

Neural Q-Learning

Stephan ten Hagen and Ben Kröse
Faculty of Science, University of Amsterdam
Kruislaan 403, 1098SJ Amsterdam
{stephanh,krose}@science.uva.nl
tel: +31 20 525 7564
fax: +31 20 525 7490

Abstract

In this paper we introduce a novel neural reinforcement learning method. Unlike existing methods our approach does not need a model of the system and can be trained directly using the measurements of the system. We achieve this by only using one function approximator and approximate the improved policy from this. An experiment using a mobile robot shows that it can be trained using a real system within reasonable time.

keywords: Reinforcement Learning, Optimal control, Feed-forward network, Nonlinear systems, Learning from real systems.

1 Introduction

General function approximators, such as neural networks, have been proposed for a wide variety of control formalisms [1, 2, 3]. For optimal control tasks involving nonlinear systems, ‘reinforcement learning’ (RL) or ‘adaptive critic’ approaches have been successfully proposed [4, 5]. In RL, a controller is optimized on the basis of a cost function representing the expected future costs (or ‘reinforcements’).

Various applications of RL in control have been described. Sofge and White used RL to obtain a controller for a manufacturing process for thermoplastic structures [6] and for a molding process and chip fabrication [7]. Other authors applied RL to obtain a thermostat controller [8, 9], a controller for satellite positioning [10], and various applications for chemical plants and reactors [9, 11] and [12]. In [13] RL is used to control a ‘fuzzy’ ball-and-beam system.

Although their apparent successes, these RL applications are to a large extent based on heuristics. Often they optimize multiple networks simultaneously, and their success will depend on the choice of learning parameters, number of hidden units etcetera. Networks are trained using other networks resulting in dependencies preventing the specification of general rules to make the choices. Besides the inability to make educated guesses about the appropriate training setup, multiple networks require more training data because of the increased number of free parameters. Existing methods are never applied directly to real systems but instead use simulations to generate a sufficient amount of data, requiring a model of the system.

In this paper we propose a continuous state and action space approach that uses only one function approximator, and that can be trained fast using real data obtained from short runs of the system.

1.1 Goal and overview

Our main goal is to increase the practical applicability of neural reinforcement learning. For this we have to be able to obtain a control policy based on measurements from the *real* system. This can only work if:

Training is fast: Fast in the sense that a very small data set is sufficient, because the real system will dictate the speed at which data becomes available. Training for several thousands of iterations is very impractical. Also it is not certain whether the increase in performance outweighs the effort and expenses of the generation of data.

Result is simple: A control policy for a real system should also make sure that the hardware does not get damaged. For this reason we rather have a simple control policy than a complicated policy that *might* be optimal.

We do not represent the control policy by a general function approximator. Instead we derive the control directly from the trained function approximator representing the future cost.

In section 2 we describe Q-learning in the discrete domain and in the continuous domain. In section 3 we introduce our novel approach that is based on a quadratic approximation of the Q-function. We tested our method on a real nonlinear control problem. The results presented in section 4 show that we can find a controller for a real system on the basis of a relative small training set.

2 Model-free RL

In Reinforcement Learning an existing control policy is used to interact with a dynamical system. Each time step a scalar immediate evaluation called “reinforcement” becomes available. A new improved policy is derived from an estimation of the sum of future reinforcements. In Q-learning the sum of future reinforcement is estimated for each state action pair, such that no model of the system is needed to derive the new policy.

2.1 Q-Learning in discrete state-action space

In a discrete state-action space the control problem is typically considered as a Markov Decision Problem. Here we have¹

- A finite set of states $\{\mathbf{x}^i\} \in \mathcal{X}$.
- A finite set of actions $\{\mathbf{u}^j\} \in \mathcal{U}$. The set of actions can depend on the state, because it is possible that not all actions are possible in all states.
- A system model \mathcal{M} , which gives the probabilities of the state transitions when a given action is applied.
- Reinforcements r_k , which indicate the costs at time step k .

The *policy* \mathbf{g} is the function that maps the states to the action, so that action $\mathbf{u} = \mathbf{g}(\mathbf{x})$ is taken in state \mathbf{x} .

¹Note that we use the same notation for discrete and continuous state-action spaces. In case of a discrete state-action spaces often the symbols s , a and π are used for the state, action and policy.

Define the Q-function as the state-action value function for policy \mathbf{g} :

$$\mathbf{Q}^{\mathbf{g}}(\mathbf{x}_k, \mathbf{u}_k) = \mathcal{E} \left\{ \sum_{i=k}^{\infty} \gamma^{i-k} r_i \right\} \quad (1)$$

This is the expected sum of future reinforcements r , when at time step k in state \mathbf{x}_k action \mathbf{u}_k is taken and in future all actions are taken according to policy \mathbf{g} . The γ is a weighting of future reinforcement and makes sure that the sum exist. In this paper we will make sure that r_k approaches zero fast enough and therefore we take $\gamma = 1$.

