1

a) Function ***load_imitations*** in ***imitations.py*** file.

b) One of the reasons why it is preferred to divide data into batches is because when dataset get large, we would need to store whole dataset in memory in order to do gradient update - and that is usually not possible (on regular machines, especially when using images). One other benefit of dividing the data into batches is that you can update the parameters of the network after every batch (e.g. Stochastic - batch gradient descent) else you would have to go through the entire data to make one update.

Epoch is an event of whole dataset passing through the network (forward and backward pass).

In lines 43-48, for each batch of data we do the following steps:

(a) Predict the classes
(b) calculate the cross entropy loss between the predictions and the ground truth
(c) Reset the gradients to be zero
(d) Backpropagate the loss
(e) Update the parameters of the network

c) Functions ***actions_to_classes***, ***scores_to_actions*** in files ***network.py*** and ***cross_entropy_loss*** in ***training.py***.

In the 'vanilla' implementation they take values: steer {-1, 0, 1}, gas {0, 0.5}, brake {0, 0.8}.

Nine classes are defined as follows:

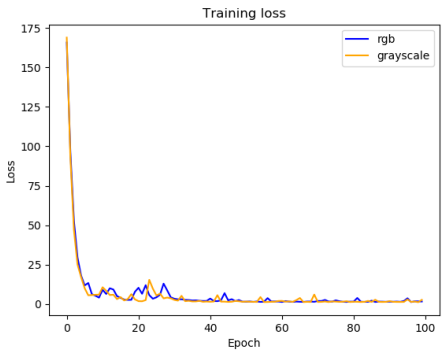
Classes	Corresponding Action
[0.0, 0.0, 0.0]	Do Nothing
[0.0, 0.5, 0.0]	Accelerate(Gas)
[0.0, 0.0, 0.8]	Brake
[1.0, 0.0, 0.0]	Turn Right
[-1.0, 0.0, 0.0]	Turn Left
[1.0, 0.5, 0.0]	Turn Right and Accelerate
[1.0, 0.0, 0.8]	Turn Right and Brake
[-1.0, 0.5, 0.0]	Turn Left and Accelerate
[-1.0, 0.0, 0.8]	Turn Left and Brake

We experimented with 7 classes too, by merging classes: accelerate and turn simultaneously (left or right) into classes: (only) turn left or turn right. We inferred that the performance of the trained network depends on expert's driving style.

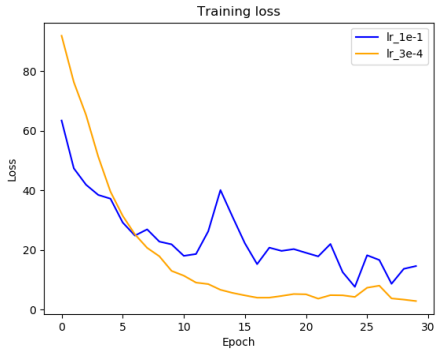
d) Function ***__init__*** of the class ***ClassificationNetwork*** in ***network.py*** file.

e) Function ***forward*** of the class ***ClassificationNetwork*** in ***network.py*** file. We experimented with both gray scaled and colored images (both normalized to [0,1]) Considering scenes on images are not complex consisting mostly of grass and the tracks, even with grey scaled images the network should learn to model all the patterns equally as good as colored. Additionally, the gray scaled images may be better choice since it computationally more demanding. On graph 1a training loss of both cases is displayed. Training time was comparable for both (485 seconds) and evaluation results were 470 for rgb and 457 for gray scale.

We can achieve better results by changing the hyperparameters. Performance of the model is influenced by the values of the hyperparameter - for example by changing the architecture we change the capacity of model and with that we directly influence what relationships we will be able to model. For example,



(a) Rgb vs Gray Scale



(b) Learning rate

by tweaking the learning rate, we affect optimization process e.g. in case of large learning rate we might 'jump' around the optimal value of the parameters and might never reach it. The graph 1b shows how training loss is affected when learning rate value is changed.

f) Function ***save_imitations*** in ***imitations.py*** file.

In this case, good training data should be the one that contains imitations of taking the appropriate action at any state. However, only perfect driving doesn't make the dataset 'good' as it doesn't contain the cases of the expert recovering from potential states which are not optimal but still can happen. In order to learn how to recover from 'bad' states (driving off tracks), with the provided implementation of ***record_imitations*** the expert had to go off the track first. This would result in model driving off the track, as it imitates expert's behaviour. To tackle this problem, we implemented **modified *record_imitations*** function which allows us to start recording only when certain key is pressed and in that way we only record recovering part and augment the data.

