Imitation Learning

Play Car Racing by OpenAI gym by training an agent using Imitation Learning

Imitation Learning

Given a set of demonstrations extract State/Action pairs, train a policy to mimic the given demonstrations. Contrast this with Reinforcement Learning where a Reward Function is used to achieve the desired results.

Behavioral Cloning

The simplest form of Imitation Learning is Behavioral Cloning: the Policy is generated by Supervised Learning. Since the State in CarRacingV0 is RGB image, a CNN model is suitable.

Known limitations of Behavioral Cloning

Recovery by the Agent after a mistake on a state the expert has never visited and never trained on, can be not possible. Overfitting to Expert's biases.

Advantages

Simple, Efficient

Other possible solutions to consider

Transfer Learning, Direct Policy Learning, Interactive Learning.
Due to time limitation, I'll proceed with Behavioral Cloning through Supervised Learning.

The problem is now easily reduced to a Supervised Learning for Classification.

Data Collection: Expert data for the agent

I wrote a simple script to collect expert data. (get-data.py)

The action a, is an array of size 3 which contains the commands for:
Direction (Left: -1, Right: 1, none: 0)
Acceleration(Accelerate: 1, none: 0)
Deceleration(Decelerate: 0.8, none: 0)

The action array is saved in a csv file.

There are two available formats to save the state: STATE_W, STATE_H (96x96) and VIDEO_W, VIDEO_H(600x400)
After a few image resizing exercises, I decided to save VIDEO_W, VIDEO_H(600x400) in rgb_mode.

The dilemma while collecting data was should I collect it as 'Expert' meaning play the game to collect the highest final reward, or should I focus on collecting a balanced dataset. I think it's not possible to collect a balanced dataset. But, there should also be some data representing the least used key(0.8 for deceleration)

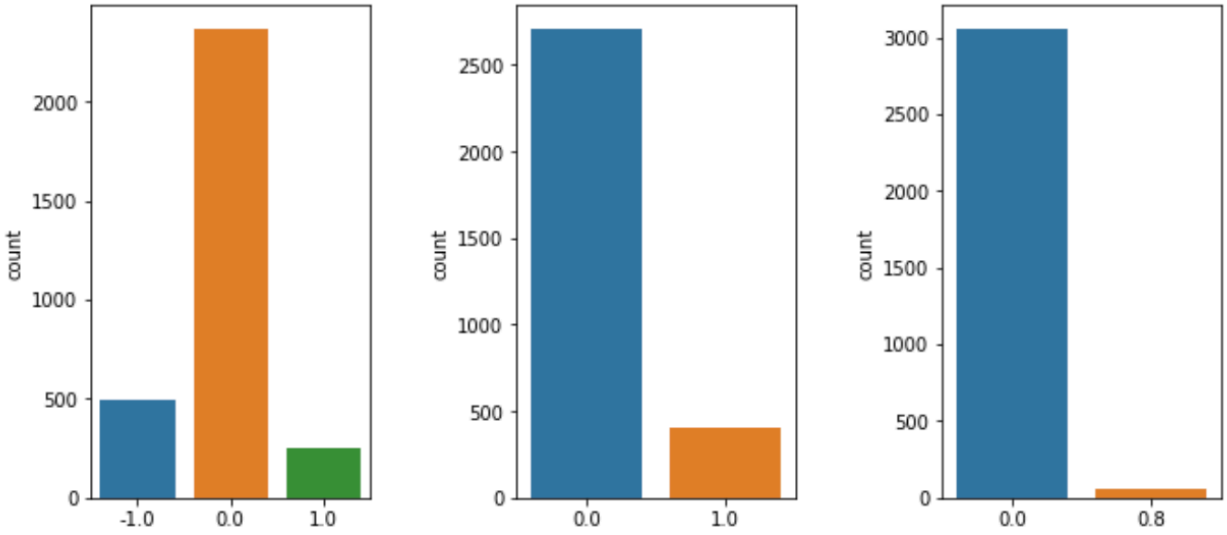
Data Processing

Exploratory data analysis

The two main objectives in this section is to check the Distribution of data

1. Distribution of Data

As expected the dataset is heavily unbalanced. Especially for Breaking(Deceleration) data. The following figure from left to right is distribution count for Direction, Acceleration, Deceleration.



One way to fix this issue is by using class weights in the model.fit. Other ways are SMOTE and k-fold cross validation. I will proceed by using calculating class weights.

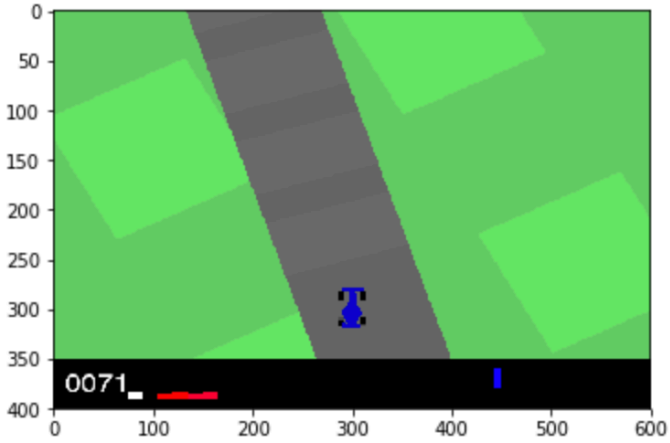
2. Preprocessing the images and class labels:

