

Snake AI

2022

Faculty or Institute

INF-1600, 2022

Amund Strøm & Morten Jansen

(amund.strom@gmail.com, morten-jansen@hotmail.com)

Abstract and Keywords

The topic for this report is the field of Reinforcement Learning called Deep Q-Learning. The project described in this report will create a Deep Q-Learning agent able to play the video game snake. The results show an agent able to perform well in its environment.

Some important keywords for this task are Reinforcement learning, Q-learning, Q-value, neural network, environment, exploitation, exploration and reward.

Contents

1 Introduction	1
<i>1.1 Background</i>	<i>1</i>
<i>1.2 Related work</i>	<i>3</i>
<i>1.3 Problem statement, problem and Aim</i>	<i>4</i>
2 Contribution	5
<i>2.1 Design</i>	<i>5</i>
<i>2.2 Results</i>	<i>9</i>
<i>2.3 Evaluation and discussion</i>	<i>10</i>
3 Conclusion	10
4 References	11

1 Introduction

In this project we are creating an agent for the game Snake. To do this, we will first implement a simple environment, the game of snake, before implementing a Q-learning agent and model to play the game. Q-learning is a type of reinforcement learning, which is reward based learning. This report will contain descriptions of our progress, implementation, design choices and thoughts on the subject and task.

1.1 Background

Machine learning was defined as early as 1959 by Arthur Samuel, who was an American pioneer within the field of computer science and artificial intelligence research.¹ He defined the term machine learning as ‘the field of study that gives computers the ability to learn without being explicitly programmed’, which still holds up to this day. A more modern definition could be ‘Machine learning is programming computers to optimize a performance criterion using example data or past experience.’² There is no universally accepted definition of this term, but the core premise would essentially be the same as these mentioned definitions.

Reinforcement Learning is the branch of Machine Learning that allows training an agent on how to select the best sequence of actions depending on the current state to maximize long term returns. Reinforcement Learning mimics the real-life process of learning through trial and error.³

This is where the term Deep Q – Learning (DQL) and Deep Q – Network (DQN) comes in, as the approach we used to create an agent to play snake. Deep Q – Learning is a type of reinforcement learning algorithm that produces approximations of the Q value for all possible actions the agent can perform within the environment. In other words, it is an algorithm within reinforcement learning that will approximate the Q value for every action.

To approximate the Q value, it utilizes a Deep Neural Network also referred to as the Deep Q – Network. This network takes the state of the agent within the environment represented as real numbers, and outputs an approximation of the Q value for every action. The largest Q value is picked as the action the agent performs.

¹ <http://infolab.stanford.edu/pub/voy/museum/samuel.html>

² <https://www.geeksforgeeks.org/introduction-machine-learning/>

³ <https://towardsdatascience.com/ai-learns-to-fly-part-2-create-your-custom-rl-environment-and-train-an-agent-b56bbd334c76>

The Q value is the result from an equation called the Bellman equation.

$$Q(s, a) = \sum_{s'} P_{ss'} [R_{ss'} + \gamma \sum_{a'} \pi(s', a') Q(s', a')]$$

Figure 1: The Bellman equation.

This equation is quite complicated, but to summarize it, 'the agent updates the current perceived value with the estimated optimal future reward which assumes that the agent takes the best current known action. In an implementation, the agent will search through all the actions for a particular state and choose the state-action pair with the highest corresponding Q-value.'⁴ To rephrase, this equation will choose the action in the current state that maximizes future rewards. This means that this Q – learning system is reward based, meaning that we reward our agent if it does a good action, getting closer to the goal of the environment, and the opposite if it gets further away from the goal.

The agent does not know the best possible action for the get go, it needs to acquire experience and learn from its mistake. This is where exploration and exploitation are important. At the start of the program the agent will explore for the most part and fill its memory with experiences by doing random actions. The longer the program runs the memory will get filled with actions that produce more rewards than others. At this point the agent stops exploring for the most part and acts on its memories, this part is referred to as the exploitation phase. With this comes some difficulties called the exploration-exploitation dilemma. At what point do we know that the agent has explored enough and should only act on its decisions? There is no real answer to this problem, but there are some different approaches that aim to solve this problem. In the case of environments such as the snake game where the goal is to increase the score indefinitely. You could tell the agent to stop exploring after it reaches a set amount of score. As mentioned, this would only work in an environment where the goal is clear. Another and more common solution would be to decrease the rate of exploration by the amount of memories the agent has acquired.

