Towards Real-Time Human Pose Estimation For Mobile Device

Ramkumar Sakthivel

TU Delft

Towards Real-Time Human Pose Estimation For Mobile Device

by

Ramkumar Sakthivel

to obtain the degree of Master of Science
at the Delft University of Technology,
to be defended publicly on October 7, 2020.

Student number:	4779460	
Project duration:	December 1, 2019 - October 7, 2020	
Thesis committee:	Prof. Dr. Dariu M. Gavrila,	TU Delft, Chair, Cognitive Robotics
	Dr. Wei Pan,	TU Delft, Supervisor, Cognitive Robotics
	Ir. Hongpeng Zhou,	TU Delft, Daily Supervisor, Cognitive Robotics
	Dr. Jan van Gemert,	TU Delft, External Member

An electronic version of this thesis is available at `http://repository.tudelft.nl/`.

TU Delft Delft University of Technology

Preface

This master thesis marks the completion of the number of credits required to obtain the Masters in Mechanical Engineering under the track 'Vehicle Engineering' at the Delft University of Technology.

I would like to thank my supervisor Dr. Wei Pan for not only giving me the opportunity to work on such an interesting research project, but also for giving me advice and motivation during the difficult times. I feel privileged to have contributed to his research work. I would also like to thank my daily supervisor Ir. Hongpeng Zhou for patiently clearing all my doubts and providing insightful suggestions for the research.

Last but not the least, I would like to thank my family for always encouraging me throughout my journey as a graduate student.

Ramkumar Sakthivel
Delft, October 2020

Abstract

Human pose estimation, a challenging computer vision task of estimating various human body joints' locations, has a wide range of applications such as pedestrian tracking for autonomous cars, baby monitoring, video surveillance, human action recognition, virtual reality, gaming, gait analysis, etc. A majority of the research on the development of models for the task of human pose estimation has been focused on improving the accuracy of the task which also increases the complexity of the models. These models demand devices with high computational power to be deployed for real-world applications. Even though a lot of research has been focused on estimating the human pose from monocular images taken from cameras, the complexity of the models makes them impossible to be implemented on edge devices and embedded devices like mobile phones that have built-in cameras. This reduces the scope of applications where human pose estimation can be used. To address the issue, the research focuses on improving the performance of a baseline human pose estimation architecture by reducing the model size(number of parameters) and thereby its inference time without a significant loss in the accuracy. To improve the performance of the model, a structured Bayesian compression algorithm is used and the network is compressed by engineering the model based on the uncertainty of the parameters. The results show that the Bayesian compression method reduces the model size by around 65 percent with only a very little drop in the model accuracy. Also, the comparison of the inference time of the original baseline and the compressed model in an android device shows that the inference time is reduced by around 50 percent because of the reduction in the number of operations in the compressed model architecture.

Contents

List of Figures

List of Tables

1 Introduction

Human pose estimation is a computer vision technique that detects and localizes human body joints in an image or video. Usually, the body joints, also called as the keypoints include head, neck, shoulder, elbow, wrist, hip, knee and ankle. Connecting these keypoints in a corresponding order produces a human skeleton figure that describes the pose of the person as shown in figure 1.1. To avoid misconception, human pose estimation technique does not recognize who is in the image but just estimates the position of the body joints. The task of human pose estimation from images is very challenging because of the high degree of freedom associated with the configuration of the limbs and also the variations such as clothing, lighting, body-type, occlusion, lighting etc. Also, finding the exact location of the body parts that occupy only very few pixels in an image is relatively complicated compared to just detecting a person in an image.

Pose estimation techniques can be used in many applications such as intelligent driver assistant systems to detect and track pedestrians, old age/assisted homes to detect human fall, baby monitoring, character animation, augmented reality, gaming, healthcare to detect abnormal patient postures, robotics, video surveillance, activity/behaviour understanding, sign-language detection and human computer interaction.

The current state of the art pose estimation methods use deep learning based techniques that use very complex architectures so as to achieve better accuracy. These models with complex architecture need a lot of computation power to estimate the human pose in realtime. However, pose estimation techniques are not used in a lot of commercial applications because of the fact that high end GPUs are expensive and are hard to be built-in on embedded devices like mobile devices. So it is necessary to build an efficient pose estimation model to make it run on embedded devices like a mobile device in realtime for widening the scope of its applications.

1.1. Research Question

The question to be addressed by this research is the following:

- Whether the structured Bayesian compression algorithm [13] prove to be successful in compressing and improving the performance of a baseline human pose estimation architecture with a very little loss in the accuracy?

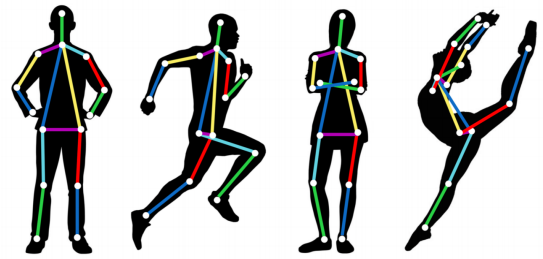


Figure 1.1: Example human pose skeletons [1]

1.2. Objectives

The objectives of this research are to:

- Build a baseline model architecture for 2D human pose estimation that is simple and capable of being converted into an android model.
- Implement the Bayesian algorithm to introduce structural sparsity on the baseline model and then engineer the sparse model to obtain the compressed model.
- Develop android applications to compare the inference time and size on mobile device.

1.3. Overview

This report addresses the above mentioned objectives in a sequential manner. In chapter 2 all the necessary concepts required to understand the methodologies used in this research are elaborated. Chapter 3 gives a detailed explanation on the literature survey carried out for this research. The network compression methodologies and the steps to engineer the model are explained in chapter 4. Chapter 5 explains the experimental setup and the results of the thesis and also includes the observations. The thesis work is concluded in Chapter 6 and the extra material is added to the appendix section.

2

Background

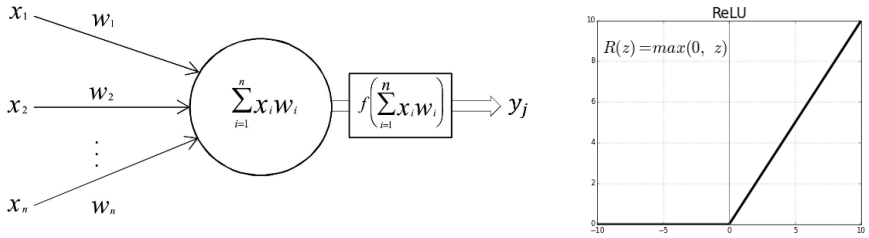
In this chapter a background theory to understand the concepts and methods used in the research are explained.

2.1. Neural Network

Neural networks, also called as artificial neural networks are a type of machine learning models that are inspired by the working of the human brain. They are proven to be efficient in mapping the input signals to desired outputs in a variety of problems. This subsection explains about the essential components of a neural network.

2.1.1. Neuron

Neural networks are made up of neurons, which are the basic building blocks. A neuron can have arbitrary number of inputs and yields a single output value. This value can be passed as an input to arbitrary number of neurons in a network. Two operations are performed by each neuron, the first operation is linear and it involves calculating the weighted sum of the inputs added with a bias. The weights and bias are called parameters that are learned during the process of training a neural network. The second operation introduces non-linearity by applying a function called as 'activation function' over the weighted sum. There are different kinds of activation functions but the most widely used activation function is the Rectified Linear Units (ReLU) [15] that only activates the neuron when the linear combination of inputs, weights and biases are positive. Figure 2.1 shows the visual representation of a neuron and the ReLU activation function.



(a) Neuron (b) ReLU

Figure 2.1: Neuron and ReLU Activation [2]

2.1.2. Loss Function

A neural network tries to fit a function for a task that maps the inputs to the desired outputs. The process of fitting a function is called as training during which the parameters of the neural network are updated such that the output generated by the neural network is close to the desired output. The process of training starts with a forward pass during which the neural network model generates an output from the input. In order to update the parameters, a value that depicts the difference between the desired output and the output from the model has to be calculated. This value is called as the loss and it is represented as a function called loss function. There are different kinds of loss functions [16] and the choice of the loss function depends on the type of problem the neural network model needs to address.

The most widely used loss function for regression type problems is the Mean Squared Error (MSE) that calculates the mean of the squared difference between the actual and the desired output as seen in equation (1).

$$L(y, \hat{y}) = \frac{1}{N} \sum_{i=0}^{N} \left(y - \hat{y}_i\right)^2 \tag{2.1}$$

where y is the desired output and $\hat{y}$ is the prediction made by the neural network model.

2.1.3. Backpropagation

The values of the parameters are updated such that the loss function is minimised. In order to update the parameters the gradient of the loss function with respect to the parameters is calculated. As the gradient points in the direction of maximising a function, updating the parameters using their negative value of the gradient helps in minimising the loss function. This process of updating the parameters is called as backpropagation [17].

The equations used to update the parameters during backpropagation are (2) and (3),

$$w := w - \alpha \frac{\partial L}{\partial w} \tag{2.2}$$

$$b := b - \alpha \frac{\partial L}{\partial b} \tag{2.3}$$

where L is the loss function, w and b are the parameters and α is called as the learning rate that updates the value of parameters in small steps.

2.1.4. Optimization

As seen in equations (2) and (3), the rate by which the value of the parameters gets updated depends on the learning rate. The lower the learning rate the slower is the convergence to the minima of the loss function. Convergence might fail for a high value of learning rate. It is a hyper-parameter that has to be chosen properly for a smooth convergence.

The algorithms that help in a faster and a smooth convergence are called as optimisation algorithms. Stochastic Gradient Descent (SGD) is an optimization algorithm that uses the mean of the gradients calculated on a small batch of data and updates the parameters in smaller steps. This helps in reducing the computation power. There are a wide variety of such optimisation algorithms and are discussed in [18].

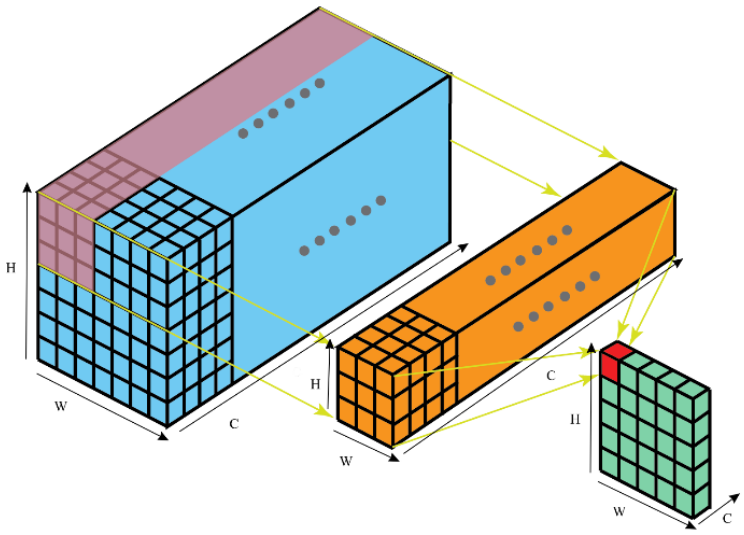


Figure 2.2: Visualization of convolution operation [3]

2.2. Convolutional Neural Network

A convolutional neural network (CNN) is a type of neural network that uses a method called as convolution to map the inputs to the output. CNN are being widely used on tasks involving images as these methods are proven to extract much information from images that can be used for prediction. This subsection explains about the essential components of a convolutional neural network.

2.2.1. Convolution

This subsection explains the convolution operation used when image is given as an input to CNN. So, the input layer consists of a stack of two dimensional matrix also called as a tensor. Each value of the tensor can be visualised as a input neuron. Every image has three channels (one for red, blue and green) and the values represent the intensities at the corresponding pixels in the image. A convolutional kernel is made up of a stack of filters whose number of input channels equals the number of output channels of the previous layer. The width and height of the kernel is a hyper-parameter that is usually an odd number. The convolutional kernel is also referred to as a convolutional filter. The filter slides across the input tensor and at each position the values of the input and filter gets multiplied and added to produce a single value corresponding to the position of the input pixel. These values are then stored in the output tensor called as the feature map. The number of filters used in the convolutional layer equals the number of channels in the output feature map. Figure 2.2 shows a visualization of the convolution operation.

In the figure 2.2, each small cube represents a neuron. The big blue cuboid represents the input with a width, height and number of channels. The orange cuboid represents the filters which can have arbitrary width and height but the number of channels of the filter has to match the number of channels of the input. At every position the filter operates on the inputs and produces an output that is added to the feature map as shown in green. The width and height of the output feature map depends on the width and height of the input and the filter. Every filter (orange box) produces one feature map channel. The feature map with multiple channels serves as the input for the upcoming convolutional layer and the process continues. At every layer the filters connect the input and the output neurons and so they act as the weights of the CNN. These weights are updated using backpropagation as explained in the previous sections. As it can be seen from the figure 2.2, every

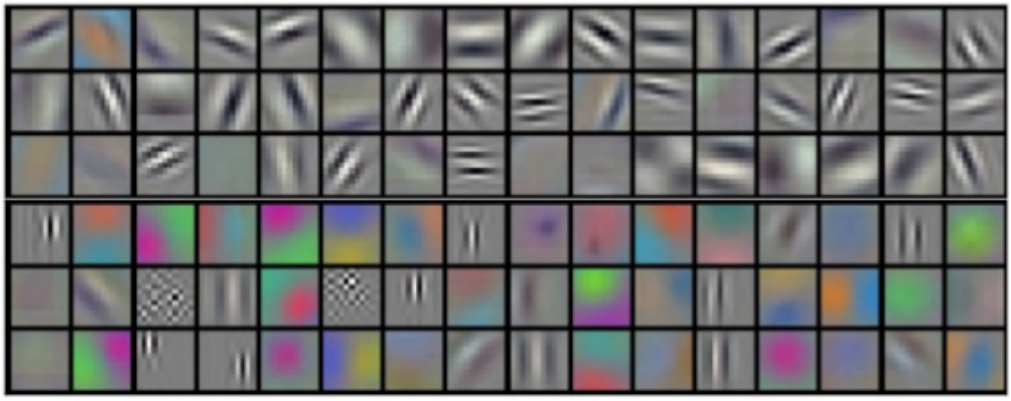


Figure 2.3: Visualization of features learned by convolution filters [4]

channel of the input neurons share the weights of one convolutional filter.

Intuitively these weights try to learn the underlying features in the image and the feature maps extract the most useful information from the images. These extracted information can be used further to map to the outputs. Figure 2.3 shows the type of features learned by the convolutional filters in a CNN.

