Traffic Sign Recognition (TSR) System

Prepared by:

Pavly Salah Zaki

Marco Magdy William

Bolis Karam Soliman

Kerolos Gamal Alexsan

Under the most appreciated supervision of:

Dr. Maher Mansour from FEHU
Dr. Magdy El-Moursy,
Eng. Kerolos Karam,
and Eng. Mark Magdy
from *Mentor Graphics*

Table of Contents

Chapter 1: Introduction

Computer Vision

With the amazing sensory of vision, it was about time that humans started wanting to capture specific moments and savor them, thus came the brilliant invention of cameras.

A statistic by YouTube in 2017 showed that on average **300 hours** of **video are uploaded** to **YouTube every minute**! This means that while you were reading this paragraph, a few days' worth of content has been uploaded to YouTube!

Therefore, it was critical for Machine Learning engineers and Data Scientists and Analysists to develop algorithms that can understand and utilize this visual data.

Computer Vision Applications [1]

1- Biological and Medical Examinations
2- Face recognition for various security systems
3- Autonomous Cars and Traffic Sign Recognition
4- Auto-captioning videos
5- Google's Image search

And many others…

Fig. 1 Some Computer Vision applications

Problem Definition

With the rapid technological advancement, automobiles have become a crucial part of our day-to-day lives. This makes the road traffic more and more complicated, which led to more traffic accidents every year. According to the Association for Safe International Road Travel (ASIRT) organization, about 1.3 million people die (including 1,600 children under 15 years of age!), and about 20-50 million are injured or disabled annually due to traffic accidents. **[2]**

There are numerous reasons that lead to those horrifying numbers of road accidents: according to San Diego Personal Injury Law Offices, the leading causes for such traumatic accidents are distracted driving and speeding. **[3]** Hence, a serious and immediate action needed to be taken, and that action was the Advanced Driver Assistant System (ADAS). **[4]** ADAS refer to high-tech in-vehicle systems that are designed to increase road safety by alerting the driver of hazardous road conditions. Examples of the crucial ADAS sub-systems are Lane Departure, Collision Avoidance, and ***Traffic Signs Recognition (TSR)***. Recently, Traffic Signs Recognition has become a hot and active research topic due to its importance; there are various difficulties presented to the drivers that hinder their ability to properly see the traffic signs. Some of those difficulties are shown in Fig. 1. Hence it was necessary to automate the traffic signs detection and recognition process efficiently.

The Traffic Signs Detection and Recognition (TSDR) System is a system that enables cars to detect various traffic signs and behave accordingly. For example, if the car detects a speed limit sign, it will monitor its current speed, notify the driver of the detected sign and change its speed accordingly.

There are two main components using which we will make our project: Deep Learning and its implementation. The implementation was done on the Imx6Q board, Raspberry Pi 3 Model B+, and TASS PreScan **[5]**.

All will be described explicitly in the next few sections of this document.

Project Overview

The main functions of the TSR system

- Display urgent contents on the road to the driver
- Warn the driver to drive at the prescribed speed
- Warn the driver of road-crossing sections
- and overall provide a favorable guarantee for safe driving

Problems that might face the TSR systems

- **Lighting conditions** are changeable, according to the time of day, the weather conditions (e.g., sunny, sandy, foggy, snowy), etc.
- **The presence of other objects** in the scene (e.g., moving cars, bicycles, pedestrians, shop signs, etc.) can partially occlude the visibility of a sign or cause ambiguity.
- The sign's appearance can change because of human vandalism or the variations respect to size, **visibility angle** and position in the image.
- **The long exposure to the sunlight** and the reaction of the paint with the air provoke that the color of traffic signs often gradually fades.
- If the image is acquired from a moving car, then it often suffers from **motion blur** and car vibration. For these reasons, the reliable detection of traffic signs from such varied scenes becomes rather challenging.

Fig. 2 Difficulties that might face the TSR system [6]

Thus, a simple classification algorithm like template matching will not able to achieve high-quality recognition because of the limited set of predefined templates and a "deeper" more advanced object detection and recognition algorithm had to be used – ***Deep Convolutional Neural Network***. **[7]**

Categories of Traffic Signs

The most essential types of traffic signs are ***Prohibitory***, ***Warning***, ***Mandatory*** and ***Informative*** signs. Note also that different signs may be adopted in different countries.

Generally speaking, the traffic signs are well-designed, by using particular colors (***yellow***, ***red***, ***blue***, ***white*** and ***black***) and shapes (***triangle***, ***rectangle***, ***circle*** and ***octagon***), to attract the drivers' attention against the natural environment for the purpose of benefiting the problem of traffic sign recognition.

Category	Shape	Color	Example
Prohibitory	Circular	Red, Blue, White & Black	
Mandatory	Rectangular & Circular	Blue, White & Black	
Danger	Triangular	Red, White & Black	

Fig. 3 categories of traffic signs according to the GTSDB, which will be talked about in the next section

Dataset(s)

In order to train a Neural Network (section B), one has to feed it a large number of images to train on, this is called a "dataset."

The datasets used in the TSR system are the ***GTSRB*** (German Traffic Sign Recognition Benchmark) **[8]** for classification and ***GTSDB*** (German Traffic Sign Detection Benchmark respectively) **[9]** for real-time detection and recognition.

GTSRB dataset is used for training and testing the Traffic Signs Recognition System algorithm (Classification part). The dataset includes over 50, 000 images and 43 classes of complex traffic signs in extreme conditions such as tilt, uneven lighting, distortion, similar background colors, and extreme weather conditions. (fig. 4)

Fig. 4 pictures that can be founded in the GTSRB dataset

GTSDB dataset is used for detecting a traffic sign during real-time testing. It contains about 900 images and 43 classes (same as the GTSRB but with much, much less data) (fig. 3)

Fig. 3 Images that can be found in the GTSDB dataset

Benefits of Using GPUs compared to CPUs

One of the main problems that might face the TSR system is the high demand for hardware computational performances. This could be solved by using Nvidia's technology CUDA (Compute Unified Device Architecture) which is a parallel computing platform and programming model created by NVIDIA and implemented by the graphics processing units (GPUs) that they produce. This is why deep learning models need very high-end GPUs with as much VRAM and CUDA cores as possible, and it's highly ill-advised to test let alone train a deep learning model on a CPU.

Hardware	Training	Classifying an image (64x64)
Nvidia GeForce GTX 650	7 min	0.05 ms
Nvidia GeForce GT 650M	12 min	0.14 ms
Intel Core i7	16 min	0.37 ms

Table 1. Difference a GPU makes in classification time

With GPUs, it is estimated that the acceleration reaches 60-80 times as compared with conventional executing on a CPU. This statistic was measured on a frame of size 1920x1080 pixels.

TSR System Host PC Specifications

All the models mentioned in the next few section were trained and initially tested on a PC of specifications listed in Table 2.

	Specifications
CPU	I7 6700K Quad-core 3.2GHz
RAM	16GB DDR3
GPU	Nvidia GTX 1070 6GB
OS	Windows 10 Pro

Table 2. PC specs

Chapter 2: Problem Solution (Deep Learning)

Artificial Neural Networks

Generally speaking, an Artificial Neural Network (ANN) is a collection of connected and tunable units (a.k.a. nodes, or neurons) which can pass a signal (usually a real-valued number) from a unit to another. The number of (layers of) units, their types, and the way they are connected to each other is called the ***network architecture.***

A neural network is a mathematical model based on connected via each other neural units – artificial neurons – similar to biological neural networks. Typically, neurons are organized in layers, and the connections are established between neurons from only adjacent layers. The input low-level feature vector is put into first layer and, moving from layer to layer, is transformed to the high-level features vector.

Notice that the output layer neurons amount is equal to the number of classifying classes. Thus, the output vector is the vector of probabilities showing the possibility that the input vector belongs to a corresponding class.

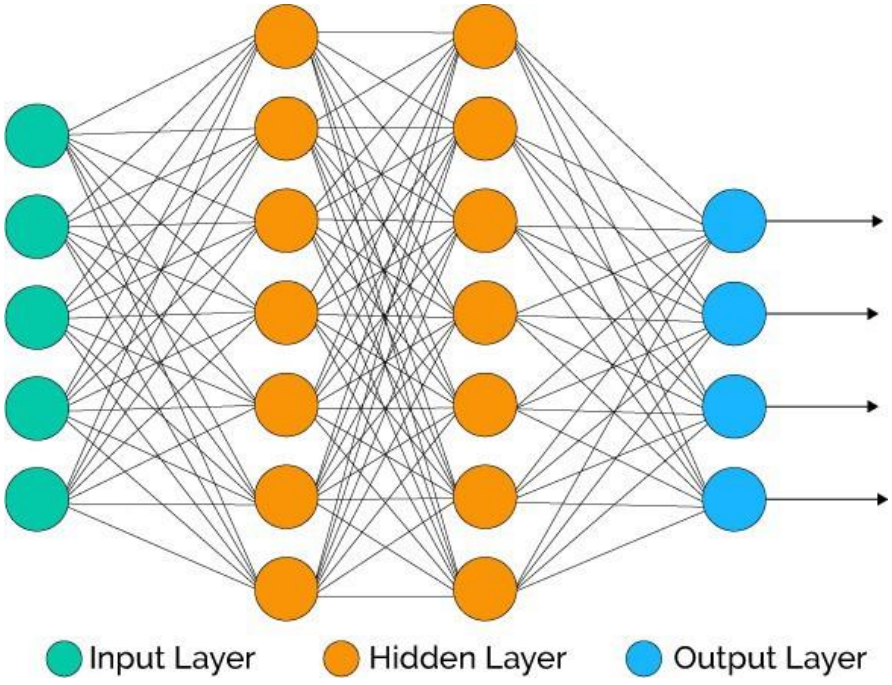

Fig. 4 Artificial Neural Networks

So, what is a Neuron?

An **artificial neuron** is a mathematical function conceived as a model of biological neurons, a neural network. Artificial neurons are elementary units in an artificial neural network. The artificial neuron receives one or more inputs and sums them to produce an output (or activation, representing a neuron's action potential which is transmitted along its axon). Usually each input is separately weighted, and the sum is passed through a non-linear function known as an **activation function** or **transfer function**. The transfer functions usually have a sigmoid shape, but they may also take the form of other non-linear functions, piecewise linear functions, or step functions.

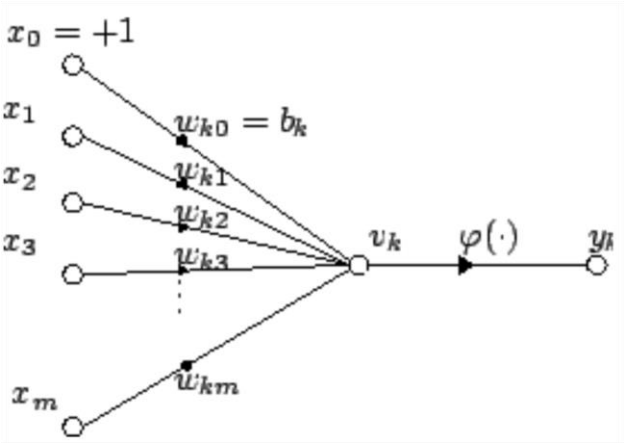

Fig. 5 an artificial neuron

$$y_k = \varphi\left(\sum_{j=0}^{m} w_{kj} x_j\right)$$

Eqn. 1 Where Φ is the transfer function.

The output is analogous to the axon of a biological neuron, and its value propagates to the input of the next layer, through a synapse. It may also exit the system, possibly as part of an output vector. Its transfer function weights are calculated and threshold value are predetermined.

Convolutional Neural Networks

Nowadays, more and more object recognition tasks are being solved with Convolutional Neural Networks (CNN). Due to its high recognition rate and fast execution, the convolutional neural networks have enhanced most of computer vision tasks, both existing and new ones.

A CNN, in specific, has one or more layers of ***convolution*** units. A convolution unit receives its input from multiple units from the previous layer which together create a proximity. Therefore, the input units (that form a small neighborhood) share their weights.

They are great for capturing local information (e.g. neighbor pixels in an image or surrounding words in a text) as well as reducing the complexity of the model (faster training, needs fewer samples, reduces the chance of overfitting).

The convolution units (as well as pooling units) are especially beneficial as they reduce the number of units in the network (since they are *many-to-one mappings*). This means, there are fewer parameters to learn which reduces the chance of ***overfitting*** as the model would be less complex than a fully connected network.

In this article, we propose several implementations for traffic signs detection and recognition algorithms using multiple convolution neural network models. Training of the models is implemented using the TensorFlow library for real-time detection. The experimental results confirmed high efficiency of the developed computer vision system.

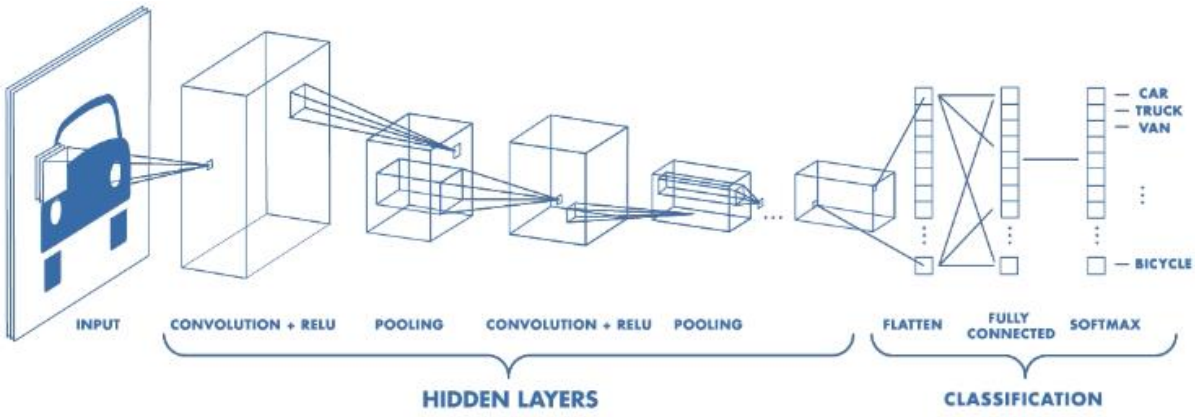

Fig. 6 CNN architecture

Convolutional Neural Networks (CNN) vs. Recurrent Neural Networks (RNN)

CNN:

1. CNN take a **fixed size** input and generate fixed-size outputs.
2. CNN is a type of feed-forward artificial neural network - are variations of multilayer perceptron which is designed to use minimal amounts of preprocessing.
3. CNNs use connectivity pattern between its neurons is inspired by the organization of the animal visual cortex, whose individual neurons are arranged in such a way that they respond to overlapping regions tiling the visual field.
4. **CNNs are ideal for images and videos processing.**

RNN:

1. RNN can handle arbitrary input/output lengths.
2. RNN, unlike feedforward neural networks, can use their internal memory to process arbitrary sequences of inputs.
3. Recurrent neural networks use time-series information (i.e. what I spoke last will impact what I will speak next.)
4. **RNNs are ideal for text and speech analysis.**

Of course there are multiple types of Neural Networks, but we are going to focus on Convolution Neural Networks since it is the type we used in our system. [10]

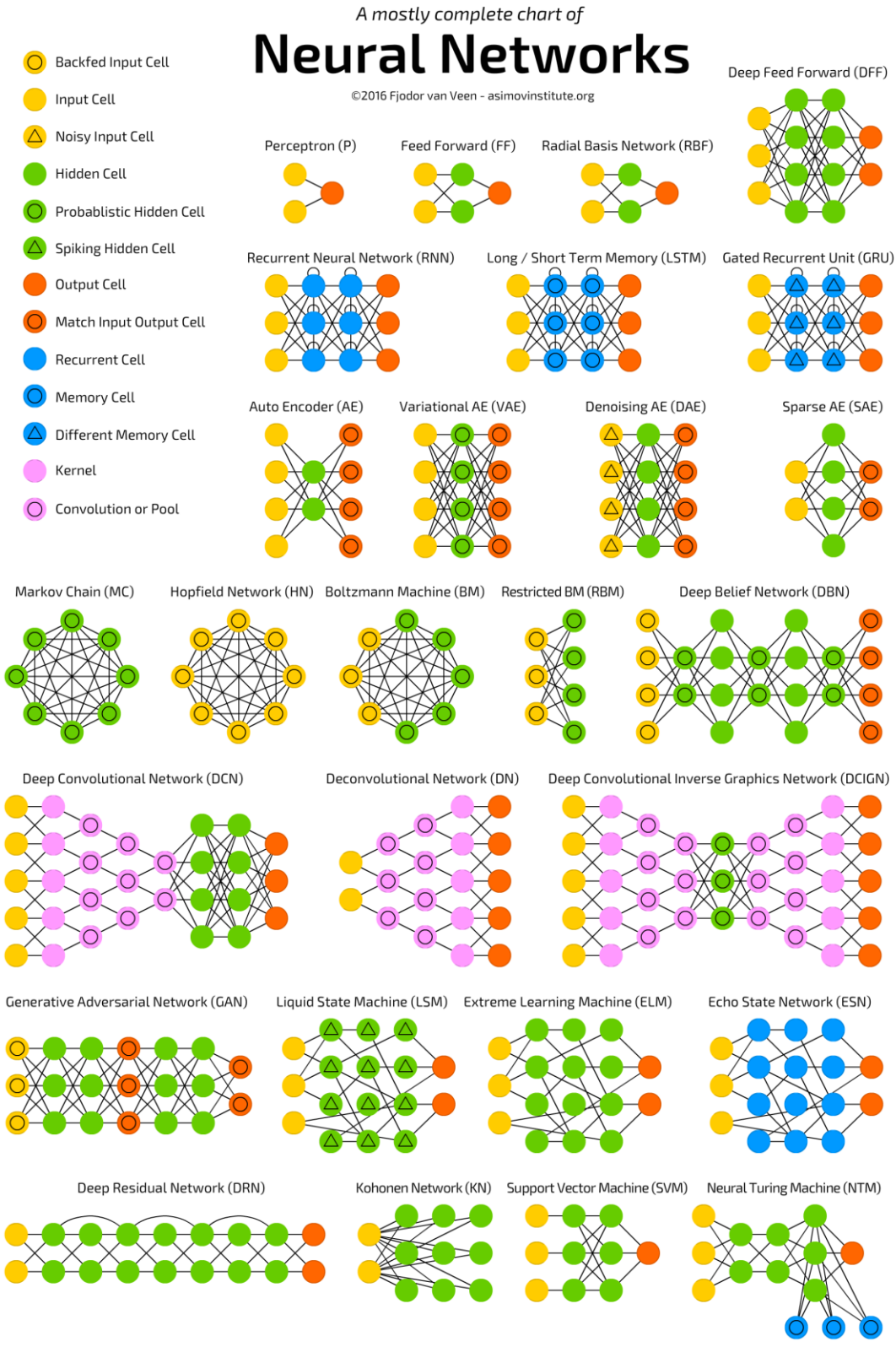

Fig. 7 types of Neural Networks

How do Neural Networks work?

