Hochschule für Angewandte Wissenschaften Hamburg
Hamburg University of Applied Sciences

Bachelorarbeit

Roland Meo

Deep Q-Learning With Features Exemplified By Pacman

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Roland Meo

Deep Q-Learning With Features Exemplified By Pacman

Bachelorarbeit eingereicht im Rahmen des WP: Lernfähige Systeme

im Studiengang Bachelor of Science Technische Informatik
am Department Informatik
der Fakultät Technik und Informatik
der Hochschule für Angewandte Wissenschaften Hamburg

Betreuender Professor: Prof. Dr. Michael Neitzke
Zweitgutachter: Prof. Dr.-Ing. Andreas Meisel

Eingereicht am:

Roland Meo

Thema der Arbeit
Deep Q-Learning With Features Exemplified By Pacman

Stichworte
Maschinen Lernen, Verstärkendes Lernen, Q-Lernen, Tiefes Lernen, Neuronale Netze, Merkmale, Regression, Pacman

Kurzzusammenfassung
Diese Bachelorarbeit behandelt die Entwicklung und Optimierung eines selbständig lernenden Pacman-Agenten, da Pacman über komplexe Zustandsdaten verfügt, welche ein häufiges Problem im Maschinen Lernen darstellen. Eine typische Herangehensweise ist hierbei die Bildung von Merkmalen, eine verallgemeinerte Abstraktion der gegebenen Daten. Im Verstärkenden Lernen werden diese Merkmale genutzt um einen Wert zu berechnen der aussagt wie vorteilhaft eine Situation. Dabei werden meist Verfahren wie lineare Funktions-approximation genutzt. Alternativ wird in dieser Thesis eine andere Herangehensweise vorgeschlagen, namentlich eine Kombination aus Verstärkenden Lernen und Tiefen Lernen mit künstlichen neuronalen Netzen. Ein neuronaler Netz-agent wird implementiert und optimiert bis diese Optimierung als genügend empfunden wird.
Roland Meo

Title of the paper
Deep Q-Learning With Features Exemplified By Pacman

Keywords
Machine Learning, Reinforcement Learning, Q-Learning, Deep Learning, Neural Networks, Features, Regression, Pacman

Abstract
This bachelor thesis deals with the development and the optimization of an autonomous learning Pacman-agent, since Pacman offers high-dimensional state data, which is a common problem in machine learning. A typical approach to this problem is using features, a high-level abstraction of the given data. In reinforcement learning these features are used to calculate a value describing how beneficial a situation is by using prediction methods like linear function approximation. This thesis suggests a different approach by mixing reinforcement learning and deep learning via an artificial neural network. A neural network agent is implemented then and optimized to a level deemed sufficient.

Contents

List of Tables

List of Figures

Listings

1 Introduction

Machine learning has become a popular research area, combining all sorts of algorithms for all kinds of tasks. But what they have in common no matter the task is that all machine learning tasks require some sort of training data. And while the task learned upon might get more complex so most likely does the depth of the input data structure, leading to the curse of dimensionality. This means that with every new parameter in the environment trained upon, the set of states in the environment grows exponentially. That again is blowing up the amount of training data needed to get a good training performance. Resulting also in an increase of time needed to train upon the gathered training data. In the end the computation time or the memory needed might end up being so gigantic, that the particular machine learning task exceeds the capabilities of the existing hardware. Therefore, in this thesis Pacman got chosen as object of study, since it suffers from the same problem. Seeming simple at first, the amount of parameters in Pacman (namely the ghosts, foods, power-pill, Pacman's location, etc.) are resulting in the curse of dimensionality.
One prevalent approach to this problem are features, a high-level abstraction of the high-dimensional input data. Each feature contains specific information of the original input data and with a proper representation of the input data from multiple features, a simpler abstraction is gained. The difficulty then becomes crafting these features, which can be done by handcrafting them or generating them with algorithms. The gained feature abstraction brings then the benefit of making the learned translatable to similar problems, but comes with the risk of not being descriptive enough, leading to learning results not performing as good as anticipated. Fortunately in this case the features are preexisting and got adopted from the paper Luuk Bom und Wiering (2013), so these difficulties aren't encountered.
But one question arises, how to actually use these features? In reinforcement learning there are methods like linear function approximation to estimate the beneficiality of feature combinations returning a descriptive value. But those methods might end up not being refined enough to provide satisfactory results or need more expert knowledge in order to use them sufficient. Instead this thesis takes a different approach by using deep learning via artificial neural networks, since they offer a more generalized regression approach. By processing the features

mapped to a reward signal to an artificial feed forward network, a regression between features and rewards takes place. But since components of a neural network are interchangeable, first finding the right network configuration for the given task isn't trivial. Therefore multiple networks get trained at the same time just being different in one component or parameter. After multiple episodes of training the best performing iteration of the network will then be taken and compared to different network configurations trained on the same task. From this adjusting one component or parameter at the time and starting a new training round for the next component until they are adjusted and some network deemed sufficient in performance is found. Then finally trying to further enhance the existing features, a new feature gets introduced and experimented on.
In order to give the reader a better understanding of the problem ahead and how to accomplish the task of building an self-learning agent, this thesis offers the basic knowledge needed to understand the principles of reinforcement learning and deep learning used, the implemented features, the architecture built and the results of experiments that helped further enhance the network performance.

2 Reinforcement Learning

Reinforcement learning originated from the idea of learning something by interacting with its environment. But before such an experiment can even designed, the task experimented on has to fulfill the Markov Property. As this property is given, the learner (agent) interacts with the environment in discrete time steps , $t = 1, 2, 3, 4, \ldots$.
Not knowing which action to take, the agent must discover which actions for each state are most promising to reach a set goal. While learning the best strategy from one situation to the next, the challenge of trade-off between exploration and exploitation arises, as one might choose the seemingly best action or explore an unknown action, that might be more rewarding in future. (Sutton und Barto, 1998, 1.1 Reinforcement Learning)

2.1 The Markov Property And Markov Decision Processes

2.1.1 The Markov Property

In a reinforcement learning framework an agent transitions from one environment state s to the next s' via an action a, while receiving a reward r from the environment after each transition in the next time step $t + 1$. If from each of these states the next one can predicted, without needing to know the preceded events, the Markow Property is given.
In mathematical terms:

$$Pr\left\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, \ldots r_1, s_0, a_0\right\} \tag{2.1}$$

In an equation 2.1 the assumption is made, that an environment in the present time t responds with an reward at a future time $t+1$ to an action made at time t. Everything that might happen in s_{t+1} is causally dependent by events in the present s_t, a_t, r_t and past $s_{t-1}, a_{t-1}, \ldots r_1, s_0, a_0$.

$$Pr\left\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\right\} \tag{2.2}$$

But looking at equation 2.2, if at the same time the prediction of the future just can be made from knowing present events, the Markov Property is given. (Sutton und Barto, 1998, 3.5 The

Markov Property)
Pacman fulfills this property, since for every state not all previous states have to be known to continue playing or to assume what action to take next. For example if a ghost approaches Pacman, Pacman doesn't need to know how the ghost came close, but can predict from present information how to escape in future states so as not to lose. This makes Pacman suitable for a reinforcement learning task.

2.1.2 Markov Decision Processes

If the Markov Property is given in a task, the task becomes a Markov Decision Processes (MDP). With finite sets of actions and states, the finite MDP is a tuple of:

- S a set of states
- A a set of actions
- $\mathcal{P}^a_{ss'}$ the probability that state s transitions to state s' via action a
- $\mathcal{R}^a_{ss'}$ the immediate reward received after transitioning from state s to state s' via action a

The probability $\mathcal{P}^a_{ss'}$ to translate from one state s to s' as via an action a as is defined as follows:

$$\mathcal{P}^a_{ss'} = Pr\left\{s_{t+1} = s \mid s_t = s, a_t = a\right\} \tag{2.3}$$

The immediate reward expected after transitioning from state s to state s' via action a in the next time step $t+1$ is defined as follows:

$$\mathcal{R}^a_{ss'} = E\left\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\right\} \tag{2.4}$$

Mapping states and actions to numerical values, the behavior of the agent can now be defined as a policy $\pi_t(s, a)$ that assigns a certain probability distribution over possible actions in each state for each time step t. The goal of the reinforcement learning task is therefore to learn the the most promising policy. The following statements in chapter 2 are made under the assumption, that the designated task, is a finite MDP.

2.2 The Agent-Environment

The agent-environment framework is a concept describing how an agent should interact with his environment in an reinforcement learning experiment. At each time step t the agent receives a representation of the environment's state $s_t \in S$, where S is the set of all possible

states. From this state the agent chooses an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of all available actions in state s_t. With the next time increment the agent receives a new state s_{t+1}) and in partially consequence of the chosen action a numerical reward $r_{t+1} \in \Re$, where $\Re$ is the set of all immediate rewards, from the environment.
In terms of Pacman this means the Pacman agent receives an initial game state, in which he chooses an action and promotes the action to the environment. The environment computes all kind of game events like foods, ghosts, etc. and replies to the agent with a new state. From this new state the Pacman agent deduces a score, chooses an action again and so on, until some final state is reached and a new game gets started.

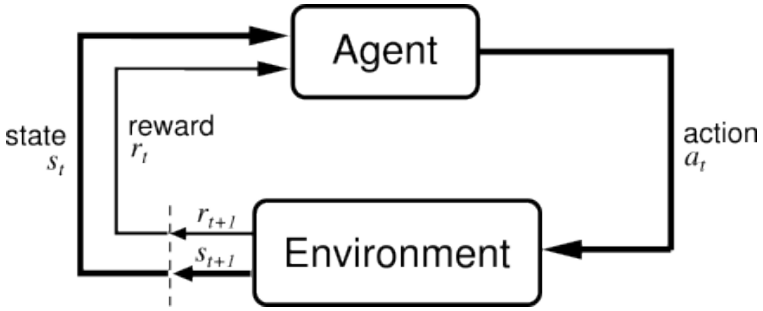


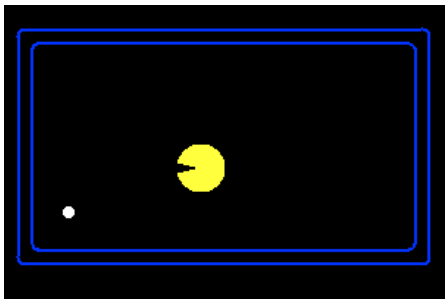
Figure 2.1: Sketch of the functionality principle of the agent-environment, from Sutton und Barto (1998). Showing the $Agent$ receiving reward r_t and state s_t and giving action a_t to the $Environment$ and therefore receiving a new reward r_{t+1} and a new state s_{t+1}.

2.3 The A,$\Re$ And S Spaces In Pacman

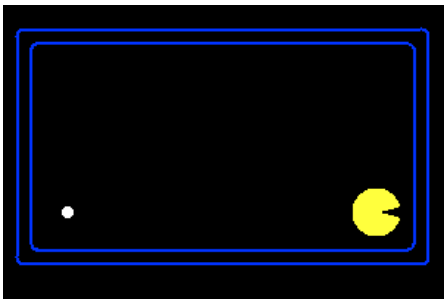
The following section discusses the spaces of actions A, rewards $\Re$ and states S for Pacman, which are essential to the reinforcement learning experiment.

2.3.1 Pacman Action Space A

For each state the agent has four possible actions, these make up the whole space of A. This small action space will become beneficial in section 2.4.3, as the introduced approach is more practical with small action spaces Geramifard u. a. (2013)[p. 388-389]. Those actions are only limited by the walls, as the Pacman cannot walk through them as seen in figure 2.2b.



(a) $\{North, South, East, West\} = A(s_t) = A$



(b) $\{North, West\} = A(s_t) \subset A$

Figure 2.2: Showing the full action space in figure 2.2a and how it changes depending on the situation exemplary shown in figure 2.2b

2.3.2 Pacman Reward Space $\Re$ And The Return

As suggested by Luuk Bom und Wiering (2013) following game events in table 2.1 are triggering the immediate rewards. The sum of all triggered immediate rewards is returned to the Agent. The possible combinations of all retrievable rewards makes up $\Re$.

