Race Car Control Lab

Race Car Localization using Convolutional Neural Network

Final Report

Mohamed AbouHussein
Mostafa Hussein

IMTEK
University of Freiburg
February 2018

Summary

This report is related to the work of implementing and comparing localization techniques for a race car using a convolutional neural network CNN to an already implemented vision system technique. The implementation is done using `tensorflow` and `keras` packages in python. The report is divided into 5 sections as follows: section 1 is an introduction about the CNN and its utilization in computer vision applications, section 2 reviews different architectures implemented and pre-processing performed, section 3 presents the evaluation metrics used to assess the performance as well as the the results and discussion, section 4 is a user manual for the code and section 5 presents the conclusion and future work and development areas.

1 Deep Learning through Neural Networks

This section gives an overview about the object detection task and how Convolutional Neural Networks (CNN) are able to perform this task, by giving an idea of their main functions and architectures.

1.1 Object Detection and Convolutional Neural Netowrk

Objects contained in images can be located and identified automatically. This is called object detection and is one of the basic problems of computer vision. As to be demonstrated, CNNs are currently the state-of-the-art solution for object detection. Object detection is a subfield of computer vision that is currently heavily based on machine learning. For the past decade, the field of machine learning has been dominated by so-called deep neural networks, which take advantage of improvements in computing power and data availability. A subtype of a neural network are the CNNs, which is well-suited for image-related tasks. The network is trained to extract different features. For example, corners, edges and colour differences. This is done across the image and. For object detection, the system has to both estimate the locations of probable objects and to classify these. A CNN is a neural network that typically contains several types of layers, one of which is a **convolutional layer**, as well as **pooling**, and **activation layers**.

Convolutional Layer

To understand what a CNN is, it is firstly important to understand how convolutions work. Imagine to have an image represented as a 5x5 matrix of values, then a 3x3 matrix was taken and slided around the image. At each position the 3x3 visits, a matrix multiplication occurs between the values of the 3x3 window and the values in the image that are currently being covered by the window. This results in a single number the represents all the values in that window of the image. An illustration of this convolutional operation is represented in figure 1.

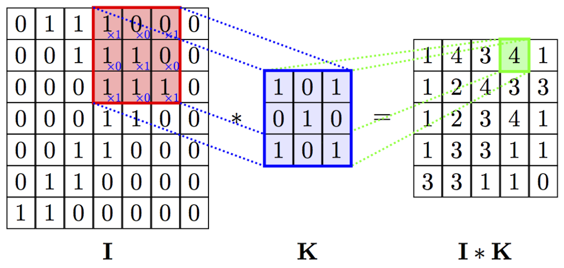


Figure 1: a two-dimensional convolutional operator as an operation of sliding the kernel matrix across the target image and recording elementwise products

The "window" that moves over the image is called a kernel. Kernels are typically square and 3x3 is a fairly common kernel size for small-ish images. The distance the window moves each time is called the stride. Additionally of note, images are sometimes padded with zeros around the perimeter when performing convolutions, which dampens the value of the convolutions around the edges of the image. The goal of a convolutional layer is filtering. As we move over an image we effective check for patterns in that section of the image. This works because of filters, stacks of weights represented as a vector, which are multiplied by the values outputed by the convolution. When training an image, these weights change, and so when it is time to evaluate an image, these weights return high values if it thinks it is seeing a pattern it has seen before. The combinations of high weights from various filters let the network predict the content of an image. This is why in CNN architecture diagrams, the convolution step is represented by a box, not by a rectangle; the third dimension represents the filters.

Pooling Layers

Pooling works very much like convoluting, where we take a kernel and move the kernel over the image, the only difference is the function that is applied to the kernel and the image window isn't linear.
Max pooling and Average pooling are the most common pooling functions used in CNNs. Max pooling takes the largest value from the window of the image currently covered by the kernel, while average pooling takes the average of all values in the window. An exmaple of such operations are illustrated below in figure 2.

