ROBOTIC LEARNING AND VISION FOR NAVIGATION REPORT

Kathakoli Sengupta(U17696524)

1.1. Data Collection:

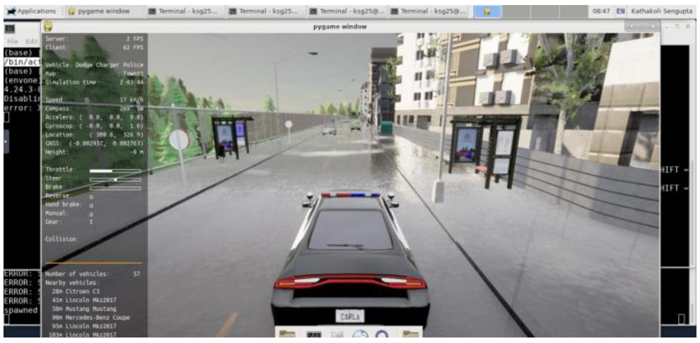


Figure 1

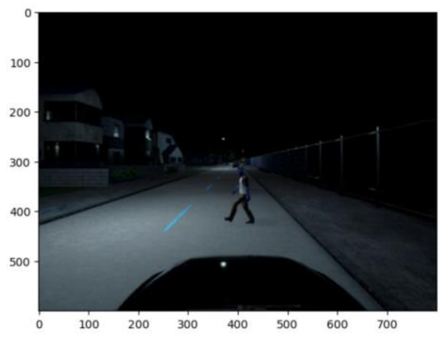
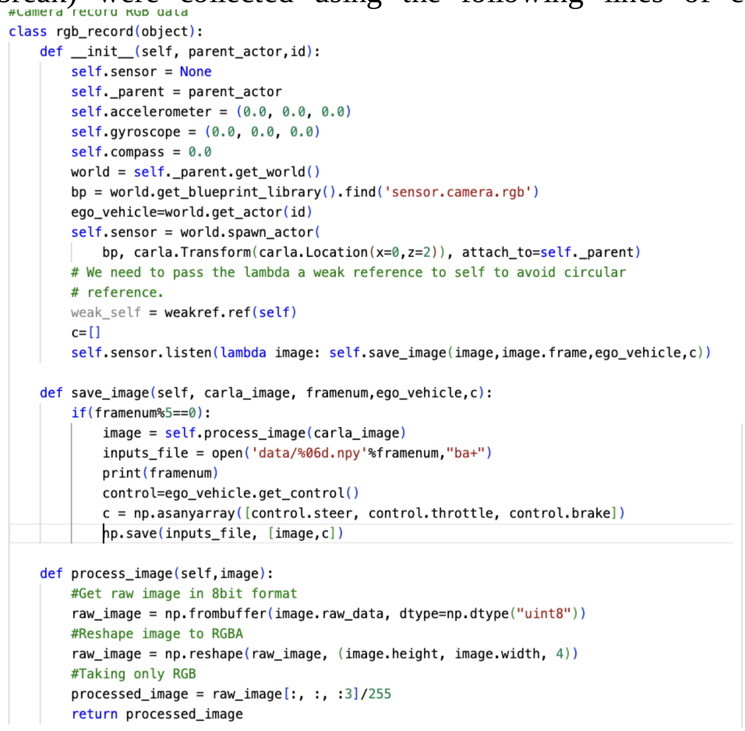


Figure2

a. As a first step, Town 01 was configured and dynamic weather was set.

b. 50 vehicles and 50 passengers were spawned as traffic in the CARLA environment out of which only 36 passengers could be spawned due to collision in the spawn environment. Fig. 1 shows the environment formed.

c. An extra RGB camera was initialized at point(0,2), images and ego vehicle's controls(steer, throttle and break) were collected using the following lines of code in **Manual_Control_datasetcollection.py.**



