Department of physics, University of Oslo

FYS-STK4155 – Applied data analysis and machine learning
Project 3 - A game of snake using reinforced and supervised learning

Jon A Ottesen

(Dated: 19. desember 2019)

The main aim of this project is to study the differences between reinforced (Q-learning) and supervised machine learning. The differences will be studied using a convolutional neural network architecture and comparing the performance of the reinforced and supervised networks when playing a simple game of snake, treating the board as a 2D image. It can be seen the best performing reinforced algorithm with CNN scored an average of 8 points with the high score being 33, whereas the supervised algorithm almost always scored either a perfect score or didn't score any points at all.

I. INTRODUCTION

Machine learning is often divided into multiple sub categories, each with its respective advantages and disadvantages. The central methods in this project will be supervised and reinforce machine learning. Supervised machine learning includes multiple types of classification and regression methods. One such example would be the use of neural network in medical classification. Reinforced learning on the other hand has different applicability, such as the very famous artificial intelligence for e.g. machinery. In this project the categories supervised and reinforced learning will be directly compared in a simple game of snake treated as a classification case with down, right, up and left as the possible solutions.

The method used for classification will be convolutional neural network where the game of snake is treated as is, which is an image of a snake head, snake tail, apple and walls. To compare supervised and reinforced learning, the learning regiment for the network will be different. The supervised method will use a training set to minimize an error function whereas reinforced learning will apply Q-learning (more about that later).

II. THEORY

A. Game of snake

Before starting on any real theory, some underlying knowledge about how a game of snake can be treated is necessary.

A game of snake can be thought of as a simple matrix such as:

$$\boldsymbol{X} = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 2 & 2 & 0 & 0 \\ 0 & 0 & 0 & 2 & 2 & 0 \\ 0 & 0 & 0 & 0 & 2 & 2 \end{bmatrix} \tag{II.1}$$

where -1 represents the apple, 0 represents blank spaces, 1 the snake head and 2 the snake tail. This board can be generalized to a size of $\boldsymbol{X} \in \mathbb{R}^{n \times m}$ or in my case and the one used in the latter theoretical statements $\boldsymbol{X} \in \mathbb{R}^{n \times n}$ where n is an even number. The reason for this is to avoid any unnecessary complications.

B. Hamiltonian circuit

A Hamiltonian circuit is a path that visits every vertex exactly once within any cycle, and where the endpoint of that path is adjacent to the

starting vertex such that they may connect. For a 6x6 matrix such as the one in equation II.1, a Hamiltonian circuit may be as followed

$$\boldsymbol{A} = \begin{bmatrix} 0 & 35 & 34 & 33 & 32 & 31 \\ 1 & 10 & 11 & 20 & 21 & 30 \\ 2 & 9 & 12 & 19 & 22 & 29 \\ 3 & 8 & 13 & 18 & 23 & 28 \\ 4 & 7 & 14 & 17 & 24 & 27 \\ 5 & 6 & 15 & 16 & 25 & 26 \end{bmatrix} \tag{II.2}$$

with the starting tile being 0. This general pattern can be repeated for all matrices with an even number of columns, for both even and odd number of rows.

It can be seen that the transpose of $\boldsymbol{A}$ keeps the Hamiltonian circuit intact and thus: *all matrices with* $\boldsymbol{A} \in \mathbb{R}^{n \times m}$ *with n or m as an even number has a Hamiltonian circuit.* Note however, I have not included any mathematical proofs on the matter, and what is previously stated is only based on observation.

An important consequence of the Hamiltonian circuit in a game of snake is that it will inevitably lead to a perfect game of snake.

1. *Pertubated Hamiltonian cycle*

The entire concept behind this idea is credited to John Tapsell[4].

A snake algorithm following a specific designated path is very boring. A method to solve this problem is to allow the snake to take shortcuts. A shortcut is considered a skip in the Hamiltonian circuit such that the next position is closer than the original path to the apple, an example of a shortcut would be to skip from position 10 to 1 in matrix II.2 if the apple was at 3. However, such a shortcut follows some specific rules to ensure survival and simplicity:

1. A shortcut can only be done with regards to the snakes current position and the current adjacent tiles. It does not take into consideration further possible positions to ensure the minimum distance.

2. The next snake head position can **never** be higher than that of the snake tail. Said another way; the next position can never be in the interval $[r_{tail}, r_{head}]$.

3. The snake only takes a shortcut if the next position is closer to the apple, as long as rule 1 and 2 are not violated.

The second rule need further explaining, and to do this the matrix in equation II.2 is needed. Assume that the head is positioned at 5, and the tail is 5 long such that the snake is positioned at 5, 4, 3, 2, 1 and 0. The apple is than positioned at 26. The shortcut taken would than be 6 - 15 - 16 - 25 - 26. Now the snake tail would be positioned at 4, thus any shortcuts involving tiles within $[4, 26]$ is forbidden.

C. Neural Network

Before starting on the theoretical parts about neural networks (NN) and convolutional neural network (CNN) some terminology is required.

Learning rate: The learning rate is factor of which the network updates the previous weights and biases based on the error of the training data. This is a hyperparameter meant for adjustments.

Epochs: The total number of times the network trains on the entirety of the training data and can be thought of as a training cycle with length of the number of data points. In stochastic gradient descent, training data may be repeated but the length of the cycle remains constant.

Episode: An episode can be compared to that of an epoch, but unlike an epoch the length of en episode does not remain constant. In a game of snake, an episode would be a single game, but the length of a single game will vary depending on the network performance. An episode is thus the number of games played and trained on.

1. 1D Neural network

Before giving a detailed explanation on convolution neural network, some background theory of regular 1D neural networks are needed. However, I will not flesh out the mathematical details involved in feed forward or back propagation. The mathematical process behind a regular feed forward neural network is explained in [3][1]. Although the mathematical knowledge is not essential, I will assume that the process behind a basic feed forward neural network is known.

Neural network takes a 1 dimensional input, and through a continuous mathematical mapping weighing of the inputs, a output is given. This mapping happens through a chosen number of hidden layers, with their respective activation function and number of neurons.

D. Convolutional neural network

The theoretical aspects in this subsection is based of [2].

