Deep Reinforcement Learning for Snake

Anton Finnson and Victor Molnö

***Abstract*—Reinforcement learning algorithms have proven to be successful at various machine learning tasks. In this paper we implement versions of deep Q-learning on the classic video game Snake. We aim to find out how this algorithm should be configured in order for it to learn to play the game as well as possible. To do this, we study how the learning performance of the algorithm depends on some of the many parameters involved, by changing one parameter at a time and recording the effects. From this we are able to set up an algorithm that learns to play the game well enough to achieve a high score of 66 points, corresponding to filling up 46% of the playing field, after just above 5 hours of training. Further, we find that the trained algorithm can cope well with an obstacle being added to the game.**

***Index Terms*—Deep reinforcement learning, Q-learning, Snake, Neural network.**

I. Introduction

In recent years, the research fields of machine learning and artificial intelligence (AI) have received rapidly increasing attention. Many prominent corporations invest large sums of money in the development of these areas. For example, in 2014 the artificial intelligence company DeepMind was acquired by Google after managing to train a computer program to play Atari games at a superhuman level [1]. This first achievement together with further research by DeepMind, for example in [2], among others, is opening up a wide range of opportunities for AI. Some different examples, such as composing music and driving cars, are given in [3].

The celebrated algorithms deployed by DeepMind in the Atari experiment were variants of reinforcement learning. Reinforcement learning is a process through which a machine learns to achieve tasks without following a strict policy set up by humans, but instead by choosing future behaviour based on previous experiences, via the use of rewards. It resembles how humans learn and more closely how humans train animals by the carrot and stick principle. As an example, a strategy for teaching a dog to sit is to give the dog a treat when it sits and thereby promote that behaviour. Reinforcement learning is adapting this strategy and applying it to a computer program instead of an animal. The programmer communicates to the machine what task to solve only by defining rewards for certain behaviour. The program then processes these rewards and is able to use them in future decision making, even for situations not encountered before.

What has proven successful in constructing the algorithms for processing of rewards is a model loosely imitating how the human brain functions, called an artificial neural network (in this text often only network or neural network), explained thoroughly in [4]. In reinforcement learning, a network is "trained" to make correct decisions. This article investigates how such a network should be set up and how it should be trained to play the game Snake as efficiently as possible.

II. Background

A. Q-learning basics

The reinforcement learning algorithm discussed in this text is Q-learning [5], an algorithm that learns to make optimal decisions in a Markov decision process, (for further reading about the Markov decision processes, see [6]). Consider a system that can be in a set of possible *states* $\mathcal{S} = \{s_i\}$ affected by a set of *actions* $\mathcal{A} = \{a_i\}$. Say that a sequence $a^{(1)}, a^{(2)}, \ldots, a^{(n)}$ of n (not necessarily distinct) actions from $\mathcal{A}$ have been taken, so that the system have gone through a sequence of (not necessarily distinct) states $s^{(0)}, s^{(1)}, s^{(2)}, \ldots, s^{(n)}$, all in $\mathcal{S}$. A Markov model says that, if this is the case, the system's change to the next state $s^{(n+1)}$ only depends on the current state $s^{(n)}$ and the action $a^{(n+1)}$; it does not depend on any of the previous states or actions.

Video games are naturally described as Markov decision processes. In this setting, the state is a set of current game parameter values, such as player position, player velocity etc., and the action is the move done by the player. A playing and learning algorithm is often referred to as an *agent*. Algorithm and agent are used interchangeably in this paper. When implementing Q-learning on a game, a reward r is associated with every state change, i.e. every action taken by the agent, see Figure 1.

The idea behind Q-learning is to decide what action to take in a certain state by defining a *Q-function* (Q for quality) and updating this function by experience, as explained in [5]. The simplest form of Q-function is the *Q-table*, a matrix where the *Q-value* in row i and column j, denoted $Q(s_i, a_j)$, represent the quality of action a_j in state s_i. When a Q-table is properly updated, the straightforward choice of taking the action a associated with the highest Q-value, i.e. choosing

$$a \in \underset{a_j \in \mathcal{A}_l(s)}{\arg\max}\, Q(s, a_j) \tag{1}$$

when in state s, is a winning game strategy. Here $\mathcal{A}_l(s) \subseteq \mathcal{A}$ is the legal action space. However, the updating or training of a Q-table or any other Q-function to give a good measure of quality is not so straightforward and usually requires lots of computational work. This is the learning part of Q-learning.

In the general sense, learning amounts to fitting a function to some data by solving an optimisation problem. In linear regression analysis [7] a linear function $\hat{y} = X\hat{\beta}$ is fitted to some measured data y by minimising the error in the form of the sum of squares $\sum_{i=1}^{N}(\hat{y}_i - y_i)^2$ between predictions $\hat{y}_i$ and measured data y_i. Similarly, in Q-learning a Q-function is found by minimising some other *loss* function of the difference between the prediction $Q(s, a)$ and some target, denoted $T(s, a)$ in this text. As mentioned in Section I, the idea behind reinforcement learning is to learn by receiving

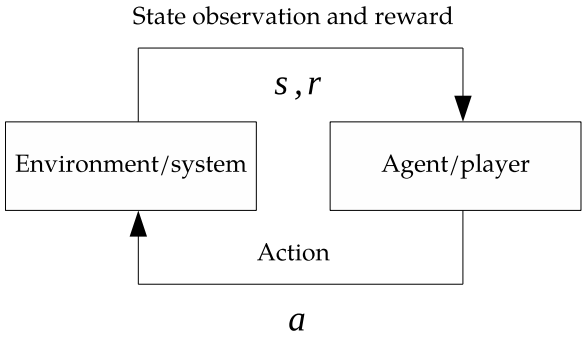

Fig. 1. Dynamics of the Markov process with a reward.

rewards based on the actions taken. This is done by letting the target be a function of the rewards. Say that the agent has taken n previous actions, so that the system is now in state $s^{(n)}$, and decides to take an action $a^{(n)}$. This will lead to some (not necessarily different) state $s^{(n+1)}$ and a reward $r^{(n+1)}$, according to Figure 1. Assume that the rewards are given in such a way that a good move returns a high reward, then a reasonable game strategy is to choose actions to maximise the expected value

$$\mathbb{E}\left[\sum_{k=0}^{\infty}\gamma^k r^{(n+1+k)}\right] \tag{2}$$

for some *discount factor* $\gamma \in [0,1]$ that compensates the increasing uncertainty of future states (explained in [5]). Observe that γ^k means "gamma to the kth power" in the usual sense, while $r^{(n+1+k)}$ means "the reward received from taking action number $n+1+k$."

We want the target function to fulfil the equation

$$T(s^{(n)}, a_j) = \mathbb{E}[r(s^{(n)}, a_j) + \gamma \max_a T(s^{(n+1)}, a)], \tag{3}$$

where $r(s^{(n)}, a_j)$ is the reward given when taking action a_j from state $s^{(n)}$. If the Q-function outputs $Q(s,a) = T(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$, then choosing actions by (1) maximises (2). If we have a target function such as in (3), the minimum of some loss function $f(Q,T)$ is acquired by taking state-action pairs (a,s) and performing steps of stochastic gradient descent

$$\Delta\theta = -\alpha\nabla_\theta f(Q(s,a), T(s,a)) \tag{4}$$

with α known as the *learning rate* and θ some set of parameters describing the Q-function (for the Q-table, θ is simply the Q-values).

To calculate T exactly it would be necessary to take into account state transition probabilities for all possible actions by the agent in all possible states of the system. If this was computationally feasible there was never a reason to invent Q-learning. Q could simply be set to equal T and the problem would be solved. However, while theoretically conceivable, the computational time required to do this is in most cases infeasibly long. Therefore, Q-learning approximates T by

$$T(s^{(n)}, a) \approx r^{(n+1)} + \gamma \max_a Q(s^{(n+1)}, a). \tag{5}$$

Now using (5) for a Q-table with

$$f(Q(s,a), T(s,a)) = \frac{|Q(s,a) - T(s,a)|^2}{2}, \tag{6}$$

(4) gives the descent formula

$$Q(s^{(n)}, a^{(n)}) \leftarrow (1-\alpha)Q(s^{(n)}, a^{(n)}) + \\ + \alpha\big(r^{(n+1)} + \gamma \max_{a_j \in \mathcal{A}} Q(s^{(n+1)}, a_j)\big), \tag{7}$$

where $a \leftarrow b$ means "update value a to be value b". As this is repeated many times for some state-action pair s_i, a_j the value of $Q(s_i, a_j)$ converges to the target value $T(s_i, a_i)$ and f is minimised. If this is done for all state-action pairs the Q-function will converge, by the proof in [8], to a point where the action taken according to (1) is always the optimal one. (7) can be modified slightly to work for other Q-functions.

