Aalborg University Copenhagen

Semester:

10

Title:

Experiments Building a Q-table Learner in a Continuous State Space

Project Period:

Spring 2018

Semester Theme:

Aalborg University Copenhagen

Frederikskaj 12,

DK-2450 Copenhagen SV

Semester Coordinator:

Secretary:

Supervisor(s):
Hendrik Purwins

Project group no.:

Members:
Jannik Vilhelm Reffstrup

Abstract:

In this project methods for building a tabular q-learner over a continuous state space is proposed and investigated. Various problem areas are found and methods to counteract these are also tested. It is shown that it is possible to create a tabular learner over a continuous state space with the use of “q-points”, but optimal methods for managing and updating these are still to be investigated.

Experiments Building a Q-table Learner in a Continuous State Space

Jannik Reffstrup
Aalborg University Copenhagen
31-05-2018

Abstract

In this project methods for building a tabular q-learner over a continuous state space is proposed and investigated. Various problem areas are found and methods to counteract these are also tested. It is shown that it is possible to create a tabular learner over a continuous state space with the use of "q-points", but optimal methods for managing and updating these are still to be investigated.

1 Intro

Artificial Intelligence (AI) used in games for controlling of opponents or companions are usually predictable and non-adaptive to the players actions. This can add a certain level of puzzle solving, as the behaviour must be figured out which is the case in games such as Hitman or Dark Souls. But more often the AI is constructed to simulate real human decision processes and behaviours with the purpose of making the player forget the systems behind the characters and feel further immersed in the game. These days there is a lot of focus on a third kind of AI controlled by machine learning with the work of Google leading the way. These neural network(NN) based AI are trained over a long timespan to master games but have rarely been used together with players as the training is often compersome and does not adapt to the player as fast as the player adapts to the AI. These methods can adapt to any behaviour and are constantly being developed upon to make them faster adapting and more stable. In a previous project it was made clear that simple reinforcement learning algorithms can be effective at providing live adaptation during gameplay, even while being far less capable systems. It seems that the agent being a fast learner out-prioritizes learning complexity. This project evolves around developing an AI capable of both fast learning and higher level complexity through using the methods of q-tables on top of a continuous state space.

2 Background

This next section will go through some of the methods influencing this project.

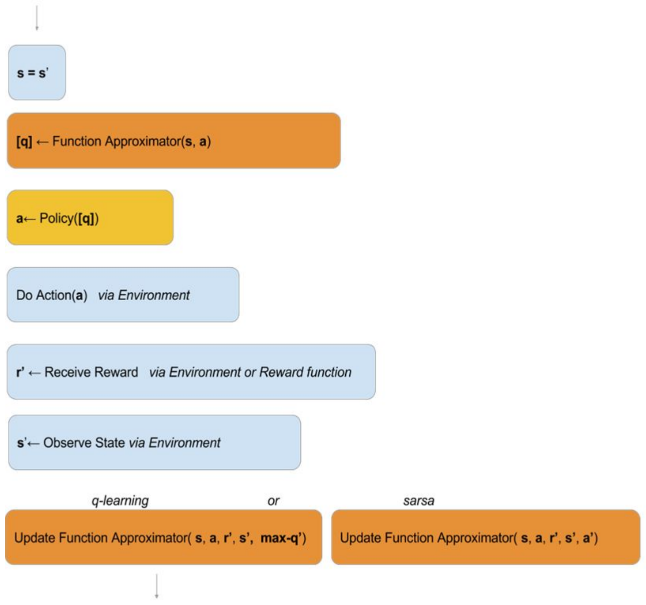
2.1 Reinforcement Learning basics

Reinforcement learning is all about creating a flexible algorithm which reinforces itself towards a more optimal version by experiencing the environment online. Reinforcement learning consists of some basic elements. The data received from the environment describing what the agent is sensing is called the state *S,* and the set of actions which the agent can choose from is denoted A. The part of the program which samples the state, makes a decision and performs an action is called the agent. In order for the agent to optimise the algorithm towards an optimum, it needs to know whether the last action in the last state was good or bad. For this a reward *R* is given by the reward function *r: S x A → R*. This can be any number but most often between -1 and 1 . After receiving a reward, the agent can be updated. This can happen after each action or at the end of an epoch or episode, such as reached goal or failure. How much the agent should change the algorithm estimating the future reward is called the value or q-function, and can be controlled by a value called the learning rate denoted as α. The higher the learning rate, the faster the agent will learn, but the higher is the risk of it not finding the average optimum as it overshoots. A discounted sum of future rewards is called *the return:* $R_t = \sum_{i=t}^{\infty} \gamma^{i-t} r_i$.

How far back the reward should affect previous actions can usually be controlled with a discount factor $\gamma \in [0, 1]$. A deterministic policy is a mapping from states to actions: $\pi : S \rightarrow A$. It is the agent's goal to maximise the expected return. The q-function or action-value function is defined as $Q^{\pi}(s_t, a_t)$ (Sutton & Barto 2012) (Andrychowicz et al. 2017).

2.2 Q-learning and SARSA

A standard machine learning agent could have a structure as the following and can be a q-learner or using SARSA, which is two methods influencing how careful the agent is behaving.



s = s'
[q] ← Function Approximator(s, a)
a← Policy([q])
Do Action(a) via Environment
r' ← Receive Reward via Environment or Reward function
s'← Observe State via Environment
q-learning
or
sarsa
Update Function Approximator(s, a, r', s', max-q')
Update Function Approximator(s, a, r', s', a')

Figure x: An example of a reinforcement agent update.

2.3 Q-table

A q-table is a representation of a finite state-space, where there is stored a q-value for each possible action in each state. This can quickly grow extremely large, and is not good at generalising, as each state-action-pair needs to be explored. q-tables can in simple environments be extremely effective, as they do not tend to suffer from catastrophic forgetting. The most classic task for a q-table agent is the grid-world environment, as this can clearly display the q-values for each state-action and thereby the decision process of the agent (Sutton & Barto 2012, p.79).

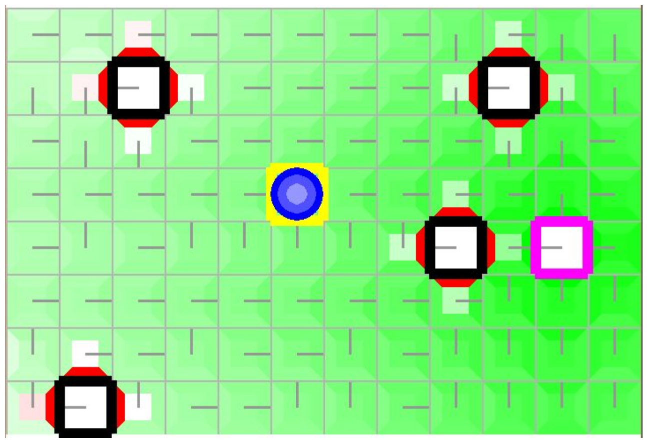


Figure x: An illustration of a grid-world environment, where the purple square is representing the goal and black squares represent a negative reward.

q-value retrieval:

$$q(s,a) = table(s,a)$$

The q-table can be updated via the temporal-difference update:

$$q'(s,a) = q(s,a) + \alpha*(r+\gamma *max(q'(s',a')) - q(s,a))$$

2.4 Feature-based-Action

Another simple method using reinforcement learning is the feature-based agent(FB agent). In contrast to the q-table agent, the feature-based agent is adapting few weights with information from a continuous state space. The feature-based agent can not learn connections between inputs, but through custom feature-functions it can evolve a kind of stomach-feeling about the state-action-pair. These feature-functions inputs the state and action, and works as models of the environment, returning exampelvise a larger value if input action is getting the agent closer to the goal and smaller if further away. The agent calculates each q-value by adding each feature-value multiplied with an respective internal weight.

q-value retrieval:

$$q(s_t,a_t) \leftarrow w_0 + w_1 f_1(s,a) + w_2 f_2(s,a) + ... + w_i f_i(s,a)$$

Each weight can be updated via the temporal-difference update:

$$w_i \leftarrow w_i + \alpha(r_{t+1} + \gamma q(s_{t+1}, a_{t+1}) - q(s_t,a_t))\ f_i$$

The feature based agent has a large potential in games, as this can be learn extremely fast and thereby make for an interesting game-AI. The downside is, that the agent is not capable of learning anything complicated without extensive pre-pro-grammed features.
Such an agent was implemented in multiple environ-ments in a previous project, and showed to learn a simplified Pac-Man environment at near human level with features helping navigating to nearest coin etc. It also functioned successfully as opponents and teammates in a top-down shooter.

