Ruprecht-Karls-Universität Heidelberg
Institute of Computer Science
Winter Semester 2017/18
Seminar: Artificial Intelligence
Lecturer: Prof. Dr. Björn Ommer

Report

Pacman - Capture the Flag

Name:	Jan Sieber
Matriculation number:	3219317
Course of studies:	Applied Computer Science(9. semester)
Email:	uni@email-master.de
Date of submission:	04.07.2017

Declaration of Authorship

I, **Jan Sieber** hereby confirm that the report I am submitting on **Pacman - Capture the Flag**, in the course **Artificial Intelligence**, is solely my own work. If any text passages or images from books, papers, the Web or other sources have been copied or in any other way used, all references have been acknowledged and fully cited. I declare that no part of the report submitted has been used for any other report, thesis or course in another higher education institution, research institution or educational institution. I am aware of the University's regulations concerning plagiarism, including those regulations concerning disciplinary actions that may result from plagiarism.

Heidelberg, 23. Februar 2018

Preamble

Due to I put some bigger images into this report and added a whole section on how to start our programs with different grids, I exceeded the report page limit of 10 pages a bit. If one drops the section „Instructions how to start code“, it would be eleven pages. The one page would be the images.
All parts were written by me. I submitted the code for the approximate q-learning agent for classic Pacman too, which is written by my partner. He also implemented a DQN, but it is submitted here. At I have no instructions how to start his DQN.

Contents

1 Introduction

Going away from static programming, one can make programs learn. In this report/project, we try to apply machine learning to playing Pacman. In general there are three types of machine learning:

- Supervised learning
- Unsupervised learning
- Reinforcement learning

Supervised learning needs prelabeled data, while unsupervised learning does not. Reinforcement learning is a trade-off between the expensive labeling of data and not having any labeled data. In reinforcement learning data is labeled while it is generated. In example the label can be the score of a specific move in a game. One could say, that reinforcement learning is generating data „on the fly“. Simultaneaously to generating data rules are learned by the algorithm, which are applied to play the game later on. This property (generating data while playing) makes it perfect to use reinforcement learning for making a program that learns to play Pacman and many other games.

2 Reinforcement learning

There are three major derivatives of reinforcement algorithms used, to learn Pacman.

- Q-learning
- Approximate Q-Learning
- Deep Q-Networks

While approximate q-learning has common parts with q-learning and deep q-networks, q-learning and deep q-networks differ completely. Still, all of those three algorithms have one thing in common. They try to find a policy to always make the best move in a given state. This is done by randomly exploring the playing field by a given factor $\epsilon \in [0, 1]$. These random explorations help the algorithm to learn, because it will choose actions that may not be the best move at the moment and update its variables properly. After one trained the algorithm, ϵ will be 0 in testing phase. This is a typical ϵ greedy approach. With this thing in mind one should read the next chapters.

2.1 Q-Learning

In q-learning one has a typical „Markow Decision Problem“ (MDP). One has

- a set of states S
- a set of actions A
- a reward r for every transition from state s to state s' by action a.

The solution of a MDP is to find a policy $\pi : S \Rightarrow A$, which gives an optimal action $a \in A$ for every state $s \in S$. Now one just needs to approximate a good policy π. This is done by encoding a state action pair into a number by the Q-function

$$Q : S \times A \Rightarrow R$$

, the so called q-value. [8] By choosing the action a with the highest q-value in state s, one can gain the policy π. More mathematically this can be expressed as

$$\max_a Q(s, a)$$

where $s \in S$ and $a \in A$. Now one just needs to know how to gain these q-values. This can be achieved by the applying the simple update rule

$$Q(s_t, a_t) = (1 - \alpha) \cdot Q(s_t, a_t) + \alpha(r_t + \gamma \cdot \max_a Q(s_{t+1}, a))$$

where $\alpha \in [0, 1]$ is the learning rate and $\gamma \in [0, 1]$ is the discount factor for future q-values. s_{t+1} is the next state to s_t. Now the q-values can be updated by playing the game, because the game can give us the actual q-values $Q(s_t, a_t)$ the reward r and the most promising future q-value $Q(s_{t+1}, a)$ If one gets a negative reward, the updated q-value will shrink. In opposite, it will rise.[1] It is an important question how one should initialize these q-values. Values over the highest reward will rise the exploration rate of the algorithm, because others not yet taken actions are better than the highest reward. The lower this initial value gets to the lowest reward, the exploration rate will also shrink.

