TeamProject Report

team number:9

leader	3180103618	范宏禹	计科 1803
members	3180102173	王子腾	软工 1801
members	3180103654	沈乐明	软工 1801
members	3180103422	徐晓丹	软工 1802

2019.07.19

CONTENTS

1. Project Introductions

1) Topic Selection

a. Project Name: Pacman(吃豆人)

b. Environment: OpenAI.Gym & Atari

c. Installation: Install from Pycharm Settings-Project Interpreter-Available Packages Installation/We download the environment from github.

2) Feedback from game

a. Termination condition: The Pacman loses his life or has eaten all the beans.

b. Observation: configuration(including the (x,y) coordinate of a character along with its traveling direction), speed, scared, etc. Ghost has two mode: classic and eatable. After pacman eat the diamond, ghost will change to eatable mode for 10 second, when it can be eaten and its speed decreasd by 50%.

c. Action: north, south, east, west, stop.

d. Actions Encoding: Use the generateSuccessor function to generate a new configuration by translating the current configuration by the action vector. Actions are movement vectors. Use Onehot code to represent the movement vectors. To be extent, (0,1), (0,-1), (1,0), (-1,0), (0,0) represent north, south, east, west and stop, respectively.

e. Reward: If the Pacman successfully eat a bean(food), the score will increase by 10; If the pacnman successfully eat a ghost in eatable mode, the score will increase by 500; If the pacman is caught by a ghost, the score decrease 500 and end game; Every move cost 1 score. Details are listed at following chart:

movement	score
eat a bean	+10
eat a ghost	+500
eaten by ghost	-500
every action	-1

3) Brief Introduction to work

a. Target: Use reinforcement learning to train the agent so as to get a relatively high score after training for certain times.

b. Algorithms: DQN and Double-DQN(failed)

a) Description: Use DQN to maximize the score, tring to use Double DQN to do the work but failed.

b) Characteristics

Advantages:

(1)Use DNN network for Q value function fitting and use end-to-end model.

(2)Apply Experience Replay mechanism(经验回放). It builds a storage to store all the samples, and removes the correlation by random sampling.

(3)Separate Target Network(双网络结构). An independent target Q-network slower than the current Q-network is constructed to calculate y, which reduces the possibility of shock and divergence in training and makes it more stable.

Disadvantages:

(1)Off-policy is limited, and it's not really online-learning.

(2)The problem of overestimation (compared to double DQN).

c) Sphere of application:

4) Development Environment & System Operation Requirements

a. Development Environment: python == 3.5.1 ; tensorflow == 0.8rc

a) Operating system: Win 10

b) Development tool: Pycharm (using Python 3.7); pycharm / VS code; jupyter-notebook

c) Packages: Gym; Numpy; TensorFlow; Atari; pandas; matpoltlib;

b. System Operation Requirements: Pac-man implementation by UC Berkeley: The Pac-man Projects - UC Berkeley (http://ai.berkeley.edu/project_overview.html)

2. Technical Details

1) Theoretical Basis

a. Value-based RL

b. Convolutional Neural Network (CNN):

The deep neural network we choose is `convolutional neural network`. A convolutional neural network consists of an input and an output layer, as

well as multiple hidden layers. The hidden layers of a CNN typically consist of a series of convolutional layers that *convolve* with a multiplication or other dot product. The activation function is commonly a `RELU layer`, and is subsequently followed by additional convolutions such as `pooling layers`, fully connected layers and normalization layers, referred to as hidden layers because their inputs and outputs are masked by the activation function and final convolution. The final convolution, in turn, often involves `backpropagation` in order to more accurately weight the end product.

c. Loss function and optimization

Considering Quality-based algorithms returns predicted quality, we use `MSE`as Loss function:

$$L(\theta) = (self.y_j - self.Q_{pred})^2$$

We choose `Adam Optimization` as the optimizer instead of `Gradient decent`. The algorithms leverages the power of adaptive learning rates methods to find individual learning rates for each parameter. It also has advantages of `Adagrad`, which works really well in settings with sparse gradients, but struggles in non-convex optimization of neural networks, and `RMSprop`, which tackles to resolve some of the problems of `Adagrad` and works really well in on-line settings.

