Machine Learning Engineer Nanodegree

Capstone Project

Aditya Gupta

Definition

Project Overview

Driving in a new country is a situation people around the world face every day, whether they are on vacation or on a business trip. Often, different countries have different set of traffic signs and sometimes the same sign in different countries may have varying meanings. For a driver, this can often be frustrating and may even pose a threat to road safety. There is a need to provide drivers with a means to quickly and easily obtain the meaning of the traffic signs whenever they are uncertain what a sign means.

In this project, I create an android application that is capable of identifying 43 different types of traffic signs in Germany. The model is trained on the German Traffic Sign Recognition Benchmark Dataset and uses Deep Learning for identifying the traffic sign. What sets this application apart from others is that the recognition of traffic signs is done locally and does not require an internet connection.

Problem Statement

The aim is to create a Deep Neural Network for image recognition and running it on an Android Device locally. The following tasks need to be carried out:

- Downloading and analyzing the data
- Preprocessing the data
- Training a classifier
- Extracting and optimizing the model for Android

- Make the model run on Android
- Loading/Capturing an image for recognition

Metrics

Since we are performing a classification task, Accuracy is the chosen metric of evaluation. The selection of Accuracy as metric of evaluation can further be justified by the fact that the difference in samples for each class is within an order of magnitude of each other.

$$Accuracy = \frac{true\ positives + true\ negatives}{total\ number\ of\ samples}$$

Further on, the performance of our model will be compared against the top competitors of the German Traffic Sign Recognition Benchmark Competition.

Analysis

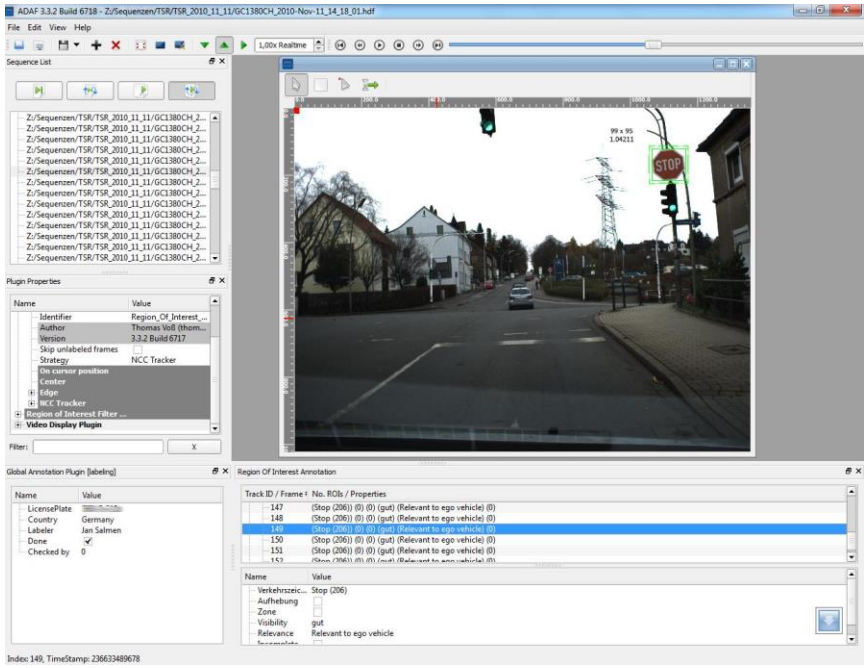
Data Exploration

The German Traffic Sign Recognition Benchmark (GTSRB) Dataset [1] has been divided for us into a training set and a testing set. The training set contains 39,209 training images in 43 classes while the test dataset contains 12,630 test images in 43 classes. Each image is a cropped photo of a traffic sign and each image contains only 1 traffic sign.

The images are stored in .ppm format and have varying sizes. The dataset is divided into a number of directories with 1 directory for each class. Each directory also contains a .CSV file for annotations. The images within each directory are divided into tracks and each track contains 30 images of one physical traffic sign.

The dataset has been created by extracting images of traffic signs from video of approximately 10 hours driving in Germany. The camera used was '*Prosilica GC 1380CH*' and the video was

recorded at a resolution of 1360x1024 pixels and 25 fps. Finally, the data collection, image extraction and annotation were done using the 'NISYS Advanced Development and Analysis Framework (ADAF)' [2] .