⁴ <https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc>

1.2 Related work

Task 5

Machine learning is the field of science that enables a computer to learn without being explicitly programmed. It will train a piece of software using an algorithm or model to make predictions and improve itself over time. There are several methods for classifying machine learning processes, these are the most common algorithms.

- Supervised learning presents the model or algorithm with inputs and outputs, it will then find connections and patterns between the inputs and outputs. It will continue to learn until the desired accuracy is met. Using this method means that we need to know that the data is correct and have maybe thousands or millions of data to meet the desired accuracy.⁵
- Unsupervised learning presents the input with no known output, meaning that the learning algorithm is on its own to find structure in its inputs. It is used to cluster together the input into different groups.⁵
- Reinforcement learning interacts with an environment and must perform a certain goal and will receive positive or negative feedback depending on if it got closer or reached the goal.⁵

Task 7

Neural Networks are inspired by the biological neural networks that exist inside animal brains. Neural networks are a series of nodes or neurons that are connected with each other. The connection is used to transmit signals with different weights. The input neurons are real numbers, these numbers are transmitted into the network and each neuron uses an algorithm to calculate the weight of the output. This process is repeated until the output neurons, and the result will be neurons with different values, and the neuron with the greatest value is what the network calculated to be the best action.⁶

The good thing with neural networks is that it works in a variety of different applications. Such as financial services, forecasting, market research, fraud detection, robotics and many more. The reason is that it only needs inputs that can be represented via a real number, then it will learn and calculate the best possible action.⁷

⁵ <https://www.geeksforgeeks.org/getting-started-machine-learning/>

⁶ https://en.wikipedia.org/wiki/Artificial_neural_network

⁷ <https://www.investopedia.com/terms/n/neuralnetwork.asp#toc-types-of-neural-networks>

Algorithms are used to learn the network and are often referred to as optimize algorithms. These algorithms have different memory usage, speed and precision. These are some important algorithms: Gradient Descent, Newton Method, Conjugate Gradient, Quasi-Newton Method, Levenberg-Marquadt algorithm.⁸

Deep learning is a subcategory of neural networks which adds 1 or more “hidden layers” which are layers with neurons or nodes in-between the input and output layer.⁹ You can find deep learning in areas such as Automatic speech recognition, image recognition, Visual art processing, Natural language processing, etc. Some of the most common combinations of networks are Feedforward neural network, which is very basic, and data only flows in one direction. Convolutional neural network (CNN) consists of 1 or more convolutional layers with very few parameters but are very deep, this makes it effective for image recognition.¹⁰

1.3 Problem statement, problem and Aim

The problem revolves around getting the snake to play the game without either eating its own body or hitting the edges of the screen while eating more apples. To accomplish this, several inputs to the reinforcement learning function must be defined and several problems must be taken care of, such as. What should reward the agent? Should there be any punishments for bad actions? What should be the relationship between exploration and exploitation

We aim to make a snake agent that is able to win the game (eat apples until the screen is filled with the snake). This could be called a secondary aim or bonus aim as it seems difficult and unrealistic to accomplish. So the main aim will be to make a snake agent which has a higher average score than we can achieve in five attempts.

⁸ https://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network

⁹ https://en.wikipedia.org/wiki/Deep_learning

¹⁰ <https://www.educba.com/deep-learning-networks/>

2 Contribution

2.1 Design

The design of the software is divided between two modules. One being environment.py which contains the environment for the game of snake, and the other being main.py, which contains the model and the agent. The environment is based on the game of snake as previously mentioned. The module contains a class called environment as well as random implementation where the snake's next action is always randomized. To start off, the environment is initialized with the width and height for the interface and the size for the food and snake as well as the speed the simulation will run at.

