models that contain cycles, so that they are no longer directed acyclic graphs. They are applicable to problems that involve sequential data; they represent the observation that regularities persist through time, by implementing a *weight sharing* constraint that requires that parameters of the replicas in time of each neuron be the same at each of the different time steps. As noted in [1] [4, Section 10.1], for any recurrent neural network operation over a finite length of time, there exists an equivalent feedforward network, by "unrolling" the network through time.

1.4 Neuroevolution

1.4.1 Overview

In *neuroevolution*, genetic algorithms are used to optimize the parameters of a neural network [11] [12]. Because of this, in contrast to backpropagation, neuroevolution is not a gradient-based learning algorithm. For an overview of the field, see [3, Section 6.6]. There are three general classes of neuroevolution: the optimization of the structure of a neural network, the optimization of the weights given a predefined structure, or the optimization of a separate network which is used to generate the solution network through some type of transformation. In our case, we study the second of these methods: the use of genetic algorithms to learn the connection weights of a neural network.

1.4.2 Genetic Algorithms

Genetic algorithms [13] [14] are a class of optimization methods which use techniques that were originally inspired by the process of evolutionary natural selection. They consist of a framework in which a *population* of *candidate solutions* is *evolved* over multiple *generations*. A population consists of a set of candidate solutions. A candidate solution is defined the same way as in traditional optimization. The process of evolution involves the steps of *selection*, *crossover* and *mutation*. We review the algorithm for neuroevolution, a particular application of genetic algorithms, in Section 4.2.

1.5 Gradient Descent Algorithms

Gradient descent in neural networks is an optimization method that aims to minimize an objective function $J(\theta)$ parameterized by the parameter vector θ of a neural network, by iteratively updating a candidate solution based on the local gradient $\nabla_\theta J(\theta)$ of the function to be optimized with regards to its parameters, evaluated on some training examples. The solution is updated by taking steps in the opposite direction of the gradient. The magnitude of each update is controlled by a learning rate η.

In full-batch gradient descent, all of the training examples are used to compute the gradient at each step. A variant, called stochastic gradient descent, uses a single training example at each step. There is yet another variant, called mini-batch gradient descent, which lies between these two concepts and is the most commonly used. It uses a subset of the training examples at each step to compute the gradient, which can lead to faster convergence behavior.

There exist more sophisticated gradient descent algorithms that aim to adaptively change the learning rate on a per parameter basis as the optimization proceeds. One of these algorithms, which we will use in this work, is RMSprop [15] which is a mini-batch varient of the Rprop [16] batch gradient descent algorithm. Other common alternatives are Adagrad [17], Adadelta [18] and Adam [19].

Chapter 2

Reinforcement Learning

2.1 Introduction

Reinforcement learning [20] [21, Section 3] is a general framework for the problem of an *agent* embedded in an unknown *environment* that learns a behavior *policy* that maximizes a *reward function*. The standard reference for the field is [20], and a survey of recent techniques is found in [22]. Under the reinforcement learning paradigm, an agent sends actions to an environment, and receives observations and rewards in response.

Reinforcement learning belongs to the fields of machine learning and artificial intelligence. For a survey of definitions of artificial intelligence, see [21]. For a broad view of artificial intelligence in general, see [23]. For a universal theory of optimal artificial intelligence and reinforcement learning, refer to [24] and [25].

As a general framework, there are many specific methods that can be described within the reinforcement learning setting. Unsupervised learning methods can also be considered part of reinforcement learning when they are employed by the agent to improve its ability to achieve rewards. For a survey of unsupervised learning applied in this context, see [3, Section 6.4].

The most common formalism used to describe this setting involves the agent observing a state s and taking an action a, and the environment responding with a reward signal r along with a subsequent state s'. This agent-environment interaction is then repeated through time, defining a sequence of experience tuples (s, a, r, s') which can be used as input to various reinforcement learning algorithms.

If the environment is *fully observable*, then the observation at a particular time corresponds to the exact configuration of the environment; in more complex problems, the environment may only be *partially observable*, in which case the agent only has access to a particular subset of the true state of the environment through its observations.

The policy of an agent represents how the agent behaves, conditional on its observations of the environment and the state of its internal memory. There are many ways that a policy can be represented. It can be as simple as a lookup table, which is equivalent to a *reflex agent* [23] in artificial intelligence. Alternatively, the agent can parameterize its policy in some way, so that its responses become a more complex function of the state of the environment and its internal state.

A further distinction is made between *model-based* reinforcement learning, in which an agent learns an explicit model of the conditional *transition probabilities* $P(s'|s, a)$ of the environment, and *model-free* reinforcement learning, in which an agent does not explicitly model the environment. There are two major approaches within the model-free reinforcement learning setting: *direct policy search* methods and *value-based methods*, both of which we cover below.

2.2 Exploration and Exploitation

An important tradeoff in reinforcement learning exists between the exploitation of current estimates to increase immediate reward and further exploration of the environment to improve

the likelihood of future reward through improved accuracy. This is commonly referred to as the *exploration versus exploitation* tradeoff [20, Chapter 1].

2.3 Function Approximation

The simplest category of reinforcement learning problems are the *tabular* case, in which each state is explicitly defined, and state-specific values are learned. This corresponds to an explicit representation in the form of a lookup table with an entry for every single state. In all but the simplest domains, this approach is not practically feasible, due to the size of the state space.

As a result, *function approximators* are used to generalize from specific states into more abstract state representations. The concept of function approximation is related to the concepts of regularization and generalization in statistics, and compression in information theory. In general, a function approximator takes as input a high-dimensional state, and maps it to a lower dimensional representation. Both linear and non-linear function approximation methods are studied. Neural networks are a particular instance of non-linear function approximators.

2.4 Direct Policy Search

In *direct policy search*, the space of possible policies is searched directly. The agent does not attempt to model the transition dynamics of the environment, nor does it attempt to explicitly learn the value of different states or actions. Instead, it iteratively attempts to improve a parameterized policy. When function approximation is used, direct policy search is related to approximate policy iteration.

Direct policy search can be broken down into gradient-based methods, also known as policy gradient methods, and methods that do not rely on the gradient. In policy gradient methods, often the gradient is not explicitly known and must approximated via sampling from the environment. An example of these methods is the REINFORCE algorithm [26]. Gradient-free methods include evolutionary algorithms, which were introduced previously. We will describe one of these algorithms in detail in section 4.2. A survey on policy search and applications to robotics is [27].

2.5 Value-Based Methods

In value-based reinforcement learning methods, an agent learns the expected reward r conditioned on a particular action a in a certain state s. This defines a function which is called the *action-value function*.

A central theme in many reinforcement learning algorithms is *temporal-difference* (TD) learning. In value-based TD learning, an approximation of the action-value function is used as an initial estimate and is compared to sample returns obtained from the environment. Iterative updates are then made based on the observed error, called the *TD error*. This is an example of *bootstrapping*.

A distinction is made between *on-policy* and *off-policy* reinforcement learning algorithms. On-policy algorithms learn the value of the policy that is actually carried out by an agent, which includes exploration steps that it may take, whereas off-policy algorithms learn the value of an optimal policy that is independent of the actual policy being followed.

2.5.1 Q-learning

Q-learning [28] is a commonly used off-policy value-based reinforcement learning algorithm. We define the action-value function as $Q(s, a)$ and define an iterative update procedure called *one-step Q-learning* [20, Chapter 6] as follows:

$$Q_{t+1}(s_t, a_t) \leftarrow Q_t(s_t, a_t) + \alpha \left(r_{t+1} + \gamma \max_a Q_t(s_{t+1}, a) - Q_t(s_t, a_t) \right)$$

where α is a learning-rate parameter, and γ is a parameter that *discounts* the value of future rewards versus present rewards.

Chapter 3

Reinforcement Learning with Neural Networks

3.1 Introduction

As mentioned previously, neural networks can be used as non-linear function approximators. They are well suited to problems where the state space is high-dimensional, due to their frequent ability to effectively learn to detect and exploit patterns in complex domains. The use of neural networks to compress a high-dimensional state space into a compact representation that is used to learn a control policy by a reinforcement learning algorithm is also referred to as *deep reinforcement learning*.

We begin with the neuroevolution direct policy search method, followed by a discussion of two value-based methods and comments on regularization in neural network based reinforcement learning.

3.2 Neuroevolution

As discussed in Section 1, neural networks can be applied to reinforcement learning as a policy search method, by representing the controller as a neural network and optimizing the parameters of the controller using a genetic algorithm.

3.3 Neural Fitted Q Iteration

In *batch reinforcement learning* [22, Chapter 2], a sequence of samples is collected from the environment in order to perform batch learning. Using a neural network function approximator with batch learning led to the *Neural Fitted Q-Iteration* (NFQ) algorithm, which is described in detail in [29] and is an extension of *experience replay* methods [30]. A practical guide to applying the algorithm is provided in [31].

3.4 Deep Q-Networks

While NFQ is a batch reinforcement learning method, it is also possible to construct an online algorithm for Q-learning using neural networks. Furthermore, a convolutional neural network for compressing high-dimensional input images can be integrated into the overall network that approximates the action-value function. These are the main extensions used in the *Deep Q-Network* [32] [33] (DQN) algorithm. Furthermore, this algorithm introduces the notion of a separate *target network* that increases the stability of online reinforcement learning with neural networks by reducing correlations between updates to the parameters and the targets used to calculate those updates.

3.5 Regularization

Regularization in machine learning consists of any method which aims to decrease the generalization error of a model without decreasing the training error [4, Chapter 5].

Regularization techniques have been studied extensively in machine learning in general, but have received less attention within the reinforcement learning literature. Recent work includes [34] and [35]. An excellent overview of the generalization ability of neural network models grounded in Solomonoff's algorithmic probability [36] and Kolmogorov complexity [37] is presented in [38]. Dropout [39] has been found to be a simple and effective regularization method for neural networks.

In [40], it is mentioned that combining regularization with NFQ can be problematic when there is a minimal amount of training data, as it can cause samples with low frequency to be regularized away despite carrying valuable information.

Regularization is also applied in *autoencoder* [4, Chapter 14] neural network models, in particular in *denoising autoencoders* and *sparse autoencoders*, which can be used to compress the input to reinforcement learning algorithms.

The original NFQ and DQN algorithms do not incorporate regularization into their neural network models. In Chapter 8, we consider the effect of adding dropout regularization to an algorithm derived from these techniques.

Chapter 4

Algorithms

4.1 Introduction

In this chapter, we present the NEUROEVOLUTION algorithm for gradient-free training of neural networks, along with an algorithm that utilizes NEUROEVOLUTION for direct policy search called POLICY-SEARCH-NEUROEVOLUTION. We will apply these algorithms to a deep reinforcement learning experiment in Chapter 7.

Finally, we introduce an algorithm that combines ideas from the Neural Fitted Q-Iteration and Deep Q-Network algorithms and adds dropout regularization, resulting in a novel variation called the *Regularized Convolutional Neural Fitted Q-Iteration* (RC-NFQ) algorithm. We will apply this algorithm to a deep reinforcement learning experiment in Chapter 8.