Table 2.1: game events and the corresponding rewards

Event	Reward	Description
Win	+50	Pacman has eaten all the food
Lose	-350	Pacman collided with a non-scared ghost
Ghost	+20	Pacman ate a scared ghost
Food	+12	Pacman ate a food
Powerpill	+3	Pacman ate a powerpill
Step	-5	Pacman made a step
Reverse	-6	Pacman made a reverse step

2.3.3 Pacman State Space S

The space or set of states in Pacman is quite big compared to the seemingly simplicity of the game. For each field of the game grid there would be $2^5 * 40 = 1280$ theoretical sub-states, since each field carries the information of:

- Pacman on field (binary)
- ghost on field (binary)

- food on field (binary)
- powerpill on field (binary)
- wall on field (binary)
- scared ghost time (counter 0 to 39)

The $|set\ of\ states|$ then differs from the size of the game grid, as the amount of sub-states gets multiplied for each field on the grid. Though a lot of the these states are not possible in the actual game, exemplary considering that Pacman cannot be at multiple places at the same time.

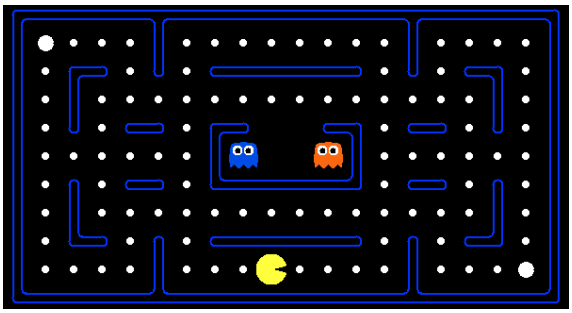


Figure 2.3: The standard game grid has a size of $20 * 11 = 220$ multiplied by 1280 for each theoretical sub-state, so there would be about 281k states, not all of them being possible though. Training an agent on so many states, that occur randomly, since the ghosts have a stochastic moving pattern, would take enormous time. That is why the concept of features gets introduced in section 2.4.3.

2.4 Q-Learning

Section 2.1.2 explained what properties must be given so a reinforcement learning experiment can be designed and section 2.3 introduced a concept of how such an experiment should be designed. Buts it is yet unclear how the Pacman agent actually learns. That's why the next section introduces a control algorithm called Q-Learning, following mostly Sutton und Barto (1998)[6.5 Q-Learning: Off-Policy TD Control].

2.4.1 Value Functions

Before any control algorithm can be implemented, there has to be a metric to measure the value of states and actions. So for measuring "how good" a state s is under a policy π a value function $V^\pi(s)$ is defined as follows:

$$V^\pi(s) = E_\pi \{R_t | s_t = s\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\} \tag{2.5}$$

The value function 2.5 returns the expected long term reward R_t for state s_t, that is the sum of discounted immediate rewards, with gamma $0 < \gamma < 1$ being the discount-factor.
Similarly the benefit of taking an action a in a state s under policy π is defined as Q^π function:

$$Q^\pi(s,a) = E_\pi \{R_t \mid s_t = s, a_t = a\} = E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, _t = a \right\} \tag{2.6}$$

The Q^π function 2.4.2 returns the expected long term reward R_t for a state-action pair (s_t, a_t), that is the sum of discounted immediate rewards, with gamma $0 < \gamma < 1$ being the discount factor.
Both functions 2.5 and 2.4.2 are approximated from experience following a policy π, converging to these functions with every single state encountered infinitely.

2.4.2 The Q-Learning Algorithm

So now having a metric measuring the value of states and state-action pairs and further knowing this only can be learned from experience, an off-policy[1] control algorithm known as Q-learning is used, in order to rate which action is best for each state, hence equation . A single Q-value update is calculated according to:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \right] \tag{2.7}$$

Following equation 2.7, if the agent picked an action a_t in state s_t, the Q-value $Q(s_t, a_t)$ gets updated by the old Q-value $Q(s_t, a_t)$ plus the product of learning rate $0 < \alpha < 1$ and the immediate reward r_{t+1} (resolving from action a_t) plus the discounted best possible Q-value $Q(s_{t+1}, a)$ (of follow state s_{t+1}) minus the old Q-value $Q(s_t, a_t)$.
Knowing how to update Q-values it is possible to dictate the Q-learn algorithm:

```
1 Initialize Q(s,a)
```

[1] Because Q-learning adds the discounted next *best* possible Q-value instead of the actually occurring in its update function, it is called off-policy, this reducing the amount of training rounds needed to learn the optimum.

```
Repeat (for each game):
    Initialize state
    Repeat (for each episode):
        Choose action from state using policy derived from Q
            (by default epsilon-greedy):
        Take action, wait for state' and reward
        Update Q(state,action)
        state = state'
    until state is terminal
```

Listing 2.1: Q-learning pseudo code

After initializing the Q-values arbitrarily, the game-environment gets started. The agent receives an initial state from the environment. From this the agent has to pick an action according to a policy derived from the approximation of equation 2.4.2. If the agent is in training an epsilon-greedy policy is chosen, meaning that by a chance epsilon $0 < \varepsilon < 1$ some random action will be chosen, since as stated in section 2.4.1 the Q^π approximated is learned from experience. The agent has to sometimes pick a random action, because it can't experience all state-action pairs, if it strictly chooses the seemingly best state-action pair with the highest value. Additionally the highest value of an early regression of equation 2.4.2, might be vague or just not correct. Therefore, while training the Pacman agent epsilon will start at 1 and linearly decrease to a set minimum of 0.1, as proposed in Lample und Chaplot (2016). If the Q^π is properly approximated, epsilon is set to zero, taking only the actions from a state with the highest Q-value. After choosing an action and giving it to the environment like in section 2.3, the agent waits for state' and the reward from the environment. With this data received, he then can update the Q-value for the last state-action pair according to function 2.7, if the agent is in training. This happens until a terminal state is reached, starting a new game and repeating the process.
Knowing this it would be possible to write a simple Q-learning agent, but as stated in section 2.3.3 the space of states is huge. That's why in section 2.4.3 the concept of features gets introduced.

2.4.3 Q-Learning With Features (Linear Function Approximation)

In order to get a handle of enormous finite state spaces with $|A| \ll |S|$, it is possible to generalize the state representation with linear function approximation. Francisco S. Melo (2007)

$$Q^\pi_\theta(s,a) = \phi(s,a)^T\theta \tag{2.8}$$

Following equation 2.8, instead of one Q-value for every state-action pair, there is a vector $\phi(s, a)$, each element is a feature and each $\phi_i(s, a) \in \mathbb{R}$ denotes the value of feature i for each state-action pair (a, s). These features are normalized and computed by feature functions. For this Pacman experiment 7 features are chosen and discussed in chapter 4.
There is the vector $\theta \in \mathbb{R}^n$ containing weights for each feature i and therefore measuring the contribution of each feature to the approximated $Q^\pi(s, a)$. As the goal is to get as close to the original Q^π function as possible the weights get adjusted each episode. To do so there is an update function as follows:

$$\theta_{t+1} = \theta_t + \alpha\delta_t\phi(s_t, a_t) \tag{2.9}$$

with

$$\delta_t = r_{t+1} + \gamma \max_a Q^\pi_{\theta_t}(s_{t+1}, a) - Q^\pi_{\theta_t}(s_t, a_t) \tag{2.10}$$

Following equation 2.9 each weight in θ_t gets updated by the learning-rate α multiplied by the error δ_t multiplied by the corresponding feature value in $\phi(s_t, a_t)$. The error δ_t following equation 2.10 consists of the reward received in the next time-step $_{t+1}$ plus the product of the discount-factor γ and the maximal Q-value of the follow state $\max_a Q^\pi_{\theta_t}(s_{t+1}, a)$ minus the current Q-value $Q^\pi_{\theta_t}(s_t, a_t)$.
This is the technique used in the reference agent presented in chapter 6.

3 Deep Learning

Behind neural networks stands the idea of imitating biological nervous systems in a computational approach, by introducing artificial neurons connected to a collection called a (artificial) neural network. Though by far not as powerful as the human brain, neural networks can be used on huge data stacks for all kind of regression and classification problems. In this instance the neural network is used to resemble a function approximation of the Q-function. Though a method for linear approximation was introduced in section 2.4.3, neural networks offer a more general approach to a more complex estimate than a linear approximation. Thus giving a better approximation and showing how to combine reinforcement learning and deep learning techniques exemplary.

3.1 Neurons

The neuron is the atomic unit of the neural network. It consists of inputs, input weights (for each input), a bias (or threshold) and an output-function.

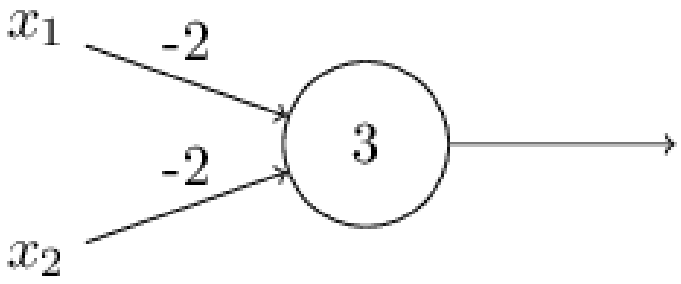


Figure 3.1: Sketch of a perceptron with two inputs x_1, x_2, weights $w_1 = w_2 = -2$ and a threshold of 3, from Nielsen (2015).

One of the simplest forms of neuron is the perceptron as shown in figure 3.1. The shown perceptron implements a $NAND$ gate as its output gets computed like:

$$output = \begin{cases} 1, & \text{if } \sum_{n=1}^{j} x_n w_n \geq threshold \\ 0, & \text{if } \sum_{n=1}^{j} x_n w_n < threshold \end{cases} \tag{3.1}$$

(**Note**: If the threshold gets shifted to the other side of the equation its called the bias b)
Though this isn't a good example of the utility of perceptrons in practice. It shows that in theory every practical computing machine could be realized, as every gate can be built from NAND gates. In practice it is more common to use other neurons than perceptrons, for example the sigmoid neuron. The difference between those and the perceptrons is, they use different output-functions. For instance the sigmoid output gets computed like:

$$\sigma_{output} = \frac{1}{1 + exp(-z)} \tag{3.2}$$

So with bigger $z = \sum_{n=1}^{j} x_n w_n + b$, the σ_{output} gets closer to 1 and with smaller z closer to 0. Knowing this it is now possible to construct more complex computing structures, by interconnecting the outputs and inputs of multiple neurons.

3.2 Deep Feedforward Networks

If multiple neurons get connected, they are called a neural network. The construction of neural networks is accomplished in layers as shown in figure 3.2

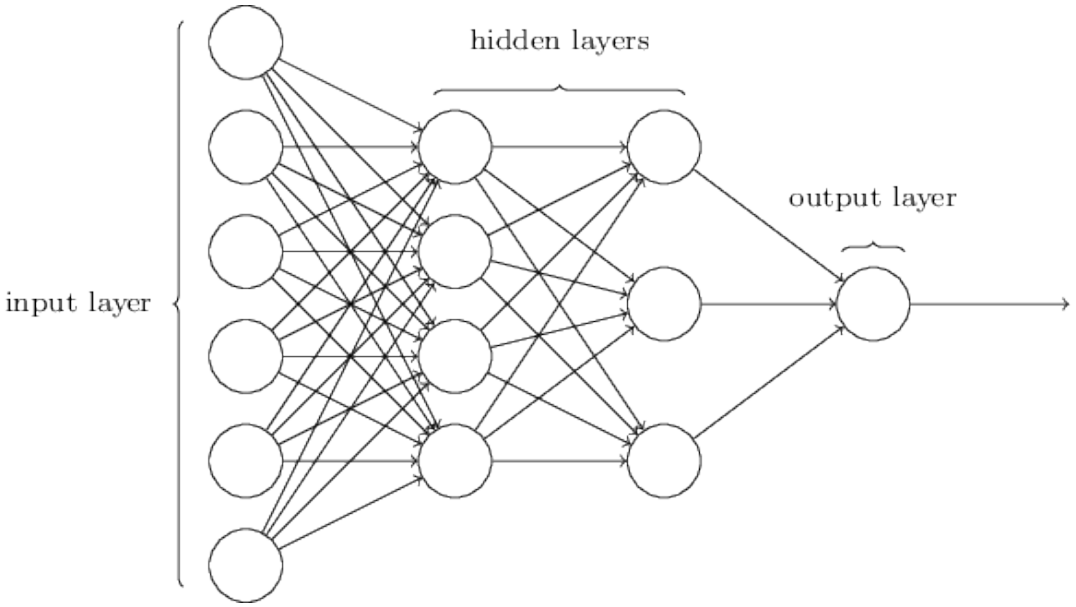


Figure 3.2: Sketch of a neural network, from Nielsen (2015).