From the figure 2.3 it can be seen that the convolutional filters learn to extract features like edges at different orientations, colours, patterns, etc. form an image.

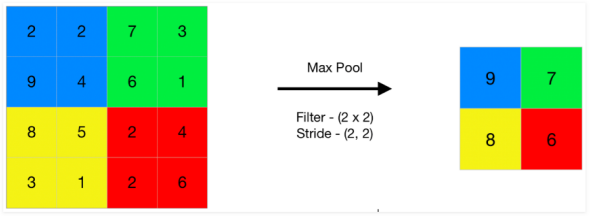
2.2.2. Pooling

The activation function is applied to the feature map outputs generated by the convolution operation. Then the pooling operation which is an optional operation can be applied on the feature map to reduce its resolution. Pooling operation uses a kxk region to slide over the feature map with a stride value s. The value of the stride decides the amount of reduction in the resolution of the feature map. For example a stride value of 2 reduces the spatial resolution of the feature map by half. Pooling operation is done to reduce the computational complexity and also make the CNN learn features with positional invariance in an image. Most widely used types of pooling are max and average pooling. To visualize the pooling operation figure 2.4 shows how pooling reduces the resolution on a single channel of a feature map. In max pooling only the maximum value in the region is retained while in average pooling the average of the values in the region is retained.

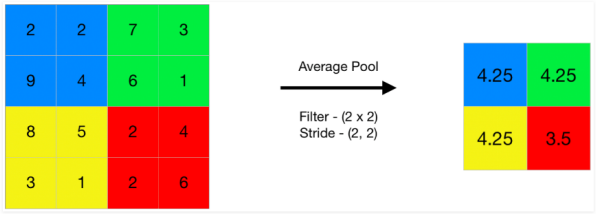
2.2.3. Deconvolution

Until now the operations explained on CNN reduce the resolution of the feature maps, but if it is necessary to increase the feature map resolution then upsampling methods are used. Instead of just re-scaling the feature maps using interpolation techniques, upsampling methods use robust operation to increase the feature map resolution such that the information lost during the downsampling operation is regained back. Among the upsampling methods deconvolution operation is the most widely used because of its robustness achieved using the learnable parameters that aid in upsampling. In fact the correct term for this operation is called as the transposed convolution that involves a transposed version of the normal convolution operation.

To better understand how transposed convolution works, let us consider a single channel of a feature map in a CNN layer with dimension $3x3$. Let us consider that the required output dimension of the feature map is $5x5$. During transposed convolution, The input feature map is padded



(a) Max Pooling



(b) Average Pooling

Figure 2.4: Pooling [5]

with sufficient rows and columns filled with zeros and then normal convolution operation is performed using a convolutional filter. Figure 2.5 shows the transposed convolution operation.

In the figure 2.5, the blue cells represent the input and the green cells represent the output. The grey cells represents the convolutional filter and the values of the filter are learned during training the CNN model. The unfilled dotted cells represent the zero cells added to match the input, output and the filter dimensions. This operation has proven to be robust in regaining the information as the parameters of the filters are learned during the process of backpropagation as explained in previous subsections.

2.3. CNN Architecture

A complete CNN architecture consists of a combination of convolution, activation function and pooling layers in a sequential manner. Based on the way these layers are arranged there are different types of architectures that have their own advantages and disadvantages. Among the architectures, the most popular one is ResNet [7] because of its ability to achieve a very high accuracy on Imagenet classification challenge.

2.3.1. ResNet Architecture

A CNN is called as a deep neural network when the number of layers in the architecture is high. Since every layer in the network learns features from the image and help in extracting useful information, intuitively the statement 'the greater the number of layers, the higher is the accuracy of the model' should be true. However this is not the case in reality and various factors that hinder this theory are overfitting and the problem of vanishing/exploding gradients.

According to the paper [7], increasing the number of layers in an architecture has an increased error during training when compared to a shallower network. This is very much counter intuitive because at least if the deeper models are not able to perform better than shallower networks, the error should not increase as the number of layer increases because the deeper networks should be able to learn the identity function and should able to match the low error of their shallow counterparts. The experiments conducted shows that this was not the case which brought a conclusion that

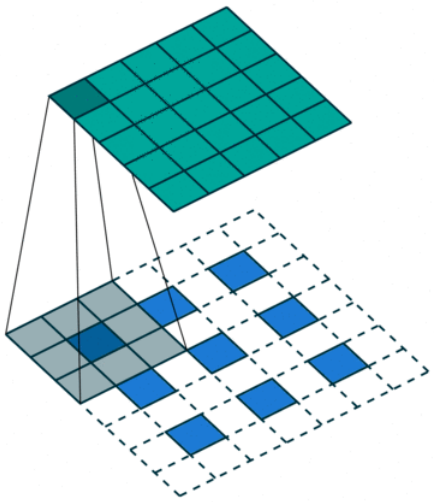
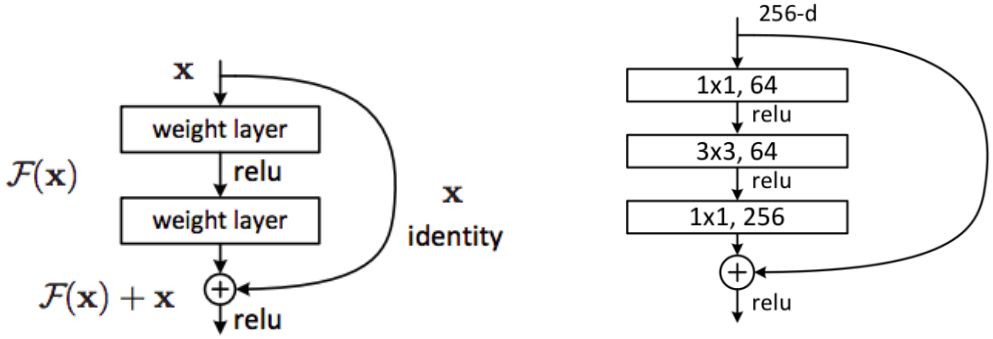


Figure 2.5: Transposed Convolution [6]



(a) Residual block (b) Operations in a block

Figure 2.6: Residual block in ResNet architecture [7]

the identity mapping is not an easy function to be learned by the model.

To alleviate this problem, the concept of residual connection or skip connection was introduced in [7]. The architecture was thus named as residual neural network architecture. The skip connections proposed in [7] help the model to learn the identity mapping easily. This helped in building deep architectures with many number of layers and improved accuracy. A residual neural network is made up of residual blocks that contain the skip connections in them. These connections aid in bypassing the information from previous layers to next layers in the network architecture. A residual block with a skip connection is shown in figure 2.6.

Nowadays newer versions of these residual blocks are being used that help in improving the accuracy of the model. The newer version of the residual block is shown in figure 2.7.

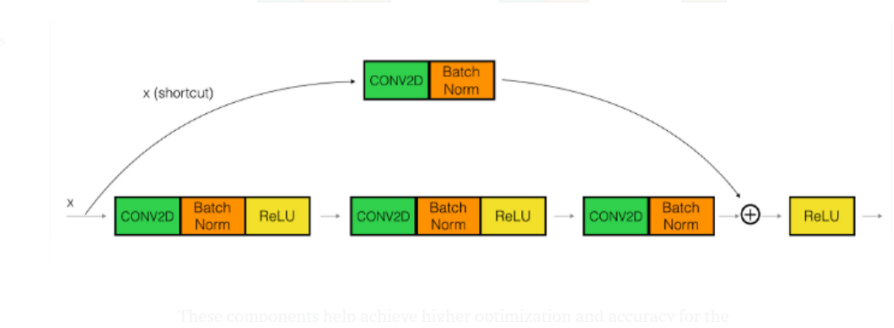


Figure 2.7: New version of the residual block

In the residual block shown in figure 2.7, batch normalisation (batch norm) operation is included that normalises the feature maps with respect to each channel so as to improve the training process of the model. In this residual block a convolution operation is performed on the skip connection as well. This gives an option for the model to either learn features or to learn the identity mapping which improves the accuracy and the robustness of the model architecture.

3

Literature Survey

The goal of the literature survey is to choose a suitable dataset, an evaluation metric and a baseline architecture. This chapter explains about the survey done on the pose estimation datasets, evaluation metrics and model architectures.

3.1. Datasets

The first step in building a model involves choosing the dataset and evaluation metric for the problem to be solved. The proper choice of the dataset and the metric plays a significant role in improving the accuracy of the models and also generalises well for real world applications. For solving the problem of pose estimation, a lot of datasets have been developed and labeled. The datasets and their annotations are made publicly available to support the research in the domain.

For the literature study and further research only '2D Single-Person' pose estimation methods and datasets are considered. 2D Single-Person pose estimation datasets can be classified into image based and video based datasets. Image based datasets contain a large collection of single frame RGB images with humans and their corresponding keypoint annotations. While the video based datasets contain short videos with humans and each frame in the video has respective keypoint annotations. Both the image based and video based datasets can be further classified based on the annotations available into upper body [19–22] and full body based pose estimation datasets. Full body annotated datasets can be further classified into indoor based datasets [23–25] that are collected from indoor settings, sport activities based datasets [26–28] that contain humans performing different sport activities and diverse activity datasets [14, 29–31] that contain people performing different activities. Different categories of the pose estimation datasets can be seen in flow chart 3.1.

3.1.1. MPII Human Pose Dataset

Among the available datasets, MPII Pose [14] is one of the most widely used dataset for benchmark in the domain of pose estimation and so it is chosen as the suitable dataset for this research. The dataset, collected and annotated by the Max Plank Institute was released in 2014 and contains 26,429 images of single person performing 491 different activities. Each image has 16 manually annotated keypoints.

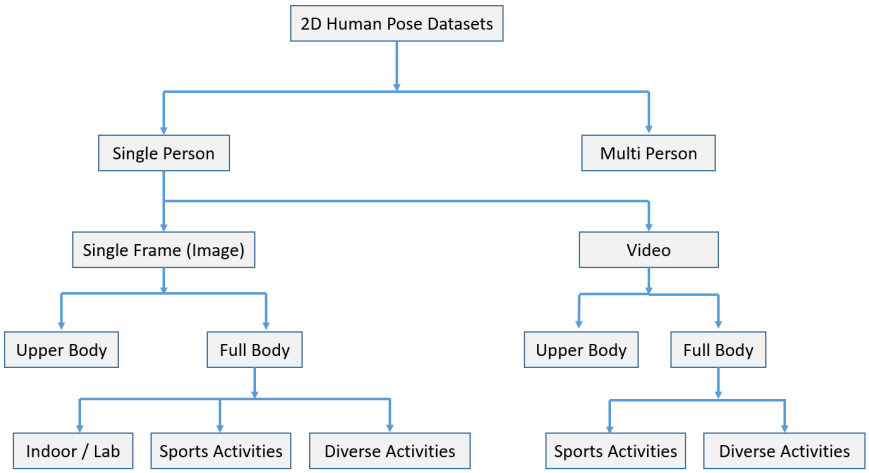


Figure 3.1: 2D Pose Estimation Datasets Classification Chart

3.2. Evaluation Metrics

Metrics for human pose estimation play a vital role in evaluating various methods by quantifying their performance on a chosen dataset. The performance measurement also helps in the comparison of different models used for the task of pose estimation. For pose estimation, the logic behind the metrics is whether or not the model has correctly estimated the keypoint. In order to judge whether the estimated keypoint is correct, metrics like Percentage of Correct Parts(PCP)[22], Percentage of Detected Joints(PDJ)[8], Percentage of Correct Key-points(PCK)[32] and Percentage of Correct Keypoints based on head segment length(PCK-h)[14] are used.

3.2.1. Percentage Of Correct Keypoints Based On Head Segment Length (PCK-h)

The most suitable evaluation metric for the research is the PCK-h metric as it alleviates all the drawbacks of the other metrics. It considers a detected keypoint to be correct if it lies within a radius equal to the fraction (usually 0.5) of the ground truth head segment length of the person. During training and evaluation, PCK-h value is obtained for each keypoint and the accuracy of the model can be measured by the mean of the PCK-h values.

3.3. Pose Estimation Methodologies

Human pose estimation methods can be classified into generative, discriminative and hybrid methods based on how the human skeleton is interpreted by them. Generative methods [32–35] model the human body as a kinematic tree and learn the parameters of different pose configurations with constraints, from the labelled images. The methods estimate a pose by choosing one of the learned configuration that best matches with the image. Discriminative methods[36–38] on the other hand learn a function that maps the input image to the output pose directly. Hybrid methods[39] combine both generative and discriminative methods to estimate the human body pose.

3.4. Deep-Learning Based Methods

Even though the deep learning based methods that use convolutional neural networks(CNNs) for the task of human pose estimation fall under the category of discriminative methods, they are discussed in a separate section because of the extensive research that has been carried out on these

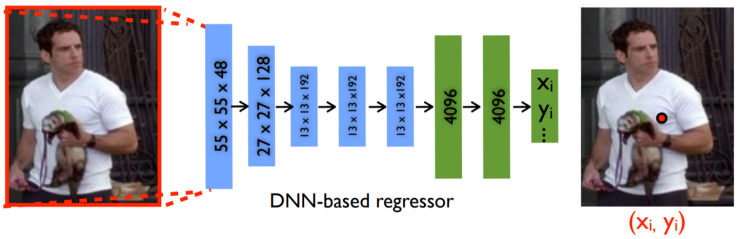


Figure 3.2: Initial stage of the regression network proposed in [8]

type of methods and also because of the fact that these deep learning based methods outperform other methods by a huge margin. On contrary to the use of hand crafted features used by the other discriminative methods, deep learning based methods extract features from images using CNN architectures that are more effective for the task of pose estimation. The models then perform various operations on the feature maps to estimate the keypoint positions. During the process of training the deep learning based model, the parameters of the CNN architecture are updated using a process called backpropagation[40] for CNN that uses gradient descent approach to minimise the cost. Backpropagation updates the weights in such a way that it minimises the loss function. For the task of human pose estimation, most widely used loss function is the mean squared error loss function. Depending on the way in which the keypoints are predicted, deep learning based 2D single-person human pose estimation methods can be classified into 'direct regression' based methods and 'heatmap' based methods. They are elaborated in the following subsections.

