ADVANCED TOPICS IN COMPUTATIONAL INTELLIGENCE
Pacman

Beata Baczynska

May 2022

Contents

1 Introduction

I have used Berkeley's implementation from: http://ai.berkeley.edu/reinforcement.html.

You can find the first code in the section .1. Some functions were already implemented. I've implemented: my_closestGhost(), my_closestFood() and my_closestCapsule(). In the function getFeatures() the features I've added are: closest-capsule (the Manhattan distance to the closest capsule with precision to 10 steps), closest-ghost(the Manhattan distance to the closest ghost with precision to 10 steps), the_closest_ghost_scaredTime(the timer how many moves left when ghost being edible), eat-last-food(if the food that is going to be eaten is the last one), getting-closer-to-food(if the action makes pacman to be closer to food - in case of distance to be greater than 10) and closest-food (this feature was present however defined differently; here it was the Manhattan distance to the closest food with precision to 10 steps). The features already present in the original implementation were: -of-ghosts-1-step-away (number of ghost 1 step away from pacman) and eats-food (binary feature, equals 1 when there is no ghosts one step away and there is a food on the next position). I have extracted all the features for all actions (East/West/North/South/Stop) so at the end I received number_of_features_per_action x number_of_actions features. 8x5=40 in this case.

All the results you can find in the tables at section .4 were obtained after training was done. The weights were fixed, exploration rate and the alpha were set to zero. The results were obtained for the same layout as during the training (smallClassic).

2 About the code

I have used python3.9 and tensorflow 2.4.1.

To run code you should give the execute permission for all the files.

To run the code from terminal use:

For training:

/home/.../bin/python /home/.../pacman.py -p QLearningAgent -x 1000 -n 1000 -l smallClassic

In the -x NUMBER and -n NUMBER the NUMBER should be the same. That is the number of episodes to train. The smallClassic is the name of the layout. The layout should be present in the layouts/ directory.

In this case training for 1000 episodes.

For testing:

/home/.../bin/python /home/.../pacman.py -p QLearningAgentTest -x 0 -n 10 -l smallClassic

Set -x to zero (-x 0), set -n to number of episodes you want to display. In this case 10 episodes will be displayed.

When using PyCharm you can run with parameters set in the Run/Debug Configurations (with you Script path).

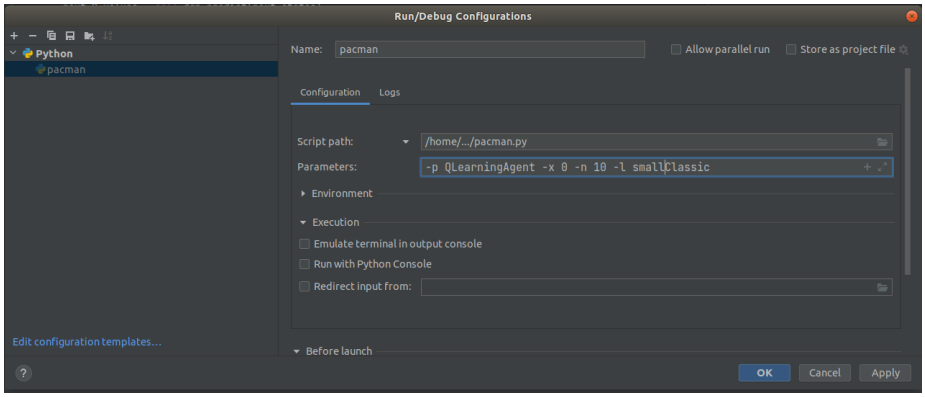

Figure 1: Run/Debug Configurations

3 Experiments

3.1 First experiment

The code (including functions that were already implemented) you can find in section .1.

At this point extracted features were checked. The DQN implementation consisted of single neural network (without target one), replay buffer of size 1000 and the batch size set to 128. The training performance: loss and reward you can find on the figure 28 and 40.

The performance after training is showed at the figure 53.

3.2 Second experiment with slightly changed features

I have simplified definition of the closest-food/ghost/capsule feature. With that I could also delete the feature getting-closer-to-food, as this knowledge could be obtained always with the closest-food feature. After these changes the number of neurons in the first layer could be decreased to 35 (that was the only change in the qLearningAgent function).

```python
def getFeatures(self, state, action):
    # extract the grid of food and wall locations and get the ghost locations
    food = state.getFood()
    walls = state.getWalls()
    ghosts = state.getGhostPositions()
    capsules = state.data.capsules
    features = util.Counter()

    # compute the location of pacman after he takes the action
    x, y = state.getPacmanPosition()
    dx, dy = Actions.directionToVector(action)
    next_x, next_y = int(x + dx), int(y + dy)

    # count the number of ghosts 1-step away
    features["#-of-ghosts-1-step-away"] = sum((next_x, next_y) in Actions.getLegalNeighbors(g, walls) for g in ghosts)

    # # if there is no danger of ghosts then add the food feature
    if not features["#-of-ghosts-1-step-away"] and food[next_x][next_y]:
        features["eats-food"] = 1.0
    else:
        features["eats-food"] = 0.0

    distCapsule, capsule_x, capsule_y = my_closestCapsule((next_x, next_y), capsules, walls)
    if distCapsule is not None:
        capsuleDist = manhattanDistance((next_x, next_y), (capsule_x, capsule_y))
        features["closest-capsule"] = float(capsuleDist/10)
    else:
        features["closest-capsule"] = -1.0

    distGhost, ghost_x, ghost_y, ghost_id = my_closestGhost((next_x, next_y), ghosts, walls)
    if distGhost is not None:
        ghostDist = manhattanDistance((next_x, next_y), (ghost_x, ghost_y))
        features["closest-ghost"] = float(ghostDist / 10)
        s = state.getGhostState(ghost_id)
        features[f"the_closest_ghost_scaredTime"] = s.scaredTimer
    else:
        features["closest-ghost"] = -1.0
        features[f"the_closest_ghost_scaredTime"] = 0

    dist, food_x, food_y = my_closestFood((next_x, next_y), food, walls)
    features["eat-last-food"] = 0
    if dist is not None:
        foodDist = manhattanDistance((next_x, next_y), (food_x, food_y))
        features["closest-food"] = float(foodDist / 10)
        if ((next_x, next_y) == (food_x, food_y)) and (sum(sum(x) for x in food)) == 1:
            features["eat-last-food"] = 1.0
    else:
        features["closest-food"] = -1.0

    return features
```

Figure 2: Feature extraction, closest-food/ghost/capsule feature modified

The performance of training you can find on the figure 29 and 41. The game results is showed at the figure 54.

The average score was slightly worse however the win rate stayed the same. I decided to stay with the features from the second experiment as I believed that it's better to keep less features (to have easier model to train), also the loss and rewards plots showed slightly better scores than the ones from experiment one.