If you have 2000 examples of faces, each one being one of your 50 friends, you would, in sequence, "train" the network on each example. To train it, you set the input nodes to be the pixel values of the image and you "run the network forward" propagating the pixel values through the connection weights ("w") to the hidden nodes, and then from the hidden nodes to the output nodes. The output node with the highest value, say node #28, is whose face the network thinks is in the image, e.g. person #28.

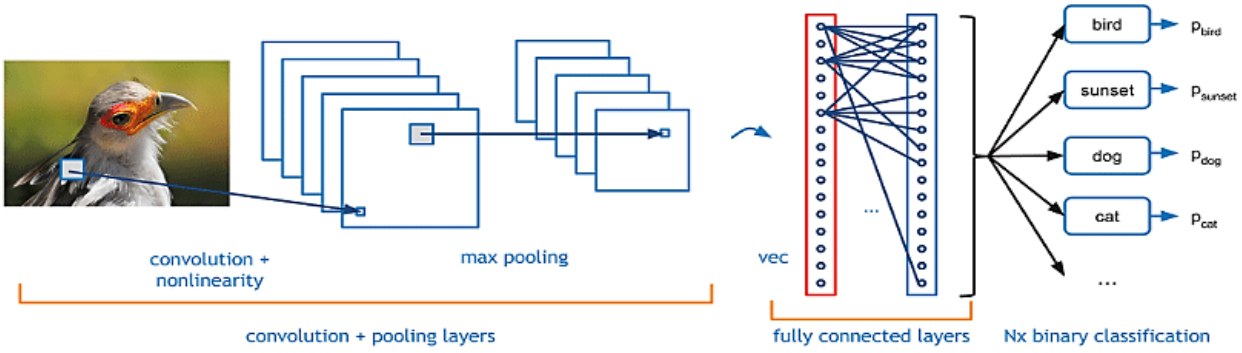

Fig. 8 NN classifying an input image of a bird

Training the network

In the beginning, the network is fed an image and is told to classify it, but it will of course get the wrong answer because it knows nothing about the given image or object. This is where the "training" and "backpropagation" comes in.

To train the network on the aforementioned face recognition example, the correct output is compared to what actually came out of the network. The correct output would be *"1.0" for node #28 and "0.0" for all other nodes*, meaning the face was of person #28 and was *not* of any of the other people. The difference between what the network said and the correct answer is the "error." The error values are propagated backward through the network using some complicated math that tells the algorithm how to modify each connection weight so that the network will get closer to the correct answer next time.

The training process is repeated over and over for all the images until the network is doing a reasonably correct job of matching the face images to the identity of the person. This is called *training iterations*.

Once the network is trained, you can "*show*" the network a new face image it has never seen before, and you hope that it will correctly guess whose face is in the image. This is a property called "***generalization***" which means that the network can generalize what it learned from the specific examples to do the right thing for inputs it has never seen before.

Generalization is often determined by the number of nodes in the hidden layer. If you have a very large number of hidden nodes, the network will often "***overfit***" and not generalize. For example, if there were 2000 hidden nodes, each hidden node might simply learn which of the 2000 examples was seen and map them to outputs "by rote" with no generalization. However, with only 200 hidden nodes, the network would be forced to find some commonalities between the input images of the same person.

Deep Learning Terminology

1. **Layers**
2. **Activation Functions**
3. **Loss Functions**
4. **Learning Rate**
5. **Optimizers**
6. **Over-fitting**
7. **Under-fitting**
8. **Adequate-fitting**
9. **Generalization**
10. **Regularization**
11. **Early-stopping**
12. **Cross-validation**
13. **Image Augmentation**
14. **Feature Maps**
15. **Transfer Learning**

CNN – Architecture (Layers)

Computers see an input image as array of pixels and it depends on the image resolution. Based on the image resolution, it will see h x w x d (h = Height, w = Width, d = Dimension).

Eg., An image of 6 x 6 x 3 array of matrix of RGB (3 refers to RGB values) and an image of 4 x 4 x 1 array of matrix of grayscale image.

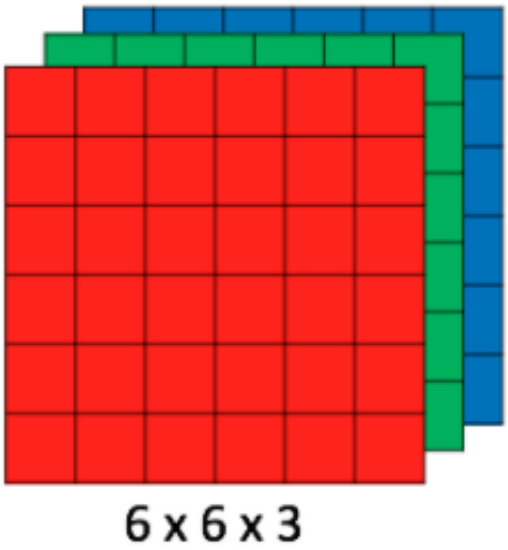

Fig. 9 a 6x6 3-channel (RGB) image

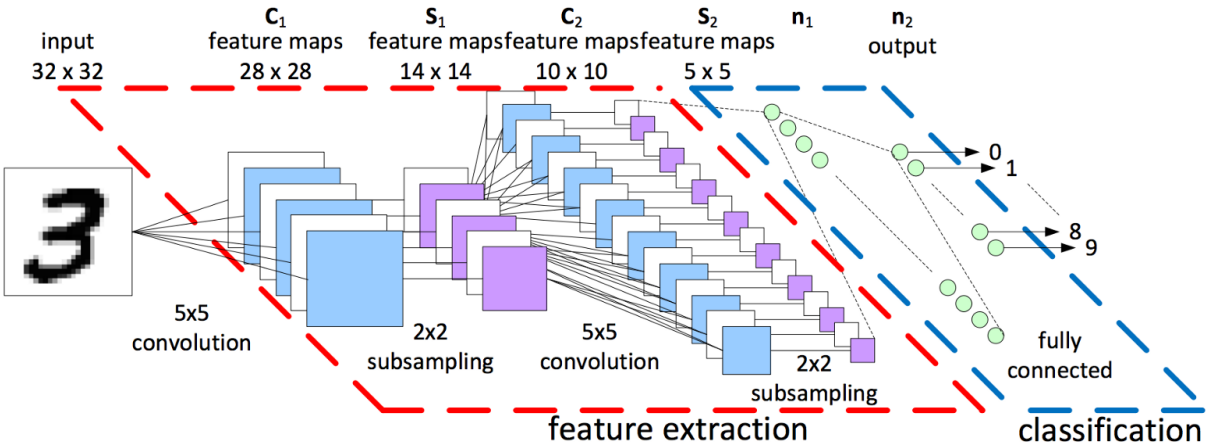

Fig. 10: CNN feature extraction and recognition process

1. Convolutional Layer (Conv2D)

Convolution is the first layer to extract features from an input image. Convolution preserves the relationship between pixels by learning image features using small squares of input data. It is a mathematical operation that takes two inputs such as image matrix and a filter or kernel.

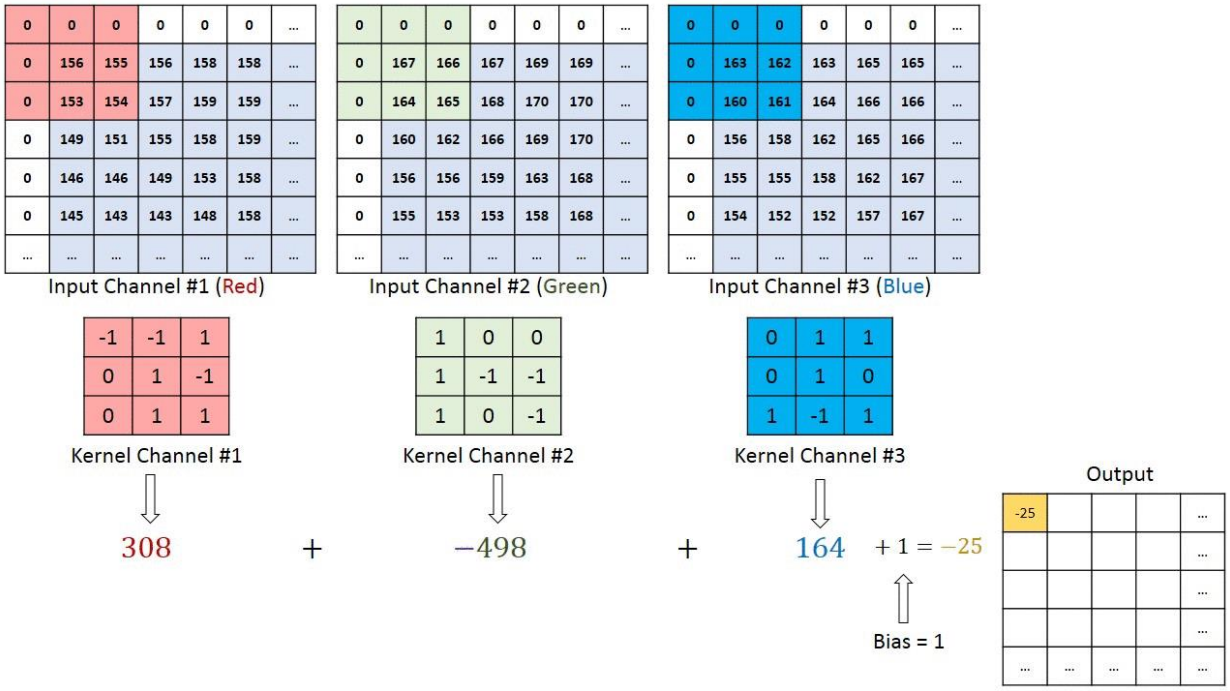

Fig. 11 how the Convolutional layer works

Stride *is the size of the step the convolution filter moves each time*. A stride size is usually 1, meaning the filter slides pixel by pixel. By increasing the stride size, your filter is sliding over the input with a larger interval and thus has less overlap between the cells. While increasing the stride may decrease the computational time, it also might decrease the accuracy of the model.

Padding

Sometimes filter does not fit perfectly fit the input image. We have two options:

- ***Zero-padding*** - Pad the picture with zeros so that it fits
- ***Valid-padding*** - Drop the part of the image where the filter did not fit and keeps only valid part of the image.

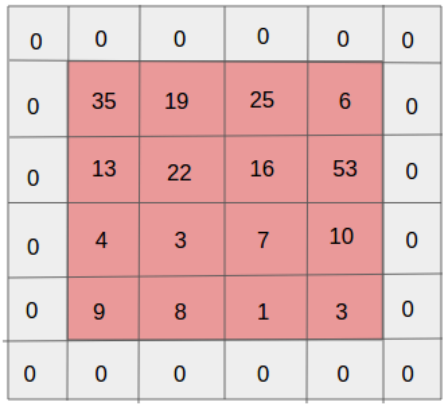

Fig. 12 zero padding for a 3x3 kernel

If a ***kernel*** (filter) of size k×k is used, then the padding size p should be chosen to be ***p=(k−1)/2***

Intuitively, for each cell of the input matrix must be placed at the center of kernel.

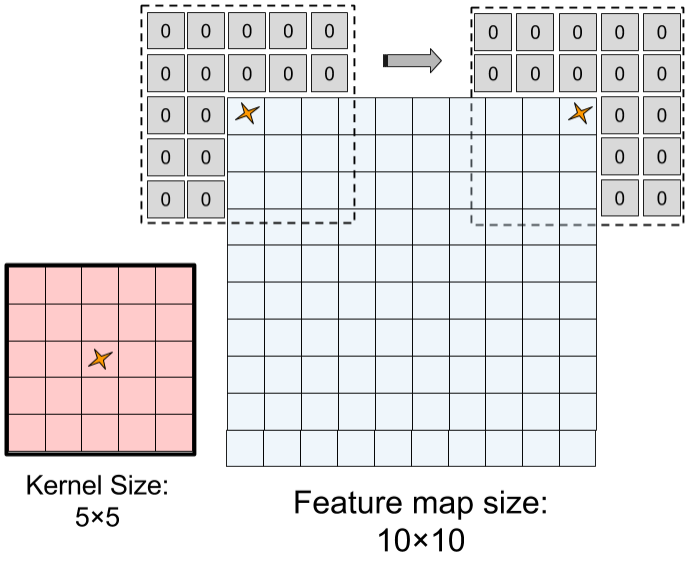

Fig. 13 zero padding for a 5x5 kernel

2. Max Pooling Layers

It is common to periodically insert a Pooling layer in-between successive Conv layers in a ConvNet architecture. Its function is to progressively reduce the spatial size of the representation to reduce the amount of parameters and computation in the network, and hence to also control overfitting. The Pooling Layer operates independently on every depth slice of the input and resizes it spatially, using the MAX operation. The most common form is a pooling layer with filters of size 2x2 applied with a stride of 2 downsamples every depth slice in the input by 2 along both width and height, discarding 75% of the activations. Every MAX operation would in this case be taking a max over 4 numbers (little 2x2 region in some depth slice).

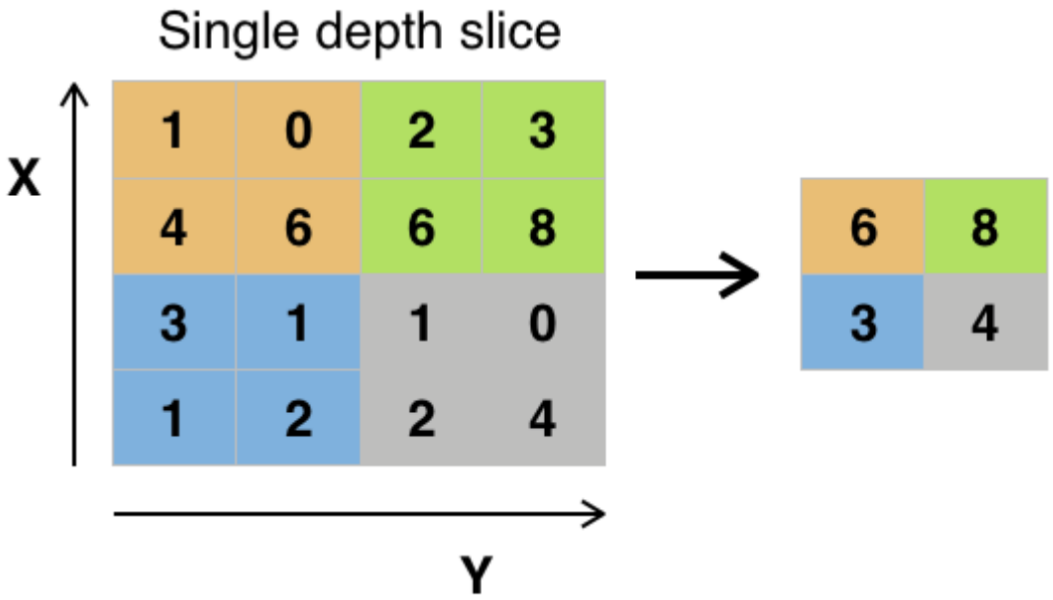

Fig. 14 Max pooling with a 2x2 filter and stride = 2

3. Flatten Layers

Usually come before ***Dense*** layers. They are used to convert the image from it's 2D form (widthxheight) to a 1D vector.

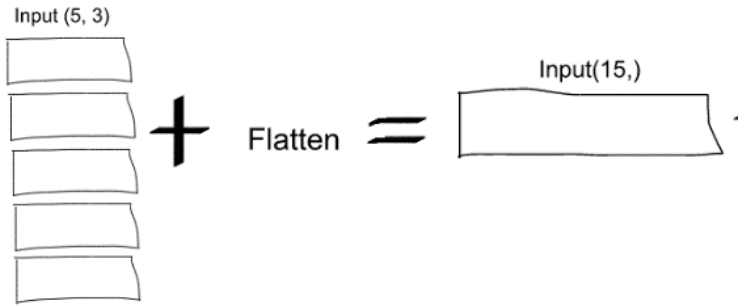

Fig. 15 Flatten layer

4. Dense (Fully Connected) Layers

Usually used as the output layer in each CNN model.
No. of nodes in the output layer = no. of classes

With the fully connected layers, we combined these features together to create a model. Finally, we have an activation function such as softmax or sigmoid to classify the outputs as cat, dog, car, truck etc.,

Neurons in a fully connected layer have full connections to all activations in the previous layer, as seen in regular Neural Networks. Their activations can hence be computed with a matrix multiplication followed by a bias offset.

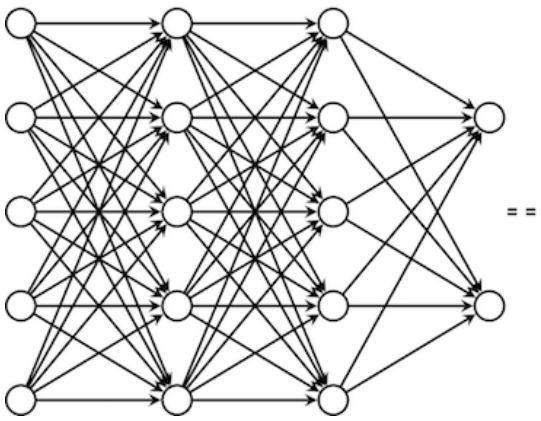

Fig. 16 Dense (Fully-connected) layer

5. Dropout Layers

Dropout refers to ignoring units (i.e. neurons) during the training phase of certain set of neurons which is chosen at random. By “ignoring”, I mean these units are not considered during a particular forward or backward pass.

More technically, At each training stage, individual nodes are either dropped out of the net with probability *1-p* or kept with probability *p*, so that a reduced network is left; incoming and outgoing edges to a dropped-out node are also removed.

