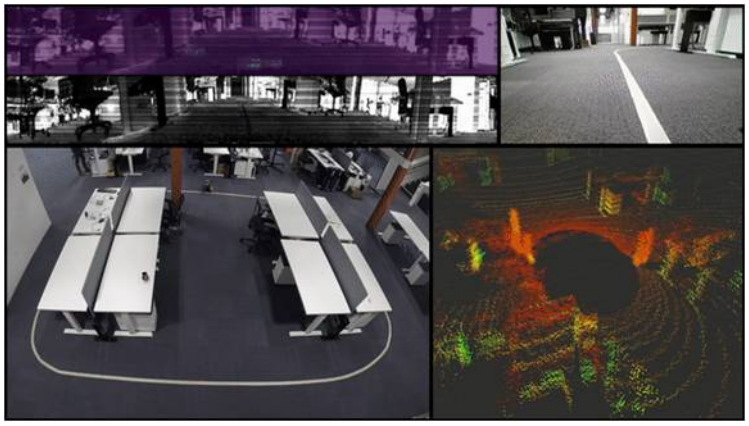
2020 年 3 月 4 日

OS-1 ライダーの反射強度データを用いた自律運転 ML アルゴリズムの開発



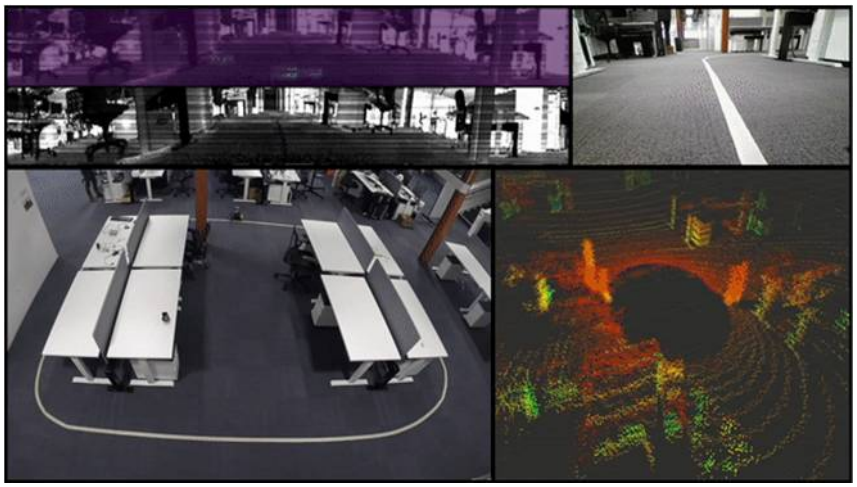
Wil Selby

センサーに関する注意：本投稿では、第一世代 OS1-64 ファームウェア 1.12 を使用しました。YouTube チャンネルをチェックして、最新センサーの最新データをご覧下さい。

本プロジェクトの目的は、機械学習モデルを開発して、Ouster OS1 ライダーセンサーを主たるセンサーの入力源とし、RC（リモコン）カーをレーストラック上で自動走行させることです。モデルは、末端間の畳み込みニューラルネットワーク（Convolutional Neural Network：CNN）となり、ライダーからの反射強度の画像データを処理して、RC カーへステアリングコマンドを出力します。

これは、Udacity 自動走行車モジュール“behavior cloning”、および、DIY ロボカーレースから影響を受けたものです。本プロジェクトは、ウェブカムにより得られたカラーカメラ画像を用いて自動で RC カーを走らせる同様のモデルを開発することであり、以前の開発に基づいて構築されています。

この投稿では、データの主たるパイプラインを開発することであり、学習データの収集、処理、そして、Google Colab を用いた ML モデルの学習、ROS を用いた RC カーへのモデルの統合と導入を含んでいます。



ライダー搭載 RC カーに基づく ML モデルの使用

プラットフォームの概要

本プロジェクトでは、ベースとなる RC プラットフォームは、Exceed Blaze (Hyper Blue)ビークルです。ハードウェアを搭載するために、プラットフォームは Standard Donkey Car Kit で改造されています。完成したシステムを以下の画像に示します。

OusterOS1 を搭載した RC カープラットフォーム

センサーに関しては、OusterOS-1-64 ライダーセンサーを使用する予定です。システムには、Genius WideCam F100 webcam と OpenMV Cam M7 も搭載されていますが、今回のプロジェクトでは使用していません。

ライダーセンサーは、従来、豊富な空間情報を含む 3D 点群をアウトプットします。しかし、今回のデータは、ML アプリケーションのカメラ画像に較べて、取り扱いが一層チャレンジングなものになります。ブログ投稿"The Lidar IS the Camera"では、OS-1 ライダーが固定の解像度から構成された深度画像、信号画像（反射強度画像）、周辺画像（アンビエント画像）を全てリアルタイムで、カメラ無しでアウトプットできる方法について詳述しています。オープンソースドライバーは、これらのデータレイヤーを、固定解像度の 360° パノラマフレームとして出力します。これにより、カメラで撮ったような画像が得られ、カラーカメラを用いて作成された同じ ML モデルと、同等に扱うことが可能になります。投稿ブログが以下に述べるように、

OS1 は近赤外線で反射強度とアンビエント画像の両方を取得できるので、データは同じ現

場の可視光画像に似たものとなり、そのため、データは自然な外観を呈し、カメラ用に開発されたアルゴリズムが適切にデータ変換できる可能性が高まります。

全プロセスは、Arduino Uno 付きの Compulab fitlet2 上で行われ、PCA 9685 Servo Driver を介して、モーターにステアリングとスロットルのコマンドが供給されます。車両は、Xbox コントローラーでリモート操作が可能となります。

Compulab fitlet2 は、ベースオペレーティングシステムとして、Ubuntu 18.04 を起動します。ROS はミドルウェアとして用いられ、ソフトウェアコンポーネントの間で、またロボット専用ライブラリとビジュアライザーと間で、プロセス間コミュニケーションをとるために使用されます。ML モデル開発に関しては、Tensorflow と Keras が Google Colab とともに用いられ、学習環境を提供します。

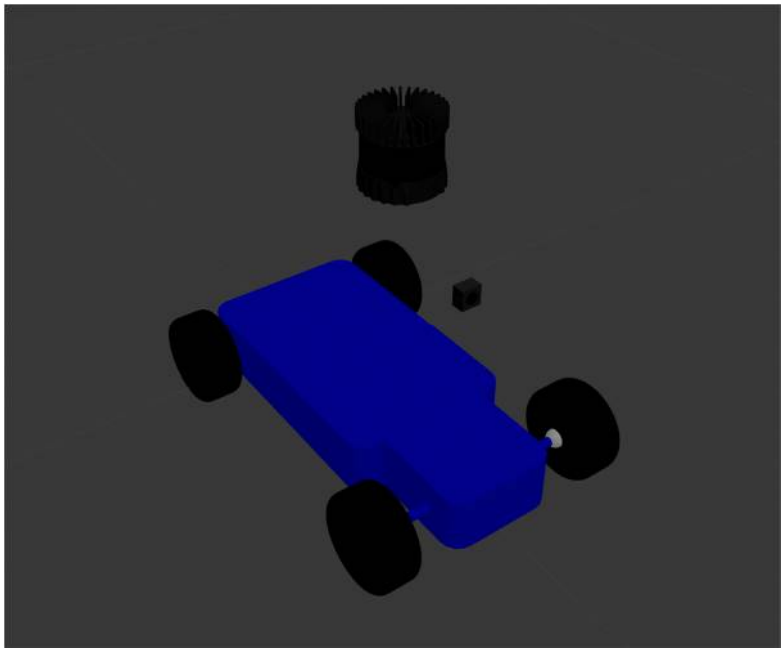
これまでの開発

今回の開発の主な動機の 1 つは、OS1 ライダーセンサーからのカメラ-似の画像が、もともとカラーカメラ画像に基づく ML モデルアーキテクチャで使用できるかどうかを見定めるものです。