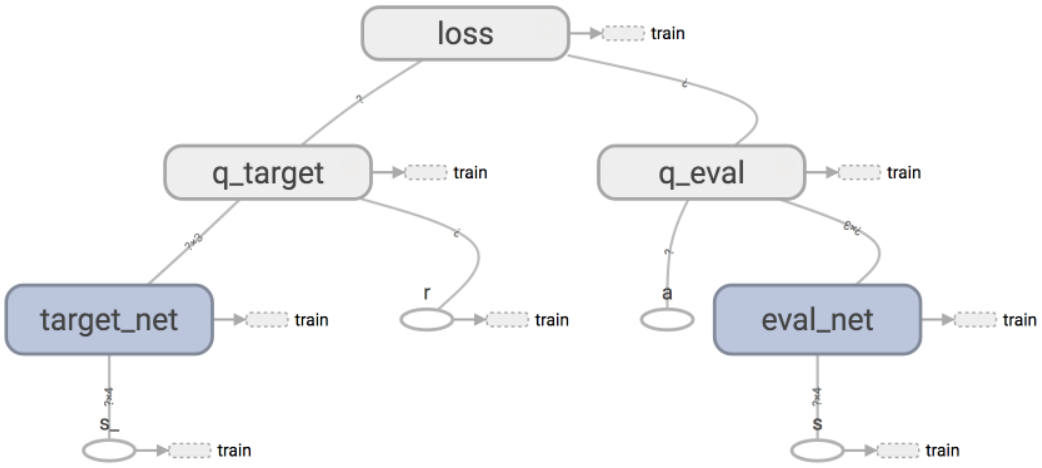
2) Algorithms

a. Deep Q-Network (DQN):

The Pseudocode of DQN is listed as follow:

Algorithm 1 Deep Q-learning with Experience Replay

Initialize replay memory $\mathcal{D}$ to capacity N
Initialize action-value function Q with random weights
for episode $= 1, M$ **do**
  Initialise sequence $s_1 = \{x_1\}$ and preprocessed sequenced $\phi_1 = \phi(s_1)$
  for $t = 1, T$ **do**
    With probability ϵ select a random action a_t
    otherwise select $a_t = \max_a Q^*(\phi(s_t), a; \theta)$
    Execute action a_t in emulator and observe reward r_t and image x_{t+1}
    Set $s_{t+1} = s_t, a_t, x_{t+1}$ and preprocess $\phi_{t+1} = \phi(s_{t+1})$
    Store transition $(\phi_t, a_t, r_t, \phi_{t+1})$ in $\mathcal{D}$
    Sample random minibatch of transitions $(\phi_j, a_j, r_j, \phi_{j+1})$ from $\mathcal{D}$
    Set $y_j = \begin{cases} r_j & \text{for terminal } \phi_{j+1} \\ r_j + \gamma \max_{a'} Q(\phi_{j+1}, a'; \theta) & \text{for non-terminal } \phi_{j+1} \end{cases}$
    Perform a gradient descent step on $(y_j - Q(\phi_j, a_j; \theta))^2$ according to equation 3
  end for
end for



b. Double Deep Q-Network:

The difference between DQN and Double-DQN lies in the `q_target`. This method**handles the problem of the overestimation of Q-values.** when we compute the `q_target`, we use two networks to decouple the action selection from the `q_target` value generation. Firstly, we use our DQN network to select what is the best action to take for the next state (the action with the highest Q value). Then we use our target network to calculate the `q_target` value of taking that action at the next state. In this way the action we choose may not be the seeming 'best' step with respect to the previous step, but it is the best step considering the whole methods:

$$Y_t^{DQN} = R_{t+1} + \gamma maxQ(S_{t+1}, a; \theta_t^-)$$

$$Y_t^{DoubleDQN} = R_{t+1} + \gamma maxQ(S_{t+1}, argmaxQ(S_{t+1}, a; \theta_t); \theta_t^-)$$

3) Technical Details

The RL*brain is written in `DQN.py`, the connection between RL*brain and environment is in class `PacmanDQN` of `pacman_Agents.py`

a. Important Functions

1. `observation_step(self, state)`

directory: panman_Agent.py ; class PanmanDQN input: `state` output: none introduction: Process current experience state and reward; Store last experience and model into memory ; use function `train(self)`to start training.

