Self organising maps for value estimation to solve reinforcement learning tasks

A. Kleiner,B. Sharp, O. Bittel
Staffordshire University

May 11, 2000

Abstract

Reinforcement learning has been applied recently more and more for the optimisation of agent behaviours. This approach became popular due to its adaptive and unsupervised learning process. One of the key ideas of this approach is to estimate the value of agent states. For huge state spaces however, it is difficult to implement this approach. As a result, various models were proposed which make use of function approximators, such as neural networks, to solve this problem. This paper focuses on an implementation of value estimation with a particular class of neural networks, known as self organising maps. Experiments with an agent moving in a "gridworld" and the autonomous robot Khepera have been carried out to show the benefit of our approach. The results clearly show that the conventional approach, done by an implementation of a look-up table to represent the value function, can be out performed in terms of memory usage and convergence speed.

Keywords: self organising maps, reinforcement learning, neural networks

1 Introduction

In this paper we discuss the credit assignment problem, and the reinforcement learning issue associated with rewarding an agent upon successful execution of a set of actions. Figure 1 illustrates the interaction between an agent and its environment. For every action, the agent performs in any state s_t, it receives an immediate reinforcement r_t and the percepts of the successor state s_{t+1}. This immediate reinforcement depends on the performed action and on the new state taken as well. For example, an agent searching for an exit in a maze might be rewarded only if this exit is reached. If this state is found, it is obvious that all former states, which contributed to this success, have to be rewarded as well.

Reinforcement learning is one solution for the credit assignment problem. The idea of reinforcement learning grew up within two different branches. One branch focused on learning by trial and error, whereas the other branch focused on the problem of optimal control. In the late 1950s Richard Bellman introduced his approach of a value function or a "optimal return function" to solve the problem of optimal control (Bellman 1957). Methods to solve this equation are nowadays known as dynamic programming. This paper focuses on a generalization of these methods, known as *temporal difference* methods, which has been introduced in 1988 by Richard Sutton (Sutton 1988). These methods assign, during an iterative procedure, a credit to every state in the state space, based on a calculated difference between these states. Roughly speaking this implies, that if a future state is desirable, the present state is as well. Sutton introduced the parameter λ to define, how far in the future states have to be taken into account, thus this generalisation is named $TD(\lambda)$. Within this paper, however, the simpler case $TD(0)$[1] is used, which only considers one successor state during a temporal update.

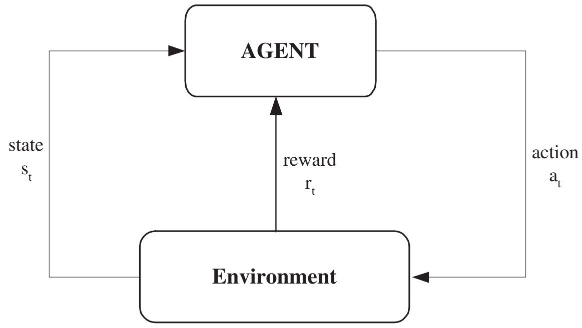

Figure 1: The agent-environment interaction in reinforcement learning

Current methods for the "optimal return function" suffer, however, under what Bellman called "the curse of dimensionality", since states from real world problems consist usually of many elements in their vectors. Therefore it makes sense to use function approximators, such as neural networks, to learn the "optimal return function".

Successful applications of reinforcement learning with neural networks are testified by many researchers. Barto and Crites (Barto & Crites 1996) describe a neural reinforcement learning approach for an elevator scheduling task. Thrun (Thrun 1996) reports the successful learning of basic control procedures of an autonomous robot. This robot learned with a neural Q learning implementation, supported by a neural network. Another successful implementation was done by Tesauro at IBM (Tesauro 1992). He combined a feed-forward network, trained by backpropagation, with $TD(\lambda)$ for the popular backgammon game. This architecture was able to find strategies using less inducement and has even defeated

[1] Also known as the value iteration method

champions during an international competition.

Besides this successful examples, which are all based on neural networks using backpropagation, there is more and more evidence, that architectures based on backpropagation converge slowly or not at all. Examples for such problematic tasks are given by (Boyan & Moore 1995) and (Gordon 1995). This difficulties arise due to the fact that backpropagation networks store information implicit. This means for the training that every new update affects former stored information as well. A convergence cannot be guaranteed anymore, since the original approach of reinforcement learning is supposed to be used with an explicit look-up table. Therefore our approach makes use of a neural network architecture with explicit knowledge representation, known as *self organising maps*.