以前、同じハードウェアとソフトウェアシステムを使用して、カラーカメラ画像を用いた ML モデルを開発し、同様なレーストラックを RC カーが確実に操作できるかどうかを検証したことがあります。

実世界のハードウェアを使用する前に、シミュレーション環境を使用して、データ・パイプライン、モデル学習、モデル導入プロセスを検証しました。ROS Gazebo をシミュレーション環境として、また RViz をシミュレーションカメラのアウトプットの可視化に使用しました。

シミュレーション画像を作成するため、シミュレーション RGB カラーカメラを MIT レースカープロジェクトから提供されたシミュレーション RC カーの上に取り付けました。

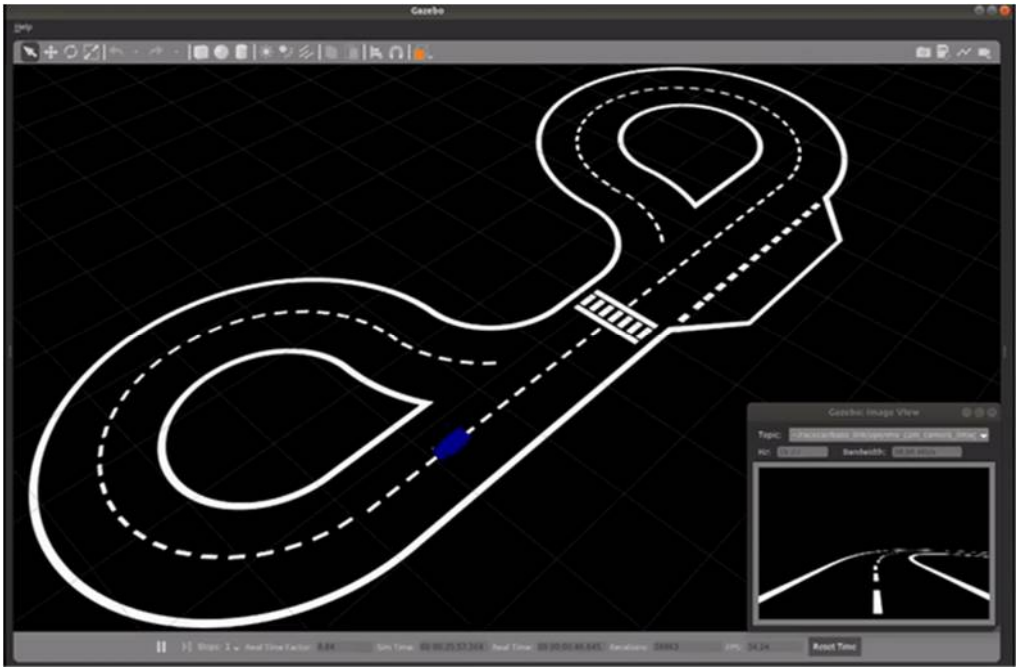


MIT レースカーGazebo シミュレーション

シミュレーション OusterOS1 も車両に搭載されましたが、その時は使用しませんでした。投稿“Simulating an Ouster OS-1 lidar Sensor in ROS Gazebo and RViz”に ROS を使用した OusterOS1 センサーのシミュレーションプロセスが詳述されています。

最後に、Valter Costa によって、論文のため開発された Conde world ファイル“Autonomous Driving Simulator for Educational Purposes”が、シミュレーション環境として使用され、システムのオペレーション、学習用のテストトラックとして提供されました。

学習データを収集するために、シミュレーション車両がマニュアル操作でトラック周りを駆動し、ステアリングコマンドとシミュレーションの画像が ROS bag フォーマットで記録されました。

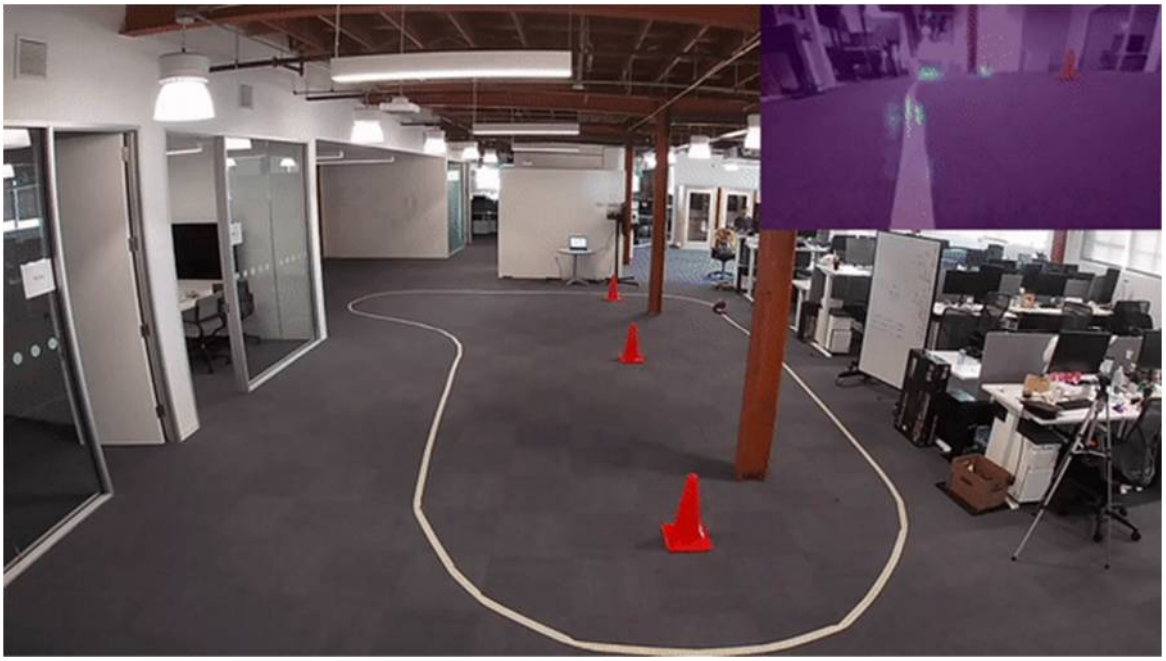


ROS GazeboML モデル学習データ収集

そして、データは処理されて、ML モデルの学習に使われます。この用途で、NVIDIA により開発され、彼らの投稿ブログ“End-to-End Deep Learning for Self-Driving Cars”（自律走行車用末端間深層学習）で述べられているモデルを実行しました。彼らは、畳み込みニューラルネットワーク（CNN）を開発し、1 つのフロントカメラから、ステアリングコマンドに、直接、生のピクセルをマップます。

