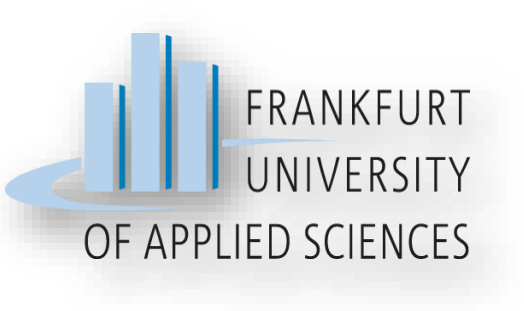


Vietnamese – German University

Electrical Engineering and Information Technology Program

HUMAN POSE ESTIMATION USING NEURAL NETWORKS

BY

Dinh Quang Vinh

Matriculation number: 1235201

BACHELOR THESIS

Submitted in partial fulfillment of the requirements for the degree of Bachelor Engineering in study program Electrical Engineering and Information Technology, Vietnamese - German University

First supervisor: Dr. Nguyen Minh Hien

Second supervisor: Mr. Bien Minh Tri

Binh Duong, Vietnam, 2022

© 2022 by Dinh Quang Vinh. All rights reserved.

Disclaimer

I hereby declare that this thesis is a product of my own work, unless otherwise referenced. I also proclaim that all opinions, results, recommendations and conclusion are mine and do not represent the policies or opinions of Vietnamese - German University.

Dinh Quang Vinh

Acknowledgement

Firstly, I would like to express my gratitude and appreciation to Dr. Nguyen Minh Hien, my first advisor. Not only did she introduce me to this topic at the very beginning of the Senior Project, but she also guided me throughout the creation of this thesis. Without her significant insight and knowledge in the topic of Deep Learning and specifically Neural Networks, I would not have finished my thesis in time. I am also grateful for her teaching and recommendation of materials that helps me bettering my understanding as well as my research skills for future projects.

Secondly, I want to thank Mr. Tri Bien Minh, lab engineer, for being my second advisor. His suggestions and assistance regarding the topic were essential for the making of this thesis.

Lastly, I also would like to pay tribute to my friends and family who have been there and motivated me from start to finish.

Abstract

With the rise of Virtual Reality (VR) and Augmented Reality (AR) applications in recent years, the Human Pose Estimation problem has also gained more recognition because understanding human pose is very critical to afore-mentioned technologies. Human pose can be tracked with motion capture (sometimes referred to as mo-cap or mocap) which utilizes various sensor and camera types to achieve real time accuracy. This technique, however, is very expensive and only produces a fraction of the data that needs to be studied. Furthermore, due to its usefulness and versatility, Human Pose Estimation finds its crucial role in fields such as gaming, healthcare, sports, etc. as well. The majority of human pose data is available in the form of videos and images which are captured by RGB cameras. Thus, the focal point of this thesis will be obtaining poses from RGB images.

Human Pose Estimation is not a new task to the Computer Vision field, it has in fact been researched and investigated throughout the last 15 years. The main goal of the task is to locate crucial points of a human body such as eyes, nose, shoulders, etc. It can either be conducted in 2D or 3D space. There are classical approaches and deep learning-based approaches to the problem. The latter approaches have gained significantly more traction thanks to the popularity of Convolution Neural Networks in image processing tasks.

This thesis aims to obtain a Neural Network model capable of properly detecting specific keypoints on human bodies in RGB images. It also details theoretical aspects, implementation, evaluations and proposals for future improvements.

Table of contents

Table of figures

Table of tables

Chapter 1: Introduction

1.1 Human Pose Estimation

1.1.1 Overview

Human Pose Estimation (HPE) aims to identify and classify certain joints on the human body. Joints, also referred to as keypoints, are a set of specific points of interest located on the human body. Keypoints may refer to face-keypoints (sometimes called landmarks), body-keypoints or even hand-keypoints. In this project, only body-keypoints are considered. Being able to extract these points precisely enables a deeper comprehension of human pose and the correlation between these body parts. Thus, HPE is heavily used and experimented in Action Recognition, Animation and any fields that require a better understanding of human pose.

Nonetheless, HPE is a very complicated computer vision task. This is because human pose in the wild is highly non-linear and unconstrained. A great proportion of data available for the task is in the form of 2D images and videos. The main challenge when working in a 2-dimensional domain is the lack of "depth" knowledge which makes some joints occluded from the frame. Furthermore, if there are more than one people in an image, they can be in different scales or sizes depending on the camera's point of view. Not only that, but people can also be in different environmental settings, lighting, clothing or activities. In traditional object detection task, a bounding box around the object is predicted, which is made of four strictly related and constrained points. In HPE, however, a point of interest or joint may be restricted within a local space, but it has no limitation within the global space, the human body. A good case in point is the human forearm, a wrist has to be in a certain distance and rotation with respect to the elbow, yet it can be anywhere or in any rotation with respect to the whole body.

Figure 1.1: Human pose examples in different activities [1].

One solution to give the images more depth information is using 3D or depth cameras such as Kinect from Microsoft [2]. This method obtains high accuracy, robust results and is very useful for AR/VR or 3D animation. However, what the majority of people have at home is a mobile phone or a laptop with 2D camera. Hence, this project focuses on detecting a single person from a 2D frame, making this technology accessible and working with what has already been available to everyone.

1.1.2 Difference between 2D and 3D Human Pose Estimation

The main difference between 2D and 3D HPE is the output. 2D Pose Estimation estimates the location of body joints in 2D space relative to input data and the location is represented with (x, y) coordinates for each keypoint. Meanwhile, 3D Pose Estimation produces (x, y, z) coordinates for each joint.

In spite of being able to use 3D input data, most methods used for 3D Pose Estimation only concentrate on 2D images or videos due to the abundance and availability of this type of data. Moreover, 2D human datasets are easily obtained while annotating 3D pose in images is time-consuming and manual labeling is not practical and expensive.

3D Human Pose Estimation is a more significant challenge than its 2D counterpart. However, the tradeoff is the prediction in 3D space from a 2D input, much like how the human brain perceives regular images.

1.1.3 Human body modeling

There are three main types of human models used for HPE in 2D and 3D planes.

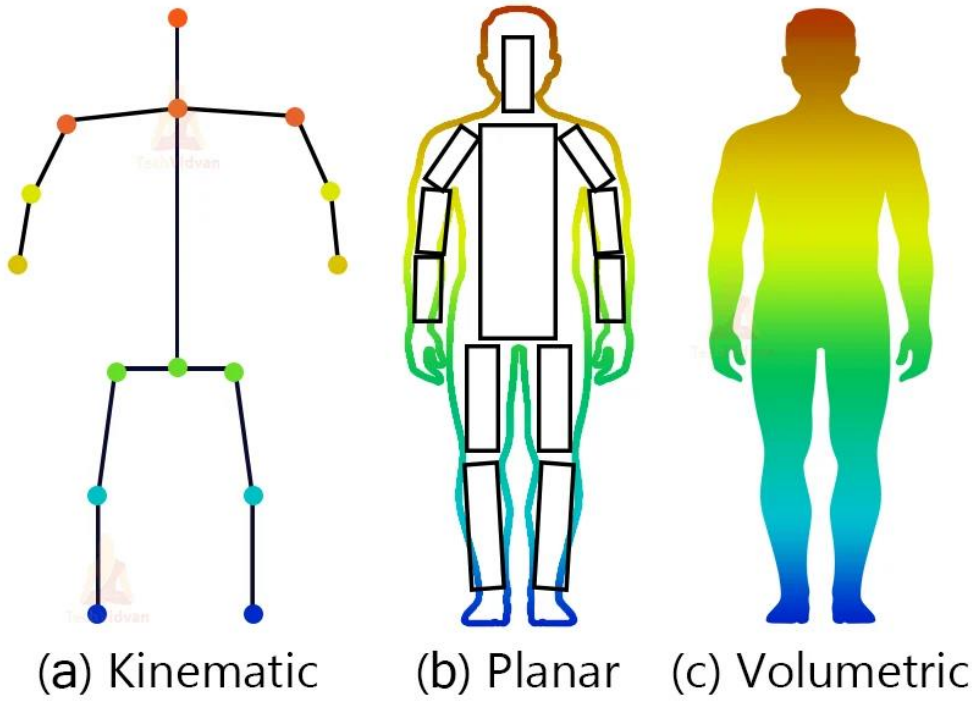


Figure 1.2: Types of human body models [3].

1.1.3.1 Kinematic

It is also known as skeleton-based model. This flexible and intuitive model uses a set of joint (keypoint) positions and limb orientations to represent human body architecture. This model is the most common and frequently applied to capture pose for both 2D and 3D tasks. However, it also has the drawback in depicting texture and shape information.

1.1.3.2 Planar model

Planar model, also known as contour-based model, is primarily used in 2D Pose Estimation. Body parts in planar model are usually represented by rectangles approximating the human contours. Therefore, unlike kinetic model, it can capture the information of shape and appearance of a human body.

1.1.3.3 Volumetric model

Sometimes it is referred to as volume-based model and is used solely for the 3D pose estimation problem. The human body is represented as geometric meshes and shapes, which can be very beneficial to observe how the human body deforms when in certain poses.

1.1.4 Classical and Deep Learning-based approaches to Human Pose Estimation

1.1.4.1 Classical approaches

Traditional HPE approaches only focus on dealing with 2D position or spatial location of human body keypoints from images and videos. They adopt different hand-crafted feature extraction techniques and algorithms for body parts. One of the early works in articulating human pose used the framework called “Pictorial Structure”. A good example is brilliant work from Matthias Dantoe et al [4]. The authors employed two-layered random forests as joint regressors to localize the coordinates of the joints. These approaches work well when the input image has good clearance and visible limbs but fail to capture occluded or deformable body parts. In order to overcome these issues, hand-crafted features were applied such as edges, contours, color histograms, HOG (Histogram Oriented Gradient), etc. Despite the addition of these features, classical approaches still faced the problem of inaccuracy and poor generalization capability. Consequently, adopting new and better approaches was just a matter of time.

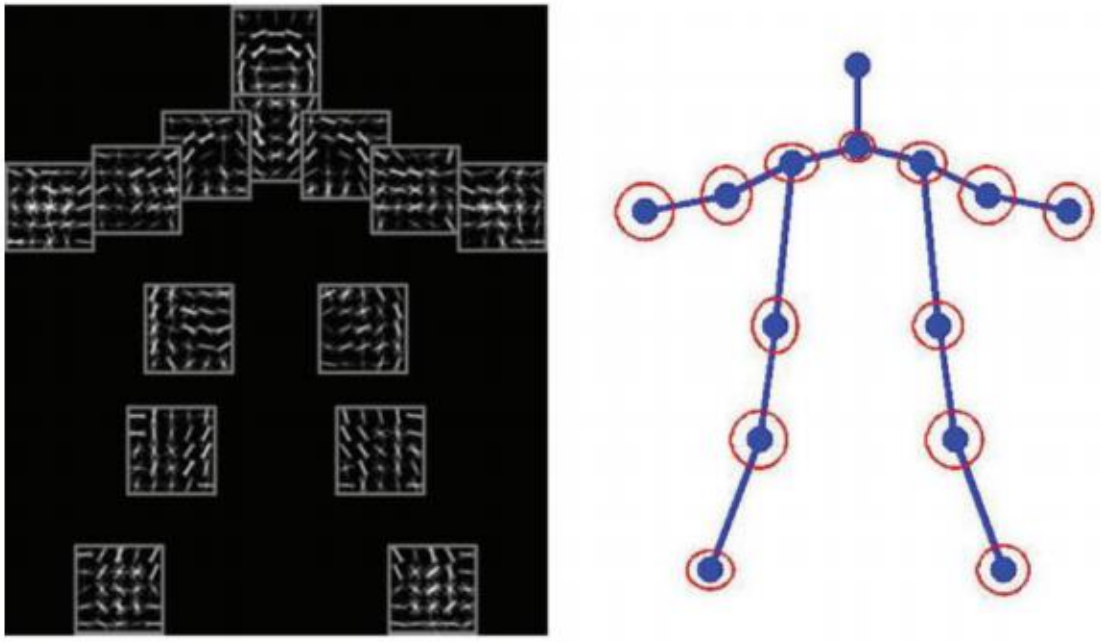


Figure 1.3: Example of HOG features for keypoints detections [5].