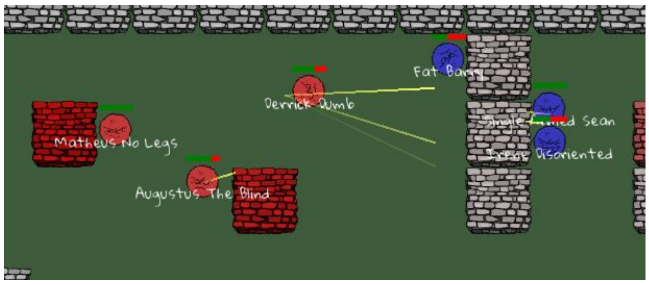
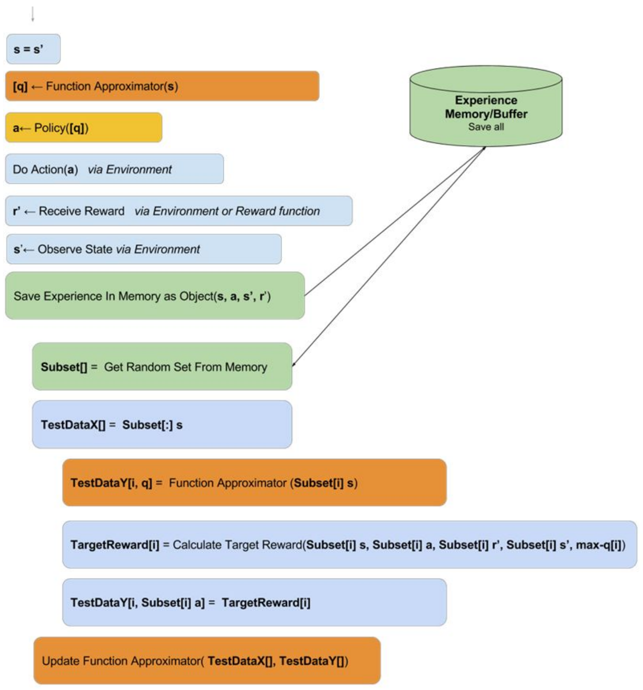


Figure x: A screenshot of the top-down shooter environment, where each soldier is a feature based agent and is containing a unique set of weights. The features being based upon distance to cover, nearest enemy, team mate and health.

2.5 Experience Replay

Experience replay is a method originally used for optimising learning with a neural network. The method can furthermore be used in combination with other learners. After each action, the experience is saved in tuples of <*s,a,s',c*> where *c* is the expected sum of future rewards. These tuples are then stored in a database, which makes it possible to use the data more than once. Experience replay is implemented in order to aid in several ways. One benefit is efficiency increase, as the sample efficiency is heightened by making reuse possible. Experience replay can also make use of mini-batch updates, which can make the computation faster. Together with efficiency, experience replay can also increase stability, which is a big deal with neural network. This makes sure that the agent is not only trained with data collected by using the most recent policy, but also previous experiences. This makes sure that the agent does not forget a previous learned policy (de Bruin et al. n.d.).

An update using the experience replay, can look like the following pseudo code.



```
s = s'
[q] ← Function Approximator(s)
a← Policy([q])
Do Action(a)   via Environment
r' ← Receive Reward   via Environment or Reward function
s'← Observe State via Environment
Save Experience In Memory as Object(s, a, s', r')
    Subset[] = Get Random Set From Memory
    TestDataX[] = Subset[:] s
        TestDataY[i, q] = Function Approximator (Subset[i] s)
        TargetReward[i] = Calculate Target Reward(Subset[i] s, Subset[i] a, Subset[i] r', Subset[i] s', max-q[i])
        TestDataY[i, Subset[i] a] = TargetReward[i]
    Update Function Approximator( TestDataX[], TestDataY[])
```

Figure x: An example of training with experience replay.

In a previous project it was clear that saving the experience in groups could make the agent more stable, as the oldest experiences are thrown out when reaching a limit.

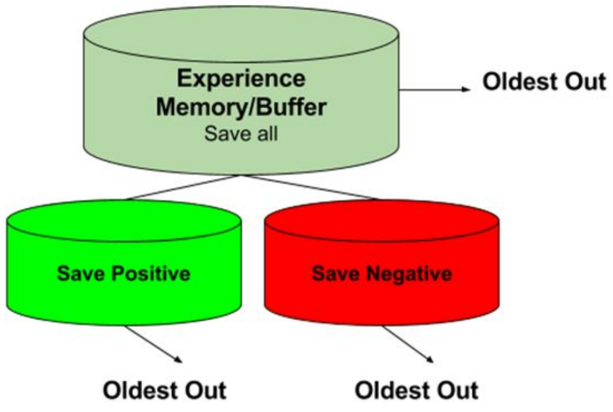


Figure x: A variant of experience replay from a previous project where positive and negative experience is saved separately.

2.6 Eligibility traces

Eligibility traces is another way of making the update more efficient. By using eligibility traces, the agent stores a number of previous state-action pairs in order of experiencing. When reaching a reward, all previous state-action qvalues are updated with the the most recent actions weighted higher. (Sutton & Barto 2012, p.154')
(Sutton & Barto 2012, p.165)

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma 2 R_{t+3} + \cdots + \gamma T-t-1 R_T$$

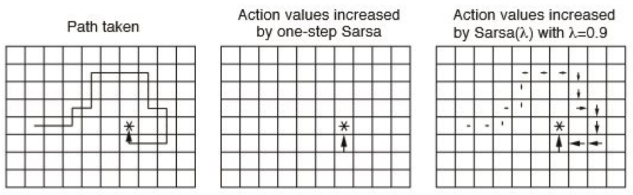
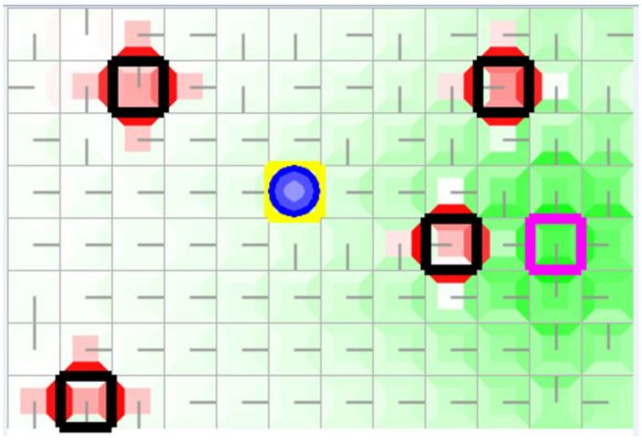

Path taken
Action values increased by one-step Sarsa
Action values increased by Sarsa(λ) with λ=0.9

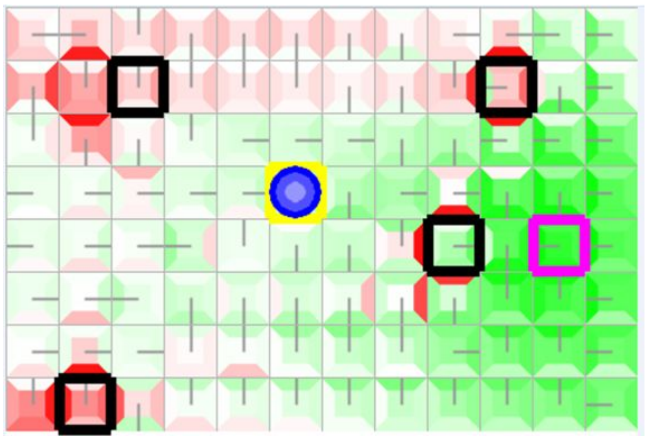
Figure 7.12: Gridworld example of the speedup of policy learning due to the use of eligibility traces.

Figure x: A figure from (Sutton & Barto 2012, p.171)

3 Challenge of Catastrophic forgetting

Previous projects pointed out the challenge of catastrophic forgetting, which is highly relevant when dealing with FB agents and NN agents. The tendency happens when the agent starts to exploit the learned policy. This results in only the states visited by the learned policy is being visited, resulting in the knowledge about the rest of the state space being forgotten terminally confusing the agent (de Bruin et al. n.d., p.3). Underneath is a figure showing q-values from a NN-agent in a grid-world environment, where the left image shows how the q-values should be. The rightmost image shows how the values get washed out when the agent starts exploiting the policy. This is especially a problem when utilising in game AI as the agent should learn while performing instead of training and then performing without changing. Q-tables and agents using experience replay tend to be a lot more stable.





