

Deep Learning for Reinforcement Learning in Pacman

Deep Learning für Reinforcement Learning in Pacman
Bachelor-Thesis von Aaron Hochländer aus Wiesbaden
Juli 2014



TECHNISCHE
UNIVERSITÄT
DARMSTADT



Deep Learning for Reinforcement Learning in Pacman
Deep Learning für Reinforcement Learning in Pacman

Vorgelegte Bachelor-Thesis von Aaron Hochländer aus Wiesbaden

1. Gutachten: Prof. Dr. Jan Peters
2. Gutachten: M.Sc. Roberto Calandra
3. Gutachten: Dr. Gerhard Neumann

Tag der Einreichung:

Please cite this document with:

URN: urn:nbn:de:tuda-tuprints-38321

URL: <http://tuprints.ulb.tu-darmstadt.de/id/eprint/3832>

Dieses Dokument wird bereitgestellt von tuprints,

E-Publishing-Service der TU Darmstadt

<http://tuprints.ulb.tu-darmstadt.de>

tuprints@ulb.tu-darmstadt.de



Die Veröffentlichung steht unter folgender Creative Commons Lizenz:

Namensnennung – Keine kommerzielle Nutzung – Keine Bearbeitung 2.0 Deutschland

<http://creativecommons.org/licenses/by-nc-nd/2.0/de/>

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

In der abgegebenen Thesis stimmen die schriftliche und elektronische Fassung überein.

Darmstadt, den 2. Juli 2014

(Aaron Hochländer)

Thesis Statement

I herewith formally declare that I have written the submitted thesis independently. I did not use any outside support except for the quoted literature and other sources mentioned in the paper. I clearly marked and separately listed all of the literature and all of the other sources which I employed when producing this academic work, either literally or in content. This thesis has not been handed in or published before in the same or similar form.

In the submitted thesis the written copies and the electronic version are identical in content.

Darmstadt, July 2, 2014

(Aaron Hochländer)

Abstract

The curse of dimensionality is a common problem for numerous machine learning methods when they are confronted with high-dimensional data. In order to deal with this problem a popular approach is the introduction of features vectors as a high-level abstraction of the input data. Designing task-specific features by hand is often very challenging and not very cost-efficient since a set of features that provides a good representation of the data for a task is usually worthless for all other tasks. Therefore, automatic feature learning is desirable. An interesting and successful approach for processing high-dimensional data hierarchically, that goes by the name of deep learning, was introduced in 2006 and has gained a lot of popularity since then. In this thesis, the applicability of the combination of deep learning and reinforcement learning is analysed on the task of learning to play the game Pacman. In order to achieve this, initially a feature representation of observations of Pacman game states is learned using a deep learning method. This feature representation is subsequently used to approximate the value functions of two reinforcement learning algorithms, Q-learning and least-squares policy iteration. While Q-learning yielded disappointing experimental results, the learning performance when using the least-squares policy iteration algorithm shows that deep learning can indeed be successfully used to learn features for reinforcement learning methods.

Zusammenfassung

Der exponentielle Anstieg der Größe des Zustandsraums ist ein häufiges Problem für viele Methoden des maschinellen Lernens, wenn mit hochdimensionalen Daten gearbeitet wird. Als Lösungsansatz für dieses Problem werden häufig Merkmalsvektoren eingeführt, die eine Abstraktion der Eingangsdaten auf hoher Ebene bilden. Diese charakterisierenden Merkmale für jede neue Aufgabe von Hand zu konzipieren erweist sich häufig als schwierig und ist zudem nicht sehr kosteneffizient, da ein Merkmalsvektor, der für die Daten einer Aufgabe eine sehr gute Repräsentation liefert, in der Regel für alle anderen Aufgaben nutzlos ist. Aus diesem Grund ist das automatische Lernen von Merkmalen wünschenswert. Ein interessanter und erfolgreicher Ansatz um hochdimensionale Daten hierarchisch zu verarbeiten, der Tiefes Lernen genannt wird, wurde 2006 eingeführt und wird seitdem immer populärer. In dieser Thesis wird die Eignung der Kombination aus Tiefem Lernen und Verstärkendem Lernen analysiert indem versucht wird, das Spiel Pacman zu lernen. Um dies durchzuführen wird zunächst ein Merkmalsvektor, der Beobachtungen von Zuständen des Pacman-Spiels repräsentieren soll, mithilfe einer Methode des Tiefen Lernens konstruiert. Dieser Merkmalsvektor wird daraufhin verwendet um die Wertefunktionen von zwei Algorithmen des Bestärkenden Lernens, *Q-learning* und *least-squares policy iteration*, zu approximieren. Während Q-learning enttäuschende Resultate Versuchsergebnisse einbrachte zeigt das Abschneiden des least-squares policy iteration Algorithmus, dass Tiefes Lernen tatsächlich erfolgreich angewendet werden kann, um automatisch Merkmalsvektoren für Methoden des Bestärkenden Lernens zu erstellen.

Acknowledgments

First of all, I would like to thank both of my supervisors Roberto Calandra and Dr. Gerhard Neumann. They assisted me in any possible way and I am very grateful for all of their advice and guidance during the process of my research and the writing of this thesis.

I also want to thank Prof. Dr. Jan Peters for providing me with this interesting topic of research and the whole Intelligent Autonomous System Lab for allowing me to use their computers for carrying out the experiments for this thesis.

Additionally, I am very grateful to my family for all their support throughout my studies and also for motivating me at times when it was necessary.

Contents

1	Introduction	2
1.1	Related Work	2
2	Reinforcement Learning	4
2.1	Markov Decision Processes	4
2.2	Q-Learning	5
2.3	Q-Learning with linear function approximation	6
2.4	Least-Squares Policy Iteration	6
3	Feature Learning	8
3.1	Deep Learning	8
3.2	Autoencoders	9
3.3	Stacked Autoencoders	9
4	Experiments	11
4.1	Setup	11
4.2	Results	11
5	Conclusion	20
	Bibliography	23

Figures and Tables


List of Figures

2.1	Visualisation of the reinforcement learning process using an MDP to describe the agent's interaction with its environment [1]	4
3.1	A multilayer perceptron with several hidden layers that provide more and more abstract representations of the input. The final hidden layer is used as input for a classifier [2].	8
4.1	The two Pacman maps on which the learning algorithms were run	12
4.2	The learning curves for learning Pacman on two training maps using the benchmark algorithms	15
4.3	Results on applying Q-learning using stacked autoencoders as feature function on the small grid	16
4.4	Learning curves and a weight progression for very long Q-learning experiments	17
4.5	Consequences for the rewards and weights for different forgetting factors	18
4.6	Learning curves for LSTDQ on both maps for different feature functions	19

List of Tables

4.1	Reconstruction errors for different 20-dimensional feature representations	13
4.2	Reconstruction errors for three layers of different dimensionalities	13
4.3	Reconstruction errors for different layer combinations on the medium grid	14
4.4	Averages of the best and last rewards during the learning process as well as the average rewards of the final policies for the benchmark algorithms	15
4.5	Best and final rewards for the experiments that led to Figure 4.3a	16
4.6	Comparison of the final policies for the three tested forgetting factors	18
4.7	Comparison of best and final rewards for all the different experiments	19

1 Introduction

Nowadays, the field of machine learning has become an enormous research area and the amount of different algorithms for different tasks that have already been developed are countless. However, independent from the specific task and method they all have one common problem. Machine learning generally requires some form of training data or teaching by an expert in order to initiate the learning process. As the complexity of a task rises, so typically does the dimensionality of the data because the data has to contain additional information. Otherwise, the quality of the representation of the task would decrease. Therefore, challenging machine learning tasks require high-dimensional input data. 

Ideally, in machine learning the aim is to be able to generalize over a lot of different possible cases in the input space, so that after proper training the behavior for previously unseen new data can be adequately predicted. Hence, the more coverage of the space of possible inputs a set of training data provides, the more useful it is for learning. The resulting problem for a high dimensional input space, which is known as the curse of dimensionality, is that the amount of training data that is necessary for a good learning performance becomes huge. In fact, the required data for maintaining the coverage of the space increases exponentially with the dimensionality of the space. This has two direct consequences. First of all, obtaining training data can be expensive for a lot of tasks, especially for supervised learning tasks where the input data has to be labeled with the correct output. Additionally, an exponential increase in training data is always accompanied by a significant increase in computation time and memory requirements.

The motivation for the combination of reinforcement learning and deep learning in this thesis is a direct consequence of these issues. Reinforcement learning as opposed to most other machine learning methods provides the possibility for a learning agent to create its own input data by applying actions to its environment and observing the consequences. So instead of requiring data that covers all of the input space the learning agent can explore this space on its own and gradually improve its behavior by considering the consequences of its past actions.

Unfortunately, the problem of high computation times in the case of high-dimensional input data still persists for reinforcement learning and although reinforcement learning allows for creating new training data, high-dimensional state and action spaces of the environment lead to a high amount of experiments that have to be run in order to collect enough data for successful learning. A common approach to solving this problem is the introduction of features. A feature contains specific information about the data and a proper feature representation of an input data set consists of various features that altogether contain all the information of the data that is required for the learning task. All the features of a data point can be stored in a feature vector which - given the features are well constructed and the input data was not originally in a high-level representation already - has a much lower dimensionality than the input data itself. Subsequently, the idea is to perform the learning algorithm on these feature vectors instead of the raw input data, thus decreasing the amount of necessary experiments and, as a consequence, also the overall computation time significantly. This approach raises the question of how to choose good features for a specific task. The most straightforward answer is to let them design by an expert for the task. For face recognition, e.g., features could include the size of the nose, the distance between the eyes or the hair color. The obvious drawback of this method is that expert knowledge is not always available for all tasks and even if it is, coming up with good features can still be very challenging and expensive for complex data. Additionally, hand-crafted features are always very task-specific and in most cases there are no possibilities of generalization in order to solve other tasks similarly. For instance, the features of the face recognition task are not very useful for character recognition although both work on digital images.

