Pedestrian Classification with Convolutional Neural Networks

Robin Roy Philip

University of
St Andrews

This thesis is submitted in partial fulfilment for the degree of MSc

at the

University of St Andrews

Date of Submission: 27/08/2017

Acknowledgements

I would like to thank Dr Carl Donovan, my supervisor at the University of St Andrews Statistics Department, who provided me with constant guidance and support throughout this master's project. His supervision was in such a way that was non –intrusive but at the same tine inquisitive in purpose. Finally, he was always present and willing to answer questions I had, helping me have an idea of the end goal I was supposed to achieve. Thank you!

Abstract

Considerable work has been done on the subject of Image Classification using Convolutional Neural Nets (CNN) especially on the subject of detecting pedestrians in an image. This work differs by fitting a cifar10-inspired CNN to a video dataset, which has high variability in the training set due to large differences in background scene settings. Despite such differences, the final model achieves considerable success, with an accuracy of 72.3 % on previously unseen test set with an AUC score of 0.92. The quality of the final is model is also evaluated; with trialling various batch sizes to evaluate model performance on shifting batch sizes. As a final step, an attempt to visualise the third convolutional layer is taken, however with less interpretable results. The report also discusses the shortcomings of the explored methods and suggest alternative approaches where future work can be based on.

I. Contents

II. Glossary of terms

Pixel: Smallest addressable element in all points addressable in a display device.

Batch Size: Defines the number of training samples that is propagated through the network

Epoch: The number of times the algorithm sees the ENTIRE data set. One epoch would equate to all the samples of the dataset seen by the Algorithm

Iteration: An "iteration" describes the number of times a "batch" of data is passed through the algorithm. One iteration would equate to a single “batch” of data seen by the network. For e.g.: With 1000 training examples, and a batch size of 500, will take two iterations to complete one epoch.

Learning Rate: Rate at which the model learns from training samples. The learning rate determines how quickly or slowly the model parameters in a network is updated. A small learning rate refers to smaller updates of the parameters in model, but suspecitble to over-fitting

One-Hot Encoding : Representation using binary digits .For the label "6" by using a vector of 10 values, all zeros but the 6th value which is 1.Useful as the format is very similar to how our neural network outputs predictions, also as a vector of 10 values. For e.g.: six can be represented as 000001000.

Stride: A stride defines the amount of overlap between each convolutional window

Zero Padding: Preserving the original image structure by adding zero’s to the new image structure

III. Introduction

Deep learning using convolutional neural networks (CNNs) is well known and studied, and is almost ubiquitous in modern Image classification problems. .CNNs in particular pose a good tool for this task and produces results similar to humans (Karpathy, github, 2014) .

In this report, we look to work with a pedestrian video set dataset obtained from www.changedetection.net (N. Goyette, Jun 2012) , with the aim of pedestrian classification and detection.

The first part of the report introduces the relevant software and theoretical tools with a classical dataset "MNIST", implemented using Tensor flow (Görner, 2015). The second part discusses the Pedestrian classification and detection problem, and looks to implement the classification problem applied to our own pedestrian dataset. The results are finally evaluated and discussed, along with recommendations for future work.

The problem of pedestrian classification and detection has been well studied and researched using Convolutional Neural Networks (Variyar, 2016) and is a challenging part of computer vision problems. (Deng, 2014).The application of such tools vary widely from self-driving vehicles to traffic cameras

The aim of this work is to use existing tools to implement a classification method in identifying pedestrians, to simply each identify a frame as containing a pedestrian or not, using a model structure similar to that built for training the Cifar-10 dataset. A possible framework for further extending this dataset for detection is discussed but not implemented due to time constraints. As a further note, this work focuses very little on feature extraction methods and uses the CNN pipeline to do feature selection itself.

A. Artificial Neural Networks and Convolutional Neural Networks:

In an information-processing paradigm, neural networks are inspired by the way biological nervous systems such as the brain process information. By definition, neural networks composes a large number of highly interconnected processing elements (neurons) working together to solve specific problems. Similar to people, ANN's learn by example. As in the case of biological systems, ANN's learn involving adjustments to the synaptic connections that exist between the neurones.

1. Definition of an Artificial Neural Network:

We start with first simply defining a neural network. A neural network can be defined as a collection of connected perceptron's. A perceptron transforms a given set of inputs into desired outputs (Fig1**).**

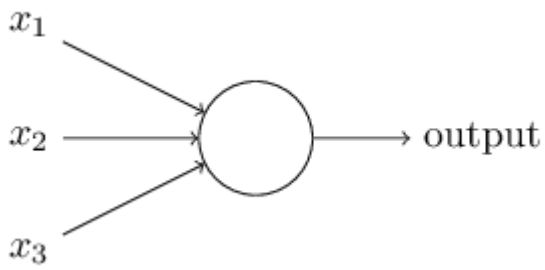


Figure 1: A single perceptron (Image from (Nielsen, 2015))

The output of the perceptron is determined by a set of weights,w_1, w_2, w_3 which expresses the importance of respective inputs to the output as in Eq1. The firing of a perceptron (output= 1), is also determined by a threshold value, which can be expressed as below:

$$output = \begin{cases} 0 & if \ \sum_j w_j x_j \ \leq threshold \\ 1 & if \ \sum_j w_j x_j \ \geq threshold \end{cases} \quad (1)$$

The threshold then determines how easy it is to get the perceptron to fire. The above, threshold can be rewritten in terms of bias (-threshold). A high bias, will easily fire a perceptron while an unbiased perceptron will output zero.

$$output = \begin{cases} 0 & if \ \ w.x + b \ \leq 0 \\ 1 & if \ w.x + b \ \geq 0 \end{cases} \quad (2)$$

2. From Perceptron to Sigmoid Neurons

A perceptron however is not suitable for small changes in weights as the function as it is non-continuous and hence non-differentiable i.e. training and updating the weights becomes difficult. The problem can be overcome by the use of sigmoid neurons, which adapts well to small changes and updates to produce smaller changes in output. A sigmoid function or a neuron can be defined as:

$$\sigma(z) \equiv \frac{1}{1+e^{-z}} \qquad (3)$$

The advantage of using sigmoid neurons, come from the fact that inputs z can be any real –valued number , while the output $\sigma(z)$ is constrained to be between 0 and 1 i.e. maps z from 0 to 1 . This can be useful when looking at the average intensities of the pixels in an image.

3. A Fully Connected Neural Network:

One is able to construct a neural network by composing the neuron into many layers. The left most layer, “the input layer” contains the input neurons. The middle layers or “the hidden layers”, which help make decisions for the output layers with the appropriate initialisation of weights. Finally, the rightmost layer is final output layer, which can obtain predictions in the required format.

A Fully Connected layer can be then formed, by connecting each neuron with every other neuron in the previous layer and connection has its own weight. Fig2. Shows a fully connected Neural Network with three hidden layers.

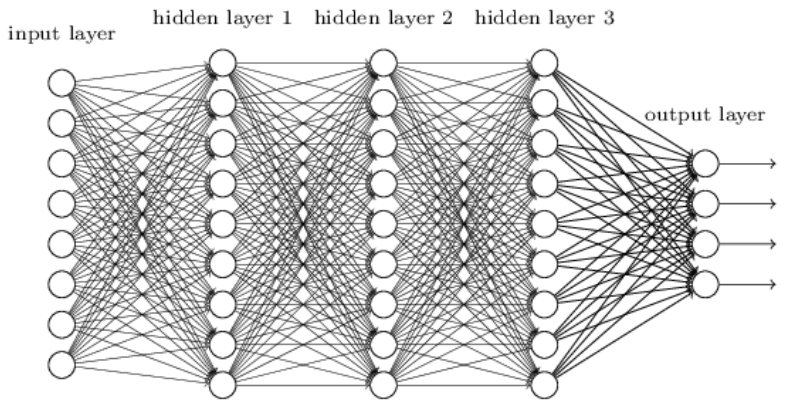


Figure 2: A Fully connected layer with three hidden layers (A.Nielsen, 2015)

4. Convolutional Neural Networks:

Convolutional Neural networks form an extension of the neural network discussed and have been applied effectively in classification of images (Y. LeCun, 1989).However CNN's have found applications in a wide range of fields from human pose estimation to document analysis . With the advent of GPU's, CNNs have gone through a renaissance phase, which allowed efficient ways to train them.

Let –Net (Yann LeCun, 1998) one of the first convolutional networks, helped propel the field of deep learning, developed by Yann LeCun after many previous successful iterations since the year 1988. Since then, AlexNet demonstrated significant improvement on the image classification tasks (Krizhevsky A. I., 2012) . Following the success of AlexNet, several publications such as Google Net, VGGNet, ZFNet and ResNet have shown to improve the image classification performance.

Convolutional Networks derives its name from the mathematical operation "Convolutional Operator". The primary purpose of the convolution is to extract features from the input image. For image classification problems, convolutional layers are able to recognise local feature such as edges or corners by restricting the respective fields of hidden units. As an example, an image with the $28 * 28 = 784$ pixels can be expressed with each pixel a neuron .However the standard NNs do not take into account the spatial structure of the images. (I.e. distance between pixels is not taken into account).

Convolutional Neural Nets take advantage of the spatial structure, which has many layer networks making it good at classifying images. We can describe the filters and feature maps in a CNN as follows:

- The neurons make connections in small-localized regions (*local receptive field*) of the input image. Each connection learns a weight and a parameter estimated.
- We slide the local receptive field across the entire input image, for each input neuron, the local filter produces a hidden neuron in the 1st hidden layer by weighting with the same weights and biases.
- Neurons in the first hidden layer detect the same feature, just at different locations in the input image also known as a "feature map". (Intuitively: a vertical edge picked up can be picked up elsewhere in the image using the same set of weights and biases) .This method also works well for translational images or rotation.
- The shared weights and bias define the kernel or filter, thereby reduces the model complexity compared to ANN's

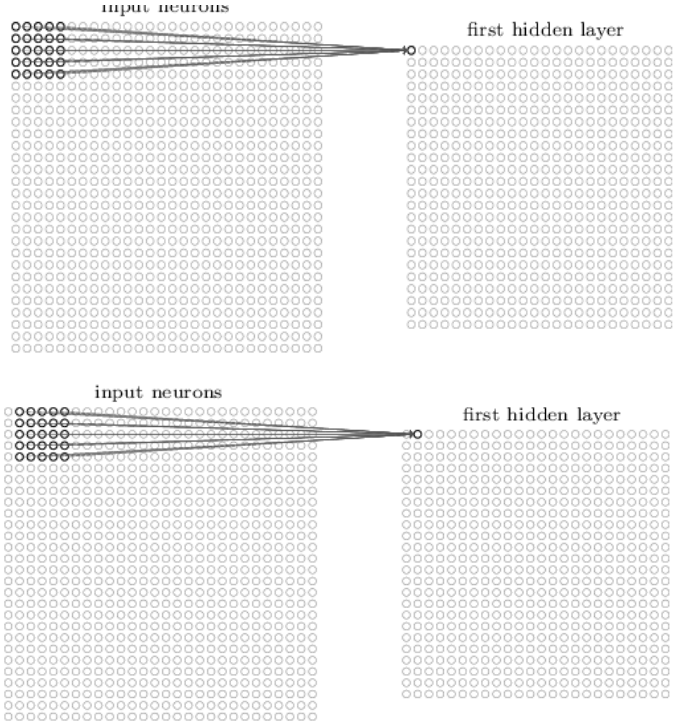


Figure 3: 1st and 2nd Neurons scanning their respective local receptive fields

- Many feature maps (a convolutional layer) can be created based on different set of shared weights for different features. For three different feature maps, the hidden layer has three hidden sets of weights.
- Advantage of sharing weights and biases for feature maps: say for $5 \times 5 = 25$, have 25 identical weights and a single shared bias. In addition, if we have 20 feature maps: 20×26 feature- maps that define single CN layer. In comparison with NN, for 784 input neurons, and 30 hidden neurons +30 biases, we get$784 \times 30 + 30 = 23550$ parameters for estimation. Furthermore, the translational invariance by the CN layer reduces the number of parameters required to get the same performance as a fully connected (each neuron per input) NN.

B. Mechanics of Convolutional Neural Networks:

Convolutional Neural Networks (CNNs) transforms a given set of inputs into outputs. They differ from Neural Networks such that they allow for easier encoding of certain features of an image into the architecture. To "train" a neural network, a large number of training examples are usually required shown, after which the network iteratively modify its weights to minimize the errors and thereby "learns".

Some of the most common learning techniques for convolutional neural networks we discuss:

1. Gradient Descent:

One of the , the most successful approaches are by popularized in recent years nit the NN community can be called "numerical " or gradient based learning .The learning machine computes a function $Y^p = F(Z^p, W)$, where Z^p is the p^{th} input pattern and W presents the collection of adjustable parameters in the system.

A loss function $E^p = D(D^p, F(W, Z^p)$ measure the discrepancy between D^p, the "correct" or desired output for the pattern Z^p , and the output produced by the system (A.Nielsen, 2015). The performance is estimated by measuring the accuracy on a set of samples disjoint from the training set, which is called the test set.

The general problem of minimizing a function with respect to a set of parameters is at the root of many issues in computer science and statistics. Gradient-based learning draws on the fact that it is generally much easier to minimize a reasonably smooth, continuous function. The loss function can be minimized by estimating the impact of small variations and the direction of the changes of the parameter values on the loss function.

2. Stochastic gradient Descent:

Another popular minimization procedure is the stochastic gradient algorithm, also called the online update (Bottou, 2010). Here, the parameters are updated using an approximated average gradient. The new set of parameters, W_k is a real valued vector, updated based on the gradient of a single sample or batch $\nabla(Q_i(w))$ with a learning rate ε as in Eqtn 4.

$$W_k = W_{k-1} - \varepsilon\nabla\big(Q_i(w)\big) \qquad (4)$$

With this procedure, the parameter vector fluctuates around an average trajectory but it converges considerably faster than regular gradient descent and second order gradient methods on large training sets.

For the purpose of this report, a stochastic gradient descent (SGD), where at each iteration our error surface is estimated with respect to each example.

The key advantage of using a stochastic gradient descent is the error surface becomes more dynamic and improves our ability to navigate across regions of the parameter space. The optimiser resulted in better convergence and validation accuracy results as discussed later. Though simple and effective, the SGD requires careful tuning of model hyper parameters, specifically the learning rate and initial parameter values.

C. Techniques to prevent overfitting in Convolutional Neural Networks:

When convolutional layers; have a huge number of neurons the network will likely be over-fitted due to the network learning the training data very well. We discuss here the implemented and most common techniques to prevent overfitting:

1. Pooling layer

The Pooling layer takes each feature map output from the convolutional layer and prepares a condensed feature map. Each unit in pooling layer may summarize a region of 2×2 neurons in the previous layer. One common method, is to take the max of each region (maxfilter) and produce a feature known as “max pooling” as seen in Figure 4 .

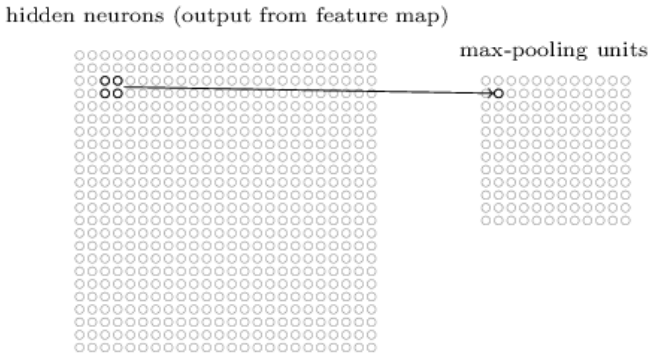


Figure 4: Pooling of layers (A.Nielsen, 2015)

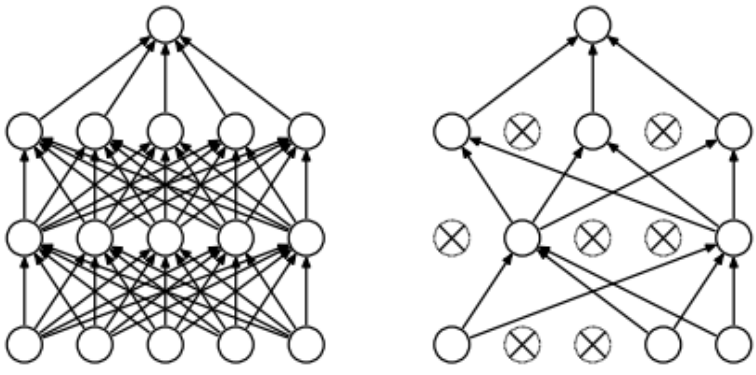
2. Max-norm Constraints

Another form of regularization is to enforce an upper bound on the magnitude of the weight vector for every neuron and use a projected gradient descent to enforce the constraint (Nitish Srivastava, 2014). In practice this, corresponds to performing the parameter updates as normal and then enforcing the constraint by clamping the weight vector of every neuron to satisfy $\left|\left|w\right|\right|_2 < c$. Typical values of "c" are in orders of three or four (Karpathy, CS231n: Convolutional Neural Networks for Visual Recognition, 2017). One major advantage of this method is that the networks do not "explode" even when learning rates are set too high because the updates are always bounded.

3. Dropout

A deep neural network has a large number of parameters and may suffer serious overfitting. *Dropout* is a technique that prevents overfitting and ability to combine exponentially many different neural networks efficiently. The term "dropout" refers to dropping out neurons in a neural network, which means temporarily removing the neurons in the network from incoming and outgoing connections (Nitish Srivastava, 2014) as in Figure 5. Dropout can be therefore interpreted as sampling a neural network within the full Neural Network, and only updating the parameters of the sampled network based on the input data (Karpathy, CS231n: Convolutional Neural Networks for Visual Recognition, 2017).

Figure 5: Left (a): A standard neural network with two hidden layers Right (b) A thinned out neural network after applying dropout. (Nitish Srivastava, 2014)



(a) Standard Neural Net (b) After applying dropout.

4. Batch Normalisation

In the process of training the weights of the network, the output distribution of the neuron in the bottom layer begins to shift. The result of the changing distribution of the neurons in the bottom layer means that the top layer not only has to learn how to make the appropriate predictions but it also needs to somehow modify itself to accommodate the shifts in incoming distribution. This shift in layers is known as “covariate shift” (Shimodaira, 2000)

This internal covariate shift significantly slows down training, and the magnitude of the problem compounds with more layers. Batch Normalisation provides an appropriate solution to this problem.

