An Embedded Deep Learning Computer Vision Method for Driver Distraction Detection

by

Benjamin Roytburd

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Engineering (Computer Engineering) in the University of Michigan–Dearborn 2021

Master's Thesis Committee:

Table of Contents

List of Tables

List of Figures

Abstract

Driver distraction is a modern issue when operating automotive vehicles. It can lead to impaired driving and potential accidents. Detecting driver distraction most often relies on analyzing a photo or video of the driver being distracted. This involves complex deep learning models which often can only be ran on computers too powerful and expensive to implement into automobiles. This thesis presents a method of detecting driver distraction using computer vision methods within an embedded environment. By taking the deep learning architecture SqueezeNet, which is optimized for embedded deployment, and benchmarking it on a Jetson Nano embedded computer, this thesis demonstrates a viable method of detecting driver distraction in real time. The method shown here involves making slight modifications to SqueezeNet to be trained on the AUC Distracted Driver Dataset, yielding accuracies as high as 93% and speeds as high as 11 FPS when detecting distracted driving. This performance is similar, and/ or better when compared to larger, more complex deep learning models trained for similar driver distraction detection applications.

Chapter 1: Introduction

Driver distraction is a cause for many roadway accidents, according to the NHTSA, in 2018, 2,841 drivers were killed, and 400,000 drivers were injured in vehicular accidents caused by distracted driving [1]. This is a shocking number, for something as preventable as being distracted while driving. Even with all of the safety mechanisms in modern cars to the reduce the chances vehicular accidents, the best way to deal with the statistics of distracted driving accidents is through mitigation. The need for mitigation of driver distraction is especially obvious when one considers how connected to technology modern vehicles are. Besides the common place distractions during driving such as talking, eating, and adjusting the vehicle radio, drivers now have to deal with smart devices such as smart phones and smart watches, as well as devices built within the vehicle itself in the form of interactive touch screen radios and cluster units. Many existing driver distraction detection methods only detect a limited amount of distraction modes or are easy for the driver to bypass. For instance, General Motors tracks the driver's head to make sure their eyes are on the road while their vehicles are autonomous [19]. Such methods of distraction detection cannot capture certain modes of distraction while driving (e.g. a driver can talk on their phone while facing the road). The automotive company Tesla uses distraction detection methods easily fooled by an end user. Tesla detects driver distraction by determining if the driver's hands are on the wheel while their vehicles are autonomous.

As of late, there have been documented cases of drivers using things like oranges and water bottles to apply weight on the wheel to bypass this distraction system [18]. This thesis proposes its own method of detecting driver distraction that covers many modes of distraction, is difficult to bypass, and is deployable onto embedded hardware. Figure 1 shows how this thesis went about determining the best way to detect driver distraction using existing research and technologies.

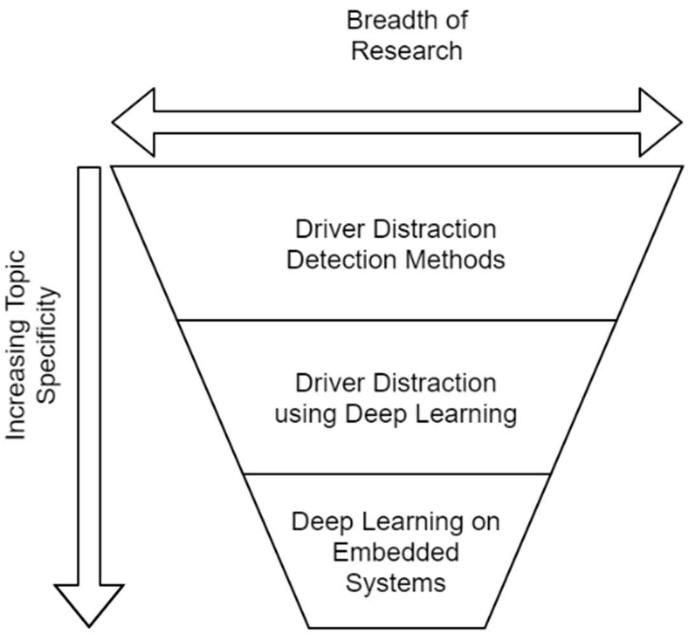

Breadth of Research
Increasing Topic Specificity
Driver Distraction Detection Methods
Driver Distraction using Deep Learning
Deep Learning on Embedded Systems

Figure 1. Research Methodology

This thesis will be organized into the following sections, a literature survey of the topics mentioned in Figure 1, a design section covering design decisions of the proposed methodology, an implementation section which deploys and tests the design, a discussion section discussing the results of the implementation, and a conclusion.

A. Driver Distraction Detection Methods

The first topic of the literature survey is driver distraction methods. Modern research on driver distraction seems to have reached a consensus on what constitutes it. Based on NHTSA guidelines most of the time, driver distraction is any activity that diverts a driver's attention from the main task of driving. Most research into driver distraction states this one way or another. What differs in most papers is what method of detecting driver distraction is used. The features that researchers monitor for are things like gaze direction [3, 5, 25, 26, 27], inertial measurement units (IMUs) [4], cognitive distraction [6, 25, 26], and physiological features [2]. The features used in these papers are fed into artificial intelligence models such as deep learning. The most common models have been support vector machines [6, 25, 26] and neural networks [2, 3, 4, 5]. Some models use more traditional computer vision techniques such as the Haar Cascade Classifier [27]. These models were trained and tested on desktop computers, and not implemented on any vehicular hardware. There was also a lack of testing and training for real-time performance metrics, such as frames per second. While the models used in the papers are highly accurate, they are not scalable for embedded deployment due to their complexity and lack of design for deployment in real-time. This thesis proposes using deep learning to detect driver distraction, but it will design its approach with real-time deployment in mind. It will do that by training a deep learning model on a local desktop, but then deploying it on embedded hardware. Real-time metrics will also be measured and optimized for. To facilitate a real-time deployment, the feature to detect for distraction must be kept simple, thus this thesis will use driver pose as its feature.

B. Driver Distraction Using Deep Learning

The proposal this thesis has thus far is to detect driver distraction by determining the driver's pose. There is already some research done in this area, and there are many different approaches to solving this problem. Most papers covering this topic use already established deep learning architectures known as convolutional neural networks (CNNs) such as VGG-16, AlexNet, or InceptionV3 [2, 7, 29, 32]. Other papers use neural networks as well, but in different forms such as a multi-layer perceptron, Feed-Forward Neural Network, or a combination of different neural networks [8, 9, 10, 28, 31]. CNNs can also be made simpler for this application, with as little as three convolutional layers [30, 33]. The methods that use deep CNNs yield high accuracy (at least 90%) and high FPS performance. Table 1 shows that the better the model performs, the more complex it and its dataset is. Table 1 also shows that the simpler models and datasets can have a higher FPS. Thus, a compromise needs to be made between accuracy and speed when choosing a model. This thesis proposes using a CNN to detect driver pose, but it will focus on an architecture and design made for deployment onto embedded hardware, keeping it simple but with high performance metrics.

C. Deep Learning on Embedded Systems

Now that this thesis has defined the feature to detect for driver distraction (driver pose), the deep learning method to facilitate this detection (CNN), what's left is to determine the hardware the CNN will be deployed on. There is existing research on deploying deep learning such as CNNs onto embedded hardware. Researchers have deployed deep learning models on microcontrollers [11, 12, 35, 36], microprocessors [13, 36], and field-programmable-gate-arrays (FPGAs) [14, 34]. These three pieces of hardware each come with their own advantages and disadvantages, microcontrollers can be too limited in memory and processing power to deploy an effective deep learning model, and FPGAs can be very expensive and require an extra step of hardware design to facilitate deep learning. Microprocessors are more effective here, such as a Raspberry PI or Jetson Nano, as they have the memory and processing power to do deep learning without the cost of something like an FPGA. This thesis proposes deploying a CNN that is built for microprocessors and is compatible with the application of computer vision.

Table 1. Driver Distraction Literature Survey

Paper	Features Used	Distraction Detection Method	Accuracy	FPS
[2]	Driver Pose (10 poses)	Deep Learning Network Ensemble	96%	52
[3]	Head Pose, Gaze, Facial Features	Combination of Computer Vision and Artificial Intelligence	>90%	26
[4]	Driver Pose (6 poses)	Deep Learning Network Ensemble	87%	40
[5]	Vehicle Dynamics, Gaze	Support Vector Machines, Deep Learning, Neural Networks	>95%	25
[6]	Eye Movement	Support Vector Machines	81%	60
[7]	Driver Pose (10 poses)	Deep Learning Network	92%	14
[8]	Driver Pose (6 poses)	Discrete Cosine Transform, Neural Network	93%	None
[9]	Head and Body Joint Position	Random Forests, Neural Networks	81%	30
[10]	Driver Pose (6 poses)	Deep Learning Network Ensemble	93%	25
[28]	Driver Pose (4 poses)	Multi-Layer Perceptron	90%	None
[29]	Driver Pose (10 poses)	Deep Learning Network	95%	None
[30]	Driver Pose (4 poses)	Small Deep Learning Network	99%	None
[31]	Driver Pose (10 poses)	Deep Learning Network Ensemble	92%	8
[32]	Driver Pose (10 poses)	Deep Learning Network	95%	42
[33]	Body Part Positions and Face Angle (17 classes)	Small Deep Learning Network	98% / 91%	29

D. Deep Learning Network Comparisons

Deep learning networks can vary quite a bit and deciding on a CNN that can be deployed onto a microprocessor for driver distraction detection requires comparison among different deep learning networks. Table 2 compares existing CNNs which have been used for a variety of computer vision applications. The network that stands out the most from this table for this thesis, will be the one with the least number of parameters, with the most classifications. These requirements will yield a network that is small enough to be deployed onto a microprocessor, while retaining accuracy when being trained on large datasets.

In the above table, SqueezeNet stands out as the network for this application. With a parameter count of 1,200,000, and a classification application of 1000 classifiers, SqueezeNet is a small network which is powerful enough to handle the AUC Distracted Driver Dataset. SqueezeNet was originally built for deployment on FPGAs, but due to its size, it is feasible to use on a microprocessor [14]. Another reason SqueezeNet stands out, is that both AlexNet and SqueezeNet are trained on the ImageNet dataset, both have the same accuracy, but SqueezeNet is many times smaller.

Table 2. Deep Learning Network Comparison

	Parameters	**Size**	**Dataset Application**	**Accuracy**	**Application**	**Classifications**
SqueezeNet	1,200,000	4.8 MB	ImageNet	80.30%	Image Classification	1000
MobileNet	4,200,000	NA	ImageNet	70.60%	Image Classification	1000
AlexNet	60,000,000	240 MB	ImageNet	80.30%	Image Classification	1000
Tiny SSD	1,130,000	2.3 MB	VOC2007/2012	61.30%	Object Detection	20
VGG-16	134,000,000	528 MB	ImageNet	92.70%	Image Classification	1000
GoogLeNet	4,000,000	NA	ImageNet	93.33%	Image Classification	1000
MicronNet	510,000	1 MB	German Traffic Sign Recognition Benchmark	98.90%	Traffic Sign Recognition	43

Chapter 2: Design

A methodology refers to practices and procedures regarding a specific discipline. This section of the thesis covers the proposed methodology to accomplish the task of detecting driver distraction on a real-time embedded system, as well as the design of the technologies used in the proposal (SqueezeNet and the AUC Distracted Driver Dataset).

A. Proposed Methodology

In this thesis, a target accuracy is posed based upon [2], this is because this is one of the papers where the AUC Distracted Driver Dataset is introduced and tested. The goal of the model used in this thesis will be to have a final accuracy of at least 95.98% and a performance speed of 52 frames per second (FPS) [2]. This thesis will be using SqueezeNet as its deep learning architecture, since this architecture is designed with limited memory in mind, the target size of the fully trained model should be less than 4.8 megabytes (MB) [14]. This size is chosen because it is the size of SqueezeNet in [14] before any kind of model compression techniques.

The deep learning model used in this thesis will be trained using the AUC Distracted Driver Dataset [2]. This dataset has 10 classes, nine of those classes are distracted driving modes, and one class is the safe driving mode. For this thesis, the classes will be changed to two, one class will be distracted, which will encompass all nine of the distracted driving modes, and the other class will be safe driving. The deep learning model that is used for this thesis will be SqueezeNet [14]. The model will be pre-trained on the ImageNet dataset and be programmed using the Python [24] language, and the deep learning framework that will be used to build the model is

TensorFlow and TensorFlow-Lite [22], which will quantize and compress the model so it can fit onto an embedded device. The hardware that this compressed model will be deployed on is the Jetson Nano 2 gigabyte (GB) version.

SqueezeNet is the chosen deep learning architecture for this thesis due to its small size and application in embedded deep learning deployment. It will need to be modified further for this thesis' application, but the network as its presented in [14] is a good baseline to begin with.
Next, this model shall be built using TensorFlow, unfortunately it is most likely that leaving the model in the TensorFlow format will cause it to be too large to deploy on any embedded hardware, even though the model is designed for embedded deployment. SqueezeNet needs to be quantized and compressed to an even smaller size, and then ran in its compressed state on the embedded target. There are many software libraries available to convert the built SqueezeNet model into an embedded ready one. A library that can be deployed on a wide range of hardware is needed, that is why TensorFlow-Lite is chosen for this thesis, Figure 2 shows this library can convert TensorFlow models, and it can be deployed on a wide variety of hardware. Finally, the embedded hardware for this thesis was chosen as the Jetson Nano since it is cheaper and contains an integrated GPU which is made for deep learning applications.