3.4.1. Direct Regression Based Methods

Direct regression based methods take the image as input, extract the features using a CNN based architecture and finally regress the body joint locations from the features. Deeppose[8] was the first deep learning model that used CNN based regression to map the input image to keypoint locations. In the initial stage the model produces a rough estimate of the keypoint locations. From the obtained rough estimates, the image is cropped and rescaled with each estimated keypoint location and the cropped images are fed as input to the next CNN based regression stage. Multiple stages are used to improve the accuracy of the initially estimated keypoint locations. The first stage of [8] is shown in Figure 3.2. Instead of using a single image as input, the model proposed in[41] uses a sequence of images from video as input and regresses the keypoint locations. An iterative error feedback model[42] was proposed which regresses the keypoint locations from input image in multiple stages. After the first stage the input to the second stage contains both the image and the predictions from the previous stage and this process is carried out recursively so as to improve the accuracy. To generalise regression based methods for both 2D and 3D human pose estimation, [43] proposed a model that represents human pose with bone structures instead of keypoints. A multi-task framework model that first detects body parts using sliding window and then regresses the keypoint locations was introduced in [44].

Though these deep learning based regression methods for human pose estimation are better compared to the discriminative methods, mapping an image directly to keypoint locations is very challenging for the model to learn as it is highly non-linear. The keypoint can lie anywhere in an image and so a very complex model is necessary to directly regress the keypoint location from the image features.

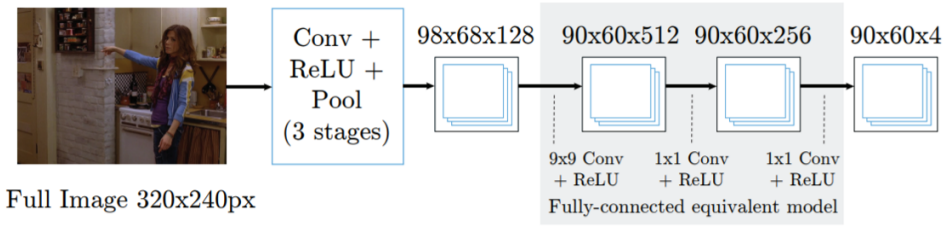


Figure 3.3: Architecture of a heatmap based deep learning method [9]

3.4.2. Heatmap Based Methods

Heatmap based methods generate feature maps that resemble a heatmap for each keypoint location. Then the keypoint locations are obtained from the generated heatmaps. Heatmap based method for the task of human pose estimation was first proposed in[45]. The pose estimation model proposed in[45], first uses a CNN architecture and produces a heatmaps based output from input monocular image. The output heatmap is a lower resolution 2D feature map of the input image, with one 2D map for each keypoint location. In each of the n heatmaps for n keypoints, the coordinate location with the highest value represents the coordinate location of the corresponding keypoint location.Intuitively this could be grasped in a way that, each value in the heatmap represents the probability that the keypoint lies in that location. So the highest value means that locations seems to have the highest probability for the corresponding keypoint to lie in that location of the image. A sample architecture of a heatmap based method can be visualised from Figure 3.3. The low resolution heatmap is then rescaled to the original image size and the coordinates are extracted which are the pixel location of the body joints.

Similar to the regression based methods, the heatmap based pose estimation methods like [9] also use a predefined backbone CNN architecture. However, heatmap based methods perform well on the task of human pose estimation and outperform the regression based methods with their robustness, the research has shifted towards heatmap based techniques. Though heatmap based methods performed better than regression based methods in the task of human pose estimation, they suffered from two issues. The first one was because of the loss of the spatial information by the use of maxpooling layers[40] in the CNN architectures. Maxpooling layers reduce the size of the feature maps by taking the maximum value from a window and by doing so they not only reduce the computational cost but also make the model invariant to the position of the objects in the image. Though max pooling layers have the advantages as mentioned earlier they lose the spacial information which is very much important for the task of human pose estimation as the task needs pixel wise spacial information to properly estimate the keypoint locations in an image. The second issue with heatmap based methods is that they cannot be trained as an end-to-end model. This drawback rises as the heatmap based methods use an argmax(returns the location of maximum value from an array) function to extract the keypoints from the heatmaps and this step cannot be included in the training procedure as the argmax function is not differentiable. Also the labeled data doesn't contain the output labels in the form of heatmaps but as vectors with the image pixel locations for all the keypoints. So to train heatmap based pose estimation models, target heatmap should be created so as to calculate the mean-squared error loss between the predicted output and target heatmaps. The target heatmaps are created from the pixel location coordinates by fitting a Gaussian with the mean as the location of the keypoint and a small standard deviation as in Figure 3.4.

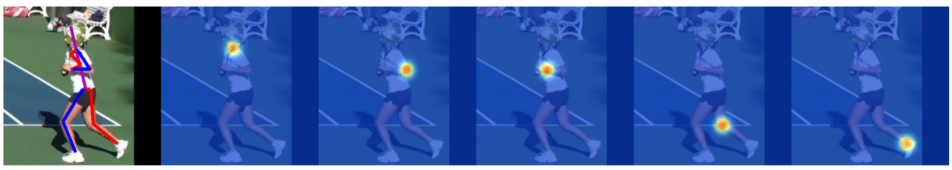


Figure 3.4: Example heatmap output produced by a network [10]

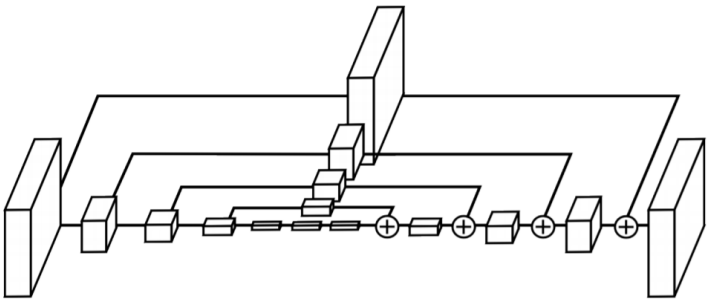


Figure 3.5: A single Hourglass pipeline [10]

3.4.2.1. Convolutional Encoder-Decoder Architectures

To overcome the issue of losing the spacial information because of the max pooling layer in the CNN architecture, Newell et al.[10] proposed a convolutional encoder-decoder based architecture for the task of pose estimation. The proposed architecture that looks like an hourglass, first uses convolution and pooling layers to reduce the resolution of the feature map and then uses upsampling layers to increase the resolution of the feature maps as shown in Figure 3.5. The information from the encoder pipeline is added to the decoder pipeline to regain the spacial information that is lost because of the pooling layers. The output the first hourglass block is then fed as an input to the next block that resembles the first. The final output of the model[10] is a keypoints heatmap that has the same resolution as that of the input image. Due to its robustness, the hourglass model[10] achieved a very high PCK-h value and outperformed all the previous heatmap based models by a huge margin on the MPII benchmark [14]

The advent of hourglass model has been a turning point in the research path of 2D single person pose estimation methods. The models[46–50], that were developed later to achieve a high PCK-h value on the MPII benchmark [14], use an convolutional encoder-decoder architecture that resembles the hourglass model[10]. The current state of the art method [51] that has the highest mean PCK-h value of 93.7 on the MPII[14] dataset also uses an complex convolutional encoder-decoder pipeline. The pose estimation methods that use an architecture similar to the hourglass model and their PCK-h value on the MPII benchmark [14] are listed in table 3.1

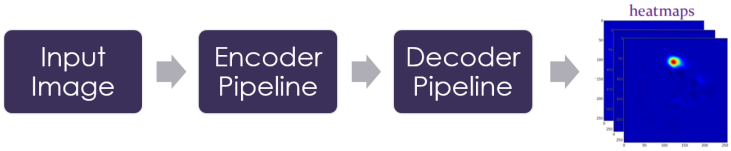


Figure 3.6: Promising baseline architecture for 2D single-person pose estimation.

Method	PCK-h
(Chu et al., 2017) [46]	91.5
(Yang et al., 2017) [47]	92
(Ke et al., 2018) [50]	92.1
(Tang et al., 2018) [48]	92.3
(Tang and Wu, 2019) [49]	92.7
(Zhihui et al., 2019) [51]	93.7

Table 3.1: Pose estimation methods that use convolutional encoder-decoder architecture similar to hourglass[10] model and their PCK-h values on the MPII benchmark dataset[14]

The most widely used baseline architecture in the pose estimation methods proposed after the advent of stacked hourglass model [10], first extracts the feature maps by downsampling the input image resolution in a series of CNN layers and then uses a sequence of upsampling techniques to increase the resolution to finally obtain the keypoints heatmap.

3.5. The Baseline Architecture

From the survey on literature, it is clear that the most promising baseline architecture for the task of 2D single-person pose estimation is a convolutional encoder-decoder type architecture that has two pipelines as shown in Figure 3.6. The encoder pipeline consists of a backbone CNN architecture that extracts the feature maps from the input image by using convolution and pooling layers. The decoder pipeline uses upsampling operations to increase the resolution of the feature maps and finally produces the output heatmap. During the process of training, the mean squared error is computed between the predicted heatmap and the target heatmap and gradient descent method to update the parameters of the architecture.

3.5.1. Encoder Pipeline Architecture

The encoder pipeline consists of a backbone CNN architecture that extracts the feature maps form the input image. There are many CNN architectures like AlexNet [52], GoogleNet [53], VGGNet [54] and the most widely used contemporary CNN architecture is the ResNet [7] model because of its robust performance in the ImageNet benchmark. It is the first preferred and 'go to' architecture by many computer vision scientists. Also the ResNet architecture is proven to extract very useful features from the images due to the presence of skip connections in the architecture. The skip connections aid the architecture to have a large number of layers and doesn't suffer from the problem of vanishing gradients. Table 3.2 shows the test error of various CNN architectures on the ImageNet benchmark.

From the table it is clear that ResNet-50 architecture [7] is good at extracting useful features from the images. Also the operations used in the architecture are not very complex which makes it a suitable for being converted into android model.

method	top-5 err.
VGG	8.43
GoogLeNet	7.89
VGG v5	7.1
PReLU-net	5.71
BN-inception	5.81
ResNet-34	5.71
ResNet-50	5.25

Table 3.2: Test error of CNN architectures on imagenet benchmark

3.5.2. Decoder Pipeline Architecture

The decoder pipeline gets the feature maps that are of small resolution compared to the image fed as input to the encoder pipeline. The decoder pipeline then performs upsampling to increase the resolution of the feature maps and finally outputs the predicted keypoint heatmaps. There are different methods of upsampling operations that are performed in a sequence of layers.

The decoder pipeline used in the stacked hourglass model [10] for pose estimation uses nearest neighbour upsampling and then adds the upsampled feature maps with its counterpart feature maps from the encoder pipeline as shown in Figure 3.5. The decoder pipeline used in [10] is symmetrical to its encoder pipeline. The upsampling method proposed in [55] initially applies an unpooling operation by inserting sufficient number of rows and columns filled with zeros to the small scaled feature map and then applies a normal convolution operation to get an upsampled feature map. This method is called as deconvolution or transposed convolution and is the most widely used efficient decoder pipeline. The deconvolutional layers are efficient and robust because the parameters of the network are learned by the model during training. Also the operations in deconvolutional layers are simple to be implemented on a mobile device which makes it the best choice among other decoder pipeline architectures.

The suitable baseline model architecture for the task of 2D single-person pose estimation, to implement it on mobile device is inspired from the model proposed in [11]. The architecture can be visualised in the figure 3.7.

Figure 3.7: Pose estimation baseline model architecture [11]

In the image 3.7, the initial downsampling layers represent the encoder pipeline for which ResNet-50 architecture is used and for the decoder pipeline represented in orange, deconvolutional layers are used as explained in [11]

4

Model Compression

The objective of the research is to check whether the structured Bayesian deep compression algorithm [13] can successfully compress the baseline pose estimation model and reduce the inference time. In this chapter, the concepts necessary to intuitively understand the Bayesian compression algorithm, the methodology and the steps to engineer the model is explained in detail.

4.1. Regularization

One of the very common issues faced while building a deep learning model is 'overfitting'. Overfitting occurs when the model adapts and fits well on the training data but fails to generalise well on unseen data. This might lead to a poor accuracy on the test set but a good accuracy on the training set. Overfitting may be caused when there is not enough data to train the model or when the model complexity is high. It is very hard to build a perfect model for any task and so in many cases a slightly complex model is built and it is regularized by adding some constraints. As the name suggests 'regularization ' is the method used to implement this behaviour. regularization [56] reduces the model complexity by imposing a constraint on the model parameters such that the parameter values are as low as possible while also achieving a minimum loss.

Let us look at an example of how regularization can reduce overfitting on a simple regression based problem. Let the independent variable be X, target variable be Y and the coefficient/parameters be β. The relation for a polynomial of degree 1 looks like,

$$Y \approx \beta_0 + \beta_1 X_1 + \beta_2 X_2 + \ldots + \beta_p X_p \tag{4.1}$$

A loss function is necessary to fit a model for this problem. Let us consider residual sum of squares or RSS as the loss function for this problem.

$$\text{RSS} = \sum_{i=1}^{n} \left(y_i - \beta_0 - \sum_{j=1}^{p} \beta_j X_{ij} \right)^2 \tag{4.2}$$

The parameters of the function in (1) can be updated by using the gradients with respect to this loss function. The following figure shows the fitting curve for different degree of the polynomials.

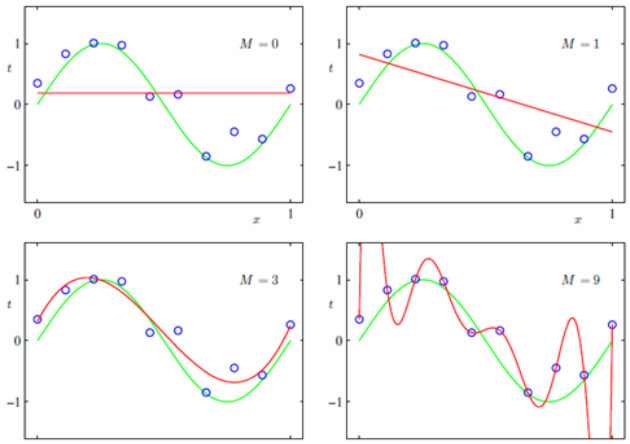


Figure 4.1: Fitting curves for different degree of polynomials [1]

In the figure 4.1 the green curve represents the optimal curve that generalises well on the data and the red curve represents the curve obtained for the corresponding polynomial degree M. As the degree of the polynomial increases the training data is fir properly but the model fails to generalise for unseen data and has a high variance. Also it is very difficult to choose the optimal degree of the polynomial. To tackle this issuer regularization helps to shrink the values of the parameters and so the curve will not tend to overfit even if the degree of the polynomial is high as the weights for those polynomials are contained to be a low value.

Based on the type of constraint equations added to the loss function, there are different types of regularization . Most widely used types of regularization are the L2 (ridge) and L1 (lasso) regularization .