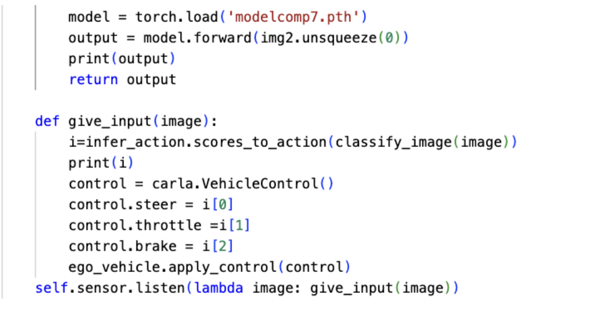
```
#camera record RGB data
class rgb_record(object):
    def __init__(self, parent_actor,id):
        self.sensor = None
        self._parent = parent_actor
        self.accelerometer = (0.0, 0.0, 0.0)
        self.gyroscope = (0.0, 0.0, 0.0)
        self.compass = 0.0
        world = self._parent.get_world()
        bp = world.get_blueprint_library().find('sensor.camera.rgb')
        ego_vehicle=world.get_actor(id)
        self.sensor = world.spawn_actor(
            bp, carla.Transform(carla.Location(x=0,z=2)), attach_to=self._parent)
        # We need to pass the lambda a weak reference to self to avoid circular
        # reference.
        weak_self = weakref.ref(self)
        c=[]
        self.sensor.listen(lambda image: self.save_image(image,image.frame,ego_vehicle,c))

    def save_image(self, carla_image, framenum,ego_vehicle,c):
        if(framenum%5==0):
            image = self.process_image(carla_image)
            inputs_file = open('data/%06d.npy'%framenum,"ba+")
            print(framenum)
            control=ego_vehicle.get_control()
            c = np.asanyarray([control.steer, control.throttle, control.brake])
            np.save(inputs_file, [image,c])

    def process_image(self,image):
        #Get raw image in 8bit format
        raw_image = np.frombuffer(image.raw_data, dtype=np.dtype("uint8"))
        #Reshape image to RGBA
        raw_image = np.reshape(raw_image, (image.height, image.width, 4))
        #Taking only RGB
        processed_image = raw_image[:, :, :3]/255
        return processed_image
```

Figure 3

A world was created and the ego vehicle is spawned in it. The images are reshaped and only RGB data is extracted as a pre-processing step. Every 5 frames are skipped to avoid redundant data. The ego vehicle ID was extracted using get_actor() function and is used to get the controls (steer, throttle and break). This controls list is concatenated with the image array and stored as .npy files as shown in Fig. 3.



```
        model = torch.load('modelcomp7.pth')
        output = model.forward(img2.unsqueeze(0))
        print(output)
        return output

    def give_input(image):
        i=infer_action.scores_to_action(classify_image(image))
        print(i)
        control = carla.VehicleControl()
        control.steer = i[0]
        control.throttle =i[1]
        control.brake = i[2]
        ego_vehicle.apply_control(control)
    self.sensor.listen(lambda image: give_input(image))
```

Figure 4

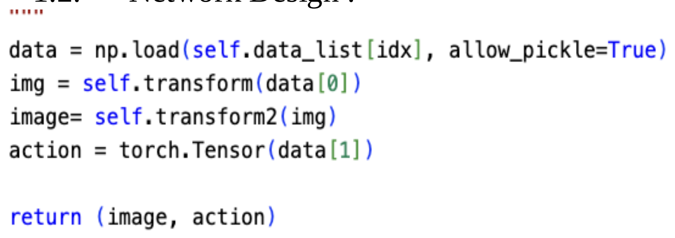
A custom script was created called **Manual_control_trial.py** as shown in Fig.4 that could load the model in real-time and run its predicted controls in CARLA. This was used to test all model performance after training

d. The controls are stored using **Manual_Control_datasetcollection.py** as shown above.

e. The steps a to d are repeated multiple times and the images are plotted in jupyter notebook as shown in Fig. 2 to check the data quality and missing values.

f. **Good Training Data:** A good training data is an unbiased data. E.g. It is possible that while collecting the data the car takes left turns quite often compared to right then it will lack instances of right turns and will be biased to take left. For the current experiment two datasets were collected , one small dataset of 6ooo datapoints to compare all the model performances and one large dataset of 50000 datapoints for the competition. The smaller dataset has a lot of red light signal images so it had learned to stop for all red lights however it does not perform so well for turns whereas the competition dataset does take proper turns but sometimes fails to stop for traffic lights. **Perfect Imitation:** In case of only perfect imitations, the agent will not learn to rectify errors in case of any wrong move, hence the agent will be overfit on a small data but it won't be able to generalise in an unknown situation.

1.2. Network Design :

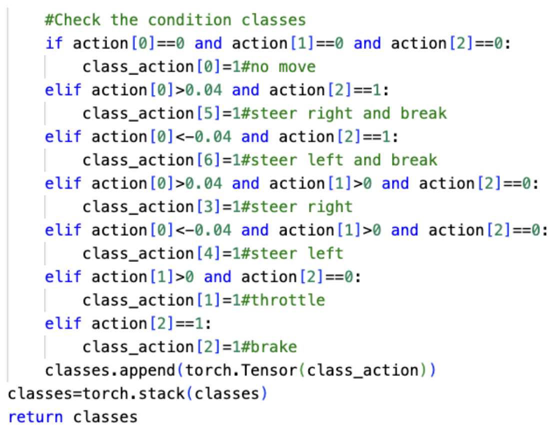


```
"""
data = np.load(self.data_list[idx], allow_pickle=True)
img = self.transform(data[0])
image= self.transform2(img)
action = torch.Tensor(data[1])

return (image, action)
```

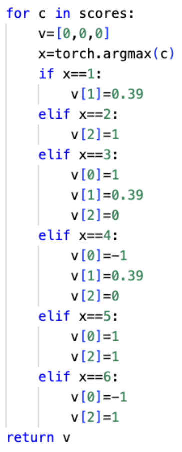
Figure 5

a. A dataloader was defined in **Dataset.py** with the lines of code given in Fig. 5 transform is for converting image numpy array to tensor and transform2 is for resizing the tensor to 96*128 dimension from 600*800. 96*128 was chosen because that would take a broader horizontal length and would consider traffic lights etc.