This paper will discuss the problems associated with the use of self organising maps (SOMs) to learn the value function and describe our modified approach to SOM applied to two problems.

2 Self organizing maps (SOM)

Self organizing maps were firstly introduced by Teuvo Kohonen in 1982 (Kohonen 1982). These kind of neural networks are a typical representative of unsupervised learning algorithms. During the learning process particular neurons are trained to represent clusters of the input data. The achieved arrangement of these clusters is such, that similar clusters, in terms of their Euclidean distance, are near to each other and different clusters are far from each other. Hence, the network builds up a topology depending on the data given to it from the input space. This topology is equal to the statistical distribution of the data. Areas of the input space, which are supported by more samples in the data, are represented more detailed than areas supported with less samples.

SOM architecture
A SOM usually consists of a two dimensional grid of neurons. Every neuron is connected via its weights to the input vector, where one weight is spent for every element of this vector. Before the training process, values of these weights are set arbitrary. During the training phase, however, the weights of each neuron are modified to represent clusters of the input space.

Mapping of pattern
After a network has been trained, a cluster for an input vector can be identified easily. To find the neuron, representing this cluster, the Euclidean distance between this vector and all weight sets of the neurons on the SOM has to be calculated. The neuron with the shortest distance represents this vector most precisely and is thus named as "winner" neuron. The Euclidean distance is calculated after the following equation:

$$d_i = \sum_{k=1}^{n} (w_{ik} - x_k)^2 \quad (1)$$

Where w_{ik} denotes the ith neurons kth weight and x_k the kth element of the input vector.

Learning of clusters.
The learning process takes place in a so called offline learning. During a fixed amount of repetitions, called epochs, all patterns of the training data are propagated through the network. At the beginning of the learning process, values of the weights are arbitrary. Therefore for every input vector x_i a neuron u_i is chosen to be its representative by random as well. To manifest the structure of the map, weights are moved in direction to their corresponding input vector. After a while the representation of

input vectors becomes more stable, since the Euclidean distance of each winner neuron decreases.

To build a topological map, it is important to adjust the weights of neighbours around the neuron as well. Therefore a special neighbourhood function has to be applied. This function should return to the winner neuron a value of *1* and to neurons with increasing distance to it a decreasing value down to zero. Usually the "sombrero hat function" or the Gaussian function is used for that. By use of the Gaussian function, the neighbourhood function is:

$$h_{ci} = e^{-\frac{|n_c - n_i|^2}{2\sigma^2}} \quad (2)$$

Where n_c denotes the winner neuron and n_i any neuron on the Kohonen Layer. The standard deviation σ denotes the neighbourhood radius.

For every input vector the following update rule will be applied to every neuron on the SOM:

$$\triangle w_{ik} = \eta \cdot h_{ci} \cdot (x_k - w_{ik}) \quad (3)$$

Where η denotes the step size.

By this update rule, weights are updated in discrete steps, defined by the step size η. The nearer neurons are to a chosen winner neuron, the more they are affected by the update. Thereby neighbouring neurons represent similar clusters, which leads to a topological map.

The advantage of SOMs is that they are able to classify samples of an input space unsupervised. During the learning process, the map adapts its structure to the input data. Depending on the data, the SOM will build clusters and order them in an appropriate manner. One disadvantage of SOMs is, however, the necessity to define a representative subset of the input space and train it over many epochs. After the SOM is trained it is only possible to add a new cluster to the representation by repeating the learning process with the old training set and the new pattern.

3 Reinforcement Learning

Classical approaches for neural networks tend to make use of specific knowledge about states and their corresponding output. This given knowledge is used for a training set and after the training it is expected to gain knowledge about unknown situations by generalization. However for many problems in the real world an appropriate training set can't be generated, since the "teacher" doesn't know the specific mapping. Nevertheless, it seems to be easy for the teacher to assess this mapping for every state. When learning to drive a car, for example, one is not told how to operate the car controls appropriately, the teacher, however, bridges the gap in learning using appropriate feedback, which improves the learning process and leads finally to the desired mapping between states and actions.