Here is where feature learning is called into action. It describes the automatic transformation of the input data into a feature representation that can afterwards be used for machine learning. Instead of using prior knowledge about the task, as is the case for hand-crafted features, unsupervised feature learning methods only use the input data to learn a transformation function that reduces the dimensionality of the data while maintaining most of its contained information. While feature learning methods like the classic dimensionality reduction technique called principal component analysis (PCA) have been known for a long time in the machine learning community, a recent approach which is known as deep learning has turned out to be very successful for feature learning. In deep learning, features are learned by creating an architecture of hierarchical feature representations that become more and more abstract for each additional layer of the architecture, thus, generating a powerful feature representation of the input data in the final layer.

1.1 Related Work

Instead of using the low-dimensional representation that the deep learning method provides as features for reinforcement learning, it is also possible to perform supervised learning algorithms on this feature representation. In 2006 Hinton and

Salakhutdinov presented a combination of classification and dimensionality reduction with the help of a deep architecture consisting of Restricted Boltzman Machines [3], which turned out to be the breakthrough of deep learning. Their implementation outperformed a classification approach that uses the standard principal component analysis for feature learning for different classification tasks and aroused a lot of attention in the machine learning community, especially because prior to these experiments there did not exist any efficient methods of training deep neural networks. Shortly thereafter, Bengio et al carried out further work on this topic and their experiments showed that deep neural networks, that are trained using the method that Hinton and Salakhutdinov introduced, lead to significantly improved performances than shallow neural networks [4]. In the same study, they introduced so-called autoencoders as an alternative to Restricted Boltzman Machines for the training of deep networks.

Because of the relative novelty of deep learning, the combination with reinforcement learning is not very common yet and it is therefore not well explored. However, a few approaches have already been developed, most notably by Lange and Riedmiller. In [5] they introduced a new algorithm called **deep fitted-q iteration (DFQ)** that combines a variant of the reinforcement learning algorithm Q-learning with a deep learning method that uses stacked autoencoders to create features out of the training data. As a first application for their approach, they chose a simple path-finding task in a grid-world. Instead of learning a very complicated policy, the main aspect of their method that they wanted to **evaluate was the suitability of the learned features** for reinforcement learning. For that purpose, the input data which they used for training only consisted of the **raw pixel values of camera images** of the grid-world. Their experiments showed that the created features were indeed useful since the algorithm found a nearly perfect policy for the learning task.

More recently, with the help of Voigtländer, they applied a similar variant of their DFQ algorithm to a real world application [6]. They chose the task of slot car racing. The **only input data that** the learning agent received were the **raw pixels of camera images of the racing track**. The learning agent was able to learn a policy which lead to resulting lap times that are comparable to those that humans achieved on the same racing track.

Riedmiller was also involved in another work that combines Q-learning with deep learning. In [7] the deep learning method of choice for Mnih et al. was a convolutional neural network. The resulting deep Q-learning algorithm was tested on several Atari 2600 games like Pong or Breakout. They took the raw frames from the Atari which have the size of **210x160 pixels and preprocessed** them to obtain input images of the **size 84x84 pixels for the convolutional neural network**. Impressively, the agent was able to learn policies that yield state-of-the-art results for six of the seven games that were tested using the same algorithm with the same values of its hyperparameters for all the games.

2 Reinforcement Learning

In the field of machine learning, there are many different approaches for teaching agents how to solve a given task. One of these approaches is primarily based on the agent's **interaction with the environment**. It is known as reinforcement learning and its aim is to find the behavior of the agent that yields maximal reward for the task at hand.

2.1 Markov Decision Processes

Most reinforcement learning problems can be formulated as Markov Decision Processes (MDPs). An MDP is a tuple (S, A, T, R) consisting of

- a set of states S ,
- a set of actions A ,
- a transition model where $T(s, a, s') = p(s'|s, a)$ is the transition probability between two states $s, s' \in S$ given an action $a \in A$,
- a reward function where $r = R(s, a, s')$ is the reward that the agent receives for performing an action a in the state s that leads to the state s' .

Figure 2.1 shows the general framework for the reinforcement learning process as described in [1]. The behavior of the agent is modeled as **a sequence of actions** in discrete time steps t . In each time step t , the system is in a current state s_t . When the agent applies an action a_t to the environment, it receives the resulting state s_{t+1} according to the transition probabilities of the system as well as the reward r_{t+1} for his action.

The behavior of the agent can now be defined as a policy $\pi(a|s)$ that assigns a **certain probability distribution** over **all possible actions in each state**. If an agent acts according to a policy, it chooses its action based upon the specified probability distribution in every state. Analogously, finding the best behavior corresponds to finding the optimal policy, i.e. **the policy π that maximizes the expectation of accumulated discounted rewards r_t over all time steps $E[R|\pi]$ with**

$$R = \sum_{t=0}^{\infty} \gamma^t r_{t+1}. \quad (2.1)$$

The parameter $\gamma \in [0, 1]$ is a discount factor that specifies whether the agent should focus on short term or long term rewards. If $\gamma = 1$ the reward has the same importance for the agent in every time step, whereas for $\gamma = 0$, only the reward of the current time step matters.

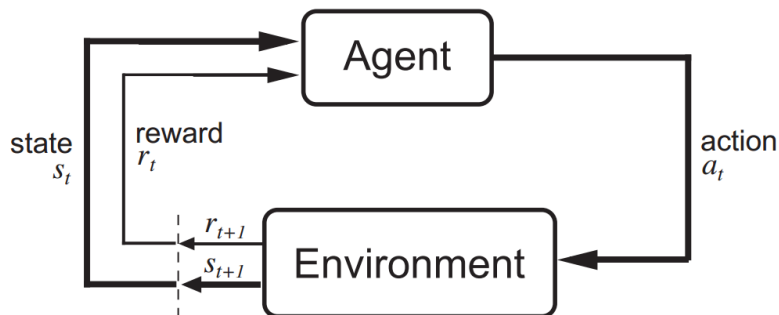


Figure 2.1: Visualisation of the reinforcement learning process using an MDP to describe the agent's interaction with its environment [1]

2.2 Q-Learning

One approach to **finding a good policy** is to first learn a **value function** for all states and then choose the policy that maximizes this value function in every state. The value function can be defined as the expectation of the accumulated discounted rewards

$$V^\pi(s) = E_\pi[R(s, a) + \gamma E_{p(s'|s, a)}[V^\pi(s')]], \quad (2.2)$$

where a is the action chosen according to the policy π , s' is the expected state that is obtained after performing action a in state s and $R(s, a)$ is the reward for this expected next state according to the reward function $R(s, a, s')$ of the MDP. V^π is called value function of the policy π . The policy will from now on be **assumed to be deterministic**, i.e. in every state it chooses the action that yields the **highest expected reward** instead of defining a probability distribution over the actions. Since the desired result of the learning process is an optimal policy π^* , the optimal action in each state has to be found using the optimal value function

$$V^*(s) = \max_a (R(s, a) + \gamma E_{p(s'|s, a)}[V^*(s')]). \quad (2.3)$$

Computing this optimal value function V^* analytically requires a perfect **model of the environment**, otherwise the next state s' can't be determined for every action in every state. As explained in the last section, the reinforcement learning approach is instead based on actually applying the actions to the environment and observing the resulting states and rewards. These collected samples are used in Q-learning, which was introduced in 1989 by Watkins in [8], to approximate the optimal value function.

Approximating the value function directly is not feasible since it would require **samples for all possible actions in a state for a single update step**. Therefore, in Q-learning the computation of the optimal value function is divided into two steps using the **optimal Q-function $Q^*(s, a)$ for all state-action pairs**. It is defined as

$$Q^*(s, a) = R(s, a) + \gamma E_{p(s'|s, a)}[V^*(s')]. \quad (2.4)$$

Using the Q-function, the value function can be written as

$$V^*(s) = \max_a Q^*(s, a). \quad (2.5)$$

Now the problem is reduced to **computing the optimal values $Q^*(s, a)$ for all state-action pairs**, which can be done iteratively by collecting samples. After initialising the values $Q_0(s, a)$, usually with zero, the Q-function is updated for every obtained sample in the form (s, a, s', r) using the following **update rule**:

$$Q_{t+1}(s, a) = Q_t(s, a) + \alpha_t [r + \gamma \max_{a'} Q_t(s', a') - Q_t(s, a)]. \quad (2.6)$$

The motivation for this rule is to calculate the **temporal difference** between the predicted Q-value $[r + \gamma \max_{a'} Q_t(s', a')]$ and its current value $Q_t(s, a)$ for every sample. Additionally, we compute the running average by summing these differences up. The introduction of the **learning rate α_t** allows for the determination of the significance of new information in relation to the existing value. The learning rate is often chosen to be constant but it can also be adjusted over time in multiple ways during the learning process.

Once either all Q-values have converged or a certain amount of iterations is reached, the algorithm terminates. The resulting **optimal policy is the one that chooses the action a in state s that maximizes $Q^*(s, a)$ for all states in the state space**:

$$\pi^*(s) = \arg \max_a Q^*(s, a). \quad (2.7)$$

The proof for the convergence of the algorithm is shown in [9].

In practice, it is necessary to introduce a strategy for the **exploration of the state space**. A popular method is the **epsilon-greedy strategy**. It introduces the parameter ϵ that defines the probability of choosing a random action in a state following a uniform distribution over all possible actions while in the $1 - \epsilon$ fraction of the time the action that maximizes the current Q-function is chosen.

2.3 Q-Learning with linear function approximation

In complex environments, the state space as well as the action space can become prohibitively large, which renders the computation of Q-values for all state-action pairs expensive. Hence, we need to **generalize over state-action pairs**. A common way to achieve this goal is to introduce a set of features that provides a high-level representation of the state and the action.