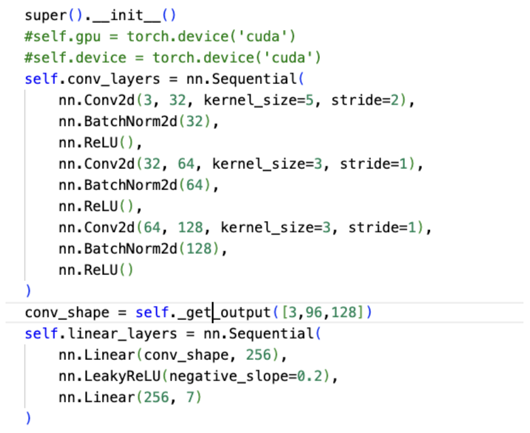
```
    #Check the condition classes
    if action[0]==0 and action[1]==0 and action[2]==0:
        class_action[0]=1#no move
    elif action[0]>0.04 and action[2]==1:
        class_action[5]=1#steer right and break
    elif action[0]<-0.04 and action[2]==1:
        class_action[6]=1#steer left and break
    elif action[0]>0.04 and action[1]>0 and action[2]==0:
        class_action[3]=1#steer right
    elif action[0]<-0.04 and action[1]>0 and action[2]==0:
        class_action[4]=1#steer left
    elif action[1]>0 and action[2]==0:
        class_action[1]=1#throttle
    elif action[2]==1:
        class_action[2]=1#brake
    classes.append(torch.Tensor(class_action))
classes=torch.stack(classes)
return classes
```

Figure 6



```
for c in scores:
    v=[0,0,0]
    x=torch.argmax(c)
    if x==1:
        v[1]=0.39
    elif x==2:
        v[2]=1
    elif x==3:
        v[0]=1
        v[1]=0.39
        v[2]=0
    elif x==4:
        v[0]=-1
        v[1]=0.39
        v[2]=0
    elif x==5:
        v[0]=1
        v[2]=1
    elif x==6:
        v[0]=-1
        v[2]=1
return v
```

Figure 7



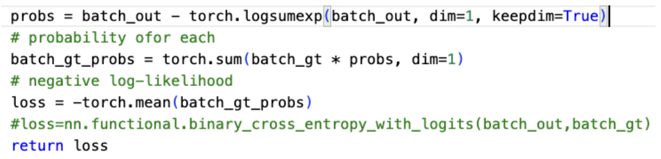
```
super().__init__()
#self.gpu = torch.device('cuda')
#self.device = torch.device('cuda')
self.conv_layers = nn.Sequential(
    nn.Conv2d(3, 32, kernel_size=5, stride=2),
    nn.BatchNorm2d(32),
    nn.ReLU(),
    nn.Conv2d(32, 64, kernel_size=3, stride=1),
    nn.BatchNorm2d(64),
    nn.ReLU(),
    nn.Conv2d(64, 128, kernel_size=3, stride=1),
    nn.BatchNorm2d(128),
    nn.ReLU()
)
conv_shape = self._get_output([3,96,128])
self.linear_layers = nn.Sequential(
    nn.Linear(conv_shape, 256),
    nn.LeakyReLU(negative_slope=0.2),
    nn.Linear(256, 7)
)
```

Figure 8

b. The classes_to_action function in defined in Fig. 6 where 7 classes are defined according to 7 possible conditions. Fig. 7 shows scores_to_action function definition where steer, throttle and brake control parameter values are determined from the class predicted where maximum throttle value given is 0.39 according to CARLA observations. The steer and throttle are thresholded at 0.04 and 0.39 values according to observation.
c. A 3 convolution-2 linear layer model was defined as shown in Fig. 8.
d. **RGB and Grayscale:** The images were converted to grayscale and training was carried out. Grayscale training was much faster than RGB since 3 channels were not processed. Though there was not much differences in performance RGB model performed slightly better.
Hyperparameter Tuning: A couple of other networks like 5 convolution layer with 3*32,32*32,32*64,64*64,64*128 dimensions and different kernel sizes like all kernels of size 5 were tried and tested but this model gave the best output possible among all. Also learning rate tuning and optimizer tuning, different batch sizes and epoch numbers was tried as discussed in 1.3f. Yes, hyperparameter tuning does give better results since it changes the type of features(low-level, high-level), number of features, number of steps during gradient descent etc which manipulates the learning procedure and proper hyperparameter choice is very important to train a good model.
e. **Importance of dividing data into batches:** Given the limited amount of GPU memory and the huge amount of image data it is impossible to load entire few tens of GBs of data for training at once. Hence, dividing into batches lets us load a part of the data train on that and then load next part. The batch size is chosen based on amount of GPU computation available. **Epoch:** An epoch is the total number of iterations required to train one batch of data over all the samples in the batch. It is one cycle of training for the particular batch.