The Reinforcement problem

The task of reinforcement learning is to use rewards to train an agent to perform successful functions. Figure 1 illustrates the typical interaction between agent and environment. The agent performs actions in its environment and receives a new state vector, caused by this action. Furthermore the agent gets feedback of whether the action was adequate. This feedback is expressed by immediate rewards, which also depend on the new state taken by the agent. A chess playing agent, for example, would receive a maximum immediate reward if it reaches a state where the opponent cannot move the king any more. This example illustrates very clearly the *credit assignment prob-*

lem. The reward achieved in the last board position is achieved after a long chain of actions. Thus all actions, done in the past, are responsible for the final success and therefore also have to be rewarded. For this problem several approaches have been proposed; a good introduction to these is found in the book by Barto and Sutton (Barto & Sutton 1998). This paper, however, focuses on one of these approaches, which is the value iteration method, also known as $TD(0)$.

Rewards [2]
In reinforcement learning, the only hints given to the successful task are immediate reinforcement signals. These signals usually come directly from the environment or can be generated artificially by an assessment of the situation. If they are generated for a problem, they should be chosen economically. Instead of rewarding many sub-solutions of a problem, only the main goal should be rewarded. For example, for a chess player agent it would not necessarily make sense to reward the taking of the opponent's pieces. The agent might find a strategy which optimises the collection of pieces of the opponent, but forgets about the importance of the king. Reinforcement learning aims to maximise the achieved reinforcement signals over a long period of time.

In some problems no terminal state can be expected, as in the case of a robot driving through a world of obstacles and learning not to collide with them. An accumulation of rewards would lead to an infinite sum. For the case where no terminal state is defined, we have to make use of a discount factor to ensure that the learning process will converge. This factor discounts rewards which might be expected in the future [3], and thus can be computed as follows:

$$R_T = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + ...$$
$$= \sum_{k=0}^{T} \gamma^k r_{t+k+1} \quad (4)$$

Where R_T denotes the rewards achieved during many steps, γ the discount factor and r_t the reward at time t. For $T = \infty$ it has to be ensured that $\gamma < 1$

The direct goal for reinforcement learning methods is to maximise R_T. To achieve this goal, however, a prediction for the expectation of rewards in the future is necessary. Therefore we need a mapping from states to their corresponding maximum expectation. As known from utility theory, this mapping is defined by the value function [4].

The value function $V^*(s)$
In order to maximise rewards over time, it has to be known for every state, what future rewards might be expected. The optimal value function $V^*(s)$ provides this knowledge with a value for every state. this return value is equal to the accumulation of maximum rewards from all successor states. Generally this function can be represented by a look-up table, where for every state an entry is necessary. This function is usually unknown and has to be learned by a reinforcement learning algorithm. One algorithm, which updates this function successive, is value iteration.

Value iteration
In contrast to other available methods, this method updates the value function after every seen state and thus is known as value iteration. This update can be imagined with an

[2] Rewards also include negative values which are equal to punishments

[3] These expectations are based on knowledge achieved in the past

[4] In terms of the utility theory originally named utility function

agent performing actions and using received rewards, caused by this actions, to update values of the former states. Since the optimal value function returns for every state the accumulation of future rewards, the update of a visited state s_t has to include the value of the successor state s_{t+1} as well. Thus the value function is learned after the following iterative equation:

$$V_{k+1}(s_t) := r(s_t, a_t) + V_k(s_{t+1}) \quad (5)$$

Where V_{k+1} and V_k denote the value function before and after the update and $r(s_t, a_t)$ refers to the immediate reinforcement achieved for the transition from state s_t to state s_{t+1} by the chosen action a_t. While applying this method, the value function approximates more and more until it reaches its optimum. That means that predictions of future rewards become successively more precise and actions can be chosen with maximum future rewards.

There is an underlying assumption that the agent's actions are chosen in an optimal manner. In value iteration, the optimal choice of an action can be done after the *greedy-policy*. This policy is, simply after its name, to chose actions which lead to maximum rewards. For an agent this means, to chose from all possible action $a \in A$ that one, which returns after equation (5) the maximum expectation. However we can see, that after equation (5) the successor state s_{t+1}, caused by action a_t, must be known. Thus a model of the environment is necessary, which provides for state s_t and action a_t the successor state s_{t+1}:

$$s_{t+1} = f(s_t, a_t) \quad (6)$$

Exploration

If all actions are chosen after the *greedy-policy*, it might happen that the learning process results in a sub-optimal solution. This is because actions are always chosen by use of knowledge gathered so far. This knowledge however can lead to a local optimal solution in the search space, where global optimal solutions never can be found. Therefore it makes sense to chose actions, with a defined likelihood, arbitrary. The policy to chose action by a propability of ε arbitrary, is called *ε-greedy* policy. Certainly there is a trade-off between exploration and exploitation of existing knowledge and the optimal adjustment of this parameter depends on the problem domain.