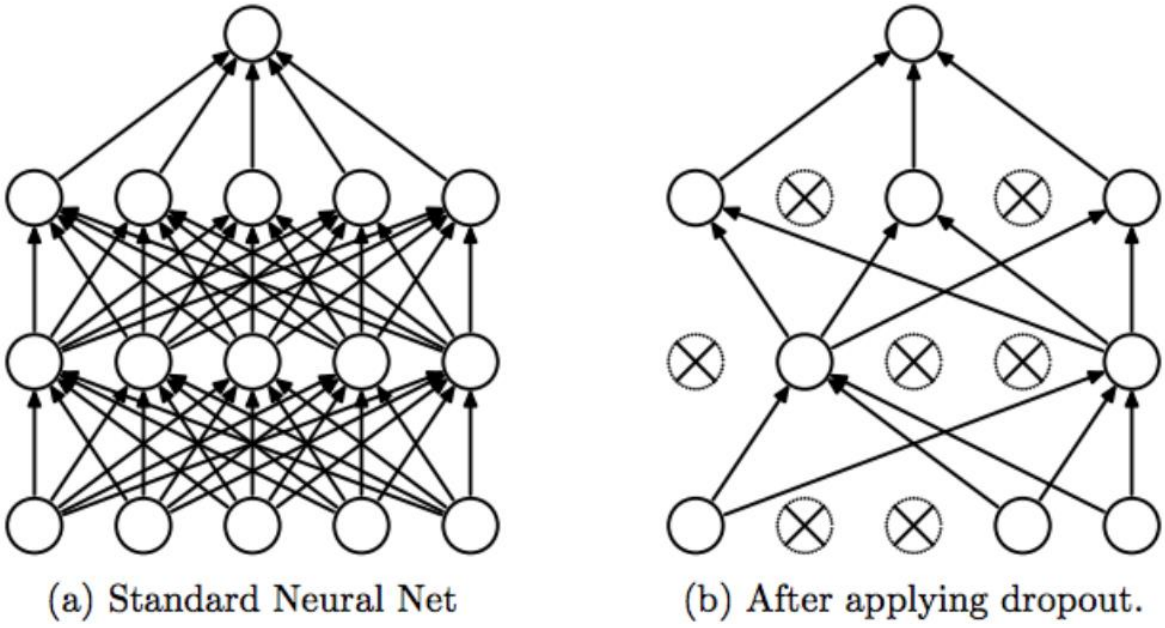

Fig. 17 Dropout layer

Why do we need Dropout?

Used for ***Regularization,*** to overcome ***overfitting***.

CNN – Transfer Functions

1. Step function

Where the output is 1 if the value of x is greater than equal to zero and 0 if the value of x is less than zero. As one can see a step function is non-differentiable at zero. At present, a neural network uses back propagation method along with gradient descent to calculate weights of different layers. Since the step function is non-differentiable at zero hence it is **not able to make progress** with the gradient descent approach and fails in the task of updating the weights.

To overcome, this problem **sigmoid** functions were introduced instead of the step function.

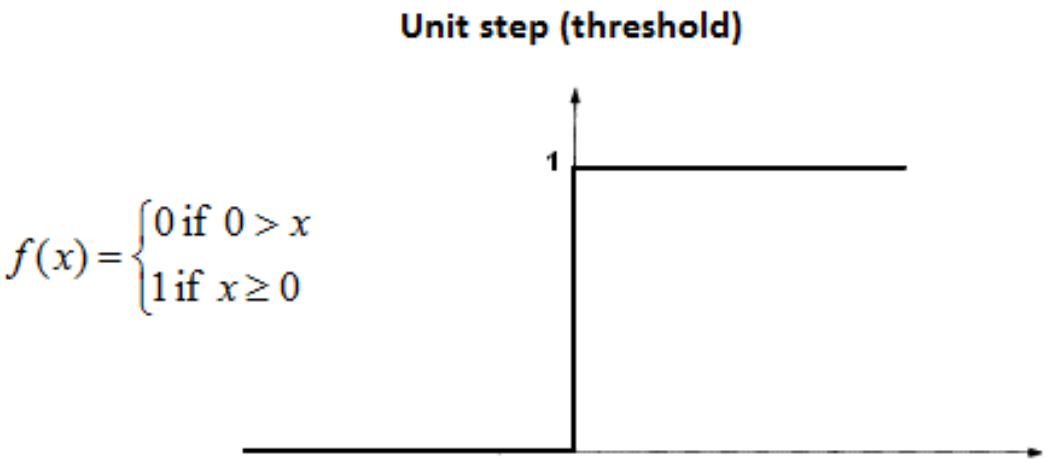

Fig. 18 Unit Step transfer function

2. Sigmoid Function

The value of the function tends to zero when z or independent variable tends to negative infinity and tends to 1 when z tends to infinity. It needs to be kept in mind that this function represents an approximation of the behavior of the dependent variable and is an assumption. Now the question arises as to why we use the sigmoid function as one of the approximation functions. There are certain simple reasons for this:

1. *It captures non-linearity in the data. Albeit in an approximated form, but the concept of non-linearity is essential for accurate modeling.*
2. *The sigmoid function is differentiable throughout and hence can be used with gradient descent and backpropagation approaches for calculating weights of different layers.*
3. *The assumption of a dependent variable to follow a sigmoid function inherently assumes a Gaussian distribution for the independent variable which is a general distribution we see for a lot of randomly occurring events and this is a good generic distribution to start with.*

However, a sigmoid function also suffers from a problem of vanishing gradients. As can be seen from the picture a sigmoid function squashes it's input into a very small output range [0,1] and has very steep gradients. Thus, there remain large regions of input space, where even a large change produces a very small change in the output. This is referred to as the problem of vanishing gradient. This problem increases with an increase in the number of layers and thus stagnates the learning of a neural network at a certain level.

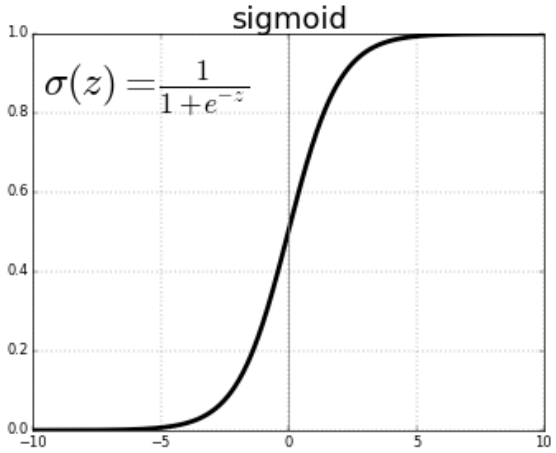

Fig. 19 Sigmoid transfer function

3. ReLU Function

The Rectified Linear Unit is the most commonly used activation function in deep learning models.

The function returns 0 if it receives any negative input, but for any positive value x, it returns that value back. So, it can be written as f(x)=max (0, x).

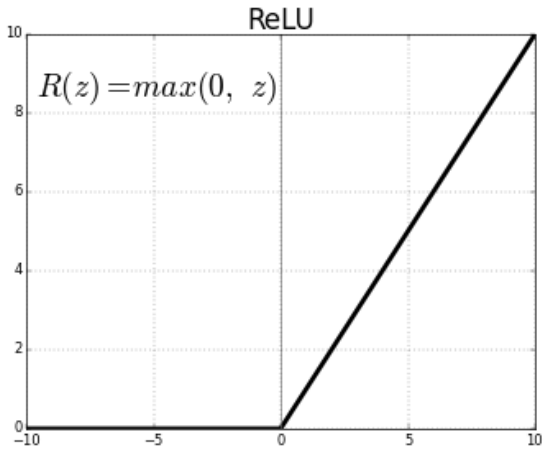

Fig. 20 ReLU transfer function

The **Leaky ReLU** is one of the most well-known. It is the same as ReLU for positive numbers. But instead of being 0 for all negative values, it has a **constant slope** (less than 1.).

That slope is a parameter the user sets when building the model, and it is frequently called α. For example, if the user sets α=0.3, the activation function is f(x) = max (0.3*x, x) *This has the theoretical advantage that, by being influenced,* ***by x at all values, it may make more complete use of the information contained in x.***

There are other alternatives, but both practitioners and researchers have generally found an insufficient benefit to justify using anything other than ReLU. In general practice as well, ReLU has found to be performing better than sigmoid or Tanh functions.

CNN – Loss Functions

Machines learn by means of a loss function. It's a method of evaluating how well specific algorithm models the given data. If predictions deviate too much from actual results, loss function would cough up a very large number. Gradually, with the help of some optimization function, loss function learns to reduce the error in prediction.

Regression losses and Classification losses.

In classification, we are trying to predict output from set of finite categorical values i.e. Given large data set of images of hand written digits, categorizing them into one of 0–9 digits. Regression, on the other hand, deals with predicting a continuous value for example given floor area, number of rooms, size of rooms, predict the price of room.

A. Regression Losses

1. Mean Square Error/ L2 Loss

As the name suggests, *Mean Square Error* is measured as the average of squared difference between predictions and actual observations. However, due to squaring, predictions which are far away from actual values are penalized heavily in comparison to less deviated predictions.

$$MSE = \frac{\sum_{i=1}^{n} (y_i - \hat{y}_i)^2}{n}$$

Eqn. 2 Mean Square Error

```
Where:
n         - Number of training examples.
i         - ith training example in a data set.
y(i)      - Ground truth label for ith training example.
y_hat(i)  - Prediction for ith training example.
```

2. Mean Absolute Error/L1 Loss

Mean absolute error, on the other hand, is measured as the average of sum of absolute differences between predictions and actual observations.

$$MAE = \frac{\sum_{i=1}^{n} \mid y_i - \hat{y}_i \mid}{n}$$

Eqn. 2 Mean Absolute Error

B. Classification Loss

1. Hinge/Multi-class SVM Loss

The score of correct category should be greater than sum of scores of all incorrect categories by some safety margin (usually one). And hence hinge loss is used most notably for ***Support Vector Machines***. **[11]**

$$SVMLoss = \sum_{j \neq y_i} max(0, s_j - s_{y_i} + 1)$$

Eqn. 3 Support Vector Machine loss

2. Cross Entropy Loss

This is the most common setting for classification problems. Cross-entropy loss increases as the predicted probability diverges from the actual label. An important aspect of this is that cross entropy loss penalizes heavily the predictions that are *confident but wrong*.

$$CrossEntropyLoss = -(y_i log(\hat{y}_i) + (1 - y_i) log(1 - \hat{y}_i))$$

i) Categorical Cross Entropy

The ground truth is provided as an n-dimensional vector in which all entries are 0 except the entry corresponding to the class, which is 1 (one-hot-encoding).

One-hot encoding

ii) Sparse Categorical Cross Entropy

The ground truth is provided as single integer unit

iii) Binary Cross Entropy

There are only two classes in the ground truth

CNN – Learning Rate

Learning rate is a hyper-parameter that controls how much we are adjusting the weights of our network with respect the loss gradient. The lower the value, the slower we travel along the downward slope. While this might be a good idea (using a low learning rate) in terms of making sure that we do not miss any local minima, it could also mean that we'll be taking a long time to converge.

Changing our weights too fast by adding or subtracting too much (i.e. taking steps that are too large) can hinder our ability to minimize the loss function.

Taking steps that are too large can mean that the algorithm will never converge to an optimum.

At the same time, we don't want to take steps that are too small, because then we might never end up with the right values for our weights.

Steps that are too small might lead to our optimizer converging on a local minimum for the loss function, but not the absolute minimum.

Learning rate is just a very small number, usually something like 0.001, that we multiply the gradients by to scale them. This ensures that any changes we make to our weights are pretty small.

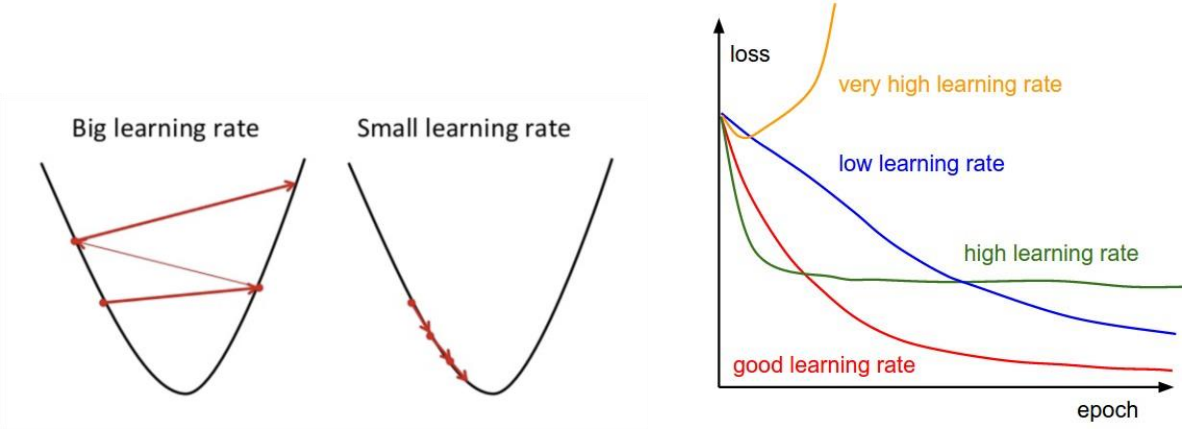

Fig. 21 learning rates

CNN – Optimizers

Optimization algorithms helps us to *minimize (or maximize)* an Objective function (*another name for Error function*) E(x) which is simply a mathematical function dependent on the Model's internal learnable parameters which are used in computing the target *values*(Y) from the set of *predictors*(X) used in the model i.e minimizing the Loss as the network trains.

1. **First-order Optimizers**
   These algorithms minimize or maximize a Loss Function **E(x)** using its ***Gradient*** values with respect to the parameters. Most widely used First order optimization algorithm is **Gradient Descent.** The First order derivative tells us whether the function is decreasing or increasing at a particular point. First order Derivative basically give us a **line** which is ***Tangential*** *to a point on its Error Surface.*

2. **Second-order Optimizers**
   Second-order methods use the second order derivative which is also called Hessian to minimize or maximize the Loss function.

So which Order Optimization Strategy to use?

1. Now **The First Order Optimization** techniques are easy to compute and less time consuming, converging pretty fast on large data sets.

2. **Second Order Techniques** are faster only when the **Second Order Derivative** is known otherwise, these methods are always slower and costly to compute in terms of both time and memory *Although, sometimes* ***Second Order Optimization*** *technique can sometimes outperform* ***First Order Gradient Descent*** *because Second-order techniques will not get stuck around paths of slow convergence around* ***saddle points.***

Types of Optimizers

1. Gradient Descent

Probably one of the most straightforward approaches is to simply go in direction opposite to the gradient.

$$y = x - \alpha \nabla f(x)$$

Eqn. 4 Gradient Descent equation

y – new parameters
x – old parameters
α – learning rate
f(x) – loss function

Disadvantages:

- The algorithm is vulnerable to Saddle points.
- The algorithm is ineffective—it requires the use of the entire training set in each iteration. This means that in every epoch we have to look at all the examples in order to perform next optimization step.

2. Mini-batch Gradient Descent

As mentioned before, Gradient Descent is highly inefficient when it comes to large datasets. So how could we possible solve this problem? One possible solution is kind of obvious really; divide your dataset into smaller batches and train subsequently on these batches, that way, the GD algorithm will operate on a much less number of example per training iteration and we will make use of vectorization. But how do we choose the batch size, you might ask? The answer is relative, as is the case for most of the deep learning problems. You just have to try it out for yourself. Though it is recommended to choose a batch size that is a power of 2 – 32, 64, 128, etc…

3. Stochastic Gradient Descent

Same as Mini-batch GD but uses one stochastic item instead of a batch.

4. Gradient Descent with Momentum

This strategy leverages exponentially weighted averages to avoid troubles with points where gradient of cost functions is close to zero. In simple words, we allow our algorithm to gain momentum, so that even if the local gradient is zero, we still move forward relying on the previously calculated values. For this reason, it is almost always a better choice than a pure gradient descent.

$$\mathbf{V}_{\mathbf{dW}}^{[l]} = \beta \mathbf{V}_{\mathbf{dW}}^{[l]} + (1 - \beta)\mathbf{dW}^{[l]}$$

$$\mathbf{V}_{\mathbf{db}}^{[l]} = \beta \mathbf{V}_{\mathbf{db}}^{[l]} + (1 - \beta)\mathbf{db}^{[l]}$$

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \mathbf{V}_{\mathbf{dW}}^{[l]}$$

$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \mathbf{V}_{\mathbf{db}}^{[l]}$$

Eqn. 5 Gradient Descent with Momentum

Where,
VdW and ***Vdb*** are the ***EWA*** of ***dW*** and ***dB***, and ***β*** is the ***range of values to be averaged*** in the EWA portion.

The use of exponentially weighted averages allows us to focus on the leading trend instead of the noise. Component indicating minimum is amplified and component responsible for the oscillations is slowly eliminated. What's more, if we obtain gradients pointing a similar direction in subsequent updates, the learning rate will increase. This leads to faster convergence and reduced oscillations. This method, however, has a disadvantage—as you approach the minimum, the momentum value increases and may become so large that the algorithm will not be able to stop at the right place.

5. RMSProp

Root Mean Squared Propagation—which is one of the most frequently used optimizers. This is another algorithm that uses exponentially weighted averages. Moreover, it is adaptive—it allows for individual adjustment of the learning rate for each parameter of the model. Subsequent parameter values are based on previous gradient values calculated for particular parameter.

$$\mathbf{S}_{\mathbf{dW}}^{[l]} = \beta \mathbf{S}_{\mathbf{dW}}^{[l]} + (1-\beta)(\mathbf{dW}^{[l]})^2$$

$$\mathbf{S}_{\mathbf{db}}^{[l]} = \beta \mathbf{S}_{\mathbf{db}}^{[l]} + (1-\beta)(\mathbf{db}^{[l]})^2$$

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\mathbf{dW}^{[l]}}{\sqrt{\mathbf{S}_{\mathbf{dW}}^{[l]}} + \varepsilon}$$

$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\mathbf{db}^{[l]}}{\sqrt{\mathbf{S}_{\mathbf{db}}^{[l]}} + \varepsilon}$$

Eqn. 6 RMSProp

In RMSProp, the learning rate is reduced faster for parameters where the gradient is large and slower for parameters where the gradient is small.