Each neuron of each layer is connected with each neuron of the next layer via output-to-input. The first layer is the input layer, where there is one neuron for each input value, following this layer are n hidden-layer and finally one output layer. Each of these layers consists of 1 to k neurons. The kind of neural network used in this experiment is called a feed forward network, simply because input gets fed to the first layer and wanders from front to back until the output layer is reached. The network used consists of an input layer of 7 input neurons, a hidden layer of 50 sigmoid neurons and an output layer of simply one neuron, whose output is real-numbered (not just from 0 to 1). With just $7+50+1 = 58$ neurons and $7*50+50*1 = 400$ weights instead of thousands of Q-value pairs the networks is approximating the Q-function. This will allow to estimate the Q-value by deriving the features of each possible follow state and feeding them to the network, retrieving possible Q-values. If the network is well trained, the action that leads to features with the highest Q-value,will be taken as the best action.

3.3 Gradient Descent With Backpropagation

3.3.1 Gradient Descent

The sections 3.2 and 6.3.3 gave an idea of how the network is constructed and computes outputs, but it is still unclear how the network learns. It seems reasonable to suppose that alteration of

weights is the way to go, but how are they altered, as small changes affect not just one neuron but all follow neurons connected. First training data is needed, meaning multiple input data or feature vectors x and the corresponding correct outputs $y(x)$. With this it is able to define a cost function $C(w, b)$, dependent on the weights w and the biases b.

$$C(w,b) = \frac{1}{2n} \sum_{x} \parallel y(x) - a \parallel^2 \tag{3.3}$$

This cost function (also known as *mean squared error*) allows to measure how close the network output a is to the training data output $y(x)$ on n training samples, with $C(w, b) \approx 0$ being the optimum. In furtherance of reducing the cost/error an algorithm called gradient descent is adjusting the weights and biases to the gradient of the function $C(w, b)$ in $0 < \eta \leq 1$ sized steps, so that for each iteration there is a negative $\triangle C$, till some minimum of the cost function is reached. For this the k weights and l biases get updated thusly:

$$\begin{aligned} w_k &\rightarrow w'_k - \eta \frac{\partial C}{\partial w_k} \\ b_l &\rightarrow b'_l - \eta \frac{\partial C}{\partial b_l} \end{aligned} \tag{3.4}$$

3.3.2 Backpropagation

The last open question is how to get $\frac{\partial C}{\partial w_k}$ and $\frac{\partial C}{\partial b_l}$ of equation 3.4 or more figuratively the gradient of of the cost function. The answer is an algorithm called backpropagation. For the backpropagation algorithm four equations are needed.

First Equation

The first equation δ^L_j computes the error in the output layer L of each j^{th} neuron.

$$\delta^L_j = \frac{\partial C}{\partial a^L_j} \sigma'(z^L_j) \tag{3.5}$$

Equation 3.5 measures how fast the cost grows for each j^{th} output activation a^L_j, multiplied by the change of the activation function σ to the weighted input z^L_j. In a matrix-based form equation 3.5 can be written as:

$$\delta^L = (a^L - y) \odot \sigma'(z^L)^1 \tag{3.6}$$

[1] $\odot$ being the Hadamard product exemplary computed like: $\begin{bmatrix}1\\2\end{bmatrix} \odot \begin{bmatrix}3\\4\end{bmatrix} = \begin{bmatrix}1 & * & 3\\2 & * & 4\end{bmatrix} = \begin{bmatrix}3\\8\end{bmatrix}$

Second Equation

The second equation δ^l computes the error in each layer l.

$$\delta^l = ((w^{l+1})^T \delta^{l+1} \odot \sigma'(z^l) \tag{3.7}$$

Multiplying the transposed weight matrix $(w^{l+1})^T$ of the next layer $l+1$ to the error of the its layer δ^{l+1} and again forming the Hadamard product of this product to $\sigma'(z^L)$, the error of the recent layer l gets calculated. Combining the equations 3.6 and 3.7 it is possible to compute the error δ^l for each layer l from back to front, starting with $\delta^L, \delta^{L-1}, \delta^{L-2}, \dots, \delta^2$.

Third Equation

The third equation $\frac{\partial C}{\partial b^l_j}$ computes the gradient of cost with respect to any bias in the network:

$$\frac{\partial C}{\partial b^l_j} = \delta^l_j \tag{3.8}$$

The third equation can be written as the matrix-based equation 3.9 and is then the same as the **second equation**:

$$\frac{\partial C}{\partial b^l} = \delta^l \tag{3.9}$$

Knowing this one unknown of the equation 3.4 hence $\frac{\partial C}{\partial b_l}$ can be solved.

Fourth Equation

The last and fourth equation $\frac{\partial C}{\partial w^l_{jk}}$ computes the gradient of cost with respect to any weight in the network:

$$\frac{\partial C}{\partial w^l_{jk}} = a^{l-1}_k \delta^l_j \tag{3.10}$$

With a^{l-1}_k being the activation of a neuron k of a previous layer $l-1$ and δ^l_j being the error of the current neuron j in the layer l. Knowing this second unknown of the equation 3.4 hence $\frac{\partial C}{\partial w_k}$ can be solved.
Keeping these four functions in mind it is possible to compute the δ of each weight and bias recursively from back to front and update it, hence the name backpropagation.

4 The State Representation (Features)

The agent has no direct view of the game grid, instead it is offered a high level abstraction of the game state via features as introduced in section 2.4.3. As these features are an abstract representation of the game state a policy transfer to another game grid is possible and will be demonstrated in chapter 6. In conclusion, the agent can just perform as good as the state abstraction is in expressiveness, hence an agent will always perform bad on an insufficient abstraction no matter how good the training.
Since the design of handcrafted features relies on expert knowledge, seven features derived from the paper Luuk Bom und Wiering (2013) are introduced. All features are normalized to $0 \leq x < 1$ with $x \in \mathbb{R}$ as suggested for the function approximation 2.4.3.

4.1 Level Progress

In Pacman the main goal is to eat all the food, while avoiding the ghosts. Hence, the first feature represents the amount of eaten food, letting the agent know his progress in the game.

- $a =$ total food
- $b =$ eaten food

The progress feature $f_{progress}$ gets computed as follow:

$$f_{progress} = (a - b)/a \tag{4.1}$$

4.2 Powerpill

When the Pacman eats a powerpill the ghosts become scared and eatable for some time, making it possible to score some extra points. The agent needs to learn when it is profitable to engage eating a ghost with the time left on the scared timer or if he should persuade other goals. By default the scared timers of the ghost are set to 0, with the consumption of a powerpill, the timer gets set or refreshed to a default scared time.

- $SCARED\_TIME$ = the maximal time of the ghost being scared
- $scared$ = the biggest scared timer of the present ghosts

The powerpill feature $f_{powerpill}$ gets computed as follow:

$$f_{powerpill} = \begin{cases} 1 - (SCARED\_TIME - scared)/SCARED\_TIME, & \text{if } scared > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

4.3 Pills (Food)

In order to make any progress Pacman needs to know where the nearest food is. Hence, the food feature (or pill feature) is rating how far the closest food is apart from Pacman. The closest food to Pacman is found via breadth-first search.

- a = Maximum path length [1]
- b = Shortest distance to next pill

The food feature f_{food} gets computed as follow:

$$f_{food} = (a - b)/a \tag{4.3}$$

4.4 Ghosts

On the contrary to feature 4.3 if Pacman wants to avoid losing, he needs to dodge the ghosts. Therefore, the ghost feature gives a rating of the distance to the closest ghost. The closest ghost is the ghost with the smallest Manhattan distance to Pacman. [2]

- a = Maximum path length [1]
- b = Shortest distance to next ghost

The ghost feature f_{ghost} gets computed as follow:

$$f_{ghost} = (a - b)/a \tag{4.4}$$

[1] The length of longest path gets calculated once with the initialization of the game grid
[2] The Manhattan distance between two vectors p, q is calculated as follow: $d(q, p) = \sum_{i=1}^{n} \mid p_i - q_i \mid$

4.5 Scared Ghosts

This feature is dependent on the powerpill feature 4.2. If there is no scared ghost, this feature is set to zero, as zero is equal to the furthest distance. If then a ghost becomes scared, its distance gets measured. Though it seems profitable to engage a scared ghost, as a collide offers a lot of extra points, the agent hast to regard that the scared timer decreases, hence the powerpill feature 4.2, risking to run into a non scared ghost and lose if the timer runs out.

- $a =$ Maximum path length [1]
- $b =$ Shortest distance to next scared ghost

The scared ghost feature f_{scared} gets computed as follow:

$$f_{scared} = \begin{cases} (a-b)/a, & \text{if } f_{powerpill} > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4.5}$$

4.6 Entrapment

Sometimes while engaging food, Pacman also engages a ghost at the same time. In this case it is beneficent to know, if there are any escape roots following the path to the food. In order to achieve this the entrapment feature is introduced, measuring the the amount of escape roots (or crossings) in the moving direction. Since every crossing with 3 or more moving directions is a possible escape root and thus considerably safe.

- $a =$ amount of all crossings in the game grid
- $b =$ amount of crossings following the present direction

The entrapment feature $f_{entrapment}$ gets computed like this:

$$f_{entrapment} = (a-b)/a \tag{4.6}$$

4.7 Action

In some games a situation occurs where 2 choices of action seem to be equally good. In this case the wanted behavior is to keep Pacman following his current direction. Thus the action feature is implemented, simply checking if the last moving direction matches the current direction.

- $d_{last} =$ the last moving direction

- $d_{current}$ = the current moving direction

The action ghost feature f_{action} gets computed like this:

$$f_{action} = \begin{cases} 1, & \text{if } d_{last} == d_{current} \\ 0, & \text{otherwise} \end{cases} \tag{4.7}$$

5 Architecture

Chapters 2 and 3 establish the needed knowledge to understand the mathematical and theoretical aspects of the Pacman agent and chapter 4 introduces the used features. This chapter is explaining the architecture and the components used to run the associated experiments in chapter 6, hence giving an idea of how the agent works software-wise and making it possible to adopt this architecture for similar problems.

5.1 Third Party Software

In this section some important third party frameworks and libraries are acknowledged, without which this Pacman experiment wouldn't have be possible in given amount of time.

5.1.1 Pacman Framework Of CS 188

The Pacman framework of the UC Berkeley's introductory artificial intelligence course, CS 188 is used as a foundation for this experiment. For it offers a framework that has a good setting for learning the basics of reinforcement learning. It has a predefined environment, with states that hold all relevant game data and a pre-defiend reward signal (,though it is not in-use in these experiment). And an agent interface, that makes it easy to implement new agents, just needing the user to keep some name conventions on the agent's name and the agents to have a function *getAction(self, state)* returning an action to the model.

If looking back at section 2.3 these conditions cover the full agent-environment concept. The inner works of the agent are then up to the user to be designed.