Implementation of Value Iteration

So far, the algorithm can be summarised in the following steps:

- select the most promising action a_t after the *ε-greedy policy*

  $$a_t = \arg\min_{a \in A(s_t)}(r(s_t, a) + V_k(f(s_t, a)))$$

- apply a_t in the environment

  $$s_t \Longrightarrow s_{t+1}$$

- adapt the value function for state s_t

  $$V_{k+1}(s_t) := r(s_t, a_t) + V_k(s_{t+1})$$

In theory, this algorithm will definitely evaluate an optimal solution for problems, such as defined at the beginning of this section. A problem to reinforcement learning however, is its application to real world situations. That is because real world situations are usually involved with huge state spaces. The value function should provide every state with an appropriate value. But most real world problems come up with a multi-dimensional state vector. The state of a robot, for example, whose task is to find a strategy to avoid obstacles, can be described by the state of its approximity sensors.

If every sensor would have a possible return value of 10 Bit and the robot itself owns eight of these sensors, the state space would consist of $1.2 * 10^{24}$ different states, emphasizing the problem of tractability in inferencing.

On the other hand, it might happen, that during a real experiment with a limited time, all states can never be visited. Thus it is likely, that even after a long training time, still unknown states are visited. But unfortunately the value function can't provide a prediction for them.

4 Modified SOM to learn the value function

The two problems previously identified for reinforcement learning, can be solved using function approximators. Neural Networks, in particular, provide the benefit of compressing the input space and furthermore the learned knowledge can be generalised. This means for the value function, that similar states will be evaluated by one neuron. Hence also unknown states can be generalized and evaluated by the policy. For this purpose the previously introduced model of self organising maps has been taken and modified.

Modification to the architecture
Usually SOMs are used for classification of input spaces, for which no output vector is necessary. To make use of SOMs as function approximator, it is necessary to extend the model by an output value. Such modifications have been first introduced by Ritter and Schulten in connection with reflex maps for complex robot movements (Ritter & Schulten 1987). The modification used here is, that every neuron of the Kohonen layer is expanded by one weight, which connects it to the scalar output. This output is used for the value function. The goal is to get a generalisation for similar situations. To achieve this, the output weights have to be trained with a neighbourhood function as well. Therefore the output weights are adapted with the following rule:

$$\delta w_i = \eta_2 h_{ci}(y - w_i) \quad (7)$$

Where η_2 is a second step size parameter and h_{ci} the same neighbourhood function as used for the input weights and y the desired output of the network.

Modification to the algorithm
As remarked previously, the learning algorithm for SOMs is supposed to be applied "offline" with a specific training set. The application of value iteration however, is an "online" process, where the knowledge increases iteratively. To solve this contradiction, the learning process of the SOM has been divided into two steps:

- First step: pre-classification of the environment
- Second step: execution of reinforcement learning with improvement of classification for visited states

For the first step a representative sample of the whole state space is necessary, to build a appropriate map of the environment. This sample will be trained, until the structure of the SOM is adequate to classify states of the problems state space. During the execution of the second step the reinforcement learning algorithm updates states with their appropriate values. These states are classified by SOMs, where one neuron is chosen as winner. The corresponding output weights of this neuron are changed to the value, calculated by the reinforcement learning algorithm. Furthermore, the output values of the neighbourhood of this neuron are modified as well to achieve the effect of generalisation.

Usually the states, necessary to solve the problem, are a subset of the whole state space. Thus the SOM has to classify only this subset, using a pre-classification. During the application of reinforcement learning, this classification will improve, since for every state visited, its representation is strengthen. States, which are visited more frequently and thus are more important for the solution of the problem, will achieve a better representation than those unimportant states, which are visited less.

5 Experiments and results

5.1 The path-planning problem

This section describes the application of our modified SOM with reinforcement learning for solving the path planning problem. The problem is to find the shortest path through a maze or simply a path on a map. For the experiment described here, a computer simulation of a "girdworld" has been taken (see Figure 2). The gridworld is represented by a two dimensional arrangement of positions. Wall piece or obstacles can occupy these positions and the agent therefore can't cross them. Other positions however, are free to its discovery. For the experiment, the upper left corner is defined as start position and the lower right corner as end position. The agent's task is to find the shortest path between these two positions, while avoiding obstacles on its way.