2.2 Approximate Q-Learning

Approximate q-learning works a little different from q-learning. While q-learning is taking the raw (state, action)-pairs, approximate q-learning takes handcrafted features $f_i(s, a)$ extracted from the (s,a)-pair and uses a single neuron to predict our q-values by

$$\sum_{i=1}^{n} f_i(s, a) w_i$$

, where w_i is the weight of the single feature $f_i(s, a)$. Instead of manipulating the q-values directly like in q-learning, we now have to manipulate the weights w_i of the neural network. This is done by the update rule

$$w_i = w_i + \alpha \cdot diff \cdot f_i(s, a)$$
$$diff = (r + \gamma \cdot \max_{a'} Q(s', a')) - (Q(s, a))$$

[9]

2.3 Deep Q-Network

After one already used neural networks in approximate q-learning, one can go one step further and use kind of a convolutional neural network, the „deep q-network“ (=DQN). This has the advantage that one does not need to handcraft features. One just needs to forward pass the actual state through the neural network, compute the error and adapt the weights with backpropagation. Still there are some subtleties to explain.

2.3.1 Architecture

The architecture of a DQN follows the classic CNN architecture, with the difference that there are no pooling layers. This seems logical, since pooling layers often destroy positional information. Pooling layers mathematically blur the image data (=loss of image data), which is not good for playing Pacman or games in general, since positional data is very important in many games.[4]

2.3.2 Replay memory

In addition to the adapted architecture, one needs to adapt, how the network learns. Instead of learning directly on each generated sample, one learns on the replay memory. The replay memory saves the most recent state, action taken in a state, the next state and reward data (for each move made). This is all we need to make the program learn later on. Later, we learn on a randomly sampled subset of this replay memory. The replay memory has a specific size. If there is a size overflow, the oldest data will be replaced. There are optimizations for this replacement strategy, which will be presented later.[6][4]

2.3.3 Backpropagation

To make the DQN work, one needs the loss for the neural network, to back-propagate the error and optimize the weights of our network.

$$loss = (r + \gamma \max_{a'} Q^*(s', a') - Q(s, a))^2$$

where s is the actual state, r is the reward, s' is the future state resulting out of state s and action a and γ is the discount factor. As one can see $r + \gamma \max_{a'} Q^*(s', a')$ is the value, which we want to achieve (=target value), while $Q(s, a)$ is only our predicted value. Both values $Q^*(s', a')$ and $Q(s, a)$ are calculated by our DQN.[5]

2.3.4 Two networks

Now that we know how to compute the loss of our network, one needs to know that there is another improvement. This is also the reason, why there is a different notation for the predicting and target network in the previous chapter even if they are both the same network. Often one has the problem, that the DQN network is overestimating some q-values. This can be seen like overfitting an outlier/focus on a local minimum. To tackle this problem, one can use two deep q-networks. One „target“ and one „prediction“ network. As the names already state, the target network is for the target $Q^*(s, a)$ values in our loss and the prediction network for the $Q(s, a)$ values. The prediction network is the network to be optimized. After a certain period of time the prediction network will be copied into the target network, to update our

more stable q-values. This can avoid local minima. [2]

2.3.5 Exploration and learning rate decay

Additionally to the double networks method, one should implement an exploration rate decay. While in the beginning one wants to make totally random movements, in the end one wants to explore further away states. To get there little faster and more often, there is a decay of the exploration rate. Additionally, the learning rate decay can be important too, because this can prevent too fast learning in the later phase of learning or too slow in the early phase. With decay of learning rate, one can improve the learning speed significantly. For general decay one can apply the formula

$$\beta_{t+1} = \beta_t \cdot b$$

where β is the learning or exploration rate and $b \in [0, 1]$ a decay constant.[2]

2.4 Further DQN improvements

Even if the above DQN network does work well already, one can further boost the performance with the following methods. Until now, all methods and improvements were implemented. The following two are not.