4.2 Direct Policy Search using Neuroevolution

The procedure for NEUROEVOLUTION is listed in Algorithm 1. The procedure takes as arguments a parameter vector θ corresponding to a parameterization of a particular neural network architecture chosen for the application, an objective function J and a set of hyperparameters. The hyperparameters are described in Table 4.1. They influence the distribution of the initial population and the behavior of the selection, crossover and mutation operations during evolution.

In the initial step of the algorithm, a population of individual candidates of size α_0 is constructed which have the shape of θ and are drawn from a Gaussian distribution with mean μ_0 and variance σ_0. Within the main loop of the algorithm, several functions are called which perform the operations of the genetic algorithm, each of which we describe next.

First, the APPLY-TOURNAMENT-SELECTION function applies the tournament selection algorithm [41] to select a batch of individuals from P which will pass to the next step of the algorithm. It conducts α_0 tournaments, where each tournament compares the fitness of α_1 individuals sampled randomly with replacement from P and chooses the individual with the highest fitness. Subsequently, with probability α_2 for each consecutive pair of individuals which were selected, the APPLY-TWO-POINT-CROSSOVER function selects a starting and ending position and swaps the parameters in the corresponding portion of the parameter vector for the first individual with the corresponding portion of the parameter vector for the second individual. Next, in the APPLY-GAUSSIAN-MUTATION function, each entry in each individual parameter vector is replaced with probability α_3 with a number drawn from a Gaussian distribution with mean μ_1 and variance σ_1, resulting in the individuals for the next generation of the population. The fitness of these individuals is then evaluated in parallel by EVALUATE-FITNESS-PARALLEL, which evaluates each individual using the objective function J, potentially across multiple processors simultaneously.

Building on NEUROEVOLUTION, POLICY-SEARCH-NEUROEVOLUTION is presented in Algorithm 2. This procedure uses NEUROEVOLUTION to directly search the space of control policies in an environment E, where the control policy R is a parameterized

recurrent neural network, and the inputs to the control policy come from a previously trained convolutional neural network C which serves as a function approximator. The objective function J used for fitness evaluation consists of evaluating the effectiveness of the current policy R by evaluating it in the environment E using the fixed compressor C. The process of evaluating intermediate policies in the environment is referred to as *policy evaluation*.

Algorithm 1 Optimize the parameters of a neural network using a genetic algorithm

Input:

J is an objective function

θ is a parameter vector for the neural network which will be optimized

$hyperparameters$, described in Table 4.1

Output:

Returns the parameters θ for the neural network from the best performing candidate

procedure NEUROEVOLUTION($\theta, J, hyperparameters$)
    $\mu, \sigma, \alpha \leftarrow hyperparameters$
    Initialize population P consisting of individuals with a shape of θ drawn from a multi-dimensional gaussian distribution with parameters $\mu_0, \sigma_0, \alpha_0$
    while within computational budget **do**
        $P \leftarrow$ APPLY-TOURNAMENT-SELECTION(P, α_1)
        $P \leftarrow$ APPLY-TWO-POINT-CROSSOVER(P, α_2)
        $P \leftarrow$ APPLY-GAUSSIAN-MUTATION($P, \mu_1, \sigma_1, \alpha_3$)
        $J_i \leftarrow$ EVALUATE-FITNESS-PARALLEL(J, P)
    end while
    Assign the parameters from the fittest individual P_{best} to θ
    return θ
end procedure

Table 4.1: Hyperparameters for the NEUROEVOLUTION algorithm

Hyperparameter	**Applies to**	**Purpose**
μ_0, σ_0	initial population	mean and variance for each entry
α_0	initial population	population size
α_1	tournament selection	tournament size
α_2	two-point crossover	crossover probability
μ_1	gaussian mutation	mutation mean
σ_1	gaussian mutation	mutation variance
α_3	gaussian mutation	mutation probability

Algorithm 2 Find the weights for a recurrent neural network that correspond to an optimal policy

Input:

E is an environment, which specifies the space of states, actions and rewards
C is a trained convolutional neural network used for compression in environment E
R is an un-trained recurrent neural network used for control in environment E
hyperparameters, identical to those described in Table 4.1

Output:

Returns a trained recurrent neural network R that defines a policy under E and C

1: **procedure** POLICY-SEARCH-NEUROEVOLUTION($E, C, R, hyperparameters$)
2: Let θ denote the parameter vector associated with R
3: Let J denote a policy evaluation function which evaluates a controller R which is being learned with respect to a fixed compressor C and an environment E
4: $\theta^* \leftarrow$ NEUROEVOLUTION($\theta, J, hyperparameters$)
5: Use θ^* to update the parameters of R
6: **return** R
7: **end procedure**

4.3 Convolutional Neural Fitted Q Iteration

The *Regularized Convolutional Neural Fitted Q Iteration* (RC-NFQ) procedure is illustrated in Algorithm 3. The hyperparameters for the algorithm are described in Table 4.2. In addition to the hyperparameters, a suitable choice of architecture for the convolutional neural network that will serve both as a function approximator and to learn the action-value function must be selected. An example of such an architecture is presented in Chapter 8.

Algorithm 3 Estimate the action-value function using a convolutional neural network with dropout regularization

Input:

E is an environment, which specifies the space of states, actions and rewards

C is an architecture for a convolutional neural network with dropout regularization layers $hyperparameters$, described in Table 4.2

Output:

Returns a learned action-value function Q

procedure RC-NFQ($E, C, hyperparameters$)
    Parameterize identical convolutional neural network models Q_0 and $\hat{Q}_0$ which will be used to learn to approximate the action-value function, using the architecture specified in C and the dropout probability α_{drop} for the dropout regularization layers
    Initialize action-value function Q_0 with a parameter vector θ of random initial weights
    Initialize target action-value function $\hat{Q}_0$ with $\theta^- = \theta$
    Initialize experience replay buffer D
    for episode $i = 0, \alpha_{eps}$ **do**
        Initialize temporary experience buffer $\tilde{D}$
        for $t = 1, \alpha_{len}$ **do**
            Select random action with probability ϵ
            Otherwise, select action a that maximizes the action-value function $Q_i(s, a)$
            Execute action a in environment E and observe r and s'
            Store transition (s, a, r, s') in $\tilde{D}$
        end for
        Append $\tilde{D}$ to the experience replay buffer D
        for iteration $k = 0, \alpha_{iters}$ **do**
            Sample random batch of $\alpha_{samples}$ from D and store in D'
            Generate a pattern set of training targets where $y_i = D'^r + \gamma \hat{Q}_i(D'^{s'}, D'^a)$
            Call RMSprop with learning rate α_{lr} to perform gradient descent on $\left(y_i - Q_i\left(D'^s, D'^a; \theta\right)\right)^2$ and store the updated parameters in Q_{i+1}
            if k is a multiple of α_{freq} **then**
                Update target action-value function $\hat{Q}_{i+1}$ with the parameters from Q_i
            else
                Copy the parameters from $\hat{Q}_i$ to $\hat{Q}_{i+1}$
            end if
        end for
    end for
    return $Q_{\alpha_{eps}}$
end procedure

Table 4.2: Hyperparameters for the RC-NFQ algorithm

Hyperparameter	**Description**
γ	discount factor for future rewards
α_{lr}	learning rate for RMSprop
α_{freq}	frequency at which the target Q-network is updated
α_{iters}	number of iterations of fitted Q-iteration to run between episodes
α_{len}	length of each episode
α_{eps}	number of episodes
$\alpha_{samples}$	number of samples to use within each iteration of fitted Q-iteration
α_{drop}	dropout probability for the dropout regularization layers

Chapter 5

A Simulated Race Car Environment

5.1 Summary

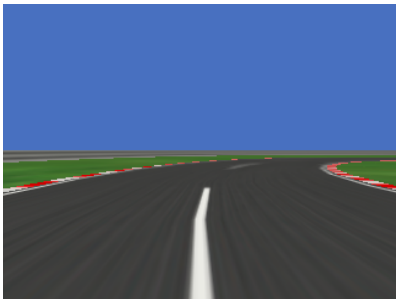
We implement a customized version of the *TORCS* race-car simulation software [42] as a simulation environment suitable for deep reinforcement learning. It implements a physics simulator and game engine for race cars on customizable race tracks. In the past, it has been used for many artificial intelligence competitions [43] based on high-level features such as car velocity and rangefinder measurements. We implement modifications to the software in order to allow the raw high-dimensional pixel data from the driver perspective to be used for control. Prior work using *TORCS* as a testbed for high-dimensional control from pixels started in 2013, from two research groups: *IDSIA* [44, 45, 46] and *Google DeepMind* [47, 48].

5.2 Environment Description

The environment consists of a race track with a user-definable layout and surroundings. The learning task is to learn a control policy using the raw high-dimensional pixel input from the driver's perspective, shown in Figure 5.1a, as input. We apply an initial dimensionality reduction step, also used in [46], in order to reduce the computational cost for the learning algorithms. This extracts the luminance channel, producing a grayscale image, and downsamples the image to 64x64 pixels, as shown in Figure 5.1b.

We recreate the custom track configurations used in [46], shown in Figure 5.2. The tracks are configured to be mirror images of each other; they are identical in every way except that all left and right turns are exchanged with each other with respect to the starting line. The purpose of constructing two training tracks in this manner is to discourage systems from memorizing the configuration of their training track without learning to generalize and take advantage of the input state for their control policy.

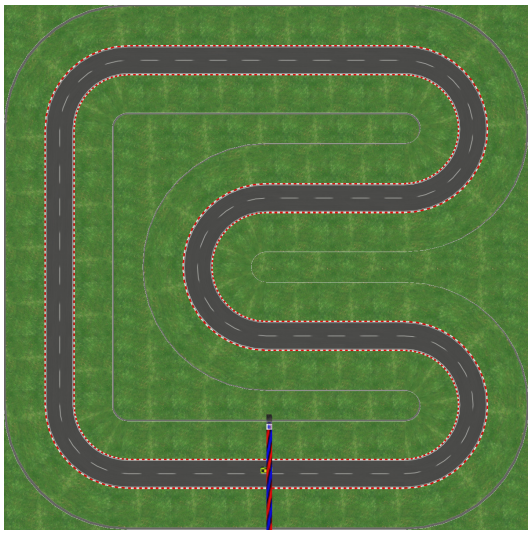
The controller sends an action to the environment which consists of a real-valued steering angle and a real-valued acceleration quantity. Hence the problem consists of learning a control policy π that maps a continuous state space $\mathcal{S} \in \mathbb{R}^{64\times64}$ to a continuous action space $\mathcal{A} \in \mathbb{R}^2$ in order to maximize a scalar reward signal $\mathcal{R}$.