ROS ノードが開発され、学習された ML モデルが推論用に使用されました。シミュレーション環境では、ML モデルは、シミュレートされたカラーカメラ画像とアウトプットされたステアリングコマンドを処理し、データ・パイプライン、モデル学習、モデル導入プロセスを検証しながら車両をコントロールしていきます。

プロセスは実世界でも繰り返され、トラック周りの学習データが収集され、モデルが学習され、物理的なプラットフォーム上にモデルが導入されていきます。システムのパフォーマンスはうまく発揮され、以下に示すように RC 車両をトラック周りで、自律的に操縦することに成功しました。



RC 車両末端間 ML モデル検証

実世界におけるデータ収集

カラーカメラ画像を用いたシステム検証が完了したら、次のステップは、システムを OusterOS1 ライダーデータに適合することです。

RC カーを扱う場合、diy_driverless_car_ROS レポジトリ、特に rover_ml パッケージが使用されます。

マニュアル操作で RC カーを操作する場合、rc_dbw_cam.launch ファイルを使用して、システムが開始されます。本プロジェクトに関しては、OS1 が 1024×20 モードで操作されました。カラーカメラのフレームレートによりマッチさせるために、より高速のフレームレートが使用されました。

$ roslaunch rover_teleop rc_dbw_cam.launch os1:=true ekf:=false

record_training.sh スクリプトが学習データを記録するのに使用されました。

$ source record_training.sh

関心をひく主要なトピックは、

/racecar/ackermann_cmd_mux/output トピック

で発行されるステアリングコマンド、および、

/img_node/intensity_image トピック上のライダーの反射強度チャンネル画像です。

車両は、トラック周りを、トータルで 10 周、各方向に 5 周づつ、マニュアルで操作されました。

学習データを収集し終わったら、体系化し、処理する必要があります。これには、ライダー画像データや ROSbag からのステアリングコマンドデータの抽出や Tensorflow と Keras で操作が容易なフォーマットでの保存も含まれます。

ライダー画像は JPG ファイルとして、それらが取得されたタイムスタンプに基づくユニークなファイル名で保存されます。ステアリングコマンドは、CSV ファイルでも抽出、保存されます。CSV ファイルの各行は、その画像に関連するステアリング、スロットル値と同様、画像のファイル名もリストされます。ライダー画像とステアリングコマンドは異なる周波数で発信されるので、基本的な補間が行われ、各ライダー画像が取得されるごとにステアリングコマンドが評価されます。

変換を実行するには、rosbag_to_csv.py スクリプトを使用します。これは、Ross Wightmans Ross Wightmans bagdump.py スクリプトを若干修正したバージョンです。

この CVS ファイル（補間された.csv）と関連する JPG 画像が学習データセットの基盤を形成します。CSV 出力の例を以下に示します。

A	B	C	D	E	F	G	H
index	timestamp	width	height	frame_id	filename	angle	speed
2019-12-26 22:54:05	1577400844913415168	1024	64		center/1577400844913415168.jpg	0.3500754054	0.05858492479
2019-12-26 22:54:05	1577400844963412992	1024	64		center/1577400844963412992.jpg	0.8928326253	0.05858492479
2019-12-26 22:54:05	1577400845013411840	1024	64		center/1577400845013411840.jpg	1.235748818	0.06160691624
2019-12-26 22:54:05	1577400845063418624	1024	64		center/1577400845063418624.jpg	1.321794062	0.06851439094
2019-12-26 22:54:05	1577400845113428224	1024	64		center/1577400845113428224.jpg	1.404983496	0.07812113386
2019-12-26 22:54:05	1577400845163436288	1024	64		center/1577400845163436288.jpg	1.453057408	0.0856718421
2019-12-26 22:54:05	1577400845213441280	1024	64		center/1577400845213441280.jpg	1.453057408	0.0856718421
2019-12-26 22:54:05	1577400845263491584	1024	64		center/1577400845263491584.jpg	1.447215803	0.08618157172
2019-12-26 22:54:05	1577400845313480960	1024	64		center/1577400845313480960.jpg	1.43246345	0.08746883958
2019-12-26 22:54:05	1577400845363720704	1024	64		center/1577400845363720704.jpg	1.417637212	0.08876255463
2019-12-26 22:54:05	1577400845413688320	1024	64		center/1577400845413688320.jpg	1.404283451	0.09203658035
2019-12-26 22:54:05	1577400845463908352	1024	64		center/1577400845463908352.jpg	1.393013211	0.09839765974

RC カーOuster ライダーデータ ML モデル学習データセット

結果は、ライダー画像のローカルディレクトリと、ステアリング角度コマンドに関連する画像ファイル名と相関を持つ interpolated.csv ファイルになります。そして、データは Google Colab での使用のため、.tar ファイルに保管されます。

Google Colab でのデータ処理とモデル学習

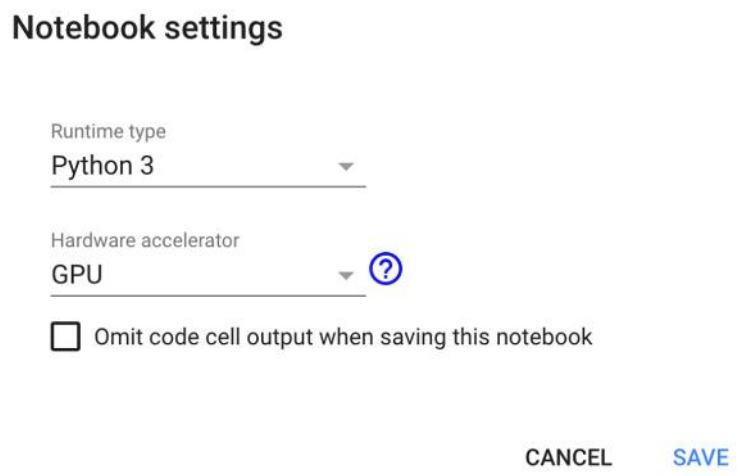
本セクションでは、RC カー内蔵デバイスの代りに、モデル学習プロセスのため Google Colab を使用するプロセスについて述べていきます。データセットサイズあるいはモデルの複雑さに応じて、内蔵のデバイスでは、ML モデルの学習が困難になることがあり得ます。

Google Colab は、機械学習教育と研究のための無料の研究ツールになります。ジュピター

ノートブック環境の１つで、ウェブブラウザを介してアクセスできます。コードは、１２GB の RAM と 320GB のディスクスペースのある仮想マシンで実行されます。Google Colab は、GPU の無料の利用と、今や、同様に TPU を提供します。これにより、大規模データセットを用いて ML モデルの学習が出来る主要なツールとなります。

