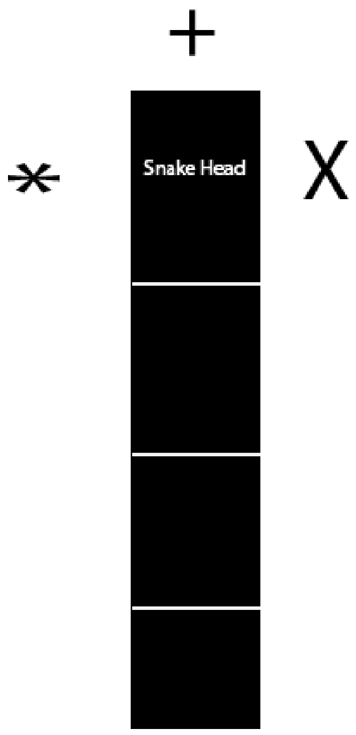
Saji Tibi 207905407

Linear Policy

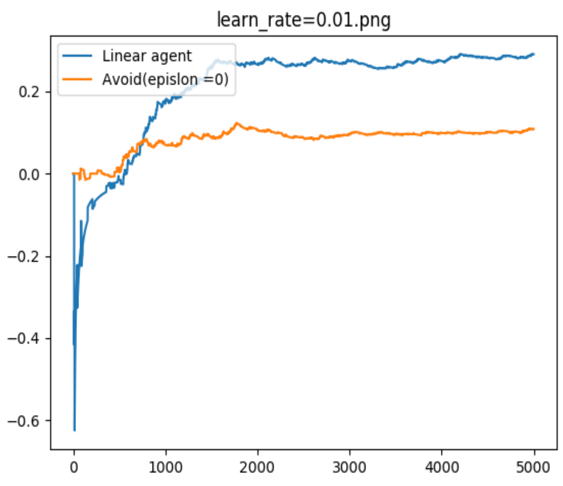
Board Representation:
I’ve used the 3 tiles, in front, to the right, to the left of the snake and based on whats found in these tiles I learn the linear function so I can approximate it.
Each of these tiles are represented as vector of size 11(so it can represent each of the [-1,9] values can be found on board)
For example:

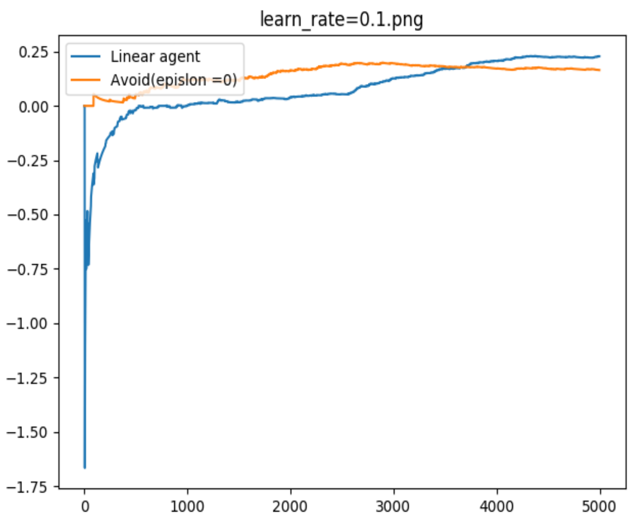


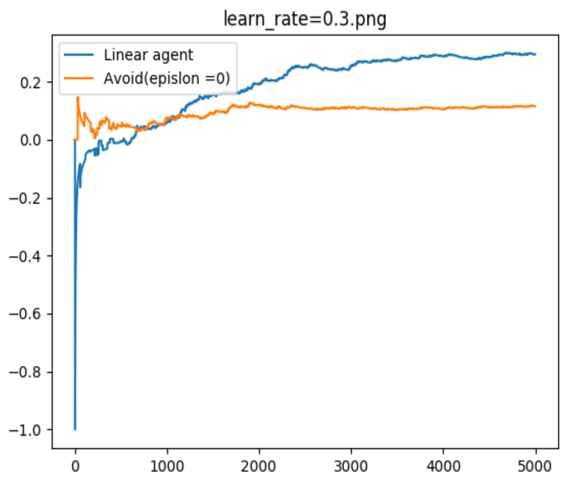
In case of * = 6, + = 5 , X =8, we would get vector [0,0,0,0,0,1,0,0,0,0,0] for left and so on.
This vector would multiplied by features vector and other vectors as well, the action that corresponds to highest value of multiplication would be chosen as next step.

