Online Feature Learning for Reinforcement Learning

Online Feature Learning for Reinforcement Learning
Bachelor-Thesis von Melvin Laux aus Darmstadt
Oktober 2014

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Online Feature Learning for Reinforcement Learning
Online Feature Learning for Reinforcement Learning

Vorgelegte Bachelor-Thesis von Melvin Laux aus Darmstadt

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

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter 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.

Darmstadt, den 10. Oktober 2014

(Melvin Laux)

Abstract

High Dimensional Reinforcement Learning requires a useful low dimensional representation of the input data to be able to train an agent. However, learning these features requires complete knowledge about the input data, which in general is not available. Initially, to acquire the needed knowledge, many experiments have to be used to explore the input data space at the before training. Preferably, the process of learning useful features should be done on the fly, during the training of the agent. One major challenge for this online feature learning is to find a feature learning method, that is time-efficient and able to adapt to changing input data distributions. One method that meets both of these criteria are incremental autoencoders. In this thesis, we evaluate the perormance of incremental autoencoders for online feature learning for Q-Learning and present limited results of applying incremental autoencoders as feature learning method on the visual pole balancing task.

Zusammenfassung

Hochdimensionales Reinforcement Learning benötigt ein gute niedrigdimensionale Repräsentation der Eingangsdaten um einen Controler zu trainieren. Um diese Features zu lernen, wird allerdings ein gewisses Maß Information über die Daten benötigt, welches zu Beginn von Reinforcement Learning für gewöhnlich nicht vorhanden ist. Um das benötigte Wissen zu extrahieren, müssen vor Beginn des Reinforcement Learning viele Experiment durchgführt werden um den Eingangsdatenraum zu erkunden. Vorzugsweise sollten gute Features gelernt werden, während der Controler trainiert wird. Eine Hauptaufgabe für dieses Online Feature Learning ist es, eine Feature Learning Methode zu finden, die sowohl zeiteffizient ist, als auch in der Lage ist, sich verändernden Verteilungen der Eingangsdaten anzupassen. Eine Methode, die beide dieser Anforderungen erfüllt, sind inkrementelle Autoencoder. In dieser Thesis wird ein Weg vorgestellt, um inkrementelle Autoencoder für Online Feature Learning für Q-Learning zu nutzen. Zum Testen wird hierbei das oft verwendete invertierte Pendel verwendet. Trotz der eher enttäuschenden Ergebnisse, zeigt dieser Ansatz eininge positive Aspekte und könnte durch weiteres Tuning der der Integration von inkrementellen Autoencodern in die Q-Learning Umgebung weiter verbessert werden.

Acknowledgments

First, I want to thank my supervisor Roberto Calandra for his advice and support during this thesis and also for providing me with this research topic.

I am also grateful to Prof. Dr. Jan Peters for accepting my thesis and providing my access to the Intelligent Autonomous Systems lab to use their computers to conduct the experiments for this thesis.

Last, but not least, I would like to thank my family and friends for always supporting me in every possible way and keeping me motivated when necessary.

Contents

1 Introduction

Many machine learning methods require high-dimensional training data to deal with the rising complexity of modern learning tasks. However, reinforcement learning does not scale up well to high-dimensional input data. This problem is known as the curse of dimensionality [1]. Traditionally, this problem is handled by using hand-crafted low-dimensional feature vectors. Creating good feature vectors is a very challenging task for complex input data and therefore usually requires an expert to the specific task. However, an expert is time-consuming and not always available. To avoid this problem, machine learning methods can be utilized to automatically learn low-dimensional feature representations of the high-dimensional input data. This process is commonly known as Feature Learning.

One common method to reduce the dimensionality of data is Deep Learning, i.e. autoencoders [2, 3, 4]. However, many existing methods for feature learning require a lot of time to produce usable results. This usually makes it infeasible to gradually improve the features during training reinforcement learning as the input data space is explored, since it does require feature learning to be done multiple times. In this thesis, we propose a method to speed up the feature learning process in order to learn features online during reinforcement learning.

1.1 Motivation

In general, at the beginning of reinforcement learning, there usually is very little to no knowledge about the input data available. This requires the agent to perform many random experiments to collect data in the beginning, since learning features on a very little subset of the input space may lead to bad low dimensional representations and therefore poor performance of the reinforcement learning. In an ideal high dimensional reinforcement learning framework, features are learned "on the fly", as the agent is exploring the state space. This Online Feature Learning requires to relearn new features of the input data many times, which leads to a significant increase in training time to find an optimal policy. To efficiently perform online feature learning for reinforcement learning, a feature learning method is needed, that : a) can be trained quickly, and b) is able to adapt changing input data distributions.

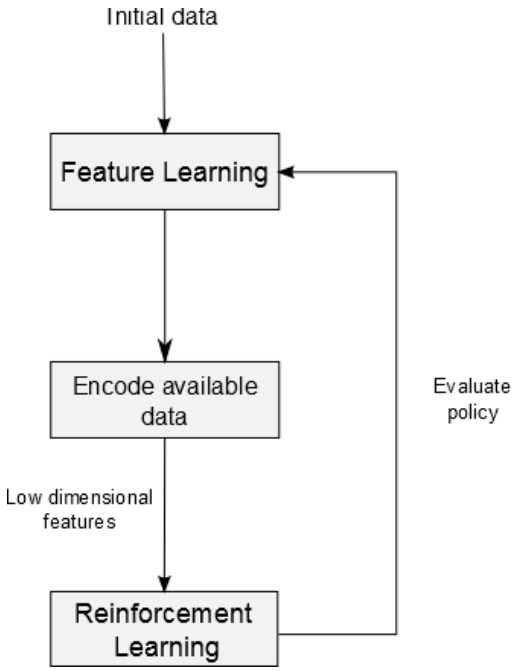


Figure 1.1: Ideal high dimensional reinforcement learning framework with online feature learning

The pole-balancing task is a popular task to explore the possibility of doing high-dimensional reinforcement learning. In 2012, Jan Mattner, Sascha Lange and and Martin Riedmiller used learned to swing up an inverted pendulum from raw visual visual data only [5]. In this approach, a stacked denoising autoencoder [6, 7] was used to learn low dimensional features from the visual input data. These features were then used to train an agent to swing up the pendulum using Fitted-Q [8] However, one of the major drawbacks of stacked autoencoders is the relatively long training time. In order to obtain useful features, the stacked denoising autoencoder was trained on 2000 previously acquired training images. To avoid having to train multiple stacked denoising autoencoders, the feature learning was removed from the loop shown in Figure 1.1 and only performed once in the beginning. However, a reinforcement learning agent may be exposed to a changing environment. Using this approach may come across difficulties, when dealing with changes in the input data distribution, while an approach, that is using online feature learning could simply adapt its feature representation to its new environment.

Therefore, in order to be able to perform online feature learning during reinforcement learning, the feature learning process needs to be sped up significantly. This way, feature learning can be put back into the loop (see Figure 1.1). One fast method to learn features in an online manner are incremental autoencoders, first introduced in 2012 by Guanyu Zhou, Kihyuk Sohn and Honglak Lee [9]. This new type of autoencoder can be trained online with a fast incremental learning algorithm. This algorithm allows to train the autoencoder on only a small fraction of the available data, profoundly speeding up the process. The algorithm is able to add new hidden neurons to the autoencoder and to merge similar ones into one. This allows the incremental autoencoder to adapt to changes in the input data distribution.

Additionally, this adds a certain robustness to the choice of the size feature vector. Choosing the amount of necessary features to represent the input data, while still being able to perform reinforcement learning in a fast way can be a very complex task. The ability of the incremental autoencoder to change the size of its hidden layer makes the this choice a less critical issue.

1.2 Contribution

The goal of this thesis is to find a method to sufficiently speed up the feature learning process to make it possible to include it into the loop of high dimensional reinforcement learning. In particular, we evaluate the possibility of using incremental autoencoders for online feature learning in reinforcement learning. For the experiments, Q-Learning [10], one the most common methods, was chosen for reinforcement learning. To test the online feature learning methods, developed during the course of this thesis, a simplified version of the popular and interesting pole-balancing task was used.