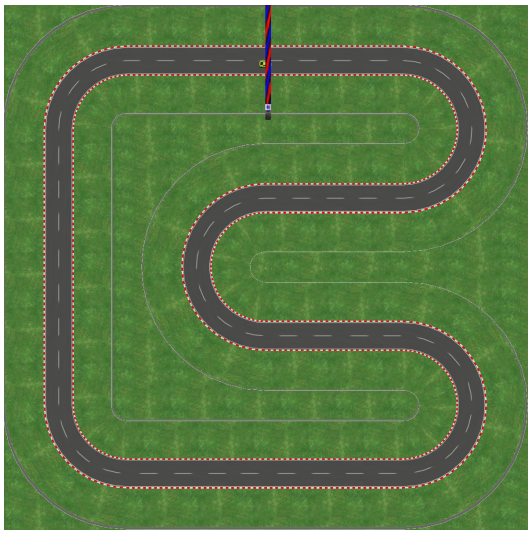
(a) Before downsampling

(b) Grayscale, after downsampling

Figure 5.1: Sample input images from the *TORCS* race-car simulator.



(a) Initial track



(b) Mirror image of initial track

Figure 5.2: Custom race tracks used for training.

5.3 Modifications to the simulator

The default *TORCS* simulator [42] with the competition extensions [43] does not support using the raw high-dimensional pixel data for control, and the modified implementations used in prior research are not publicly available.

As a result, we implemented several significant modifications to the *TORCS* codebase to support the experiment.

5.3.1 Features introduced

- Stream the raw pixel video frames from the agent using the *ZeroMQ* [49] messaging framework
- Allow multiple racing servers to run in parallel
- Allow the simulator to be started and terminated automatically from Python
- Disable the "Ready-Set-Go" start screen message
- Start the car exactly at the starting line
- Allow the race track to be chosen programmatically
- Allow rendering when the window is not in focus

Chapter 6

A Robotic Car Testbed

6.1 Summary

We also designed our own testbed for deep reinforcement learning experiments applied to robotics. The testbed consists of a wheeled robot on a racetrack, equipped with a camera and an on-board computer connected to a server over a wireless network for control. In this section, we will describe the architecture of the robot and its accompanying vision system, the environment, and the server, along with a formal description in the reinforcement learning setting.

6.2 Robot Description

The robot is based on a customized version of the *LEGO Mindstorms EV3* platform. It is equipped with four large servo motors, attached to each wheel. It has a custom firmware running the *ev3dev* programming interface. A color sensor is mounted on the bottom, and a bumper sensor is mounted to the front, which are used for calculating the reward signal, but are not used for control. It also has a *Raspberry Pi Camera* and a *Raspberry Pi* on-board computer which are used to capture the raw video inputs used for control. Both of the on-board computers are connected via a wireless network to the server. Video is streamed using the *ZeroMQ* messaging framework.

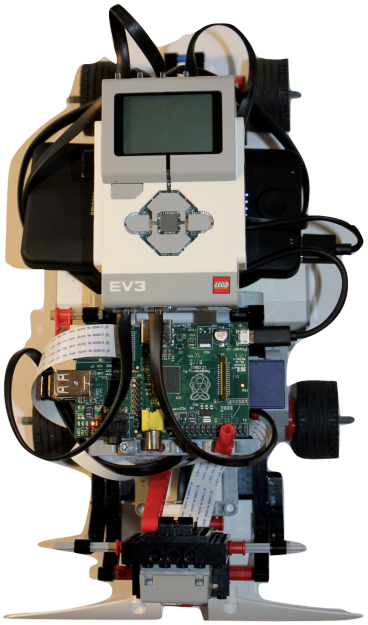
6.3 Server Description

The server implements the learning algorithms and is connected to the robot by a wireless network.

6.4 Environment Description

The environment consists of a 48-inch by 72-inch white surface, with surrounding walls and inner walls, which defines a racetrack shape. The racetrack surface has a thick middle stripe, and on the inner side has a thin stripe and a series of angled markers.

A series of yellow and red marker lines are placed at regular intervals along the racetrack. These marker lines are detected by the on-board color sensor in order to compute a reward signal that indicates progress along the racetrack.



(a) The robot, a customized version of the *LEGO Mindstorms EV3* platform.

(b) The robot race track environment.

Figure 6.1: The robot and its environment.

6.5 Reward Signal

A finite state machine monitors detection of the yellow and red marker lines. If they are detected in the order "yellow, red" then a positive reward of $+5$ is administered. If they are detected in the order "red, yellow" then a negative reward of -0.1 is administered.

The front bumper sensor administers a negative reward of -2 when triggered. Driving in reverse is penalized by a negative reward of -1 per timestep.

6.6 Control Loop

The robot runs a continuous control loop. Within this loop, time is discretized by incrementing the time step every 5 iterations, at which point a reward and a new state are generated. Between time steps, the previous action is maintained. The reward and new state are passed to the server, which returns an action. Hence, the interaction between the robot and the server can be defined in terms of (s, a, r, s') tuples.

6.7 State Definitions

The state is defined by the high-dimensional visual input from the robot camera. Video is captured on the robot at 30 frames per second with 64x64 pixel resolution. When it is received by the server, it is converted to grayscale and sampled once per time step, which is used as the state representation. Hence, the robot operates in a continuous state space of $\mathcal{S} \in \mathbb{R}^{64\times64}$.

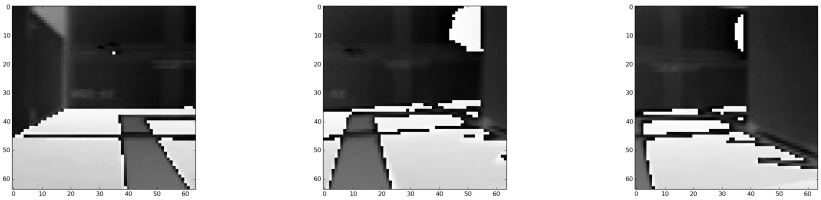


Figure 6.2: View from the robot camera as it turns right.

6.8 Action Definitions

The action space is defined as a discrete mapping to robot motor torques from the action set $a \in \{forward, backward, left, right\}$. Hence, the robot has a discrete action space with $n = 4$ possible actions.

Chapter 7

Experiment: Direct Policy Search using Neuroevolution

7.1 Summary

We apply POLICY-SEARCH-NEUROEVOLUTION as described in Algorithm 2 to the *TORCS* simulated race car environment described in Chapter 5. We reproduce parts of the experiment presented in [46] and analyze the learning problem and methods presented in that study.

The convolutional neural network and the recurrent neural network were implemented using the *Keras* [50] deep learning library. The evolutionary algorithms were implemented using the *DEAP* [51] framework for distributed evolutionary algorithms.

7.2 Convolutional Neural Network

We use a convolutional neural network to compress the high-dimensional input signal into a low-dimensional representation that will be fed into the recurrent neural network, which will then learn a control policy based on that low-dimensional signal.

We reproduce the architecture described in [46]. The architecture is pictured in Figure 7.1, and the detailed configuration of each layer is specified in Table 7.1.

In order to generate a training set, we used a built-in car controller to drive the car around each of the two tracks while capturing images. We then selected 20 images from each of the two tracks that were representative of different viewpoints and features that were encountered, in order to form a training set of 40 images.

We then optimized the parameters of the network using neuroevolution, described in Algorithm 1. As illustrated in Table 7.1, the network has a total of 993 parameters. The objective function was designed to maximize the ability of the network to discriminate between the training images.

Let $\phi(\mathcal{D}_i)$ denote the output feature vector computed by the convolutional neural network given an input image $\mathcal{D}_i$. We maximized the sums of the average and minimum pairwise

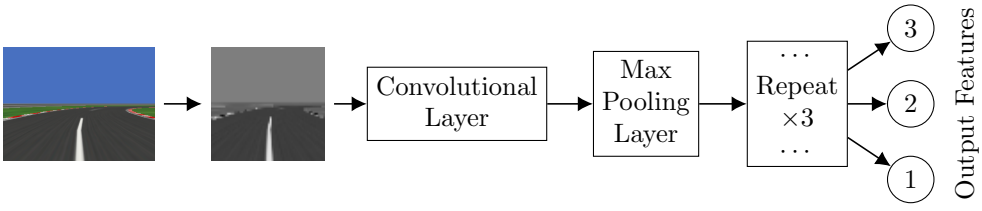


Figure 7.1: Architecture of the convolutional neural network.

Table 7.1: Convolutional neural network configuration

Layer	Input	Output	Filter Width	Filters	Stride	Parameters
Convolutional 1	(1, 64, 64)	(10, 63, 63)	2	10	(1, 1)	50
ReLU 1	(10, 63, 63)	(10, 63, 63)	-	-	-	-
Max Pooling 1	(10, 63, 63)	(10, 21, 21)	3	-	-	-
Convolutional 2	(10, 21, 21)	(10, 20, 20)	2	10	(1, 1)	410
ReLU 2	(10, 20, 20)	(10, 20, 20)	-	-	-	-
Max Pooling 2	(10, 20, 20)	(10, 10, 10)	2	-	-	-
Convolutional 3	(10, 10, 10)	(10, 9, 9)	2	10	(1, 1)	410
ReLU 3	(10, 9, 9)	(10, 9, 9)	-	-	-	-
Max Pooling 3	(10, 9, 9)	(10, 3, 3)	3	-	-	-
Convolutional 4	(10, 3, 3)	(3, 2, 2)	2	3	(1, 1)	123
ReLU 4	(3, 2, 2)	(3, 2, 2)	-	-	-	-
Max Pooling 4	(3, 2, 2)	(3, 1, 1)	2	-	-	-

euclidean distances between output features ϕ_i and $\phi_j, \forall(\mathcal{D}_i, \mathcal{D}_j)|\mathcal{D}_i \neq \mathcal{D}_j$ computed from the set of input images $\mathcal{D}$. Hence, the optimization problem was to find the parameters θ for the convolutional neural network that satisfied

$$\max_{\theta} \left(\frac{1}{|\mathcal{D}|} \sum_{(\phi_i, \phi_j)} \|\phi_i - \phi_j\| + \min_{(\phi_i, \phi_j)} \|\phi_i - \phi_j\| \right).$$

7.3 Recurrent Neural Network

The recurrent neural network used as a controller is a simple RNN, also known as an Elman network [52]. It receives an input vector of length 3 from the output of the convolutional neural network. The input passes through a hidden layer with 3 nodes, which is connected to an output layer which also consists of 3 nodes. The total number of trainable parameters is 33.

7.4 Control Signal

The *TORCS* simulator runs at a frequency of one time step per 0.022 seconds. In the experiments, we utilize a control frequency of 5 Hz; between control signals, the previous action is repeated.

The purpose of the RNN is to learn a control policy to drive the car. Since the car accepts an action vector $\mathcal{A} \in \mathbb{R}^2$, it is necessary to transform the output of the RNN to produce a suitable control signal. This is accomplished as in [46] by generating the steering signal from the mean of the first and second output nodes, and the acceleration signal from the third output node. The steering signal is clipped to keep it within the range $[-1, 1]$ and the acceleration signal is clipped to keep it within the range $[0, 1]$. We note that it may be feasible to slightly modify the RNN architecture that was used so that only two output nodes are needed, with one of them directly mapped to the acceleration signal. However, we chose to utilize the same architecture as [46] for comparison purposes.