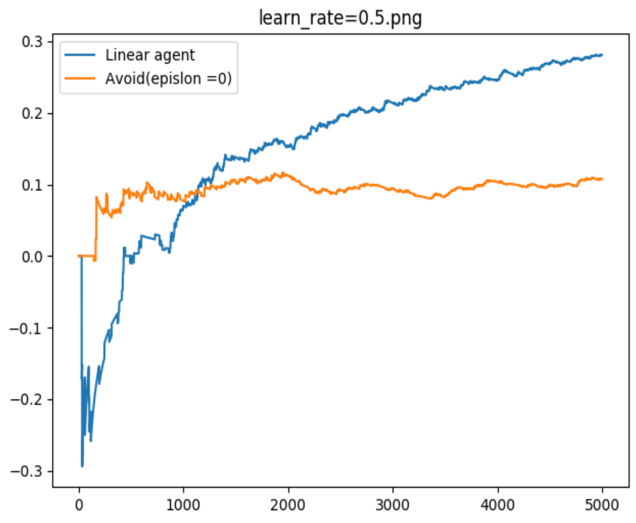
Learning rate: I've tried different values as seen below and the best value was LT = 0.1.
(Note that I've ran multiple times each LT value, graphs below seemed to be the average)
estimated run with 5000 rounds ,score scope 1000, pat = 0.005 plt = 0.01

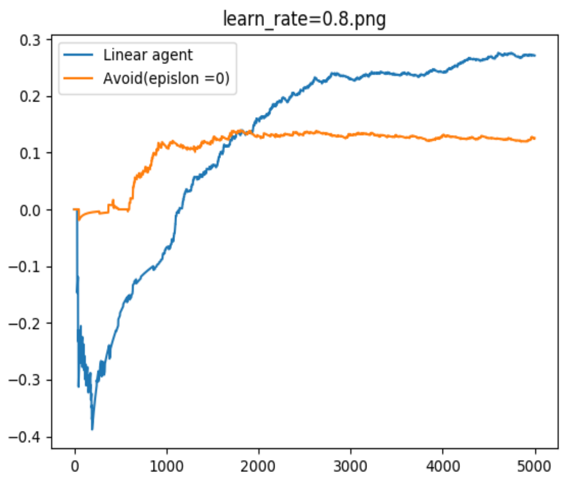
```
Please not that all graphs are made such that each point (x,y) represent x=round,
y= mean of rewards until this round and not last 100 rounds.
```