3.3 3rd experiment: exploration probability decay

I have introduced the linear decay for the epsilon parameter (instead of fixed value during all training). With maxepsilon = 1.0 and minepsilon = 0.1 I was updating the epsilon value, after each episode, as presented on the figure 4.

```python
class QLearningAgent(Agent):
    """
      Q-Learning Agent

      Functions you should fill in:
        - computeValueFromQValues
        - computeActionFromQValues
        - getQValue
        - getAction
        - update

      Instance variables you have access to
        - self.epsilon (exploration prob)
        - self.alpha (learning rate)
        - self.discount (discount rate)

      Functions you should use
        - self.getLegalActions(state)
          which returns legal actions for a state
    """
    def __init__(self, actionFn = None, numTraining=100, epsilon=0.15, alpha=0.5, gamma=0.99):

        self.featExtractor = SimpleExtractor()
        self.seen_states = []
        self.dqn = keras.models.Sequential([
            keras.layers.Dense(64, activation="elu", input_shape=[35]),
            keras.layers.LayerNormalization(),
            keras.layers.Dense(32, activation="linear"),
            keras.layers.LayerNormalization(),
            keras.layers.Dense(5)
        ])
        # uncomment for testing
        # self.dqn = keras.models.load_model("weights")
        self.maxepsilon = 1.0 # = None for testing
        self.minepsilon = 0.1
        if actionFn == None:
            actionFn = lambda state: state.getLegalActions()
        self.actionFn = actionFn
        self.episodesSoFar = 0
        self.accumTrainRewards = 0.0
        self.accumTestRewards = 0.0
        self.numTraining = int(numTraining)
        self.epsilon = float(epsilon)
        self.alpha = float(alpha)
        self.discount = float(gamma)
        self.memory = deque(maxlen=1000)
        self.batch_size = 128
        self.learning_rate = 0.0001
        self.loss_array = []
        self.tmp_loss_array = []
        self.reward_array = []
        self.epizode_array = []
```

Figure 3: Q-Learning Agent

```
def final(self, state):
    """
      Called by Pacman game at the terminal state
    """
    deltaReward = state.getScore() - self.lastState.getScore()
    self.observeTransition(self.lastState, self.lastAction, state, deltaReward)
    self.stopEpisode()

    # Make sure we have this var
    if not 'episodeStartTime' in self.__dict__:
        self.episodeStartTime = time.time()
    if not 'lastWindowAccumRewards' in self.__dict__:
        self.lastWindowAccumRewards = 0.0
    self.lastWindowAccumRewards += state.getScore()

    if len(self.memory) >= self.batch_size:
        self.training_step(self.batch_size)

    if self.maxepsilon:
        self.epsilon = max(self.maxepsilon - (self.episodesSoFar / self.numTraining), self.minepsilon)
        print(f"Epsilon: {self.epsilon}")
```

Figure 4: Epsilon decay

Both plots, loss and reward, improved. Also the quality of the Pacman's game. I've stayed with implemented changes.

3.4 4th experiment: add information about last two actions

When displaying Pacman's behaviour for previous implementation, I've noticed Pacman's hesitations - choosing left and right (or top and down) directions alternately. I thought that maybe by adding information of last two chosen actions can help with that.

In the code you can see self.last_action_memory added as well as extended input_shape by extra 10 neurons (5 neurons per one hot-coded action).

```
class QLearningAgent(Agent):
    """
      Q-Learning Agent

      Functions you should fill in:
        - computeValueFromQValues
        - computeActionFromQValues
        - getQValue
        - getAction
        - update

      Instance variables you have access to
        - self.epsilon (exploration prob)
        - self.alpha (learning rate)
        - self.discount (discount rate)

      Functions you should use
        - self.getLegalActions(state)
          which returns legal actions for a state
    """
    def __init__(self, actionFn = None, numTraining=100, epsilon=0.15, alpha=0.5, gamma=0.99):

        self.featExtractor = SimpleExtractor()
        self.seen_states = []
        self.dqn = keras.models.Sequential([
            keras.layers.Dense(64, activation="elu", input_shape=[45]),
            keras.layers.LayerNormalization(),
            keras.layers.Dense(32, activation="linear"),
            keras.layers.LayerNormalization(),
            keras.layers.Dense(5)
        ])
        # uncomment for testing
        # self.dqn = keras.models.load_model("weights")
        self.maxepsilon = 1.0 # = None for testing
        self.minepsilon = 0.1
        if actionFn == None:
            actionFn = lambda state: state.getLegalActions()
        self.actionFn = actionFn
        self.episodesSoFar = 0
        self.accumTrainRewards = 0.0
        self.accumTestRewards = 0.0
        self.numTraining = int(numTraining)
        self.epsilon = float(epsilon)
        self.alpha = float(alpha)
        self.discount = float(gamma)
        self.memory = deque(maxlen=1000)
        self.batch_size = 128
        self.learning_rate = 0.0001
        self.loss_array = []
        self.tmp_loss_array = []
        self.reward_array = []
        self.epizode_array = []
        self.last_action_memory = deque([[0, 0, 0, 0, 1], [0, 0, 0, 0, 1]], maxlen=2)
```

Figure 5: Q-Learning Agent

Results, especially the table at figure 56 showed worse performance of the agent compering to previous approach. I thought that maybe by extending the input layer I should also increase number of neurons in next ones, to allow to extract and code more features and information in deeper layers.

3.5 5th experiment: larger network

For the reason mentioned above, the number of neurons in the first layer were increased (from 64 to 128) also one more layer were added (and number of neurons were tuned).

```
def __init__(self, actionFn = None, numTraining=0, epsilon=0.15, alpha=0.5, gamma=0.99):

    self.featExtractor = SimpleExtractor()
    self.seen_states = []
    self.dqn = keras.models.Sequential([
        keras.layers.Dense(128, activation="elu", input_shape=[45]),
        keras.layers.LayerNormalization(),
        keras.layers.Dense(64, activation="linear"),
        keras.layers.LayerNormalization(),
        keras.layers.Dense(16, activation="linear"),
        keras.layers.LayerNormalization(),
        keras.layers.Dense(5)
    ])
    self.dqn = keras.models.load_model("weights")
    # self.maxalpha = 0.7
    # self.minalpha = 0.5
    self.maxepsilon = None # zmniejszono epsilon ze względu na znaczne zwiększenie liczby epizodów
    # self.minepsilon = 0.1
    if actionFn == None:
        actionFn = lambda state: state.getLegalActions()
    self.actionFn = actionFn
    self.episodesSoFar = 0
    self.accumTrainRewards = 0.0
    self.accumTestRewards = 0.0
    self.numTraining = int(numTraining)
    # self.epsilon = float(epsilon)
    # self.alpha = float(alpha)
    self.alpha = 0.0
    self.epsilon = 0.0
    self.discount = float(gamma)
    self.memory = deque(maxlen=1000)
    self.batch_size = 128
    self.learning_rate = 0.0001
    self.loss_array = []
    self.tmp_loss_array = []
    self.reward_array = []
    self.epizode_array = []
    self.last_action_memory = deque([[0, 0, 0, 0, 1], [0, 0, 0, 0, 1]], maxlen=2)
```

Figure 6: Q-Learning Agent

Obtained results were not better. I decided to come back to the approach 3 (section 3.3).