5.1.2 Keras - Deep Learning Library For Theano And TensorFlow

Keras is a high-level deep learning library Chollet (2015) running on top of the libraries Theano a library for defining, optimizing and evaluating mathematical expressions with multi-dimensional arrays Theano Development Team (2016) and Tensorflow is an interface for expressing machine learning algorithms, and an implementation for executing such algorithms

Abadi u. a. (2015).
In the end Keras, which made implementing all kind of networks variations used easier, is used with a Theano back-end. Since at the time building the agent, the current Tensorflow-built is causing to much swaps on the memory of the server and hence is much slower than Theano for this experiment.

5.2 Main Classes

Using the Pacman framework of section 5.1.1, the implemented agent has to take states from the already given environment and give an action to the environment every discrete time step. Looking at figure 5.1 the agent *NeuroKAgent*[1] is dependent on three other classes. These being the *NeuralControllerKeras*[2] class for handling the neural network, the *RewardHandler* class that creates a signal according to the section 2.3.2 and the *StateRepresenter* class extracting the 7 features mentioned in section 4.

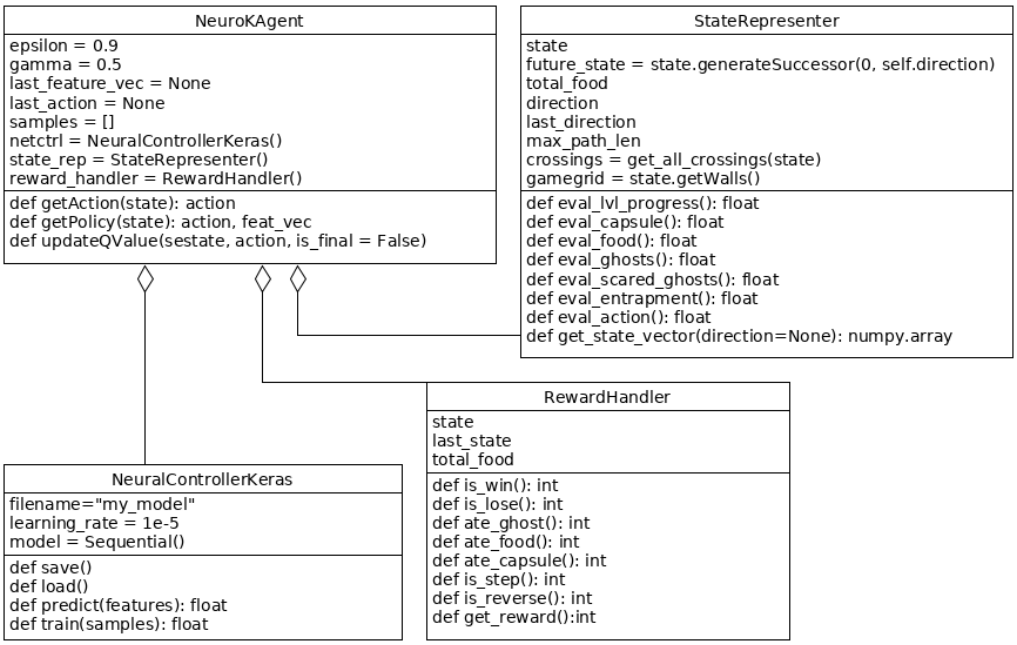


Figure 5.1: Class diagram of the most important classes.

The following sections will discuss the classes shown in figure 5.1 int detail.

[1] NeuroKAgent standing for Neural Keras Agent
[2] Since there is more initial experimentation with different neural network frameworks, the name convention NeuralController_ emerges

5.2.1 RewardHandler

The RewardHandler class creates the reward signal for the agent according to table 2.1. In order to do so the RewardHandler needs the last state, the current state and the current action. Thanks to the framework the following data can just be read from the current state:

- is lose
- is win
- ate ghost

By comparing the last state and the current state the RewardHandler can register if:

- Pacman ate food
- Pacman ate a powerpill
- Pacman made a step

And at last with the current action, the RewardHandler can measure if Pacman has done a step backwards, since the future Pacman position following the action, might lead to the same the position of the last state. The RewardHandler multiplies the binary signal values to the corresponding score, summarizes them and returns the sum to the agent.

5.2.2 StateRepresenter

The StateRepresenter implements the features listed in chapter 4 and returns an array of real numbered values between 0 to 1.

5.2.3 NeuralControllerKeras

The NeuralControllerKeras class controls the neural network. With its initialization it creates a model of the network according to the given parameters, with random weights. From this the NeuralControllerKeras can:

- $load$ a different model and its weights
- $save$ the current model and its weights
- $predict$ game-score according to the needed input (commonly the 7 features)
- $train$ the network on sample data provided by the agent (commonly a set of feature-combinations and their correlating game-scores)

5.2.4 NeuroKAgent

The NeuroKAgent is the centric class and holds one instance of RewardHandler, StateRepresenter, NeuralControllerKeras each. It also defines the reinforcement learning parameters needed for the series of experiments. These being $\varepsilon = 0.9$ for the random policy choice linearly decreasing with further training episodes and $\gamma = 0.5$ the discount-rate for the previous reward. [3] The NeuroKAgent class is responsibly for choosing an action according to the policy and giving it to the Pacman-framework, receiving rewards from the environment and mapping them to the last state's features adding them to a data set and retraining the network after the final game of a training episode. For this the class offers four methods:

- $getPolicy$ is choosing an action to an epsilon-greedy policy (if not in training $\varepsilon = 0$)
- $updateQValue$ named according to the Q-Learning algorithm this method is adding data samples to the data set
- $getAction$ the only method demanded by the Pacman-framework for receiving a new state and returning a chosen action
- $final$ an optional method offered by the Pacman-framework, called after every game, in this instance called to write logs, add the data of the final action to the data set and if it should be the last training game retrain the network

5.3 Class Interaction

After explaining how each of the main classes work on their own, the sequence diagram in figure 5.2 shows how these classes and the enviroment interact witch each other.
The class entities collaborate in four major steps.

1. $start\ framework$:
   Via command-line the framework gets started and initializes the agent given in the arguments.

2. $getAction(state)$:
   After everything is set up, the environment offers a state to the agent, waiting for an action. (If this is the first game the agent initializes his helper classes and the network)

   a) $getPolicy(state)$:
   The agent picks an action, by estimating all possible future states, getting their

[3] Note that α is not needed for the deep learning variant of the agent, since it comes with its own learning rate or step-size.

features vectors and getting those evaluated from the neural network. He then choses an action according to an ε-greedy policy (with $\varepsilon = 0$ if not in training)

b) $updateQValue(feat\_vec, action)$:
If the agent is in training mode, he "updates" the Q-value of the state-action pair (except this turn is the first turn). In reality the correct Q-value will be approximated adding this value and the feature vector of the last state-action pair to the training samples for the neural network.

The agent return the chosen action to the environment. These steps repeat until a final state is reached, hence Pacman lost or won.

3. $final(state)$:
If a final state is reached, the environment calls the final function of the agent, passing him a final state with the final score.

   a) $updateQValue(feat\_vec, action)$:
   If the agent trains, he adds a training sample for the feature vector that lead to the final state.

   b) $train(random\_samples)$:
   If this game happens to be the nth training game in a series of training games, the agent picks a percentage of random training samples of all samples gathered and trains the network on them. After the network is done learning, it will return a loss, hence the mean squared error of the training samples to its outputs.

   c) $write\ to\ logs$:
   Every time a game is done, the agent will write relevant data to logs.

4. $quit\ framework$:
If the last game was the last game in a series of games, the framework will shutdown.

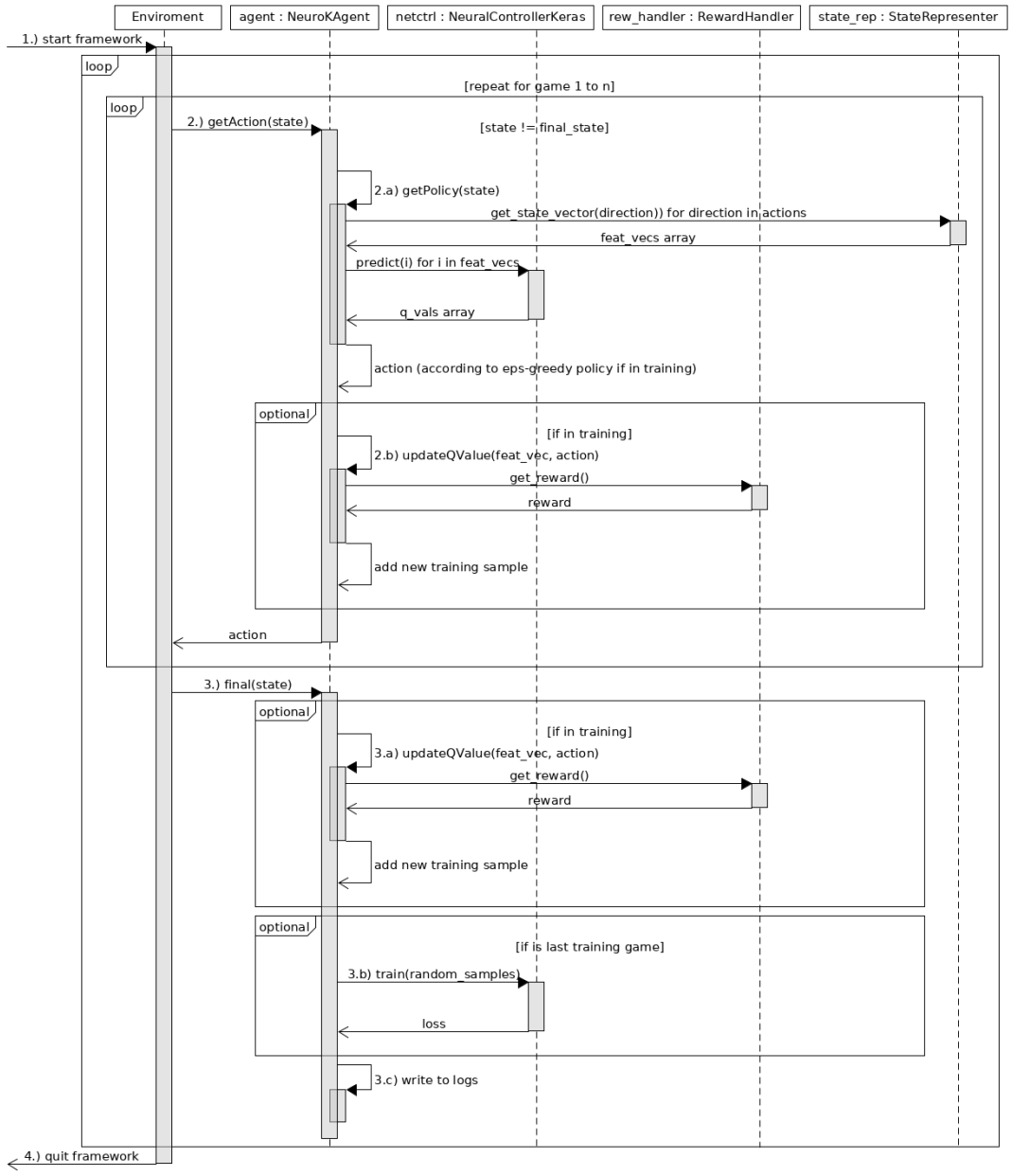


Figure 5.2: Sequence diagram of the main classes interacting

6 Experiments

Still, In this chapter the effort of proving the assumption that artificial neural networks offer an alternative solution to classic regression approaches gets proven via experiments. For this an agent is implemented using the linear approximation in order to compare its performance to the more refined neural network agents. Further experiments attempt finding a sufficient neural network configuration, as the attempt to find the right components and (hyper-) parameters isn't a trivial one. Therefore, different types of neurons, learners (or trainers) and step sizes (or learning rates) are experimented on and compared in order to find a fitting configuration. Notice that the result doesn't raise the claim of being the optimal solution, as solutions for these kind of problems rely strongly on the given task and an optimization is only pursued to a level deemed sufficient to the task.
After finding a satisfactory neural network a comparison to a linear approximation agent is made. In order to prove the assumption, that behavior learned from features translates to different game-grids, an agent with the best network configuration found in the former experiments is playing on the so called "contest game-grid". From that seeing that the features of chapter 4 might be improved by a new feature named "Powerpill-distance", an agent with the new feature gets trained and compared to the former best "7 feature" agent on the training as well as the contest game-grid. Showing that there is still room for improvement in the features suggested by the paper Luuk Bom und Wiering (2013).