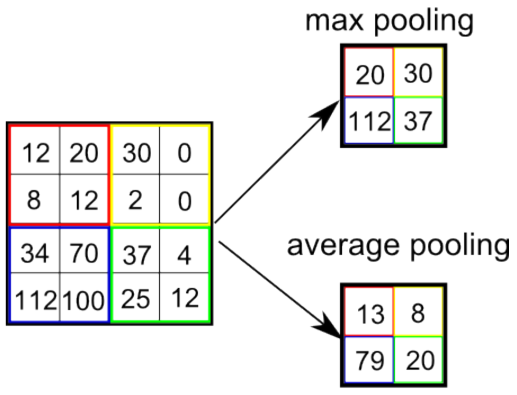
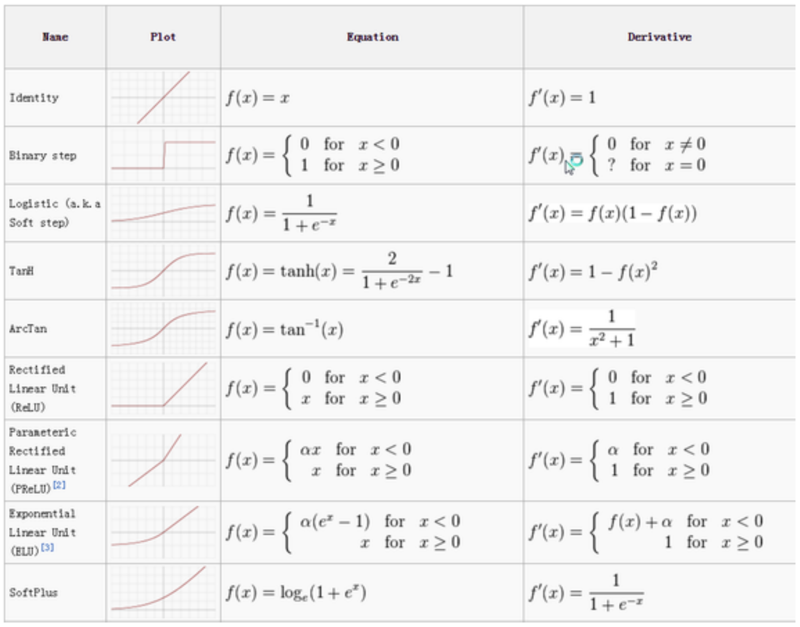


Figure 2: Pooling Operations

Activation Layers

Activation layers work exactly as in other neural networks, a value is passed through a function that squashes the value into a range. Here's a bunch of common ones in figure 3:



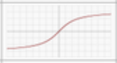
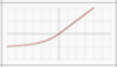
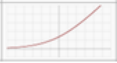
Name	Plot	Equation	Derivative
Identity		$f(x) = x$	$f'(x) = 1$
Binary step		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x \neq 0 \\ ? & \text{for } x = 0 \end{cases}$
Logistic (a.k.a Soft step)		$f(x) = \frac{1}{1+e^{-x}}$	$f'(x) = f(x)(1 - f(x))$
TanH		$f(x) = \tanh(x) = \frac{2}{1+e^{-2x}} - 1$	$f'(x) = 1 - f(x)^2$
ArcTan		$f(x) = \tan^{-1}(x)$	$f'(x) = \frac{1}{x^2+1}$
Rectified Linear Unit (ReLU)		$f(x) = \begin{cases} 0 & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} 0 & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Parameteric Rectified Linear Unit (PReLU)[2]		$f(x) = \begin{cases} \alpha x & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
Exponential Linear Unit (ELU)[3]		$f(x) = \begin{cases} \alpha(e^x - 1) & \text{for } x < 0 \\ x & \text{for } x \geq 0 \end{cases}$	$f'(x) = \begin{cases} f(x) + \alpha & \text{for } x < 0 \\ 1 & \text{for } x \geq 0 \end{cases}$
SoftPlus		$f(x) = \log_e(1 + e^x)$	$f'(x) = \frac{1}{1+e^{-x}}$

Figure 3: list of common activation functions

The most used activation function in CNNs is the relu (Rectified Linear Unit). There are a bunch of reasons that people like relus, but a big one is because they are really cheap to perform, if the number is negative: zero, else: the number. Being cheap makes it faster to train network.

