Playing Arcade Game with Object Recognition and Reinforcement Learning

Hongyu Tu, University of Massachusetts Amherst (hongyutu@umass.edu)
Dec. 1st - Final Report

Abstract

This paper is a report of an attempt to train a control algorithm that can autonomously play arcade games, using techniques within the field of computer vision and reinforcement learning. More specifically, three neural network models are trained to work together and come up with the best action to take in real-time given only visual input. The logic behind the control algorithm highly resembles the logic that was used in today's self-driving vehicles and can potentially lead to implications that are way more generalizable than the game itself. I found that it is possible to train a network to play the game without much prior knowledge on how to logic behind the game works and that in cases like this reinforcement learning has a significant advantage over supervised learning when it comes to learning a control policy for a game.

1. Introduction

Within the arcade video game, Crossy Road, players tries to cross the lanes of roads ahead as much as possible by circling around obstacles and avoid getting hit by moving vehicles. An advanced goal for this game is to collect coins that randomly appear on the map. This makes a great problem that can be solved with reinforcement learning, and this project aims to find a policy that will enable the computer to play this game without human interaction.

1.1. Details of the game

In the game, the player can control the character to go either left, right, forward, or backward. The road in front is infinite but only a limited amount of lanes will be shown at any given moment. As for horizontal movements, there's a bound of 9 positions per lane, which means if the character is currently at the center of a lane, it can only go 4 steps to the left and 4 steps to the right, no further than that. On the map, there are relatively safe lanes that have only obstacles, including trees, tree stumps, and stones that are stationary, which will only block the character and will not kill them.

When trying to leave the safe zone and make a road-crossing attempt, there will be vehicles that can be moving from either left or right toward the opposite side of the road, these vehicles have different velocities which will make it either harder or easier to find the exact timing to make the attempt because the character will die when hit by vehicles on the road.

Lastly, there are water lanes in which the player needs to step on water lilies (stationary) or floating logs (non-stationary) to avoid drowning. The floating logs have movement directions of either left or right as well and the player will move with it, if the player couldn't get off the log before the log goes out of bounds, the player dies as well. Lastly, the map (camera) is constantly moving forward so the player needs to constantly try to move forward to avoid the game from terminating.

1.2. Overall Plan

The way this paper purpose to solve this problem is by using a combination of three neural network models. Since the game is not open source, the only way an algorithm can gather information about this game is the same way humans do – visual. Therefore, the first neural network will be trained to identify objects in the game (make sense of the given scene) and gather the information that will be important for the next step. To make reasonable moves during the game, a reinforcement learning model will be trained with the information gathered in the previous step to obtain an optimal policy, which is the main goal of this project. And lastly, to put everything together, a third model will be trained to identify the status of the game (before game, during game, the game ends) so that it can act as an environment wrapper around the game so we can utilize our prior result and play the game.

2. Related Work

The reason that I come up with this project is from the inspiration of the work done by Kinsley[1] back in 2018. For his project, he tried to train a network model that gets screenshots from a computer game, processes them, and then tries to predict the keyboard output that turns the steering the wheel so the car can perform autonomous driving within the game. For the majority part, he used OpenCV for image recognition and then used Adam for the later pre-

diction.

One paper that has the most to do with my project is Playing Atari with Deep Reinforcement Learning, by Minh *et al*. [2]. In the paper, they passed in the game screenshots in pixels into a deep network model and used a variant of Q-Learning to learn the control policy for each game and was able to achieve near-human performance. Due to the resolution of the Atari games are low and therefore not much reduction is needed, however, in this particular case, the game crossy road has a lot more resolution from gameplay and it's heavily animated, therefore, the density of information is a lower for the problem that this paper aims to solve. I decided to go with a different variant of Q-Learning, which will be discussed in detail in the next section, and how the density of information per pixel is increased will be discussed as well.

Another paper from the same author above, by Minh *et al*. [3], was found to have more details about the algorithm that they used, specifically how they implemented the deep learning features. It's great when it comes to trying to learn one model for a lot of games. However, deep learning has gone beyond the skills that I have right now as well as the computational power that was needed. Therefore, I had my project set up to be a simpler version of the combination of the three projects that I have mentioned in this section and had some adjustments so that the three parts can work together.