If (1) holds, it is clear that the Q-value at the present time step must equal the received reinforcement plus the Q-value at the next time step. Based on this observation the Temporal Difference (TD) learning rule was introduced in [14]. The update:

$$\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k) \rightarrow \mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k) + \alpha(r_k + \mathbf{Q}(\mathbf{x}_{k+1}, \mathbf{g}(\mathbf{x}_{k+1})) - \mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k)) \quad (2)$$

with α as the learning rate, is the temporal difference update for the estimation of the Q-function². The new estimate of $\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k)$ is changed proportional to the difference between the current estimation and what it should be according to the received reinforcement plus the estimation at the next time step.

For a discrete state-action space a look-up table can be used for the Q-function in (1). In each state the action with the lowest Q-value can be selected:

$$\mathbf{u}'_k = \underset{\mathbf{u}}{\operatorname{argmin}} \mathbf{Q}(\mathbf{x}_k, \mathbf{u}). \quad (3)$$

Applying this to all states results in the *greedy* policy. If the Q-function (1) is approximated correctly then the greedy policy will be an improvement compared to the policy used to generate the data. The process of generating data and selecting the greedy policy can be repeated and this will eventually converge to the optimal policy [15, 16].

2.2 Continuous state-action space

In a continuous state-action space the control problem can be described as a Markov process. Here we have

- A infinite set of states $\{\mathbf{x}\} \in \mathcal{X}$ with $\mathcal{X} \subset \mathfrak{R}^{n_x}$.
- A finite set of actions $\{\mathbf{u}\} \in \mathcal{U}$, with $\mathcal{U} \subset \mathfrak{R}^{n_u}$. The set of actions can depend on the state, because it is possible that not all actions are possible in all states.
- A system model \mathcal{M} , which gives the next state transition given the current state and the control action. Usually this is a deterministic functional mapping with the uncertainty represented as additional noise.
- Reinforcements r_k , which indicate the costs at time step k .

²Note that this is the policy iteration update, where the estimation of the Q-values is separated from the improvement of the policy. The original Q-Learning in [15] was based on value iteration where the estimation of the Q-value and the policy improvement are one step. Instead of $\mathbf{g}(\mathbf{x}_{k+1})$ in (2), the original update uses $\underset{\mathbf{u}}{\operatorname{argmin}} \mathbf{Q}(\mathbf{x}_{k+1}, \mathbf{u})$ as the action at the next time step.

The *policy* \mathbf{g} is the function that maps the states to the action, so that action $\mathbf{u} = \mathbf{g}(\mathbf{x})$ is taken in state \mathbf{x} .

The continuous state and action space can be quantized so that look-up tables can be used. An other possibility is to replace the look-up tables with a general function approximators. If we take a function approximator $\mathbf{Q}(\mathbf{x}, \mathbf{u}, \mathbf{w})$, with \mathbf{w} the parameters, then TD learning becomes minimizing the temporal difference error:

$$E = \sum_{k=0}^{N-1} (r_k + \mathbf{Q}(\mathbf{x}_{k+1}, \mathbf{g}(\mathbf{x}_{k+1}), \mathbf{w}) - \mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}))^2. \quad (4)$$

Here N is the number of time steps data is available. The effect of minimization is that Q values at two successive time steps will only differ in the reinforcement that is received in between.

Given the function approximator the computation of the greedy control policy according to (3) can lead to two problems:

- The extremum of a nonlinear function has to be computed which is in general not a trivial task.
- The greedy control policy does not have to be smooth and continuous even when the approximated Q -function is smooth and continuous. Near the discontinuities a little noise can have a high influence on the control actions, so that the behavior of the system becomes very unpredictable.

The second problem can be overcome by introducing a Dynamic Output Element [12]. The policy is followed by a low pass filter to removing the effects of noise at the discontinuities. The resulting control policy is no longer a function of the current state alone, but also of past state values. The first problem is still not solved.

Both problems are solved in the actor-critic configuration. The approximator representing the Q -function is called the critic and a second approximator called the actor is introduced to form the control policy. The actor can be trained by using the critic as the “error function” that has to be minimized [6]. The training itself solves the first problem and selecting an actor that can only be continuous solves the second problem. One major drawback of actor critic approaches is that they involve the training of two or more function approximators. This not only makes the training very difficult, a more serious problem is that the analysis of the outcome is too difficult. If the approach fails, it is unclear whether this is due to the settings of the training parameters, the choice of approximators or the use of insufficient exploration in generating the data.

There exist architectures which use a critic representing a ‘Value-function’: the expected future costs as a function of the state only. Such models need a system model, often represented by an additional function approximator [4, 5]. With a critic representing a Q -function no model is needed, but the training requires many episodes in which data is generated and the actor is adapted. Therefore the training is always performed using a simulation of the system, such that a model is still being used.