For a layer with d-dimensions and k^{th}input $x^{(k)}$, each dimension can be normalised with the expectation and variance of the input given by :

$$x^{(k)} = \frac{x^{(k)} - E[x^{(k)}]}{\sqrt{Var[x^{(k)}]}} \qquad (5)$$

In other words, each batch is mean-centred and scaled according to unit standard deviation as in Eqtn 5.

We use mini-batches to produce estimates means and variance of each activation. Moreover, the resulting statistics can fully participate in the gradient back-propogation.Adding batch Normalisation to a state-of-the-art image classification model, yields a substantial speed up in training with reducing the side effects of overfitting (Sergey Ioffe, 2015).

5. Data Augmentation

Another common method to reduce overfitting on image data is to artificially enlarge the size of the dataset by applying various translations and reflections. We discuss more this more in detail in our Methodology.

D. Hyper-Parameter Optimisation:

The primary challenge in optimizing deep learning models is that we are forced to use minimal local information to infer global structure of the error surface i.e. find optimal choice of hyper parameters . In practice and in this report, an educated guess is made of the CNN's network architecture .However, specific algorithms exist where the optimisation can be applied in a structured way.

Many optimisation procedures exist for this task, such as the Grid search, Random Search etc. (Karpathy, CS231n: Convolutional Neural Networks for Visual Recognition. , 2017). However, in such a case the hyper-parameter space is large and expensive so we discuss an evolutionary algorithm (Erik Bochinski, 2017).

Evolutionary Algorithms (EA) are biologically inspired by Darwin's theory of evolution. The core aspect of EA are based on the concept of survival of the fittest in a population *P* that evolved in a twostep manner:

- At First, individuals are modified by crossover or mutation.
- Fitness based selection is applied to produce the next generation.

Therefore, it is an iterative process and the optimum is sought from different positions simultaneously. Hence, the problem of trapping in a local minimum is avoided.

IV. Software and Implementation

Deep learning and most of the available toolkits are mostly implemented in Python. Tensor Flow, one of the leading tools in deep learning used, with model implemented using Keras, which is explained later in this section.

A. Tensor Flow

The Tensor Flow API and a reference implementation were released as an open-source package under the Apache 2.0 license in November 2015 and are available at www.tensorflow.org.

Tensor Flow allows makes it easier to build, design and train deep learning models. It is a Python library that allows users to express arbitrary computation as a graph of data flows, which improves computation speed and allows for parallel programming (Abadi, et al., 2015). Nodes in the graph represent mathematical operations, whereas edges represent data that is communicated from one node to another. Data in Tensor Flow represented as multidimensional arrays known as tensors. This representation allows for clean, expressive method for implementing methods all the while taking advantage of faster computation times by GPU acceleration of parallel tensor operations.

B. Using Tensor Flow variables and Operations:

A deep learning model and its parameters are defined by the following properties in Tensor Flow:

- Variables are explicitly initialized before a graph is used for the first time.
- Gradient methods to modify variables after each iteration , to find optimum parameter values
- Values stored in a variable so can be saved to disk and restore them for later use.

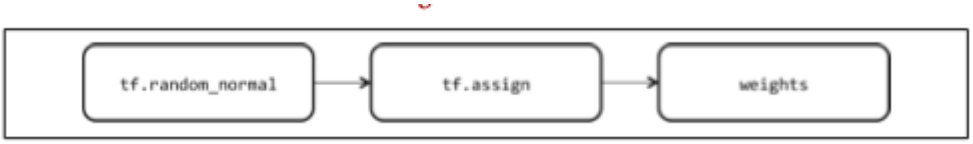


Figure 6: Three operations are added to the graph when instantiating a Tensor Flow variable. In this example, we instantiate the variable weights using a random normal initializer. (Buduma, 2017)

Tensors are similar to multi-dimensional arrays or matrices can have operations applied to them .Tensor Flow operations represent abstract transformations that are applied to tensors in the computation graph. An operation consists of one or more kernels that represent device-specific implementation. Therefore, an operation may have separate CPU and GPU kernels because it can be expressed more efficiently in a GPU setting.

C. Placeholder and Sessions in Tensor Flow:

Inputs to a model during train and test times are defined using a placeholder, unlike a variable, placeholders can be populated each time the computation graph is run.

Sessions in Tensor Flow allow the program to interact with the computation graph. A Tensor Flow session is therefore responsible for building the initial graph and can be used to initialize all variables appropriately and to run an example computational graph as seen in Figure 7.

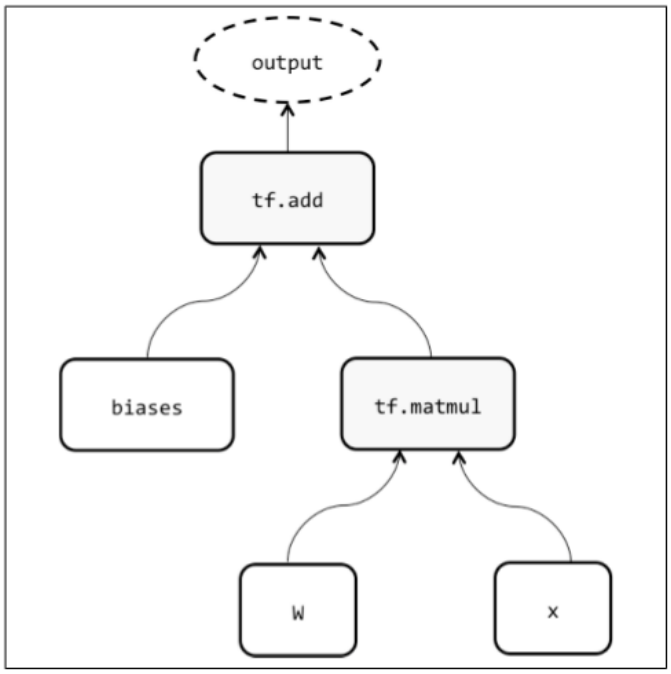


Figure 7 : An outline of an example computational graph in Tensor Flow for a feed forward Neural Network (Buduma, 2017)

As seen above, the weights w and biases are initialised as variables using *sess.run (init_op)* in Tensor Flow. The graph is then run with all the operations (tf.matmul ,tf.add) by calling *sess.run* ,in addition with a feed dictionary that fills the placeholders with the necessary input data x. The *sess.run* is then used to train networks (Buduma, 2017).

D. Keras and TF-learn using Tensor Flow as backend

Keras and TF-learn are simply wrappers around more complex numerical computation engines such as Tensor Flow .Using Keras abstracts away much of complexity of building a deep neural network from scratch, by allowing an easier user interface to work with (Keras, 2017). For the purpose of this report, we start to work with Tensor Flow for the classical MNIST dataset, however resort to Keras for the pedestrian dataset for ease of use.

E. CPU & GPU Programming:

Due to the relative size and ease of the MNIST dataset, tensor flow calculations are implemented using a ASUS CPU with intel Core 13. While computations of all models for the pedestrian dataset are carried out using a designated computer in the Jack Cole Building of the Computer Science Department at the University of St Andrews. The computer had a model name “Inter R Core ™ i5-4460 NVIDIA GTX 1080 GPU and 4 CPU Cores. The OS installed in the computer was Ubuntu 16.04. The computer also has a GPU accelerated version of tensor flow installed in a Python virtual environment. A 3.5.3 Python version and a Tensor flow 0.12.0 was used for this project.

F. OPENCV:

An open source library, written in optimized C, which has bindings in Python, is used as main tool to process images and videos.

V. Tensor Flow using MNIST DATASET

This Section acts as an exemplar which focuses on applying Tensor Flow and its tools to the MNIST dataset and reproducing results similar to the current best performing models.

MNIST dataset represents a database of handwritten digits by high school students and employees of the United States Census Bureau (LeCun, Cortes, & Burges, n.d.), with a training set of 60,000 examples and a test set of 10,000. Here we simply replicate the previous error rate achieved with convolutional networks (Cires¸an, Meier, & Schmidhuber, 2012), using various neural and convolutional network architecture.

A. Evaluating the results for MNIST dataset:

Cross entropy is commonly used to quantify the difference between two probability distributions.

The cross entropy function can be defined as:

$$H_{\hat{y}}(y) := -\sum \hat{y}_i \log(y_i) \qquad (6)$$

Where $\hat{y}$ is the predicted probability distribution and the y_i is the actual probability distribution. For multi-classification problems such as this, classification error is a very crude measure error as we see in section IX , a highly imbalanced response class will return good accuracies however a very useless model in terms of predictive power.

B. Results

Each model architecture is implemented using tensor flow and the model run for 10K iterations with a batch size of 100 images .The results are displayed in the table below (results are slightly variable dependent on the runs) :

Model	**Number of Layers**	**Hidden-layer Output function**	**Learning Rate**	**Model Training Accuracy**	**Model Test accuracy**	**Convolutional Neural Network**
1	1	SIGMOID	0.005	93%	92.62%	No
2	5	SIGMOID	0.005	98%	96.31%	No
3	5	RELU	Exponential decay with each iteration	100%	98.18%	No
4	5	SIGMOID	0.005	100%	98.81	Yes

Table 1 : Results for the MNIST data implemented using Tensor Flow.

C. Discussion

1. Model 1: Linear Classifier using a single Layer with softmax function:

The simplest classifier can be regarded as a linear classifier, each input pixel value contributes to a weighted sum for each output unit

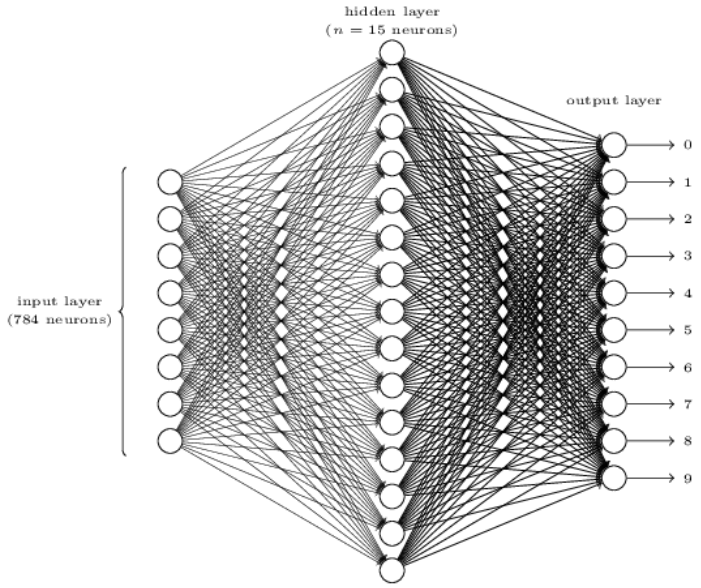


Figure 8: A Simple linear classifier using a softmax function (A.Nielsen, 2015)

The input layer comprises of 784 neurons (28*28 pixels) and output layer producing 10 digits. The input values contain values in greyscale 0 for white, 1 for black with digits one-hot encoded. Each input pixel value contributes to a weighted sum for each output unit. The output unit with the highest sum (including the contribution of a bias constant) indicates the class of input character, here the output function is the softmax function – a generalisation of the logistic function that "squashes" a K-dimensional vector z of arbitrary real values in to a K-dimensional vector given by:

$$Softmax = \frac{e^{z_j}}{\Sigma_{k=1}^{K} e^{z_k}} \; for\; j = 1, \ldots., K. \qquad (7)$$

In relation to probability theory, the softmax function can be considered as a multinomial logistic regression problem with K different possible outcomes. The results show the model performs the worst in comparison with training and test errors when compared with other models.

2. Model 2& 3: A Five layer Neural Network with Sigmoid and Relu as output functions in the hidden layers

To improve on Model 1, we create a deeper model with five hidden layers and evaluate the model using Sigmoid and Relu as different output functions. Furthermore, in Model 3, the learning rate lr is changed to an exponential decay function as in Eqtn 8. We define a maximum and minimum learning rate (lr_{max}, lr_{min}) and allow the learning rate to decay with each increasing iteration "I" with "Total I" iterations altogether. Hence, the model starts learning quickly and then slows down to avoid overfitting and thereby improve the test accuracy (Görner, 2015).

$$lr = (lr_{max} - lr_{min}) \times e^{-\frac{i}{Total\ I}} \qquad (8)$$

Although Model2, has great improvement in training and test accuracy we obtain a 100% training error in Model 3 , when hidden layers are fitted using a Relu function. However the test error performs slightly in worse in such case, this can be apparent when plotting the cross entropy loss over the iterations.

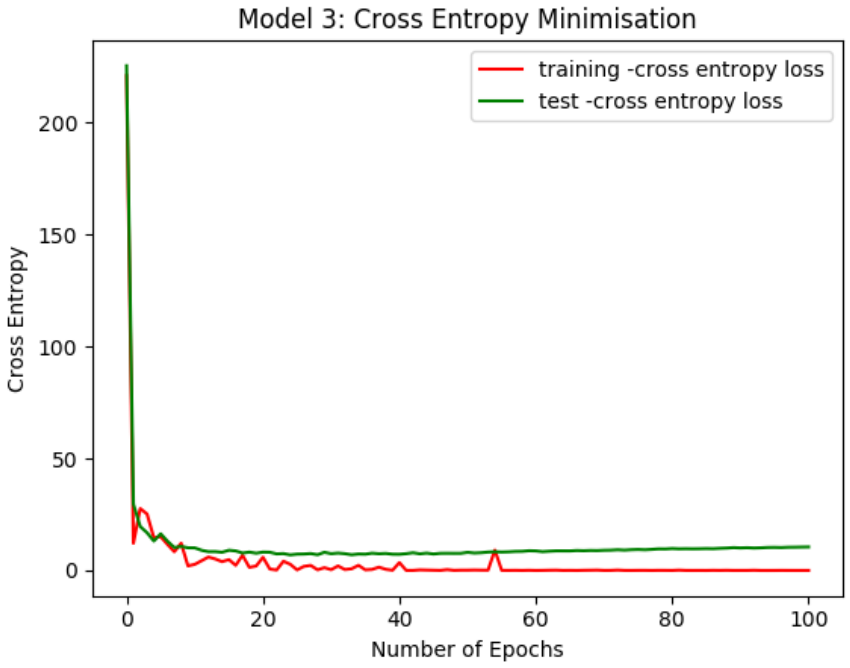


Figure 9: Cross Entropy Minimisation for Model 3

We see the cross-entropy curves for test and training data start disconnecting after a couple thousand iterations. As the learning algorithm works on training data only and optimises for the training cross-entropy accordingly. The algorithm never sees test data, and therefore the loss function stops dropping and bounces back up, due to overfitting.

3. Model 4: Convolutional Neural Nets (CNN's) on MNIST dataset

The previous models have the disadvantage of flattening the initial images (28*28) into a single vector with one dimension. The procedure of flattening images loses information (as the pixels are related to neighbours), convolution layers have the advantage they retain the spatial information. The main differences can be outlined as follows:

- The weights used are used the same weights and biases for one whole image (each neuron (28×28) has the same weights in both directions).
- Rescan the image with different set of weights producing a different convolutional layer.
- Output values same as the pixels no of original image, by zero padding.
- The neurons decrease by a factor of 2 ,for each layer
  $784 \times 4 => 14 \times 14 \times 8 => 7 \times 7 \times 12 => 200$

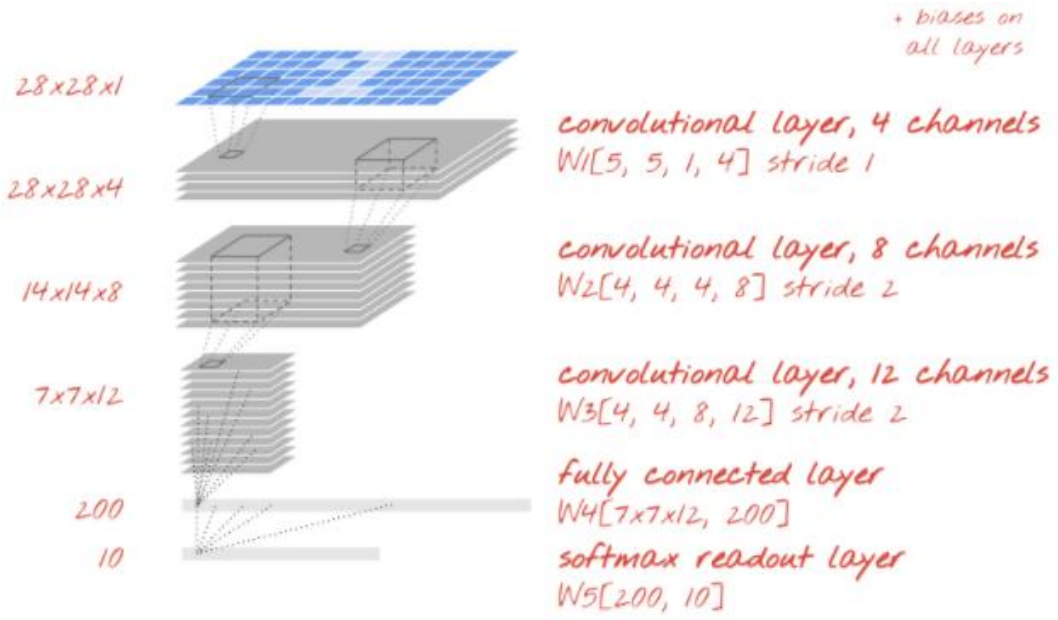


Figure 10 : An illustration of the implemented CNN (Görner, 2015)

For completeness, we plot the training and test accuracies for the 5-layer convolutional layer is plotted as seen in Figure 11.