3. Approach

This section will discuss in detail how I decided to implement the three models. To provide a big picture here, for the final model, there will be an infinite loop being run. Within the loop, a screenshot will first be taken from the game, after some pre-processing, the model for identifying game status will first try to see if the character is in-game, and start the game if not. In cases that the agent is in-game, we continue to use the object detection model to make sense of the environment, and lastly, pass the information from the object detection model into our learned policy from the reinforcement learning model to make a prediction.

3.1. Screenshot Pre-Processing

For my particular setting, I'm running the game in 666 $\times$ 1280, which will be the size of the initial input image size. Since the camera angle of the game is distorted (it's in 3D), the warp Perspective function from OpenCV will be used to compress the input image into 2D, which makes the size of the images smaller. Lastly, we will bring the image size down and grayscale it as the color isn't necessary for the algorithm to learn the game.

The process image function is written to convert a screenshot into usable data that has a smaller size and correct 2D perspective.

a) Take screenshot

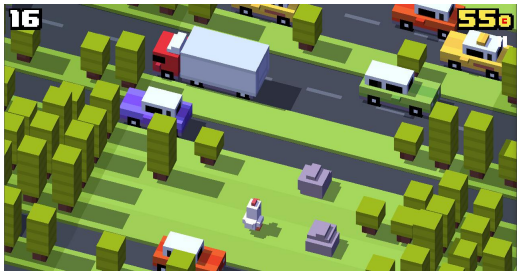

Figure 1. Unprocessed Screenshot

b) Use filter to keep only the ROI & grayscale. (The detailed coordinate math is shown in the top image below, and the second image is an example showing the output after this step.)

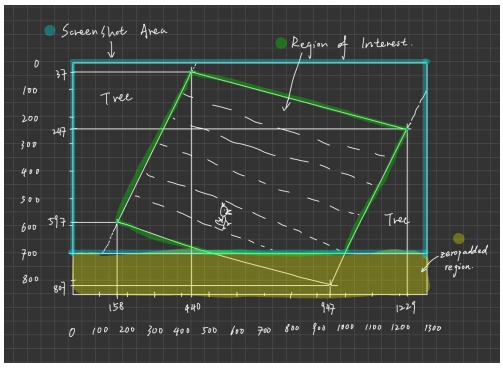

Figure 2. Coordinate information

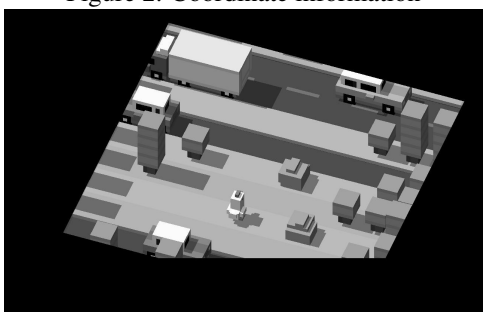

Figure 3. Screenshot after ROI

c) Warp Perspective

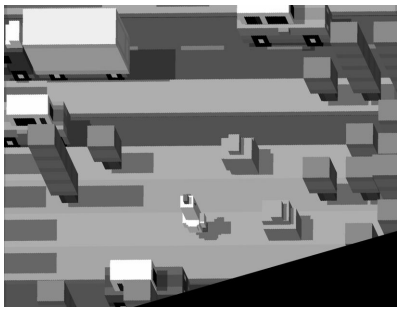

Figure 4. Screenshot transformed to 2D

d) Re-scale (Optional)

Figure 5. Input resized to 32 by 32

3.2. Model for Object Detection

The model for Object Detection I decided to go with is YOLO v5. Since the graphic style of the game is more pixel-like compared to the real world, pre-trained models performed extremely badly. Therefore, the weights have to be trained with the screenshots of the game. More specifically, we have identified the types of objects in the game that the model should be able to distinguish in order to make good decisions, which includes: [**agent**, **obstacle**, **vehicle**, **water lilies**, **coin**, **log**], and they are encoded by integer 0 to 5. With the custom data passed into YOLO v5, the output will include the coordinates of identified objects along with the predicted label.