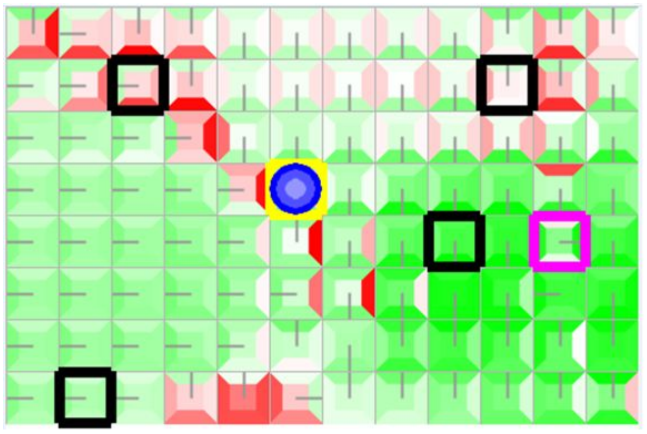


Figure x: A grid-world environment with displayed q-values. The first figure showing the optimal policy and the 2. and 3. showing how the agent forgets the previously learned as exploiting the polity.

4 Flexible q-learning

The method proposed in this project is to lay out a q-table over a continuous state space. This could be done by simply dividing the state space into areas, such that 2 features explaining x and y-position and ranging from 0-1 could be divided into n-spaces.

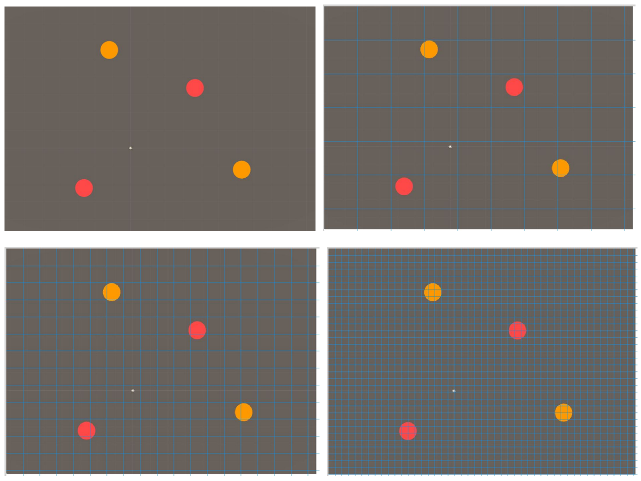


Figure x: Screenshots from an environment where the agent can navigate up, down, left and right through a continuous *state space. The red dots being negative reward areas and yellow positive. The blue grid shows how the state space could be divided in larger or smaller areas.*

In these spaces, the agent will perform the same action until reaching a new space. With this method normal q-table learning can be used, but problems arise if the state space-division is not fine enough, as the agent will get confused and when sometimes missing the goal. A too fine division will cause a need of an enormous q-table resulting in slow learning and need of high computer power and memory.

What is proposed in this project is to find a way of dividing the state space into areas without having a specific size of the q-table. Where instead of using a q-table in grid form, q-points are stored. These points contains the standard q-table-data, <s, q(s,a)>. The agent looks up the q-value for a given state by finding the q-point containing state-values. A q-table utilising q-points could look like the following.

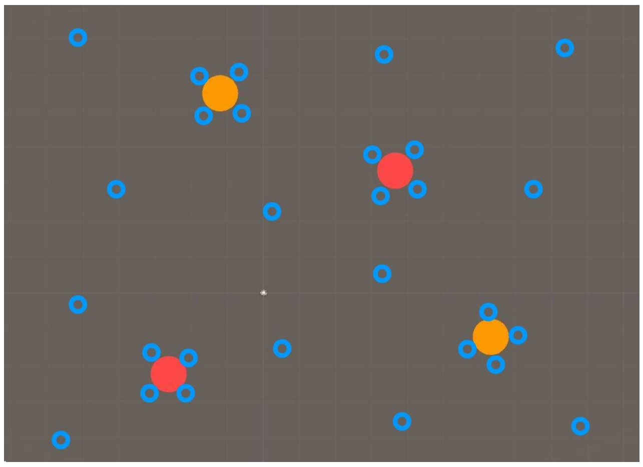


Figure x: An environment based on a q-table using q-points. The q-points are displayed as the blue circles.

In an optimal example, only the q-points needed are stored, which can make learning extremely fast and computation time even faster. The challenge with this algorithm is to find out where the q-points should be placed.

q-value-retrieval:

q(s,a) = FindNearestQpointInDatabase(s, a)

The q-point can be updated via the temporal-difference update:

qPoints'(s,a) = qPoints(s,a) + learningrate (r+discount*max(qPoints'(s',a')) - qPoints(s,a))*

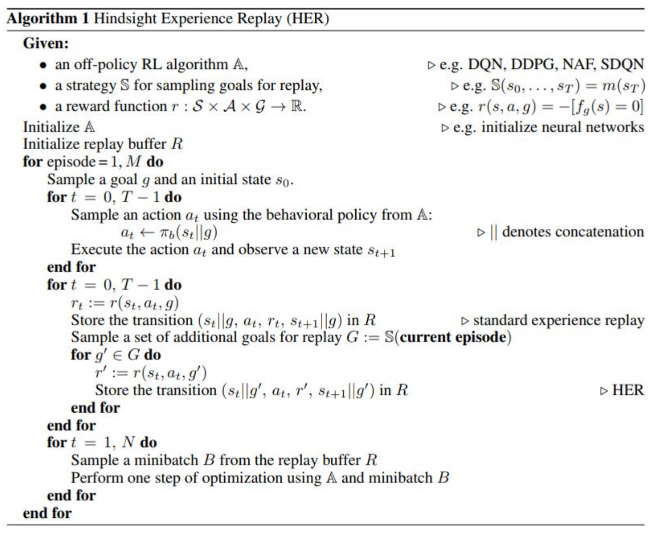
5 SOTA

5.1 Hindsight Experience Replay

These days there has been a wide succes in reinforcement learning with the use of neural networks. An agent has learned to play Atari (Mnih et al. 2015) and defeat the best human player in the game of GO(Silver et al. 2016). For these feats the policies has been feeded an enormous amount of training data, but often this an unrealistic luxury. To make more out of the collected data a method of combining experience replay(Zhao et al. 2016) and universal policies(Ioffe & Szegedy 2015) has been created called Hindsight Experience Replay(HER) (Andrychowicz et al. 2017).

HER uses binary rewards, which means that it is not needed to create convoluted reward functions. Traditionally this means that the agent receives a 1 on task completion and otherwise 0. The focus on HER is to make data use more efficient. This is done by using experience replay which stores the collected data for reusability and by using the philosophy of universal policies where not only the current state is given as input, but also a goal. The example is an agent learning to play hockey and misses the goal to the right. Usually the agent would receive a 0 for not completing the task, but here the initial state and outcome state is stored in the experience databank. When training the agent using this datapoint, the state is inputted and the outcome becomes the new goal, which means that the agent receives a reward of 1, as it would have scored if the goal had been to the right. This way the agent can use all the data collected from failing too and gets a stronger understanding of the environment (Andrychowicz et al. 2017).