3 Neural Q-Learning

In this section we introduce our novel approach, which is based on the following observations:

- The two methods from section 2.2 may result in a control policy that in principle can be anything. But controlling real systems using a “potentially” arbitrary control policy can be

very risky. A safer approach is to define beforehand a class of feasible control policies from which the best is derived based on the approximated Q-function.

- Many real systems cannot generate sufficient training data fast enough to guarantee an accurately approximated Q-function. This implies that the greedy policy derived from this Q-function does not have to perform well. The approximated Q-function should only be regarded as a rough indication of the expected performance.

Similar to the original Q-learning approach, we make sure that the new control policy directly follows from the approximated Q-function. Therefore we will refer to it as Neural Q-learning.

3.1 The linear control policy

Suppose a function approximator $\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w})$ was trained by minimizing (4). How can we derive a new control policy from this function? Instead of using the methods described in section 2.2, we start by approximating a quadratic function from $\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w})$ and derive the control policy from this quadratic function.

In this paper we consider tasks for which all immediate reinforcements $r_k = r(\mathbf{x}_k, \mathbf{u}_k)$ have a minimum and a zero gradient in the origin. This implies that the origin is our “target” state and the objective is to optimize the trajectory towards the origin. For this situation the approximated quadratic function should have a global minimum in the origin, so we can use the same quadratic function as in [17, 18, 19]. The greedy policy for this Q-function is a linear function of the state.

A second order Taylor expansion of $\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w})$ around $\mathbf{x} = \mathbf{0}$ and $\mathbf{u} = \mathbf{0}$ can be made. This results in the quadratic function:³

$$\bar{\mathbf{Q}}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}) = c + \begin{bmatrix} \mathbf{G}_x & \mathbf{G}_u \end{bmatrix} \begin{bmatrix} \mathbf{x}_k \\ \mathbf{u}_k \end{bmatrix} + \begin{bmatrix} \mathbf{x}_k^T & \mathbf{u}_k^T \end{bmatrix} \begin{bmatrix} \mathbf{H}_{xx} & \mathbf{H}_{ux} \\ \mathbf{H}_{xu} & \mathbf{H}_{uu} \end{bmatrix} \begin{bmatrix} \mathbf{x}_k \\ \mathbf{u}_k \end{bmatrix}, \quad (5)$$

where the \mathbf{G} and \mathbf{H} parameters are formed by the gradient and Hessian:

$$c = \mathbf{Q}(\mathbf{0}, \mathbf{0}, \mathbf{w}) \quad \mathbf{G}_i = \nabla_i \mathbf{Q}(\mathbf{0}, \mathbf{0}, \mathbf{w}) = \mathbf{0} \quad \mathbf{H}_{ij} = \frac{1}{2} \nabla_{ij}^2 \mathbf{Q}(\mathbf{0}, \mathbf{0}, \mathbf{w}).$$

The \mathbf{G} is zero because of our choice of immediate reinforcements.

The quadratic approximation simplifies the minimization of the Q function with respect to the action. If matrix \mathbf{H} is positive definite, the gradient of $\bar{\mathbf{Q}}$ with respect to \mathbf{u} can be set to zero and solved. Let \mathbf{u}'_k be the action for which the gradient is zero, then:

$$\begin{aligned} \nabla_{\mathbf{u}} \bar{\mathbf{Q}} &= 2\mathbf{H}_{uu}\mathbf{u}'_k + 2\mathbf{H}_{ux}\mathbf{x}_k = \mathbf{0} \\ \mathbf{u}'_k &= -\mathbf{H}_{uu}^{-1}\mathbf{H}_{ux}\mathbf{x}_k \end{aligned} \quad (6)$$

So the resulting control policy is given by

$$\mathbf{u}'_k = \bar{\mathbf{L}}\mathbf{x}_k, \quad (7)$$

which is linear function of the state.

³We use the bar to indicate the quadratic Q function and the results that follow from it.

3.2 The nonlinear control policy

Instead of using a general approximator we can estimate the parameters of a quadratic function to obtain (5). This has one major drawback. Training data obtained from areas where the Q-function deviates from a quadratic function will have large errors and therefore influence the estimated quadratic function too much. So directly estimating the parameters will only have a reliable outcome when the true Q-function is quadratic. For an unknown nonlinear system a general function approximator $\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w})$ has to be used.