3.6 6th experiment: done and numTrainingEpsilonDecrease variables

I've noticed the missing 'done' parameter in the formula. I corrected it and also added the numTrainingEpsilonDecrease parameter that I've used in the epsilon decay formula. By setting this parameter to lower value than numTraining it was possible to train the neural network longer when having lower exploration probability rate. I believed that it is beneficial to at the end when learned policy is already very good. I've set the numTrainingEpsilonDecrease to 5000 and the training for the 7000 episodes (however I stooped the training after 6000 because the scores were not longer improving).

```
target_Q_values = rewards + (1 - dones) * self.discount * max_next_Q_values
```

Figure 7: Done added to the equation

The results were impressively good so I have decided to check how the agent works on different layout (mediumClassic).

The agent was playing so long that I've decided to stop the episode. The agent was avoiding ghosts really well however he was not looking for food.

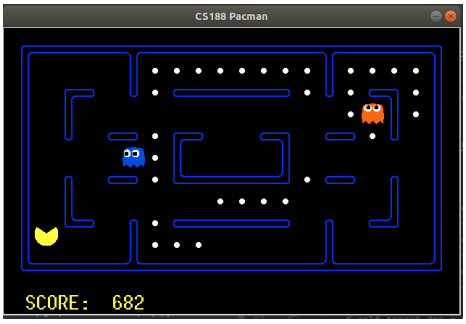

Figure 8: Agent playing on the mediumClassic layout.

3.7 7th experiment: target DQN updated every 100 episodes

Next I implemented the target dqn network. I was updating the weights every 100 episodes.

The result were slightly worse that in the previous approach.

3.8 8th experiment: target DQN updated every episode with theta parameter

I repeated the target network implementation but with updating the weights every episode but with the theta parameter (theta=0.999).

```
def final(self, state):
    """
      Called by Pacman game at the terminal state
    """
    deltaReward = state.getScore() - self.lastState.getScore()
    self.observeTransition(self.lastState, self.lastAction, state, deltaReward)
    self.stopEpisode()

    # Make sure we have this var
    if not 'episodeStartTime' in self.__dict__:
        self.episodeStartTime = time.time()
    if not 'lastWindowAccumRewards' in self.__dict__:
        self.lastWindowAccumRewards = 0.0
    self.lastWindowAccumRewards += state.getScore()

    if len(self.memory) >= self.batch_size:
        self.training_step(self.batch_size)

    if self.maxepsilon:
        self.epsilon = max(self.maxepsilon - (self.episodesSoFar / self.numTrainingEpsilonDecrease), self.minepsilon)
        print(f"Epsilon: {self.epsilon}")

    new_weights = [self.theta * wt + (1 - self.theta) * w for (wt, w) in zip(self.target_dqn.get_weights(), self.dqn.get_weights())]
    self.target_dqn.set_weights(new_weights)
```

Figure 9: Weights update with theta parameter

3.9 9th experiment: Double DQN

I've stayed with DQN updated every episode with theta parameter as described in section 3.8. I've added a double DQN improvement.

```python
def training_step(self, batch_size):

    loss_fn = keras.losses.mean_squared_error
    optimizer = keras.optimizers.Adam(learning_rate=self.learning_rate)

    states, actions, rewards, next_states, dones = self.sample_experiences(batch_size)

    next_Q_values = self.dqn.predict(next_states)
    action_selected = np.argmax(next_Q_values, axis=1)
    max_next_Q_values = [pred[idx] for (pred, idx) in zip(self.target_dqn.predict(next_states), action_selected)]

    target_Q_values = np.array([reward + self.discount * next_Q if not done else reward for (reward, done, next_Q) in zip(rewards, dones, max_next_Q_values)])
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, 5)
    with tf.GradientTape() as tape:
        all_Q_values = self.dqn(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
        print(loss)
    grads = tape.gradient(loss, self.dqn.trainable_variables)
    optimizer.apply_gradients(zip(grads, self.dqn.trainable_variables))
    self.tmp_loss_array.append(loss)
```

Figure 10: Double DQN

The results on the smallClassic layout were very good (figure 62).

I've also tried the agent on new layout: mediumCLassic and results were impressive. We could see that agent was performing really well on the layout that he saw first time.

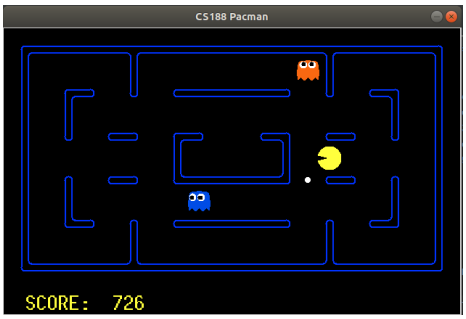

Figure 11: Game on the mediumClassic layout by agent trained on smallClassic one

```
Average Score: 846.4
Scores:        1234.0, 40.0, 1355.0, 1239.0, 1435.0, -650.0, 1148.0, 1282.0, 1011.0, 370.0
Win Rate:      7/10 (0.70)
Record:        Win, Loss, Win, Win, Win, Loss, Win, Win, Win, Loss
```

Figure 12: Results for the mediumClassic layout

3.10 10th experiment: Prioritized Experience Replay

For the next experiment I've implemented the Prioritized Experience Replay.

```
self.offset = 0.1
# starts from low and increase during training,
# opposit to epsilon, so I will use epsilon to define
self.beta = (1.0 - self.epsilon) + 0.1
self.per_alpha = 0.9
```

Figure 13: Initialization of new parameters

```
if len(self.memory) == 0:
    initial_priority = 1
else:
    initial_priority = max(self.memory, key=lambda k:k[-1])[-1]

self.memory.append(([my_state, my_action_idx, reward, my_nextState, int(done), initial_priority]))
```

Figure 14: Priority initialization

```
def sample_experiences(self):

    priority_sum = sum((self.memory[k][-1])**self.per_alpha for k in range(len(self.memory)))
    probabilities = [(self.memory[k][-1])**self.per_alpha/priority_sum for k in range(len(self.memory))]

    indices = np.random.choice(len(self.memory), p=probabilities, size=self.batch_size)
    used_prob = [probabilities[index] for index in indices]

    importances = [(self.batch_size * p) ** (-self.beta) for p in used_prob]
    max_imp = max(importances)
    importances = [imp/max_imp for imp in importances]

    batch = [self.memory[index] for index in indices]
    states, actions, rewards, next_states, dones, priorities = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(6)]
    return (states, actions, rewards, next_states, dones), indices, importances
```