1.1.4.2 Deep learning-based approaches

Conventional methodology has its limitations but by utilizing Deep Learning (DL), researchers have found new solutions to overcome such challenges. Deep learning-based approaches have not only reshaped entirely the HPE problem but also other Computer Vision tasks such as classification, detection and segmentation. Out of all the DL-based approaches, Neural Networks (NNs) protrude and specifically Convolutional Neural Networks (CNNs) surpasses other algorithms, this stays true even within HPE landscape. Some of the works that employ different types of NNs such as recurrent neural network [6] show promising results but need to be studied and experimented further.

Classical solutions require hand-crafted features, which is time consuming and extremely intricate. CNN, on the other hand, has the ability to extract patterns and representations from a given input image with higher accuracy and precision. When provided enough training-testing-validation data, CNN can learn very complex features on its own.

Over the years, more and more papers that adopt CNN as the base for their research have been published. In addition to showcasing the improvements in tackling HPE, they also enrich our understanding and new perspectives to CNN. Major milestones can be DeepPose which was

proposed by researchers at Google [7], DensePose from Facebook researchers [8], Convolutional Pose Machines [9], High-Resolution Net (HRNet) [10], etc.

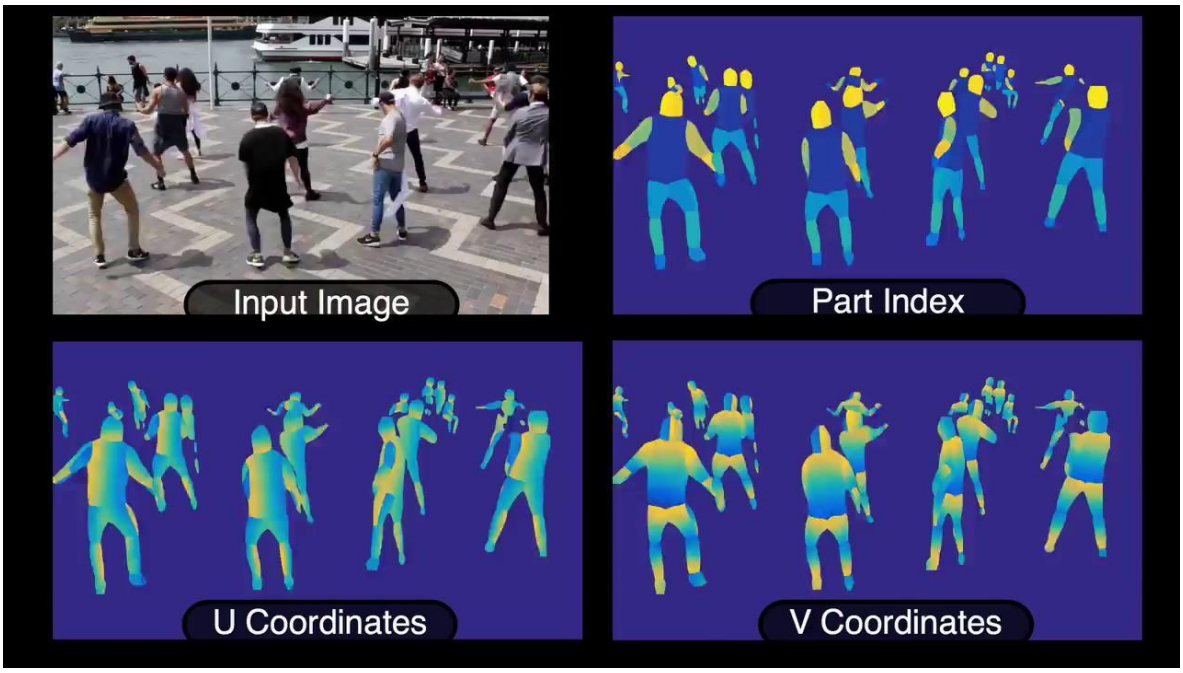


Figure 1.4: DensePose example [8].

1.1.5 Single- and multi-person Human Pose Estimation

Single person HPE simply pivots around processing one person at a time. Normally, the person of interest would be at the very center of the frame.

As more and more research started to take off in HPE, it brought up more exciting challenges. One of such is multi-person Pose Estimation. Deep Neural Networks are very proficient at figuring out one single person pose but they struggle when it comes to more than one in the image because of several reasons:

- One image can contain more than one person that are in different positions and scales (can be far away or up close).
- As the number of people increases, the interactions between these people can result in very complex and time-consuming computations.
- When the image has a group of people standing or sitting next to each other (a crowd), it is even harder.

So as to address these particular challenges, two structures were introduced:

1. Top-down: Perform a localization of all possible humans in the image or video and then estimate the pose of each person. This method has high accuracy but heavily depends on the human detector. It also limits how many people the human detector can detect and generally takes more time to process one image.
2. Bottom-up: Estimate all keypoints in the image then assemble and assign which keypoints belong to which person. In addition to being faster than the top-down approach, this method can predict more people. However, it has lower accuracy in comparison.

1.1.6 Human Pose Estimation datasets

There are a lot of public datasets available both for 3D and 2D Pose Estimation.

3D Pose Estimation datasets:

- DensePose [11]
- UP-3D [12]
- Human3.6m [13]
- 3D Poses in the Wild [14]
- Total Capture [15]

2D Pose Estimation datasets:

- MPII Human Pose Dataset [16]
- Leeds Sports Pose [17]
- Frame Labeled in Cinema [18]
- Frame Labeled in Cinema Plus [19]
- YouTube Pose (VGG) [20]
- Microsoft COCO: Common Objects in Context [21]

1.2 Objective

The purpose of this thesis is to research, implement and thoroughly evaluate a deep CNN that can be used to estimate human body pose from 2D images. The thesis also goes into detail of how the data is preprocessed, the architecture and the implementation of the CNN model. Additionally, from the obtained predictions of the model, evaluation is conducted to test its performance and accuracy. Different strategies are also discussed to observe which provides better outcomes. Suggestions to improve the NN are also discussed.

1.3 Scope and limitations

Firstly, HPE is a difficult problem domain due to numerous reasons as mentioned in Section 1.1.1. In order to achieve feasible results in a timely manner, only monocular RGB images are used. Additionally, the main focus is single person in single frame and the human body configuration is kinematic (see Section 1.1.3.1). The whole body is considered and there are no restrictions in clothing, lighting, activities or joints visibilities. Secondly, the main approach to HPE in this thesis is Deep Learning-based, namely, a Convolutional Neural Network. Lastly, COCO dataset is chosen (see Section 1.1.6) for training, testing and evaluation.

Chapter 2: Introduction Neural Networks

Neural Networks (NNs), also referred to as Artificial Neural Networks (ANNs), are one of many Deep Learning (DL) techniques. It is considered the most well-known and prominent in the DL field. As their names suggest, NNs are inspired by the human brain's function and structure.

2.1 Neurons

2.1.1 Biological neurons

Approximately, there are 86 billion neurons in the human brain. They are the fundamental building blocks of brain and nervous system. These interconnected nerve cells are in charge of processing the incoming inputs and forwarding chemical and electrical responses accordingly. Altogether, they create a dense and sophisticated signals exchange net which defines our interactions with other human beings or the surrounding environment.

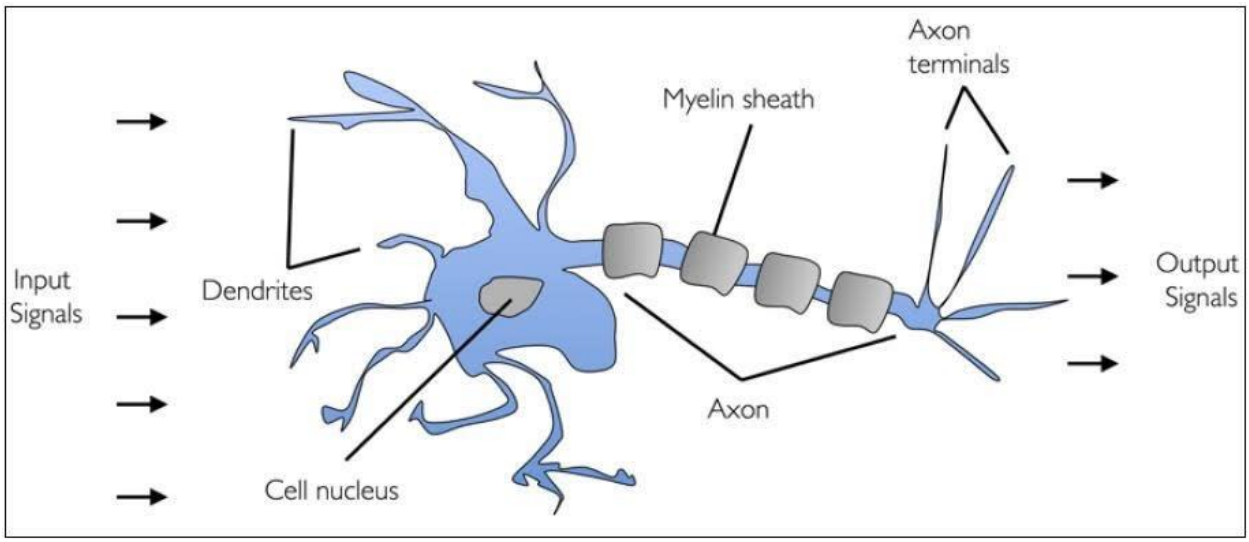


Figure 2.1: Diagram of biological neuron [22].

As shown in Fig. 2.1, dendrites receive information from previous neurons. A cell nucleus (a soma) then collects and processes input signals from the dendrites. Afterward, the processed information is passed along the axon to different terminals. Each terminal will be connected to the next neuron via synapse and then recurse.

2.1.2 Artificial neurons

An artificial neuron is simply a mathematical function that mimics the behavior of a biological neuron. Each neuron accumulates the weighted input values and forwards this sum into a nonlinear function to generate output. All signals are represented as numerical values.

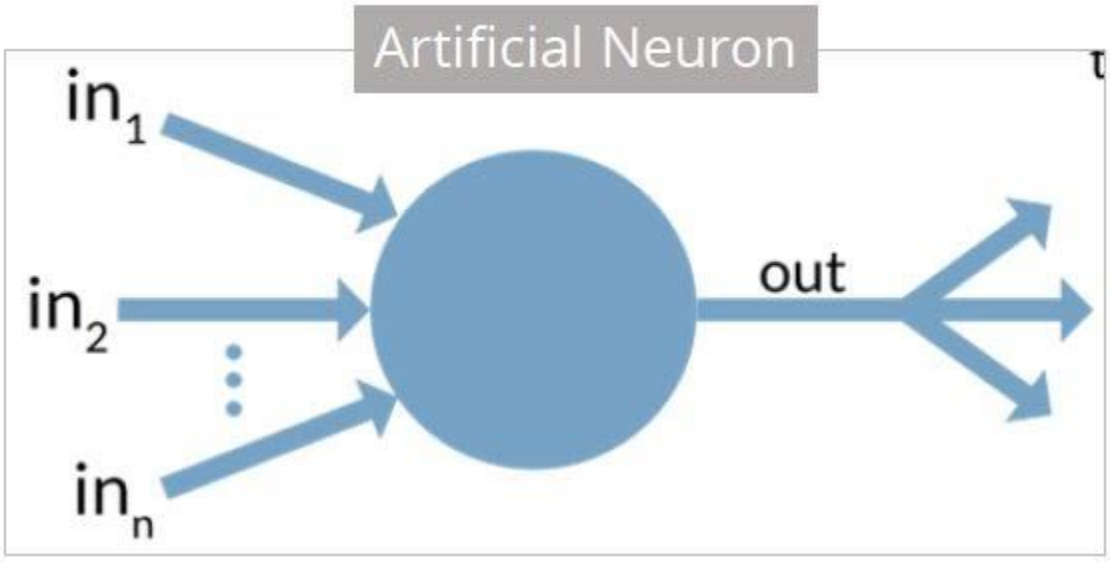


Figure 2.2: Diagram on an artificial neuron [22].