Google Colab ノートブックセットアップ

我々は、Development of an End-to-End ML Model for Navigating an RC car with a Lidar ノートブックを使用して、データセットの処理、ML モデルの学習、ML モデル性能の評価を実行します。まず、runtime 環境が適切にセットアップされているかを確認します。runtime メニューから、オプションを Change runtime type に変更し、ノートブックの設定メニューにアクセスします。



Google Colab ノートブック設定

ノートブックは、Python2、Python3 と互換性がありますが、ハードウェアアクセラレータドロップダウンメニューから GPU が選択されていることを確認してください。

ノートブックが適切にセットアップされたら、ノートブックの実行を開始できます。ノートブックは現時点では、TF2.0 とは互換ではありません。

%tensorflow_version 1.x コマンドは、TF2.0 の代わりに TF1.0 が使われることを確保します。セルは、Tensorflow と Keras のバージョンを出力します。

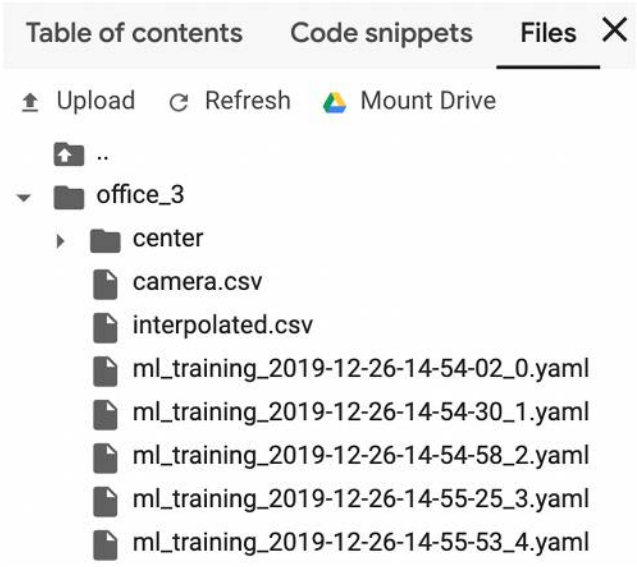
```
Tensorflow Version: 1.15.0
Tensorflow Keras Version: 2.2.4-tf
```

最後に、選択した GPU が利用可能かどうか確認します。GPU が利用可能であれば、その名前も出力されます。現状、Nvidia Tesla P100 が利用可能です。

データプロセス

ノートブックが適切に設定され、環境がセットアップされると、学習用データのロードを開始できます。

データセットは、.tar ファイルとして保存され、AWS でホストされます。データベースは、Colab 環境にダウンロードされ、抽出されます。



Google Colab での RC カーOuster ライダー学習データセットファイル構造

次に、interpolated.csv ファイルを解析し、情報を Pandas データフレームに読み込みます。データフレームのサマリーがセル出力として提供されます。

```
Dataset Dimensions:
(6253, 8)

Dataset Summary:
```

	index	timestamp	width	height	frame_id	filename	angle	speed
0	2019-12-26 22:54:03.912922880	1577400843912922880	1024	64	NaN	center/1577400843912922880.jpg	0.0	0.0
1	2019-12-26 22:54:03.963078912	1577400843963078912	1024	64	NaN	center/1577400843963078912.jpg	0.0	0.0
2	2019-12-26 22:54:04.013053184	1577400844013053184	1024	64	NaN	center/1577400844013053184.jpg	0.0	0.0
3	2019-12-26 22:54:04.063207680	1577400844063207680	1024	64	NaN	center/1577400844063207680.jpg	0.0	0.0
4	2019-12-26 22:54:04.113175552	1577400844113175552	1024	64	NaN	center/1577400844113175552.jpg	0.0	0.0

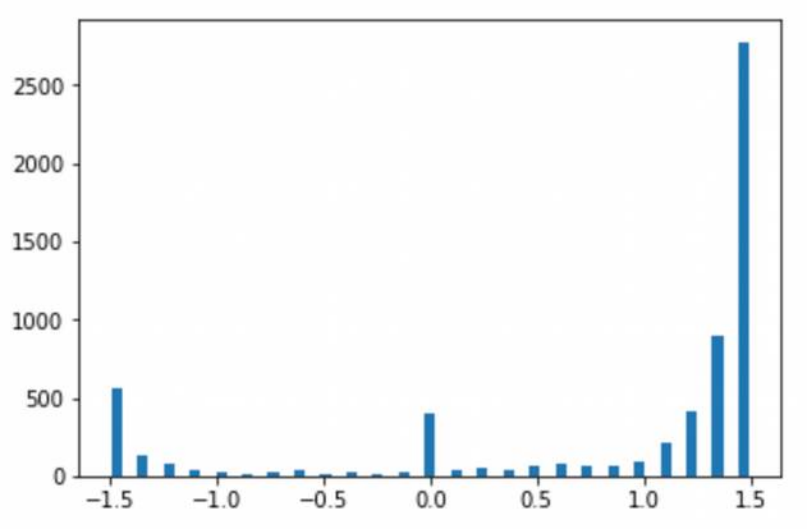
RC カーOuster ライダー学習データセット Pandas データフレーム

本データセットには 6000 以上のサンプルがあります。

次に、データセット自体の処理を行いたいです。これには、未使用の列、あらゆる NULL 値、0 スロットル値の欄の削除が含まれます。我々は、RC カーが動いている箇所のデータだけをモデルに学習させたいと考えます。

ヒストグラムを用いて、ステアリングコマンドの分布を視覚化できます。ウィジェットで

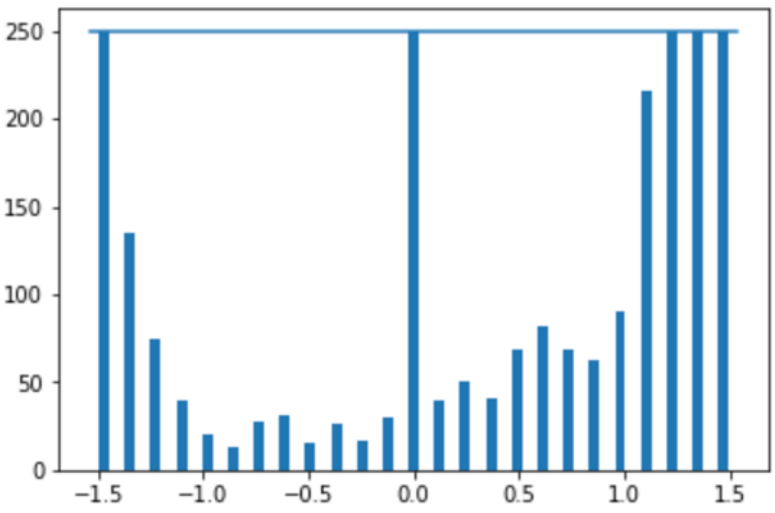
は、ユーザーがヒストグラムで使う瓶の数を選択できます。デフォルトの 25 瓶を使用すると以下のヒストグラムになります。



