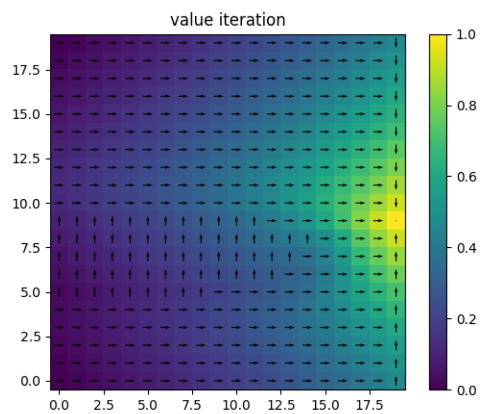
CS 237B: Principles of Robot Autonomy II
Problem Set 1

Name: Abhyudit Singh Manhas
SUID: 06645995

Problem 1: Markovian Drone

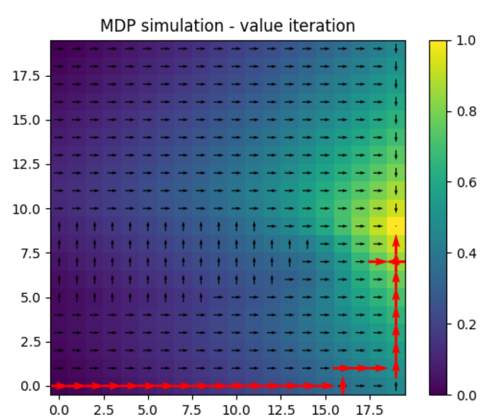
(i) Completed `value_iteration.py`.

(ii) The heatmap of the optimal value function is shown below. The optimal policy is visualised by the black arrows.



(iii) Computed an optimal policy in the `visualize_value_function()` function of `utils.py`. Used this optimal policy to simulate the MDP in the `simulate_MDP()` function of `utils.py`.

(iv) The heatmap is shown below. The simulated drone trajectory is denoted by the red arrows.



The storm's influence is strongest at its center and decays farther from the center. There is a slight chance that the storm will cause the drone to move in a uniformly random direction instead of following the optimal policy. This is seen above and we can observe that the optimal policy corrects this whenever the drone deviates from the optimal trajectory. This is because it is a closed-loop control policy.

(v) Sampled 10^5 state transition tuples in `q_learning.py`.

(vi) The expectation form of the optimal Q-function is:

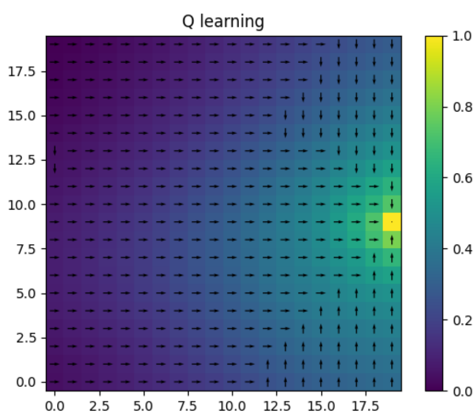
$$Q^*(\boldsymbol{x}_t, \boldsymbol{u}_t) = \begin{cases} E[r(\boldsymbol{x}_t, \boldsymbol{u}_t) + \gamma \max_{\boldsymbol{u}' \in \mathrm{U}} Q^*(\boldsymbol{x}_{t+1}, \boldsymbol{u}')], & \text{if } \boldsymbol{x}_t \text{ is not a terminal state} \\ E[r(\boldsymbol{x}_t, \boldsymbol{u}_t)], & \text{otherwise} \end{cases} \tag{1}$$

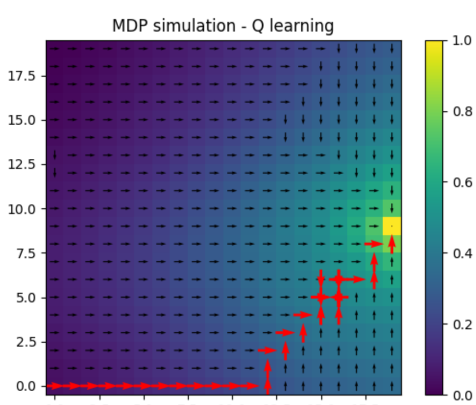
(vii) Created the feed forward neural network in `q_learning.py`.

(viii) Completed `Q_learning()` in `q_learning.py`.

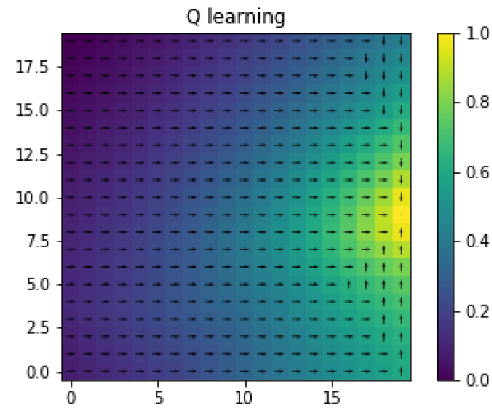
(ix) Q-learning is a model-free RL algorithm, whereas value iteration is a model-based algorithm. Thus, using Q-learning would be easier than using value iteration when the model dynamics are unknown. For this case, Q-learning would be preferred over value iteration if we didn't know the transition probabilities. Q-learning will be preferred for applications that involve an unknown environment, such as a self-driving car or a mobile robot/robot manipulator in an unknown environment. It could even be used in something like stock trading, where the market dynamics are very unpredictable and essentially unknown.