Unlike a 1-dimensional neural networks that assumes that the input is an array of information, a CNN assumes that the input is an image. This allows multiple properties to be encoded into the architecture of the method itself. This ensures a high degree of efficiency by reducing the number of parameters. A concrete example would be an image of size $256 \times 256 \times 3$ (last dimension is the RGB color scheme). For a regular NN, this would require a total of

$$196608 \times n \tag{II.3}$$

weights for the first hidden layer alone, where n is the number of hidden neurons. In a CNN this number is vastly reduced

[1] After having read through my own project 2, I must admit, the English need some. Hopefully it will be better this time around.

We will begin by considering a simple image

$$\boldsymbol{J} \in \mathbb{R}^{h \times w \times i} \tag{II.4}$$

or for simplicity $\boldsymbol{J} \in \mathbb{R}^{h \times w \times 3}$ with the first two indexes representing the height and width respectively and the last index representing the RGB color scheme.

The next issue at hand will be to introduce some mathematical terminology. The *grand sum* is the sum of all elements in a matrix, and for a two dimensional matrix this can be written as

$$\mathbb{G}(\boldsymbol{C}) = \boldsymbol{e}_{\boldsymbol{n_r},\boldsymbol{1}}^{T} \boldsymbol{C} \boldsymbol{e}_{\boldsymbol{n_c},\boldsymbol{1}} \tag{II.5}$$

with C as some arbitrary matrix $\boldsymbol{C} \in \mathbb{R}^{n_r \times n_c}$ and $\boldsymbol{e}_{\boldsymbol{n_r},\boldsymbol{1}}, \boldsymbol{e}_{\boldsymbol{n_c},\boldsymbol{1}}$ as two column vectors with length n_r and n_c respectively filled with ones. For tensors of higher dimension, the *grand sum* sums over all elements in all dimensions.

1. Convolution layer

A convolutional layer consist of many kernels (often referred to as convolutional filters). A single kernel for the input image takes the following shape $\boldsymbol{K} \in \mathbb{R}^{g \times k \times 3}$ where g and k is less than or equal to the height and width of the input image. A generalized convolutional layer with all kernels will have the following shape

$$\boldsymbol{K} \in \mathbb{R}^{n \times g \times k \times i} \tag{II.6}$$

with n is the number of kernels, g is the height, k is the width and i being the depth. The depth i of a single kernel must be equal to the depth of the previous layer. The kernel layer can be thought of as a matrix representation of the weights in a regular 1D NN, and are the variables that will be updated. Unlike regular feed forward NN, the feed forward process in a CNN is non-linear.

The mathematical formalism behind the forward process in a convolutional layer can be presented using matrix notation. It is worth noting that I am using the input image $\boldsymbol{J}$ with an unspecified depth i to preserve generality. The

mapping process through a single kernel onto a matrix can be written as

$$\boldsymbol{M}_{t,m} = \mathbb{G}\left(\boldsymbol{X}^{(l-1)}_{t:t+g,m:m+k} \circ \boldsymbol{K}_p\right) \tag{II.7}$$

where $\circ$ is the hadamard multiplication and the subscript represents a submatrix in $\boldsymbol{X}$. The submatrix is the matrix spanned from the height t to t+g including the last index and width m to m+k including the last index, and the entire depth of the input $\boldsymbol{X}^{(l-1)}$ is included. Remember that the grand sum also sums up in the depth dimension. However, the result is only the matrix for a specific depth n in the convolutional layer.

For a python implementation, the indexing would be written as $t : t+g+1$ and $m : m+k+1$. Equation II.7 restricts the kernel size such that $g \leq h$ and $k \leq w$. Furthermore, a common troupe is to enforce g and k to be odd numbers, however this is not required.

The result from equation II.7 is not the output of the convolutional layer at a specifc depth. The output of the convolutional layer for a specifc depth is given by

$$X^{(l)}_n = f_a(\boldsymbol{M}) \tag{II.8}$$

where f_a is the activation function for that layer, often being the Relu activation function given by $max(x, 0)$. The matrix for a given depth in the next layer is called a feature map, since the entire process is simply mapping a image through a weighting matrix called a kernel. Combining equation II.7 and II.8 with the general kernel tensor in II.6, the feed forward process in a convolutional layer is given by

$$\boldsymbol{X}^{(l)}_{t,m,n} = f_a\left(\mathbb{G}\left(\boldsymbol{X}^{(l-1)}_{t-s:t+s+1,m-b:m+b+1} \circ \boldsymbol{K}_n\right) + \boldsymbol{b}^{(l)}\right) \tag{II.9}$$

and the dimensions of $\boldsymbol{X}^{(l-1)}$ is the same as II.4. While the dimensions of $\boldsymbol{X}^{(l)} \in \mathbb{R}^{h-g+1 \times w-k+1 \times n}$, such that new depth is given by the number of kernels n. The $\boldsymbol{b}^{(l)}$ term added is the bias term and is an additive term with the shape $\boldsymbol{b}^{(l)} \in \mathbb{R}^{1\times1\times n}$, such that the additive term differs depending on the depth.

Before ending I wanna make it explicitly clear that it's the terms in the kernels in $\boldsymbol{K}$ and the biases $\boldsymbol{b}^{(l)}$ that are updated with gradient descent methods. Furthermore, the kernels and biases differs depending one the layer, as with the weights in a NN.

As a last note on the topic of a convolutional layer, I want to showcase the explicit advantage of using kernels instead of the NN approach of weighing each element in the input. The number of element wise calculations in a convolutional layer is given by(excluding the activation function and additive operations)

$$N_{CNN} = g \cdot k \cdot i \cdot (h - g + 1)\,(w - k + 1) \cdot n \tag{II.10}$$

i.e the number of kernels multiplied with the elements in the kernel multiplied with the shape of the output matrix. The number of calculations in a regular NN is given by

$$N_{NN} = n_n \cdot h \cdot w \cdot i \tag{II.11}$$

where n_n is the number of hidden nodes. The ratio is therefore given by

$$R^{CNN}_{NN} = \frac{n \cdot g \cdot k \cdot (h - g + 1)\,(w - k + 1)}{n_n \cdot h \cdot w}. \tag{II.12}$$

The number of nodes n_n in the NN is for simplicity enforced to equal the number of elements in the output tensor (the output after the kernel calculation). This implies

$$R^{CNN}_{NN} = \frac{g \cdot k}{h \cdot w} \tag{II.13}$$

and for regular kernel size choices such as 3x3 and 5x5 in a 256x256 image results in a ratio of $1.3 \cdot 10^{-4}$ for a 3x3 and $3.8 \cdot 10^{-4}$ for a 5x5 kernel.