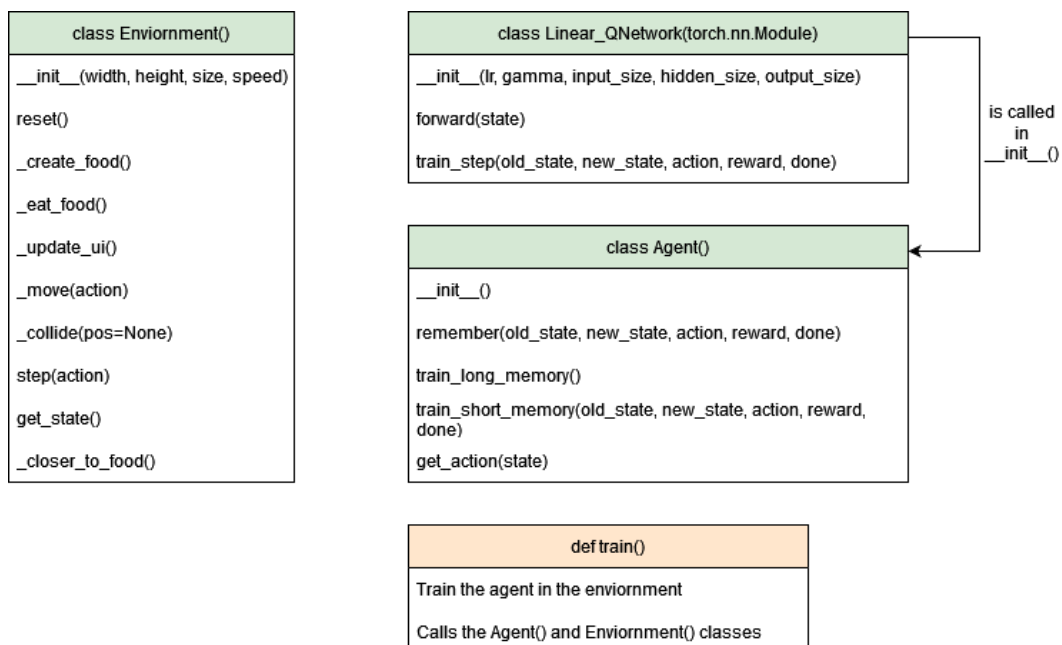


Figure 2: Class diagram over the agent and environment implementation.

The first method in the environment class is the reset method. This method contains several attributes such as iteration, score, the possible directions, the direction, all of the snake's positions as well as its food. Iteration and score is initially set to zero and food to none. The default direction of the snake is going right. The method ends with using the method `create_food` to create the first piece of food.

`_create_food`

Then there is the `_create_food` method, which creates new food at a random location, but not if the snake is there. If the food is placed where the snake is, the method calls itself again and initializes a new location for the food.

`_eat_food`

Next is the method `_eat_food` which initially sets reward to be 0. The reward is a variable to adjust the snake's behavior to be beneficial. If the food and snake head collide, the score is incremented, reward set to 10, a new piece inserted to the snake list and `_create_food` is called. Lastly, outside of the if statement, reward is returned.

`_update_ui`

`_Update_ui` is a method to update the user interface, it goes through all the positions of the snake and draws them and does the same with the food position. The score display is also updated here. Finally, the method calls the pygame display update method.

`_move`

The move method is used to move the snake in a given direction. It takes in an action, the action can either continue on the same path, go left or go right.

`_collide`

This is a method to determine if the snake is colliding with either a wall or itself. If no position argument is passed it will check the head of the snake by default. If it is passed a position, it will check for that position. This method is used both to determine if the snake has died and also to see if the next position of the snake in a given case will result in a collision.

`step`

The step method is a key method for the program. It is used to play the next action of the snake. It takes in a snake object and an action as an argument. Step is responsible for checking if the user has quit the program. The method also checks if the snake is moving closer to the food, which gives it a +1 reward. This method uses `_collide` to determine if the snake has died, in which case it sets the reward to -10 and game over to be true before it returns score, game_over and reward. Lastly the method updates the ui and ticks the clock before returning game_over, score and reward.

`get_state`

Another key method which returns the inputs for the network model is `get_state`. It starts off by checking the direction of the snake and the next possible position. The method defines an array of 1s and 0s which are the inputs. The state has the following eleven elements: danger straight, danger right, danger left, direction right, direction down, direction left, direction up, food is right, food is down, food is left and food is up. These are all determined by logical operations which check if they are true or false. The state is returned as a numpy array of ints.