6. ADAM

Adaptive Moment Estimation. It is an algorithm which, like RMSProp, works well in a wide range of applications. It takes advantage of the biggest pros of RMSProp, and combine them with ideas known from momentum optimization. The result is a strategy that allows for quick and effective optimization.

$$\mathbf{V}_{\mathbf{dW}}^{[l]} = \beta_1 \mathbf{V}_{\mathbf{dW}}^{[l]} + (1 - \beta_1)\mathbf{dW}^{[l]}$$

$$\mathbf{V}_{\mathbf{db}}^{[l]} = \beta_1 \mathbf{V}_{\mathbf{db}}^{[l]} + (1 - \beta_1)\mathbf{db}^{[l]}$$

$$\mathbf{S}_{\mathbf{dW}}^{[l]} = \beta_2 \mathbf{S}_{\mathbf{dW}}^{[l]} + (1 - \beta_2)(\mathbf{dW}^{[l]})^2$$

$$\mathbf{S}_{\mathbf{db}}^{[l]} = \beta_2 \mathbf{S}_{\mathbf{db}}^{[l]} + (1 - \beta_2)(\mathbf{db}^{[l]})^2$$

$$\hat{\mathbf{V}}_{\mathbf{dW}}^{[l]} = \frac{\mathbf{V}_{\mathbf{dW}}^{[l]}}{1 - \beta_1^t}$$

$$\hat{\mathbf{V}}_{\mathbf{db}}^{[l]} = \frac{\mathbf{V}_{\mathbf{db}}^{[l]}}{1 - \beta_1^t}$$

$$\hat{\mathbf{S}}_{\mathbf{dW}}^{[l]} = \frac{\mathbf{S}_{\mathbf{dW}}^{[l]}}{1 - \beta_2^t}$$

$$\hat{\mathbf{S}}_{\mathbf{db}}^{[l]} = \frac{\mathbf{S}_{\mathbf{db}}^{[l]}}{1 - \beta_2^t}$$

$$\mathbf{W}^{[l]} = \mathbf{W}^{[l]} - \alpha \frac{\hat{\mathbf{V}}_{\mathbf{dW}}^{[l]}}{\sqrt{\mathbf{S}_{\mathbf{dW}}^{[l]} + \varepsilon}}$$

$$\mathbf{b}^{[l]} = \mathbf{b}^{[l]} - \alpha \frac{\hat{\mathbf{V}}_{\mathbf{db}}^{[l]}}{\sqrt{\mathbf{S}_{\mathbf{db}}^{[l]} + \varepsilon}}$$

Eqn. 7 ADAM optimizer

Over-fitting, Under-fitting and Adequate-fitting

Sometimes the model performs poorly during the testing phase, that maybe because the model is too simple to describe the target, or too complex to express the target. So, why does this happen? Two things–***over-fitting*** and ***under-fitting***.

By looking at Fig. 22, the graph on the left side we can predict that the line does not cover all the points shown in the graph. Such models tend to cause ***under-fitting*** of data. It also called ***High Bias***.

Whereas the graph on right side, shows the predicted line covers all the points in graph. In such condition you can also think that it's a good graph which cover all the points. But that's not actually true, the predicted line into the graph covers all points which are noise and outlier. Such models are also responsible to predict poor result due to its complexity. This causes the model to ***over-fit*** on the training data. It is also called ***High Variance***.

Now, looking at the middle graph it shows a pretty good predicted line. It covers majority of the point in graph and also maintains the balance between bias and variance. This is called ***Adequate-fitting***, ***Robust-fitting*** or ***Appropriate-fitting***.

"Bias is reduced and variance is increased in relation to model complexity. As more and more parameters are added to a model, the complexity of the model rises and variance becomes our primary concern while bias steadily falls."

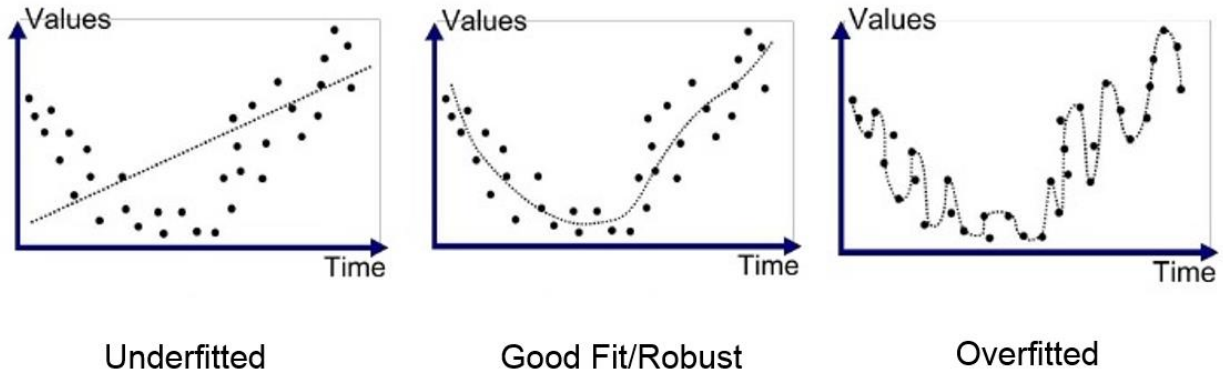

Figure 22. Under-fitting, Robust-fitting and Over-fitting

To sum up in simpler terms;

1. **Adequate/Robust-fitting**

   Ideally, the case when the model makes the predictions with 0 error, is said to have a *good fit* on the data. This situation is achievable at a spot between overfitting and under-fitting. In order to understand it we will have to look at the performance of our model with the passage of time, while it is learning from training dataset

2. **Over-fitting**

   Definition

   The model over-trains on the training data, leading to the model learning noise in given data as features.

   E.g. say we're training a model to classify traffic signs. We give our model a bunch of training data including the image in figure 23. Over-fitting here would mean that the model would 'study' this image so much that it thinks the leaves of the tree is part of the traffic sign and if given an image of a traffic signs with no leaves it would not recognize it.

   A non-technical example would be if you're studying for an exam and you study a mathematical problem with it's given numbers, so if in the exam the professor changed the numbers in a problem, the student would not be able to solve it.

Figure 23. Noise in training data that the model may learn as features thus producing terrible results

   How to overcome

   a) **Regularization**
   b) **Cross-validation**
   c) **Early Stopping**
   d) **Image Augmentation**

   Each will be covered soon

3. Under-fitting

Definition

A statistical model or a machine learning algorithm is said to have under-fitting when it cannot capture the underlying trend of the data. Under-fitting destroys the accuracy of our machine learning model. Its occurrence simply means that our model or the algorithm does not fit the data well enough.

Causes

The model is simply too small or flexible to be able to learn new features.

Another cause for under-fitting may be insufficient training data – you can't possibly give a Deep CNN model say 10 images as the training dataset and expect it to do well.

How to overcome

a) Collect as much training data as possible, at the very least 200 images per class
b) Do not over-use Regularization techniques

Generalization

A model is said to be a good machine learning model, if it ***generalizes*** any new input data from the problem domain in a proper way. This helps us to make predictions in the future on ***data that the model has never seen***.

As an example, say I were to show you an image of dog and ask you to "classify" that image for me; assuming you correctly identified it as a dog, would you still be able to identify it as a dog if I just moved the dog three pixels to the left? What about if I turned it upside? Would you still be able to identify the dog if I replaced it with a dog from a different breed?

To ensure that the model would generalize, the machine learning engineer must ensure first that the model doesn't over-fit or under-fit.

Cross-validation

This process of deciding whether the numerical results quantifying hypothesized relationships between variables, are acceptable as descriptions of the data, is known as ***validation.***

In **K Fold cross validation** (fig. 24), the data is divided into k subsets. Now the holdout method is repeated k times, such that ***each time, one of the k subsets is used as the test set/ validation set and the other k-1 subsets are put together to form a training set***. The *error estimation is averaged over all k trials to get total effectiveness of our model*. As can be seen, every data point gets to be in a validation set exactly once, and gets to be in a training set *k-1*times. ***This significantly reduces bias as we are using most of the data for fitting, and also significantly reduces variance as most of the data is also being used in validation set.*** Interchanging the training and test sets also adds to the effectiveness of this method.

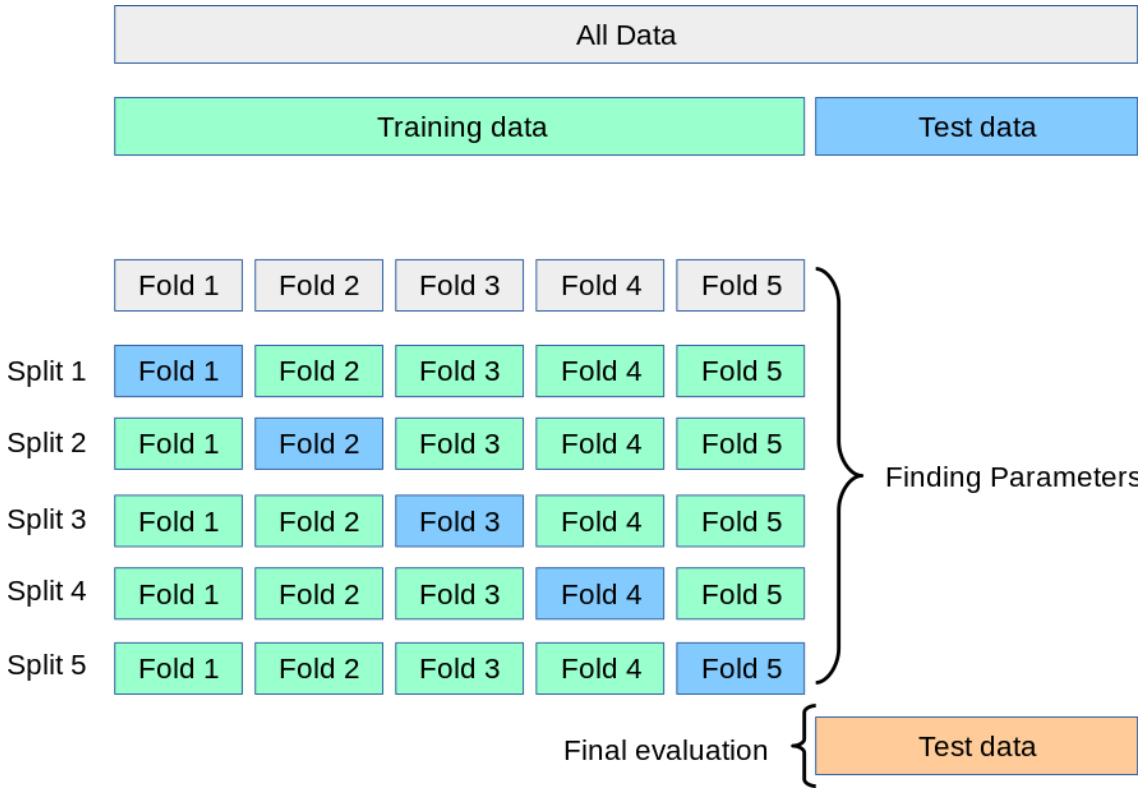

Figure 24. K-Fold Cross-Validation

Early-stopping

A validation-based technique to counter over-fitting in which the training automatically stops as soon as the loss function converges to avoid unnecessary computations and noise-learning.

Regularization

This is a form of regression, that constrains/ regularizes or shrinks the coefficient estimates towards zero. In other words, *this technique discourages learning a more complex or flexible model, so as to avoid the risk of overfitting. I.e.* If there is noise in the training data, then the estimated coefficients won't generalize well to the future data. This is where regularization comes in and shrinks or regularizes these learned estimates towards zero.

***Regularization, significantly reduces the variance of the model, without substantial increase in its bias*.**

Batch-Normalization

We normalize the input layer by adjusting and scaling the activations. For example, when we have features from 0 to 1 and some from 1 to 1000, we should normalize them to speed up learning. If the input layer is benefiting from it, why not do the same thing also for the values in the hidden layers, that are changing all the time, and get 10 times or more improvement in the training speed.

Also, batch normalization allows each layer of a network to learn by itself a little bit more independently of other layers.

- We can use higher learning rates because batch normalization makes sure that there's no activation that's gone really high or really low. And by that, things that previously couldn't get to train, it will start to train.
- It reduces overfitting because it has a slight regularization effects. Similar to dropout, it adds some noise to each hidden layer's activations. Therefore, if we use batch normalization, we will use less dropout. However, its ill-advised to solely depend on one and ignore the other.

Image Augmentation

A convolutional neural network that can robustly classify objects even if its placed in different orientations is said to have the property called invariance. More specifically, a CNN can be invariant to translation, viewpoint, size or illumination (Or a combination of the above).

This essentially is the premise of data augmentation. In the real world scenario, we may have a dataset of images taken in a limited set of conditions. But, our target application may exist in a variety of conditions, such as different orientation, location, scale, brightness etc. We account for these situations by training our neural network with additional synthetically modified data.

Remember— ***"Your neural network is only as good as the data you feed it."***

Popular augmentation techniques:

1. **Flip**
2. **Rotation**
3. **Scale**
4. **Crop**
5. **Skew**
6. **Translation**
7. **Gaussian Noise**
8. **Conditional GANs (Advanced)**

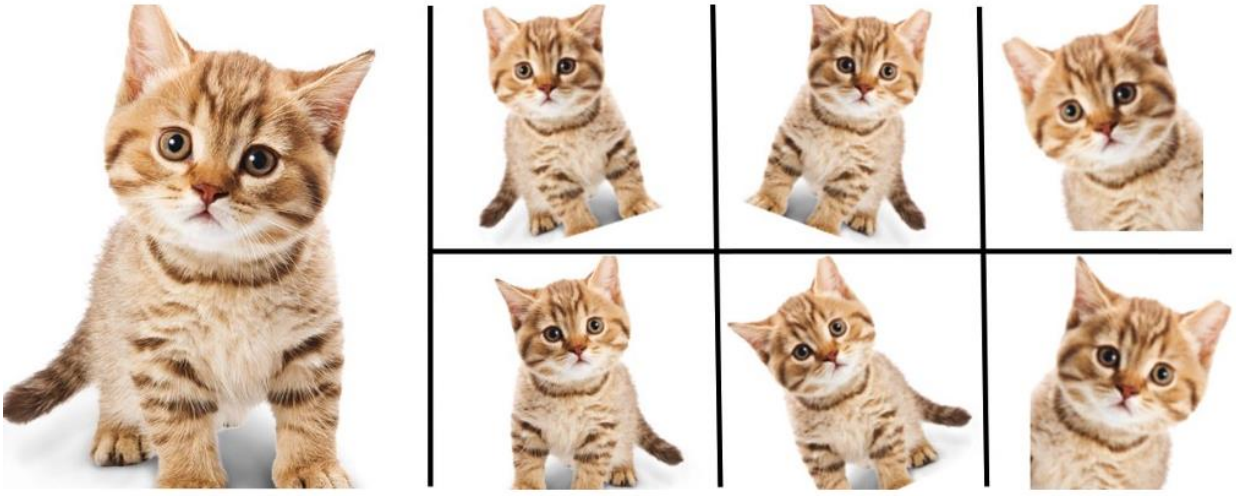

Figure 25. Image augmentation is used to tremendously enlarge small datasets

Feature Maps

Think of it as representations of the input. The number of filters (kernel) you will use on the input will result in same amount of feature maps.

The Convolution layer uses a filter matrix over the array of image pixels and performs convolution operation to obtain a convolved feature map.

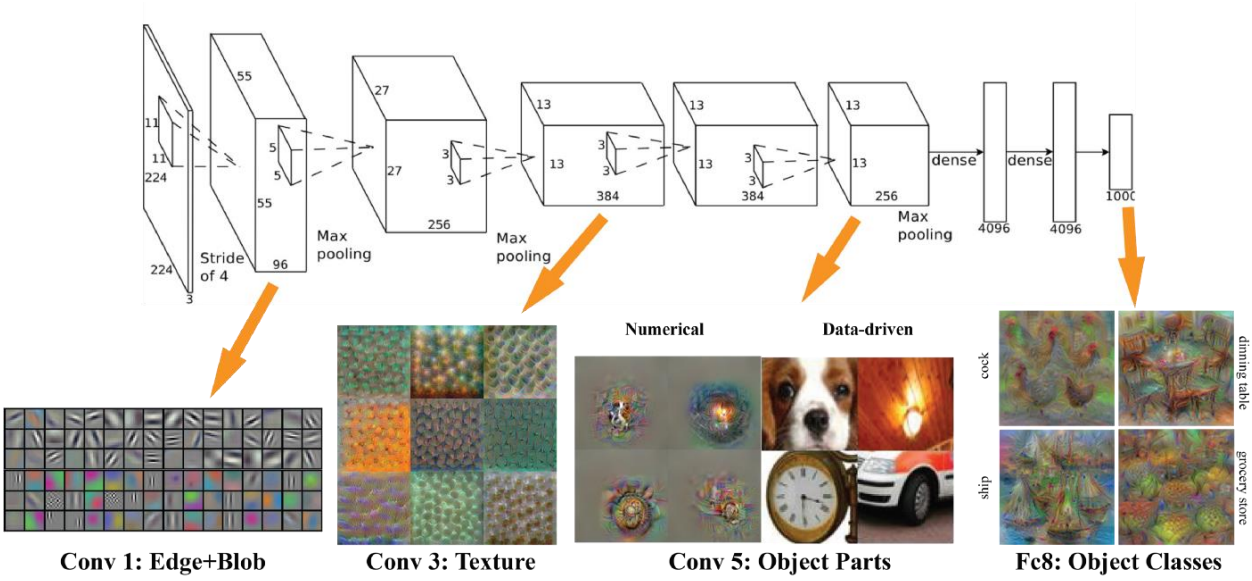

Figure 26. Feature maps

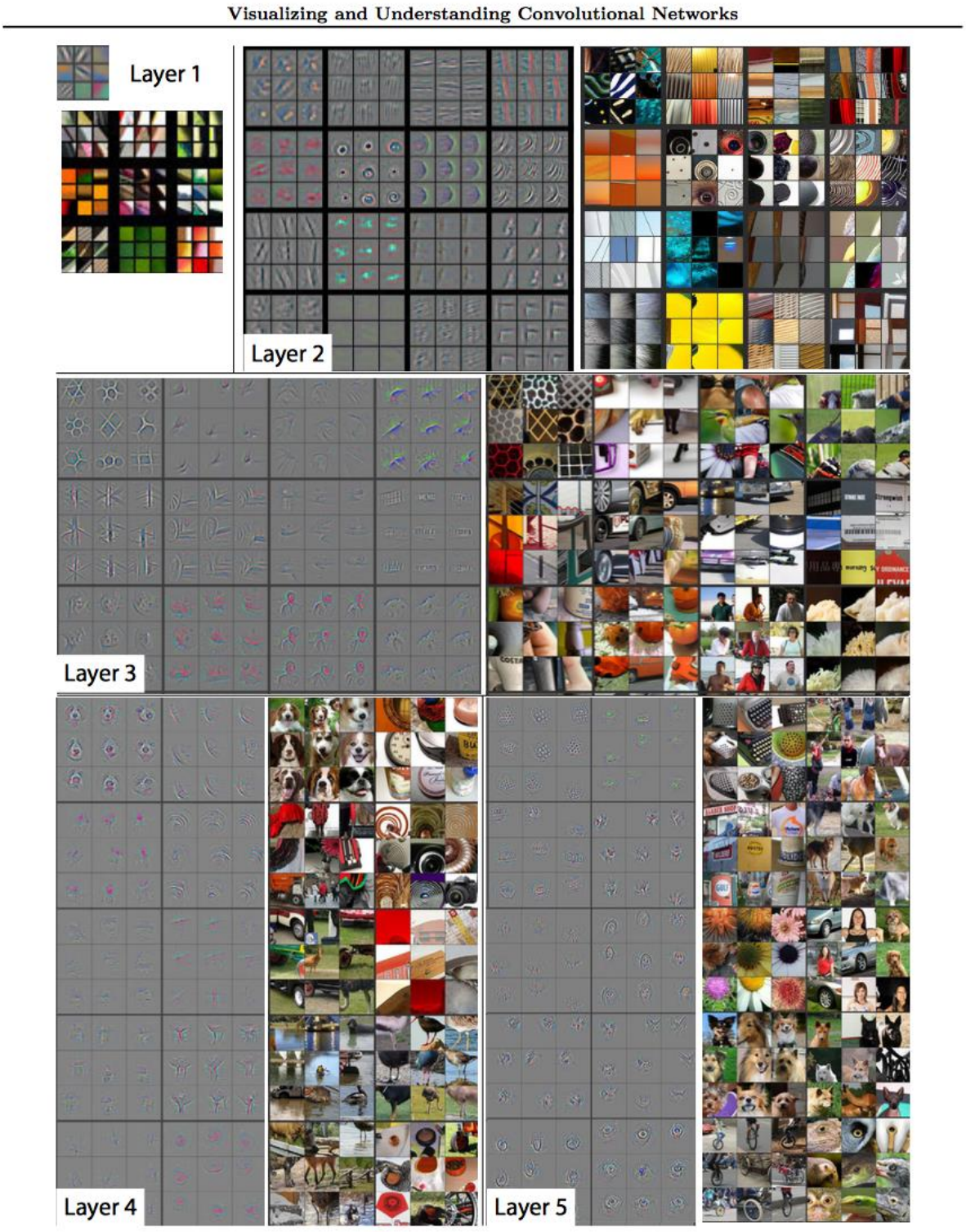

Figure 27. Feature maps for different sets of layers [12]

Transfer Learning

We – humans – transfer and leverage our knowledge from what we have learnt in the past!

