A picture containing drawing

Description automatically generated

Project report

Université Antonine

Artificial Intelligence

Al Sayyed Hadi Ibrahim

# Abstract

The purpose of this project is to attempt to classify the state of a driver, either drunk of sober, with the possibility to add more states in the future, based on a dashcam. Data was gathered manually for this purpose, and many models were created and tested, until finally being able to somewhat accurately make predictions of the state of the driver, with a footage that is either pre-recorded or live.

# Data Gathering

Gathering data was a challenge, and it was done on many steps. First, research was done to find a complete dataset of first-person driving videos (dashcam), but the search bore no results. Hence, the data was carefully and manually handpicked from several different videos on YouTube and Shutterstock (the original videos are located under the Sources folder), and then clipped by hand to focus on the most important parts for our AI to learn from.

But that was not it, since most of the videos had watermarks on them. While it’s a crime by itself, it has other dangerous effects on our program. For example, if most of our drunk labeled data had watermarks, the AI might infer that a watermark => drunk driver, which is totally unrelated and wrong. Therefore, the decision was clear, the clips need to be edited to remove the watermark. Unfortunately, all decent video editing software are not free, and with a lack in bandwidth, creativity had to come in place. The next paragraph will explain how and why the scripts “convert.py” and “script.py” were written.

Microsoft PowerPoint has the option to crop and resize, rescale videos. But the process must be done manually, and it is quite time consuming. For each video clip, we would need to create a PowerPoint file, insert the video, crop it, re-scale the width and the height to preserve the aspect ratio consistency and export as MP4. While it’s a straightforward process, it can be automated and improved. The “script.py” creates a PowerPoint template, and then places an unedited video inside the “media” directory (Where ppt saves the media for presentations), and then converts the whole directory to a .pptx format file. It loops through every video and does the same job. This way, we have saved a lot of time creating empty ppt files and getting them ready. Next, we crop and resize quickly each video, and finally, we run the “convert.py” script, which converts every .pptx file with a video to a .mp4 format file.

Data augmentation was used to make the most out of the relatively few clips that we have.

After getting the clips, we need to find a way to parse it to numerical values. Luckily, TensorFlow’s Keras Video Generators can handle this job for us. This source was used for more information : <https://medium.com/smileinnovation/training-neural-network-with-image-sequence-an-example-with-video-as-input-c3407f7a0b0f>.  
Then, we can label the videos by separating them in 2 different and labeled directories, drunk and normal. For each video in each directory, the video generator will take the video, and classify it to an array of 5 numerical values: {N,F,W,H,C}, and then distributing these input and labels to two sets, the training set and the prediction set. With out datasets ready, we’re now eligible to go to the next step.

A picture containing photo, different, showing, window

Description automatically generated

# Modeling

Now that we have our datasets ready, it’s time to create our models for our RNN. First, our data must go through a convolutional neural network for us to be able to use it in our own RNN. We have many options for this phase, but we will experiment with 2: building our own simple convNet (see the build\_convnet method) and using a pre-trained convnet from Keras applications called MobileNet, with a little bit of tweaking to work with our data.

Next, we will build our own recurrent neural network, that will take as first layer our CNN + time distributed sequence of frames (2D convNet + TimeDistributed). Then, we add a GRU layer to optimize our recurrent neural network. We will also try using LSTM once everything is setup. After that, we can concentrate on our neural network. We will try to use 4 hidden layers with “relu” as activation function, and use “softmax” for the output layer. Because our dataset is very limited, a dropout value of 0.5 was chosen, to try and prevent overfitting. “Adam” optimizer was used, and the categorical cross-entropy as a loss function, in case we want to add more categories to our model in the future. Finally, we set our training epochs to 60 and made callbacks to the model available by saving the weights in a file.

# Testing and optimization

Let’s test the results with the previous parameters. Note that we are still using the standard video generator at this point, with a simple convnet that we built.  
  
A screenshot of a cell phone

Description automatically generated

Great start. Let’s try using Keras’ MobileNet convolution neural network now.   
  
A close up of a map

Description automatically generated

Our performance has greatly worsened, it was not a great idea for our model as it led it to underfit. We’ll stick to using our simple built CNN for now at least.

Let’s try to optimize it further. Instead of using a simple VideoFrameGenerator, we’ll use a SlidingFrameGenerator, which will look for many available sequences in each video, creating a lot more data for us to learn from, with now over 650 samples instead of 90.

A black and silver text on a white background

Description automatically generated

After 8hs of processing on a GeForce GTX 1050ti and an i7 8750-U, with TensorFlow GPU enabled, using the latest CUDA and CUDNN versions, we got the following results:

A screenshot of a cell phone

Description automatically generated

The model converged to almost 90% accuracy after the epoch 40, which is the estimated best time for our training process.

After numerous other tried and experiments, we decided to go with this model because it’s the one that had the best generalization ability over all the others, who were either underfit or slightly overfit. So, we save the configurations and weights in a file, to use it directly next time.

# Making predictions

Making video classifications is straightforward with [opencv](https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_gui/py_video_display/py_video_display.html). All what we need to do is grab “N” frames and then feed them to our model. Iterate until there’s either no more frames or manual interruption. The output seems to be flickering, so we take the average of all the values sequences prediction values, and only output that. Let’s try it on a few examples.

A picture containing ceiling, monitor, car, sitting

Description automatically generated

Out of 5 tests, 4 were correctly classified. Which is good enough, seeing how limited our dataset is. We can move to the final step now.

# Live Predictions

Finally, it’s time for the final step. Using the same algorithm as the memory predictions, which is grabbing 5 frames at a time, making a prediction, and then taking the mean of all predictions, we can apply it on the PC’s webcam to constantly grab 5 frames and predict the state until an order is received to stop. (Pressing the “q” button)

A screenshot of a computer screen

Description automatically generated

Unfortunately, I couldn't head out and test it in real time, so we just have to rely on the theoretic results.

# Conclusion

This model is still in its alpha version. Many improvements can be made, especially by adding valuable data to our dataset. The project is extensible, we can add many more features to our detection by adding a new subfolder under the Data directory and labeling it, then placing all the related data in it. More tests could have been conducted, but the processing power and time needed are above my PC’s ability. Better accuracy can be achieved with better data.