At first, we evaluate the performance of incremental autoencoders as a feature learning method compared to standard (denoising) autoencoders intensively with respect to time and reconstruction performance. The results show, that they can indeed greatly speed up the feature learning process in comparison to standard (denoising) autoencoders without having a less quality reconstruction. Additionally, the incremental autoencoder is able to adapt to changing input data distributions without having to be retrained

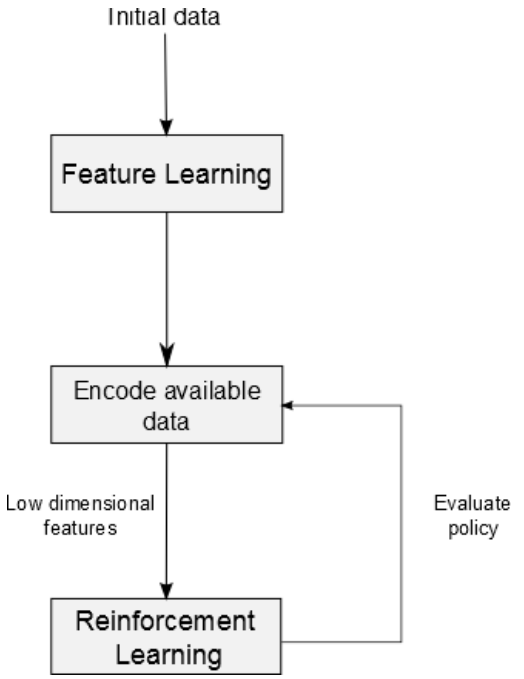


Figure 1.2: Visualization of the high dimensional reinforcement learning process in [5]

We then propose an algorithm using incremental autoencoders for online feature learning in reinforcement learning, i.e. Q-Learning and subsequently test this algorithm on the pole-balancing task. Using this algorithm allows to learn good features in a stepwise during the training of the reinforcement learning agent. The resulting high dimensional reinforcement learning framework is illustrated in Figure 3.1. The evaluation of this approach shows limited results, yet great promise for improvement.

1.3 Outline

The remainder of this thesis is divided into the following chapters: In chapter 3 of this thesis we will give a brief overview of Feature Learning using artificial neural networks. First, we will describe how to change the dimensionality of data with autoencoders and how to prevent them from overfitting the training data. In the subsequent section the incremental autoencoder will be introduced. Chapter 4 gives an introduction to Reinforcement Learning. The chapter begins with an explanation of Q-Learning, one of the most commonly used RL-methods and in the following describe how feature learning with autoencoders can be applied to it, in order to perform high dimensional reinforcement learning. Experimental results are shown and explained within the fourth chapter. Finally, chapter 5 will give a conclusion of the experiments and describe possible future work on this particular topic.

2 Feature Learning

One of the main problems of Reinforcement Learning stems from the fact, that it does not scale up well to high dimensional input data. This issue is usually handled by using low dimensional feature vectors to represent high dimensional inputs. However, finding useful feature vectors can be a difficult task, which usually requires a good a priori knowledge about the problem. The process of using machine learning methods to automatically find useful low dimensional representations of the input data is called Feature Learning. In this thesis, we decided to use deep learning (autoencoders) for learning features, however many other methods exist, e.g. principal component analysis.

2.1 Autoencoders

One method to change the dimensionality of data are so called autoencoders. Autoencoders are artificial neural networks with one hidden layer, which has the desired dimensionality of the input data. The input and output layers both have the same amount of units - one for each dimension of the given input data. The network is trained to reproduce the input values at the output layer. The autoencoder learns two functions, an encoder function and a decoder function. The encoder function transforms the input data $\mathbf{x}$ into a hidden representation

$$\mathbf{y} = f(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.1}$$

The decoder function reconstructs the hidden representation $\mathbf{x}$ back into its original representation

$$\mathbf{z} = f(\mathbf{W_h}\mathbf{y} + \mathbf{b_h}) \tag{2.2}$$

Therefore, the network has the parameters $\theta = [\mathbf{W}, \mathbf{W_h}, \mathbf{b}, \mathbf{b_h}]$, where $\mathbf{W}$ and $\mathbf{W_h}$ represent the weights of the neural network and $\mathbf{b}, \mathbf{b_h}$ its biases. Additionally, the network requires a transfer function f, e.g. the log-sigmoid function. During training, the parameters are optimized with respect to a given cost function such as the least-squares-error or cross-entropy. A common method to reduce the amount of parameters of this optimization problem is to use tied weights, which is done by adding the the following constraint: $\mathbf{W_h} = \mathbf{W}^T$.

The training of the autoencoder can be viewed as the following optimization problem:

$$\text{argmin}_\theta \; E(\mathbf{z}, \; \mathbf{x}), \tag{2.3}$$

where $E(\mathbf{z}, \; \mathbf{x})$ denotes the error between the input and the output of the autoencoder. In this thesis, we decided to use the mean squared error (MSE) as error function. This optimization problem can be solved with common methods such as Gradient Descent [11] or RPROP [12]. The necessary gradients of θ can be efficiently computed with the backpropagation algorithm. However, autoencoders often overfit to the training data.

2.2 Denoising Autoencoders

One method to prevent autoencoders from overfitting the training data and achieve a better generalization, is to corrupt the input data during training. During training, a corrupted input data $\tilde{\mathbf{x}}$ is created by adding noise noise to the original input, e.g. zeroing noise, salt-n-pepper noise or gaussian noise. The network is then trained to reconstruct the original input data $\mathbf{x}$ from the corrupted data $\tilde{\mathbf{x}}$. Just as the standard autoencoder, the denoising version is trained learns an encoder (see equation 2.1) and decoder (see equation 2.2) function. However during training modified in the following way to include the added noise:

$$\tilde{\mathbf{y}} = f(\mathbf{W}\tilde{\mathbf{x}} + \mathbf{b}) \tag{2.4}$$

$$\tilde{\mathbf{z}} = f(\mathbf{W_h}\tilde{\mathbf{y}} + \mathbf{b_h}) \tag{2.5}$$

Therefore, the training of the denoising autoencoder can be viewed as the following optimization problem:

$$\text{argmin}_\theta \; E(\tilde{\mathbf{z}}, \mathbf{x}) \tag{2.6}$$

While denoising autoencoders generalize the training data fairly well, they are not able to efficiently adapt to changes in the input data distribution. This requires a complete retraining from scratch, whenever the distribution changes, e.g. when new input data is discovered during reinforcement learning.

2.3 Incremental Autoencoders

During Reinforcement Learning the amount of available data is constantly increasing. This data can be used to find a better low dimensional representation. However, relearning new features whenever new data is collected requires training a new (denoising) autoencoder on the entire data, which usually takes up a lot of time.

An alternative has been presented by [9], the incremental autoencoder. The incremental autoencoder can be trained in an incremental manner and is able to adapt to changing input data distributions. Additionally, the incremental autoencoder can change the size of its hidden layer during training.

Since finding the optimal dimensionality of data can be rather difficult, it would be useful to automatically adapt the amount of hidden units of the autoencoder from the given input data. This can be done with incremental autoencoders. After each training iteration the reconstruction error is evaluated and the amount of hidden units is changed appropriately. If the error is very large, new features are added. If the error is decreasing, similar features are merged to decrease the complexity of the model. By doing this, over time the amount of feature converges to a reasonable amount, which is enough to return a good reconstruction of the input data while keeping the complexity as low a possible.

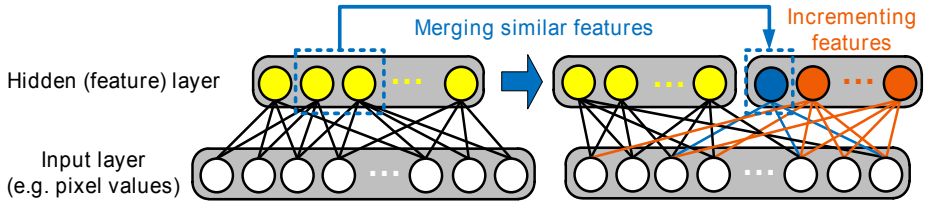


Figure 2.1: Visualization of an incremental autoencoder (taken from [9])

During training, each presented data point with a reconstruction error that is higher than the average loss of the past several data points is added to a small subset B of difficult training examples (see lines 2 and 3). Once the size of the set "hard" examples B has reached a certain value τ, the set is used to add new features, merge similar existing features of the incremental autoencoder and train the newly created features. The incremental autoencoder to use the small subset of data points B to train the new added features. This allows the incremental autoencoder to adapt to new data by optimize a small set of new parameters on a fraction of the previously seen data instead of having to be retrained completely on the entire available data.