RC カーOuster ライダー学習データセットヒストグラム

データセットは不均一なステアリングコマンド分布を含んでいることが明白です。

この分布をノーマライズすることで、より均一な分布にして、ML モデルを様々な入力で学習させ、オーバーフィットさせないよう確保することが出来ます。hist チェックボックスが選択されると、データセットは削られて、瓶ごとに一定数の入力エントリ以上にはならないいようになります。この値は、samples_per_bin 変数を使って設定され、デフォルトでは 150 です。残ったエントリのヒストグラムはプロットされ、オリジナルのヒストグラムよりもずっと均一見えるものになります。



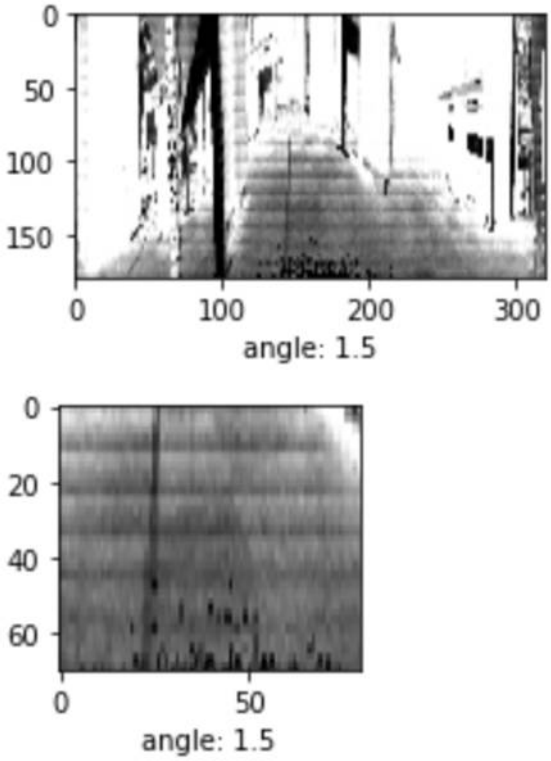
RC カーOuster ライダー学習データセット標準化ヒストグラム

画像処理

データセットがノーマライズされたら、画像を可視化してその品質の評価に進むことが出来ます。画像の処理には、幾つかのステップが存在します。

まず最初に、画像を元来の 1024×64 を 320×180 にリサイズします。これは、元来のアスペクト比を変更しますが、画像で失われる詳細は、今回のタスクでは重要ではありません。学習タスクと関係のない画像領域は、切り取って削除されます。これにより、モデルが環境に対してユニークな特徴を学習するのを防ぎます。

以下にオリジナルの画像、320×180 にリサイズされ、更に、切り取られたバージョンの画像を示します。切り取られた画像には、床とレーンは残っていますが、残りのオフィス環境は削除されています。



Ouster ライダー学習データ切り取られたライダー強度画像

これらの画像処理関数は、ML モデル定義および学習プロセスにおいて後続のステップとして組み込まれます。

学習および検証データセットの定義

データセットが処理され、レビューされたら、データを学習セットと検証セットに分ける必要があります。学習セットは、学習中モデルの重み付けを調整するのに使われます。検証セッ

トはモデルの重み付けには影響しませんが、モデルが学習データにオーバーフィット（過学習）していないかを調べるのに使われます。

データセットを分割する前に、データ拡張について調査します。我々のデータセットは比較的小さいので、データ拡張というテクニックを用いて、学習用データのわずかに変更されたバージョンを作成して、データセットを大幅に拡大します。これにより、最終モデルをより高精度に、そして、環境の変化に対してよりロバストにすることが出来ます。

Keras は、ImageDataGenerator クラスを有し、画像を操作するための幾つかの内蔵関数を提供します。これには、画像の垂直あるいは水平な移動、画像へのズーミング、輝度の調整が含まれます。投稿“Exploring Image Data Augmentation with Keras and Tensorflow”は、それぞれの例題付きで、そのオプションに関して良い概要を提供します。

大きなデータセットを取り扱う場合、Keras データジェネレータを使用して、読み込みデータを扱うのが有用です。これらは特に画像を含むデータセットの操作に有効です。この例題では、基本のデータジェネレータを定義します。将来的には、ImageDataGenerator クラスのような Keras 実装を使用できるようになるでしょう。

データジェネレータは、サンプルデータのデータセットと同様、バッチサイズのハイパーパラメータ値とデータを拡張するかどうか決めるフラグも取り込みます。

```
def generator(samples, batch_size=32, aug=0)
```

データジェネレータは、画像を読み込み、それをリサイズします。そして、その画像に対応するステアリング角コマンドを読み込みます。**Aug** フラグが有効化されると、拡張データジェネレータを用いて、基本的な画像拡張を実施します。拡張があっても無くても、画像と角度は、

別々のアレイに保存されます。これらはシャッフルされて、戻されます。

```
yield sklearn.utils.shuffle(X_train, y_train)
```

最後に、sklearn ライブラリから train_test_split 関数を使用し、データセットを学習用と検証用サンプルに分けます。

```
train_samples, validation_samples = train_test_split(samples, test_size=0.2)
```

データの 80%を学習用に、20%を検証用に使用します。これは、最終的には、約 2900 の学習サンプルと、700 の検証サンプルとなります。

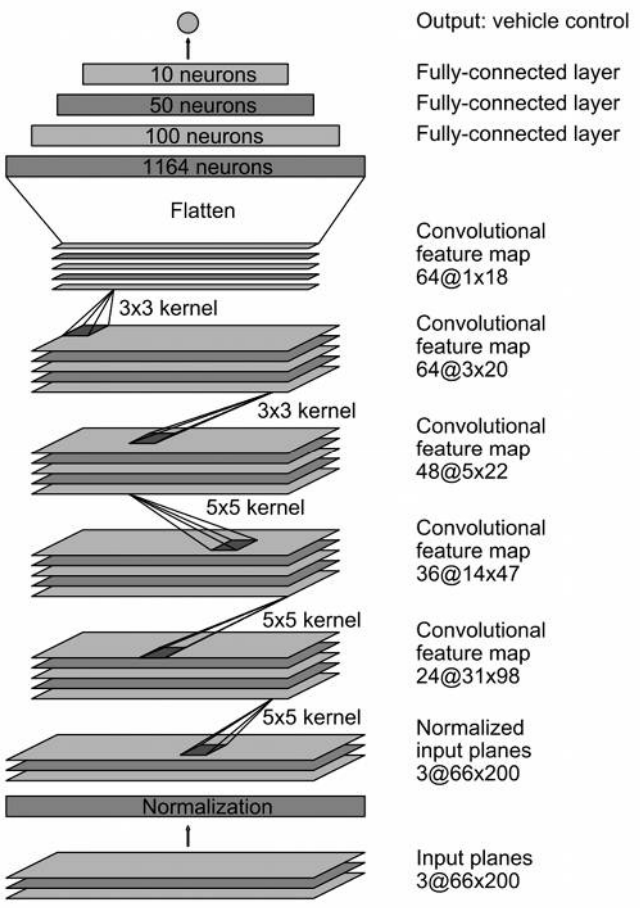
これらはデータセットジェネレータ関数に入力され、学習データジェネレータと検証データジェネレータとなります。

```
train_generator = generator(train_samples, batch_size=batch_size_value, aug=1)
validation_generator = generator(validation_samples, batch_size=batch_size_value,
aug=0)
```