An overview of the HER update can be seen below



```
Algorithm 1 Hindsight Experience Replay (HER)
Given:
  • an off-policy RL algorithm $\mathbb{A}$,                          ▷ e.g. DQN, DDPG, NAF, SDQN
  • a strategy $\mathbb{S}$ for sampling goals for replay,             ▷ e.g. $\mathbb{S}(s_0, \dots, s_T) = m(s_T)$
  • a reward function $r : \mathcal{S} \times \mathcal{A} \times \mathcal{G} \to \mathbb{R}$.   ▷ e.g. $r(s, a, g) = -[f_g(s) = 0]$
Initialize $\mathbb{A}$                                                ▷ e.g. initialize neural networks
Initialize replay buffer $R$
for episode = $1, M$ do
    Sample a goal $g$ and an initial state $s_0$.
    for $t = 0, T - 1$ do
        Sample an action $a_t$ using the behavioral policy from $\mathbb{A}$:
                $a_t \leftarrow \pi_b(s_t||g)$                         ▷ || denotes concatenation
        Execute the action $a_t$ and observe a new state $s_{t+1}$
    end for
    for $t = 0, T - 1$ do
        $r_t := r(s_t, a_t, g)$
        Store the transition $(s_t||g, a_t, r_t, s_{t+1}||g)$ in $R$     ▷ standard experience replay
        Sample a set of additional goals for replay $G := \mathbb{S}$(current episode)
        for $g' \in G$ do
            $r' := r(s_t, a_t, g')$
            Store the transition $(s_t||g', a_t, r', s_{t+1}||g')$ in $R$   ▷ HER
        end for
    end for
    for $t = 1, N$ do
        Sample a minibatch $B$ from the replay buffer $R$
        Perform one step of optimization using $\mathbb{A}$ and minibatch $B$
    end for
end for
```

Figure x: Illustration from (Andrychowicz et al. 2017).

6 Implementation

6.1 The test environment

The implementation of this novel method was not without challenges. A test environment was created which bares assembly to the gridworld environment but with continuous states. In this environment the agent has to navigate to the goals without hitting the traps. The agent can navigate up, down, right or left and the q-points are displayed at state-positions and with an arrow pointing in the direction corresponding to the action associated with the highest q-value. In this environment it is clear whether the agent has learned to navigate correctly, and whether the q-point locations are positioned correctly.

6.2 The implementations

The first implementation was built upon a little different concept. Here the q-point consisted of the state-values, a single overall q-value and a list of references to the q-notes led to by the respective actions. The q-value was retrieved in the following manner:

q(s,a) = FindNearestQpointInDatabase(s, qPoint(a))

This method was implemented was inspired by the methods of experience replay, and the idea was that each q-update could ripple backwards without needing to store separate experiences, as each q-point contained the next q-points.
This implementation was capable of mastering the simple environment, but not more advanced environments. Because of the lack of performance and transparency, the princip was simplified towards a standard q-table. This second implementation was more stable and because of simplicity easier to troubleshoot.

6.2.1 Q-update variations

The first update method was the simplest where the agent found the q-values by finding the closest q-point, which works, but when in between states, should it not be influenced by all nearby states? The thought was that the agent was part of a location between 3 q-points creating a polygon in the same way as with a Barycentric coordinate. The agent would then collect q-values from all 3 q-points and then scale the average according to distance, such that the nearest q-points had a larger influence on the decision. When receiving a reward, the credit would then also be scaled and delivered to all 3 points. First experiments were made where q-values were gained from the 3-nearest points, but it was clear that this was wrong, as the nearest 3 could be placed to one side from the agent as shown below. Another problem is to find the correct polygon, as multiple could be possible as shown below. Furthermore it was not clear whether the quad would need to be in more dimensions, as the dimensions of the state-space increases. The idea was therefore saved for further testing in another project.

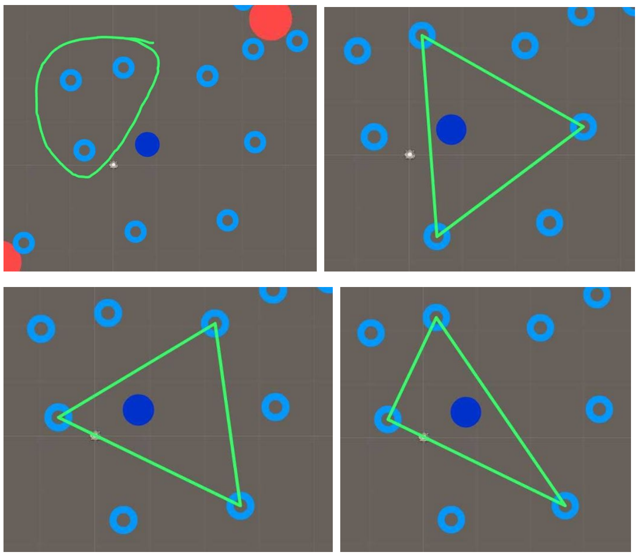


Figure x: 4 situations where the agent, illustrated by a solid blue, dot can be associated with near q-positions in different ways.

6.3 Placement of q-points

The largest challenge of this project is to determine where to place the q-points, as only the necessary amount is wanted. If the algorithm responsible for creating the points is too generous, the amount of q-points will spiral into infinity handicapping both learning speed and cpu. If too few points are placed, the agent will miss the goal and thereby get confused.

In this project two methods were examined.

The first created a new q-point when:

- receiving a reward being non-0.
- when entering a new state resulting in a q-value being too different than expected.
- when the reward is different than what was lastly given in this state.

The second method included:

- chance of randomly creating a q-point with the current state values.
- chance of randomly combining two points if both highest q-value was at the same position.
- chance of creating a q-point when receiving a non-0 reward

It turned out that the second method was the most effective, but needed to be used with care, as too high chances of creating or combining q-points, could spoil the learning.

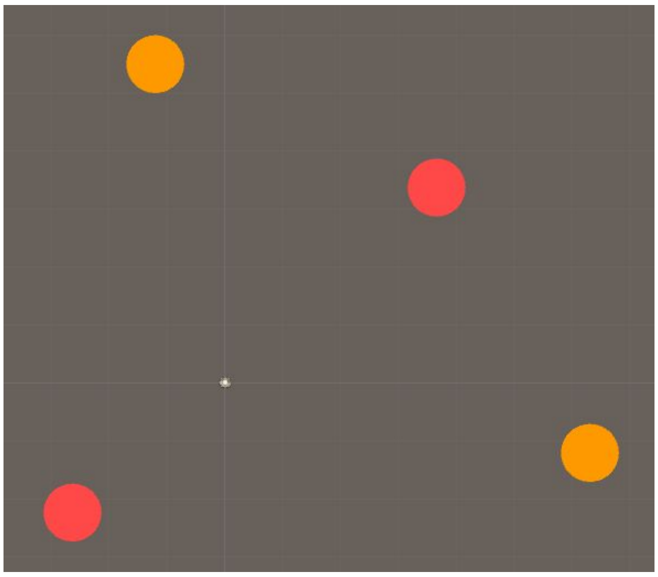
7 Experiments

7.1 The test environments

Several environments were used in this project to test different levels of learning.

7.1.1 Simpel navigation

The firs and simplest environment is a navigation environment much like the gridworld, but since the agent must learn to master a continuous state space there are no grid and all state-values are continuous. The state-values consists of x and y position of the agent which can choose between the actions up, down, left and right. In the test environment there can be placed a number of reward-objects and punishment-objects. If the agent gets within a certain distance of an object a reward of 1 or -1 is returned respectively.



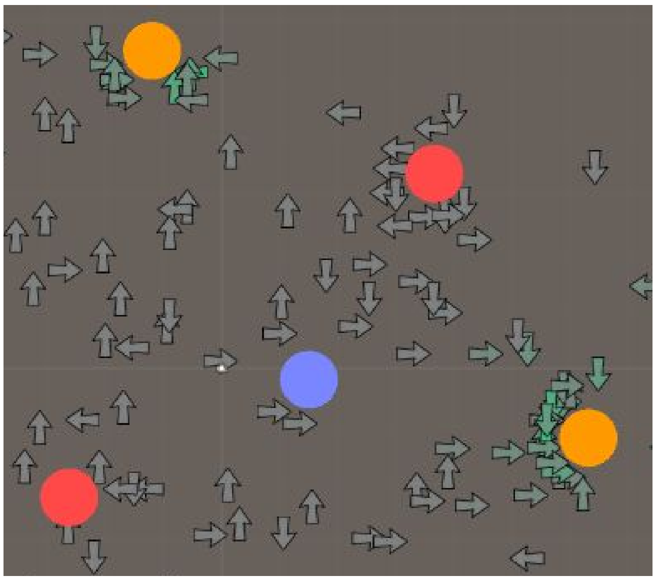


Figure x:The two figures showing screenshots from the navigation environment with the yellow circles being reward-objects and red circles being punishment-objects. The blue circle represents the agent. The arrows shows the placed q-points.

The environment can also be expanded to 3 dimensions.

