
Pitfalls Encountered When Implementing Complex Neural Networks Like Progressive Neural Networks

Dennis Verheijden
s4455770

Joost Besseling
s4796799

Abstract

We show that it is possible to implement a deep Q-network in Chainer. We use different techniques to train the network efficiently. We manage to achieve super human performance in a number of games.

1 Introduction

The Atari Gym Brockman et al. [2016] environment is a modern environment to train and test various reinforcement learning algorithms, on a difficult, real time, task. The gym has a large variety of different games that we can test our algorithms on. We wanted to implement and train a novel neural network in Chainer Tokui et al. [2015]. We implemented the deep Q-learning network as introduced by Mnih et al. [2013] and Mnih et al. [2015].

In this paper we will give an overview of the various techniques that are needed when training a deep Q-network. We will also give a short introduction to the problem of reinforcement learning.

2 Background

2.1 Atari

On a high level the Atari environment works as follows. An agent is playing a game, choosing different actions and different times. When the agent manages to score a point, or beat a level, the game will return a reward. If the agent dies, the game will return a different reward. It is easy to see how this leads to difficulties, since there might not be a direct relation between the last action and the last reward.

An example from the game Pacman. The agent has to make sure it is not trapped by the little ghosts. If it is trapped, it will not immediately die, the death will only occur after some amount of time. So the immediate positive reward the agent gained by eating a dot, might lead to the agents demise.

On a lower level, the atari environment works as follows: it uses a default setup for all the games available. First it has to be initialized, after that we can iterate through the screen by calling the *next* function on the environment with an action. The network will return the next state, the reward, and whether the game is finished or not. When the game is finished, we know we have to either reset the environment, or we are done with the training or evaluation on the game.

2.2 Reinforcement Learning

As stated before, the Atari environment provides a specific learning environment. The agent receives positive or negative rewards and has to change its actions based on these rewards. This structure is called reinforcement learning. Reinforcement learning differs from both supervised and unsupervised learning in a number of ways. It differs from supervised learning because we don't have a training

set of labeled data, we only gain the ‘labels’ after performing the action. Different actions result in different rewards.

It differs from unsupervised learning too, the network does receive information about whether the performed action was good or bad. So we are not simply learning the general structure of an Atari game, as is common in unsupervised methods.

This poses a number of different challenges. For example, the distance between an action and the reward. Also, the agent will see different parts of the game, based on the actions that it takes. This has as a consequence that there is a balance between exploration and exploitation, if the agent always chooses the same action that has an immediate reward, the agent might get stuck in the game. But if it acts too randomly it might not learn to get past a difficult part of the game, in which it has to take a dangerous action first Sutton and Barto [2017].

2.3 Q-learning

One of the ways to deal with the difficulties is Q-learning. Q-learning was popularized in the famous paper from Google Deepmind Tokui et al. [2015]. The assumption of Q-learning is that there is some function called $Q(s, a)$ that can predict the value of doing action a when the agent is in state s . If we assume that we have this function available, we can utilize it by evaluating it at every state and choosing the action with the highest value. This will then result in perfect play. The problem is, of course, that we don’t know this Q -function.

One approach to approximate the Q function is the so called tabular Q learning. The main idea is to use the Bellman equation:

$$Q(s, a) = r + \gamma \max_{a'} (Q(s', a')),$$

here s is the current state, a is the chosen action, r and s' are the result and the next state resulting from action a . In tabular Q-learning, Q is approximated by a table of size $S \times A$, where S is the size of the state-space and A is the size of the action space. Unfortunately, the state-space is really big in just the simple Atari games, so this method is not feasible for anything but the smallest problems.

To tackle this specific problem, we replace the table with a large neural network, which will be trained to estimate the Q function for every state. There are numerous challenges when training a deep Q-network. In this paper we will discuss a few of the problems that have to be considered.