`_closer_to_food`

The last method of the environment class is the method which checks if the snake is closer to the food or not. It works by subtracting the previous position of the snake with the food, and same with the new position. Then gets the sum of x and y for both and checks if the new position is lesser in which case returns 1, if not it returns 0. This is then added to the reward in the step function.

`main.py`

The second module, which is the main module that contains both the class for the model and the agent.

`Class Linear_QNetwork`

The neural network is implemented in the `Linear_QNetwork` class. This class inherits from the `torch.nn.Model` class, which is the base class for all neural networks in the torch library. The model is built from 4 linear layers, one for the input layer, two hidden layers and one for output. The input layer size is 11, one for each element in the snakes' state. The hidden layers both have 256 nodes and the output has one node for each output which is 3. The optimiser we chose is Adam, which is a standard optimiser for training deep learning algorithms. The loss function chosen was MSE, which is short for mean squared error. MSE takes the difference between y and a prediction, takes the square of this result and divides it by y. So it takes the distance of all points to a prediction and finds the mean of this, squaring it to avoid negative numbers.

`forward`

The first method for the network class is the forward function, which is responsible for the forward propagation of the network. This is an important step in any neural network which determines how the inputs from the last layer are calculated by the next. In this case, the RELU activation function has been implemented.

`train_step`

`Train_step` is one of the most important methods in the entire program. The method begins by defining an old state, new state, action and reward as tensors. These are values that are used in torch's network models. It is then checked if the inputs are either an array or a single value, in which case the inputs are converted with the `unsqueeze` method from torch to fit with the rest of the code. It then defines the old q value using the forward method with the old state. To get the new q value it uses the bellman equation previously explained. The function then empties the gradients, uses MSE to calculate loss, backward propagates to update gradients and finally performs an optimization step.

Class Agent

The second class in the main file is the Agent class. This class is used to hold variables and methods for the agent to train and become a better snake player. Some important variables are the epsilon and the gamma variable. The epsilon value here is used to determine the amount of randomness or exploration. The gamma value is used in part to calculate the next step with the bellman equation. It also holds all the components for the memory which is to be trained using the model.

remember

The remember method indexes a new value to the five different parameters of the memory. It resets the index when it reaches the max memory constant value. The memory consists of five different arrays, the old state, new state, the action, reward and the done parameter.

train_long_memory

The method train_long_memory first checks if the memory contains enough information to be trained. It uses the batch_size to determine this. Then it defines the max memory variable, which is the same as the memory counter until it reaches the batch size. The method then makes an array with the batch size as the amount of elements and max_mem as the highest index and uses this to create arrays for the 5 different memories. These five arrays are then trained with the train_step method.

train_short_memory

Train_short_memory is a method to train one batch of memory, it just uses the train_step method with the agent model.

get_action

To get a new action for the agent, the get_action method is used. This method uses the epsilon value to determine whether it should do a random move, also known as exploration. Or run the state through the model to make a prediction to what is the best move, which is called exploitation. The function returns an action.

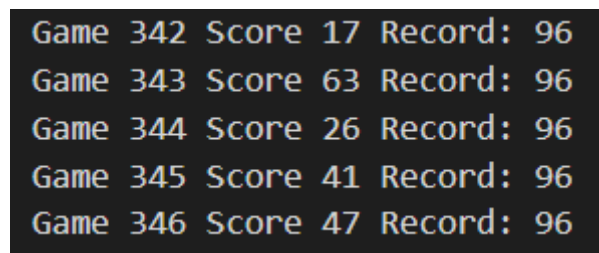
Finally there is the function used that runs the program, called train. The program starts off by defining several important variables such as the record, agent and environment. It then enters the game loop which is only stopped if the user exits the program. Initially the old state is defined using the get_state method on the environment. Then an action is fetched using the get_action method on the agent with the old state. Then the variables done, score and reward are extracted with the step method for the environment and a new state is defined subsequently. The program then trains the short memory with train_short_memory and saves the old state, new state, action, reward and done with the remember method. The program then checks if done is True, meaning if the game is over. If it is not, the program loops back. If it is, however, the environment is reset with the method reset, number of games is incremented and the agent's long memory is trained with train_long_memory. After this the program

checks if the new score is bigger than the record, in which case the record is updated. Lastly the n_games and score and record is printed after each episode.