B. SqueezeNet Architecture

SqueezeNet was designed to be a smaller version of AlexNet [15], which was a very deep convolutional neural network that was created to classify the ImageNet dataset, a dataset of 1000 classifications of images. AlexNet achieved an accuracy of 80.3% on this dataset, the model was 240 MB in size and contained 60,000,000 parameters. SqueezeNet with its design was able to achieve this accuracy of 80.3% using only 4.8 MB of memory with 1,200,000 parameters. This is a massive reduction in size, with little to no performance change of the model. AlexNet was one of the models used on the AUC Distracted Driver Dataset [2].

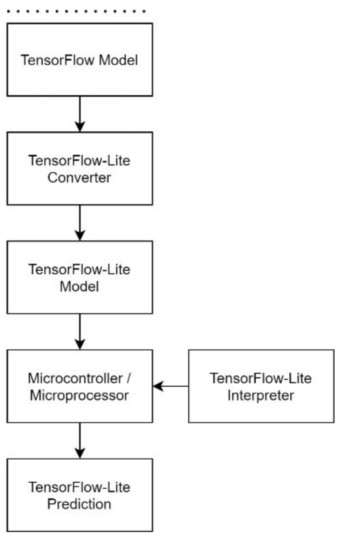

TensorFlow Model
TensorFlow-Lite Converter
TensorFlow-Lite Model
Microcontroller / Microprocessor
TensorFlow-Lite Interpreter
TensorFlow-Lite Prediction

Figure 2. TensorFlow Deployment Process

One of the main features of SqueezeNet is the fire module, it is a specialized component of a neural network that uses a mix of filter sizes. A fire module is a component or layer of a CNN that has a mix of filter sizes, Figure 3 shows that the fire modules in SqueezeNet are a mix of 1 x 1 and 3 x 3 convolutional filters.

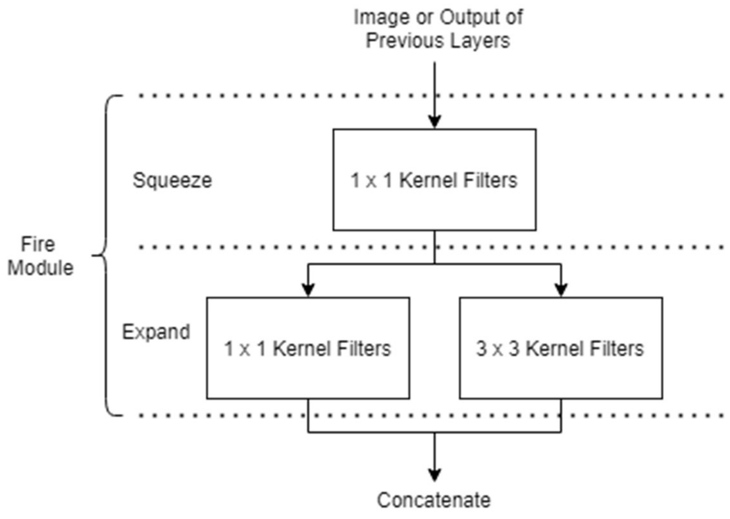


Figure 3. Fire Module

The fire module is one of the reasons this thesis selects SqueezeNet as its chosen deep learning architecture. It is one of the ways that the network can reach Alex-Net level accuracies while being 50 times smaller. Smaller architectures are more conducive to embedded devices due to their smaller memory footprint. To maximize the limited memory and processing power of an embedded device, a network that can perform as well as its larger counterparts is needed [14].

C. AUC Distracted Driver Dataset Structure

Researchers at The American University in Cairo (AUC) created a distracted driving dataset in response to currently available distracted driving datasets based on posture. Existing datasets of the same type are the StateFarm and Southeast University (SEU) datasets. The downsides of the StateFarm dataset are that it was made for the express purpose of a deep learning competition and is not available for further research. The downside of the SEU dataset is that it only contains four postures, compared to StateFarm's 10 postures [2]. Figure 4 shows that The AUC distracted driver dataset contains postures of the same kind as the StateFarm dataset. These postures are as follows: drinking, adjusting the radio, safe driving, adjusting hair and/or makeup, reaching behind the driver, talking to passengers, talking on a cell phone using the left and right hand, and texting on a cell phone using the left and right hand. A big advantage that the AUC dataset has is that it uses a very diverse driver pool, sampling 31 drivers of different gender and race from seven different countries. These countries are Egypt, Germany, USA, Canada, Uganda, Palestine, and Morocco. 22 of the drivers are male, and 9 are female [2]. The images are parsed from videos, so the data is formatted in "streams" of various sizes. There are streams of drivers performing the distraction pose mode with varying lengths, some of these streams are as large as 190 images, or as small as three images. This thesis proposes determining distraction based on one image at a time, since this is a real-time implementation of detecting driver distraction.

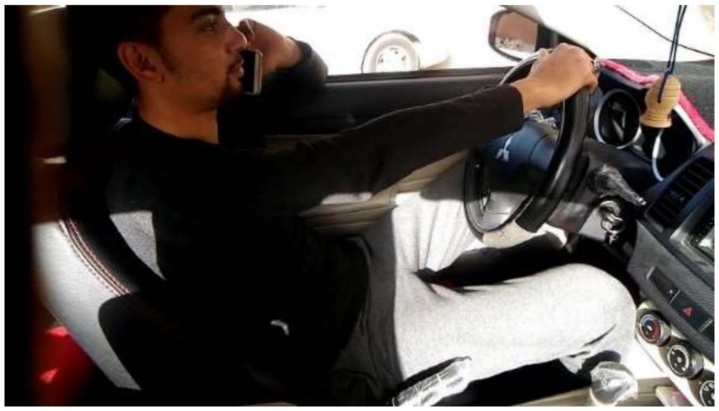


Figure 4. AUC Distracted Driver Dataset Example Image

Next, this thesis proposes taking classes two through ten (every class besides safe driving) and combining them into one class, “distracted driving”. The logic behind this, is that any form of distracted driving is unacceptable and must be mitigated. When detecting driver distraction, it is important to make a distinction when the driver is driving safe, and if the driver is operating the vehicle in any distracted state. It also simplifies the deep learning problem from a multi-class one to a binary classification one. As will be shown later in the thesis, this reduces parameters in the deep learning architecture of choice, SqueezeNet. Unfortunately, this creates a challenge with the dataset, there are much more images of distracted drivers as there are safe drivers. This creates a lopsided dataset, with much more representation of distracted driving. This lopsidedness could be kept, but it may cause some classification issues with safe driving, as will be shown later in the thesis.

In the original paper where the AUC dataset was introduced, three drivers were used as validation, and the other 28 drivers were used to train the deep learning network. This thesis uses six drivers for testing, and 24 drivers for training. The drivers selected for testing are drivers two, 12, 14, 21, 27, and 30. These drivers were selected by selecting six random drivers at a time as the test dataset, and training and testing SqueezeNet. This selection of drivers yielded the highest accuracy.

The poses for all the drivers in the AUC dataset are shown in Figure 5. While these poses may not represent every possible scenario that a driver may be in during driving, they represent the most common distraction situations that drivers may be in [2]. Some of these poses may even become more outdated with the advent on hands free calling in most modern vehicles (i.e. talking on cell phone left and right hand classes), but the methods shown in this thesis could work on other potential poses or be trained to detect one pose more than the other. It could also be argued that a driver distraction detection system such as this would encourage more drivers to use hands free functionalities on modern vehicles.

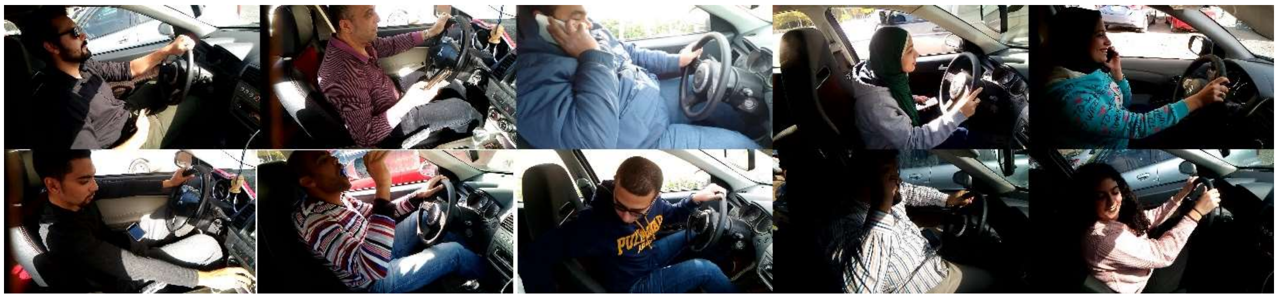


Figure 5. Driver Distraction Classes

In Figure 5, the classifications are as follows (starting from the top left image), safe driving, texting on a cell phone using the right hand, calling on a cell phone using the right hand, texting on a cell phone using the left hand, calling on a cell phone using the left hand, adjusting the radio, drinking, reaching behind the driver, adjusting hair and/or makeup, and talking to passengers.

D. Software Design

Before any software is written, a design must be proposed. Given the previous discussion of the methodology, SqueezeNet architecture, and AUC dataset, all these requirements need to be translated into a software implementation. The essentials of the software implementation will be to load and potentially augment the AUC Distracted Driver Dataset, to build and modify SqueezeNet, and then to train the modified SqueezeNet with the augmented dataset. Figure 6 shows if all these requirements are met during implementation, a trained version of SqueezeNet will be available for validation and deployment into an embedded environment.

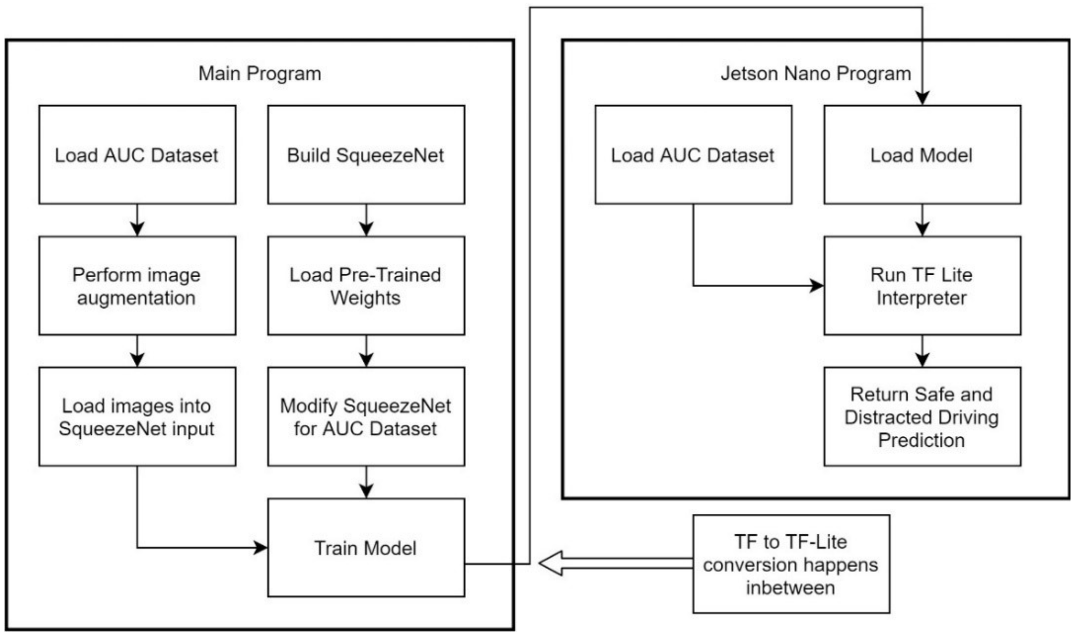

Main Program
Load AUC Dataset
Build SqueezeNet
Perform image augmentation
Load Pre-Trained Weights
Load images into SqueezeNet input
Modify SqueezeNet for AUC Dataset
Train Model
Jetson Nano Program
Load AUC Dataset
Load Model
Run TF Lite Interpreter
Return Safe and Distracted Driving Prediction
TF to TF-Lite conversion happens inbetween

Figure 6. Software Design

Chapter 3: Implementation

The method proposed in this thesis will require more than using SqueezeNet on the AUC dataset, both mentioned concepts will require their own modifications and tuning. This section will cover modifications to SqueezeNet and the AUC Distracted Driver Dataset so that the technologies can be integrated together, and then deployed onto the Jetson Nano hardware.

A. SqueezeNet Modifications and Tuning

The principal of SqueezeNet, is to have less neurons and computation, without sacrificing accuracy of the model. Keeping this in mind, there have been other versions of SqueezeNet that attempt to take this idea even further. The version of SqueezeNet that this thesis uses is known as SqueezeNet 1.1 [16]. This version of SqueezeNet has less parameters, without sacrificing any accuracy. While SqueezeNet 1.0 has 1,248,424 parameters, SqueezeNet 1.1 has 1,235,496 parameters. This is not a significant drop in parameters, but [16] claims that this results in 2.4 times less computation. The size of SqueezeNet in this thesis is 895,554 parameters, which is 1.4 times less parameters than that of SqueezeNet 1.0. When replacing the final layer of SqueezeNet with two classes, it decreases the number of neurons from 1000 to two. The images used in the AUC dataset are scaled down to 224 by 224 pixels for input. The final modification made to SqueezeNet for this thesis was to introduce L2 regularization to the neurons in the network.