Algorithm 1: Incremental Learning Algorithm

```
1  repeat
2      compute Error(x) of data sample x
3      if Error(x) > μ then
4          B = B ∪ x
5      if |B| > τ then
6          merge ΔM pairs of features
7          add ΔN new features
8          B = ∅
9          update ΔN and ΔM
10     finetune on complete data
11 until convergence
```

2.3.1 Adding New Features

The process of adding new features $y_{\mathcal{N}}$ begins with creating δN new features, i.e. neurons in the hidden layer, initialized with random weights. These new features are then trained on the subset of the collected hard training examples. The optimization problem is posed as the following:

$$\arg\min_{\theta_{\mathcal{N}}} \frac{1}{|B|} \sum_{x \in B} \mathcal{L}(z, x). \tag{2.7}$$

Note that the network is only optimized over the parameters of the new features, while the weights and biases of the already existing neurons remain unchanged. Now the encoder function for these new features $y_{\mathcal{N}}$ can be written as the following:

$$y_{\mathcal{N}} = f(W_{\mathcal{N}} x + b_{\mathcal{N}}) \tag{2.8}$$
$$z = f(W_{\mathcal{N}}^T y_{\mathcal{N}} + W_{\mathcal{O}}^T y_{\mathcal{O}} + c) \tag{2.9}$$

The decoder depends on both, the new features $y_{\mathcal{N}}$ and the old features $y_{\mathcal{O}}$. During optimization, the values of the old features are fixed and can therefore be precomputed and be seen as an additional dynamic decoding bias $c_d(y_{\mathcal{O}})$:

$$c_d(y_{\mathcal{O}}) = W_{\mathcal{O}}^T y_{\mathcal{O}} + c \tag{2.10}$$

This allows to compute the values of $c_d(y_{\mathcal{O}})$ only once and be reused during the training of the new features, in order to further speed up the optimization. The decoder function, can now be written as the following:

$$z = f(W_{\mathcal{N}}^T y_{\mathcal{N}} + c_d(y_{\mathcal{O}})) \tag{2.11}$$

2.3.2 Merging Similar Features

The merging process of similar features begins with creating δN pairs $\mathcal{M}$ of features which which have the least distance to each other.

$$\mathcal{M} = \arg\min_{m1,m2} d(W_{m1}, W_{m2}) \tag{2.12}$$

Each of these pairs is replaced by a new feature, whose weights are initialized with a weighted average of the two old values. These new features are then trained on the subset of the collected hard training examples. The optimization problem is then again posed as the following:

$$\arg\min_{\theta_{\mathcal{N}}} \frac{1}{|B|} \sum_{x \in B} \mathcal{L}(z, x) \tag{2.13}$$

This results in the new encoder and decoder functions for the new features, including the dynamic decoding bias:

$$y_{\mathcal{M}} = f(W_{\mathcal{N}} x + b_{\mathcal{N}}) \tag{2.14}$$
$$z = f(W_{\mathcal{N}}^T y_{\mathcal{N}} + c_d(y_{\mathcal{O}})) \tag{2.15}$$

2.3.3 Updating the parameters

After every iteration of adding/merging features the incremental learning algorithm requires to update the parameters ΔN and ΔM. This can be be done in various ways. In this thesis the parameters have been updated with the following heuristic:

$$\Delta N_{t+1} = \begin{cases} \Delta N_t, & \text{if } n \text{ is even} \\ \Delta N_t, & \text{if } n \text{ is odd} \\ \Delta N_t, & \text{otherwise} \end{cases} \tag{2.16}$$

$$\Delta M_{t+1} = \Delta N_t \tag{2.17}$$

This is one of the three methods that have been tried by [9].

3 Reinforcement Learning

A common problem in machine learning is to teach an agent to perform actions in order to achieve a given goal within its environment. One method to train such an agent is called Reinforcement Learning which has been inspired by behavioral psychology. The standard Reinforcement Learning set-up can be described as a Markov-Decision-Process, which consists of

1. A finite set of states S, representing the environment.
2. A finite set of actions A, which can be performed by the agent.
3. A reward function $r = R(s, a, s')$, determining the the immediate reward of performing a given action a in a given state s, that results in state s'.
4. A transition model $T(s, a, s') = p(s'|s, a)$, describing the probability of a transition between states s and s' when performing a given action a.

In Reinforcement Learning, the agent interacts with its environment for a finite amount of time. At every discrete time step t, the agent receives an observation o_t and chooses an action a_t from the available actions A. The environment changes its state from s_t to s_{t+1} according to the transition function T and returns a reward r_t. The behavior of the learning agent can described as a policy $\pi(a|s)$ which assigns a probability for each possible action to every state. The goal of the agent is to find an optimal policy $\pi^*(a|s)$, i.e. maximize the expected accumulated discounted reward $E[R|\pi]$ with

$$R = \sum_{t=0}^{\infty} \gamma^t \; r_{t+1}, \tag{3.1}$$

where $\gamma \in [0, 1]$ is representing the discount factor. The discount factor determines how much previous rewards are taken into account. For $\gamma = 0$, only the reward of the last time step is used, as for $\gamma = 1$ all previous rewards are used with the same importance as the current reward.

3.1 Q-Learning

One approach to find an optimal policy π^* is to use a value-function of all possible states and maximize this function in every state. This value function can be defined as the expected accumulated discounted reward

$$V^{\pi}(s) = E_{\pi}[R(s, a) + \gamma \; E_{p(s'|s,a)}[V^{\pi}(s')]]. \tag{3.2}$$

To find an optimal policy, it is necessary to always choose the optimal action in each state by making use of the optimal value function

$$V^*(s) = \max_a (R(s, a) + \gamma \; E_{p(s'|s,a)}[V^*(s')]). \tag{3.3}$$

However, computing the optimal value function V^* analytically needs a perfect model of the environment. Reinforcement Learning is collecting samples of tuples (s, a, r, s') by actually using the current policy. Q-Learning, first introduced by Watkins in 1989, uses these collected samples to build an approximation of V^*. Q-Learning divides the approximation of the value-function in two steps, by introducing an optimal Q-function $Q^*(s, a)$ for every state-action pair, which is defined as

$$Q^*(s, a) = R(s, a) + \gamma \; E_{p(s'|s,a)}[V^*(s')] \tag{3.4}$$

The problem of approximating the optimal value function is now reduced to calculating the Q-values for all state-action pairs by reformulating V^* as

$$V^*(s) = \max_a(Q^*(s,a)). \tag{3.5}$$

After initializing all Q-values (usually with zero), the optimal Q-values $Q^*(s,a)$ can be approximated iteratively with the collected observations (s,a,r,s'). For every sample, the Q-function is updated with the following rule:

$$Q_{t+1}(s,a) = Q_t(s,a) + \alpha_t \left[r + \gamma \max_{a'}(Q_t(s',a')) - Q_t(s,a)\right], \tag{3.6}$$

where the learning rate α_t determines how much the newly found information is taken into account in relation to the current value. The training process is completed, once all Q-values have converged, or a maximum number of iterations are completed. This results in an optimal policy, that always chooses the action with the maximum Q-value for a given state:

$$\pi^*(s) = max_a(Q^*(s,a)) \tag{3.7}$$

The complete process of training an agent using Q-Learning can be described with the following algorithm:

Algorithm 2: Q-Learning

```
set parameters α, γ, ϵ
initialize Matrix of Q-values Q
repeat
    observe current state s
    compute random number r ∈ (0, 1)
    if r < ϵ then
        select action a = arg max_a(Q(s, a))
    else
        select random action a
    determine next state s' using s, a
    calculate reward r = R(s, a, s')
    update Q(s, a) = Q(s, a) + α [r + γ max_a'(Q(s', a')) − Q(s, a)]
until convergence
```

The parameter ϵ determines the exploration rate of the algorithm, which allows to tune how much the state space is explored, i.e. a random action is chosen, and how much the already learned information is exploited, i.e. the action that maximizes the current Q-value for the given state is chosen.

3.2 Visual Q-Learning

As explained in the previous chapters, it is necessary to use low-dimensional feature vectors to represent high dimensional input data, e.g. in the task of swinging up and balancing an inverted pendulum in an upright position from visual input data only. One method to learn features for this task is to use (denoising) autoencoders as explained in chapter 2. However, this requires to collect enough input data samples to find a useful representation. In general, the process of teaching an agent with Q-Learning from high dimensional input data only looks like the following:

Algorithm 3: Visual Q-Learning

collect initial input data D by performing a sequence of random actions
train autoencoder on available input data D

set parameters α, γ, ϵ
initialize Matrix of Q-values Q
repeat
    observe current visual state o
    transform observation into low dimensional representation s
    compute random number $r \in (0,1)$
    if $r < \epsilon$ **then**
        select action $a = \arg\max_a(Q(s,a))$
    else
        select random action a
    determine next state s' using s, a
    calculate reward $r = R(s,a,s')$
    update $Q(s,a) = Q(s,a) + \alpha\,[r + \gamma\ \max_{a'}(Q(s',a')) - Q(s,a)]$
until *convergence*

In this algorithm, the autoencoder is trained on the initially given input data only. Afterwards the parameters $\alpha, \gamma\epsilon$ and the Q-Matrix are initialized. Then the Q-Learning loop is entered, where the current observed state is transformed into its low dimensional feature representation using the autoencoder. Using this feature vector, the Q-Learning is performed as explained above. To collect the necessary data in order to be able to find a useful feature vector, we need to do many experiments before actually beginning to train the agent.

3.3 Visual Online Q-Learning

In order not to "waste" time and experiments to collect data for training the autoencoder, it is needed to learn a new feature representation whenever new input data is available. This also leads to changing feature representations of the input data over time. To avoid having to start Q-Learning from scratch every time a new feature representation is learned, the new Q-Matrix can be reinitialized using experience replay.

