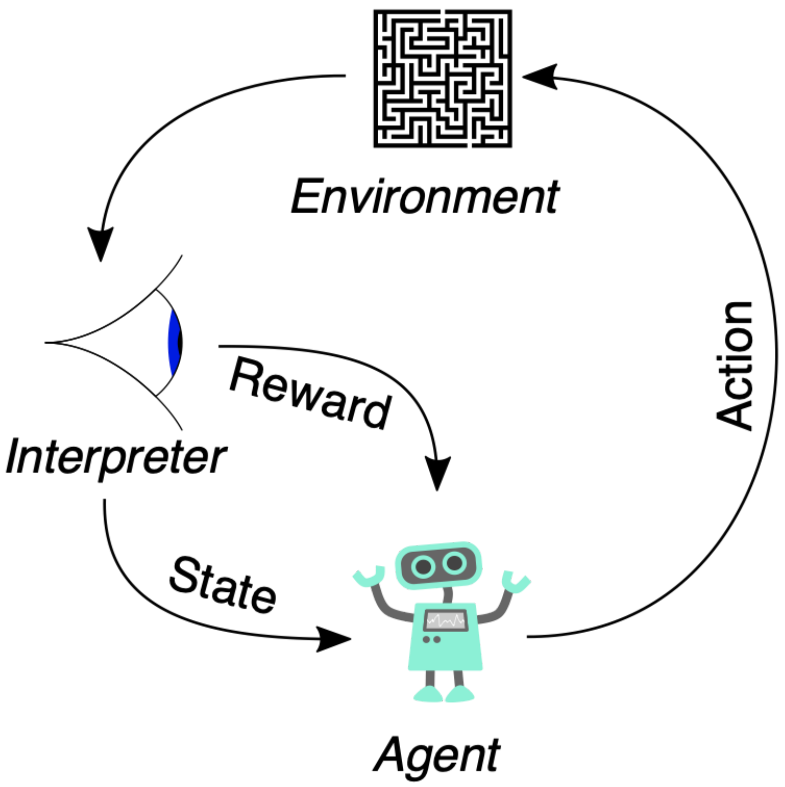
Reinforcement Learning: Pac-Man

Intro To Reinforcement Learning

Reinforcement learning concerns an area of machine learning where a software agent is trained to take actions leading to the maximum possible reward in an environment. Consider this: You're playing chess for the very first time, and nobody bothers to explain the rules to you. The only thing you're told is what the valid moves are in chess. Thus, you begin your journey towards becoming the next grandmaster. In the beginning you basically make random moves, because you know nothing [Jon Snow]. However, for each move you make, you observe the result of your move, and the move made by your opponent as a response. Essentially, you receive a positive or negative reward based on your moves. Gradually, you begin to understand what's considered good moves and you begin to play better.

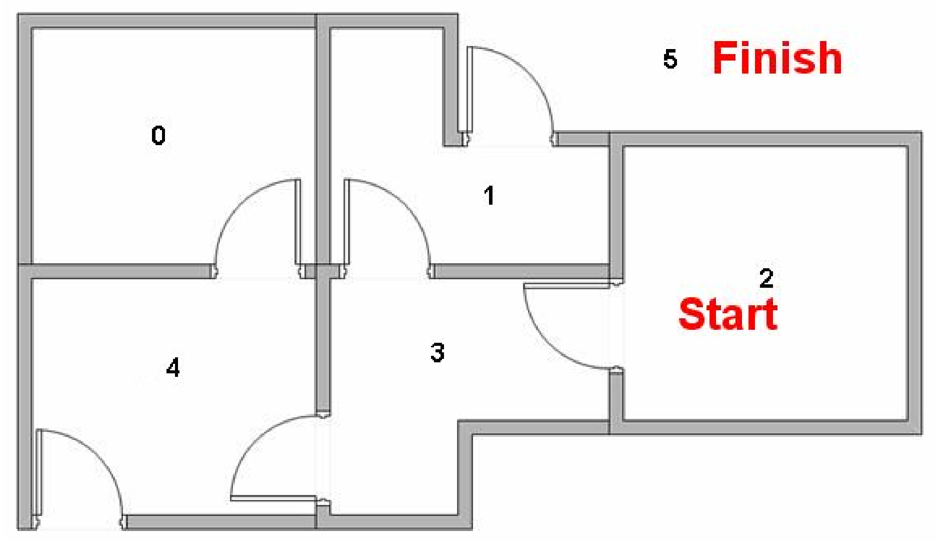
In the context of agents and reinforcement learning, every move would have a corresponding reward associated with. By taking out one of the opponents pieces, the agent would receive a positive reward. Similarly, it would receive a negative reward by losing a piece. Taking actions which ultimately leads to the checkmate of the opponent's king, would be considered maximizing the attainable reward.