B. Dataset Modification and Reconstructing

The AUC Distracted Driver Dataset contains ten classes of drivers, nine of those classes are distracted forms of driving, and one of those classes is safe driving. For the purposes of this thesis, the nine distracted classes are combined into one class, called “distracted driving”, and the class of safe driving is kept as its own class, called “safe driving”. This creates an issue since there are now many more images of distracted driving than safe driving. Specifically, there are 2720 images of safe driving, and 8958 images of distracted driving. Training SqueezeNet with this data may cause it to overfit / bias towards detecting everything as distracted. Certain measures are taken to compensate for this. One way is to use a validation dataset to measure how well SqueezeNet is generalizing epoch to epoch during training. The drivers used for validation are drivers 2, 12, 14, 21, 27, and 30, while the rest of the drivers are used for training. For validation, the images are fed into the network as a size of 224 by 224 pixels, but no further image preprocessing is done on them. For training, there is image preprocessing done to prevent overfitting. Using Keras and TensorFlow, as the images are being fed into SqueezeNet during training, they are randomly rotated 30 degrees, shifted by width and height up to 10 percent, sheared by up to 20 percent, zoomed in on by up to 20 percent, and any missing pixels caused by this manipulation are filled in by the nearest pixels of the original image. Initially, SqueezeNet was validated using all of the validation images during training, but later stages of training involved using just one image of each validation driver as validation. This is because the way this driver distraction detection method is designed, distraction is detected frame by frame (or image by image), so a validation method of a single image should show the performance of the model just as well as an entire stream of images. The dataset was fed into SqueezeNet in batches of 64 images over 50 epochs, the batch size was the same for the training and validation datasets when the entire stream of images was used for validation.

C. SqueezeNet Compression and Quantization

SqueezeNet is already a reduced deep learning neural network, but there are further methods that can be taken to compress a network besides the methods covered so far. In the original SqueezeNet paper, the size of the fully trained network is 4.8 MB. With further compression techniques such as quantization, deep compression, and pruning, SqueezeNet can get as small as 0.47 MB [14]. This yields a huge reduction in size of the network and is what allows these deep learning models to be deployed on embedded devices with limited memory. Within [14], the data type of the original SqueezeNet is 32 bits, through 8-bit quantization this data type can be shrunk down to 8 bits. This creates a network with the size of 0.66 MB. In SqueezeNet this compression resulted in no loss in accuracy on the ImageNet dataset. For this thesis, TensorFlow is used to build and train a version of SqueezeNet for embedded deployment. This thesis also uses TensorFlow-Lite to compress SqueezeNet to a smaller version for embedded deployment. The uncompressed version of SqueezeNet in this thesis is a size of 4.26 MB, which is smaller than the original SqueezeNet size of 4.8 MB. After compression using TensorFlow-Lite, SqueezeNet is compressed to a size of 0.904 MB. The original model that SqueezeNet is based on, AlexNet, is a size of 240 MB uncompressed. When comparing AlexNet to this thesis' version of SqueezeNet, that is a size reduction of around 265 times. Considering that the original implementation of the AUC Distracted Driver Dataset used four trained versions of AlexNet in combination with other deep learning networks to detect driver distraction, this is a huge space saver [2].

D. Jetson Nano Hardware Deployment

To deploy deep learning networks on embedded hardware, often something more powerful than a typical MCU is needed. This creates an issue for low-cost solutions though, as highly integrated single board computers can be very expensive. For the Jetson Nano 2GB, this is not necessarily the case. At the price point of around 60 dollars, it is one of the more affordable options for embedded computing out there. The specifications that the Jetson Nano has also suits it for deep learning applications. It has a quad-core ARM CPU with a clock speed of 1.57 GHz, and it has a 128-core Maxwell GPU. The Jetson Nano uses the Tegra X1 processor which is built on the Maxwell GPU architecture and is the enabling factor in using embedded deep learning [17]. In Figure 7, the Jetson Nano is shown within the context of this thesis' methodology. The Tegra X1 chip is the workhorse behind processing the inputs and outputs of the SqueezeNet architecture.

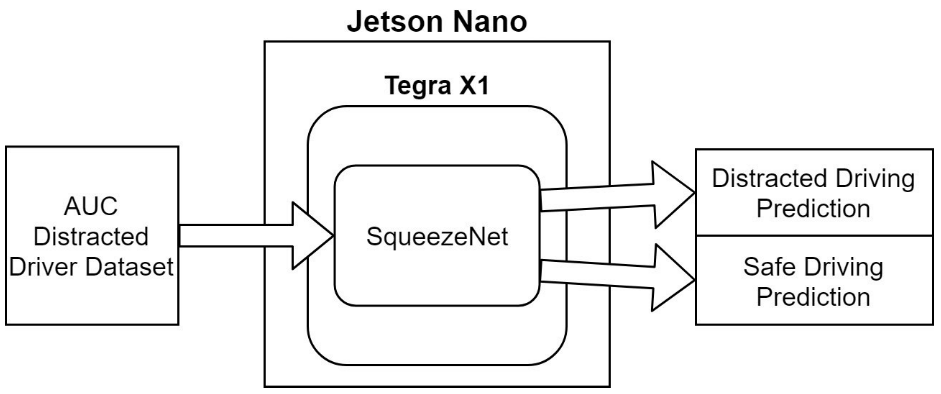


Figure 7. System Hardware Diagram

To show how powerful the hardware is, and how it can power deep learning, Figure 8 [20] draws the hardware block diagram of the Jetson Nano, all the I/O allows for the development kit to be easily started up and developed on. The Tegra X1 processor is built on the Maxwell GPU architecture and is the enabling factor in using embedded deep learning on this Jetson Nano, as it is a powerful GPU on embedded hardware.

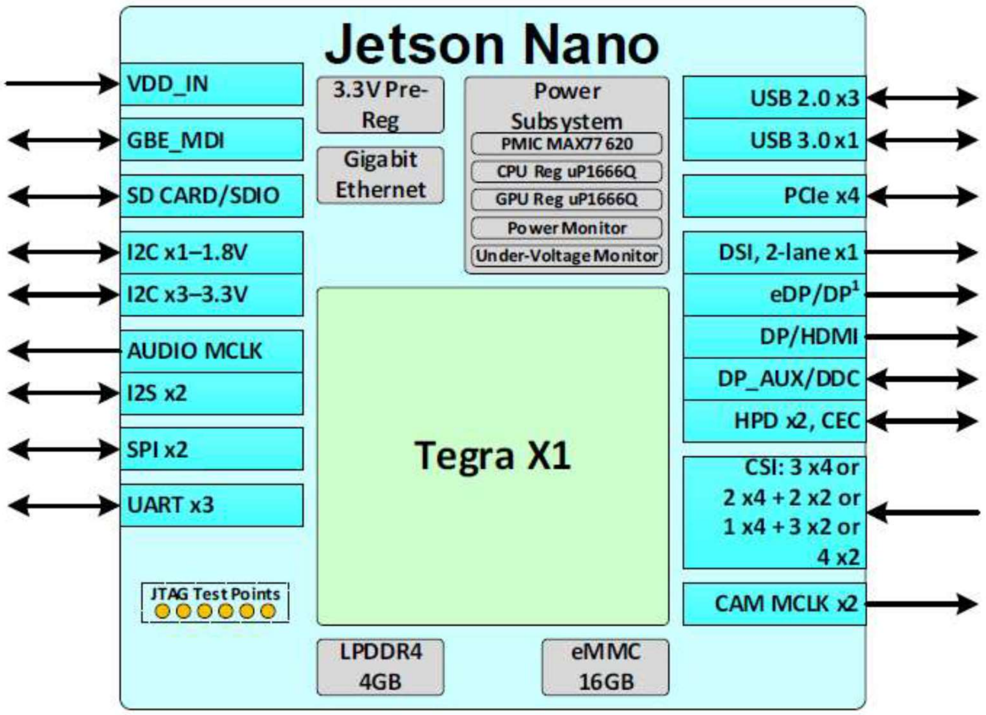


Figure 8. Jetson Nano Hardware Diagram

The Jetson Nano runs the Jetpack SDK which includes a Linux operating system and APIs for deep learning and computer vision. Software on the Jetson Nano was written in Python and used the TensorFlow / TensorFlow-Lite API. This thesis used the 2 GB RAM version of the Jetson Nano, but there is a 4 GB RAM version of the Jetson Nano that could yield even higher performance. The software could also be written in C++ instead of Python, which could yield more performance boosts. The implementations of this driver distraction can vary widely on the target hardware, but the design of the network and its method of deployment is what allows for higher performance in an embedded environment.

Figure 9 shows all the methods and processes used to make SqueezeNet compatible with the AUC Distracted Driver Dataset, and for it to be deployed onto the Jetson Nano Hardware. The chart from left to right shows the steps taken with SqueezeNet and with the AUC Distracted Driver Dataset for compatibility and deployment.

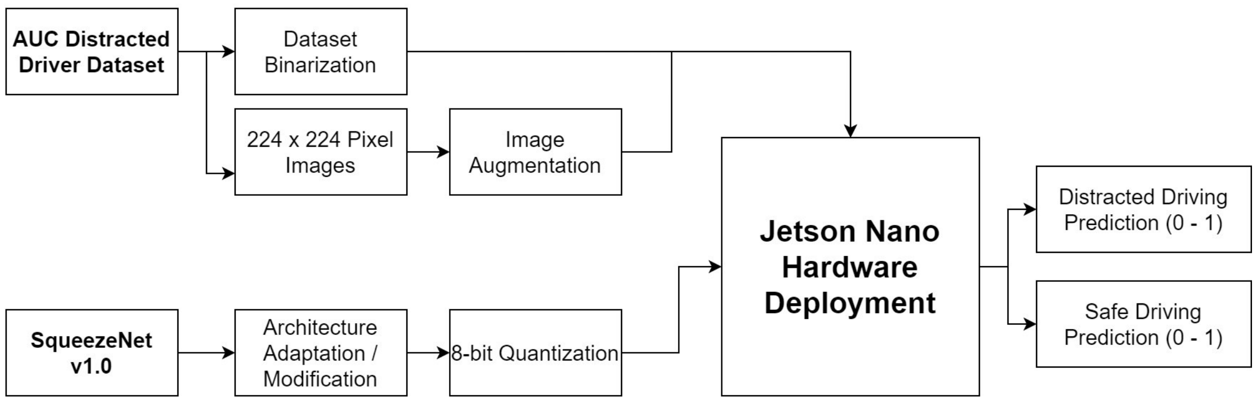

AUC Distracted Driver Dataset
Dataset Binarization
224 x 224 Pixel Images
Image Augmentation
Jetson Nano Hardware Deployment
Distracted Driving Prediction (0 - 1)
Safe Driving Prediction (0 - 1)
SqueezeNet v1.0
Architecture Adaptation / Modification
8-bit Quantization

Figure 9. Methodology Flowchart

E. Desktop Software Development

The software implementation for this thesis was written in Python using its built in libraries, TensorFlow and its libraries, and NumPy and its libraries [23]. To train and create a desktop version of SqueezeNet v1.1, the following libraries were imported in Python: TensorFlow, NumPy, OS, and Shutil. A call graph was created to demonstrate which functions are called from the "main" of the Python program to build the dataset and train SqueezeNet. Figure 10 shows a generalized call graph, so it does not show the detailed TensorFlow calls, and demonstrates the time flow of the program using the term "time_step" to indicate sequentially what occurs when the program runs. The functions within the call graph will be detailed in this section. The call graph has its functions defined by Table 3, which details the sequence that the functions in the call graph are called.