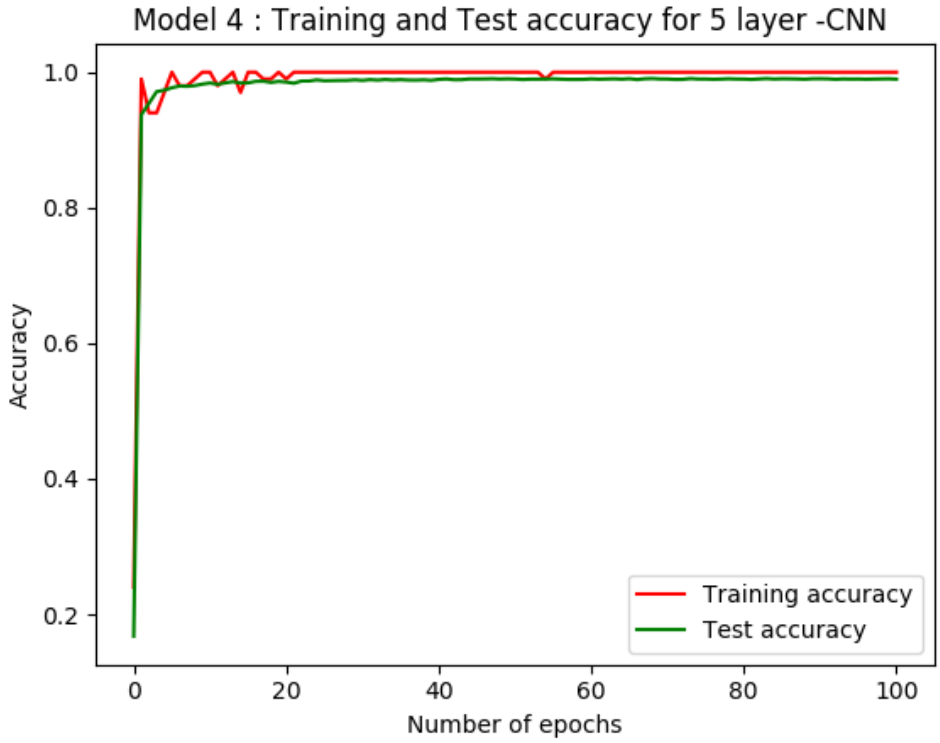


Figure 11 : Model 4: Training and Test accuracies

The test accuracy obtained is 0.81% off from the current world record of 99.7% (LeCun, Cortes, & Burges, n.d.)

D. Conclusion

The section of the report, introduced neural networks and convolutional neural networks along with the theory involved. The software necessary for implementing CNN's in python is described and applied tools to a classical dataset such as the MNSIST dataset, as a pre-requisite learning step before working with the actual pedestrian dataset. The readymade nature of the dataset required for no pre-processing steps combined with the fact a large amount of work done in literature allows for easy reproducibility. A simple logistic model is also fitted and compared with various neural nets and convolutional neural networks with the results summarized. The next section of this report looks at implementing the developed tools and theory to a real life problem of Pedestrian detection using the afore-mentioned Pedestrian dataset.

VI. The Pedestrian Detection Dataset

The Pedestrian dataset used in the modelling process, is part of a subset of the ChangeDetection2014 dataset. The dataset contains 10 videos, ~30K frames that mostly contain pedestrians from 10 different scene settings.

Scene a: Backdoor

Scene b: Skating

Scene c: peopleInshade

Scene d: copy Machine

Figure 12: A random subset of frames containing pedestrians from pedestrian detection dataset

For each frame, the following information is provided as a ground truth .txt file:

- The number of pedestrians in a frame
- The coordinates of the bounding boxes ,provided in the form of (frameID,0 x,y,width ,height)

The specific dataset was chosen with in mind to understand the performance of CNN's with a dataset that provided with a hard classification problem. This is apparent in Figure 12 , where the training images vary widely across the scenes mentioned.

A. A Brief Introduction to Pedestrian Detection Problems

Pedestrian detection is an important aspect of autonomous vehicle driving as recognizing pedestrian's helps in reducing accidents between the vehicles and the pedestrians. In literature, feature based approaches have been mostly used for pedestrian detection; here a Convolutional neural network is tried with various degrees of success. Images are classified as either containing a pedestrian or not.

It is important note is to make the distinction between classification and detection problems. Classification problems as such, which this report is focussed on, are mainly concerned with defining the class of an image, while detection tasks are concerned with identifying the coordinates of an object on the image then classifying the image post-detection. Hence, classification can be considered a subclass of pedestrian detection problems. One of the earliest work of detecting people was carried out Shirai in 1985, looking at detection of people from a sequence of TV images.

Person Detection problems can be divided into the following categories (Chen, 2012). :

1. Static problems:

Static pedestrian detection focuses on methods, which are model-based, and feature based classifiers. As static problems do not require motion information, we can apply the model for single images or frames. The two methods can be described as follows:

1. *Model-based Pedestrian detection*
   In Model based Pedestrian detection, a specific pedestrian model is built and an image is scanned for the matched region.

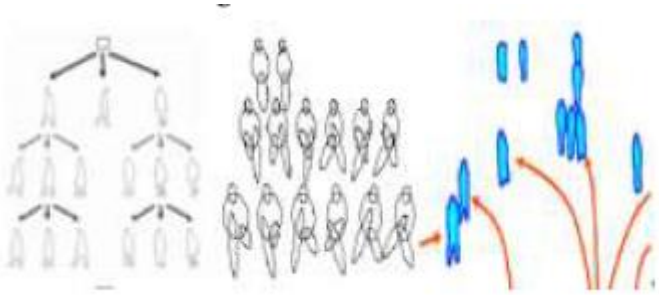


Figure 13: Various shape models of pedestrians (Chen, 2012)

Among them, shape model is the most commonly used one, where the shape of the pedestrians can be used to detect them in Images. Work by P.Meer in 2003 uses histogram of human appearance to detect pedestrians. The disadvantage of such models is that the detection method is sensitive to shape variability and clutters such as background objects and occlusion.

2. *Feature –Classifier based pedestrian detection*

   Most pedestrian detection methods are based on the general process of feature extraction, which can be described as follows:

   First, a detection window (usually sliding window) is extracted from the image. Next, a feature vector of the sub-image is generated inside the detection window. Consequently, the original image is mapped to a specific point in the multidimensional feature space. Finally, a weak classifier is trained using a pool of training samples (Pedestrian or Non-Pedestrian). The whole process, can be mapped into a structure of a convolutional neural network as CNN's have inherent feature selection associated with them.

   Several other feature-based classifiers exist, from Histogram of Oriented Gradient Feature Vectors (HOG) to Principal Component Analysis to perform feature selection. However, this report focuses entirely on CNN's to perform all of the feature selection as proposed by Variyar.

2. Dynamic problems:

Dynamic Pedestrian Detection makes use of the inherent motion associated with video data. It also assumes that the camera and background do not have relative motion to each other.

As previous, a variety of techniques exist for video classification.N.Dalal proposes a Histogram of Oriented Flow (HOF) to detect pedestrian, which compute HOG features in optical flow images.

While, R. Cutler propose a periodicity detection method to detect the motion of human legs analogous of two pendulums out of phase 180° in gravity, using time-frequency analysis.

3. Combination of Multiple Features:

Pedestrian detection problems are inherently complex, and one single model or feature might not prove adequate to solving the problem. Problems such as occlusion (Figure 15), background noise, and variations in pose, clothing, body shape and illumination can make the signal difficult to model.

S.Walk in 2010 proposed a combined feature of HOG, HOF and CSS (Colour Self-Similarity) to outperform the state-of-the art by up to 20% on Caltech and TUD-Brussels database. P.Viola combine Haar wavelet feature and frame difference method to generate a robust pedestrian detector. J.Yao extract covariance feature on foreground image based on probabilistic field.

W.R.Schwartz construct a feature set including edge, colour and texture information to detect pedestrians.

These methods therefore discreetly choose features to be combined into multi-features. They are hence to some extent complimentary and additional features should focus on different part of information, which is not picked up.

4. Part Based Pedestrian Detection:

In crowded scenes with frequent occlusions, detectors based on the human body is unable to find all pedestrians. The problem is solved by segmenting the human body into different parts and detecting each part respectively. This compromises computation run time for model accuracy. An advantage of the method is, the pedestrian detector can effectively do well even when partly occluded, but the resolution should be high enough to detect human body parts.

B. Moving from Classification to Detection

In order to move from classification to detection ,an essential next step in object detection is localisation, i.e. quantifying the coordinates of the object by a rectangle (x, y, width, height) .For the pedestrian dataset, each image has one class (pedestrian or not) and bounding boxes (corresponding to the coordinates of the pedestrian). The problem can be therefore compressed into Classification +Localisation. Localisation in this context can be however treated as a regression problem, where the loss function looks to minimise the distance between the coordinates outputted to the ground truth coordinates available.

1. Evaluating the accuracy of the detection system:

The standard evaluation metrics for the accuracy of the detection can be evaluated using the Intersection over Union (IOU) approach using the ground truth-bounding box provided (Dollar, 2009). For each bounding box, if the bounding box specified by the CNN does no not overlap by more than 50% with the bounding box provided by the ground truth file then it is considered to be a false negative or miss match. Furthermore, if the bounding box specified by the algorithm does not contain a pedestrian then it is considered a False Positive.

2. Region- Based Convolutionl Neural Networks (RCNN) , Fast RCNN etc.

As mentioned above, treating object detection as a two-stage detection process, there exists many variants of a CNN, which can be implemented to perform such tasks.

a) Region- Based Convolutionl Neural Networks (RCNN)

An RCNN can be considered as a CNN with Regional propels added in. Hence, the RCNN takes an image as an input and outputs the bounding boxes and labels for each object in the image.

The mechanism of the bounding box proposals by RCNN is based on the process called “Selective Search”. At a high level, looks at the image through different sizes and for each size tries to group together adjacent pixels by colour, texture, or intensity to identify objects.

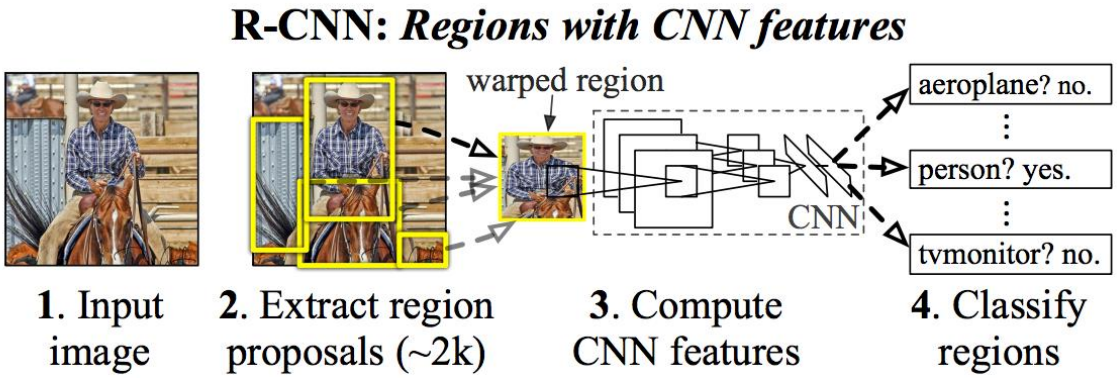


Figure 14; Mechanism of a RCNN (Ross Girshick, 2013)

The proposals are created and then RCNN wraps the region to a standard square size and passes it through a modified version of Alex Net – a pertained convolutional network on ImageNet Large Scale Visual Recognition Challenge in 2012 (Krizhevsky A. I., 2012).

b) Fast Region- Based Convolutionl Neural Networks (Fast RCNN)

RCNN works well but due to long computation times is not preferred. Two major reasons exist:

- It requires a forward pass of the CNN for every single region proposal for every single image
- The general pipeline of a RCNN is slow: uses a CNN to generate features, classifier to predict the class and a final regression model to improve the accuracy of the bounding boxes.

Fast RCNN improves on the RCNN model by two ways:

- **Region of Interest Pooling (ROI Pool)**: The CNN features for each region are obtained by selecting a corresponding region from the CNN’s feature map. The features in each region are then selected using some pooling techniques usually max pooling. This down samples the amount of iterations required to run the original RCNN model.
- **Unified Framework:** Fast RCNN combines all the three separate tasks mentioned above into one framework, which allows for faster computation times.

Faster RCNN’s, Region based Fully Convolutional Networks etc. extensions of the above architecture looks to obtain even faster computation times by speeding up the region proposal algorithms.

VII. Methodology

Detecting pedestrians using CNNs , is a motivation obtained from a similar work carried out by Variyar in 2016.For the purpose of this report, we consider a simple binary classification problem, either a frame containing pedestrian labelled [1, 0] or no pedestrian [0, 1]. The count of persons in the image is therefore ignored and only the existence of a person is considered. A picture containing multiple pedestrians will be therefore classified as a pedestrian in Figure 15. No Region of Interest pooling for proposal generation or bounding box calculation has been done, as localisation is not Considered.

.

Figure 15: Multiple pedestrian's occlusion

The high intra-class variability between 10 different scenarios make this a difficult problem for a neural network to learn (Figure 12). With this in mind, the model is supplied images of 50×50 pixels size with 9 different scenarios .There are 23,498 frames for use as training data (80%) and a further validation set (20%). With the remaining one scene (2750 frames -"sofa") is used as the final test set. However, the frames form part of video, and hence redundancy between frames is an issue, hence multitudes of image augmentation steps are taken to increase the variability in the training data (M.Paulin, 2014). They are created as follows:

- Randomly Rotate Images
- Randomly shift images horizontally by a certain pixel length
- Randomly shift images vertically by a certain pixel length
- Randomly Flip images horizontally or mirror images


Figure 16: Result of applying a horizontal flip to a pedestrian image

A. Pre-Processing

An important step in working with image data is pre-processing of the images into required formats before model fitting (Pal & Sudeep, 2016). This allows for improvement in performance in model accuracy as the data is scaled to a range, rather than arbitrarily large or small values .It also allows for reducing the background noise in the data. Here, the following pre-processing steps are considered:

1. Feature Standardisation:

Feature Standardisation represents a technique to standardize the pixel values across the entire dataset. Hence to standardise, a column of x:

$$\frac{(x-\mu(x))}{std(x)} \qquad (9)$$

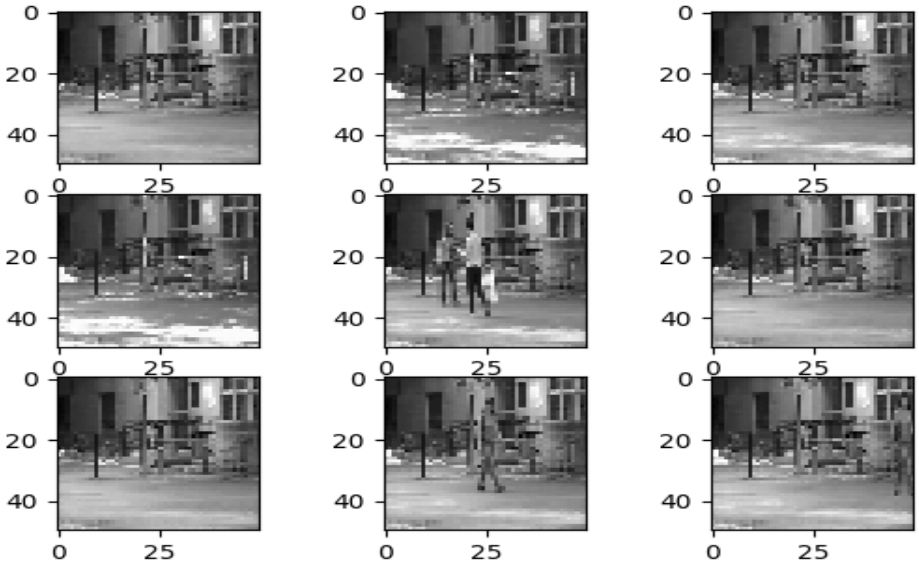
This scales each sample by the specific standard deviation with mean centring.

2. ZCA Whitening:

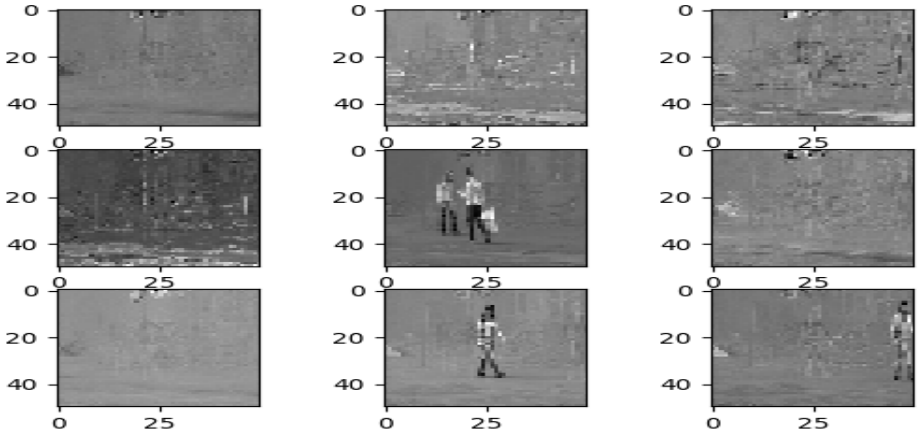
A whitening transform of an image is a linear algebra operation that reduces the redundancy of in the matrix of pixel images. The pedestrian dataset is a sequence of frames formed from videos, and hence highly correlated. The goal of whitening is to make the input data less redundant and hence the model is able to see features that are:

- Less correlated with each other
- Features that all have the same variance

Unlike Principal Component Analysis, images after ZCA still resemble the original image and is a major advantage when using CNN is to model the resultant images (Krizhevsky A. , 2009).As an Example, Figure 17 shows the result of the applying a ZCA transform to a subset of the training set.



a) Training images before zca whitening



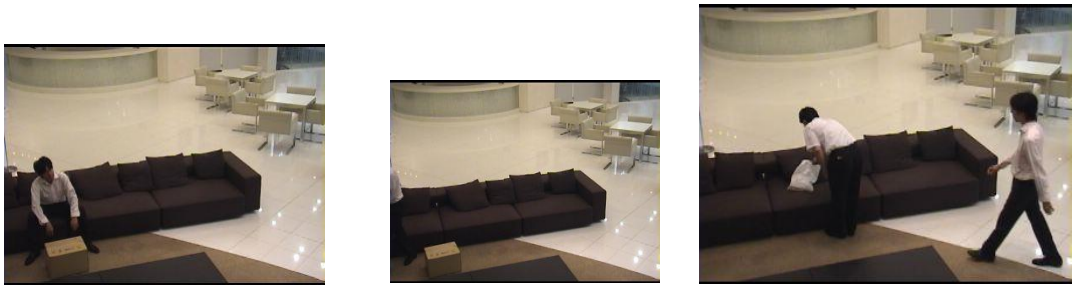
b) Training images after zca whitening

Figure 17 Pedestrian Images before and after applying ZCA whitening

B. Convolutional Neural Network architecture and Implementation

The model was designed initially with a simple 2-layer model .Deeper models were subsequently added, inspired by the architecture used by the Cifar10-dataset.This allowed for easier training for the initial models and testing for convergence. Furthermore , a variety of optimisers were trialled for minimising the loss function (categorical cross entropy) but the Stochastic Gradient Descent with an exponential decay resulted in the best convergence of the training and validation accuracy.