Due to the fact, that the agent is supposed to learn the "cheapest" path, it is punished for every move with -1 and rewarded with 0 if it reaches the goal. Beside these reinforcement signals, the agent gets no other information, about where it can find the goal or which direction should be preferred. If it faces an obstacle, the possible actions are reduced to that actions, which lead to free positions around.

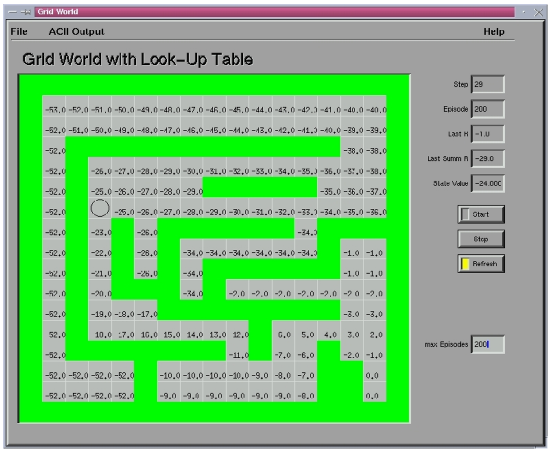

Figure 2: The gridworld experiment

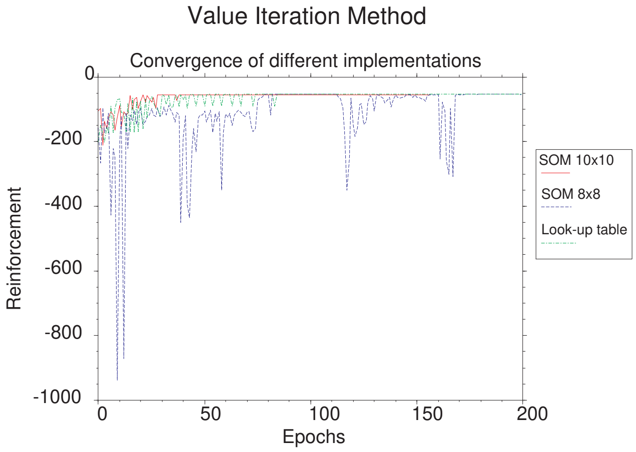

Figure 3: Achieved rewards, during learning of a behaviour for the gridworld experiment

Two implementations of a modified SOM with 8x8 neurons and 10x10 neurons have been used. For comparison, the experiment has been carried out with a look-up table, where every entry represents a state, as well. This look-up table consists of 289 entries, due to the used grid size is 17x17 positions.

Results
The result of this experiment is shown in figure 3. In this graph the achieved rewards for each implementation after every episode can be seen. The optimal path is found, if the accumulated reinforcement during one episode is *-53*, since the agent needs at least 53 steps to reach its goal. In the graph can be seen, that the implementation of the modified SOM with 10x10 neurons leads to a faster result than the look-up table. After 30 episodes the agent, equipped with the modified SOM, found the cheapest path.

5.2 Learning obstacle avoidance with a robot

A common problem in robotics is the autonomous drive of a robot. For such a drive there are various processes. One process might bring it to a far destination, lead by a path finding algorithm. For simple movement, however, a process is necessary to avoid obstacles. In this problem, it is very difficult to define appropriate actions for particular situations. On the other hand, we can easily assess the resulting actions. Therefore this problem seems to be appropriate for the reinforcement learning approach.

In this experiment the autonomous miniature robot Khepera, which was developed at the EPFL in Lausanne, has been used (see figure 4). This 5 cm huge robot is equipped with eight approximity sensors, where two are mounted at the front, two at the back, two at the side and two in 45° to the front. These sensors give a return value between 0 and 1024, which is corresponding to a range of about 5 cm. The robots drive consists of two servo motors, which can turn the two wheels with 2 m per second in negative and positive directions. By this configuration, the robot is able to do 360° rotations without moving in x or y direction. Therefore

