Project 1: Navigation

Author: Harald Wilbertz

The project uses Reinforcement Learning methods, in particular:Deep Q-learning and several variants, to learn a suitable policy using the Unity banana environment. The environment consists of a continuous state space with the goal to collect yellow bananas (reward: +1) while avoiding blue bananas (reward: -1). The state space has 37 dimensions and contains the agent's velocity, along with ray-based perception of objects around agent's forward direction.

The 35 dimensions of ray perception contain the following bits of informaton: - 7 rays are projecting from the agent at the following angles (measured in degrees): [20, 90, 160, 45, 135, 70, 110] where 90 is directly in front of the agent. - Each ray is 5 dimensional. If it encounters a detectable object (i.e. yellow banana, wall, blue banana, agent), the value at that position in the array is set to 1.

The last number within each ray is a distance measure which is a fraction of the ray length normalized to 1. Each ray is [Yellow Banana, Wall, Blue Banana, Agent, Distance]. For example, [0,1,1,0,0,0.2] means that there is a blue banana detected at 20% of the distance.

The velocity of the agent is two dimensional: left/right velocity and forward/backward velocity.

There are 4 actions to choose from: move left, move right, move forward and move backward.

The report contains three parts:

- **Design and Implementation**
- **Results**
- **Future Improvements**

Design and Implementation

At the core of the learning algorithm isDeep Q-learning, which exceeded human-level performance in 49 Atari games. The algorithm is a modification of Q-learning. Q-learning belongs to the class of Temporal-Difference learning (TD-learning) methods. The idea of Q-learning is to learn an action-value function, that estimates future rewards given a state and a choosen action.

Due to the fact that our space is continous, a traditional tabular representation is difficult to use. Deep neural networks are used as universal function approximators instead. Using SGD (Stochastic Gradient Descent), we update the weights of the neural network in order to get an approximation for the action-value function.

We choose a 2-hidden layer network with layers having 64 and 32 hidden units with relu (rectified linear unit) activation applied after each fully-connected layer. Adam (Adaptive Moment Estimation) was used as the optimizer for finding the optimal weights. The network implementation can be found in the model.py file. Additional layers did not improve the algorithm.

- **Exploration vs. Exploitation**: To address this exploration vs. exploitation dilemma, an ε-greedy algorithm was implemented.The agent "explores" by picking a random action with some probability epsilon ϵ.

However, the agent continues to "exploit" its knowledge of the environment by choosing actions based on the policy with probability (1-ϵ). The value of epsilon is decayed over time, therefore the agent favors exploration initially, but favors exploitation later. The starting and ending values for epsilon and the decays rate are hyperparameters for the algorithm.

- **Fixed Q-targets**: When we perform an update of our network parameters w to make Q closer to the desired result, we indirectly can alter the values for Q and other states nearby. This can make training quite unstable. When we update Q for a state s, then on subsequent states s' Q(s', a') might become worse. To make training more stable, a target network can be used. This target network is a copy of our network. This network is synchronized with the main network periodically. Therefore we have two separate networks, one is the online network being learned and the other being the target network. The weights of the target network are taken from the online network itself by freezing the model parameters for a few iterations and updating it periodically after a few steps. By freezing the parameters this way, it ensures that the target network parameters are significantly different from the online network parameters.

- **Experience Buffer**: We are trying to approximate a complex linear function Q with a neural network using SGD in order to estimate the network weights. One of the fundamental requirements for SGD (Stochastic Gradient Descent) is that the training data are idependent and identically distributed. (abbreviated as i.i.d.) But our samples are not independent. Even with a large batch of samples, most of them will be quite close to each other, since they belong to the same episode. In addition the distribution of training data is not identical to the samples provided by the optimal policy, that should be learned. To avoid these problems, a large buffer of past experiences is used. The current implementation uses a buffer of fixed size, with new data added to the end of the buffer, pushing older experiences out of it.

All of the above mentioned techniques were incorporated. The entire implementation was done in PyTorch. Also, various other improvements have been proposed upon the original DQN algorithm, and this repository contains the implementations of two of those:

- **Double DQN**: The basic DQN has a tendency to overestimate the values for Q. This can be harmful for training performance and results in suboptimal policies. The main reason for this behaviour is the use of the max operation in the Bellman equation. In a paper Deep Reinforcement Learning with Double Q-Learning the authors van Hasselt, Guez and Silver proposed a solution. The idea of Double DQN is to disentangle the calculation of the Q-targets into finding the best action and then calculating the Q-value for that action in the given state. Double DQN use one network to choose the best action and the other to evaluate that action. The idea is that if one network chose an action as the best one by mistake, chances are that the other network wouldn't have a large Q-value for the sub-optimal action.

- **Dueling DQN**: In a paper Dueling Network architecture for Deep Reinforcement Learningthe authors Ziyu Wang, Tom Schaul, Matteo Hessel, Hado van Hasselt, Marc Lanctot, Nando de Freitas described further improvements for DQN's. Their dueling network uses two separate estimators: one for the state value function and one for the state-dependent action advantage function. The main benefit of this factoring is to generalize learning across actions without imposing any change to the underlying reinforcement learning algorithm. Their results show that this architecture leads to better policy evaluation in the presence of many similar-valued actions.

The implementaion of Double DQN and Dueling DQN used code snippets from Deep Reinforcement Learning

Hands-On, Maxim Lapan, Packt Publishing, Chapter 7.

Hyperparameters

The code uses a lot of hyperparameters. The values a are given below

Hyperparameter	Value
Experience buffer size	100000
Batch size	64
Gamma (discount factor)	0.99
Tau	1e-3
Number of episodes	2000
Update interval	5
Learning rate	1e-3
Max number of steps per episode	2000
Epsilon start	1.0
Epsilon minimum	0.1
Epsilon decay	0.99

Results

The best performance was achieved by**DQN** where the reward of +13 was achieved in**338** episodes. Double DQN required 430 episodes, Dueling DQN 442 episodes. The plots of the rewards for the different variants of DQN is shown below:

Double DQN	DQN	Dueling DQN
double-dqn	dqn	dueling double dqn

Ideas for improvement

In a paper Rainbow: Combining Improvements in Deep Reinforcement Learningthe authors Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, David Silver describe several improvement to DQN. Double DQN and Dueling DQL have already been implemented, but there are still other possible improvements:

- N-steps DQN: A simple unrolling the Bellman equation could help to improve speed and stability.

- Noisy neworks: Adding noise to the network weights could make exploration more efficient. Either independent gaussian noise or factoried gaussian noise can be used.
- Prioritized replay buffer: By assigning priorities to buffer samples according to training loss and sampling the buffer proportional to these priorities training might improve.
- Categorical DQN: Q values might be replaced with more generic Q-value probability distributions.

Last not least, hyperparameter search should improve the performance too.