In order to provide generalization over the state space and action space for a certain task, an appropriate feature function $\phi(s, a)$ should yield similar feature vectors for similar state-action pairs. If such a function is given, the Q-function can be approximated as a linear combination of these features

$$Q_{\theta}^{\pi}(s, a) = \sum_{i=1}^M \phi_i(s, a) \theta_i = \phi(s, a)^T \theta, \quad (2.8)$$

with a weight vector θ of the same dimensionality M as the feature vector. Now instead of learning optimal Q-values for all state-action pairs directly, it is sufficient to learn the weights θ that lead to an optimal approximation of the Q-function [10]. This can be done in a very similar way as before, by iteratively computing the average over all samples (s, a, s', r) . The **resulting update rule is**

$$\theta_{t+1} = \theta_t + \alpha_t [r + \gamma \max_{a'} Q_{\theta_t}^{\pi}(s', a') - Q_{\theta_t}^{\pi}(s, a)] \phi(s, a). \quad (2.9)$$

It can be derived using **stochastic gradient descent optimization** with the **temporal difference error as loss function**. The problem about this algorithm is that it is not proven to converge for all feature functions. But if it does, $Q_{\theta_t}^{\pi}$ will be an optimal approximation of Q^* with regard to the choice of the feature function. For a feature function that provides a good representation of the states, the resulting policy will therefore be optimal. However, finding an appropriate feature function for a task is not trivial. This topic will be discussed in-depth in Chapter 3.

2.4 Least-Squares Policy Iteration

In order to solve the problem of the **possible divergence** of the Q-learning approach with linear function approximation, a more sophisticated algorithm was introduced by Lagoudakis and Parr in [11]. It is called **least-squares policy iteration** and is based on the same principle as Q-learning, i.e. approximating the Q-function as a linear combination of features $\phi(s, a)$ and weights θ and using samples to find good values for the weights. The difference is that instead of using a stochastic gradient descent method to obtain the optimal weights, **a least-squares optimization is performed**. Because of that, the optimization is usually faster for least-squares policy iteration compared to Q-learning and there is no learning rate that has to be tuned.

In Section 2.2 the optimal Q-function was shown to be

$$Q^*(s, a) = E_{\pi}[R(s, a) + \gamma \max_{a'} Q^*(s', a')], \quad (2.10)$$

as a combination of Equations 2.4 and 2.5. Thus, the exact Q-function in terms of a policy π is defined as

$$Q^{\pi}(s, a) = R(s, a) + \gamma \sum_{s'} p(s'|s, a) Q^{\pi}(s', \pi(s')), \quad (2.11)$$

where $p(s'|s, a)$ is the **transition model** of the underlying MDP and $R(s, a) = \sum_{s'} p(s'|s, a) R(s, a, s')$. This can also be written in matrix form as

$$\mathbf{Q}^{\pi} = \mathbf{R} + \gamma \mathbf{P}^{\pi} \mathbf{Q}^{\pi}, \quad (2.12)$$

where \mathbf{Q}^{π} and \mathbf{R} become vectors that contain the q-values and rewards of all state-action pairs (s, a) and \mathbf{P}^{π} is a matrix that contains the transition probabilities between all state-action pairs (s, a) and $(s', \pi(s'))$.

As in the previous section, the Q-function is approximated using a **feature function $\phi(s, a)$** and a **weight vector θ** , see Equation 2.8. If the features are written as a $|S||A| \times M$ matrix Φ , \mathbf{Q}^{π} can be substituted in Equation 2.12 by the approximation of the form

$$\mathbf{Q}^{\pi} = \Phi \theta. \quad (2.13)$$

The resulting equation for the Q-function becomes

$$\Phi \theta = \mathbf{R} + \gamma \mathbf{P}^{\pi} \Phi \theta, \quad (2.14)$$

which, rewritten, leads to

$$(\Phi - \gamma \mathbf{P}^\pi \Phi) \theta = \mathbf{R}. \quad (2.15)$$

The sought weight vector θ^π is the one that yields a Q-function $Q^\pi = \Phi \theta^\pi$ which is invariant under an update step followed by the **orthogonal projection to the Q-function space** $\Phi(\Phi^T \Phi)^{-1} \Phi^T$:

$$\Phi \theta^\pi = \Phi(\Phi^T \Phi)^{-1} \Phi^T (\mathbf{R} + \gamma \mathbf{P}^\pi \Phi \theta^\pi). \quad (2.16)$$

Solving the equation for θ^π yields the weight vector

$$\theta^\pi = (\Phi^T (\Phi - \gamma \mathbf{P}^\pi \Phi))^{-1} \Phi^T \mathbf{R}, \quad (2.17)$$

that **results in an approximated Q-function which is the least-squares fixed-point approximation of the exact Q-function**, as shown in detail in [12].

After obtaining this solution for the weights, the key of the least-squares policy iteration algorithm is to approximate both the matrix $\mathbf{A} := \Phi^T (\Phi - \gamma \mathbf{P}^\pi \Phi)$ and the vector $\mathbf{b} := \Phi^T \mathbf{R}$ by sampling. Instead of updating the weights after each new sample, the update steps are performed a lot less frequently. All L samples that are obtained between two update steps are stored in a set $\mathbf{D} = \{(s_i, a_i, s'_i, r_i) | i = 1, 2, \dots, L\}$. Averaging over the complete set of samples yields the following equations for the approximations of \mathbf{A} and \mathbf{b} :

$$\tilde{\mathbf{A}} = \frac{1}{L} \sum_{i=1}^L \phi(s_i, a_i) (\phi(s_i, a_i) - \gamma \phi(s'_i, \pi(s'_i)))^T, \quad \tilde{\mathbf{b}} = \frac{1}{L} \sum_{i=1}^L \phi(s_i, a_i) r_i. \quad (2.18)$$

After computing $\tilde{\mathbf{A}}$ and $\tilde{\mathbf{b}}$, the new weights can be obtained by

$$\theta = \tilde{\mathbf{A}}^{-1} \tilde{\mathbf{b}}, \quad (2.19)$$

according to Equation 2.17. Analogously to standard Q-learning, a change of the weights results in a new Q-function Q_θ^π and therefore also in a new policy π , which is still defined as choosing the action a in state s that maximizes $Q_\theta^\pi(s, a)$.

3 Feature Learning

The input data for reinforcement learning problems is often very high-dimensional. Therefore, it is in most cases not feasible to apply learning algorithms directly on this input data as the amount of samples that is required to fill a state space increases exponentially with its amount of dimensions which leads to excessive computation times for algorithms that work on unprocessed data.

There are different ways to deal with this problem. The approach covered in this thesis is called feature learning. Feature learning consists of the transformation of high-dimensional input data $\mathbf{x} \in X^D$ into a lower-dimensional space $\mathbf{h} \in H^d$ with $d < D$ using a feature function $\phi(\mathbf{x})$. This feature function has to be learned in a way that the transformation is not accompanied by a huge loss of information since the features should yield an expressive representation of the input data in order to achieve good results for the reinforcement learning methods. It can be learned using different automatic dimensionality reduction techniques that are also utilized in other fields of machine learning. These features do not require prior knowledge about the learning task at hand which is an advantage over the common manual design of task-specific feature representations. Instead, they work directly on the data and can often be used in a similar way for different tasks.

3.1 Deep Learning

The term deep learning describes numerous algorithms that use deep architectures to generate low-dimensional transformations of high-dimensional data. Instead of directly computing features out of the input data, deep architectures consist of multiple computational layers with different abstraction levels. In each layer, a certain non-linear transformation function, that depends on which deep learning algorithm is chosen, computes a new representation of the data. The result is a hierarchical network in which usually the dimensionality of the data gradually decreases layer per layer, using the output of one layer as input for the next layer.

A motivation for the use of deep architectures is given by Bengio in [13]. He compares the concept of deep learning to the human brain, which also appears to use different hierarchical abstraction levels in order to get a deep understanding of new information it receives. Thus, the idea of deep learning is to learn high-level features of high-dimensional input data that become more powerful with the increase of the abstraction in each layer.

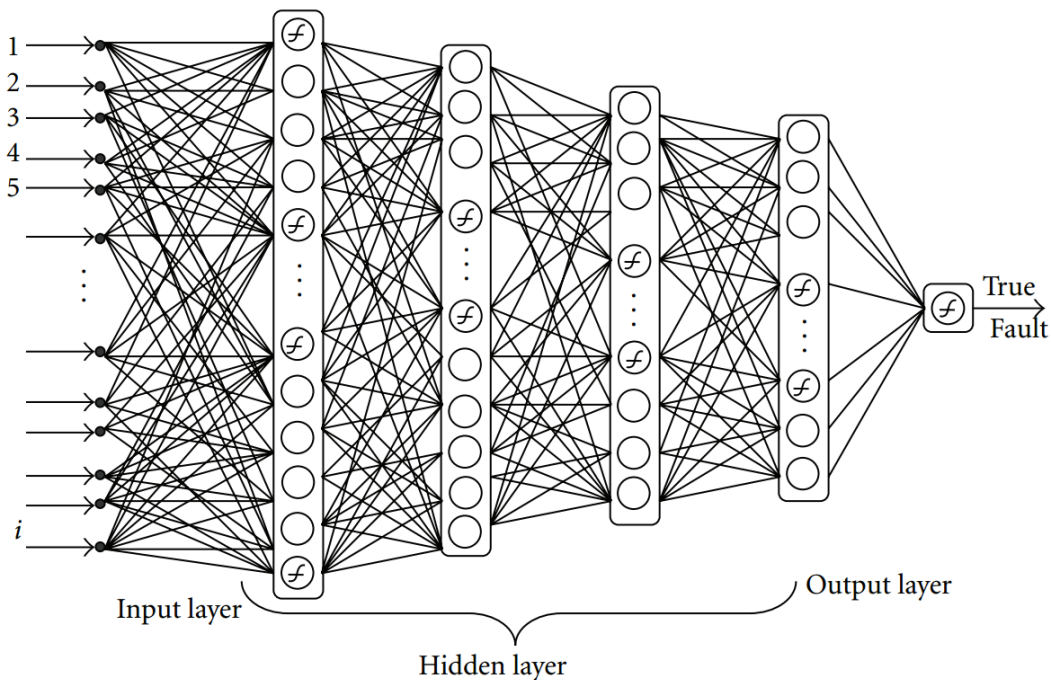


Figure 3.1: A multilayer perceptron with several hidden layers that provide more and more abstract representations of the input. The final hidden layer is used as input for a classifier [2].

A simple form of a deep architecture is a multilayer perceptron, as shown in Figure 3.1. For a long time multilayer perceptrons with a lot of hidden layers were less effective compared to shallower multilayer perceptrons with only one or two hidden layers due to the complexity of their training [4]. The cause for this is that most learning algorithms for multilayer neural networks use gradient descent methods for the optimization of its parameters. Therefore, it is possible that the learning process leads to poor local optima instead of reaching the desired global optimum if the parameters are initialised randomly.