4.1.1. L2 Regularization

In the case of L2 regularization [57], the extra constrain function is added to the loss function that restricts the value of the parameters such that they have a low value thereby reducing model complexity and overfitting. For our example regression problem the new loss function becomes,

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j X_{ij}\right)^2+\lambda\sum_{j=1}^{p}\beta_j^2=\text{RSS}+\lambda\sum_{j=1}^{p}\beta_j^2 \tag{4.3}$$

From the equation it can be seen that the objective of the model to reduce the loss can be achieved by reducing both the RSS loss and also the squared value of the parameters. This helps in imposing the constraint on the parameters to shrink their value. The λ is a hyper-parameter that decides the penalty on the model complexity. Higher the value of λ, less flexible/complex is the model and it generalises well on unseen data. So the proper choice of λ is very important and it is mostly found out empirically. If the value of λ is zero then there is no effect of regularization and if the value of λ tends to infinity then the parameters values tend to zero.

The following figure shows a visual representation of how L2 regularization works.

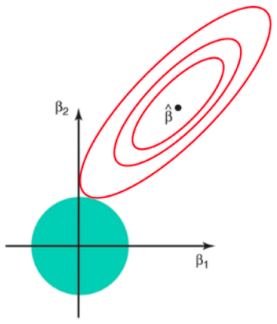


Figure 4.2: L2 regularization visualisation [12]

The green area in the figure 4.2 represents the function that imposes the constraint and the ellipses represent the contours of the RSS loss function. The optimal value for the parameters is the point of intersection of the contour and the green area. This regularization is named as L2 as it reduces the L2 norm of the weight vector.

4.1.2. L1 Regularization

L1 regularization [57] is similar to L2 regularization except a minor change to the constraint function. Instead of the imposing a loss using the squared value of the parameters as in L2, L1 regularization uses the absolute value of parameters as show in the equation below,

$$\sum_{i=1}^{n}\left(y_i-\beta_0-\sum_{j=1}^{p}\beta_j X_{ij}\right)^2+\lambda\sum_{j=1}^{p}|\beta_j|=\text{RSS}+\lambda\sum_{j=1}^{p}|\beta_j| \tag{4.4}$$

This regularization is named as L1 as it reduces the L1 norm of the weight vector. The following figure shows a visual representation of how L1 regularization works.

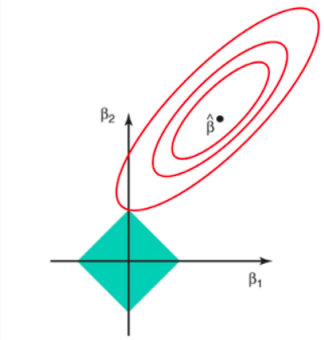


Figure 4.3: L1 regularization visualisation [12]

As explained in the previous subsection, the optimal value for the parameters is the point where the contour and the green area meet. The difference between the constraint functions of L1 and L2 regularization can be seen using the green area in figure 4.2 and 4.3 respectively.

4.2. Sparsity Using Regularization

As discussed in the earlier sections, it is very difficult to build a deep learning model that is perfect for a task but rather building a complex model and regularising it using some constraints is easy.

Intuitively regularization can be thought of as a mechanism that identifies the most redundant parameters that do not offer any contribution for the model in the task of prediction or classification, and makes them close to zero. The advantage of finding the redundant parameters and making them equal to zero helps to remove the neurons to which the parameters are connected as the value of activations from the neurons also become zero when multiplied with those parameters. Thereby, regularization helps in making the complex model architecture sparse by removing the redundant activations (neurons).

4.2.1. L1 Regularized Random Sparsity

In order to introduce sparsity in the model architecture L1 regularization is preferred over L2 regularization because L1 regularization has the capability of reducing the value of redundant weights to zero whereas L2 regularization cannot do that. This can be visualised from the images 4.2 and 4.3.

Random sparsity can be introduced by applying the constraint function as seen in equation (4.4) on each weight. Intuitively is like reducing the L1 norm of each weight. After performing the L1 regularization on each weight individually a threshold is applied to set the weight values below the threshold to zero. Since the weights set to zero are not in a structured arrangement this method only creates randomly sparse weight matrices. To reduce the model size, the sparse weight matrices of the trained model are stored in special data structures that contains the non-zero weight values and their corresponding weight locations. During inference the non-zero weight values are loaded into the model in their corresponding location while the other weight values are set to zero. This process has proven to reduce the model size but hasn't proven to save the inference time as the model architecture remains the same.

Figure 4.5 shows how the random sparse weight matrix looks like.

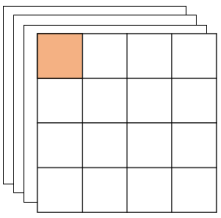


Figure 4.4: Random sparsity on the convolutional filter. Coloured square represents the redundant weight to be pruned. [13]

4.2.2. Group Lasso Regularized Sparsity

The random sparsity introduced by L1 regularization can only reduce the model size but not the inference time. To achieve reduction in both the model size and inference time structured sparsity has to be introduced which aids in removal of the weights and the corresponding feature map activations from the baseline architecture. The method to achieve this structured sparsity is proposed in [58].

Consider a convolutional weight with a dimension (N, C, W, H), where N is the number of filters, C is the number of channels, W is the width of the filter and H is the height of the filter. Let l denote a convolutional layer. The equation for structured sparsity as proposed in [58] is,

$$E(\boldsymbol{W}) = E_D(\boldsymbol{W}) + \lambda \cdot R(\boldsymbol{W}) + \lambda_g \cdot \sum_{l=1}^{L} R_g\left(\boldsymbol{W}^{(l)}\right) \tag{4.5}$$

Here W is the grouped weight, E_D is the loss incurred on the data, $R(\cdot)$ is the normal regu-

larization (L1 or L2) applied on each weight individually, $R_g(\cdot)$ is the group lasso regularization constraint that effectively zeros out the redundant weight groups [59]. For a weight group w^g, $R_g(\boldsymbol{w}) = \sum_{g=1}^{G} \left\| \boldsymbol{w}^{(g)} \right\|_g$ where G is the total number of groups. In the equation, $\| \cdot \| g$ is the group lasso which can be represented mathematically as $\left\| \boldsymbol{w}^{(g)} \right\|_g = \sqrt{\sum_{i=1}^{|\boldsymbol{w}^{(g)}|} \left(w_i^{(g)} \right)^2}$.

Figure 4.5 shows how the structured sparse weight matrix looks like.

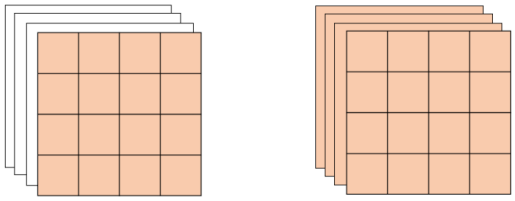


Figure 4.5: Structural sparsity on the convolutional filter. Coloured square represents the redundant weight to be pruned. [13]

4.3. Bayesian Learning

Bayesian learning method are a set of approaches in statistics that use Bayesian theorem to calculate the posterior probability of a hypothesis and to update it based on the new evidence. The model parameters are treated as random variables in Bayesian learning and are not deterministic. The prior knowledge on the distribution of parameter values forms the initial belief and the data points are the observations for which the likelihood has to be maximised. This subsection elaborates on the important concepts of Bayesian learning.

4.3.1. Bayes theorem

Bayes theorem can be used to calculate the posterior probability of an event if the likelihood and prior information are known. Bayes theorem can be expressed mathematically as in equation (4.6)

$$P(A \mid B) = \frac{P(B \mid A)P(A)}{P(B)} \tag{4.6}$$

where $P(A \mid B)$ is the posterior probability of the event A occurring given that event B has occurred, $P(B \mid A)$ is the likelihood of event B occurring given event A has occurred, $P(A)$ is the prior belief of the event A and $P(B)$ is the marginal probability of event B and it is also called as the normalising constant.

4.3.2. Bayesian Inference

Bayesian inference assumes that the parameters of a machine learning model are random variables that belong to a distribution and the posterior distribution of the parameters can be calculated using the Bayes theorem.

Let us assume that,

- W: Weights of the model.
- D: Data point or observation with independent and target variable X and Y

Therefore the posterior distribution of the parameters given the data points $P(W \mid D)$ can be calculated using the equation (4.7),

$$P(W \mid D) = \frac{P(D \mid W)P(W)}{P(D)} \tag{4.7}$$

where $P(D \mid W)$ is called as the likelihood of the data given the weights and the objective of the machine learning model is to maximise this likelihood, $P(W)$ is the initial belief also called as the prior distribution of the weights and the denominator $P(D)$ is the marginal distribution of the data points (observations). This equation updates the posterior distribution based on the observations and the prior information. Even though the equation looks simple, the term in the denominator which is also referred to as the normalising constant becomes intractable when the dimension of the data becomes high. This makes the problem of calculating the posterior probability distribution of parameters intractable.

The solution to this issue is to approximate the posterior distribution such that the approximate distribution is very close to the original distribution. There are different methods for Bayesian inference approximation like Monte Carlo approximation [60], Laplacian approximation [61], and variational inference approximation [62].

4.3.3. Inference Using Laplacian Approximation

Among all the available Bayesian approximation methods Laplacian approximation is the widely used method especially for neural networks as it is simple and can handle huge dimensions.

Let us assume an unnormalised probability density $P(x)$, with a normalising constant

$$Z_p = \int P(x)\,dx \tag{4.8}$$

let the peak of the distribution P(x) lie at x_0. Using Taylor's expansion,

$$\ln P(x) \simeq \ln P(x_0) - \frac{c_1}{2}(x - x_0) - \frac{c_2}{2}(x - x_0)^2 + \ldots \tag{4.9}$$

where,

$$c_1 = \frac{-\partial}{\partial x} \ln p(x)\bigg|_{x=x_0} = 0 \tag{4.10}$$

and so the equation for second order Taylor expansion becomes,

$$\ln P(x) \simeq \ln P(x_0) - \frac{c}{2}\left(x - x_0^2\right) \tag{4.11}$$

where,

$$c = \frac{-\partial^2}{\partial x^2} \ln p(x)\bigg|_{x=x_0} \tag{4.12}$$

The distribution $P(x)$ can be approximated by an unnormalised gaussian distribution $Q(x)$ using,

$$Q(x) = P(x_0)\, e^{\left[\frac{-c}{2}(x-x_0)^2\right]} \tag{4.13}$$

The normalising constant can also be approximated using

$$\mathcal{Z}_Q = P(x_0)\sqrt{\frac{2\pi}{c}} \tag{4.14}$$

This integral can be generalised for Z_p over K dimensional space using Hessian of the parameter matrix, A. The hessian matrix contains second order derivatives of the parameters and has the same dimension as that of the parameter matrix.

Thus the approximated distribution can be obtained using,

$$\ln P(x) \simeq \ln P(x_0) - \frac{1}{2}(x - x_0)^T A (t - x_0) \tag{4.15}$$

with the normalising constant,

$$Z_p \simeq Z_a = P(x_0)\sqrt{\frac{(2\pi)^k}{\det A}} \tag{4.16}$$

4.3.4. Structural Bayesian Deep Compression

After calculating the distribution of the parameters of the CNN using the laplacian approximation, their uncertainty needs to be calculated that gives a measure of how redundant the parameters are in the model. Parameters that have a low uncertainty or in other words high probability of being close to zero are the redundant parameters that can be safely set to zero in the model. Figure 4.6 helps to visualise how the convolution operation in CNN looks like with Bayesian approach.

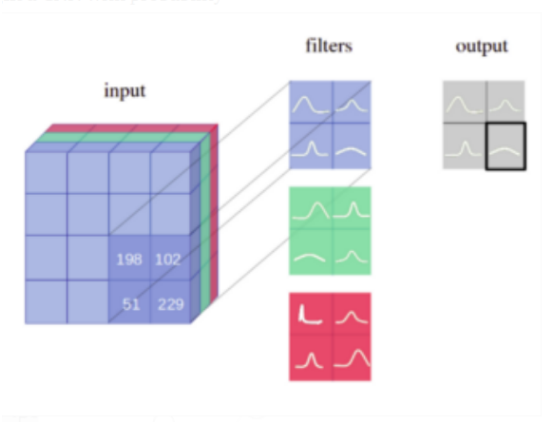


Figure 4.6: Weights as probability distributions

In order to calculate the uncertainty of the parameters the algorithm explained in the appendix B of [13] is used. One of the objectives of this research is to check whether the algorithm elaborated in [13] works for a human pose estimation model architecture.

Algorithm 1: Structural Bayesian Deep Compression

Initialization: $\forall l = 1,\ldots,L, \omega^l(0), \gamma^l(0) = 1; \lambda^l \in \mathbb{R}^+$;

Iteration ;

for $t = 1$ *to* $T_{\max}$ **do**

1. Maximum likelihood with regularization:

$$\min_{\mathscr{W}} E_D(\cdot) + \sum_{l=1}^{L} \lambda^l R\left(\omega^l(t) \circ \mathscr{W}^l\right)$$

2. Compute the Hessian for convolutional layers.
3. Update parameter values: Update$(\cdot)$ specifies how to update parameters as shown in Table 3.

$$\begin{aligned}
&\gamma^l(t) \leftarrow \text{Update}\left(\omega^l(t-1), \mathscr{W}^l(t)\right), \Gamma^l(t) = \left[\gamma^l(t)\right] \\
&C^l(t) \leftarrow \left(\left(\Gamma^l(t)\right)^{-1} + \text{diag}\left(H^l(t)\right)\right)^{-1} \\
&\alpha^l(t) \leftarrow -\frac{C^l(t)}{\gamma^l(t)^2} + \frac{1}{\gamma^l(t)} \{\text{ element-wise division }\} \\
&\omega^t(t) \leftarrow \text{Update}\left(\alpha^t(t)\right)
\end{aligned}$$

4. Prune the unimportant connections.

end

The algorithm uses reweighted group Lasso methods for solving the marginal likelihood $\int P(D \mid \mathscr{W}) P(\mathscr{W}) d\mathscr{W}$, maximisation problem. The initial belief distribution $P(\mathscr{W})$ is initialised as sparse priors over network weights as shown in Table 3. In the term for regularization each weight is scaled by ω. Hessian of the weights W is used to update the value of ω. The values of γ, C and ω are updated after every iteration. Here l represent the l-th convolutional layer in the architecture and α is used as a temporary variable to update ω. The summary of the pseudo code is shown in Algorithm 1.