2 Implemented Architectures and their Configurations

This section reviews the several network architectures implemented with its different configurations.

2.1 Initial Convolutional Neural Network Architecture

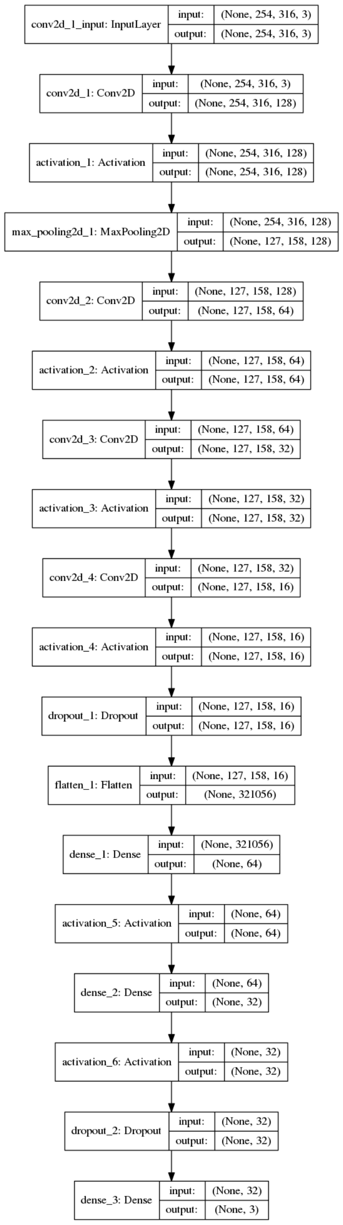

conv2d_1_input: InputLayer
input: (None, 254, 316, 3)
output: (None, 254, 316, 3)
conv2d_1: Conv2D
input: (None, 254, 316, 3)
output: (None, 254, 316, 128)
activation_1: Activation
input: (None, 254, 316, 128)
output: (None, 254, 316, 128)
max_pooling2d_1: MaxPooling2D
input: (None, 254, 316, 128)
output: (None, 127, 158, 128)
conv2d_2: Conv2D
input: (None, 127, 158, 128)
output: (None, 127, 158, 64)
activation_2: Activation
input: (None, 127, 158, 64)
output: (None, 127, 158, 64)
conv2d_3: Conv2D
input: (None, 127, 158, 64)
output: (None, 127, 158, 32)
activation_3: Activation
input: (None, 127, 158, 32)
output: (None, 127, 158, 32)
conv2d_4: Conv2D
input: (None, 127, 158, 32)
output: (None, 127, 158, 16)
activation_4: Activation
input: (None, 127, 158, 16)
output: (None, 127, 158, 16)
dropout_1: Dropout
input: (None, 127, 158, 16)
output: (None, 127, 158, 16)
flatten_1: Flatten
input: (None, 127, 158, 16)
output: (None, 321056)
dense_1: Dense
input: (None, 321056)
output: (None, 64)
activation_5: Activation
input: (None, 64)
output: (None, 64)
dense_2: Dense
input: (None, 64)
output: (None, 32)
activation_6: Activation
input: (None, 32)
output: (None, 32)
dropout_2: Dropout
input: (None, 32)
output: (None, 32)
dense_3: Dense
input: (None, 32)
output: (None, 3)

Figure 4: The initial CNN architecture implemented.