Algorithm 4: Visual Online Q-Learning

set parameters $\alpha, \gamma, \epsilon, numExperiments$
initialize experience $E = \emptyset$
repeat
    train autoencoder on available input data D
    initialize Matrix of Q-values Q using experience replay of E
    repeat
        $i = 0$
        observe current visual state o
        add observation to available data $D = D \cup o$
        transform observation into low dimensional representation s
        compute random number $r \in (0,1)$
        if $r < \epsilon$ **then**
            select action $a = \arg\max_a(Q(s,a))$
        else
            select random action a
        determine next state s' using s, a
        calculate reward $r = R(s,a,s')$
        update $Q(s,a) = Q(s,a) + \alpha\,[r + \gamma\ \max_{a'}(Q(s',a')) - Q(s,a)]$
        update experience $E = E \cup (s,a,r,s')$
    until $i = numExperiments$
until *convergence*

This algorithm relearns a new feature representation after every $numIterations$ performed actions. After the parameters $\alpha, \gamma, \epsilon, numExperiments$ and the experience E are initialized, the outer loop is entered. In this loop the currently available data is used to train a new autoencoder and the Q-Matrix is initialized using experience replay. To do this, every tuple (s, a, r, s') during Q-Learning in E (see line 19 of the algorithm). During experience replay, every tuple in E is sequentially used to update Q. Additionally, during each Q-Learning step, the observation is added to the currently available dataset D (see line 9). The major drawback of this algorithm is however, that it requires a lot of time to train multiple autoencoders.

3.4 Visual Incremental Online Q-Learning

To improve the speed of online feature learning, it is possible use an incremental autoencoder instead of a standard autoencoder. Feature Learning with incremental autoencoders has the benefit, that the incremental autoencoder only has to be trained on the newly acquired data, and not on the entire available data set.

This algorithm is very similar to the Visual Online Q-Learning algorithm described in the previous section. It, however, uses an incremental autoencoder instead of a standard autoencoder. As described in the previous chapter, an incremental autoencoder can be retrained on the new observations collected by the Q-Learning loop. Note, that in line 5 of this algorithm, the input data for the feature learning is reset to the empty set. With this algorithm can significantly speed up the feature learning process by utilizing the properties of the incremental autoencoder and therefore perform online feature learning in a reasonable amount of time. The resulting high dimensional reinforcement learning framework is illustrated in Figure 3.1.

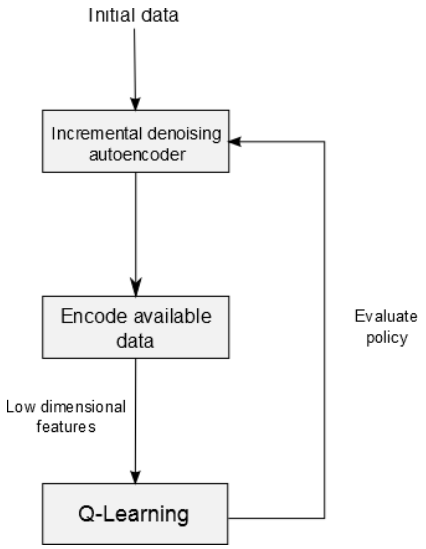


Figure 3.1: Illustration of the developed high dimensional reinforcement learning framework

Algorithm 5: Visual Incremental Online Q-Learning

```
1  set parameters α, γ, ε, numExperiments
2  initialize experience E = ∅
3  repeat
4      train incremental autoencoder on new available input data D
5      D = ∅
6      initialize Matrix of Q-values Q using experience replay of E
7      repeat
8          i = 0
9          observe current visual state o
10         add observation to available data D = D ∪ o
11         transform observation into low dimensional representation s
12         compute random number r ∈ (0, 1)
13         if r < ε then
14             select action a = arg max_a(Q(s, a))
15         else
16             select random action a
17         determine next state s' using s, a
18         calculate reward r = R(s, a, s')
19         update Q(s, a) = Q(s, a) + α [r + γ max_a'(Q(s', a')) − Q(s, a)]
20         update experience E = E ∪ (s, a, r, s')
21     until i = numExperiments
22 until convergence
```

4 Experimental Results

In this section we will describe both, the setup and the results of the experiments, that were conducted for this thesis.

4.1 Pendulum Simulator

In order to perform experiments, we implemented a simplified simulator of an inverted pendulum. The dynamics of the pendulum are defined by the following function

$$q_{t+1} = q_t + 0.2\ action \tag{4.1}$$

where q is the current angle of the pendulum and $action$ represents the control which is applied to the pendulum.

Using this simulator, a data set consisting of 1000 images has been created. These images of the pendulum served as visual input data during the experiments. Every of these images has a resolution of 32x32 pixels, leading to a 1024-dimensional input data set. Some example images of the data set can be seen in Figure 4.1.

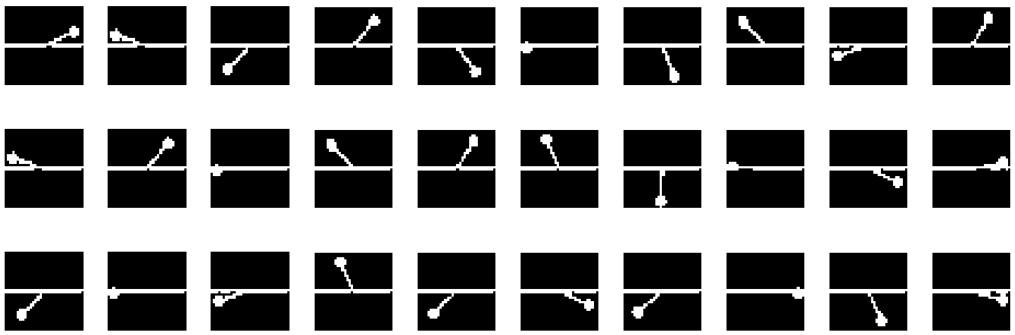


Figure 4.1: Example images of the pendulum image data set

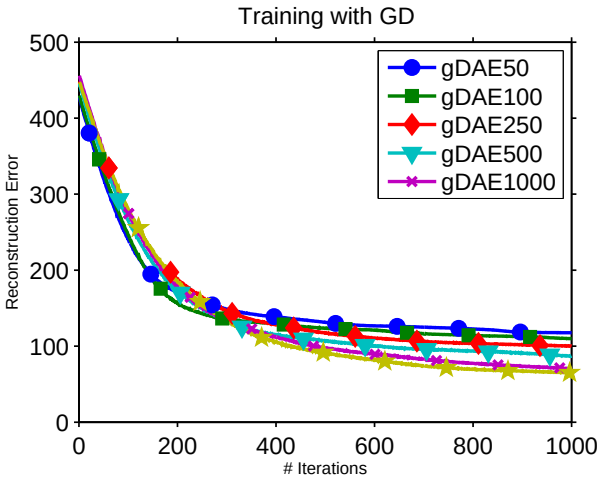
4.2 Results of Feature Learning Experiments

In the following section, the experiments on feature learning using autoencoders will be presented.

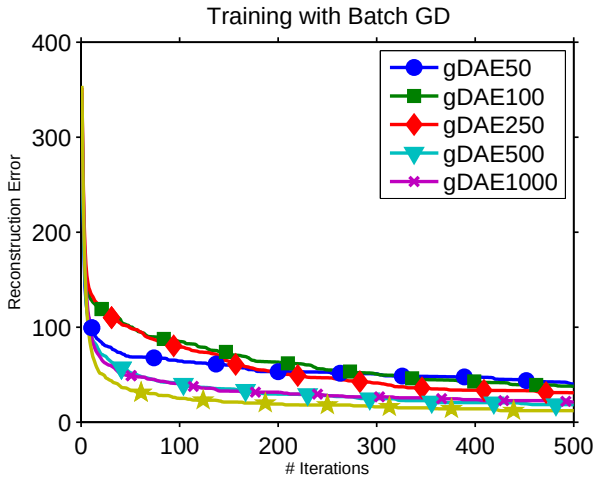
4.2.1 Autoencoders

To test the effectiveness of (denoising) autoencoders for feature learning we trained various types of (denoising) autoencoders on the pendulum data set. The effect of different types of noise, numbers of hidden units, optimization methods and transfer functions were tested.