Hinton and Salakhutdinov gave a solution to this problem [3]. They presented an approach that involves greedy unsupervised training in each of the layers of the architecture, one at a time, in order to obtain good initial parameters for the following optimization of the complete network.

3.2 Autoencoders

A common method that can be used for greedy layer-wise training of deep architectures is the use of deterministic autoencoders [14]. An autoencoder learns two transformation functions. First of all, the input \mathbf{x} of the autoencoder is encoded with the help of an encoder function of the form

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b}). \quad (3.1)$$

The result is a hidden representation \mathbf{y} of the input, which in the next step is decoded using a similar function to compute a reconstruction \mathbf{z} of the input data by

$$\mathbf{z} = f(\mathbf{W}'\mathbf{y} + \mathbf{b}_h). \quad (3.2)$$

The matrices \mathbf{W} and \mathbf{W}' as well as the vectors \mathbf{b} and \mathbf{b}_h are the parameters of the network. A common constraint in order to decrease the amount of parameters is $\mathbf{W}' = \mathbf{W}^T$. The function f is a non-linear activation function with the sigmoid being a popular choice. Hence, the prerequisite is $\mathbf{x} \in [0, 1]^D$ and according to the transformation functions this leads to $\mathbf{y} \in [0, 1]^H$ and $\mathbf{z} \in [0, 1]^D$ as well.

Instead of predicting an output as is the case for most neural networks, the autoencoder learns parameters which allow for a good reconstruction of its input. Therefore, training these parameters is an unsupervised learning task that can be solved by minimizing the average reconstruction error, i.e. the error between input \mathbf{x} and reconstructed input \mathbf{z} . Alternatively, because of $\mathbf{x}, \mathbf{z} \in [0, 1]^D$ they can be seen as independent Bernoulli distributions [15] which encourages the use of the cross-entropy loss as loss function, which is a more sophisticated approach than using the standard mean-squared error. The cross-entropy loss is defined as

$$L_H(\mathbf{x}, \mathbf{z}) = - \sum_{k=1}^D [x_k \log z_k + (1 - x_k) \log(1 - z_k)], \quad (3.3)$$

and can be minimized using a gradient descent method to find optimal values for the parameters. The required samples for the learning process don't need to be labeled since the learning is unsupervised.

3.3 Stacked Autoencoders

By using autoencoders, it is possible to construct a deep architecture by stacking multiple autoencoders sequentially [14]. The first layer of the network is an autoencoder which is used to minimize its reconstruction error. The hidden representation of this autoencoder becomes the input for the second layer which also consists of an autoencoder that computes a reconstruction of this new input, while again producing another hidden output. This procedure is repeated until the desired amount of layers is reached. The hidden values of the final layer can be treated in two different ways depending on the task that the stacked autoencoders are used for: either as an input for a supervised layer for tasks like classification or as final output of the stacked autoencoders for dimensionality reduction tasks.

The training of stacked autoencoders is usually decomposed into two steps. Initially, the parameters in each layer are trained individually by minimizing the reconstruction error as described in the last section. This procedure is called pretraining and it is done sequentially for all layers starting with the first one. As motivated in Section 3.1, the purpose of pretraining is to obtain good initial parameters for the subsequent finetuning. Finetuning is still necessary in order to evaluate the output of the whole network as opposed to the separate transformations in each layer that are regarded during pretraining. It can be performed by backpropagating the resulting final error iteratively through the complete network. While pretraining is always unsupervised, finetuning can be supervised if the final feature representation is used for a supervised learning task. Otherwise, it is unsupervised as well.

For classification tasks, the final error is the error between the labels of the training samples and the predicted outputs in the supervised layer on top of the last autoencoder. Whereas for unsupervised dimensionality reduction tasks the finetuning error can be defined as the cross-entropy loss between the input and its global reconstruction. The latter is obtained by first unfolding all autoencoders, i.e. encoding the input in the first autoencoder, taking the hidden output and encoding it in the second autoencoder, etc., until the final layer, and then analogously computing all the reconstructions by going back through the network in the reverse order. The reconstruction of the first autoencoder is the sought global reconstruction.

4 Experiments

In this chapter I present the setup as well as the results of the experiments for the combination of Q-learning and least-squares policy iteration with automatic feature learning using stacked autoencoders. The application that was chosen for the experiments in this thesis is the popular Pacman game.

4.1 Setup

The Pacman gameplay takes place in a two-dimensional grid-world that is filled with food in the form of so-called pac-dots. The player controls Pacman who has to eat all the pac-dots in order to win the game and advance to the next level. While pursuing this goal he has to be aware of the four ghosts that act as his enemies. If Pacman gets caught by one of them it loses a life and once he has lost all his lives the game is over. Additionally, eating one of the larger pac-dots that are called power pellets enables Pacman to eat ghosts in order to collect additional points for a short time period instead of dying when he collides with one of them. The Pacman implementation that was used as application for the algorithms was developed at the University of California in Berkeley and is written in Python [16]. In this implementation, the player does not have multiple lives and does not advance to a next level when it successfully completed a game, so it is a little simplified in order to improve the usability of learning algorithms. The main advantage of Pacman that makes it attractive for reinforcement learning methods is that both the state space and the action space are discrete and defining the game as an MDP is a straightforward process:

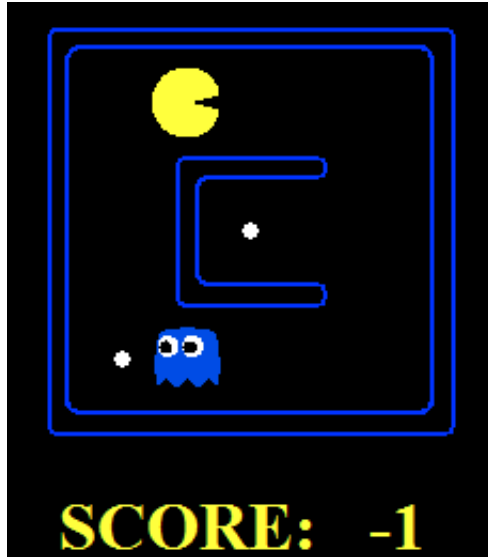
- A state $s \in S$ consists of the map layout and the current game score as well as the positions of Pacman, all ghosts and all remaining pac-dots.
- There are a maximum of five actions $a \in A$ available in each state: moving one step in one of the four cardinal directions as long as the path is not obstructed by a wall or standing still.
- The transition model $T(s, a, s')$ depends on the behavior of the ghosts. While there is no noise when applying an action, i.e. if the action a is going eastwards Pacman's position in the next state s' will always be one step further to the east than in s , the ghosts behave in a stochastic way.
- The reward function $R(s, a, s')$ is defined as the difference in game score that occurs when performing the action a in s that leads to s' . If Pacman eats a pac-dot he gains 10 points, if he wins the game by eating the final pac-dot on the map he gains 500 points and if he loses by colliding with a ghost he is punished by receiving -500 points. Additionally, in each time step the score is reduced by one to encourage fast play. In the implementation that was used eating a power pellet is not rewarded with any points but eating a ghost afterwards leads to 200 extra points.

For feature learning, similarly to other deep reinforcement learning approaches that were presented in Section 1.1, instead of using an abstract state representation the deep learning method only receives observations of the states as input data. The representation of an observation that was chosen is slightly simplified in comparison to the actual state as it does not take the current game score into account and power pellets are displayed in the same way as standard pac-dots. This leaves three possibilities for each field of an $n \times m$ map, assuming that the agents, i.e. Pacman and the ghosts, are treated separately: it can contain a wall or a pac-dot, otherwise it is empty. Every agent has a current position (x, y) with $x \in [0, n - 1]$ and $y \in [0, m - 1]$. The resulting representation of the observation is a binary vector using hot-one encoding to describe the contents of all fields of the map and the current positions of all the agents. Hence, the dimensionality of the observation of an $n \times m$ map with k total agents is $3(nm) + k(n + m)$.

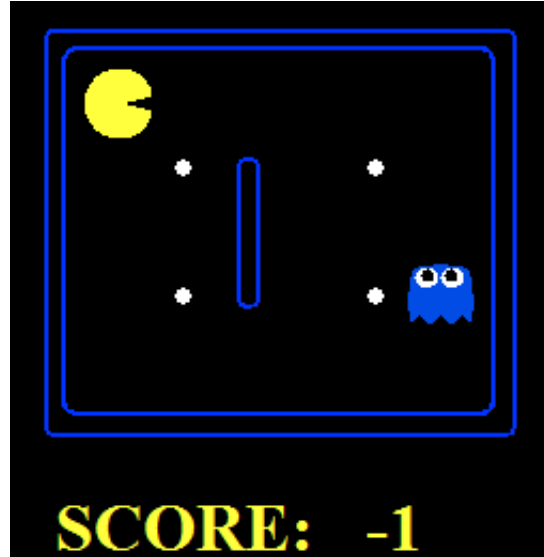
Ultimately, for the implementation of the stacked autoencoders the open source Python library Theano [17] was utilised.

4.2 Results

At first, the quality of the feature representations of observed Pacman states that were computed using the deep learning method will be presented individually. In order to determine if those feature representations do not only allow for a good reconstruction of the input but if they are also expressive enough to be useful for reinforcement learning, they were evaluated on two different Pacman maps using both Q-learning with linear function approximation and a least-squares



(a) The small grid



(b) The medium grid

Figure 4.1: The two Pacman maps on which the learning algorithms were run

policy iteration algorithm called LSTDQ. As a benchmark, standard Q-learning without the use of features as well as Q-learning with handcrafted features were applied to the same Pacman challenges.

The original Pacman map has a size of 28×31 fields and contains four ghosts. For the purpose of this thesis the experiments were carried out on significantly smaller maps with map sizes of 7×7 and 8×7 respectively. Additionally, they feature only a single ghost each and very few pac-dots. These two maps will from now on be referred to as small grid and medium grid.