The difficulty of the learning problem was also reduced by imposing a velocity limit on the car of 10 kilometers per hour. We note that [48] also studied a "slow car" setting, in addition to a "fast car" setting.

7.5 Objective Function

The fitness was calculated as in [46]. The agent was rewarded for traveling a large distance along the race track in the correct direction. In addition, it was rewarded slightly for increasing its

maximum velocity, and was penalized slightly for collisions and for high variance in its control signals. The objective function used was

$$distance - 0.003 \cdot damage + 0.2 \cdot max\_velocity - 100 \cdot c$$

where *distance* corresponds to the distance traveled along the race track, *damage* corresponds to the collision signals generated by the simulator if the car collides with the walls, *max_velocity* corresponds to the maximum velocity attained during a trial, and c corresponds to the cumulative differences in successive control signals.

Each fitness evaluation consisted of evaluating the candidate controller on the two test tracks described in Section 5.2 for 500 time steps, or approximately 11 seconds of simulated time. The fitness of each of the two races was recorded, and the minimum fitness was reported as the fitness of the candidate. By requiring the candidate to perform well on both the first track and its mirror image, the training procedure aims to discourage the learning algorithm from simply memorizing the training track without using input images to discriminate between track conditions.

7.6 Results

7.6.1 Convolutional Neural Network

We trained the convolutional neural network using the NEUROEVOLUTION algorithm. We used a population size of 100, a crossover probability of 0.5 with two-point crossover, and tournament selection with a tournament size of 10. We compared the performance over different choices of the mutation probability hyperparameter from the following set:

$$\{0.05, 0.10, 0.20, 0.30, 0.40, 0.50\}.$$

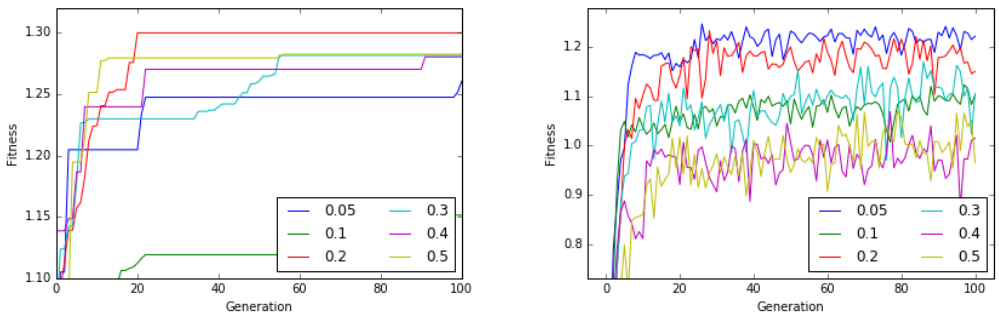
Mutations were applied with a probability of 0.05 to each element of individuals chosen for mutation from a gaussian distribution with mean 0 and standard deviation 1.5.

For each choice of mutation probability, the maximum fitness achieved per generation is illustrated in Figure 7.2a, and the mean fitness achieved is illustrated in Figure 7.2b. The maximum fitness was 1.30, achieved with a mutation probability of 0.20.

The evolution of the feature vectors corresponding to this maximum fitness is presented in Figure 7.3. The initial features are clustered together and progressively become better separated as the number of generations increases.

7.6.2 Recurrent Neural Network

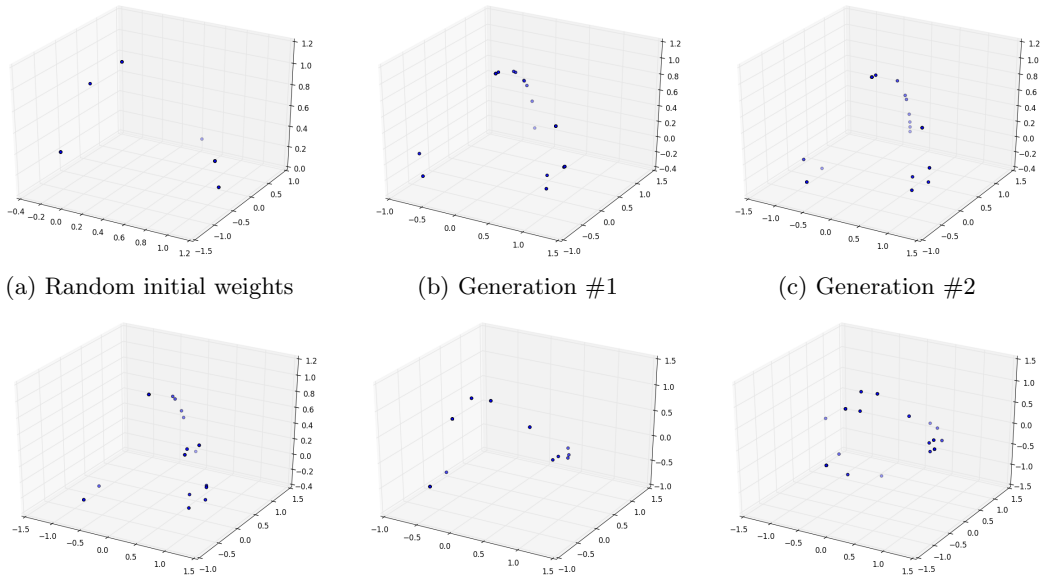
We trained the recurrent neural network controller in the *TORCS* environment using the POLICY-SEARCH-NEUROEVOLUTION algorithm. We used a population size of 96, a crossover probability of 0.5 with two-point crossover, and tournament selection.



(a) Maximum fitness

(b) Mean fitness

Figure 7.2: Fitness per generation by mutation probability.



(a) Random initial weights (b) Generation #1 (c) Generation #2

(d) Generation #4 (e) Generation #16 (f) Generation #100

Figure 7.3: Feature vectors produced as evolution advances and higher fitness parameter settings for the convolutional neural network are found, better separating the input images in feature space.

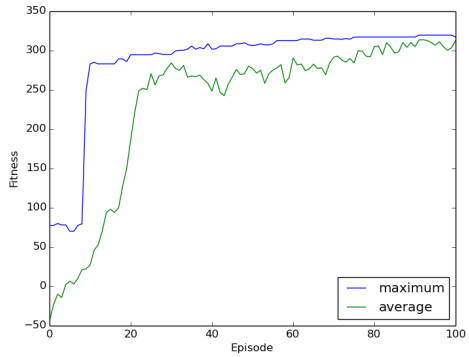
Training was conducted for 100 generations for each choice of hyperparameters. By parallelizing the fitness evaluation function using the extensions to the *TORCS* framework that we implemented, we were able to evaluate 12 candidate solutions simultaneously across each of the CPU cores on the server. Each experiment ran for approximately 3 hours.

We compared performance across two different hyperparameters: the mutation probability was chosen from the set $\{0.02, 0.20\}$, and the tournament size was chosen from the set $\{2, 10\}$. The results are shown in Figure 7.4.

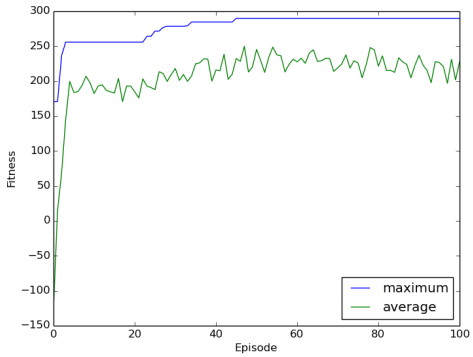
The best solution found had a fitness of 330.7, with a mutation probability of 0.20 and a tournament size of 10. The best solution was found after 39 generations, although a solution that was nearly as good with fitness of 330.1 was found much earlier at generation 12.

The second best solution found had a fitness of 319.7, with a mutation probability of 0.02 and a tournament size of 2. By generation 10, it had reached a fitness of 283.1. After that point, there were a large number of incremental improvements in maximum fitness, more so than in any other trial. It is also notable that the average fitness gradually began to approach the maximum fitness over time.

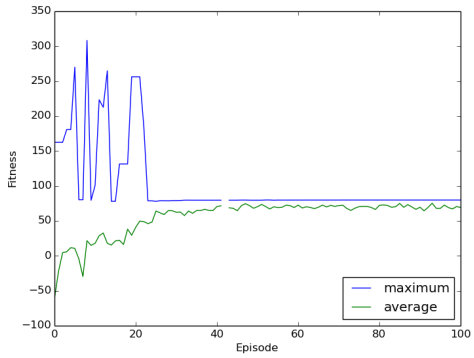
With a mutation probability of 0.02 and a tournament size of 10, a slightly lower maximum fitness was achieved, and the average fitness did not approach the maximum fitness. The worst result came from a mutation probability of 0.20 and a tournament size of 2; in that case, although a solution with a fitness of 308.3 was found early at generation 8, the population rapidly degraded to hugely suboptimal solutions, falling to a maximum fitness of 78.7 at generation 23, and never increasing again.



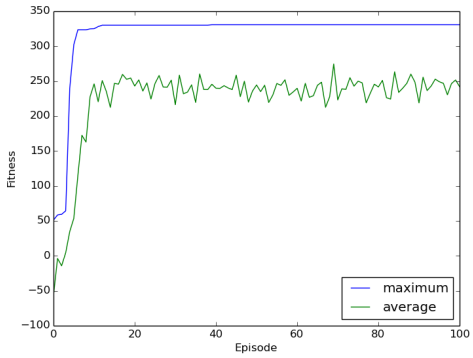
(a) Mutation probability of 0.02, tournament size of 2. Maximum fitness of 319.7 at generation 91.



(b) Mutation probability of 0.02, tournament size of 10. Maximum fitness of 290.0 at generation 45.



(c) Mutation probability of 0.20, tournament size of 2. Maximum fitness of 308.3 at generation 8. Maximum fitness degrades afterwards.



(d) Mutation probability of 0.20, tournament size of 10. Maximum fitness of 330.7 at generation 39.

Figure 7.4: Comparison of different hyperparameter choices for evolving the recurrent neural network controller.

7.7 Analysis

The training procedure produced a control policy that was able to successfully navigate the car through the turns encountered within the tested time duration on the two tracks.

The NEUROEVOLUTION training procedure and objective function of maximizing discrimination between training images for the convolutional neural network was very simple, but yielded features that were sufficient for effectively learning a control policy.

It is notable that most of the different choices of mutation probability when evolving the convolutional neural network rapidly arrived at a solution near the best solution found. We can also observe that, for higher mutation probabilities, the mean fitness of the population tends to remain lower.

In training the recurrent neural network, the high mutation probability in the configuration illustrated in Figure 7.4c is likely the reason that the average fitness remained significantly below the maximum fitness for all of the remaining episodes in that trial.

Overall, we found that the POLICY-SEARCH-NEUROEVOLUTION algorithm was fairly robust to the different choices of hyperparameters that we tested, with 3 out of the 4 choices producing reasonably good results.