2.4.1 Prioritized experience/memory replay

Instead of just replacing the oldest data from the replay memory, one can replace the data with the most surprising outcome for the neural network. The replay memory is there to „generate“ some training data by learning it multiple times. Due to that it is beneficial to learn on the most surprising data outcome, because the others are already well predicted by the neural network. [7][3] For example one can take the loss as scoring function. In addition to that one could introduce a time to live for each sample. Like that, one can prevent overfitting on special states.

2.4.2 Multi-step learning

To increase the learning speed and look some steps ahead, one can implement multi-step learning. As the name already states, multi-step learning only

looks some iterations into the future, accumulates the reward and takes this as the actual target reward. The formula for the accumulation looks like

$$r_t = \sum_{k=0}^{n-1} \gamma_t^k r_{t+k+1}$$

where r_x is the reward at time x and γ is the discount factor. [3]

3 Game and Framework

The framework comes from the „UC Berkeley CS188 Intro to AI"-course [9]. It gives the ability to start very quickly and to fine tune rewards like time penalties. There are two different Environments used. Due to that, some things were double implement.

- Classic Pacman [9]
- Pacman - Capture the Flag [10]

In both environments one can create a map and load it.

3.1 Classic Pacman

Everyone should know the classic Pacman.

- Pacman needs to eat dots and gets a reward for this.
- Once all dots are eaten Pacman wins.
- Ghosts try to eat Pacman, what makes Pacman lose.
- If Pacman eats a big yellow dot the ghosts get scared and can be eaten by Pacman. The eaten ghost will respawn in the mid-box.

The environment for classic Pacman already had a full implementation of Pacman and an additional training environment for q-learning and approximate q-learning. One just needed to implement the algorithms and make them learn. Additionally, one could change the rewards and penalties with some search in the code. The DQN solution needed to be implemented manually by inheriting from the appropriate class and defining the appropriate functions. [9]

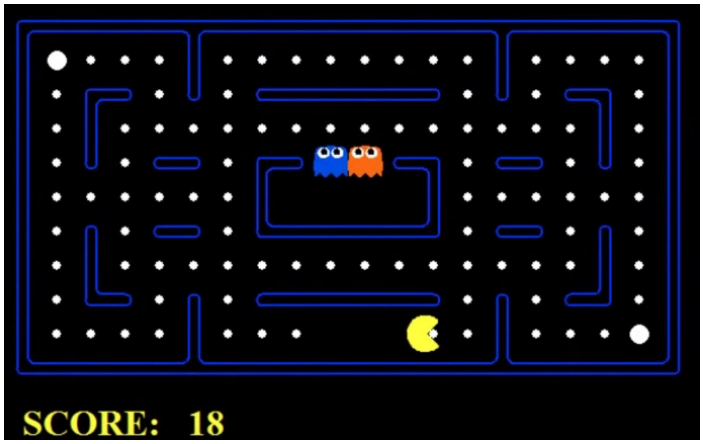

Figure 1: Classic Pacman on a modified grid.

3.2 Pacman - Capture the Flag

„Pacman - Capture the Flag“ differs a bit from the classic Pacman. In this game mode the field is biparted and each side has two ghosts. One is a defensive, the other an offensive ghost. The offensive ghost is able to go to the enemies half of the playing field. Passing the middle border will transform the offensive ghost into a Pacman. The goal of the offensive ghost is to eat dots from the enemies side and return them to the own side. If the offensive ghost/Pacman dies, he loses all dots eaten. The defensive ghost only can move at the own half and its goal is to defend the dots from the enemie Pacman. The Pacman - CTF environment only implements the game mechanics and