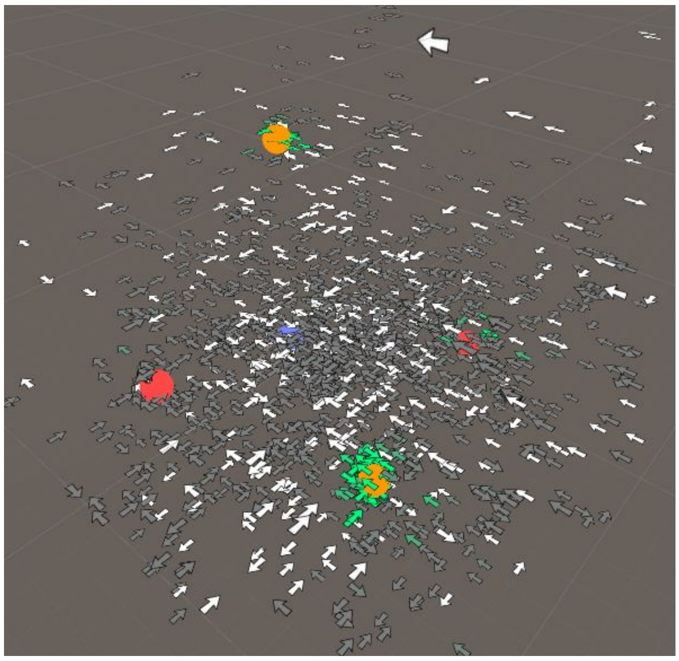


Figure x: Screenshot from the navigation environment expanded to 3 dimensions

7.1.2 Coin pickup

The coin-pickup environment is also quite simple, as this only consists of a 2 dimensional states-pace being angle and distance to the nearest coin. Here the agent is controlling a tank, which can use the 3 actions: turn right, turn left and move forward. The environment is swarmed with coins which respawn at a new random location upon pickup. The challenge with this environment is, that there is a local optimum where the agent can just continue forward and still pickup coins.

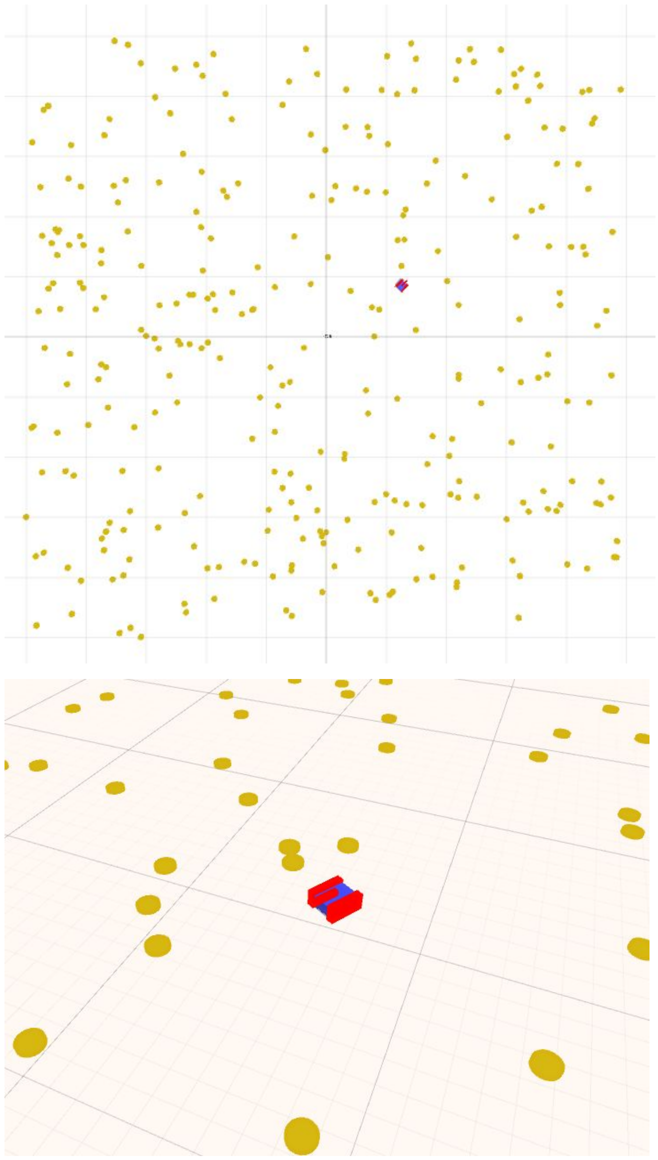


Figure x:Screenshots from the coin-pickup environment

7.1.3 Pac-Man

Pac-Man is a classic machine learning environment, which was used in a previous project together with feature based learning agents. Today Pac-Man is often used with neural network given raw pixel-input. Pac-Man is a good test, as it contains different levels of strategies and can be simplified:

level 1: pickup the coins and run away from a ghosts
level 2: include use of power pallets
level 3: avoid getting trapped by multiple ghosts
level 4: learn the different movement patterns of the ghosts and time when to use the power-pallets. The coins are also pickup in an efficient order.

In Pac-Man the agent has to navigate the maze collecting coins and avoid being caught by the ghosts. Additional elements can give points such as eating risky bonus-fruits or consuming one of the 4 power-pallets, which are located in each corner. The power-pallets turns the ghosts into a scared-state in where they can be consumed for extra points (gameinternals 2010).

In this project the pac-man-level is almost an exact replica from the original, with the exception of the ghost house and warping-tunnels.

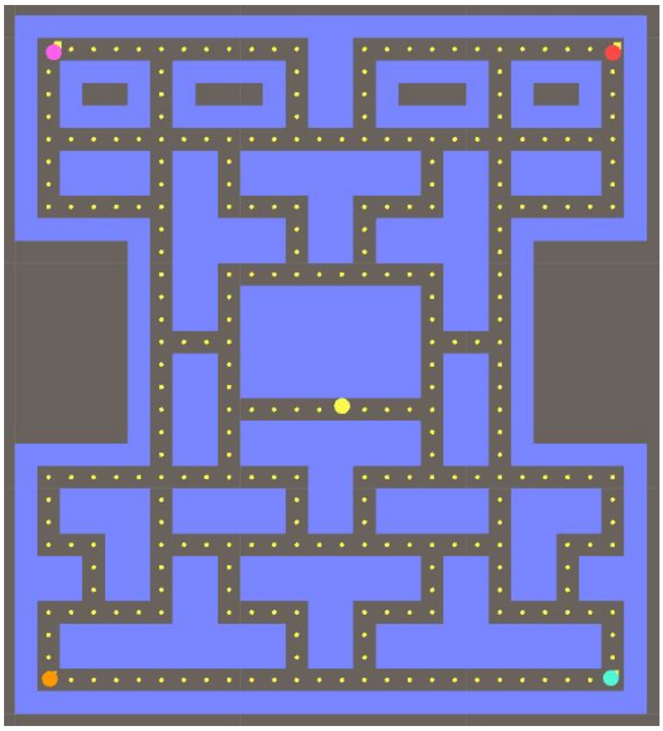


Figure x:Screenshot from the Pac-Man environment.

Movement rules

The entire pacman maze is build on top of a grid where there are strict rules to the movement of both the ghosts and pacman. The most noteworthy are that the ghosts can never turn around unless scared and does only decide where to go next when entering the center of a crossway. Pac-Man can always turn around and change direction as long as not choosing an action pointing towards a wall. Pac-Man can furthermore put an action in queue, so that it is performed as soon as possible. In this version of the Pac-Man environment Pac-Man moves a little faster than the ghosts. This would change in the original, as the levels progressed.

The four ghosts hunting Pac-Man furthermore moves in very specific patterns. In this environment only the two first ghosts are utilised, which are the red and pink. When a decision about which direction to turn is necessary, the choice is made based on which tile adjoining the intersection will put the ghost nearest to its target tile, measured in a straight line. This means that the ghost not always chooses the shortest route. The different ghosts shows different behaviours as they aim for different tiles. The red ghost aims directly for the tile containing Pac-Man leading to a direct chase. The pink aims 4 tiles ahead of the direction Pac-Man is going leading to an ambushing effect (gameinternals 2010).

The agent is rewarded by a small value on coin-pickup, a larger on consumption of a ghost and a negative reward on collision with a ghost.

The environment is reset on emptying the maze for coins or hitting a ghost.