In all the cases, the algorithm was able to find a fairly good solution with a fitness above 250 within 10 generations. The most promising choice of hyperparameters appears to be a mutation probability of 0.02 and a tournament size of 2. This corresponded to a maximum fitness curve that rapidly reached a good level of performance, and then consistently found

incremental improvements over the duration of the experiment. Furthermore, the average fitness gradually converged towards the maximum fitness.

In that configuration, the mutation probability is somewhat low, which allows promising configurations to be more stable with less risk of changing due to mutation. The choice of a tournament size of 2 allows greater diversity to persist in the population, since each round of tournament selection will randomly choose two individuals to compete in the tournament, allowing a greater probability of survival for any particular individual. These two factors correspond to the *exploration versus exploitation* tradeoff mentioned in Section 2.

These results were achieved using a standard genetic algorithm with two-point crossover, while in [46] the more complex CoSyNE [53] algorithm was used.

Chapter 8

Experiment: Regularized Convolutional Neural Fitted Q Iteration

8.1 Summary

In this experiment, we apply the Regularized Convolutional Neural Fitted Q Iteration algorithm (RC-NFQ) to a robotic car that receives raw pixels from a video camera as input, described in detail in Chapter 6. The objective is to learn a driving policy in a racetrack environment. We also consider the effects of dropout regularization and the choice of learning rates on Q-value learning and demonstrate initial steps towards learning an effective control policy.

During training, we use an *epsilon-greedy* exploration strategy [20], in which the robot follows its current estimate of an optimal policy at each time step with probability $(1 - \epsilon)$ and selects a random action with probability ϵ. The value of ϵ was annealed over the four training episodes according to a schedule of $\epsilon \in (1.0, 0.5, 0.5, 0.2)$. Using epsilon-greedy exploration is meant to encourage the robot to explore the state space, and also has the effect of perturbing the robot if it gets stuck in a suboptimal state that it believes is optimal.

The neural network was implemented using the *Keras* [50] deep learning library. The modified fitted Q iteration algorithm was implemented from scratch. The server uses an *Intel i7-4930K* processor with 32GB of RAM and an *Nvidia Titan X* graphics processing unit with 12GB of RAM.

8.2 Q-network

We use a convolutional neural network to compress the input images from the video camera. The convolutional layers are similar to those used in [32] and [48] with several modifications. The state input consists of only one input image, and the action is also fed as input, encoded as a one-hot vector. We add dropout layers after the first and second convolutional layers, and after the first fully connected layer. The dropout probability used is $p = 0.25$. The details of the convolutional neural network are shown in Table 8.1, and the architecture of the Q-network is shown in Table 8.2.

The total number of parameters to be trained is 305713. We train the network using RMSprop.

Table 8.1: Convolutional neural network architecture

Layer	Input	Output	Filter Width	Filters	Stride	Parameters
Convolutional 1	(1, 64, 64)	(16, 15, 15)	8	16	(4, 4)	1040
Activation 1	(16, 15, 15)	(16, 15, 15)	-	-	-	-
Dropout 1	(16, 15, 15)	(16, 15, 15)	-	-	-	-
Convolutional 2	(16, 15, 15)	(32, 6, 6)	4	32	(2, 2)	8224
Activation 2	(32, 6, 6)	(32, 6, 6)	-	-	-	-
Dropout 2	(32, 6, 6)	(32, 6, 6)	-	-	-	-
Flatten	(32, 6, 6)	(1152)	-	-	-	-

Table 8.2: Q-network architecture

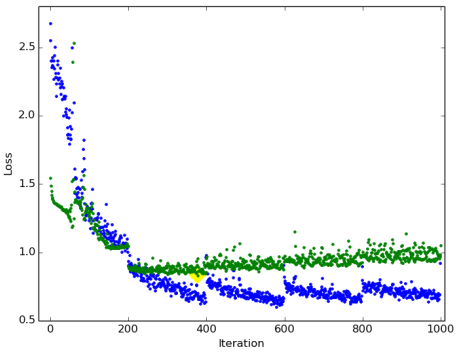
Layer	Input	Output	Parameters
ConvNet	(1, 64, 64)	(1152)	9264
Action Input	(4)	(4)	-
Merge	(1152) and (4)	(1156)	-
Dense 1	(1156)	(256)	296192
Activation	(256)	(256)	-
Dropout	(256)	(256)	-
Dense 2	(256)	(1)	257

8.3 Results

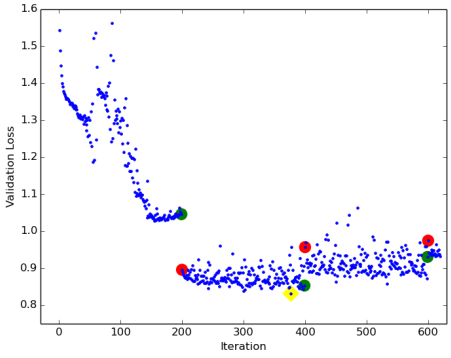
8.3.1 Training

We divide the data collection on the robot into four episodes. The data from each episode forms a batch for training. Each batch consists of 5000 time steps, and the data is split into 90% training and 10% validation sets.

After the first episode, we run the NFQ algorithm for 1000 iterations, with a learning rate of $\eta = 10^{-3}$ using full-batch RMSprop. The results are illustrated in Figure 8.1 with the best validation loss highlighted. The results of subsequent training batches are discussed in the next section along with an analysis of various hyperparameters.



(a) Training loss marked in blue; validation loss marked in green; lowest validation loss highlighted in yellow. The target network was updated every 200 iterations.



(b) Enlarged region showing the validation loss only. The last iterations before target network updates are marked in green, and the first iterations after the updates are marked in red. Lowest validation loss highlighted in yellow.

Figure 8.1: Illustration of NFQ training and validation loss between target network updates, during the first batch of training.

8.3.2 Comparing gradient descent hyperparameters

For the second batch of fitted Q iteration, we begin by running a preliminary batch in which we compare the effect of different learning rates η and the choice of mini-batch versus full-batch training on the learning curves for the RMSprop optimizer. Fitted Q iteration is performed with a separate target network updated every 200 epochs. The results are illustrated in Figure 8.2.

As shown in Table 8.3, the best validation loss achieved was 1.00, and the range of best validation losses was between 1.00 and 1.07. Hence, the choice of parameters did not have a very large effect on the ability to find a low validation loss at some point during training. However, if we compare the learning curves in Figure 8.2 by examining the validation set loss over time, we will note significant differences between the hyperparameter choices.

Table 8.3: Best validation loss compared for different RMSprop hyperparameters after the preliminary run of the second training batch.

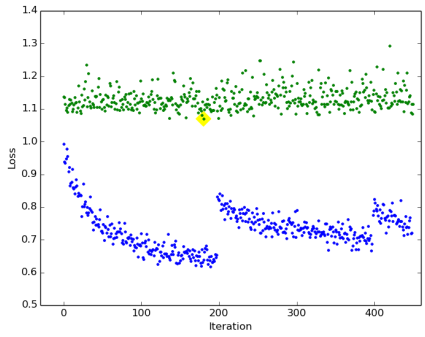
Learning Rate	**Full-batch**	**Mini-batch**
10^{-3}	1.07	1.06
2.5×10^{-4}	1.03	1.06
10^{-4}	1.03	1.05
2.5×10^{-5}	1.00	1.07

The smoothest validation set learning curve was obtained with an RMSprop learning rate of $\eta = 2.5 \times 10^{-5}$, which reached a validation loss of 1.07 at epoch number 396.

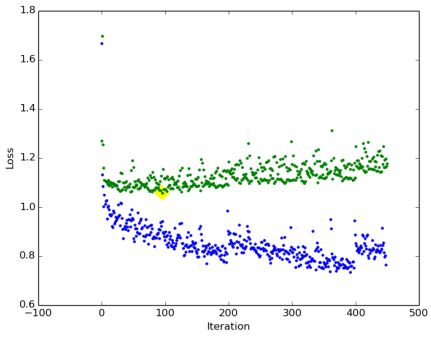
We then used the $\eta = 2.5 \times 10^{-5}$ learning rate and applied it using fitted Q iteration with the target network updated less frequently, every 400 epochs, and ran the optimizer for 1650 epochs in total. The batch was run by starting with the result of the first batch, in order to produce an updated result for the second batch. Mini-batch and full-batch updates are compared in Figure 8.3. For full-batch, the best validation loss was 1.05 at epoch 759. For mini-batch, the best validation loss was 1.02 at epoch 717.

We note that when using full-batch, the training loss is noisier, and the validation loss is significantly less noisy. In fact, the validation loss is nearly monotonically improving between each target network update.

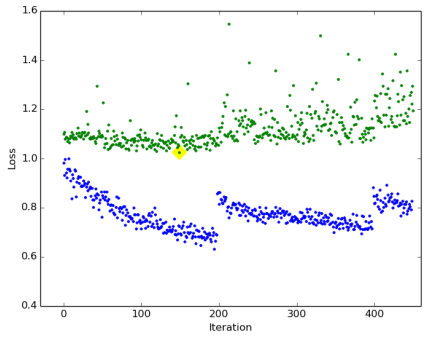
Although the mini-batch update achieved a validation loss of 1.02 that was slightly lower than full-batch at 1.05, we selected the full-batch result, due to the similar performance with far lower noise on the validation set.



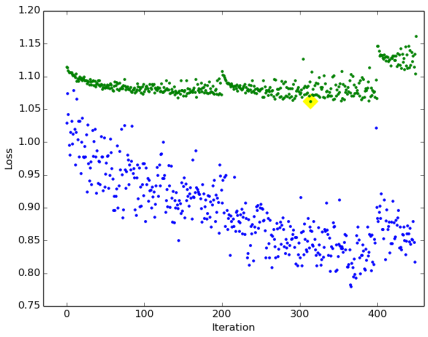
(a) mini-batch, $\eta = 10^{-3}$



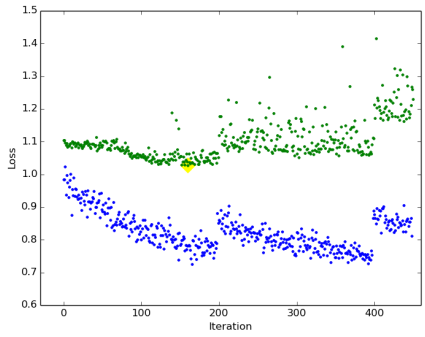
(b) full-batch, $\eta = 10^{-3}$



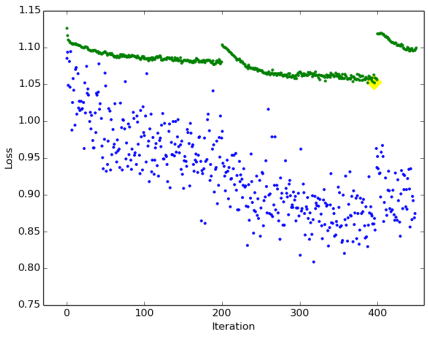
(c) mini-batch, $\eta = 2.5 \times 10^{-4}$