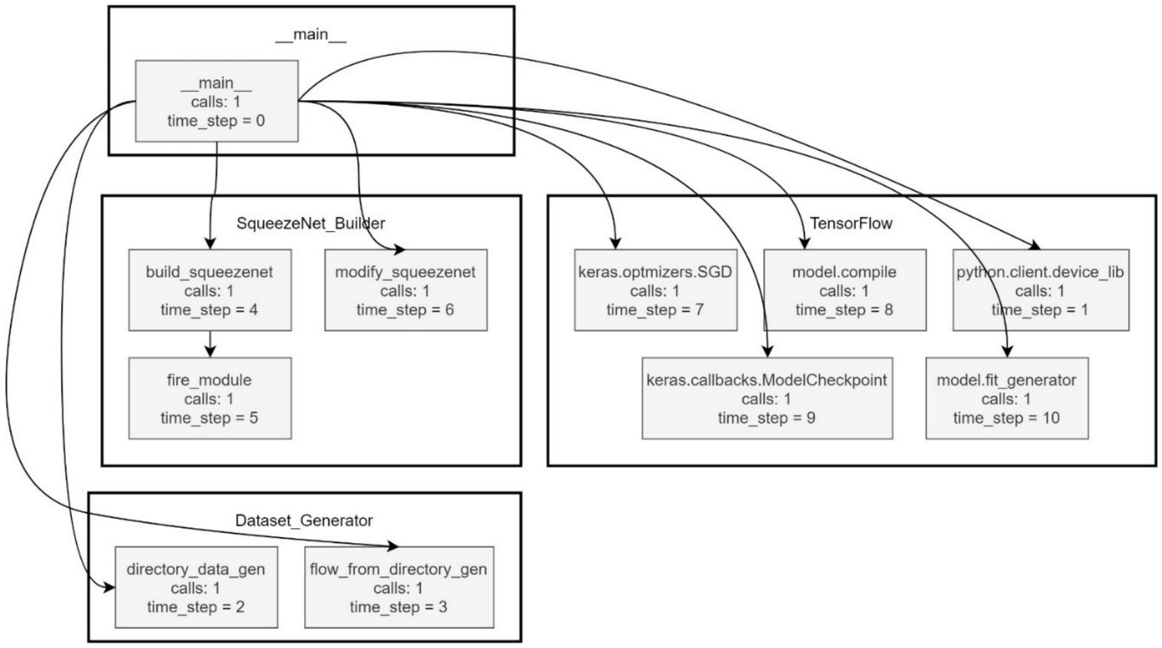
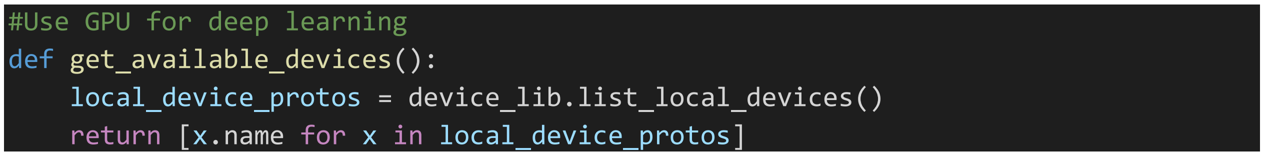


Figure 10. Call Graph

Table 3. List of Functions

	Time Step	Calls	File / Resource
__main__	0	1	__main__
python.client.device_lib	1	1	TensorFlow
directory_data_gen	2	1	Dataset_Generator.py
flow_from_directory_gen	3	1	Dataset_Generator.py
build_squeezenet	4	1	SqueezeNet_Builder.py
fire_module	5	1	SqueezeNet_Builder.py
modify_squeezenet	6	1	SqueezeNet_Builder.py
keras.optimizers.SGD	7	1	TensorFlow
model.compile	8	1	TensorFlow
keras.callbacks.ModelCheckpoint	9	1	TensorFlow
model.fit_generator	10	1	TensorFlow

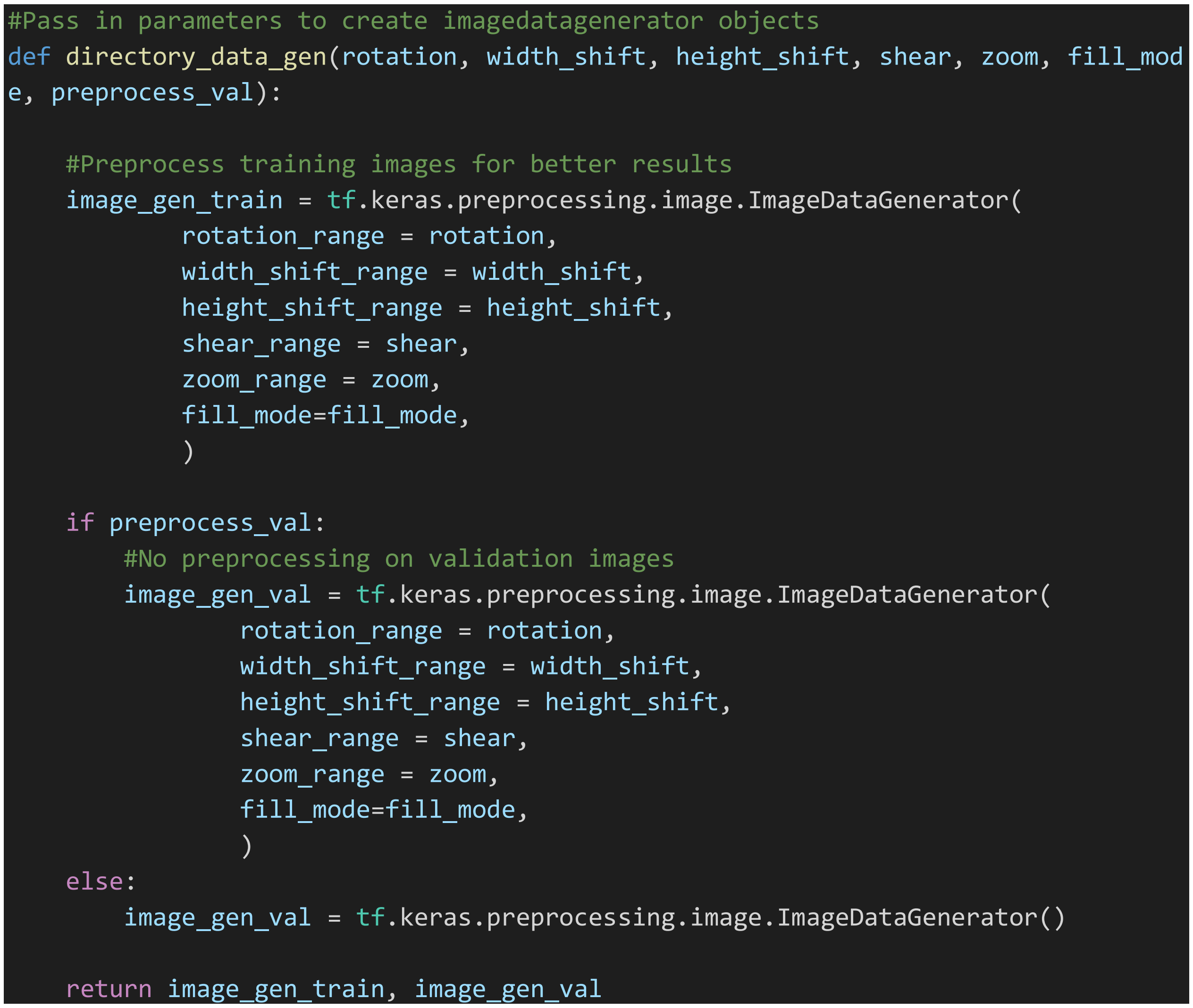
First, to use the desktop's GPU to train and test SqueezeNet, TensorFlow was used (Figure 11).



```
#Use GPU for deep learning
def get_available_devices():
    local_device_protos = device_lib.list_local_devices()
    return [x.name for x in local_device_protos]
```

Figure 11. get_available_devices

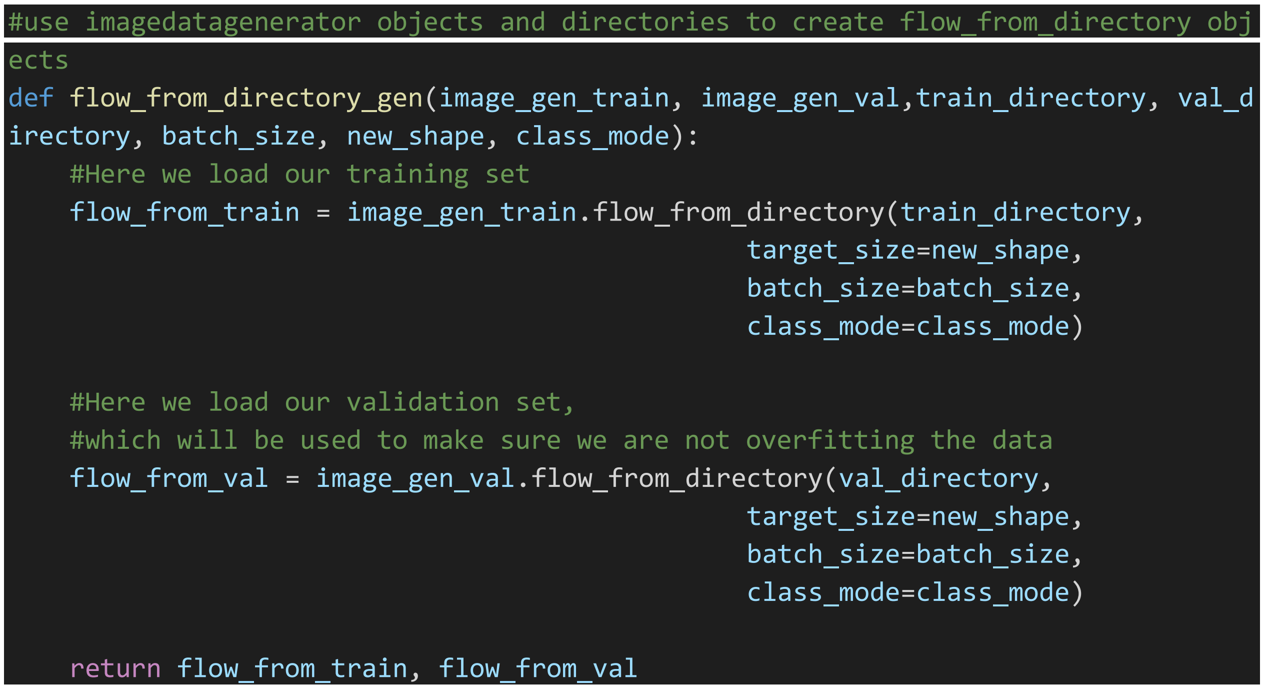
TensorFlow selects the available GPU on the desktop for deep learning. Next, the directories for the training and validation data of the AUC Distracted Driver Dataset need to be loaded and modified. Image augmentation then takes place on the training data of the AUC dataset (Figure 12).



```
#Pass in parameters to create imagedatagenerator objects
def directory_data_gen(rotation, width_shift, height_shift, shear, zoom, fill_mod
e, preprocess_val):

    #Preprocess training images for better results
    image_gen_train = tf.keras.preprocessing.image.ImageDataGenerator(
            rotation_range = rotation,
            width_shift_range = width_shift,
            height_shift_range = height_shift,
            shear_range = shear,
            zoom_range = zoom,
            fill_mode=fill_mode,
            )

    if preprocess_val:
        #No preprocessing on validation images
        image_gen_val = tf.keras.preprocessing.image.ImageDataGenerator(
                rotation_range = rotation,
                width_shift_range = width_shift,
                height_shift_range = height_shift,
                shear_range = shear,
                zoom_range = zoom,
                fill_mode=fill_mode,
                )
    else:
        image_gen_val = tf.keras.preprocessing.image.ImageDataGenerator()

    return image_gen_train, image_gen_val
```

Figure 12. directory_data_gen

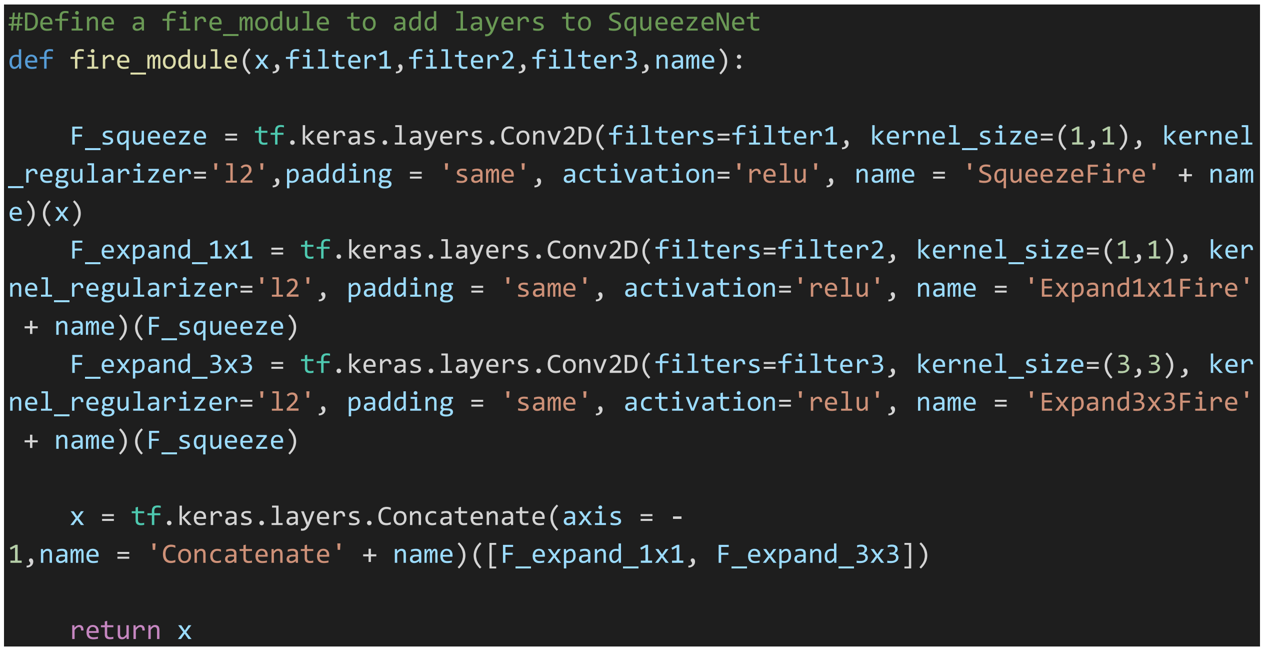
The validation data has no image augmentation done on it, in order to match possible deployment of SqueezeNet, and potentially increase validation accuracy. To load the directories as images that TensorFlow can operate on, a batch size of 64 needs to be defined. The image size of the dataset is also redefined to 224 by 224 pixels. The dataset is defined as a "categorical" classification problem, even though it is a binary classification problem (Figure 13). This is done to accommodate SqueezeNet modifications which will be shown later.



```
#use imagedatagenerator objects and directories to create flow_from_directory obj
ects
def flow_from_directory_gen(image_gen_train, image_gen_val,train_directory, val_d
irectory, batch_size, new_shape, class_mode):
    #Here we load our training set
    flow_from_train = image_gen_train.flow_from_directory(train_directory,
                                                            target_size=new_shape,
                                                            batch_size=batch_size,
                                                            class_mode=class_mode)

    #Here we load our validation set,
    #which will be used to make sure we are not overfitting the data
    flow_from_val = image_gen_val.flow_from_directory(val_directory,
                                                            target_size=new_shape,
                                                            batch_size=batch_size,
                                                            class_mode=class_mode)

    return flow_from_train, flow_from_val
```