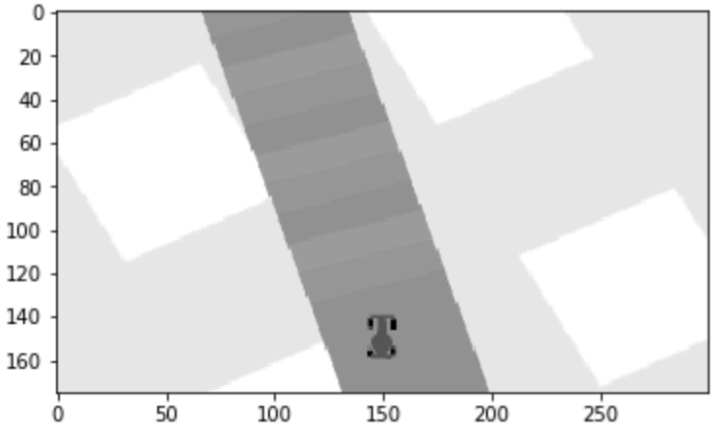
There are a few steps I want to make before feeding the input data to a CNN model. In order to have a proper comparison of the different models, the training data is the same for all models.

These are:
1. One hot encoding of class labels
2. Convert to Greyscale
3. Crop lower progress bar
4. Rescale image
5. Convert image to array
6. Append image array to training_data
7. Append output labels to training_data
8. Shuffle data
9. Normalize data
10. Save data in pickle format

I removed the progress bar at the bottom of the image because I don't want the model to train on the frame number shown.

Example of image before and after Image Processing:

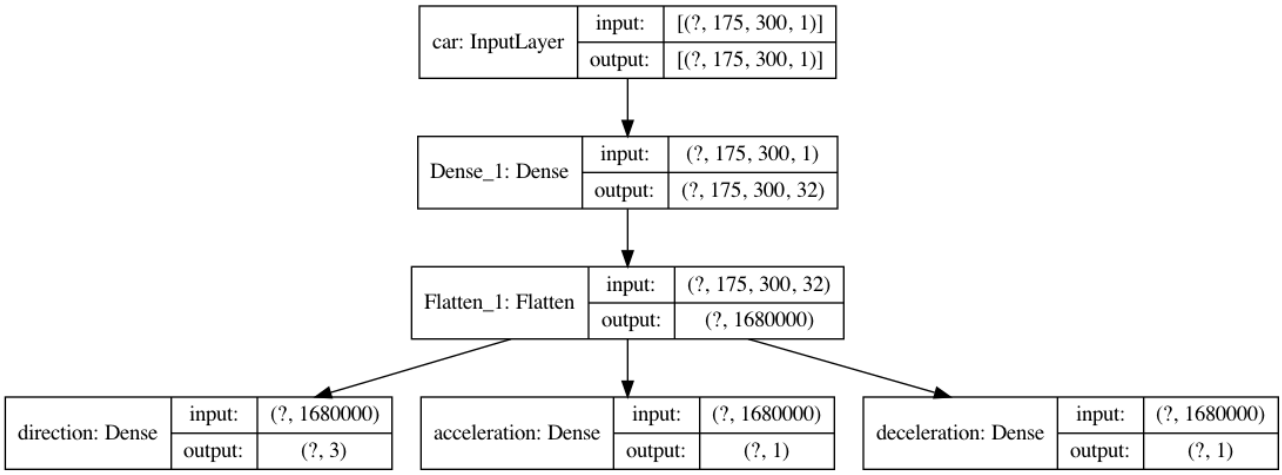




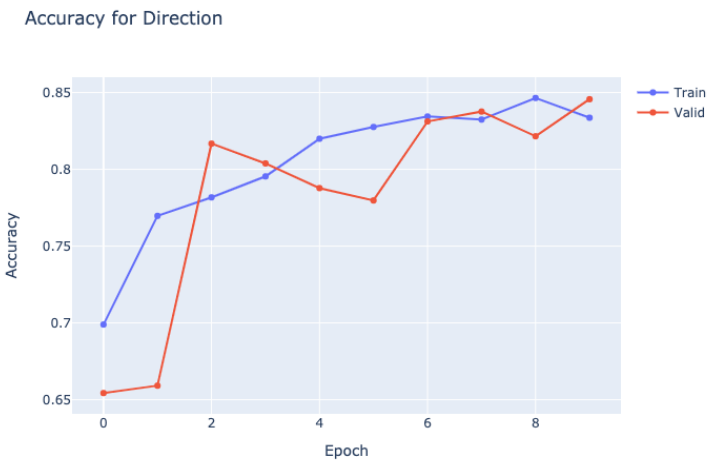
Model Building

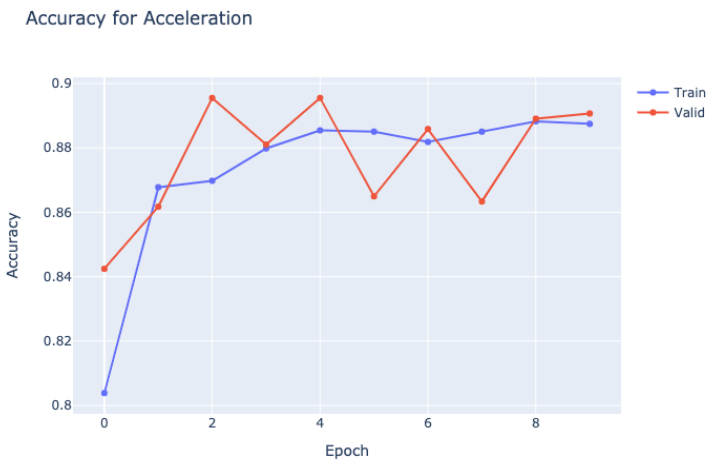
Baseline Model

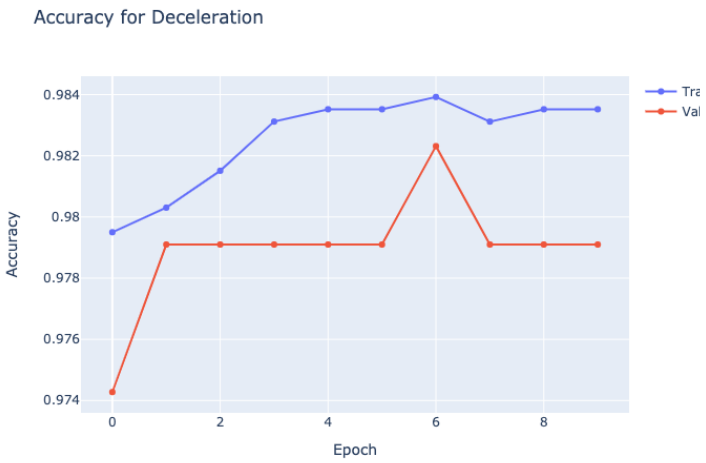
Creating a Baseline Model is important to compare Classification results. For this Baseline model, I created a multi-output-CNN with a couple of layers.

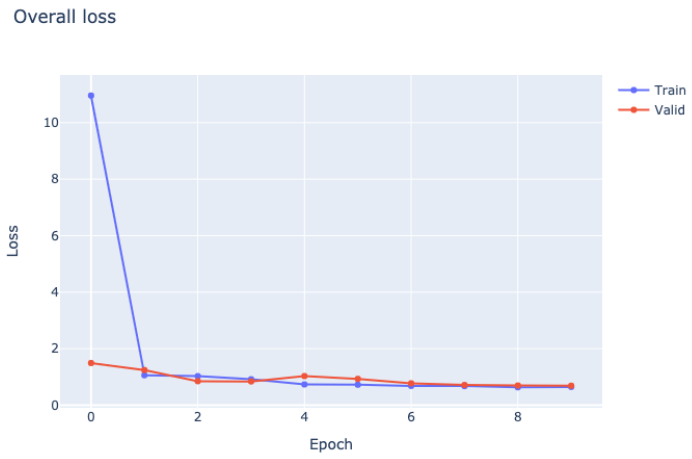


The training and validation results are as follows for [Direction, Acceleration, Deceleration] and the overall loss:



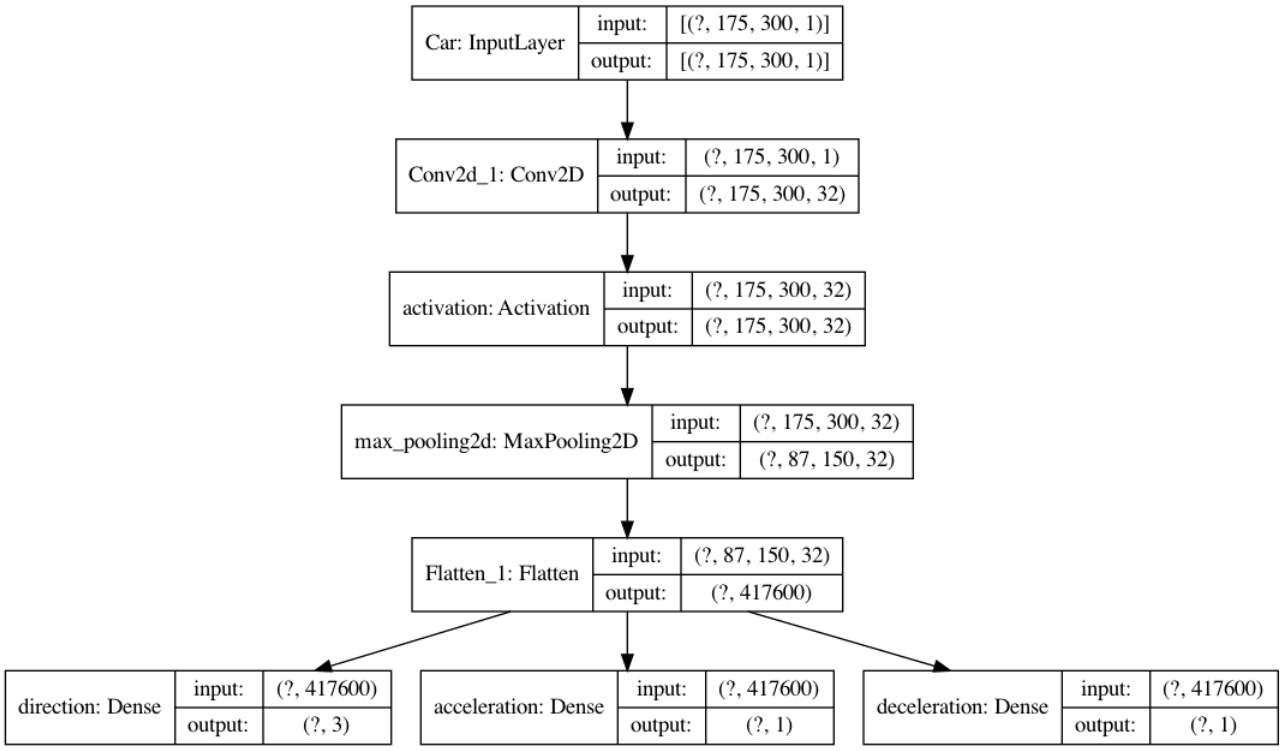






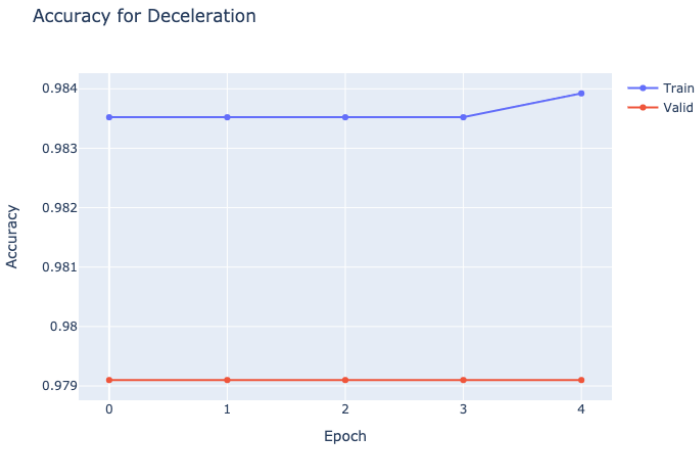
Model 1: A simple CNN

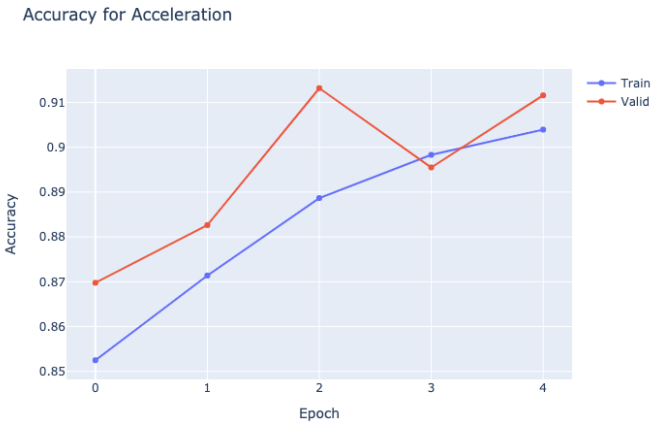
After a brief literature survey, I learnt that simple CNN architectures work well for Car Racing type of problems. In addition to that, I observed that the model works well for low number of Epochs.

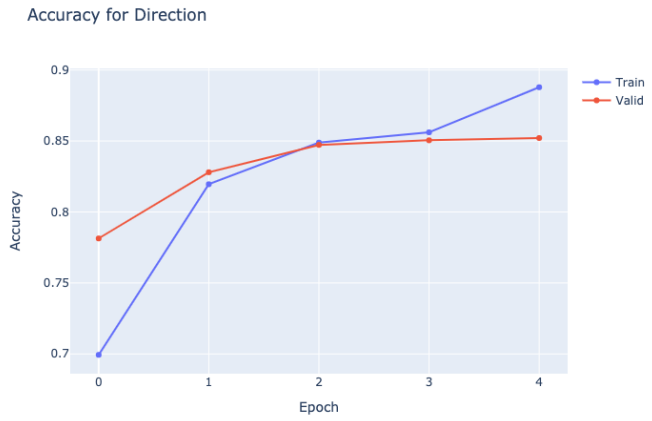


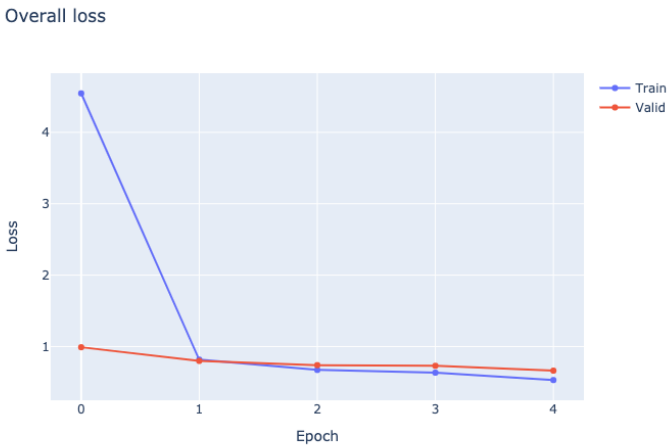
Expectation:

1. The model may overfit
2. The model doesn't learn for Deceleration and other low representations of data classes
3. Irregular training-validation accuracy curves over epochs due to stochastic nature of neural nets



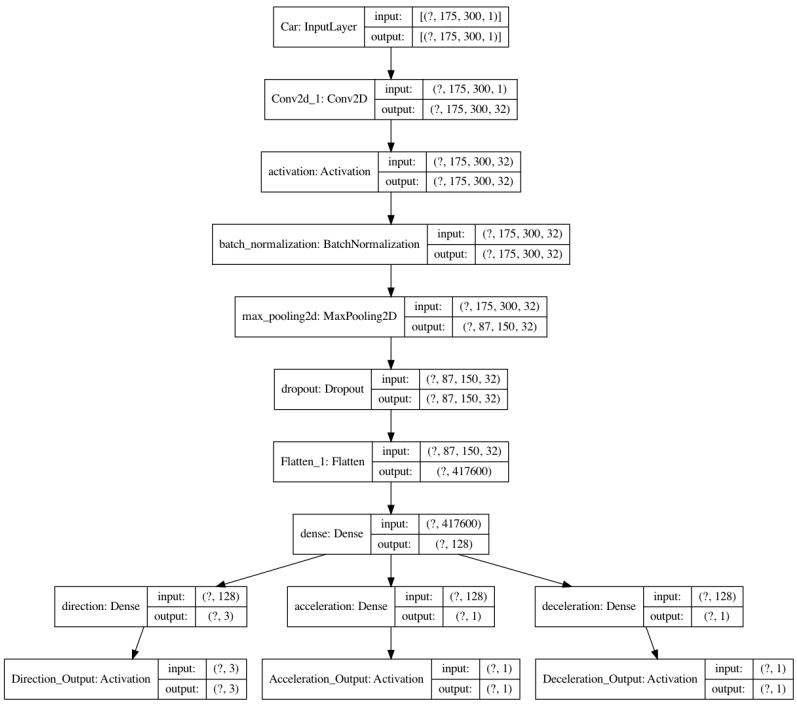




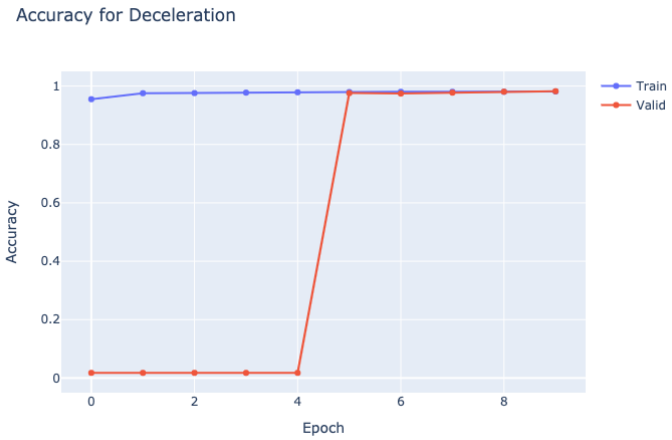


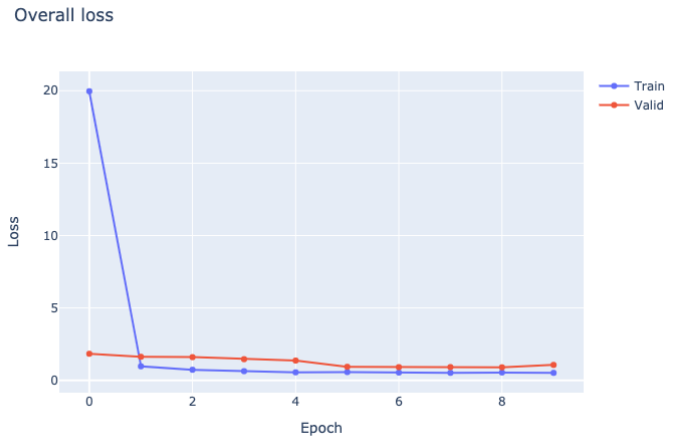
Model 2: Multi Output CNN model, with more layers

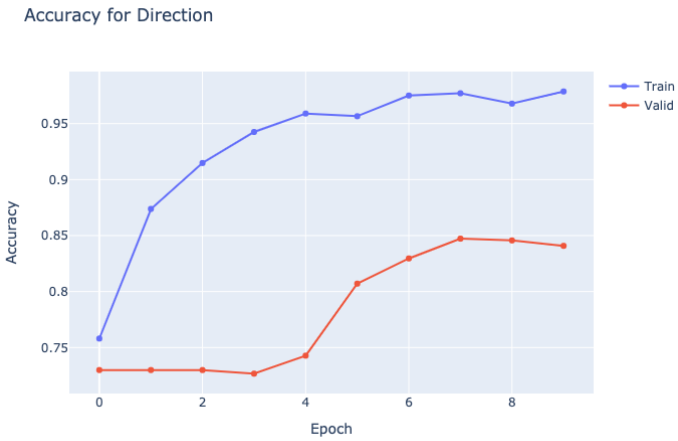
To improve accuracy while decreasing overfitting, I wanted to make a deeper model and add more layers to each branch.