Each of the trialled models were tested on the independent test dataset as discussed before. A sample of images used for testing is shown in Figure 18.



a) Pedestrian *b) No Pedestrian* *c) Pedestrian*

Figure 18: Test images for the Pedestrian Dataset-"Sofa"

For completeness, the final implemented architecture is discussed below, with default hyper-parameters can be found in the script associated with kerasmodel.py

The **final** implemented model architecture can be summarised as follows:

- A Convolutional input layer, with 32 feature maps with a size of 3×3, a RELU function as an activation function and a weight constraint of max norm set to three.
- Followed by a **second** Convolutional layer, with 32 feature maps with a size of 3×3, a rectifier activation function and a weight constraint of max norm set to 3
- Max pool layer with a size 2×2
- Drop out set to 75%
- A **third** Convolutional Layer with 64 Feature maps with a size of 3×3, a rectifier activation function and a weight constraint of max norm set to 3
- A **fourth** Convolutional Layer with 64 Feature maps with a size of 3×3, a rectifier activation function and a weight constraint of max norm set to 3
- A Max pool layer with a filter size of 2×2
- A Flatten layer.
- Fully connected layer with 512 units and a rectifier activation function.
- Dropout set to 50%.
- Fully connected output layer with 10 units and a softmax activation function.

C. Implemented Code Layout

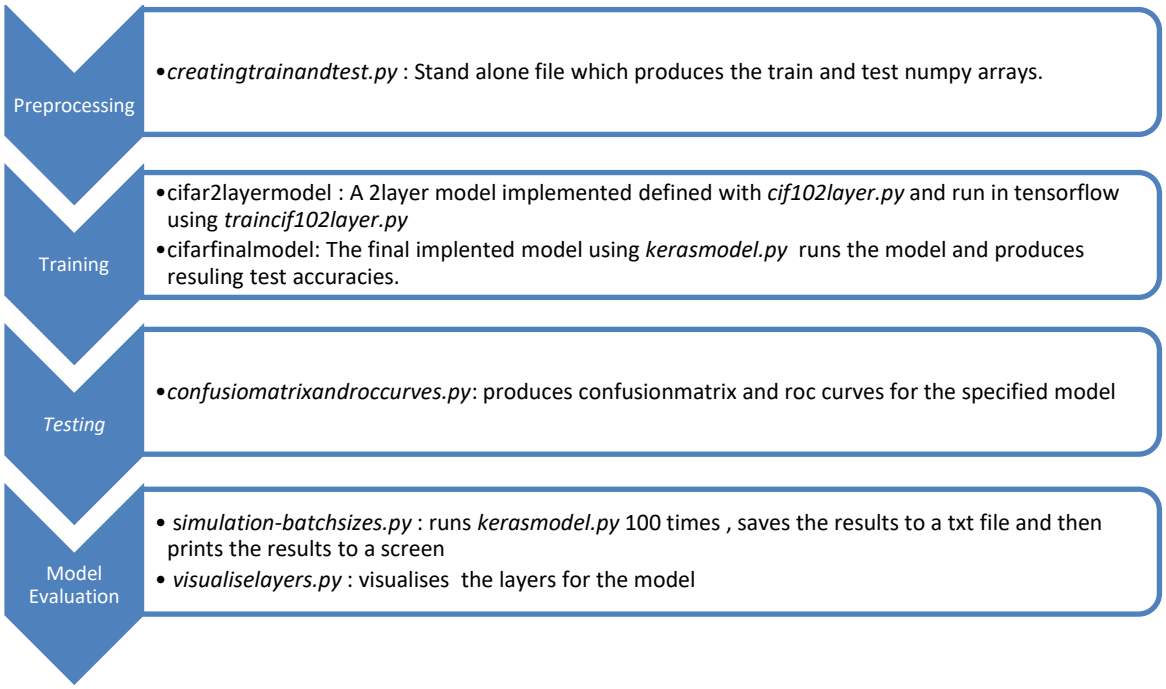


Figure 19 : Code layout for the pedestrian detection dataset

Figure 19, provides a brief overview of the implemented code structure. The initial work of "Pre-processing "consisted of transforming the jpeg files into numpy arrays as well as labelling them with the no of corresponding pedestrians, using OPENCV. The next step, of "training" the model, started with a number of trial and error runs for the 2layer model, which allowed for choosing appropriate starting values for the deeper cifarfinalmodel. Subsequent analysis of the model is done in the model evaluation step.

VIII. Results

As discussed, the cifar model is trained on a GTX NVIDIA 1080 graphics card, with the inclusion of pre-processing steps requiring longer running times (10-20 minutes). A simple 2-layer model initially trained on a CPU achieved promising results with validation accuracies reaching in excess of 65% as in Figure 20 but did not produce comparable test results i.e. was over fitted - hence was discarded and favoured for a deeper cifar10 –model.

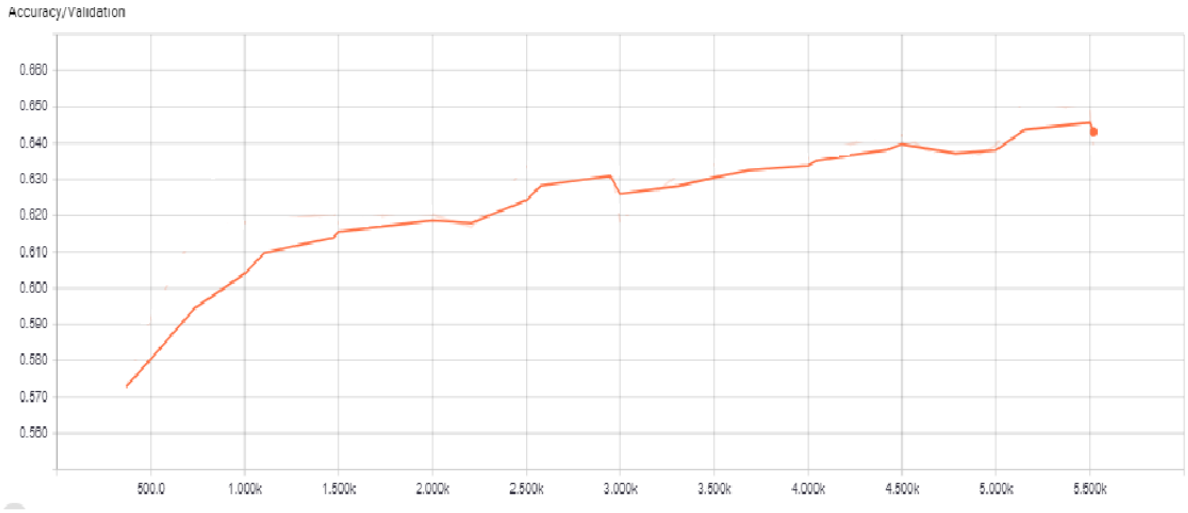


Figure 20 : Validation accuracy for the 2-hidden layer model obtained using tensor board

A sample of the predictions produced is presented in Figure 21. We see the 2- layer model, only correctly labels 5 out of the 16 images correctly, and with a test accuracy of 50.9%, which only performs 0.9% better than a Random Guess Classifier (50%).

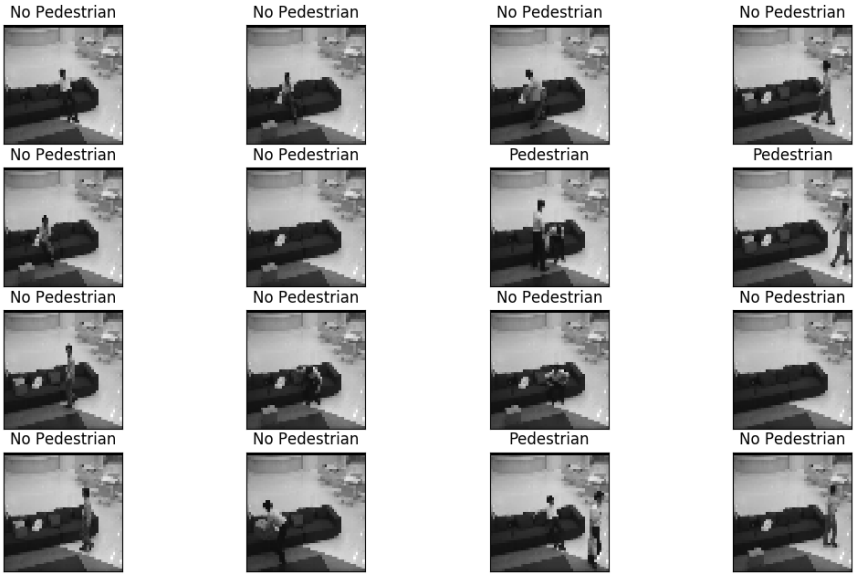


*Figure 21: Model Predictions for simple 2-layer model on 4*4 grid of "sofa" test images*

For Cifar10 model, the Training and validation accuracies reach above 80% when training for larger number of epoch's (variant on no of epochs run). Considering the same sample of test set returns as previous, the deeper model returns 16 out of 16 correctly labelled images as seen in Figure 22.

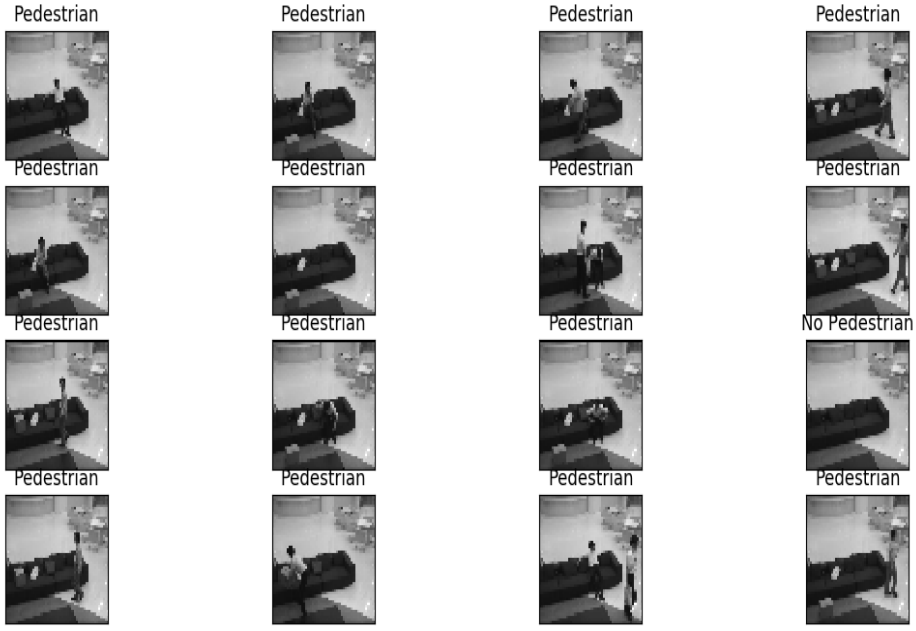


*Figure 22: Model Predictions for cifar-10 model on 4*4 grid of "sofa" test images*

As a binary classification, the model's performance is evaluated on the test data using a confusion matrix presented in Table 2

	Predicted : No Pedestrians	**Predicted : Pedestrians**
Actual : No Pedestrians	130	611
Actual : Pedestrians	147	1862

Table 2: Binary Confusion Matrix using the Cifar10-model

For a binary classification problem, we calculate the following classification performance metrics for the test set.

- $Accuracy = \frac{TP+TN}{Total} = 72.43\%$
- $Weighted\ Precision\ = \frac{(\mathrm{p}_{c_1} \times \mathrm{c}_1)+(\mathrm{p}_{c_2} \times \mathrm{c}_2)}{c_1+c_2} = \ 67.65\%$
- $Weighted\ Recall = \frac{(\mathrm{r}_{c_1} \times \mathrm{c}_1)+(\mathrm{r}_{c_2} \times \mathrm{c}_2)}{c_1+c_2} = = 72.43\%$

Where TP, TN, FP, FN are True Positives, True Negatives, False Positives and False Negatives Respectively.

In the above and for future, the Recall (r) = $\frac{TP}{TP+FN}$and Precision (p) =$\frac{TP}{TP+FP}$ are calculated for each label (c_1, c_2), and with their average, weighted by support (the number of true instances for each label).This allows to account for an imbalanced dataset such as this (Pedregosa, 2017).

Furthermore, we compute the Area under the Curve (AUC) as **0.92** of a Receiver Operating Curve of the model in Figure 23. As AUC is not a function of threshold, it can be used as an evaluation of the classifier as threshold varies over all possible values. Hence can be seen as a broader metric, testing the quality of the internal value that the classifier generates and then compares to a threshold i.e. indiscriminate to the choice of particular threshold values.

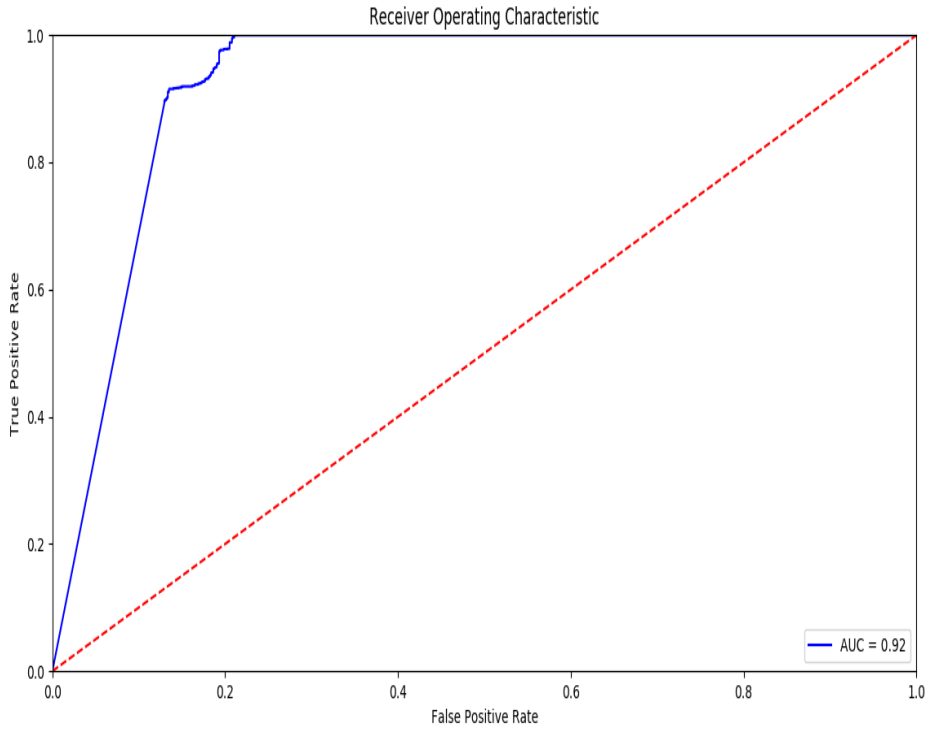


Figure 23: ROC curve for the cifar10 model

IX. Evaluation

Convolutional Neural Networks are considered a black box model, with many hyper-parameters to learn. As a result, the model performs very differently to the batches of training set provided for the model to learn. Considerable research has been done on how CNN's are particularly susceptible to adversarial perturbations to the input images (Christian Szegedy, 2014) ; (Ian J Goodfellow, 2015); (Seyed-Mohsen Moosavi-Dezfooli, 2015).

In order to understand the stability of the model, we run a simulation based on different batch sizes (32,200,300) 100 times and compare it to the precision and recall of the original model with batch size-96. In simpler terms, we look to answer the following:

- How often do we achieve the precision and accuracy achieved by the cifar10 model if we run the model a 100 times
- Is a certain batch size preferred to another in terms of better precision and recall values?

The results of the simulation are plotted as seen in Figure 24 & Figure 25.

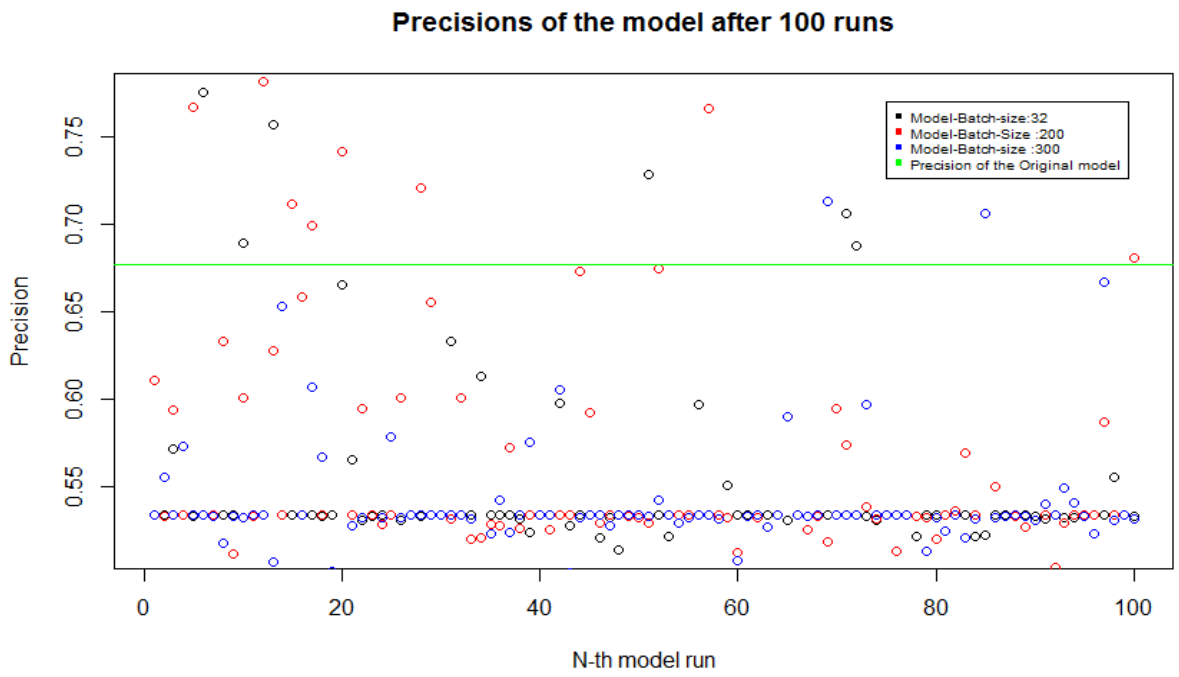


Figure 24 : Precision of the three models with batch sizes (32,200,300) compared to the original model