Figure 6. Input image before going through YOLO

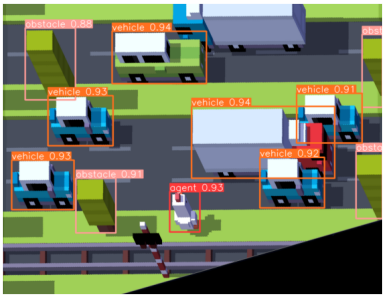

Figure 7. Output of YOLO with detected objects boxed

3.3. RL State Representation

Continuing from the Object Detection part, the YOLO model outputs a list of matches that it finds, and each row contains the predicted label and the x, y coordinates of the detected object. With that information, it's easy to just throw away the background and keep only the object information.

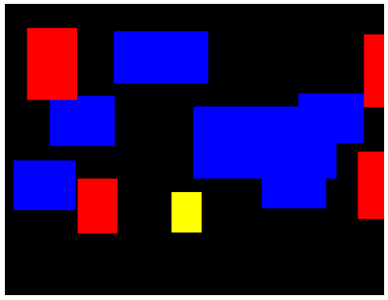

Figure 8. Before state transformation

However, this is still too much unnecessary information left. To further reduce the redundant information, recall that each screenshot contains 9 vertical lanes and each lane contains 9 positions, which is equivalent to a 9 by 9 matrix. To obtain the correct transformation, edge detection was used to figure out the exact position of each block.

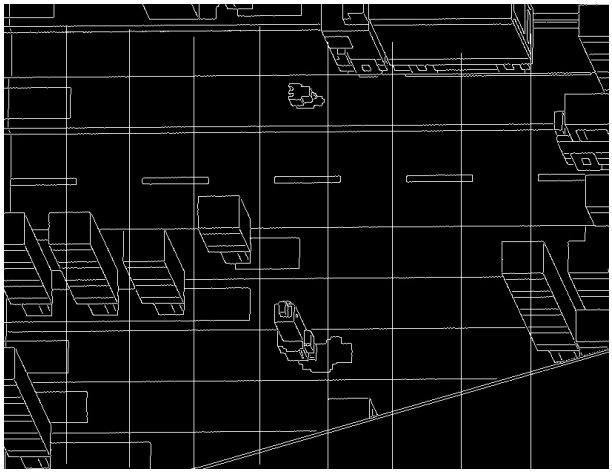

Figure 9. Image with edge detection and showing grid

The block corners with given row and column index (r, c) can be calculated by:

$$x_{min} = \max(0, c \times 90 - 5)$$
$$x_{max} = (c+1) \times 90 - 5$$
$$y_{min} = r \times 68 + 30$$
$$y_{max} = (r+1) \times 68 + 30$$

Another problem that surfaced during the implementation of this process is that there are a lot of times the grids don't perfectly align, and the view is still not strictly 2D even with the wrap perspective. For example, in figure 9, row 3, column 3, it should be empty as the tree stems from index row 4 column 3. However, the algorithm doesn't have that knowledge. Another example is that in figure 8, the huge blue block on the mid-right area would suggest it's filled with vehicles, but, from human reasoning, we know it's only because the red truck in figure 7 blocks the empty lane above it. To fix this problem, a threshold was added to determine the correct positioning:

- A block will be considered as taken by a type of object only if the number of pixels of that type of object filled over 40 percent of the entire block. If there are two different types of objects in the same block that both fit the criteria, pick whichever is higher.
- When there are two consecutive blocks in a certain column identified as filled with vehicle or obstacle, keep only the lower one
- Continuing on the previous point, in case of the agent taking up two consecutive blocks in a column, the top block of the two will be set to the second most seen object in that block but set to empty if the second-largest type contains less than 10 percent of the block. This is important for the agent to see the item in front of it as it always gets blocked out a lot by the agent.