4.2.1 Deep Learning

As described in Section 4.1, the input data for the stacked autoencoders is obtained by generating binary observation vectors from Pacman states. When the game is played on the small grid these observations are already 175-dimensional while for the medium grid the dimensionality increases to 198. Therefore, training the stacked autoencoders requires a lot of initial training data. The data was drawn by running simple Q-learning algorithms on both of the maps without any learned features and storing the observations from all the visited states during the course of the learning process. The final training sets consisted of 41762 observations for the small grid and 46811 for the medium grid.

The training of the stacked autoencoders was carried out exactly as described in Section 3.1. First of all, the autoencoders were pretrained individually, one layer at a time, using a gradient descent optimisation with mini-batches. The cross-entropy loss between the input of the autoencoder and the reconstruction of the input after transforming it into the hidden representation acted as the loss function. For each layer the optimisation was carried out for a total of 250 update steps with a learning rate of 0.01. Afterwards, the whole network was finetuned using the global reconstruction error as a loss function for a backpropagation optimisation which was again carried out by the gradient descent method. The finetuning ran for 600 update steps with a learning rate of 0.1.

The evaluation of the learned parameters of the stacked autoencoders was carried out on the training set. Usually, this is not a good idea since the possibility of overfitting to the training data can not be excluded if the training set is used for testing, but in this case with the small Pacman maps the training data represents the complete state space very well and it would be very hard to create any new validation data which is not yet covered by the training set. In addition to the final cross-entropy loss between the input and the global reconstruction, the overall root-mean-square error as well as the maximum reconstruction error for a single bit of the observations were calculated.

In the beginning, multiple experiments were run on the observations of the small grid that all produced feature representations of the dimensionality $M = 20$, with different layer sizes and amounts of layers. Table 4.1 shows the resulting reconstruction errors. The notation 80-40-20 e.g. means that the stacked autoencoders consist of three layers, where the first one receives the 175-dimensional input and computes an 80-dimensional hidden representation which is transformed into a 40-dimensional representation in the second layer until finally the third layer returns a 20-dimensional feature representation. All layer combinations except for the 80-40-20 variant yielded maximal errors that are close to one. This means that for each of them there existed at least one case where one bit of the observation was reconstructed completely wrong, i.e. it was either zero in the input and one in the reconstruction or vice versa. While a maximal

error of 0.5651 for the 80-40-20 layers is still far from optimal, it is significantly better than the remaining setup and the superiority of this combination of layers is also noticeable when the cross-entropy loss and the root-mean-square error are regarded. It performed even better than the 400-150-60-20 variant which was a little unexpected because other researchers obtained good results when choosing an output of a higher dimensionality than the one of its input in the first layer (see [3] e.g.). Using only a single autoencoder to compute the features resulted in very high errors. **This is a sign of the power of deep architectures.**

structure of the layers	cross-entropy loss	root-mean-square error	maximal error
20	2.2734168	0.0602623	0.9999940
80-40-20	0.0044549	0.0009910	0.5651185
140-110-80-50-20	0.0155585	0.0046332	0.9718128
160-140-120-100-80-40-20	1.2829919	0.0387994	0.9999997
400-150-60-20	0.0436514	0.0092135	0.9761342

Table 4.1: Reconstruction errors for different 20-dimensional feature representations

Subsequently, more experiments on the small grid were carried out for stacked autoencoders that were all constructed out of three layers, but with different dimensionalities in the layers. The calculated errors for these experiments are displayed in Table 4.2. **Higher dimensionalities led to lower reconstruction errors which is the expected outcome for dimensionality reduction methods.** For the layer combinations 120-80-50 and 150-125-100 the maximal errors were miniscule which basically means that the learned parameters of the stacked autoencoders nearly led to a perfect reconstruction of the input.

structure of the layers	cross-entropy loss	root-mean-square error	maximal error
70-28-12	4.0262617	0.0725132	0.9999997
80-40-20	0.0044549	0.0009910	0.5651185
120-80-50	0.0003492	0.0000137	0.0059330
150-125-100	0.0002406	0.0000080	0.0034054

Table 4.2: Reconstruction errors for three layers of different dimensionalities

After obtaining all the results an attempt of improving the performance by adding prior knowledge into the algorithm was made. The idea was to not use the sigmoid anymore for computing the reconstructions in the autoencoders. Instead, the global reconstruction was **transformed in the end before being compared to the input.** This transformation function was designed to compute the softmax separately over every three bits of the observation vector that represent a single field of the Pacman map, as well as computing the separate softmax over every group of bits that represented a coordinate of the position of one of the agents. Not using the sigmoid anymore for the reconstruction meant that the reconstructions between the hidden layers were not guaranteed to always be in the interval $[0, 1]$ any longer, **therefore the cross-entropy loss was replaced by the mean squared error.** Unfortunately, this approach failed in improving the performance. For the same 80-40-20 layer combination that was used before, the resulting root-mean-square error of the reconstruction turned out to be a lot higher than before, 0.06423 instead of 0.00099. In addition, the maximal error increased from 0.5651 to 0.9986.

Therefore, the stacked autoencoders were ultimately provided with the collected observations from the medium grid and trained in the standard way without any prior knowledge. For similar dimensionalities the reconstruction errors on the medium grid were significantly larger than those on the small grid as can be seen in Table 4.3. However, for a combination of 160–120–80 layers the maximal error becomes negligible again and all reconstruction errors decreased even further for higher dimensionalities.

Although the implementation of stacked autoencoders made it possible to create feature representation of significantly reduced dimensionalities in comparison to the input data, this does not necessarily mean that the features are useful when used in reinforcement learning. Therefore, the next sections show the outcome of learning Pacman using Q-learning and LSTDQ with the help of these features.

structure of the layers	cross-entropy loss	root-mean-square error	maximal error
100-50-25	0.0062014	0.0023961	0.9170911
160-120-80	0.0003447	0.0000154	0.0070545
180-160-140	0.0002661	0.0000113	0.0064320
190-180-170	0.0002576	0.0000112	0.0041474
200-200-200	0.0002257	0.0000096	0.0052879

Table 4.3: Reconstruction errors for different layer combinations on the medium grid

4.2.2 Benchmark

For the purpose of appropriate evaluation of the learned feature representation for reinforcement learning algorithms a form of benchmark is necessary. Therefore, prior to experimenting with the automatically learned features the learning problems for both the small grid and the medium grid were solved with basic reinforcement learning methods. First of all, basic Q-learning without was applied, i.e. the Q-function was directly approximated for all possible state-action pairs. Afterwards, the Q-learning algorithm was expanded by linear function approximation of the Q-function using a hand-crafted feature representation $\phi(s, a)$ over states and actions. It was designed by the developers of the Pacman implementation from Berkeley [16] and focuses on the field that Pacman will arrive at after performing the action a in state s . The representation consists of four separate features:

- The number of ghosts that are one step away of Pacman’s resulting position, i.e. the number of ghosts that could reach the same position in the next state.
- The occurrence of a pac-dot in Pacman’s resulting position. This feature is 1 if there are no ghosts one step away and there is a pac-dot in this field, otherwise it is 0.
- The distance to the closest pac-dot from Pacman’s resulting position. It is computed as the number as the amount of fields that are in between Pacman’s new position and the closest pac-dot divided by the total map size.
- A bias which is always 1.

Here the advantage of having expert knowledge about a learning task becomes very clear. Automatic feature learning can only rely on the input data itself which limits the potential reduction of the dimensionality because at a certain point the information loss becomes too significant. If instead the features are designed manually it is possible to provide the most important information about a state-action pair with the help of only four features.

As seen in Section 2.2 there are several hyperparameters for Q-learning that often have to be adjusted for different variants of the Q-learning algorithm. However, in this case a proper scaling of the features allowed for the use of the same hyperparameters for both algorithms. The learning rate was set to $\alpha = 0.2$, the discount factor was chosen to be $\gamma = 0.8$ and the exploration hyperparameter of the epsilon-greedy strategy that was used to explore the state-space was $\epsilon = 0.05$.

For every experiment the Q-values or the weights were learned for a total of 30000 learning episodes, where each episode corresponds to one game. Afterwards, the resulting policy was evaluated for 100 episodes. All experiments were performed eight times and the resulting average learning curves are shown in Figure 4.2 for both learning algorithms on the two different maps. The learning curves were computed by averaging over the last 100 final rewards of the completed episodes every 100 episodes. The final reward of an episode is simply defined as the final score when the game is over. Since there are only a few pac-dots in both of the maps, a successful game usually results in a final reward of about 500 points whereas a lost game analogously leads to about -500 points. Additionally to the mean of the average rewards over all experiments the 95% confidence interval is approximated by visualising the area that lies in between two standard deviations around the mean.

Because of the epsilon-greedy exploration with $\epsilon = 0.05$ Pacman chooses a random action 5% of the times. While this is necessary in order to find a good policy it can often cause suboptimal play by Pacman. Therefore, the average rewards that are depicted in the learning curves are generally worse than those that would emerge if the policies were evaluated with $\epsilon = 0$ at a certain point of time. Because of that, an exact evaluation can only be accomplished for the final policy and the resulting values in comparison to the rewards during the learning process are presented in Table 4.4. On the small grid the learning curve for standard Q-learning becomes stable after approximately 2500 episodes and the algorithm always produces optimal or near-optimal final policies. Q-learning with handcrafted features seems