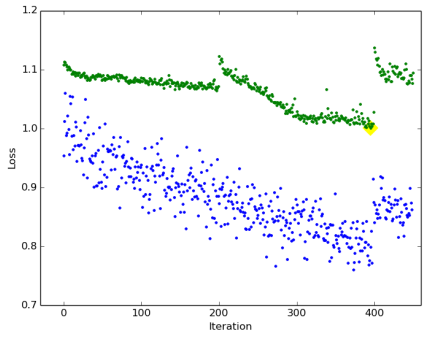
(d) full-batch, $\eta = 2.5 \times 10^{-4}$



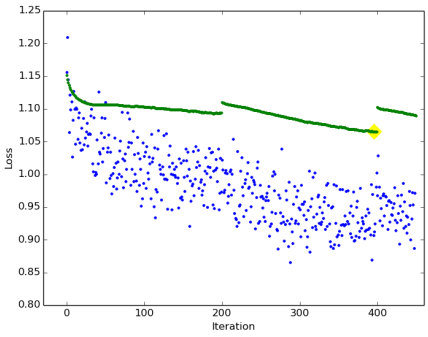
(e) mini-batch, $\eta = 10^{-4}$



(f) full-batch, $\eta = 10^{-4}$

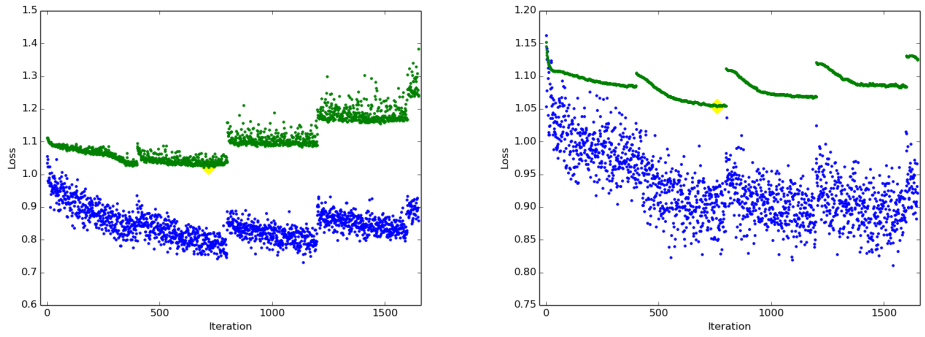


(g) mini-batch, $\eta = 2.5 \times 10^{-5}$



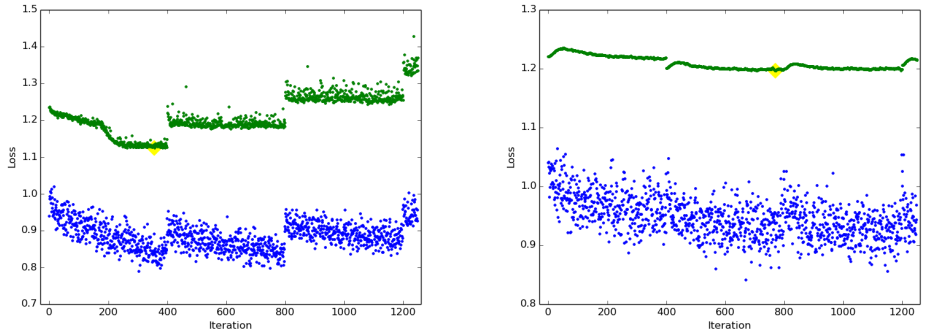
(h) full-batch, $\eta = 2.5 \times 10^{-5}$

Figure 8.2: Comparison of learning curves for different learning rates η and mini-batch versus full-batch using RMSprop for convolutional NFQ with dropout. Training loss marked in blue; validation loss marked in green; lowest validation loss highlighted in yellow. The target network was updated every 200 iterations.



(a) mini-batch (b) full-batch

Figure 8.3: Learning curves for $\eta = 2.5 \times 10^{-5}$ with 400 iterations between NFQ target network updates after episode 2. Training loss marked in blue; validation loss marked in green; lowest validation loss highlighted in yellow.



(a) mini-batch (b) full-batch

Figure 8.4: Learning curves for $\eta = 2.5 \times 10^{-5}$ with 400 iterations between NFQ target network updates after episode 3. Training loss marked in blue; validation loss marked in green; lowest validation loss highlighted in yellow.

For the third training batch, we used the same learning rate of $\eta = 2.5 \times 10^{-5}$ and compared mini-batch and full-batch training, as shown in Figure 8.4. The mini-batch method reached a significantly better solution with a fitness of 1.12 during the first group of fitted Q iteration at epoch 356, compared to a fitness of 1.20 found by full-batch training at epoch 770. We chose the parameters from this mini-batch training epoch for the subsequent episode.

8.3.3 Analysis of learning progress

The performance of the robot measured in terms of average reward and total reward is illustrated in Figure 8.5 according to the results listed in Table 8.4. We can observe a general trend towards increased performance, although the relatively small episode count did not result in convergence.

The number of times each action was taken is listed in Table 8.5, along with details on the predicted action-values within each episode. In the final episode, the robot usually preferred the single action of turning left. This means that the Q-value for that action tended to be higher than that for other actions, in the states which were evaluated. It is interesting to ask the question of whether the Q-network was simply learning an average Q-value without discriminating based on state, or learned to vary the Q-value, which could indicate that it developed an ability to discriminate between states in predicting Q-values. In order to gain insight in this regard, we analyzed how much the predicted Q-values varied between different

Table 8.4: Reward per episode

Episode	Average reward	Total reward
1	-0.476	-2381.5
2	-0.226	-1131.1
3	-0.234	-1171.7
4	-0.122	-609.3

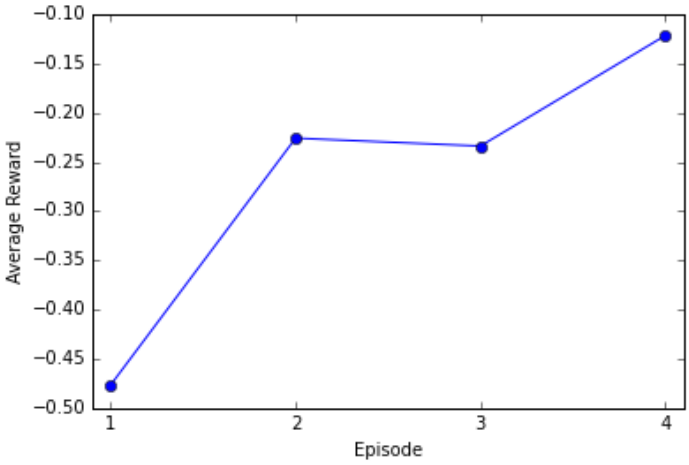
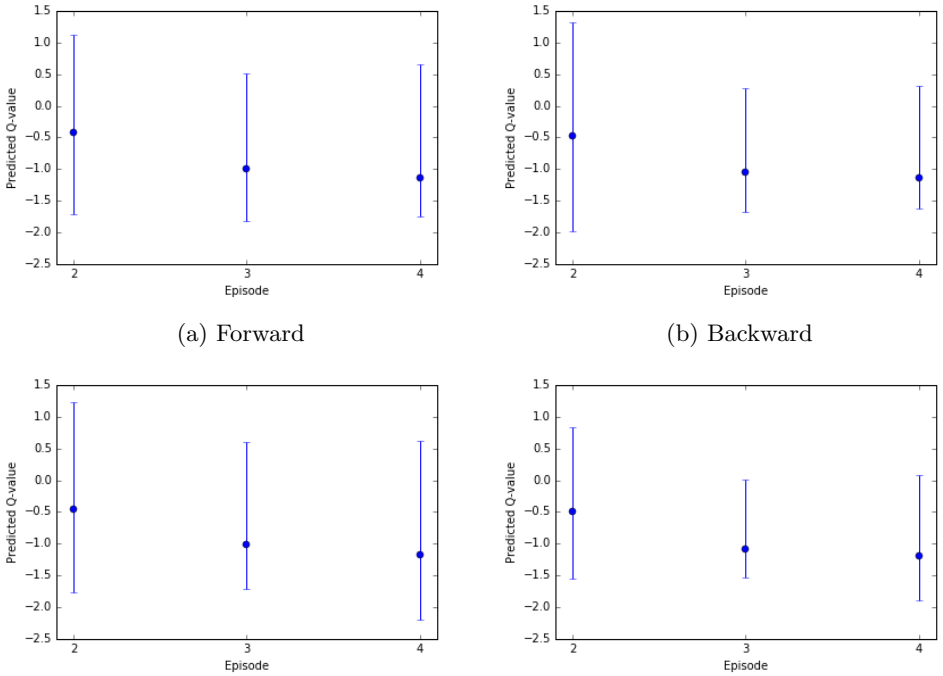


Figure 8.5: Average reward per episode



(a) Forward

(b) Backward

(c) Left

(d) Right

Figure 8.6: Predicted Q-values (mean, mininum and maximum) per action taken.

Table 8.5: Predicted action-values per episode

Episode	Action	Mean	Minimum	Maximum	Action Count
1	forward	-	-	-	1296
1	backward	-	-	-	1214
1	left	-	-	-	1214
1	right	-	-	-	1276
2	forward	-0.43	-1.72	1.12	629
2	backward	-0.47	-1.97	1.32	617
2	left	-0.46	-1.76	1.23	3109
2	right	-0.49	-1.55	0.84	645
3	forward	-1.00	-1.81	0.51	648
3	backward	-1.05	-1.67	0.27	661
3	left	-1.02	-1.72	0.61	3087
3	right	-1.09	-1.54	0.01	604
4	forward	-1.14	-1.75	0.65	286
4	backward	-1.13	-1.62	0.32	230
4	left	-1.18	-2.19	0.62	4231
4	right	-1.20	-1.89	0.09	253

states, across the episodes. The predicted Q-values from Table 8.5 are illustrated graphically in Figure 8.6. The first episode is not illustrated, since it consisted of randomly chosen actions with $\epsilon = 1.0$. We can see that the mean values and upper and lower limits of the predictions changed over time, and did not converge on a single narrow prediction range. This supports the hypothesis that the states did have predictive value for the Q-network.

The optimal strategy would involve learning to efficiently complete laps around the race-track in the correct direction, while avoiding collisions. This would require frequent use of the "forward" action, with slightly less frequent use of the "turn left" action. We can see in Table 8.5 that, in the final episode, although the "turn left" action was chosen most frequently, when the "forward" action was chosen, it had a higher mean and maximum predicted Q-value.

These results provide some evidence for learning progress but would benefit from a much higher episode count. The robotic platform as implemented required a lot of time to capture episode data, as a human was required to monitor the robot and repair it in frequent cases of mechanical failure. In future work, the robot platform should be made more robust to allow for more efficient data collection in order to investigate the learning performance over a longer timespan.