Up to this point, we can continue on the example above and obtain a 9 by 9 representation like this:

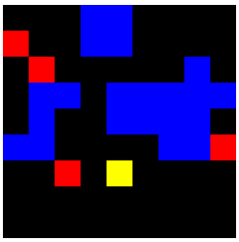

Figure 10. Full State Representation of size 9 by 9

To further simply the state representation, the final representation will always be centered at the agent (at index r = 2, c = 2). In the case of the agent being on the side, the 9 by 9 representation will be padded with obstacles before taking the 5 by 5 slice. And here is the visual representation of the final state representation that is used for the next section.

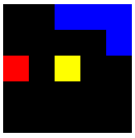

Figure 11. Partial State Representation of size 5 by 5

3.4. Reinforcement Learning

To identify the game as a reinforcement learning problem, usually, an MDP including states, actions, transition function, and reward, needs to be formally defined. However, this game isn't strictly Markovian, as the future states depend on randomly generated items defined within the game logic, and the whole game is only partially observable. Therefore, I decided to go with defining the state representation, action set, heuristic reward, transition function, and state-action value approximation. To start with, the action set will include five actions: [**Left** , **Right**, **Forward**, **Backward**, **NOP**].

For the simplicity of the model, the agent will look at its surrendering environment within a distance of 2, which can be represented by a 5 by 5 matrix. Since the center is defined to always be the player, the middle will always be treated as an empty space, and all empty spaces will be indicated by value 0. For all the other entries, the value that can be taken will be integer value from 1 to 5, corresponding to the types of the object identified (More detail on this can be found in section 3.2). Due to the fact that the location information is extremely important, I made the decision to keep the representation in 2 dimensional instead of flattening it.

For the heuristic move function, it basically implements the rules that the agent should know, or in other words, it tells the agent the consequences when taking certain actions at a certain state. Note that indicating the rules is not the same as letting the agent what to do, and this makes the problem a reinforcement learning problem instead of a supervised learning problem. This move function could have been done by simply capturing the game history, but the game runs in real-time, and can only take one action at a given time, which will make the training process extremely slow and eliminates our attempt to check what happens when we try different actions at a given state. Therefore, a function that takes in the state and action called step takes care of the reward and transition function at the same time. First, it looks at the action, if it moves forward, a row will be padded in front, the probability that the next row has vehicles is 0.75, and the probability of the next row has only obstacle is 0.25. For backward, left, right, all will be padded with a row full of obstacles and then crop out the last row/column in the opposite direction of movement. The next step is to simulate the movement of vehicles and logs, which will go either left or right with equal probability. This is the rule of how the game is updated, which will be our transition function.

For the reward part, we check if the action was a success or a fail by checking the center value. If it's empty, which means the agent has no problem standing there, it's a successful movement and will get a default reward of -1 for cases that are not moving forward. On the other hand, if the success has resulted from a forward action, it will have a reward of -1, so that it will always be motivated to move forward. Another case that indicates a successful action is that the center value indicates water lilies or logs, which means the agent learned to step on those and avoid water, it will be given a reward of 2 as it values more than simply moving forward on an empty space. Lastly, the agent will get a reward of 10 if the center value is a coin. If the new center is now an obstacle or a car, the action will be considered as failed, and the difference is that steps on obstacle will revert the state to its original state (considered as no movement) with the game keeps going, with a reward of -2, and steps on a car will terminate the game with reward -10. In cases that the action failed but the game is not terminating, the entire representation will have the last row replaced by a

row of an obstacle as the map is constantly moving forward. Lastly, a special case is that if the row immediately behind the agent is all obstacle, that indicates that the agent is all the way at the bottom of the screen. If the action at that time is backward or NOP, the game will terminate, as the game was designed so. Overall this is a simplified version of the rules that are in the game, so it's not perfect, however, this helps a lot with training time.