2.2 Results

The final results show an agent that is able to learn and act independently in its environment. On average it uses about 20-30 games before it starts doing anything significant. At this point in the runtime it mostly chooses random actions and learns what result an action will produce within each state. In games 40-60 you see the agent get a higher record and mostly chooses actions from its memory that it thinks is the best possible action in its current state, but it will also choose some random actions and learn from it. In games 70-90 the snake chooses only from its memory and will reach much higher records. At this point the agent will somewhat stop learning since it will start overwriting its memory. It is of course possible to increase its memory, but we found that our computers started slowing down if we increased the size with any significant effect.

We also did a test where the agent stopped doing random actions in games after 200, to see if it would increase the skill of the agent. The program ran for about two hours and played a total of 346 games. The highest score it got was 96, and that happened in game 207. Seven games after it stopped doing occasional random moves.



Game 342	Score 17	Record: 96
Game 343	Score 63	Record: 96
Game 344	Score 26	Record: 96
Game 345	Score 41	Record: 96
Game 346	Score 47	Record: 96

Figure 3: Small sample of stats after running a total of 346 games

looking at the stats there is also a huge difference in score between each game, almost as if it is random if the agent performs well or not.

2.3 Evaluation and discussion

The program is far from perfect which was our original aim but it performs well enough in its environment and is able to do better than we could do in 5 games, which was our second aim. Learning from our test where we ran the program for 346 games, it proves that either the program decays in accuracy over time, or that it is mostly random. Regardless if it is random or not, we believe there are several factors that could improve the performance of our agent. Such as having a larger memory and increasing the number of memories to choose from during each step (called the batch size). Change the reward algorithm during runtime, so the agent would pick a smarter path to the apple rather than taking the shortest path, as it tries doing in our algorithm. Such an algorithm could look like taking a longer path so the agent does not trap itself inside its own body. Also implementing a way for the agent to remember where the rest of its body is, since our implementation only recognizes the state of its head. This would be difficult to implement since the agent knows very little about the actual environment, it only calculates one step ahead and is blind about everything else. Such an implementation where the agent remembers everything in its environment is way past our capability. We could also experiment with different activation functions, optimizers and loss functions, but we believe that Deep Q - Learning is the correct algorithm in the field of reinforcement learning.

Future work for this program will be to further optimize the rewards system. To make the snake behave in a more effective manner, it would be smart to implement as mentioned earlier some incentivization for not locking itself in. It would also be beneficial to implement more features to the UI to make it easier to change key values, thus making it far less efficient to experiment and find the best parameters. It could also be interesting to implement a model to adjust the parameters for the program to further improve it.

3 Conclusion

The task we chose turned out to be harder than anticipated. There were many challenges along the way. After finishing the first complete draft of the code and testing out the model, we quickly realized the snake would not get better at eating the food. Eventually realized that not only rewarding the snake for eating the food, but also moving in the direction of the food was important. Previously the snake would only get good at not crashing in itself and walls. This however spawned the new problem mentioned in 2.3 Evaluation and discussion, which is when the snake grows in size it still takes the shortest path to the food while not considering that it will lock itself in and eventually fail. Overall I would say we are happy with the results when compared to our goals and have learned a lot.

4 References

<https://www.geeksforgeeks.org/ai-driven-snake-game-using-deep-q-learning/>

<https://www.freecodecamp.org/news/train-an-ai-to-play-a-snake-game-using-python/>

<https://towardsdatascience.com/ai-learns-to-fly-part-2-create-your-custom-rl-environment-and-train-an-agent-b56bbd334c76>

<https://www.geeksforgeeks.org/introduction-machine-learning/>

<http://infolab.stanford.edu/pub/voy/museum/samuel.html>

<https://www.geeksforgeeks.org/getting-started-machine-learning/>

https://en.wikipedia.org/wiki/Artificial_neural_network

<https://www.investopedia.com/terms/n/neuralnetwork.asp#toc-types-of-neural-networks>

https://www.neuraldesigner.com/blog/5_algorithms_to_train_a_neural_network

https://en.wikipedia.org/wiki/Deep_learning

<https://www.educba.com/deep-learning-networks/>