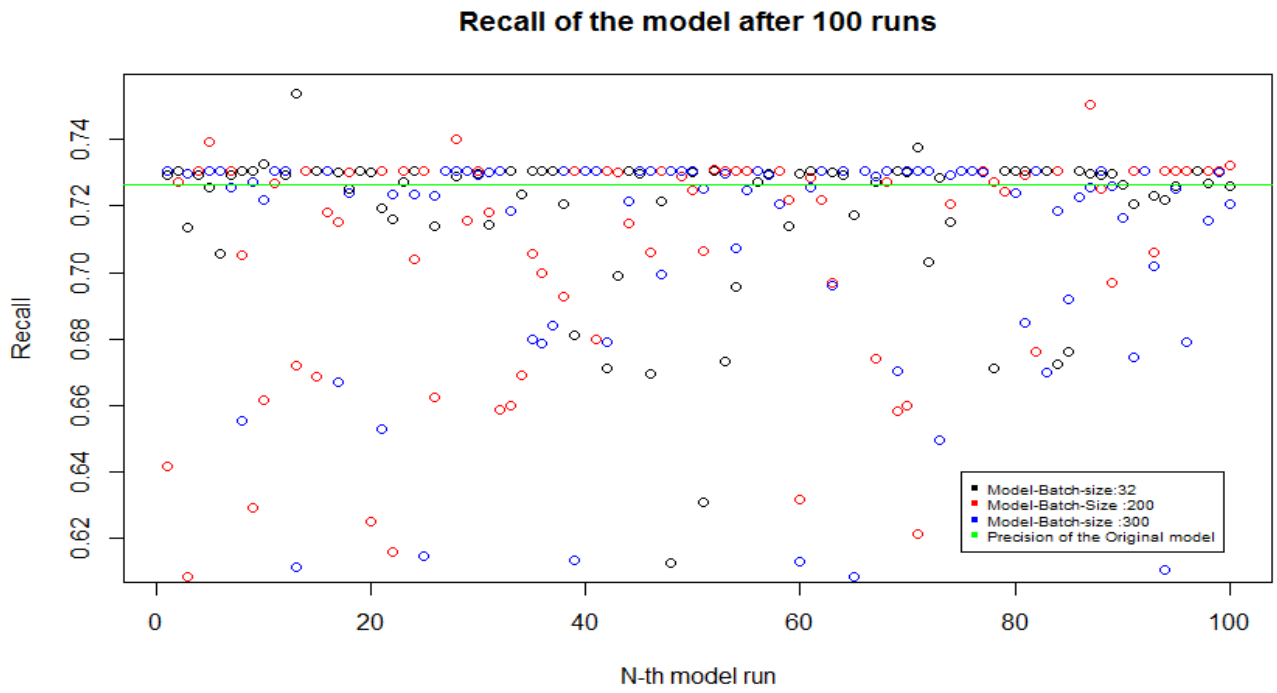


Figure 25: Recall of the three models with batch sizes (32,200,300) compared to the original model

The original model precision or better is only achieved 16 out of 300 model runs, while model has a much better recall, invariant to batch size. The simulation also aims to point out reproducibility of the results achieved by the implemented model. In terms of batch size the model, 32, 200 achieved better precision values compared to a larger batch size as 300, and can be used for future optimisation procedures.

1. Visualising Convolutional Layers:

In an attempt to understand the results achieved, a further step is taken to visualise the third convolutional layer in the cifar10 model and compare it to a very ineffective performing model with the following results.

	Predicted : No Pedestrians	**Predicted : Pedestrians**
Actual : No Pedestrians	0	741
Actual : Pedestrians	0	2009

Table 3: Binary Confusion Matrix using the Ineffective Cifar10-model

The ineffective model has a higher test accuracy at 73.05% as well as AUC score of one, with a Recall at 73.05%. However, the model performs much worse in weighted precision with 53.36% or a precision of zero. Often referred to as the “Accuracy Paradox”, the model beats the cifar10 model by simply predicting one’s for all classes and hence a very useless model for predictive purposes.

A feature map is produced for both the models which is supplied with a pedestrian containing image, which is then visualised for the 3rd convolutional layer as seen in (Figure 26, Figure 27, Figure 28)

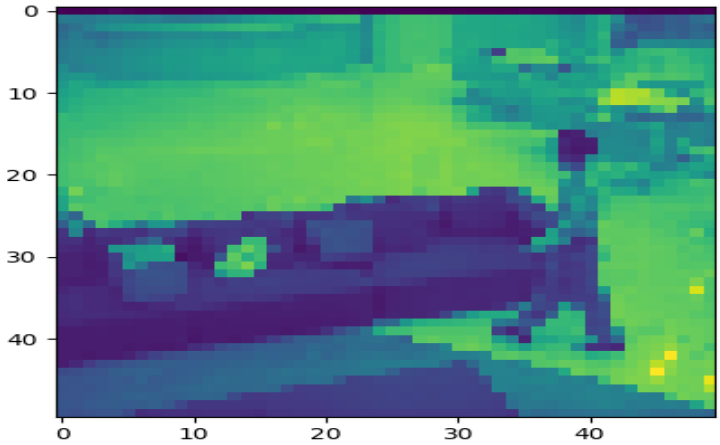


Figure 26 : A Pedestrian image from test dataset (plotted cmap="jet")

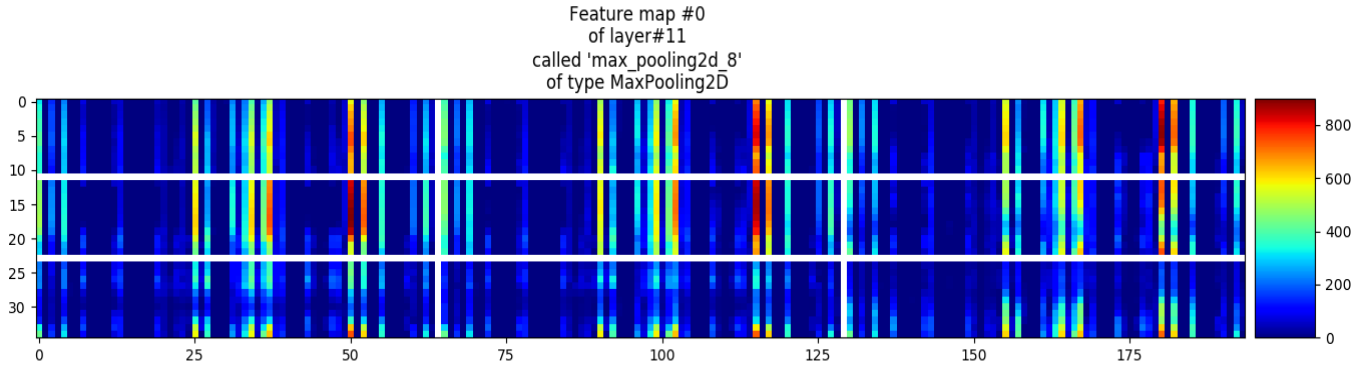


Figure 27: Feature map of third Convolutional Layer after Max-pooling in the cifar10 model plotted for layer#11

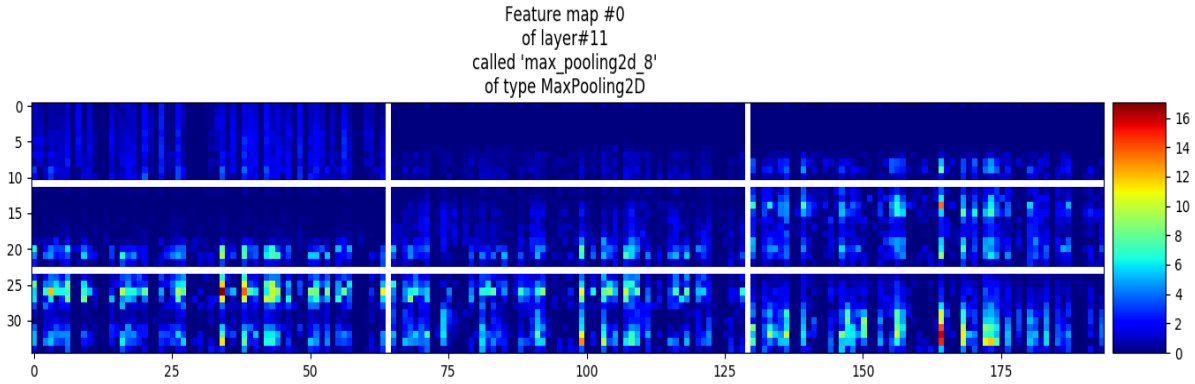


Figure 28: Feature map of third Convolutional Layer after Max-pooling in the "ineffective" cifar10 model

In the above figures (27&28), every box shows an activation map corresponding to some filter. Notice that the activations are sparse (most values are zero, in this visualization shown in dark blue) and mostly local, while the activated regions correspond to red.

We notice stark contrasts between the two feature maps in cifar10 and the ineffective model. There exists stronger regions of activations (high intensity) closer to the presence of the pedestrian in the cifar10 model while, very low activations (mostly blue) are seen in the ineffective model. Furthermore, activations appear random and inconsistent with the ineffective model, while we see consistent strips of activation in the cifar10 model. In addition, we hypothesise; the cifar model performs better as it is able to pick signals, which are generalizable to varied datasets compared to the ineffective model.

X. Conclusion

In this report, we looked at how Convolutional Neural networks can be fitted to a relative simple pedestrian classification problem , to perform a binary classification .The model results obtained are promising despite , the low number of training samples for the model to learn and high variability of training samples between scenes as seen in Figure 12. We also evaluated model performance using classification metrics and tested the stability of the model by performing simulations with varying batch sizes.

A. Discussion

The results show that the CNN model is a black box model, with tiny perturbations in the dataset making the difference between good performing model and worse ones. As suggested before as seen in Figure 12 , the scenes between training samples vary vastly in terms of background settings, pedestrian features (clothing, position, lighting etc.). All this, result in an unstable model, which performs vastly different dependent on the training batches.

It also becomes clear in such cases; training and validation accuracies are not good indicators of the model's performance on unseen test set. Hence, classification metrics such as precision and recall on a test set can serve as better performance indicators for general datasets.

B. Future Work

In respect with the findings, it becomes clear the pedestrian dataset is not particular suited in a static problem setting. However, one can look to improve upon the current classification model in this setting by implementing some of the below:

- **Transfer- Learning**: Training using a Pre-trained convolutional network on ImageNet such as the VGGnet, Resnet models with its associated weights and biases. Minor modifications to the final layers of these neural nets can hence prove useful in classification tasks such as ours.
- **Hyper-parameter tuning**: Another way of improving classifying performance is to tune the hyper-parameters of the model. In order to achieve better performance the architecture of the CNN requires tuning combined with the hyper-parameters such no of epochs, batch size, learning rates etc. As a next step, the Genetic Evolutionary algorithm discussed earlier in Page 16 can be implemented to perform network selection, than fitting a generic CNN architecture.
- **Feature Based Selection:** Using feature based selection methods such as PCA can improve the signal to noise ratio in the model, which is certainly applicable to the cifar10 model described. One such work is carried out in (Ernesto Mastrocinque, 2014) , where the results of a PCA can be used to find the principal variables and is supplied as input into an Artificial Neural Network.

An alternative approach to producing a better performing classification model is it to make use of the connectivity of the video data, and hence to consider it as a "*Dynamic Problem*" as discussed in Pages: 27 .

CNNS as seen previously powerful class of models for image recognition problems. An extension of this can be done on video classification problems. This can be done by extending the connectivity of a CNN in the time domain to take advantage of spatial temporal information.

The Standard approach to video classification involves three main stages:

1) Local Visual features that describe a region of the video are extracted either densely or at a sparse set of interest points. (H. Wang, 2009)

2) Features are combined into fixed sized video level description. A common approach is to quantize all features using learned kmeans dictionary and accumulate the visual words over the duration of videos into histograms of varying spatio-temporal positions and extents. (I. Laptev, 2008)

3) A classifier is trained on the resulting "bag of words" representation to distinguish among the visual classes of interest.

However, an *improved* method relies on treating space and time as equivalent dimensions of the input and perform convolutions in both time and space. One such method involves fusing time information in CNN's (Karpathy, et al., 2014) . The mechanisms are discussed as follows:

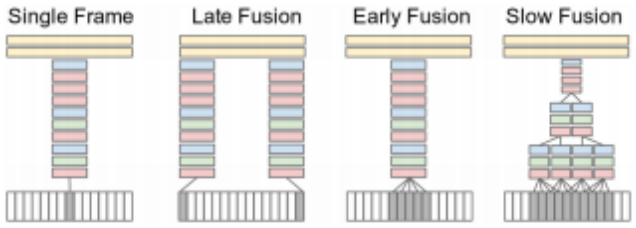


Figure 29: Various ways of Fusing frames (Karpathy, et al., 2014)

- Single Frame: A single frame architecture similar to the winning ImageNet model.
- Late Fusion: Places two separate single-frame networks with shared parameters at a distance of 15 frames apart and then merges the two streams in a fully connected layer.
- Early Fusion: Combines information across an entire time window by extending the filter by one dimension and condensing the temporal information in one-step.
- Slow Fusion: Balanced mix of the two former networks that slowly fuses temporal information throughout the network. Implemented via 3D convolutions with some temporal extent and stride.

However, using temporal information to classification problems comes at a cost in terms of computation time even with the use of modern GPU's. (Karpathy, et al., 2014)

Finally, the pedestrian detection dataset can be used outside the scope of CNN's. For example, Deep forest networks or deep belief networks can be trained using our data.

XI. Appendix

A. MNIST DATASET CODE

1. Model1.py

```
''' Creata a single layer NN using tensorflow , report its accuracy for a
1K iterations -- labelled as Model 1 in Report ----------------''

100 images at a time - batch - 10 digits : MNIST dataset '''


'Output : Accuracy  after 10K iteration:test or train  accuracy  :,60K
images , batch size of 100, 600 iterations to com' \
'plete one epoch ,  '

import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data as mnist_data

tf.set_random_seed(0)

# neural network with 1 layer of 10 softmax neurons

# · · · · · · · · · · ·       (input data, flattened pixels)       X [batch,
784]        # 784 = 28 * 28
# \x/x\x/x\x/x\x/x\x/x\x/    -- fully connected layer (softmax)      W [784,
10]     b[10]
#   · · · · · · · ·                                                 Y [batch,
10]

# The model is:
#
# Y = softmax( X * W + b)
#              X: matrix for 100 grayscale images of 28x28 pixels,
flattened (there are 100 images in a mini-batch)
#              W: weight matrix with 784 lines and 10 columns
#              b: bias vector with 10 dimensions
#              +: add with broadcasting: adds the vector to each line of
the matrix (numpy)
#              softmax(matrix) applies softmax on each line
#              softmax(line) applies an exp to each value then divides by
the norm of the resulting line
#              Y: output matrix with 100 lines and 10 columns

# Test :10K images +labels , train :60 K images + labels

# Download images and labels into mnist.test (10K images+labels) and
mnist.train (60K images+labels)
mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False,
validation_size=0)


## Place holders for the X and the Y_ labels:

# input X: 28x28 grayscale images, the first dimension (None) will index
the images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# correct answers will go here
Y_ = tf.placeholder(tf.float32, [None, 10])
```

```
# weights W[784, 10]   784=28*28
W = tf.Variable(tf.zeros([784, 10]))  ## Variables are all the parameters
that your training al
# biases b[10]
b = tf.Variable(tf.zeros([10]))

# flatten the images into a single line of pixels
# -1 in the shape definition means "the only possible dimension that will
preserve the number of elements"
XX = tf.reshape(X, [-1, 784])

# The m00000odel
Y = tf.nn.softmax(tf.matmul(XX, W) + b)

# loss function: cross-entropy = - sum( Y_i * log(Yi) )
#                           Y: the computed output vector
#                           Y_: the desired output vector

# cross-entropy
# log takes the log of each element, * multiplies the tensors element by
element
# reduce_mean will add all the components in the tensor
# so here we end up with the total cross-entropy for all images in the
batch
cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0  # normalized for
batches of 100 images,
                                                          # *10 because
"mean" included an unwanted division by 10

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# training, learning rate = 0.005
train_step =
tf.train.GradientDescentOptimizer(0.005).minimize(cross_entropy)
# GradienctDescent optimiser : optimising the weights and biases , where it
is know to minimise the loss function :
#cross entropy .


## Define  the session , so the nodes can be executed :
init = tf.global_variables_initializer()
sess=tf.Session()
sess.run(init)



'''Explanation of the below loop :'''
## Feed dictionary , feed on all the missing data that the placeholders :
get images 100 images at a time
# Training loop :pass 100 images at a time , train it ;
# COmpute the gradient on the current 100 images and labels , add alittle
fraction of this to our cweights and biases :train step
# Start over with a new batch of images and labels :(60K) :100K if 1000
iterations :
# Learning rate : little fraction of gradient that you will be adding to
the weights and baises: how to choose the right gradient ?
```

```
#Tensor flow has a deffered excecution model :the tf. statements only
produces a computation graph
# Tf done primaarly done for distributed computing ,helps a lot with
distributing the graphs to multiple systems :
# sess.run excutes the nodes : i.e need to run this everytime you need to
execute something + a feed dict _

for i in range(10000):
    #load batch of images and labells
    batch_X ,batch_Y=mnist.train.next_batch(100)
    train_data={X:batch_X,Y_:batch_Y}
    # train
    # Whats is the train step : It computes the gradient , dereives the
deltas for the weights and biases  and updates the
    # weights and biases .: It makes the weights and biases move in the
right direction
    sess.run(train_step,feed_dict=train_data)


    if (i%100==0):
        #check for every 100 iterations what the sucess  is on training and
test it :
        #sucess?

train_a,train_c=sess.run([accuracy,cross_entropy],feed_dict=train_data)
        print(str(i) + ": Resubstitution  accuracy:" + str(train_a) + "
loss: " + str(train_c))

        # sucess on test data?
        test_data={X:mnist.test.images,Y_:mnist.test.labels}

test_a,test_c=sess.run([accuracy,cross_entropy],feed_dict=test_data)
        print(str(i) + ": accuracy:" + str(test_a) + " loss: " +
str(test_c))


print("train accuracy at " + str(i)+" th iteration :
"+str(train_a*100)+"%")

print("accuracy using test datset at " + str(i)+" th iteration :
"+str(test_a*100)+"%")
```