The table below shows the components of an artificial neuron and its counterpart:

Biological neuron	Artificial neuron
Cell nucleus	Node
Dendrites	Input
Synapse	Weights
Axon	Output

Table 2.1: Comparison between biological and artificial neuron.

2.2 Neural Networks: Overview

NNs have been around for a very long time and were first introduced by Warren McCulloh and Walter Pitts in 1943 [23]. A NN is an interconnected collection of artificial neurons. A given neuron is connected to multiple neurons via links which are equivalent to the axon-synapse-dendrite connections of biological neurons. Each of the links carries a weight that represents the importance or influence of one neuron on another. The larger the weight value is the more impact the neuron has.

Perceptron is a simple NN architecture that was proposed in 1957 by Frank Rosenblatt [24]. Functionally, a perceptron is an artificial neuron. Output of artificial neuron ranges typically from $[0.0, 1.0]$, a perceptron's output on the other hand can only be either 0 or 1. Thus, perceptron is used primarily in binary classifiers.

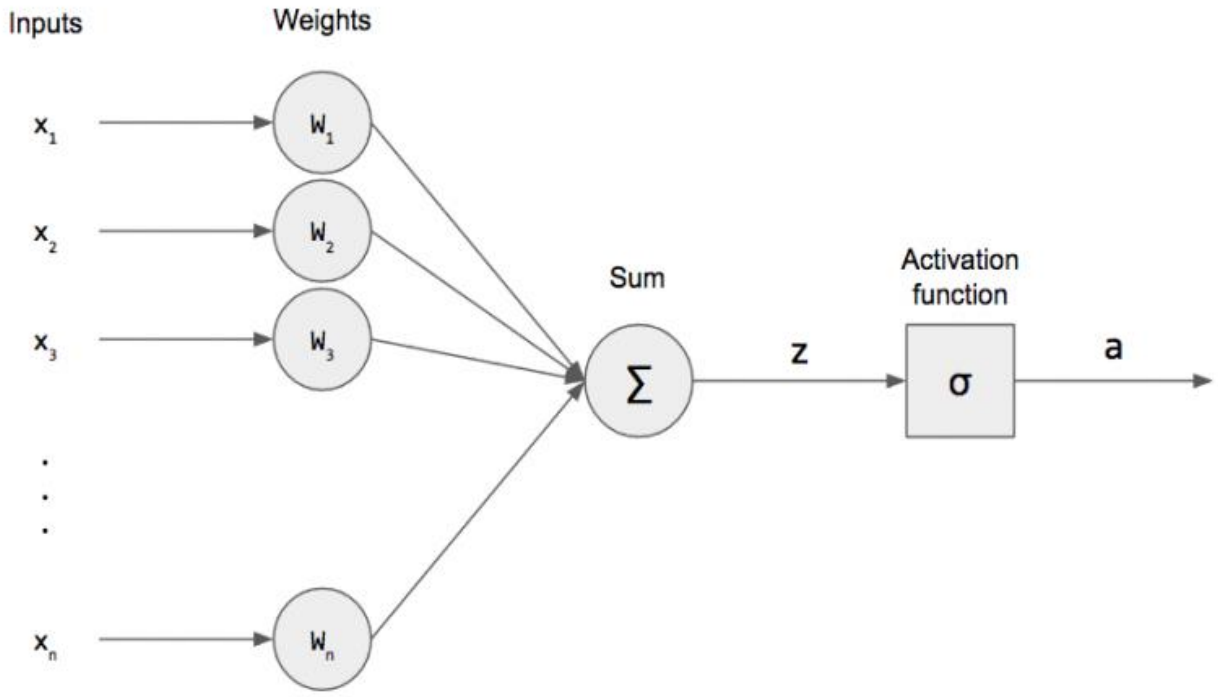


Figure 2.3: Mathematical model of a perceptron.

A NN can contain multiple perceptrons or artificial neurons. In any case, it always has the structure of one input layer, one or more hidden layers and one output layer. The complexity or "deepness" of a network is characterized by the number of hidden layers. Nevertheless,

increasing the number of hidden layers may amplify negligible errors in a less hidden layers network and increase the required training time. Depending on the situation, a sufficient number of hidden layers is desirable.

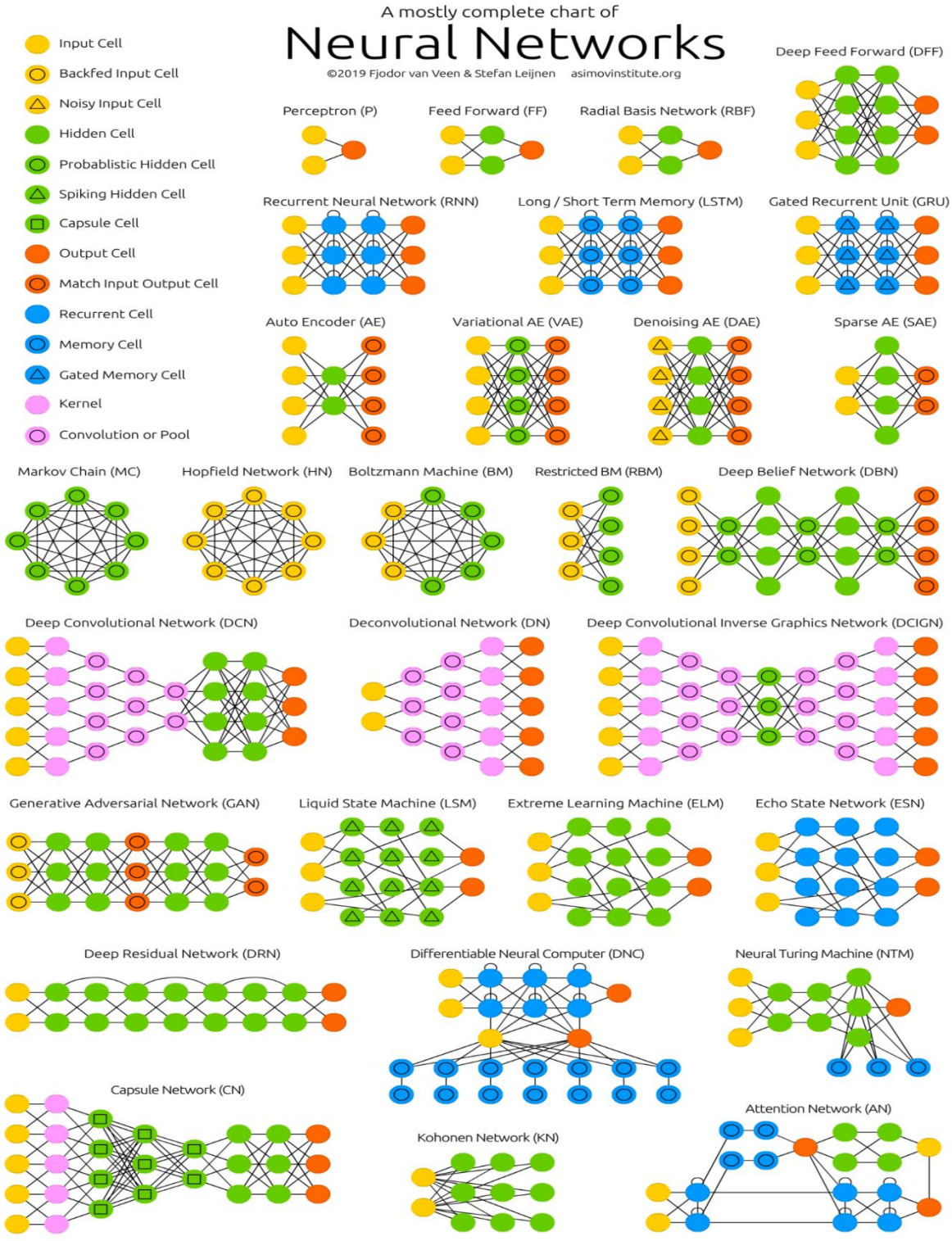


Figure 2.4: Various Neural Network Architectures [25].

2.3 Elements of Neural Networks

2.3.1 Layers

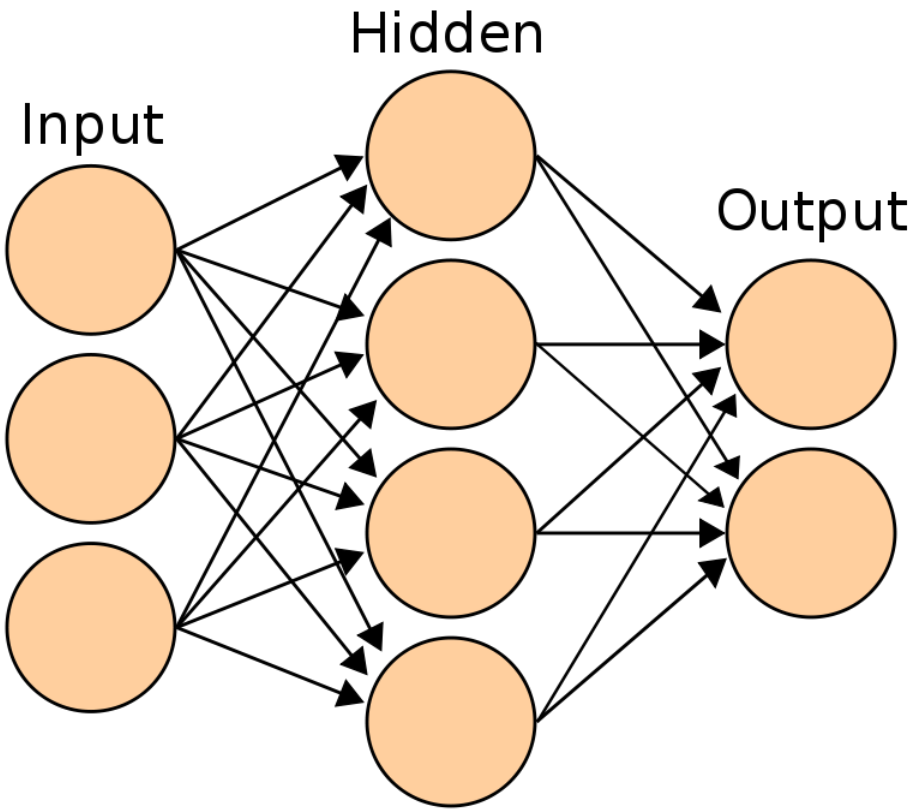


Figure 2.5: A simple NN layout.

2.3.1.1 Input layer

The very first layer of the NN that contains raw or preprocessed data, the number of nodes (neurons) in this layer should always equal the quantity of features or attributes of an example in the dataset. An example, also referred to as an instance, is a pair (X, y) where X refers to a set of independent variables (features) and y is the target (label).

2.3.1.2 Hidden layers

NNs usually have multiple hidden layers but for simple networks one layer is adequate. These layers hold a critical role in the outstanding performance and complexity of the network.

Each layer attempts to learn different aspects about the data by minimizing an error/cost function. The true values of these layer's nodes are unknown in the training hence they are "hidden".