Figure 13. flow_from_directory_gen

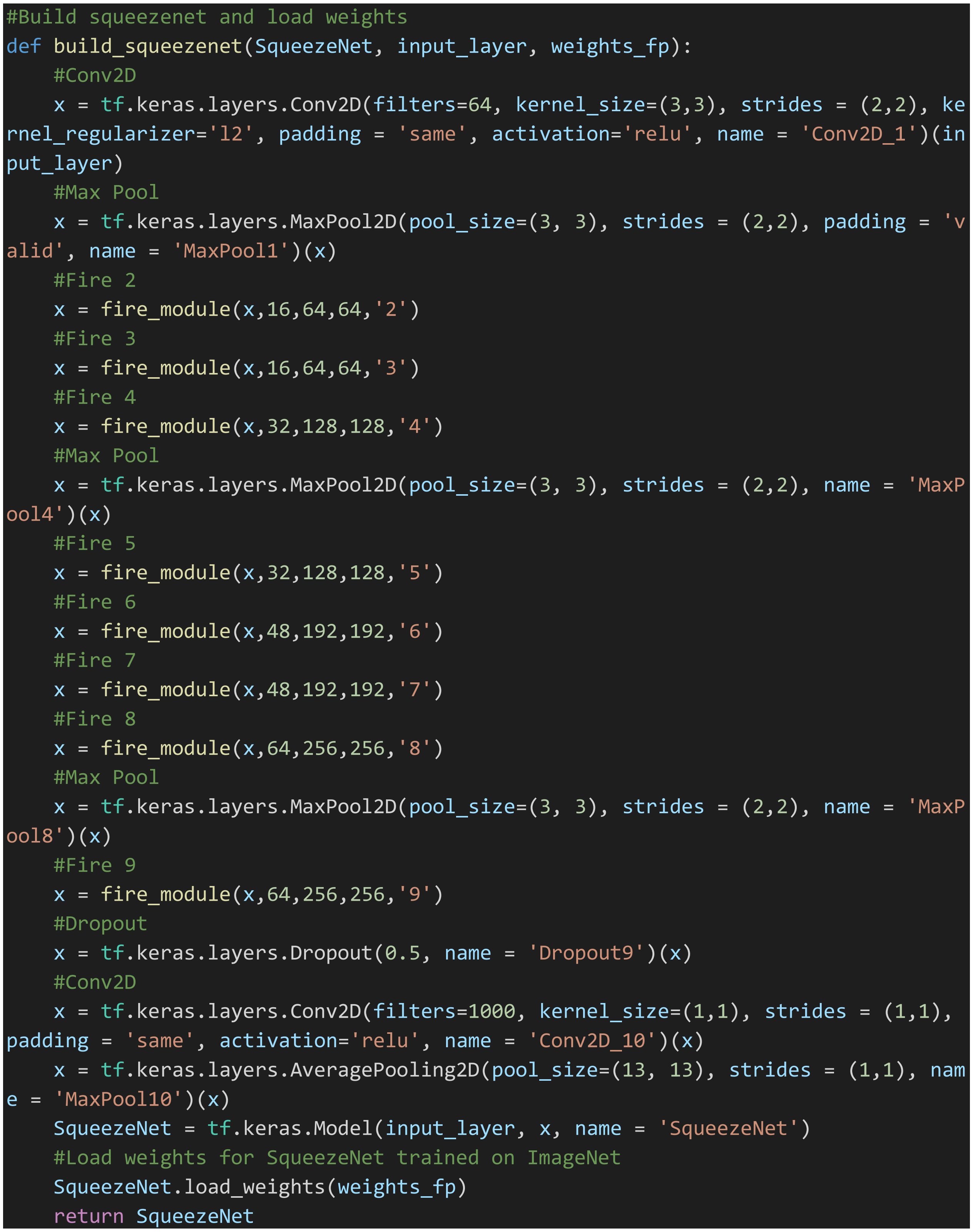
Before SqueezeNet can be built, a "fire" module needs to be created. SqueezeNet is defined using the TensorFlow API, so the fire module needs to also be defined using this API. The fire module function takes a TensorFlow model layer, three filter sizes, and a name as an input. It takes the input layer and feeds it into a "squeeze" module of kernel size (1, 1). This layer is then fed into two expand layers, one of kernel size (1, 1), and another of kernel size (3, 3). These layers have filter sizes also defined by the inputs to the fire module function. The expand layers are concatenated into one layer and returned from the function (Figure 14).



```python
#Define a fire_module to add layers to SqueezeNet
def fire_module(x,filter1,filter2,filter3,name):

    F_squeeze = tf.keras.layers.Conv2D(filters=filter1, kernel_size=(1,1), kernel
_regularizer='l2',padding = 'same', activation='relu', name = 'SqueezeFire' + nam
e)(x)
    F_expand_1x1 = tf.keras.layers.Conv2D(filters=filter2, kernel_size=(1,1), ker
nel_regularizer='l2', padding = 'same', activation='relu', name = 'Expand1x1Fire'
 + name)(F_squeeze)
    F_expand_3x3 = tf.keras.layers.Conv2D(filters=filter3, kernel_size=(3,3), ker
nel_regularizer='l2', padding = 'same', activation='relu', name = 'Expand3x3Fire'
 + name)(F_squeeze)

    x = tf.keras.layers.Concatenate(axis = -
1,name = 'Concatenate' + name)([F_expand_1x1, F_expand_3x3])

    return x
```

Figure 14. fire_module

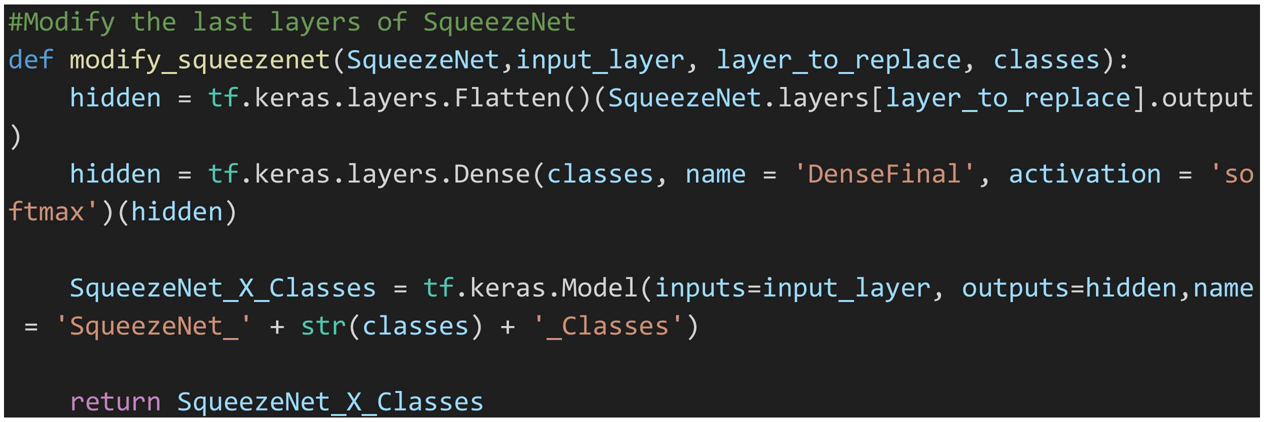
Once the fire module is defined as a function, SqueezeNet v1.1 in its original form can be built. It is built layer by layer using TensorFlow. Certain layers use L2 regularization (fire module layers and 2D convolutional layers). Once SqueezeNet is built, a trained model (h5 file) is loaded in. This model's weights are trained using ImageNet to an accuracy of 80.3 % (Figure 15).



```
#Build squeezenet and load weights
def build_squeezenet(SqueezeNet, input_layer, weights_fp):
    #Conv2D
    x = tf.keras.layers.Conv2D(filters=64, kernel_size=(3,3), strides = (2,2), ke
rnel_regularizer='l2', padding = 'same', activation='relu', name = 'Conv2D_1')(in
put_layer)
    #Max Pool
    x = tf.keras.layers.MaxPool2D(pool_size=(3, 3), strides = (2,2), padding = 'v
alid', name = 'MaxPool1')(x)
    #Fire 2
    x = fire_module(x,16,64,64,'2')
    #Fire 3
    x = fire_module(x,16,64,64,'3')
    #Fire 4
    x = fire_module(x,32,128,128,'4')
    #Max Pool
    x = tf.keras.layers.MaxPool2D(pool_size=(3, 3), strides = (2,2), name = 'MaxP
ool4')(x)
    #Fire 5
    x = fire_module(x,32,128,128,'5')
    #Fire 6
    x = fire_module(x,48,192,192,'6')
    #Fire 7
    x = fire_module(x,48,192,192,'7')
    #Fire 8
    x = fire_module(x,64,256,256,'8')
    #Max Pool
    x = tf.keras.layers.MaxPool2D(pool_size=(3, 3), strides = (2,2), name = 'MaxP
ool8')(x)
    #Fire 9
    x = fire_module(x,64,256,256,'9')
    #Dropout
    x = tf.keras.layers.Dropout(0.5, name = 'Dropout9')(x)
    #Conv2D
    x = tf.keras.layers.Conv2D(filters=1000, kernel_size=(1,1), strides = (1,1),
padding = 'same', activation='relu', name = 'Conv2D_10')(x)
    x = tf.keras.layers.AveragePooling2D(pool_size=(13, 13), strides = (1,1), nam
e = 'MaxPool10')(x)
    SqueezeNet = tf.keras.Model(input_layer, x, name = 'SqueezeNet')
    #Load weights for SqueezeNet trained on ImageNet
    SqueezeNet.load_weights(weights_fp)
    return SqueezeNet
```

Figure 15. build_squeezenet

SqueezeNet was originally built to classify 1000 classes, so it needs to be modified to classify the AUC dataset combined into two classes. To do this, the last layer of SqueezeNet is replaced with a dense layer of two neurons with a softmax activation function. (Figure 16). Figure 17 shows a graphical representation of SqueezeNet after this modification, referred to as SqueezeNet_AUC.



```
#Modify the last layers of SqueezeNet
def modify_squeezenet(SqueezeNet,input_layer, layer_to_replace, classes):
    hidden = tf.keras.layers.Flatten()(SqueezeNet.layers[layer_to_replace].output
)
    hidden = tf.keras.layers.Dense(classes, name = 'DenseFinal', activation = 'so
ftmax')(hidden)

    SqueezeNet_X_Classes = tf.keras.Model(inputs=input_layer, outputs=hidden,name
 = 'SqueezeNet_' + str(classes) + '_Classes')

    return SqueezeNet_X_Classes
```