Figure 15: Sampling experience according to priority

```
def update_transition_priority(self, target_Q, Q, indices):

    for tq, q, idx in zip(target_Q, Q, indices):
        self.memory[idx][-1] = (abs(tq - q) + self.offset).numpy()[0]

def training_step(self):

    loss_fn = keras.losses.mean_squared_error
    optimizer = keras.optimizers.Adam(learning_rate=self.learning_rate)

    (states, actions, rewards, next_states, dones), indices, importances = self.sample_experiences()

    next_Q_values = self.dqn.predict(next_states)
    action_selected = np.argmax(next_Q_values, axis=1)
    max_next_Q_values = [pred[idx] for (pred, idx) in zip(self.target_dqn.predict(next_states), action_selected)]

    target_Q_values = rewards + (1 - dones) * self.discount * max_next_Q_values
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, 5)
    with tf.GradientTape() as tape:
        all_Q_values = self.dqn(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        self.update_transition_priority(target_Q_values, Q_values, indices)
        weighted_error = loss_fn(target_Q_values, Q_values)*importances
        loss = tf.reduce_mean(weighted_error)
        print(loss)
    grads = tape.gradient(loss, self.dqn.trainable_variables)
    optimizer.apply_gradients(zip(grads, self.dqn.trainable_variables))
    self.tmp_loss_array.append(loss)
```

Figure 16: Importance sampling and priority update

The performance of agent was worse than in the previous approach. I've noticed that agent was usually dying when two ghost trapped him. Like in the corridor when one ghost was at the beginning and another at the end. Then it was not possible for pacman to escape.

On the loss plot we can see that initial loss was much bigger than for training without the Prioritized Experience Replay. It seem correct as we were not choosing the samples randomly but with the priority for the ones with the biggest error.

3.11 11th experiment: Prioritized Experience Replay, trained on several layouts

I have repeated the learning when different layouts were choosing randomly during training. We can see on the reward function that it doesn't reach high score.

I was thinking that maybe because of the POMDP approach the problem can be with the situations when pacman sees the closest ghost but actually two ghosts are close and they are surrounding him. As the feature extraction allows to have the information only about one of them he cannot differentiate safe situations with unsafe.

I was also wondering why the problem was not present when training without Prioritized Experience Replay I think that it could be because in the current approach we probable are repeating the training again and again for the same situations (similar feature vector, as we cannot see if the second ghost is also close or not, but huge difference for the reward, as the pacman sometime dies). I think that maybe when updating weights again and again for the same kind of problem (that is impossible to solve with that POMDP) we affect the general performance of the agent.

For this experiment I didn't upload the table with results as the agent's performance was really weak.

3.12 12th experiment: Prioritized Experience Replay, trained on several layouts with information about second ghost

I wanted to check a little bit my theory so I have repeated the previous training but when the information about the distance to two ghosts was extracted. Also only two layouts were chosen for training (smallClassic and mediumClassic as the rest of them has one or three ghosts).

The performance improved. By checking the figure 51 I can expect that with longer training I could obtain better performance (however I was not able to train the agent longer in that case).

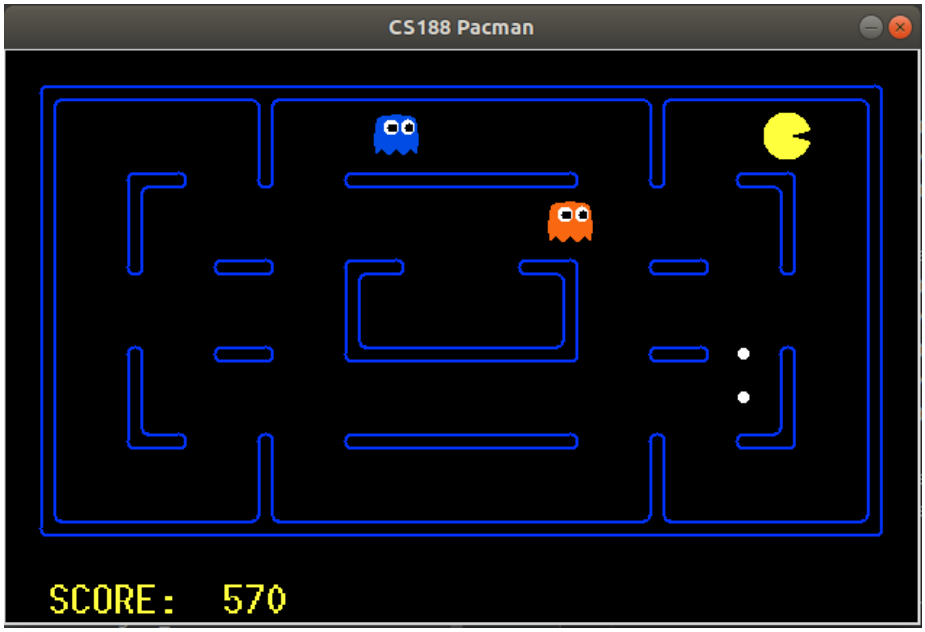

Figure 17: Pacman's game

When checking the agent's behaviour I've noticed that this time the agent was using the capsules (and eating ghosts after that) more often than anytime before (so the Prioritized Experience Replay was successfully used in that case as probably by selecting samples randomly this situation could be quite rare). However the pacman was not eating the food quick. I think that for MDP approach the Prioritized Experience Replay can be very helpful however for the POMDP it depends how well the agent's perception in defined (as for the previous experiment the performance was decreased because of the lack

of full information for the agent about the difference between good and risky behaviour).

4 Conclusion

I think that proper definition of feature vector has a big influence on the performance. For sure it would be easier to perceive full image (however I wanted to decrease amount of information to train).

The best performance I've receive when using Experience Replay buffer, target dqn network and double DQN.

For the Prioritized Experience Replay the performance decreased but in this case I think the POMDP approach could be the reason why. However I could also noticed that for the rare situations (like eating ghost after eating capsule) the Prioritized Experience Replay was helpful.