Looking back at the Q-table, it is only reasonable to set up a table of values corresponding to state-action combinations for systems with small sets of pairs. Say that the $Q(s_i, a_j)$ are initialised randomly, then before the table can tell the quality of taking a_j in s_i the agent must have visited that state and taken that action at least once. For simple problems such as in [9], a Q-table is perfectly viable. But for many applications, games like Chess, Go, Atari, and Snake for example, the training process to exhaustively update values for all state-action pairs is computationally unfeasible (we will get to the limitations of a Q-table for Snake in Section IV-B). The solution is the artificial neural networks mentioned in Section I. Instead of taking indices i and j for states and actions and returning Q values, the network takes the value of some parameters describing the state and returns Q-values for all possible actions. Inputs with similar values may then have similar outputs and thereby the network generalises experience from one state to another so that the algorithm doesn't have to update all of them individually.

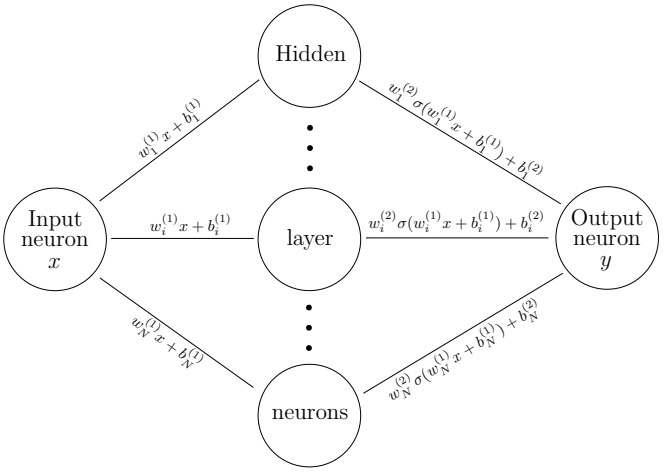

Fig. 2. Schematic representation of an artificial neural network with 1 input, 1 output and 1 hidden layer.

B. Neural networks

A *shallow* neural network with one input and one output is a function of the type

$$y(x) = \sum_{i=1}^{N} w_i^{(2)} \sigma(w_i^{(1)} x + b_i^{(1)}) + b_i^{(2)}, \tag{8}$$

where all $w_i^{(k)}$ are scalars called *weights* and $b_i^{(k)}$ are some other scalars called *biases*. The function $\sigma(x)$ is an *activation function*. Figure 2 shows a schematic graph of a shallow network with one input and one output. This can be generalised to handle multiple inputs and outputs.

Neurons with neither their input nor output as part of the input or output of the neural network are said to be in *hidden* layers. A network with more than one hidden layer is called a *deep network*. In *deep* reinforcement learning, *deep* neural networks are used. Figure 3 shows the idea behind deep sequential networks. They are similar to shallow networks but have more than one hidden layer. That the network is sequential means that the input for each layer is the output of the previous. Further, the input X to and the output Y of the deep network are in general vector elements in $\mathbb{R}^{m_0}$ and $\mathbb{R}^{n_{k+1}}$ respectively for some dimensions m_0 and n_{k+1}. The analogue of (8) for a deep sequential neural network with k hidden layers looks like

$$Y = W^{(k+1)}\sigma^{(k+1)}(...W^{(2)}\sigma^{(1)}(W^{(1)}X + \\ + b^{(1)}) + b^{(2)}...) + b^{(k+1)} \tag{9}$$

with weight matrices

$$W^{(i)} = \begin{bmatrix} w_{11}^{(i)} & \cdots & w_{1n_i}^{(i)} \\ \vdots & & \vdots \\ w_{m_i 1}^{(i)} & \cdots & w_{m_i n_i}^{(i)} \end{bmatrix} \tag{10}$$

for the ith layer, and bias vectors

$$b^{(i)} = \begin{bmatrix} b_1^{(i)} \\ \vdots \\ b_{m_i}^{(i)} \end{bmatrix} \tag{11}$$

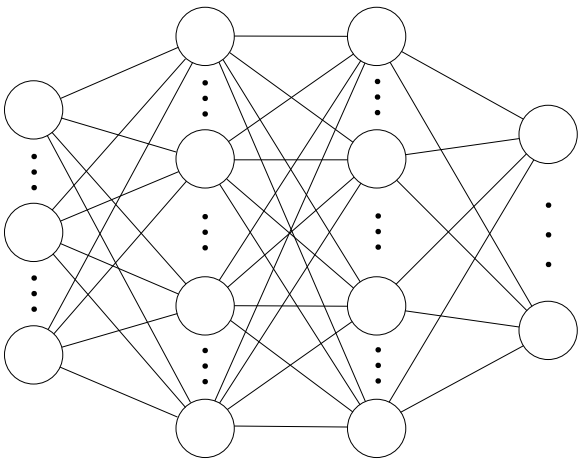

Fig. 3. Structure of deep fully connected neural network with 2 hidden layers. In general there can be any number of hidden layers.

for the same layer. The activation functions $\sigma^{(i)}$ are functions in one variable that apply component wise.

If the Q function $Q(s)$ is a deep neural network which outputs a vector of Q-values for the different actions in $\mathcal{A}$, then θ from (4) are all the weights $w_{jl}^{(i)}$ and biases $b_j^{(i)}$. So training comes down to updating weights and biases, which is explained in detail in [10].

In the Q-table setting, states are represented by values of a single index. When $\mathcal{S}$ becomes too large, it is useful to introduce some representation that parameterises the states. For a video game, one way to represent the states is to take X as a stretched out vector of the colour values for each pixel in the video frames. X can then be fed into a deep neural network similar to (9). However in images, pixels that are close to each other geometrically often come together to form patterns or *features*. Therefore it is often better to let X be a matrix in order to preserve the features of the image. This can reduce complexity of the neural network through use of convolution and pooling layers (for example as done by DeepMind in [2]). The convolutional layer, see Figure 4, takes an n-tensor input (in this text n is always 2, i.e. a matrix). A convolutional layer has m filters (tensors with elements a_{ij}), each of size $K \times K$, where K is called the *kernel size*, and the output of the layer is computed in steps. At each step a $K \times K$ subarea of the input layer, called the *receptive field*, is considered (shown as a 2×2 green area in Figure 4). Starting at the top left of the input layer, the receptive field is displaced S columns to the right between each step, with S being the *stride length*. This is done until the receptive field cannot be moved farther to the right. The receptive field is then moved S rows down and to the far left of the input layer. This process continues until the receptive field is in the lower right corner of the input layer. At each position of the receptive field a scalar product is calculated between the receptive field and each of the m filters. This gives a tensor of scalar products as the output of the convolution layer. By updating the a_{ij} values of the filters,

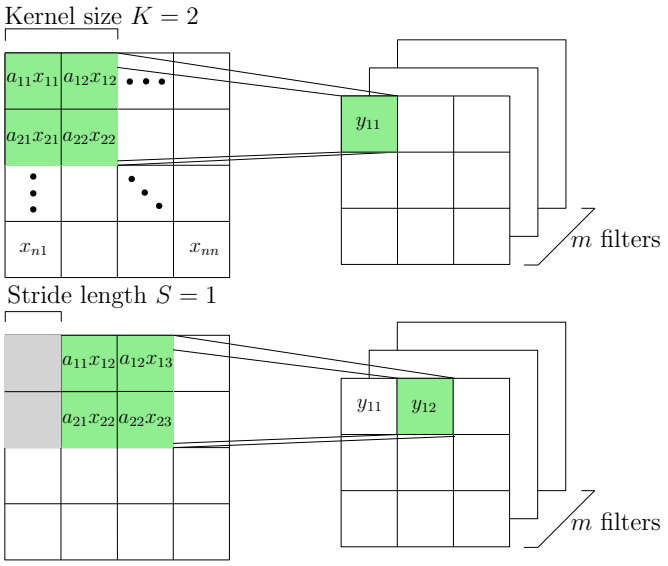

Fig. 4. Convolutional neural network with stride length $S = 1$ and kernel size $K = 2$.

the convolution layer can be trained to distinguish important features in the game. In Snake for example, one feature to distinguish could be that the head is close to the apple. More details about convolutional neural networks can be found in [11].

Pooling layers are usually used in combination with convolution layers to reduce the dimension of the signal. Like convolution layers they also have a kernel that is slid across the input. For every stop the kernel picks out some summary scalar of the receptive field. When striding more than one step the output is of smaller dimension than the input. Average pooling returns the average, max pooling (used in this project) returns the max value.

C. Training strategy

As mentioned in Section II, the tactic (1) only maximises (2) when the Q-function is already trained. To train the Q-function, the learning algorithm first needs to gather lots of training data. Here a dilemma occurs. If following (1) from the beginning, the algorithm might tend to repeat a few action sequences that have been initialised favourably, and thereby not gather sufficiently diverse training data. On the other hand, if the algorithm makes only completely random choices in the training phase, it will never reach the higher game levels and then never gather higher level training data.

A common and efficient way to overcome this dilemma is to use an *ε-greedy* policy. This is used in [2]. The policy is to choose *greedy* actions (following (1)) with probability $1-\varepsilon$. Set $0 \leq \varepsilon \leq 1$, then for every decision, sample a ξ from a standard uniform $U(0,1)$ distribution and compare ξ to ε. If $\xi \leq \varepsilon$, take a random action, else take the greedy action.