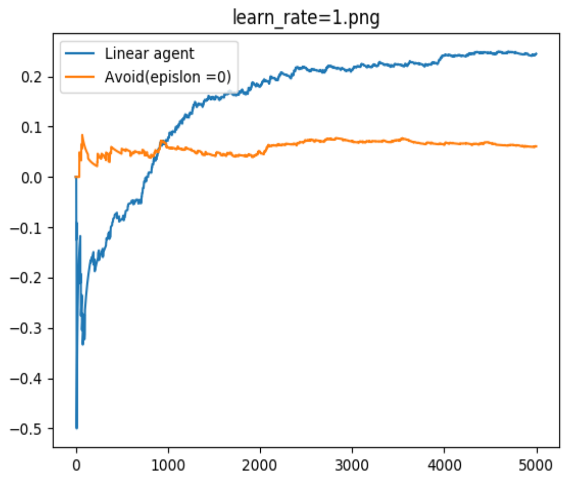




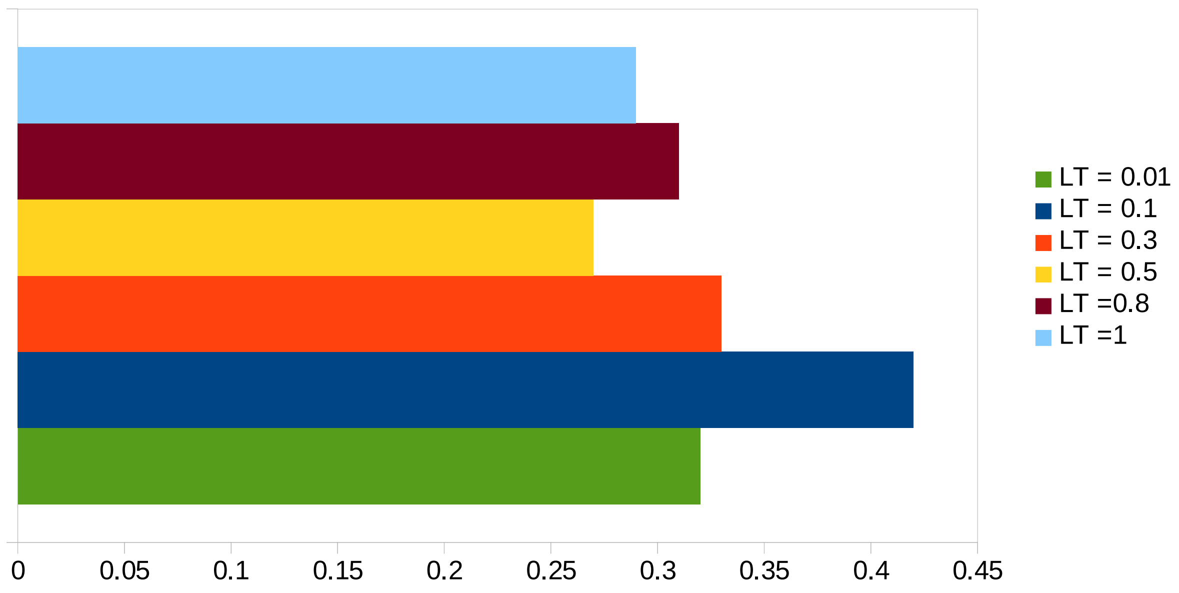






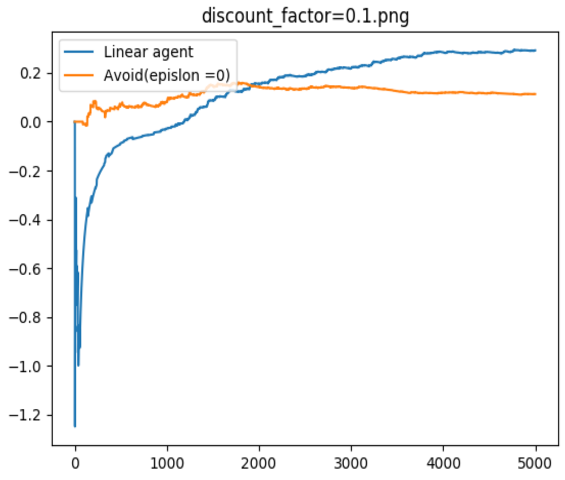


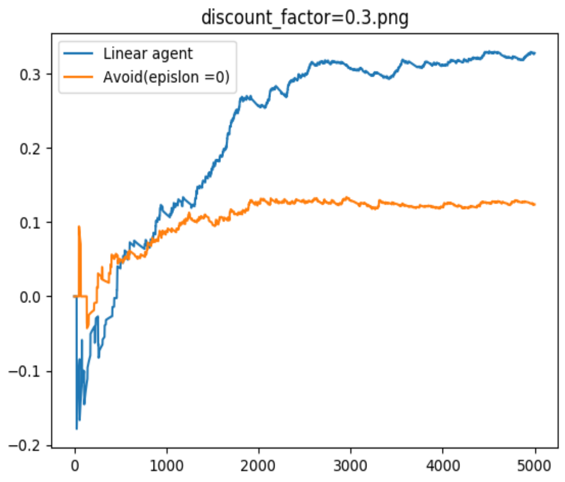
Average on 5 runs per value

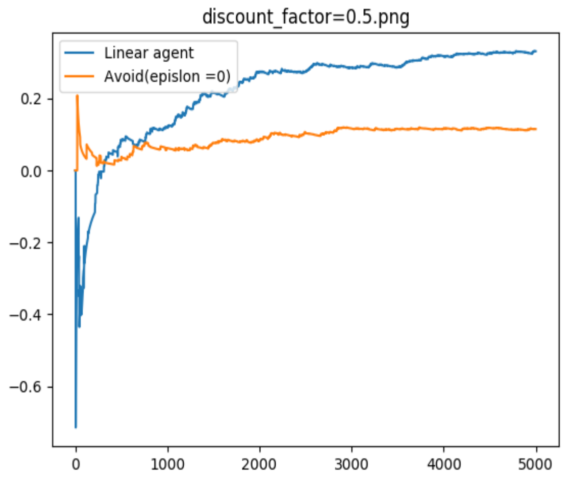


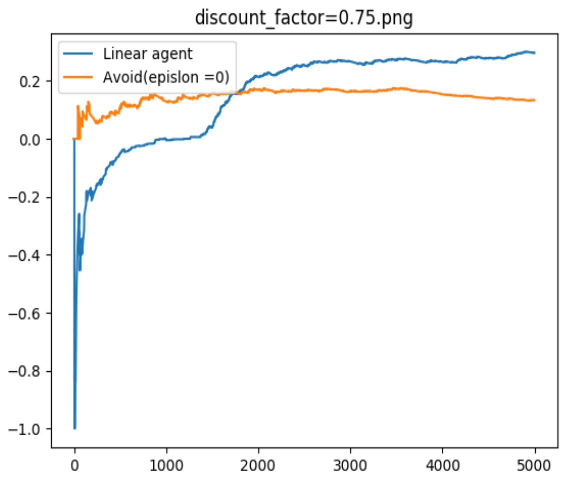
Some values had reached high peak like 0.8 but in the long run they get worse, 0.1 is the best match for the given evaluation specifics.

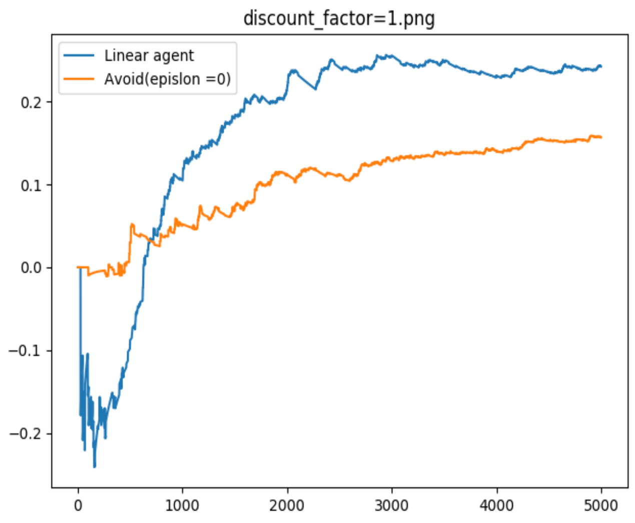
as for discount rate I've choose 0.1 also I made my choice after running various values multiple times. Also since my linear policy only looks at whats a head of it small discount will give better results.











As we can see above high discount rate cause the learning to be less stable hence we look for future rewards with an algorithm that doesnt plan ahead.

<u>Exploration-exploitation trade-off:</u>
Running various parameters led to believe that dynamic epsilon is the best, hence if we have constant epsilon, if its high then probably we are not exploiting as we should and if its low then we are not learning new things (exploring), so start with epsilon 1 and decreasing each 200 round by 0.05 until we reach 0.01 then we stop deceasing gave best results.

Custom Policy

<u>Board Representation:</u>
n the custom policy similar to linear but instead of taking only 3 tiles, I take 3 tiles to left and to right along with more 7 tiles for height of 5

	X	X	X	X	X	X	X		
	X	X	X	X	X	X	X		
	X	X	X	X	X	X	X		
	X	X	X	X	X	X	X		
	X	X	X	X	X	X	X		
	X	X	X	Head, direction forward	X	X	X		
				Tail					

All the X mark what I take as an input, this way we can represent the board with in clever and more expressive way.

<u>Custom Model Learning:</u>
I choose to use Deep Q-Learning as for second agent, the reason I choose such model is because the simplicity of implication in the game, no need to hardcore how to find paths to positive food or such also this way we don't need to care about the values each object represent, suppose one run X = 5, next run X=6, using DQL, Neural network for reinforcement learning.
The network I choose gets an input as explained above, for each tile of the 41 tiles there is a vector of size 11, so overall an input for the network would be of size 41*11 = 451, followed by two hidden layers the size of those layers isn't exactly technically chosen, I've tried some different variations and those sizes worked best. I also used this reference.
The algorithm in this agent uses this equation as ‘loss’ function:

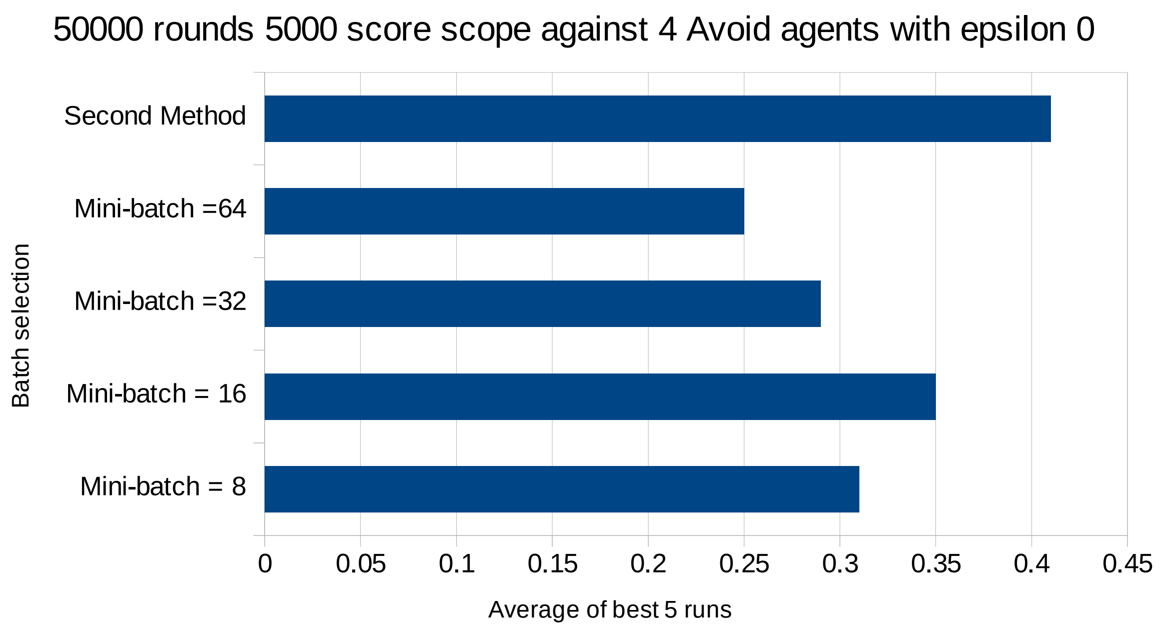
$$loss = \Big(\underbrace{\overset{\text{Reward}}{r} + \overset{\text{Decay Rate}}{\gamma} \max_{a`} \hat{Q}(s, a`)}_{\text{Target}} - \underbrace{Q(s, a)}_{\text{Prediction}}\Big)^2$$

the target is calculated using previous reward along with new state and max action , the prediction is of the previous state and action which is calculated by using neural network prediction of the previous state representation.

Exploration-exploitation trade-off:
I’ve used ε-Greedy Exploration with ε=1 in the beginning and each 200 round I reduce its value by 0.005 until we hit the minimum ε which is 0.001, I've tried various values to start with and with different reduction speed, those parameters worked out best for me, giving lower rates of reduction will cause a lot of random movements even after we have good model in hand and would be waste. Same goes with discount rate I've settled on 0.9 since future rewards are important(below there is an explanation with graphs)

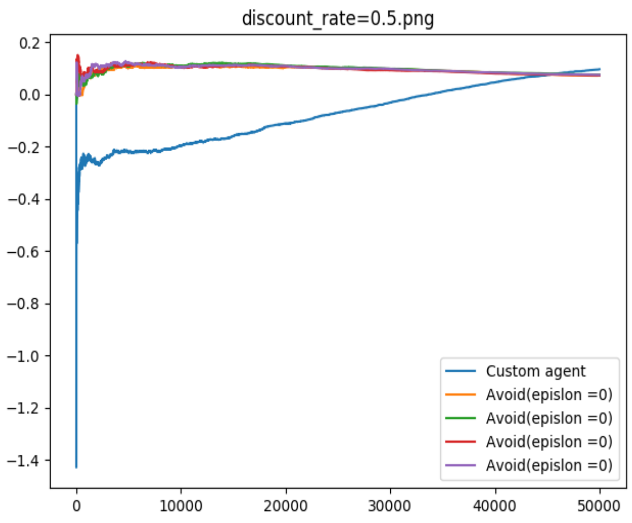
The learning and results:
* First method: I tried recording the game in buffer of various sizes, and using mini batch on the buffer to learn, however this method was successful but not very, since each 5 rounds we get to learn taking a mini batch of size>5 would probably have old samples we already saw if we train each 5 rounds, and if we train per x time suppose each 25 rounds, if we take mini batch of size 25 sampled randomly from the buffer of size>> 25 we would probably miss some samples when it fills up(some samples might never go to the training) and if the buffer size is close to 25 then its not really a mini batch its only a batch training of big size which cause problems.