モデルアーキテクチャと学習

前述したように、Nvidia により開発されたモデルを用いて、ライダー画像から直接、ステアリングコマンドに、生のピクセルをマッピングします。

ネットワークは 9 レイヤーを含み、1 つの正規化レイヤーと、5 つの畳み込みレイヤーと、3 つの完全に繋がったレイヤーから成ります。論文からのモデルを可視化したものを以下に示します。:



NVIDIA 末端間 ML モデル

このモデルは、Keras API を用いて、コンパイルされます。入力画像は、事前に見出された切り取り値を用いて切り整えられます。

```
model.add(Cropping2D(cropping=((height_min,bottom_crop), (width_min,right_crop)),
input_shape=(180,320,3)))
```

次の関数は、入ってくるデータのある基本的なプリプロセスを実行します。画像は正規化され、ピクセル値は、わずかな標準偏差で0にセンタリングされます。

model.add(Lambda(lambda x: (x / 255.0) - 0.5))

残りの行は、特定のモデルレイヤーを定義します。モデルが定義されたら、モデルはコンパイルされます。

model.compile(loss='mse', optimizer=Adam(lr=0.001), metrics=['mse','mae','mape','cosine'])

損失関数の平均自乗誤差が措定され、学習中に最小化されます。この測定基準は回帰タスクに対して有用です。Adam optimizer がオプティマイザ（最適化アルゴリズム）引数に対して措定されます。

これにより、701,559 の学習可能なパラメータ付きのモデルが得られます。

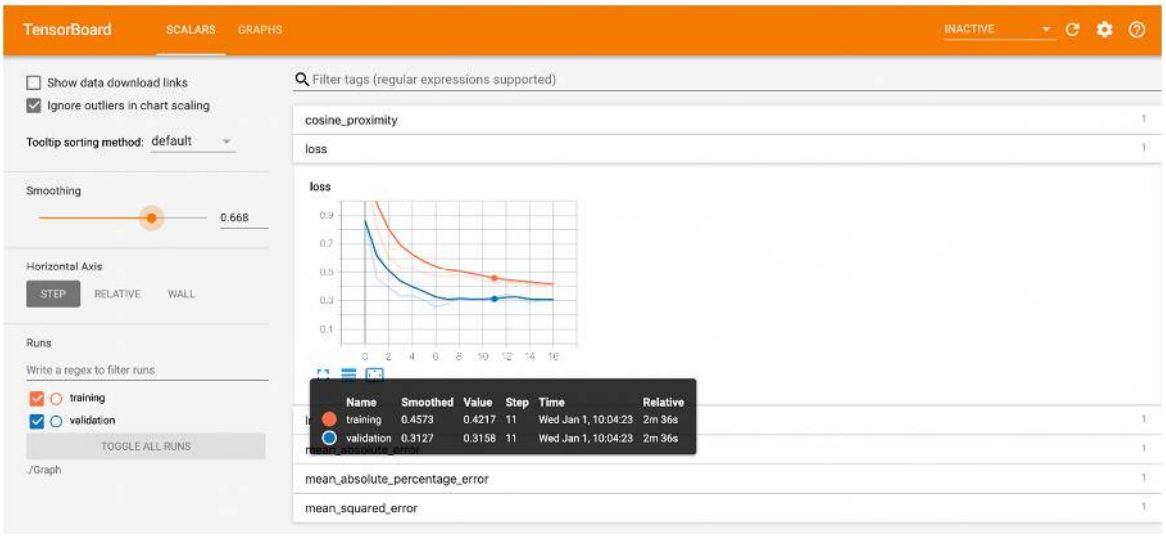
学習の前に、幾つかのコールバックも定義されます。コールバックとは、ユーティリティ関数のことで、モデル学習プロセス中に、起動され、呼ばれます。

最初に、checkpoints がセットアップされます。これにより、学習プロセス中、モデルの重みを保存し、システムの場合、モデルを再ロードできるようにします。これは、Colab 環境では有用です。なぜならば、一定のタイムアウト期間の後、インスタンスの接続を断つことが出来るからです。

checkpoint = ModelCheckpoint(filepath, monitor='val_loss', verbose=1,
save_best_only=True, mode='auto', period=1)

モデルのパフォーマンスが停滞すれば、EarlyStopping コールバックが使用されて、学習プロセスが終了されます。これは、オーバーフィッティング（過学習）の防止に役立ちます。ReduceLROnPlateau 関数も同様です。モデルのパフォーマンスが改善をやめた場合、Adam オプティマイザーの学習率は自動的に調整されます。

最後に、Tensorboard が設定されます。TensorBoard は、TensorFlow が提供する可視化ツールです。モデルグラフの可視化同様、損失や精度のような追跡実験指標を有効化できます。セルを実行した時、リンクが Tensorboard クラスへのアウトプットに貼られます。学習プロセス中の Tensorboard のイメージ例を以下に示します。



Tensorboard での Google ColabML モデル学習

モデルがコンパイルされ、コールバックのセットが定義されると、モデルの学習の準備が出来たことになります。学習の前に、幾つかのハイパーパラメータが定義されます。

最初に、学習及び検証用データのステップサイズが定義されます。

```
steps_per_epoch=(len(train_samples) / batch_size_value)
validation_steps=(len(validation_samples) / batch_size_value)
```

これは、1 つのエポックが終わり次のエポックが開始される前にデータジェネレータから産出されるステップ総数（サンプルのバッチ）を定義します。

次に、エポック数が定義されます。

```
n_epoch = 50
```

最後に、fit_generator 関数が使用され、モデル学習プロセスが開始されます。

```
history_object = model.fit_generator(
generator=train_generator,
steps_per_epoch=STEP_SIZE_TRAIN,
validation_data=validation_generator,
validation_steps=STEP_SIZE_VALID,
callbacks=callbacks_list,
use_multiprocessing=True,
epochs=n_epoch)
```

学習が開始されると、セルは各エポックの指標をアウトプットします。学習が完了すると、モデルは Colab インスタンスに保存することが出来、ローカルワークステーションにダウ

ンロード出来るようになります。

モデルの評価

モデルの学習プロセスは、アウトプットされるパフォーマンス指標を介して、モデルのパフォーマンスに関する洞察（insight）を提供します。しかし、モデルがどのように機能するかをより良く理解するためには、学習モデルを使用して、サンプルデータの幾つかに関して予測を行い、結果を評価します。