2.4 Network Architecture

model beschrijven in termen van *states rewards and actions* + Q-learning (is pure value iteration aka rewards propagaten van de terminal state naar de eerste zet in episodes)

The network architecture that we will be using has a number of different layers. Since we are dealing with raw pixel data, we have chosen to use three convolutional layers. Convolutional layers are especially good at handling visual input, because they make use of the special structure of the pixel data. After these layers we have to fully connected layers. The output consists of the possible actions the agent can take. Between each of the layers we add rectifier non-linearity.

3 Related Work

Not sure... Misschien wat vertellen over de guru paper Mnih et al. [2013] + mogelijke verbeteringen in NATURE Mnih et al. [2015]

Reinforcement learning is a very active field, so there is a lot of related work. In 2013 Mnih et al introduced a novel way to tackle the problem of reinforcement learning which they called Q-learning. This method is highly effective, achieving super human play in a number of games.

More recently, Google achieved another major success when they managed to beat the worlds best Go players, using a version of reinforcement learning

Another problem of Neural Networks is the so called Catastrophic Forgetting which means that if we take a network, and train it on problem 1, for example one of the games in Atari. Then we train it on a different game, problem 2, the network will ‘forget’ how to play problem 1.

Another problem is that of knowledge transfer. Some games are fairly similar, so learning one game might help you play another game. But can a neural network transfer the knowledge from one game to the next? There are a number of ways to tackle these problems. For example the work that Rusu did on Progressive Neural Networks Rusu et al. [2016]. In this paper they introduce a network architecture that adds a new network for every problem that it encounters, but incorporates all computations of the previous networks. The idea is that these computations speed up the learning of the next network, because some of the knowledge transfers from one game to the next.

4 Deep Q-Learning

Because deep q-learning is computationally expensive, we have to use a lot of optimizations. There is also a lot of instability in deep Q-learning, so we use some methods to reduce the instability of learning. In this section the techniques used to tackle these problems are described.

4.1 Preprocessing

Hoe we data eerst verwerken voor complexiteit vermindering + wat input model is (stacked frames) zie Mnih et al. [2013]

Due to the complex nature of raw pixel data, it is a good idea to do a manual preprocessing steps on the data. Some of the difficulties of the raw pixel data: There are a lot of unused pixels in the game, for example the top of Pong only contains the score, so there is no useful information on this part of the screen. We can remove this from the input of the network, to reduce the complexity of the game. Another example is the fact that there is coloured data, the initial input exist of three (i.e. red, green, and blue) channels. By combining them into one input layer, we reduce the complexity even more, without throwing away necessary information.

Another problem with Atari games is that one screen doesn’t provide enough information to be able to infer the state of the game (e.g. what direction is the ball moving toward? What direction is the opponent going? Or in pacman, what directions are the ghosts moving in?). In order to mitigate this effect, first we used the difference between the previous screen and the next screen. But since this doesn’t provide that much extra information, we decided to use another technique in which we stack the last n frames Mnih et al. [2013]. Essentially, we introduce 4 channels, in which each channel contains an entire screen of the game. The idea is that the combination of the last four frames contain enough information to be able to accurately predict the state of the game.

Other possible preprocessing techniques could be ... ???

4.2 Reward Clipping

Different environments in the Atari gym have different reward structures, which means that training on the different games would mean that we need different hyper parameters and learning rates. To reduce this effect, we clip all non-zero rewards to either 1, or -1 . Of course, this will have a negative effect on the agent, since it cannot differentiate between small and big rewards. We assume that this effect is not too big Mnih et al. [2015].

4.3 Replay Memory

Om patronen te voorkomen Mnih et al. [2013]

Since we are playing the game on real time in the atari environment, there will be a high correlation between successive game-states. Which might lead the algorithm to get stuck in a negative feedback loop. Consider the scenario in which moving to the right in the beginning of the game gives an immediate positive reward, but makes the player get stuck in some ?????!???

