BananaFeast report

Learning Algorithm

QNetwork

The QNetwork is modeled with pytorch and is a simple Neural Net consisting of three fully connected layers of the sizes 64, 32 and 16 respectively. The layers are connected in the order 64>32>16 and the first two are activated with RELU.

Another model with five fully connected layers was tested, but failed to achieve better results although they were satisfactory. Dropout layers were also tested but delivered sub par results and had no visible effect on model robustness.

DQN Agent

The learning algorithm is modeled after the Double DQN as proposed by Hasselt. The Agent uses three main components and has four main functions.

Parameters:

n_episodes (int):	maximum number of training episodes
max_t (int):	maximum number of timesteps per episode
EPSILON (float):	Exploration Rate
eps_start (float):	starting value of epsilon, for epsilon-greedy action selection
eps_end (float):	minimum value of epsilon
eps_decay (float):	multiplicative factor (per episode) for decreasing epsilon
BUFFER_SIZE (int):	Size of Memory Buffer
BATCH_SIZE (int):	Size of memory batch trained on
GAMMA (float):	discount factor for future rewards
TAU (float):	for soft update of target parameters
LR (float):	learning rate
UPDATE_INTERVAL (int):	how often to update the network
model (boolean):	use small (True) or big (False) Net

Components:

The first component is the agent’s **memory**. This is a memory buffer that saves the experiences of the agent. For efficiency, the memory is implemented using a deque. The memory component handles the saving of experiences as well as serving batches of experience for training.

The **localNet** is the local Q Network of the Agent expressed by the QNetwork Neural Net. It is updated every update_interval and contains the agents local brain. It’s main purpose is the selection of actions based on the current state as well as updating the targetNet. The localNet is updated using the ADAM-optimizer and by factor of the Learning Rate (LR).

The **targetNet** is modeled identically to the localNet. It is mostly used to calculate the values of the next steps the agent may take. It’s paramerters are updated indirectly using the localNet and the soft update factor TAU. It is also updated in the same interval as the localNet

Functions:

get_action returns the action the agent should take in the current state. The agent estimates the best choice by feeding the state to the localNet. The resulting action is the greedy action to take in the current state. With the chance of EPSILON, the agent takes a random action. This together results in an epsilon-greedy action selection.

Step updates the agents state (usually after an action is taken). It handles the memory saving as well as triggering the learn function if the update_interval is reached.

Learn updates the localNet and targetNet. The localNet is updated using the double DQN approach, calculating the best actions using the localNet and target Q Values of those actions using the targetNet. This improves performance by reducing overestimation. The localNet is then updated using the ADAM optimizer and then calling update_targetNet.

update_targetNet updates the targetNet by the softupdate through TAU*localNet.

Training

The training is done locally and hyperparameters were tuned using both intuition and a grid optimization. The optimal chosen parameters are explained in the following section.

"n_episodes" : [1500]

The number of episodes was set to 1500. Although the Agent often already achieves the required score by 1000 – 1200 episodes, the performance does improve to a score of 17 at usual maximum. The Agent was mostly trained using two thousand episodes to see long term stability as well as potential performance gains. Nevertheless, the agent rarely improves significantly after the 1500 episode mark and this was chosen as the sweet spot between best performance and training ressources.

"BUFFER_SIZE" : [int(1e5)]

The Buffer size was set to 100000. Increasing the buffer size had no traceable improvement, so this was mainly left unchanged.

"BATCH_SIZE" : [64]

The batch size was set to 64. Both sizes of 32 and 128 were tested but offered no real improvements.

"GAMMA" : [0.99]

Gamma was set to 0.99. After testing, both values of 0.9 and 0.999 proved to decrease the performance of the agent, albeit only by small margins. Agents with large GAMMA seemed to favor efficient movements less. Low GAMMA values also decreased performance. Maybe this is analogous to the traveling salesman, where getting the most immediate reward does not usually result in the most efficient path.

"TAU" : [1e-3]

TAU was one of the more relevant parameters. Setting it to higher or lower values did not decrease maximum performance, but the models had worse learning gains on average, leading to less reliable results.

"LR" : [5e-4]

The Learning Rate was the single most relevant parameter. Especially low LR of <= 0.005 resulted in the Agent topping off at scores of 8 – 11. High LRs performed a bit worse than the chosen value.