f. The log-softmax was applied on the output before implementing cross-entropy as given in part c. Probabilities multiplied by predicted outputs are calculated and their mean was calculated to find the loss.

g. The model was trained and the loss plot is shown in Fig. 10.



```
probs = batch_out - torch.logsumexp(batch_out, dim=1, keepdim=True)
# probability ofor each
batch_gt_probs = torch.sum(batch_gt * probs, dim=1)
# negative log-likelihood
loss = -torch.mean(batch_gt_probs)
#loss=nn.functional.binary_cross_entropy_with_logits(batch_out,batch_gt)
return loss
```

Figure 9

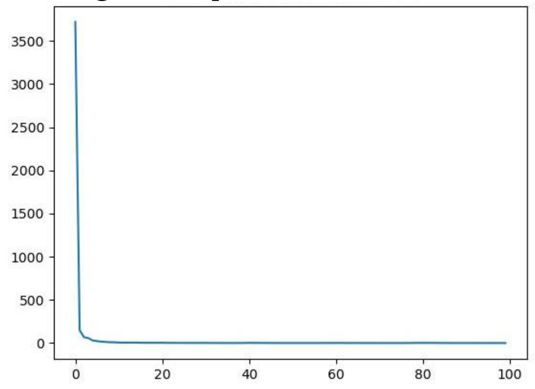


Figure 10

1.3. Network Improvements:

a. **Sensors :** The sensors included in manual_control.py are Lane Invasion Sensor, Gnss Sensor, IMU Sensor, Radar Sensor , Lidar, Dynamic Vision Sensor, Collision Sensor. The sensors except LIDAR and DVS(they are computationally very heavy), location, accelerometer, velocity data was collected by passing the parameters defined in HUD inside manual_control.py using the code.

```
self.camera_sensor=rgb_record(self.player,self.player.id,self.gnss_sensor,self.imu_sensor,self.lane_invasion_sensor,self.collision_sensor)
```

The IMU sensor and radar values helped the network to give better performance whereas the other sensors didn't have any impact.

b.

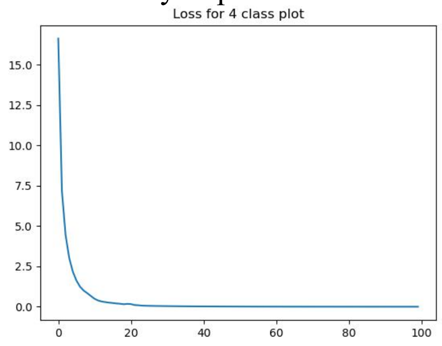


Figure11

Multiclass Classification: Four classes are defined for keyboard actions and the classification network was modified to 4 neuron heads to give 4 class classification. A sigmoid activation function is inserted after the last linear layer to convert the output to 0 and 1. BCE loss is used as loss function since now we need every neuron head to give 0 and 1 instead of one-hot vector. The loss plot is given in Figure 1 for the multiclass classification. The performance got better when multiclass classification was implemented. Learning rate of 10^-5 was used in this case.

c.

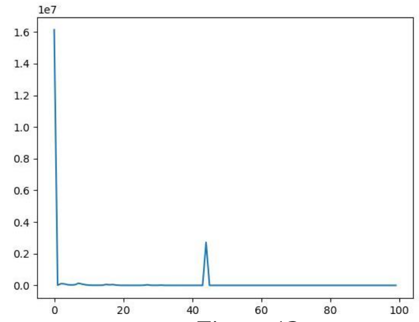


Figure 12

Regression:

Mean-Squared error loss function was used in this case where we find the square-root of mean of square of difference between each predicted output and ground truth.

Advantages/Drawbacks: Regression is more useful when there is a continuous output value. It maps particular input values to particular output values which can be a drawback for some problems.

In this case, classification worked far better than regression, probably because regression was doing input to output mapping and could not generalise control output controls in case of unknown trajectories.

d.