6.1 Metrics

Indispensable to any kind of experiment are certain metrics to evaluate the success of the experiment Originally the assumption that taking the loss of the MSE function and seeing it converge to zero would be a sufficient metric to evaluate the training success is made. But looking at figure 6.1, even for a successful agent the MSE doesn't converge fast enough towards zero. In addition, it is noisy, therefore making it hard to derive any information from it.

Figure 6.1: The MSE returns while training the sigmoid agent in section 6.3.

Ergo, as a practical alternative to the MSE, the mean score of 10 fixed [1] games at the end of each episode is used as a metric, since the goal of the agent is to maximize the game-score via cumulated rewards. The paper Mnih u. a. (2013) also suggests the use of the average total reward as a satisfactory metric, even though the average total reward metric tends to be very noisy, because small changes to the weights of a policy can lead to large changes in the distribution of states the policy visits. For this reason, the following figures showing the learning behavior of the agents are having a regression curve plotted above them, making it easier to evaluate the overall training progression.

6.2 Lineaer Approximation

Trying an old less refined method called linear function approximation, an agent tries to learn weights for each of seven features in chapter 4. These weights rate how beneficial their corresponding feature is in return. Hence, the agents plays 1000 episodes of 50 games each, taking a snapshot of the weights after each episode, while consistently updating the weights each turn in the game according to the formulas in chapter 2.4.3.

[1] fixed games means, that the usual random ghost movement is calculated with a fixed seed for pseudorandom numbers, resulting in the same ghost behavior every series of games.

Figure 6.2: Sketch showing the learning behavior of the linear approximation agent over 1000 episodes

From this an agent behaving as seen in figure 6.2 is gathered. The agent has a steep learning curve in the beginning and reaches its possible best after about 200 to 300 episodes. But it doesn't really get better after this. Looking at the figure again, there is a lot of noise in the on-going later episodes. This might come from the fact, that a linear approximation isn't just enough to comprehend the complex state-action space of Pacman with the features and this method given. Seeing that the maximum mean score reached is about 300 points and there aren't much optimization possibilities, it is concluded that this approach is just not satisfactory.

6.3 Different Networks

When First learning of neural networks most literature chooses the example of a classification problem with sigmoid neurons and a stochastic gradient descent (SGD) trainer [2]. But since playing Pacman or more precise the approximation of the Q-function via feature abstractions is a regression task, it is unclear how good these approaches translate. Though thanks to Luuk Bom und Wiering (2013) there is a suggestion regarding the amount of neurons in each layer, but it is unclear what neurons and trainer (or learner) to use for more satisfactorily results.

[2] SGD trains on multiple random subsets of the provided training data (called minibatches), instead of the whole training set like gradient descent in section 3.3.1

6.3.1 Training Setup

In order to train the different network setups each agent will have a neural network consisting:

- input layer of 7 input neurons [3]
- hidden layer of 50 sigmoid/rectifier neurons
- output layer of 1 linear neuron
- and a network trainer for backpropagation

A training session lasts for 1000 episodes, each episode consisting again of 50 random games on the training game-grid to gather a training set and train the network from random batches of this set. After the retraining of the network is conducted, the agent will play 10 fixed evaluation games, take the mean score of these games as a measure of performance. If the agent should be the so far most promising agent, a backup of the current network will be taken.
After a training session the best performing networks will be taken for comparison. Playing 200 fixed games on the training grid.

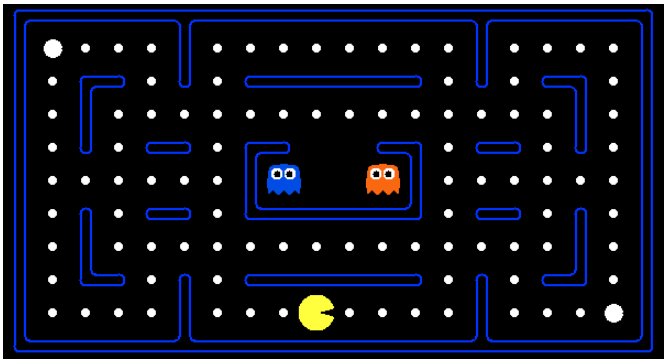


Figure 6.3: Sketch of the training game-grid

6.3.2 Sigmoid Vs Rectifier

In common neural network literature like Nielsen (2015) or Goodfellow u. a. (2016) the neurons referred or used are mostly sigmoid neurons. But Glorot u. a. (2011) suggests that rectifier neurons (also known as ReLU [4]) hold more benefit (especially in deeper networks), as they are closer to the actual activation function of biological neurons. Also looking at 6.4 for the sigmoid function with z [5] getting bigger (or smaller), the gradient of the sigmoid function

[3] one for each feature
[4] ReLU stands for rectified linear unit
[5] $z = \sum_{n=1}^{j} x_n w_n + b$ being the sum of all inputs x multiplied by their weights w plus the bias b

vanishes. While at the same time for $z > 0$ the gradient of the rectifier function is constant at 1. Resulting in an overall slower learning for sigmoid neurons compared to ReLU.

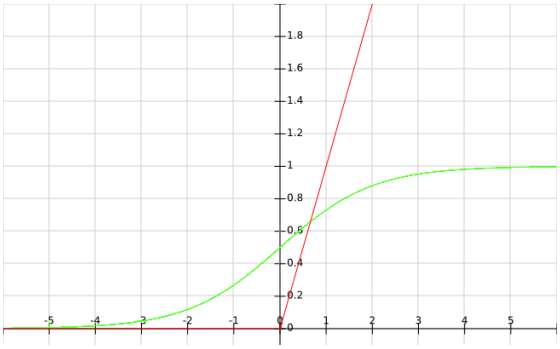


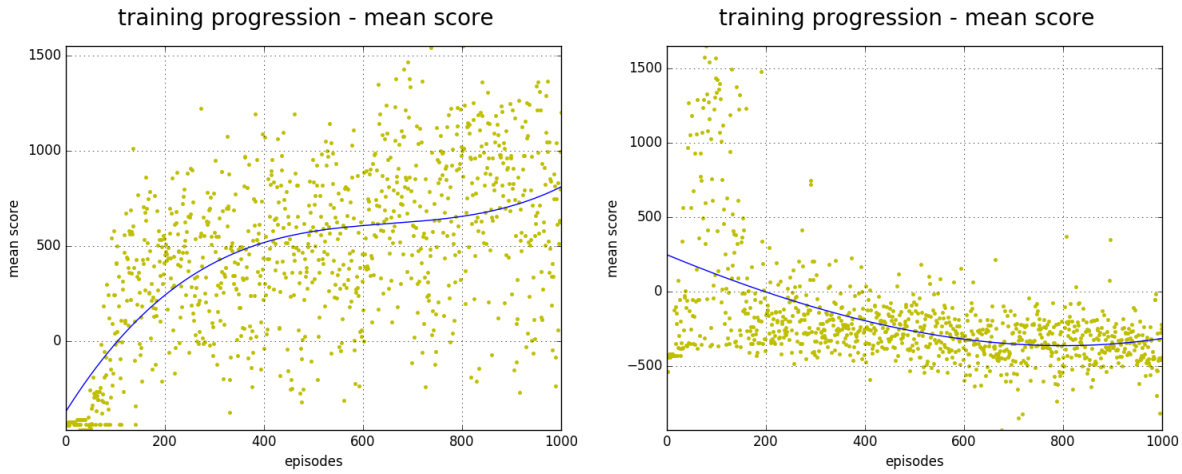
Figure 6.4: Sketch showing the sigmoid function (green) and the rectifier function (red)

Looking at 6.4, one of the key differences between sigmoid and ReLU is the function range:

- $\sigma_{(z)} : \mathbb{R} \rightarrow (0, 1)$
- $ReLU_{(z)} : \mathbb{R} \rightarrow [0, z]$

Another possible benefit is that the neural network gains sparsity, which potentially improves the estimation for rarer occurring training events, like the event of eating a ghost. Since following equation 6.1 inputs $z \leq 0$ will be forwarded with zero, instead of small values close to zeroes as the sigmoid function does, causing to add real zeroes to the neural network, hence sparsity. Meaning that neurons that would directly contribute to the scared ghost feature only fire then their $z > 0$, but since all neurons of each layer are inter-connected with the neurons of the next layer this might not happen, still the agent might take some benefits from this to some degree.

$$ReLU_{output} = \begin{cases} z, & \text{if } z > 0 \\ 0, & \text{if } z \leq 0 \end{cases} \tag{6.1}$$



(a) Sigmoid agent

(b) Rectifier agent

Figure 6.5: Training progression of the two agents with different neurons, taking the mean scores of the 10 fixed games after each episode and plotting a regression curve above them.

Looking at the graphs in figure 6.5a the sigmoid agent slowly gets better per average with the episodes, having its best performing neural network in episode 801. In figure 6.5b by comparison the rectifier agent learns much faster and spikes to his maximum in episode 79, from this constantly getting worse. Since neural networks are very hard to debug, the assumption is that this might come from a too big step size.
Taking the best performing networks of each agent, each agent plays 200 games with the results presented in table 6.1.

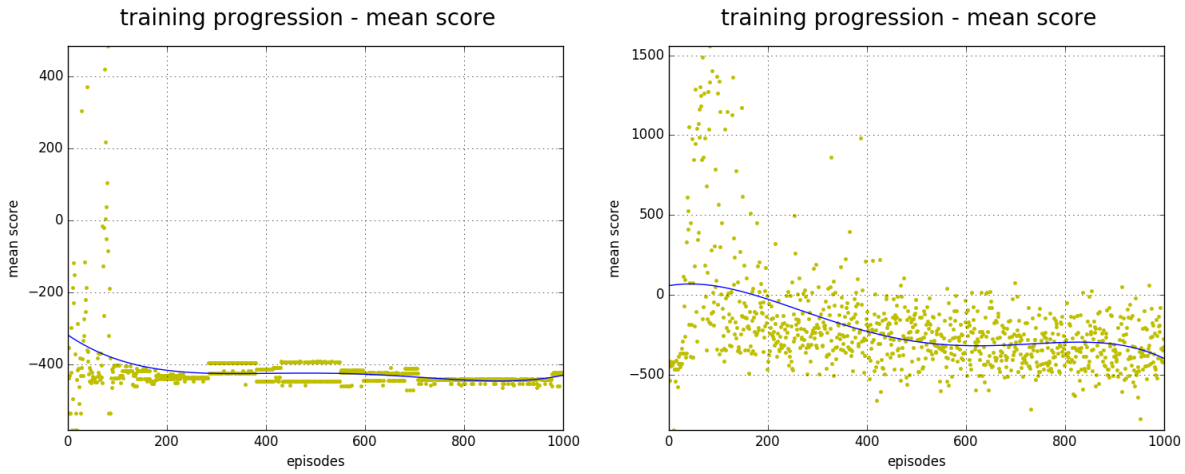
Table 6.1: Results of 200 fixed games with the best rectifier and the best sigmoid network

Network	**Win ratio**	**Avg. score**	**Avg. food eaten/ game**	**Avg. ghosts eaten/ game**
Sigmoid_{801}	69%	1090	88%	1.140
Rectifier_{79}	81.5%	1261	92%	1.300

Examining figure 6.5 the sigmoid agent shows a more consisting learning behavior than the rectifier agent, but looking at table 6.1 the best rectifier agent still wins about 12.5% more games than its sigmoid competitor. The grief problem is that in the current configuration, the learning curve of the rectifier agent is unstable, probably caused by a too big step size parameter η in the network trainer. Still, primarily wanting to enhance the performance of the agent, the rectifier neurons are chosen for further experiments, trying to resolve the unstable learning curve in section 6.3.4.