Advanced Development & Analysis Framework (ADAF) by Nisys GmbH

The annotation file contains the following information:

- Filename: Filename of corresponding image
- Width: Width of the image
- Height: Height of the image
- ROI.x1: X-coordinate of top-left corner of traffic sign bounding box
- ROI.y1: Y-coordinate of top-left corner of traffic sign bounding box
- ROI.x2: X-coordinate of bottom-right corner of traffic sign bounding box
- ROI.y2: Y-coordinate of bottom-right corner of traffic sign bounding box
- ClassId: Assigned class label

Sample of all classes [3]

The images are colored and of varying sizes. The sizes vary from 25 x 25 to 243 x 225 and contain random backgrounds behind the traffic signs.

	Width	Height
count	39209.000000	39209.000000
mean	50.835880	50.328930
std	24.306933	23.115423
min	25.000000	25.000000
25%	35.000000	35.000000
50%	43.000000	43.000000
75%	58.000000	58.000000
max	243.000000	225.000000

Training set statistics

Algorithms and Techniques

The classifier that I will be using is Deep Convolutional Neural Network. It is one of the most common and one of the most effective techniques when the data to be processed involves images. A convolution layer as compared to a fully connected layer, has fewer trainable parameters and therefore the size of the dataset required is smaller than that for a fully connected layer model, yet still requires a large number of samples. Our dataset is sufficiently large enough to train the model appropriately.

Deep Convolution Networks:

The problem with vanilla Multi-Layer Perceptron Deep Neural Networks when dealing with image data is that they take as input an input vector and hence when dealing with image data the image needs to be flattened to convert it into a vector form. This causes us to lose the important structural information that we have about the intensity values in the image. Moreover, in MLP model, each node in a layer is connected to every other node in the previous layer. This is often not needed as intensity values in one region of the image need not influence the intensity values in other parts of the image. This also leads to an extremely large number of parameters to train and leads to excessive training time.

A convolution layer addresses the above issue by considering only a small portion of the image at a time and also maintains the image structure. In this each perceptron has access to only a small area of the image that is under it. The activation values are obtained by sliding a small window over the image and recording the values obtained. The window essentially is the weights in a matrix form which is called a filter. The values are obtained by multiplying the weight values with the intensity values in the corresponding locations within the window. The values obtained using one filter form one activation map. Many such filters may be present in one convolution layer. Each layer can have the following parameters – number of filters, filter shape, stride (how much to move the filter in each step), padding (what to do if movement doesn't cover entire image), etc.

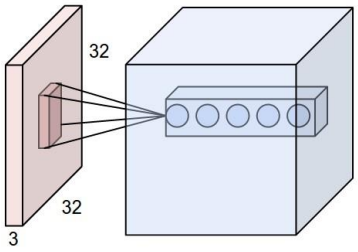


Fig: Convolution Layer [5]

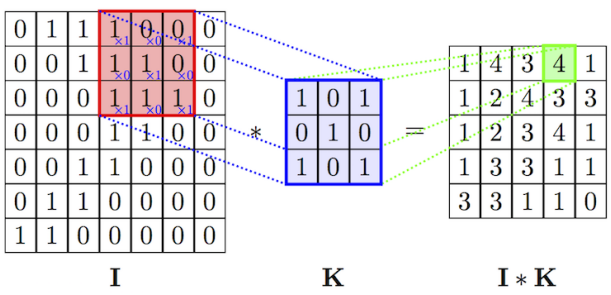


Fig: Activation map [6]

There can also be non-linear layers such as Max pooling layer which picks the maximum value within the window, Average pooling layer which takes the average of all values inside window etc.

The weights for convolution network are trained in the same way as other layers.

A deep convolution network consists of multiple convolution layers stacked on top of each other. The upper layers recognize low level features such as lines, colors, blobs etc. As we move deeper, the layers start detecting more complex shapes and features such as circles, faces, textures, etc. The layers convert an image which has lower breadth than width/height, to forms having more depth and lesser width/height. The deepest layers contain no special information but contain information like “is there a wheel?”, “is there a tail?”, etc. Hence, the last layer can be safely flattened and passed to a fully connected layer which can then learn relationships among them. This is how a Deep Convolutional network works.