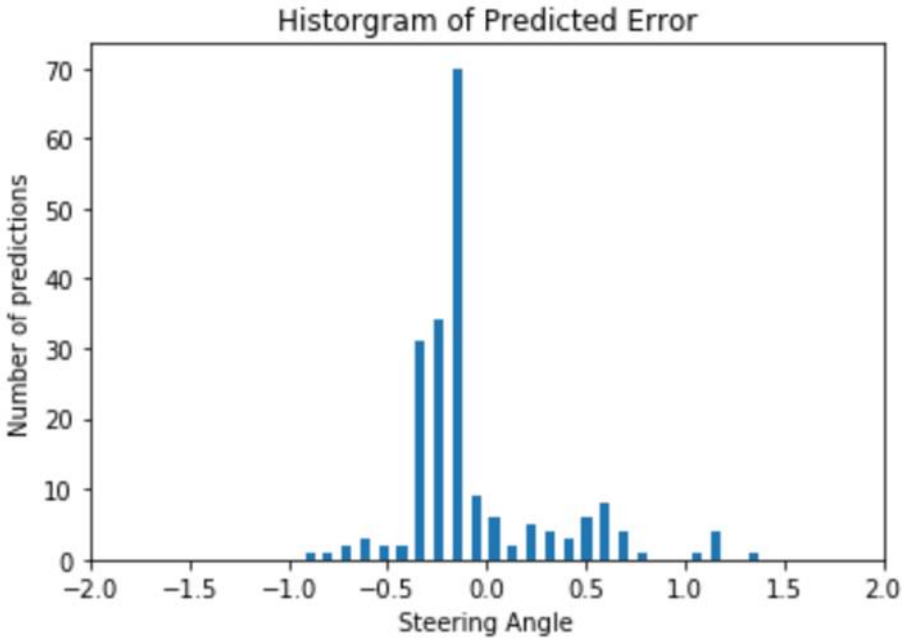
学習パフォーマンスは、学習プロセス中の学習及び検証用データセットの両方の損失関数をプロットすることにより可視化することが出来ます。

Ouster ライダーML モデル学習損失プロット

損失が初期に急激に降下しますが、学習プロセスが続いても改善は緩やかになります。学習用および検証用の両方のプロットは、同様のプロファイルを有します。これはモデルがデータにオーバーフィットしていないことを示します。