2. *Pooling layer*

The next layer type to be discussed is the pooling layer and is often applied after the convolutional layer/layers. The main aim for the pooling layer is to reduce the the number of parameters needed in the network, thus reducing the

chance of overfitting and reducing the computational load of the network.

Unlike the the previous subsection revolving around weighing variables with kernels. A pooling layer is a simple downsampling of the data in the previous layer. Using the tensor in equation II.4 and renaming it to $\boldsymbol{X}^{(l-1)}$, the pooling layer tensor can be written as

$$\boldsymbol{X}^{(l)}_{t,m,n} = \max\left[\boldsymbol{X}^{(l-1)}_{n_s t:n_s t+f,n_s m:n_s m+f,n}\right] \tag{II.14}$$

where f is the size of the square submatrix in $\boldsymbol{X}^{(l-1)}$ such that $f < h$ and $f < w$. The parameter n_s is a positive integer called the stride, and is often restricted by $n_s \leq f/2+1$, but not enforced. The most used combination is $f = 2$ and $n_s = 2$. The stride can be thought of a slide parameter for the submatrix, and specifies the change in position for the submatrix when calculating a new element in the output $\boldsymbol{X}^{(l)}$. The stride parameter could be included in equation II.9 as done in II.14, however, I have omitted because I will not be using the stride parameter in the convolutional layer.

A small comment before ending this subsection, there exist multiple types of pooling methods. The one discussed above is called max pooling and is the most commen method. Other methods exists such as average pooling, but these methods will not be discussed here. The difference between the methods revolve around the function applied on the submatrix in $\boldsymbol{X}^{(l-1)}$.

3. Techniques etc

There exist multiple other techniques applied in CNN, however, I have restricted myself to only use the main mentioned above. A technique widely used is padding, and revolves around applying n frames of zeros in height and width dimension. For a game of snake where almost the entire board is zeros, this technique seem unnecessary.

Flattening is another technique used in CNN. By flattening the input $\boldsymbol{X}^{(l-1)}$ the entire tensor is flattened to a 1D input. By flattening in a specific layer, the usage of the CNN architecture such as the convolutional layer becomes difficult. After flattening the remaining network can therefore be treated as a regular NN.

E. Supervised learning

Supervised ML (Machine learning) is a branch of ML with many different subcategories of algorithms and methods, therefore this subsection will revolve around the methodology of this branch of ML.

In supervised ML, the system is introduced to a problem with the answer already known. The system is than trained to minimize some cost function with regards to the training data with the problem and answer. Supervised ML does therefore require a large amount of data for training. For a CNN and NN, the system or network is training by optimizing the weights such that the output of the network minimizes the cost function.

The cost function trained to minimize is dependent on the problem at hand. For classification task, such as snake. A possible cost function would be the cross entropy used in categorical problems such as the down, right, up and left categories in snake.

F. Reinforced learning

The theoretical aspects this subsection is credited [1] and [5].

Reinforced ML differs from supervised ML in multiple ways. The most predominant of which is that it does not require any pre-created training data, and instead generates its own by interacting with an environment. In a game of snake, the network will train itself by repeatedly replaying snake, and thus interacting with the environment. An important aspect of the learning is that it does not have any hard-coded

patterns when training unlike supervised learning.

When playing a game of snake, a reinforced algorithm ties to maximize the end score at the end of the game, whereas the supervised learning maximizes the correct answer for each movement based on the training set. To make the difference as concrete as possible: in reinforced learning time matter, whereas supervised learning is indifferent to time.

When all is said and done, a reinforced learning algorithm is at its very core a supervised learning algorithm. The difference being how it accumulates training data.

1. *Q-learning*

One of the most well-known methods in reinforced learning is Q-learning, and it's the method that will be applied in a game of snake to represent reinforced learning.

Probably the most important quantity in Q-learning is the Q-factor[2], but I will come to this later.

For the network to interact with the environment, it requires some stimuli based on its actions. These stimuli is called rewards and is denoted by r. In a game of snake, the rewards are as followed

$$r = \begin{cases} \text{Eats apple,} & 1 \\ \text{Dies,} & -1 \\ \text{Something else,} & \beta \end{cases} \tag{II.15}$$

where $\beta \in [0, -1]$ and is the parameter determining whether the snake is penalized for moving or not. Rewards are based on the possible actions the algorithm may take at a given state. I will therefore introduce the state-space s, which is just a collection of possible states. The state-space has the following form

$$s \in \mathbb{R}^{n \times i \times j} \tag{II.16}$$

where n denotes the number of states, and i and j represents the matrix or image.

The next quantity is called the action-space, and the action-space is a collection of all possible actions for a given state, such that the state s_i has the following actions given by a_i. The action-space has the following shape

$$a \in \mathbb{R}^{n \times k} \tag{II.17}$$

where k is the number of actions. All possible actions give a specific reward, and this is denoted by the reward space r

$$r \in \mathbb{R}^{n \times k}. \tag{II.18}$$

The entire state-space and the corresponding action and reward-space could be calculated beforehand, but this is a bad idea for large games such as snake. These spaces are therefore accumulated by having the network play the game and automatically creating a state-space with a corresponding action and reward-space. Therefore, the more the more the game played, the larger does the accumulated spaces become.

With the state, action and reward-space defined, the learned Q-value is given by

$$Q_{loss,i} = r_i + \gamma \cdot r_{max,i} \tag{II.19}$$

and with this, the Q-space is given by

$$Q_{i+1} = (1 - \alpha)\, Q_i + \alpha Q_{loss,i} \tag{II.20}$$

where Q_i is a row vector with all possible Q-values for all possible actions at a given state. The initial Q_0 is a row vector of all zeros. The α-parameter is some learning rate and $\gamma \in [0, 1)$ is the discount factor which is the importance of future rewards. The two last terms are r_i which is a row vector for the rewards for all actions in the state i, the last term $r_{max,i}$ is a bit tricky, since it's not the maximum possible reward in the current state. The parameter $r_{max,i}$ is the maximum reward in the next state

[2] Rather self explanatory from the name.

for the corresponding action in this state. This is best explained using an example. The snake has four possible actions, and lets assume the snake takes a right. The next state after the right turn has again four possible actions with their corresponding rewards. The maximum of these rewards would be $r_{max,i,j}$ for the right action in the initial state. The same goes for the three next directions. In my terminology I have deviated a bit from the norm with regards to the $r_{max,i}$ name, usually called $\max(Q')$.