The quadratic function (5) is only used to obtain $\bar{\mathbf{L}}$, which allows us to make a new local approximation around the point $(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k)$. We approximate it by

$$\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w}) \approx \begin{bmatrix} \mathbf{x}_k^\top & \mathbf{u}_k^\top \end{bmatrix} \begin{bmatrix} \mathbf{H}_{xx}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) & \mathbf{H}_{ux}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) \\ \mathbf{H}_{xu}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) & \mathbf{H}_{uu}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) \end{bmatrix} \begin{bmatrix} \mathbf{x}_k \\ \mathbf{u}_k \end{bmatrix} \quad (8)$$

with

$$\mathbf{H}_{ij}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) = \frac{1}{2} \nabla_{ij}^2 \mathbf{Q}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k, \mathbf{w}).$$

If $\mathbf{H}_{ij}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k)$ equals the \mathbf{H}_{ij} in (5) then $\bar{\mathbf{L}}\mathbf{x}_k$ is the correct greedy action to take. This is true for all states if $\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w})$ is a quadratic function.

In case $\mathbf{H}_{ij}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) \neq \mathbf{H}_{ij}$ the $\mathbf{Q}(\mathbf{x}_k, \mathbf{u}_k, \mathbf{w})$ is not a quadratic function. This implies that $\bar{\mathbf{L}}\mathbf{x}_k$ is not the action that minimizes (8). We can compute that action analogue to (6):

$$\mathbf{u}'_k = -\mathbf{H}_{uu}^{-1}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) \mathbf{H}_{ux}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) \mathbf{x}_k = \mathbf{L}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k) \mathbf{x}_k \quad (9)$$

This results in a nonlinear control policy.

The above describe procedure always results in (9) as control policy. If the true Q-function is quadratic the result will be linear. If the Hessian around $(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k)$ indicates a local deviation from this quadratic function then locally an alternative quadratic function is used to derive the control action.

3.3 The implementation

The control policy should follow directly from the approximated Q-function, because we do not use an actor. This is already possible by the quadratic approximations. The parameters of the quadratic approximations have to be expressed using the parameters and architecture of the general function approximator. To simplify this the quadratic functions can also be replaced by linear functions with quadratic inputs.

Consider the following scalar example:

$$\begin{aligned} \bar{\mathbf{Q}}(x, u) &= \begin{bmatrix} x & u \end{bmatrix} \begin{bmatrix} H_{xx} & H_{ux} \\ H_{xu} & H_{uu} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \\ &= \begin{bmatrix} H_{xx} & H_{ux} + H_{xu} & H_{uu} \end{bmatrix} \begin{bmatrix} x^2 \\ xu \\ u^2 \end{bmatrix} = \boldsymbol{\theta}^\top \boldsymbol{\phi} = \bar{\mathbf{Q}}(\boldsymbol{\phi}) \end{aligned} \quad (10)$$

From the linear approximator $\bar{\mathbf{Q}}(\boldsymbol{\phi})$ with input $\boldsymbol{\phi}$ and parameters $\boldsymbol{\theta}$ the parameters of the quadratic function $\bar{\mathbf{Q}}(x, u)$ can be obtained because \mathbf{H} is symmetric. This is important because it means that we still can use \mathbf{H} to compute $\bar{\mathbf{L}}$.

Using the quadratic inputs $\phi^T = [x_0^2, x_0x_1, \dots, x_0u_0, \dots]^T$ has several advantages. The function formed by the general approximator is already “shaped” to form quadratic like functions, making the quadratic approximations (5) and (8) more reliable. The task of the general approximator is reduced to only model the deviation with a quadratic function. In case of a linear system and quadratic immediate reinforcements the true Q-function is quadratic so, that only the parameters of a linear function have to be estimated. The minimization of (4) can be expressed as a linear least square estimation problem and is guaranteed to converge to the optimal linear control policy [17, 18, 19, 20].

For the $\mathbf{Q}(\phi) = \boldsymbol{\theta}^T \phi$ in (10) the parameters can also be found using the gradient $\nabla_{\phi} \mathbf{Q}(\phi) = \boldsymbol{\theta}^T$. For a general approximator $\boldsymbol{\theta}^T(\phi_k, \mathbf{w})$ we can derive the parameters similarly:

$$\boldsymbol{\theta}^T(\phi_k, \mathbf{w}) = \nabla_{\phi_k} \mathbf{Q}(\phi_k, \mathbf{w})$$

We have to make sure that we choose a function approximator for which we can derive the linear and nonlinear control policy. Take a function approximator for which the gradient consist of a constant and a input dependent part:

$$\nabla_{\phi_k} \mathbf{Q}(\phi_k, \mathbf{w}) = \bar{\boldsymbol{\theta}}^T + \tilde{\boldsymbol{\theta}}^T(\phi_k, \mathbf{w}) \quad (11)$$

The constant part $\bar{\boldsymbol{\theta}}$ does not depend on the input so it can be regarded as a representation of the global quadratic approximation (5). From this $\bar{\mathbf{L}}$ can be computed according to (7), which can be used to express the input ϕ_k as a quadratic combinations of \mathbf{x}_k and $\bar{\mathbf{L}}\mathbf{x}_k$. With the ϕ_k the variable part $\tilde{\boldsymbol{\theta}}^T(\phi_k, \mathbf{w})$ of (11) can be computed, which then can be rearranged to form $\mathbf{H}(\mathbf{x}_k, \bar{\mathbf{L}}\mathbf{x}_k)$ in (8).