"UPDATE_INTERVAL" : [20]

The update interval behaved fairly stable for values between 15 and 50. High Values like 100 decreased performance.

```
"model" : [True]
```

A very simple Model, described earlier, showed satisfyingly good performance. Choosing a model with more layers or dropout layers decreased the performance and stability. Especially the dropout layers were expected to give the agent an edge by improving robustness. This did not happen.

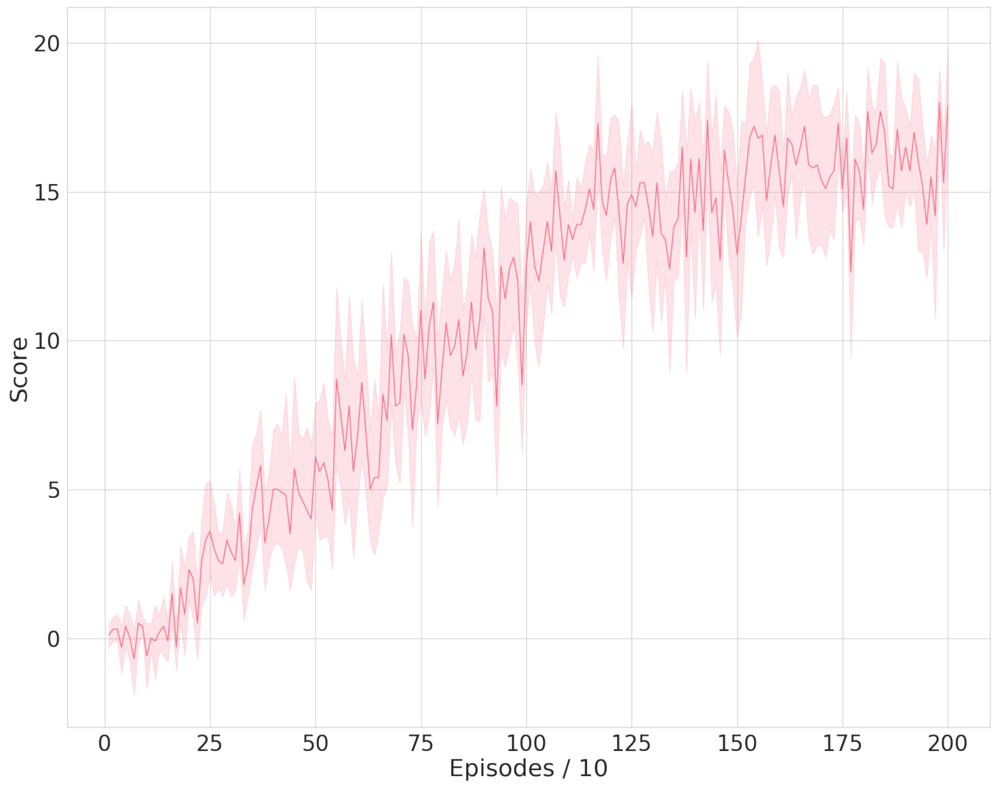
```
"eps_decay" : [0.99]
```

The decay also had a large effect on performance. Lower values led to the agent not learning past the first few hundred episodes and thereby gaining too little variance in the experiences. Higher values kept the agent from optimizing and led to very shallow learning performances.

Performance:

The model is reliably able to achieve a score of 13+. The performance is reached at around 1100 episodes and can increase to an average performance of around 16 – 17 at 1500 – 1800 episodes. There may still be room for optimization in the parameters, although most were tested to a satisfying degree. The Agent does experience some minor and rare problems. It was observed to have trouble when having a banana at both the left and right side of the screen in similar distance. This resulted in the agent jolting to both sides in turn, effectively paralyzing it. This was observed only once.

Plot of rewards



Ideas for future work

The algorithm used is a simple double DQN. This could be expanded by both a dueling DQN and prioritized experience replay, as described in the paper on rainbow. Furthermore, more testing could be done on the tuning of the hyperparameters, especially computing larger combinations. These, however, do take quite an amount of time to process. Lastly, being able to adjust the rewards returned by the environment may be a way to further optimize the the agent. Especially when trying to replicate human behavior, as the actions of the current agent are definitely effective, but far from elegant.