Appendices

.1 Code used for first experiment

.1.1 Feature extraction

```
def closestFood(pos, food, walls):
    fringe = [(pos[0], pos[1], 0)]
    expanded = set()
    while fringe:
        pos_x, pos_y, dist = fringe.pop(0)
        if (pos_x, pos_y) in expanded:
            continue
        expanded.add((pos_x, pos_y))
        # if we find a food at this location then exit
        if food[pos_x][pos_y]:
            return dist
        # otherwise spread out from the location to its neighbours
        nbrs = Actions.getLegalNeighbors((pos_x, pos_y), walls)
        for nbr_x, nbr_y in nbrs:
            fringe.append((nbr_x, nbr_y, dist+1))
    # no food found
    return None
```

Figure 18: Original function: closestFood

```python
def my_closestGhost(pos, ghosts, walls):
    fringe = [(pos[0], pos[1], 0)]
    expanded = set()
    while fringe:
        pos_x, pos_y, dist = fringe.pop(0)
        if (pos_x, pos_y) in expanded:
            continue
        expanded.add((pos_x, pos_y))
        # if we find a food at this location then exit
        if (pos_x, pos_y) in ghosts:
            return dist, pos_x, pos_y, ghosts.index((pos_x, pos_y)) + 1
        # otherwise spread out from the location to its neighbours
        nbrs = Actions.getLegalNeighbors((pos_x, pos_y), walls)
        for nbr_x, nbr_y in nbrs:
            fringe.append((nbr_x, nbr_y, dist + 1))
    # no food found
    return None, None, None, None

def my_closestFood(pos, food, walls):
    fringe = [(pos[0], pos[1], 0)]
    expanded = set()
    while fringe:
        pos_x, pos_y, dist = fringe.pop(0)
        if (pos_x, pos_y) in expanded:
            continue
        expanded.add((pos_x, pos_y))
        # if we find a food at this location then exit
        if food[pos_x][pos_y]:
            return dist, pos_x, pos_y
        # otherwise spread out from the location to its neighbours
        nbrs = Actions.getLegalNeighbors((pos_x, pos_y), walls)
        for nbr_x, nbr_y in nbrs:
            fringe.append((nbr_x, nbr_y, dist + 1))
    # no food found
    return None, None, None

def my_closestCapsule(pos, capsules, walls):
    fringe = [(pos[0], pos[1], 0)]
    expanded = set()
    while fringe:
        pos_x, pos_y, dist = fringe.pop(0)
        if (pos_x, pos_y) in expanded:
            continue
        expanded.add((pos_x, pos_y))
        # if we find a food at this location then exit
        if (pos_x, pos_y) in capsules:
            return dist, pos_x, pos_y
        # otherwise spread out from the location to its neighbours
        nbrs = Actions.getLegalNeighbors((pos_x, pos_y), walls)
        for nbr_x, nbr_y in nbrs:
            fringe.append((nbr_x, nbr_y, dist + 1))
    # no food found
    return None, None, None
```

Figure 19: Added functions: my_closestGhost, my_closestFood and my_closestCapsule

```python
def getFeatures(self, state, action):
    # extract the grid of food and wall locations and get the ghost locations
    food = state.getFood()
    walls = state.getWalls()
    ghosts = state.getGhostPositions()
    capsules = state.data.capsules
    features = util.Counter()
    # compute the location of pacman after he takes the action
    x, y = state.getPacmanPosition()
    dx, dy = Actions.directionToVector(action)
    next_x, next_y = int(x + dx), int(y + dy)
    # count the number of ghosts 1-step away
    features["#-of-ghosts-1-step-away"] = sum((next_x, next_y) in Actions.getLegalNeighbors(g, walls) for g in ghosts)
    # if there is no danger of ghosts then add the food feature
    if not features["#-of-ghosts-1-step-away"] and food[next_x][next_y]:
        features["eats-food"] = 1.0
    else:
        features["eats-food"] = 0.0
    distCapsule, capsule_x, capsule_y = my_closestCapsule((next_x, next_y), capsules, walls)
    if distCapsule is not None:
        # make the distance a number less than one otherwise the update
        # will diverge wildly
        capsuleDist = manhattanDistance((next_x, next_y), (capsule_x, capsule_y))
        capsuleDist = float(capsuleDist/10) if capsuleDist < 10 else 1.0
        features["closest-capsule"] = capsuleDist
    else:
        features["closest-capsule"] = -1.0
    distGhost, ghost_x, ghost_y, ghost_id = my_closestGhost((next_x, next_y), ghosts, walls)
    if distGhost is not None:
        ghostDist = manhattanDistance((next_x, next_y), (ghost_x, ghost_y))
        ghostDist = float(ghostDist / 10) if ghostDist < 10 else 1.0
        features["closest-ghost"] = ghostDist
        s = state.getGhostState(ghost_id)
        features[f"the_closest_ghost_scaredTime"] = s.scaredTimer
    else:
        features["closest-ghost"] = -1.0
        features[f"the_closest_ghost_scaredTime"] = 0
    dist, food_x, food_y = my_closestFood((next_x, next_y), food, walls)
    _, foodNow_x, foodNow_y = my_closestFood((x, y), food, walls)
    features["eat-last-food"] = 0
    features["getting-closer-to-food"] = 0
    if dist is not None:
        foodDistNow = manhattanDistance((x, y), (foodNow_x, foodNow_y))
        foodDist = manhattanDistance((next_x, next_y), (food_x, food_y))
        features["getting-closer-to-food"] = 1 if foodDist < foodDistNow else 0
        foodDist = float(foodDist / 10) if foodDist < 10 else 1.0
        features["closest-food"] = foodDist
        if ((next_x, next_y) == (food_x, food_y)) and (sum(sum(x) for x in food)) == 1:
            features["eat-last-food"] = 1.0
    else:
        features["closest-food"] = -1.0

    return features
```

Figure 20: Feature extraction, first idea

.1.2 Q-learning Agent

```
class QLearningAgent(Agent):
    """
      Q-Learning Agent

      Functions you should fill in:
        - computeValueFromQValues
        - computeActionFromQValues
        - getQValue
        - getAction
        - update

      Instance variables you have access to
        - self.epsilon (exploration prob)
        - self.alpha (learning rate)
        - self.discount (discount rate)

      Functions you should use
        - self.getLegalActions(state)
          which returns legal actions for a state
    """
    def __init__(self, actionFn = None, numTraining=100, epsilon=0.4, alpha=0.5, gamma=0.99):

        self.featExtractor = SimpleExtractor()
        self.seen_states = []
        self.dqn = keras.models.Sequential([
            keras.layers.Dense(64, activation="elu", input_shape=[40]),
            keras.layers.LayerNormalization(),
            keras.layers.Dense(32, activation="linear"),
            keras.layers.LayerNormalization(),
            keras.layers.Dense(5)
        ])
        # uncomment for testing
        # self.dqn = keras.models.load_model("weights")
        if actionFn == None:
            actionFn = lambda state: state.getLegalActions()
        self.actionFn = actionFn
        self.episodesSoFar = 0
        self.accumTrainRewards = 0.0
        self.accumTestRewards = 0.0
        self.numTraining = int(numTraining)
        self.epsilon = float(epsilon)
        self.alpha = float(alpha)
        self.discount = float(gamma)
        self.memory = deque(maxlen=1000)
        self.batch_size = 128
        self.learning_rate = 0.0001
        self.loss_array = []
        self.tmp_loss_array = []
        self.reward_array = []
        self.epizode_array = []
```