Further, it is customary not to keep ε constant throughout the training but rather to initialise it high and decrease it gradually as to take more random actions early and more greedy actions late in the training process.

Another problem that occurs in deep Q-learning is overestimation. This is explained in [12] along with a proposed solution, double deep Q-network. DDQN, for short, utilises two neural networks: one *online* network, denoted simply Q as before, for deciding on action a by (1) and one *target* network, Q_{target}, for evaluating $Q(s^{(n+1)}, a)$ in (5). The online network is trained continuously per a version of (7) where in the last term, $Q(s^{(n+1)}, a)$ is given by $Q_{target}(s^{(n+1)}, a)$, while the target network is updated to match the online network periodically, $Q_{target} \leftarrow Q$.

III. CartPole

Before taking on the task to construct an algorithm learning to play Snake, we instead consider an easier task, namely to construct an algorithm learning the game CartPole from OpenAI's package `gym` [13]. The goal of the game is to balance a pole placed on a cart which is placed on a one-dimensional track. At each time-step the player can choose to push the cart left or right. A game, or *episode*, finishes when the angle between the pole and the vertical axis exceeds 15°, the cart moves too far to either side, or at 500 time steps. Each episode is initialised by sampling a value from a standard uniform distribution $U(-0.05, 0.05)$ for each of the four variables describing the state of the game. We construct two different algorithms: one using plain Q-table Q-learning, and one using deep Q-learning. Both are implemented in Python 3.

A. Q-table approach

The first algorithm takes 4 binary numbers, representing the sign (i.e. whether the value is positive or negative) of the position of the cart, the velocity of the cart, the angle of the pole relative to the normal, and the angular velocity of the pole. This gives $4^2 = 16$ distinct states. Combined with the two possible actions, the Q-table becomes a 2×16 table of Q-values. Initially all Q-values are set to equal 0.

An ε-greedy policy is applied as explained in II-C, to increase exploration. ε is initially set to be 1 and is decreased according to $\varepsilon \leftarrow \varepsilon(1 - 0.11\varepsilon)$ at every random action.

After an action has been chosen, a reward $r = 1$ is given to the agent as long as the next state is non-terminal.[1] With these values, (7) is used to update the Q-table, with parameter values $\alpha = 0.1,\ \gamma = 1$.

B. Deep Q-learning approach

Instead of just considering whether each value is positive or negative we now want an algorithm that takes the floating point values of the four quantities representing the state of the cart-pole system, thus utilising all available information. To create a Q-table where each possible combination of values for the quantities is a state would be practically impossible. To overcome this problem we use a neural network, which we construct with the package `keras`[2] running on top of Tensorflow. This network takes as input the four floating point numbers representing the state, and outputs two Q-values for the the two possible actions. In between the input and output there are two fully connected hidden layers, consisting of 24 nodes each, with the Rectified Linear Unit activation function $\sigma(x) = \text{ReLU}(x) := \max\{0, x\}$.[3]

Based on the two numbers in the output, an action is chosen in the same way as in the first algorithm. The reward is also the same, as this is something connected to the environment rather than the agent, but this time ε is decreased in a different way. In this algorithm the network is trained in between each action by looking at old state-action-reward combinations, by choosing a batch of eight earlier moves at random from the last 10,000 moves. For each step taken in this fashion, ε is decreased by a factor 0.999, until it reaches a minimum value of 0.01. The learning rate is set to be $\alpha = 0.001$, and the discount factor is set to $\gamma = 0.95$.

C. Condition to stop training

To study a way of avoiding overtraining we implement a condition to discontinue the training, both for the Q-table and

[1] Falling is still bad, as this will lead to zero future rewards.

[2] Keras version 2.2.4

[3] This is probably not the optimal setup and just one layer will most likely suffice in this simple case.

the neural network. If the average score over five consecutive episodes is 350 points or more the training will cease completely, i.e. the Q-functions will not be updated further.

D. Results and discussion of CartPole

Iterating through 10,000 steps with the Q-table algorithm presented in Section III-A yields a trained Q-table frequently getting average scores over 350 points over ten consecutive games, as seen in Figure 5. It is however clear that the score fluctuates a lot, often going down to averaging between 50 and 100 points. The results of the neural network are overall worse but more consistent. The fact that the line representing the results of the agent trained with the neural network is smoother than the line of the agent trained with a Q-table is, at least partly, explained by the fact that the neural network trains eight times between each step that it takes, compared to the one time of the Q-table. This leads to the scores of the neural network to be farther apart along the x-axis, as an episode with the neural network consists of eight times more steps than an equally-scoring episode played with the Q-table. This

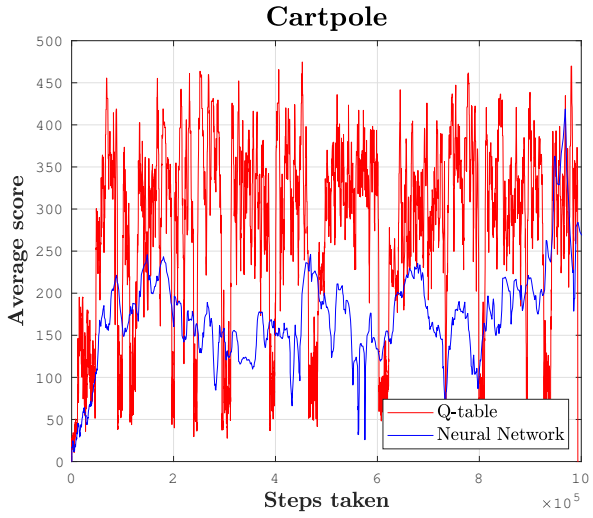

Fig. 5. Scores for CartPole at the end of episodes. The number of steps taken is the number of times that each algorithms has updated its Q-function. Average taken over the score at the end of the last 10 episodes.

is without the "smart" training-stopping condition presented in Section III-C. The results using this condition are presented in Figure 6. The scores from the agent training with a Q-table now quickly stabilises at a score somewhere in between 300 and 350 points, albeit with a large variation. The agent training with the neural network takes much longer to reach the threshold scores necessary for the training to cease, but when it does, around 70,000 steps in, it stabilises at a score of almost 450 points. Not only is the stable score higher than that of the Q-table trained agent, but the variation around this stable score is way smaller as well. This huge difference in variation cannot be explained by the factor eight difference in steps taken each game. A probable explanation is that the network receives and makes use of much more information about the system than the Q-table does. A final difference worth noting is that the Q-table algorithm runs much quicker

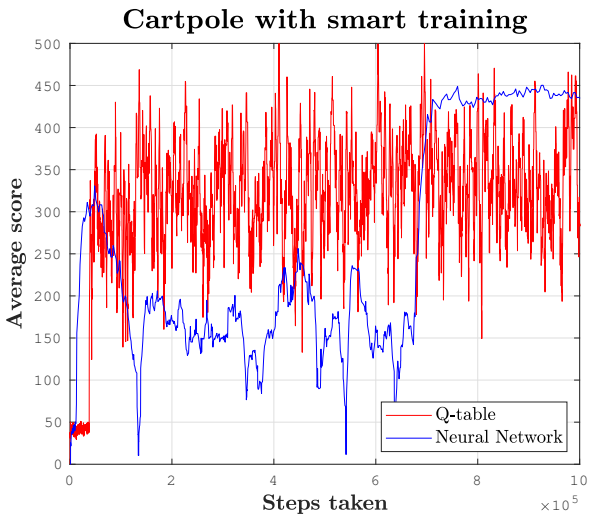

Fig. 6. Scores for CartPole, at the end of episodes, when a condition to stop training is applied. The number of steps taken is the number of times that each algorithms has updated its Q-function, or, if the training has stopped, the number of times it would have been updated had it not been for the no-training condition. Average taken over the score at the end of the last 10 episodes.

than the neural network algorithm. If speed is highly valued the Q-table approach thus has the upper hand.

E. Conclusions from CartPole

From the results of this investigation of learning by a Q-table versus by a neural network we can conclude a few things. One is that the Q-table learns to play well much quicker than the network. Another thing we conclude is that both methods have troubles with the scores dropping back down if the training continues. This, combined with the observation that this problem mostly goes away when training is stopped, suggests that this happens due to overtraining. Another thing we can conclude is that the neural network seems to give better results than the Q-table when training is discontinued in a clever way, the opposite of what seems to be true when no such thing is implemented. Overall the simple Q-table approach seems to be sufficient for playing CartPole, and maybe even better than the more complicated and computationally heavier neural network approach.

IV. Snake

Prepared by the work done on the game CartPole we now take on the main task of this project—to find out how to set up and train a deep Q-learning algorithm for it to learn to play the game Snake well.

A. Construction of the game Snake

To be able to study and develop learning algorithms for Snake we need a version of the game suited for our purposes. For that reason we construct our own Snake game from scratch using Python 3. The rendering of the game, seen in Figures 7 and 8, is done using the package `gym` once again [13]. The game is written in a similar fashion to the CartPole game to be able to act as an environment in which an agent can learn.