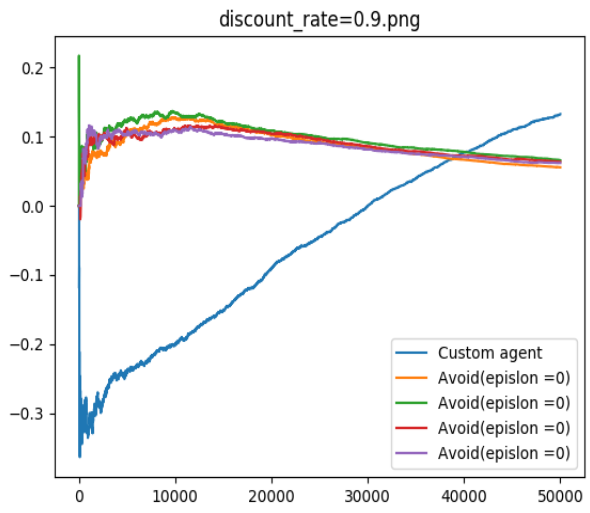
* second method: Instead of taking mini-batch, we train the network each 5 rounds with the very recent 5 rounds this way we guarantee each sample is seen in network and we have small input of samples, only 5, so the trade-off between memory efficiency and speed is more successful in our situation this method gave better results.



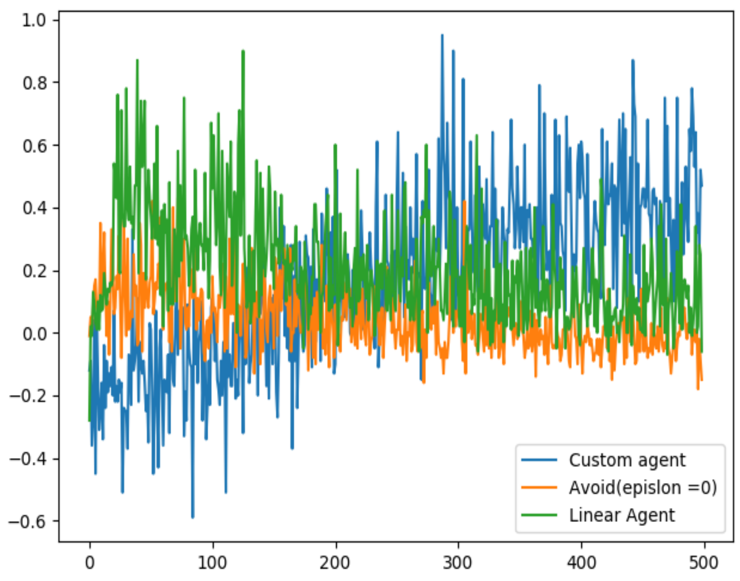
In custom agent I didn't change the learning rate, I've tried several values but without specificity the learning rate Adam optimizer performed better.

As seen below , the network takes time to start actually winning, around 35K rounds however adjusting discount rate, learn time, epsilon and other parameters gave early winnings, around 3K but in the long run it give worse results than current implementation, in conclusion acute slope (faster to win) in the beginning caused fast convergence to low average win rate. (we care about final results so those algorithms aren't quit helpful)





Note how lower discount gave faster winning and got close to other agents but it won by small difference while high discount rate had at first lower results but in the end it won by huge difference.
Below graph of 50000 rounds 5000 score scope, here we have each point represent last 100 round and not all rounds until
the point.



Its clearly obvious that custom agent started with high loss and then performed better while linear agent had less step of improvement over time, the avoid agent stayed almost the same.

Other solutions:
* I had tried using more complex board representation but it required more time to learn and predict I also achieved better results with more complex network architecture but also it needed more time so I settled on the current configurations as best suited for the purpose.
* A good extra solution would be adding more than whats found on nearby tiles, I tried using some features from the linear learner to find what values represent good food, and adding an angle as an input to network between snake head and the good food, however this solution also required some extra time and couldn’t be done in given specs.