2. Model2.py

```
''' Creata a Five layer NN using tensorflow , report its accuracy for a 2K
iterations -using Relu function '''


'100 images at a time - batch - 10 digits : MNIST dataset '''


'Output : Accuracy  after 200 0 iterations: '
"Accuracy for training data set : 95.22%"
"accuracy for test dataset : 92.0%"



import tensorflow as tf
```

```
from tensorflow.examples.tutorials.mnist import input_data as mnist_data

tf.set_random_seed(0)

# neural network with 5 layers
#
# · · · · · · · · · · ·          (input data, flattened pixels)       X
[batch, 784]   # 784 = 28*28
# \x/x\x/x\x/x\x/x\x/x\x/       -- fully connected layer (sigmoid)      W1
[784, 200]      B1[200]
#  · · · · · · · · · ·                                                 Y1
[batch, 200]
#   \x/x\x/x\x/x\x/          -- fully connected layer (sigmoid)      W2
[200, 100]      B2[100]
#    · · · · · · ·                                                     Y2
[batch, 100]
#     \x/x\x/x\x/           -- fully connected layer (sigmoid)      W3
[100, 60]       B3[60]
#      · · · · ·                                                       Y3
[batch, 60]
#       \x/x\x/              -- fully connected layer (sigmoid)      W4 [60,
30]        B4[30]
#        · · ·                                                        Y4
[batch, 30]
#         \x/                 -- fully connected layer (softmax)      W5 [30,
10]        B5[10]
#
# The model is:
#
# Y = softmax( X * W + b)
#              W1:

# Test :10K images +labels , train :60 K images + labels

# Download images and labels into mnist.test (10K images+labels) and
mnist.train (60K images+labels)
mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False,
validation_size=0)


## Place holders for the X and the Y_ labels:

# input X: 28x28 grayscale images, the first dimension (None) will index
the images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# correct answers will go here
Y_ = tf.placeholder(tf.float32, [None, 10])

## Initialise the Neuorons
L=200
M=100
N=60
O=30
P=10

# 5 weights to be initiallised using truncated normal  :
W1=tf.Variable(tf.truncated_normal([784, L], stddev=0.1))
W2=tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
W3=tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
W4=tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
W5=tf.Variable(tf.truncated_normal([O, P], stddev=0.1))
```

```
#Initialise the Biases
b1 = tf.Variable(tf.zeros([L]))
b2 = tf.Variable(tf.zeros([M]))
b3 = tf.Variable(tf.zeros([N]))
b4 = tf.Variable(tf.zeros([O]))
b5 = tf.Variable(tf.zeros([P]))

# flatten the images into a single line of pixels
# -1 in the shape definition means "the only possible dimension that will
preserve the number of elements"
XX = tf.reshape(X, [-1, 784])


## Build the model :
Y1=tf.nn.sigmoid(tf.matmul(XX,W1)+b1)
Y2=tf.nn.sigmoid(tf.matmul(Y1,W2)+b2)
Y3=tf.nn.sigmoid(tf.matmul(Y2,W3)+b3)
Y4=tf.nn.sigmoid(tf.matmul(Y3,W4)+b4)
Y=tf.nn.softmax(tf.matmul(Y4,W5)+b5)




# loss function: cross-entropy = - sum( Y_i * log(Yi) )
#                           Y: the computed output vector
#                           Y_: the desired output vector

# cross-entropy
# log takes the log of each element, * multiplies the tensors element by
element
# reduce_mean will add all the components in the tensor
# so here we end up with the total cross-entropy for all images in the
batch
cross_entropy = -tf.reduce_mean(Y_ * tf.log(Y)) * 1000.0  # normalized for
batches of 100 images,
                                                          # *10 because
"mean" included an unwanted division by 10

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# training, learning rate = 0.005
train_step =
tf.train.GradientDescentOptimizer(0.005).minimize(cross_entropy)
## Define  the session , so the nodes can be executed :
init = tf.global_variables_initializer()
sess=tf.Session()
sess.run(init)



'''Explanation of the below loop :'''
## Feed dictionary , feed on all the missing data that the placeholders :
get images 100 images at a time
# Training loop :pass 100 images at a time , train it ;
```

```
# COmpute the gradient on the current 100 images and labels , add alittle
fraction of this to our cweights and biases :train step
# Start over with a new batch of images and labels :(60K) :100K if 1000
iterations :
# Learning rate : little fraction of gradient that you will be adding to
the weights and baises: how to choose the right gradient ?


#Tensor flow has a deffered excecution model :the tf. statements only
produces a computation graph
# Tf done primaarly done for distributed computing ,helps a lot with
distributing the graphs to multiple systems :
# sess.run excutes the nodes : i.e need to run this everytime you need to
execute something + a feed dict _

for i in range(10000):
    #load batch of images and labells
    batch_X ,batch_Y=mnist.train.next_batch(100)
    train_data={X:batch_X,Y_:batch_Y}
    # train
    # Whats is the train step : It computes the gradient , dereives the
deltas for the weights and biases  and updates the
    # weights and biases .: It makes the weights and biases move in the
right direction
    sess.run(train_step,feed_dict=train_data)


    if (i%100==0):
        #check for every 100 iterations what the sucess  is on training and
test it :
        #sucess?

train_a,train_c=sess.run([accuracy,cross_entropy],feed_dict=train_data)
       # print(str(i) + ": Resubstitution  accuracy:" + str(train_a) + "
loss: " + str(train_c))

        # sucess on test data?
        test_data={X:mnist.test.images,Y_:mnist.test.labels}

test_a,test_c=sess.run([accuracy,cross_entropy],feed_dict=test_data)
        #print(str(i) + ": accuracy:" + str(test_a) + " loss: " +
str(test_c))



print("train accuracy at " + str(i)+" th iteration :
"+str(train_a*100)+"%")

print("accuracy using test datset at " + str(i)+" th iteration :
"+str(test_a*100)+"%")
```

3. Model3.py

```
'''Five layer NN- using the Relu function-- Model 2 '''


'''The output after 2K iterations :
Trainign accuracy :100%
Test accuracy :97.61% '''


''' Change also the learning rate decay to  0.0001'''
```

```
#

'100 images at a time - batch - 10 digits : MNIST dataset '''


'Output : Accuracy  after 2000 iterations: '
"Accuracy for training data set : 100% !!!"
"accuracy for test dataset : 97.32%"

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data as mnist_data

tf.set_random_seed(0)

# neural network with 5 layers
#
# · · · · · · · · · · ·          (input data, flattened pixels)       X
[batch, 784]   # 784 = 28*28
# \x/x\x/x\x/x\x/x\x/x\x/       -- fully connected layer (sigmoid)      W1
[784, 200]      B1[200]
#  · · · · · · · · · ·                                                 Y1
[batch, 200]
#   \x/x\x/x\x/x\x/x\x/         -- fully connected layer (sigmoid)      W2
[200, 100]      B2[100]
#    · · · · · · · ·                                                   Y2
[batch, 100]
#     \x/x\x/x\x/x\x/           -- fully connected layer (sigmoid)      W3
[100, 60]       B3[60]
#      · · · · · ·                                                     Y3
[batch, 60]
#       \x/x\x/x\x/             -- fully connected layer (sigmoid)      W4 [60,
30]        B4[30]
#        · · · ·                                                       Y4
[batch, 30]
#         \x/x\x/               -- fully connected layer (softmax)      W5 [30,
10]        B5[10]
#
# The model is:
#
# Y = softmax( X * W + b)
#              W1:

# Test :10K images +labels , train :60 K images + labels

# Download images and labels into mnist.test (10K images+labels) and
mnist.train (60K images+labels)
mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False,
validation_size=0)


##Initialise some variables to store for plotting
trainaccuracy=np.array([])
testaccuracy=np.array([])

cross_entropy_train=np.array([])
cross_entroppy_test=np.array([])


## Place holders for the X and the Y_ labels:
```

```
# input X: 28x28 grayscale images, the first dimension (None) will index
the images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# correct answers will go here
Y_ = tf.placeholder(tf.float32, [None, 10])

# Placeholder for the adamoptimizer (define the placeholder and use it for
adamoptimizer and then later populate with
# the results from the run session
lr=tf.placeholder(tf.float32)



## Initialise the Neuorons
L=200
M=100
N=60
O=30
P=10

# 5 weights to be initiallised using truncated normal  :
W1=tf.Variable(tf.truncated_normal([784, L], stddev=0.1))
W2=tf.Variable(tf.truncated_normal([L, M], stddev=0.1))
W3=tf.Variable(tf.truncated_normal([M, N], stddev=0.1))
W4=tf.Variable(tf.truncated_normal([N, O], stddev=0.1))
W5=tf.Variable(tf.truncated_normal([O, P], stddev=0.1))


#Initialise the Biases
b1 = tf.Variable(tf.ones([L])/10)
b2 = tf.Variable(tf.ones([M])/10)
b3 = tf.Variable(tf.ones([N])/10)
b4 = tf.Variable(tf.ones([O])/10)
b5 = tf.Variable(tf.ones([P])/10)

# flatten the images into a single line of pixels
# -1 in the shape definition means "the only possible dimension that will
preserve the number of elements"
XX = tf.reshape(X, [-1, 784])


## Build the model :
Y1=tf.nn.relu(tf.matmul(XX,W1)+b1)
Y2=tf.nn.relu(tf.matmul(Y1,W2)+b2)
Y3=tf.nn.relu(tf.matmul(Y2,W3)+b3)
Y4=tf.nn.relu(tf.matmul(Y3,W4)+b4)

## Replace Y4 with a logits function :
Ylogits=tf.matmul(Y4,W5)+b5
Y=tf.nn.softmax(Ylogits)




# loss function: cross-entropy = - sum( Y_i * log(Yi) )
#                           Y: the computed output vector
#                           Y_: the desired output vector
```

```
# cross-entropy
# log takes the log of each element, * multiplies the tensors element by
element
# reduce_mean will add all the components in the tensor
# so here we end up with the total cross-entropy for all images in the
batch

cross_entropy_logit=tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits,
labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy_logit) * 100.0  # normalized
for batches of 100 images,
                                                             # *10 because
"mean" included an unwanted division by 10

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# training, learning rate = 0.005

# Adam optimizer : Saddle points frequent in 10K weights and biases .
#Points are not local minima , but where the gradient is neverthless and
get stuck there .
# Adam optimizer gets around that problem :
train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy) ## Lr added

## Define  the session , so the nodes can be executed :
init = tf.global_variables_initializer()
sess=tf.Session()
sess.run(init)



'''Explanation of the below loop :'''
## Feed dictionary , feed on all the missing data that the placeholders :
get images 100 images at a time
# Training loop :pass 100 images at a time , train it ;
# COmpute the gradient on the current 100 images and labels , add alittle
fraction of this to our cweights and biases :train step
# Start over with a new batch of images and labels :(60K) :100K if 1000
iterations :
# Learning rate : little fraction of gradient that you will be adding to
the weights and baises: how to choose the right gradient ?


#Tensor flow has a deffered excecution model :the tf. statements only
produces a computation graph
# Tf done primaarly done for distributed computing ,helps a lot with
distributing the graphs to multiple systems :
# sess.run excutes the nodes : i.e need to run this everytime you need to
execute something + a feed dict _


''' Learning rate decay added to the relu code :'''
#Define a min learning and max min rate :
lrmin=0.0001
lrmax=0.003
```

```
for i in range(10000+1):
    '''Optimizing using  decying learning rate:'''
    # feed the placeholder with the lrdecay with each iteration:
    learningrate = lrmin + (lrmax - lrmin) * np.exp(-i / 2000)



    #load batch of images and labells
    batch_X ,batch_Y=mnist.train.next_batch(100)

    # Add in the learning rate :
    train_data={X:batch_X,Y_:batch_Y,lr:learningrate}

    # Whats is the train step : It computes the gradient , dereives the
deltas for the weights and biases  and updates the
    # weights and biases .: It makes the weights and biases move in the
right direction
    sess.run(train_step,feed_dict=train_data)



    if (i%100==0):
        #check for every 100 iterations what the sucess  is on training and
test it :
        #sucess?

train_a,train_c=sess.run([accuracy,cross_entropy],feed_dict={X:batch_X,Y_:b
atch_Y})
        #print(str(i) + ": Resubstitution  accuracy:" + str(train_a) + "
loss: " + str(train_c))

        # sucess on test data?
        test_data={X:mnist.test.images,Y_:mnist.test.labels}

test_a,test_c=sess.run([accuracy,cross_entropy],feed_dict=test_data)
        #print(str(i) + ": accuracy:" + str(test_a) + " loss: " +
str(test_c))

        #populate the train and test accuract for visualising the
plotting :
        trainaccuracy=np.append(trainaccuracy,train_a)
        testaccuracy=np.append(testaccuracy,test_a)
        #populate the cross entropy for train and test data :

        cross_entropy_train=np.append(cross_entropy_train,train_c)
        cross_entroppy_test=np.append(cross_entroppy_test,test_c)


print("train accuracy at " + str(i)+" th iteration :
"+str(train_a*100)+"%")

print("accurac  y using test datset at " + str(i)+" th iteration :
"+str(test_a*100)+"%")



## Function for plotting train and test  accuracy for 2K iterations :
# use matplot from the pyplot library to plot the  functions :
import matplotlib.pyplot as plt
```

```
#

#plt.subplot(211)
#create legends.
x1=list(range(len(trainaccuracy)))
y1=trainaccuracy
x2=list(range(len(testaccuracy)))
y2=testaccuracy



plt.plot(x1,y1,c='r',label='training accuracy')
plt.plot(x2,y2,c='g',label='test accuracy')
plt.legend()
plt.show()


## What happens to the cross entropy loss :
x3=list(range(len(cross_entropy_train)))
y3=cross_entropy_train
x4=list(range(len(cross_entroppy_test)))
y4=cross_entroppy_test



plt.plot(x3,y3,c='r',label='training -cross entropy loss')
plt.plot(x4,y4,c='g',label='test -cross entropy loss')
plt.legend()
plt.show()
```

4. Model4.py

```
''' Implement the Covnet model for the MNIST dataset- Model 4 convolutional
neural network  '''



## THe Relu function : simpler and performs better than the sigmoid
function for deep NN
# Works similar to biological neurons : 0 when no signal ..
## Difference between sgmoid functions : change the biases to  small
positive values :




'100 images at a time - batch - 10 digits : MNIST dataset '''


'Output : Accuracy  after 200 0 iterations: '
"Accuracy for training data set : 100% !!!"
"accuracy for test dataset : 97.32%"

import matplotlib.pyplot as plt
import numpy as np
import tensorflow as tf
from tensorflow.examples.tutorials.mnist import input_data as mnist_data

tf.set_random_seed(0)
```

```
#
# Download images and labels into mnist.test (10K images+labels) and
mnist.train (60K images+labels)
mnist = mnist_data.read_data_sets("data", one_hot=True, reshape=False,
validation_size=0)


##Initialise some variables to store for plotting
trainaccuracy=np.array([])
testaccuracy=np.array([])

cross_entropy_train=np.array([])
cross_entroppy_test=np.array([])


## Place holders for the X and the Y_ labels:

# input X: 28x28 grayscale images, the first dimension (None) will index
the images in the mini-batch
X = tf.placeholder(tf.float32, [None, 28, 28, 1])
# correct answers will go here
Y_ = tf.placeholder(tf.float32, [None, 10])

# Placeholder for the adamoptimizer (define the placeholder and use it for
adamoptimizer and then later populate with
# the results from the run session
lr=tf.placeholder(tf.float32)



## Initialise the output  channels or feature maps  (no more neurons as the
same as pixels)

L=4          # first convolutional layer
M=8          # 2nd convolutional layer
N=12         # 3rd convolutional layer
O=200        # Fully connected layers
P=10         # Final Output layer

# Input :
# An image of 28*28*1 instead of  a single vector of 784*1

# 5 weights to be initiallised using truncated normal  :
#shape[5,5,1,4] : corresponds to : 5,5 patch size for the scanning , 1
input channel , 4 output channel
W1=tf.Variable(tf.truncated_normal([5,5,1,L],stddev=0.1)) # 28*28*4 1st CN
# Filter size (

W2=tf.Variable(tf.truncated_normal([4,4,L,M],stddev=0.1)) #14*14*8 as 8
output  , notice stride =2 as it halves


W3=tf.Variable(tf.truncated_normal([4,4,M,N],stddev=0.1)) #7*7*12 stride:2

W4=tf.Variable(tf.truncated_normal([7*7*12,O],stddev=0.1))# Fully connected
layer: Input 7*7*12 ,output:200 Neurons

W5=tf.Variable(tf.truncated_normal([O,P],stddev=0.1)) # Input :200 Neuron ,
Output :10 classes
```

```
#Initialise the Biases
b1 = tf.Variable(tf.ones([L])/10)
b2 = tf.Variable(tf.ones([M])/10)
b3 = tf.Variable(tf.ones([N])/10)
b4 = tf.Variable(tf.ones([O])/10)
b5 = tf.Variable(tf.ones([P])/10)

# flatten the images into a single line of pixels
# -1 in the shape definition means "the only possible dimension that will
preserve the number of elements"
#XX = tf.reshape(X, [-1, 784]) -- No more flattening of the input image
required


## Build the model :

# stride notation [1,H,W,1] : H is the height ,W is the width as the neuron
is scanning in both directions
# First layer :
stride=1
Y1cn=tf.nn.conv2d(X,W1,strides=[1,stride,stride,1],padding='SAME') #
padding :output map size as the same as input map size
# stride defines the amount of overlapness between each convoluion window :
A stride of 5 , for a patch size of 5 weill#
# mean no overlapping convolution windows when scanning .

Y1=tf.nn.relu(Y1cn+b1)


#2nd Layer :
stride=2
Y2cn=tf.nn.conv2d(Y1,W2,strides=[1,stride,stride,1],padding='SAME') #
padding :output map size as the same as input map size
# stride defines the amount of overlapness between each convoluion window :
A stride of 5 , for a patch size of 5 weill#
# mean no overlapping convolution windows when scanning .

Y2=tf.nn.relu(Y2cn+b2)

#3rd Layer :
stride=2
Y3cn=tf.nn.conv2d(Y2,W3,strides=[1,stride,stride,1],padding='SAME') #
padding :output map size as the same as input map size
# stride defines the amount of overlapness between each convoluion window :
A stride of 5 , for a patch size of 5 weill#
# mean no overlapping convolution windows when scanning .

Y3=tf.nn.relu(Y3cn+b3)
#4th Layer(full CN layer) :
# Use Reshape to convert from a convolution layer to fully connected
layer :
YY=tf.reshape(Y3,shape=[-1,7*7*N])


Y4=tf.nn.relu(tf.matmul(YY,W4)+b4)
# stride defines the amount of overlapness between each convoluion window :
A stride of 5 , for a patch size of 5 weill#
# mean no overlapping convolution windows when scanning .
```

```
# Final Out put function from 200 Neurons :
## Replace Y4 with a logits function :
Ylogits=tf.matmul(Y4,W5)+b5
Y=tf.nn.softmax(Ylogits)




# loss function: cross-entropy = - sum( Y_i * log(Yi) )
#                           Y: the computed output vector
#                           Y_: the desired output vector

# cross-entropy
# log takes the log of each element, * multiplies the tensors element by
element
# reduce_mean will add all the components in the tensor
# so here we end up with the total cross-entropy for all images in the
batch

cross_entropy_logit=tf.nn.softmax_cross_entropy_with_logits(logits=Ylogits,
labels=Y_)
cross_entropy = tf.reduce_mean(cross_entropy_logit) * 100.0  # normalized
for batches of 100 images,
                                                             # *10 because
"mean" included an unwanted division by 10

# accuracy of the trained model, between 0 (worst) and 1 (best)
correct_prediction = tf.equal(tf.argmax(Y, 1), tf.argmax(Y_, 1))
accuracy = tf.reduce_mean(tf.cast(correct_prediction, tf.float32))

# training, learning rate = 0.005

# Adam optimizer : Saddle points frequent in 10K weights and biases .
#Points are not local minima , but where the gradient is neverthless and
get stuck there .
# Adam optimizer gets around that problem :
train_step = tf.train.AdamOptimizer(lr).minimize(cross_entropy) ## Lr added

## Define  the session , so the nodes can be executed :
init = tf.global_variables_initializer()
sess=tf.Session()
sess.run(init)



'''Explanation of the below loop :'''
## Feed dictionary , feed on all the missing data that the placeholders :
get images 100 images at a time
# Training loop :pass 100 images at a time , train it ;
# COmpute the gradient on the current 100 images and labels , add alittle
fraction of this to our cweights and biases :train step
# Start over with a new batch of images and labels :(60K) :100K if 1000
iterations :
# Learning rate : little fraction of gradient that you will be adding to
the weights and baises: how to choose the right gradient ?


#Tensor flow has a deffered excecution model :the tf. statements only
produces a computation graph
```

```
# Tf done primaarly done for distributed computing ,helps a lot with
distributing the graphs to multiple systems :
# sess.run excutes the nodes : i.e need to run this everytime you need to
execute something + a feed dict _


''' Learning rate decay added to the relu code :'''
#Define a min learning and max min rate :
lrmin=0.0001
lrmax=0.003



for i in range(10000+1):
    '''Optimizing using  decying learning rate:'''
    # feed the placeholder with the lrdecay with each iteration:
    learningrate = lrmin + (lrmax - lrmin) * np.exp(-i / 2000)




    #load batch of images and labells
    batch_X ,batch_Y=mnist.train.next_batch(100)

    # Add in the learning rate :
    train_data={X:batch_X,Y_:batch_Y,lr:learningrate}

    # Whats is the train step : It computes the gradient , dereives the
deltas for the weights and biases  and updates the
    # weights and biases .: It makes the weights and biases move in the
right direction
    sess.run(train_step,feed_dict=train_data)



    if (i%100==0):
        #check for every 100 iterations what the sucess  is on training and
test it :
        #sucess?

train_a,train_c=sess.run([accuracy,cross_entropy],feed_dict={X:batch_X,Y_:b
atch_Y})
        #print(str(i) + ": Resubstitution  accuracy:" + str(train_a) + "
loss: " + str(train_c))

        # sucess on test data?
        test_data={X:mnist.test.images,Y_:mnist.test.labels}

test_a,test_c=sess.run([accuracy,cross_entropy],feed_dict=test_data)
        #print(str(i) + ": accuracy:" + str(test_a) + " loss: " +
str(test_c))

        #populate the train and test accuract for visualising the
plotting :
        trainaccuracy=np.append(trainaccuracy,train_a)
        testaccuracy=np.append(testaccuracy,test_a)
        #populate the cross entropy for train and test data :

        cross_entropy_train=np.append(cross_entropy_train,train_c)
        cross_entroppy_test=np.append(cross_entroppy_test,test_c)
```

```
print("train accuracy at " + str(i)+" th iteration :
"+str(train_a*100)+"%")
print("accuracy using test datset at " + str(i)+" th iteration :
"+str(test_a*100)+"%")



## Function for plotting train and test  accuracy for 2K iterations :
# use matplot from the pyplot library to plot the  functions :
import matplotlib.pyplot as plt

#

#plt.subplot(211)
#create legends.
x1=list(range(len(trainaccuracy)))
y1=trainaccuracy
x2=list(range(len(testaccuracy)))
y2=testaccuracy



plt.plot(x1,y1,c='r',label='Training accuracy')
plt.plot(x2,y2,c='g',label='Test accuracy')
plt.legend()
plt.show()


## What happens to the cross entropy loss :
x3=list(range(len(cross_entropy_train)))
y3=cross_entropy_train
x4=list(range(len(cross_entroppy_test)))
y4=cross_entroppy_test



plt.plot(x3,y3,c='r',label='training -cross entropy loss')
plt.plot(x4,y4,c='g',label='test -cross entropy loss')
plt.legend()
plt.show()
```