Transfer learning is the idea of overcoming the isolated learning paradigm and utilizing knowledge acquired for one task to solve related ones.

Basically;

Transfer Learning is a technique in which a pre-trained model is used and re-trained on a new dataset instead of building a new model from scratch.

This can help eliminate the pain and time consumption of building an advanced deep learning model from the ground up and instead *fine-tune* pre-trained parameters.

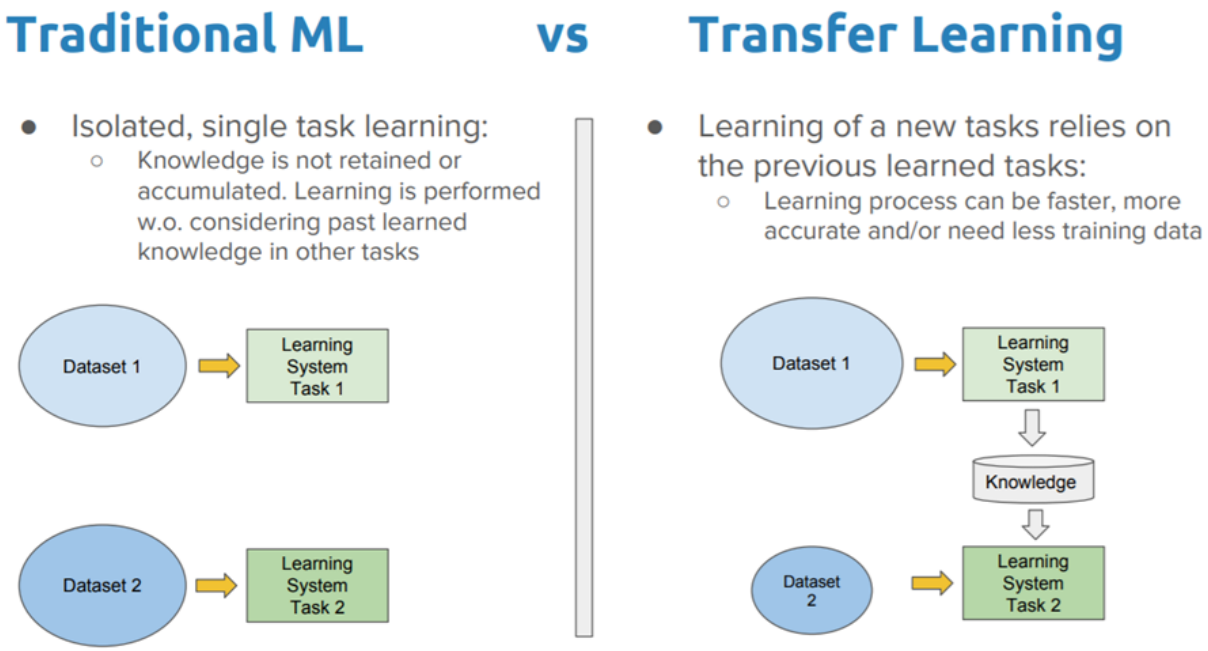

Figure 28. Traditional ML vs. Transfer Learning

Now enough with the introduction and let's start exploring our Traffic Signs Recognition System.

Our project is split into three main parts:

I. Classification

II. Real-time Detection and Recognition (2nd Term)

III. Real-time Implementation (2nd Term)

Chapter 3: Traffic Signs Recognition – Classification

Introduction

A classification problem is concerned with classifying an image to a certain class. Say, if a model is given a picture of a dog, it would classify it to the 'Dog' class/category. (Given, of course, that it has been previously trained to recognize dog pictures).

Traffic Signs Classification Journey

We aimed to create a deep learning model to classify images of traffic signs regardless of the aforementioned complex conditions they may be found in.

A. Dataset

We used the German Traffic Signs Benchmark dataset, which is the de facto for traffic signs classification for myriad of reasons:

- It is large enough to avoid under-fitting
  - It has 39,209 training images
  - 12,630 testing images
  - 43 classes

  Averaging around image per class, which is more than enough to train a deep learning model.
- It contains various complex situations in the training data so that the model can adapt to classify images in such harsh conditions.

Figure 29. GTSRB samples

B. Models

In this section, we will give a brief overview about the deep learning models we used – yes, "models."

1. **Custom model**
2. **VGG-16**
3. **MobileNet v1**
4. **MobileNet v2**

1. Custom Model

Layer
Convolutional, stride 2, kernel 7x7x4
Convolutional, stride 2, kernel 5x5x8
Convolutional, stride 2, kernel 3x3x16
Convolutional, stride 2, kernel 3x3x32
Convolutional, stride 1, kernel 2x2x16
Convolutional, stride 1, kernel 2x2x8
Convolutional, stride 1, kernel 2x2x4
Fully connected-64
Fully connected-16
Softmax

Figure 30. Custom model architecture

At first we tried a very simple CNN, but unfortunately this model did not perform well at all. Producing and ***accuracy of 60%.***

2. VGG-16

- VGG-16 is a model developed by **Virtual Geometry Group** in **2014**
- It was trained on **4 GPUs (Titan X) for 2–3 weeks**.
- It achieved **92.5%** accuracy in ImageNet Large Scale Visual Recognition Competition (**ILSVRC**) competition securing the second place after ***Res-Net.***
- It has 16 layers and **138M** parameters
- VGG-16 is a **very heavy** model of size **528MB**

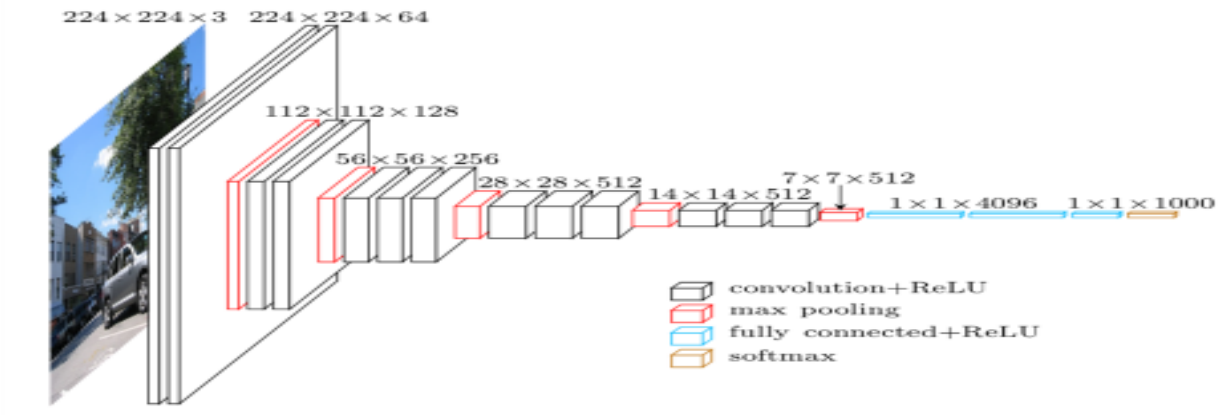

Figure 31. VGG-16 architecture

We unfortunately couldn't train VGG-16 on our dataset (GTSRB) because it was too heavy for my laptop at the time which had an Nvidia GTX 1050 GPU. This is why we decided to go for a lighter model like the **MobileNet**.

3. MobileNet

- Mobile-Net is developed by Google
- Suitable for mobile and embedded based vision applications where there is lack of compute power.
- According to MachineThink*, Mobile-Net v2 is 10x faster than VGG-16 (30 FPS) and consumes almost half the power VGG-16 consumes on the 2018 10.5” iPad Pro
- Mobile-Net v2 consists of 17x fig. 32 followed by the classification layer.
- It has 3.24M parameters

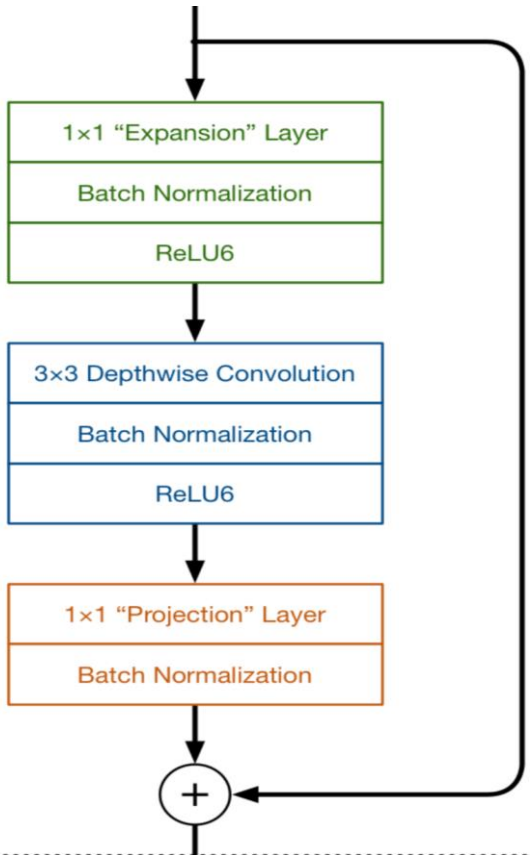

Figure 32. Part of the MobileNetv2 architecture

MobileNetv1

Model	MobileNet v1
Number of epochs	10
Initial learning rate	0.0002
Data augmentation	• Random grey scale • Random 25% crop • Random vertical flip • Random 20-60% brightness shift • Random contrast shift
Number of classes	43
Number of training samples	39, 209
Number of validation samples	12, 630
Total loss achieved	0.54
Training accuracy	**82%**
Testing accuracy	**78%**

Table 3. Parameters used for re-training MobileNet v1

However, we were not satisfied with the achieved accuracy for our system.

We have tried increasing the number of training iterations, but the model ended up over-fitting and we got even worse accuracy.

MobileNetv2

Model	MobileNet v2
Number of epochs	10
Initial learning rate	0.0002
Data augmentation	• Random grey scale • Random 25% crop • Random vertical flip • Random 20-60% brightness shift • Random contrast shift
Number of classes	43
Number of training samples	39,200
Number of validation samples	12,800
Total loss achieved	0.04
Training accuracy	**99.9%**
Testing accuracy	**99.8%**

Table 4. Parameters used for re-training MobileNet v2

Accuracy of ***99.8%*** is very satisfactory.

Notice that we could not use horizontal flip in image augmentation because that would completely change properties of some classes like '***Turn Right***' and '***Turn Left.***'

Results

Fig. 33 shows the confusion matrix of the MobileNetv2 model

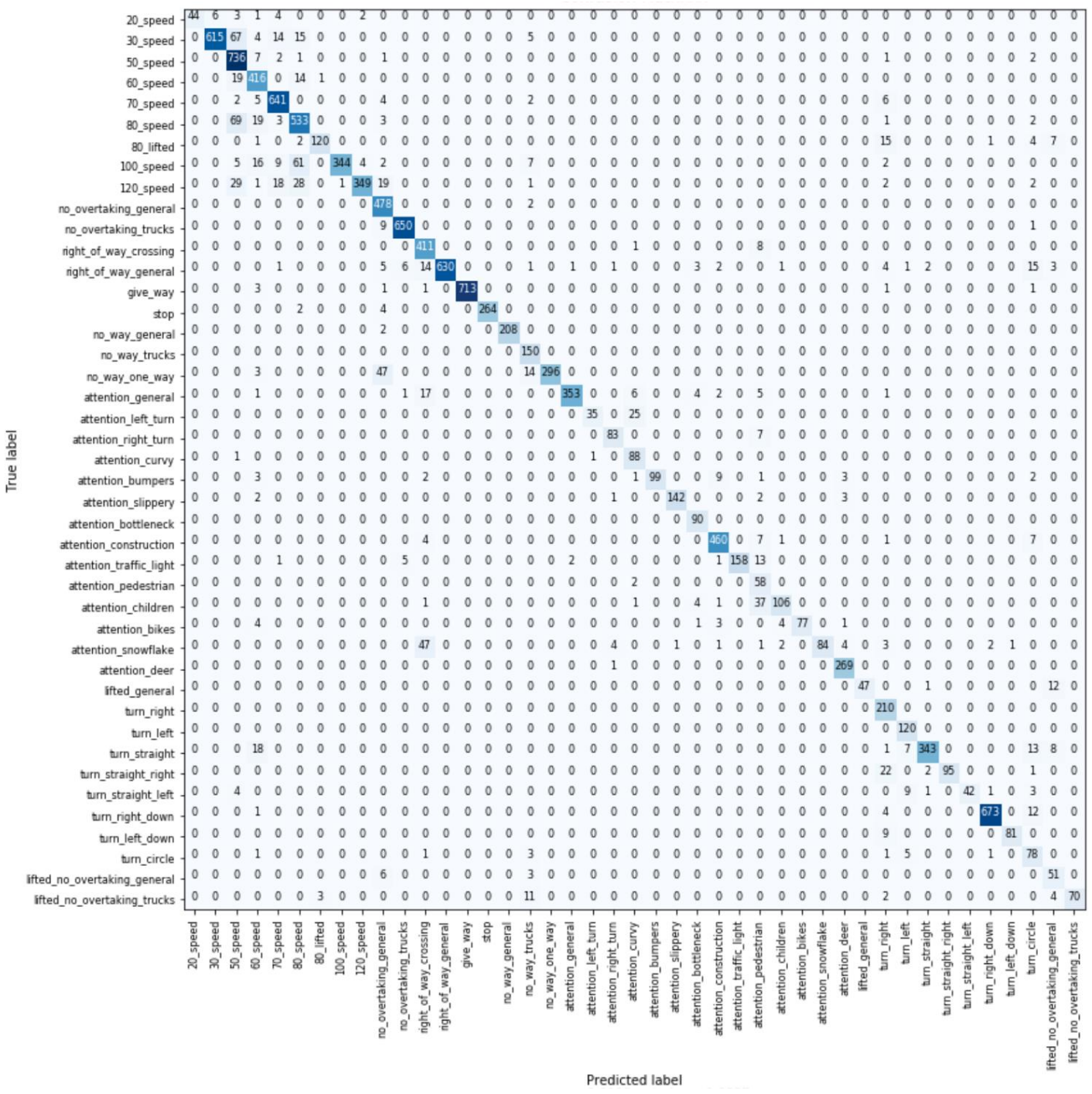

Figure 33. Confusion matrix

GTSRB competition

Our variation of MobileNet v2 scored a testing accuracy of 99.8% which is an all-time record according to the GTSRB Competition. [13]

The number 1 algorithm used CNN and scored an accuracy of ***99.7%***

TEAM	METHOD	TOTAL	SUBSET All signs
[156] DeepKnowledge Seville	CNN with 3 Spatial Transformers	99.71%	99.71%
[3] IDSIA ⭐	Committee of CNNs	99.46%	99.46%
[155] COSFIRE	Color-blob-based COSFIRE filters for object recogn	98.97%	98.97%
[1] INI-RTCV ⭐	Human Performance	98.84%	98.84%
[4] sermanet ⭐	Multi-Scale CNNs	98.31%	98.31%
[2] CAOR ⭐	Random Forests	96.14%	96.14%
[6] INI-RTCV	LDA on HOG 2	95.68%	95.68%
[5] INI-RTCV	LDA on HOG 1	93.18%	93.18%
[7] INI-RTCV	LDA on HOG 3	92.34%	92.34%

Table 5. GTSRB Competition [13]

Steps of training and testing

Now we will go through the exact steps, from A-Z on how to replicate the results we got. From installing python to testing your first deep learning model.

1. Install python
2. Install dependencies
    a. Scipy
    b. Matplotlib
    c. OpenCV or Pillow
    d. Tensorflow
    e. Keras
3. Run test_mobilenet_tsrs.py attached with this book
4. See the results for yourself!