Figure 21: Q-Learning Agent

```
def getLegalActions(self,state):
    """
      Get the actions available for a given
      state. This is what you should use to
      obtain legal actions for a state
    """
    return self.actionFn(state)

def observeTransition(self, state,action,nextState,deltaReward):
    """
        Called by environment to inform agent that a transition has
        been observed. This will result in a call to self.update
        on the same arguments

        NOTE: Do *not* override or call this function
    """
    self.episodeRewards += deltaReward
    self.update(state,action,nextState,deltaReward)

def startEpisode(self):
    """
      Called by environment when new episode is starting
    """
    self.lastState = None
    self.lastAction = None
    self.episodeRewards = 0.0

def stopEpisode(self):
    """
      Called by environment when episode is done
    """
    if self.episodesSoFar < self.numTraining:
        self.accumTrainRewards += self.episodeRewards
    else:
        self.accumTestRewards += self.episodeRewards
    self.episodesSoFar += 1
    if self.episodesSoFar >= self.numTraining:
        # Take off the training wheels
        self.epsilon = 0.0    # no exploration
        self.alpha = 0.0      # no learning

def isInTraining(self):
    return self.episodesSoFar < self.numTraining

def isInTesting(self):
    return not self.isInTraining()

def setEpsilon(self, epsilon):
    self.epsilon = epsilon

def setLearningRate(self, alpha):
    self.alpha = alpha
```

Figure 22: Q-Learning Agent

```
def setDiscount(self, discount):
    self.discount = discount

def doAction(self,state,action):
    """
        Called by inherited class when
        an action is taken in a state
    """
    self.lastState = state
    self.lastAction = action
    actionArr = [0, 0, 0, 0, 0]
    actionArr[arrWithActions.index(action)] = 1
    self.last_action_memory.append(actionArr)

def observationFunction(self, state):
    """
        This is where we ended up after our last action.
        The simulation should somehow ensure this is called
    """
    if not self.lastState is None:
        reward = state.getScore() - self.lastState.getScore()
        self.observeTransition(self.lastState, self.lastAction, state, reward)
    return state

def registerInitialState(self, state):
    self.startEpisode()
    if self.episodesSoFar == 0:
        print('Beginning %d episodes of Training' % (self.numTraining))

def training_step(self, batch_size):

    loss_fn = keras.losses.mean_squared_error
    optimizer = keras.optimizers.Adam(learning_rate=self.learning_rate)

    states, actions, rewards, next_states = self.sample_experiences(batch_size)

    next_Q_values = self.dqn.predict(next_states)
    max_next_Q_values = np.max(next_Q_values, axis=1)

    target_Q_values = (rewards + self.discount * np.array(max_next_Q_values))
    target_Q_values = target_Q_values.reshape(-1, 1)
    mask = tf.one_hot(actions, 5)
    with tf.GradientTape() as tape:
        all_Q_values = self.dqn(states)
        Q_values = tf.reduce_sum(all_Q_values * mask, axis=1, keepdims=True)
        loss = tf.reduce_mean(loss_fn(target_Q_values, Q_values))
        print(loss)
    grads = tape.gradient(loss, self.dqn.trainable_variables)
    optimizer.apply_gradients(zip(grads, self.dqn.trainable_variables))
    self.tmp_loss_array.append(loss)
```

Figure 23: Q-Learning Agent

```
def final(self, state):
    """
      Called by Pacman game at the terminal state
    """
    deltaReward = state.getScore() - self.lastState.getScore()
    self.observeTransition(self.lastState, self.lastAction, state, deltaReward)
    self.stopEpisode()

    # Make sure we have this var
    if not 'episodeStartTime' in self.__dict__:
        self.episodeStartTime = time.time()
    if not 'lastWindowAccumRewards' in self.__dict__:
        self.lastWindowAccumRewards = 0.0
    self.lastWindowAccumRewards += state.getScore()

    if len(self.memory) >= self.batch_size:
        self.training_step(self.batch_size)

    if self.maxepsilon:
        self.epsilon = max(self.maxepsilon - (self.episodesSoFar / self.numTraining), self.minepsilon)
        print(f"Epsilon: {self.epsilon}")

    NUM_EPS_UPDATE = 100
    if self.episodesSoFar % NUM_EPS_UPDATE == 0:

        self.dqn.save("weights")
        print("Weights updated")

        print('Reinforcement Learning Status:')
        windowAvg = self.lastWindowAccumRewards / float(NUM_EPS_UPDATE)
        if self.episodesSoFar <= self.numTraining:
            trainAvg = self.accumTrainRewards / float(self.episodesSoFar)
            print('\tCompleted %d out of %d training episodes' % (
                   self.episodesSoFar,self.numTraining))
            print('\tAverage Rewards over all training: %.2f' % (
                    trainAvg))
        else:
            testAvg = float(self.accumTestRewards) / (self.episodesSoFar - self.numTraining)
            print('\tCompleted %d test episodes' % (self.episodesSoFar - self.numTraining))
            print('\tAverage Rewards over testing: %.2f' % testAvg)
        print('\tAverage Rewards for last %d episodes: %.2f'  % (
                NUM_EPS_UPDATE,windowAvg))
        print('\tEpisode took %.2f seconds' % (time.time() - self.episodeStartTime))

        self.lastWindowAccumRewards = 0.0
        self.episodeStartTime = time.time()

    if self.episodesSoFar == self.numTraining:
        self.alpha = 0.0
        self.epsilon = 0.0
        msg = 'Training Done (turning off epsilon and alpha)'
        print('%s\n%s' % (msg,'-' * len(msg)))
```

Figure 24: Q-Learning Agent

```
def sample_experiences(self, batch_size):

    indices = np.random.randint(len(self.memory), size=batch_size)
    batch = [self.memory[index] for index in indices]
    states, actions, rewards, next_states = [
        np.array([experience[field_index] for experience in batch])
        for field_index in range(4)]
    return states, actions, rewards, next_states

def __state_to_mystate(self, state):

    my_state_vector = []

    for a in arrWithActions:
        feats = self.featExtractor.getFeatures(state, a)
        for f in feats.keys():
            if type(feats[f]) == list:
                for x in feats[f]:
                    my_state_vector.append(x)
            else:
                my_state_vector.append(feats[f])

    my_state_vector.extend([item for sublist in self.last_action_memory for item in sublist])
    return np.array(my_state_vector)

def getQValue(self, state, action):
    """
      Returns Q(state,action)
      Should return 0.0 if we have never seen a state
      or the Q node value otherwise
    """
    my_state = self.__state_to_mystate(state)

    idx_of_state = arrWithActions.index(action)

    if any(all(list == my_state) for list in self.seen_states):
        Q_values = self.dqn.predict(np.array([my_state, ]))[0]
        return Q_values[idx_of_state]
    else:
        return 0.0
```