As you may see, the entire Q-value process of creating a Q-space with a corresponding s-space is the same as creating a training set by rewarding good actions. However, cumulative scores such as the Q-value table created by equation II.20 will not work as training data. Therefore, the loss function for the Q-values is given by

$$C = ((Q_{loss,i} - y_i')^2 \tag{II.21}$$

where y' are the predicted values from the network at a given state. The ideal situation is that the network is not only capable of predicted the next reward, but also future rewards. That is the reason for the inclusion of the $r_{max,i}$ term. Notice that equation II.21 is the regular MSE. The Q-learning process has by the inclusion of equation II.21 been made into a regular optimization problem with regards to the the Q_{loss} space.

As mentioned earlier in this subsection, the network itself is responsible for creating the different spaces. However, it's highly likely that the network will be stuck in some kind of infinite loop. To avoid this, a usual strategy is to implement a exploration parameter $\epsilon \in [0, 1]$. By drawing a random number R, the next action is decided based on

$$a_{next} = \begin{cases} R \leq \epsilon & \text{Random movement} \\ R > \epsilon & \text{Network decides movement.} \end{cases} \tag{II.22}$$

Furthermore, the ϵ doesn't need to remain constant. The method I will apply for a decaying exploration parameter is

$$\epsilon(\text{episode}) = \epsilon_0 - \epsilon_0 \cdot \text{episode}/\text{episodes}. \tag{II.23}$$

A generalized and simplified algorithm of the Q-learning process is somewhat akin to:

Algorithm 1 Q-learning

```
Define lists for the states and Q_loss
for episode in episodes do
    while Not Game over do
        Find all possible states one turn forward
        r = the rewards for those positions
        Find all possible states two turns forward
        r_max,i = the highest rewards in two turns
        Q_loss,i = r + γ r_max,i
        Save the state and its corresponding Q_loss, i
        ϵ-exploration or model moves.
    end while
    Train model using the states and Q_loss
end for
```

G. Model evaluation

Some usual metrics for evaluating categorical models include the accuracy and F1 score with the accuracy being defined as

$$\text{Accuracy} = \frac{1}{n}\sum_{i=0}^{n} I(y_i = \tilde{y}_i) \tag{II.24}$$

and is the fraction of correct predictions. However, in a game of snake, such metrics are mostly obsolete. Instead the game score

$$GS = n_{\text{apples}} \tag{II.25}$$

which is the number of apples eaten. Using the height and width from equation II.4, the maximum score is given by

$$n_{\text{max score}} = h \cdot w - 2. \tag{II.26}$$

The minus 2 factor is added because the initial length of the snake is 2. Another metric that will be used in the evaluation of the models is the game efficiency

$$GE_i = \frac{\Delta s_i}{h \cdot w} \tag{II.27}$$

where Δs is the number of movements partaken by the snake between each score increment. Thus Δs_1 is the number of steps taken by the snake from the start of the game until the first apple is eaten.

III. METHOD

A. Code implementation

1. Snake

I will keep this brief since there is not much worth mentioning about a snake game implementation. However, a brief explanation would be that the game of snake is implemented in a matrix format in numpy using two-dimensional arrays. The initial start position for both the apple and the head (with one tail) is randomized when the game is initiated.

A major development decision that may impact the performance of the ML algorithms is that the snake can move in all four directions at any time: down, right, up and left. This instance is unorthodox from other version where only three movement options are allowed: left, right and forward.

2. Convolutional neural network

The entire implementation of the CNN was done using the open-source module Keras. The network itself took following shape:

1. Input layer of size $w \times h$ with a depth of three. The width and height were either 8×8 or 20×20 depending on the snake game.

2. The second layer is a convolutional layer with 64 kernels all of size 3×3 and the relu activation function with a stride of 1 as in equation II.9.

3. The last layer is a simple flattened output layer with 4 outputs each belonging to specific direction. The activation function is the softmax activation for the supervised learning and a linear function for the reinforced learning.

The network just described is the network that gave the best results for a 8×8 snake game using reinforced learning.

Before continuing with any further explanation I would like to introduce an important concept in a game of snake: boundaries. Without boundaries in the game, the reinforced learning algorithm is not capable of understanding what not to hit[3]. The boundary was added by extending the matrix by one in all directions, and giving the new extensions the same value as given to the snake tail.

A peculiar thing to noticing about my CNN is the input layer. Earlier when I introduced a game of snake in matrix notation in equation 1, the game board only had a depth of one, not three as specified in my first layer. This is because I extended the matrix dimension depth too a depth of three where each dimension contains a specific element, and only that element. The first depth contains only the element 1 from the original board. The second depth contains only elements denoted with 2 (this also includes the added wall), the rest of the elements are zero. The third depth does therefore contain the apple, and only the apple. In each depth, the specific element is denoted by a 1, whether the element is the tail, head or apple unlike II.1. Changing this might have some impact on the performance of the CNN, but this aspect will not be explored.

[3] I learned the hard way, wasting a couple days not getting any somewhat decent results. Adding boundaries doubled or tripled the performance of the algorithm.

B. Supervised Learning

The first objective when training a supervised network is to ensure proper training data. However, for a game of snake this is rather difficult. This problem was solved by the use of the Hamiltonian circuit discussed in subsection II B.

The two Hamiltonian methods: the regular Hamiltonian circuit and the perturbed Hamiltonian circuit were implemented for a square board of size $n \times n$. With these surefire methods implemented, multiple games of snake were played with both methods, but only two games were saved for each training set, one clockwise and the other counter-clockwise (different directions on the Hamiltonian circuit). The states and actions belonging to the two games are the training set for the supervised model. The remaining data was used to calculate the mean game efficiency for the Hamiltonian path, both perturbed and un-perturbed. This was all done for both a 20x20 and 8x8 board.

Using the training sets (regular and perturbed for both 20x20 and 8x8) four models with the same architecture was trained with different number of epochs. The accuracy was than calculated with respect to the entire data set for each data set - epoch combination. All models created with different training sets and epochs than played a total of N=10 games of snake where the game score was stored. The scores when then categorized into three categories: perfect, all zeros and any score in-between prefect and 0.