B. Pedestrian Dataset:

1. Pre-Processing

a) Creatingtrainandtest.py

```
import numpy as np
import os
import os.path
import shutil
import cv2
import tflearn
from tflearn.layers.conv import conv_2d, max_pool_2d
from tflearn.layers.core import input_data, dropout, fully_connected
from tflearn.layers.estimator import regression
from random import shuffle
from tqdm import tqdm
import tensorflow as tf
import matplotlib.pyplot as plt

### Seperates the data into folders with background , and no of pedestrians
## read in a hthe labels of the pedestrian -datatset ##

#tf.reset_default_graph() #reset the graph after ecreating new model :

#backdoor_file=
"C:/Users/creem/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian detection dataset/pedestrian detection
dataset/backdoor/gt.txt"

#backdoor_folder_path="C:/Users/creem/Dropbox/Dissertation/Dissertation/Fin
alDataset-Change/pedestrian detection dataset/pedestrian detection
dataset/backdoor/in#put"

## seperate images for all the folders in the pedestrian datset:


## Replace with the folder name for the full pedestrian dataset :

big_folder="C:/Users/creem/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian detection dataset/pedestrian detection dataset"


IMG_SIZE=50
LR=1*10^(-3)


# Call thename for a model

## Name the model
MODEL_NAME ='ped-class-{}-{}.model'.format(LR,'Preprocessed- 2layer cifar
structure -Dropout deactivated-ZCA whitening-norotation ')


def text_reader(file):
    #initialise a empty list
    lab=[]
    # read fille
    labels= open(file, "r")
    lines=labels.readlines()
## print each line
    for line  in lines:
```

```
        parts=line.split()
        #get the first column of the line

        if (len(parts)!=0):
            col1=parts[0]
            lab.append(int(col1))
        #print(col1)
        #lab1=lab.append(col1)


    return (lab)

#def load_images_numpy :


def  get_index_for_pedestrians(folder_path,gt_file):
    ## Get the image label and split it with its index accoriding
    # to if the label comes


#folder_path = "test"
    images = [f for f in os.listdir(folder_path) if
os.path.isfile(os.path.join(folder_path, f))]
    gtlabels=text_reader(file=gt_file)
    #print(gtlabels)
    #print('images',images)

    ## get the image index for each of the images
    for image in images:

        folder_name=image.split('n')[0]
        file_index=image.split('n')[1]
        file_index=file_index.split('.')[0]
        ## remove the leading zeroes
        file_index=file_index.lstrip("0")
        image_index=int(file_index)

        if image_index in gtlabels:
            #rename the pedestrian pictures :
            print("renaming pedestrian Labels")

os.rename(os.path.join(folder_path,image),os.path.join(folder_path,"ped"+st
r(image_index))+'.jpg')
            #ped.append(image_index)

        else :
        #print("Background images:",image_index)
            print("renaming background labels")

os.rename(os.path.join(folder_path,image),os.path.join(folder_path,"no-
ped"+str(image_index)+'.jpg'))
            #no_ped.append(image_index)


    #return ( {"ped":np.array(ped),"background":np.array(no_ped)})


def label_images_for_pedestrians(img):
    #function to label images for classification according to no background
and pedestrians .
```

```
     word_label=img.split('.')[-2]
     word_label=word_label.split('/')[-1][:2]
     #print(word_label)
     if word_label=="no":return([0,1]) # no pedestrian
     elif word_label=="pe":return([1,0]) #pedestrian


def create_train_data(TRAIN_DIR):
    training_data=[]
    for img in tqdm(os.listdir(TRAIN_DIR)):     # iterate through each of
the images in the folder
        label=label_images_for_pedestrians(img)
        path=os.path.join(TRAIN_DIR,img)

img=cv2.resize(cv2.imread(path,cv2.IMREAD_GRAYSCALE),(IMG_SIZE,IMG_SIZE))
        training_data.append([np.array(img),np.array(label)])
        #print(training_data)
    shuffle(training_data)
    #np.save("train_data.npy",training_data) -- do not save the np array
here ,save it after all the 6 iterations
    return training_data



def create_test_data(TEST_DIR):
    testing_data=[]
    for img in tqdm(os.listdir(TEST_DIR)):
        label=label_images_for_pedestrians(img)
        img_num=img.split('.')[0] # get the id for each test for future
refrence # need to compare with gt labels .
        path=os.path.join(TEST_DIR,img)

img=cv2.resize(cv2.imread(path,cv2.IMREAD_GRAYSCALE),(IMG_SIZE,IMG_SIZE))
        testing_data.append([np.array(img),np.array(label)])
    #np.save("testdata.npy",testing_data) # no need to label test data
    shuffle(testing_data)
    return testing_data

def train_test_label(function="train"):
## Wrapper function to test /train and label the data
    if function=="train":
        ## create an empty numpy array
        final_train_dataset=np.empty((0,2),int)# data for the full 6
folders
        for i in os.listdir(big_folder)[:9]:

                path=os.path.join(big_folder,i)
                file=os.path.join(path,'gt.txt')

                #print(file)
                image_path=os.path.join(path,"input")
#print(file)
                #print(image_path)
                train_data=create_train_data(image_path)


final_train_dataset=np.append(final_train_dataset,train_data,axis=0)

        np.save("traindata.npy",final_train_dataset)
    elif function=="test":
```

```
    # create test data fro the remaining folders
        final_test_dataset=np.empty((0,2),int)
        for i in os.listdir(big_folder)[9:]:

                path=os.path.join(big_folder,i)
                file=os.path.join(path,'gt.txt')

                #print(file)
                image_path=os.path.join(path,"input")
#print(file)

                test_data=create_test_data(image_path)

final_test_dataset=np.append(final_test_dataset,test_data,axis=0)

        np.save("testdata.npy",final_test_dataset)
    elif function=="rename":
        for i in os.listdir(big_folder):

                path=os.path.join(big_folder,i)
                file=os.path.join(path,'gt.txt')

                #print(file)
                image_path=os.path.join(path,"input")
#print(file)


get_index_for_pedestrians(folder_path=image_path,gt_file=file)

    else :
        print("Type either test/train /rename")


## Folders in my current directory  after converted numpy and labelling

# create a property file and read me file

#train_data="C:/Users/creem/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/traindata.npy"
#test_data="C:/Users/creem/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/testdata.npy"
## Load them :

#train_data=np.load(train_data)
#test_data=np.load(test_data)
## function to  create folders according to n pedestrains:

#

if __name__=='__main__':



# Uncomment below to test or train or rename the data

    train_test_label("train")    # produce train data
    train_test_label("test")     # produce test data

## Load them :
```

```
train_data=np.load("traindata.npy")
test_data=np.load(test_data)
print(train_data.shape) #check if it worked ...
```

2. Training

a) Kerasmodel.py

```
from __future__ import print_function
import keras
from keras.datasets import cifar10
from keras.preprocessing.image import ImageDataGenerator
from keras.models import Sequential
from keras.layers import Dense, Dropout, Activation, Flatten
from keras.layers import Conv2D, MaxPooling2D
import pandas
from pandas import read_csv
import numpy
import numpy as np
import math
from keras.models import Sequential
from keras.layers import Dense
from keras.optimizers import SGD
from sklearn.preprocessing import LabelEncoder
from keras.callbacks import LearningRateScheduler
import tensorflow as tf
import sklearn.model_selection as sk
from sklearn.metrics import *

IMG_SIZE=50
batch_size = 96
num_classes = 2
epochs = 5
data_augmentation = True

## Get the traind data


train_data="/home/ha46/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/traindata.npy"
test_data="/home/ha46/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/testdata.npy"


#train_data="C:/Users/Creem/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/traindata.npy"
#test_data="C:/Users/Creem//Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/testdata.npy"


train_data=np.load(train_data)
test_data=np.load(test_data)


# The data, shuffled and split between train and test sets:
```

```
X= np.array([i[0] for i in train_data]).reshape(-1,IMG_SIZE,IMG_SIZE,1)##
get the 1st element which is the image and the 2nd element to ge thte
labels
Y=np.array([i[1]for i in train_data ]) # fit the labels for the Y daata

X=X.astype('float32')
#Y=Y.astype('float32')


x_train, x_test, y_train, y_test = sk.train_test_split(X, Y, test_size=0.2)



print('x_train shape:', x_train.shape)
print(x_train.shape[0], 'train samples')
print(x_test.shape[0], 'test samples')

# Convert class vectors to binary class matrices.
#y_train = keras.utils.to_categorical(y_train, num_classes)
#y_test = keras.utils.to_categorical(y_test, num_classes)



model = Sequential()

model.add(Conv2D(32, (3, 3), padding='same',
                 input_shape=x_train.shape[1:]))
model.add(Activation('relu'))
model.add(Conv2D(32, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.75))

model.add(Conv2D(64, (3, 3), padding='same'))
model.add(Activation('relu'))
model.add(Conv2D(64, (3, 3)))
model.add(Activation('relu'))
model.add(MaxPooling2D(pool_size=(2, 2)))
model.add(Dropout(0.5))

model.add(Flatten())
model.add(Dense(512))
model.add(Activation('relu'))
model.add(Dropout(0.5))
model.add(Dense(num_classes))
model.add(Activation('softmax'))


## Train using a learning rate algorithm
# learning rate schedule


# initiate RMSprop optimizer
opt = keras.optimizers.SGD(lr=0.0001, decay=1e-6)

# Let's train the model using RMSprop
model.compile(loss='categorical_crossentropy',
              optimizer=opt,
              metrics=['accuracy'])
```

```
if not data_augmentation:
    print('Not using data augmentation.')
    model.fit(x_train, y_train,
              batch_size=batch_size,
              epochs=epochs,
              validation_data=(x_test, y_test),
              shuffle=True)
else:
    print('Using real-time data augmentation.')
    # This will do preprocessing and realtime data augmentation:
    datagen = ImageDataGenerator(
        featurewise_center=False,  # set input mean to 0 over the dataset
        samplewise_center=False,  # set each sample mean to 0
        featurewise_std_normalization=False,  # divide inputs by std of the
dataset
        samplewise_std_normalization=False,  # divide each input by its std
        zca_whitening=False,  # apply ZCA whitening
        rotation_range=2.0,  # randomly rotate images in the range
(degrees, 0 to 180)
        width_shift_range=0.1,  # randomly shift images horizontally
(fraction of total width)
        height_shift_range=0.1,  # randomly shift images vertically
(fraction of total height)
        horizontal_flip=True,  # randomly flip images
        vertical_flip=False)  # randomly flip images

    # Compute quantities required for feature-wise normalization
    # (std, mean, and principal components if ZCA whitening is applied).
    datagen.fit(x_train)

    # Fit the model on the batches generated by datagen.flow().
    hist=model.fit_generator(datagen.flow(x_train, y_train,

  batch_size=batch_size),
                        steps_per_epoch=x_train.shape[0] // batch_size,
                        epochs=epochs,
                        validation_data=(x_test, y_test),verbose=0)


true_vals=[]
pred_vals=[]
prob_scores=[]

def predict_values():
    for num, data in enumerate(test_data[:]):
        # cat: [1,0]
        # dog: [0,1]dircd


        # print(data)
        img_num = data[1]
        img_data = data[0]
        # print(img_num)
        # print(img_data)
        # y = fig.add_subplot(4,4,num+1)
        orig = img_data
        data = img_data.reshape(-1, IMG_SIZE, IMG_SIZE, 1)  # Rehsape the
daya to be supllied correctly into tensorflow
```

```
        model_out = model.predict([data])[0]  ### produce a label of
predicitions

        if np.argmax(img_num) == 1: true_vals.append(0)
        if np.argmax(img_num) == 1: prob_scores.append(
            model_out[1])  ## return the index which is the maximum
argument :
        if np.argmax(img_num) == 0: true_vals.append(1)
        if np.argmax(img_num) == 0: prob_scores.append(model_out[0])

        if np.argmax(model_out) == 1:
            pred_vals.append(0)  ## return the index which is the maximum
argument :
        elif np.argmax(model_out) == 0:
            pred_vals.append(1)

    return (np.array(true_vals), np.array(pred_vals),
np.array(prob_scores))


# confusion_matrix




a=predict_values()

print("Accuracy ",accuracy_score(a[0], a[1]))
print("precision:",precision_score(a[0], a[1],average="weighted"))
print("roc_auc_score:",roc_auc_score(a[0], a[2]))
print("Recall:",recall_score(a[0], a[1], average='weighted'))
print(confusion_matrix(a[0], a[1]))
```