Lastly, with all the pieces ready, I implemented a gradient-based Q learning algorithm with PyTorch, using softmax as the exploring factor, where the state-action value function is approximated with a CNN network followed by fully connected layers. The reason why I didn't have a tabular Q Learning step up is that the table storing the q values will simply way too big, as there are 24^6 possible states (rows) and 5 actions (columns), which gives us a matrix with a billion entries. Due to the nature of Q-Learning being an off-policy algorithm, at each time step, the action will be picked randomly from the probability being the softmax of all actions' q values. The picked action along with the state will be fed into the step function described in the last paragraph to retrieve the status (if the game has terminated, the next state, and the reward). The q value corresponding to the state and action will be updated following this rule below (α is the momentum and γ is the discount factor):

$$q(s,a) = (1-\alpha)\cdot q(s,a) + \alpha \cdot (r(s,a) + \gamma \cdot \max(q(s',\cdot))$$

3.5. Model for Identifying Game Status

This is a relatively simple task as the three statuses (before game, in game, and end game) look quite different, and we have only three classes to deal with. Here, a neural network consisting of 4 CNN layers, connected by activation, max-pooling, and dropouts, and ends with 2 Fully-Connected layers is used to fit the screenshots of a random mix of different statuses. The output would simply be either 0, 1, or 2 suggesting the status of the game that the character is in.

4. Experiment

Since not much study has been done on this specific game, all the data that will be used in training will be collected by the author that comes from the run time screenshots of the game, with labels marked by the author depending on usage. More specifically, we will have three data sets for the three models we introduced before. They will be discussed in detail in the sub-sections below.

4.1. YOLO for Object Detection

For the object detection model, x is the screenshots from gameplay, and y is the coordinates of objects and their corresponding label. There are in total 197 images collected for Object detection. To label the images for Object detection, I used an online tool called 'makesense.ai' and boxed all the objects in the 197 images, and matched the box with a label.

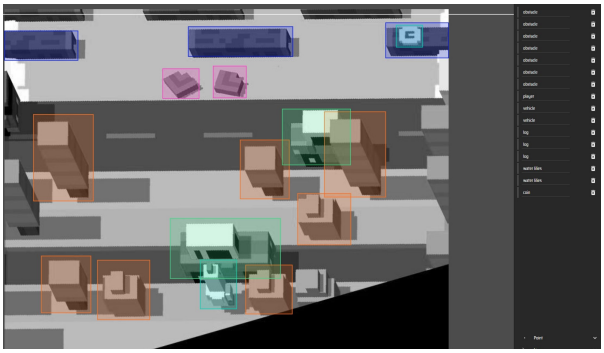

Figure 12. Screenshot of the control panel of makesense.ai for Image Labeling

With the GPU I have in hand, the largest model I can train is based on the yolov5l, which has 46.5 million parameters, almost 7 times the amount that is in yolov5s. I tried to train my custom model from yolov5s, yolov5m, and yolov5l, up to now the yolov5l yields the best result with no sign of overfitting, and the output on run time seems decent to me.

Table 1. yolov5l.pt epoch = 80

Class	mAP@.5	mAP@.5:.95:100%
all	0.981	0.788
agent	0.995	0.821
obstacle	0.954	0.721
vehicle	0.952	0.805
water lilies	0.995	0.738
coin	0.995	0.792
log	0.995	0.849