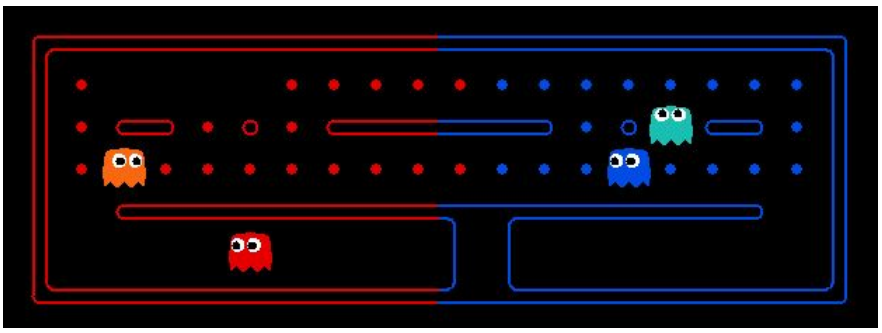

Figure 2: Pacman - Capture the flag on a modified grid.

rewards, which can be modified. One needs to implement the training and

everything else.[10]

4 Experiments on Pacman

4.1 Q-Learning

As one saw from the previous sections, the q-learning rule is very simple. Due to that, there was not much room for experiments on the classic Pacman game. The q-learning algorithm achieves the following results with the following setting

Grid	ER	LR	Training games	Average Score	Winrate
SmallGrid	0.05	0.2	1000	-24168.2	0.52
SmallGrid	0.05	0.2	2000	406.4	1.0
MediumGrid	0.05	0.2	1000	-49519.1	0.0
MediumGrid	0.05	0.2	2000	-49510.3	0.0
MediumGrid	0.05	0.2	12000	-4878.7	0.85
MediumGrid	0.05	0.2	32000	1800.3	0.98
MediumClassic	0.05	0.2	32000	-32567.4	0.0

As one can see, mostly it was trained on exploration rate=0.05 and learning rate=0.2. That is, because these were the best parameters in our runs. One can also see, that the smallGrid wins 100% of the games after 2000 training iterations what is not that much. The smallGrid is a 6x7 playing field. One can explain the fast learning with the small field. While there are good results on the smallGrid, the mediumGrid needed to be trained much longer, while the MediumClassic did not learn that much after 32000 episodes. This comes from the fact, that q-learning tries to remember state action pairs and its q-values. If q-learning has not seen a state action pair yet, it does not know how to react. As one may remember, this will get better with the approximate q-learning algorithm.

4.2 Approximate Q-Learning

While pure q-learning takes pure state action pairs, approximate q-learning extracts the features and encodes the state. We used following features for different extractors:

- SimpleExtractor uses features

  - Number of ghosts one field away after action
  - Input if he ate food with action
  - Distance to next food

- FeatureExtractor uses features
  - Ignore scared ghosts one field away after action

- AdvancedFeatureExtractor uses features
  - Distance to scared ghosts
  - Input if he ate scared ghost

SimpleExtractor					FeatureExtractor					AdvancedFeatureExtractor				
Iteration	Avg. Score	Won games	Avg. Score	Win % all	Iteration	Avg. Score	Won games	Avg. Score	Win % all	Iteration	Avg. Score	Won games	Avg. Score	Win % all
1	642,65	52	635,312	50,70%	1	920,86	62	845,25	57,40%	1	853,39	60	836,448	56,10%
2	683,71	54			2	799,66	54			2	966,41	64		
3	687,7	54			3	806,89	56			3	855,61	58		
4	556,66	44			4	818,87	57			4	913,25	61		
5	626,19	50			5	825,25	57			5	865,41	58		
6	613,56	50			6	813,71	57			6	830,88	56		
7	663,3	53			7	768,87	53			7	685,55	45		
8	648,44	52			8	864,41	56			8	690,16	46		
9	614,36	49			9	979,12	67			9	796,89	53		
10	616,55	49			10	854,86	55			10	906,93	60		

Figure 3: Results for multiple runs on each feature extractor. Each iteration is 800 training games and 100 test games. The Avg. Score and Win % all is the average over the 10 iterations. Data created by Johannes Daub.

4.3 Deep Q-Network

First the smallGrid was tried. Due to it is very small, there was not much to do and it worked pretty fast.