Figure 16. modify_squeezenet

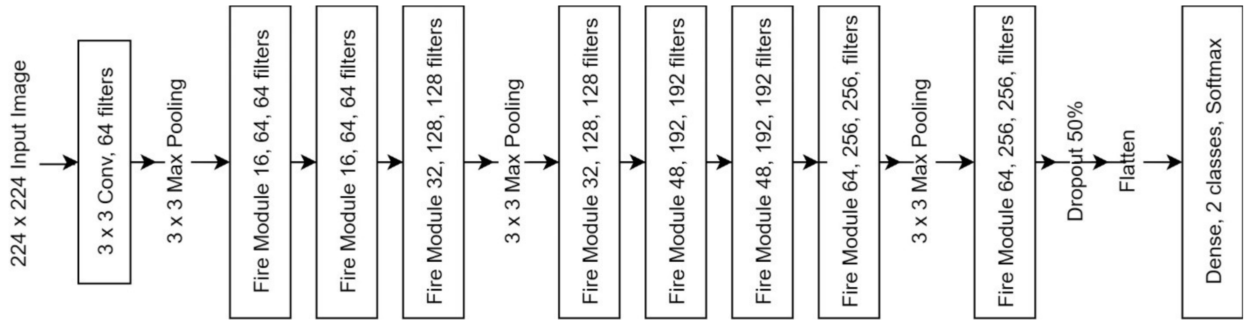
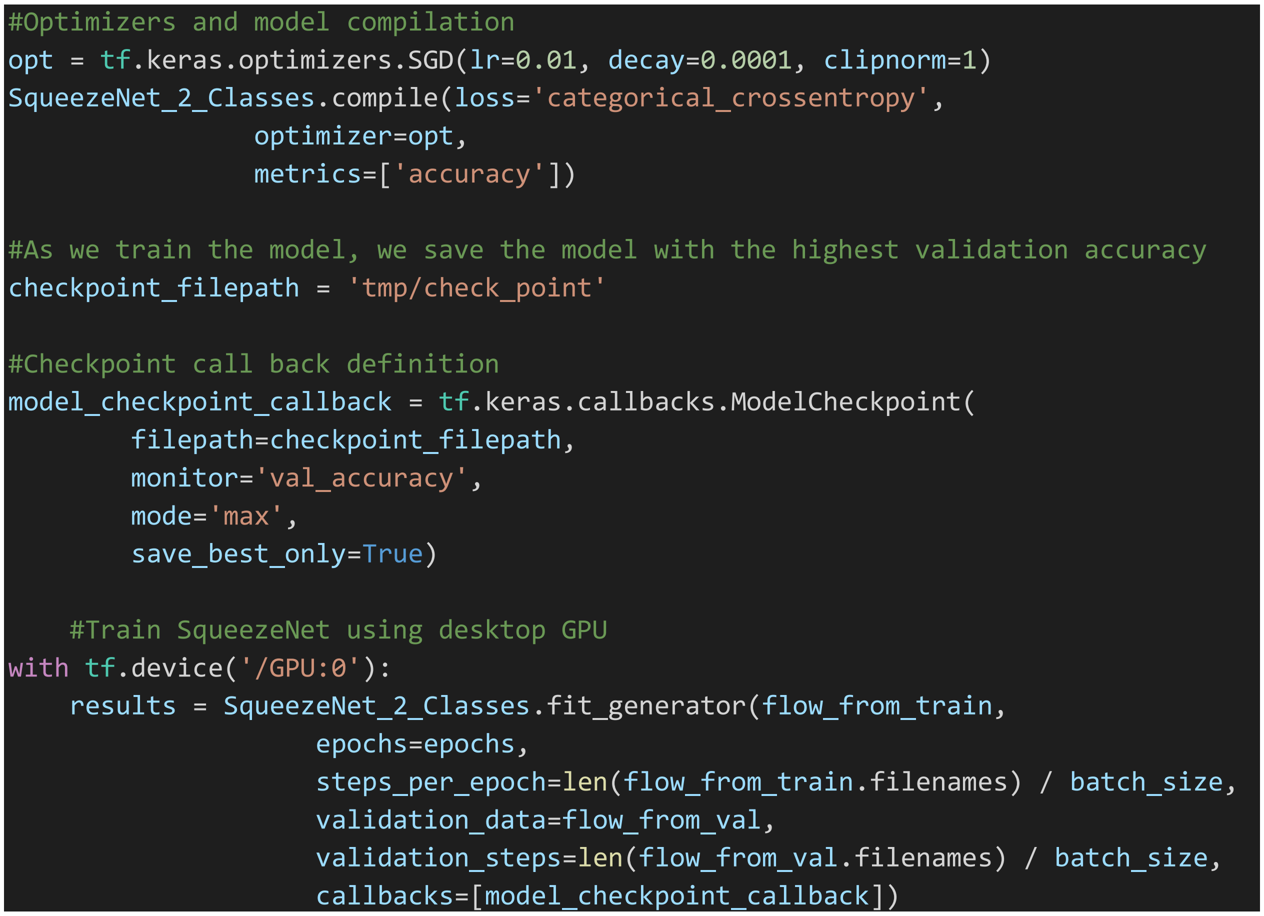

224 x 224 Input Image
3 x 3 Conv, 64 filters
3 x 3 Max Pooling
Fire Module 16, 64, 64 filters
Fire Module 16, 64, 64 filters
Fire Module 32, 128, 128 filters
3 x 3 Max Pooling
Fire Module 32, 128, 128 filters
Fire Module 48, 192, 192 filters
Fire Module 48, 192, 192 filters
Fire Module 64, 256, 256, filters
3 x 3 Max Pooling
Fire Module 64, 256, 256, filters
Dropout 50%
Flatten
Dense, 2 classes, Softmax

Figure 17. SqueezeNet_AUC

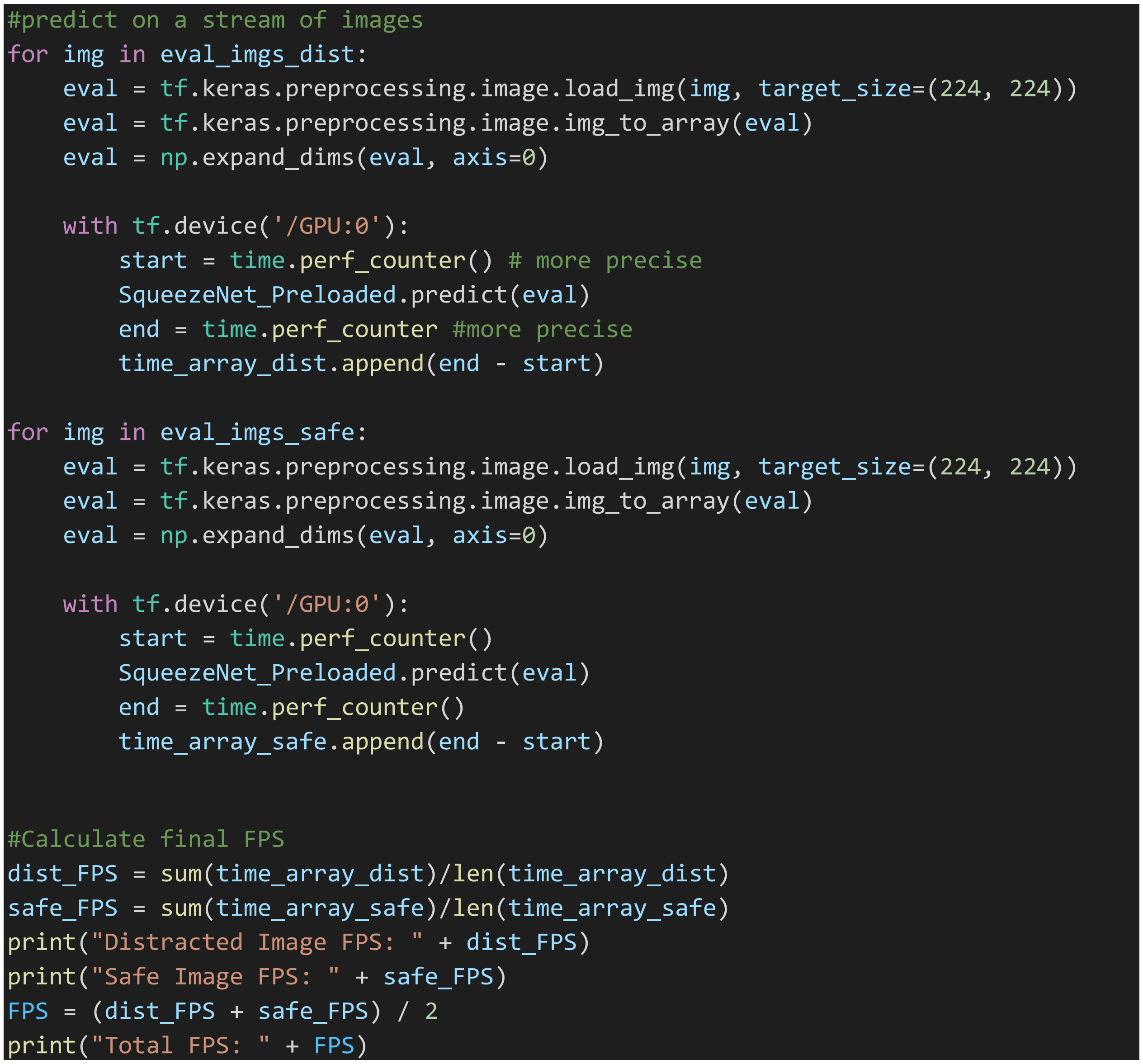
This new version of SqueezeNet is then optimized and compiled to begin training and testing. Next, training the model is done using checkpoints, the model is saved every time the highest validation accuracy over the course of 50 epochs is reached. Training is done with the GPU using TensorFlow, this increases the speed and efficiency of training the model (Figure 18).



```
#Optimizers and model compilation
opt = tf.keras.optimizers.SGD(lr=0.01, decay=0.0001, clipnorm=1)
SqueezeNet_2_Classes.compile(loss='categorical_crossentropy',
                optimizer=opt,
                metrics=['accuracy'])

#As we train the model, we save the model with the highest validation accuracy
checkpoint_filepath = 'tmp/check_point'

#Checkpoint call back definition
model_checkpoint_callback = tf.keras.callbacks.ModelCheckpoint(
        filepath=checkpoint_filepath,
        monitor='val_accuracy',
        mode='max',
        save_best_only=True)

    #Train SqueezeNet using desktop GPU
with tf.device('/GPU:0'):
    results = SqueezeNet_2_Classes.fit_generator(flow_from_train,
                    epochs=epochs,
                    steps_per_epoch=len(flow_from_train.filenames) / batch_size,
                    validation_data=flow_from_val,
                    validation_steps=len(flow_from_val.filenames) / batch_size,
                    callbacks=[model_checkpoint_callback])
```

Figure 18. SqueezeNet Training

Using the above training method is what yields the final training and validation accuracy. To measure the frames per second performance of the model on a desktop, the Python time library was used. SqueezeNet is loaded with the TensorFlow API, and it predicts on each validation image in the dataset. Each prediction's execution time is measured, and then used to calculate an average FPS value of the model on the desktop (Figure 19). This is done to benchmark the desktop performance of SqueezeNet using its TensorFlow deployment.



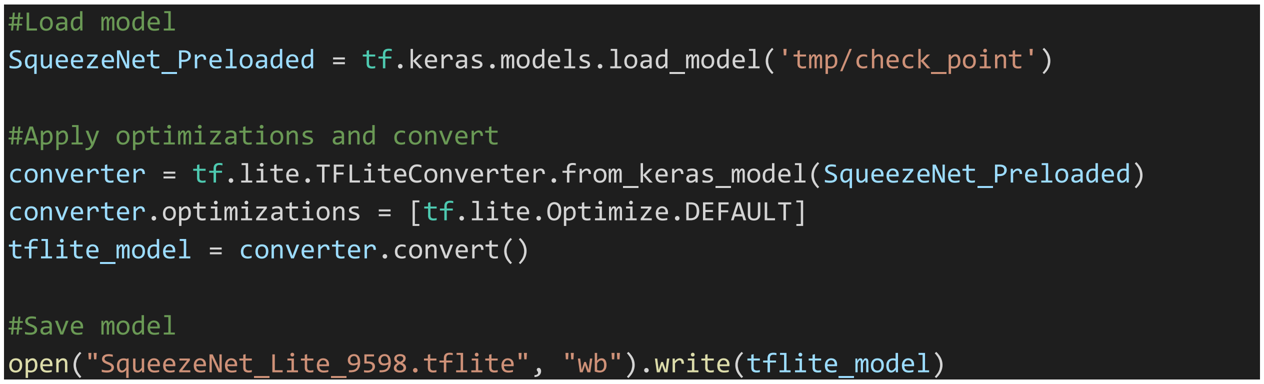
```python
#predict on a stream of images
for img in eval_imgs_dist:
    eval = tf.keras.preprocessing.image.load_img(img, target_size=(224, 224))
    eval = tf.keras.preprocessing.image.img_to_array(eval)
    eval = np.expand_dims(eval, axis=0)

    with tf.device('/GPU:0'):
        start = time.perf_counter() # more precise
        SqueezeNet_Preloaded.predict(eval)
        end = time.perf_counter #more precise
        time_array_dist.append(end - start)

for img in eval_imgs_safe:
    eval = tf.keras.preprocessing.image.load_img(img, target_size=(224, 224))
    eval = tf.keras.preprocessing.image.img_to_array(eval)
    eval = np.expand_dims(eval, axis=0)

    with tf.device('/GPU:0'):
        start = time.perf_counter()
        SqueezeNet_Preloaded.predict(eval)
        end = time.perf_counter()
        time_array_safe.append(end - start)


#Calculate final FPS
dist_FPS = sum(time_array_dist)/len(time_array_dist)
safe_FPS = sum(time_array_safe)/len(time_array_safe)
print("Distracted Image FPS: " + dist_FPS)
print("Safe Image FPS: " + safe_FPS)
FPS = (dist_FPS + safe_FPS) / 2
print("Total FPS: " + FPS)
```

Figure 19. FPS Evaluator

This software implementation builds and describes SqueezeNet on a desktop environment using a GPU, it is a good benchmark of SqueezeNet's performance, but to evaluate it in an embedded context, this software needs to be deployed onto a Jetson Nano, and its performance needs to be compared.