2.3.1.3 Output layer

The last layer is where the number of nodes needs to be the number of possible cases or labels that we desire. For example, in a system to distinguish between cats and dogs, there are exactly two output nodes.

2.3.2 Weights and biases

Weights and biases are the learnable parameters of most machine learning models, including NNs. A trainable network will randomize both the weight and bias values prior to the training process. Both parameters are modified toward desirable values while the training is happening. The pair influences the incoming input data but to a distinct extent.

As suggested in Section 2.2 there is a link or connection between a particular pair of neurons. The order of significance of this link is a numerical value called weight. In other words, a weight dictates how much influence the input will induce on the output. Weights can be negative, which depict inhibitory connection, or positive, which indicate excitatory connection.

Bias is an additional input to the next layer. Simply, it is a constant node that is not affected by the previous layer because it does not have any incoming connections. It always has the value of 1 but it does have outgoing connections to next neurons with different weights. The bias unit assures that an activation in a neuron is feasible even when all the inputs are zero.

The relation of bias and weight with respect to input and output can be demonstrated as follows:

$$Output = \sum_{i} weight_i * input_i + bias. \tag{2.1}$$

2.3.3 Activation functions

It can be pointed out in Fig. 2.3 that in order to produce the output, a weighted sum (sometimes referred as activation) is passed through a (usually non-linear) function called activation function. Additionally, a bias is added to the sum before forwarding to the function. An activation function, as its name implies, decides whether a node should be active or not (on or off). There are many different activation functions with their own pros and cons, a few commonly used functions are mentioned below.

2.3.3.1 Binary step

Binary step function compares the given input to a threshold value. If the input is greater than the threshold, then the neuron is active, otherwise it is deactivated.

Mathematically, it can be represented as

$$f(x) = \begin{cases} 0, & x < 0 \\ 1, & x \geq 0 \end{cases}. \tag{2.2}$$

Figure 2.6 illustrates the binary step activation function (2.2).

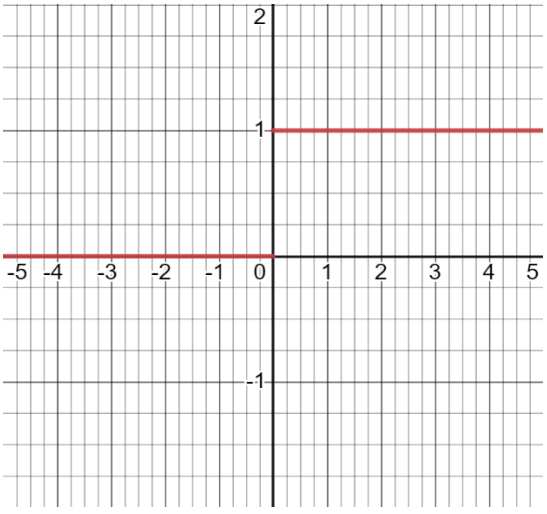


Figure 2.6: Binary step function.

2.3.3.2 Linear activation

Linear activation function, also referred as the identity function, produces output that is proportional to the input, as follows:

$$f(x) = x. \tag{2.3}$$

Figure 2.7 illustrates the linear activation function (2.3).

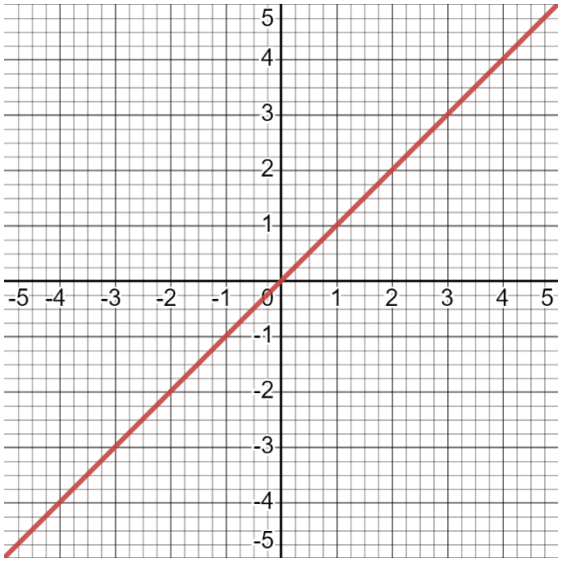


Figure 2.7: Linear function.

2.3.3.3 Sigmoid

This function takes in any real values as input and output is in the range of 0 to 1. The larger the input is, the closer the output approaches 1.0, whereas the smaller the input is, the closer the output is to 0.0. The function is defined as

$$f(x) = \frac{1}{1 + e^{-x}}. \tag{2.4}$$

and is illustrated in Fig. 2.8.

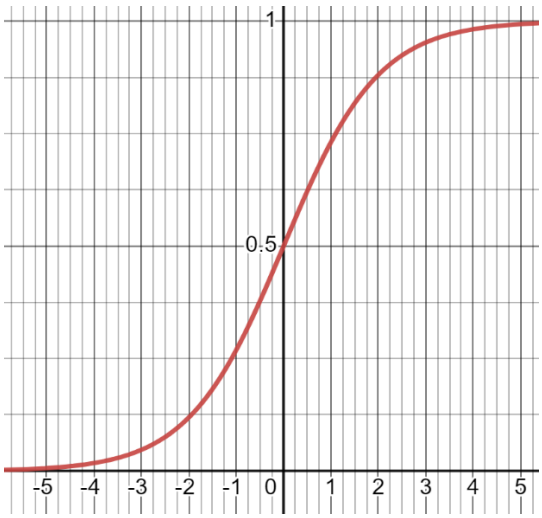


Figure 2.8: Sigmoid function.

2.3.3.4 Tanh

Tanh function is similar to sigmoid function. The only difference is that the output of tanh function ranges from -1 to 1. The function is as follows:

$$f(x) = \frac{(e^{x} - e^{-x})}{(e^{x} + e^{-x})}. \tag{2.5}$$

and is illustrated in Fig. 2.9.

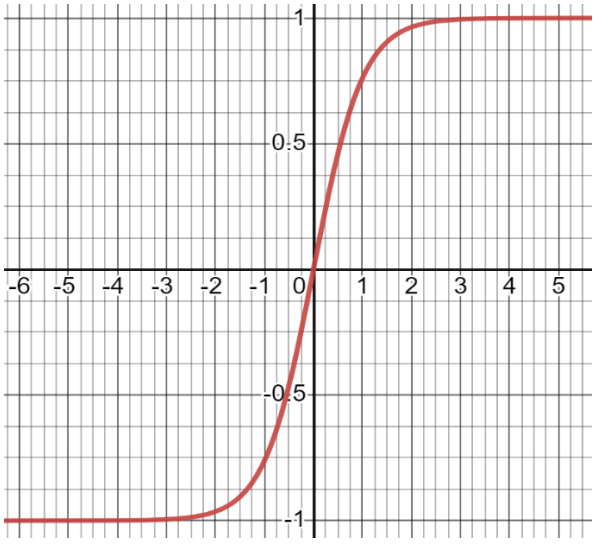


Figure 2.9: Tanh function.

2.3.3.5 Rectified linear unit (ReLU)

The ReLU activation function is similar to the linear activation function. Although ReLU deactivates the neuron if the input is less than 0. It is defined as follows:

$$f(x) = \max(0, x). \tag{2.6}$$

and is illustrated in Fig. 2.10.

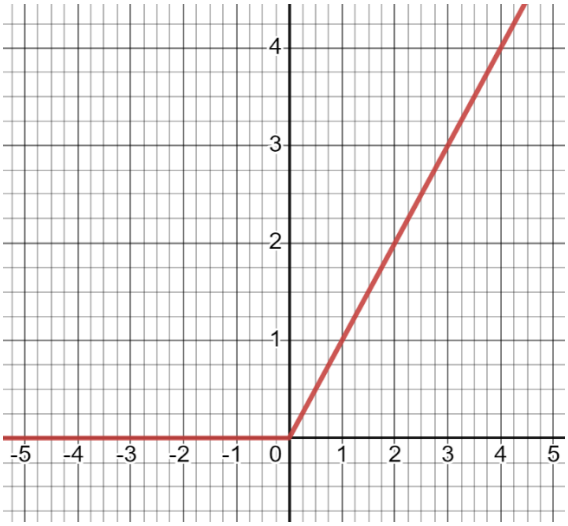


Figure 2.10: ReLU function.

2.4 Training a Neural Network

NNs learn by processing examples or instances, each of which contains a known pair of "input" and "result" or as referred to as (X, y) in Section 2.3.1.1. This type of learning is called supervised learning. Additionally, there are unsupervised learning, reinforcement learning and self-learning.

2.4.1 Forward propagation and Backpropagation

As the name implies, forward propagation is the forward flow of information. Each layer accepts input data from the prior layer, processes it and then passes it to the successive layer until it reaches the output layer. During forward propagation the data should be fed in the forward

direction solely. Such network configuration is known as feed-forward network. Moreover, during this process at each neuron in the hidden layer(s) or output layer, a weighted sum of inputs from previous layers is calculated and the activation function of said neutron will decide whether to forward this value. In other words, the purpose of forward propagation is the calculation and storage of immediate variables at each node in the hidden layer(s) and output layer.

Contrary to forward propagation, backpropagation is simply the method that traverses information in the reverse order. In essence, backpropagation is how NNs learn. At the end of forward propagation, the output is compared to a desired output (label) via a cost function to evaluate the accuracy or performance of the network. Backpropagation is all about feeding the loss backwards in a way that weights can be updated to minimize the loss in the next iteration. However, the weights cannot directly be adjusted in an arbitrary way. Optimization algorithms such as gradient descent assists to find weights that prompts smaller loss. One thing to be noted, such algorithms do not guarantee smaller loss every time, many other factors can play a critical role.

The main goal to strive for during training an NN, is lower error rate (loss). Proper tuning of weights and biases ensures a smaller error rate at every cycle of forward and backpropagation. As a result, the network becomes more accurate and reliable by increasing its generalization and "learning" from its past mistakes.

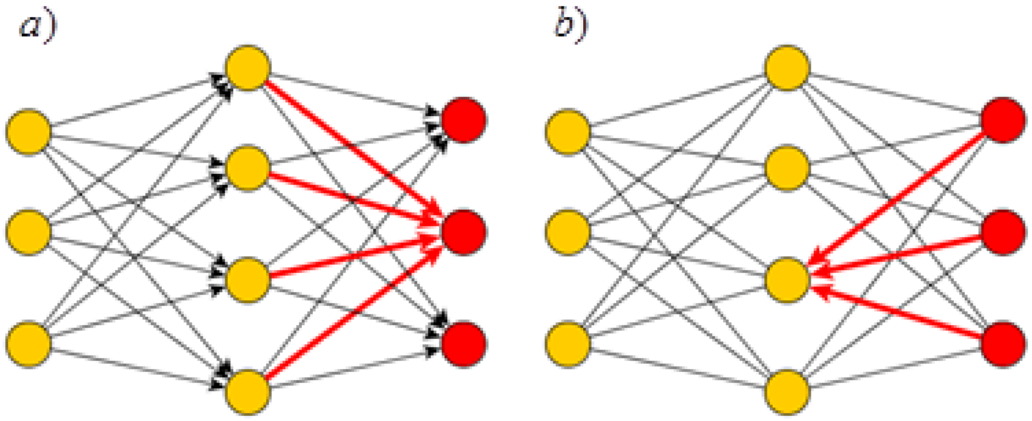


Figure 2.11: a) Forward propagation and b) Backpropagation.

2.4.2 Cost functions

During the very first training phase, the network randomly initializes weights and tries to make predictions on training data. Nonetheless, how can the network comprehend how badly it is behaving/predicting or how far off it is from the truth? That is when the cost function comes into the picture. Cost function is a metric that evaluates how well the network performs by determining the offset of predictions with respect to actual results during training. There are different types of cost functions for different ML problems.