For the design of the CNN network, the procedure utilized is to create a single convolutional layer at a time. The layer parameters are randomly tried out from a range of values. The best estimated random values are the ones implemented. Afterwards, another layer is added and the same procedure is tried out stacking multiple layers. In addition, the hypermaraters is also set in a similar manner [1]. The previous procedure has resulted in the following: The first conv layer included 128 `Conv2D()` filters, 3x3 `kernel_size`. The activation function for the first layer is `relu`. Three other `Conv2D()` layers are stacked that has the same specs, kernel size and activation function but of size 64, 32 and 16 respectively. In addition, we add a `Dropout()` function to avoid data over-fitting of value 0.2. Finally, we flatten the filters through 3 `Dense()` layers of size 64, 32 and 3 (the output size) and in between activation function is `relu`. The architecture can be depicted in the figure 4 This architecture did not yield decent results as to be shown in section 3. Thereby, the next architecture is implemented.
The initial used optimization algorithm is the stochastic gradient decent. The conversion rate is slow. Thereby, later on the adam optimizer is utilized. The learning rate is of value 0.001 and the decay is of value 0.0005.

2.2 VGG-Net Convolutional Neural Network Architecture

The VGG-Net is a very deep CNN for image recognition [1]. The network architecture § proposed by the Visual Geometry Group (VGG) department in the Oxford University. The results were part of the ImageNet competition submission that the team achieved the first place in the localization competition track. Thereby, the hypothesis is that it would be quite suitable to our application. The VGG-Net architecture consists of eleven stacked filter layers. All the kernels used at each layer are of size 3x3. The first and the second layer includes 64 filters followed by a `MaxPooling2D()` of size 2x2. The third and the fourth layers includes 128 filters followed by a `MaxPooling2D()` of size 2x2. The fifth, sixth and seventh have have 256 filters followed by a `MaxPooling2D()` of size 2x2. The eighth, ninth and tenth layers consists of 512 filters followed by a `MaxPooling2D()` of size 2x2. The final layer includes 8 filters. All layers activation functions is the `relu` function. Lastly the features are flattened with a `Flatten()` and densed using a fully connected layer of size of 4096 with activation function `relu` and `Dropout()`. Finally, a `Dense()` layer of 3 is used which represents the output of the x position, y position and ψ orientation. The architecture is depicted in figure 5.
The VGG-Net used the adam optimizer as the gradient decent algorithm. The learning rate is of value 0.001 and a the decay is of value 0.0005.

[1] The implemented networks can be found in the RaceCar_Dataset CNN in the files cnn1.py to cnn8.py. The described network here is the one in the file cnn8.py