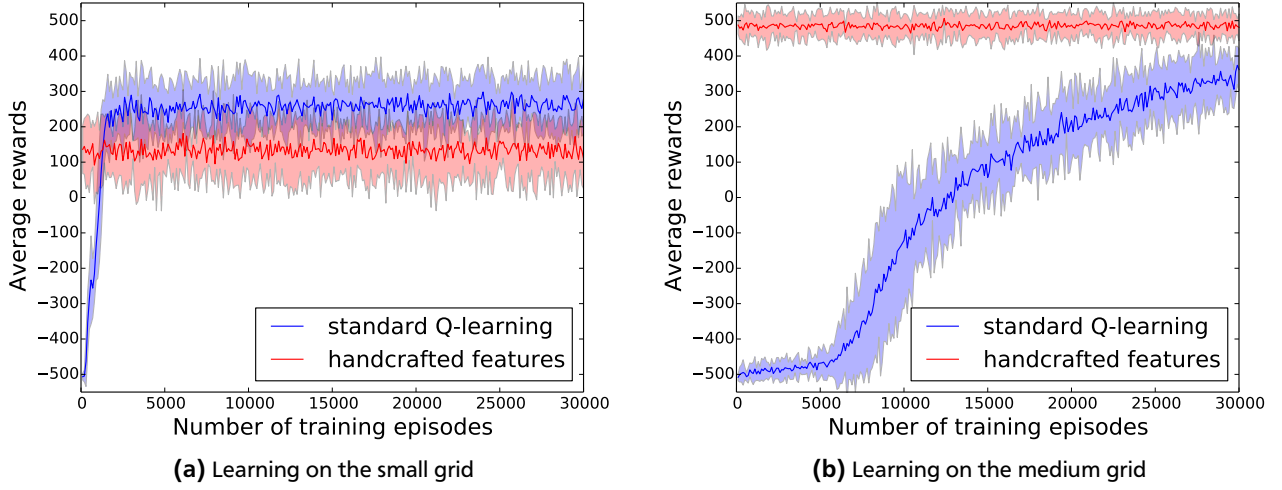


Figure 4.2: The learning curves for learning Pacman on two training maps using the benchmark algorithms

variant of the algorithm	best while learning	last while learning	final policy
small grid, standard Q-learning	298.6	279.7	499.9
small grid, handcrafted features	181.2	110.6	225.8
medium grid, standard Q-learning	373.9	361.6	483.6
medium grid, handcrafted features	506.5	481.2	527.5

Table 4.4: Averages of the best and last rewards during the learning process as well as the average rewards of the final policies for the benchmark algorithms

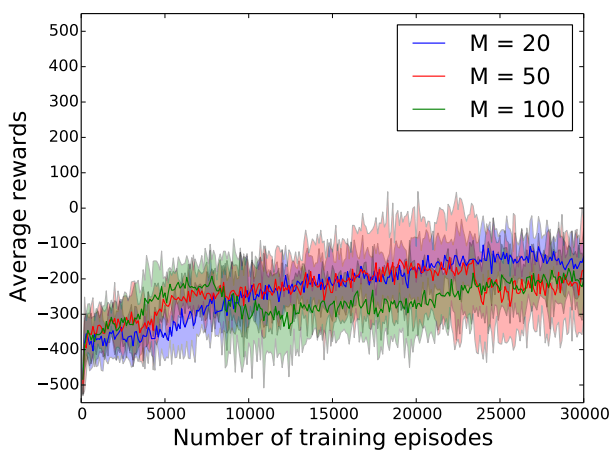
to find a decent policy already at some point during the first 100 episodes but it is **unable to further improve the policy** and the **final result is worse than the result from standard Q-learning**. However, Q-learning with handcrafted features clearly outperforms standard Q-learning on the medium grid, again finding a good policy after very few episodes which is near-optimal in this case as opposed to standard Q-learning which progresses a lot slower than on the small grid.

The reason for this is that on the medium grid the amount of **possible game states** is a **lot higher than on the small grid**. Since in standard Q-learning the Q-function is learned directly for every state-action pair it takes a very large amount of update steps until the algorithm converges. While it is apparent that using the handcrafted features accelerates the convergence significantly the performance on the small grid shows a flaw of the used set of features. In order to reach the pac-dot in the middle of the map Pacman has to enter a blind lane where he can be caught by a ghost if the second pac-dot is still on the map. During standard Q-learning the Q-values for entering this trap are adjusted appropriately so that the pac-dot in the middle is only pursued after the other one was already eaten, but the handcrafted features don't provide any consideration of the outcome when Pacman enters a blind lane.

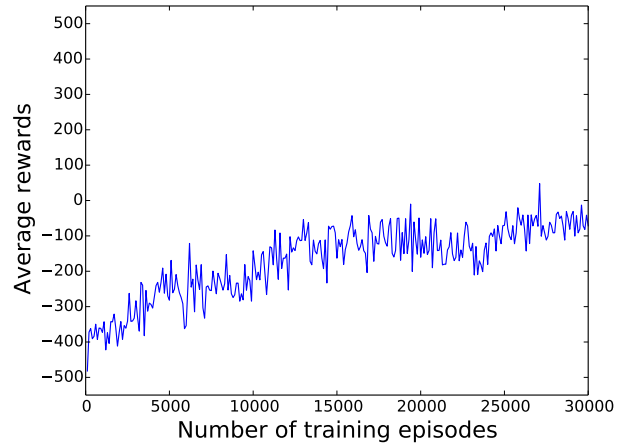
4.2.3 Q-Learning with linear function approximation combined with deep learning

After obtaining proper benchmarks the Q-learning algorithm could be performed using the learned transformation function of the stacked autoencoders as feature function. But first a slight change had to be applied to the algorithm. While in general feature functions are functions over both states and actions the resulting feature representation after training the stacked autoencoders uses only **the observations of states as inputs** without any regard to the action. Since the set of actions of the Pacman game is discrete, the solution to this problem is straightforward. Instead of a single weight vector five separate weight vectors were used, one for each action, **so in every update step only the weights for the current sampled action are updated**:

$$\theta_{t+1}^a = \theta_t^a + \alpha_t [r + \gamma \max_{a'} Q_{\theta_t}^\pi(s', a') - Q_{\theta_t}^\pi(s, a)] \phi(s, a) \quad (4.1)$$



(a) Comparison between different feature functions



(b) An experiment with an average learning curve that resulted in a very good final policy

Figure 4.3: Results on applying Q-learning using stacked autoencoders as feature function on the small grid

dimensionality of the features	best while learning	last while learning	final policy
M = 20	-104.2	-137.1	6.6
M = 50	-138.8	-174.7	-180.2
M = 100	-170.6	-204.3	-215.3

Table 4.5: Best and final rewards for the experiments that led to Figure 4.3a

The new approximation of the Q-function is accordingly defined as

$$Q_{\theta}^{\pi}(s, a) = \sum_{i=1}^M \phi_i(s) \theta_{t_i}^a = \phi(s)^T \theta_t^a \quad (4.2)$$

in every time step t , where $\phi(s)$ is the output of the final layer of the stacked autoencoders when an observation of the state s is plugged in as input of the first layer.

When the experiments were carried out, the first issue that came up was the adjustment of the learning rate. With $\alpha = 0.2$ as in the benchmark algorithms there was no learning progress at all. The learning rate had to be reduced drastically to values in the area of 10^{-5} in order to see some progress without the learning curve dropping after a while because of oversized updates of the weights. Unfortunately, an extremely low learning rate is accompanied by very slow learning progress. Therefore, even 30000 episodes were not enough to learn a policy that comes even remotely close to the results of standard Q-learning and Q-learning with handcrafted features. Figure 4.3a and Table 4.5 show the learning curves and final performances for feature functions of different dimensionalities and the hyperparameters $\alpha = 10^{-5}$, $\gamma = 0.8$ and $\epsilon = 0.05$ on the small grid. Apparently, for Q-learning with these values for the hyperparameters features of smaller dimensionalities led to overall better learning performances than features of higher dimensionalities. For $M = 100$ there is a noticeable drop in the average rewards after about 8000 episodes which probably means that the learning rate would have to be reduced even further leading once again to slower learning. The same applies to the learning curve for $M = 50$ after about 23000 training episodes.

For the feature functions with $M = 20$ and $M = 50$ a lot of experiments were carried out with different values for the three hyperparameters in the ranges of $\alpha \in [5 \cdot 10^{-6}, 10^{-4}]$, $\gamma \in [0.4, 0.9]$ and $\epsilon \in [0.01, 0.05]$. None of them led to learning curves with significantly better average rewards. However, the final policy that was evaluated after the 30000 episodes turned out to be very good on rare occasions although the learning curve looked average. Figure 4.3b shows such an experiment with $M = 20$, $\alpha = 2 \cdot 10^{-5}$, $\gamma = 0.6$ and $\epsilon = 0.05$ where the learning curve does not look any better than in the average case but the final policy yielded an average reward of 412.8. The probable reason for these outliers is that the weights never converge during the 30000 episodes, thus during the learning process the algorithm might temporarily calculate approximated Q-values that led to near-optimal policies on occasion but they are replaced by worse policies after further updates. So the assumption is that the outcome of the final policy is very luck-dependent because it might turn out completely different if the experiment is run for a couple of additional episodes.

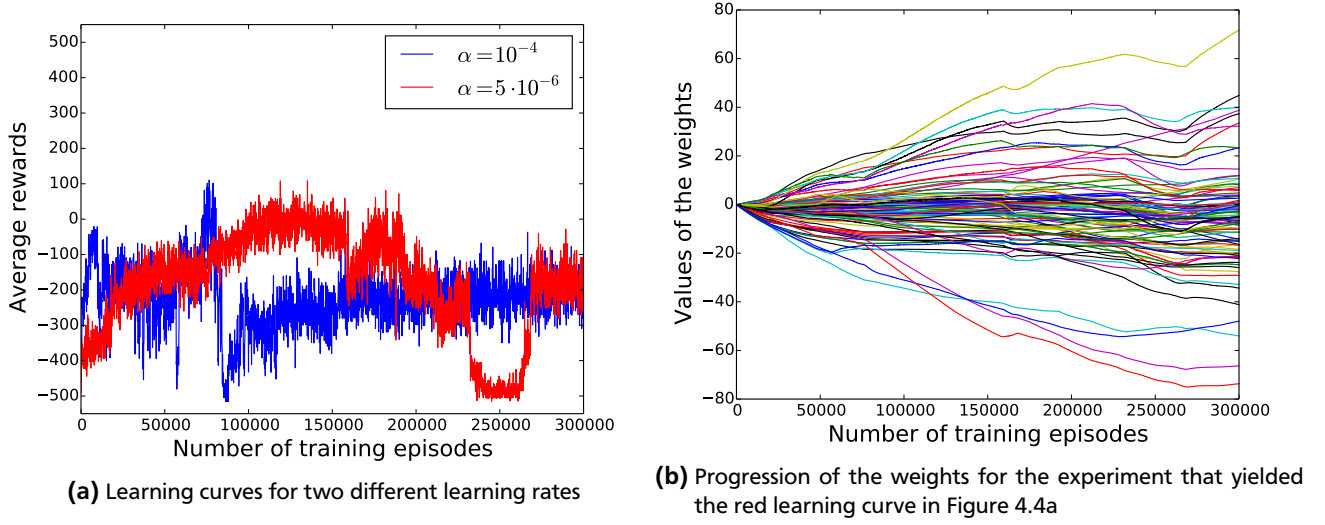


Figure 4.4: Learning curves and a weight progression for very long Q-learning experiments

After obtaining all these unsatisfactory results the experiment was run for an excessive length of up to 300000 episodes a couple of times. In Figure 4.4a the resulting learning curves of two of these experiments is shown and again after some time - the higher the learning rate the sooner - the average rewards stagnate until they suddenly drop, leading to strange behavior afterwards as well as final rewards, 33.4 for $\alpha = 10^{-4}$ and -64.93 for $\alpha = 5 \cdot 10^{-6}$. As depicted exemplarily for one case in Figure 4.4b the weights did not converge either for all of the long experiments. Reducing the learning rate even further while extending the total learning duration was not a viable option anymore since each of these experiments already took a couple of days and the success of somehow finding optimal policies with Q-learning seemed very questionable at this point. Furthermore, a few experiments on the medium grid showed that the problem with the convergence occurred there as well, therefore it seems to be independent from the specifics of the map.