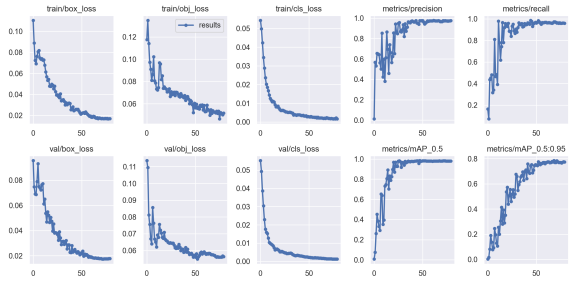

Figure 13. Error curve of YOLO Training

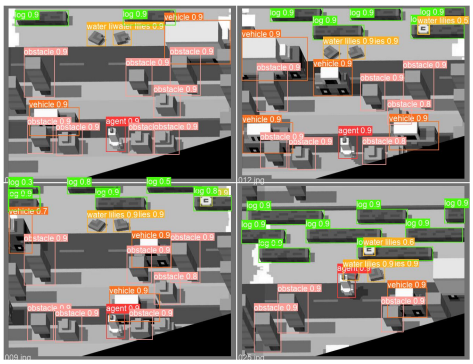

Figure 14. Predicted label on validation batch 0

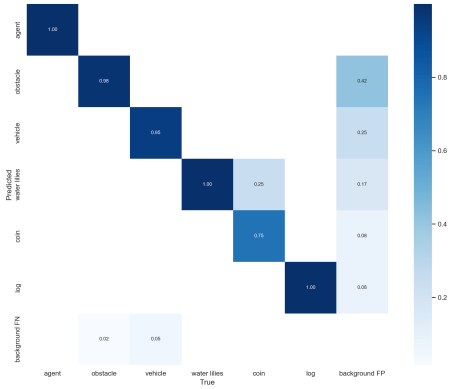

Figure 15. Confusion Matrix with all 6 classes

4.2. CNN for identifying Game Status

I used TensorFlow to construct my CNN model with structure shown below.

```
Model: "sequential"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d (Conv2D)              (None, 32, 32, 16)        448
_________________________________________________________________
leaky_re_lu (LeakyReLU)      (None, 32, 32, 16)        0
_________________________________________________________________
conv2d_1 (Conv2D)            (None, 32, 32, 32)        4640
_________________________________________________________________
leaky_re_lu_1 (LeakyReLU)    (None, 32, 32, 32)        0
_________________________________________________________________
max_pooling2d (MaxPooling2D) (None, 16, 16, 32)        0
_________________________________________________________________
dropout (Dropout)            (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_2 (Conv2D)            (None, 16, 16, 32)        9248
_________________________________________________________________
leaky_re_lu_2 (LeakyReLU)    (None, 16, 16, 32)        0
_________________________________________________________________
conv2d_3 (Conv2D)            (None, 16, 16, 64)        18496
_________________________________________________________________
leaky_re_lu_3 (LeakyReLU)    (None, 16, 16, 64)        0
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 8, 8, 64)          0
_________________________________________________________________
dropout_1 (Dropout)          (None, 8, 8, 64)          0
_________________________________________________________________
flatten (Flatten)            (None, 4096)              0
_________________________________________________________________
dense (Dense)                (None, 256)               1048832
_________________________________________________________________
dropout_2 (Dropout)          (None, 256)               0
_________________________________________________________________
dense_1 (Dense)              (None, 3)                 771
=================================================================
Total params: 1,082,435
Trainable params: 1,082,435
Non-trainable params: 0
_________________________________________________________________
```

Figure 16. Screenshot of the model structure

For the model that determines the status of the game, x contains screenshots again and y is the corresponding status (before game, in game, end game will be encoded by 0, 1, 2). With a 8:2 split, 476 images were assigned to training and 120 for testing. With learning rate = 0.01 and epochs = 5, the training accuracy is 98.11% and the test accuracy is 100%.

4.3. Prep for RL

For the reinforcement model, the data set was collected again through game recordings and there were 1141 out of 1374 images that were able to go through the transformation function, and we obtained a matrix of size 1141 by 5 by 5 as x. Here are a few examples of the data:

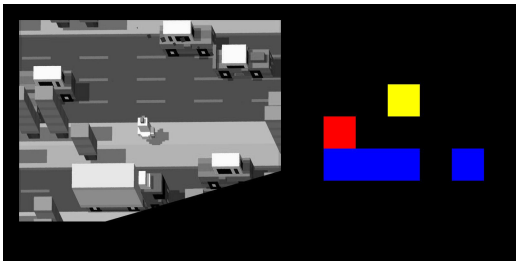

Figure 17. Example 1

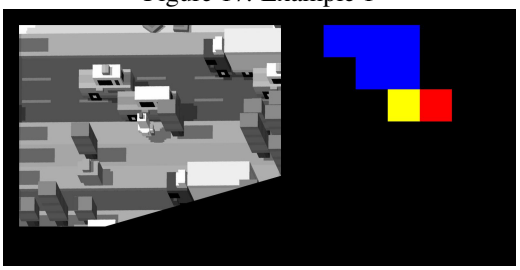

Figure 18. Example 2

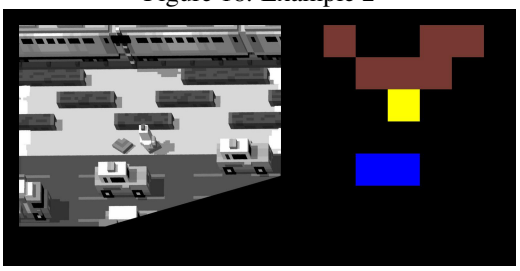

Figure 19. Example 3

Note that the examples above are just the visual representation and the actual data are all consist of just 5 by 5 per image. And we are ready for the next stage.

4.4. Supervised learning attempt

Before attempting the reinforcement learning approach, I decided to use a supervised approach as the control group. So I hand-labeled all 1141 images from the last step as if I were playing the game. For example, the label of the 3 examples on top will be $[0, 4, 0]$ indicating I think the agent

at those situations should go $[Forward, NOP, Forward]$. Once I have the labels for all the images, they were encoded into one-hot vectors and that gives me a matrix of 1141 by 5 as the y for my supervised learning model. In the end, the output of the model basically will give me a probability distribution of which action should be taken. Therefore, I used cross-entropy loss for training which automatically applies a softmax to the output and find the difference between my prediction and the one-hot encoded label, and the default Adam optimizer that was built in PyTorch was used for the optimization. The model structure is shown below:

```
Sequential(
  (0): Conv2d(1, 5, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): LeakyReLU(negative_slope=0.1)
  (2): Conv2d(5, 10, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (4): Dropout(p=0.25, inplace=False)
  (5): Flatten()
  (6): Linear(in_features=90, out_features=45, bias=True)
  (7): Dropout(p=0.25, inplace=False)
  (8): LeakyReLU(negative_slope=0.1)
  (9): Linear(in_features=45, out_features=5, bias=True)
)
```

Figure 20. Supervised Learning Model

Again, I'm using a 4:1 split for the train and test set, so each will contain 913 and 228 images. Through 15 epochs, the model was able to improve the accuracy from below 50% to over 80% for the training set and around 70% for the test set.

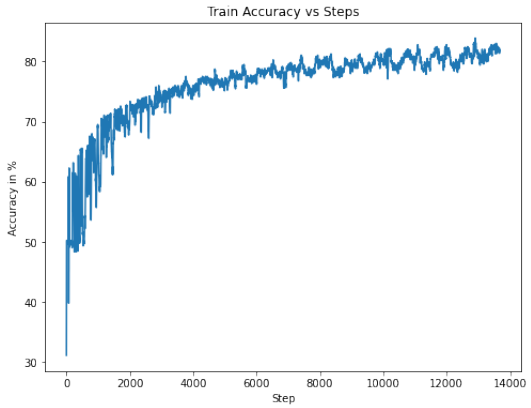

Figure 21. Accuracy Curve for training set

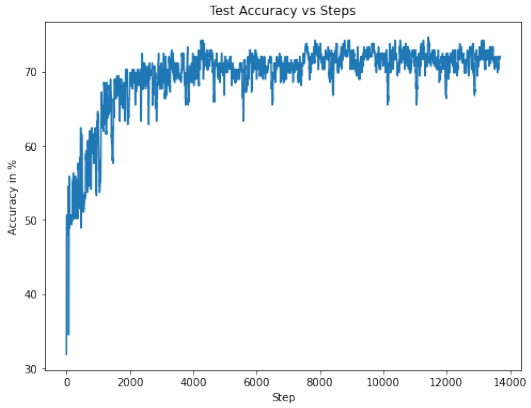

Figure 22. Accuracy Curve for test set

After Epoch 15, despite the accuracy for the training set is still increasing, the accuracy for the test set stopped improve and the loss for the test set tends to increase, which is a clear sign of over-fitting. This limits the training to stop at epoch 15.