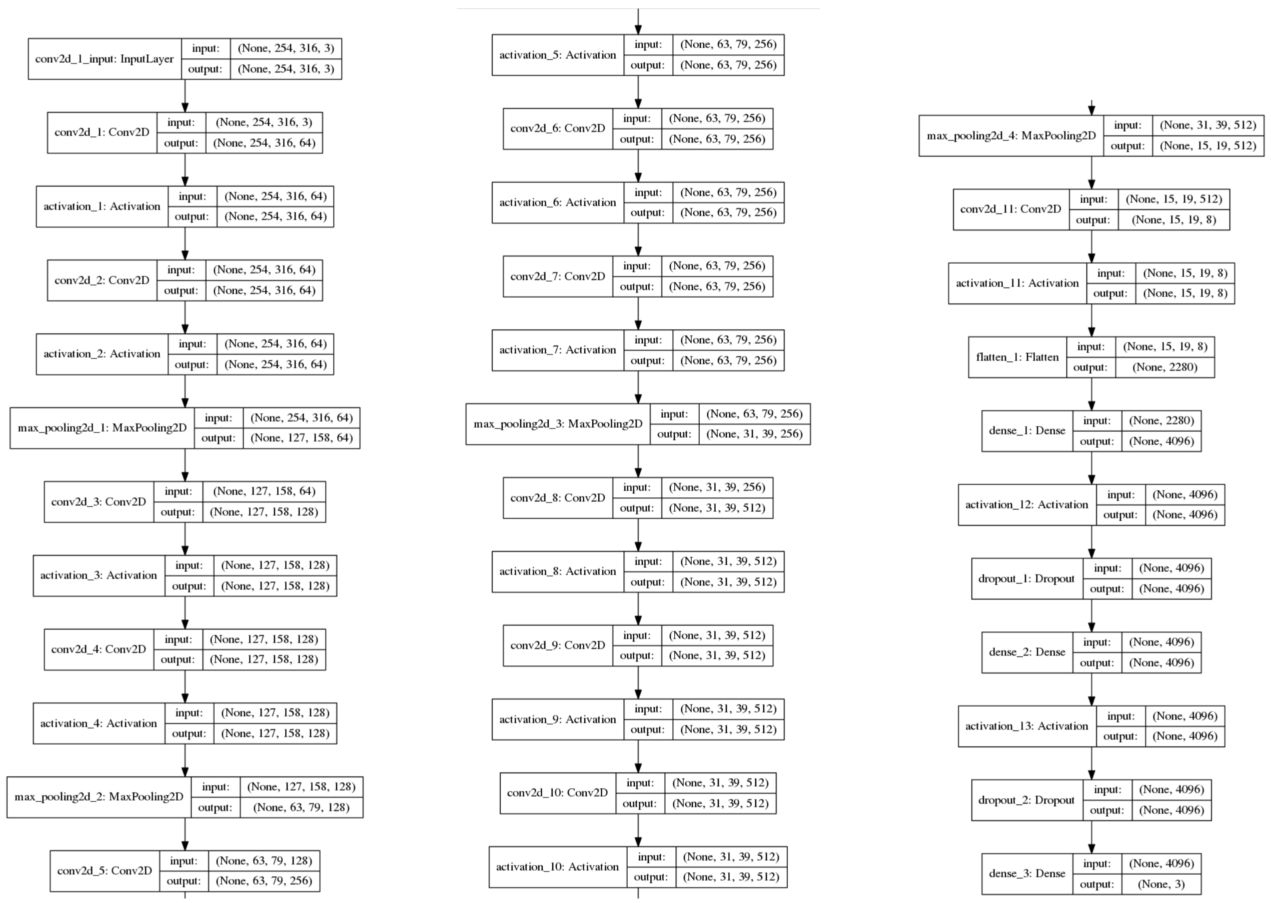

conv2d_1_input: InputLayer input: (None, 254, 316, 3) output: (None, 254, 316, 3)
conv2d_1: Conv2D input: (None, 254, 316, 3) output: (None, 254, 316, 64)
activation_1: Activation input: (None, 254, 316, 64) output: (None, 254, 316, 64)
conv2d_2: Conv2D input: (None, 254, 316, 64) output: (None, 254, 316, 64)
activation_2: Activation input: (None, 254, 316, 64) output: (None, 254, 316, 64)
max_pooling2d_1: MaxPooling2D input: (None, 254, 316, 64) output: (None, 127, 158, 64)
conv2d_3: Conv2D input: (None, 127, 158, 64) output: (None, 127, 158, 128)
activation_3: Activation input: (None, 127, 158, 128) output: (None, 127, 158, 128)
conv2d_4: Conv2D input: (None, 127, 158, 128) output: (None, 127, 158, 128)
activation_4: Activation input: (None, 127, 158, 128) output: (None, 127, 158, 128)
max_pooling2d_2: MaxPooling2D input: (None, 127, 158, 128) output: (None, 63, 79, 128)
conv2d_5: Conv2D input: (None, 63, 79, 128) output: (None, 63, 79, 256)
activation_5: Activation input: (None, 63, 79, 256) output: (None, 63, 79, 256)
conv2d_6: Conv2D input: (None, 63, 79, 256) output: (None, 63, 79, 256)
activation_6: Activation input: (None, 63, 79, 256) output: (None, 63, 79, 256)
conv2d_7: Conv2D input: (None, 63, 79, 256) output: (None, 63, 79, 256)
activation_7: Activation input: (None, 63, 79, 256) output: (None, 63, 79, 256)
max_pooling2d_3: MaxPooling2D input: (None, 63, 79, 256) output: (None, 31, 39, 256)
conv2d_8: Conv2D input: (None, 31, 39, 256) output: (None, 31, 39, 512)
activation_8: Activation input: (None, 31, 39, 512) output: (None, 31, 39, 512)
conv2d_9: Conv2D input: (None, 31, 39, 512) output: (None, 31, 39, 512)
activation_9: Activation input: (None, 31, 39, 512) output: (None, 31, 39, 512)
conv2d_10: Conv2D input: (None, 31, 39, 512) output: (None, 31, 39, 512)
activation_10: Activation input: (None, 31, 39, 512) output: (None, 31, 39, 512)
max_pooling2d_4: MaxPooling2D input: (None, 31, 39, 512) output: (None, 15, 19, 512)
conv2d_11: Conv2D input: (None, 15, 19, 512) output: (None, 15, 19, 8)
activation_11: Activation input: (None, 15, 19, 8) output: (None, 15, 19, 8)
flatten_1: Flatten input: (None, 15, 19, 8) output: (None, 2280)
dense_1: Dense input: (None, 2280) output: (None, 4096)
activation_12: Activation input: (None, 4096) output: (None, 4096)
dropout_1: Dropout input: (None, 4096) output: (None, 4096)
dense_2: Dense input: (None, 4096) output: (None, 4096)
activation_13: Activation input: (None, 4096) output: (None, 4096)
dropout_2: Dropout input: (None, 4096) output: (None, 4096)
dense_3: Dense input: (None, 4096) output: (None, 3)