C. Reinforced Learning

During the Q-learning process I will keep a constant number of epochs when training the network, this being 10. However, the number of data in the training data will not remain constant, because of the accumulation of data points during the Q-learning process. The batch size used will remain constant at 50.

With regards to the implementation of the Q-learning process, it is following the code example 1 given in the theory section. However, there are a some small details worth mentioning with regards to training the CNN with the Q-values accumulated:

1. If the counter in the game goes on longer than some tolerance, the game ends. However, if the snake gets point the counter resets.

2. If the game ends because of the counter, and the snake has **not** accumulated any points, this game is not added to the training set. Effectively down-sampling the data set to avoid repetitive patterns.

3. The network is **not** trained after each episode, only after specified episodes. Where it's trained on the entire accumulated data set for 10 epochs. The used intervals are 50 episodes apart, thus the set is first trained at 50 episodes[4].

4. Before the model is trained, the data is down-sampled to avoid any duplicate sets in the training data by removing any duplicates.

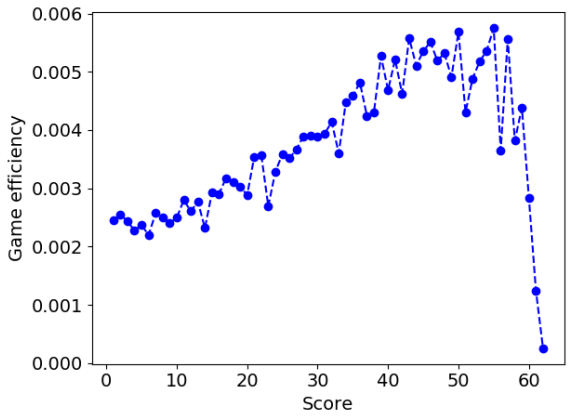
To avoid having to repeat myself multiple times, the calculation of the game score is independent of the method used. The game score was calculated by letting only the network trained at 0, 50, 100, 150...., 600 episodes play 250 games of snake where the mean score was taken for each episode. Thus resulting in mean game score for 0, 50... episodes.

With the formalities explained, the actual network training and parameter fine-tuning can begin. The first parameter to fine-tune is the learning rate of the network in the ADAM gradient descent algorithm, not the learning rate given in equation II.20. This was done by training the network $N = 5$ times for learning rates

[4] This is purely a design choice to make it easier to evaluation of the model easier at specific stages.

of $[1e-1, 1e-2, 1e-3]$ and evaluating the game score for each learning rate at 50 episodes apart from 0 to 600 episodes in total. The mean and maximum of those N=5 retraining runs were than plotted for their learning rate against their corresponding episode.

With a proper learning rate, the next issue revolves around optimizing the exploration parameter. Thus simply repeating the measurements done in the previous paragraph with a exploration parameter of $[0.1, 0.5, 0.9]$, but with a constant learning rate. The next issue revolves around optimizing the penalty hyperparameter for movement β from equation II.15. Just like previously, the other values are kept constant except the parameter measured, the values tested were $[-0.1, -0.05, 0, 0.1]$. The last hyperparameter to test is the γ-discount factor, the tested values here are $[0, 0.1, 0.4, 0.7]$.

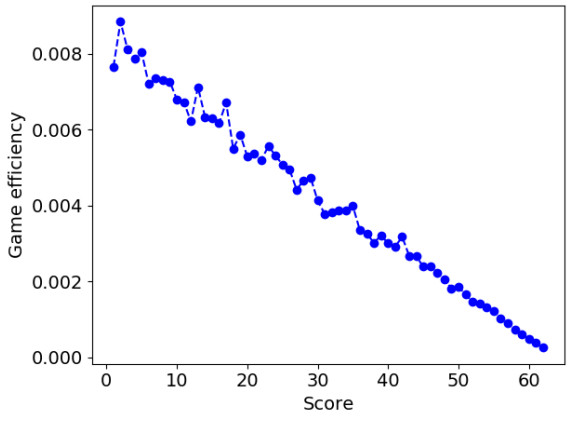


Figur 1: The mean efficiency score for N = 100 runs for a 8x8 snake grid using the pertubated Hamiltonian path.

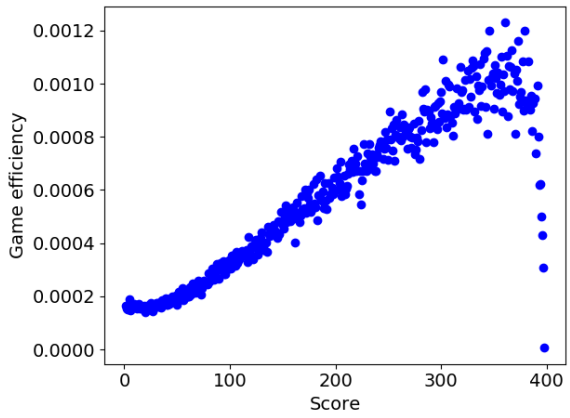
IV. RESULTS

A. Hamiltonian circuit

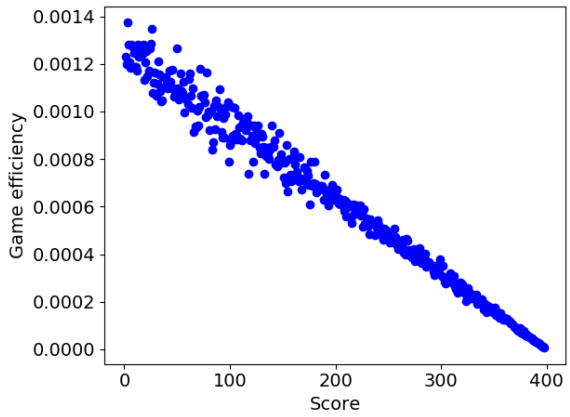
The game efficiency for the pertubated Hamiltonian circuit is shown in figure 1 and 3 for a 8x8 grid and a 20x20 grid respectively. The game efficiency for the entire Hamiltonian circuit is shown in figure 2 and 4 respectively. The plotted data points are the means of N = 100 random runs. During all the games, not a single game failed, thus performing flawlessly. This is not for the trained network, but for the Hamiltonian algorithms.