3.4 Using a feed forward network

To demonstrate the above mentioned approach we will show that a conventional feed-forward network can be used as function approximator. Unlike the adaptive critic approach where the state \mathbf{x} and action \mathbf{u} are used as inputs, the quadratic combinations ϕ are used as input. Take a network with one hidden layer and a linear output unit:

$$\mathbf{Q}(\phi_k, \mathbf{w}) = \mathbf{w}_o^T \boldsymbol{\Gamma} \left(\begin{bmatrix} \mathbf{w}_{h,1}^T \\ \dots \\ \mathbf{w}_{h,n_h}^T \end{bmatrix} \phi_k + \mathbf{b}_h \right) + b_o$$

The weights \mathbf{w}_o and $\mathbf{w}_{h,j}$ and biases \mathbf{b}_h and b_o form the weights \mathbf{w} of the network and n_h is the number of hidden units. The $\boldsymbol{\Gamma}$ represent the vector of activations (outputs) of the hidden units. Take units with hyperbolic tangent activations:

$$\Gamma_{h,j}(\phi_k) = \tanh(\mathbf{w}_{h,j}\phi_k + b_{h,j}).$$

For this network we can compute the parameters θ_i by taking the partial derivative with respect to the corresponding input ϕ_i :

$$\begin{aligned} \theta_i(\phi_k, \mathbf{w}) &= \frac{\partial \mathbf{Q}(\phi_k, \mathbf{w})}{\partial \phi_{k,i}} \\ &= \sum_{j=1}^{n_h} w_{o,j} w_{h,j,i} (1 - \tanh^2(\mathbf{w}_{h,j}\phi_k + b_{h,j})) \\ &= \underbrace{\sum_{j=1}^{n_h} w_{o,j} w_{h,j,i}}_{\tilde{\theta}_i} - \underbrace{\sum_{j=1}^{n_h} w_{o,j} w_{h,j,i} \tanh^2(\mathbf{w}_{h,j}\phi_k + b_{h,j})}_{\tilde{\theta}_i(\phi_k, \mathbf{w})}. \end{aligned} \quad (12)$$

The partial derivative consist of a constant and input dependent part. The constant parts $\bar{\theta}_i$ together form $\bar{\boldsymbol{\theta}}$ that is used to compute $\bar{\mathbf{L}}$. This is used to compute the $\boldsymbol{\phi}_k$ in the variable part $\tilde{\theta}_i(\boldsymbol{\phi}_k, \mathbf{w})$.

The variable part $\tilde{\theta}_i(\boldsymbol{\phi}_k, \mathbf{w})$ is very small when $\mathbf{w}_{h,j}$ and \mathbf{b}_h are very small. An almost quadratic function is formed by the approximator. This is a good way to initialize the network. The result will be a linear control policy unless the minimization of (4) creates a deviation from a quadratic function.

4 Experiment

We applied the method to a control task involving a real mobile robot. Our mobile robot is an “engineered artifact” so we had a reasonable understanding about the dynamics and kinematics. A controller to make the robot perform a certain task can be easier obtained by designing it manually using the systems model. However, the knowledge about the model made it possible to interpret the result.

4.1 The task

We used a mobile robot that moves according to:

$$\begin{aligned} x_{k+1} &= x_k + \frac{v_k}{\omega_k} (\sin(\psi_k + T\omega_k) - \sin(\psi_k)) \\ y_{k+1} &= y_k + \frac{v_k}{\omega_k} (\cos(\psi_k) - \cos(\psi_k + T\omega_k)) \\ \psi_{k+1} &= \psi_k + \omega_k T \end{aligned} \quad (13)$$

where x , y are the positions in meters and ψ the orientation in radians. The v and ω are the traversal and rotational speed and T the duration of each time step. Kinematic model (13) holds for any $\omega \neq 0$ and gives a trajectory of the robot along a circle with radius $\frac{v}{\omega}$. If $\omega_k = 0$, the orientation does not change and $x_{k+1} = x_k + Tv_k \sin(\psi_k)$ and $y_{k+1} = y_k + Tv_k \cos(\psi_k)$.