F. Jetson Nano Software Development

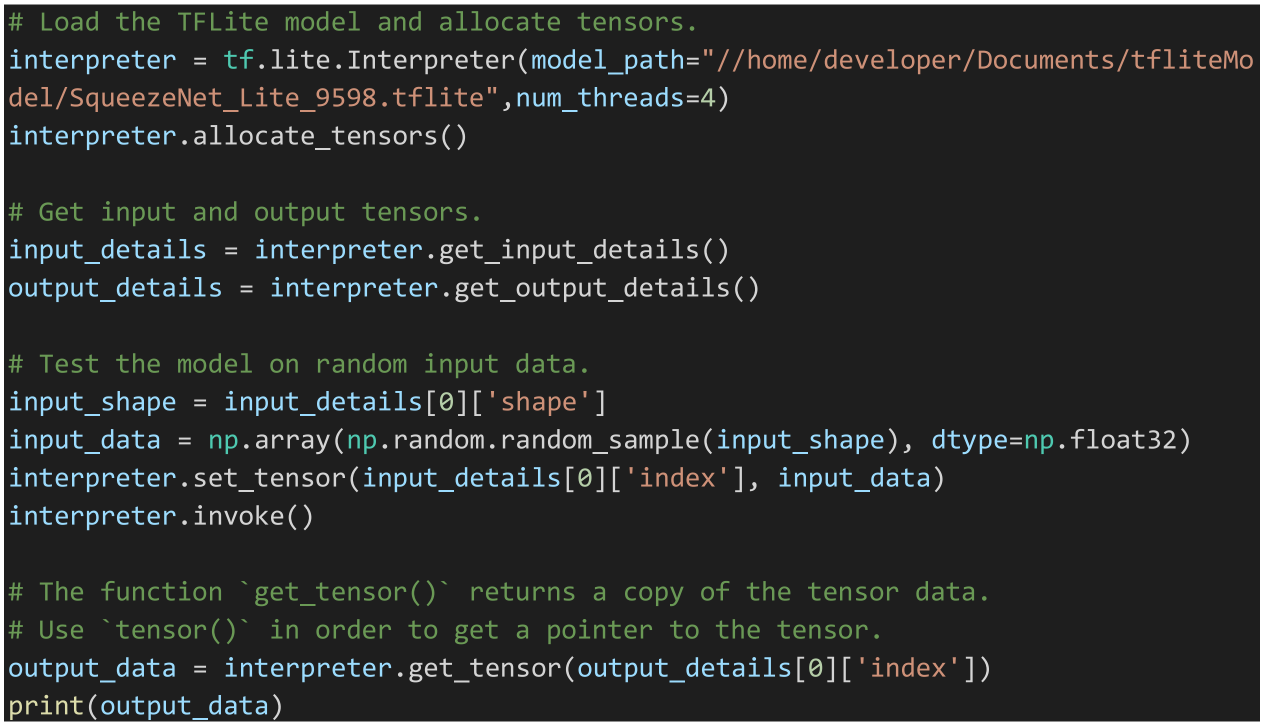
Before SqueezeNet can be deployed onto the Jetson Nano, it needs to be converted from a TensorFlow model to a TensorFlow-Lite model. The TensorFlow library also takes care of this and converts the trained SqueezeNet model created from the previous section's code using default optimizers (Figure 20).



```
#Load model
SqueezeNet_Preloaded = tf.keras.models.load_model('tmp/check_point')

#Apply optimizations and convert
converter = tf.lite.TFLiteConverter.from_keras_model(SqueezeNet_Preloaded)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
tflite_model = converter.convert()

#Save model
open("SqueezeNet_Lite_9598.tflite", "wb").write(tflite_model)
```

Figure 20. SqueezeNet Conversion

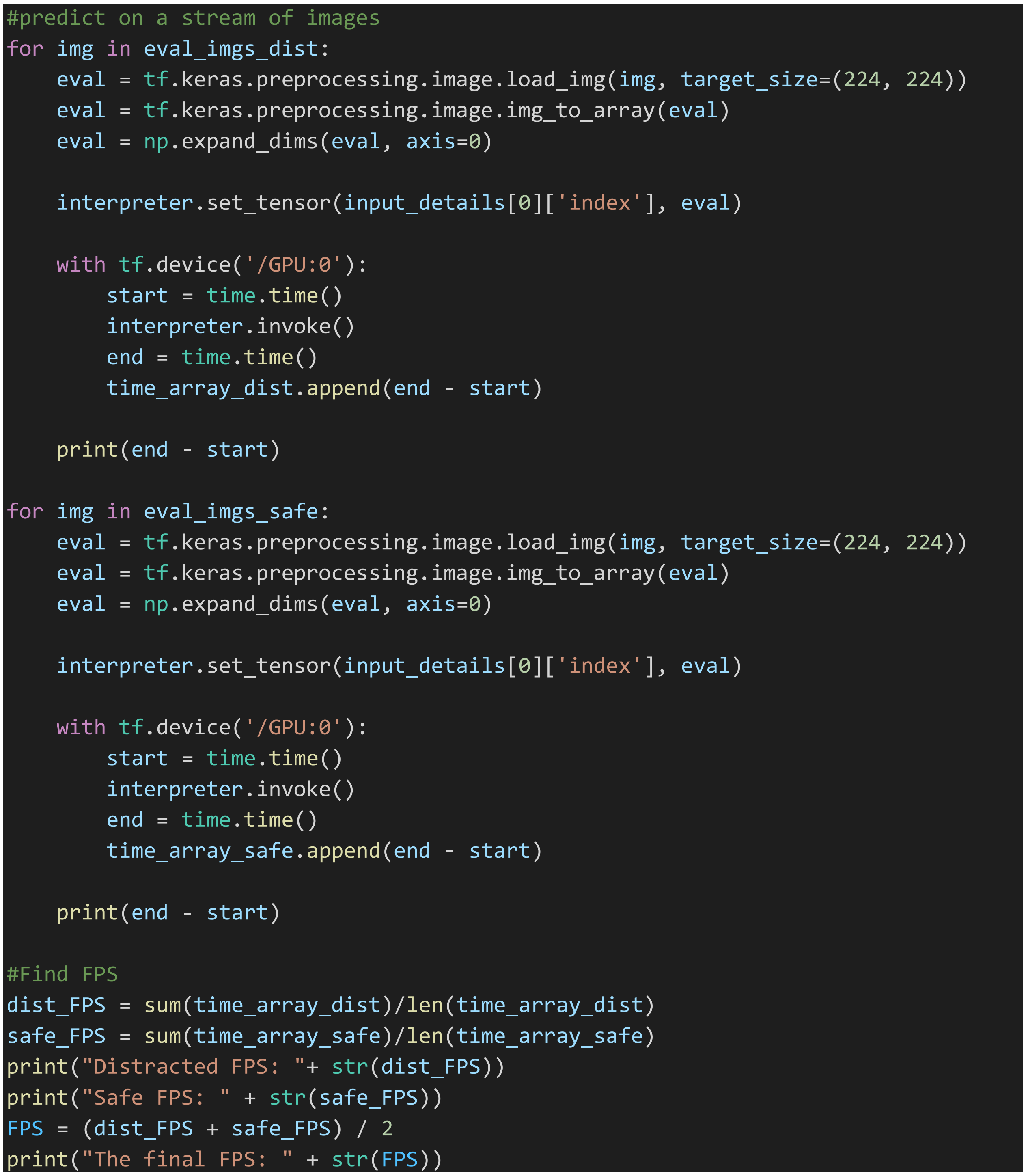
To see the frames per second performance of the TensorFlow-Lite model on the Jetson Nano, a similar method is employed as the desktop version, except the TensorFlow-Lite API is used for the prediction. To load the TensorFlow-Lite model, the saved model must be loaded into an "interpreter", this interpreter then tests the model on some random data to make sure that the model is functioning properly on the device (Figure 21).



```
# Load the TFLite model and allocate tensors.
interpreter = tf.lite.Interpreter(model_path="//home/developer/Documents/tfliteModel/SqueezeNet_Lite_9598.tflite",num_threads=4)
interpreter.allocate_tensors()

# Get input and output tensors.
input_details = interpreter.get_input_details()
output_details = interpreter.get_output_details()

# Test the model on random input data.
input_shape = input_details[0]['shape']
input_data = np.array(np.random.random_sample(input_shape), dtype=np.float32)
interpreter.set_tensor(input_details[0]['index'], input_data)
interpreter.invoke()

# The function `get_tensor()` returns a copy of the tensor data.
# Use `tensor()` in order to get a pointer to the tensor.
output_data = interpreter.get_tensor(output_details[0]['index'])
print(output_data)
```

Figure 21. TensorFlow-Lite Interpreter

To measure the FPS of the model on the Jetson Nano, the interpreter is “invoked”, and the execution time of that invocation is measured for all the validation images. A total FPS calculation is then performed (Figure 22).



```python
#predict on a stream of images
for img in eval_imgs_dist:
    eval = tf.keras.preprocessing.image.load_img(img, target_size=(224, 224))
    eval = tf.keras.preprocessing.image.img_to_array(eval)
    eval = np.expand_dims(eval, axis=0)

    interpreter.set_tensor(input_details[0]['index'], eval)

    with tf.device('/GPU:0'):
        start = time.time()
        interpreter.invoke()
        end = time.time()
        time_array_dist.append(end - start)

    print(end - start)

for img in eval_imgs_safe:
    eval = tf.keras.preprocessing.image.load_img(img, target_size=(224, 224))
    eval = tf.keras.preprocessing.image.img_to_array(eval)
    eval = np.expand_dims(eval, axis=0)

    interpreter.set_tensor(input_details[0]['index'], eval)

    with tf.device('/GPU:0'):
        start = time.time()
        interpreter.invoke()
        end = time.time()
        time_array_safe.append(end - start)

    print(end - start)

#Find FPS
dist_FPS = sum(time_array_dist)/len(time_array_dist)
safe_FPS = sum(time_array_safe)/len(time_array_safe)
print("Distracted FPS: "+ str(dist_FPS))
print("Safe FPS: " + str(safe_FPS))
FPS = (dist_FPS + safe_FPS) / 2
print("The final FPS: " + str(FPS))
```

Figure 22. Jetson Nano FPS Evaluator

Now that the implementation in software is done for both the desktop and the Jetson Nano, the results must be validated, and performance compared. This is needed to demonstrate the viability of deploying deep learning on embedded devices for driver distraction detection.

Chapter 4: Validation

Now that an implementation has been laid out for the proposed methodology, it needs to be validated in practice. This section describes the deployment of the methodology on a desktop and on embedded hardware and concludes with a performance evaluation of the deployments to contextualize them with the goals of this thesis.

A. Desktop Performance

For the performance measurements of SqueezeNet, the following parameters were measured: accuracy and frames per second performance of the model. Over the course of this research, there were many different models tested and trained using different data methods and techniques. Two models will be covered in this section, a model that was validated during training by the entire streams of images for drivers 2, 12, 14, 21 , 27, and 30, and the model that was validated using only image of each stream of each of the aforementioned drivers. The model that was validated using the entire stream of driver images for each driver will be referred to as "StreamVal", and the model that was validated using single images from each stream will be referred to as "SingleImgVal". Both were trained using the same data, and both share the same validation drivers, but SingleImgVal uses less images in validation. StreamVal uses 2461 images for validation during training from epoch to epoch, while SingleImgVal uses only 59 images for validation. Ideally, StreamVal and SingleImgVal should perform the same, and the benefit of using SingleImgVal is that training is much quicker, since there are 2402 images less to process at the end of each training epoch. StreamVal had a training accuracy of 95.98% after training, and a validation accuracy of 91.58%. SingleImgVal had a training accuracy of 93.12% after training, and a validation accuracy of 97.22%. At first, it seems that the validation accuracy is much higher in SingleImgVal, so therefore it must be the better performing model. But, when examining the confusion matrices for each of these trained models, this is not the case.

Table 4. StreamVal Validation Confusion Matrix

2461 Images		Predicted	
		Distracted Driving	Safe Driving
True Label	Distracted Driving	1728 (94%)	119 (6%)
	Safe Driving	88 (14%)	526 (86%)

In Table 4, the distracted driver accuracy is 94%, and the accuracy of detecting safe driving is 86%, this is consistent with the validation accuracy achieved during training but shows that the model has more trouble detecting safe driving than distracted.

Table 5. SingleImgVal Validation Confusion Matrix

2461 Images		Predicted	
		Distracted Driving	Safe Driving
True Label	Distracted Driving	1729 (94%)	118 (6%)
	Safe Driving	234 (38%)	380 (62%)

In Table 5, the distracted driving accuracy is 94%, and the safe driving accuracy is 62%. This is very different from the trend in the training confusion matrix of StreamVal, which yielded 94% accuracy for distracted driving, and 86% accuracy for safe driving. These numbers show that SingleImgVal performs worse, as it detects safe driving at a lower rate. Yet, this poor rate of safe driving classification was not caught during training, this may be because of the small validation set that was used. When only validating with such a small set of images, a good performance on the training data does not necessarily correspond to good performance on the larger set of validation images.

Next, to measure the FPS (frames per second) at which the model (StreamVal) operates on a desktop computer, the time it takes to predict a single image is measured using the Python "Time" library, and the average of the times measured for all the images is the frames per second of the model. Measuring FPS this way yielded 32 FPS on a NVIDIA GeForce GTX 1060 3GB GPU.

B. Jetson Nano Performance

Before deploying SqueezeNet onto the Jetson Nano hardware, the model must first be converted into a TensorFlow-Lite model. A confusion matrix of the compressed SqueezeNet Model (StreamVal) is show below.

Table 6. TensorFlow Lite Validation Confusion Matrix

2461 Images		Predicted	
		Distracted Driving	Safe Driving
True Label	Distracted Driving	1719 (93%)	128 (7%)
	Safe Driving	81 (13%)	533 (87%)

In Table 6, there is little drop in accuracy of the TensorFlow-Lite model. The validation confusion matrix is within a tolerance of 1% change in accuracy for both distracted and safe driving before compression. This shows that compression does not significantly hurt the accuracy of SqueezeNet in the case of this thesis. To measure the FPS of a TensorFlow-Lite model, a similar method is employed to the regular TensorFlow model. The Python "Time" library is used to measure how long it takes for the TensorFlow-Lite interpreter to be "invoked" and make a prediction on an image. This is done for all of the validation images, and the average is taken for a final FPS. When deploying the TensorFlow-Lite model onto the Jetson Nano, the model ran at 11 FPS. While this speed is not as fast as the desktop PC, it is comparable to other implementations of detecting driver distraction using embedded NVIDIA devices. For instance, in [7] GoogleNet was ran on a set of driver postures at a rate of 11 FPS using the NVIDIA Jetson TX1, which is a higher power embedded computer than the Nano.