4.2.4 Least-Squares Policy Iteration combined with deep learning

Instead of further tweaking the Q-learning hyperparameters finding a solution for the Pacman task was also attempted with the help of the least-squares policy iteration algorithm LSTDQ. For LSTDQ the Q-function is again approximated as a weighted linear combination of the features which are assumed to be a function over both state and action. While splitting the weight vector in five different vectors, one for each action, was easily applicable for Q-learning the weight updates for LSTDQ are more complicated as described in Section 2.4. Therefore, the feature function was adjusted slightly. Originally the feature function $\phi(s)$ was a function that receives the observation of a state and returns the output of the final autoencoder of the architecture of stacked autoencoders, an M -dimensional vector. The new feature-function $\phi'(s, a)$ for LSTDQ is a function of a state and an action that returns a vector of the dimensionality $5M$ and is defined by

$$\phi'(s, a_1) = \begin{pmatrix} \phi(s) \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}, \quad \phi'(s, a_2) = \begin{pmatrix} \mathbf{0} \\ \phi(s) \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}, \quad \phi'(s, a_3) = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \phi(s) \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}, \quad \phi'(s, a_4) = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \phi(s) \\ \mathbf{0} \end{pmatrix}, \quad \phi'(s, a_5) = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \\ \phi(s) \end{pmatrix} \quad (4.3)$$

where $\mathbf{0}$ has the same dimensionality M as $\phi(s)$. Hence, the resulting approximation of the Q-function

$$Q_{\theta}^{\pi}(s, a) = \sum_{i=1}^{5M} \phi'_i(s, a) \theta_i \quad (4.4)$$

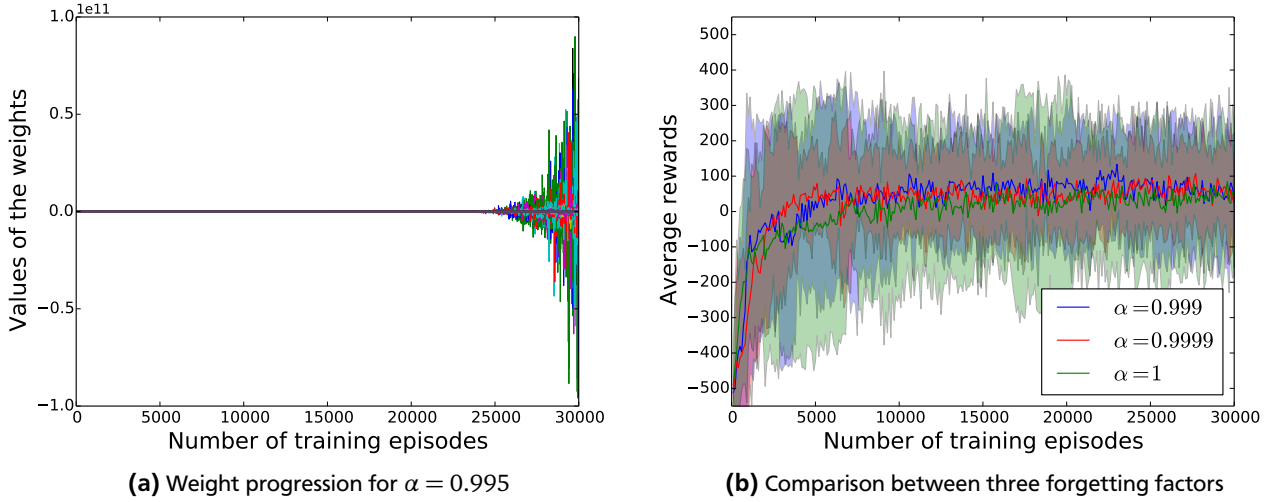


Figure 4.5: Consequences for the rewards and weights for different forgetting factors

value of the forgetting factor	best while learning	last while learning	final policy
$\alpha = 0.999$	133.5	31.9	433.7
$\alpha = 0.9999$	107.4	23.4	410.0
$\alpha = 1$	84.4	35.4	388.8

Table 4.6: Comparison of the final policies for the three tested forgetting factors

results in the same representation as in the Q-learning approach. The only difference is that instead of having five weight vectors for the five actions the single weight vector contains the weights for all the actions and by computing the cross-product of $\phi'(s, a)$ and θ only the weight values that correspond to the current action are included in the calculation. This is required because it allows for the approximation of a single \mathbf{A} matrix and \mathbf{b} vector for the LSTDQ algorithm.

Regarding the hyperparameters the biggest advantage of LSTDQ over Q-learning is that there is no learning rate that has to be tuned, which can be a very tedious task as the Q-learning experiments have shown. But on the other hand some new hyperparameters arise: is the frequency of update steps and the initialisation of \mathbf{A} . After a little experimenting an update step was performed after every 10 training episodes, i.e. usually every 50 – 150 received samples depending on the map and the current policy. This amount of samples is high enough so that an update led to significant improvements of the policy and low enough so that the learning process is not unnecessarily prolonged. \mathbf{A} as well as \mathbf{b} should ideally be initialised as $\mathbf{0}$ since they are both computed by summing over all samples. The problem is that the update equation of the weights $\theta = \tilde{\mathbf{A}}^{-1}\tilde{\mathbf{b}}$ involves the inversion of \mathbf{A} . If there have not been seen enough samples yet \mathbf{A} might very well be singular early on. In order to avoid this numerical problem \mathbf{A} is initialised as $\lambda\mathbf{I}$ with a $\lambda \in (0, 1)$. While $\lambda = 10^{-6}$ led to very spiky weight progressions $\lambda = 10^{-2}$ resulted in decent outcomes.

After determining these two hyperparameters another issue when using LSTDQ had to be tackled. Ideally, \mathbf{A} and \mathbf{b} are computed out of the collected samples once for every weight update and reset to their initial values before the next update step. However, this would require large amounts of samples between two update steps in order to yield a good performance. In praxis, there are two alternatives. First of all, the \mathbf{A} and \mathbf{b} values after the previous update steps can simply be used as new initial values for a new update step which means that in the end \mathbf{A} and \mathbf{b} are the average over the samples of all the policies that were seen during the whole learning process. The drawback of this approach is that the approximations of \mathbf{A} and \mathbf{b} are not very good approximations in the beginning of the learning, so if these values are always considered for the whole learning process the final policy might be suboptimal since it is the result of an approximation of the Q-function that is computed using flawed weights.

The second alternative features the multiplication of \mathbf{A} and \mathbf{b} with a so-called forgetting factor $\alpha \in (0, 1)$ after every update step. Thereby, for each new update all previous approximated values become a little less significant for the new values in order to stem the problem of averaging over poor approximations. Since an update is performed every 10 training episodes this factor should be very close to one, otherwise \mathbf{A} can become close to being singular after a while leading to unstable weights. Figure 4.5a shows the weight progressions for $\alpha = 0.995$ and a feature representation with $M = 50$ that becomes very spiky after about 25,000 episodes, similar to the case of initialising \mathbf{A} as $\lambda\mathbf{I}$ with a very small

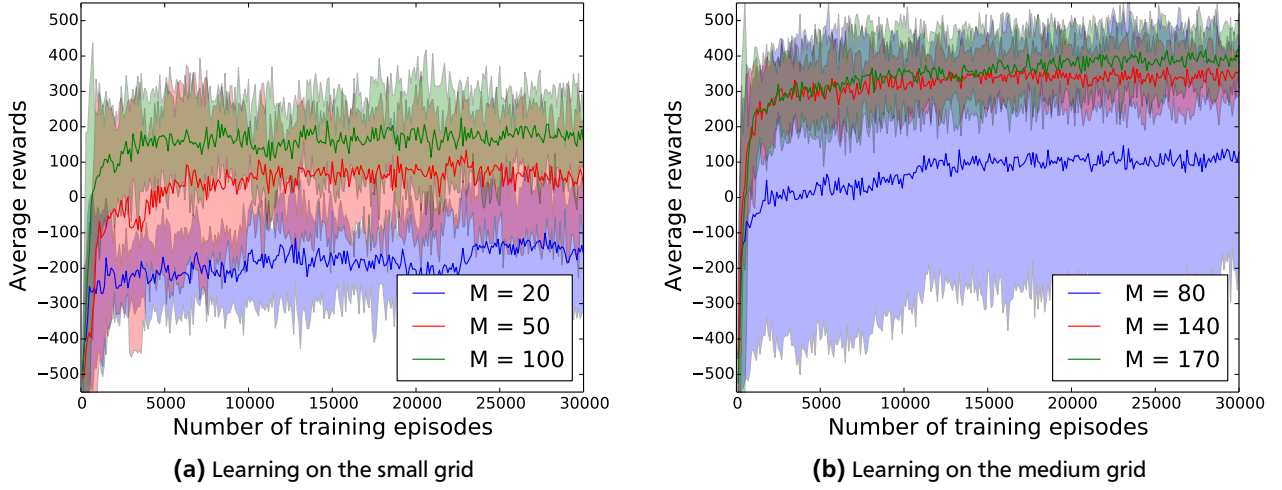


Figure 4.6: Learning curves for LSTDQ on both maps for different feature functions

variant of the algorithm	best while learning	last while learning	final policy
small grid, $M = 20$	-100.3	-161.3	-11.9
small grid, $M = 50$	133.5	31.9	433.7
small grid, $M = 100$	224.1	153.6	490.9
medium grid, $M = 80$	147.2	128.3	211.0
medium grid, $M = 140$	369.1	359.5	432.2
medium grid, $M = 170$	418.0	415.3	473.5

Table 4.7: Comparison of best and final rewards for all the different experiments

λ . In the opposite case, if α is too high, its effect becomes negligible. In Figure 4.5b the learning curves that emerge after applying the algorithm on the small grid with $\alpha = 0.999$, $\alpha = 0.9999$ and $\alpha = 1$, which is equivalent to not having any forgetting factor at all, are compared. They are all very similar to each other but the analysis of the final policies, which is summarised in Table 4.6, showed that choosing a forgetting factor of $\alpha = 0.999$ is the best choice out of all the values that were tested. The final two hyperparameters that have to be addressed remained at $\gamma = 0.8$ and $\epsilon = 0.05$ for all LSTDQ experiments which allows for better comparison to the other methods.