The task of the robot is to follow a straight line that is defined in the world. The state is given by the distance δ to that line and the orientation α with respect to that line. So the state is given by $\mathbf{x}^T = [\alpha \ \delta]$, where δ is given in meters and α in radians. Using (13) we can express the state transition for the robot given this task:

$$\begin{aligned} \alpha_{k+1} &= \alpha_k + \omega_k T \\ \delta_{k+1} &= \delta_k + \frac{v_k}{\omega_k} (\cos(\alpha_k) - \cos(\alpha_k + T\omega_k)). \end{aligned}$$

We set the traversal speed v at a fixed value of 0.1 meters per second. As control action u we took the rotation speed ω . Without loss of generality we can take the x-axis of the world as the line to follow. Our setup is shown in figure 1, where we take $\delta = y$ and $\alpha = \psi$.

We used as immediate reinforcement quadratic costs, so

$$r_k = r(\mathbf{x}_k, \mathbf{u}_k) = \mathbf{x}_k^T \begin{bmatrix} S_\alpha & 0 \\ 0 & S_\delta \end{bmatrix} \mathbf{x}_k + u_k R u_k, \quad (14)$$

with

$$S_\alpha = 0.1, \quad S_\delta = 1 \quad \text{and} \quad R = 1.$$

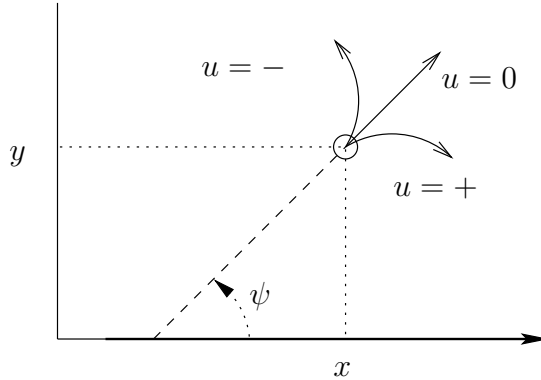


Figure 1: The circle is the robot in the world domain at position (x, y) and with orientation ψ .

This indicates that we want to minimize the distance to the line without steering too much. The S_α is given a small positive value is to assign costs to following the line in the wrong direction. Note that (14) has the global minimum with zero gradient in the origin.

We can reason about the control policy that can do the task. Assume that the robot is facing in the correct direction. Then the robot has to turn towards the line. If it is close to the line it does not have to rotate much but if it is further away the first priority is to face the line. However, there is a maximum required change in orientation. If the robot is far or very far away for the line, it does not have to rotate more than a quarter turn. If we would take a linear control policy then beyond a certain point the robot start making circular movements because it will rotate too much. The only way is to reduce the parameters of the linear policy, but that leads to a loss of performance near the line. The solution is to have a policy that is steep near the line for good performance and changes smoothly to a constant value further away and so extend the functionality in a larger part of the state space.

4.2 The implementation

4.2.1 The Robot

The robot we used in the experiments is a Nomad Super Scout II. This is a mobile robot with a two wheel differential drive at its geometric center. The drive motors of both wheels are independent and the width of the robot is 41 cm. The maximum speed is 1 m/s at an acceleration of 2 m/s².

The robot has a MC68332 processor board for the low level processes. These include sending the control commands to the drive motors, but also keeping track of the position and orientation of the robot by means of odometry. All other software runs on a second board equipped with a Pentium II 233Mhz processor.

The real robot does not exactly move like the ideal kinematic model, described above. The main two differences are:

Noise: The wheels can slip leading to a different displacement of the robot and due to the limited acceleration the trajectories are not exactly circular. The duration of the time steps may also vary because the state information is obtained via a communication protocol between the two processor boards.

Safety bounds: Because a real robot can be ruined we incorporated a safety measure by limiting the action applied to the robot. Actions outside the safety bound were replaced by

the value of the safety bound.

4.2.2 The data generation

The experiment consisted of three parts: the generation of data, the training of the network and testing the result. To generate the data we had to control the robot using an existing policy and store the values of the state and control actions. Also we had to incorporate exploration to make sure that alternative actions were tried.

We started the robot at an initial state and controlled it with a linear policy

$$u_k = \mathbf{L}\mathbf{x}_k + \mathbf{e}_k,$$

where we used $\mathbf{L} = \begin{bmatrix} -10^{-3} & -10^{-3} \end{bmatrix}$. This policy was chosen to approach the line very slowly, so that the costs r_k will approach zero. This policy was chosen too small to prevent the incorporation of too much prior knowledge. The sample time T was 0.35 seconds. We controlled the robot for 57 time steps, which corresponds to approximately 20 seconds.

The \mathbf{e}_k represents the exploration. We used zero-mean Gaussian white noise with variance $\sigma^2 = 0.01$. The values of the states and control actions, together with the cost r computed using (14) were stored every time step and form a train set. To make sure that the control policy will be valid for a large part of the state space, we started the robot at four different positions. We chose the initial orientation $\alpha_0 = 0$ and used four different initial distances: $\delta_0 = -1.75$, $\delta_0 = -0.75$, $\delta_0 = 0.75$ and $\delta_0 = 1.75$. This resulted in four train sets, which were combined to form one large train set.