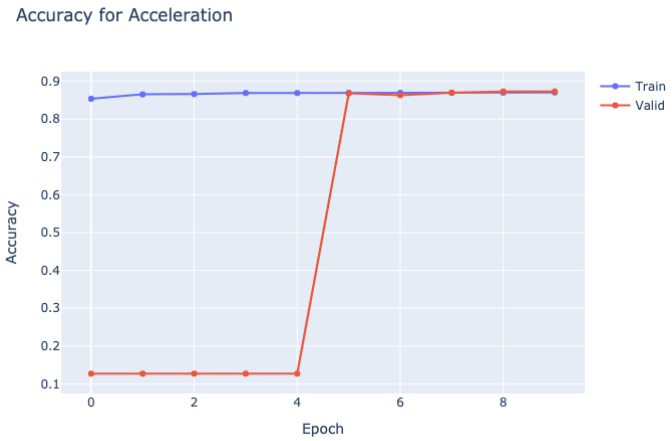


As expected, lower number for epochs is good enough. The comparable difference between testing and validation curves in direction branch shows that there is still the problem of overfitting









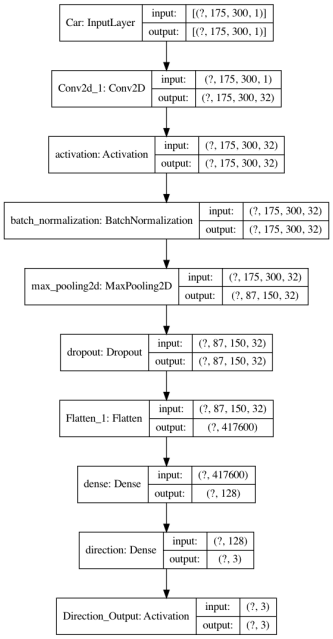
One problem I faced with using Keras Functional API is that it doesn't accept Class Weights for a MultiOutput model. Popular opinion online seems to be it's a bug in the later versions of Keras.

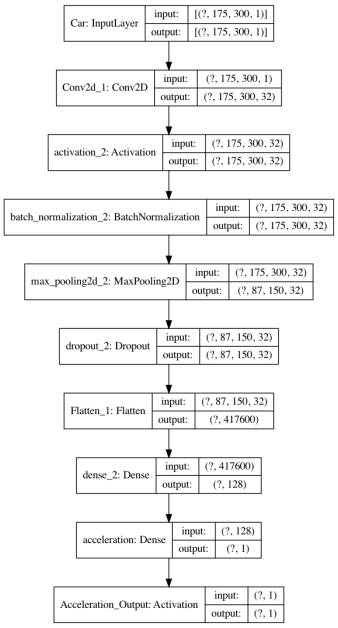
I still strongly feel the need to use class weights. I didn't want to waste time experimenting with Keras versions, and I also wanted to check if Multi model architecture is better than a multi-output architecture. The general assumption is that for classification problems, a multi model architecture is better suited than a multi-output model. I wanted to see if holds true for this problem.

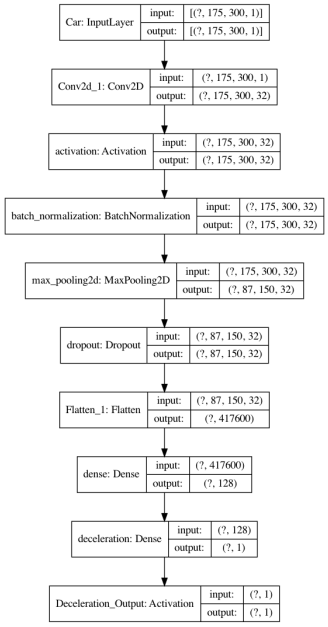
Model 3 - Multi Model CNN

Instead of having three branches from a single CNN, this architecture has 3 different models for each of Direction, Acceleration, and Deceleration. The input data used is still the same, except for the mapping of the input features and output labels.

I was able to utilize the class weights that I calculated early on. Unfortunately, the issue of overfitting an under sampled dataset was not resolved.







Summary for Direction model:

```
Epoch 1/10
78/78 [==============================] - 70s 897ms/step - accuracy: 0.873
4 - loss: 0.3744 - val_loss: 1.2807 - val_accuracy: 0.7781
Epoch 2/10
78/78 [==============================] - 68s 873ms/step - accuracy: 0.946
5 - loss: 0.1551 - val_loss: 0.7778 - val_accuracy: 0.8215
Epoch 3/10
78/78 [==============================] - 68s 874ms/step - accuracy: 0.941
7 - loss: 0.1397 - val_loss: 1.1567 - val_accuracy: 0.6109
Epoch 4/10
78/78 [==============================] - 70s 893ms/step - accuracy: 0.958
6 - loss: 0.0840 - val_loss: 0.5521 - val_accuracy: 0.8441
Epoch 5/10
78/78 [==============================] - 70s 901ms/step - accuracy: 0.973
5 - loss: 0.0646 - val_loss: 0.5412 - val_accuracy: 0.8505
Epoch 6/10
78/78 [==============================] - 70s 894ms/step - accuracy: 0.983
5 - loss: 0.0395 - val_loss: 0.6968 - val_accuracy: 0.8296
Epoch 7/10
78/78 [==============================] - 70s 901ms/step - accuracy: 0.983
5 - loss: 0.0281 - val_loss: 0.6733 - val_accuracy: 0.8650
Epoch 8/10
78/78 [==============================] - 71s 913ms/step - accuracy: 0.990
4 - loss: 0.0141 - val_loss: 0.7234 - val_accuracy: 0.8714
Epoch 9/10
78/78 [==============================] - 69s 886ms/step - accuracy: 0.991
6 - loss: 0.0206 - val_loss: 0.7636 - val_accuracy: 0.8826
Epoch 10/10
78/78 [==============================] - 69s 883ms/step - accuracy: 0.965
8 - loss: 0.0816 - val_loss: 1.1229 - val_accuracy: 0.8537
```