The following are the parameters that could be optimized:

- Preprocessing technique
- Input size
- Number of training epochs
- Batch Size
- Optimizer
- Learning Rate
- Weight Decay
- Dropout probability
- Architecture of Neural Network
    - Number of layers
    - Size of each layer
    - Types of layers

During Training both training dataset as well as validation dataset are loaded and are fed into the network in the form of uniformly sized but randomly picked batches. In our case we use the Adam optimizer for training. On the training machine, we use a GPU to significantly improve training time since GPUs provide a lot of computation power. A Nvidia Tesla K80 with 11GB of VRAM was used on the training machine. However, on the mobile device, tensorflow does not support GPU utilization on mobile devices at the time of writing and hence the inference on mobile device was done using only the CPU. This is okay since inference is much less computation and memory intensive that training a model.

Benchmark

Traffic signs in general are very simple to identify for human being if they know what each traffic sign looks like. This task is easier for humans than, say for example, Handwritten digit recognition. This is because traffic signs have a very uniform structure compared to handwritten digits which can vary a lot in their structure depending on who is writing the digits. Hence, it felt appropriate to benchmark our model against scores obtained by manual labelling done by humans.

The accuracy score obtained by human labelling is 98.4% as given on [4] and 96.14% accuracy is obtained by the 4th place team COAR which used the method of Random Forests to perform the classification.

Keeping in mind the memory and processing constraint on mobile devices, the model finally selected should have low memory use during inference and low processing cost. Therefore, it is not possible to work with extremely large images and large number of layers.

Methodology

Data Preprocessing

All of the data preprocessing work is done in the preprocessing notebook. This preprocessing is done on the dataset which is already preprocessed to some extent. The dataset contains images of traffic signs that have already been cropped out of the original picture using certain segmentation techniques.

The following steps have been performed in the preprocessing notebook:

- Analyzing the provided .csv files to get information regarding number of images, information regarding the size of images, etc. This information is used to select an appropriate input size of images.
- Using the information provided in the annotations file, the images are further cropped to the exact bounding box of the traffic sign in the picture.
- Images are resized to 32 x 32.
- Images are stored in a single vector.
- Images in the dataset are then mean normalized.
- Image labels are converted to one-hot encoding form.

- The images are then shuffled so as to randomize the occurrence of images of the various classes in the dataset.
- The image and label vectors are saved as a .pkl file.

The steps above are those that are carried out on the training and testing datasets. On the mobile device, when the user selects a picture, he/she is asked to first crop out the background and select a rectangle that contains the symbol of the traffic sign. This is done because the model will learn to recognize the important structure of the traffic sign, but since we cannot have possible trained on all possible backgrounds, we exclude the background portion of the image. After that, the cropped image is resized and normalized and then fed into the model for inference.

Initially the training dataset is stored as a single vector but while training the model, the data is randomly split into the training and validation dataset.

Implementation

Since our model will be trained and used on two separate types of devices, it is required that the implementation be carried out in 2 phases.

PHASE I

This phase is the preprocessing, creation and training of the classifier.

First the data is preprocessed and saved so that it can be used for training the model. Then, the architecture of the model is considered keeping in mind the constraints that are imposed by the memory and processing capabilities of the mobile device on which inference is carried out and also the model architecture is limited by computing power and memory available for training the classifier. The memory constraint on GPU is especially important during the training phase as it directly affects the batch size and puts an upper bound on the number of samples pet batch, and since the batch size is limited, the accuracy is also reduced marginally depending on the size of the batch compared to the overall size of the dataset.

The training and evaluation is done in the train_network notebook.

The following steps are carried out in the train_network notebook:

1. Loading the training images and the labels.
2. Splitting the training data into training and validation set.
3. Defining the network architecture.
4. Setting training parameters such as number of epochs, batch size, learning rate and keep probability for dropout.
5. Specifying the optimizer, the loss function to be used and the metric for evaluation.
6. Train the network and track the accuracy and loss.
7. If the accuracy is not satisfactory, go back to refining the network architecture.
8. Once the accuracy is satisfactory, the model is frozen and saved as a tensorflow graph.

The basic model considered has the following architecture:

- A flattening layer
- Fully connected layer with 1024 neurons (1)
- Fully connected layer with 1024 neurons (2)
- Fully connected layer with 1024 neurons (3)
- Fully connected output layer with 43 neurons

This basic model is able to achieve an accuracy of 87%.