Code

```
# ### Importing libraries
from keras.models import model_from_json
import numpy as np
import os
import cv2
import matplotlib.pyplot as plt

from keras.preprocessing.image import load_img
from keras.preprocessing.image import img_to_array
from keras.applications.mobilenet import preprocess_input


# ### Class names
class_names = [
    '20_speed',
    '30_speed',
    '50_speed',
    '60_speed',
    '70_speed',
    '80_speed',
    '80_lifted',
    '100_speed',
    '120_speed',
    'no_overtaking_general',
    'no_overtaking_trucks',
    'right_of_way_crossing',
    'right_of_way_general',
    'give_way',
    'stop',
    'no_way_general',
    'no_way_trucks',
    'no_way_one_way',
    'attention_general',
    'attention_left_turn',
    'attention_right_turn',
    'attention_curvy',
    'attention_bumpers',
    'attention_slippery',
    'attention_bottleneck',
    'attention_construction',
    'attention_traffic_light',
    'attention_pedestrian',
    'attention_children',
```

```
    'attention_bikes',
    'attention_snowflake',
    'attention_deer',
    'lifted_general',
    'turn_right',
    'turn_left',
    'turn_straight',
    'turn_straight_right',
    'turn_straight_left',
    'turn_right_down',
    'turn_left_down',
    'turn_circle',
    'lifted_no_overtaking_general',
    'lifted_no_overtaking_trucks'
]


# ### Loading model

# Model reconstruction from JSON file
with open('MobileNet_model_architecture.json', 'r') as f:
    loaded_model = model_from_json(f.read())

# Load weights into the new model
loaded_model.load_weights('MobileNet_model_weights.h5')

loaded_model.summary()


# ### Loading images from test_images folder

# dimensions of images
img_width, img_height = 224, 224
folder_path = r"test_images"


# takes a shitload of time. [insert totally worth it meme here]
images = []

for root, directories, file_paths in os.walk(folder_path):
    for filename in file_paths:
        img_path = os.path.join(root, filename)

        image = load_img(img_path, target_size=(img_width, img_height))
```

```
        image = img_to_array(image)

        image = image.reshape((1, image.shape[0], image.shape[1],
image.shape[2]))

        image = preprocess_input(image)

        images.append(image)


# ### Predicting images
predictions = []

for image in images:
    yhat = loaded_model.predict(image)

    predictions_int = np.argmax(yhat)

    predictions.append(class_names[predictions_int])


print(predictions)


# ### Plotting images and predicted labels
images_cv2 = []
for root, directories, file_paths in os.walk(folder_path):
    for filename in file_paths:
        img_path = os.path.join(root, filename)
        img = cv2.imread(img_path, 1)
        img = cv2.cvtColor(img, cv2.COLOR_BGR2RGB)
        images_cv2.append(img)


ROWS = 5
COLS = 5
plt.figure(figsize=(10, 10))
for i in range(25):
    plt.subplot(ROWS, COLS, i+1)
    plt.imshow(images_cv2[i])
    plt.subplots_adjust(hspace=0.5)
    plt.title(predictions[i], fontsize=15, color='green')
    plt.xticks([])
    plt.yticks([])

plt.show()
```

Chapter 4: Traffic Signs Recognition – Detection and Recognition

Now to the main part of the project – Traffic Signs Detection and Recognition.

In this part of the system, our goal is ***to detect and classify traffic sign in real-time***.

We have, again, tried various models and variations to see which would produce the best results. Let's have a look.

At first all the training and testing for the TSR system was done on a ***Host PC***.

After that we implemented the best models on the ***iMX6Q board***, ***TASS PreScan***, and ***Raspberry Pi 3 Model B+***.

Implementation on Host PC

As I mentioned before, the first phase of training and testing the TSR was done on a desktop PC with specifications shown in table 6.

	Host PC
CPU	**I7 6700k Quad-core 3.2GHz**
RAM	**16GB DDR3**
GPU	**Nvidia GTX 1070 6GB**

Table 6. Host PC specs

We used these models respectively and got very different results for reasons that were only discovered by data analysis for the GTSDB.

Models used respectively:

1. **SSD MobileNetv1**
2. **FRCNN ResNet50**
3. **FRCNN Inceptionv2**
4. **Yolov2**

And we're going to discuss each one in details in the very next section.

Model #1: *SSD MobileNetv1*

The very first model we tried is the SSD MobileNetv1 simply because after brief research we found out that it was one of the lightest models available for object detection using Tensorflow.

Overview

MobileNetv1

MobileNets are small, low-latency, low-power models parameterized to meet the resource constraints of a variety of use cases. They can be built upon for classification, detection, embedding and segmentation similar to how other popular large scale models, such as Inception, are used.

Note that MobileNetv1 architecture is mentioned before in the Classification section.

Single Shot Multi-Box Detector (SSD) [14]

- ***Single Shot*:** this means that the tasks of object localization and classification are done in a *single forward pass* of the network
- ***Multi-Box*:** Able to detect multiple objects in a single frame/image
- ***Detector*:** The network is an object detector that also classifies detected objects

  Note that SSD's input image size is 300x300 because we're going to visit that later.

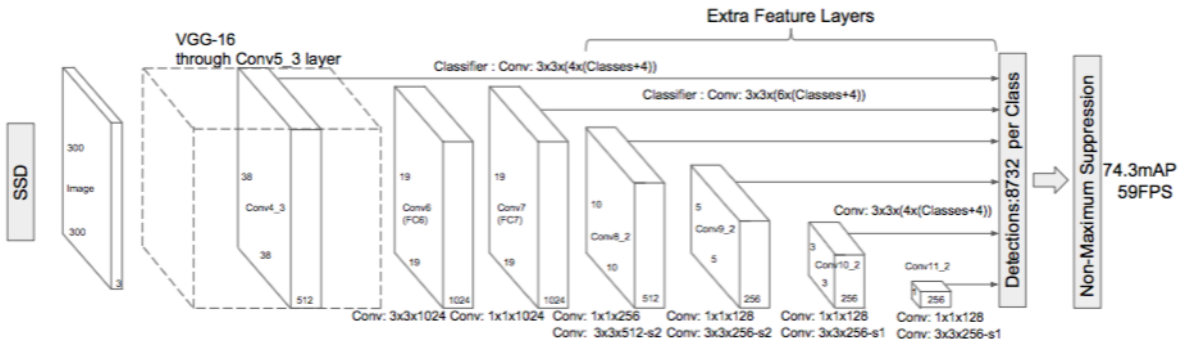

Figure 34. SSD architecture

Training parameters

Table 7. shows the training configuration we used for the SSD MboileNetv1 model. Of course these are ***NOT*** the initial configurations; but the ones that gave us the best possible results.

Model	**SSD MobileNet v1**
Number of epochs	20k
Initial learning rate	0.0002
Learning rate decay	-
Image resizer*	SSD image resizer (300x300)
IoU*	0.6
L2 regularization	0.0004
Batch normalization	True Decay: 0.9997 Epsilon: 0.001
Rescore*	NMS
Data augmentation	Random grey scale Random 25% crop Random Vertical flip*
Number of classes	43
Number of training samples	800
Total loss	1.5

Table 7. TSR configuration for SSD MobileNetv1 model

Let's quickly go over the parameters we have not discussed before:

Image resizer: SSD image resizer automatically resizes the input image to 300pixels * 300pixels. This is done to increase the training and detection speeds.

Intersection over Union (IoU) is how much the detected box (during training phase) intersects or overlaps over the ground truth detection box. If the model's detected box overlaps with 60% of the ground truth box, this box is said to be an object.

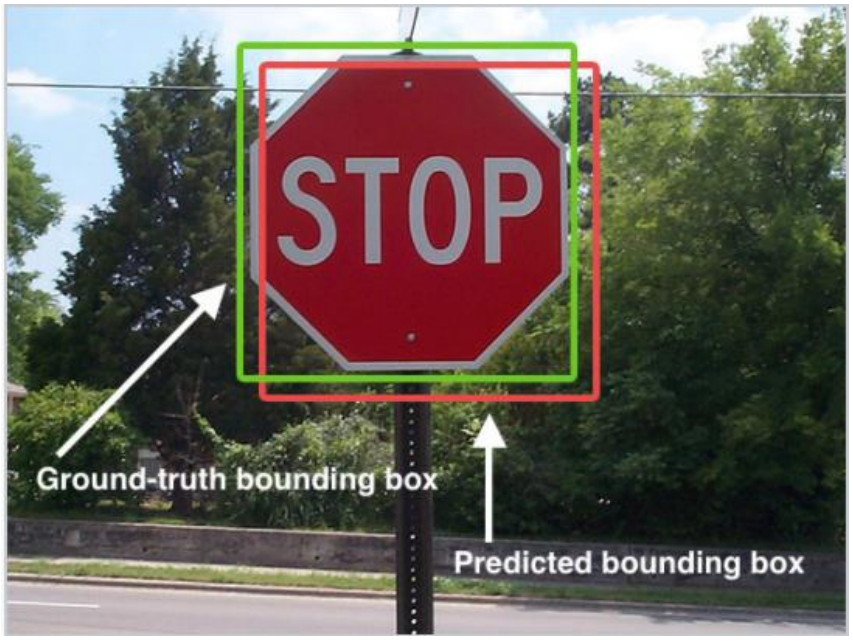

Figure 35. Intersection over Union

Rescore: the model usually does not predict just one object; it predicts 10s or maybe 100s of boxes per object. The Rescore algorithm Non Maximum Suppression eliminates all the superfluous boxes and leaves only one box per object***. [15]***

Note that in the data augmentation we did not use horizontal flip, because, as mentioned before that would give a very negative effect on classes like 'Turn Right' and 'Turn Left'.

Results

Figure 36. Results from SSD MobileNetv1

From the results shown in figure 36, we can clearly see that the accuracy isn't that great. In fact, with calculations we got an ***mAP of 0.46*** (***46% accuracy***).

So, why did we get such low accuracy, you might ask?

The answer to that question can only be answered if we did some ***data analysis.***

After days of analyzing where the problem might be, I decided to do some data visualization on the GTSDB and that's what I found:

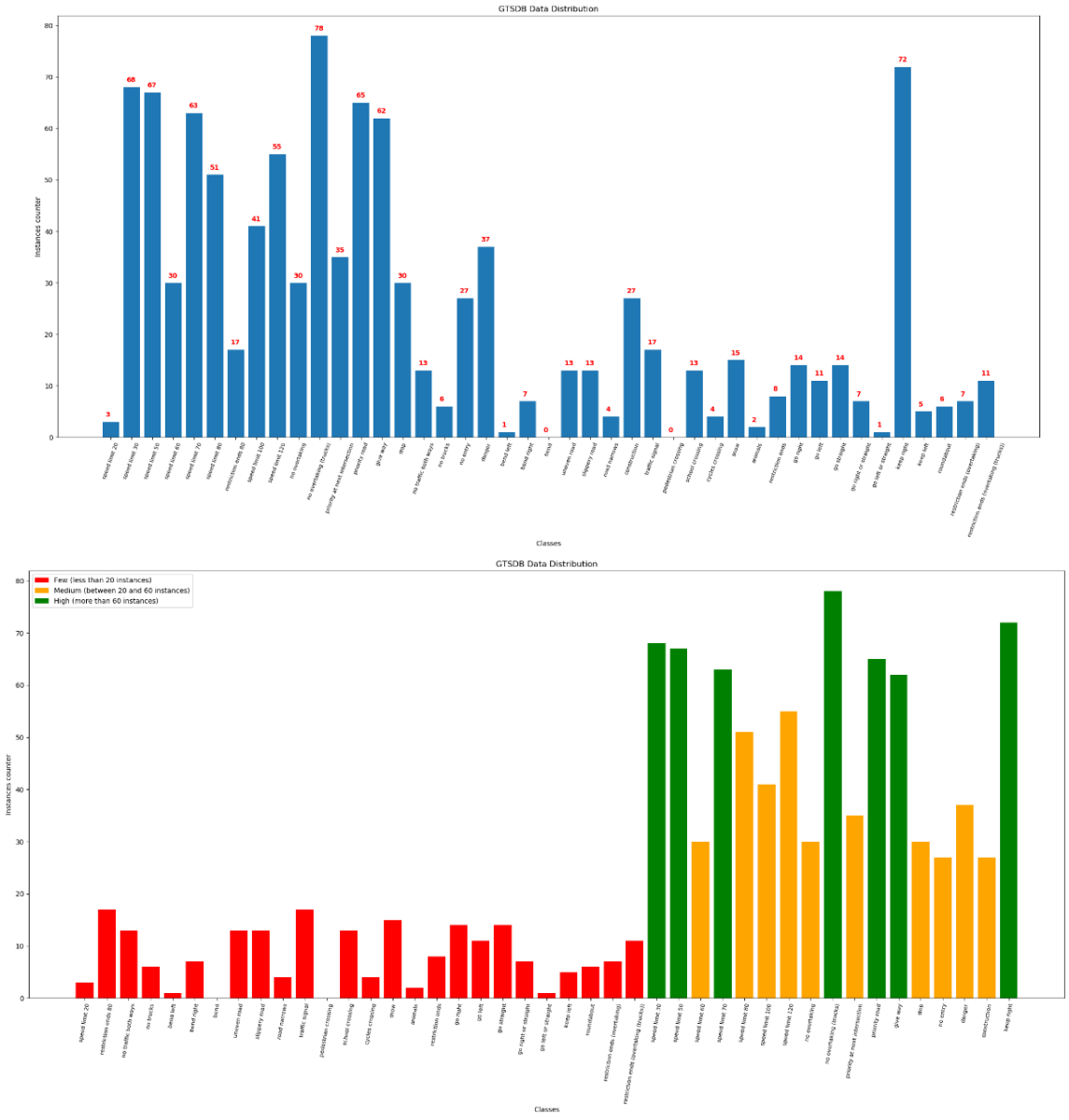

Figure 37. data visualization of GTSDB

Looking at the graph from figure 37, we get one very important thing: some classes have **less than 20 training instances!** And all classes have less than 100 instances. **This is obviously not nearly enough to train a deep learning model.**

Another thing is the entire **GTSDB dataset**. There are **only 900 training samples** (for 43 classes!), which is, again, never enough to train deep learning models.

So, how do we solve this?

One simple solution would be to use 4 classes instead of 43, this would solve the average number of instances per class.

The new data visualization is shown in figure 38.

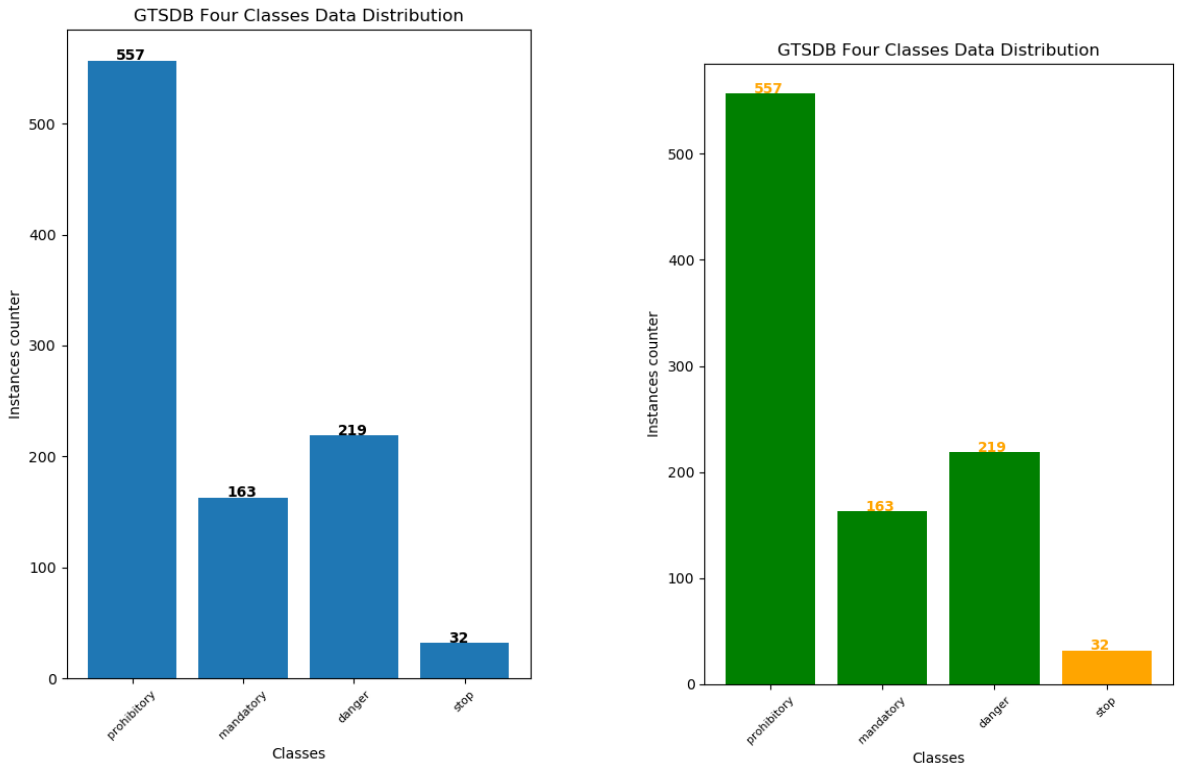

Figure 38. data visualization for 4 classes of the GTSDB

One other thing we found after research is that ***SSD is not suitable for small classes*** since it automatically resizes the input images to 300 * 300 which makes small objects even smaller and thus very, very hard to detect. In addition, **SSD is known for its *high speed* and *low accuracy***.

Therefore, we decided to ditch SSD and try another method – ***Faster RCNN.***

Faster RCNN

R-CNN is a special type of CNN that is able to locate and detect objects in images: the output is generally a set of bounding boxes that closely match each of the detected objects, as well as a class output for each detected object.

Regional Proposal Network (RPN) based approaches such as R-CNN need two shots to detect an object, one for generating region proposals, one for detecting the object of each proposal.

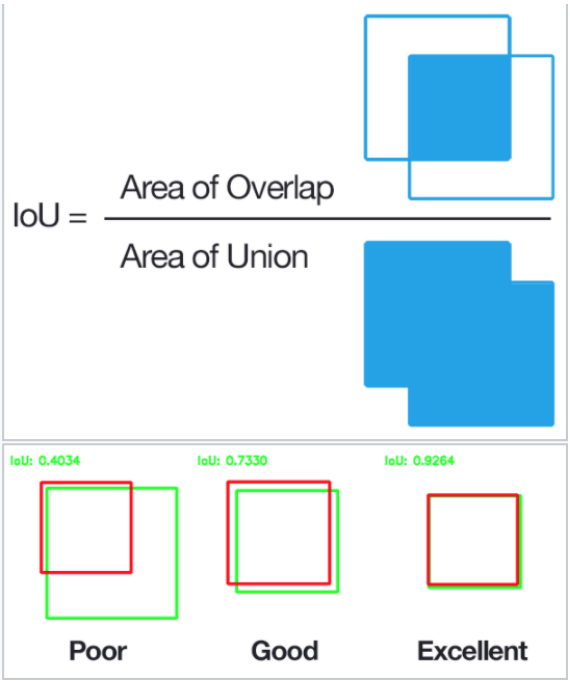

Figure 39. Intersection over Union in FRCNN

Model #1: *FRCNN Inceptionv2*

Training parameters

Model	**F-RCNN Inception v2**
Number of epochs	8.5k
Initial learning rate	0.0002
Learning rate decay	1e-3 every 3000 steps
IoU	0.6
L2 regularization	0.0004
Batch normalization	True Decay: 0.9997 Epsilon: 0.001
Rescore	NMS
Data augmentation	Random grey scale Random 25% crop Random Vertical flip Random horizontal flip
Number of classes	4
Number of training samples	900
Total loss	0.4

Table 8. TSR configuration for FRCNN Inception v2 model

Results


danger: 97%
mandatory: 97%
mandatory: 98%


danger: 96%
prohibitory: 60%


prohibitory: 97%


danger: 80%


danger: 91%


danger: 99%
prohibitory: 99%
30
danger: 93%

Figure 40. Results from the FRCNN Inception v2 model

From the results we can clearly see that FRCNN Inception v2 on 4 classes is much better than SSD MobileNet v1 on 43 classes.