State values

<u>For the general agent</u>

The state values returned from the environment is consisting minimum of 9 values with following meaning:

s[0](s): angle to nearest coin
s[1](s): dist to nearest coin
s[2](s): angle to ghost[n]
s[3](s): dist to ghost[n]
s[4](s): state of ghost[n]
s[5](s): wall ahead in direction 0
s[6](s): wall ahead in direction 1
s[7](s): wall ahead in direction 2
s[8](s): wall ahead in direction 3

<u>For the FB-agent</u>

The number of state values are greatly reduced when using a FB agent, as all the transitions are calculated in the environment. Here the agent only asks to get the state values for a specific action when in a specific state. The features are as following:

f0(s,a): dist to nearest coin
f1(s,a): dist to hunting ghost[n]
f2(s,a): dist to scared ghost[n]
f3(s,a): wall ahead[n]

7.2 Different Experiments

7.2.1 Success of the q-point agent

The flexible q-table learner managed to find routes through the navigation environment both in 2 and 3 dimensions, but the efficiency of the routes is hard to measure.

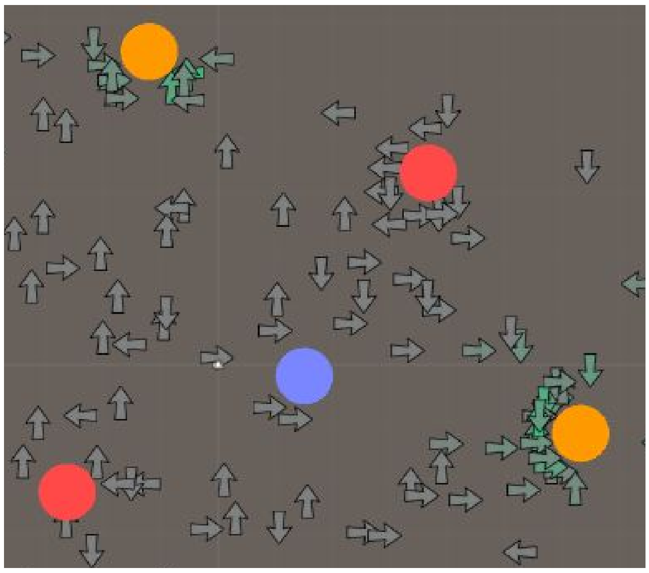


Figure x:Screenshot from the navigation environment test.

In the coin-pickup environment with the q-points pre-placed the agent performed almost as well as the pre-programmed "optimal" agent as shown below, but when responsible for managing the q-points itself it only collects one third of the amount of coins compared to the pre-programmed.

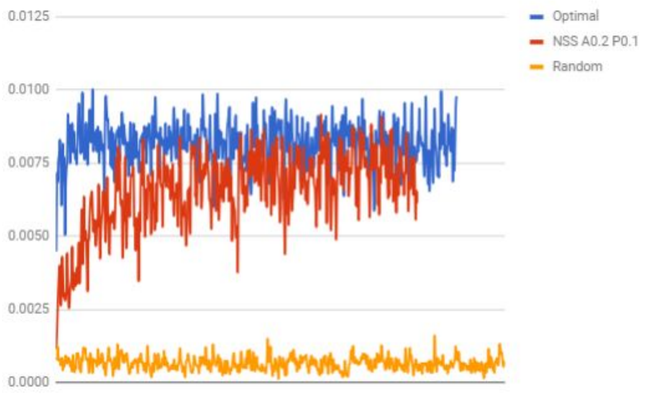


Figure x: Graphs showing difference between the pre-programmed agent (Optimal), the q-point learner provided q-points at start (NSS) and the random baseline in the 1 ghost Pac-Man environment.

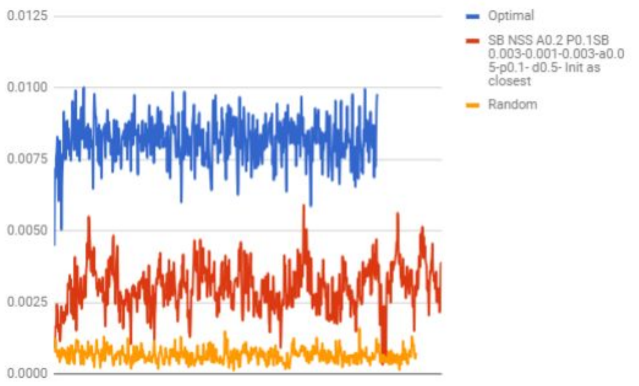


Figure x: Graphs showing difference between the pre-programmed agent (Optimal), the q-point learner (SB) and the random baseline in the 1 ghost Pac-Man environment.

When the state space is this simple, the q-points can also be viewed the same way as within the navigation environment. Below is displayed the q-points of the pre-programmed agent which shows to turn right when the angle to the closest coin is negative, left when it is positive and move forward when within 10 degrees of dead on.

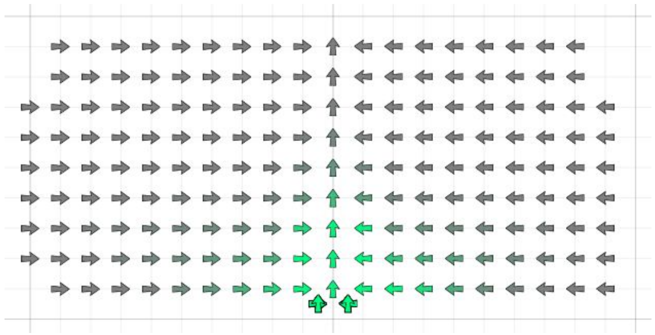


Figure x: Figure displaying the q-points for an agent who has mastered the coin pickup environment.

Below is the learned policy when the q-points are created on beforehand.

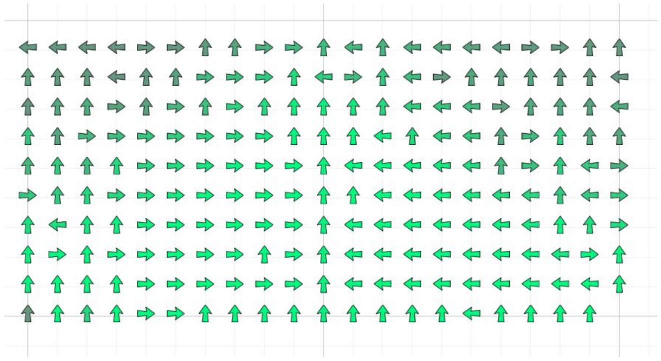


Figure x: Figure displaying the q-points for an agent with pre-created q-points with no step scaling policy 0.1 - 9298501 steps and average between 0.007 and 0.008.

In the Pac-Man environment, the q-point agent was hold up against the FB-agent and a random agent, which set the baseline. Here the agents just have to face one ghost.

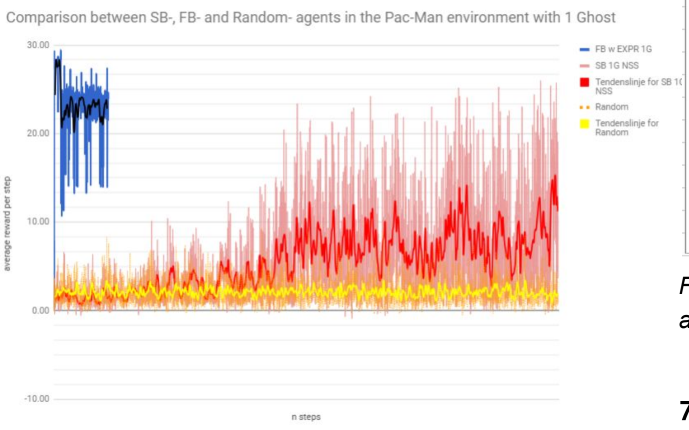


Figure x: Graphs showing difference in earned reward between the q-point learner(SB), FB learner and the random baseline.

When facing two ghosts

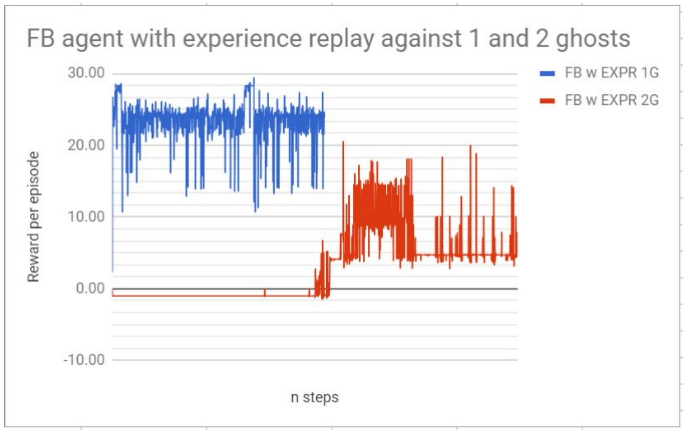


Figure x: Graphs showing the difference between the FB learner facing 1 ghost(1G) and 2 ghosts(2G).