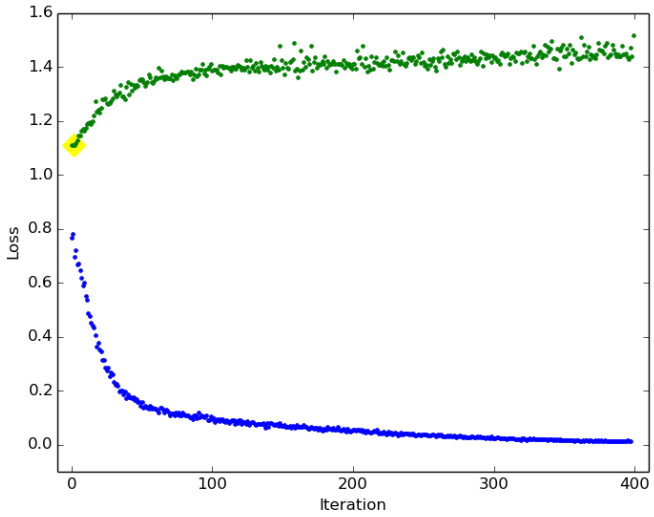


Figure 8.7: NFQ without dropout regularization, mini-batch RMSprop, $\eta = 2.5 \times 10^{-4}$. Training loss marked in blue; validation loss marked in green; lowest validation loss highlighted in yellow.

8.4 Analysis

For comparison purposes, we also include the result of running the NFQ algorithm on training data without using dropout regularization in Figure 8.7. The training loss steadily converges to zero, and the validation loss diverges immediately. This suggests that the procedure overfits to the training data. The learning curves encountered when dropout regularization is used do not display this pattern; instead, they demonstrate both the validation and training loss decreasing until they plateau at a certain level before updating the target network, as illustrated previously in Figures 8.1, 8.2, 8.3 and 8.4. This is evidence that dropout regularization may be a useful addition to neural fitted Q iteration with convolutional neural networks.

Mini-batch training produced a noiser learning curve but in general arrived at a better solution than full-batch gradient descent. The choice of learning rates for the RMSprop algorithm made a significant difference in the shape of the learning curves. The best validation loss was generally achieved within the first one or two target network updates, with subsequent updates producing higher validation losses.

The overall final validation loss was still significantly far above zero, suggesting that the network had not yet arrived near a correct approximation of the target Q-function. However, as discussed in the previous section, initial evidence of learning was observed.

Chapter 9

Future Directions

9.1 Direct Policy Search

For direct policy search with neuroevolution, it would be interesting to also compare Covariance Matrix Adaptation Evolution Strategy (CMA-ES) [54], and to compare in detail the performance of the simple genetic algorithm used here with CMA-ES and CoSyNE [53].

Another worthwhile extension would be to eliminate the velocity limit (while modifying the control frequency accordingly) and to test generalization ability on the entire track and on different testing tracks, as in [46].

The convolutional neural network training procedure could also be replaced with a procedure using *convolutional autoencoders* for unsupervised feature learning as in [55], and the resulting impact on controller performance could be studied.

It would also be informative to implement and study online end-to-end learning for the convolutional neural network and recurrent neural network simultaneously, as in [45].

Furthermore, an analysis of the relative performance of stochastic gradient algorithms such as REINFORCE [26] compared to neuroevolution for direct policy search would be an interesting research direction.

9.2 Value-based Methods

The most important subsequent investigation proposed is to improve the robustness and efficiency of the robot testbed to allow significantly more data to be collected to investigate the convergence of the RC-NFQ algorithm with additional iterations and a larger pool of training data.

An immediate extension of the convolutional neural network architecture with dropout layers as described would include the addition of an LSTM [56] recurrent neural network layer to allow the controller to take advantage of memory of prior states. In order to extend the integration of dropout regularization layers with the NFQ algorithm and convolutional neural networks to LSTM networks, the improved methods for effectively combining dropout with RNNs presented in [57] could be applied.

An alternative simpler way to add a minimal amount of memory would be to provide the network with several recent frames as an input tensor to the convolutional neural network, as in [48] [32] [33].

Extending the RC-NFQ algorithm to the online case, resulting in a variant of the Deep Q-Network (DQN) algorithm utilizing dropout regularization, would also be a possible next step. Modifying the Q-network further by adding batch normalization layers [58] and analyzing their effect on learning in both the NFQ and DQN settings could also be worthwhile.

In [59], more sophisticated techniques for sampling from an experience replay memory are analyzed, and could be tested in this setting as well. Additional variations would consist of varying the various hyperparameter values in the RC-NFQ algorithm.

If a suitably similar environment were constructed in simulation, then experiments could be conducted that took advantage of the faster timescales and parallelization possible in simulated learning, with an adaptation and transfer of the learned policies to the real robot.

Actor-critic [20, Chapter 6] methods and advantage learning [60] [61] [62] [63] would be another worthwhile direction in which to extend the RC-NFQ algorithm. Additionally, Monte-Carlo tree search methods have recently been combined with convolutional neural networks for control in *TORCS* [64] and *Go* [65] and present a promising research direction.

9.3 Exploration

Due to our observation that the robot tended to prefer one particular action after multiple rounds of training despite not yet finding an optimal policy, it is likely the case that improved exploration strategies beyond epsilon-greedy might help improve the efficiency of learning.

A simple addition would be to test softmax action selection rather than winner-take-all action selection based on the estimated Q-values.

More complex additions would include the addition of an intrinsic motivation function which rewards the agent for discovering novel states; in [66], the agent was rewarded for discovering states that it did not yet know how to compress well, with the intention of driving the compressor to improve its performance over time, and by extension, to improve the ability of the controller to find a more effective control policy.

In recent work [67], an approach to deep (temporally-extended) exploration is applied to DQN and could also be extended to this setting.

Chapter 10

Conclusion

A brief overview of neural networks and reinforcement learning was presented, and the deep reinforcement learning paradigm was introduced. This was followed by a description of a method which uses neural networks for deep reinforcement learning via direct policy search with neuroevolution, and the value-based methods Neural Fitted Q Iteration (NFQ) and Deep Q-Networks (DQN).

Detailed algorithms were presented for direct policy search using neuroevolution and for a novel extension of NFQ called *Regularized Convolutional Neural Fitted Q Iteration* (RC-NFQ), which adds convolutional neural networks and dropout regularization to the NFQ algorithm along with several elements from the DQN algorithm.

A modified version of the *TORCS* race-car simulator and a real robotic car testbed were presented as environments for developing and testing deep reinforcement learning algorithms which learn control policies from high-dimensional vision input in simulated or real environments.

Experiments applying direct policy search using neuroevolution to the *TORCS* race-car simulator were presented, illustrating successful learning of certain control policies and comparing different hyperparameter settings.

Additionally, preliminary experiments applying the proposed RC-NFQ algorithm to the problem of robotic control using the robotic car testbed were presented. The effects of dropout regularization and various hyperparameters on learning were analyzed. Initial evidence of learning progress and a beneficial effect of regularization were found, and should be investigated further in subsequent work. Finally, several additional extensions were proposed to compare learning algorithms and exploration policies.

Chapter 11

Supplementary Materials

11.1 Summary

Source code for our implementation of the algorithms is available at the following links, along with datasets used for training and a sample video of the neuroevolution process.

11.2 Source Code

11.2.1 Neuroevolution

The neuroevolution implementation is available here:
`https://github.com/cosmoharrigan/neuroevolution`

11.2.2 RC-NFQ

The RC-NFQ implementation is available here:
`https://github.com/cosmoharrigan/rc-nfq`

11.3 Datasets

11.3.1 Neuroevolution

The training images for the convolutional neural network are available here:
`http://machineintelligence.org/neuroevolution/training_images.zip`

11.3.2 RC-NFQ

The experience tuples collected from the robot are available here:
`http://machineintelligence.org/rc-nfq/experience_tuples.zip`

11.4 Videos

11.4.1 Neuroevolution

A video sample of the training process is available here:
`https://youtu.be/rS4F2V2VdoQ`

Acknowledgements

I wish to extend my appreciation to Dr. Dieter Fox and Arunkumar Byravan from the University of Washington Department of Computer Science & Engineering for their support and feedback during this project. I would also like to thank Phil Snyder for valuable feedback on an earlier version of this document, and Dr. Marina Meila, Dr. Daniel Weld, Dr. Ben Goertzel and Dr. Joscha Bach for their additional guidance during my research.

Bibliography

[1] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. "Learning Internal Representations by Error Propagation". In: (1985). URL: http://oai.dtic.mil/oai/oai?verb=getRecord{\&}metadataPrefix=html{\&}identifier=ADA164453.

[2] Yann LeCun et al. "Gradient-based learning applied to document recognition". In: *Proceedings of the IEEE* 86.11 (1998), pp. 2278–2323. ISSN: 00189219. DOI: 10.1109/5.726791. arXiv: 1102.0183.

[3] Jürgen Schmidhuber. "Deep learning in neural networks: An overview". In: *Neural Networks* 61 (2015), pp. 85–117.

[4] Yoshua Bengio, Ian J Goodfellow, and Aaron Courville. *Deep Learning*. 2015.

[5] Y. LeCun et al. "Backpropagation Applied to Handwritten Zip Code Recognition". In: *Neural Computation* 1.4 (1989), pp. 541–551. ISSN: 0899-7667. DOI: 10.1162/neco.1989.1.4.541. URL: http://dl.acm.org/citation.cfm?id=1351079.1351090.

[6] Kunihiko Fukushima. "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position". In: *Biological cybernetics* 36.4 (1980), pp. 193–202.

[7] Olga Russakovsky et al. "ImageNet Large Scale Visual Recognition Challenge". In: *International Journal of Computer Vision* 115.3 (2015), pp. 211–252. ISSN: 0920-5691. DOI: 10.1007/s11263-015-0816-y. URL: http://link.springer.com/10.1007/s11263-015-0816-y.

[8] David H Hubel and Torsten N Wiesel. "Receptive fields and functional architecture of monkey striate cortex". In: *The Journal of physiology* 195.1 (1968), pp. 215–243.

[9] Fabio Anselmi and Tomaso Poggio. *Representation learning in sensory cortex: a theory*. Tech. rep. Center for Brains, Minds and Machines (CBMM), 2014.

[10] Jost Tobias Springenberg et al. "Striving for Simplicity: The All Convolutional Net". In: (2014). arXiv: 1412.6806. URL: http://arxiv.org/abs/1412.6806.

[11] D. Whitley. "Genetic Algorithms and Neural Networks". In: *Genetic Algorithms in Engineering and Computer Science*. John WIley and Sons, 1995.

[12] DJ Montana and L Davis. "Training Feedforward Neural Networks Using Genetic Algorithms". In: *ijcai.org* (). URL: http://www.ijcai.org/PastProceedings/IJCAI-89-VOL1/PDF/122.pdf.

[13] John H Holland. *Adaptation in natural and artificial systems: an introductory analysis with applications to biology, control, and artificial intelligence*. U Michigan Press, 1975.

[14] Melanie Mitchell. *An Introduction to Genetic Algorithms*. MIT Press, 1998, p. 209. ISBN: 0262631857. URL: https://books.google.com/books?hl=en{\&}lr={\&}id=0eznlz0TF-IC{\&}pgis=1.