In fact, the recorded **accuracy is 96%**, which according to the GTSDB competition **[16]**, would have put our model in the top-30 spot.

The accuracy is very satisfactory, but we had to keep in mind that the FRCNN Inception v2 model is a very heavy one, thus we had to train yet another model for faster performance.

TEAM	METHOD	AREA UNDER CURVE	AVERAGE OVERLAP
wgy@HIT501	ELL_SVM2 (Prohibitive)	100 %	90.08 %
visics	boosted_intChn_ratio_scales (Prohibitive)	100 %	88.22 %
LITS1	intColorShapeAppearance (Prohibitive)	100 %	86.91 %
glc12125	robok (Prohibitive)	100 %	85.57 %
DSL_clusters	ToneMap+AcfChunks (Prohibitive)	100 %	83.56 %
qichang.hu@adelaide.edu.au	sp_Cov_P (Prohibitive)	100 %	82.03 %
huqc@msn.com	spCov+chns_P (Prohibitive)	99.99 %	82.62 %
shawn_pan	intChn_50scales (Prohibitive)	99.98 %	88.99 %
BolognaCVLab	MSER-1+HOG-2+SVM+Heights+TL (Prohibitive)	99.98 %	84.95 %
BolognaCVLab	MSER-1+HOG+SVM+Heights+TL (Prohibitive)	99.97 %	84.37 %
qcom	VSSD3 (Prohibitive)	99.89 %	91.93 %
wgy@HIT501	HOG_LDA_SVM (Prohibitive)	99.78 %	80.81 %
visics	boosted_intChn_7ops (Prohibitive)	99.12 %	88.88 %
huqc@msn.com	First try (Prohibitive)	99 %	82.54 %
s_trafficsign	DL2 (prohibitory) (Prohibitive)	99 %	86.34 %
wgy@HIT501	ELL_SVM (Prohibitive)	98.99 %	90.17 %
hb@hit501	hb_pro_0.8_0.6_13 (Prohibitive)	98.98 %	79.6 %
LITS1	hog+svm1 (Prohibitive)	98.97 %	88.49 %
VJ+Filter	VJ+LBP+HSV+colorSIFT (Prohibitive)	98.96 %	86.33 %
exp	Multi-features filter (Prohibitive)	98.96 %	86.33 %
exp	exp-multi-pro (Prohibitive)	98.91 %	86.59 %
VJ+Filter	VJ+LBP+HSV (Prohibitive)	98.87 %	86.34 %
NII-UIT	light-colorSIFT_V2 (Prohibitive)	98.11 %	81.71 %
genius07	vgg_detection (Prohibitive)	98 %	90.55 %
VJ+Filter	VJ+LBP+HSV+SIFT (Prohibitive)	97.96 %	86.33 %
VJ+Filter	test (Prohibitive)	97.94 %	86.58 %
LITS1	hog+svm (Prohibitive)	97.73 %	87.8 %
huqc@msn.com	Second try (Prohibitive)	97.02 %	83.02 %
temp	frir (Prohibitive)	97 %	91.79 %
s_trafficsign	DL (prohibitory) (Prohibitive)	97 %	86.33 %
BolognaCVLab	MSER+HOG+SVM+Heights+TL (Prohibitive)	96.01 %	80.45 %
bkmw	haar_upscale (Prohibitive)	95.82 %	84.48 %
exp	Bag of visual word (Prohibitive)	95.23 %	87.64 %
aboltabol	aboltabol3 (Prohibitive)	95 %	92.49 %
NII-UIT	denseSIFT (Prohibitive)	94.65 %	87.48 %
ducluu	lbp_filter_16_20_wsize (Prohibitive)	93.12 %	85.37 %
exp	Bag of visual words (Prohibitive)	92.34 %	87.55 %
ShapeSaliency	ShapeSaliency+Color (Prohibitive)	92.16 %	85.49 %
Test	12_STAGES_UPSIZE (Prohibitive)	91.82 %	82.91 %
Test	test_all (Prohibitive)	91.49 %	82.94 %
code monkey	11D3 prohibitory (Prohibitive)	91.4 %	88.1 %
Test	12 STAGES_UPSIZE (Prohibitive)	91.15 %	82.87 %
hoqmawla	hoqmawla3 (Prohibitive)	91.14 %	83.73 %
INI-RTCV	Viola-Jones (Prohibitive)	90.81 %	87.85 %
code monkey	Modified SSD (Prohibitive)	88.8 %	90.27 %
mraouf.cairo.4	Prohibitory_13_21__ (Prohibitive)	88.6 %	93.28 %
shawn_pan	intChn_MultiScale (Prohibitive)	88.35 %	87.94 %
huqc@msn.com	Combine1 (Prohibitive)	88.11 %	83.66 %
mraouf.cairo.3	Prohibitory_Testing_13_20 (Prohibitive)	87.64 %	93.16 %

Table 9. GTSDB competition

Model #3 YOLOv2

As good as the FRCNN Inception v2 model's accuracy was, the model itself was too heavy for implementation on any embedded system. So we opted to go for YOLO – a model known for its high speed and having very acceptable accuracy as well.

YOLO v2 is the second iteration of the YOLO it is said to be 'faster, stronger and netter' than its older brother.

Type	Filters	Size/Stride	Output
Convolutional	32	3×3	224×224
Maxpool		$2 \times 2/2$	112×112
Convolutional	64	3×3	112×112
Maxpool		$2 \times 2/2$	56×56
Convolutional	128	3×3	56×56
Convolutional	64	1×1	56×56
Convolutional	128	3×3	56×56
Maxpool		$2 \times 2/2$	28×28
Convolutional	256	3×3	28×28
Convolutional	128	1×1	28×28
Convolutional	256	3×3	28×28
Maxpool		$2 \times 2/2$	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Convolutional	256	1×1	14×14
Convolutional	512	3×3	14×14
Maxpool		$2 \times 2/2$	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	512	1×1	7×7
Convolutional	1024	3×3	7×7
Convolutional	1000	1×1	7×7
Avgpool		Global	1000
Softmax			

Table 41. YOLOv2 architecture

We trained a smaller version of YOLOv2 called ***Tiny-YOLOv2*** because it is recommended by the official YOLO GitHub repository to train YOLOv2-full on a Titan XP or higher.

Tiny-YOLO is much faster than YOLO-full, which makes it perfect for embedded systems. However, it's accuracy is considerably lower.

Training parameters

Model	**Tiny YOLOv2**
Number of epochs	1.5k
Initial learning rate	0.0002
Learning rate decay	-
Data augmentation	Random grey scale Random Vertical flip Random horizontal flip
Number of classes	4
Number of training samples	900
Total loss	**2.8**

Table 10. TSR Tiny-YOLOv2 configuration

Results

Figure 42. Results for YOLOv2 model

The Tiny-YOLOv2 model achieved an accuracy of 73% and an average FPS of 70FPS which is really good for real-time.

Conclusion

Model	SSD MobileNet v1	F-RCNN Inception v2	Tiny YOLOv2
Number of epochs	20k	8.5k	1.5k
Initial learning rate	0.0002	0.0002	0.0002
Learning rate decay	-	1e-3 every 3000 steps	-
Data augmentation	Random grey scale Random 25% crop Random Vertical flip*	Random grey scale Random 25% crop Random Vertical flip Random horizontal flip	Random grey scale Random Vertical flip Random horizontal flip
Number of classes	43	4	4
Number of training samples	800	900	900
Total loss	1.5	**0.4**	2.8
Accuracy	-	**95%**	73%
Speed per image	-	0.32	**0.02**
Speed (FPS)	-	25FPS	**70FPS**

Table 11. Comparison between the 3 main detection model

Steps of re-training and testing pre-trained models

Pre-requisites:

1. Python 3

2. Anaconda

It is recommended to install all the upcoming packages in a Conda environment just in case anything goes wrong you can very easily delete the environment and start all over again.

3. SciPy (NumPy, Matplotlib, etc..)

4. OpenCV2

5. Pillow

6. Tensorflow

7. Download and install the Tensorflow Object Detection API

You will also need to install:

- Visual Studio C++ 14 or above

- Protoc and convert the object detection files from .proto to .py

- Pycocotools

8. Keras (optional, but preferable)

Steps

1. Download at the very least 200 images per class to act as your dataset

2. Label the classes present in each image using LabelImg

This will output an .xml file corresponding to each image

3. Split the images accompanying their .xml files into two sets (folders) -- training and validation

4. Convert the .xml files to .csv (optional)

Now you should have two .csv files (containing the images names, RoIs and classes present in each);

one for training and one for validation

5. Create TFRecords for each set of images

6. Create a label_map.pbtxt file containing the ids and names of your classes

8. Download a model of your desire (e.g. SSD MobileNet v1, FRCNN Inception v2) from the Tensorflow Object Detection API model zoo

9. Configure the .config file adequately

You can also find pre-tuned configuration files in the Tensorflow Sample Configs, and then you will have to fine tune some parameters before training

10. Train the model

11. Evaluate the model

This step is very important to get the mAP for each class and see how your model would do in real-life situations,

It is also recommended to train and evaluate your model simultaneously (if you have a high-end Nvidia GPU) in order to see when exactly the loss converges and apply early-stopping manually

12. Export inference graph for the model

This is the saved model you'd use in production.

14. Use the graph to test the model

Code

1. Testing in real-time

```
print("[INFO] Importing libraries...")
import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tarfile
import tensorflow as tf
import zipfile

from collections import defaultdict
from io import StringIO

from matplotlib import pyplot as plt
import matplotlib; matplotlib.use('Agg')  # pylint: disable=multiple-statements
from PIL import Image

import cv2
cap = cv2.VideoCapture(0)

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")
from utils import label_map_util
from utils import visualization_utils as vis_util

print("[INFO] Defining global constants...")
# What model to download.
# F-RCNN INCEPTION V2
MODEL_NAME = 'TSDRS_FOUR_FRCNN_INCV2_inference_graph'
PATH_TO_LABELS = os.path.join('data', 'TSDRS_label_map_four.pbtxt')

# Path to frozen detection graph. This is the actual model that is used for the object detection.
PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'

# Load a (frozen) Tensorflow model into memory
print("[INFO] Loading inference graph...")
detection_graph = tf.Graph()
with detection_graph.as_default():
  od_graph_def = tf.GraphDef()
```

```
  with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')

# Loading label map
print("[INFO] Loading label map...")
category_index =
label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
use_display_name=True)

def load_image_into_numpy_array(image):
  (im_width, im_height) = image.size
  return np.array(image.getdata()).reshape(
      (im_height, im_width, 3)).astype(np.uint8)

# Size, in inches, of the output images.
IMAGE_SIZE = (12, 8)

# configure Tensorflow to utilize the GPU and CUDA
config = tf.ConfigProto()
config.gpu_options.allow_growth = True

def tsdrs_rt_main():
  with detection_graph.as_default():
    with tf.Session(graph=detection_graph, config=config) as sess:
      while True:
        ret, image_np = cap.read()

        # Expand dimensions since the model expects images to have shape:
[1, None, None, 3]
        image_np_expanded = np.expand_dims(image_np, axis=0)
        image_tensor =
detection_graph.get_tensor_by_name('image_tensor:0')

        # Each box represents a part of the image where a particular
object was detected.
        boxes = detection_graph.get_tensor_by_name('detection_boxes:0')

        # Each score represents how level of confidence for each of the
objects.
        # Score is shown on the result image, together with the class
label.
        scores = detection_graph.get_tensor_by_name('detection_scores:0')
```

```
76.         classes =
    detection_graph.get_tensor_by_name('detection_classes:0')
77.         num_detections =
    detection_graph.get_tensor_by_name('num_detections:0')
78.
79.         # Actual detection.
80.         (boxes, scores, classes, num_detections) = sess.run(
81.             [boxes, scores, classes, num_detections],
82.             feed_dict={image_tensor: image_np_expanded})
83.
84.         # Visualization of the results of a detection.
85.         vis_util.visualize_boxes_and_labels_on_image_array(
86.             image_np,
87.             np.squeeze(boxes),
88.             np.squeeze(classes).astype(np.int32),
89.             np.squeeze(scores),
90.             category_index,
91.             use_normalized_coordinates=True,
92.             line_thickness=8)
93.
94.         cv2.imshow('object detection', cv2.resize(image_np, (800,600)))
95.         if cv2.waitKey(27) & 0xFF == ord('q'):
96.           cv2.destroyAllWindows()
97.           break
98.
99.if __name__ == '__main__':
100.         tsdrs_rt_main()
```

2. Testing on static images

```
print("[INFO] Importing libraries...")
import numpy as np
import os
import six.moves.urllib as urllib
import sys
import tensorflow as tf

from collections import defaultdict
from io import StringIO

from PIL import Image
import cv2

# This is needed since the notebook is stored in the object_detection folder.
sys.path.append("..")
from utils import label_map_util
from utils import visualization_utils as vis_util

print("[INFO] Defining global constants...")
# FRCNN_ResNet50 (1000)
# MODEL_NAME = 'TSRP_FRCNN_RN50_inference_graph'
# PATH_TO_LABELS = os.path.join('data', 'TSRP_label_map.pbtxt')

# FRCNN_Incv2
MODEL_NAME = 'TSDRS_FOUR_FRCNN_INCV2_inference_graph'
PATH_TO_LABELS = os.path.join('data', 'TSDRS_label_map_four.pbtxt')

# Path to frozen detection graph. This is the actual model that is used for the object detection.
PATH_TO_CKPT = MODEL_NAME + '/frozen_inference_graph.pb'

# ## Load a (frozen) Tensorflow model into memory
print("[INFO] Loading inference graph...")
detection_graph = tf.Graph()
with detection_graph.as_default():
  od_graph_def = tf.GraphDef()
  with tf.gfile.GFile(PATH_TO_CKPT, 'rb') as fid:
    serialized_graph = fid.read()
    od_graph_def.ParseFromString(serialized_graph)
    tf.import_graph_def(od_graph_def, name='')

# ## Loading label map
```

```
print("[INFO] Loading label map...")
category_index =
label_map_util.create_category_index_from_labelmap(PATH_TO_LABELS,
use_display_name=True)

def load_image_into_numpy_array(image):
  (im_width, im_height) = image.size
  return np.array(image.getdata()).reshape(
      (im_height, im_width, 3)).astype(np.uint8)

# For the sake of simplicity we will use only 2 images:
PATH_TO_TEST_IMAGES_DIR = 'test_images'
TEST_IMAGE_PATHS = [ os.path.join(PATH_TO_TEST_IMAGES_DIR,
'image{}.jpg'.format(i)) for i in range(1, 36) ]

# Size, in inches, of the output images.
IMAGE_SIZE = (12, 8)

# configure Tensorflow to utilize the GPU and CUDA
config = tf.ConfigProto()
config.gpu_options.allow_growth = True

def tsdrs_static_images_main():
  print("[INFO] Creating a session...")
  with detection_graph.as_default():
    with tf.Session(graph=detection_graph, config=config) as sess:
      for image_path in TEST_IMAGE_PATHS:
          print("[INFO] Pre-processing image...")
          image = Image.open(image_path)
          image_np = load_image_into_numpy_array(image)

          # Expand dimensions since the model expects images to have
shape: [1, None, None, 3]
          image_np_expanded = np.expand_dims(image_np, axis=0)
          image_tensor =
detection_graph.get_tensor_by_name('image_tensor:0')

          boxes = detection_graph.get_tensor_by_name('detection_boxes:0')
          scores =
detection_graph.get_tensor_by_name('detection_scores:0')
          classes =
detection_graph.get_tensor_by_name('detection_classes:0')
          num_detections =
detection_graph.get_tensor_by_name('num_detections:0')

```

```
79.           # Actual detection
80.           print("[INFO] Detecting image...")
81.           (boxes, scores, classes, num_detections) = sess.run(
82.               [boxes, scores, classes, num_detections],
83.               feed_dict={image_tensor: image_np_expanded})
84.
85.           # predictions = [category_index.get(value) for index, value in
    enumerate(classes[0]) if scores[0, index] > 0.5]
86.           # if predictions:
87.           #   predicted_class = predictions[0]['name']
88.           #   print(predicted_class, scores[0][0])
89.
90.
91.           # Visualization of the results of a detection
92.           print("[INFO] Visualizing detected image...")
93.           vis_util.visualize_boxes_and_labels_on_image_array(
94.               image_np,
95.               np.squeeze(boxes),
96.               np.squeeze(classes).astype(np.int32),
97.               np.squeeze(scores),
98.               category_index,
99.               use_normalized_coordinates=True,
100.                    # min_score_thresh=0.2,
101.                    line_thickness=8)
102.
103.                 print("[INFO] Showing detected image...")
104.                 image_RGB = cv2.cvtColor(image_np, cv2.COLOR_BGR2RGB)
105.                 cv2.imshow("Image RGB", image_RGB)
106.                 cv2.waitKey(0)
107.                 cv2.destroyAllWindows()
108.
109.             print("DONE")
110.             print("HAIL HYDRA!")
111.
112.     if __name__ == '__main__':
113.       tsdrs_static_images_main()
```

Chapter 5: Implementation

Now that we have successfully trained and tested the models and got a god grasp over their respective results, let's implement the system on various systems.

The Traffic Signs Recognition System was implemented on:

1. iMX6Q board
2. Raspberry Pi 3 model B +
3. TASS PreScan

Implementation on iMX6Q

At first the iMX6Q [18] board was the only available board for us to try the TSR system on. Table 12 shows the board's specs. Looking at the table it is very apparent that it has low-end and it would be a struggle running any deep learning model on.