Category	Sparse prior	$R\left(\omega^l \circ \mathscr{W}^l\right)$	ω^l	γ^l
Channel wise	$\prod_{c_l} \mathscr{N}\left(\mathbf{0}, \gamma_{c_l} \mathbf{I}_{n_l m_l k_l}\right)$	$\sum_{c_l=1}^{C_l} \left\|\omega^l_{:,c_l,:,:} \circ \mathscr{W}^l_{:,c_l,:,:}\right\|_2$	$\omega^l_o = \sqrt{\sum_{c_l} \left\|\alpha^l_{:,c_l,:,:}\right\|}$ $\omega^l_{:,c_l,:,:} = \omega^l_o \cdot \mathbf{I}^l_{:,c_l,:,:}$	$\gamma^l_o = \frac{\left\|\mathscr{W}^l_{:,c_l,:,:}\right\|_2}{\omega^l_{:,c_l,:,:}(t-1)}$ $\gamma^l_{:,c_l,:,:} = \gamma^l_o \cdot \mathbf{I}_{:,c_l,:,:}$
Filter wise	$\prod_{n_l} \mathscr{N}\left(\mathbf{0}, \gamma_{n_l} \mathbf{I}_{c_l m_l k_l}\right)$	$\sum_{n_l=1}^{N_l} \left\|\omega^l_{n_l,:,:,:} \circ \mathscr{W}^l_{n_l,:,:,:}\right\|_2$	$\omega^l_o = \sqrt{\sum_{n_l} \left\|\alpha^l_{n_l,:,:,:}\right\|}$ $\omega^l_{n_l,:,:,:} = \omega^l_o \cdot \mathbf{I}^l_{n_l,:,:,:}$	$\gamma^l_o = \frac{\left\|\mathscr{W}^l_{n_l,:,:,:}\right\|_2}{\omega^l_{n_l,:,:,:}(t-1)}$ $\gamma^l_{n_l,:,:,:} = \gamma^l_o \cdot \mathbf{I}^l_{n_l,:,:,:}$

Table 4.1: Sparse priors and update rule used in Algorithm 1 for channel and filter wise pruning.

The detailed derivation for all the equations mentioned in the algorithm and in the table are elaborated in [13]. I strongly recommend the readers to go through the paper [13] to get more detailed understanding on structural Bayesian deep compression.

4.4. Engineering The Sparse Model

The model trained using the structured Bayesian deep compression algorithm contains a lot of convolutional weights that are masked with zeros. These weights that are masked are found redundant by the algorithm based on their uncertainty and they can be safely removed from the model. This model with all the redundant weights set to zero is called as the sparse model. Since the weights are masked in a structured manner, the feature maps/activations that are connected to those redun-

dant weights can be removed from the model architecture. This is possible because the presence of those activations does not play any role in the prediction as their weights are zero. Hence the redundant weights and their corresponding activations/neurons can be removed from the model. This process is referred to as engineering the model architecture.

Engineering a model architecture is not easy as it sounds. It involves careful removal of the activations and weights from the model such that the accuracy of the model doesn't get affected by a significant amount. To understand how the model is engineered, let us consider a simple example. Let $L1$,$L2$ be 2 consecutive layers in a CNN architecture. Let the size of the convolutional filter between layers $L1$ and $L2$ be (N,C,W,H), where W and H represent the kernel width and height (similar in most cases), C represents the number of filter channels which should match the number of feature map channels in layer $L1$, N represents the number of filters which should match the number of feature map channels in layer $L2$. There are different scenarios to be handled when engineering the sparse model.

The first scenario is when the entire convolutional filter is redundant and only contains zeros. In this case the entire feature map in the layer $L1$ can be safely removed. However it should be made sure that the feature map size previous to layer $L1$ matches with the size in layer $L2$, after the removal of the feature maps from $L1$. The second scenario is when a few of the N filters are redundant. In this case, the filters along with the corresponding feature map channel in layer $L2$ can be safely removed. Again it should be noted that the removal of the feature map channel doesn't affect the dimension of the filters that are present after layer $L2$. The third scenario is when all a particular channel among C is redundant is redundant in all the N filters. In this case, the filters along with the corresponding feature map channel in layer $L1$ can be safely removed. Also, care must be taken to make sure that the dimensions of the filters and feature maps always match with that of the previous and next layer.

If during some scenarios when there is a mismatch between the dimensions of the feature maps between previous and the next layer, then a few feature maps has to be removed to match the dimensions. This process has to be done repeatedly with several feature map channels as the redundant channel cannot the identified. To validate the removal of a channel that is not redundant, corresponding filter or filter channel has to be masked with zero and the accuracy of the new masked model has to be validated. The channel can be safely removed only after making sure that there is only a very little drop in the accuracy (PCK-h). More importantly, if the CNN architecture contains shortcuts or skip connections, then care must be taken while adding the activations of 2 layers through a skip connection as the dimensions should match. If while engineering the model there is a mismatch in the dimension then enough zero channels should be padded so as to make the dimensions match.

The process explained above is explained with an example to get a proper intuitive understanding of how the model is engineered after introducing sparsity. It should be noted that the engineering process can be applied only when structural sparsity is introduced in the model and not random sparsity. Let us assume that the figure 4.7 represents the feature maps and convolutional filter in random layers of a CNN.

- Let the two consecutive CNN layers be n,$n+1$.
- The shape of 'feature map 1' in layer n is $(3xW_{f1}xH_{f1})$ where 3 is number of channels and W_{f1}, H_{f1} are width and height respectively.
- The shape of 'convolutional filter 1' is $(4x3xK_1xK_1)$ where 4 represents the number of output

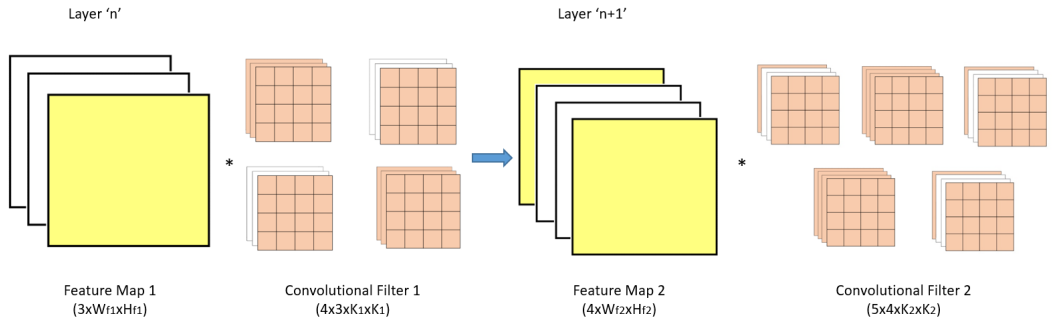


Figure 4.7: Engineering the model scenario 1

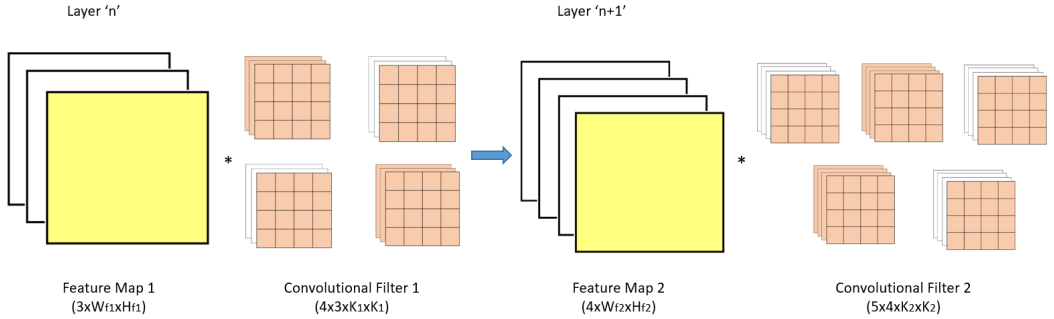


Figure 4.8: Engineering the model scenario 2

channels that is shown as the four separate filters, 3 represents the number of input channels that are stacked in each filter and K_1 represents the kernel size. The number of input channels of this convolutional filter matches the number of channels of 'feature map 1'.

- The shape of 'feature map 2' in layer $n+1$ is $(4xW_{f2}xH_{f2})$ where 4 is number of channels and W_{f2}, H_{f2} are width and height respectively. The number of channels of this feature map matches the number of output channels of the 'convolutional filter 1'.

- The shape of 'convolutional filter 2' is $(5x4xK_2xK_2)$ where 5 represents the number of output channels, 4 represents the number of input channels and K_2 represents the kernel size. The number of input channels of this convolutional filter matches the number of channels of 'feature map 2'.

- After structural sparsity is introduced in the model, the parameters of the convolutional filters that are found redundant are masked to zeros. These group masked filters channels are filled with orange color in the image 4.7. They represent the filter channels that can be safely removed from the model.

- Intuitively, the filter channels that are redundant show that the corresponding feature map channels are also redundant and can be removed from the model architecture. The corresponding channels of the redundant feature map activations that can be removed safely are filled with yellow color in image 4.7.

- However before removing the feature map channels, the dimensions of the layers have to be checked to make sure that they match even after removing the filter and feature map channels.

- As in the case shown in figure 4.7 the channels of the feature maps and the convolutional filters can be removed as the indices of the masked output channels of the convolutional filter between layers n and $n+1$ match with the indices of the input channels of 'convolutional filter 2'.
- However as shown in 4.8, even though the fourth channel of 'feature map 2' seems redundant with respect to masked 'convolutional filter 1', the corresponding input channels of masked 'convolutional filter 2' are not redundant. The removal of output channels form 'convolutional filter 1' removes the channels from the 'feature map 2', but the shape of the new feature map doesn't match with the shape of the 'convolutional filter 2' as only one input channel is removed from 'convolutional filter 2'.
- When such a mismatch happens the decision has to be made whether to remove the corresponding the filter channels even if the weights are non redundant. This decision is made empirically by forcefully setting the corresponding filter channel to zeros and then checking the model accuracy. The corresponding convolutional filter map channel is removed only if the loss in accuracy is very low.
- These steps are repeated recursively for every pair of consecutive layers and the corresponding redundant model parameters and feature map activations are pruned.

After engineering the model, a new architecture is built with the specific feature maps and convolutional filter shapes. The parameters of the compressed/engineered model are then loaded into this new compressed architecture during inference.

4.4.1. Fine-Tuning The Compressed model

The compressed model that is obtained by engineering the sparse model has to be fine-tuned to regain some amount of accuracy that was lost when engineering the model architecture. The process of fine-tuning involves retraining the model starting with the weights that were retained. This process ensures that the accuracy loss that occurs because of the removal of the redundant weights and activations is recovered, even if not fully, partially.

5

Experiments and Results

5.1. Dataset

The dataset chosen for the task 2D human pose estimation for this research is the MPII [14] human pose dataset. The open source dataset is collected, labelled and maintained by Max Plank Institute. It is one of the most cited benchmarks for the task and contains around 25k images. The images are collected from YouTube videos of people performing every day activities like walking, sports, dancing, cycling etc. The images are in 'jpg' format and the annotations are in the form of 'json' file in which the pixel location of the keypoints on a particular image is stored in the form of a vector. For every body joint a (x,y) pixel coordinate value is stored as the label. For every image the label vector has 16 (x,y) values corresponding to the indices "0 - right ankle, 1 - right knee, 2 - right hip, 3 - left hip, 4 - left knee, 5 - left ankle, 6 - pelvis, 7 - thorax, 8 - upper neck, 9 - head top, 10 - right wrist, 11 - right elbow, 12 - right shoulder, 13 - left shoulder, 14 - left elbow and 15 - left wrist" [14]. The label also contains the head segment length of the person in the image.

The dataset containing approximately 25k images are split into 22k images for training, 1k for validating and 2k for testing.

5.1.1. Preprocessing

The first step in preprocessing involved resizing the samples as the dataset contains images of different sizes and the keypoint position labels are with respect to the original size of each image. Every CNN architecture only accepts data samples with similar size. To tackle this problem, affine transformation was used to resize the images and their labels such that the input size is constant and the keypoint locations are with respect to this new input image size. All the images were reshaped to a size (256x256x3), where 3 corresponds to red, green and blue channels.

The second step involved subtracting the mean and dividing by the standard deviation for the red, green and blue channels of images. The images were initially divided by 255 to bring the values of the pixels from 0 to 1. Then the mean was subtracted from each channel and the result was divided by the standard deviation. This step was done to make sure that normalised data is given as an input to the model.

Then it was necessary to transform the keypoint location label vectors into heatmaps because as explained in the literature survey chapter a heatmap based regression method was chosen as the model architecture. The transformation was done such that the number of heatmap channels are equal to the number of keypoints (16 in our case) and the size is similar to the shape of the input

(256x256). So the shape of each heatmap created is of shape (256x256x16). For each keypoint location, a 2D Gaussian distribution with a standard deviation (σ) of 2 as suggested in [14] is fit with its mean at the location of the keypoint in the corresponding heatmap channel. The target heatmap can be visualised from the image 3.4.

Input image shape	256x256x3
Heatmap label shape	256x256x16
Mean of red channel	0.485
Mean of red channel	0.456
Mean of red channel	0.406
σ of red channel	0.229
σ of red channel	0.224
σ of red channel	0.225

Table 5.1: Parameters used for preprocessing step

The attributes of preprocessing steps are shown in table 5.1. The σ is the standard deviation with which the mean subtracted values.

5.2. Pose Estimation Model

The baseline model for this research consists of an encoder and a decoder (convolution-deconvolution) type architecture. The model was coded in the programming language 'Python' using the deep learning framework 'Pytorch'.

The encoder architecture is the standard ResNet-50 [7] model. The model architecture is made up of residual blocks and contains layers that perform convolution, pooling, batch normalisation and ReLU activation. All the layers are implemented using the standard functions available in the deep learning framework. The architecture contains 50 layers in total. The input shape of the image to the model is (256x256x3) which is the RGB image and the output shape of the feature map obtained from the encoder model is (8x8x2048).

The feature map output from the ResNet model serves as the input to the decoder architecture. The decoder model consists of five deconvolutional layers with batch normalisation and ReLU activation. The output feature map obtained from the decoder architecture is of shape (256x256x16). The number of layers and the filter shapes that were used were as recommended in the paper [11].

5.2.1. Loss Function

The loss to be minimised by the model is called as the 'Joints Mean Squared Error' loss that is calculated using the desired and output heatmaps. The equation for the loss is,

$$\mathscr{J}_{\text{mse}} = \frac{1}{K}\sum_{k=1}^{K} \|\boldsymbol{m}_k - \hat{\boldsymbol{m}}_k\|_2^2 \tag{5.1}$$

where m_k is the heat map for the keypoint k and K is the total number of keypoints. This loss can be intuitively interpreted as the mean of pixel wise squared difference loss between the predicted and original heatmap.