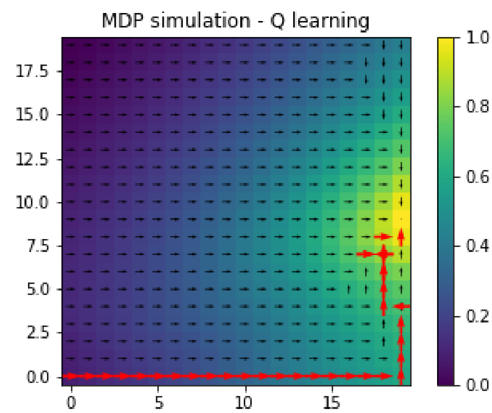
(x) The heatmap and simulated trajectory for a learning rate of 1e-2 is shown below.



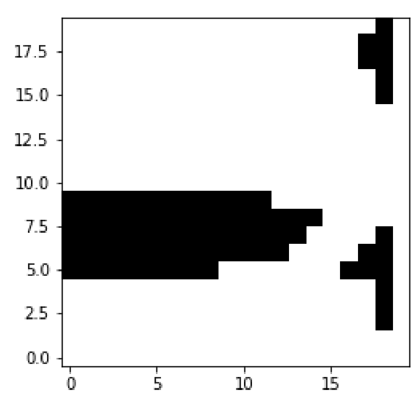


The heatmap and simulated trajectory for a learning rate of 1e-3 is shown below.





A learning rate of 1e-3 works best. For this case, a binary heat map of where the approximate Q-network policy agrees with the value iteration optimal policy is shown below. The regions where they agree is denoted by white, and where they disagree is denoted by black.



Problem 2: Classification and Sliding Window Detection

(i) Completed `get_bottleneck_dataset()` in `retrain.py`.

(ii) Created a linear classifier in `retrain.py`.

(iii) Merged both Inception-v3 and Linear Classifier models in `retrain.py`.

(iv) The dimension of each bottleneck image is 2048. We are optimizing 6147 parameters in the retraining phase. This is shown below.

```
 mixed10 (Concatenate)          (None, 8, 8, 2048)   0           ['activation_85[0][0]',
                                                                  'mixed9_1[0][0]',
                                                                  'concatenate_1[0][0]',
                                                                  'activation_93[0][0]']

 global_average_pooling2d (Glob  (None, 2048)        0           ['mixed10[0][0]']
 alAveragePooling2D)

==================================================================================================
Total params: 21,802,784
Trainable params: 21,768,352
Non-trainable params: 34,432
__________________________________________________________________________________________________
Generating Bottleneck Dataset... this may take some minutes.
Found 450 images belonging to 3 classes.
Done generating Bottleneck Dataset
WARNING:absl:`lr` is deprecated, please use `learning_rate` instead, or use the legacy optimizer, e.g.,tf.keras.optimizers.legacy.SGD.
Model: "model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 2048)]            0

 classifier (Dense)          (None, 3)                 6147

=================================================================
Total params: 6,147
Trainable params: 6,147
Non-trainable params: 0
_________________________________________________________________
5000/5000 [==============================] - 7s 1ms/step - loss: 0.0566 - accuracy: 0.9928
```

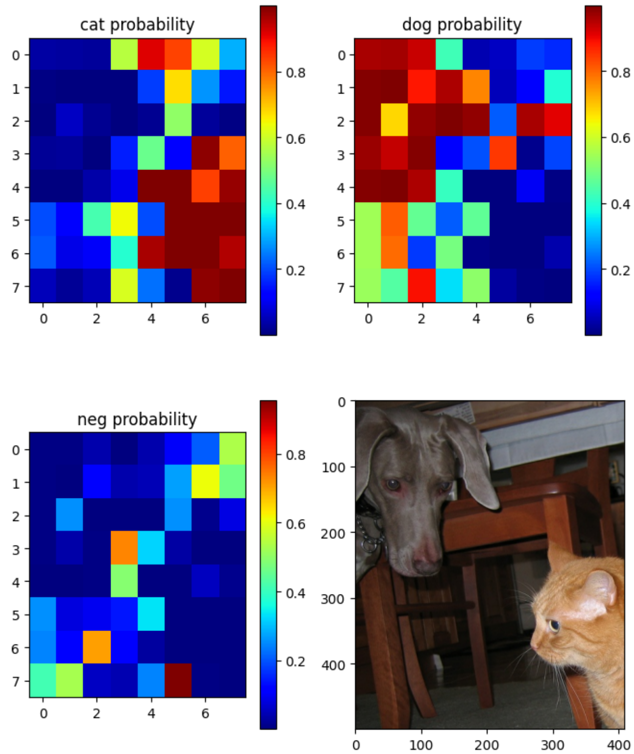
(v) Having dropout layers during training helps prevent overfitting. A dropout layer randomly sets input units to 0 with a frequency of `rate` (in `tf.keras.layers.Dropout(rate)`) at each step during training time. Inputs not set to 0 are scaled up by 1/(1 - `rate`) such that the sum over all inputs is unchanged. Hence dropout layers can saturate the nodes which can cause non-convergence of the classifier's parameters.