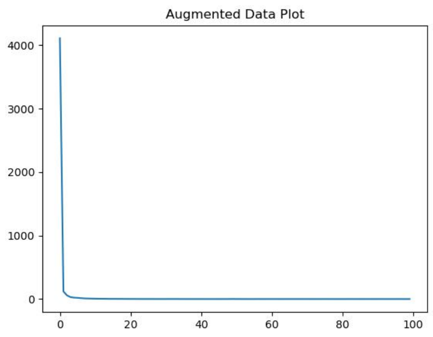


Figure 13

Data Augmentation: Two types of data augmentation are used

1. In the first case brightness, contrast, saturation etc. of the image are modified.
2. In the second case noise is introduced in the image to make it more robust to any kind of noise in the surrounding.

Hence, two complete new sets of data are concatenated with the previous dataset. This data augmentation improved the performance of the network.

```
self.transform3= transforms.Compose([transforms.ColorJitter(brightness=0.5, contrast=0.5, saturation=0.5, hue=0.1)])
self.transform4 = transforms.Compose([transforms.GaussianBlur(kernel_size=3, sigma=(0.1, 2.0))])
```

e.

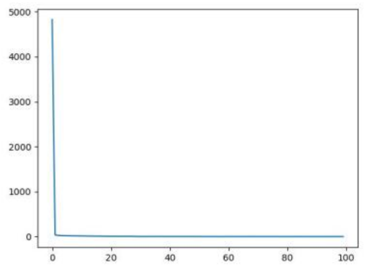
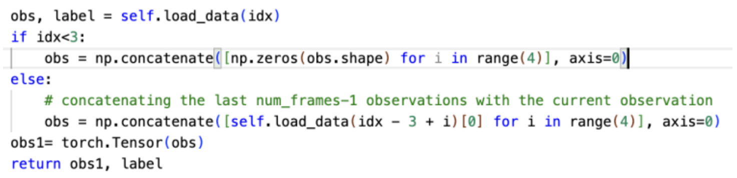


Figure 14

Historical Observations:



```
obs, label = self.load_data(idx)
if idx<3:
    obs = np.concatenate([np.zeros(obs.shape) for i in range(4)], axis=0)
else:
    # concatenating the last num_frames-1 observations with the current observation
    obs = np.concatenate([self.load_data(idx - 3 + i)[0] for i in range(4)], axis=0)
obs1= torch.Tensor(obs)
return obs1, label
```

The following lines of code are added to insert 4 previous frames in every image data forming a 12*96*128 pixel image.

This process improved the performance of the classifier since the classifier is now trained on 4 times more data than others.

f. Hyperparameter Tuning:

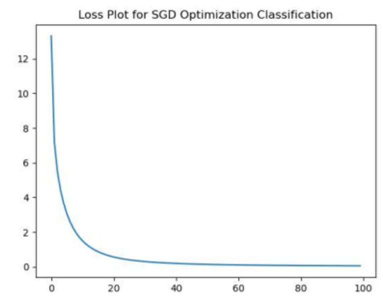


Figure 15

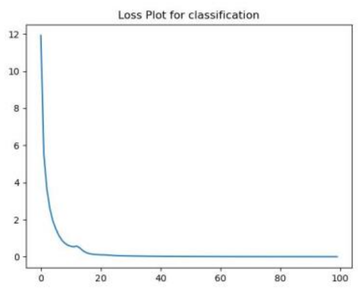


Figure 16

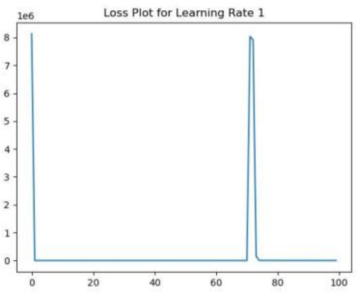


Figure 17

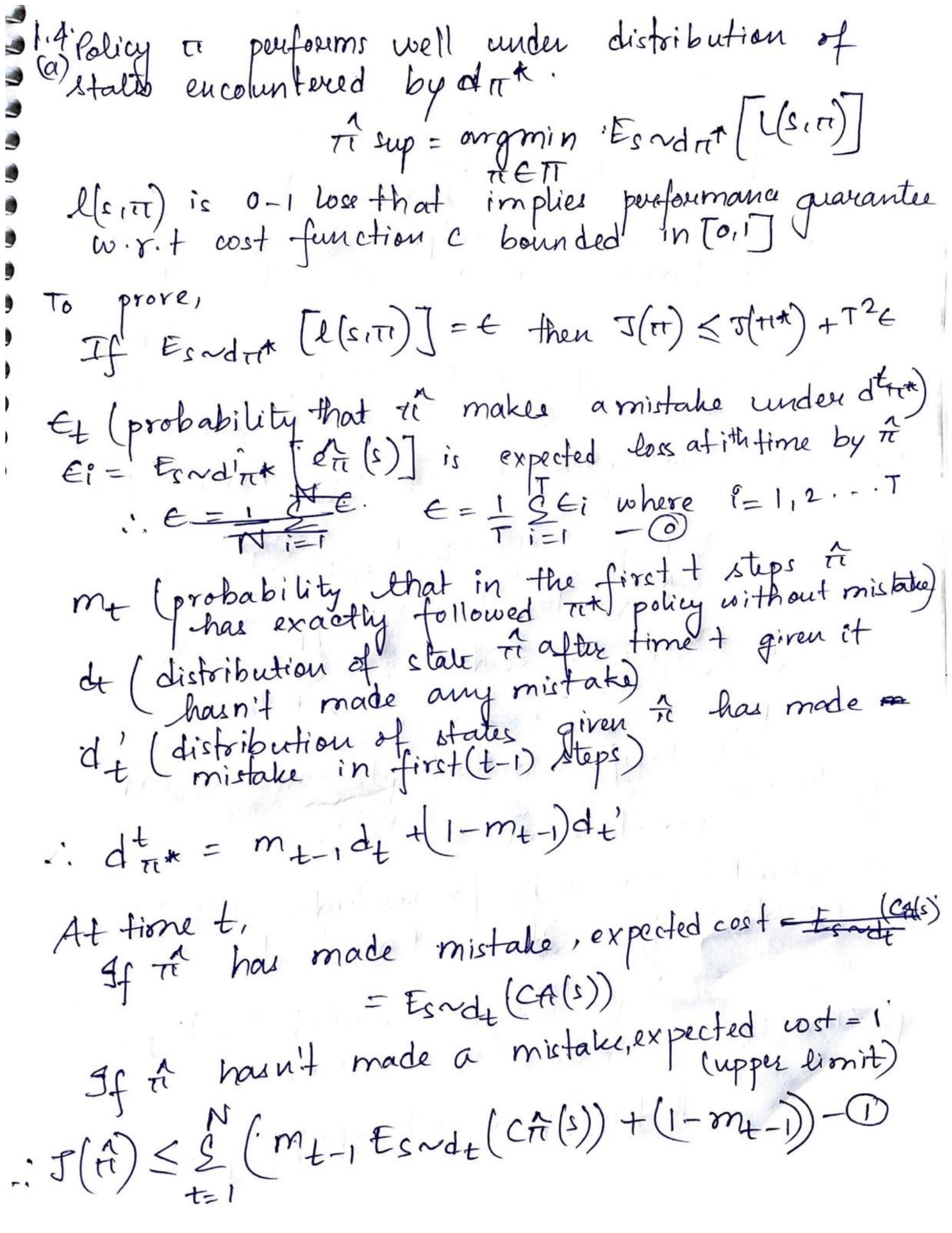
Two types of hyperparameter tuning were tried

1. In Figure 15, SGD Optimizer is used instead of Adam optimizer.
2. Figure 16 and 17 corresponds to loss curves for learning rate 1 and 10^-5 instead of 10^-2.

In all the cases performance reduced. The best performance among this was observed by using Adam Optimizer and 10^-2 learning rate.

1.4. Dagger:

a.



1.4 (a) Policy π performs well under distribution of states encountered by d_{π^*}.

$$\hat{\pi}_{sup} = \underset{\pi \in \Pi}{\arg\min}\ E_{s \sim d_{\pi^*}}\left[l(s,\pi)\right]$$

$l(s,\pi)$ is 0-1 loss that implies performance guarantee w.r.t cost function C bounded in $[0,1]$ ✓

To prove,

If $E_{s\sim d_{\pi^*}}\left[l(s,\pi)\right] = \epsilon$ then $J(\pi) \leq J(\pi^*) + T^2\epsilon$

ϵ_t (probability that $\hat{\pi}$ makes a mistake under $d^t_{\pi^*}$)

$\epsilon_i = E_{s\sim d^i_{\pi^*}}\left[e_{\hat{\pi}}(s)\right]$ is expected loss at i^{th} time by $\hat{\pi}$

~~$\therefore \epsilon = \frac{1}{N}\sum_{i=1}^{N}\epsilon$.~~ $\epsilon = \frac{1}{T}\sum_{i=1}^{T}\epsilon_i$ where $i = 1, 2 \dots T$ $-\textcircled{0}$

m_t (probability that in the first t steps $\hat{\pi}$ has exactly followed π^* policy without mistake)

d_t (distribution of state $\hat{\pi}$ after time t given it hasn't made any mistake)

d'_t (distribution of states given $\hat{\pi}$ has made ~~m~~ mistake in first $(t-1)$ steps)