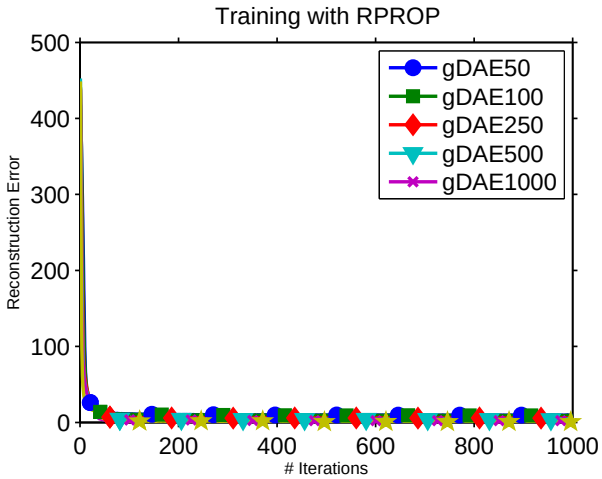
Figure **??** shows the training errors of multiple denoising autoencoders when using different optimization methods. All autoencoders were using gaussian noise with a mean of 0 and a standard deviation of 0.1. It is clearly visible, that the more hidden units are used, the the better reconstruction error gets. However, using more hidden units increases the required training time. When comparing the three used optimization methods, RPROP clearly gives the best performance. It does not only converge a lot faster than standard gradient descent or batch gradient descent, but also results in a lower reconstruction error than the other two methods.



(a) Training Reconstruction Error vs. Number of Iterations of multiple denoising autoencoders with different numbers of hidden units using the Gradient Descent optimization method



(b) Training Reconstruction Error vs. Number of Iterations of multiple denoising autoencoders with different numbers of hidden units using the Batch Graadient Descent optimization method



(c) Training Reconstruction Error vs. Number of Iterations of multiple denoising autoencoders with different numbers of hidden units using the RPROP optimization method

Figure 4.2: Comparison of different optimization methods for training autoencoders

4.2.2 Feature Learning on shuffled data

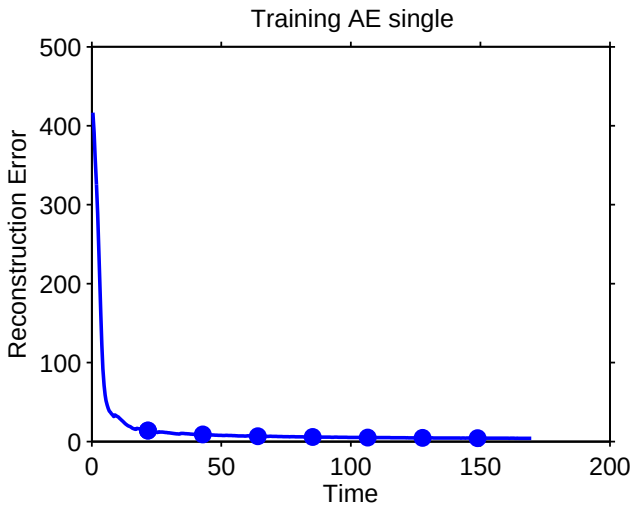
Since during Reinforcement Learning the amount of available data is increasing over time, we conducted experiments to test the speed and reconstruction errors of learning features from an iteratively growing data set using different methods of feature learning.

Figure 4.3: Example images found in one shuffled data subset

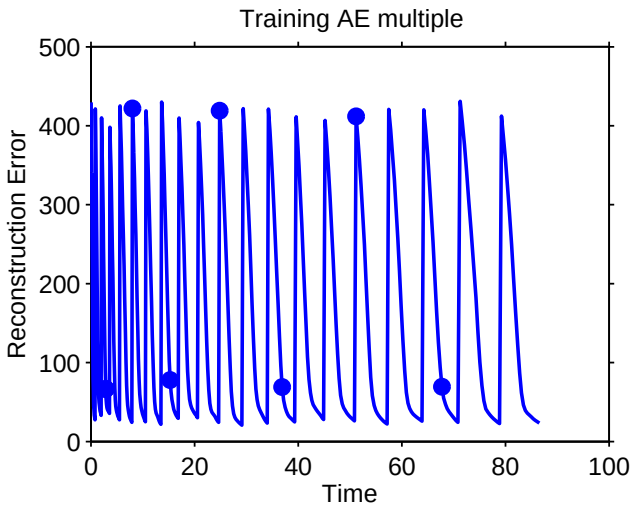
To simulate the growing amount of data, the complete data set of 1000 images was first shuffled, and then split into 20 subsets of 50 images. These subsets were used to train an an incremental denoising autoencoder over 20 iterations. The time and error of this experiment were compared to training a) a single denoising autoencoder trained on the entire available data at each iteration and b) multiple denoising autoencoders from scratch, one at each iteration, on the entire available data. Both cases were tested using the initial (50) as well as the final (76) amount of hidden units of the incremental denoising autoencoder. All of the autoencoders were trained using gaussian noise with a mean of 0 and a standard deviation of 0.1.

Figure 4.4 shows the reconstruction error during training time of all previously described feature learning setups. Training multiple autoencoders from scratch at every iteration, does lead to a very "spikey" learning curve. This makes it difficult to know when the training is actually done. The reconstruction error of the incremental autoencoder is the lowest of all five tested setups. Also, the training of the incremental autoencoder is done in less time it takes to train a single autoencoder multiple times, which in fact returns a nearly as good reconstruction error (see Figure 4.4e).

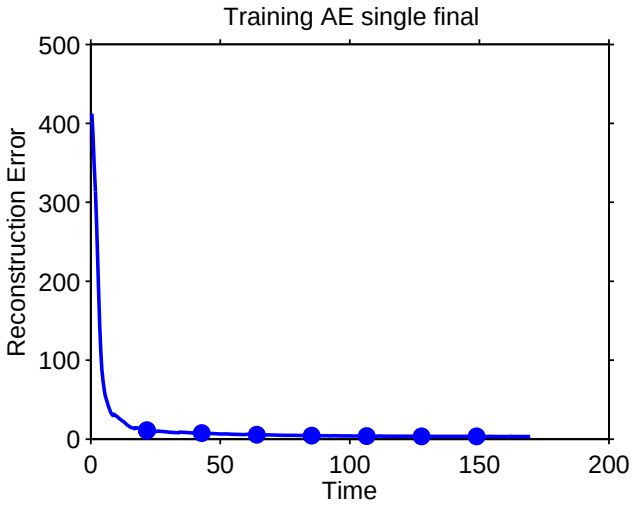
When comparing Figure 4.4a and Figure 4.4b, it becomes clear, that the retraining of the autoencoders from scratch leads to many undesirable spikes, while training the existing autoencoder with the newly acquired data is increasing in reconstruction error as the input data distribution changes. The difference in the number of hidden neurons does not impact greatly on the resulting reconstruction error (compare Figure 4.4a and Figure 4.4c or Figure 4.4b and Figure 4.4d respectively). However, when the autoencoder are retrained from scratch at every iteration, the training time is affected more by the size of the hidden layer than in the case of training a single autoencoder multiple times.



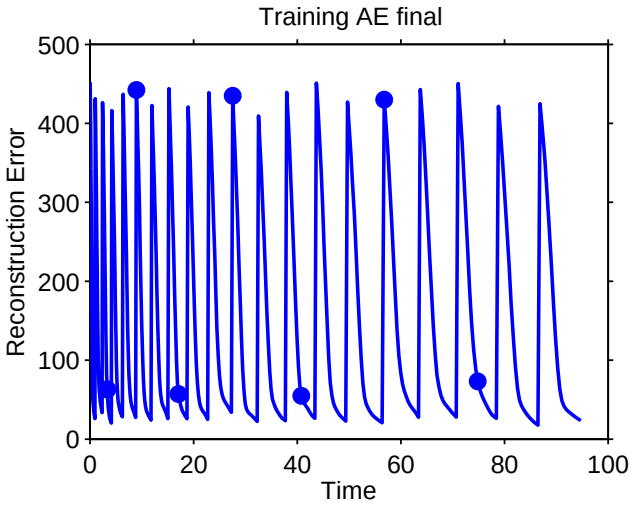
(a) Single autoencoder with 50 hidden units



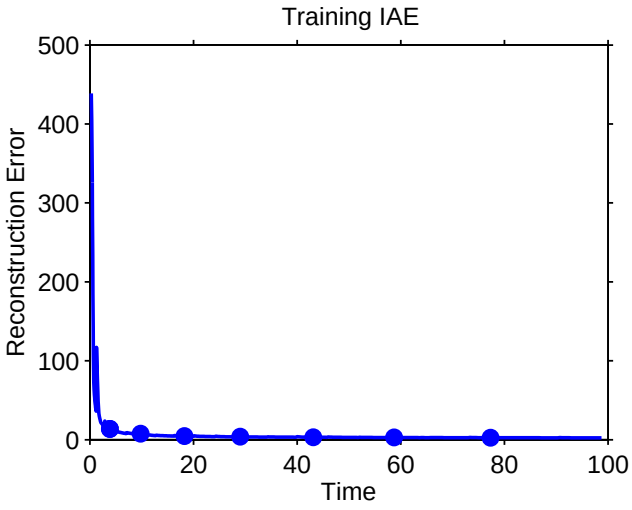
(b) Multiple autoencoders with 50 hidden units



(c) Single autoencoder with 76 hidden units



(d) Multiple autoencoders with 76 hidden units



(e) Incremental autoencoder with initially 50 hidden units

Figure 4.4: Reconstruction error of different autoencoder types during training on the shuffled pendulum dataset

4.2.3 Feature Learning on unshuffled data

In order to simulate a way of incrementing the available data to the feature learning, that is closer to the the data Reinforcement Learning methods would observe on the pendulum task, we also performed the same experiments from the previous section without shuffling the data set.