2. `train(self)`

directory: panman*Agent.py ; class PanmanDQN input: none output:none introduction: This function is the bridge between `rungame` and main training function, which is `train(batch*s,

batch*a, batch*t, batch*n, batch*r)`in`DQN.py`.It randomly select batch of previous memory and feed the data into function`train(batch*s, batch*a, batch*t, batch*n, batch*r*).batch*s`is **state**,`batch*a`is **actions**,batch*t`is **terminal state**,`batch*n`is **next state**,batch*r`is **rewards**.

3. `train(batch_s, batch_a, batch_t, batch_n, batch_r)`

directory: DQN.py ; class DQN input:`batch_s` is **state**, `batch_a` is **actions**,`batch_t`is **terminal state**, `batch_n` is **next state**, `batch_r`is **rewards**. output: `cnt`: times of training； `` `cost``: the value of Loss introduction: Inside the function is the main structure of DQN, where we use our network to calculate `cost`

4. `build_layers(x)`

directory: DQN.py ; class DQN input: x, which is the quality of previous step. output: prediction of next step's quality. introduction: Inside the function is a neural network of 1 convolution layer, 1 fully connected layer and an output layer.

b. Mathematical Basis

Bellman equation(Q_target): $Q^*(s,a) = E_{s',\delta}|r + \gamma max Q^*(s',a')|s,a$

Loss function:

$L_i(\theta_i) = (self.y_j - self.Q_{pred}(s,a,\theta_i))^2$ where $y_i = E_{s',\delta}|r + \gamma max Q^*(s',a',\theta_{i-1})|s,a$

Backward pass:$\frac{\partial L_i(\theta_i)}{\partial \theta_i} = E_{s',\delta}|r + \gamma max Q^*(s',a',\theta_{i-1}) - Q(s,a;\theta_i)|s,a \frac{\partial Q(s,a;\theta_i)}{\partial \theta_i}$

c. Highlight

1. Neural Network Structure

We designed three types of Neural Network Structure:

number	convolution layer(filters)	fully connected layer(hiddens)	output
A	2(16)	1(256)	1
B	**1(16)**	**1(256)**	**1**
C	0	1(256)	1

The experiments showed that A is indeed accurate, but the speed is unbearably slow. It takes a CPU to run a day for 4000 epochs. B is also accurate, and the speed is way much faster than A. We finished the training(4000 epochs) in 4 hours. C is a lot more faster, the experiment finished in 20 minutes, but the Pacman seemed so dumb that it had no difference with a pre-trained Pacman or me.

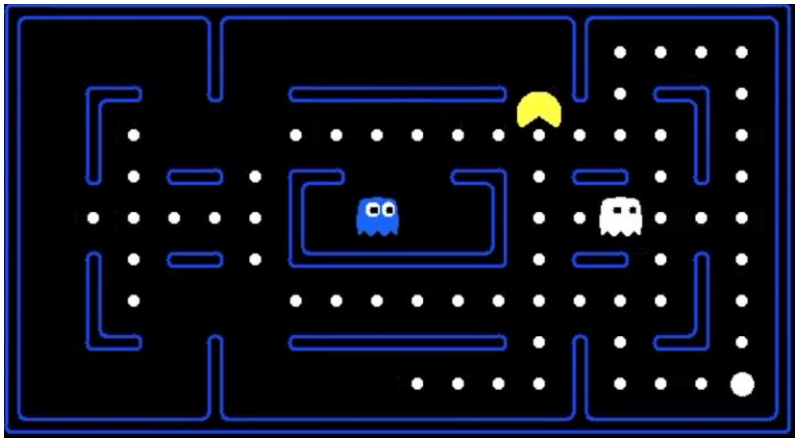
To balance speed and accuracy, Network B is our choice.

1. New function `` `build_layers``: Simplify the training code by move the neural network into a function.

3. Experiment Results

The result is good in maps such as `testClassic` and `smallGrid`, where Pacman can never die. Our evaluating map is `smallClassic`, which is more complicated than the previous two map, and the Pacman has a chanse of 70% to win after training.

smallClassic:



testClassic:

Training results

```
#  995 | steps: 11734 | steps_t:  6192 | t: 55.956927 | r:  -507.000000 | e:   0.380800 | Q: -258.520386 | won: False
#  996 | steps: 11746 | steps_t:  6203 | t: 56.048681 | r:  -511.000000 | e:   0.379700 | Q: -282.435028 | won: False
#  997 | steps: 11768 | steps_t:  6224 | t: 56.238175 | r:  -510.000000 | e:   0.377600 | Q: -256.501343 | won: False
#  998 | steps: 11781 | steps_t:  6236 | t: 56.342894 | r:  -501.000000 | e:   0.376400 | Q: -261.505310 | won: False
#  999 | steps: 11796 | steps_t:  6250 | t: 56.467563 | r:  -503.000000 | e:   0.375000 | Q: -246.733215 | won: False
# 1000 | steps: 11810 | steps_t:  6263 | t: 56.580294 | r:  -513.000000 | e:   0.373700 | Q: -288.123535 | won: False
# 1001 | steps: 11829 | steps_t:  6281 | t: 56.729859 | r:  -507.000000 | e:   0.371900 | Q: -293.471222 | won: False
# 1002 | steps: 11839 | steps_t:  6290 | t: 56.807651 | r:  -509.000000 | e:   0.371000 | Q: -281.803589 | won: False
# 1003 | steps: 11844 | steps_t:  6294 | t: 56.846547 | r:  -504.000000 | e:   0.370600 | Q: -267.124420 | won: False
# 1004 | steps: 11862 | steps_t:  6311 | t: 56.993156 | r:  -506.000000 | e:   0.368900 | Q: -244.101440 | won: False
# 1005 | steps: 11877 | steps_t:  6325 | t: 57.120816 | r:  -514.000000 | e:   0.367500 | Q: -240.003601 | won: False
```

In the first 1000 trainings, the chance of winning is nearly zero, and the choice of movement is still random.

```
# 2795 | steps: 54111 | steps_t: 46769 | t: 431.815266 | r:  -510.000000 | e:   0.100000 | Q: 183.188217 | won: False
# 2796 | steps: 54146 | steps_t: 46803 | t: 432.125436 | r:  -523.000000 | e:   0.100000 | Q: 116.780281 | won: False
# 2797 | steps: 54187 | steps_t: 46843 | t: 432.474551 | r:    71.000000 | e:   0.100000 | Q: 187.322235 | won: True
# 2798 | steps: 54267 | steps_t: 46922 | t: 433.157714 | r:  -568.000000 | e:   0.100000 | Q: 149.530502 | won: False
# 2799 | steps: 54304 | steps_t: 46958 | t: 433.473871 | r:    75.000000 | e:   0.100000 | Q: 235.172562 | won: True
# 2800 | steps: 54340 | steps_t: 46993 | t: 433.789985 | r:    76.000000 | e:   0.100000 | Q: 339.309235 | won: True
# 2801 | steps: 54393 | steps_t: 47045 | t: 434.364451 | r:  -541.000000 | e:   0.100000 | Q: 181.515518 | won: False
# 2802 | steps: 54408 | steps_t: 47059 | t: 434.519036 | r:  -503.000000 | e:   0.100000 | Q: 124.135292 | won: False
# 2803 | steps: 54455 | steps_t: 47105 | t: 434.947892 | r:  -535.000000 | e:   0.100000 | Q: 206.744156 | won: False
# 2804 | steps: 54496 | steps_t: 47145 | t: 435.293966 | r:  -529.000000 | e:   0.100000 | Q: 106.387047 | won: False
# 2805 | steps: 54563 | steps_t: 47211 | t: 435.844493 | r:  -555.000000 | e:   0.100000 | Q: 276.593170 | won: False
```