Unexpectedly the model is overfitting in-spite of using Class Weights.

Summary for Acceleration model:

```
Epoch 1/10
78/78 - 73s - accuracy: 0.8666 - val_loss: 0.5680 - val_accuracy: 0.8714
- loss: 0.5051
Epoch 2/10
78/78 - 71s - accuracy: 0.8666 - val_loss: 0.5680 - val_accuracy: 0.8714
- loss: 0.5051
Epoch 3/10
78/78 - 71s - accuracy: 0.8666 - val_loss: 0.5680 - val_accuracy: 0.8714
- loss: 0.5051
Epoch 4/10
78/78 - 71s - accuracy: 0.8666 - val_loss: 0.5680 - val_accuracy: 0.8714
- loss: 0.5051
Epoch 5/10
78/78 - 71s - accuracy: 0.8666 - val_loss: 0.5680 - val_accuracy: 0.8714
- loss: 0.5051
Epoch 6/10
78/78 - 71s - accuracy: 0.8666 - val_loss: 0.5680 - val_accuracy: 0.8714
- loss: 0.5051
Epoch 7/10
78/78 - 69s - accuracy: 0.8666 - val_loss: 0.5680 - val_accuracy: 0.8714
- loss: 0.5051
Epoch 8/10
78/78 - 70s - accuracy: 0.8489 - val_loss: 0.6946 - val_accuracy: 0.8714
- loss: 0.4761
Epoch 9/10
78/78 - 70s - accuracy: 0.8577 - val_loss: 0.7331 - val_accuracy: 0.7379
- loss: 0.4681
Epoch 10/10
78/78 - 73s - accuracy: 0.8569 - val_loss: 0.6232 - val_accuracy: 0.8682
- loss: 0.4757
```

Summary for Deceleration model:

```
Epoch 1/10
78/78 - 67s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 2/10
78/78 - 69s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 3/10
78/78 - 71s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 4/10
78/78 - 69s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 5/10
78/78 - 72s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 6/10
78/78 - 66s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 7/10
78/78 - 67s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 8/10
78/78 - 68s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 9/10
78/78 - 68s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
Epoch 10/10
78/78 - 69s - accuracy: 0.9723 - val_loss: 0.5129 - val_accuracy: 0.9807
- loss: 0.5181
```

Testing

For testing the classification models, new data was procured and saved in the same format as data used for training and validation.

Classification report for the various models:

Classification report for Baseline model

	precision	recall	f1-score	support
left	0.16	0.16	0.16	216
no action	0.77	0.80	0.79	1157
right	0.09	0.06	0.07	115
accuracy			0.65	1488
macro avg	0.34	0.34	0.34	1488
weighted avg	0.63	0.65	0.64	1488

	precision	recall	f1-score	support
0.0	0.88	1.00	0.94	1308
1.0	0.00	0.00	0.00	180
accuracy			0.88	1488
macro avg	0.44	0.50	0.47	1488
weighted avg	0.77	0.88	0.82	1488

	precision	recall	f1-score	support
0.0	0.98	1.00	0.99	1463
1.0	0.00	0.00	0.00	25
accuracy			0.98	1488
macro avg	0.49	0.50	0.50	1488
weighted avg	0.97	0.98	0.97	1488

Classification report for Simple CNN model

	precision	recall	f1-score	support
left	0.00	0.00	0.00	216
no action	0.78	1.00	0.87	1157
right	0.00	0.00	0.00	115
accuracy			0.78	1488
macro avg	0.26	0.33	0.29	1488
weighted avg	0.60	0.78	0.68	1488

	precision	recall	f1-score	support
0.0	0.88	1.00	0.94	1308
1.0	0.00	0.00	0.00	180
accuracy			0.88	1488
macro avg	0.44	0.50	0.47	1488
weighted avg	0.77	0.88	0.82	1488

	precision	recall	f1-score	support
0.0	0.98	1.00	0.99	1463
1.0	0.00	0.00	0.00	25
accuracy			0.98	1488
macro avg	0.49	0.50	0.50	1488
weighted avg	0.97	0.98	0.97	1488