(vi) Completed `classify()` in `classify.py`. We get an accuracy of 90% on the test set.

```
Correct label is dog

Incorrectly classified: dog\008890.jpg
Predicted label is cat
Correct label is dog

Incorrectly classified: neg\000352.jpg
Predicted label is dog
Correct label is neg

Incorrectly classified: neg\002193.jpg
Predicted label is dog
Correct label is neg

Incorrectly classified: neg\004831.jpg
Predicted label is cat
Correct label is neg

Incorrectly classified: neg\006718.jpg
Predicted label is dog
Correct label is neg

Incorrectly classified: neg\007991.jpg
Predicted label is dog
Correct label is neg

Incorrectly classified: neg\008879.jpg
Predicted label is cat
Correct label is neg

Evaluated on 150 samples.
Accuracy: 90%
```

(vii) Completed `compute_brute_force_classification()` in `detect.py`.

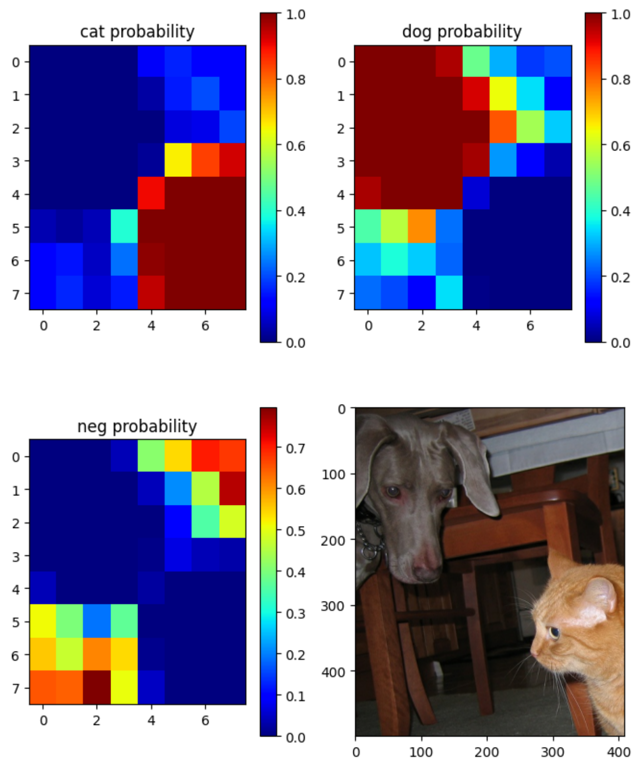
(viii) The detection plot is shown below:



(ix) The `mixed_10` layer is a concatenate layer, which concatenates `activation_85` (shape:(8,8,320)), `mixed9_1` (shape:(8,8,768)), `concatenate_1` (shape:(8,8,768)) and `activation_93` (shape:(8,8,192)). The feature vector for the image does not work very well, because in brute force classification we are using the image classifier on smaller sections (windows) of the image. This is different from what we did in training and testing, where we used the entire image of the cat or dog.

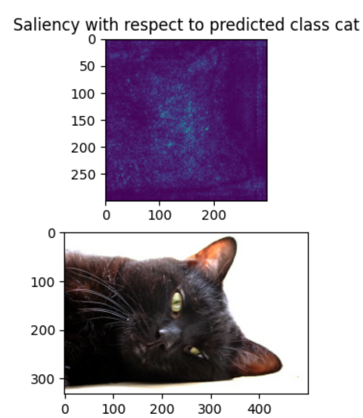
(x) Completed `compute_convolutional_KxK_classification()` in `detect.py`.

(xi) The detection plot is shown below:

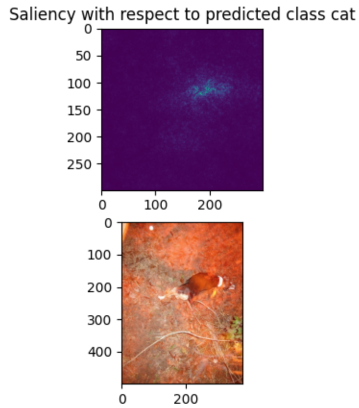


(xii) Completed `compute_and_plot_saliency()` in `detect.py`.

(xiii) The saliency plot for a correctly classified image is shown below:



The saliency plot for an incorrectly classified image is shown below:


Saliency with respect to predicted class cat
0
50
100
150
200
250
0 100 200
0
100
200
300
400
0 200