Then we selected the results after 1800 more trainings, from which we can see the chance of winning grows to about 30%, and the Pacman becomes more smart from the previous experience.

```
# 4095 | steps: 98041 | steps_t: 89399 | t: 837.488427 | r:    87.000000 | e:   0.100000 | Q: 242.318939 | won: True
# 4096 | steps: 98075 | steps_t: 89432 | t: 837.809543 | r:    78.000000 | e:   0.100000 | Q: 269.661438 | won: True
# 4097 | steps: 98095 | steps_t: 89451 | t: 837.994027 | r:  -508.000000 | e:   0.100000 | Q: 165.508850 | won: False
# 4098 | steps: 98132 | steps_t: 89487 | t: 838.330128 | r:    75.000000 | e:   0.100000 | Q: 373.421326 | won: True
# 4099 | steps: 98165 | steps_t: 89519 | t: 839.217892 | r:    79.000000 | e:   0.100000 | Q: 318.603424 | won: True
# 4100 | steps: 98185 | steps_t: 89538 | t: 839.376532 | r:    92.000000 | e:   0.100000 | Q: 233.918228 | won: True
# 4101 | steps: 98227 | steps_t: 89579 | t: 839.722545 | r:  -530.000000 | e:   0.100000 | Q: 249.614807 | won: False
# 4102 | steps: 98248 | steps_t: 89599 | t: 839.904098 | r:    91.000000 | e:   0.100000 | Q: 188.969559 | won: True
# 4103 | steps: 98268 | steps_t: 89618 | t: 840.087569 | r:    92.000000 | e:   0.100000 | Q: 193.612762 | won: True
# 4104 | steps: 98302 | steps_t: 89651 | t: 840.399733 | r:    78.000000 | e:   0.100000 | Q: 274.017609 | won: True
```

Gradually, after 4100 trains, we print the states and results on the screen. Compared with two previous condition, this time the winning possibility grows to nearly 70%, and the program was able to make decision with current information. A simple AI is able to work out the game Pacman more skillfully.

Example usage

Run a model on `smallGrid` layout for 6000 episodes, of which 5000 episodes are used for training.

```
$ python3 pacman.py -p PacmanDQN -n 6000 -x 5000 -l smallGrid
```

Layouts

Different layouts can be found and created in the `layouts` directory

Parameters

Parameters can be found in the `params` dictionary in `pacmanDQN_Agents.py`. Models are saved as "checkpoint" files in the `/saves` directory. Load and save filenames can be set using the `load_file` and `save_file` parameters. Episodes before training starts: `train_start` Size of replay memory batch size: `batch_size` Amount of experience tuples in replay memory: `mem_size`

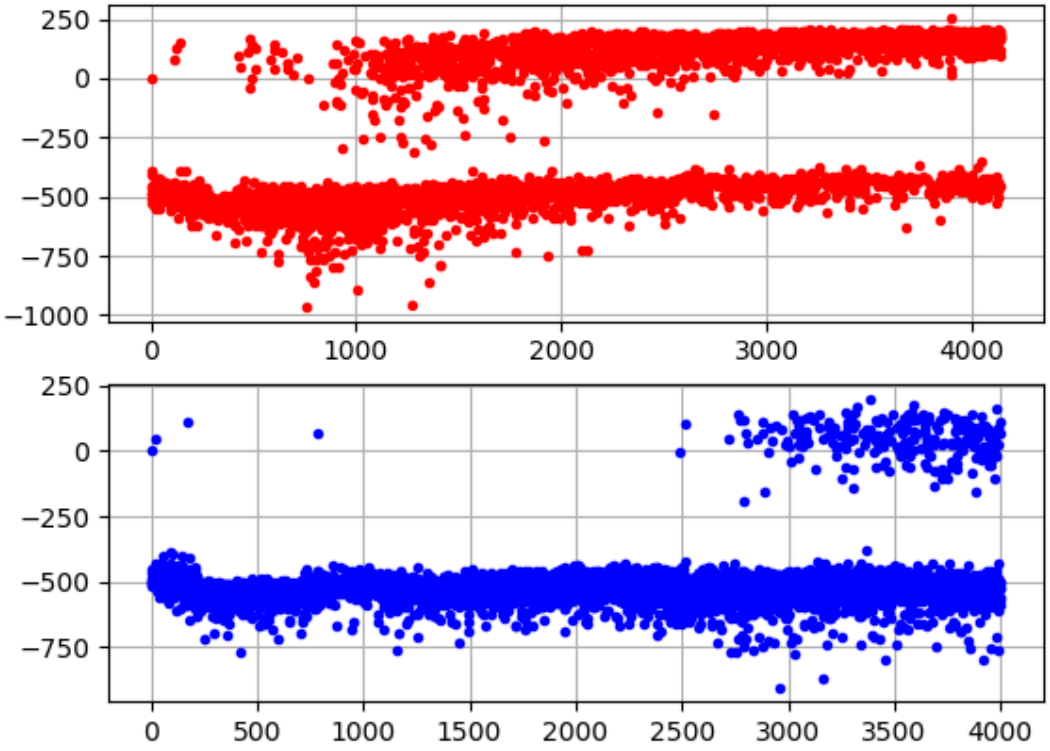
Discount rate (gamma value): `discount` Learning rate: `lr` Number of steps between start and final epsilon value (linear): `eps_step`