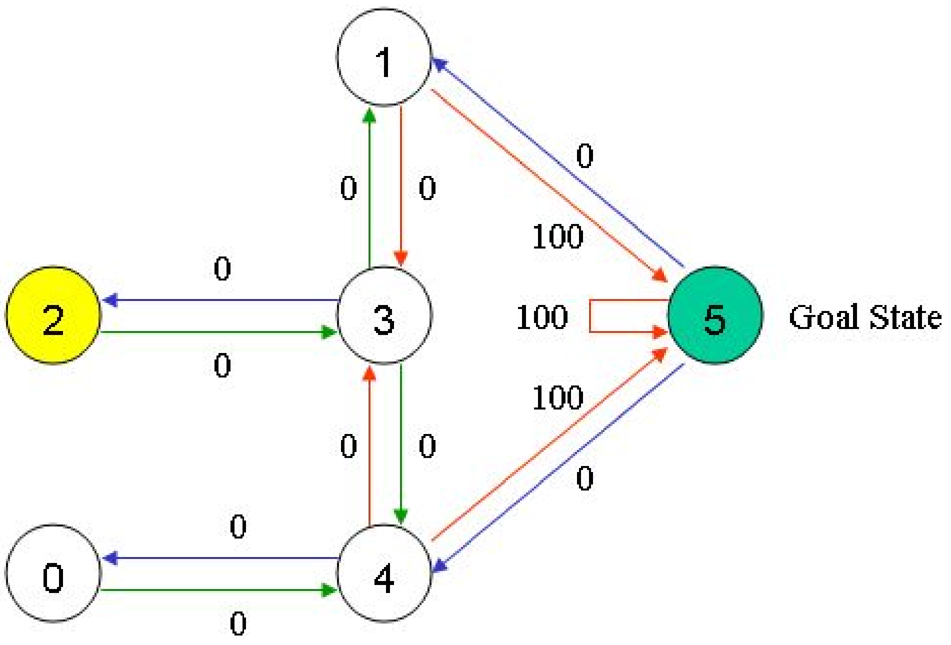
The image above illustrates the reinforcement learning process in general. An agent executes one of the possible actions for a given environment. Based on how the environment and the agent are affected by the action, the agent receives an appropriate reward and the next state of the environment, as a result of the action. By following one of many reinforcement learning strategies/algorithms, the agent can utilize the rewards received over many trials and form a policy for which action to take in each state it is in. Looking back at the chess example, after playing for a little while, you would learn that taking out an opponent piece, when it is unprotected, is usually a good move (unless you're playing Carlsen. Then he probably wants you to do it, and in reality you're screwed).

Q-Learning

Q-learning is a reinforcement learning algorithm, which learns Q-values for state-action pairs in the environment. A Q-value is a measure of utility, or quality, of an action in a given state. Thus, for each state, the agent will pick the action with the highest Q-value. However, these Q-values need to be learned.



Consider a maze consisting of different rooms, representing different states. Say that our goal is to reach state 5 from state 2. For each room, there are a set of valid actions, namely moving to adjacent rooms/states. Say that by entering state 5, the agent receives +100 reward, while all other states return 0 reward. In a state diagram, it would look like this:



Additionally, we have a Q-value matrix, which specifies the Q-value for each action in each state. These values are initialized to zero.

$$
Q = \begin{array}{c} \\ 0 \\ 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{array}
\begin{array}{c}
\begin{array}{cccccc} 0 & 1 & 2 & 3 & 4 & 5 \end{array} \\
\begin{bmatrix}
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0
\end{bmatrix}
\end{array}
$$

In each state, the agent will, in general, pick the action with highest Q-value. However, there is a certain probability that the agent will pick a random action. This is following the concept of exploitation vs exploration. By picking the action of the highest Q-value, the agent exploits the knowledge it has collected up until this point. By picking a random action, the agent gets the possibility to explore unknown territories, and possibly find a better path to the goal. When all possible actions have equal Q-values, as it is initially, a random action will be made.