Figur 2: The mean efficiency score for N = 100 runs for a 8x8 snake grid using the regular Hamiltonian path.



Figur 3: The mean efficiency score for N = 100 runs for a 20x20 snake grid using the pertubated Hamiltonian path.



Figur 4: The mean efficiency score for N = 100 runs for a 20x20 snake grid using regular Hamiltonian path.

B. Supervised Hamiltonian path

Although this is a bit unorthodox, I will not be diving my data set from the Hamiltonian path into test and training data. Thus the accuracy calculated is for the entire data set used to train in the model. The entire result section for the supervised path can be summarized in a couple of tables. In table I and II are the number of 0-scores, perfect scores and non-prefect scores shown for N=10 games on a 20x20 board for both the model trained on the unperturbed and perturbed path respectively. Table III and IV are for a 8x8 board instead of the 20x20 board.

Tabell I: The game scores for N=10 games on a 20x20 snake board, using model trained on the full Hamiltonian path.

Epochs	0-score	$0 < \text{score} < n_{max}$	n_{max}	Accuracy
1	7	0	3	0.99997
2	0	1	0	1
3	0	0	10	1

Tabell II: The game scores for N=10 games on a 20x20 snake board, using model trained on the perturbed Hamiltonian path.

Epochs	0-score	$0 < \text{score} < n_{max}$	n_{max}	Accuracy
1	10	0	0	0.97
2	5	0	5	1
3	3	0	7	1

Tabell III: The game scores for N=10 games on a 8x8 snake board, using model trained on the full Hamiltonian path.

Epochs	0-score	$0 < \text{score} < n_{max}$	n_{max}	Accuracy
1	8	2	0	0.967
5	4	6	4	0.975
10	7	3	4	0.978

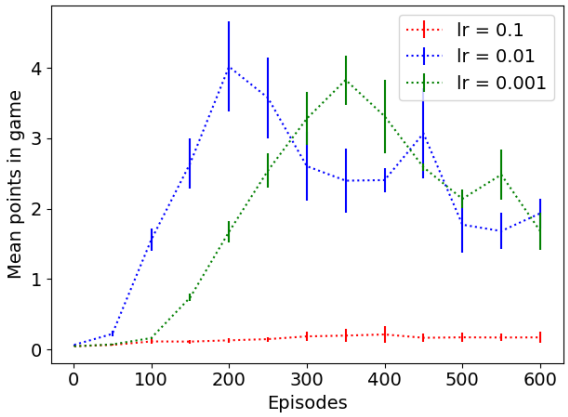
Tabell IV: The game scores for N=10 games on a 8x8 snake board, using model trained on the perturbed Hamiltonian path.

Epochs	0-score	$0 < \text{score} < n_{max}$	n_{max}	Accuracy
1	8	2	0	0.66
5	6	4	0	0.97
10	5	5	0	1

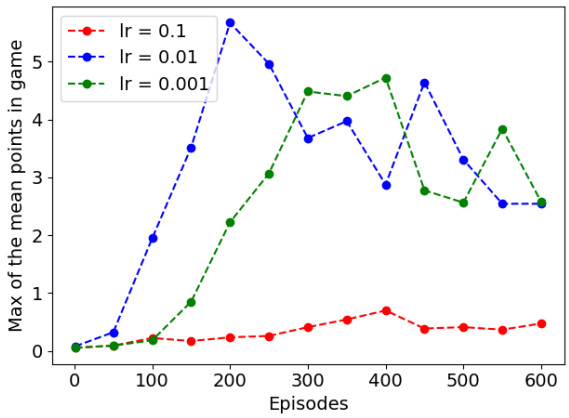
C. Reinforced Learning

A small comment regarding figure 7 to 10: the last data point can ignored and is because a small error in the plotting program.

The mean game score for different learning rates over a total of 250 games played by a model trained on $[0, 50, 100, ...550, 600]$ episodes is plotted in figure 5. The plotted vales are the mean of having re-trained the model N = 5 times, with the mean and the standard error $\sigma/\sqrt{N}$ calculated between the N=5 models. In figure 6 are the maximum means of the game scores plotted, not the means over the N=5 re-training times.

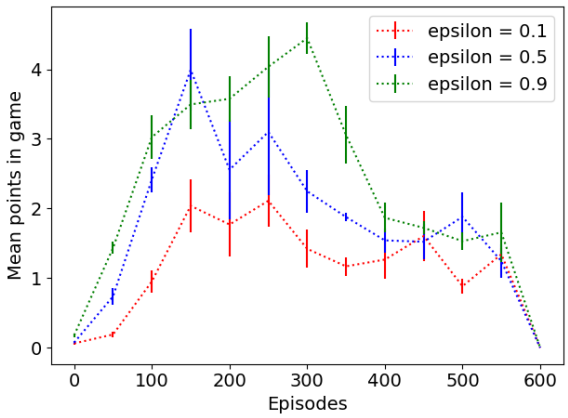


Figur 5: The mean game score of 250 games played on 5 re-trained models. The error bars represent the standard error between the N = 5 re-trained models, not the 250 games. This is done for three different learning rates in the CNN.

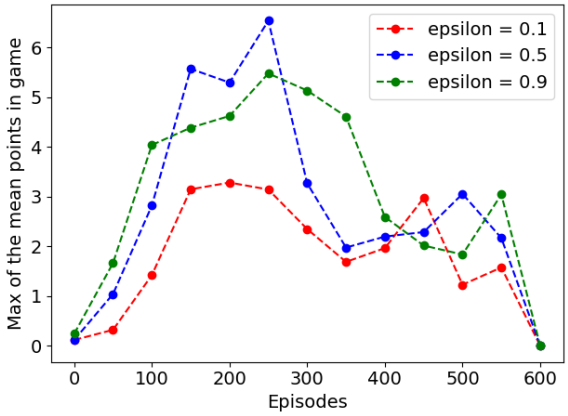


Figur 6: The mean game score of 250 games played on 5 re-trained models. The plotted values are the maximum values achieved across the N = 5 re-trained models. This is done for three different learning rates in the CNN.

Using a learning rate of $lr = 0.01$, the game score is plotted against the number of episodes the network is trained for different initial exploration parameters ϵ in figure 7 and 8. In figure 7, the plotted values are the mean of the mean of 250 games for N = 5 re-trained models, the errorbars are the standard error. In figure 8 are the maximum values of the N = 5 re-trained models plotted.