The technically detailed coding process for preprocessing is as follows:

1) Ipython notebooks (Python 3.5) were set up for step by step implementation.
2) First, the Pandas library was used to load the .CSV file containing the meta data about the images. This was then used to get a summary of the images sizes and other statistics. This also provided information regarding the traffic sign bounding boxes.
3) Then, since the images were stored in .ppm format, the Python Imaging Library (PIL) was used to load the images one at a time, crop them to their bounding boxes, and resize them to 32x32 size. Following this, the image was converted to a numpy array which allows for easy and efficient manipulation of the data. Then, the colored image is converted to grayscale using the OpenCV library. Finally, the image is converted from integer format to floating point format and added to the list of images along with its label.

4) After this the image dataset's mean and average are calculated and saved as a text file using numpy functions.
5) The images in the dataset are then normalized using the statistics calculated in the previous step. The labels are also converted to one-hot encoded form. This also done using numpy.
6) Using the sklearn library we then shuffle the dataset and labels together.
7) Finally, the dataset and the labels are saved as pickle files which is a python library to store python objects to disk so that they may be read into memory again at any time.
8) Steps 2 to 7 are repeated for the testing data.

The technically detailed coding process for training is as follows:

1) First the image data is loaded into memory from the pickle files.
2) Using keras library we use the Sequential model to specify the basic model architecture, optimizer, metrics and loss function and train the model on the training data and evaluate it on the testing set.
3) Similarly, we proceed with the specification and training of the final model. At this stage we try many different architectures and hyperparameters and arrive at the final model which performs the best.
4) Finally, we use the tools available in the tensorflow library to freeze, optimize and save the model to disk so that it can be used in the android application.

One issue that occurred during this process is the appropriate selection of preprocessing steps and image format. I began by selecting image size of 64x64, using colored images, and only performing normalization to bring values of images between 0 and 1. This led to small models not being able to generalize properly to the images and led to bigger models which seemed highly inefficient for mobile devices. It took some experimentation to determine the final sequence of preprocessing steps as mentioned above.

PHASE II

This phase involves the creation of the android application that will perform inference from the trained model.

First, the android app is created and its basic user interface is completed. This involves providing the facility to user to capture images or used saved images and then allow the user to crop the image selected to the appropriate bounding box of the traffic sign.

After setting up the user interface, the tensorflow libraries are included in the android application and a utility to perform the inference from the model on a given input is set up.

The cropped image is then preprocessed to bring it to the appropriate form and then fed into the model for the inference to be performed. The output is then to be shown to the user. One thing to be noted here is that we require that the image be preprocessed in the same way as the training data was processed. This required that OpenCV library be added to the android application for preprocessing the input image in the required way.

We used the [7] library to allow the user to select image and then crop it.

One major issue that was encountered here was the preprocessing of image. In the earlier stages, it was not felt necessary to use OpenCV and I decided to use the native android java functions to resize the image and convert to grayscale. However, this created issues as the techniques used in these functions were different from the ones that our python functions utilized. This caused the android device to not even correctly identify samples from the training dataset. Specifically, this issue was caused by the bad quality of resizing by the native functions. Even the formula used for grayscale conversion differed but this did not cause a major issue. To tackle this issue, I setup the OpenCV library on android to perform the required preprocessing.

Refinement

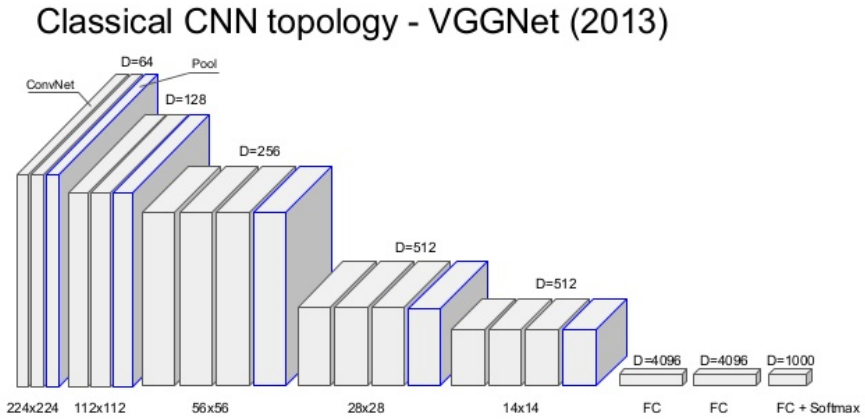
The basic model has very poor performance and requires a lot of improvement. As mentioned in the benchmark section, the accuracy obtained by human labelling is 98.4% and hence the aim was to improve the model to have at least comparable accuracy to human labelling if not better. As a result, a variety of combinations of convolution, fully connected and pooling layers was experimented with and even parameters like epochs, batch size, etc. were tuned to improve performance.