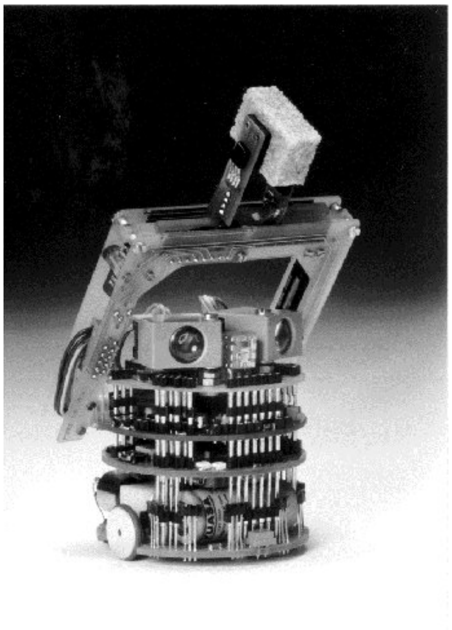

Figure 4: Autonomous robot Khepera

the robot is very manoeuvrable and should be able to deal with most situations. Furthermore the robot is equipped with two rechargeable batteries, which enable it to drive for about 20 minutes autonomously. For execution of programs, there also exists a CPU from Motorola and a RAM area of 512KB on the robot.

Experiment
Due to the fact, that for value iteration a model of the environment is required, the robot has been first trained using a computer simulation. Afterwards the experiment continued on a normal office desk, where obstacles and walls were built up with wooden blocks.

In the reinforcement learning algorithm, the state of the robot was represented by the eight sensor values. The allowed actions have been reduced to the three actions: left turn, right turn and straight forward. Also the reinforcement signals were chosen in the most trivial way. If the robot collides with an obstacle, it

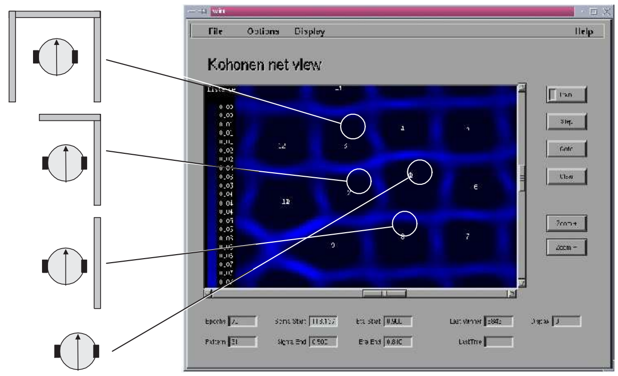

Figure 5: A learned classification of the sensor space

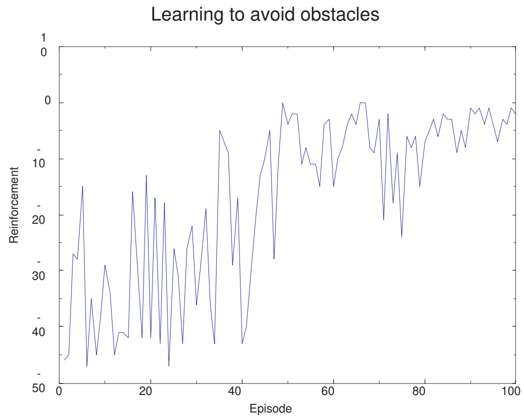

Figure 6: Collisions during the autonomous learning of an obstacle avoidance strategy

gets a punishment of -1, otherwise a reward of 0. The experiment has been carried out over multiple episodes. One episode has been limited to 50 steps. Therefore the discount factor γ has been set to 1.0. For exploration purposes the factor ε has been adjusted to 0.01, which is equal to the probability actions are chosen arbitrary. Concerning to the state vector, the input vector of the SOM consists of eight elements as well. For the Kohonen Layer an arrangement of 30x30 neurons has been chosen.

Before the application of the reinforcement learning algorithm, the SOM had to be preclassified. Therefore a training set of typical situations from an obstacle world has been trained over 90 epochs. With the help of visualisation tools it could be ensured that the situations are adequately classified, as illustrated in figure 5.

During the episodes of the value iteration method, identified situations were relearned with a small neighbourhood of $\sigma = 0.1$ and also small learning step rate of $\eta = 0.3$.

Results
The result of the learning process of the robot can be seen in figure 6. In this graph the accumulated rewards for every episode are shown. Hence for every collision the robot has been punished with *-1*, the reinforcement for every episode is equal to the caused collisions. After 45 episodes the number of collisions became significantly less. During the early episodes, the value of achieved reinforcement signals sways strongly. This results from the fact, that after the robot overcame a situation, it encountered a new situation again, where another behaviour had to be learned as well. As we see in the graph, the robot learned to manage most situations after a sufficient time of proceeding. After the training the learned control has been tested on the real robot. Although the achieved behaviour was not elegant, it proved, that the robot obviously learned the ability to avoid obstacles.