4.2.3 The network

The train set was used to train the network, by minimizing the quadratic temporal difference error using (4). We did this using steepest descent. We chose the following initial settings for the network:

- Number of hidden units: 3.
A low number of hidden units was used to prevent over fitting.
- The value of the initial output weights: 0.
Note that this implies that $\boldsymbol{\theta} = \mathbf{0}$ so that the initial network does not specify a linear control policy.
- Values of initial weights and biases of hidden units: Random between -10^{-4} and 10^{-4} .
These weights are taken small so that the input dependent part of (12) is almost zero. In this way the nonlinear control policy is a consequence of the training data.

Because the result also depends on the initial weights, we trained the network for ten different initial weights. We selected the network with the lowest quadratic temporal difference error.

4.3 The results

We defined a task where the policy is a $\mathbb{R}^2 \rightarrow \mathbb{R}$ function, so we can plot the control action for all state values. In figure 2 we see that the result of the Neural Q-Learning approach is a linear controller with a smooth nonlinear correction. Also we see that the control actions are large. In fact, the linear policy derived from the network is too large. We see that in some parts of the state space the control action deviates from the linear policy. These are the areas in which we generated

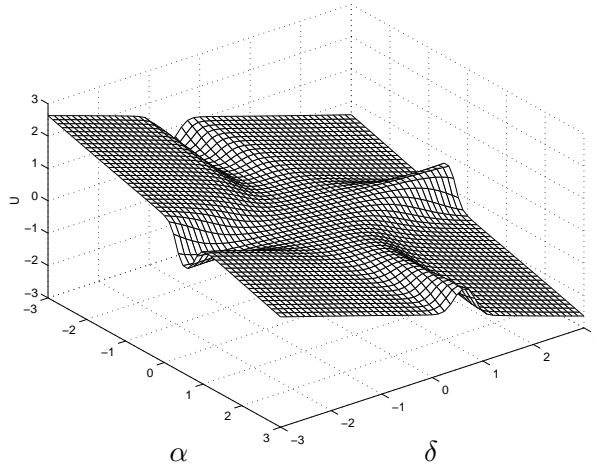


Figure 2: The resulting control policy.

data to train the network. The nonlinearity corrects the steep linear policy and makes that even at a large distance from the line the control action remains low enough. At these states the control actions are between the safety bounds of the real robot, so these actions can be applied.

We tested this control policy by using it to control the robot. We computed the total cost:

$$J = \sum_{k=0}^N r_k.$$

Because of the size of the room we could only test the robot for one minute, so we took $N = 171$. We tested the policy for each of the four initial positions used to generate the train set. The resulting total costs are shown in table 1.

In figure 3 we see that the two trajectories that start close to the line will approach the line. We also see that for starting points far from the line, the robot first starts to rotate very fast towards the line. The first few time steps the action is higher than the safety limit, so we only let the robot rotate with maximum speed. After that all actions are within the safety limit and the robot starts following the line. The high linear policy from the network makes the robot move to the line very efficiently, when it is close to the line. The linear policy will result in control actions that are too large far from the line. Due to the nonlinear correction the size of these actions are reduced. In figure 2 we see that this correction is for parts of the state space that were visited during the generation of the train set. For other parts of the state space the control actions are still too large.

The Neural Q-Learning approach results in a control policy that is linear in most parts of the state space. In parts of the state space where the training set indicates that a correction is required, the control action deviates from the linear policy. Note that we trained the Q-function on only a very small training set of 4×57 samples, acquired with only 80 seconds driving. Also note that this experiment represents only one episode. From a linear policy that cannot do the task properly

δ_0	-1.75	-0.75	0.75	1.75
J	82.362	11.600	11.471	82.189

Table 1: The total costs for the different starting positions when using the policy from figure 2.

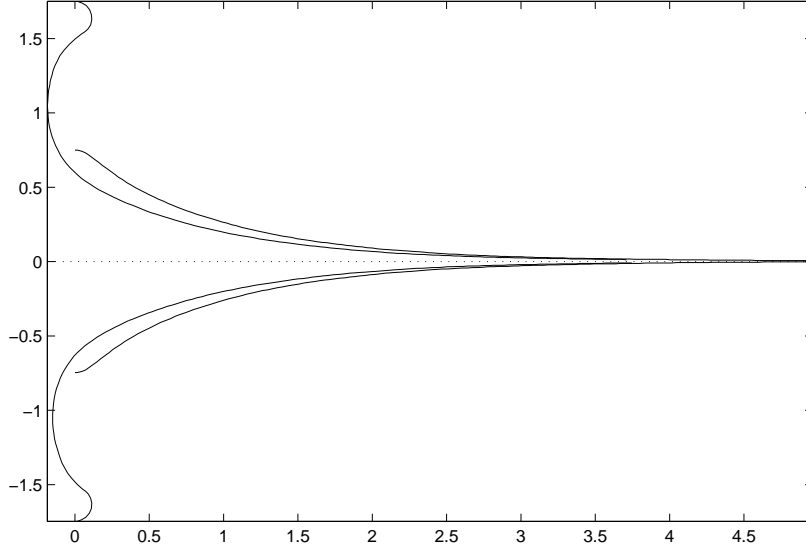


Figure 3: The trajectories in world space for the robot using the policy from figure 2.

a nonlinear function is derived that is capable of performing the task! We believe this is the first neural RL experiment where learning is based on real data from the system and that no model of the system is used.