Due to the memory constraint we could not include an extremely large number of layers and neither could we work with large inputs and hence the output size of each section the network was gradually reduced via pooling and varying the strides of the convolution layers.

Finally, a modification of the famous VGGNet [5] architecture was implemented and was found to have the best accuracy. The continuous reduction in output size of each section allowed the model to be relatively small and capable of running on mobile devices while providing sufficient room for learning the structure of the data.

The majority of the refinement process revolved around finding a good set of parameters for the convolution layers. I tried varying the depth of the model as well as the number of filters in each layer and the final model as given in the next section outperformed all others by approximately 5%. All other models tested gave about 87% to 91% accuracy. I then proceeded to further improve the model by adding the dropout layer and tested the model with different values of dropout and learning rates and obtained the following results:

Learning Rate	Keep Probability	Accuracy
0.003	0.5	96.2%
0.003	0.65	96.5%
0.003	0.8	97.6%
0.001	0.8	97.3%
0.005	0.8	95.0%



Results

Model Evaluation and Validation

The model finally chosen is a modification of VGGNet as it performed the best on the dataset.

The architecture is as follows (in order):

- 32*32*1 Input image (Grayscale)
- 32 3*3 Convolution layers with unit stride (1)
- 32 3*3 Convolution layers with unit stride (2)
- Max Pooling layer with stride 2
- 64 3*3 Convolution layers with unit stride (1)
- 64 3*3 Convolution layers with unit stride (2)
- Max Pooling layer with stride 2
- A flattening layer
- Fully connected layer with 512 neurons (1)
- A dropout layer
- Fully connected output layer with 43 neurons

The model parameters were tuned to the following values:

- Learning rate = 0.002
- Batch size = 1024
- Keep Probability for dropout = 0.8
- Epochs = 20

The model files generated after training and freezing the graph are about 15 megabytes in size and this is compatible with the memory available on modern smartphones.

The training is performed for 20 epochs, and achieved a peak accuracy of 99.5 % on the training set.

After the training, the model was tested on a test dataset and achieved an accuracy of 97.67% which beats the Random Forest classifier's accuracy (96.1%) and is comparable to the accuracy of human labelling (98.4%).

Justification

Since we were able to achieve a training accuracy of 99.5% and a test accuracy of about 97.67%, it can be safely concluded that the model was able to successfully capture the required structure in the images. Hence if given the cropped images, the performance of the model would be similar to that of the human labeled images.

Moreover, using Stratified K-fold Cross validation over 10 folds, the following results were obtained:

Fold	1	2	3	4	5	6	7	8	9	10	mean	stddev
Accuracy	99.5949	99.7878	99.6335	99.5949	99.537	99.6335	99.7878	99.7878	99.7299	99.5177	99.6605	0.0995773

The constant good performance of the model is indicative of it's robustness and fitness to the task at hand.

The application was tested on a OnePlus 3T android device, and the application was able to correctly identify all the sign that were provided as input by the user.

It is to be noted that the model can only identify German traffic signs as the dataset contained only those traffic signs. Further improvement has been discussed in the Improvement section.