6.3.3 The Right Network Trainer

Too further enhance the performance of the neural network, the previously black-box treated network trainer is exchanged for another. For this reason the commonly proposed SGD trainer is competing against a network trainer called *Adam*, which holds benefits to noisy and/or sparse gradients which is desirable since one can't comprehend the $\vec{feature}$-reward space, introduces momentum (meaning that some fraction of the previous update is added to the current update, speeding up the learning process) and computing individual adaptive learning rates coming from gradient estimates. Kingma und Ba (2014)
In order to compare both approaches two agents are trained 1000 episodes, both agents only distinguished by the trainer. After the training session the data presented in figure 6.6 is gained.

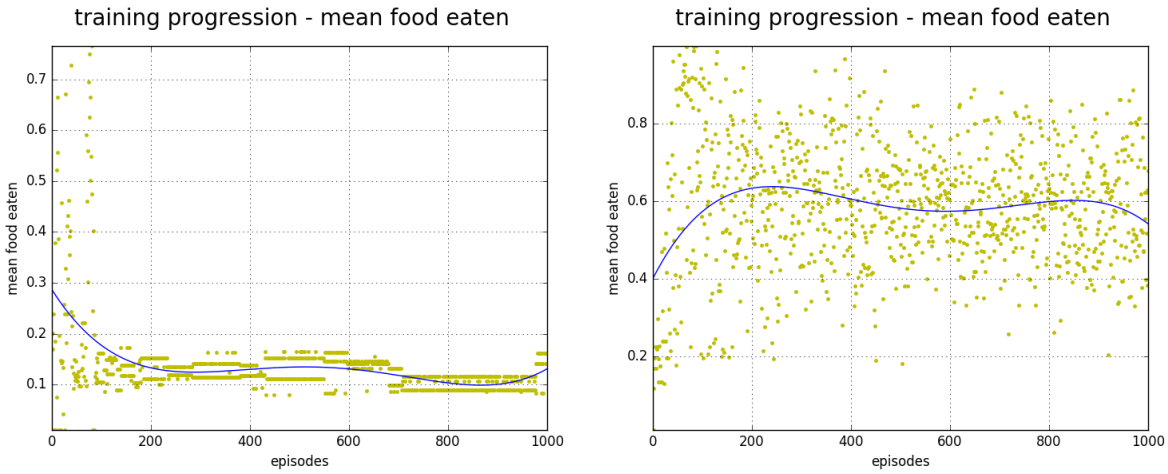


(a) SGD agent

(b) Adam agent

Figure 6.6: Training progression of the two agents with different trainers, taking the mean scores of the 10 fixed games after each episode and plotting a regression curve above them.

Looking at figure 6.6 the SGD agent behaves similar to the Adam agent, progressing steeply in early episodes but getting worse with further training. So for a more differentiated comparison the average amount of eaten food over the 10 fixed games is taken. It might be that the absolute training goal of the agent is to maximize the game-score, but it is also interesting to see the percentage of food eaten, since a game-grid with no food left is considered a win. Hence the amount of food eaten is equivalent to the game progress. Therefore, in figure 6.7 it is visible, that the Adam agent out-performs the SGD agent by far since it averagely eats three times as much food as its competitor.



(a) SGD agent

(b) Adam agent

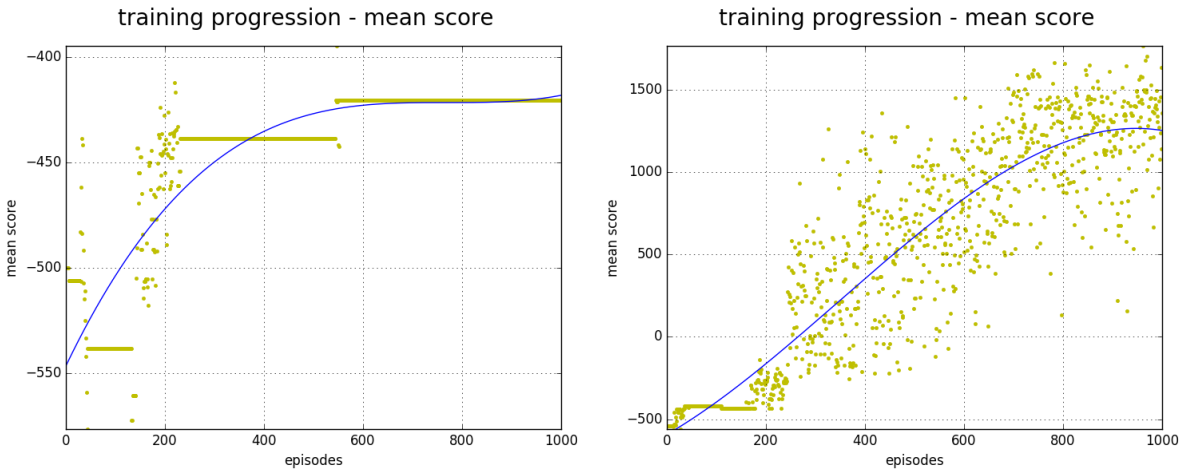
Figure 6.7: Training progression of the two agents with different trainers, taking the average amount of food eaten in 10 fixed games after each episode and plotting a regression curve above them.

Coming to the conclusion that even though the Adam agent digresses in performance visible in figure 6.6b, it somehow maintains some average utility observing figure 6.7b, hence the Adam trainer is used for further experiments.
But even with a better trainer the problem of unstable or decreasing learning behavior is still present. The assumption is made that this might come from a to big step size $0 < \eta \leq 1$, which is set to 0.001 by default. This is resulting in overshooting the optimum while trying to follow the gradient in the $\vec{feature}$-reward space.
Accordingly, the next goal is to balance the step size, that neither overshoots nor converges too slowly.

6.3.4 Step size

The step size $0 < \eta \leq 1$ is a so called hyperparameter and cannot be learned from the standard deep Q-learning task, instead it has to be predefined. Section 3.3.1 states that the step size defines by how much the weights of the network get updated, resulting in the risk that a too small step size leads to slow learning and a too large step size might lead to overshooting the minima of the gradient.
In figure 6.6b the training behavior of an agent with to big step size parameter $\eta = 0.001$ is visible. In order to improve, two agents with smaller step size parameters ($\eta_{agent1} = 10^{-4}$ and

$\eta_{agent2} = 10^{-5}$) are trained, potentially resulting in stable training behavior.
Again after a training session the data presented in figure 6.8 is gained.



(a) agent$_1$ with $\eta = 10^{-5}$

(b) agent$_2$ with $\eta = 10^{-4}$

Figure 6.8: Training progression of the two agents with different step size parameter η, taking the mean score in 10 fixed games after each episode and plotting a regression curve above them.

Looking at figure 6.8 both regression curves are constantly increasing, which is an improvement compared to the curve in figure 6.6b. But looking at the y-axis of figure 6.8a, the rewards never leave the negative value range. This supports the assumption that the step size $\eta = 10^{-5}$ is just too small, since the agent is consecutively improving but much too slow.
Analyzing figure 6.8b gives the wanted results. The agent learns steady, increasing its mean score to a maximum of 1418. It is observable that up to episode 200 the agents constantly achieves a mean score about -500 points, afterwards the mean score scatters in a positive direction, from this constantly getting better observing the regression curve. Following the Regression curve it shows that a maximum in learning behavior is reached about episode 950, though it is unclear how much improvement there would be possible with further learning. Nevertheless a stable learning behavior is achieved, which was the goal of this series of experiments of section 6.3. But wanting to know if the best agent of $\eta = 10^{-4}$ is performing better as the formerly best performing agent of $\eta = 10^{-3}$ with unstable learning behavior, each of these agents is playing 200 fixed games on the training game-grid, emerging in the results presented in table 6.2.

Table 6.2: Results of 200 fixed games with the best network of Adam agent with $\eta = 10^{-3}$ and the best network of agent$_2$ with $\eta = 10^{-4}$

Network	Win ratio	Avg. score	Avg. food eaten/ game	Avg. ghosts eaten/ game
Adam$_{79}$	81%	1261	93%	1.300
agent$_{2_{961}}$	89%	1418	95%	1.310

Looking at table 6.2 it is now clear that not just the learning behavior is stabilized but also the quality of the agent has improved. $agent_2$ with $\eta = 10^{-4}$ of episode 961 has a higher win ratio and higher average score than the former best agent with $\eta = 10^{-3}$.
At this point it can be confirmed that an agent with an Adam trainer with ReLUs in his hidden layer and a step size $\eta = 10^{-4}$ is advisable for such an experiment like this one.

6.4 Knowledge Transition

In chapter 1 the assumption was made, that since features are a high level abstraction of the state a transition of the learned to other game-grids is possible. Hence, proving this assumption the best neural agent from the last section 6.3.4 is chosen and playing 200 games on the contest game-grid (observable in sketch 6.9) given by the framework.

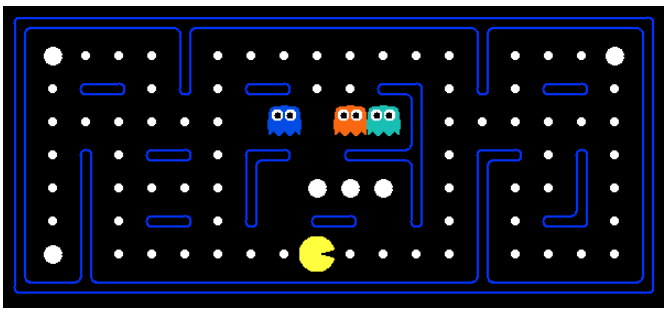


Figure 6.9: Sketch of the contest game-grid

Comparing the contest game-grid to the training grid the training grid has a size of $11*20 = 220$ fields while the contest grid consists of $9 * 20 = 180$ fields. So the training game-grid is about 22% bigger. The training game-grid contains 97 portions of food, while the contest grid only has 69. The training game-grid holds 41% more food, intuitively one would think that might lead to higher scores in the training game-grid, but since there are 3 times more Powerpills in the contest game-grid and therefore more potential scared ghosts, the agent could score higher through eating ghosts. But the difficulty of the contest grid is increased, as mentioned the number of ghosts is upped to 3. Further more it holds potential danger due to the dead end

in the lower left corner if not eating the Powerpill. And though it has more Powerpills than the training game-grid, looking at the features in chapter 4, the agent can't measure the distance to the next Powerpill, thusly not fully benefiting of more Powerpills. Therefore, estimating the out-come of the transition to the contest game-grid isn't obvious, though one might guess due the increase of difficulty the score might be less than before.

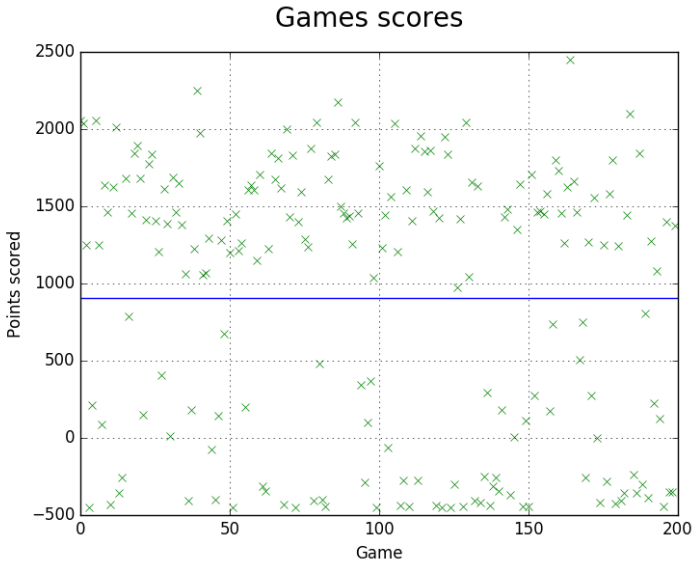


Figure 6.10: Sketch of the 200 fixed games on the contest grid

Looking at figure 6.10 the agent achieves mixed results, though being mostly positive it doesn't score as high as on the training grid. But this is not surprising since it is expected to perform not as good as on the training grid trained on.
Table 6.3.4 is summarizing the key data gathered. Comparing this data to the results from table 6.3 the win ratio on the training game-grid is 29% higher, the avg. score is also 56% higher on the training game-grid, the avg. food eaten/ game differs by 19%, but the avg. ghosts eaten/ game are about the same.