Figure 25: Q-Learning Agent

```
def computeValueFromQValues(self, state):
    """
      Returns max_action Q(state,action)
      where the max is over legal actions.  Note that if
      there are no legal actions, which is the case at the
      terminal state, you should return a value of 0.0.
    """
    legal_actions = self.getLegalActions(state)

    if legal_actions:
        legal_actions = [0 if act not in legal_actions else 1 for act in arrWithActions]
        Q_values = self.dqn.predict(np.array([self.__state_to_mystate(state), ]))[0]

        legal_values = []

        for (value, legal) in zip(Q_values, legal_actions):
            if legal:
                legal_values.append(value)

        return max(legal_values)
    else:
        return 0.0

def computeActionFromQValues(self, state):
    """
      Compute the best action to take in a state.  Note that if there
      are no legal actions, which is the case at the terminal state,
      you should return None.
    """
    to_chose_from = {}

    legal_actions = self.getLegalActions(state)

    for action in legal_actions:
        to_chose_from[action] = self.getQValue(state, action)

    if to_chose_from:
        return max(to_chose_from, key=to_chose_from.get)
    else:
        return None
```

Figure 26: Q-Learning Agent

```
def getAction(self, state):
    """
      Compute the action to take in the current state.  With
      probability self.epsilon, we should take a random action and
      take the best policy action otherwise.  Note that if there are
      no legal actions, which is the case at the terminal state, you
      should choose None as the action.
    """
    # Pick Action
    legalActions = self.getLegalActions(state)
    action = None

    legalActionsB = [0 if act not in legalActions else 1 for act in arrWithActions]
    rnd = random.uniform(0, 1)
    my_state = self.__state_to_mystate(state)

    if rnd <= self.epsilon:
        action = random.choice(legalActions)
    else:
        Q_values = self.dqn.predict(np.array([my_state, ]))[0]
        legal_values = []

        for (value, legal) in zip(Q_values, legalActionsB):
            if legal:
                legal_values.append(value)

        max_legalValue = max(legal_values)
        idx_action = list(Q_values).index(max_legalValue)
        action = arrWithActions[idx_action]
    self.doAction(state, action)

    return action

def update(self, state, action, nextState, reward):
    """
      The parent class calls this to observe a
      state = action => nextState and reward transition.
      You should do your Q-Value update here

      NOTE: You should never call this function,
      it will be called on your behalf
    """
    my_state = self.__state_to_mystate(state)
    my_nextState = self.__state_to_mystate(nextState)
    self.seen_states.append(my_state)

    my_action_idx = ['East', 'West', 'North', 'South', 'Stop'].index(action)
    self.memory.append([my_state, my_action_idx, reward, my_nextState])

def getPolicy(self, state):
    return self.computeActionFromQValues(state)

def getValue(self, state):
    return self.computeValueFromQValues(state)
```