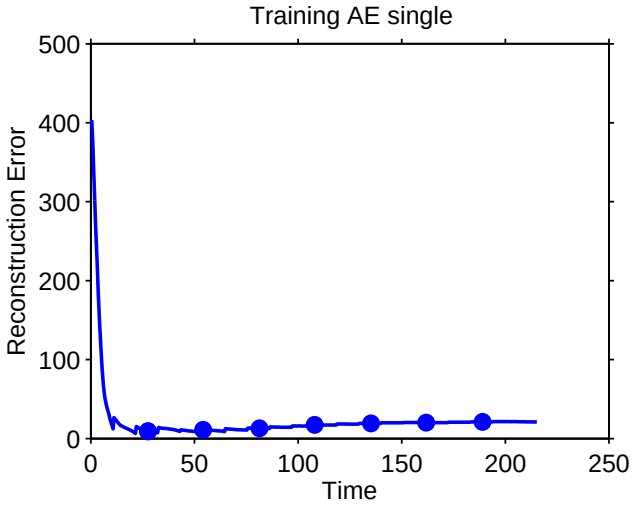
Figure 4.5: Example images found in one unshuffled data subset

By feeding the data in this unshuffled fashion to the autoencoder, it is necessary to adapt to changing distributions of the input data.

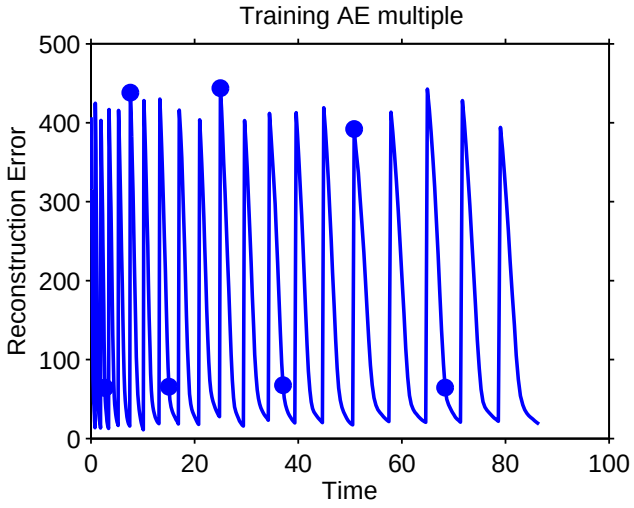
Figure 4.6 shows, that training on the unshuffled data set leads to an increase of the reconstruction error over time when training a single autoencoder. This shows that, it is not able to adapat to the changing data distribution well. Training multiple autoencoders, while dealing with the same "spike-issue" as before does get better results than the single autoencoder setting. The incremental autoencoder is not only able to adapt to the changing distribution of input data, but also can be trained in a time-efficient manner (see Figure 4.6e).

When comparing Figure 4.6a and Figure 4.6b, just as in the previous experiment, the retraining of the autoencoders from scratch leads to many spikes. However, training the same autoencoder with the newly acquired data multiple times, shows an even bigger increase in reconstruction error as the input data distribution changes. This further shows the inability of the standard autoencoder to adapt to its changing input data distribution. The difference in the number of hidden neurons has the same effects as in the previous experiment on the shuffled data set.

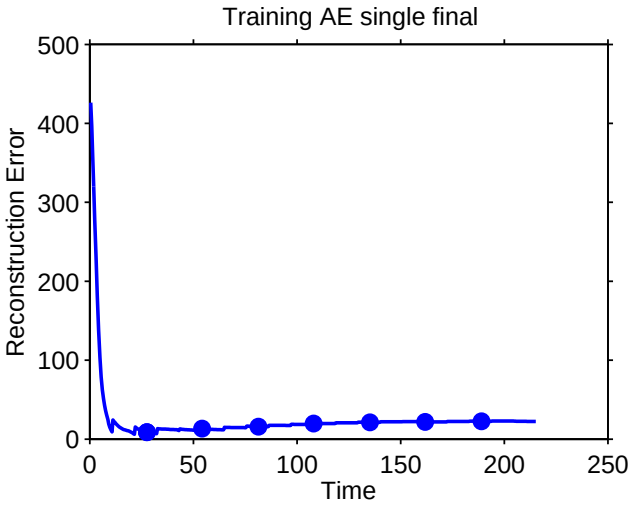
In Figure 4.7, only the reconstruction error at the end of each iteration is shown for the experiments on both, the shuffled and the unshuffled data set. When looking at Figure 4.7a, it becomes clear, that when no change in the data distribution occurs (using the shuffled data set), the single autoencoder can achieve a similar reconstruction error as its incremental counterpart. With respect to the training time, it is however significantly outperformed by the incremental autoencoder. Figure 4.7b shows, that all autoencoder variants show an increase in reconstruction error when dealing with the changing input data distribution of the unshuffled dataset. However, the incremental autoencoder outperforms all other variants significantly. Additionally, the incremental autoencoder, shows no significant increase in training time when compared to the experiments on the shuffled data set. The single autoencoder, in contrast requires a lot more time to optimize in each iteration when dealing with the changing distribution.



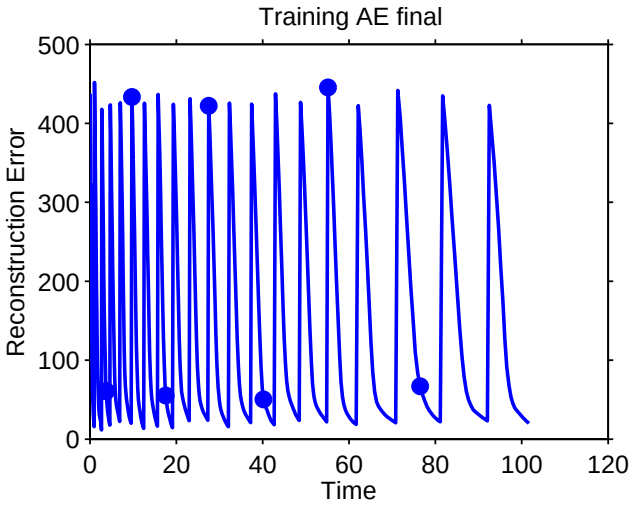
(a) Single autoencoder with 50 hidden units



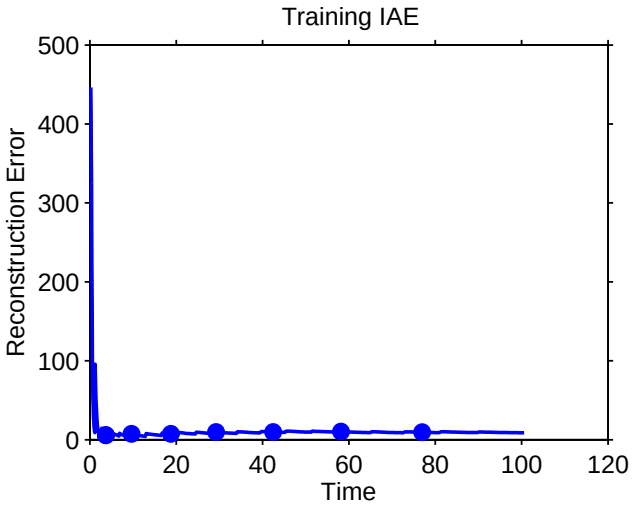
(b) Multiple autoencoders with 50 hidden units