Table 6.3: Results of 200 fixed games with the best network of section 6.3.4 on the contest game-grid

Network	**Win ratio**	**Avg. score**	**Avg. food eaten/ game**	**Avg. ghosts eaten/ game**
agent_{2961}	60%	909	76%	1.305

Surprisingly the increase of Powerpills doesn't lead to more eaten ghosts, there is even a slight decrease by 0.5%. This might not be much, but since expecting the agent to eat more

ghosts on the contest game-grid due the increase of Powerpills this seems odd at first. But remembering that there is no feature in respect to the Powerpill distance and looking at the training game-grid's layout, where Powerpills just lay in the shortest Path between two foods and comparing it to the contest game-grid with exactly the same amount of Powerpills in a similar constellation, it seems only these Powerpills get eaten, this would indicate why there is no increase in eaten ghosts. Number-wise the key difference is in the win ratio and avg. food eaten/ game, roughly wining about 60% the agent is performing decently with 76% avg. food eaten/ game. The decrease of wins strengthens the assumption that the difficulty in the contest game-grid is increased for an agent with no ability to detect Powerpills.
In an effort to improve the benefit from the Powerpill hence the agent performance, a new feature is introduced in the next experiment of section 6.5.

6.5 Powerpill-Distance Feature

Since Pacman isn't able to evaluate the distance to the next Powerpill, this experiment introduces a feature in respect to the Powerpill-distance, in order to take more benefit from the Powerpills on the game-grid.

6.5.1 The Feature Equation

In order to detect the closest Powerpill and evaluate its distance the feature is dependent on:

- $a =$ Maximum path length
- $b =$ Shortest path length to closest Powerpill [6]

The feature $f_{Powerpill-distance}$ gets computed as follows:

$$f_{Powerpill-distance} = (a - b)/a \tag{6.2}$$

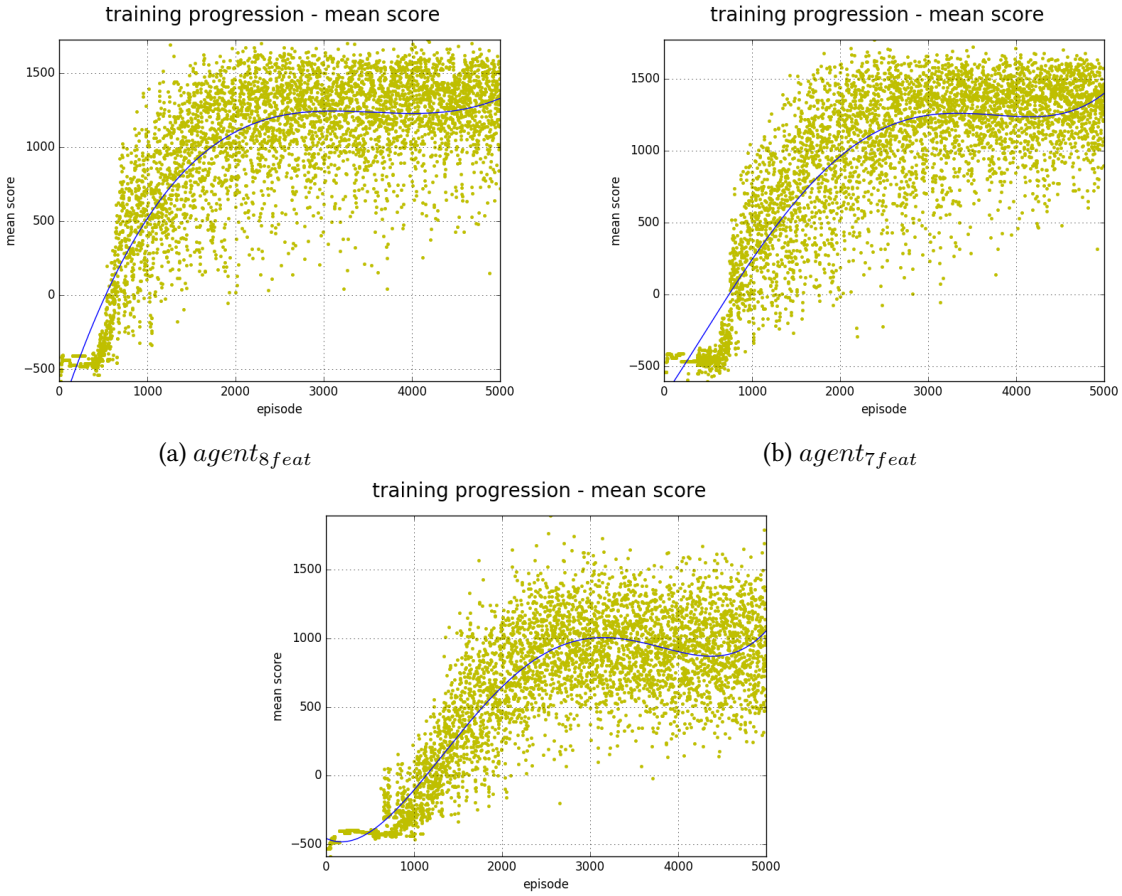
6.5.2 Training The "8 Feature" Agent

After adding the new feature, three different agents are trained with the parameters acquired in section 6.3 for 5000 episodes as referred in section 6.3.1. The number of training-episodes is increased by a number seemed fitting, since the regression of $\vec{feature}$ to reward is expected to be more complex with the added new feature. To estimate the benefits of the new feature the three agents mentioned are:

[6] The closest Powerpill is found by a breadth-first search.

- $agent_{8feat}$ an 8 feature agent, trained on the standard training game-grid
- $agent_{7feat}$ a 7 feature agent, with the best performing configuration of section 6.3, trained on the training game-grid
- $agent_{8alt}$ an 8 feature agent, trained on the contest game-grid

$agent_{8feat}$ and $agent_{7feat}$ are trained for a direct comparison of performance between the 8 feature and the 7 feature variant of the agent.
$agent_{8alt}$ is added, since the contest game-grid contains more Powerpills, which might lead to more eaten ghosts therefore directly taking more benefit of the Powerpill hence the new feature.
From this the training behaviors visible in figure 6.11 are observed.



(a) $agent_{8feat}$

(b) $agent_{7feat}$

(c) $agent_{8alt}$

Figure 6.11: Training progression of $agent_{8feat}$, $agent_{7feat}$ & $agent_{8alt}$, taking the average score of 10 fixed games after each episode and plotting a regression curve above them.

Looking at figure 6.11 the first observation is that even with a new feature or a different game-grid the configuration acquired in section 6.3 holds up for a stable learning behavior with all three agents. But by direct comparison the learning curve of the $agent_{8feat}$ is steeper than its competitors'. All agents learn steady, while the $agent_{8feat}$ seams to plateau after 2600 episodes and the $agent_{7feat}$ seams to plateau after 3100 episodes. It's only $agent_{8alt}$ that seems to get worse in performance after episode 3100 for a while, but since it is trained on the harder contest game-grid it's a special case. It's also noticeable that the mean score reached for agent $agent_{8feat}$ and $agent_{7feat}$ is averagely higher than for $agent_{8alt}$. This happening despite the fact that the potential reachable score of both game-grids is similar, due to similar the amount of food balanced by the amount of Powerpills. Remembering that $agent_{8feat}$ and $agent_{8alt}$ are learning on the same configurations except the game-grid, it indicates that the contest game-grid has a higher difficulty than the training game-grid. Nonetheless the best performing $agent_{8feat}$ occurs in episode 4177, the best $agent_{7feat}$ in episode 2390 and the best performing agent of $agent_{8alt}$ in episode 2549. These are used for further comparison.

6.5.3 Comparing The "8 Feature" Agent

To compare the actual performance of the trained agents and the new feature, the best performing agents and their corresponding last episode agents of section 6.5.2 are chosen to play 200 fixed games on the standard game-grid and again 200 fixed games on the contest game-grid. The results of these games are summed up in table 6.4 and table 6.5.

Table 6.4: Results of 200 fixed games with the best $agent_{8feat}$, $agent_{7feat}$ & $agent_{8alt}$ and the last episode agents on the training game-grid

Agent	Win ratio	Avg. score	Avg. food eaten/game	Avg. ghosts eaten/game
$agent_{8feat_{4177}}$	**93%**	**1439**	**96%**	1.075
$agent_{8feat_{5000}}$	90%	1395	95%	1.055
$agent_{7feat_{2390}}$	81%	1286	93%	1.120
$agent_{7feat_{5000}}$	81%	1266	92%	1.150
$agent_{8alt_{2549}}$	41%	739	87%	**1.295**
$agent_{8alt_{5000}}$	65%	997	92%	1.070

Table 6.5: Results of 200 fixed games with the best $agent_{8feat}$, $agent_{7feat}$ & $agent_{8alt}$ and the last episode agents on the contest game-grid

Agent	Win ratio	Avg. score	Avg. food eaten/game	Avg. ghosts eaten/game
$agent_{8feat_{4177}}$	68%	1036	83%	1.225
$agent_{8feat_{5000}}$	**77%**	**1173**	88%	1.365
$agent_{7feat_{2390}}$	72%	1138	85%	1.400
$agent_{7feat_{5000}}$	75%	1152	85%	1.330
$agent_{8alt_{2549}}$	53%	957	79%	1.590
$agent_{8alt_{5000}}$	65%	1168	**89%**	**1.855**

Starting with table 6.4, all best agents trained on the training game-grid are roughly scoring the same or slightly better as the corresponding agent of the last episode. The exception being $agent_{8alt}$, the last episode agent $agent_{8alt_{5000}}$ wins 24% more games than the deemed best agent $agent_{8alt_{2549}}$ on the training game-grid. Furthermore the agents with the additional new feature trained on the training game-grid score higher than their 7 feature competitors, even higher than $agent_{2_{961}}$ of section 6.3.4, which scores higher than the 7 feature agents of this section despite using the same training configuration. But in exchange the 7 feature configurations eat more ghosts per game than the 8 feature agents, still keeping in mind that the standard training game-grid contains only 2 Powerpills, therefore the new feature might be less beneficial than it could be. Hence the $agent_{8alt}$ is trained, the $agent_{8alt_{2549}}$ is averagely eating the highest amount of ghosts of all agents but also scoring the worst and $agent_{8alt_{5000}}$ is roughly eating as much ghosts as the other better performing 8 feature variants.
So it is kept unclear in which way the new feature is bringing extra beneficiality to the agents performance on the training game-grid.
Continuing with table 6.5, all last training episode agents are outperforming their formally deemed best corresponding agent. This means that though performing slightly worse on the training game-grid they better translate to other game-grids. Again the best agent is a variant with the new feature, namely agent $agent_{8feat_{5000}}$, and again the best 8 feature agent trained on the training game-grid isn't significantly eating a higher amount of ghosts than its 7 feature competitors. But looking at the 8 feature variants trained on the contest gaming-grid, the $agent_{8alt_{5000}}$ is eating 36% more ghosts per game than the best performing agent $agent_{8feat_{5000}}$. This indicates that the newly added feature leads to more ghosts eaten per game on a game-grid with more Powerpills. Comparing $agent_{8alt_{2549}}$ with $agent_{8alt_{5000}}$ another problem arises as $agent_{8alt_{2549}}$ is believed to perform best on the contest gaming-grid in the $agent_{8alt}$ training session, but looking at table 6.4 $agent_{8alt_{5000}}$ wins 12% more games than $agent_{8alt_{2549}}$. This comes from an uncertainty in the metric of section 6.1 used

for all experiments, since the 10 fixed games are chosen arbitrarily, working well for agents training on the training game-grid, but not for those training on the contest game-grid. For example taking just 10 fixed games with $agent_{8alt_{2549}}$ results in an avg. score of 1890.8, but for $agent_{8alt_{5000}}$ the avg. score for the same games is 850.6.
Hence, a higher number in evaluation games should be chosen for future experiments. But not too high, since with the increase of evaluation games the amount of training time increases proportionally.