Conv1	Conv2	Conv3	FC1	FC2	FC3
3x16 (3,2)	16x32 (3,1)			288x265	265x5

Table: Input channels x output channels (kernel size x stride)

with

- Learning rate of 0.0002 and a decay factor of 0.75 every 500th episode
- Exploration rate of 1.0 with decay to 0.1 and a decay factor of 0.8 every 200th episode

- Copy of the target and prediction network every 200th episode
- Replay memory size of 10000 with learning batch size of 32
- Trained 12 hours

In the following one can see a score-episode plot, which shows you the learning curve.

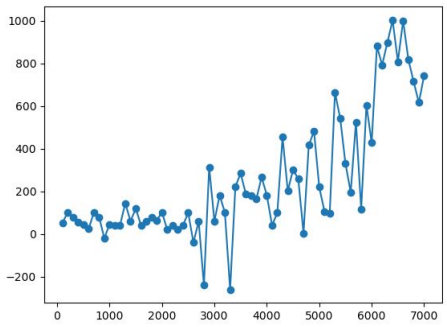

Figure 4: Score-episode plot with score in y-axis and episodes on x-axis.

Further more, several DQN architectures were tried on the mediumClassic with minimal changes.

Conv1	Conv2	Conv3	FC1	FC2	FC3
3x16 (3,1)	16x32 (3,1)	32x32 (3,1)	5440x512	512x256	256x5
3x16 (4,1)	16x32 (4,1)	32x32 (4,1)	1984x512	512x256	256x5
3x16 (4,1)	16x32 (4,1)	32x32 (4,1)		1984x256	256x5
changed rewards					
3x16 (3,2)	16x32 (3,1)			1632x512	512x5
3x16 (3,1)	16x32 (3,1)	32x64 (3,1)		10880x512	512x5
playing with learning rate and exploration rate decay					
3x16 (3,1)	16x32 (3,1)	32x64 (3,1)		10880x512	512x5

Table: Input channels x output channels (kernel size x stride)

A double image is taken as input generated from the last and actual state. Replay memory, two networks and decays are used here too. In the end, the following setup looked most promising:

- Learning rate of 0.002 with decay to 0.00002 and a decay factor of 0.9 every 5000th episode

- Exploration rate of 1.0 with decay to 0.1 and a decay factor of 0.9 every 2000th episode
- Copy of the target and prediction network every 200th episode
- Replay memory size of 100000 with learning batch size of 64
- Trained 18 hours

A more detailed timeline one can find in the git repository at the „training_mediumClassic“ branch. My partner also programmed another DQN.

5 Experiments on Pacman - Capture the Flag

5.1 Deep Q-Network

After one tried to make the DQN work on the classic Pacman and tried approximate q-learning on the CTF mode, one tried to make DQN work on the CTF mode. Due to we already knew the approximate parameters, there was not that much to experiment in this section. Here also a double image is taken as input generated from the last and actual state. Replay memory, two networks and decays are used here too. The CTF-contest was trained on the mediumCapture and tinyCapture grid. Both showed the same behavior. The trained (red) ghost, most of the time is waiting at the border to eat the enemie Pacman or get some points, while the enemie defensive ghost is not near the border. The mediumCapture was trained like the following.

- Learning rate of 0.0002 down to 0.00002 with a decay factor of 0.1 every 10000th episode
- Exploration rate of 1.0 with decay to 0.2 and a decay factor of 0.9 every 1000th episode
- Copy of the target and prediction network every 200th episode
- Replay memory size of 10000 with learning batch size of 128
- Trained 14 hours

6 Conclusion

Main objective for me/us was to get the DQN working on Pacman and Pacman - CTF. Due to I am new to reinforcement learning, I wanted to cover a broad spectrum of algorithms. That is why I tested algorithms on Pacman first and decided to make it work on Pacman - CTF. As one can see, it worked not that good but not that bad too. While Pacman - CTF is standing at the border, classic Pacman still dies and most of the time only moves if he is followed by a ghost. As mentioned, longer training and fine tuning of rewards will fix these problems.