The game Snake is made up of a hungry apple-loving snake moving around on a field with the goal of eating as many apples as possible. If the snake runs into a wall or itself, it dies and the game is over. Each apple eaten increases the score, but also makes the snake grow longer, thus increasing the difficulty of the game. The details of the game construction are given below. A 12×17 matrix is set up, with each element

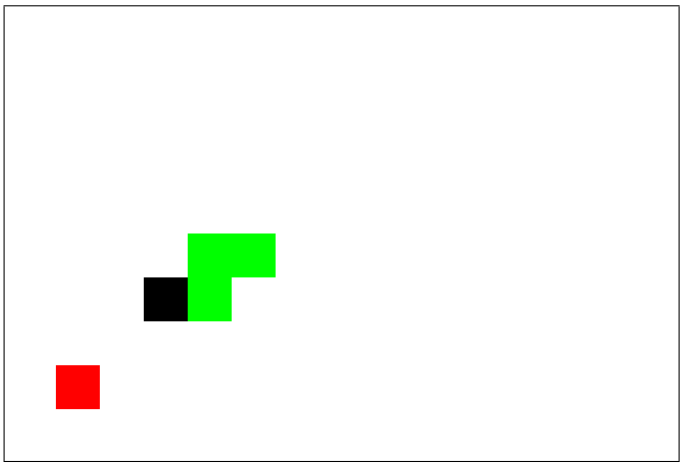

Fig. 7. The snake after eating one apple, i.e. at a score of 1 point.

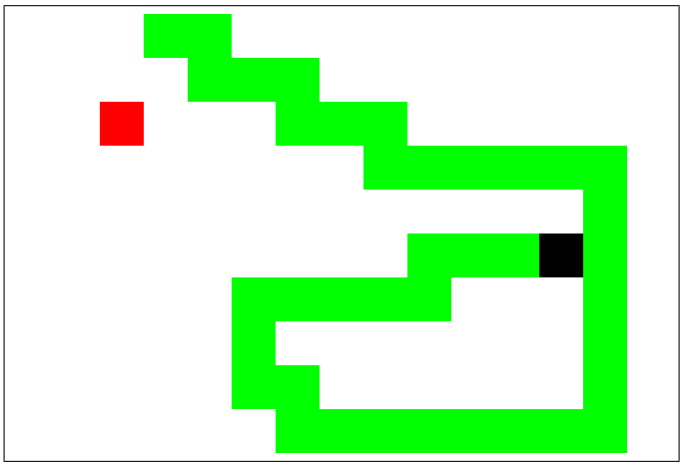

Fig. 8. The snake at a score of 36 points. In this situation it is vital for the agent to understand that turning upwards is the move to make—if it turns downwards it will have no way of surviving to eat another apple.

of the matrix representing a pixel in the game. The value of each element is set to one of five integers between 0 and 4 to represent whether the corresponding pixel is empty, contains the snake's head, contains a part of the snake's body, contains an apple, or contains a wall piece. The standard setup of the game has all border pixels set to be walls, giving the snake a 10×15 grid to move around on. This 10×15 grid is shown in two different stages of a game in Figures 7 and 8.

The game is initialised with one apple (represented by one pixel, shown in red in the figures), and a snake, facing right and consisting of three pixels placed horizontally: one head (shown in black) and two body pieces (shown in green) one and two pixels to the left of the head. The positions of the apple as well as the snake are random.

From this state the agent chooses an action from the current legal action space. Initially the action space consists of the three actions "move the head one pixel up", "move the head one pixel down", and "move the head one pixel to the right". When one of these actions are taken the snake is moved by updating the head's location according to the chosen action, replacing the pixel where the head was previously with a body piece pixel, and finally removing the body piece at the end of the snake (i.e. setting that pixel to be empty). By doing this it looks like the snake crawls according to the chosen action. This process is repeated, with the legal action space updated in between each move so that it always consists of three moves (it can never move the head "backwards", i.e. in the direction of the body piece connected to the head). When the head happens to move into an apple-pixel, four things happen: the score is increased by 1 point, the agent is given a reward r_{apple}, the end of the snake is *not* removed (which effectively means the snake's length is increased by 1 pixel), and a new apple is spawned at a random empty location. If the chosen action instead results in the head moving into a body or wall pixel the game ends, the agent is given a reward r_{dying}, and the final score is displayed. The rewards are set to $r_{\text{apple}} = 1,\ r_{\text{dying}} = 0$ unless otherwise stated.

B. Algorithm for the training of the agent

As the Snake game has way too many state-action combinations for a Q-table approach to be feasible[4], we use neural networks to build an agent able to learn the game. We study both regular DQN and DDQN as explained in Section II-C. Due to the poor performance of DQN for this problem the main focus is put on studying agents using DDQN. To create the agent we first initialise the target network and the online network identically with Keras' default initialisers. We vary the structure and parameters of the networks—the details of this are described later. With the networks initiated, Algorithm 1 is run.[5] We use $t = 4,\ T = 1{,}000\ B = 32$ unless otherwise stated. For the first 2,000 runs, the training step is skipped as there is not yet enough data stored to sample from and train on.

An ε-greedy policy, as explained in Section II-C, is used to choose the action $a^{(n)}$. We let ε decrease linearly from some $\varepsilon_{\text{start}}$ to some ε_{min} by setting $\varepsilon \leftarrow \max\{\varepsilon - \Delta\varepsilon, \varepsilon_{\text{min}}\}$ each iteration, for some $\Delta\varepsilon$. As default settings for these parameters we are using $\varepsilon_{\text{start}} = 1,\ \Delta\varepsilon = 2 \cdot 10^{-6},\ \varepsilon_{\text{min}} = 10^{-4}$.

The replay memory D is a list with a maximum length of 50,000 elements.[6] If the replay memory is full, the oldest element will be removed when a new one is added. The discount factor is $\gamma = 0.97$ unless otherwise stated. As a starting point for our code to implement the agent-learning algorithm we are using the code in the file `train.py` in the GitHub project by Pacheco [14].

[4]Already at snake length 3 there are roughly 150 apple positions $\times$ 149 head positions $\times$ 4 $\times$ 3 body positions $\approx$ 268,000 different states and at higher levels even more.

[5]One such run of two million iterations takes between 6 and 24 hours to complete, depending on the network and computer used.

[6]We use a structure called deque to do this efficiently.

for $n = 1$ *through* $2 \cdot 10^6$ **do**
  take action $a^{(n)}$ from action space;
  if *apple eaten* **then**
    increase score by 1;
  end
  save tuple of $s^{(n)}$, $a^{(n)}$, $r^{(n+1)}$, $s^{(n+1)}$ to replay memory D;
  if t *divides* n **then**
    sample a batch of B saved tuples from D;
    train the online network on this batch with target $r^{(n+1)} + \gamma \max_a Q_{\text{target}}(s^{(n+1)}, a)$;
  end
  if *action taken led to game ending* **then**
    reset game;
  end
  if T *divides* n **then**
    set the target network equal to the online network;
  end
end

Algorithm 1: training a network

C. General testing procedure

We will in the following sections explain the many different setups tested and studied. All of these tests consists of letting an algorithm train using some setup and evaluating the algorithm based on a `.csv` file containing data about this *training run*. For the setups found interesting to study further we run the algorithm again, this time without training and using the weights of the neural network at the point of the algorithm's highest-scoring training run. This is called the *evaluation run*. Just as in the training run a `.csv` file is generated with data about this evaluation run. From the scores at the end of each played game in a run, average and median scores are calculated. These calculations depends on the two parameters N_{avg} and N_{med}. The average score $k_{\text{avg,n}}$ at n played games is calculated as

$$k_{\text{avg,n}} = \frac{1}{N_{\text{avg}}} \sum_{i=n-N_{\text{avg}}+1}^{n} k_i, \tag{12}$$

where k_i is the score at the end of game i. In other words, the average is taken over the N_{avg} most recent games. The median score $k_{\text{med,n}}$ at n played games is calculated as

$$k_{\text{med,n}} = \text{median}\{k_{\text{avg,n+j}}\}_{j=-N_{\text{med}}/2}^{N_{\text{med}}/2-1}. \tag{13}$$

In other words, the median is taken over the scores at the end of the N_{med} games "closest in time" to game n.