$$\therefore d^t_{\pi^*} = m_{t-1}d_t + (1-m_{t-1})d_t'$$

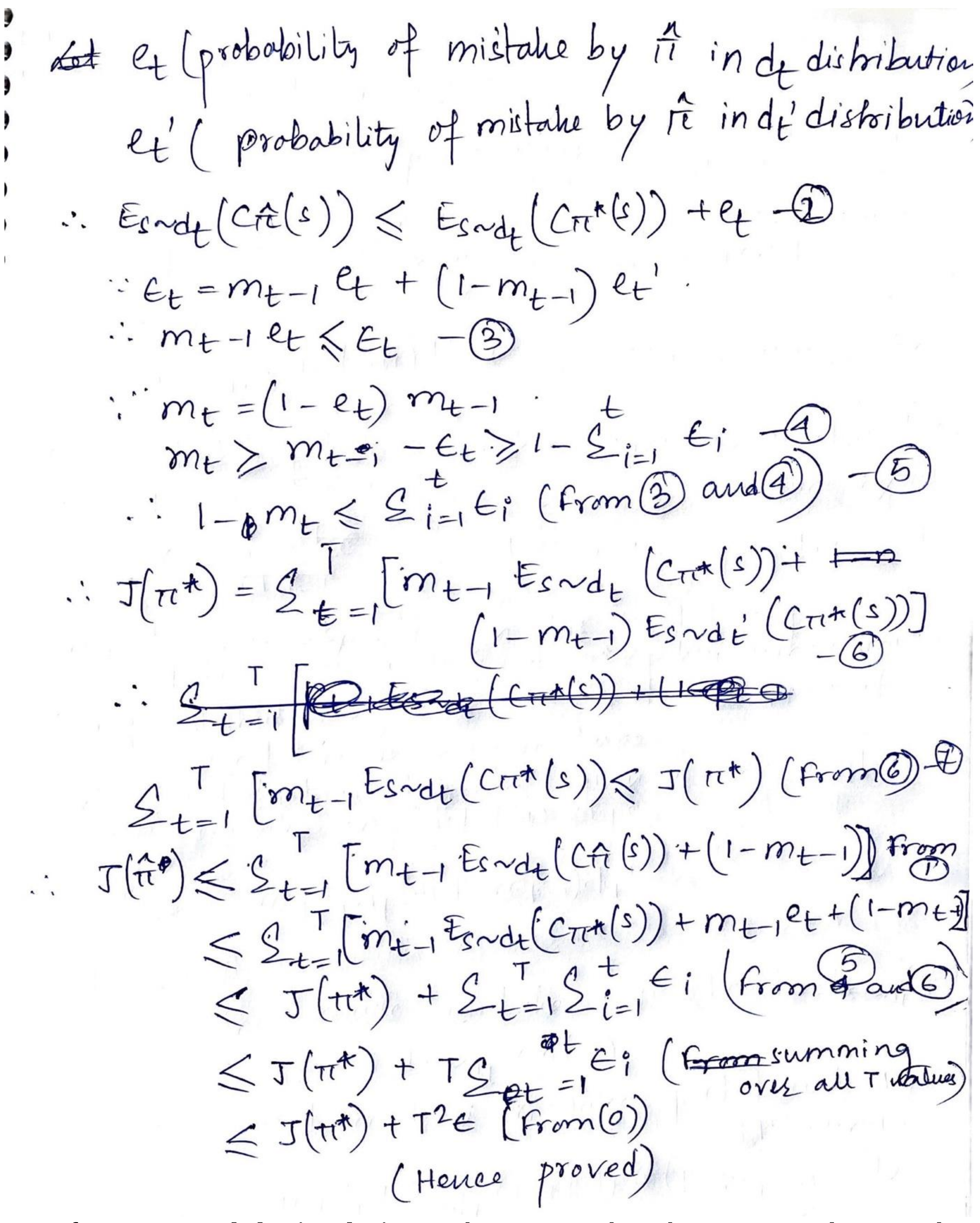
At time t,

If $\hat{\pi}$ has made mistake, expected cost ~~$= E_{s\sim d_t}$~~ $(C_{\hat{\pi}}(s))$

$= E_{s\sim d_t}(C_{\hat{\pi}}(s))$

If $\hat{\pi}$ hasn't made a mistake, expected cost $= 1$ (upper limit)

$$\therefore J(\hat{\pi}) \leq \sum_{t=1}^{N}\left(m_{t-1}E_{s\sim d_t}(C_{\hat{\pi}}(s)) + (1-m_{t-1})\right) -\textcircled{1}$$



~~Let~~ e_t (probability of mistake by $\hat{\pi}$ in d_t distribution

e_t' (probability of mistake by $\hat{\pi}$ in d_t' distribution

$\therefore E_{s\sim d_t}(C_{\hat{\pi}}(s)) \leq E_{s\sim d_t}(C_{\pi^*}(s)) + e_t$ — $\textcircled{2}$

$\because \epsilon_t = m_{t-1} e_t + (1-m_{t-1}) e_t'$.