(c) Single autoencoder with 76 hidden units

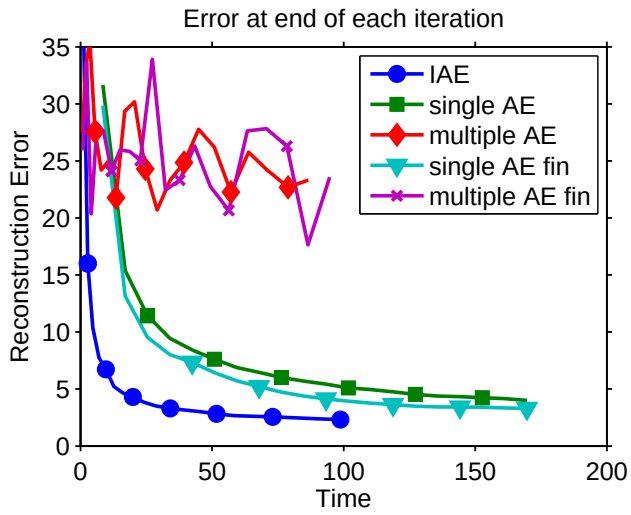


(d) Multiple autoencoders with 76 hidden units

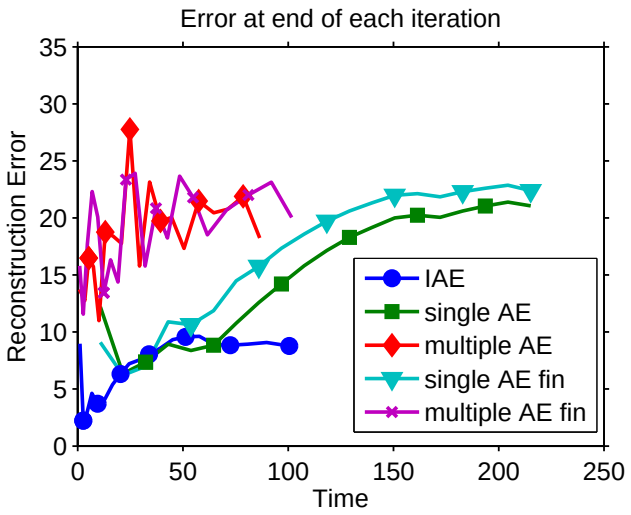


(e) Incremental autoencoder with initially 50 hidden units

Figure 4.6: Reconstruction error of different autoencoder types during training on the unshuffled pendulum data set



(a) Reconstruction Error at the end of each iteration vs. time of different types of autoencoders trained on the shuffled data set



(b) Reconstruction Error at the end of each iteration vs. time of different types of autoencoders trained on the unshuffled data set

Figure 4.7: Performance comparison of training on the shuffled and unshuffled data set

4.3 Results of Reinforcement Learning Experiments

In this section, all experiments on the various forms of Q-Learning described in chapter 3 and their experiments will be presented and explained.

4.3.1 Q-Learning

To test our implementation of the pendulum simulator, a standard version of Q-Learning was implemented. The state space was discretized into 360 states, one for every degree of the pendulums angle. The agent was given a set of five actions [big move in counterclockwise direction (-10), small move in counterclockwise direction (-3), do nothing (0), big move in clockwise direction (3), big move in clockwise direction (10)]. For all Q-Learning experiments a learning rate α of 0.2, a discount factor γ of 0.9 and an exploration rate ϵ of 0.1 were used. In the following one episode of 50 Q-Learning iterations will be referred to as one experiment. The *summed reward* of an experiment is the sum of the reward of all transitions, that were done during this experiment. Every run of training a Q-Learning agent consists of multiple experiments.

Figure 4.8 shows, that the summed reward of standard Q-Learning is converging to the optimal policy, which is reached after circa 1500 experiments with a summed reward of circa 400. The figure shows the mean and variance of the summed reward of 150 runs.

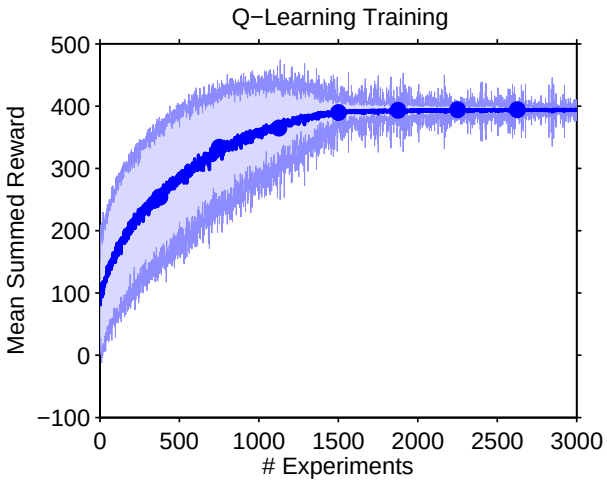
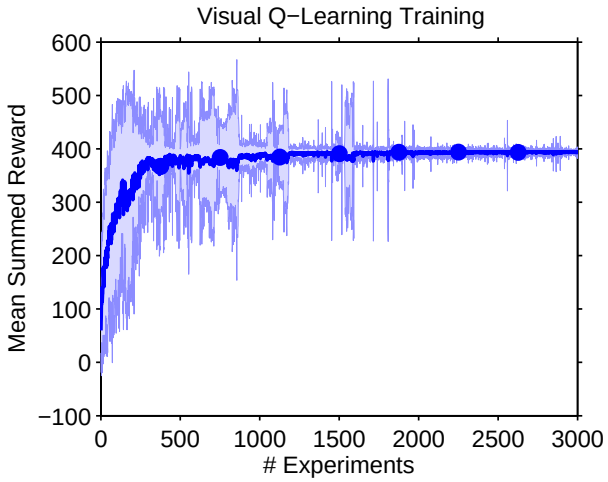


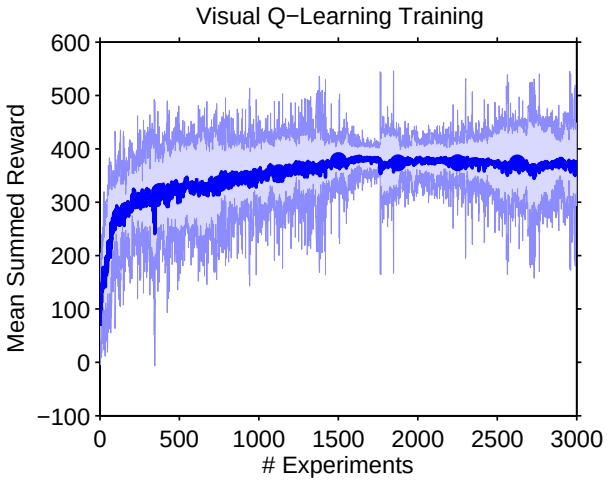
Figure 4.8: Mean reward and variance vs. number of experiments using standard Q-Learning

4.3.2 Visual Q-Learning

Next, we performed visual Q-Learning on the created pendulum data set using algorithm as seen in chapter 3. Again, all hyperparameters ($\alpha, \gamma and \epsilon$) were set to the same values as in the standard Q-Learning. For the feature learning, a denoising autoencoder with gaussian noise was used to reduce the dimensionality of the input data from 1024 to 10. It was trained on the entire pendulum data set for 500 iterations using the RPROP optimization method. Once the feature learning was done, visual Q-Learning was performed using the 10-dimensional features. To do this, the values of the feature vector were discretized into five units.



(a) Mean reward and variance vs. number of experiments using visual Q-Learning



(b) Mean reward and variance vs. number of experiments using visual Q-Learning after learning features from only partial data

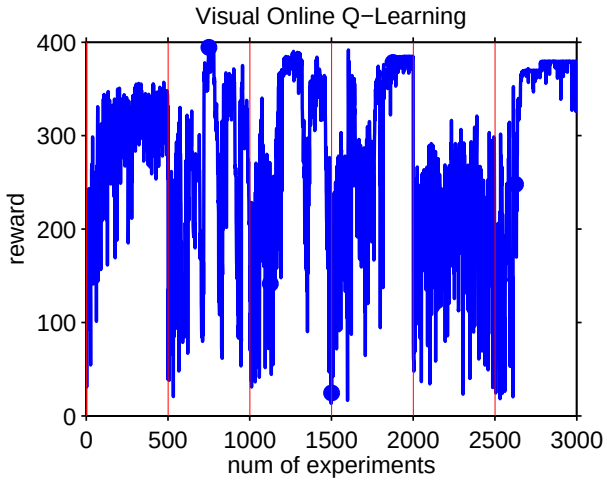
Figure 4.9a shows, that the learned 10-dimensional features can indeed be used for Q-Learning. After circa 1000 experiments, the summed reward is converged to the value that was produced by standard Q-Learning.

In Reinforcement Learning, initially there is not the entire data available to be used for feature learning. To test the usefulness of the learned features from training the autoencoder only on a partial data set, we repeated the same experiment again, while training the autoencoder on only a third of the entire pendulum data set. The images chosen for this partial data set, were the ones closest to the initial position of the pendulum.

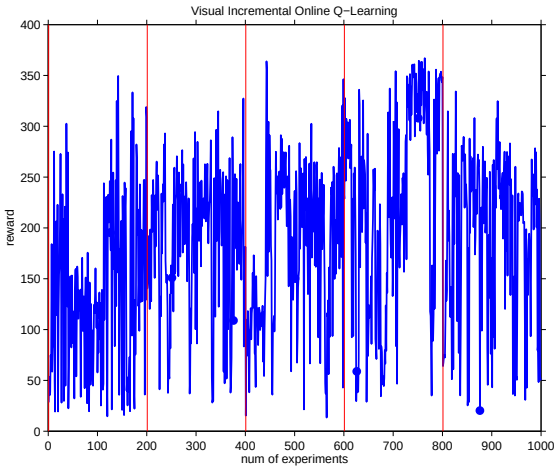
In Figure 4.9b, the summed reward of the visual Q-Learning run with the features learned from this autoencoder are shown. It is obvious, that policy learned from these features is outperformed by the one from the complete data set. The learned features are apparently not good enough to be used for learning the optimal policy.