After each action, the Q-value for the chosen action is updated for that state, where the action was performed. This is done with this simple, but complicated-looking formula:

$$Q(s_t,a_t) \leftarrow \underbrace{Q(s_t,a_t)}_{\text{old value}} + \underbrace{\alpha}_{\text{learning rate}} \cdot \left(\overbrace{\underbrace{r_t}_{\text{reward}} + \underbrace{\gamma}_{\text{discount factor}} \cdot \underbrace{\max_a Q(s_{t+1},a)}_{\text{estimate of optimal future value}}}^{\text{learned value}} - \underbrace{Q(s_t,a_t)}_{\text{old value}} \right)$$

maxQ(s+1, *a*) = Maximum Q-value in the state resulting from choosing action *a* in current state
α = a scaling factor to prevent changing the Q-value too much. Range: [0, 1]
γ = a discount factor to put greater weight on immediate reward received from executing action in current state. Range: [0, 1]

Consider if the agent was already in state 1, given the current zero-matrix of Q-values. The possible actions are: go to state 3, go to state 5. Say the agent picks state 5 by random.

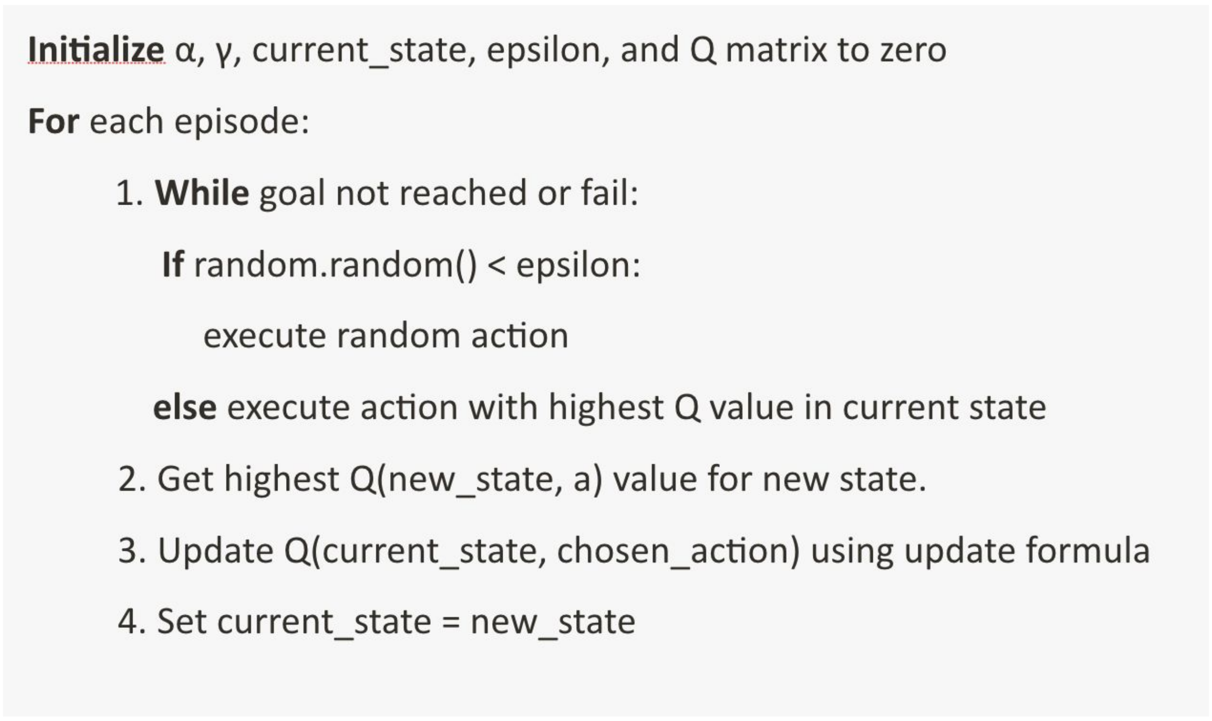
The Q-value calculation for "action state 5 in state 1", given α = 1.0 and γ = 0.8 would be:
Q(1, 5) += 1.0 * (R(1, 5) + 0.8 * Max[Q(5, 1), Q(5, 4), Q(5, 5)] - Q(1, 5)) = 1.0 * (100 + 0.8 * 0 - 0) = 100. This results in this Q-table:

$$Q= \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 2 & 0 & 0 & 0 & 0 & 0 & 100 \\ 3 & 0 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 5 & 0 & 0 & 0 & 0 & 0 & 0 \end{array}$$

Because we reached a goal state, we have finished one episode, and new episode commences, where the objective is the same: reach state 5. At some point, the Q-values will converge and the optimal path will be clear.

$$Q= \begin{array}{c|cccccc} & 0 & 1 & 2 & 3 & 4 & 5 \\ \hline 0 & 0 & 0 & 0 & 0 & 400 & 0 \\ 1 & 0 & 0 & 0 & 320 & 0 & 500 \\ 2 & 0 & 0 & 0 & 320 & 0 & 0 \\ 3 & 0 & 400 & 256 & 0 & 400 & 0 \\ 4 & 320 & 0 & 0 & 320 & 0 & 500 \\ 5 & 0 & 400 & 0 & 0 & 400 & 500 \end{array}$$

Q-learning algorithm



```
Initialize α, γ, current_state, epsilon, and Q matrix to zero
For each episode:
        1. While goal not reached or fail:
              If random.random() < epsilon:
                  execute random action
             else execute action with highest Q value in current state
        2. Get highest Q(new_state, a) value for new state.
        3. Update Q(current_state, chosen_action) using update formula
        4. Set current_state = new_state
```

Deep Q-Learning

Deep Q-Learning can be implemented using vastly different approaches and variations. For this workshop we will keep it simple and use one neural network which will work as a state - action mapper. Think of it as replacing the q table from Q-Learning with a neural network who takes a state as input and give out the optimal action in that state.

If you have never worked with a neural network before this task will be a little bit challenging and confusing. But there is a code “solution” and Malte and Manu are always glad to help, so try if you feel like it.

In addition to that, we use something called experience replay. Whenever you make a move, you save the current_state, action, reward and next_state. When you then train the neural network, instead of training it on the move just made (which is how regular q learning works), we take a random sample from the experience memory storage and train the neural network on those instead. That is because as the pacman manages to progress in the game it will forget what it learned earlier to get to the point it is now. This results in poor performance because pacman forgets old information. But by saving old and new info, pacman never forgets.

You will need to run the code on a GPU on bigger maps. Ask Manu or Malte!

Assignments

Assignment 0: Clone repo and install requirements

- clone the repository from https://github.com/knowit/ml-pacman
- the default branch is q_learn. If you're not on it, do git checkout q_learn
- start a virtual environment (recommended). Ask if you're unsure what that is!
- run *pip install -r requirements.txt*
- run *pacman/main.py* and play some pacman!

Assignment 1: Q-Learning

In this assignment we will implement Q-Learning and solve simple pacman maps. Larger maps will have a big runtime because the state space increases when the board size increases. If you're wondering about something, ASK MALTE OR MANU!!! :-)

The recipe to solve it by q_learning is described here. If you want to try to code it by yourself you can try. If you get stuck, code (for each of the steps in this recipe) that solves pacman is available further down this document. Look in the code for helpful tips.

1. Make your own map in boards/level-0.txt. You can see level-2.txt for inspiration. However, there are two rules when creating the map: For now, make the map size a maximum of 6x6 and cover the entire outer ring with walls (these two rules are already present). Fill in the rest of the board with one pacman, one ghost, some coins and more walls.
   Symbols to fill in:
   Pacman = P
   Ghost = G
   Coin = .
   Wall = %

2. NOTE: The rest of the code is to be implemented in qlearning/q_learning.py, qlearning/q_utils.py and qlearning/q_config.py. Look in the code for helpful tips. If you get stuck, there is code further down that solves the given point you're stuck at. If you're really lazy you can just git checkout pacman_runnable...