6 Conclusion

The problem of a huge state space in real world applications and the fact that mostly some but unlikely all states of a state space can be encountered, have been tackled by use of a modified SOM. The SOMs abilities to compress the input space and generalize from known situations to unknown made it possible to achieve a reasonable solution. However, it was necessary to split the standard algorithm for SOMs into two parts. Once learning by a pre-classification and twice learning during value iteration. With this modification, the algorithm can be applied to an online learning process, which is given by the value iteration method.

The applied modifications to the standard SOM have been evaluated within the gridworld example and the problem of obstacle avoidance of an autonomous robot. These experiments showed two main advantages of the modified SOM to a standard implementation with a look-up table. First, the example of obstacle avoidance proved that even for enormous state spaces, a strategy can be learned. Second, the path finding example showed, that the use of a modified SOM can lead to faster results, since the agent is able to generalize situations instead of learning a value for all of them.

For the Value Iteration algorithm, applied to the experiments described here, a model of the environment is necessary. For real world problems, such as the problem of obstacle avoidance, however, an appropriate model can hardly be provided. Sensor signals are normally noisy or even it might be that a sensor is damaged or don't work properly. Thus it is recommendable to use another reinforcement learning implementation, which makes it not any more necessary to provide a model of the environment. One commonly used variant of reinforcement learning is the Q-Learning. In this algorithm states are represented by the tuple of state and action, thus a model of the environment is not required. We belief that a combination of Q-Learning with the modified SOM, proposed in this paper, yields better results.

One of the big disadvantages encountered however, is that the modification of the SOM during the second step changes the generalisation behaviour of the network. If states are relearned frequently with a small neighbourhood function, the learned knowledge becomes too specific and generalisation is reduced. To overcome this problem, it would be necessary to relearn the complete structure of the SOM with an offline algorithm. Unfortunately, experience in terms of training examples are lost after the online process. A possible solution and probably subject of another work, is to store "critical" pattern temporarily during the online process by dynamic neurons. With "critical" pattern we mean those, which come with a far Euclidean distance to all existing neurons in the network and thus their classification by this neurons would not be appropriate. Given the set of these dynamically allocated neurons and the set of neurons on the SOM, a new arrangement with better topological representation can be trained by an offline algorithm[5]. The execution of this offline algorithm can be done during a phase of no input to the learner and is motivated by a human post processing of information, known as REM phase.

References

Barto, A. & Crites, R. (1996), Improving elevator performance using reinforcement learning, *in* M. C. Hasselmo, M. C. Mozer & D. S. Touretzky, eds, 'Advances in Neural Information Processing Systems', Vol. 8.

[5] For example the standard learning algorithm for SOMs

Barto, A. & Sutton, R. (1998), *Reinfocement Learning - An Introduction*, MIT Press, Cambridge.

Bellman, R. E. (1957), *Dynamic Programming*, Princeton University Press, Princeton.

Boyan, A. J. & Moore, A. W. (1995), Generalization in Renforcemen Learning: Savely Approximating the Value Function, *in* T. K. Leen, G. Tesauro & D. S. Touretzky, eds, 'Information Processing Systems', Vol. 7, MIT Press, Cambridge MA.

Gordon, G. (1995), Stable function approximation in dynamic programming, *in* 'Proceedings of the 12th International Conference on Machine Learning', Morgan Kaufmann, San Fransisco, Calif., pp. 261–268.

Kohonen, T. (1982), Self-Organized Formation of Topologically Correct Feature Maps, *in* 'Biol. Cybernetics', Vol. 43, pp. 59–69.

Ritter, H. & Schulten, K. (1987), Extending Kohonen's Self-Organizing Mapping Algorithm to Learn Ballistic Movements, *in* 'Neural Computers', Springer Verlag Heidelberg, pp. 393–406.

Sutton, R. (1988), Learning to predict by the methods of temporal differences, *in* 'Machine Learning', Vol. 3, pp. 9–44.

Tesauro, G. (1992), Practical issues in temporal difference learning, *in* 'Machine Learning', Vol. 8, pp. 257–277.

Thrun, S. (1996), *Explanation-based neural network learning: A lifelong learning approach*, Kluwer Academic Publishers, Bosten.