4.3.3 Visual (Incremental) Online Q-Learning

Eventually, we performed visual online Q-learning. To do so, we used the partial input data set from the previous experiment as initially available data. We used both versions of online Q-Learning, the non-incremental one as well as the incremental one. After every 500 experiments, the features were relearned using all observations made by the agent during Q-Learning. To prevent the amount of training data from growing too large, all duplicate observations were removed from the data set before training the autoencoders.



(a) Summed reward vs. number of experiments using visual online Q-Learning. Vertical red lines indicate relearning of features.



(b) Summed reward vs. number of experiments using visual incremental online Q-Learning. Vertical red lines indicate relearning of features.

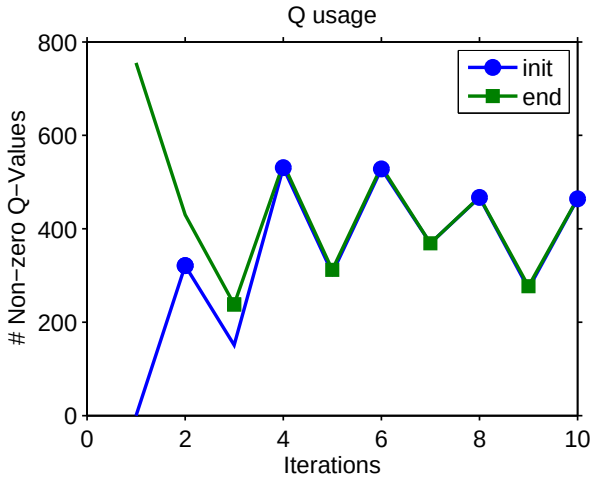
Figure 4.10: Results for online Q-Learning

Figure 4.10a shows, that Q-Learning is able to find good policies when using the learned features. However, the downward spikes of the summed reward every time new feature representations of the input data are learned, lead to the conclusion that whenever new features are introduced, the Q-Learning is basically starting from scratch each time. This most likely is a result of initially rather poor feature representations. In this case, it may happen, that multiple states are mapped to the same state. The corresponding Q-value is then built as a weighted average of all these states mapped to the same feature vector during experience replay. Q-Learning then requires further experiments to solve the ambiguities.

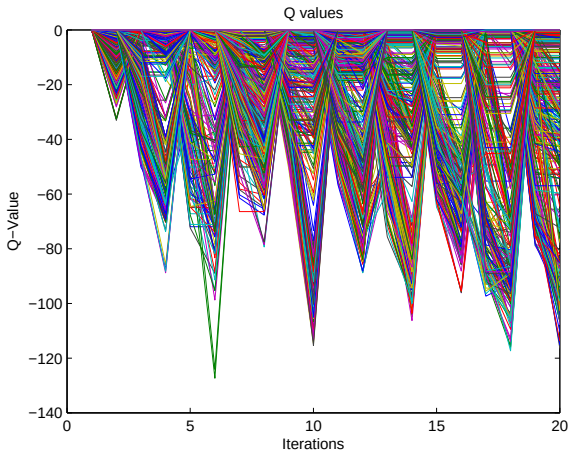
The visual incremental online Q-Learning, as shown in Figure 4.10b, does suffer from the same problem as the visual Q-Learning as the non-incremental version, where ambiguities in the feature representation of the input data lead to messed up Q-values.

In order to further investigate the reasons for the disappointing performance of the online Q-Learning, we analyzed how the Q-values change during training.

First, we looked at the amount of non-zero Q-values after every iteration, once at the beginning, i.e. after initializing with experience replay, and once at the end of all experiments of the iteration, shown in Figure 4.11a. The fact that only less than 1000 values are used, supports the hypothesis, that the learned features lead to ambiguities. Also, the actual Q-values were analyzed during online Q-Learning. In Figure 4.11b it can be seen, that every time the features are relearned



(a) Number of non-zero Q-values during training



(b) Non-zero Q-Values during training

Figure 4.11: Q-Value analysis during online Q-Learning

an entirely different feature representation is learned. Again, this can lead to the previously mentioned ambiguities if multiple states with different Q-values are mapped to the same feature vector. In this case, the new Q-values will be messed up during reinitializing them with experience replay.

5 Conclusion

Finding good features in high dimensional data can be a complex and time-consuming problem. It usually requires a certain amount of knowledge about the data. In order to gain this necessary knowledge at the beginning of reinforcement learning, it is necessary to spend many experiments to explore the data space. Preferably, the learned features are improved simultaneously as the agent is learning. In order to make this approach feasible, it is necessary to improve the speed of feature learning, which was the goal of the thesis.

In the process of this thesis, we initially investigated the performance of various types of autoencoders as a method for feature learning with different numbers of hidden units, optimization methods and noise functions. While all autoencoders were able to get a decent reconstruction error, the incremental autoencoder gave the best performance with respect to speed. Furthermore, the incremental autoencoder showed the best results when dealing with a data set, which is growing over time, making it the the best choice out of the tested methods for online feature learning for reinforcement learning.

When comparing the results for all tested optimization methods, it becomes obvious, that RPROP could achieve the best results in both reconstruction error and necessary iterations until convergence, as seen in chapter 4. These observations appear consistent throughout all tested configurations of noise function and number of hidden neurons.

We then tested these methods in combination with reinforcement learning, i.e. Q-Learning, using different algorithms (see chapter 3). The results of these experiments show, that the learned features can indeed be used to perform the simplified pole-balancing task and find an optimal policy, if the required knowledge is available to the autoencoder. However, if the denoising autoencoder had only partial data available, the learned features could not be used to find the optimal policy. This fact, again, underlines the importance of being able to perform online feature learning. A reinforcement learning agent may be exposed to changes in its environment, leading to bad performance due to poor feature representations unless it is able to adapt its feature representation of the environment it is observing. When testing the online Q-Learning, we received relatively poor performances of the reinforcement learning. These rather disappointing results were examined intensively. The main issue has been identified as the lack of separation of the learned features, which lead to ambiguities during reinforcement learning. As discussed in the previous chapter 4, these ambiguities in the feature representation of the observed states require the agent to relearn multiple Q-values whenever its encoder function for obtaining the feature vector of an observation is changed. This appears to be a fixable problem, however, could not be solved in this thesis due to lack of time.

A starting point to solve these issues in the future is to improve the learned feature space by using deep learning, i.e. stacked incremental autoencoders. A deep structure is more likely to find a good low dimensional feature representation of the high dimensional input data. Additionally, a deep autoencoder might produce better features (for visual input data), when adding convolutional layers to its structure to obtain some information of the spatial relations between each input value. Furthermore, it might be interesting to use a more sophisticated reinforcement learning method than Q-Learning for online reinforcement learning and move to a more complex application than the very simplified pole-balancing task used in this thesis, once all of the previously mentioned problems are fixed.

Bibliography

[1] D. L. Donoho, "High-Dimensional Data Analysis: The Curses and Blessings of Dimensionality," 2000.

[2] G. E. Hinton and R. R. Salakhutdinov, "Reducing the dimensionality of data with neural networks," *Science*, vol. 313, pp. 504–507, July 2006.

[3] 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, 2006.

[4] Y. Bengio, "Learning Deep Architectures for AI," *Foundations and Trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[5] J. Mattner, S. Lange, and M. Riedmiller, "Learn to swing up and balance a real pole based on raw visual input data," in *Neural Information Processing - 19th International Conference, ICONIP*, pp. 126–133, 2012.

[6] 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.

[7] 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," *JMLR*, vol. 11, pp. 3371–3408, Dec. 2010.

[8] M. Riedmiller, "Neural fitted q iteration – first experiences with a data efficient neural reinforcement learning method," in *In 16th European Conference on Machine Learning*, pp. 317–328, Springer, 2005.

[9] G. Zhou, K. Sohn, and H. Lee, "Online incremental feature learning with denoising autoencoders," in *International Conference on Artificial Intelligence and Statistics*, pp. 1453–1461, 2012.

[10] C. J. C. H. Watkins, *Learning from Delayed Rewards*. PhD thesis, King's College, Cambridge, UK, 1989.

[11] Y. Le Cun, L. Bottou, G. B. Orr, and K.-R. Müller, "Efficient backprop," in *Neural Networks, Tricks of the Trade*, Lecture Notes in Computer Science LNCS 1524, Springer Verlag, 1998.

[12] M. Riedmiller and H. Braun, "Rprop - a fast adaptive learning algorithm," tech. rep., Proc. of ISCIS VII), Universitat, 1992.