5.2.2. Model Training

The hyper-parameters used for training and validating the model are the same as used in the research paper [11]. Most important hyper-parameters used to train the model are listed in table 5.2.

Input image shape	256x256x3
Output heatmap shape	256x256x16
Number of epochs	140
Learning rate	0.001
Optimiser	SGD
Batch size	16

Table 5.2: Baseline model hyper-parameters

PCK-h is used as an evaluation metric to validate the model. It is calculated using the predicted and desired heatmap values along with the head segment length of the person in the image.

The parameters of the trained pose estimation model are stored in the form of a dictionary file that contains the layer names and their corresponding parameters. During inference this parameter dictionary is loaded into the model architecture and the predictions are made. The size of the dictionary file depicts the number of parameters used by the model architecture. The size of the file is referred to as the size of the model. The larger the model size means that the larger the number of parameters in the model and so greater is the inference time.

The model size, number of parameters and the pck-h of the trained baseline model are listed in table 5.3.

Model Size	136.3 MB
Number of parameters	33.9 M
PCK-h	87.8

Table 5.3: Baseline model properties

5.3. L1 Regularized Sparsity

To introduce the sparsity in the model parameter tensors using L1 regularization , the L1 constraint function as explained in section 4.2.1 is added to the joints mean squared error loss function (5.1). The model is trained with this new loss function and the trained model parameters are stored in a dictionary file. On this L1 regularized model, a threshold is applied and the value of the parameters (convolutional weights/filters) that are below the threshold are set to zero using a mask. This model in which a few parameters are set to zeros is called as the sparse model. Since there is no preset value for the λ and threshold the model is trained with different values and the sparsity, PCK-h and model size are logged that are listed in table 5.4.

The sparsity value refers to the amount of the model parameters that can be retained from the baseline model. For example a sparsity value of 0.350 means that only 35% of the baseline model parameters are retained and other redundant weights are set to zero. In other words 65% of the weights are redundant in the baseline model and can be removed while only loosing a very little

Lambda (λ)	Threshold	Sparsity	PCK-h	Size in MB (theoretical)
1.00 E-07	0.01	0.243	75.28	33.1
	0.001	0.287	79.74	39.1
	0.0001	0.297	79.75	40.4
3.00 E-08	0.01	0.302	79.42	41.2
	0.001	0.325	79.49	44.3
	0.0001	0.343	79.50	46.8
1.00 E-08	0.01	0.351	82.41	47.8
	0.001	0.383	82.46	52.2
	0.0001	0.394	82.46	53.7
5.00 E-09	**0.01**	**0.405**	**83.80**	**55.2**
	0.001	0.450	83.86	61.3
	0.0001	0.455	83.96	62.1
3.00 E-09	0.01	0.466	84.983	63.5
	0.001	0.489	85.066	66.7
	0.0001	0.494	85.167	67.3
1.00 E-09	0.01	0.501	85.592	68.2
	0.001	0.539	85.61	73.4
	0.0001	0.546	85.62	74.4

Table 5.4: Theoretical model size and PCK-h value comparison for L1 regularized sparse models.

accuracy. Even though the redundant weights of the sparse model are zeros, the size of the model remains the same as the baseline model because the weights are not yet completely removed. However the theoretical size of the model is calculated using the sparsity and the original baseline model size as shown in table 5.4.

To actually remove the model parameters, the steps explained in section 4.3.5 need to be followed. The model in which all the redundant parameters are removed is called as the compressed model. After training several sparse model by implementing L1 regularization on the baseline human pose estimation model, the best sparse model is chosen based on a trade-off between the model size and PCK-h value. This sparse model cannot be engineered as L1 regularization only induces random sparsity on the parameters as shown in image 4.5. Even though the size of the model can be reduced by storing the non-redundant weights and their locations in the architecture on a specific data structure, this doesn't help in reducing the inference time. Due to this drawback the L1 regularized sparse models are not engineered.

5.4. Group Lasso Regularized Structural Sparsity

Group lasso regularized structural sparsity is implemented on the baseline pose estimation model. The E_D in equation (4.5) is replaced by the MSE loss in equation (5.1). The parameters are grouped channel wise and the regularization is applied. The parameter tensors are grouped channel wise into single vectors and the lasso regularization is applied on those vectors. The lasso norm of the vector is reduced close to zero if that particular channel in the filter is considered redundant. Again similar to steps mentioned in L1 regularization , a threshold is applied and the redundant values are set to zero. However in this case an entire channel of the filter or an entire filter is set to zero instead of an individual parameter value.

The theoretical model size and sparsity for sparse model obtained after implementing the group lasso regularization on the baseline model for different value of lambda and threshold are shown in Table 5.5

Lambda (λ)	Threshold	Sparsity	PCK-h	Size in MB (theoretical)
1.00 E-07	0.01	0.228	76.4	31.0
	0.001	0.287	76.63	39.1
	0.0001	0.301	76.82	41.1
1.00 E-08	0.01	0.311	80.57	42.3
	0.001	0.345	80.72	47.1
	0.0001	0.354	80.81	48.2
3.00 E-09	**0.01**	**0.362**	**83.12**	**49.3**
	0.001	0.389	83.67	53.1
	0.0001	0.416	83.92	56.7
1.00 E-09	0.01	0.404	84.97	55.1
	0.001	0.441	85.01	60.2
	0.0001	0.482	85.12	65.6
5.00 E-10	0.01	0.473	85.21	64.4
	0.001	0.521	85.24	71.1
	0.0001	0.546	85.26	74.4
3.00 E-10	0.01	0.564	86.23	76.8
	0.001	0.659	86.3	89.8
	0.0001	0.676	86.39	92.1

Table 5.5: Theoretical model size and PCK-h value comparison for group lasso regularized sparse models.

As seen from the table the best group lasso regularized sparse model is obtained for the λ value of $3.00E-9$ and a threshold of 0.01. To compress the sparse model the steps elaborated in sections 4.3.5 are followed. After removing the redundant parameters it is necessary to build a new architecture with CNN layers that match the dimensions of the compressed model parameters. Building this new architecture can be visualised removing the redundant neurons as their activations do not contribute to the model prediction. After modifying the model architecture and obtaining the compressed model, it is retrained to regain the accuracy that is lost during the compression. During retraining it must be noted that the weights of the compressed model are initialised instead of random initialisation that happens when training is done from scratch.

The model size and the PCK-h value of the group lasso regularized sparse model that has been engineered and retrained are shown in Table 5.6.

Model Size	58.4 MB
Number of parameters	14.5 M
PCK-h	84.22

Table 5.6: Properties of model compressed using group lasso regularization method

5.5. Structural Bayesian Deep Compression

The structural Bayesian deep algorithm [13] is implemented on the baseline pose estimation model. One of the objectives of this research is to check whether the algorithm can successfully compress the human pose estimation model. In the equation shown in algorithm 1, $minE_D(\cdot)$ is replaced with the joints MSE loss (5.1). The Bayesian compression algorithm is run for 20 outers and each outer consists of 100 epochs. After every outer the values are updated as shown in algorithm 1 and the parameters (filter values) are masked to zeros if their uncertainty (γ) is below a threshold. After the training process is complete a sparse model is obtained. The model size and sparsity for sparse model obtained after implementing the Bayesian structured compression on the baseline model for different value of lambda and threshold are shown in Table 5.7

Lambda (λ)	Threshold	Sparsity	PCK-h	Size (theoretical)
5.00 E-08	0.01	0.277	82.32	37.7
	0.001	0.298	82.39	40.6
	0.0001	0.311	82.43	42.3
1.00 E-08	**0.01**	**0.301**	**84.96**	**41.0**
	0.001	0.338	85.06	46.0
	0.0001	0.342	85.17	46.6
1.00 E-09	0.01	0.496	86.32	67.6
	0.001	0.532	86.38	72.5
	0.0001	0.541	86.42	73.7

Table 5.7: Theoretical model size and PCK-h value comparison on Bayesian sparse models.

As seen from the table the best group lasso regularized sparse model is obtained for the λ value of 1.00E − 8 and a threshold of 0.01. This sparse model is then engineered and fine-tuned using the steps elaborated in sections 4.3.5 and 4.3.6. The properties of the model after these steps are shown in Table 5.8.

Model Size	45.6 MB
Number of parameters	11.3 M
PCK-h	86.42

Table 5.8: Properties of model compressed using Bayesian structured deep compression method

The relation between PCK-h and sparsity of the models compressed using the Bayesian method is shown in figure 5.1. It can be seen that the relation is almost linear and it the PCK-h drops suddenly after the sparsity value 0.3 which means the important parameters and activations are lost when more than 30% of parameters are removed.

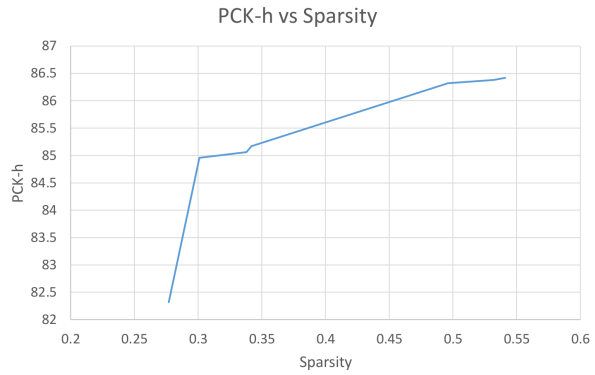


Figure 5.1: Relation between PCK-h and sparsity of the compressed models using the Bayesian method

5.5.1. Observation

From the table 5.8 it is clear that the Bayesian deep compression algorithm 1 can successfully compress a human pose estimation model architecture. The Bayesian method has reduced the model size from 136.3 MB to 45.6 MB and the number of parameters from 33.9 M to 11.3 M. Even though the model is compressed by almost one third of the original size the PCK-h has only dropped from 87.80 to 86.42.

Also it can be observed that even though the size of the models compressed after implementing the group lasso and L1 regularization are higher than the size of the model that is compressed using Bayesian structured compression [13], the PCK-h value is lesser than the later which shows that the Bayesian algorithm is good at finding the most redundant features compared to other methods as the uncertainty of the parameters are involved in the Bayesian method.

5.6. Mobile Application For Human Pose Estimation

To compare the inference time of the baseline model and the model compressed using the Bayesian structured deep compression algorithm, two mobile applications are developed. The applications are built on 'Android Studio' using 'Java' programming language. The baseline and the compressed pose estimation models that are trained using GPUs are converted into an android model using 'PyTorch Mobile'. PyTorch Mobile is a deep learning framework specially developed to deploy the deep learning models into smartphones.

The operations in the pose estimation model architectures are serialised using the 'JIT' module present in the 'PyTorch' framework. Since the 'Java' programming language can only accept single dimensional arrays, the heatmap tensor is converted into a 1D array using linear indices. This model is then loaded into the android application using 'Pytorch Mobile' framework.

The preprocessing steps necessary such as resizing and normalising as explained in section 5.1.1 are carried out in the mobile device using the 'OpenCV' library developed for android platform. After preprocessing the image the application loads the model and the prediction is carried out using the smartphone's CPU. The output is a 1D array with the pixel locations. The image along with the keypoint pixel locations obtained are then rescaled to the original image size.

The user interface of the application contains a button with an option to load any image from the smartphone's gallery as seen in image 5.2.(a). Once the image is loaded the pose estimation android model which is installed along with the application starts predicting the location of the keypoints

and outputs the single dimensional array. From the array the pixel locations are extracted, rescaled and the original image with the location of the predicted body joints marked with small green dots is displayed on the screen as shown in figure 5.2.(b).

The app also displays the inference time taken by the model to do the prediction on the image. This inference time excludes the time taken for the preprocessing and displaying the image as the focus of this research is only the inference time of the pose estimation model.

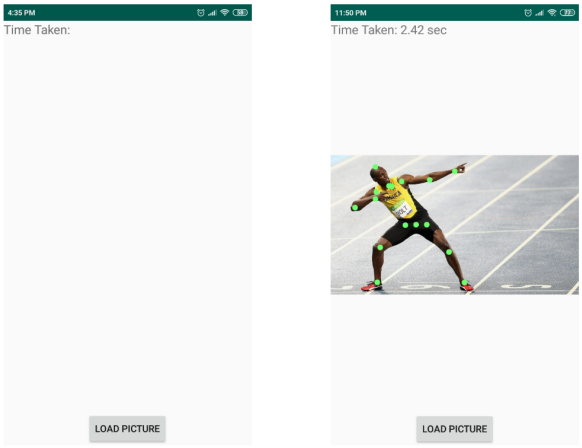


Figure 5.2: Mobile Application User Interface

The application size and the average inference time of the human pose estimation app developed with the baseline model and Bayesian compressed model are compared in table 5.9. The size and time are compared on Redmi MI Note 3 (2016) mobile device.

Model	App Size	Inference Time
Baseline	335 MB	2.37 sec/image
Compressed	146 MB	1.23 sec/image

Table 5.9: Inference time and app size for the baseline and compressed model android application

It can be seen from the table 5.9 that the size of the mobile application is much larger than the size of the model as shown in tables 5.3 and 5.8. This is because of the inclusion of other libraries like 'OpenCV', 'PyTorch Mobile', etc. during the installation of the application.

A few more images of the mobile application test cases are added in the appendix section. It can be seen that the model fails when the image resolution is high, when the human is not positioned in the centre of the image and when the human is performing a weird pose. This might be because the images in the dataset contains person at the centre of the image and there are no images with a contrast background and silhouette as seen in the 7.7. The code for developing the pose estimation android application using 'PyTorch Mobile' is also added in the appendix section of the report.

6

Conclusion

The objectives and the research questions have been addressed successfully. The baseline human pose estimation model architecture was chosen from the literature survey and was built and trained on a GPU. The Bayesian structural deep compression algorithm was implemented on the baseline human pose estimation model and the model was engineered to obtain a compressed model architecture. The compression algorithm successfully compressed the model by around 65 percent. Mobile applications were developed to compare the inference time of the baseline and the compressed model and it was found that the compressed model saved upto 50 percent of the inference time of the baseline model. In the future, the GPUs built in the smartphones can be used to run the model which can still reduce the inference time of the model. This research has proved that the Bayesian compression algorithm [13] can be used to reduce the model size and computational time for a 2D human pose estimation architecture. This compression method can be implemented on efficient architectures in the future to achieve a real-time inference on edge devices. To alleviate the failure of the model in estimating the pose on a particular type of images that are explained in section 5.6, in the future a task specific dataset can be used to train the human pose estimation model.

Bibliography

[1] Amy L. Bearman, Stanford, and Catherine Dong. Human pose estimation and activity classification using convolutional neural networks. 2015.