D. Network structure

We study many different neural network configurations. All the networks have a fully connected layer with 4 nodes as the final layer, denoted $D(4)$, one for each action.[7] They all contain a flattening layer F as well. This converts the input of a matrix to a vector, which is necessary to do before the layer $D(4)$. The most basic network consists of a flattening layer F, two hidden layers in the form of fully connected layers $D(n_1), D(n_2)$, where n_1, n_2 are the number of nodes in the first and second hidden layer respectively, and finally the layer $D(4)$. This type of layer, with $n_1 = 128,\ n_2 = 64$, is represented as network 1 in Table I. A downside with this type of network is that it treats the pixels as individual, not as ordered in a two-dimensional grid. To get a network which treats the input as the two-dimensional data it actually is we use convolutional layers $C_S^K(m)$, where S is the stride length (same in both directions), K is the kernel size (in the sense that the kernel is $K \times K$), and m is the number of filters in the layer. Two such layers are used in networks 2–5 in Table I. For all fully connected and convolutional layers we use the activation function $\text{ReLU}(x) = \max\{x, 0\}$. An issue arising when settling with just these types of layers is that the number of connections between the layers grow very large, resulting in more time consuming computations. To avoid this a fourth type of layer is used—the max pooling layer M_S^K, where S, K mean the same thing as for the convolutional layer. Both convolutional and max pooling layers are explained Section II-B.

These layers can be combined in so many different ways, with different values for n, S, K, and m, and different amount of layers, that it, considered the limited extent of this project, is unrealistic to methodically go through all reasonable combinations to find the absolute best one. Instead we have tried some of the combinations to find a good one, even if it may not be the very optimal one.

E. Parameters

We now study the effects of varying different parameters involved in the agent. We are mainly focused on studying the effects of varying $\Delta\varepsilon, \varepsilon_{\min}$, and γ, but also briefly examine what happens when t and T (in Algorithm 1) are varied.
The values studied are $\Delta\varepsilon \in \{10^{-6}, 2 \cdot 10^{-6}, 10^{-5}\},\ \varepsilon_{\min} \in \{0.1, 0.01, 0.001, 0.0001\},\ \gamma \in \{0.9, 0.95, 0.97, 0.99\}$. For t we are setting $t = 1$ and $t = 4$, and for T we are using values of $T = 1,\ T = 1{,}000$, and $T = 10{,}000$. Note that setting $T = 1$ is precisely studying regular DQN instead of DDQN, as explained in IV-B. This choice of values to test is partly based on the values used in [12] and [14].

TABLE I
A SELECTION OF THE NETWORKS WE ARE TESTING. NOTATION AS IN SECTION IV-D.

NN 1	NN 2	NN 3	NN 4	NN 5	NN 6
	$C_1^4(32)$	$C_1^4(32)$	$C_1^4(64)$	$C_1^2(32)$	$C_1^4(32)$
	M_2^2	M_2^2	M_2^2	M_2^2	
	$C_1^2(128)$	$C_1^2(128)$	$C_1^2(64)$	$C_2^2(128)$	$C_1^2(128)$
	M_1^2	M_1^2	M_1^2	M_1^2	
F	F	F	F	F	F
$D(128)$	$D(256)$				
$D(64)$					
$D(4)$	$D(4)$	$D(4)$	$D(4)$	$D(4)$	$D(4)$

[7] D stands for *Dense,* which is the name of fully connected layers in Keras.

F. Environment

Finally we also study what happens when the environment is tweaked. This boils down to two different things being changed and the resulting effects analysed. Firstly, the rewards given to the agent are varied. The pairs of rewards (for eating apples and dying respectively) used are $(r_{\text{apple}}, r_{\text{dying}}) \in \{(1,0),(10,-1),(100,-1)\}$. Secondly, the map on which the snake can move around is modified to have a 2×2 block of wall-pixels in the middle. However, we only use this modification in the evaluation of already-trained agents; all agents always train on an obstacle-free map.

G. Variations on the training algorithm

Algorithm 1 is the algorithm for the agent that we use in every test, when something else is not explicitly stated. That being said, we are also studying four different variations on this algorithm.

1) : The first variation is to let the value for $\varepsilon_{\min}$ be higher (0.05 rather than in the range 0.0001–0.01) when the score is zero or one. This alteration is motivated by the fact that the snake is too short to run into itself when below a score of two, and thus more likely to get stuck in loops where it neither eats apples nor dies. Having a higher probability for the action to be chosen randomly can possibly solve this problem.

2) : Another variation also focused on the ways actions are chosen randomly or not in step 1 of the algorithm is the following one. In this alteration we do not use scalar values for $\varepsilon_{\text{start}}$, $\Delta\varepsilon$, and ε, but rather have these as vectors, with one position for each attainable score. This means that, for $\boldsymbol{\varepsilon} = (\varepsilon_0, \varepsilon_1, \ldots, \varepsilon_m)^T$, this vector is first initialised as $\boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}_{\text{start}}$, and then at each following iteration updated according to the equation

$$\boldsymbol{\varepsilon} = \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \vdots \\ \varepsilon_k \\ \vdots \\ \varepsilon_m \end{pmatrix} \leftarrow \begin{pmatrix} \varepsilon_0 \\ \varepsilon_1 \\ \vdots \\ \max\{\varepsilon_k - (\Delta\varepsilon)_k, (\varepsilon_{\min})_k\} \\ \vdots \\ \varepsilon_m \end{pmatrix} \tag{14}$$

when at some score k. The probability for an action to be chosen at random is then dependent on the score, so that the probability is ε_k when the score is k. We motivate this alteration by the fact that the game changes slightly as the snake grows longer. This alteration makes sure that exploration is done on each score, and could thus lead to the agent finding optimal strategies at each level, rather than developing a strategy that works well on low scores, but maybe not as the snake gets longer.

3) : A third variation consists of splitting the replay memory into two or three separate lists, each designated for some range of values of the current score, so that the tuple is saved in the first replay memory if the score is below some threshold value τ_1, in the second if the score is between τ_1 and some τ_2, and in the third replay memory (if there is one) otherwise. With these replay memories in place the sampling from the replay memory is done in the following way. A check is made to see what memories are eligible to be selected: these are the memories in which at least $32 \cdot 5 = 160$ tuples are saved. Then one of these eligible replay memories is selected at random, in such a way that each replay memory is four times as likely to be chosen as the previous one. The idea behind this variation to the algorithm is to make the agent train more on the difficult situations it can face when playing as a long snake once it has already mastered playing as a short snake.

4) : A fourth variation of the algorithm keeps track of how many times it has added a tuple to the replay memory, and does this separately for each value of the score, i.e. it counts how many tuples it has added while at score 0, while at score 1, while at score 2, and so on. After reaching some threshold number of added tuples for some score, it no longer adds tuples to the replay memory while at that score. The idea is, just as in the previous alteration, to train no more than sufficient on low levels, and focus the computational power to train in what could be a more efficient way.

H. Default settings

In the tests that we have explained in the preceding Sections IV-D–IV-G we are always varying just one parameter at a time. Table II shows the values we use as default for the different parameters. For instance, when we study the effect of varying $\Delta\varepsilon$, all other parameters are set to the value of Table II.

TABLE II
DEFAULT PARAMETER VALUES AND SETUPS. THE NETWORK VALUE REFERS TO THE NUMBERING IN TABLE I.

Parameter	Default value
$\varepsilon_{\text{start}}$	1
$\varepsilon_{\min}$	0.001
$\Delta\varepsilon$	$2 \cdot 10^{-6}$
γ	0.97
Network	2
$(r_{\text{apple}}, r_{\text{dying}})$	(1,0)
Replay memory size	50,000
B	32
T	1,000
t	4

I. Finding the optimal settings

After testing parameters individually we classify what value or configuration gave the best result, for each of the tested parameters. We then run an agent using this set of presumably good parameter values and compare its performance to the performance of an agent using the default parameters.

V. RESULTS AND DISCUSSION

The results from testing different configurations of the DDQN are shown in Figures 9–21. In all of the Figures in this section the bright-green lines and dots show the performance of an agent using the default settings presented in Table II, and are thus the same throughout all of the figures in this section. It is worth noting that the x-axis shows the number of moves, or steps, taken by the agent, while the data points

each correspond to one played game. This is done as the number of steps taken is directly proportional to the training done;[8] as games vary in length, it is not true for the number of played games. Due to time constraints, each test was only run one time and no confidence intervals or error measures were made for the learning curves. However our experience from having to restart tests is that the variances are low. This is also supported by what we present in Section V-E, which in essence is that we find an improved algorithm by combining what these single runs say are the best parameter values.

A. Neural Networks

In Figure 9 we present the results from testing the performance of agents using the different neural networks in Table I. Figure 10 compares the same agents once again, this time playing evaluation games according to the procedure presented in IV-C. The first thing that is clear from these figures is that the agent using network 1 barely learns anything at all. What it does learn is to move around in a square of four adjacent pixels seemingly indefinitely. While this strategy lets the snake live forever, it does not lead to any apples being eaten and as a result the score is never increased. This indefinite looping is indirectly seen in Figure 10 as there is only one red point (in the lower left corner). The second game did not finish before the 10^5 evaluation steps were over. As the main difference between this network and the five others is the lack of convolutional layers, it clearly seems like the convolutional networks are vital for the network to learn. This is expected as a network without these disregards the structure of the image-matrix presented to the network, and without this structure the game does not make any sense.