[15] Tijmen Tieleman and Geoffrey Hinton. "Lecture 6.5-rmsprop: Divide the gradient by a running average of its recent magnitude". In: *COURSERA: Neural Networks for Machine Learning* 4 (2012), p. 2.

[16] Martin Riedmiller and Heinrich Braun. "A direct adaptive method for faster backpropagation learning: The RPROP algorithm". In: *Neural Networks, 1993., IEEE International Conference on*. IEEE. 1993, pp. 586–591.

[17] John Duchi, Elad Hazan, and Yoram Singer. "Adaptive subgradient methods for online learning and stochastic optimization". In: *The Journal of Machine Learning Research* 12 (2011), pp. 2121–2159.

[18] Matthew D Zeiler. "ADADELTA: an adaptive learning rate method". In: *arXiv preprint arXiv:1212.5701* (2012).

[19] Diederik Kingma and Jimmy Ba. "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980* (2014).

[20] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. Vol. 1. 1. MIT press Cambridge, 1998.

[21] Shane Legg and Marcus Hutter. "Universal intelligence: A definition of machine intelligence". In: *Minds and Machines* 17.4 (2007), pp. 391–444.

[22] Marco Wiering and Martijn van Otterlo. *Reinforcement Learning: State-of-the-art*. Vol. 12. Springer Science & Business Media, 2012.

[23] Stuart Russell, Peter Norvig, and Artificial Intelligence. "A modern approach". In: *Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs* 25 (1995), p. 27.

[24] Marcus Hutter. *Universal Artificial Intelligence*. Vol. 1. 2. 2005, pp. 1–82. ISBN: 9783540221395. DOI: 10.1145/1358628.1358961.

[25] Shane Legg. "Machine super intelligence". PhD thesis. University of Lugano, 2008.

[26] Ronald J Williams. "Simple statistical gradient-following algorithms for connectionist reinforcement learning". In: *Machine learning* 8.3-4 (1992), pp. 229–256.

[27] Marc Peter Deisenroth, Gerhard Neumann, Jan Peters, et al. "A Survey on Policy Search for Robotics." In: *Foundations and Trends in Robotics* 2.1-2 (2013), pp. 1–142.

[28] Christopher J C H Watkins and Peter Dayan. "Q-learning". In: *Machine learning* 8.3-4 (1992), pp. 279–292.

[29] Martin Riedmiller. *Neural fitted q iteration first experiences with a data efficient neural reinforcement learning method*. Ed. by João Gama et al. Vol. 3720. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 317–328. ISBN: 978-3-540-29243-2. DOI: 10.1007/11564096. URL: http://dl.acm.org/citation.cfm?id=2130928.2130962.

[30] Long-Ji Lin. "Self-Improving Reactive Agents Based on Reinforcement Learning, Planning and Teaching". en. In: *Machine Learning* 8.3-4 (), pp. 293–321. ISSN: 1573-0565. DOI: 10.1023/A:1022628806385. URL: http://link.springer.com/article/10.1023/A{\%}3A1022628806385.

[31] Martin Riedmiller. "10 Steps and Some Tricks to Set up Neural Reinforcement Controllers". In: *Neural Networks: Tricks of the Trade*. Springer, 2012, pp. 735–757.

[32] Volodymyr Mnih et al. "Playing Atari with Deep Reinforcement Learning". In: (2013). arXiv: 1312.5602. URL: http://arxiv.org/abs/1312.5602.

[33] Volodymyr Mnih et al. "Human-level control through deep reinforcement learning". In: *Nature* 518.7540 (2015), pp. 529–533. ISSN: 0028-0836. DOI: 10.1038/nature14236. arXiv: 1312.5602. URL: http://dx.doi.org/10.1038/nature14236.

[34] Amir M Farahmand et al. "Regularized policy iteration". In: *Advances in Neural Information Processing Systems*. 2009, pp. 441–448.

[35] Bo Liu, Sridhar Mahadevan, and Ji Liu. "Regularized off-policy TD-learning". In: *Advances in Neural Information Processing Systems*. 2012, pp. 836–844.

[36] Ray J Solomonoff. "A formal theory of inductive inference. Part I". In: *Information and control* 7.1 (1964), pp. 1–22.

[37] Andrei N Kolmogorov. "Three approaches to the quantitative definition ofinformation'". In: *Problems of information transmission* 1.1 (1965), pp. 1–7.

[38] Jürgen Schmidhuber. "Discovering Neural Nets with Low Kolmogorov Complexity and High Generalization Capability". In: *Neural Networks* 10.5 (1997), pp. 857–873. ISSN: 08936080. DOI: 10.1016/S0893-6080(96)00127-X. URL: http://www.sciencedirect.com/science/article/pii/S089360809600127X.

[39] Nitish Srivastava et al. "Dropout: A simple way to prevent neural networks from overfitting". In: *The Journal of Machine Learning Research* 15.1 (2014), pp. 1929–1958.

[40] Thomas Lampe and Martin Riedmiller. "Approximate model-assisted neural fitted Q-iteration". In: *Neural Networks (IJCNN), 2014 International Joint Conference on*. IEEE. 2014, pp. 2698–2704.

[41] David E Goldberg and Kalyanmoy Deb. "A comparative analysis of selection schemes used in genetic algorithms". In: ().

[42] Bernhard Wymann et al. "TORCS : The open racing car simulator e". In: (2013), pp. 1–4.

[43] Daniele Loiacono, Luigi Cardamone, and Pier Luca Lanzi. "Simulated Car Racing Championship: Competition Software Manual". In: April (2013). arXiv: 1304.1672. URL: http://arxiv.org/abs/1304.1672.

[44] Jan Koutník et al. "Evolving large-scale neural networks for vision-based reinforcement learning". In: *Proceeding of the fifteenth annual conference on Genetic and evolutionary computation conference - GECCO '13*. New York, New York, USA: ACM Press, 2013, p. 1061. ISBN: 9781450319638. DOI: 10.1145/2463372.2463509. URL: http://dl.acm.org/citation.cfm?id=2463372.2463509.

[45] Jan Koutník, Jürgen Schmidhuber, and Faustino Gomez. "Online evolution of deep convolutional network for vision-based reinforcement learning". In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 8575 LNAI (2014), pp. 260–269. ISSN: 16113349. DOI: 10.1007/978-3-319-08864-8-25.

[46] Jan Koutník, Jürgen Schmidhuber, and Faustino Gomez. "Evolving deep unsupervised convolutional networks for vision-based reinforcement learning". In: *Proceedings of the 2014 conference on Genetic and evolutionary computation*. ACM. 2014, pp. 541–548.

[47] Timothy P. Lillicrap et al. "Continuous control with deep reinforcement learning". In: (2015), p. 10. arXiv: 1509.02971. URL: http://arxiv.org/abs/1509.02971.

[48] Volodymyr Mnih et al. "Asynchronous Methods for Deep Reinforcement Learning". In: (2016). arXiv: 1602.01783. URL: http://arxiv.org/abs/1602.01783.

[49] Pieter Hintjens. *ZeroMQ: Messaging for Many Applications*. O'Reilly Media, Inc., 2013.

[50] Francois Chollet. *Keras: Deep Learning library for Theano and TensorFlow*. https://github.com/fchollet/keras. 2015.

[51] Félix-Antoine Fortin et al. "DEAP: Evolutionary Algorithms Made Easy". In: *Journal of Machine Learning Research* 13 (2012), pp. 2171–2175.

[52] J Elman. "Finding structure in time". In: *Cognitive Science* 14.2 (1990), pp. 179–211. ISSN: 03640213. DOI: 10.1016/0364-0213(90)90002-E. URL: http://www.sciencedirect.com/science/article/pii/036402139090002E.

[53] Faustino Gomez, Jürgen Schmidhuber, and Risto Miikkulainen. "Accelerated Neural Evolution through Cooperatively Coevolved Synapses". In: *The Journal of Machine Learning Research* 9 (2008), pp. 937–965. ISSN: 1532-4435. URL: http://dl.acm.org/citation.cfm?id=1390681.1390712.

[54] Nikolaus Hansen and Andreas Ostermeier. "Completely derandomized self-adaptation in evolution strategies". In: *Evolutionary computation* 9.2 (2001), pp. 159–195.

[55] Sascha Lange and Martin Riedmiller. “Deep auto-encoder neural networks in reinforcement learning”. English. In: *The 2010 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2010, pp. 1–8. ISBN: 978-1-4244-6916-1. DOI: 10.1109/IJCNN.2010.5596468. URL: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=5596468.

[56] Sepp Hochreiter and Jürgen Schmidhuber. “Long short-term memory”. In: *Neural computation* 9.8 (1997), pp. 1735–1780.

[57] Yarin Gal. “A Theoretically Grounded Application of Dropout in Recurrent Neural Networks”. In: *arXiv preprint arXiv:1512.05287* (2015).

[58] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: (2015). arXiv: 1502.03167. URL: http://arxiv.org/abs/1502.03167.

[59] Tom Schaul et al. “Prioritized Experience Replay”. In: (2015). arXiv: 1511.05952. URL: http://arxiv.org/abs/1511.05952.

[60] Leemon C. Baird III. “Advantage Updating”. In: (1993). URL: http://oai.dtic.mil/oai/oai?verb=getRecord{\&}metadataPrefix=html{\&}identifier=ADA280862.

[61] Mance E. Harmon and Leemon C. Baird. “MultiPlayer Residual Advantage Learning With General Function Approximation”. In: (1996).

[62] Ziyu Wang, Nando de Freitas, and Marc Lanctot. “Dueling Network Architectures for Deep Reinforcement Learning”. In: (2015), p. 14. arXiv: 1511.06581. URL: http://arxiv.org/abs/1511.06581.

[63] John Schulman et al. “High-Dimensional Continuous Control Using Generalized Advantage Estimation”. In: (2015). arXiv: 1506.02438. URL: http://arxiv.org/abs/1506.02438.

[64] Xiaoxiao Guo et al. “Deep Learning for Real-Time Atari Game Play Using Offline Monte-Carlo Tree Search Planning”. In: *Advances in Neural Information Processing Systems*. 2014, pp. 3338–3346. URL: http://papers.nips.cc/paper/5421-scalable-inference-for-neuronal-connectivity-from-calcium-imaging.

[65] David Silver et al. “Mastering the game of Go with deep neural networks and tree search”. In: *Nature* 529.7587 (2016), pp. 484–489.

[66] Giuseppe Cuccu et al. “Intrinsically motivated neuroevolution for vision-based reinforcement learning”. English. In: *2011 IEEE International Conference on Development and Learning (ICDL)*. Vol. 2. IEEE, 2011, pp. 1–7. ISBN: 978-1-61284-989-8. DOI: 10.1109/DEVLRN.2011.6037324. URL: http://ieeexplore.ieee.org/articleDetails.jsp?arnumber=6037324.

[67] Ian Osband et al. “Deep Exploration via Bootstrapped DQN”. In: (2016). arXiv: 1602.04621. URL: http://arxiv.org/abs/1602.04621.