2.4.2.1 Cost functions for regression problem

Regression problem is where a real-value quantity needs to be predicted.

Let us denote y is the actual result, $\hat{y}$ is the predicted output and n is the number of examples in the dataset.

- Mean Error (ME):

$$ME = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i). \tag{2.7}$$

- Mean Square Error (MSE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{2.8}$$

- Mean Absolute Error (MAE):

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2. \tag{2.9}$$

2.4.2.2 Cost functions for classification problem

Classification problem is a problem where an example is determined to which class it belongs to. If there are two classes, the problem is called binary classification. If there are more than two, it is referred to as multi-class classification.

- Categorical cross entropy (for multi-class classification):

$$Loss = -\sum_{i=1}^{n} y_i \cdot \log \hat{y}_i. \tag{2.10}$$

- Binary cross entropy (for binary classification):

$$Loss = -\frac{1}{n}\sum_{i=1}^{n} (y_i \cdot \log \hat{y}_i + (1 - y_i) \cdot \log(1 - \hat{y}_i)), \tag{2.11}$$

where y_i is the true probability distribution, $\hat{y}_i$ is the predicted probability distribution and n is the total number of examples in the dataset.

2.4.3 Gradient descent

Even though a cost function evaluates how an NN performs, it does not provide insight on how to improve the network. Gradient descent is an optimizer algorithm that provides directional indication towards the minimum value of error of the network. To put it another way, it is used for finding local minima of a cost function.

Mathematically, the gradient of a function is the vector of partial derivatives with respect to all independent variables. A gradient, also known as the slope of a function, can be interpreted as the "direction and rate of fastest increase". Since gradient is a vector, its direction is the direction in which the function increases quickest, and the magnitude is the rate of increase in that direction.

In vector calculus, for function $f: \mathbb{R}^n \to \mathbb{R}$, its gradient $\nabla f: \mathbb{R}^n \to \mathbb{R}$ is defined at the point $p = (x_1, \dots, x_n)$ in n-dimensional space as

$$\nabla f(p) = \begin{bmatrix} \frac{\partial f}{x_1}(p) \\ \vdots \\ \frac{\partial f}{\partial x_n}(p) \end{bmatrix}. \tag{2.12}$$

Since ∇f points in the direction of greatest increase of the function, $-\nabla f$ must point in the direction of greatest decrease of the function hence it is called gradient descent (as opposed to gradient ascent).

For a parabola gradient descent would be the slope where it is smaller than 0.

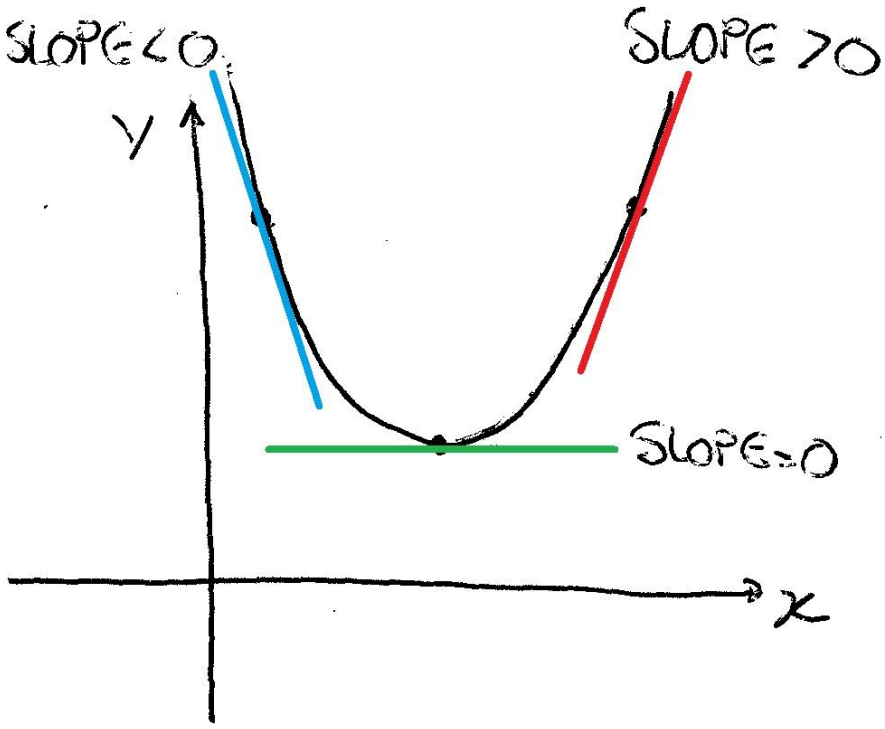


Figure 2.12: Slopes of a parabola.

2.4.4 How weights and biases are updated

Consider the following simple NN:

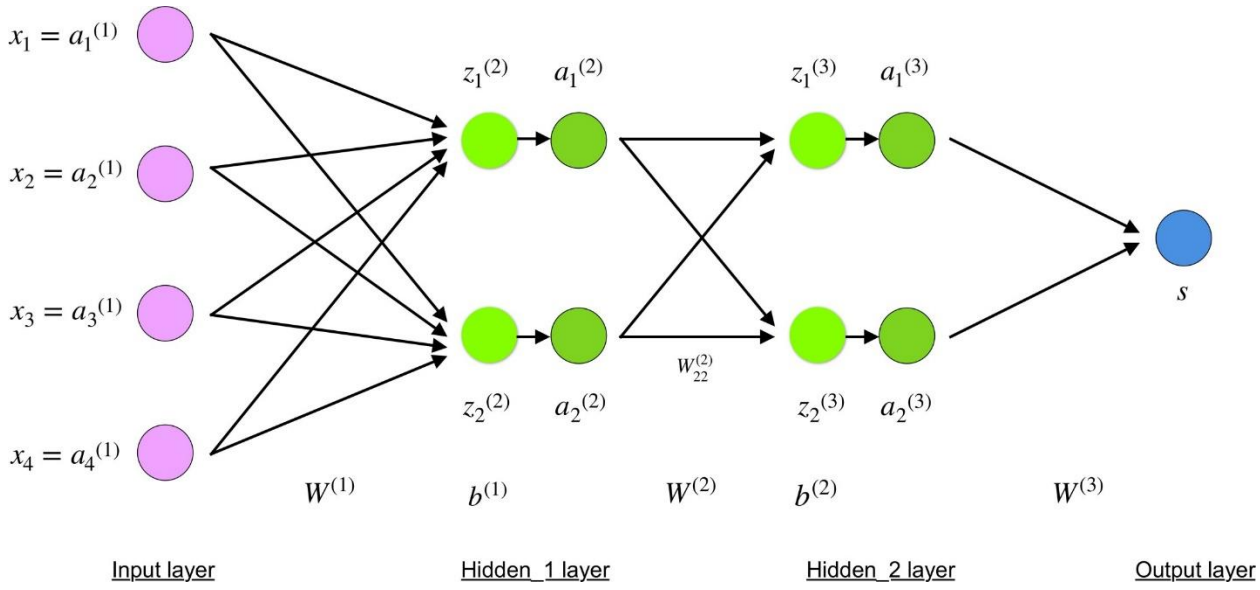


Figure 2.13: Diagram of an NN [26].

2.4.4.1 Important notation

The following table shows some important notations:

Symbol	Definition
L	Number of layers in the network
l	Layer index
j	Node index for layer l
k	Node index for layer $l-1$
y_j	The expected value of node j in the output layer L for a single training example
C_0	Loss function of the network for a single training example
$w_j^{(l)}$	The vector of weights connecting all nodes in the layer $l-1$ to node j in layer l
$w_{jk}^{(l)}$	The weight that connects node k in the layer $l-1$ to node j in layer l
$b_j^{(l)}$	The bias for node j in the layer l
$z_j^{(l)}$	The input for node j in layer l
$g^{(l)}$	The activation function used for layer l
$a_j^{(l)}$	The activation output of node j in layer l

Table 2.2: Notations for NN.

For instance, the input of node j in layer l is the weighted sum of activation outputs (of N neurons) from prior layer $l-1$ and bias for that node:

$$z_j^{(l)} = \sum_{k=1}^{N} w_{jk}^{(l)} a_k^{(l-1)} + b_j^{(l)}. \tag{2.13}$$

Activation output of the given node j is the result of passing its input through activation function:

$$a_j^{(l)} = g^{(l)}\left(z_j^{(l)}\right). \tag{2.14}$$

If the node j is in layer L (output layer) and MSE loss function is used, the loss of the node would be as following:

$$C_{0_j} = \left(a_j^{(l)} - y_j\right)^2. \tag{2.15}$$

It is to be noticed that y_j is constant so C_{0_j} can be expressed as the function of $a_j^{(L)}$:

$$C_{0_j}\left(a_j^{(l)}\right). \tag{2.16}$$

Moreover, the input of node j is in layer l can be expressed as the function of $w_{jk}^{(L)}$ and $b_j^{(L)}$:

$$z_j^{(l)} = \left(w_{jk}^{(l)}, \quad b_j^{(l)}\right). \tag{2.17}$$

From Eq. (2.14), (2.16) and (2.17) , we have C_{0_j} as a composition of functions:

$$C_{0_j}\left(a_j^{(l)}\left(z_j^{(l)}\left(w_{jk}^{(l)}, \quad b_j^{(l)}\right)\right)\right). \tag{2.18}$$

The overall loss of the network is defined as

$$C_0 = \frac{1}{N}\sum_{j=1}^{N} C_{0_j}. \tag{2.19}$$

2.4.4.2 Finding gradient descent

Consider the simplified version of the NN shown in 2.14.

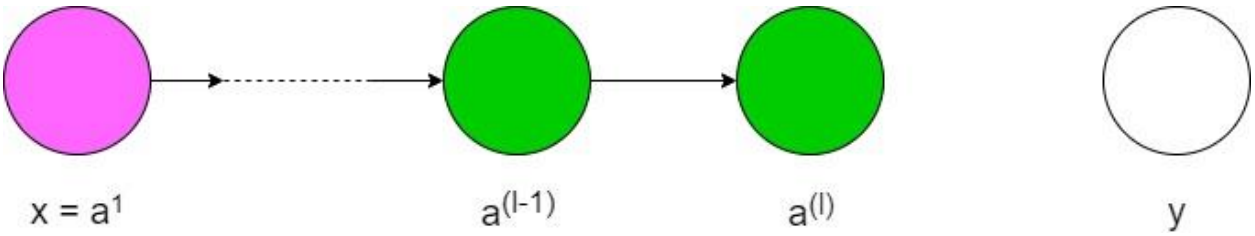


Figure 2.14: A simplification of NN.

here all layers only have one neuron each, hence, node index j is omitted. Purple node represents input, the first green node represents activation output of that node in layer $l-1$, similarly the second one is activation output in layer l or for this case it is the output and y is the desired output.

Let us have a closer look at the node in layer l:

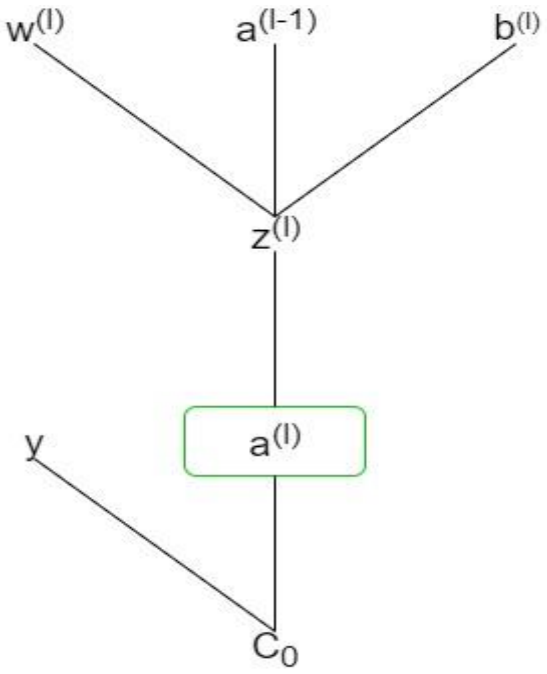


Figure 2.15: The node (green) in layer j.