Ultimately, after all the hyperparameters had been tuned, the experiments could be carried out using different learned feature representations. Figure 4.6 as well as Table 4.7 show the learning curves and final results of those experiments. It is evident that applying LSTDQ was indeed a major upgrade in comparison to Q-learning since the performance was much better. Interestingly, LSTDQ works better on higher-dimensional features as opposed to what was observed for Q-learning. Using a 100-dimensional feature function for the small grid and a 170-dimensional feature function for the medium grid yielded near-optimal final policies with average rewards of 490.9 and 473.5 respectively, which is equivalent to winning the game almost all the time. This performance is also comparable to the one of the two benchmark algorithms. Of course the high-level handcrafted features make a convergence velocity possible that is beyond reach for any automatic feature learning methods, but the deep learning features do not share the flaws of the handcrafted features and allow for a better final policy on the small grid. On the medium grid the advantage of having a learned feature function over standard Q-learning without any features already becomes very clear as the learning process for LSTDQ was significantly more efficient in terms of the convergence speed.

5 Conclusion

In this thesis, I studied the applicability of deep learning for feature learning in conjunction with reinforcement learning. I developed a novel learning approach using stacked autoencoders as feature learning method. This approach does not require any structured input data as opposed to most other reinforcement learning methods. Instead, the method is only provided with raw binary observations of states of the Pacman game.

First of all, it was shown that the dimensionality of the input data could be reduced greatly using the stacked autoencoders while retaining a small reconstruction error. Additionally, the experimental results validate the hypothesis that, provided the layers are pretrained individually in the beginning, networks with multiple layers outperform shallow networks with only one layer significantly. However, considering only the reconstruction error as an evaluation method for the learned feature representations is not sufficient for showing their suitability for reinforcement learning. Thus, in the second part of my experiments I used the learned features for two reinforcement learning methods.

Initially, I applied Q-learning with linear approximation of the Q-function using features that were generated by the stacked autoencoders. The results were disappointing in comparison to using a simple set of handcrafted features or even standard Q-learning without any features. The problem about Q-learning with linear function approximation is that the algorithm is not guaranteed to converge for all feature functions. This is exactly what happened with the deep learned features, although the experiments were carried out for very large amounts of training episodes. Related to the failure of converging, the tuning of the learning rate was a big issue that could not satisfactorily be resolved.

In order to overcome these limitations, I subsequently applied least-squares policy iteration. LSTDQ emerged as a fairly efficient algorithm for the task. Its performance on the small grid after tuning the hyperparameters and choosing a good feature representation was comparable to the one of standard Q-learning, both in terms of final rewards and number of required experiments until the learning curve reached a stable optimum. On the same map it outperformed the Q-learning algorithm that used handcrafted features. On the medium grid, LSTDQ already learned a lot faster than standard Q-learning with similar final rewards, but it was inferior to the handcrafted features.

The major novelty of the proposed approach lies in the use of simple vectors of bits as input data. There have been numerous learning approaches for Pacman in the past, but they usually incorporated a great deal of prior knowledge about the game in order to achieve good results. This is, to the best of my knowledge, the first time that Pacman was learned successfully from raw observations of the game. Looking forward, this is another hint of the applicability of deep learning in order to improve machine learning methods that directly work on pixels of real-life camera images, which is one of the current challenges in reinforcement learning.

On the other hand, the researched methods have shown some practical drawbacks. During the experiments two main issues were observed. The first one is the tuning of the hyperparameters for the reinforcement learning tasks. Especially for LSTDQ, this procedure took a long time. Although the problem of finding a good learning rate does not arise, the total amount of hyperparameters in comparison to Q-learning increases. Especially finding a good forgetting factor is not an easy task since it also depends on the frequency of update steps. In general, this would not be a big problem, but in combination with the second issue, the general computation time of the approach, it considerably slowed down the whole evaluation process. Training stacked autoencoders with multiple layers is computationally expensive since every layer has to be pretrained individually before the actual finetuning of the whole network can be performed. On top of that, both Q-learning and LSTDQ are algorithms that have to call the feature function many times in between every two actions of the agent since in every state the action that yields the maximal Q-value has to be determined and the Q-value is computed directly as the cross product between the feature function and the weights. This means that every time a previously unseen state is observed, all the layers of the stacked autoencoders have to be unfolded in order to compute the new features. This takes a lot more time than calling the simple handcrafted feature function e.g., thus leading to time-consuming experiments, especially when the available computational power is limited.

The proposed approach has another limitation. The deep learning method and the reinforcement learning algorithms ran completely separately, which is probably a reason for the slow convergence of the Q-learning algorithm. A possible extension of the proposed algorithm would be changing the way in which the observations that are necessary for training the stacked autoencoders are obtained. Currently, the observations are collected using the already existing implementations of standard Q-learning and Q-learning with handcrafted features. This is not a very realistic scenario because usually the learned policies from basic reinforcement learning algorithms generate data which does not provide a good coverage of the state space. Therefore, an improved algorithm could alternate between reinforcement learning and deep learning, improving both the policy and the feature representation over time. In the beginning, the reinforcement learning algorithm could be initialised with a random policy. After a certain amount of training episodes, all observations of the sampled states are used to train the stacked autoencoders. Now the trained stacked autoencoders can be used as a

feature-function for the reinforcement learning method which again runs for a fixed amount of training episodes, thus updating its weights and the policy. All the new observed states are once again transferred to the stacked autoencoders as new training data to provide an improved feature function. This whole process is repeated iteratively until a good policy is found.



Bibliography

- [1] R. S. Sutton, “Reinforcement Learning,” *The MIT Encyclopedia of the Cognitive Sciences*, 1999.
- [2] R. Bohlouli, B. Rostami, and J. Keighobadi, “Application of Neuro-wavelet Algorithm in Ultrasonic-phased Array Nondestructive Testing of Polyethylene Pipelines,” *J. Control Sci. Eng.*, vol. 2012, pp. 19:19–19:19, Jan. 2012.
- [3] G. E. Hinton and R. R. Salakhutdinov, “Reducing the dimensionality of data with neural networks,” *Science*, vol. 313, pp. 504–507, July 2006.
- [4] Y. Bengio, P. Lamblin, D. Popovici, and H. Larochelle, “Greedy Layer-Wise Training of Deep Networks,” in *Advances in Neural Information Processing systems (NIPS)*, pp. 1–17, 2007.
- [5] S. Lange and M. Riedmiller, “Deep auto-encoder neural networks in reinforcement learning,” in *Neural Networks (IJCNN), The 2010 International Joint Conference on*, pp. 1–8, July 2010.
- [6] S. Lange, M. Riedmiller, and A. Voigtländer, “Autonomous reinforcement learning on raw visual input data in a real world application,” in *IJCNN*, pp. 1–8, IEEE, 2012.
- [7] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, “Playing Atari with Deep Reinforcement Learning,” *CoRR*, vol. abs/1312.5602, 2013.
- [8] C. J. C. H. Watkins, *Learning from Delayed Rewards*. PhD thesis, King’s College, Cambridge, UK, May 1989.
- [9] C. J. C. H. Watkins and P. Dayan, “Q-learning,” *Machine Learning*, vol. 8, no. 3-4, pp. 279–292, 1992.
- [10] F. S. Melo and M. I. Ribeiro, “Convergence of Q-learning with linear function approximation,” in *Proceedings of the 2007 European Control Conference*, pp. 2671–2678, 2007.
- [11] M. G. Lagoudakis and R. Parr, “Model-Free Least-Squares Policy Iteration,” in *NIPS* (T. G. Dietterich, S. Becker, and Z. Ghahramani, eds.), pp. 1547–1554, MIT Press, 2001.
- [12] M. G. Lagoudakis and R. Parr, “Least-squares Policy Iteration,” *The Journal of Machine Learning Research (JMLR)*, vol. 4, pp. 1107–1149, Dec. 2003.
- [13] Y. Bengio, “Learning Deep Architectures for AI,” *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.
- [14] P. Vincent, H. Larochelle, Y. Bengio, and P.-A. Manzagol, “Extracting and Composing Robust Features with Denoising Autoencoders,” in *ICML*, pp. 1096–1103, ACM, 2008.
- [15] P. Vincent, H. Larochelle, I. Lajoie, Y. Bengio, and P.-A. Manzagol, “Stacked Denoising Autoencoders: Learning Useful Representations in a Deep Network with a Local Denoising Criterion,” *The Journal of Machine Learning Research (JMLR)*, vol. 11, pp. 3371–3408, Dec. 2010.
- [16] J. DeNero and D. Klein, “The Pac-Man Projects.” <http://inst.eecs.berkeley.edu/~cs188/pacman/pacman.html>.
- [17] J. Bergstra, O. Breuleux, F. Bastien, P. Lamblin, R. Pascanu, G. Desjardins, J. Turian, D. Warde-Farley, and Y. Bengio, “Theano: a CPU and GPU Math Expression Compiler,” in *Proceedings of the Python for Scientific Computing Conference (SciPy)*, June 2010. Oral Presentation.