With the training done, the model was saved and then loaded into a separate driver code that passes actual ongoing game images into the model and has the model decide which step to take next. I was surprised to see that the model was able to learn when to wait and go, circling around obstacles, with only 1000 plus images' training. More analysis of this model will be continued as a new form of evaluation metrics need to be introduced.

4.5. Gradient based Q learning

Unlike labeling what to do for each state, all the model here has is the heuristic function that takes in state and action, and outputs status (if the action has caused the game to terminate), reward, and the next state. The algorithm basically follows the generic tabular Q-Learning with one modification: instead of doing a table lookup for each q value, the network model will predict the q values. Here below is the network model structure:

```
Sequential(
  (0): Conv2d(1, 5, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
  (1): LeakyReLU(negative_slope=0.1)
  (2): Conv2d(5, 10, kernel_size=(2, 2), stride=(1, 1), padding=(1, 1))
  (3): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  (4): Dropout(p=0.25, inplace=False)
  (5): Flatten()
  (6): Linear(in_features=90, out_features=30, bias=True)
  (7): Dropout(p=0.25, inplace=False)
  (8): LeakyReLU(negative_slope=0.1)
  (9): Linear(in_features=30, out_features=5, bias=True)
)
```

Figure 23. State-action value approximation model

More details about the hyperparameters and the training setting: although for this part, the model outputs a 1 by 5 vector as output as well (same as the previous part), the meaning of this vector is not the same. For the previous part, it's a probability distribution, and for this part, it's the estimated state-action value (suggesting the estimated return for this state and action). Therefore, the loss function this time is not cross-entropy but mean squared error, and there's no softmax involved when calculating the loss. Specifically for the Q Learning algorithm, the γ is set to 0.8, α is set to 0.05, the timeout was set to 100 steps per episode.

When it comes to the evaluation, I decided to look at two import values: 1) the total steps per episode, which indicates how long the agent can survive in the game. Since the camera angle is constantly moving forward and the agent will die if it decides to stay on the same lane. Therefore, the longer the agent survives, the further it will need to move forward in order to stay alive, which is our main objective. 2) The reward, the indicator of how good the actions that have been taken with those many steps within each episode. Moving left or right for 90 percent of the time will not help

the agent go a long way ahead when compared to moving forward 90 percent of the time. Here below is a table showing the comparison of the performance of a few variances of the model (including the supervised model that I have trained from the previous part.)

Table 2. Performance with step limit = 100 and 3000 attempts

Model	Avg Steps	Avg Reward
Supervised, 80% data	61	28.915
Supervised, all data	73	30.132
Q learning, alpha = 0.1	72	55.394
Q learning, alpha = 0.05	83	74.851

5. Conclusion

Overall, I think this is a successful experiment with both object detection and reinforcement learning and it's been a great hands-on experience. Without any instructions on when it's safe to move forward, the agent was able to learn how to play the game. More importantly, in terms of the reward, the policy found by the reinforcement learning model was able to perform significantly better than the supervised model as it has the knowledge to predict the consequences of the future steps.

With that said, there are still a lot of places that can be improved. First of all, although the reinforcement learning model managed to do really well in the algorithm itself when it comes to estimation of how good the policy is. Often times when I'm testing in the actual game, due to the loss in precision during the object detection and the compression from raw image data to the 5 by 5 state representation, error in the pre-processing stage leads to big problems later. For example, when there are water lilies to the front left of the agent but it was not recognized, the agent will think it's the safe lane in front and just go straight, which causes the agent to get killed in the game. Another potential improvement is that currently, only one image was used for the agent so there's no way for the agent to predict the direction of the movement of the vehicles. Have the model was designed to take in three images, maybe the direction of the movement can be figured out and the agent can take advantage of that additional information as well during the decision-making process.

References

[1] Harrison Kinsley. Training self-driving car neural network-python plays gta v. 1

[2] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning, 2013. 2

[3] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 02 2015. 2