6.5.4 Retraining The "8 Feature" Agent On The Contest Game-Grid With More Evaluation Games

Though it seems that the new feature improves the overall performance of the agent, the fact that the expected best $agent_{8alt}$ underperforms the last episode's agent is unsatisfactory. So, in an attempt to adjust this irregularity, the number of evaluation games at the end of each training episode is increased up to 30 games. After this the following training behavior is gained as shown in table 6.12.

Figure 6.12: Training progression of $agent_{8alt}$ taking the average score of 30 fixed games after each episode and plotting a regression curve above them.

Comparing figure 6.12 with figure 6.11c the agent with increased evaluation games shows similar learning behavior as the $agent_{8alt}$, yet the adjusted $agent_{8alt}$ scores higher overall.

Again the best agent and the last episode agent of the adjusted $agent_{8alt}$ are playing 200 fixed games on both game-grids. The results get summarized in the tables 6.6 and 6.6.

Table 6.6: Results of 200 fixed games with the best $agent_{8alt}$ and the last episode agent on the training game-grid

Agent	Win ratio	Avg. score	Avg. food eaten/game	Avg. ghosts eaten/game
$agent_{8alt_{3226}}$	71%	1131	93%	1.280
$agent_{8alt_{5000}}$	80%	1226	94%	1.170

Table 6.7: Results of 200 fixed games with the best $agent_{8alt}$ and the last episode agent on the contest game-grid

Agent	Win ratio	Avg. score	Avg. food eaten/game	Avg. ghosts eaten/game
$agent_{8alt_{3226}}$	76%	1195	86%	1.500
$agent_{8alt_{5000}}$	71%	1068	80%	1.285

Observing tables 6.6 and 6.6 the agents now show similar behavior as the agents $agent_{8feat}$ and $agent_{7feat}$. This means that the deemed best performing agent is actually performing better on the trained upon game-grid than the agent of the last episode, since $agent_{8alt_{3226}}$ wins 5% more games and scores about 12% more points than $agent_{8alt_{5000}}$ on the contest game-grid. Also the agent of the last episode translates better to a different game-grid, as $agent_{8alt_{5000}}$ wins 9% more games and scores about 12% more points than $agent_{8alt_{3226}}$ on the contest game-grid. In addition, the current $agent_{8alt}$ scores lower on the contest game-grid it is trained on, than on the training game-grid it is transferred to. Yet $agent_{8feat}$ and $agent_{7feat}$ which are trained on the training game-grid are scoring expectedly higher on the training game-grid than on the contest game-grid. This is a strong indicator for the increased difficulty of the contest game-grid.

An unexpected result is that by comparing the $agent_{8alt}$s in table 6.7 and table 6.5 the amount of ghosts eaten decreases with the further training in the adjusted $agent_{8alt}$ unlike the former $agent_{8alt}$ of section 6.5.3. Still, overall the performance is increased with the raise of evaluation games. Comparing $agent_{8feat_{5000}}$ with the current $agent_{8alt_{3226}}$ the $agent_{8feat_{5000}}$ wins 1% more games than its competitor, but the agents are setup to improve the reward hence the game-score and $agent_{8alt_{3226}}$ scores about 2,3% more points per game than $agent_{8feat_{5000}}$. Nonetheless, the differences are minor and considering that training the current $agent_{8alt}$ takes significantly more time and performs worse on the training game-grid, the effort isn't really worthwhile.

Therefore, one could make the assumption that an agent that is trained on a more general

game-grid holds more overall benefits than an agent trained on a more difficult game-grid. But this again raises the question how to judge the difficulty of a game-grid, which is something that needs itself some kind of expert knowledge. Since previously in this series of experiments indeed the assumption is made that the contest game-grid is more difficult than the training game-grid, indicators for the correctness of this assumption are gained after the experiments.

7 Conclusion

For this thesis I studied the needed knowledge in reinforcement learning and deep learning, in order to approach artificial neural networks as an alternative for classic regression algorithms while handling features. I did implement the features of paper Luuk Bom und Wiering (2013) in the architecture presented. By doing this modular, single components were interchangeable for different experiments. In this instance these components were linear function approximation and deep learning modules. Though the second approach was pursued mostly, since the focus laid on deep learning as an alternative for classic regression algorithms.
First linear function approximation agents stagnated pretty fast in learning and performed poorly, therefore got dismissed in the early stages. This indicates that linear function approximation wasn't refined enough for this particular task. I used a simple feed forward network, mostly utilized for regression and classification tasks, to estimate the beneficiality of possible feature combinations for estimating which action given the current state might be the best. The first attempted artificial network agent, trained on data sets of feature combinations mapped to received rewards, averagely outscored the best linear function approximation agent's score by triple the points.
Sure there were more refined regression methods than linear function approximation, but these would have also needed more expert knowledge to implement. Artificial neural networks or better their libraries were like a toolbox, one could have potentially built a network without the premiss of knowing every detail. This was giving more options for adjustments in the current network design, without the need to interchange or learn a complete new method as with the regression approach. The challenge in using such networks laid with understanding enough or having enough experience to determine the network-size, choosing the right components and finding stable hyper parameters (like the step-size), etc. In this instance at the beginning of the experiments no experience was preexisting, so the understanding gained from different papers had to be tested and verified with the different agents built and trained.
Looking back at the experiments, it becomes clear that artificial neural networks offer a good alternative performance-wise. Since the best agent was able to reach an 89% win-ratio on the given training game-grid. After establishing artificial neural networks as a good alternative

to classic regression, the assumption that training on features makes the learned translatable to different game-grids is proven. The best agent trained on the training game-grid still won around 60% of the games on the more difficult contest game-grid. But while doing so I noticed that the features presented in Luuk Bom und Wiering (2013) were missing a feature in respect to the Powerpill distance as the contest game-grid held three times the Powerpills than the training game-grid. Pacman was already able to evaluate the elapsed time of the scared ghost duration after eating a Powerpill and the distance to a scared ghost, but not its distance to the next Powerpill. After implementing this new feature a new best agent improved its win-ratio to 93% on the training game-grid and 68% on the contest game-grid. This shows that finding an appropriate abstraction of the state space isn't easy and that there might also be more room for improvement. Another observation was that by taking a later iteration of the best agent with more training on the training game-grid (but with slightly worse performance on the same game-grid) and letting it play on the contest-game grid, led to out-scoring the deemed best agent significantly. This was interesting, since the agents seemed to stagnate in learning as the score didn't improve in later training episodes with further training, but indicated that some kind of learning seemed to be going on as agents with more training translated better to other game-grids. This happening, despite the fact that the only learning goal of the agents was to improve the gained rewards, was unexpected. Thusly again hinting at the complexity of Pacman and that understanding the math or principles behind the back-prop. algorithms wasn't enough to comprehend the actual learning of the agent.

Although positive results with artificial neural networks were obtained, there was a practical downside to them as they differed in sizes and components. Their sheer amount of different components and size made them almost impossible to debug. If the network behaved oddly or wouldn't learn properly, all one can do was observe the behavior and make assumptions to figure out what was wrong. This trial and error approach depending on the complexity of the task learned took a lot of time. For this series of experiments the time needed for one training session differed from one day to almost a week depending on the amount of episodes.

Still, with the knowledge gained it will be possible to build an even more complex network architecture, which can potentially craft its own features. This will make it possible to train an agent from raw byte or image input data, while feeding these newly gained features into a network structure acquired in this thesis. Therefore, having an architecture that could learn all kind of Pacman-esque games, just by exchanging the environment.

For this reason I would promote this approach under the idea of pursuing a more advanced network architecture in the future, using the knowledge gained in this thesis.

Bibliography

[Abadi u. a. 2015] Abadi, Martín ; Agarwal, Ashish ; Barham, Paul ; Brevdo, Eugene ; Chen, Zhifeng ; Citro, Craig ; Corrado, Greg S. ; Davis, Andy ; Dean, Jeffrey ; Devin, Matthieu ; Ghemawat, Sanjay ; Goodfellow, Ian ; Harp, Andrew ; Irving, Geoffrey ; Isard, Michael ; Jia, Yangqing ; Jozefowicz, Rafal ; Kaiser, Lukasz ; Kudlur, Manjunath ; Levenberg, Josh ; Mané, Dan ; Monga, Rajat ; Moore, Sherry ; Murray, Derek ; Olah, Chris ; Schuster, Mike ; Shlens, Jonathon ; Steiner, Benoit ; Sutskever, Ilya ; Talwar, Kunal ; Tucker, Paul ; Vanhoucke, Vincent ; Vasudevan, Vijay ; Viégas, Fernanda ; Vinyals, Oriol ; Warden, Pete ; Wattenberg, Martin ; Wicke, Martin ; Yu, Yuan ; Zheng, Xiaoqiang: *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. – URL http://tensorflow.org/. – Software available from tensorflow.org

[Chollet 2015] Chollet, François: *keras*. https://github.com/fchollet/keras. 2015

[Francisco S. Melo 2007] Francisco S. Melo, M. Isabel R.: *Q-learning with linear function approximation*. 2007

[Geramifard u. a. 2013] Geramifard, Alborz ; Walsh, Thomas J. ; Tellex, Stefanie: *A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning*. Hanover, MA, USA : Now Publishers Inc., 2013. – ISBN 1601987609, 9781601987600

[Glorot u. a. 2011] Glorot, Xavier ; Bordes, Antoine ; Bengio, Yoshua: Deep Sparse Rectifier Neural Networks. In: Gordon, Geoffrey J. (Hrsg.) ; Dunson, David B. (Hrsg.): *Proceedings of the Fourteenth International Conference on Artificial Intelligence and Statistics (AISTATS-11)* Bd. 15, Journal of Machine Learning Research - Workshop and Conference Proceedings, 2011, S. 315–323. – URL http://www.jmlr.org/proceedings/papers/v15/glorot11a/glorot11a.pdf

[Goodfellow u. a. 2016] Goodfellow, Ian ; Bengio, Yoshua ; Courville, Aaron: *Deep Learning*. MIT Press, 2016. – http://www.deeplearningbook.org

[Kingma und Ba 2014] Kingma, Diederik P. ; Ba, Jimmy: Adam: A Method for Stochastic Optimization. In: *CoRR* abs/1412.6980 (2014). – URL http://arxiv.org/abs/1412.6980

[Lample und Chaplot 2016] Lample, Guillaume ; Chaplot, Devendra S.: Playing FPS Games with Deep Reinforcement Learning. (2016). – URL http://arxiv.org/abs/1609.05521

[Luuk Bom und Wiering 2013] Luuk Bom, Ruud H. ; Wiering, Marco: *Reinforcement Learning to Train Ms. Pac-Man Using Higher-order Action-relative Inputs*. 2013

[Mnih u. a. 2013] Mnih, Volodymyr ; Kavukcuoglu, Koray ; Silver, David ; Graves, Alex ; Antonoglou, Ioannis ; Wierstra, Daan ; Riedmiller, Martin A.: Playing Atari with Deep Reinforcement Learning. In: *CoRR* abs/1312.5602 (2013). – URL http://arxiv.org/abs/1312.5602

[Nielsen 2015] Nielsen, Michael A.: *Neural Networks and Deep Learning*. Determination Press, 2015

[Sutton und Barto 1998] Sutton, Richard S. ; Barto, Andrew G.: *Introduction to Reinforcement Learning*. 1st. Cambridge, MA, USA : MIT Press, 1998. – ISBN 0262193981

[Theano Development Team 2016] Theano Development Team: Theano: A Python framework for fast computation of mathematical expressions. In: *arXiv e-prints* abs/1605.02688 (2016), Mai. – URL http://arxiv.org/abs/1605.02688

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe.

Hamburg, Roland Meo