C. Performance Evaluation

The trained model itself performs worse on safe driving images, this section will explore why that is, as well as possible ways to bypass this. To further deep dive as to why StreamVal has an accuracy of 86% for safe drivers, and 94% for distracted drivers, the accuracy was calculated image by image for each stream of the validation drivers. For each stream in each validation driver, a miss percentage was calculated. The percentages correspond to how many images were misclassified, i.e. 100% means 100 percent of the images were misclassified. Within this calculation, the original classes of the AUC Distracted Driver dataset are also accounted for, to potentially show more behaviors in the model.

Table 7. SqueezeNet StreamVal Stream Misclassification Percentages

	C0	**C1**	**C2**	**C3**	**C4**	**C5**	**C6**	**C7**	**C8**	**C9**
Driver 12	7	0	0	0	0	0	0	0	0	12
Driver 14	3	8	0	11	0	0	10	0	0	12
Driver 2	0	0	0	85	0	0	0	0	0	0
Driver 21	20	0	0	0	91	0	0	0	0	4
Driver 27	30	9	0	38	0	0	0	0	16	2
Driver 30	34	0	0	none	0	0	0	0	8	0

In Table 7, the six validation drivers' images are organized by class (c0 is class 0, c1 is class 1, etc.). Something that stands out is that some streams in class 3 and class 4 (touching phone and talking on phone left from original AUC dataset) have very high misclassification rates. For instance, Driver 21 class 4 has a misclassification rate of 91% (or an accuracy of only 9%). This may be happening because classes 3 and 4 looks very similar to safe driving, there are only minute differences between having both hands on the wheel and having a cell phone in one hand while the hand facing the camera is still on the wheel (Figure 23).

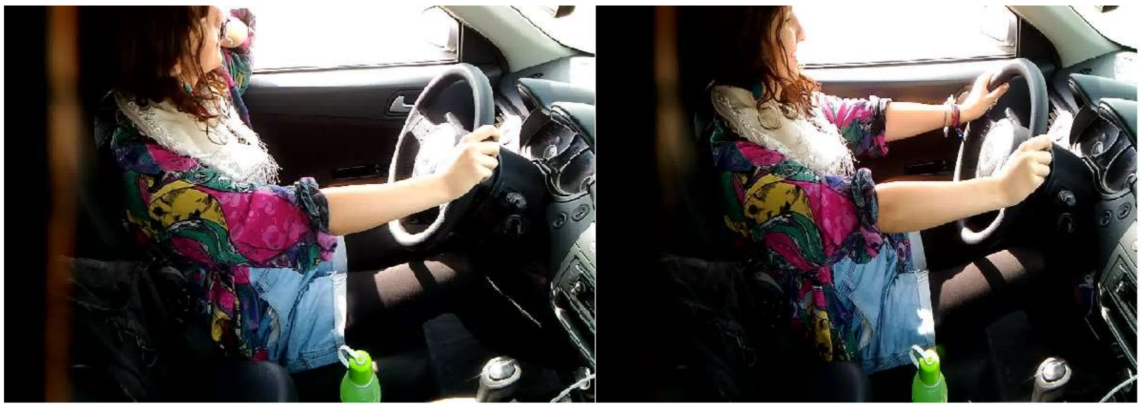


Figure 23. Driver 21 Class 4 vs Driver 21 Class 0

The safe driving streams (class 0) also have higher misclassification rates than the other classes, this may be because there are many more images of distracted drivers than there are safe drivers. This overrepresentation causes the model to be good at classifying distracted drivers, but not as good at classifying safe drivers.

To remedy these issues in the validation dataset, more safe driving images would be required. These images could be obtained organically through just capturing more images of more drivers being safe, or through further image augmentation to artificially create more safe driving data than distracted data. More safe driving images would allow for higher rates of classification of safe driving, and also allow the model to differentiate classes that are close to safe driving (classes 3 and 4) better.

Chapter 5: Conclusion

In conclusion, embedded driver distraction detection is very feasible based on the results of this thesis. Small neural nets like SqueezeNet are capable of classifying images of drivers in various poses to accurately classify distracted driving and safe driving. Embedded deployment of these networks for distraction detection is also possible since network compression techniques result in little loss in accuracy (<1 %). The performance of the models on the embedded devices is also promising, since even on a Jetson Nano, which cannot run TensorFlow-Lite models to their full potential, SqueezeNet can operate at 11 FPS. The accuracy of SqueezeNet on the AUC Distracted Driver Dataset was also comparable to other implementations on the AUC Distracted Driver Dataset. The distracted accuracy of 94% was comparable to accuracies on individual classes in [2], the original implementation of the dataset. Overall, this thesis completes what it sets out to do, it shows that an embedded driver distraction system using small deep learning networks is possible.

References

[1] National Center for Statistics and Analysis, “Distracted Driving 2018,” *DOT HS 812 926*, no. April, pp. 1–7, 2020, [Online]. Available: https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812926.

[2] Y. Abouelnaga, H. M. Eraqi, and M. N. Moustafa, “Real-time distracted driver posture classification,” *arXiv*, no. NIPS, 2017, [Online]. Available: https://arxiv.org/abs/1706.09498.

[3] F. Vicente, Z. Huang, X. Xiong, F. De La Torre, W. Zhang, and D. Levi, “Driver gaze tracking and eyes off the road detection system,” *IEEE Trans. Intell. Transp. Syst.*, vol. 16, no. 4, pp. 2014–2027, 2015, doi: 10.1109/TITS.2015.2396031.

[4] C. Streiffer, R. Raghavendra, T. Benson, and M. Srivatsa, “DarNet: A deep learning solution for distracted driving detection,” *Middlew. 2017 - Proc. 2017 Int. Middlew. Conf. (Industrial Track)*, pp. 22–28, 2017, doi: 10.1145/3154448.3154452.

[5] F. Tango and M. Botta, “Real-time detection system of driver distraction using machine learning,” *IEEE Trans. Intell. Transp. Syst.*, vol. 14, no. 2, pp. 894–905, 2013, doi: 10.1109/TITS.2013.2247760.

[6] Y. Liang, M. L. Reyes, and J. D. Lee, “Real-time detection of driver cognitive distraction using support vector machines,” *IEEE Trans. Intell. Transp. Syst.*, vol. 8, no. 2, pp. 340–350, 2007, doi: 10.1109/TITS.2007.895298.

[7] D. Tran, H. M. Do, W. Sheng, H. Bai, and G. Chowdhary, “Real-time detection of distracted driving based on deep learning,” *IET Intell. Transp. Syst.*, vol. 12, no. 10, pp. 1210–1219, 2018, doi: 10.1049/iet-its.2018.5172.

[8] R. Gupta, P. Mangalraj, A. Agrawal, and A. Kumar, “Posture recognition for safe driving,” *Proc. 2015 3rd Int. Conf. Image Inf. Process. ICIIP 2015*, pp. 141–146, 2016, doi: 10.1109/ICIIP.2015.7414755.

[9] Y. Xing *et al.*, “Identification and Analysis of Driver Postures for In-Vehicle Driving Activities and Secondary Tasks Recognition,” *IEEE Trans. Comput. Soc. Syst.*, vol. 5, no. 1, pp. 95–108, 2018, doi: 10.1109/TCSS.2017.2766884.

[10] Y. Ma, Z. Yin, and L. Nie, “Driver distraction detection with a two-stream convolutional neural network,” *SAE Tech. Pap.*, vol. April, no. April, pp. 1–8, 2020, doi: 10.4271/2020-01-1039.

[11] F. Iandola and K. Keutzer, “Keynote ESWEEK 2017: Small Neural Nets are beautiful: Enabling embedded systems with small Deep-Neural-Network architectures,” *arXiv*, 2017, [Online]. Available: https://arxiv.org/abs/1710.02759.

[12] G. Cerutti, R. Prasad, and E. Farella, "Convolutional neural network on embedded platform for people presence detection in low resolution thermal images," *ICASSP, IEEE Int. Conf. Acoust. Speech Signal Process. - Proc.*, vol. May, pp. 7610–7614, 2019, doi: 10.1109/ICASSP.2019.8682998.

[13] B. Reddy, Y. H. Kim, S. Yun, C. Seo, and J. Jang, "Real-Time Driver Drowsiness Detection for Embedded System Using Model Compression of Deep Neural Networks," *IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, vol. July, pp. 438–445, 2017, doi: 10.1109/CVPRW.2017.59.

[14] F. N. Iandola, S. Han, M. W. Moskewicz, K. Ashraf, W. J. Dally, and K. Keutzer, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <0.5MB model size," pp. 1–13, 2016, [Online]. Available: http://arxiv.org/abs/1602.07360.

[15] B. A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2012, [Online]. Available: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[16] Forresti (2018) SqueezeNet [SqueezeNet_v1.1] https://github.com/forresti/SqueezeNet

[17] Jetson nano developer kit. (2021, January 28). Retrieved March 09, 2021, from https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[18] R. Stumpf, "People Keep Coming Up With Simple Ways to Fool Tesla's Autopilot - The Drive," *The Drive*. 2018, [Online]. Available: https://www.thedrive.com/sheetmetal/18168/people-keep-coming-up-with-ways-to-fool-teslas-autopilot.

[19] G. Cadillac, "Super Cruise: Hands-Free Driving, Cutting Edge Technology." 2018, [Online]. Available: https://www.cadillac.com/ownership/vehicle-technology/super-cruise.

[20] NVIDIA Corporation. NVIDIA Jetson Nano Product Design Guide (2020) Accessed: 2021 [Online]. Available: https://developer.nvidia.com/jetson

[19] A. Wong, M. J. Shafiee, and M. St Jules, "MicronNet: A highly compact deep convolutional neural network architecture for real-time embedded traffic sign classification," *IEEE Access*, vol. 6, pp. 59803–59810, 2018, doi: 10.1109/ACCESS.2018.2873948.

[21] A. Agarwal *et al.*, "TensorFlow : large-scale machine learning on heterogeneous distributed systems," 2015.

[22] M. Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016, [Online]. Available: http://arxiv.org/abs/1603.04467.

[23] R.Manoharan and S.Chandrakala, "Distraction monitoring system," *2015 Int. Conf. Comput. Commun. Technol.*, pp. 262–266, 2015.

[24] G. Van Rossum, & F. L. Drake (2009). Python 3 Reference Manual. Scotts Valley, CA: CreateSpace.

[25] M. Kutila, M. Jokela, G. Markkula, and M. R. Rué, “Driver distraction detection with a camera vision system,” *Proc. - Int. Conf. Image Process. ICIP*, vol. 6, no. 3 1, pp. 201–204, 2006, doi: 10.1109/ICIP.2007.4379556.

[26] R. O. Mbouna, S. G. Kong, and M. G. Chun, “Visual analysis of eye state and head pose for driver alertness monitoring,” *IEEE Trans. Intell. Transp. Syst.*, vol. 14, no. 3, pp. 1462–1469, 2013, doi: 10.1109/TITS.2013.2262098.

[27] V. Rathod and R. Agrawal, “Camera based driver distraction system using image processing,” *Proc. - 2018 4th Int. Conf. Comput. Commun. Control Autom. ICCUBEA 2018*, pp. 1–6, 2018, doi: 10.1109/ICCUBEA.2018.8697463.

[28] C. Zhao, Y. Gao, J. He, and J. Lian, “Recognition of driving postures by multiwavelet transform and multilayer perceptron classifier,” *Eng. Appl. Artif. Intell.*, vol. 25, no. 8, pp. 1677–1686, 2012, doi: 10.1016/j.engappai.2012.09.018.

[29] V. Tamas and V. Maties, “Real-time distracted drivers detection using deep learning,” *Am. J. Artif. Intell.*, vol. 3, no. 1, p. 1, 2019, doi: 10.11648/j.ajai.20190301.11.

[30] C. Yan, F. Coenen, and B. Zhang, “Driving posture recognition by convolutional neural networks,” *IET Comput. Vis.*, vol. 10, no. 2, pp. 103–114, 2016, doi: 10.1049/iet-cvi.2015.0175.

[31] M. Alotaibi and B. Alotaibi, “Distracted driver classification using deep learning,” *Signal, Image Video Process.*, vol. 14, no. 3, pp. 617–624, 2020, doi: 10.1007/s11760-019-01589-z.

[32] B. Baheti, S. Gajre, and S. Talbar, “Detection of distracted driver using convolutional neural network,” *IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit. Work.*, vol. June, pp. 1145–1151, 2018, doi: 10.1109/CVPRW.2018.00150.

[33] K. Okuno *et al.*, “Body posture and face orientation estimation by convolutional network with heterogeneous learning,” *2018 Int. Work. Adv. Image Technol. IWAIT 2018*, pp. 1–4, 2018, doi: 10.1109/IWAIT.2018.8369677.

[34] M. Bettoni, G. Urgese, Y. Kobayashi, E. Macii, and A. Acquaviva, “A convolutional neural network fully implemented on FPGA for embedded platforms,” *Proc. - 2017 1st New Gener. CAS, NGCAS 2017*, no. May, pp. 49–52, 2017, doi: 10.1109/NGCAS.2017.16.

[35] D. Gutierrez-Galan *et al.*, “Embedded neural network for real-time animal behavior classification,” *Neurocomputing*, vol. 272, pp. 17–26, 2018, doi: 10.1016/j.neucom.2017.03.090.

[36] S. A. Manzano, D. T. Hughes, C. R. Simpson, R. Patel, and N. Correll, “Embedded neural networks for robot autonomy,” *arXiv*, pp. 1–16, 2019, [Online]. Available: https://arxiv.org/abs/1911.03848.