Here it is clear, why FB-agents can get into problems, as it performs significantly worse when facing two ghost contra just one. The problem being, that it is not possible for the agent to recognize a trap unless given very specific features. In this case the q-point agent should have an advantage.

As seen on the figure below, the q-point agent performs as well as the FB-agent when opposing two ghosts, it even seems to be the case that the q-point agent surpasses the FB-agent and continues to increase the average reward though extremely fluctuating and slowly.

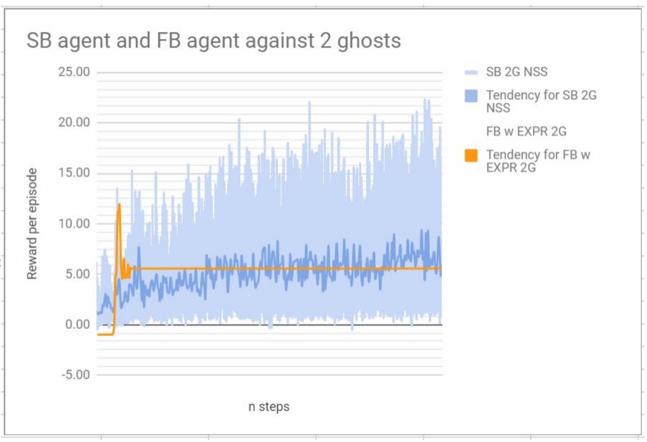


Figure x: Graph showing the tendency lines for the FB agent and q-point agent (SB)

7.2.2 Step Scale problem

Challenges of unevenly placed q-points

Another challenge was that the q-points are placed in different densities. This meant that the q-points which are further away from the goal gets the same credit as the q-points close to the goal. This effect can be seen on the figure below where the green arrows form a circle with the low density placed q-points are dragged out. If optimal, the circle would be perfectly round no matter the density, and the goal would be at center.

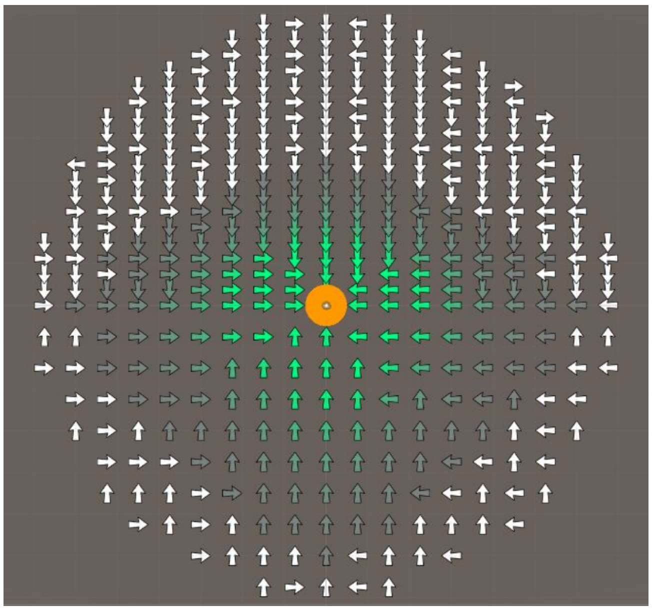


Figure x: Screenshot from the simple navigation environment, where q-points have been instantiated with variating proximity. The yellow dot is the goal and the greener the arrow, the higher the q-value. Discount 5, threshold 0.02

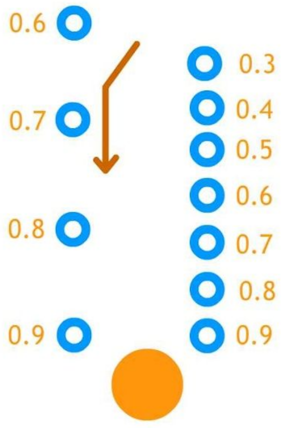


Figure x: Illustration showing how the q-values gets scaled wrongly and the agent therefore will choose the path to the left towards the yellow goal.

This means that the agent prioritises routes with states spread out wider, as the reward has not been discounted as many times. This is not good, as a less dense route means a more insecure route, and can result in unnecessary long routes.
One method of scaling the q-value to accommodate for the density is to calculate the average number of actions taken in each state for a given action. This takes up extra memory, but seems to be a possible method for addressing the problem.

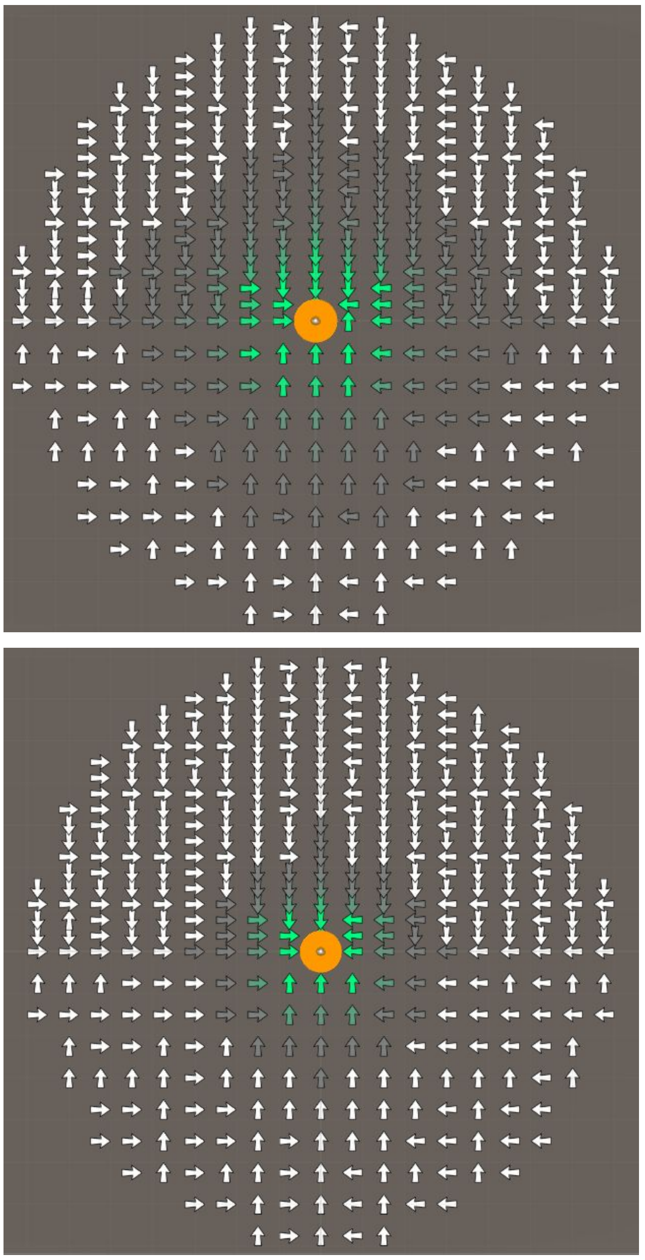


Figure x: Here the q-value is scaled with the average number of actions. To the right the scaling is quadratically

Another solution is to use eligibility traces where the learningrate is scaled with the total number of steps for that episode.

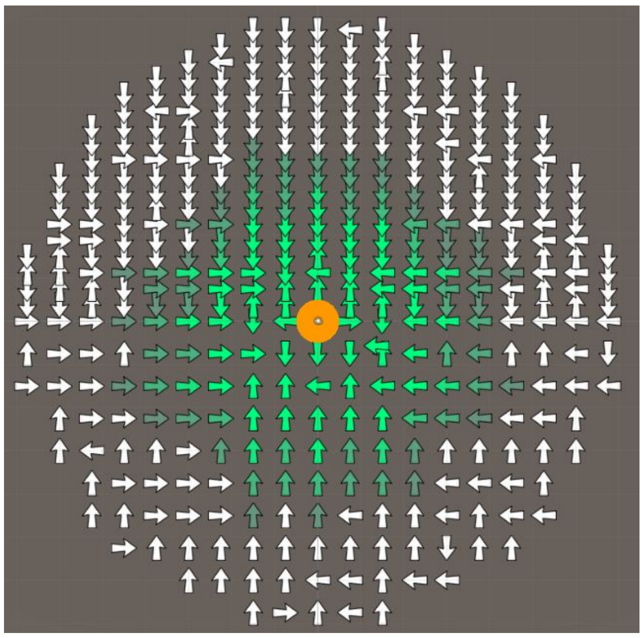


Figure x:Figure illustrating the step-scaling problem possible solved by eligibility traces.