Figure 5: Implemented VGG-Net architecture (each column of layers is stacked under the column to its left).

3 Results and Performance Evaluation

The following section reviews the evaluation metrics implemented and its results.

3.1 Evaluation Metrics

Loss

The loss function is used to guide the training process of a neural network. Mean squared error is commonly used in machine learning. MSE is the straight line between two points in Euclidian space. In neural network, back propagation algorithm is applied to iteratively minimize the MSE, so the network can learn from the dataset, next time when the network see a similar data, the inference result should be similar to the output of the training output. It is calculated by measuring the average of the squares of the errors or deviations, that is, the difference between the estimator and what is estimated.

Mean squared error: $\text{MSE} = \frac{1}{n}\sum_{i=1}^{i} e_i^2$

where e_i^2 is the error difference between the true label y_i and the predicted label $\tilde{y}_i$

Average Error per dimension

Another indication used for the model evaluation is calculating the average error per dimension for the testing data set. Basically,50 images are used as test data-set.They are passed to the trained model to see the average error that the model will output for each 3 dimensions; x, y and ψ orientation. The difference, is that MSE is applied during the training phase, and the other metric is calculated in the test phase.

3.2 Results

Initial Model

The results of the initial model described in section 2.1, showed a training loss of around 2.03, and a validation loss of around 1.787. The results are shown below in figure 6.

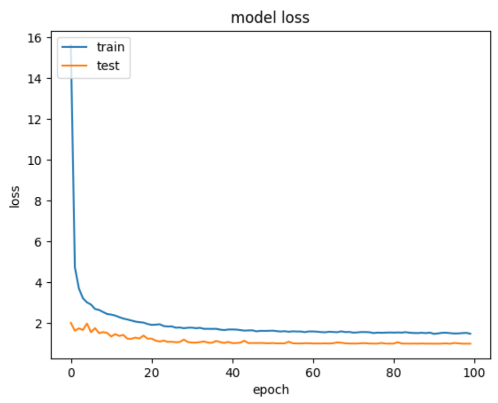


Figure 6: Loss for initial network (Training vs. Validation)

The average error per dimension for the tested images is 0.25 m in $x-dimension$, 0.31 m in $y-dimension$ and 1.26 in ψ. Clearly, the low performance of the proposed network makes it inapplicable to our application. Thereby, the network in the following subsection is tested for our application.