The difference between the performance of other five networks is small. From Figure 9 we see that network 4 learns to reach a median score of 25 slightly quicker than the rest. Network 2 has the highest score at the end of the two million steps taken in training mode, but takes much longer to run—the fully connected layer that separates it from networks 3–6 leads to weight files roughly 32 times bigger and the training steps taking more than four times as long to perform. By comparing the training runs of networks 3 and 5 we see that the impact of changing the stride length of the convolutional layers is small. If anything the default settings used by network 3 are slightly better than those of network 5. The effects of having max pooling layers can be seen by comparing networks 3 and 6. As with the stride lengths the effects are small, but it seems that using max pooling layers gives better results. In Figure 10 we see that networks 2 and 3 seem to be performing the best during evaluation, while networks 4, 5, and 6 attain similar, slightly lower scores.

With these things in mind we argue that network 4 is the best one to use. It reaches high scores quicker than network 3 while running significantly faster than the slightly-higher-scoring network 2. That being said, it does not perform as well during evaluation, but the difference is small, and could maybe be explained by the fact that network 3 reached its high score much earlier during training than network 4. This means

[8]The algorithms train on 32 tuples between each step.

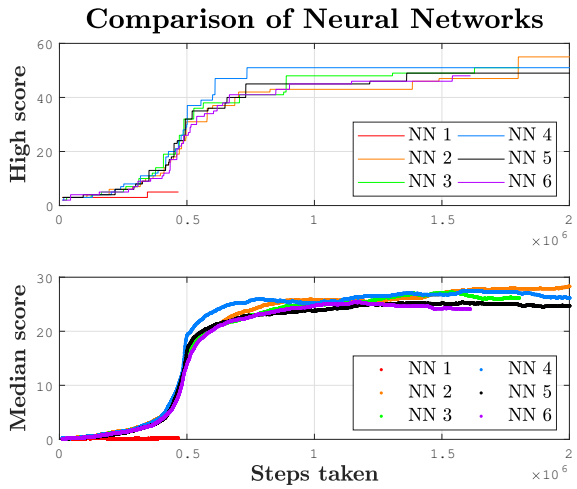

Fig. 9. Score progression when training using different types of neural networks (NN). The numbers refer to the networks listed in Table I. The median score is calculated by using $N_{\text{med}} = 1{,}000$ and $N_{\text{avg}} = 10$ in (12) and (13).

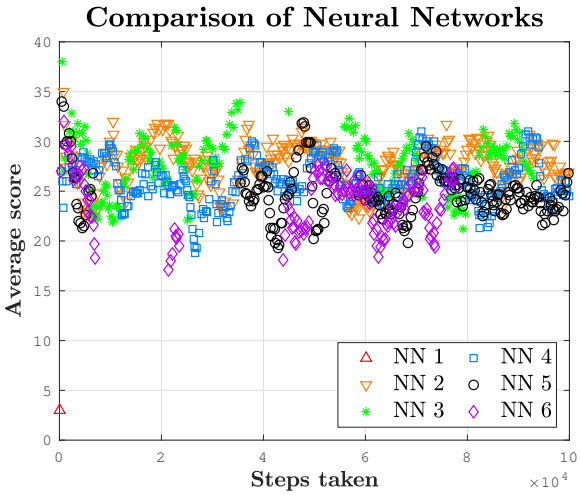

Fig. 10. Scores from evaluation games with agents trained using different types of neural networks (NN). The numbers refer to the networks listed in Table I. The average score is calculated by using $N_{\text{avg}} = 10$ in (12).

that network 3 evaluates with the weights and biases it had after training during approximately $0.7 \cdot 10^6$ steps, while the evaluation of network 4 is done with the weights and biases it had after about $1.6 \cdot 10^6$ steps, as can be seen in the upper half of Figure 9. Maybe this does not fully explain the difference in evaluation performance, but we mean that the faster learning of network 4 still outweighs the slightly worse evaluation run.

B. Parameters

We are comparing the performance of agents using different values of the parameter $\Delta\varepsilon$ in Figures 11 and 12. In Figure 11 the agents are compared while training, and in Figure 12 their evaluation runs are compared.

It is clear that the $\Delta\varepsilon$ affects the number of training steps needed to reach a median score of 20. By taking a closer look

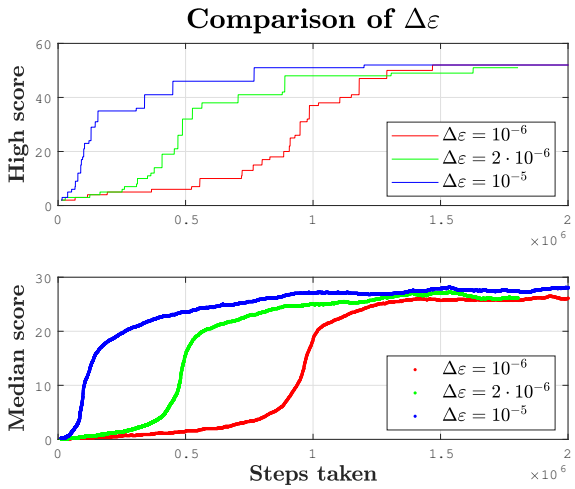

Fig. 11. Score progression when training using different values for $\Delta\varepsilon$. The median score is calculated by using $N_{\text{med}} = 1{,}000$ and $N_{\text{avg}} = 10$ in (12) and (13).

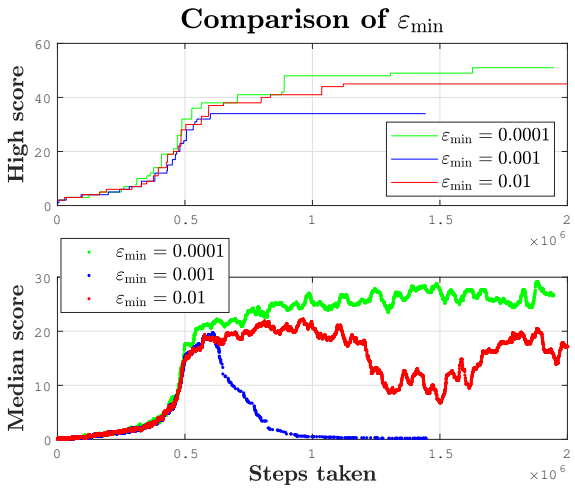

Fig. 13. Score progression when training using different lower bounds on ε, i.e. different $\varepsilon_{\min}$. The median score is calculated by using $N_{\text{med}} = 100$ and $N_{\text{avg}} = 10$ in (12) and (13).

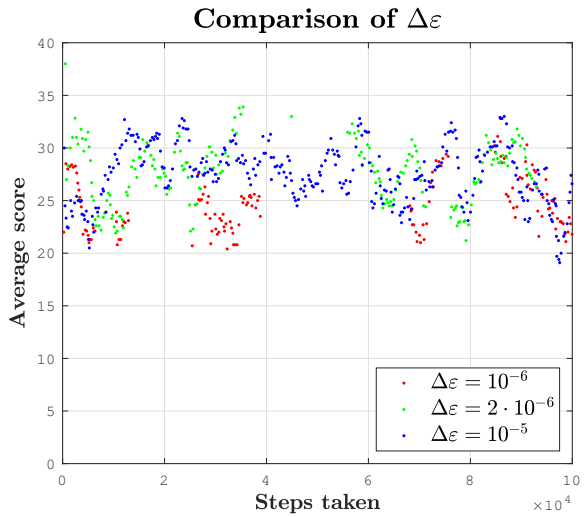

Fig. 12. Scores from evaluation games with agents trained using different values for $\Delta\varepsilon$. The average score is calculated by using $N_{\text{avg}} = 10$ in (12).

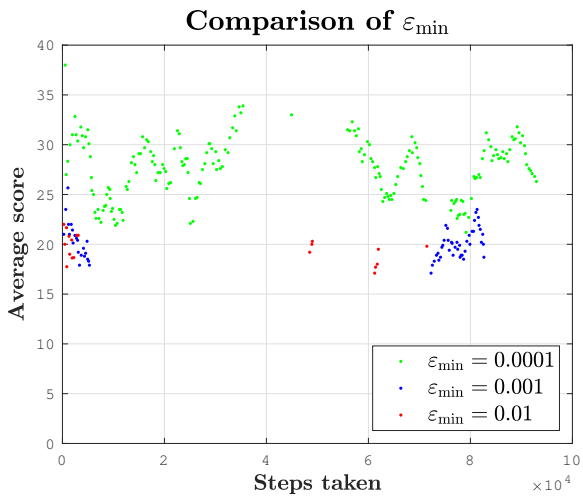

Fig. 14. Scores from evaluation games with agents trained using different lower bounds on ε, i.e. different $\varepsilon_{\min}$. The average score is calculated by using $N_{\text{avg}} = 10$ in (12).

at Figure 11 one can notice that the growth of the median score is very high for a short period around the time where the value of ε has decreased to $\varepsilon_{\min}$. However, after two million steps this difference has decreased to the point that they all perform very similarly. In Figure 12 we can see that there are "holes" in the green and red data sets. This is the result of the snake getting stuck in loops—this leads to some very long (and often low-scoring) games, which creates a large gap on the x-axis between two succeeding games.