If MSE is used as the loss function, the cost would be

$$C_0 = (a^{(l)} - y)^2. \tag{2.20}$$

The activation output is described as

$$a^{(l)} = g^{(l)}\left(z^{(l)}\right). \tag{2.21}$$

and the input of the node $z^{(l)}$ is expressed as

$$z^{(l)} = w^{(l)}a^{(l-1)} + b^{(l)}, \tag{2.22}$$

where $w^{(l)}$ is the weight vector for the node, but since there is only one node in the prior layer, the vector only consists of one weight. $a^{(l-1)}$ is the activation output of layer $l-1$. Lastly, $b^{(l)}$ is the bias.

As demonstrated in Section 2.4.4.1, the cost function is a composition of function:

$$C_0 = \left(a^{(l)}\left(z^{(l)}\left(w^{(l)}, b^{(l)}\right)\right)\right). \tag{2.23}$$

The above function implies that the cost function is a function of weight $w^{(l)}$ and bias $b^{(l)}$. Therefore, gradient of C_0 is

$$\nabla C_0 = \begin{bmatrix} \frac{\partial C_0}{\partial w^{(l)}} \\ \frac{\partial C_0}{\partial b^{(l)}} \end{bmatrix}. \tag{2.24}$$

Partial derivative of C_0 with respect to $w^{(l)}$ using chain rule give the following expression:

$$\frac{\partial C_0}{\partial w^{(l)}} = \frac{\partial C_0}{\partial a^{(l)}}\frac{\partial a^{(l)}}{\partial z^{(l)}}\frac{\partial z^{(l)}}{\partial w^{(l)}}. \tag{2.25}$$

Using Eq. (2.20), the first term can be expressed as

$$\frac{\partial C_0}{\partial a^{(l)}} = 2\left(a^{(l)} - y\right). \tag{2.26}$$

Using Eq. (2.21), the second term can be expressed as

$$\frac{\partial a^{(l)}}{\partial z^{(l)}} = g'^{(l)}\left(z^{(l)}\right). \tag{2.27}$$

Using Eq. (2.22), the last term can be expressed as

$$\frac{\partial z^{(l)}}{\partial w^{(l)}} = a^{(l-1)}. \tag{2.28}$$

From Eq. (2.25), (2.26), (2.27) and (2.28):

$$\frac{\partial C_0}{\partial w^{(l)}} = 2\left(a^{(l)} - y\right)\left(g'^{(l)}\left(z^{(l)}\right)\right)\left(a^{(l-1)}\right). \tag{2.29}$$

Similarly for bias $b^{(l)}$:

$$\frac{\partial C_0}{\partial b^{(l)}} = 1\left(g'^{(l)}\left(z^{(l)}\right)\right)\left(a^{(l-1)}\right). \tag{2.30}$$

However, the weight and bias are only adjusted by a fraction of gradient descent:

$$\begin{cases} w^{(l)} = w^{(l)} - \alpha \dfrac{\partial C_0}{\partial w^{(l)}} \\ b^{(l)} = b^{(l)} - \alpha \dfrac{\partial C_0}{\partial b^{(l)}} \end{cases}, \tag{2.31}$$

where α is also known as learning rate.

After updating the weight and bias that influence node $a^{(l)}$, we then carry on with node $a^{(l-1)}$ and the node before that until reaching the input layer.

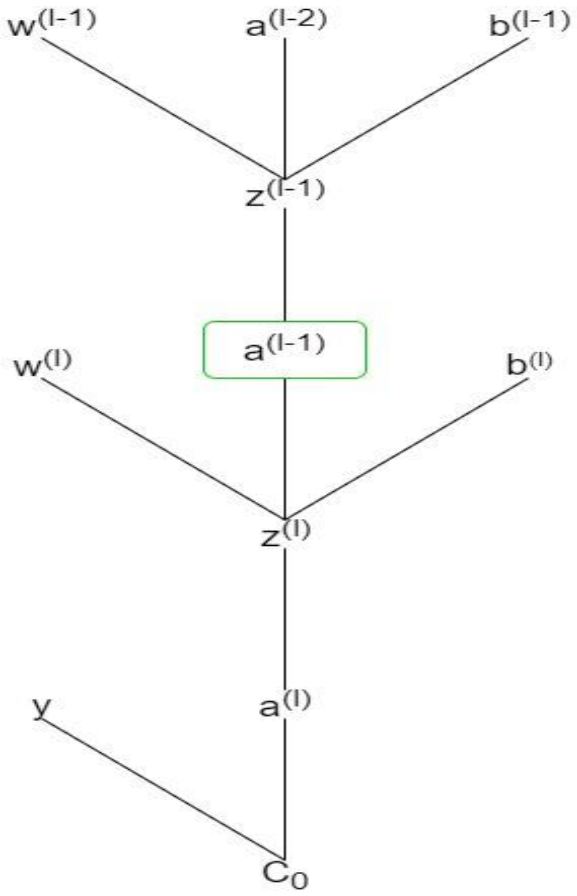


Figure 2.16: The node at layer l-1.

In case there are multiple nodes in one layer, the gradient of the cost function is

$$\nabla C_0 = \begin{bmatrix} \frac{\partial C_0}{\partial w^{(1)}} \\ \frac{\partial C_0}{\partial b^{(1)}} \\ \vdots \\ \frac{\partial C_0}{\partial w^{(l)}} \\ \frac{\partial C_0}{\partial b^{(l)}} \end{bmatrix}. \tag{2.32}$$

The process should be the same for each weight and bias.

2.5 Hyperparameters

In ML, a hyperparameter is a parameter that controls the learning process. By contrast, the values of other parameters (such as weights and biases) are derived from training. Hyperparameters can be viewed as a network's configuration settings, which specifies how the network structure is laid out and how it will function. They cannot be estimated or modified during the training process. How the hyperparameters are decided and calibrated has a direct and significant impact on how the network performs. Thus, the process of "hyperparameter optimization" is the selection and fine-tuning of these parameters to produce the best outcome. There are several methods to optimize but the easiest approach is trial and error. It is simply trying different combinations of hyperparameter values and observing what gives the best result. The number of hyperparameters can be overwhelming. Therefore, sometimes it is better to focus on a handful of common hyperparameters.

2.5.1 Network architecture

The network itself is a hyperparameter that is inherent in all types of ANN. This broad hyperparameter consists of settings such as the number of layers, number of neurons for each layer, activation functions used for layers, weights and biases configurations, dropout, etc. Nonetheless, all of the mentioned settings increase the complexity of a network, which can be double-edged. A complicated network may produce a better output, though not certain, it also requires a greater magnitude of processing power.

2.5.2 Loss function

It is critical to choose the suitable loss function, also referred to as cost function, for the task since each loss function is designed for some specific scenarios, not all (see Section 2.4.2). The choice of loss function depends on the configuration of the output layer. Furthermore, some loss function requires less time to compute, hence less time to train, but can be less accurate (i.e., ReLU).

2.5.3 Learning rate

As introduced in Section 2.4.4.2, learning rate is a small percentage of gradient descent of the cost function. Learning rate determines the step size at each iteration while moving toward the minimum of the loss function. A sufficient learning rate is needed to reduce training time while not overshooting the minimum point.

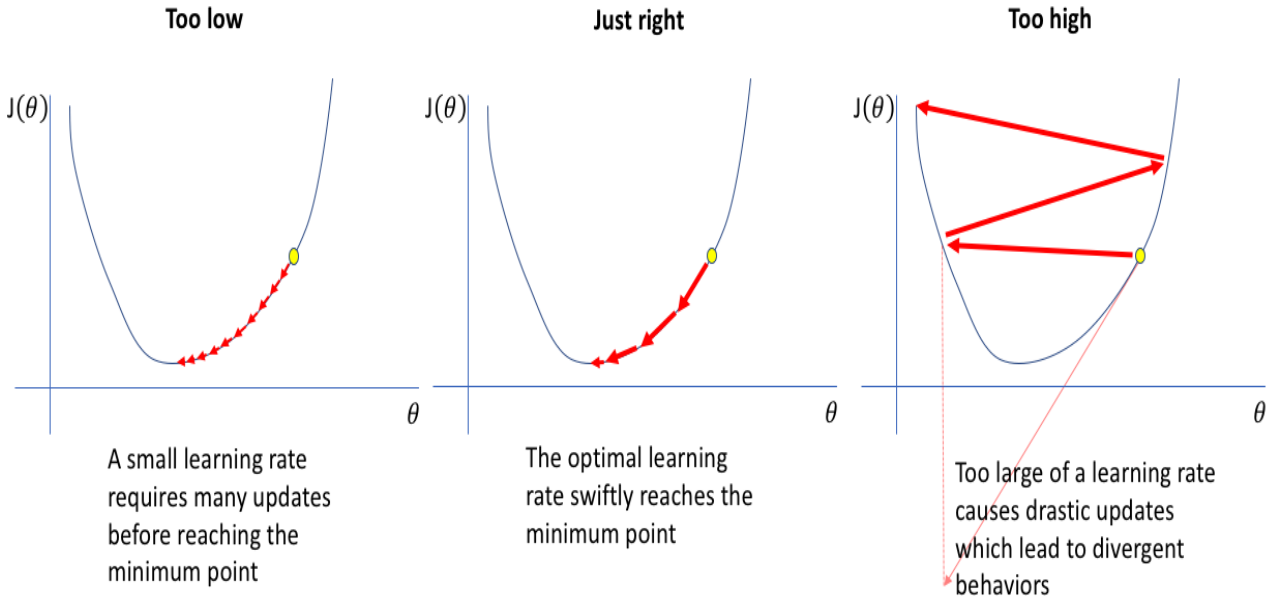


Figure 2.17: The effect of different learning rates [27].

2.5.4 Batch size

The number of training examples utilized in one iteration is referred to as batch size. The higher the batch size is, the more memory the model needs. Sometimes, when dealing with extremely large datasets it is not possible to fit the whole dataset into the network. Splitting the dataset into smaller subsets benefits the parallel operation of the processing unit, especially with GPU or TPU. As a result, the training is shortened. That is because we can update the weights after each batch rather than just once if we used all the examples during propagation. However, each batch should be a smaller representation of the whole dataset. For example, if a dataset has 500 images of dogs and 500 images of cats, then the batch should contain approximately 50% of cat images and the rest is dog images.

2.5.5 Optimizer

Not to be confused with hyperparameter optimization, an optimizer is the algorithm that updates the weights of the network during training, using output from the loss function along with other parameters. Commonly adopted optimizers such as SGD (Stochastic Gradient Descent), Adam (Adaptive Momentum Estimation) or RMSProp (Root Mean Square Propagation) are based on gradient descent (see Section 2.4.3). Other algorithms such as Bayesian Optimization [28] draw inspiration from statistics or Particle Swarm Optimization [29] are inspired by natural phenomena and do not use gradients.

2.5.6 Epoch

One epoch is one forward and one backward propagation of all training examples. It is not the same with an iteration. The number of iterations is the number of passes (forward and backward) using batch size. If the whole dataset is fed, one iteration is one epoch, otherwise one epoch should contain multiple iterations. For example, consider a dataset with 1000 training examples and the batch size is 500 examples, then it will take 2 iterations to complete one epoch.

If the number of epochs is too little, the network will underfit or not be trained enough. On the other hand, if the number of epochs is too much, the network overlearns and results in overfitting.