VGG-Net

The results of the VGG-Net described in section 2.2, showed a drop in the training loss from 3.748 to 0.0874, and a drop in the validation loss from 0.918 to 0.1323. The results are shown below in figure 7.

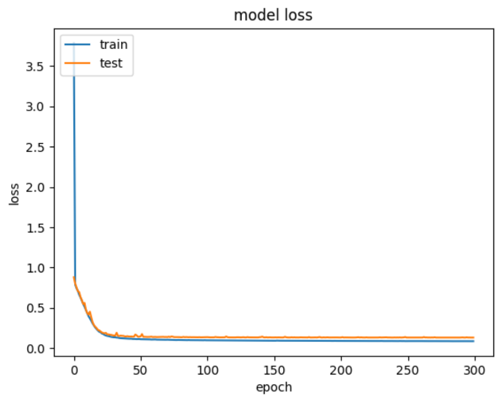


Figure 7: Loss for VGG-Net (Training vs. Validation)

The average error per dimension for the tested images is 0.0142 m in $x-dimension$, 0.0153 m in $y-dimension$ and 0.1701 in ψ. As the results show, this network clearly outperformed the priorly implemented architecture. Furthermore, the current results make the network quite applicable in real-life application. In addition, the average time for a single image to predicted is 0.0155 seconds, which is quite low. It is important to note that several `Dropout()` values were tested. The model which does not contain any `Dropout()` gave out the highest score. Our hypothesis is, that our application is training only on fixed background images with no variations, which is not the case for the localization example implemented by the VGG team.

4 User Manual

4.1 Setup Installation

The code depends on several packages to compile and run. Below is the list of packages and their installation prompt commands. This is only applicable for linux based OS. For other operating systems check the relevant installation method.

- pip package handler

```
        $ sudo apt-get pip
```

- opencv

```
        $ sudo pip install opencv-python
```

- numpy

```
        $ sudo pip install numpy
```

- imutils

```
        $ sudo pip install imutils
```

- tensorflow

```
    $ pip install tensorflow-gpu
```

- keras

```
    $ sudo pip install keras
```

- matplotlib

```
    $ psudo pip install matplotlib
```

- skimage

```
    $ sudo apt-get install python-skimage
```

4.2 Training and Testing

Within the project directory, the models reviewed in this report are saved in cnn_8.py and cnn_vggnet.py files respectively. The dataset is inside directory images_dataset. The test images are found inside images_test directory. The labels of the data are saved inside the csv file labels.csv. To run the network training, insert the following command in the linux terminal:

```
    $ python <cnn_model.py> images_dataset/ labels.csv
```

For testing on single image, run the following command:

```
    $ python test_cnn_model_single.py <image>
```

For running on multiple test images withing the same directory (it generates the average error in each dimension):

```
    $ python <test_cnn_model_batch.py> images_test/ labels_test.csv
```

5 Conclusion and Future Work

We propose a machine learning approach using convolutional neural network as a solution for the race car localization problem. The work in this project implemented two neural networks from which, one generated close to benchmark results. The model architecture used is based on the VGG team [1]. The average error is 0.0142, 0.0153 and 0.1701 in $x-dimension$, $y-dimension$ and ψ orientation respectively. The advantage of using this approach than using normal classical techniques for object detection and localization, is that neural networks are capable of detecting and localizing objects upon training without any preprocessing. Data Preprocessing requires humans to come up with feature extractors (or do feature engineering). However, Convnets learn the best possible features from data.

For future work, several ideas can be implemented for better generic results. Firstly, training on background noisy data can generalize the feature extraction to the race car location. Consequently, images containing unidentified objects, can still be successfully localized by the trained network. Secondly, the output of the network architecture can be modified to compensate for other objects localization in the image. For example, localizing several cars or even obstacles.

References

Literature

[1] Karen Simonyan and Andrew Zisserman. "Very deep convolutional networks for large-scale image recognition". In: *arXiv preprint arXiv:1409.1556* (2014) (cit. on pp. 6, 10).