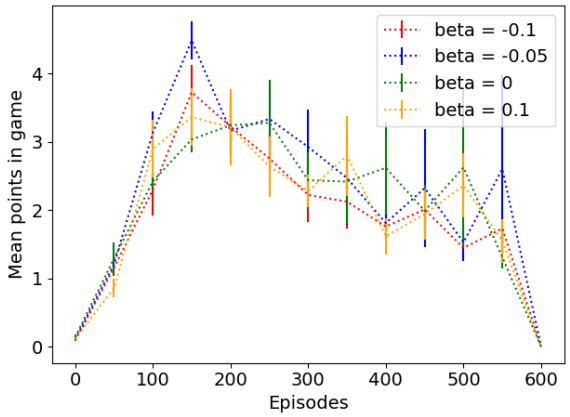


Figur 7: The mean game score of 250 games played on 5 re-trained models. The error bars represent the standard error between the N = 5 re-trained models. This is done for three different initial exploration factors in the reinforced learning algorithm.

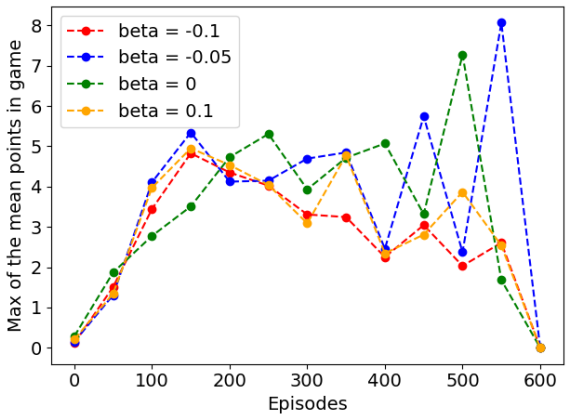


Figur 8: The mean game score of 250 games played on 5 re-trained models. The plotted values are the maximum values achieved across the N = 5 re-trained models. This is done for three different initial exploration factors in the reinforced learning algorithm.

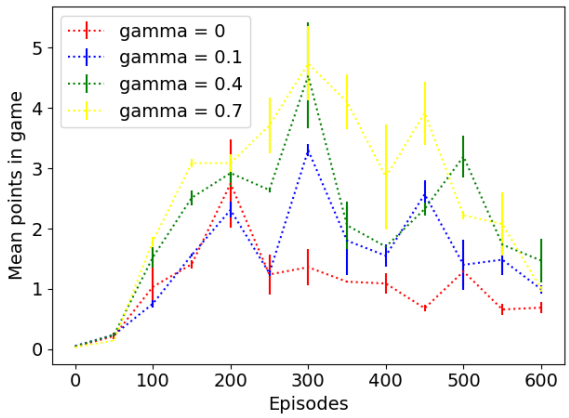
With an exploration parameter of 0.9, the next issue at hand is to find the optimal movement penalty parameter β. As previously, in figure 9 is the mean value of N = 5 re-trained networks for the mean of 250 games at specific episodes within the training regiment for different β-vales. In figure 10 is the maximum mean of the N = 5 re-trained models plotted.



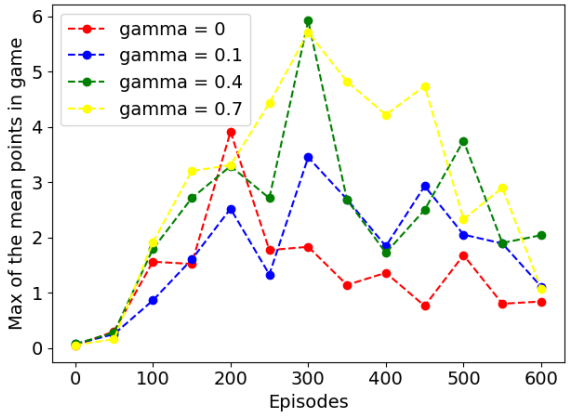
Figur 9: The mean game score of 250 games played on 5 re-trained models. The error bars represent the standard error between the N = 5 re-trained models. This is done for four different movement penalties β in the reinforced learning algorithm.



Figur 10: The mean game score of 250 games played on 5 re-trained models. The plotted values are the maximum values achieved across the N = 5 re-trained models. This is done for four different movement penalties β in the reinforced learning algorithm.



Figur 11: The mean game score of 250 games played on 5 re-trained models. The error bars represent the standard error between the N = 5 re-trained models. This is done for four different discount factors γ in the reinforced learning algorithm.

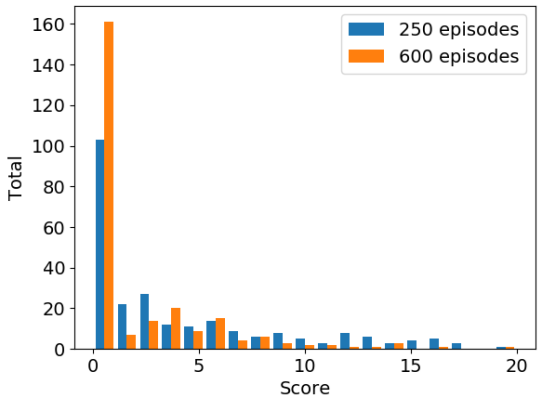


Figur 12: The mean game score of 250 games played on 5 re-trained models. The plotted values are the maximum values achieved across the N = 5 re-trained models. This is done for four different discount factor γ in the reinforced learning algorithm.

The last hyper-parameter is the discount factor γ, the movement parameter is set to $\beta = -0.05$. In figure 11 is the mean of the mean game score for 250 for 5 re-trained models plotted with the respective standard error for the 5-retrained models. In figure 12 is the maximum of the mean game score for 250 games for 5 re-trained models plotted. In both plot the curves represent multiple discount factors as seen.

Figure 13 is a histogram of the scoring for the 250 games played at different episodes. During

all the games played in figure 1 to 12 the highest scores for each 250 games played, were around 20-30, with the highest score achieved being 33.



Figur 13: A histogram of the number of scored points for 250 games between a model trained on 250 episodes and one trained on 600 episodes.