[2] S. Vieira, Walter H. L. Pinaya, and A. Mechelli. Using deep learning to investigate the neuroimaging correlates of psychiatric and neurological disorders: Methods and applications. *Neuroscience Biobehavioral Reviews*, 74:58–75, 2017.

[3] A comprehensive introduction to different types of convolutions in deep learning., `https://towardsdatascience.com/a-comprehensive-introduction-to-different-types-of-convolutions-in-deep-learning-6692`.

[4] Matthew D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *ECCV*, 2014.

[5] Introduction to Pooling Layer, `https://www.geeksforgeeks.org/cnn-introduction-to-pooling-layer/`.

[6] Vincent Dumoulin and Francesco Visin. A guide to convolution arithmetic for deep learning. *ArXiv e-prints*, mar 2016.

[7] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 770–778, 2016.

[8] Alexander Toshev and Christian Szegedy. Deeppose: Human pose estimation via deep neural networks. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1653–1660, 2014.

[9] Wei Yang, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. End-to-end learning of deformable mixture of parts and deep convolutional neural networks for human pose estimation. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 3073–3082, 2016.

[10] Alejandro Newell, Kaiyu Yang, and Jia Deng. Stacked hourglass networks for human pose estimation. In *ECCV*, 2016.

[11] Bin Xiao, Haiping Wu, and Yichen Wei. Simple baselines for human pose estimation and tracking. In *ECCV*, 2018.

[12] Jan Kukacka, V. Golkov, and D. Cremers. Regularization for deep learning: A taxonomy. *ArXiv*, abs/1710.10686, 2017.

[13] Hongpeng Zhou, M. Yang, J. Wang, and W. Pan. Bayesnas: A bayesian approach for neural architecture search. In *ICML*, 2019.

[14] Mykhaylo Andriluka, Leonid Pishchulin, Peter V. Gehler, and Bernt Schiele. 2d human pose estimation: New benchmark and state of the art analysis. *2014 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3686–3693, 2014.

[15] Chigozie Nwankpa, W. Ijomah, A. Gachagan, and S. Marshall. Activation functions: Comparison of trends in practice and research for deep learning. *ArXiv*, abs/1811.03378, 2018.

[16] Q. Wang, Yue Ma, Kun Zhao, and Y. Tian. A comprehensive survey of loss functions in machine learning. *Annals of Data Science*, pages 1–26, 2020.

[17] Y. L. Cun. A theoretical framework for back-propagation. 1988.

[18] Sebastian Ruder. An overview of gradient descent optimization algorithms. *ArXiv*, abs/1609.04747, 2016.

[19] Marcus Rohrbach, Sikandar Amin, Mykhaylo Andriluka, and Bernt Schiele. A database for fine grained activity detection of cooking activities. *2012 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1194–1201, 2012.

[20] James Charles, Tomas Pfister, Mark Everingham, and Andrew Zisserman. Automatic and efficient human pose estimation for sign language videos. *International Journal of Computer Vision*, 110:70–90, 2013.

[21] Benjamin Sapp and Ben Taskar. Modec: Multimodal decomposable models for human pose estimation. *2013 IEEE Conference on Computer Vision and Pattern Recognition*, pages 3674–3681, 2013.

[22] Vittorio Ferrari, Manuel J. Marín-Jiménez, and Andrew Zisserman. Progressive search space reduction for human pose estimation. *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[23] L. Sigal, A. Balan, and M. J. Black. HumanEva: Synchronized video and motion capture dataset and baseline algorithm for evaluation of articulated human motion. *International Journal of Computer Vision*, 87(1):4–27, March 2010.

[24] Catalin Ionescu, Dragos Papava, Vlad Olaru, and Cristian Sminchisescu. Human3.6m: Large scale datasets and predictive methods for 3d human sensing in natural environments. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36:1325–1339, 2014.

[25] Sergio Escalera, Xavier Baró, Jordi Gonzàlez, Miguel Ángel Bautista, Meysam Madadi, Miguel Reyes, Víctor Ponce-López, Hugo Jair Escalante, Jamie Shotton, and Isabelle Guyon. Chalearn looking at people challenge 2014: Dataset and results. In *ECCV Workshops*, 2014.

[26] Sam Johnson and Mark Everingham. Clustered pose and nonlinear appearance models for human pose estimation. In *BMVC*, 2010.

[27] Deva Ramanan. Learning to parse images of articulated bodies. In *NIPS*, 2006.

[28] Weiyu Zhang, Menglong Zhu, and Konstantinos G. Derpanis. From actemes to action: A strongly-supervised representation for detailed action understanding. *2013 IEEE International Conference on Computer Vision*, pages 2248–2255, 2013.

[29] Ke Gong, Xiaodan Liang, Dongyu Zhang, Xiaohui Shen, and Liang Lin. Look into person: Self-supervised structure-sensitive learning and a new benchmark for human parsing. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 6757–6765, 2017.

[30] Hueihan Jhuang, Juergen Gall, Silvia Zuffi, Cordelia Schmid, and Michael J. Black. Towards understanding action recognition. *2013 IEEE International Conference on Computer Vision*, pages 3192–3199, 2013.

[31] Lubomir D. Bourdev and Jitendra Malik. Poselets: Body part detectors trained using 3d human pose annotations. *2009 IEEE 12th International Conference on Computer Vision*, pages 1365–1372, 2009.

[32] Yi Yang and Deva Ramanan. Articulated human detection with flexible mixtures of parts. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 35:2878–2890, 2013.

[33] Pedro F. Felzenszwalb and Daniel P. Huttenlocher. Pictorial structures for object recognition. *International Journal of Computer Vision*, 61:55–79, 2004.

[34] Leonid Pishchulin, Mykhaylo Andriluka, Peter V. Gehler, and Bernt Schiele. Strong appearance and expressive spatial models for human pose estimation. *2013 IEEE International Conference on Computer Vision*, pages 3487–3494, 2013.

[35] Martin Kiefel and Peter V. Gehler. Human pose estimation with fields of parts. In *ECCV*, 2014.

[36] Liefeng Bo, Cristian Sminchisescu, Atul Kanaujia, and Dimitris N. Metaxas. Fast algorithms for large scale conditional 2d prediction. *2008 IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–8, 2008.

[37] Pedro F. Felzenszwalb, Ross B. Girshick, David A. McAllester, and Deva Ramanan. Object detection with discriminatively trained part based models. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 32:1627–1645, 2009.

[38] Farid Flitti, Mohammed Bennamoun, Du Q. Huynh, and Robyn A. Owens. Probabilistic human pose recovery from 2d images. *2010 IEEE International Conference on Image Processing*, pages 1517–1520, 2010.

[39] Rómer Rosales and Stan Sclaroff. Combining generative and discriminative models in a framework for articulated pose estimation. *International Journal of Computer Vision*, 67:251–276, 2006.

[40] Simon Haykin and Bart Kosko. Gradientbased learning applied to document recognition. 2001.

[41] Tomas Pfister, James Charles, and Andrew Zisserman. Flowing convnets for human pose estimation in videos. *2015 IEEE International Conference on Computer Vision (ICCV)*, pages 1913–1921, 2015.

[42] João Carreira, Pulkit Agrawal, Katerina Fragkiadaki, and Jitendra Malik. Human pose estimation with iterative error feedback. *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 4733–4742, 2016.

[43] Xiao Sun, Jiaxiang Shang, Shuang Liang, and Yichen Wei. Compositional human pose regression. *ArXiv*, abs/1704.00159, 2018.

[44] Sijin Li, Zhi-Qiang Liu, and Antoni B. Chan. Heterogeneous multi-task learning for human pose estimation with deep convolutional neural network. *International Journal of Computer Vision*, 113:19–36, 2014.

[45] Jonathan Tompson, Arjun Jain, Yann LeCun, and Christoph Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *NIPS*, 2014.

[46] Xiao Chu, Wei Yang, Wanli Ouyang, Cheng Ma, Alan L. Yuille, and Xiaogang Wang. Multi-context attention for human pose estimation. *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 5669–5678, 2017.

[47] Wei Yang, Shuang Li, Wanli Ouyang, Hongsheng Li, and Xiaogang Wang. Learning feature pyramids for human pose estimation. *2017 IEEE International Conference on Computer Vision (ICCV)*, pages 1290–1299, 2017.

[48] Wei Tang, Pei Yu, and Ying Wu. Deeply learned compositional models for human pose estimation. In *ECCV*, 2018.

[49] Wei Tang and Ying Wu. Does learning specific features for related parts help human pose estimation? *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1107–1116, 2019.

[50] Lipeng Ke, Ming-Ching Chang, Honggang Qi, and Siwei Lyu. Multi-scale structure-aware network for human pose estimation. *ArXiv*, abs/1803.09894, 2018.

[51] Zhihui Su, Ming Ye, Guohui Zhang, Lei Dai, and Jianda Sheng. Cascade feature aggregation for human pose estimation. *arXiv: Computer Vision and Pattern Recognition*, 2019.

[52] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. *ArXiv*, abs/1404.5997, 2014.

[53] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–9, 2015.

[54] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2015.

[55] Alexey Dosovitskiy, Jost Tobias Springenberg, and Thomas Brox. Learning to generate chairs with convolutional neural networks. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1538–1546, 2015.

[56] Benyamin Ghojogh and Mark Crowley. The theory behind overfitting, cross validation, regularization, bagging, and boosting: Tutorial. *ArXiv*, abs/1905.12787, 2019.

[57] Zongben Xu, Hai Zhang, Yao Wang, XiangYu Chang, and Yong Liang. L 1/2 regularization. *Science China Information Sciences*, 53(6):1159–1169, 2010.

[58] Wei Wen, Chunpeng Wu, Yandan Wang, Yiran Chen, and Hai Li. Learning structured sparsity in deep neural networks. *ArXiv*, abs/1608.03665, 2016.

[59] M. Yuan and Y. Lin. Model selection and estimation in regression with grouped variables 4 ming yuan and. 2004.

[60] Zoubin Ghahramani and Carl E Rasmussen. Bayesian monte carlo. In *Advances in neural information processing systems*, pages 505–512, 2003.

[61] J. S. Hsu. Generalized laplacian approximations in bayesian inference. *Canadian Journal of Statistics-revue Canadienne De Statistique*, 23:399–410, 1995.

[62] Manikanta Srikar Yellapragada and Konkimalla Chandra Prakash. Variational bayes: A report on approaches and applications. *ArXiv*, abs/1905.10744, 2019.

7

Appendix

7.0.1. Pose Estimation Android Application

In this section the steps to develop the pose estimation mobile application is explained.

- Install 'Android Studio'.
- Add the mobile version of the deep learning framework PyTorch and convert the model into an android model using instructions in the site 'PyTorch Mobile'
- Install OpenCV for android using the instructions in the video 'Installing OpenCV for Android'
- The code for the UI design of the application is as follows,

```
<?xml version="1.0" encoding="utf-8"?>
<LinearLayout

    xmlns:android="http://schemas.android.com/apk/res/android"
    android:orientation="vertical"
    android:layout_width="fill_parent"
    android:layout_height="fill_parent">

    <TextView
        android:id="@+id/text"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="@string/time_taken"
        android:textSize="18sp" />

    <ImageView
        android:id="@+id/image"
        android:layout_width="match_parent"
        android:layout_height="match_parent"
        android:layout_weight="1"
        android:contentDescription="TODO" />

    <Button
        android:id="@+id/button"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:layout_weight="0"
```

```
        android:text="@string/load_picture"
        android:layout_gravity="center"/>

</LinearLayout>
```

- The code for the application

```
package com.example.CompressedPose;

import java.io.File;
import java.io.FileOutputStream;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;

import android.content.Context;
import android.app.Activity;
import android.content.Intent;
import android.graphics.Bitmap;
import android.net.Uri;
import android.os.Bundle;
import android.provider.MediaStore;
import android.view.View;
import android.widget.Button;
import android.widget.ImageView;
import android.util.Log;
import android.widget.TextView;

import org.pytorch.IValue;
import org.pytorch.Module;
import org.pytorch.Tensor;
import org.pytorch.torchvision.TensorImageUtils;

import org.opencv.android.Utils;
import org.opencv.core.Mat;
import org.opencv.core.Size;
import org.opencv.core.Point;
import org.opencv.core.Scalar;
import org.opencv.imgproc.Imgproc;



public class MainActivity extends Activity {

    static {
        System.loadLibrary("opencv_java3");
    }


    ImageView imageView;
    Button button;
    private static final int PICK_IMAGE = 100;
    Uri imageUri;
    private Bitmap orig_bitmap;
```