Effects in other environments

The effects of using step scaling (WSS) uppersite no step scaling (NSS) seemed to greatly handicap the agent in the coin-pickup environment as seen on figure x and x, but did not seem to do much difference in the Pac-Man environment as seen on the graphs below.

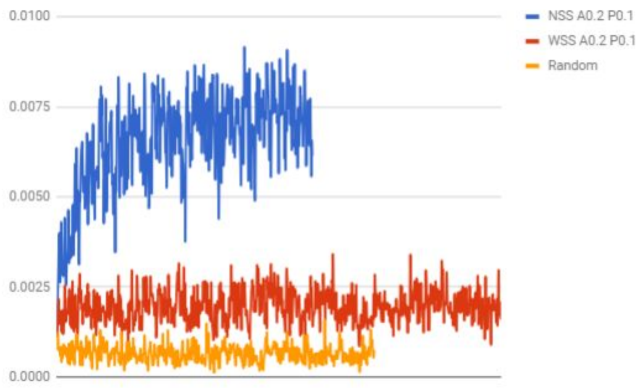


Figure x: Graphs showing the average reward of the q-point agent with (WSS) and without(NSS) the use of step scale.

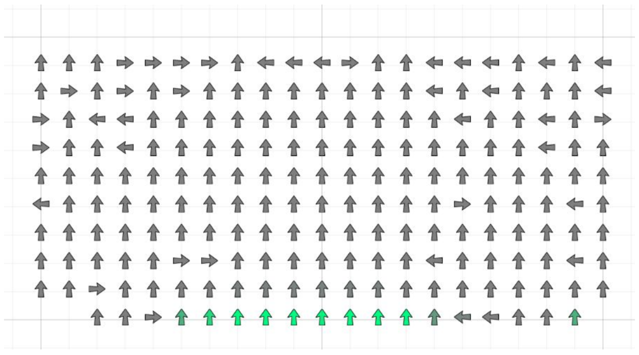


Figure x: Figure displaying the q-points for an agent with pre-created q-points with step scaling.

Adding the Eligibility traces also only seemed to handicap the agent in the coin-pickup environment as shown below both in shape of the average reward graph and q-point visualisation. The q-point figure shows that the agent using eligibility traces is confused.

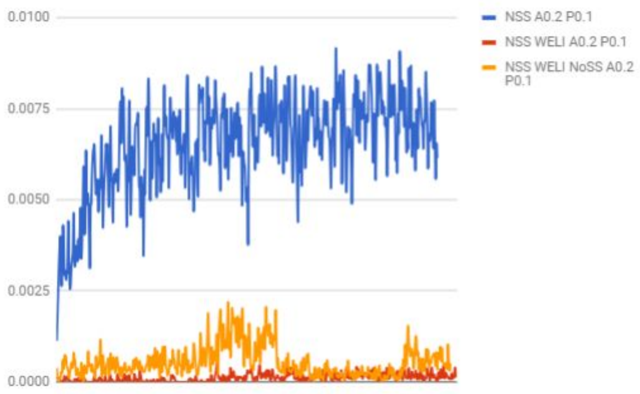


Figure x:Graphs showing the q-point agent without eligibility traces (NSS), with normal eligibility traces(NSS WELI NoSS) and eligibility traces which is step scales (NSS WELI)

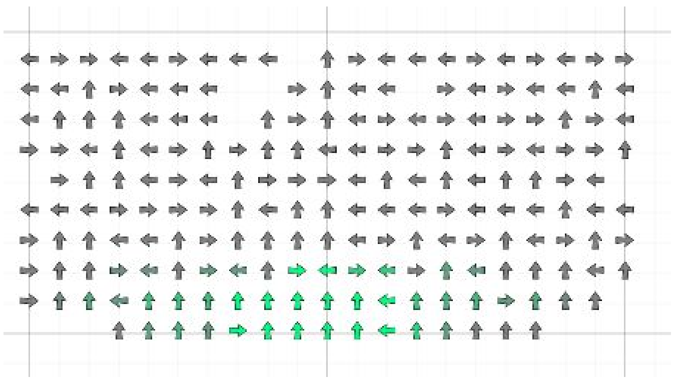
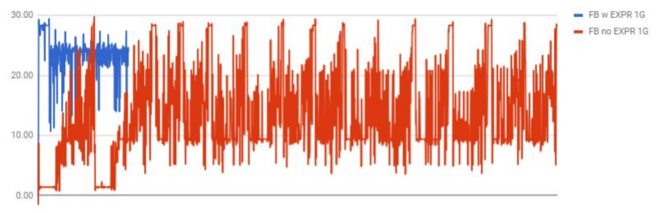


Figure x: Figure displaying the q-points for an agent using eligibility traces

7.2.3 Catastrophic forgetting

The Q-point agent and Q-tables should be free of catastrophic forgetting, but not the FB-agent if not paired with experience replay as shown below. The graph clearly shows how the agent learns a good policy and then falling back over and over.



8 Discussion

The implementation of the Q-point agent was with mixed success, as the agent was capable of placing and managing q-points in the navigation environment and to some degree in the coin-pickup environment. It was clear that the learning was a lot more stable when the q-points were placed on beforehand, which leads to believe that the q-point management algorithm is far from optimal. It also performed poorly in the Pac-Man environment compared to the FB agent using experience replay. It performed almost as well as the FB agent when facing two ghosts, but both agents should be capable of performing better.
A possible problem was discovered with the q-point method, as the q-values were discounted further when q-point density were lowered. Methods were investigated to counteract the tendency as both a step-scale and eligibility traces was implemented, but with no positive results when facing the environments. For future work a more successful method should be found.
A better result could possible have been achieved with the use of a periodic/epoch update where a binary reward is given upon completion or the total reward. Whether an episode/epoch update would still be fitted for game AI could furthermore be investigated. To encourage faster completion the reward could be divided with number of actions taken. The Q-point agent could also have benefitted from being compared to an implementation of a Q-table agent and a neural network -agent. Different algorithms for managing the q-points should also be investigated. An example could be to have an algorithm analysing the different data points for then to place the q-points at optimal positions.Future work could also include experimentation with experience replay together with the Q-point agent and even a mergure with the HER agent.

9 Conclusion

The implementation and test of the novel q-point agent was with varied success. A proof of concept was shown, as the agent was capable of near mastering a navigation environment and coin-pickup environment. The agent was though held back by the poor q-point management and possibly also update. The q-point agent furthermore had problems outperforming the FB agent in the Pac-Man environment where it should have advantages. It seems that the agent has some fundamental challenges which needs to be overcome for the project to be successful, which leads a lot of space for further development and experimentation with other update models and state management.

10 References

Andrychowicz, M. et al., 2017. Hindsight Experience Replay. In I. Guyon et al., eds. *Advances in Neural Information Processing Systems 30*. Curran Associates, Inc., pp. 5048–5058.

de Bruin, T. et al., The importance of experience replay database composition in deep reinforcement learning.

gameinternals, 2010. Understanding Pac-Man Ghost Behavior. *GameInternals*. Available at: http://gameinternals.com/post/2072558330/understanding-pac-man-ghost-behavior [Accessed May 31, 2018].

Ioffe, S. & Szegedy, C., 2015. Proceedings of the 32nd International Conference on Machine Learning.

Mnih, V. et al., 2015. Human-level control through deep reinforcement learning. *Nature*, 518(7540), pp.529–533.

Silver, D. et al., 2016. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587), pp.484–489.

Sutton, R.S. & Barto, A.G., 2012. *Reinforcement Learning: An Introduction*, Cambridge, Massachusetts London, England: The MIT Press.

Zhao, D. et al., 2016. Deep reinforcement learning with experience replay based on SARSA. In *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*. Available at: http://dx.doi.org/10.1109/ssci.2016.7849837.