Summary:

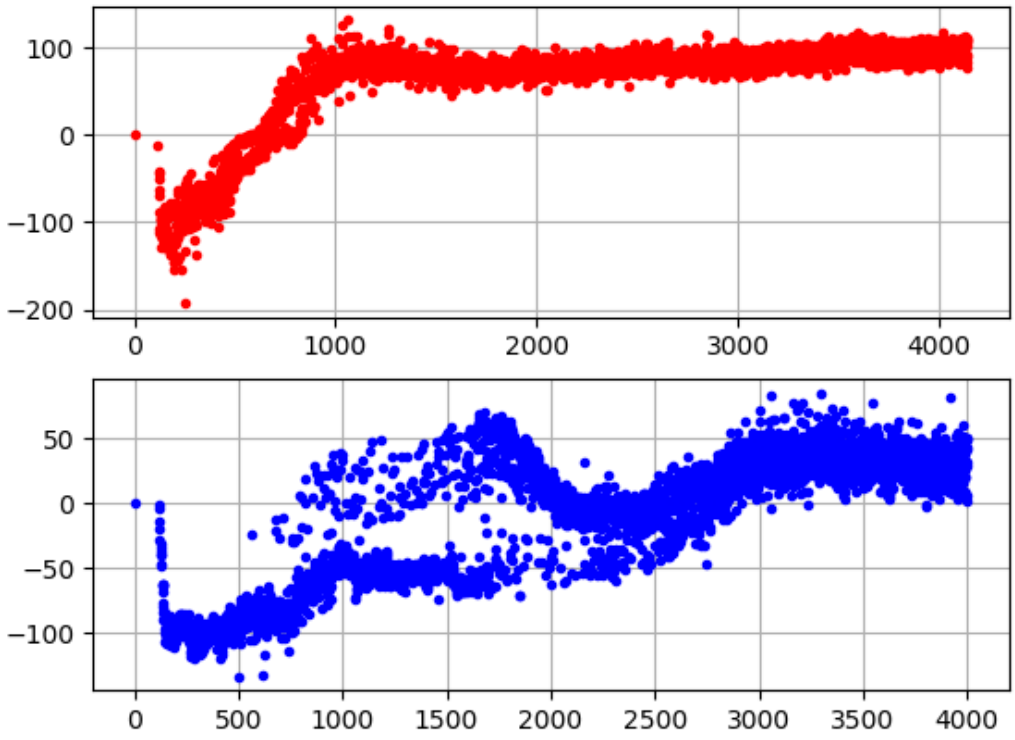
We decided to firstly choose the environment then the algorithms. After choosing Pacman as environment, we started at the easiest algorithm: DQN. The result is good, the network can be trained but is very slow. Therefore we spend a lot of time optimizing the Neural Network Structure and the result is positive.

We tried to write double-DQN on the basis of DQN.py, and the program CAN run but it always seems to initialize after one training. We estimate the bug is in our connection between the algorithm and the environment. To be frank this Pacman environment provided by UCB is a little messy, and due to our lack of time and knowledge, we failed to find it out.

***Reward*:**



Quality:



After visulization the result is in the up two graphs. In each graph is the result of two type of MAP: **testClassic** and **smallClassic**, which has different complexity. The **Reward graph** shows that after 3000 training the Pacman starts to win regularly, the more complex the map is, the slower Pacman begins to win regularly. The **Quality graph** shows that as the complexity grows, the stability decreases. To sum up, DQN is only suitable at easy map such as **testClassic**, complex map such as **smallClassic** and **minimaxClassic** requires a better algothrim or more training time.

4. References

1) Python-tutorial

2) Same Stats, Different Graphs: Generating Datasets with Varied Appearance and Identical Statistics through Simulated Annealing, Justin Matejka and George Fitzmaurice

3) Reinforcement Learning: An Introduction, Richard S. Sutton and Andrew G. Barto

4) Statistics and Machine Learning in Python, Edouard Duchesnay, Tommy Löfstedt

5) 《深度学习》

6) Introduction to Computer Science Using Python(A Computational Problem-Solving Focus)

7) Data Structures and Algorithms Using Python