Chapter 3: Convolutional Neural Network (CNN)

Conventional NNs, where presumably each neuron of a layer is fully connected to all the neurons of the next layer (also known as Fully connected neural network), do not scale well to full images. For instance, images of CIFAR-10 dataset [30] have size of only $32x32x3$, so one neuron of a hidden layer would have $32 * 32 * 3 = 3072$ weights. However, these images are relatively small in comparison to regularly used images. An image with medium size of $240x240x3$ would result in $240 * 240 * 3 = 172800$ weights for a single neuron. Furthermore, one NN can have multiple hidden layers where each layer has a respectable quantity of neurons. Briefly, it is utterly challenging for ANNs to work with image data directly. Instead, concrete features need to be provided manually. For example, a NN that categorizes different types of flowers, data points such as petal length or color of the flower should be given rather than the whole image of the flower.

Convolutional Neural Network (CNN) is a class of ANN. CNN is frequently applied to analyze visual imagery due to its perks in processing data that has a grid pattern such as images. Inspired by the organization of animal visual cortex [31, 32], CNN is designed to be adaptive and automatic in learning spatial hierarchies of features, from low- to high-level patterns. Instead of the need for manually measuring and categorizing each individual feature, CNN gathers these features by itself. Consequently, CNN is more favorable in computer vision and image classification tasks than its predecessors.

3.1 Architecture overview

Generally, CNN has a similar architecture to ANN with three primary types of layers (input, hidden and output). However, hidden layers are assorted into subtypes according to the operation needed to perform.

A typical CNN has these layers: input layer, convolutional layer, pooling layer and fully connected layer, as demonstrated as follows:

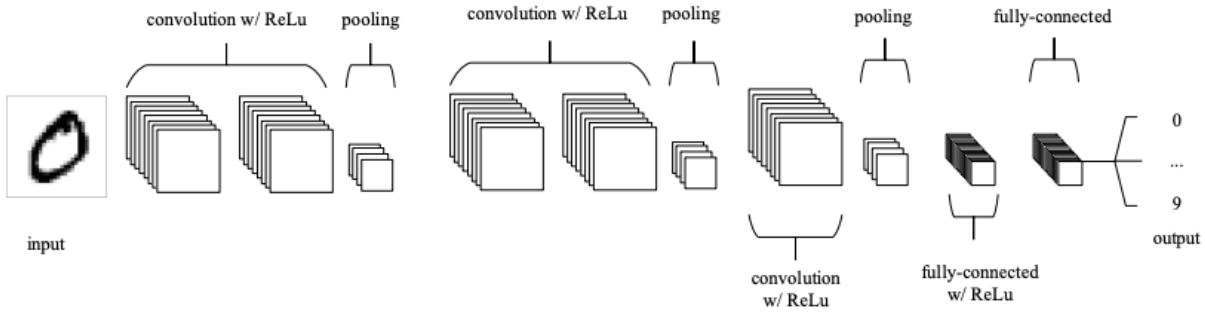


Figure 3.1: A convolutional neural network with 3 convolution layers followed by 3 pooling layers [33].

3.1.1 Input layer

As opposed to ANN, which relies on valid data inputs, CNN works in a compatible way with images as input data. Therefore, the input to the whole network is a pixel matrix of the image. An example of input can be a 28x28 grayscale image. The image does not need to be flattened to a 1-D array as in ANN, hence the input can be introduced to the network in 2-D as a 28x28 matrix. This makes capturing spatial relationships much easier.

3.1.2 Convolutional layer

The convolutional layer is the core building block of CNN. This layer conducts an operation called convolution. In mathematics, convolution is an operation on two functions f and g that produces a third function $f * g$ that expresses how the shape of one is modified by the other. In CNN context, the layer performs a dot product between the input and the filters, also known as kernel. The product is not applied on the whole input array but rather a patch of size of the filter, which has a small receptive field. Afterwards, the operation is carried out across the width and height of the array by "sliding" the filter from left to right and then top to bottom. In other words, each filter is convolved along the dimension on the input array. Since dot product always results in a single value (also referred as scalar product), the systematic application of the same filter over an image multiple times produces a two-dimensional array of output values that

represents the filtered input. This output array is known as a feature map. Once a feature map is obtained, it can be passed through an activation function such as ReLU, similar to normal NNs.

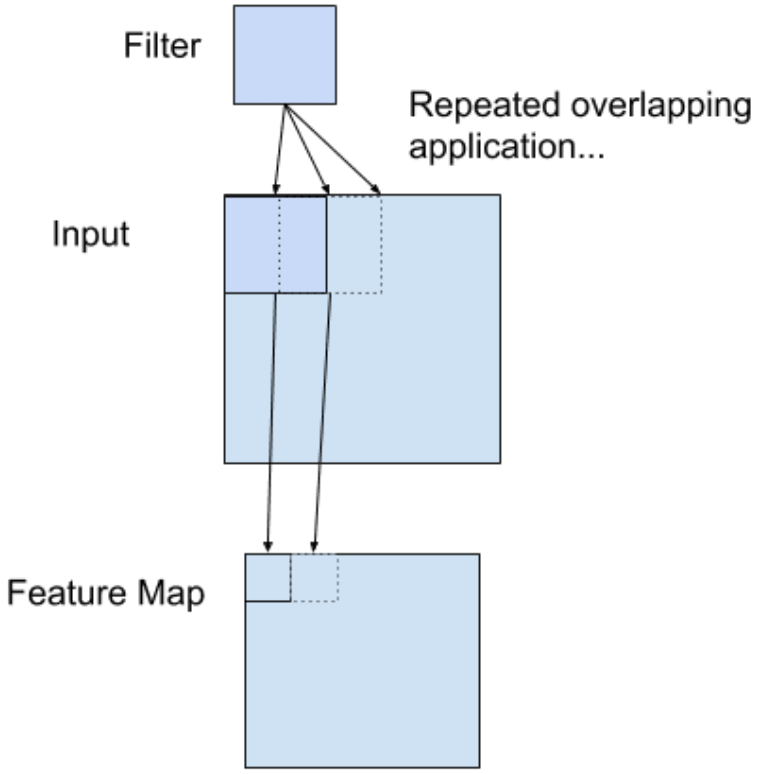


Figure 3.2: Example of a filter applied to a 2-D input to create a feature map [34].

As shown in Fig. 3.2, the size of the feature map is different from the input. It depends on several parameters including kernel size, stride and padding. Stride simply defines the number of jumping steps of the kernel over the image and padding indicates the number of rows and columns of zeros that has to be padded around the image. Zeros are added to avoid the loss of information at the edges of the image. The size of feature map is defined as the following equations:

$$\begin{cases} w_{out} = \dfrac{w_{in} - K_w + 2p}{s_x} + 1 \\ h_{out} = \dfrac{h_{in} - K_h + 2p}{s_y} + 1 \end{cases}, \tag{3.1}$$

where w_{out}, h_{out} stand for the output width and height of the feature map, w_{in}, h_{in} are the width and height of the input, K_w, K_h represent width and height of the kernel, p is the amount of padding for the edges of the input, s_x, s_y refer the stride value along x- and y-axis.

If a squared input image is considered, kernel is also a square and the strides are vertically and horizontally the same, Eq. (3.1) can be simplified as

$$N_{out} = \frac{N_{in} - K + 2p}{s} + 1, \tag{3.2}$$

where N_{out}, N_{in} are the sizes of output and input respectively, K is the kernel's size, p is the amount of padding for the edges of the input, s is stride.

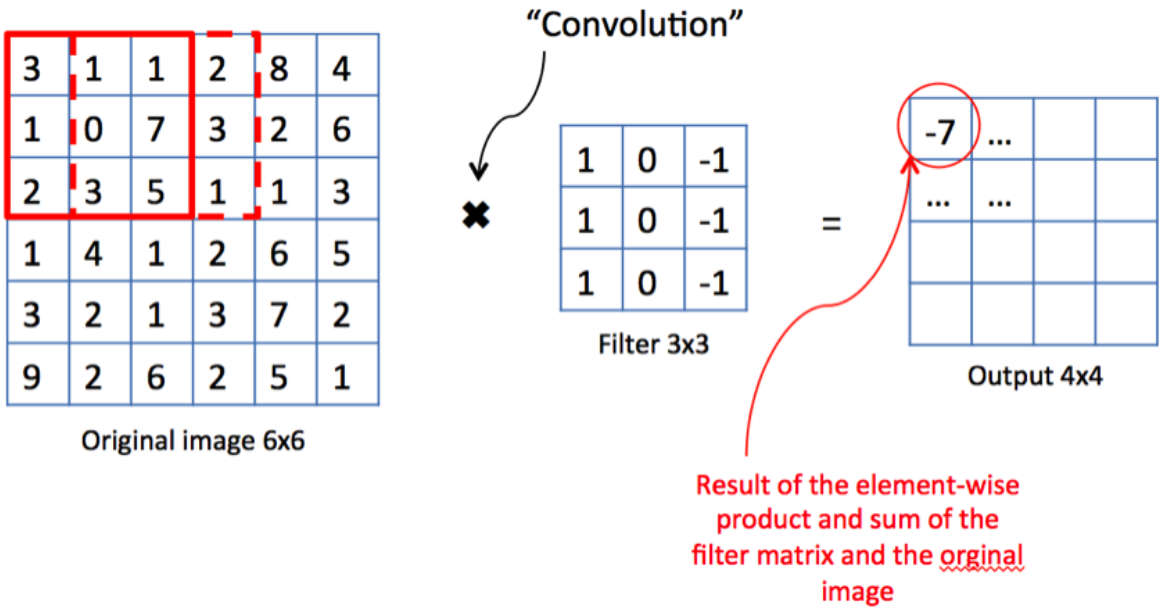


Figure 3.3: Example of convolutional operation in CNN [35].

For example, with $N_{in} = 6$, $K = 3$, $s = 1$ and $p = 0$ as shown in above figure, $N_{in} = \frac{6-3+0}{1} + 1 = 4$.

A common term that is related to CNN topics is "channel". Channel can also be referred to as the layer's depth. For instance, an RGB image has three channels: Red, Green and Blue. When a Convolutional Layer is created, as many filters can be specified as needed; and each

filter can be distinguished and retain different information. The number of filters will dictate the depth of the output or the number of channels the output will have.

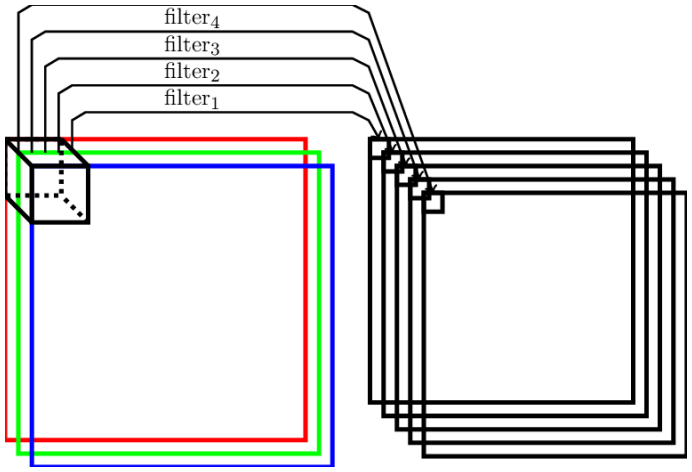


Figure 3.4: Four filters are used to create four feature maps [36].

Convolutional layer is not only applied directly on images or input data, but it can also be applied to the output of other convolutional layers. Simply put, convolutional layers are stackable. The lower hierarchical order of the layer, the lower-level features it can extract. Namely, the convolutional layer that operates directly on the raw pixel values will adapt to extract low level features such as lines. The latter convolutional layer that works on top of the previous layer may extract features that are combinations of lower-level features, such as shapes, which are made of lines. One thing worth noting is that the detail of features is not guaranteed as they depend on the network's adjustment during training. However, the abstraction of features from low to higher orders as depth of the network increased is certain.