5 Conclusion

In this paper we proposed Neural Q-Learning as a continuous state-action space equivalent of the discrete state-action space Q-Learning algorithm. In Neural Q-Learning only one function approximator is used to approximate the expected sum of future reinforcements. The new control policy is derived directly from the approximator by first approximating a global quadratic function, resulting in a linear policy. The linear policy is used to compute the state and action around which a locally a new policy is approximated. Unlike computing the greedy policy, our new policy is always smooth. The method is model-free and does not require a large number of training samples. This makes this approach more appropriate for direct use on real systems than other continuous state/action space reinforcement learning approaches. The experiment with the real robot confirms this.

References

- [1] Narendra, K. and Parthasarathy, K. (1990). Identification and control for dynamic systems using neural networks. *IEEE transaction on Neural networks*.
- [2] Choi, J. and Farrell, J. (2000). Nonlinear adaptive control using networks of piecewise linear approximators. *IEEE Transactions on Neural Networks*, 11(2):390–401.
- [3] Zhang, Y., Peng, P., and Jiang, Z. (2000). Stable neural controller design for unknown nonlinear systems using backstepping. *IEEE Transactions on Neural Networks*, 11(6):1347–1360.

- [4] Werbos, P. (1992). Approximate dynamic programming for real-time control and neural modeling. In White, D. A. and Sofge, D. A. W., editors, *Handbook of Intelligent Control: Neural, Fuzzy, and Adaptive Approaches*. Van Nostrand Reinhold.
- [5] Prokhorov, D. and Wunch II, D. (1997). Adaptive critic design. *IEEE transactions on Neural Networks*.
- [6] Sofge, D. and White, D. (1990). Neural network based process optimization and control. In *Proceedings of the 29th conf. on Decision and Control*.
- [7] Sofge, D. and White, D. (1992). *Handbook of Intelligent Control, Neural Fuzzy, and Adaptive Approaches*, chapter Applied learning: optimal control for manufacturing. Van Nostrand Reinhold.
- [8] Anderson, C., Hittle, D., Katz, A., and Kretchmar, R. (1996). Synthesis of reinforcement learning, neural networks, and pi control applied to a simulated heating coil. *Journal of Artificial Intelligence in Engineering*.
- [9] Riedmiller, M. (1996). Application of sequential reinforcement learning to control dynamical systems. In *Proceedings of the IEEE International Conference on Neural networks*.
- [10] Schram, G., Kröse, B., Babuska, R., and Krijgsman, A. (1996). Neurocontrol by reinforcement learning. *Journal A (Journal on Automatic Control), Special Issue on Neurocontrol*, 37(3):59–64.
- [11] Miller, S. and Williams, R. (1995). *Applications of Artificial Neural Networks*, chapter Temporal Difference Learning: A Chemical Process Control Application. Kluwer.
- [12] Riedmiller, M. (1999). Concepts and facilities of a neural reinforcement learning control architecture for technical process control. In *Neural Computing and Application Journal*.
- [13] Eaton, P., Prokhorov, D., and Wunch II, D. (2000). Neurocontroller alternatives for "fuzzy" ball-and-beam systems with nonuniform nonlinear friction. *IEEE transactions on Neural Networks*.
- [14] Sutton, R. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*.
- [15] Watkins, C. (1989). *Learning from Delayed Rewards*. PhD thesis, University of Cambridge.
- [16] Watkins, C. and Dayan, P. (1992). Technical note: Q learning. *Machine Learning*.
- [17] Bradtke, S. (1993). Reinforcement learning applied to linear quadratic regulation. In *Advances in Neural Information Processing Systems*.
- [18] Bradtke, S., Ydstie, B., and Barto, A. (1994). Adaptive linear quadratic control using policy iteration. Technical report, University of Massachusetts.
- [19] Landelius, T. (1997). *Reinforcement learning and Distributed Local Model Synthesis*. PhD thesis, Linköping University.
- [20] ten Hagen, S. (2001). *Continuous State Space Q-Learning for Control of Nonlinear Systems*. PhD thesis, Computer Science Institute, University of Amsterdam, The Netherlands.