From this we conclude that a value of $\Delta\varepsilon = 10^{-5}$ is preferable, as it yields high scores quicker without performing worse later on, and at the same time seems to avoid getting stuck in loops.

In Figures 13 and 14 the shown comparisons are done in the same way as above, with the only difference being that we are keeping $\Delta\varepsilon$ fixed and instead vary $\varepsilon_{\min}$. We see that changing $\varepsilon_{\min}$ seem to have great impact on the performance. The learning is identical up to the point where $\varepsilon = \varepsilon_{\min}$, that is around 500,000 steps in to training. From here on the agents using $\varepsilon_{\min} \in \{0.01, 0.001\}$ starts getting stuck in loops, leading to very long and low-scoring games, as seen in Figure 14. This leads to heavily decreasing scores, especially for the agent using $\varepsilon_{\min} = 0.01$. In the case $\varepsilon_{\min} = 0.001$ the median score never drops down to zero and increases again after about 1.5 million steps. It is clear that $\varepsilon_{\min} = 0.0001$ is the best value of those presented in Figures 13 and 14. As mentioned in Section IV-E we also did (preliminary) tests with $\varepsilon_{\min} = 0.1$. This did not seem to lead to any high scores and was therefore not studied more carefully. This lack of high scores could be because the algorithm is adapting to its own tendency to sometimes select game ending moves, in the sense that it learns to avoid dangerous states from which it could run into a wall or itself. To reach high scores it is often necessary for the algorithm to move into more dangerous states in order to get

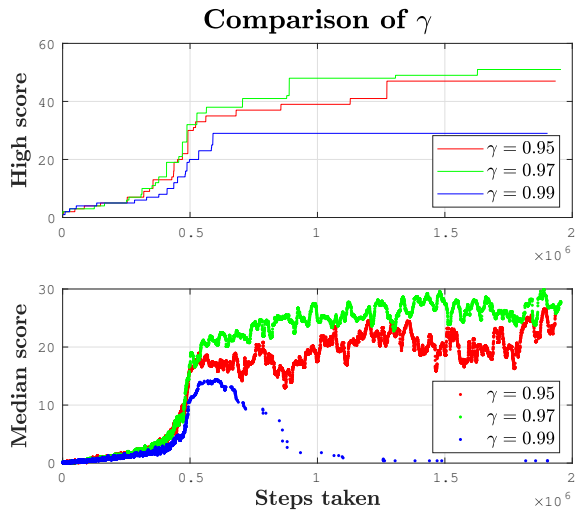

Fig. 15. Score progression when training using different values for the parameter γ. The median score is calculated by using $N_{\text{med}} = 50$ and $N_{\text{avg}} = 10$ in (12) and (13).

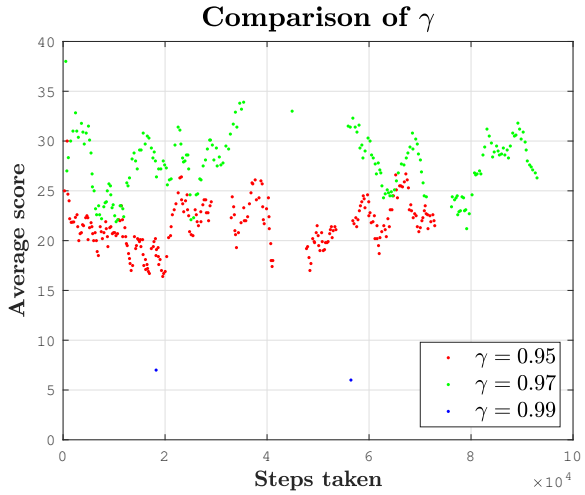

Fig. 16. Scores from evaluation games with agents trained using different values for the parameter γ. The average score is calculated by using $N_{\text{avg}} = 10$ in (12).

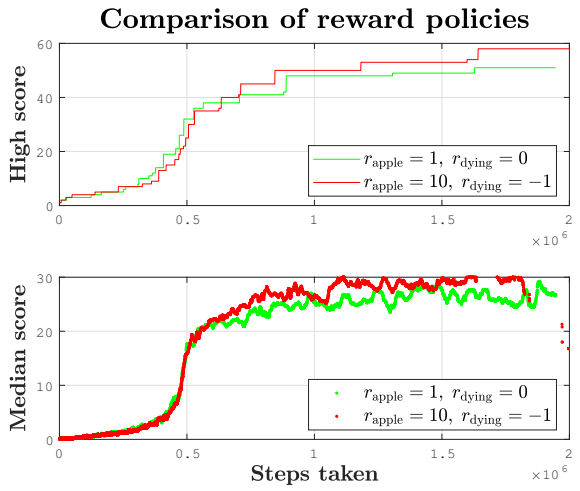

Fig. 17. Scores when training using different policies for giving rewards to the agent. The median score is calculated by using $N_{\text{med}} = 100$ and $N_{\text{avg}} = 10$ in (12) and (13).

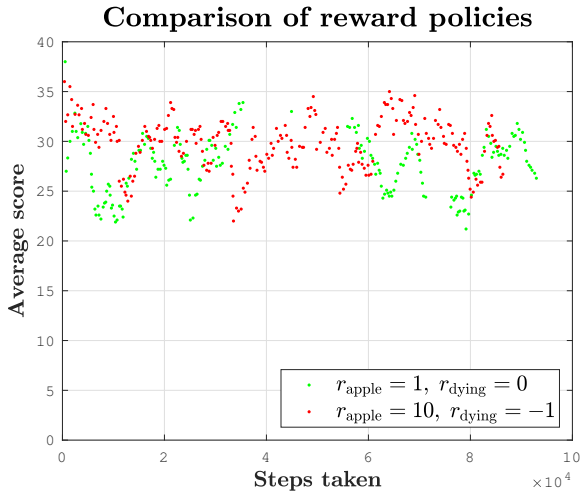

Fig. 18. Scores from evaluation games with agents trained using different policies for giving rewards to the agent. The average score is calculated by using $N_{\text{avg}} = 10$ in (12).

to the apples, which would explain why $\varepsilon_{\min} = 0.1$ gives poor results.

In Figures 15 and 16 we are comparing agents using different values for γ. They are compared both while training (Figure 15) and while evaluating (Figure 16). The results indicate that the value of γ has a big impact on performance. We see that a value of $\gamma = 0.99$ leads to the snake getting stuck in loops very frequently, as seen by the median score falling of in Figure 15 and the two lone blue dots in Figure 16. Using $\gamma = 0.95$ seems to lead to slower progression after the first 500,000 steps compared to $\gamma = 0.97$, and this difference is seen once again in the evaluation games presented Figure 16. Outside of the values of γ presented in these figures we also did tests with $\gamma = 0.9$, but as these tests indicated poor performance for this value no further tests were done. We can confidently choose $\gamma = 0.97$ as the best value of γ of those tested. The fact that the results from the different γ-values are so different even though the values themselves are close together leads us to think that a study of more values of $\gamma \in (0.95, 0.99)$ could lead to even better performance than that of $\gamma = 0.97$.

C. Comparison of different reward policies

A comparison between the performance of two different reward policies, i.e. two different pairs of values of the rewards $r_{\text{apple}}, r_{\text{dying}}$, is shown in Figures 17 and 18. In the first of these figures the learning is compared; in the second their evaluation runs are compared, with the weights from the end of their respective highest-scoring training games. From these test there is no clear difference between the two pairs of values, although $(r_{\text{apple}}, r_{\text{dying}}) = (10, -1)$ seems to give slightly better performance. With that being said, the score for this pair of values drops at the end of the

training run. Altogether this parameter seems to be of minor importance to the overall performance, but our results indicate $(r_{\text{apple}}, r_{\text{dying}}) = (10, -1)$ to be the slightly better choice. The tests done with $(r_{\text{apple}}, r_{\text{dying}}) = (100, -1)$ mentioned in Section IV-F seemed not to lead to to any noticeable difference, but as we did not study this more thoroughly it is nothing we can conclude with certainty.

D. Other algorithms

The four alterations to the main algorithm, presented in Section IV-G, were all tested but appeared to give no or little improvement.

The first variation, which uses a higher ε_{min} while at scores 0 and 1, seems to reduce the problem of the snake getting stuck in loops, but as there already are ways to tune the existing parameters of the network so that this problem is avoided this seems to be an unnecessary addition to the algorithm.

The second alteration uses different ε-values while at different scores. The effect of this were not clear from our tests, but it did not seem to improve the algorithm. To get any conclusive results about this alteration more tests are needed.

The third variation seemed to mostly worsen the results. The first shorter trial runs with this algorithm were promising, but over longer runs the scores never got as high as without using this variation. It seems as if almost only training on steps taken while the score was high leads the algorithm to forget how to play when the snake is short without really playing better when the snake is long.

The fourth alteration to the initial algorithm gave bad results, but was only tested briefly.[9] A careful tuning of all threshold numbers could maybe give different results, but as we did not have any source on that this really could be fruitful we did not study this alteration further.