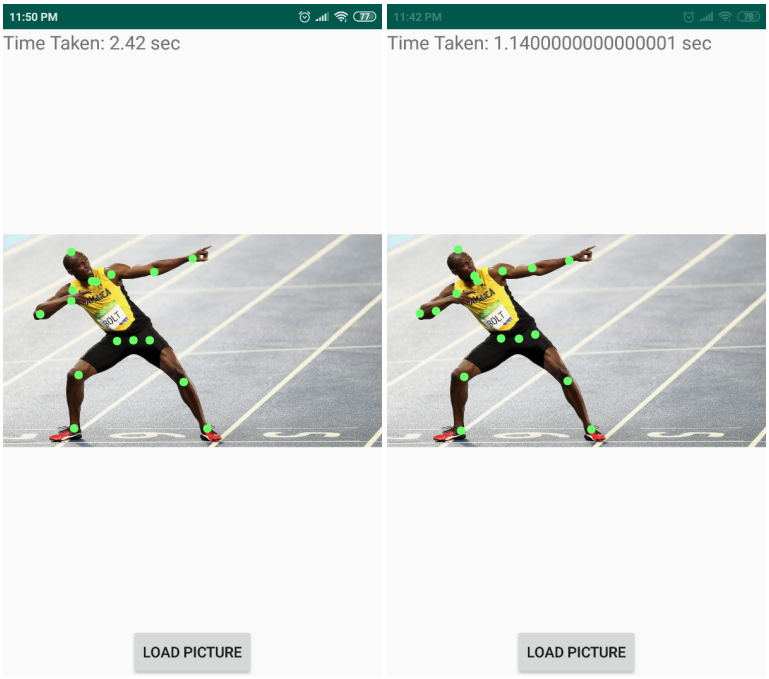
```
@Override
public void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_main);
    imageView = (ImageView)findViewById(R.id.image);
    button = (Button)findViewById(R.id.button);
    button.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View v) {
            openGallery();
        }
    });
}

private void openGallery() {
    Intent gallery = new Intent(Intent.ACTION_PICK,
        MediaStore.Images.Media.INTERNAL_CONTENT_URI);
    startActivityForResult(gallery, PICK_IMAGE);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent
    data){
    super.onActivityResult(requestCode, resultCode, data);
    if (resultCode == RESULT_OK && requestCode == PICK_IMAGE){
        imageUri = data.getData();
        try {
             orig_bitmap =
                 MediaStore.Images.Media.getBitmap(this.getContentResolver(),
                 imageUri);
        } catch (IOException e) {
            e.printStackTrace();
        }


        Module module = null;

        try {
            // loading serialized torchscript module from packaged into app
                android asset model.pt,
            // app/src/model/assets/model.pt
            module = Module.load(assetFilePath(this, "model.pt"));
        } catch (IOException e) {
            Log.e("PytorchHelloWorld", "Error reading assets", e);
            finish();
        }

        Mat orig_mat = new Mat();

        Utils.bitmapToMat(orig_bitmap,orig_mat);

        Size origSize = orig_mat.size();

        double origRows = origSize.height;
        double origCols = origSize.width;
```

```
            int rezRows = 256;
            int rezCols = 256;

            double scaleX = origCols/256;
            double scaleY = origRows/256;


            Mat rez_mat = new Mat();

            Size sz = new Size(rezRows,rezCols);

            Imgproc.resize(orig_mat, rez_mat, sz, 0, 0, Imgproc.INTER_AREA );

            Bitmap rez_bitmap = Bitmap.createBitmap(rez_mat.cols(),
                rez_mat.rows(),Bitmap.Config.ARGB_8888);
            Utils.matToBitmap(rez_mat, rez_bitmap);

            // preparing input tensor
            final Tensor inputTensor =
                TensorImageUtils.bitmapToFloat32Tensor(rez_bitmap,
                    TensorImageUtils.TORCHVISION_NORM_MEAN_RGB,
                        TensorImageUtils.TORCHVISION_NORM_STD_RGB);

            // running the model
            long startTime = System.nanoTime();
            final Tensor outputTensor =
                module.forward(IValue.from(inputTensor)).toTensor();
            long endTime  = System.nanoTime();
            long totalTime = (endTime - startTime)/10000000;

            final float[] scores = outputTensor.getDataAsFloatArray();


            for (int i = 0; i < 16; i++) {
                int x = (int)(Math.round(((scores[i]%256)*scaleX)));
                int y = (int)(Math.round(((scores[i]/256)*scaleY)));
                Imgproc.circle(orig_mat, new Point(x,y), 1, new Scalar(0, 255,
                    0, 150), 30);
            }

            Bitmap bm = Bitmap.createBitmap(orig_mat.cols(),
                orig_mat.rows(),Bitmap.Config.ARGB_8888);
            Utils.matToBitmap(orig_mat, bm);

            imageView = findViewById(R.id.image);
            imageView.setImageBitmap(bm);

            //Text
            double time_taken = (totalTime * 0.01);
            String print_time = "Time Taken: "+ time_taken +" sec";
            // showing time taken on UI
            TextView textView = findViewById(R.id.text);
            textView.setText(print_time);
        }
    }
```

```
    public static String assetFilePath(Context context, String assetName)
        throws IOException {
        File file = new File(context.getFilesDir(), assetName);
        if (file.exists() && file.length() > 0) {
            return file.getAbsolutePath();
        }

        try (InputStream is = context.getAssets().open(assetName)) {
            try (OutputStream os = new FileOutputStream(file)) {
                byte[] buffer = new byte[4 * 1024];
                int read;
                while ((read = is.read(buffer)) != -1) {
                    os.write(buffer, 0, read);
                }
                os.flush();
            }
            return file.getAbsolutePath();
        }
    }
}
```

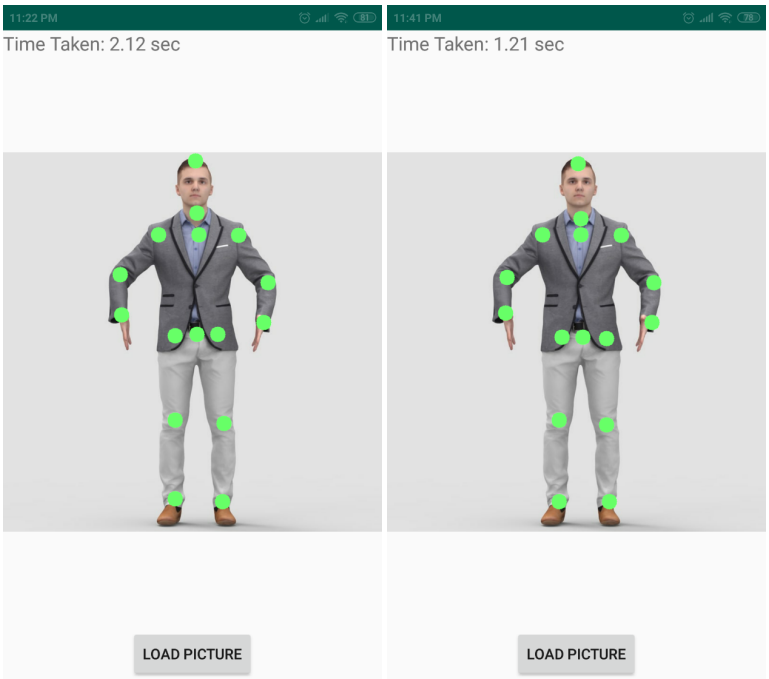
- The libraries and built in functions used in the code were majorly referred from the website 'Stack Overflow'.

7.0.2. Pose Estimation App Images



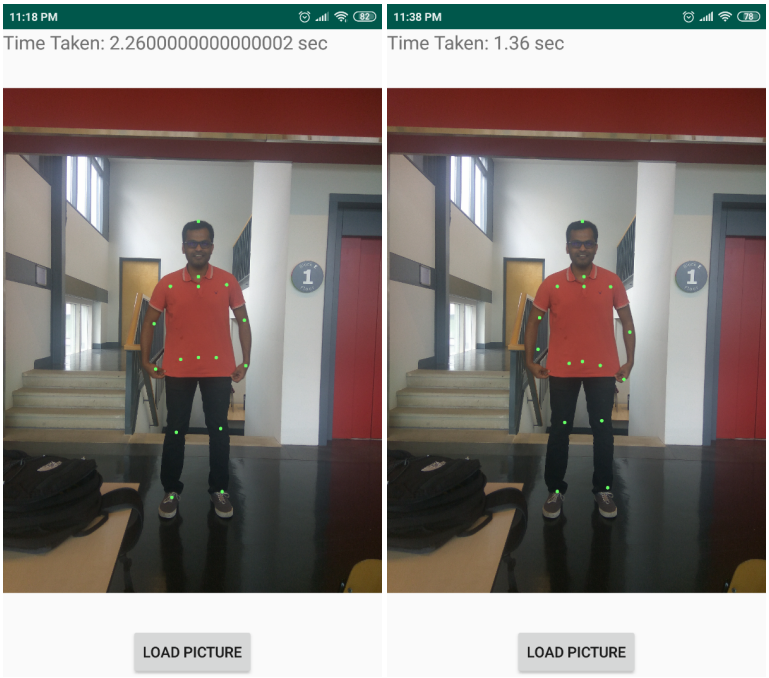
(a) Baseline model app (b) Compressed model app

Figure 7.1: App screenshot 1



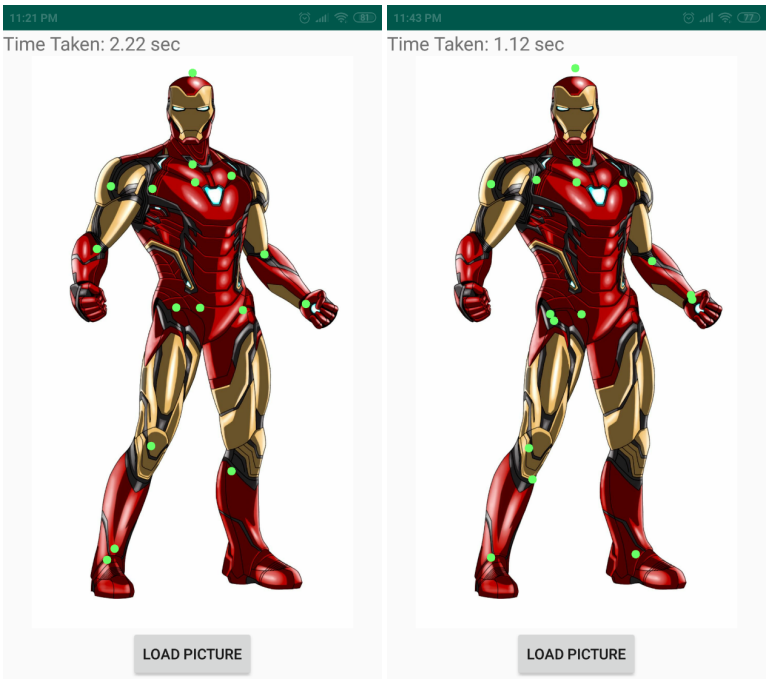
(a) Baseline model app (b) Compressed model app

Figure 7.2: App screenshot 2



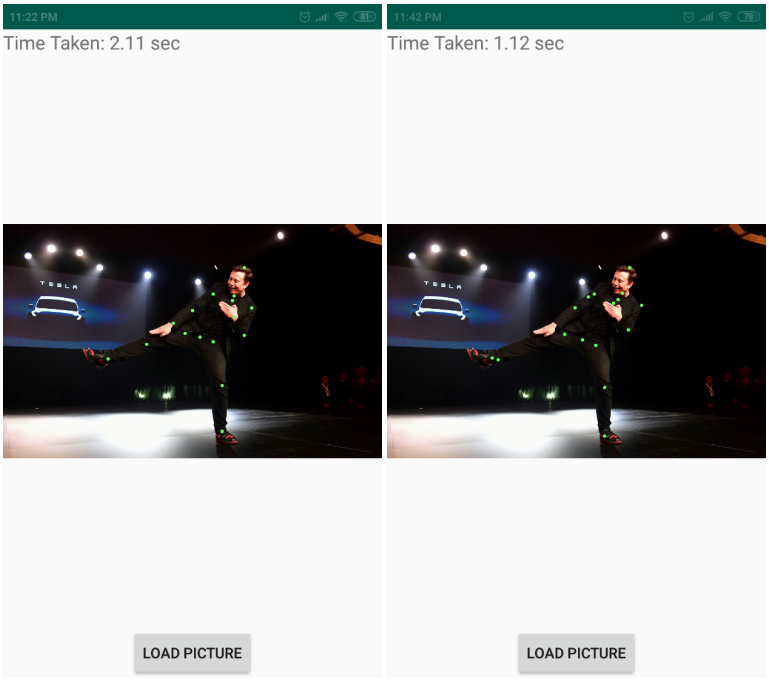
(a) Baseline model app (b) Compressed model app

Figure 7.3: App screenshot 3



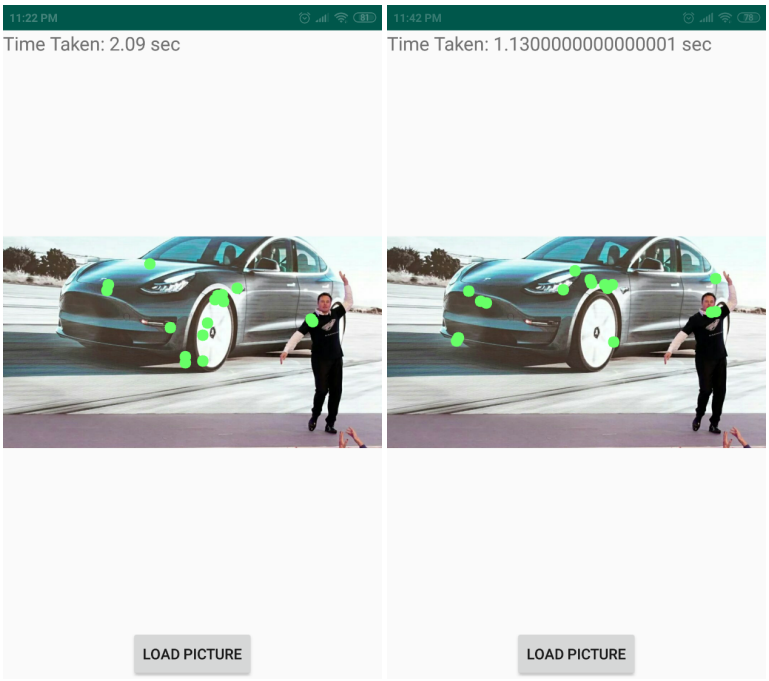
(a) Baseline model app (b) Compressed model app

Figure 7.4: App screenshot 4



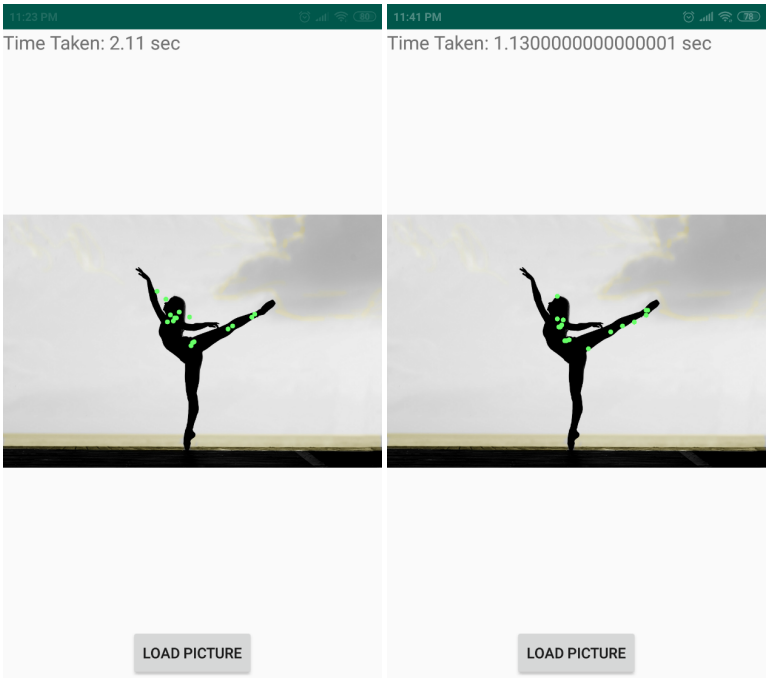
(a) Baseline model app (b) Compressed model app

Figure 7.5: App screenshot 5



(a) Baseline model app (b) Compressed model app

Figure 7.6: App screenshot 6



(a) Baseline model app (b) Compressed model app

Figure 7.7: App screenshot 7