次に、モデルはデータのサブセットの各画像に対応するステアリングコマンド値を予測するために使用できます。予測値は、既知の真値と比較され、誤差が可視化されます。

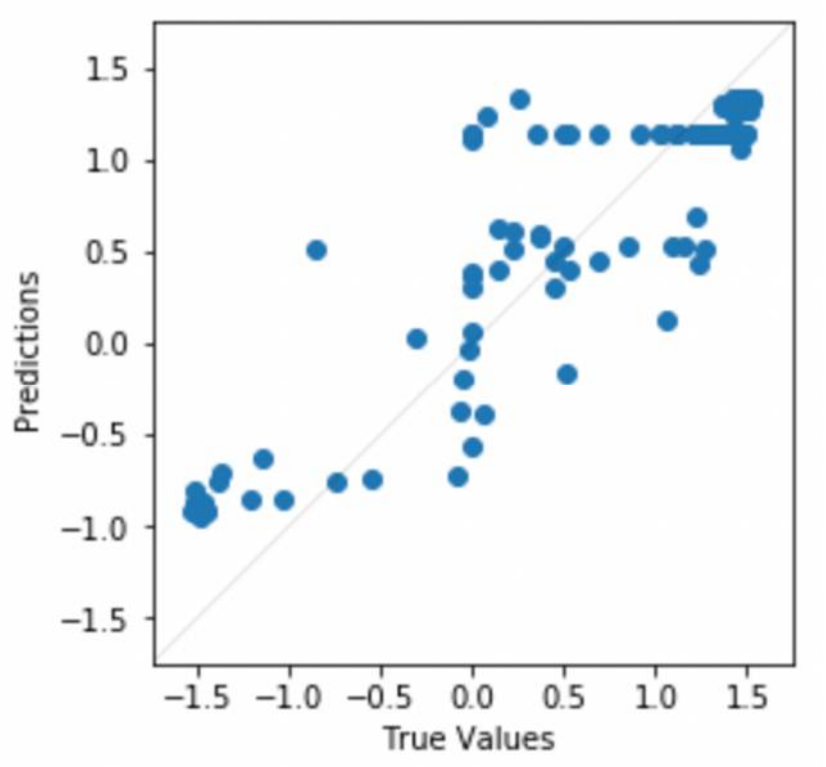
最初に、誤差はヒストグラムとして表示されます。



Ouster ライダーML モデル予測誤差

誤差は、良好である0近傍を中心にかなり均等に分布しているようにみえます。

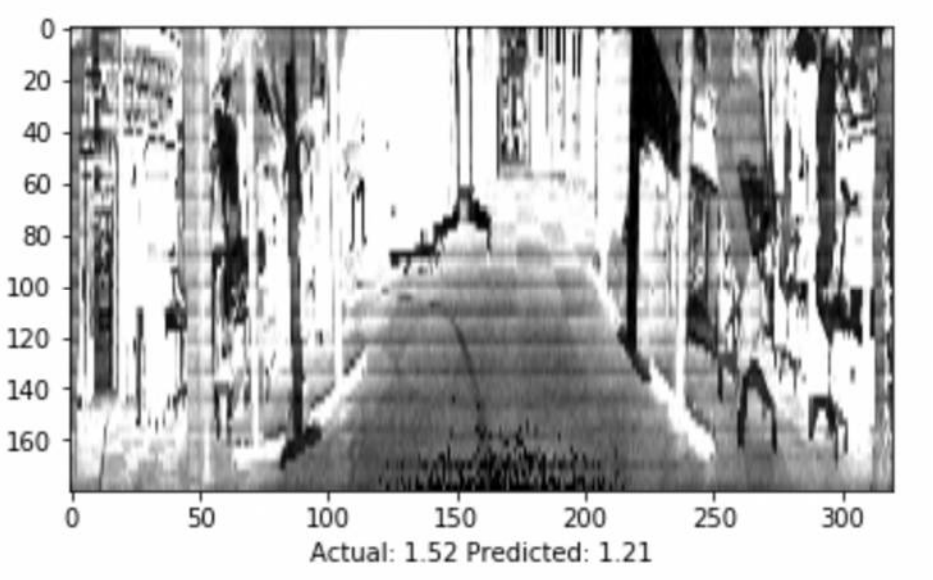
また、誤差は散布図としてもプロットできます。



Ouster ライダーML モデル予測誤差散布図

ここにおいて、モデルが角度０°のコマンドよりも、最小、最大ステアリングコマンド角のほうが、パフォーマンスは良いことが明白です。

モデルのパフォーマンスをおおよそ理解するために、個々の画像を閲覧することも可能です。下の図は、１つの画像に対する予測及び真のステアリング角度を示します。



Ouster ライダーML モデルライダー強度画像予測

最後に、

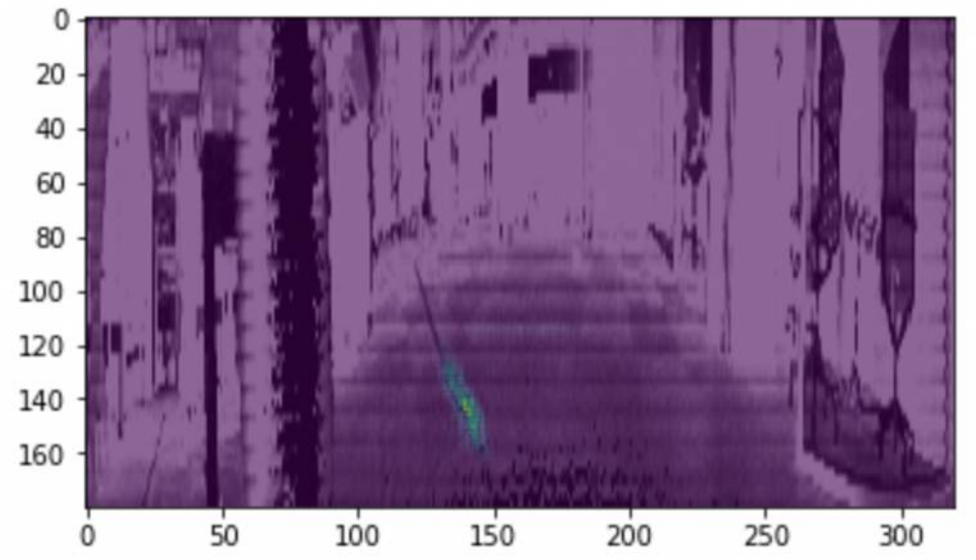
visualize_saliency 関数を使用してヒートマップを作成し、モデルのアウトプットに最も寄与する画像の領域を表示するができます。

サリエンシーマップは、論文“Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps”の中で提唱されました。それらは勾配に基づき、インプット画像の各ピクセルに関して、アウトプットの勾配を計算します。これは、インプット画像ピクセルの小さな変更に関連してアウトプットがどのように変化するかを示します。

```
grads = visualize_saliency(model,
layer_idx,
filter_indices=None,
seed_input=sample_image_mod,
grad_modifier='absolute',
backprop_modifier='guided')
```

以下の画像の例では、ネットワークがレーンのマーク箇所近傍のピクセルに対して最も感度

か高いことが明白です。



Ouster ライダーML モデルサリエンシー画像

モデルの導入と検証

モデルが学習されると、組み込みデバイス上に配置され、RC カー制御システムと統合されます。モデルは、ウェブカムから入力された画像に基づいて、リアルタイムで、ステアリングコマンドを予測します。

ROS ノードが作成され、カメラにより発行される画像トピックや、ML モデルから発行されるコントロールコマンドを受信（購読）します。drive_model.py ファイルを用いてこれが行われます。

model_control_node が作成され、/img_node/intensity_image トピックからライダー画像を受信（購読）し、コントロールコマンドを、/platform_control/cmd_vel トピックに発行します。

受信された各画像に対して、まず、180×320 画像へのリサイズが行われます。これは、モデルが要求するインプット画像サイズになります。

self.resized_image = cv2.resize(self.latestImage, (320,180))

本プロジェクトでは、ステアリング角度コマンドのみを評価しました。従って、スロットルは一定値に設定されています。

self.cmdvel.linear.x = 0.13

次に、モデルが入力画像を処理し、予測ステアリング角度を返します。

self.angle = float(model.predict(image_array[None, :, :, :], batch_size=1))

そして、予測アウトプットは、車両により許容される最大ステアリングインプット値に制限されます。

self.angle = -1.57 if self.angle < -1.57 else 1.57 if self.angle > 1.57 else self.angle

最後に、ステアリング角度コマンドと、お望みのスロットル値が、geometry_msgs/Twist メッセージとして発行されます。

self.cmdVel_publish(self.cmdvel)

システムはこの予測ループを連続的に完結させて、アップデートされたステアリング角度コマンドを車両に供給します。

システムを完全な形で起動するためには、rc_mlmodel_cam.launch を使用します。この起動ファイルは、rc_dbw_cam.launch に定義される共通 RC 車両インプットとコントロールノードを読み込むことに加えて、drive_model.pyROS ノードを読み込みます。

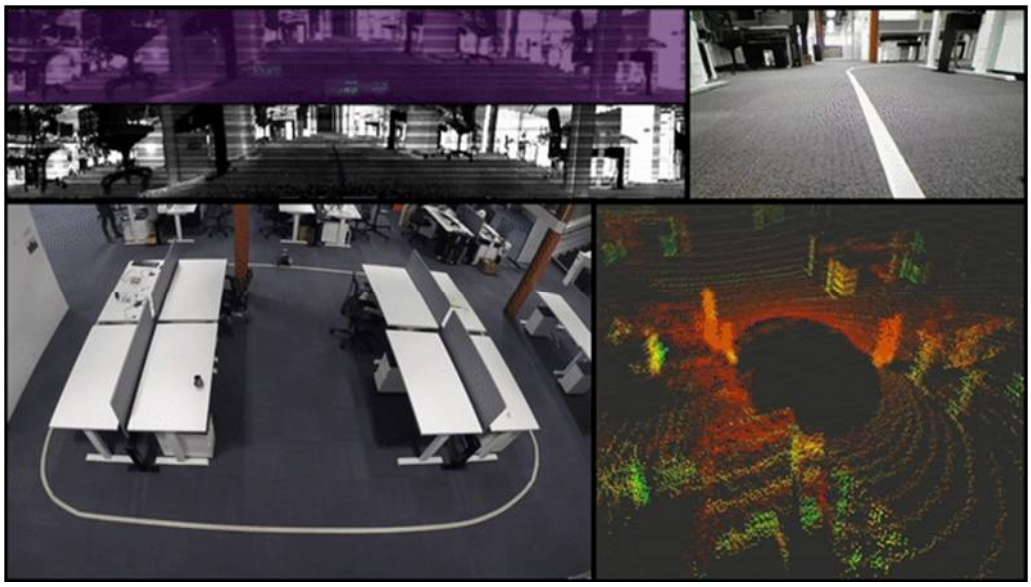
システムが起動中に、車両は２つの異なるインプットにより制御されます。Xbox コントローラーからの制御コマンドは、/joy_teleop/cmd_vel トピックに常に受信されています。更に、ML モデルは制御コマンドを、/platform_control/cmd_vel に対して発行しています。

yocs_cmd_vel_mux ノードは、コマンド速度インプットの多重化装置として機能します。幾つかのトピックから入力される cmd_vel メッセージ間の仲裁を行い、プライオリティに基づいて 1 度に 1 トピックがロボットに命令を出すことが出来るようにします。全てのトピックはそのプライオリティとタイムアウトと一緒にされ、実行時に再ロード可能な YAML ファイルを介して構成されます。これは、rover_cmd_vel_mux.yaml 構成ファイルに定義されます。

システムが適切に構成されると、RC 車両はトラック上に置かれ、システムは、rc_mlmodel_cam.launch 起動ファイルを使用して実行されます。

$ roslaunch behavior_cloning rc_mlmodel_cam.launch os1:=true ekf:=false

以下のビデオには、本プロセスで定義されたモデルを使用して、完全に自律的に周回を行う様子が描写されます。ウェブカムに記録された画像および、ステアリングコマンドのアウトプットに対して最も貢献された画像エリアを描写するサリエンシー画像の重ね合わせを同様にご覧になることが出来ます。



RC カー ライダーに基づく ML モデルの導入

本投稿は、本来 Wil のブログに掲載されたものです。:

https://www.wilselby.com/2020/02/rc-car-ml-model-development-with-ouster-os1-lidar/#more-9702