7 Instructions to start the code

Q-learning on classic Pacman (uses python 3.5):

- Open terminal in folder „qlearning“
- Start with „sudo python3 pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid“
- It will start to train and play 10 test games on the smallGrid
- One also can use „sudo python3 pacman.py -p PacmanQAgent -x 32000 -n 32010 -l mediumGrid“ to train and play on the larger mediumGrid. WARNING: This will take MUCH longer.

Approximate q-learning on classic Pacman (uses python 2.7):

- Open terminal in folder „Approximate-classic“
- Execute the right command from 0command.txt. There is also another DQN from my partner.

DQN on classic Pacman on smallGrid (uses python 3.5):

- Open terminal in folder „DQN-classic-smallGrid“
- Start with „sudo python3 pacman.py -p PacmanDQN -x 0 -n 10000 -l smallGrid -a pacPretrained=saves/0toplay.pt“

DQN on classic Pacman on mediumClassic grid (uses python 3.5):

- Open terminal in folder „DQN-classic-mediumClassic“
- Start with „sudo python3 pacman.py -p PacmanDQN -x 0 -n 10 -l mediumClassic -a pacPretrained=saves/56900th-episode.pt“

DQN on Pacman-CTF on tinyCapture (red one is trained) (uses python 3.5):

- Open terminal in folder „DQN-CTF-tinygrid“
- Start with „sudo python3 capture.py -r baselineTeam3 -b baselineTeam -x 0 -n 10 -l tinyCapture –pacPretrained saves/0toplay.pt“

DQN on Pacman-CTF on mediumCapture (red one is trained) (uses python 3.5):

- Open terminal in folder „DQN-CTF-tinygrid“
- Start with „sudo python3 capture.py -r baselineTeam3 -b baselineTeam -x 0 -n 10 -l mediumCapture –pacPretrained saves/0toplay.pt“

Literatur

[1] Eden, Tim, Anthony Knittel und Raphael van Uffelen. Reinforcement learning, unknown year. URL `https://www.cse.unsw.edu.au/~cs9417ml/RL1/algorithms.html`. [Online; accessed-February-2018].

[2] Fiszel, Ruben. Reinforcement learning and dqn, learning to play from pixels, 2016. URL `https://rubenfiszel.github.io/posts/rl4j/2016-08-24-Reinforcement-Learning-and-DQN.html`. [Online; accessed-February-2018].

[3] Hessel, Matteo, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Daniel Horgan, Bilal Piot, Mohammad Gheshlaghi Azar und David Silver. Rainbow: Combining improvements in deep reinforcement learning. *CoRR*, abs/1710.02298, 2017. URL `http://arxiv.org/abs/1710.02298`.

[4] Juliani, Arthur. Simple reinforcement learning with tensorflow part 4: Deep q-networks and beyond, 2016. URL `https://medium.com/@awjuliani/simple-reinforcement-learning-with-tensorflow-part-4-deep-q-networks-and-beyond-` [Online; accessed-February-2018].

[5] Keon. Deep q-learning with keras and gym, 2017. URL `https://keon.io/deep-q-learning/`. [Online; accessed-February-2018].

[6] Paszke, Adam. Reinforcement learning (dqn) tutorial, 2017. URL `http://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html#dqn-algorithm`. [Online; accessed-February-2018].

[7] Schaul, Tom, John Quan, Ioannis Antonoglou und David Silver. Prioritized experience replay. *CoRR*, abs/1511.05952, 2015. URL `http://arxiv.org/abs/1511.05952`.

[8] Stachniss, Cyrill und Wolfram Burgard. The markov decision problem, 2003. URL `http://ais.informatik.uni-freiburg.de/teaching/ss03/ams/DecisionProblems.pdf`. [Online; accessed-February-2018].

[9] university, Berkeley. Project 3: Reinforcement learning, 2014. URL `http://ai.berkeley.edu/reinforcement.html`. [Online; accessed-February-2018].

[10] university, Berkeley. Contest: Pacman capture the flag, 2015. URL `http://ai.berkeley.edu/contest.html`. [Online; accessed-February-2018].