2

a) Function ***extract_sensor_values*** takes the bottom part of the image crops regions with sensor information which are displayed as 2d pixel arrays (matrices) and converts their values to numeric representation. We can utilize these values as the network will surely benefit from explicit sensor state information e.g. it should learn that acceleration should happen only up to a certain accelerometer value.

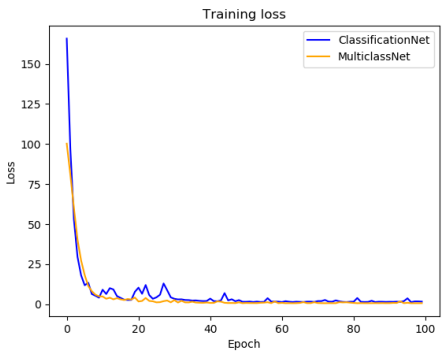
One could argue that convolutional network would be able to extract this data from the image itself (if we wouldn't have cropped the bottom part of the image before inputting it to the first layer), but inputting sensor values directly to fully connected layers we force the network to use exactly this information in addition to features extracted by the convolutional part of the network.

Including sensor information drastically improved agents behaviour and it was able to achieve better rewards. Just as predicted, agent was able to learn when to accelerate or brake.

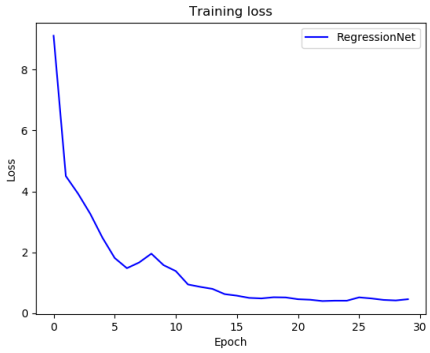
b) Implementation of the class ***MulticlassClassificationNetwork*** in the file ***network.py***. Loss function for multilabel classification is binary crossentropy on each of the sigmoid outputs and summed in end. The way we implemented ***cross_entropy_loss*** function it intrinsically supports exactly described logic so we didn't have to implement new loss function.

Average evaluation score that we observed was 519 which better than the scores achieved by the networks implemented so far. We also observed much stable convergence which can be seen on 2a.

c) Implementation of the class ***RegressionNetwork*** in the file ***network.py***. This problem can be formulated as a regression problem of predicting continuous values for each action field - namely predicting values for steer, accelerate and brake. With regression problem suitable loss is ***mean_squared_error***, implemented in file ***training.py***. Advantage of using this formulation is that we can use the output scores of the regression network as actions (without mapping it to classes). This makes implementation easier. If the training dataset contained more variable values for actions e.g. steer could take any real value between [-1,1] model would be able to predict those, where in case of classification, these values would have to be discretized. From the training process, one might conclude that because the convergence is better 2b that evaluation score would be too. We observed that this is not the case as avergagevaluation score was



(a) Classification vs Multi-Class



(b) Regression Network

117. The drawback that could be observed from the behaviour of the agent was that because the values of actions were predicted as continuous variables, agent performed most of the times all of the actions at the same time, even though the values were small e.g. when the optimal action would be [0, 0.5, 0] agent predicted [-0.1, 0.5, 0.05].

d) We augmented the dataset by flipping(taking the mirror images) the left images and actions to be right and correspondingly. The code is implemented in Lines 32-56 of the ***training.py*** file . Flipping the images increased the ability of our model to take the right turns more accurately as before flipping it contained only images where the car was almost taking the left turns and hence the car wasn't able to learn taking a right turn suitably.

e) We tried different techniques for improving the performance of the network. As the dataset was imbalanced we implemented weights of the classes in cross entropy loss in which we penalize the network for mistakes on instances of underbalanced classes more. This showed improvement to some extent, but if the imbalance was severe as in our case, it sometimes neglected overbalanced class even though performance of that class was more important.

We also used Batch-Normalization after convolutional layers and fully connected layers to normalize the input to each layer and it helped in the faster convergence of our model.

We used learning rate adaptation for the sake of faster and better convergence which showed as effective.

As a regularizer we used weight decay, but omitted dropout as it didn't seem necessary. Furthermore following paper suggested that use of these two doesn't play well.

We also experimented with different learning rates and found that if the learning rate $l_r = 1e-1$ then the training loss was flickering and wasn't able to reach an optimal state. Whereas, with learning rate $l_r = 3e-4$ our model was able to reach at an optimal state and hence the performance was much better.

We also experimented with different architectures but the 'vanilla' one proved to be sufficient as the image resolution was not very large and the content was not complicated.

For the training of our model what seemed to help the most was augmenting data with additional recordings.