Lastly, setting $T = 1$, i.e. effectively using regular DQN without the separate target network, resulted in bad performance. Similarly with $t = 1$, which amounts to training between every step, the agent never scored as high as with the default value $t = 4$.

Altogether the first variation could be one to consider using, and it could possibly be worth to study the second one further. The third variation seems to be of little use. The fourth variation did not seem promising but was studied too briefly for us to be able to draw any conclusions.

E. Optimal settings

For each parameter and setting tested, the best performing one has been presented in Sections V-A–V-C. All these best performing parameters and settings are presented together in Table III. A comparison between an agent using the parameters in Table III and an agent using the default parameters (Table II) is shown in Figures 19 and 20. The agent using the seemingly better settings, referred to in Figures 19 and 20 as Agent B, is seen to achieve higher scores throughout training as well as evaluation. The high score of 66 points is the highest score in a single game seen in any of our tests. These results indicate strongly that this really is the most well-performing setup of those tried in this project. That being said, there is likely some tweaks that can be done to the most sensitive parameters, γ and ε_{min}, to improve this score. Such an improvement would probably not, however, be of more than a few points, based on what we have seen throughout our tests. To really improve

[9]With bad results we mean that the scores were significantly lower than the scores set with our default settings.

TABLE III
THE BEST SETTINGS OF PARAMETERS WHEN TWEAKED INDIVIDUALLY. THE NETWORK NUMBER REFERS TO TABLE I.

Parameter	Value
Network	4
$\Delta\varepsilon$	10^{-5}
ε_{min}	0.0001
γ	0.97
$(r_{\text{apple}}, r_{\text{dying}})$	(10,-1)

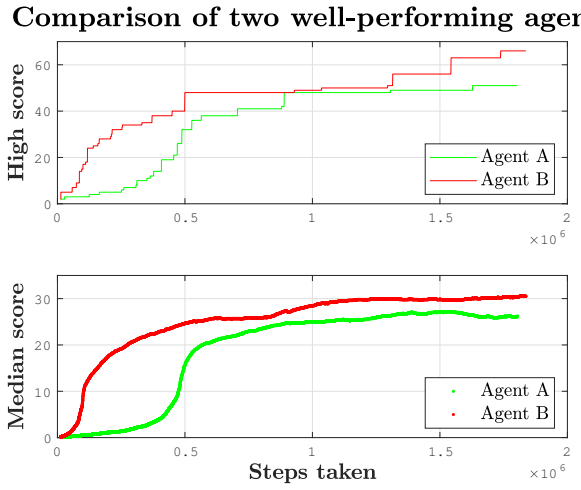

Fig. 19. Score progression when training using default settings (agent A) and using the settings presented in Table III (agent B). The median score is calculated by using $N_{\text{med}} = 1{,}000$ and $N_{\text{avg}} = 10$ in (12) and (13).

Fig. 20. Scores from evaluation games using default settings (agent A) and using the settings presented in Table III (agent B). The average score is calculated by using $N_{\text{avg}} = 10$ in (12).

the performance some other approach is most likely needed.

F. Performance with added obstacle

Figure 21 presents how an agent, trained with the settings of Table III, performs when an obstacle in the form of a 2×2 block of wall pieces is added to the middle of the map, compared to how it performs without this obstacle. We see that the score is only reduced by a few points, and the highest scores set are about the same in both the obstacle and the obstacle-free cases. It is remarkable that the agent can perform so well even though it was trained in an environment where going through the middle of the map would not end the game. This ability to perform in a new environment indicates the power of reinforcement learning.

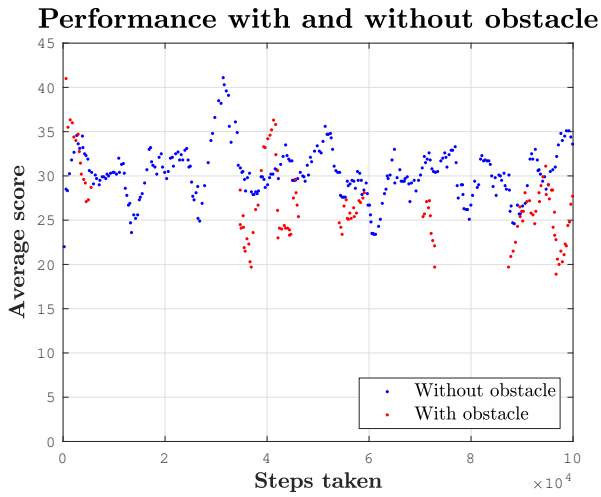

Fig. 21. Scores when testing the same agent on different environments. The average score is calculated by using $N_{\text{avg}} = 10$ in (12).

VI. Conclusions

The most efficient way to train an agent to play the game Snake, out of all the approaches we have tested, is to use the double deep Q-network presented in Algorithm 1 with the parameters of Table III. The neural network of this algorithm consists of two convolutional layers, two max pooling layers, a flattening layer, and one fully connected layer. This network is presented in detail in Table I as network 4. It is worth emphasising that this is the best approach out of the ones we have tried—it is certainly possible to find even better approaches.

An agent using our best setup learns to consistently attain average scores of 20 points after training on roughly eight million state-action combinations and 30 points after about training on 34 million state-action combinations.[10] After training on 56 million state-action pairs, which took slightly more than 5 hours, it manages to set a high score of 66 points, which amounts to covering 46% of the available space. For comparison, if a human would play through as many moves of snake as the agent did to reach 66 points, assuming five moves per second, it would take just over four entire days of constant play.

[10]As the algorithm trains 32 times for each step that it takes in game, these values differ from the values on the x-axes of all figures in Section V by a factor of 32.

A. Future work

As mentioned is Section V, no error or variance measures were made during the testing. Given more time, this would be the first thing to improve upon, maybe to give some statistical significance of the results.

Further, the analysis in this project compares algorithms mainly by their performance as function of the number of training steps. An interesting investigation would be to measure time efficiency of the different algorithms. Some networks require more computation power per training step than others. Thereby the most training step efficient setup might not be the most time efficient. Comparing time efficiency (on the same hardware) would give a fairer measure of best algorithm.

It could also be of interest to study the effects of varying γ more thoroughly, as we found the performance to be sensitive to this parameter. Doing so could give an even better Snake-playing algorithm.

Acknowledgment

We want to thank our supervisors Erik Berglund and Inês Lourenço for their continuous interest in our work and insightful comments throughout the time of the project. We could not have wished for better cooperation.

Special thanks to all students in computer lab Röd for showing patience with us occupying ten computers at a time during our testing.

References

[1] DeepMind. Solve intelligence. Use it to make the world a better place. DeepMind Technologies Limited. London, United Kingdom. (April, 2019). [Online]. Available: https://deepmind.com/about/

[2] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, February 2015. [Online]. Available: https://www.nature.com/articles/nature14236

[3] E. Newton-Rex. The state of AI. (April, 2019). [Online]. Available: https://medium.com/on-coding/the-state-of-ai-9aae385c2038

[4] S. Shanmuganathan, "Artificial Neural Network Modelling: An Introduction," in *Artificial Neural Network Modelling*, 1st ed. Basel, Switzerland: Springer Intl., 2016, ch. 1, pp. 1–15.

[5] R. S. Sutton, *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: MIT Press, 1998.

[6] M. L. Littman, "Markov decision processes," in *International Encyclopedia of the Social & Behavioral Sciences*. Amsterdam, Netherlands: Elsevier Ltd., 2015, pp. 573–575.

[7] D. C. Montgomery, E. A. Peck, and G. G. Vining, *Introduction to Linear Regression Analysis*. Hoboken, New Jersey: A John Wiley Sons, INC., 2012.

[8] F. S. Melo. Convergence of Q-learning: a simple proof. Instituto Superior Tecnico, Institute for Systems and Robotics. Lisbon, Portugal. (April, 2019). [Online]. Available: http://users.isr.ist.utl.pt/~mtjspaan/readingGroup/ProofQlearning.pdf

[9] L. Vazquez. Understanding Q-Learning, the Cliff Walking problem. (April, 2019). [Online]. Available: https://medium.com/init27-labs/understanding-q-learning-the-cliff-walking-problem-80198921abbc

[10] P. J. Werbos, "Backpropagation Through Time: What It Does and How to Do It," *Proc. IEEE*, vol. 78, 1990.

[11] H. Habibi Aghdam and E. Jahani Heravi, *Guide to Convolutional Neural Networks A Practical Application to Traffic-Sign Detection and Classification*, 1st ed. Basel, Switzerland: Springer Intl., 2017.
[12] H. van Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-learning," *CoRR*, vol. abs/1509.06461, 2015. [Online]. Available: http://arxiv.org/abs/1509.06461
[13] OpenAI gym. (April, 2019). [Online]. Available: https://gym.openai.com/
[14] M. Pacheco. Snake through deep reinforcement learning. (April, 2019). [Online]. Available: https://github.com/michael-pacheco/snake-deep-q-learning