V. DISCUSSION

I want to begin the discussion with a possible method of improvement. Instead of treating each move separately, and forcing the network to take actions where multiple answers may be correct. A possible method would be to have the network output a image where it draws the path to the apple. Such an implementation would allow for easily generated training data and the use of the Hamiltonian circuit for training would be unnecessary.

Now for the discussion about the results. The game-efficiency in figure 1 to 4 are important to discuss since they represent the generated training data. More specifically, how 'human like' the data sets are. In the beginning of a game, humans will tend to take the shortest route to the apple similarly to figure 1 and 3 for the perturbed path. As the grow longer, a possible reaction would be to follow a specific path more closely as done in the Hamiltonian circuit such as figure 2 and 4. An improvement for the training data used in the supervised model would be to use the perturbed path for the beginning, than switch to the full Hamiltonian path. Thus minimizing the total number of moves used.

A. Supervised model

There is not really much to discuss on this topic. As seen from table I and III, the model is either perfectly able to remember the Hamiltonian path or not. By doing so, it's able to always clear the game, or die very fast. I myself belive this to be because I have not given the model enough training data. If I were to give model more games played it may not die so easily. I believe the problem lies in the fact that it's not able to determine whether to move clockwise or counter-clockwise in Hamiltonian path. Although, I ensured that the model had data corresponding to both clockwise and counter-clockwise paths. If the game were to start without any tail, the model would always clear the since it is able to remember the path. Training a model this way allows for perfect clearance of the game, but only if nothing unexpected happens along the path. Making it unable to adjust as seen with the path direction. This theory is supported by the extremely small accuracy error that may come from only one or two wrong predictions.

For table II and IV there were not a single perfect game, instead almost all games ended with a score of 0. Furthermore, most games were canceled because of the tolerance limit. What I had hoped to see was that the game was capable of adjusting the Hamiltonian path to an unperturbed path when the snake tail was short. As shown clearly from the tables, this was not the case. Instead the model continuously predicted the snake to move in a given pattern, essentially trapping the snake in an infinite loop and the snake dies by the tolerance limit. It can also be seen from table IV that the snake was perfectly able the reproduce the training data with 100% accuracy. However, it still failed once introduced to an unknown state.

The supervised method either gives a perfect game when trained on the entire Hamiltonian path in one direction, or a perfect infinite loop when trained with shortcuts. It is therefore very sensitive to unknowns and will easily fail if these are introduced.

B. Reinforced learning

My initial plan was to have a larger game board than the 8x8 board I ended up using. However, this simply took too long and were to difficult computationally. I felt that was worth noting, since this problem is far less common using the supervised method. The reason for this is that the generation of the Hamiltonian path is faster than the Q-vale generation.

From figure 5 it's clear that a learning rate of 0.01 or 0.001 both work reasonably decent, and a learning rate of 0.1 does not work at all. In regards to the hyperparameters, there are some clear benefits by optimizing them. The first hyperparameter is the exploration parameter ϵ. From both figure 7 and 8 there is a clear trend that increasing the exploration increase the performance of the model. It can be safely assumed from these figures, that for a game of snake, a high exploration is beneficial. However, it is difficult to determine whether $\epsilon = 0.9$ were better than $\epsilon = 0.5$.

From figure 9 and 10 there seem to be no real correlation between the hyperparamter β and performance of the model. A reasonable assumption would than be that whether the snake is penalized for moving does not matter. However, figure 10 may suggest some advantages with a small penalty, but this may be because of the randomly initialized weights in the kernel.

The last two figure 11 and 12 clearly showcases the importance of the future reward. Whit a $\gamma = 0$ the model had a horrible performance with a score between 1-2 at it's peak. With an increasing discount factor, the performance of the model also increases. A possible option would to make a model of $lr = 0.01$, $\beta = -0.05$, $\epsilon = [0.5, 0.9]$ and $\gamma = 0.7$ multiple times to get the best performing model by having good randomized initial weights.

A general trend I would like to discuss is the mean number of apples accumulated during one game. The best performing models was capable of scoring on average of 8-points. This is not a very good result since the board size is 8x8. Furthermore, in figure 13, it clear that most games end up with a score of 0. However, some games are performing rather good as seen where the highest score in histogram was 19. If all the 0 performance models were removed, the model itself would be more robust than the supervised, since it's able to react to unknown parameters.

A small comment in the end. I figure 5 to 12 there are clear trends of overfitting, and the best models are often found at around 200-400 episodes. This shows that my snake model is highly susceptible to overfitting, and fine tuning the training may drastically improve the performance and remove most 0-scores.

VI. CONCLUSION

Both the supervised and reinforced algorithm showed capabilities of learning to accumulate apples. However, neither were perfect. The supervised algorithm either performed excellent, remembering the entire Hamiltonian path or failing and dying because of the clockwise/counter-clockwise problem. When introducing the perturbed Hamiltonian path, the model completely failed and mostly died when being stuck in an infinite loop. The problem with the supervised learning method is that it is not capable of adjusting to unknown parameters.

The reinforced algorithm, albeit being able to adjust to unknown parameters did perform considerably worse than the supervised method with the Hamiltonian path, but better than the perturbed Hamiltonian path. The highest score achieved was 33 on a 8x8 board, and the highest mean was of 8 points pr game. For any environ-

ment where a surefire method such as Hamiltonian path does not exists, the adaptability of the reinforced method may prove superior.

---

[1] Mauro Comi. How to teach ai to play games: Deep reinforcement learning. https://towardsdatascience.com/how-to-teach-an-ai-to-play-games-deep-reinforcement-learning-28f9b920440a, 2019. [Online; accessed 09-December-2019].

[2] Andrej Karpathy. Convolutional neural networks (cnns / convnets). http://cs231n.github.io/convolutional-networks/, 2019. [Online; accessed 09-December-2019].

[3] Jon Andre Ottesen. Logistic regression and neural networks. https://github.com/JonOttesen/FYS-STK-Projects/tree/master/Project%202, 2019. [Online; accessed 12-Desember-2019].

[4] John Tapsell. Nokia 6110 part 3 – algorithms. https://johnflux.com/2015/05/02/nokia-6110-part-3-algorithms/, 2019. [Online; accessed 03-Desember-2019].

[5] Wikipedia. Reinforcement learning — Wikipedia, the free encyclopedia. http://en.wikipedia.org/w/index.php?title=Reinforcement%20learning&oldid=931421705, 2019. [Online; accessed 19-December-2019].