In order to mitigate this effect, we use experience replay Mnih et al. [2013]. We let the network gather some experience, before we start training the network; we let the network play the game, and we save

the current game state, the taken action, the resulting reward and the resulting game state to a list of experiences. After the network has gained enough experience, we start training the network. Training is done by uniformly taking experiences from the experience list, and calculating the necessary things to train the network. More details on this will be provided in chapter 4.4.

4.4 Training details

model initializatie + parameters etc. (qua hidden layers e.d.) Huber loss vs. MSE

The model is trained by utilizing the techniques mentioned before. The network will be trained to approximate the Q^* function. Using the simple loss function:

$$L = \left(r + \gamma \max_{a'} (Q(\theta_i, s', a')) - Q(\theta_i, s, a) \right)^2$$

in which θ_i are the current weights, s' is the state that is obtained after performing a on the state s , r is the reward obtained by performing action a on state s .

4.5 Double DQN

Because the standard version of deep Q-learning uses the same Q function to estimate the future reward and predict the current reward, there might arise feedback loops while training. To prevent this from happening, we can use two deep Q-networks, one is used to predict the future reward and the action, and one is used to approximate the value that the network should predict. The adaptation of the loss function looks as follows

$$L = \left(r + \gamma \max_{a'} (Q(\bar{\theta}_i, s', a')) - Q(\theta_i, s, a) \right)^2$$

in this function $\bar{\theta}_i$ denotes the fact that the weights of this network are constant while training a single batch. Because we assume that the network gets better along the way, we still want to periodically update both Q networks. This is the only change when utilizing the double DQN approach.

4.5.1 Training

Start the training by initializing the necessary things: initialize the weights of the network, initialize the experience list, initialize the Atari environment and set the current screen to be all zeros. After that, we can start training the network.

Then we loop for some number of entire games over the entire next phase of the training. We use the network to decide the actions taken in the game. If a game is finished, we simply reset the environment and continue training, we also have to reinitialize the current game state. Every time an action is taken, we collect the current state, the action, the reward, and the next state and store these in the experience list. After we have gathered sufficient experience, we start training on this experience. This is done by sampling random experiences from the list, and using the values from the experience to calculate the loss, and do backpropagation with this loss, to estimate how we have to update the network.

5 Results

6 Conclusion

We have shown that the Google Deepmind paper provides a very good baseline algorithm for future reinforcement learning tasks. It is possible to train a network and achieve similar results as they got in their paper. We have utilized many of the same techniques and achieve very similar results.

7 Discussion

Although we have implemented the standard deep Q network, there are many different network structures that each have their own advantage and disadvantages. One particularly interesting network

architecture is the Progressive Neural Network, this structure is specifically strong when learning different games with the same network.

Although we can train our network to act in the different games with relative ease, we have to start over from scratch each time we train the network.

There are many other applications for reinforcement learning, of which we probably have only seen the beginning!

7.1 Difficulties

7.2 PNN/ future work

References

- Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016. URL <https://github.com/openai/gym>.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. Original Atari DQN Paper.
- Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. Improvements Upon Original NIPS Paper.
- Andrei A Rusu, Neil C Rabinowitz, Guillaume Desjardins, Hubert Soyer, James Kirkpatrick, Koray Kavukcuoglu, Razvan Pascanu, and Raia Hadsell. Progressive neural networks. *arXiv preprint arXiv:1606.04671*, 2016. URL <https://arxiv.org/pdf/1606.04671>.
- Richard Sutton and Andrew Barto. *Reinforcement Learning: An Introduction (Complete Draft)*. The MIT Press, 2017. URL <http://incompleteideas.net/book/bookdraft2017nov5.pdf>.
- Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. In *Proceedings of Workshop on Machine Learning Systems (LearningSys) in The Twenty-ninth Annual Conference on Neural Information Processing Systems (NIPS)*, 2015. URL http://learningsys.org/papers/LearningSys_2015_paper_33.pdf.