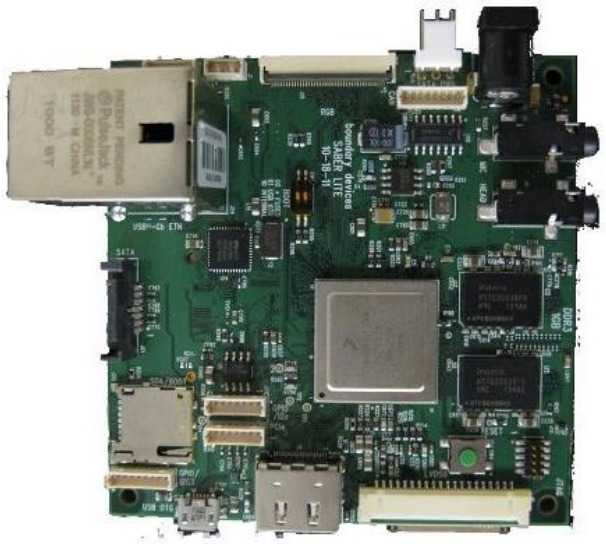

Figure 43. iMX6Q board

	iMX6Q
CPU	**ARM Cortex-A9 (Quad-Core) 1 GHZ**
RAM	**1GB DDR3**
GPU	**Vivante GC2000**

Table 12. iMX6Q specs

Model used: SSDLite MobileNet v2

Knowing the specs of the board, we decided to go with the lightest model available – SSDLite MobileNet. However, even with such a light model, Tensorflow was too heavy for the board's CPU that it crashed every time we tried to create a variable and initiate a session. We have tried other models and researching the problem we found that it's almost impossible to run any model on the board.

Core dumped

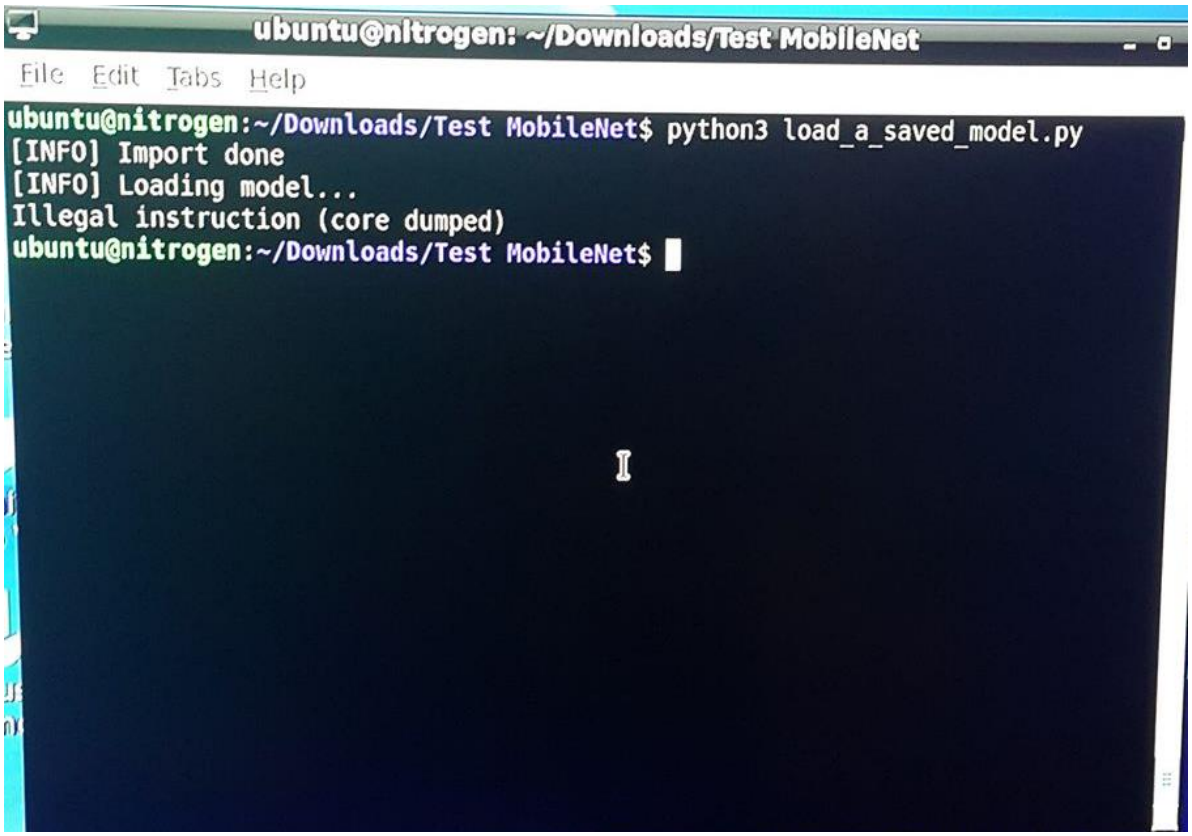

Figure 44. Tensorflow would crash every time we run a session

Implementation on RPi 3B+

After having very little luck testing the TSR on the iMX6Q, we decided to buy a Raspberry Pi 3 Model B+ to test our system. Table 13 shows the specs of the RPi 3B+.

It is also worth noting that the process of installing Tensorflow and its dependencies on the Raspberry Pi was so much easier than on the iMX6Q since it is officially supported by Tensorflow.

Figure 44. RPi 3B+ board

	RPi 3B+
CPU	ARM Cortex-A53 (Quad-Core) 1.2GHz
RAM	1GB LP DDR2
GPU	Dual Core Video Core IV

Table 13. Raspberry Pi 3 Model B + specs

Model #1: SSDLite MobileNet v2

At first we had to make sure that the Raspberry Pi could run Tensorflow and the Object Detection API, so we decided to try the exact same model we tried on the iMX6Q board – SSDLite MobileNet v2.

SSDLite is an even lighter version of SSD used solely for embedded and mobile systems due to the lack of computational power in such systems.

To our elate, the model actually worked! It detected (some) images successfully and it had a speed of 1.5FPS!

However, shortly after we rest assured that the model works we decided to go for a heavier, more accurate model – ***SSD MobileNetv2*** (NOT SSDLite).

Model #2: SSD MobileNetv2 4 classes

After training the SSD MobileNet v2 model on the host PC for 1, 000 iterations, we tested it on the Raspberry Pi 3 and got an accuracy of around 75% and an average speed of 1FPS – which is to my knowledge, enough for real-time applications.

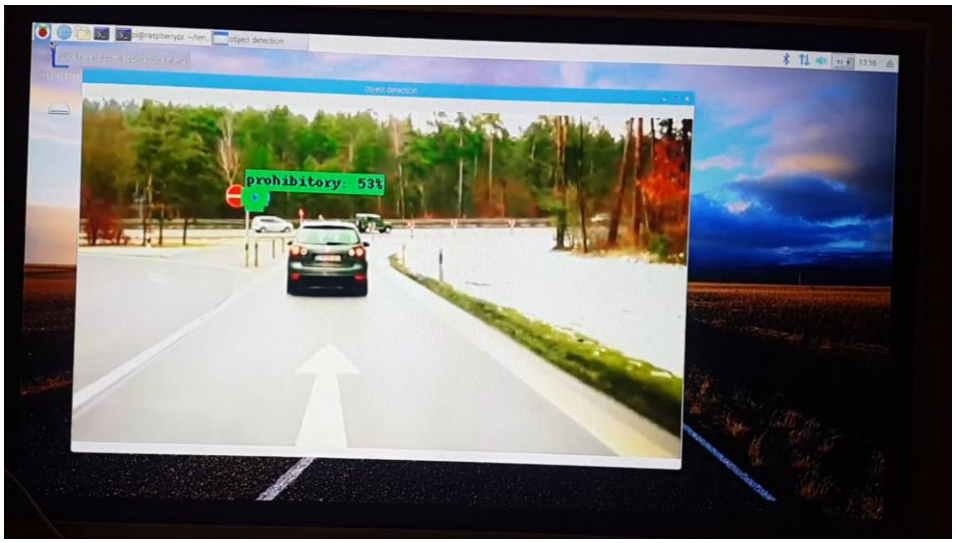

Figure 45. TSR results on the RPi and SSD MobileNet v2 model

Model # 3: SSD MobileNetv1 (43 classes)

After successfully testing the SSD MobileNetv2 (on 4 classes) on the RPi, we decided to give the very early, very low accuracy SSD MobileNetv1 (43 classes) model.

It worked pretty fine for a slightly heavy (in comparison to the RPi specs) giving an accuracy of around 50% and a speed of 1FPS on a **video resolution: 640x480 (480p)**

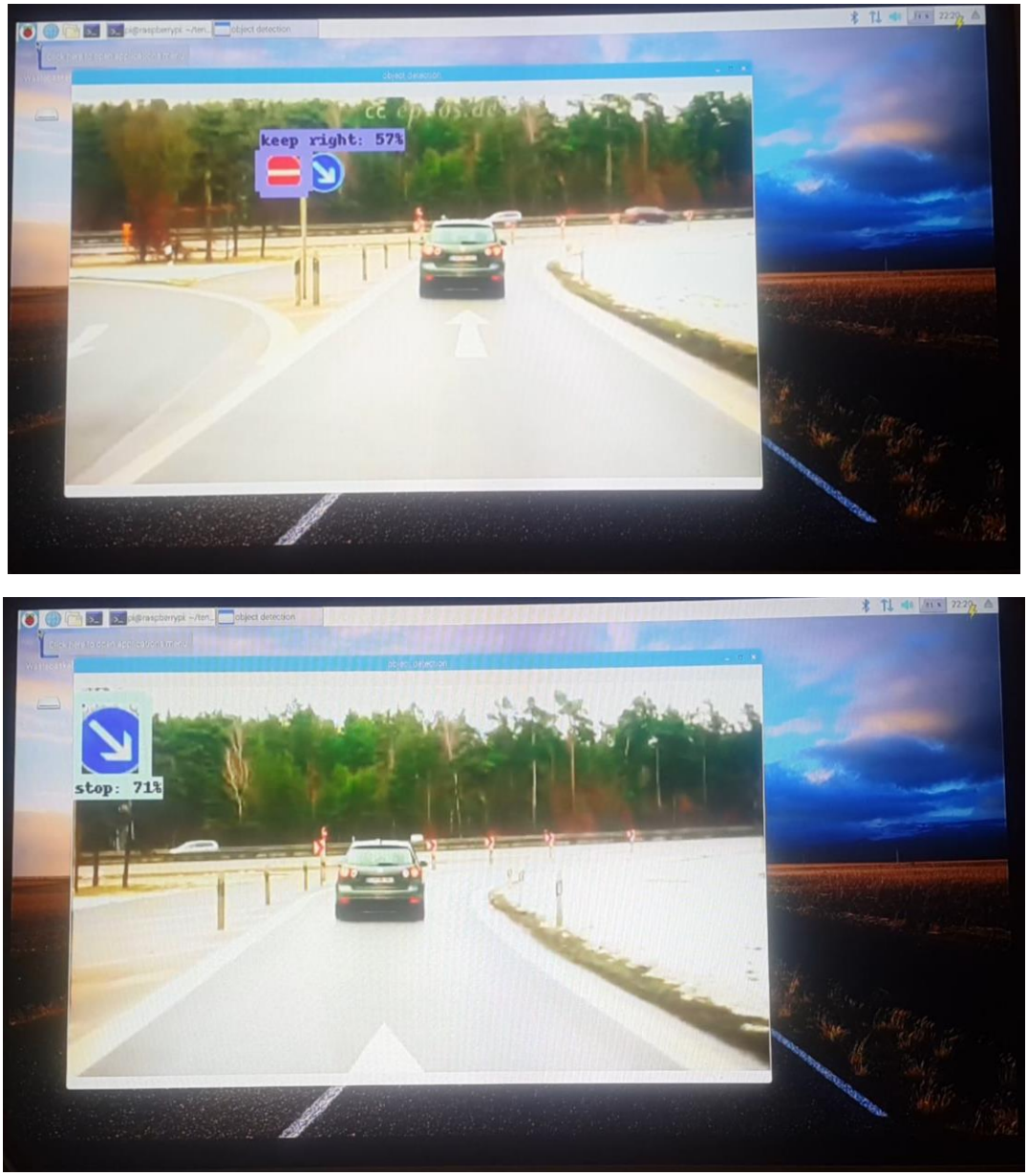

Figure 45. TSR results on the RPi 3B+ using the SSD MobileNetv1 (43 classes) model

Implementation on TASS PreScan

Model used: FRCNN Inception v2

TASS PreScan is a virtual simulation for a self-driving car developed by ***Siemens*** in ***20{}*** with the purpose to
simulate real-life scenarios that might face a self-driving car.

We've had the pleasure to implement our TSDRS in TASS.

Process:

- A real-time video is generated by TASS.
- Each frame is flattened to 1-D.
- The 1-D frame is sent to our Host PC.
- Our algorithm converts the received bytes 1-D frame into a 2D RGB frame.
- The 2D frame is then passed to the F-RCNN Inception v2 model.
- The resulting detected frame is shown and saved in the Host PC.

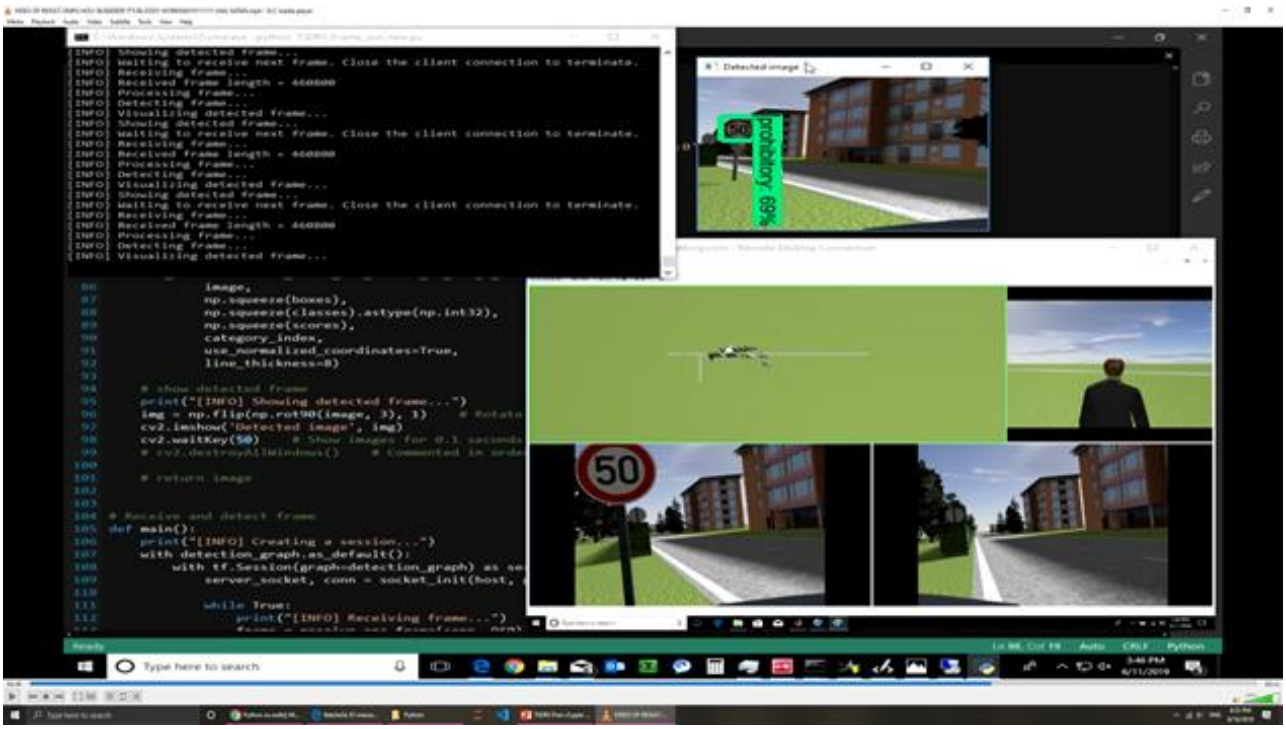

Figure 46. TSR results on TASS PreScan

Models Implementation Conclusion

Parameters	Results		
Model	SSD MobileNet v2 (4)	F-RCNN Inception v2	Tiny YOLOv2
Accuracy	75%	**95%**	73%
Speed per image (Host PC)	0.21	0.32	**0.02**
FPS (Host PC)	30	25FPS	**70FPS**
Speed TASS PreScan	Not tested	**TASS' speed – 4 frames**	Not tested
Speed RPi 3B+	**1FPS**	Could not be loaded	Not tested

Table 14. Comparison between different models and how they performed in the TSR system

References

[1] "A Photo-Realistic Simulation for Computer Vision Applications," by Matthias Mueller, Vincent Casser, Jean Lahoud, Bernard Ghanem, in 2018

[2] "ASIRT Organization," [Online].
Available: http://asirt.org/initiatives/informing-road-users/road-safety-facts/road-crash-statistics.

[3] "San Diego Personal Injury Law Offices," [Online]
Available: https://seriousaccidents.com/legal-advice/top-causes-of-car-accidents/

[4] Ludovic Simon, Jean-Philippe Tarel and Roland Bremond, "Alerting the Drivers about Road Signs with Poor Visual Saliency," in Laboratory for Road Operation, Perception, Simulation and Simulators Universite Paris Est, LEPSIS, LCPC-INRETS, in 2009

[5] "TASS PreScan," [Online]
Available: https://tass.plm.automation.siemens.com/prescan

[6] Chuanqi Tan, Fuchun Sun, Tao Kong, Wenchang Zhang, Chao Yang, Chunfang Liu, "A Survey on Deep Transfer Learning," in Cornell University, 2018

[7] Alexander Shustanova, Pavel Yakimova, "CNN Design for Real-Time Traffic Sign Recognition" in April 2017

[8] "GTSRB," [Online]
Available:

[9] "GTSDB," [Online]
Available: http://benchmark.ini.rub.de/?section=gtsdb&subsection=news

[10] "The Mostly Complete Guide of Neural Networks, explained," [Online]
Available: https://towardsdatascience.com/the-mostly-complete-chart-of-neural-networks-explained-3fb6f2367464

[11] Evgeniou, Theodoros & Pontil, Massimiliano, "Support Vector Machines: Theory and Applications," 2049. 249-257. 10.1007/3-540-44673-7_12, in 2001

[12] Matthew D. Zeiler, Rob Fergus, "Visualizing and Understanding Convolutional Networks," arXiv:1311.2901v3, in Nov 2013

[13] "GTSRB Competition," [Online]
Available: http://benchmark.ini.rub.de/?section=gtsrb&subsection=results

[14] Wei Liu, Dragomir Anguelov , Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, Alexander C. Berg, “SSD: Single Shot Multi-Box Detector, ” arXiv:1512.02325v5, in Dec. 2016

[15] A. Neubeck and L. Van Gool, "Efficient Non-Maximum Suppression," *18th International Conference on Pattern Recognition (ICPR'06)*, Hong Kong, 2006, pp. 850-855.

[16] “GTSDB Competition,” [Online] Available: http://benchmark.ini.rub.de/?section=gtsdb&subsection=results

Other useful links

CS231n Stanford University course on Neural Networks and Computer Vision
https://www.youtube.com/watch?v=vT1JzLTH4G4&list=PLzUTmXVwsnXod6WNdg57Yc3zFx_f-RYsq

http://cs231n.github.io/convolutional-networks/

https://towardsdatascience.com/