$\therefore m_{t-1} e_t \leqslant \epsilon_t$ — $\textcircled{3}$

$\because m_t = (1-e_t)\, m_{t-1}$

$m_t \geq m_{t-1} - \epsilon_t \geqslant 1 - \sum_{i=1}^{t} \epsilon_i$ — $\textcircled{4}$

$\therefore 1 - m_t \leqslant \sum_{i=1}^{t} \epsilon_i$ (from $\textcircled{3}$ and $\textcircled{4}$) — $\textcircled{5}$

$\therefore J(\pi^*) = \sum_{t=1}^{T} [m_{t-1} E_{s\sim d_t}(C_{\pi^*}(s)) +$ ~~[illegible]~~

$(1-m_{t-1}) E_{s\sim d_t'}(C_{\pi^*}(s))]$ — $\textcircled{6}$

$\therefore \sum_{t=1}^{T} [$ ~~[illegible] $E_{s\sim d_t}(C_{\pi^*}(s)) + (1$ [illegible]~~

$\sum_{t=1}^{T} [m_{t-1} E_{s\sim d_t}(C_{\pi^*}(s)) \leqslant J(\pi^*)$ (from $\textcircled{6}$) — $\textcircled{7}$

$\therefore J(\hat{\pi}) \leqslant \sum_{t=1}^{T} [m_{t-1} E_{s\sim d_t}(C_{\hat{\pi}}(s)) + (1-m_{t-1})]$ from $\textcircled{1}$

$\leq \sum_{t=1}^{T} [m_{t-1} E_{s\sim d_t}(C_{\pi^*}(s)) + m_{t-1} e_t + (1-m_{t-1})]$

$\leqslant J(\pi^*) + \sum_{t=1}^{T} \sum_{i=1}^{t} \epsilon_i$ (from ~~4~~ $\textcircled{5}$ and $\textcircled{6}$)

$\leqslant J(\pi^*) + T \sum_{t=1}^{t} \epsilon_i$ (~~from~~ summing over all T values)

$\leqslant J(\pi^*) + T^2 \epsilon$ (From (0))

(Hence proved)

Dagger performs same as behavior cloning: In the situation where there is no new data to explore and dagger is provided with the same environment as supervised learning, dagger performs same as behavior cloning. Also if the supervised learning and agent policy are the same with similar dataset for both much improvement will not be observed in case of dagger.

Quadratic cost in Urban Driving: In urban driving situations where there is a huge traffic jam there are more chances of accidents so probability of life or belonging loss can increase quadratically experiencing a quadratic loss.

b. DAGGER will perform better since at every step the expert policy is correcting all the unseen situations by the agent hence it is continuously learning new policies.

   Situations where dagger performs better: Dagger can perform better where there can be infinite unseen cases than that is present in the supervised learning dataset like self-driving cars can encounter new roads whose data are not registered. Language models can be asked new questions that it hasn't encountered yet. In these cases Dagger can help it learn from expert policy and improve.

Appendix

Zip Content

1. Data Folder : Manual_control_datasetcollection.py(general data collection)
   Manual_control_datasetcollection_sensor.py(sensor data collection)
   Manual_control_trial.py(real-time execution of model in CARLA)
2. Classification : modelclasssmalldataset.pth(Model trained with small dataset for comparison)'
   Modelcomp1.pth(Competition Model1)
   Modelcomp2.pth(CompetitionModel2)
   Dataset.py(Dataloader)
   Network.py
   Training.py
3. Regression : Modelregnew.pth(regression model)
   Trainingreg.py
   Networkreg.py
4. Multiclass: model4class.pth
   Network4class.py
   Training4class.py
5. Data Augmentation: modelaugmentdata.pth
   augmenteddata.py(Dataloader with augmentation implemented)
   Networkaugmented.py
   Trainingaugmented.py
6. Historical data: prevframes.pth
   modifieddataset.py(Dataloader for collecting 4 previous frames)
   networkprevframes.py
   trainingprevframes.py
7. Hyperparameter: SGDwithloss(Model with SGD implemented)
   Modellearn5.pth(Model with learning rate 10^-5)
   Modellearn1.pth(Model with learning rate 1)
8. Dagger: trainingdagger.py
   Networkdagger.py
9. Dataloader: It contains all types of dataloaders used in the entire assignment
   All the models have their loss curves in the same folder

   Kindly use Modelcomp1.pth or Modelcomp2.pth for competition.

   All models were not getting uploaded due to large size, kindly refer to the google drive for all other models :
   https://drive.google.com/drive/folders/115vWDtX3qKstBXQ8zv3gsminFMuZWnCO?usp=share_link