Conclusion

Free Form Visualization

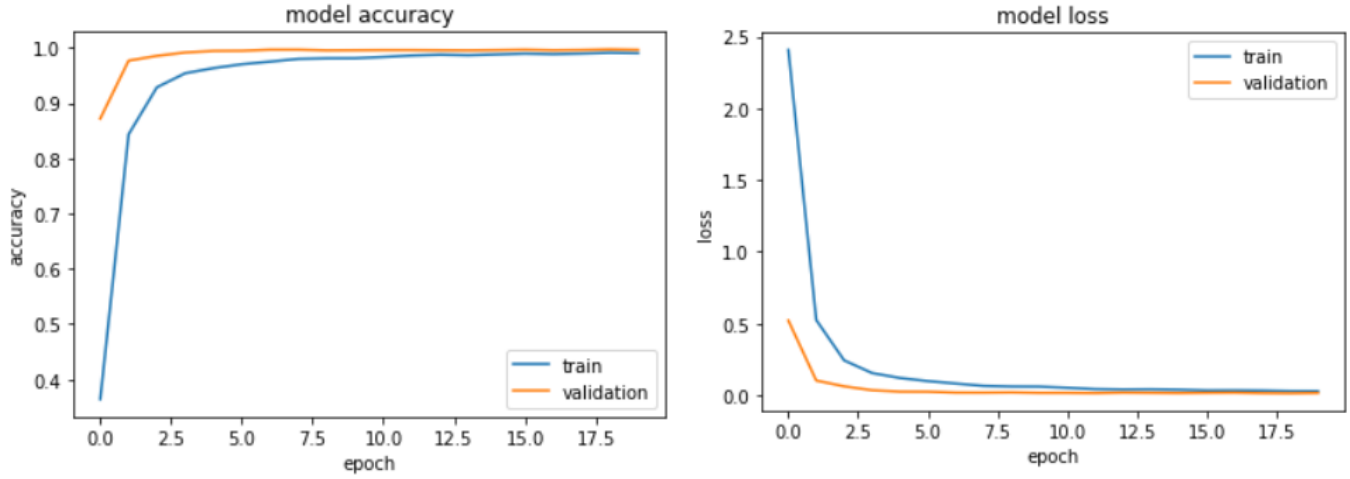


Figure: Images after preprocessing

Reflection

The entire process carried out in the completion of the project can be summarized as follows:

➔ A relevant problem statement and public dataset relevant to the problem was found.

- ➔ The data was analyzed so as to determine the appropriate input format for the model.
- ➔ The data is then preprocessed and consolidated into a format that can be easily used in the later parts of the project.
- ➔ A 3rd party benchmark and a basic model is determined to form a baseline for measuring and comparing the performance of the final model that design and train.
- ➔ The model was then trained on the data and the appropriate hyperparameters were fine tuned following an iterative experimental process.
- ➔ The trained model and other required information was saved so that it can be used on the mobile device.
- ➔ Tensorflow classifier was set up on the android device using the demo code available online.
- ➔ The application was developed to provide the required functionality.
- ➔ OpenCV was set up on the android device for preprocessing of the input data into the format that is required by the model we have trained.
- ➔ The model is tested on the mobile app using images downloaded from the internet.

An interesting thing that came up during the creation of the android application (at the time of writing) was the lack of documentation regarding setting up of and running tensorflow on android. There were few examples with little to no explanation of their working. This was one of the biggest challenge faced in this project since we were not just trying to run an ordinary MLP model but were looking to perform convolutions. Hence, adapting the tensorflow example code to work with image data was one of the most interesting parts of the project as it was something that had not been done much and required exploring and understanding the tensorflow android API.

Improvement and Issues

One of the most obvious improvement that is possible is to included more classes so as to cover the widest range of traffic signs so that the application is not restricted to being used in one country only. The challenge in this is to keep the model size small enough so that it can be easily and efficiently run on a mobile device.

Another improvement possible is to enable the application to automatically identify traffic signs present in a live feed of video.

One issue that becomes apparent when using the app is that it is not able to identify traffic signs correctly when the image has been skewed heavily or the viewing angle is very sharp. This

can be considered as a preprocessing problem and can be tackled by implementing a way to apply perspective transform to the input image.

References

[1] [Online]. Available: http://benchmark.ini.rub.de/?section=gtsrb&subsection=dataset.

[2] [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608012000457?via%3Dihub#fn000005.

[3] [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0893608012000457?via%3Dihub.

[4] [Online]. Available: http://benchmark.ini.rub.de/?section=gtsrb&subsection=results#.

[5] [Online]. Available: http://cs231n.github.io/convolutional-networks/

[6] [Online]. Available: https://cambridgespark.com/content/tutorials/convolutional-neural-networks-with-keras/index.html

[7] [Online]. Available: https://github.com/ArthurHub/Android-Image-Cropper