Figure 27: Q-Learning Agent

.2 Loss

Table 1: Loss plot during training for experiments 1-6

Figure 28: For experiment 1

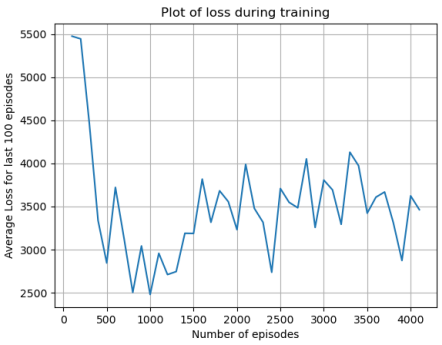

Figure 29: For experiment 2

Figure 30: For experiment 3

Figure 31: For experiment 4

Figure 32: For experiment 5

Figure 33: For experiment 6

Table 2: Loss plot during training for experiments 7-12

Figure 34: For experiment 7

Figure 35: For experiment 8

Figure 36: For experiment 9

Figure 37: For experiment 10

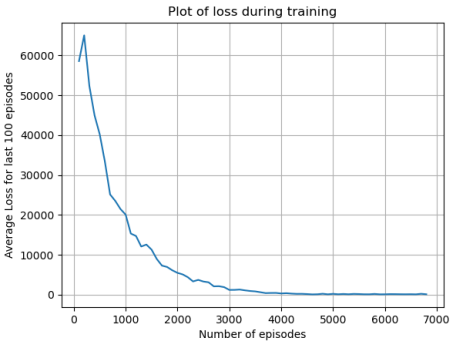

Figure 38: For experiment 11

Figure 39: For experiment 12

.3 Reward

Table 3: Reward plot during training for experiments 1-6

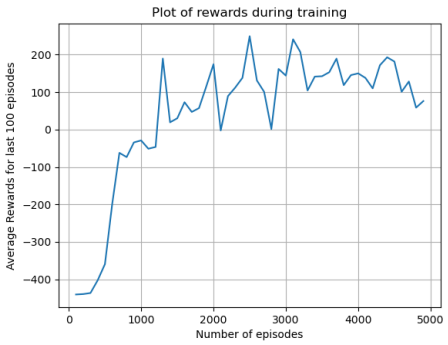

Figure 40: For experiment 1

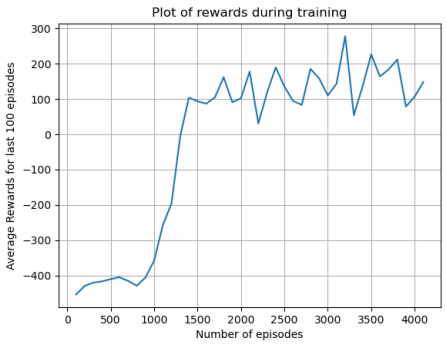

Figure 41: For experiment 2

Figure 42: For experiment 3

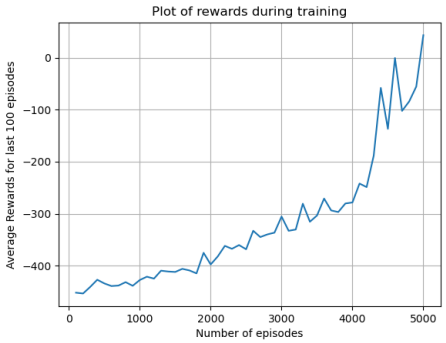

Figure 43: For experiment 4

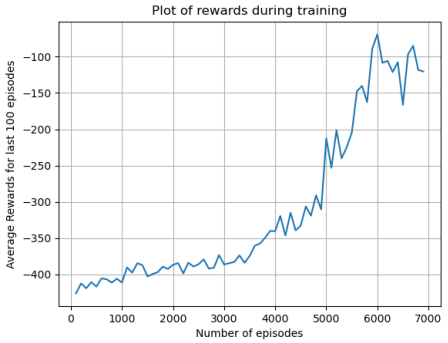

Figure 44: For experiment 5

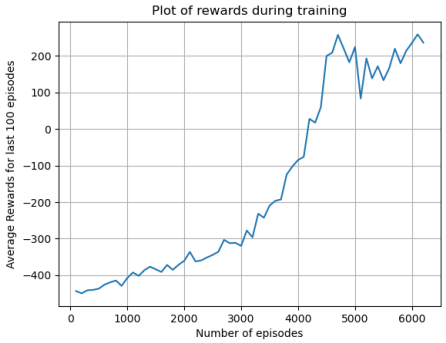

Figure 45: For experiment 6

Table 4: Reward plot during training for experiments 7-12

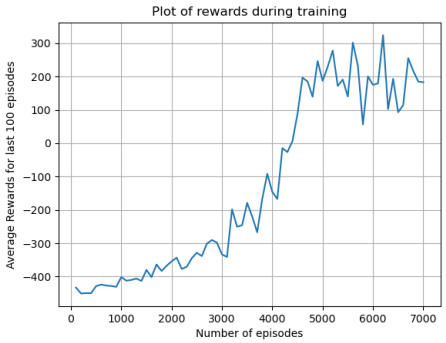

Figure 46: For experiment 7

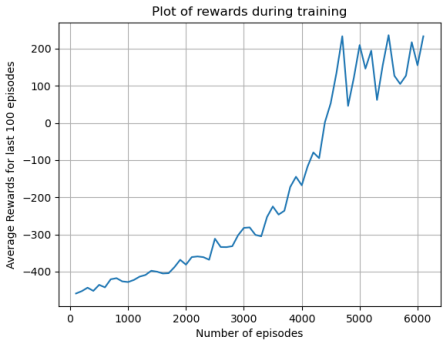

Figure 47: For experiment 8

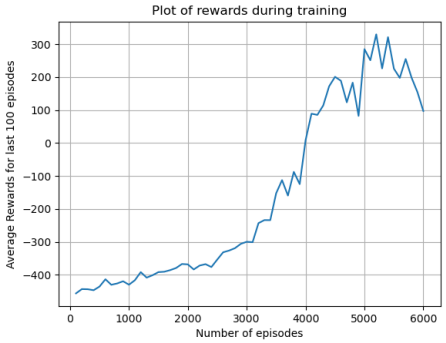

Figure 48: For experiment 9

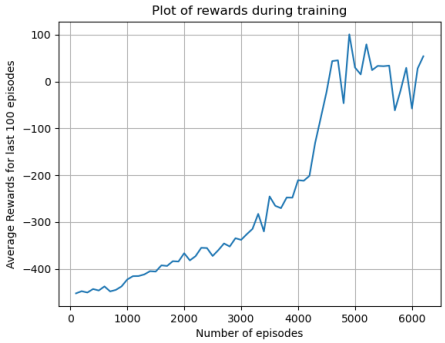

Figure 49: For experiment 10

Figure 50: For experiment 11

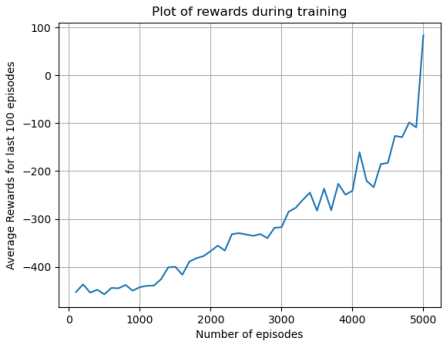

Figure 51: For experiment 12

.4 Table of performance for the same layout (smallClassic)

Figure 52: Performance on the smallClassic layout (the one used for training) for experiments 1-6

```
Average Score: 473.5
Scores:        -159.0, -389.0, -106.0, 943.0, 983.0, 856.0, -275.0, 954.0, 965.0, 963.0
Win Rate:      6/10 (0.60)
Record:        Loss, Loss, Loss, Win, Win, Win, Loss, Win, Win, Win
```

Figure 53: For experiment 1

```
Average Score: 448.7
Scores:        805.0, 925.0, -242.0, 966.0, 966.0, 949.0, -106.0, -545.0, 973.0, -204.0
Win Rate:      6/10 (0.60)
Record:        Win, Win, Loss, Win, Win, Win, Loss, Loss, Win, Loss
```

Figure 54: For experiment 2

```
Average Score: 671.0
Scores:        1035.0, 988.0, 844.0, 809.0, -338.0, 806.0, -93.0, 872.0, 962.0, 825.0
Win Rate:      8/10 (0.80)
Record:        Win, Win, Win, Win, Loss, Win, Loss, Win, Win, Win
```

Figure 55: For experiment 3

```
Average Score: 352.0
Scores:        953.0, 855.0, -319.0, 903.0, -264.0, 970.0, -160.0, -58.0, -321.0, 961.0
Win Rate:      5/10 (0.50)
Record:        Win, Win, Loss, Win, Loss, Win, Loss, Loss, Loss, Win
```

Figure 56: For experiment 4

```
Average Score: 348.1
Scores:        -405.0, 797.0, 25.0, 931.0, -85.0, 895.0, 925.0, 938.0, -325.0, -215.0
Win Rate:      5/10 (0.50)
Record:        Loss, Win, Loss, Win, Loss, Win, Win, Win, Loss, Loss
```

Figure 57: For experiment 5

```
Average Score: 938.1
Scores:        957.0, 1112.0, 972.0, 967.0, 874.0, 808.0, 954.0, 943.0, 822.0, 972.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Figure 58: For experiment 6

Figure 59: Performance on the smallClassic layout (the one used for training) for experiments 7-12

```
Average Score: 792.3
Scores:        829.0, 961.0, 959.0, 919.0, 858.0, 958.0, -150.0, 962.0, 809.0, 818.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Win, Win, Win, Loss, Win, Win, Win
```

Figure 60: For experiment 7

```
Average Score: 811.8
Scores:        939.0, 921.0, 958.0, 987.0, 912.0, 877.0, 898.0, -339.0, 982.0, 983.0
Win Rate:      9/10 (0.90)
Record:        Win, Win, Win, Win, Win, Win, Win, Loss, Win, Win
```

Figure 61: For experiment 8

```
Average Score: 961.1
Scores:        982.0, 921.0, 970.0, 935.0, 981.0, 951.0, 1161.0, 768.0, 967.0, 975.0
Win Rate:      10/10 (1.00)
Record:        Win, Win, Win, Win, Win, Win, Win, Win, Win, Win
```

Figure 62: For experiment 9

```
Average Score: 621.9
Scores:        -280.0, 938.0, 1355.0, -390.0, 913.0, 1026.0, 933.0, 800.0, 964.0, -40.0
Win Rate:      7/10 (0.70)
Record:        Loss, Win, Win, Loss, Win, Win, Win, Win, Win, Loss
```

Figure 63: For experiment 10

```
Average Score: 463.2
Scores:        997.0, -67.0, 1035.0, 980.0, 1362.0, -313.0, -7.0, 343.0, 0.0, 302.0
Win Rate:      4/10 (0.40)
Record:        Win, Loss, Win, Win, Win, Loss, Loss, Loss, Loss, Loss
```

Figure 64: For experiment 12 (tested on smallClassic and mediumClassic layouts (randomly))