3. Remove the *pass* keyword in the constructor of QLearn and initialize q_table as a dictionary and q_config.

4. Implement pick_action() in qlearning/q_learning.py

5. Implement compute_value_from_q_values() in qlearning/q_learning.py

6. Implement pick_optimal_action() in qlearning/q_learning.py

7. Implement calculate_reward_for_move() in qlearning/q_utils.py

8. In run() (qlearning/q_utils.py) implement the update rule for the q table

9. Now we are ready to try to run the code… Run qlearning/q_learning.py. May the reinforcement learning-Gods be with you…

10. If your pacman is really stupid and aren't able to solve your level, try to change the rewards, the exploration probability or number of episodes. Or maybe your q table update function is wrong? Alpha and epsilon values should be ok.

Assignment 2: Deep Q-Learning

The two files to touch now is qlearning/deep_q_learning.py and qlearning/q_config.py.

1. Create your own map again. This time only your imagination is the limit!!! (Not really, start with a rather small map and if you solve it make it bigger)

2. Implement init_model() - qlearning/deep_q_learning.py

3. Implement convert_state_to_int() - qlearning/deep_q_learning.py. This is a hard one…

4. Implement pick_optimal_action() - qlearning/deep_q_learning.py

5. Make sure the input layer of the neural network is the same size as the output of the convert_state_to_int() method.

6. Run the code on a GPU. We haven't run the code on big maps on a GPU ourselves yet, so let's solve world problems together :-)

GPU guide:

Code to run Q-Learning. Note some things (such as exploration_prob and rewards are not optimal, but they work).

1. No code
2. No code

3. self.q_table = {}
   self.config = QConfig()

4.
```
exploration_prob = 0.35
if exploration_prob > np.random.rand():
    # Explore
    return np.random.choice(Action.get_all_actions())
else:
    # Exploit
    return self.pick_optimal_action(game_state)
```

5.
```
if state not in self.q_table:
    self.q_table[state] = {key: 0.0 for key in Action.get_all_actions()}
```

```
return max(self.q_table[state].values())
```

6.
```
if state not in self.q_table:
    self.q_table[state] = {key: 0.0 for key in Action.get_all_actions()}

max_value = max(self.q_table[state].values())
actions = [key for key in self.q_table[state] if self.q_table[state][key] == max_value]

if printing:
    print(state)
    print(self.q_table[state])

return random.choice(actions)
```

7.
```
def calculate_reward_for_move(action_event):
    if action_event == ActionEvent.DOT:
        return 1
    elif action_event == ActionEvent.CAPTURED_BY_GHOST:
        return -5
    elif action_event == ActionEvent.NONE:
        return -0.1
    elif action_event == ActionEvent.WALL:
        return -0.1
    elif action_event == ActionEvent.WON:
        return 10
    elif action_event == ActionEvent.LOST:
        return -10
    return 0
```

8.
```
self.q_table[current_game_state][action] = self.q_table[current_game_state][action] + self.config.alpha *
(reward + (self.config.discount * self.compute_value_from_q_values(next_game_state)) -
self.q_table[current_game_state][action])
```
(This is one line btw)

Code for Deep Q Learning:

1. No code

2.
```
self.model = Sequential()
self.model.add(Dense(self.config.hidden_size, input_shape=(self.config.input_size,), activation='relu'))
self.model.add(Dense(self.config.hidden_size, activation='relu'))
self.model.add(Dense(self.config.num_actions))
self.model.compile(sgd(lr=.01), "mse")

return self.model
```

3.
```
string_rep = state.__str__()
r = np.array([])

for char in string_rep:
    # if char == '%':
    #     r = np.concatenate([r, [0, 0, 0, 0, 1]])
    if char == ' ':
        r = np.concatenate([r, [0, 0, 0, 1, 0]])
    if char == 'P':
        r = np.concatenate([r, [0, 0, 1, 0, 0]])
    if char == 'G':
        r = np.concatenate([r, [0, 1, 0, 0, 0]])
    if char == '.':
        r = np.concatenate([r, [1, 0, 0, 0, 0]])

return r.reshape(1, self.config.input_size)
```

4.
```
q = self.model.predict(self.convert_state_to_input(state))
return Action.get_all_actions()[np.argmax(q[0])]
```