Classification report for Many layer multi output CNN model

	precision	recall	f1-score	support
left	0.14	0.51	0.22	216
no action	0.77	0.46	0.58	1157
right	0.00	0.00	0.00	115
accuracy			0.43	1488
macro avg	0.30	0.32	0.27	1488
weighted avg	0.62	0.43	0.48	1488

	precision	recall	f1-score	support
0.0	0.88	1.00	0.94	1308
1.0	0.00	0.00	0.00	180
accuracy			0.88	1488
macro avg	0.44	0.50	0.47	1488
weighted avg	0.77	0.88	0.82	1488

	precision	recall	f1-score	support
0.0	0.98	1.00	0.99	1463
1.0	0.00	0.00	0.00	25
accuracy			0.98	1488
macro avg	0.49	0.50	0.50	1488
weighted avg	0.97	0.98	0.97	1488

Classification report for Multiple model CNN

	precision	recall	f1-score	support
left	0.14	0.70	0.23	216
no action	0.77	0.27	0.40	1157
right	0.00	0.00	0.00	115
accuracy			0.31	1488
macro avg	0.30	0.32	0.21	1488
weighted avg	0.62	0.31	0.34	1488

	precision	recall	f1-score	support
0.0	0.88	1.00	0.94	1308
1.0	0.00	0.00	0.00	180
accuracy			0.88	1488
macro avg	0.44	0.50	0.47	1488
weighted avg	0.77	0.88	0.82	1488

	precision	recall	f1-score	support
0.0	0.98	1.00	0.99	1463
1.0	0.00	0.00	0.00	25
accuracy			0.98	1488
macro avg	0.49	0.50	0.50	1488
weighted avg	0.97	0.98	0.97	1488

Since there weren't many samples for all classes, the Precision and F-score were automatically set to 0.

The evaluation metric for this problem is Accuracy

Models \ Accuracy(%)	Direction	Accelerate	Decelerate	Average
Baseline Multi-output model	65	88	99	84
Simple Multi-output CNN	**78**	**88**	**98**	**88**
Many layer Multi-output CNN	43	88	98	76.3
Multi model CNN	31	88	98	72.3

The simple multi-output CNN model has the best overall average, and individual averages for Direction, Acceleration, and Break. So this is the model chosen to be used in predicting the action values in playing CarRacingV0.

Playing CarRacingV0

During the implementation of the agent, there was some incompatibility between my env, tensorflow, and gym. I spent some time to trying to fix the issue but haven't been successful so far. If it works, my hypothesis is that the predictions for the first 100-150 frames are excellent, after which, if there is a curve in the race track, the agent may not be able to recover from a fatal mistake and losses the game.

Discussions

1. Unbalanced dataset:
    1. Since the training data is from an expert playing the game, it leads to biases. At this point the question that needs to be answered is:
        1. What is the aim of training an agent? Is it to play exactly like one player or is it to generalize for all players.
        2. If the aim is to get the highest reward at the end of one session, then reward has to be a feature in building the model.
        3. Even with methods of compensating for unbalanced dataset such as k-fold cross validation, SMOTE, class weights etc., I strongly recommend collecting data from n number of expert players.
        4. Since the simulation environment is pretty simple, I don't think there's a need to generate huge amounts of data. Perhaps varied data to balance all the classes.
2. Improving; other algorithms to try:
    1. A promising algorithm to try in the future is Direct Policy Learning, DAgger, and Inverse Reinforcement Learning. These 3 seem to be more suited for applications such as automobile navigation.

3. Something I wanted to try out was if there is a difference or advantage when using RGB vs greyscale. I used greyscale to reduce the input dimensions to facilitate faster training times.
4. Treating this problem as i.i.d (independent and identically distributed) has its advantages especially with quickly setting up a model. But, if the aim is to maximize the total reward, then data history is also important.

Possible improvements

1. Dataset:
    1. More varied data
2. Algorithm:
    1. Interactive learning algorithm, updating the reward function etc
3. Development side:
    1. Using subclasses for calling keras layers
    2. K-fold cross validation
    3. Color vs. greyscale input
    4. Functions for redundant code

Conclusion

Behavioral Cloning, a form of Imitation Learning was boiled down to Supervised Learning for Classification based on image input with multiple outputs. With the evaluation metric of Accuracy, a Simple Multi-Output CNN model is found to outperform Many-layer-Multi-output CNN model and Multi-model CNN model.
However, there are multiple issues with the prepared models. The biggest of which is overfitting and unbalanced dataset.
While Imitation Learning(Behavioral Cloning) has its disadvantages, it is extremely useful in quickly building a model and when the use case is a simple application.

Github link:

https://github.com/PaulthiV/ImitationLearning-CarRacingV0

Resources:

1. https://smartlabai.medium.com/a-brief-overview-of-imitation-learning-8a8a75c44a9c#:~:text=The%20simplest%20form%20of%20imitation,steering%20angles%20and%20drive%20autonomously.
2. https://sites.google.com/view/icml2018-imitation-learning/
3. Imitation Learning: A Survey of Learning Methods, Hussein et al. https://core.ac.uk/download/pdf/141207521.pdf
4. https://towardsdatascience.com/my-learning-from-udacity-sdc-nanodegree-do-we-really-need-a-complex-cnn-and-hours-of-training-4f80e28af90b
5. https://keras.io/guides/functional_api/
6. https://blog.statsbot.co/introduction-to-imitation-learning-32334c3b1e7a