3. Testing

a) Confusionmatrixandroccurves.py

```
''' Plot the roc curves  and confusion matrixes for models '''

from __future__ import print_function

import matplotlib.pyplot as plt
from sklearn.metrics import roc_curve ,auc
#from confusionmatrixmodels  import *
import numpy as np

##File to obtain confusion and roc curves for the best and worst performing
models##########################################


from hyperopt import Trials, STATUS_OK, tpe
from hyperas import optim
from hyperas.distributions import uniform
from keras.models import Sequential
from keras.layers.core import Dense, Dropout, Activation, Flatten
from keras.layers.convolutional import Convolution2D, MaxPooling2D
from keras.optimizers import SGD
from keras.preprocessing.image import ImageDataGenerator
```

```
from keras.datasets import cifar10
from keras.utils import np_utils
import sklearn.model_selection as sk
from sklearn.metrics import *



import numpy as np
from keras.models import load_model
# Save the model as png file
from keras.utils import plot_model
#from labelimagesconfustionmatrix import *
import matplotlib.pyplot as plt

model_1 = load_model('bestmodel.h5') ### model with the best results
model_2=load_model("badmodel.h5") ## model with the worst results
#model_3=load_model("bestmodel.h5")
 # load the saved model
#train_data="/home/ha46/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/traindata.npy"
#test_data="/home/ha46/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/testdata.npy"

train_data="C:/Users/Creem/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/traindata.npy"
test_data="C:/Users/Creem//Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/testdata.npy"


IMG_SIZE=50
train_data=np.load(train_data)
test_data=np.load(test_data)

#MODEL_NAME="ped-class--9-Preprocessed- 2layer cifar structure -Dropout
deactivated-.model" ## Best Model-Predictions
#odel.save(MODEL_NAME)


#plot_model(model_1, to_file='model.png', show_shapes=True)

##save  agraph from tensorbaorda=



#model.load(MODEL_NAME)
#fig=plt.figure()

#
#print(model_1)

#y_actu = [2, 0, 2, 2, 0, 1, 1, 2, 2, 0, 1, 2]
#y_pred = [0, 0, 2, 1, 0, 2, 1, 0, 2, 0, 2, 2]
#confusion_matrix(y_actu, y_pred)



## initiliase a empty numoy array
```

```
true_vals=[]
pred_vals=[]
prob_scores=[]

def predict_values(model=model_1):
    for num,data in enumerate(test_data[:]):
        # cat: [1,0]
        # dog: [0,1]dircd


        #print(data)
        img_num = data[1]
        img_data = data[0]
        #print(img_num)
        #print(img_data)
        #y = fig.add_subplot(4,4,num+1)
        orig = img_data
        data = img_data.reshape(-1,IMG_SIZE,IMG_SIZE,1)# Rehsape the daya
to be supllied correctly into tensorflow

        model_out = model.predict([data])[0]  ### produce a label of
predicitions




        if np.argmax(img_num) == 1: true_vals.append(0)
        if np.argmax(img_num)==1: prob_scores.append(model_out[1])  ##
return the index which is the maximum argument :
        if np.argmax(img_num) == 0: true_vals.append(1)
        if np.argmax(img_num)==0: prob_scores.append(model_out[0])

        if np.argmax(model_out) == 1: pred_vals.append(0)  ## return the
index which is the maximum argument :
        elif np.argmax(model_out) == 0: pred_vals.append(1)

    return (np.array(true_vals),np.array(pred_vals),np.array(prob_scores))

#confusion_matrix



a=predict_values(model=model_1)

print("Accuracy ",accuracy_score(a[0], a[1]))
print("precision:",precision_score(a[0], a[1],average="weighted"))
print("roc_auc_score:",roc_auc_score(a[0], a[2]))
print("Recall:",recall_score(a[0], a[1], average='weighted'))
print(confusion_matrix(a[0], a[1]))




fpr, tpr, threshold = roc_curve(a[0], a[2])
roc_auc = auc(fpr, tpr)

plt.title('Receiver Operating Characteristic')
plt.plot(fpr, tpr, 'b', label = 'AUC = %0.2f' % roc_auc)
plt.legend(loc = 'lower right')
plt.plot([0, 1], [0, 1],'r--')
```

```
plt.xlim([0, 1])
plt.ylim([0, 1])
plt.ylabel('True Positive Rate')
plt.xlabel('False Positive Rate')
plt.show()
```

4. Model Evaluation

a) Visualiselayers.py

```
''' Script to look at how the model differentiates with a pedestrian
between a good model and worse performing model'''


## get a pedestrian image and pass it as a image to both the models , see

#from labelimagesconfustionmatrix import *
#from confusionmatrixkerasmodel import *
from keras.models import Model



IMG_SIZE =50

## change for data structure :


train_data="C:/Users/Creem/Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/traindata.npy"
test_data="C:/Users/Creem//Dropbox/Dissertation/Dissertation/FinalDataset-
Change/pedestrian classification
script/modelforconfusionmatrix/testdata.npy"


model_1="bestmodel.h5"
model_2="badmodel.h5"

def get_pedestrian(lab=0):
    ped_image_data=[]
        #function to get a pedestrian picture
    for  img,data in enumerate(train_data[:]):
          label=data[1]
          img_data=data[0]
          if np.argmax(label)==lab:
            ped_image_data.append(img_data)
    return(np.array(ped_image_data))


def visualize_pedestrian(model, pedestrian,layer):
    # Keras expects batches of images, so we have to add a dimension to
trick it into being nice
    pedestrian_batch = pedestrian.reshape(-1,IMG_SIZE,IMG_SIZE,1)
    ##Choose the right no of layers
    #layer_name = 'my_layer'
    intermediate_layer_model = Model(inputs=model.input,
                                     outputs=model.layers[layer].output)
    conv_pedestrian = intermediate_layer_model.predict(pedestrian_batch)
    print(conv_pedestrian)
    conv_pedestrian = np.squeeze(conv_pedestrian, axis=0)
    print(conv_pedestrian.shape)
```

```
    plt.imshow(conv_pedestrian)
    plt.show()



def make_mosaic(im, nrows, ncols, border=1):

    ## helper function for the below visualisations###
    """From
http://nbviewer.jupyter.org/github/julienr/ipynb_playground/blob/master/keras/convmnist/keras_cnn_mnist.ipynb
    """
    import numpy.ma as ma

    nimgs = len(im)
    imshape = im[0].shape

    mosaic = ma.masked_all((nrows * imshape[0] + (nrows - 1) * border,
                            ncols * imshape[1] + (ncols - 1) * border),
                            dtype=np.float32)

    paddedh = imshape[0] + border
    paddedw = imshape[1] + border
    im
    for i in range(nimgs):

        row = int(np.floor(i / ncols))
        col = i % ncols

        mosaic[row * paddedh:row * paddedh + imshape[0],
               col * paddedw:col * paddedw + imshape[1]] = im[i]

    return mosaic
## Pass a single ped image and see how the final layer looks like for the working model  :
#visualize_pedestrian(model=model_1,pedestrian=get_pedestrian()[1],layer=1)
def plot_feature_map(model, layer_id, X, n=256, ax=None, **kwargs):
    """

    need to author code
    """
    import keras.backend as K
    import matplotlib.pyplot as plt
    from mpl_toolkits.axes_grid1 import make_axes_locatable

    layer = model.layers[layer_id]

    try:
        get_activations = K.function([model.layers[0].input,
K.learning_phase()], [layer.output,])
        activations = get_activations([X, 0])[0]
    except:
        # Ugly catch, a cleaner logic is welcome here.
        raise Exception("This layer cannot be plotted.")

    # For now we only handle feature map with 4 dimensions
    if activations.ndim != 4:
        raise Exception("Feature map of '{}' has {} dimensions which is not supported.".format(layer.name,
```

```
activations.ndim))

    # Set default matplotlib parameters
    if not 'interpolation' in kwargs.keys():
        kwargs['interpolation'] = "none"

    if not 'cmap' in kwargs.keys():
        kwargs['cmap'] = "jet"

    fig = plt.figure(figsize=(15, 15))

    # Compute nrows and ncols for images
    n_mosaic = len(activations)
    nrows = int(np.round(np.sqrt(n_mosaic)))
    ncols = int(nrows)
    if (nrows ** 2) < n_mosaic:
        ncols +=1

    # Compute nrows and ncols for mosaics
    if activations[0].shape[0] < n:
        n = activations[0].shape[0]

    nrows_inside_mosaic = int(np.round(np.sqrt(n)))
    ncols_inside_mosaic = int(nrows_inside_mosaic)

    if nrows_inside_mosaic ** 2 < n:
        ncols_inside_mosaic += 1

    for i, feature_map in enumerate(activations):

        mosaic = make_mosaic(feature_map[:n], nrows_inside_mosaic,
ncols_inside_mosaic, border=1)

        ax = fig.add_subplot(nrows, ncols, i+1)

        im = ax.imshow(mosaic, **kwargs)
        ax.set_title("Feature map #{} \nof layer#{} \ncalled '{}' \nof type
{} ".format(i, layer_id,

layer.name,

layer.__class__.__name__))

        divider = make_axes_locatable(ax)
        cax = divider.append_axes("right", size="5%", pad=0.1)
        plt.colorbar(im, cax=cax)

    fig.tight_layout()
    plt.show()
    return fig
    #
    #
if __name__=="__main__":


#img=cv2.resize(cv2.imread(path,cv2.IMREAD_GRAYSCALE),(IMG_SIZE,IMG_SIZE))

    reshaped_image=get_pedestrian(lab=1)[5].reshape(-1,IMG_SIZE,IMG_SIZE,1)
    plt.imshow(get_pedestrian(lab=1)[5])
    plot_feature_map(model_1,0,reshaped_image, n=9)
```

```
    plot_feature_map(model_2,0, reshaped_image, n=9)
```

b) *simulation-batchsizes.py*

```
'''Simulation file to run the keras model a 100 times , see what the
average accuracy , precision and TPR are : used for obtaining graphsin
Fig24 and Fig 25 in Report'''



'''Code to run the model a 100 times , change batch sizes in kerasmdel.py
and run the below code '''

import subprocess
import sys
from tqdm import *
for scriptInstance in tqdm(range(100)):
    sys.stdout = open('resultbatch32%s.txt' % scriptInstance, 'w')
    subprocess.check_call(['python', 'kerasmodel.py'],
                          stdout=sys.stdout, stderr=subprocess.STDOUT)

for scriptInstance in tqdm(range(100)):
    sys.stdout = open('resultbatch200%s.txt' % scriptInstance, 'w')
    subprocess.check_call(['python', 'kerasmodel200.py'],
                          stdout=sys.stdout, stderr=subprocess.STDOUT)

for scriptInstance in tqdm(range(100)):
    sys.stdout = open('resultbatch300%s.txt' % scriptInstance, 'w')
    subprocess.check_call(['python', 'kerasmodel300.py'],
                          stdout=sys.stdout, stderr=subprocess.STDOUT)

## the results are printed to a txt file

##########################################################################
##########################################################################
############



##########################################################################
##########################################################################
########################

### The below code , combines all the text file , into one and then takes
the accuracy , roc , recall etc and prints to the screen #######

import glob
read_files = glob.glob("*.txt")

with open("result200.txt", "wb") as outfile:
    for f in read_files:
        with open(f, "rb") as infile:
            outfile.write(infile.read())



Accuracy=[]
```

```
Precision=[]
roc_auc_score=[]
Recall=[]

flag=False
flag1=False
with open('result200.txt','r') as f:
    for line in f:
        if line.startswith('Accuracy '):
            flag=True
        if flag:
            Accuracy.append(line)
        if line.strip().startswith('Recall:'):
           flag=False

print( ''.join(Accuracy))


        if line.startswith('precision:'):
            flag=True
         if flag:
             Precision.append(line)
        if line.startswith('roc_auc_score:'):
            flag=True
         if flag:
             roc_auc_score.append(line)
        if line.startswith('Recall:'):
            flag=True
         if flag:
             Recall.append(line)


print (Accuracy,Precision,roc_auc_score,Recall)



##########################################################################
########################################
```

XII. References

A.Nielsen, M. (2015). Retrieved from neuralnetworksanddeeplearning: http://neuralnetworksanddeeplearning.com/chap1.html

Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., & al., G. S. (2015). *Large-Scale Machine Learning on Heterogeneous Distributed Systems.* Tensor Flow. Retrieved from https://www.tensorflow.org/.

Bottou, L. (2010). Large-Scale Machine Learning with Stochastic Gradient Descent. In L. Bottou, *Proceedings of the 19th International Conference on Computational Statistics (COMPSTAT'2010)* (pp. 177-187). Paris: Springer.

Bowden, P. a. (2001). An Improved Adaptive Background Mixture Model for Realtime Trackign with Shadow detection. *KluwerAcademic Publishers*, (pp. 1-5).

Buduma, N. (2017). *Fundementals of Deep Learning : Designing Next-Generation Machine Intelligence Algorithms.* United States of America: O'Reilly.

Chen, G. Z. (2012). A review on vision-based pedestrian detection. *2012 IEEE Global High Tech Congress on Electronics*, (pp. 49-54). Shenzhen.

Christian Szegedy, W. Z. (2014). Intriguing properties of neural networks. *ICLR* .

Cires¸an, D., Meier, U., & Schmidhuber, J. (2012). Multi-column deep neural networks for image classification. *IEEE Conference on Computer Vision and Pattern Recognition*, (pp. 3642-3649).

Deng, Y. (2014). Pedestrian attribute recognition at far distance. *Proceedings of the ACM International.*

Diederik P.Kingma, J. L. (2015). ADAM : A METHOD FOR STOCHASTIC OPTIMIZATION. *ICLR*, (pp. 1-15).

Dollar, P. C. (2009). Pedestrian Detection : A Bench mark . *IEE Conference on Computer Vision and Pattern Recognition.*

Erik Bochinski, T. S. (2017). *HYPER-PARAMETER OPTIMIZATION FOR CONVOLUTIONAL NEURAL NETWORK COMMITTEES BASED ON EVOLUTIONARY ALGORITHMS.* IVIU: Image & Video Interpretation & Understanding: IVIU-REC Object Recognition and Classification.

Ernesto Mastrocinque, M. S. (2014). Neural network design and feature selection using principal component analysis and Taguchi method for identifying wood veneer defects. *Production & Manufacturing Research*.

Görner, M. (2015). *TensorFlow and DeepLearning without a PHD*. Retrieved from https://codelabs.developers.google.com/codelabs/cloud-tensorflow-mnist/#0.

Graves, A. (2012). "Supervised sequence labelling with recurrent neural networks". *Springer Vol 385*.

H. Wang, M. M. (2009). Evaluation of local spatio-temporal features for action recognition. *BMVC*.

I. Laptev, M. M. (2008). Learning realistic human actions from movies. *CVPR*.

Ian J Goodfellow, J. S. (2015). Explaining and harnessing adversarial examples. *ICLR* .

J.Yao, J. (2008). Fast human detection from videos using covariance features. *Proceedings European Conference on Computer Vision*.

Karpathy, A. (2014, Sep 2). *github*. Retrieved from Andrej karpathy blog: http://karpathy.github.io/2014/09/02/what-i-learned-from-competing-against-a-convnet-on-imagenet/

Karpathy, A. (2017). *CS231n: Convolutional Neural Networks for Visual Recognition*. Retrieved from http://cs231n.github.io/neural-networks-2/#reg.

Karpathy, A. (2017). *CS231n: Convolutional Neural Networks for Visual Recognition.* . Retrieved from http://cs231n.github.io/neural-networks-3/#hyper.

Karpathy, A., Toderici, G., Shetty, S., Leung, T., Sukthankar, R., & Fei-Fei, L. (2014). Large-scale Video Classification with Convolutional Neural Networks. *Computer Vision and Pattern Recognition (CVPR).* Columbus ,USA: IEEE.

Keras. (2017). *Keras Documentation*. Retrieved from https://keras.io/.

Krizhevsky, A. (2009). *Learning Multiple Layers of Features from Tiny Images.* University of Toronto.

Krizhevsky, A. I. (2012). "Imagenet classification with deep convolutional neural networks". *Advances in neural information processing systems*.

LeCun, Y., Cortes, C., & Burges, C. J. (n.d.). *MNIST handwritten digit database, Yann LeCun, Corinna Cortes and Chris Burges*. Retrieved from http://yann.lecun.com/exdb/mnist/

M.Paulin, J. ,. (2014). Transformation pursuit for image classification. *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, (pp. 3646-3653).

N. Goyette, P.-M. J. (Jun 2012). changedetection.net: A new change detection benchmark dataset. *in Proc. IEEE Workshop* , (pp. 16-21).

N.Dalal, B. C. (2006). Human detection using oriented histograms of flow and appearance. *Proceedings of the IEEE European Conference on Computer Vision.*

Nielsen, M. (2015). *Chap1*. Retrieved from Neuralnetworksanddeeplearning: http://neuralnetworksanddeeplearning.com/chap1.html

Nitish Srivastava, G. H. (2014). Droput: A Simple way to Prevent Neural Netwoks from Overfitting. *Journal of Machine Learning Research*, 1929-1958.

P.Meer, D. V. (2003). “Kernel based object tracking”. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 564-575.

P.Viola, M. D. (2005). Detecting pedestrians using patterns of motion and appearance,. *International Journal of Computer Vision*, 153-161.

Pal, K. K., & Sudeep, K. S. (2016). Preprocessing for image classification by convolutional neural networks. *IEEE International Conference on Recent Trends in Electronics, Information & Communication Technology (RTEICT).*

Pedregosa, F. a. (2017). *scikit learn*. Retrieved from http://scikit-learn.org/stable/modules/generated/sklearn.metrics.precision_recall_fscore_support.html.

R.Cutler, L. (2000). Robust real-time periodic motion detection,analysis and applications. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 781-796.

Ross Girshick, J. D. (2013). Rich feature hierarchies for accurate object detection and semantic segmentation. *Computer and Vision Pattern recognition*.

S.Walk, N. K. (2010). "New features and insights for pedestrian detection. *Proceedings of the IEEE International Conference on Computer Vision and Pattern Recognition.*

Sergey Ioffe, C. S. (2015). Batch Normalization: Accelerating deep network training by reducing Internal covariate shift. *Proceedings of the 32 nd International Conference on Machine learning*.

Seyed-Mohsen Moosavi-Dezfooli, A. F. (2015). Deepfool: a simple and accurate method to fool deep neural networks. *CVPR*.

Shimodaira, H. (2000). Improving predictive inference under covariate shift by weighting the log-likelihood function. *Jorunal of Statiscal Planning and Inference* , 227-244.

Shirai, T. T. (1985). Detection of the movements of persons from a sparse sequence of tv image. *Pattern Recognition* , 18(3–4):207 – 213,.

Smithson, S. C., Yang, G., Gross, W. J., & Meyer, B. H. (2016). Neural Networks Designing Neural Networks:Multi-Objective Hyper-Parameter Optimization. *Sean C. Smithson Guang Yang Warren J. Gross Brett H. Meyer.* Austin ,Texas.

Variyar, A. (2016). Application of Convoluted Neural Networks for Pedestrian Detection.

W.R.Schwartz, A. D. (2009). "Human detection using partial least squares analysis. *Proceedings of IEEE Conference on Computer Vision*, 24-31.

Wang, Q., Jinfang, Z., Xiaohui, H., & Yang, W. (2016). Automatic Detection and Classification of Oil Tanks in Optical Satellite Images Based on Convolutional Neural Network. In *Image and Signal Processing* (pp. 304-313). Morroco: Springer.

Y. LeCun, B. J. (1989). Backpropogation Applied to Handwritten Zip Code Recognition. *Neural Computation*, 541-551.

Yann LeCun, L. B. (1998). Gradient-Based Learning Applied to Document Recognition. *PROC OF THE IEEE.*

Zivkovic, Z. (2004). Improved Adaptive Gaussian Mixture Model for Background Subtraction. *IPCR*, (pp. 1-4).