3.1.3 Pooling layer

Pooling is another significant concept of CNN, which is a form of non-linear down-sampling. The pooling layer performs pooling to reduce the spatial dimensionality of the input. This decreases the quantity of parameters, consequently, shortens learning time and lessens computation power. There are several non-linear functions used to implement pooling but the most common is max pooling. Usually, a 2x2 filter with a stride of 2 that outputs the maximum value as it slides over the input image. This filter discards 75% of the activations passed through

it and every max pooling operation would take in a max over four numbers. The depth remains unchanged.

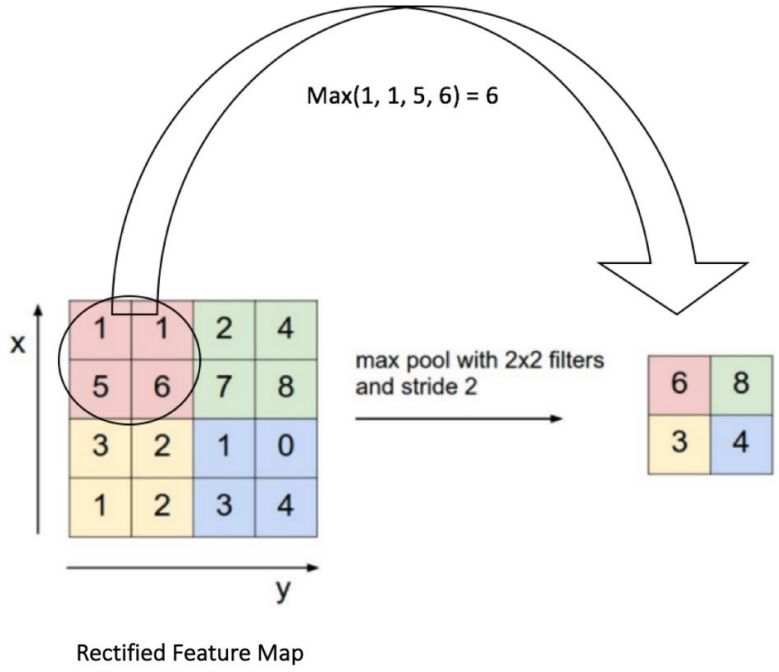


Figure 3.5: Max pooling of size 2x2 filter [37].

If the dimension of input and output matrices are denoted as $W_1 \times H_1 \times D_1$ and $W_2 \times H_2 \times D_2$ respectively, where W is width, H is height and D is depth, the output dimension can be calculated as below:

$$\begin{cases} W_2 = \dfrac{W_1 - K_w}{s_x} + 1 \\ H_2 = \dfrac{H_1 - K_h}{s_y} + 1 \\ D_2 = D_1 \end{cases}. \tag{3.3}$$

3.1.4 Fully connected layer

It is the same as a traditional ANN, all the nodes from the previous layer are connected to this fully connected layer, which is in charge of making prediction or classification of the image. Typically, the input of this layer is flattened.

3.2 Hyperparameters

In addition to the hyperparameters of a conventional ANN, CNN can have more hyperparameters:

- Kernel size: Dimension of a kernel. Commonly used kernels are 2×2 and 3×3.
- Padding: Number of rows or columns of zeros on the border of the image to prevent data loss at the edges. Two types used primarily are valid and same padding.
- Strides: Number of pixels that the sliding window (kernel) moves at a time. The kernel can move at different paces on x- and y-axis but usually it is the same amount.
- Number of filters/kernels: The depth of a layer, also referred to as channels. It dictates the number of feature maps output at each layer. The layers near the input layer tend to have fewer filters while further layers can have more.
- Pooling type and size: Max pooling, often with dimensions of $2x2$, is typically employed. Besides, min pooling or average pooling can also be used.
- Dilation: Within a kernel, certain pixels can be ignored hence reducing even more memory/processing power needed without significant loss of the signal.

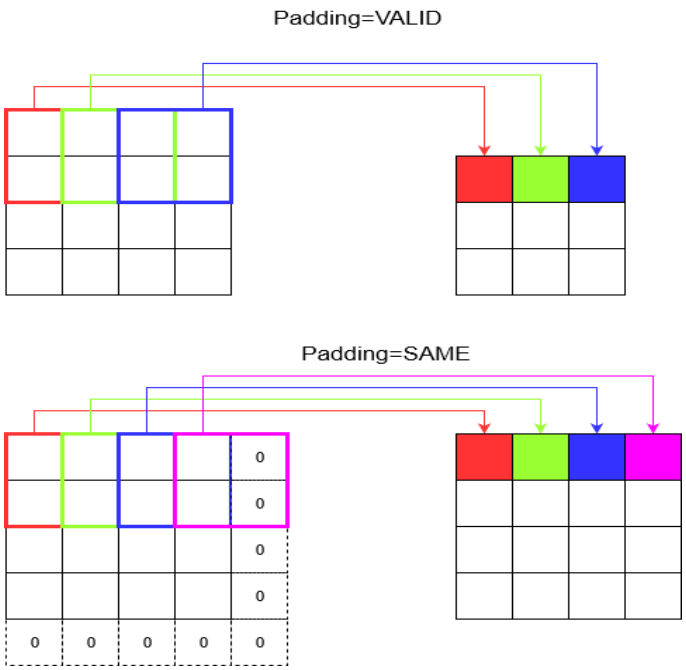


Figure 3.6: Two types of padding.

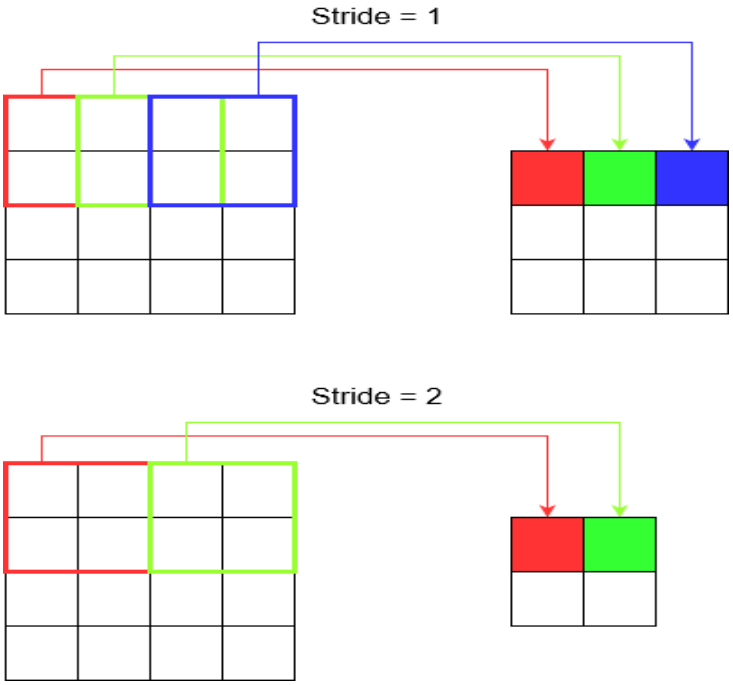


Figure 3.7: Stride of 1 and stride of 2.

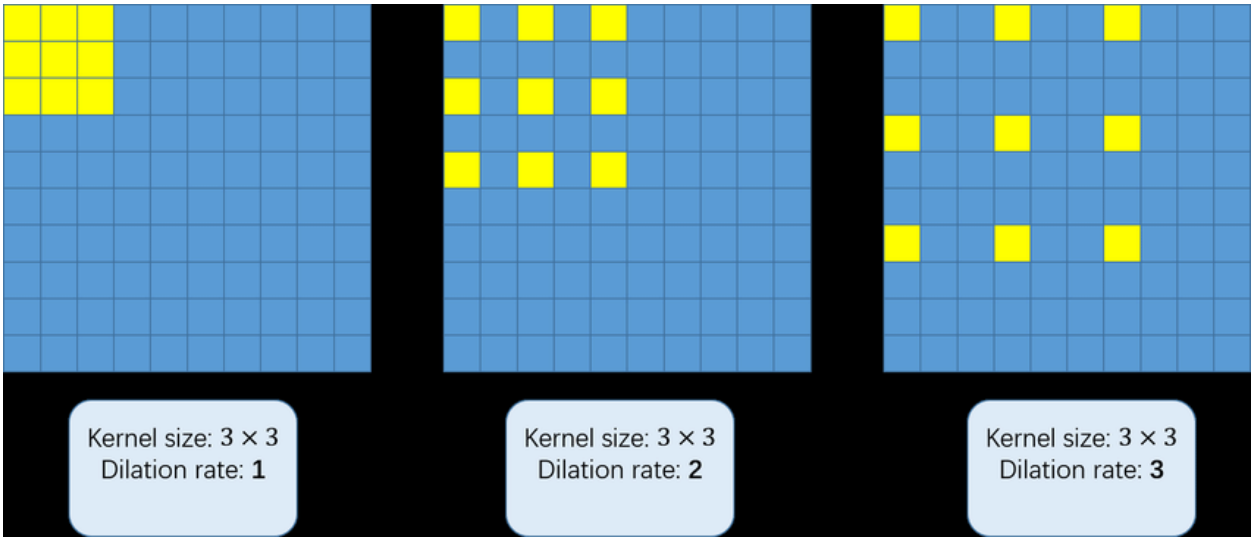


Figure 3.8: 3 × 3 convolution kernels with different dilation rate as 1, 2, and 3 [38].

Chapter 4: A CNN-based approach to HPE

The network used in this thesis is an implementation of the Stacked Hourglass architecture [39]. It consists of a series of stacked U-Nets [40] that resemble hourglass shape; hence it is called Stacked Hourglass Network (SHN). Unlike typical U-Net structures where unpooling or deconvolutional layers are used during the decoder stage, this network simply employs nearest neighbor upsampling. Additionally, feature levels of the same spatial resolutions are combined in the encoding and decoding stages. The published paper is a landmark that introduced a novel and intuitive CNN architecture and showed superiorities over its predecessors in HPE.

4.1 Residual block

The superiority of NNs is the capability to capture abstract features as information propagates deeper into the network. However, deep networks more often than not suffer from vanishing/exploding gradient due to the loss of data during propagation and performance degradation. As the networks grow in depth, accuracy saturates and stumbles. So as to overcome this problem Residual Neural Networks (or ResNets) [41] were introduced. Fundamentally, this network architecture consists of a stack of “residual blocks”, which is also the building block of SHN. The utmost importance of a residual block is the skip connection or residual connection. This connection happens every two or three convolutional layers and creates “highways” information that enables the flow of information from earlier parts of the network to the deeper part of the network. In other words, they help maintain the signal propagation which allows networks to have more layers or become “deeper”.

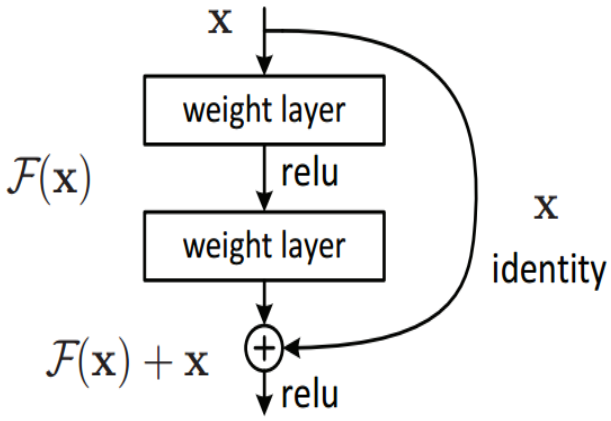


Figure 4.1: Residual block of a ResNet [41].

Instead of learning a mapping function $\mathcal{F}$ that maps input x to the output $\mathcal{F}(x)$, only a small offset or residuum $\mathcal{F}(x)$ of the identity is learned which leads to the output $\mathcal{F}(x) + x$.

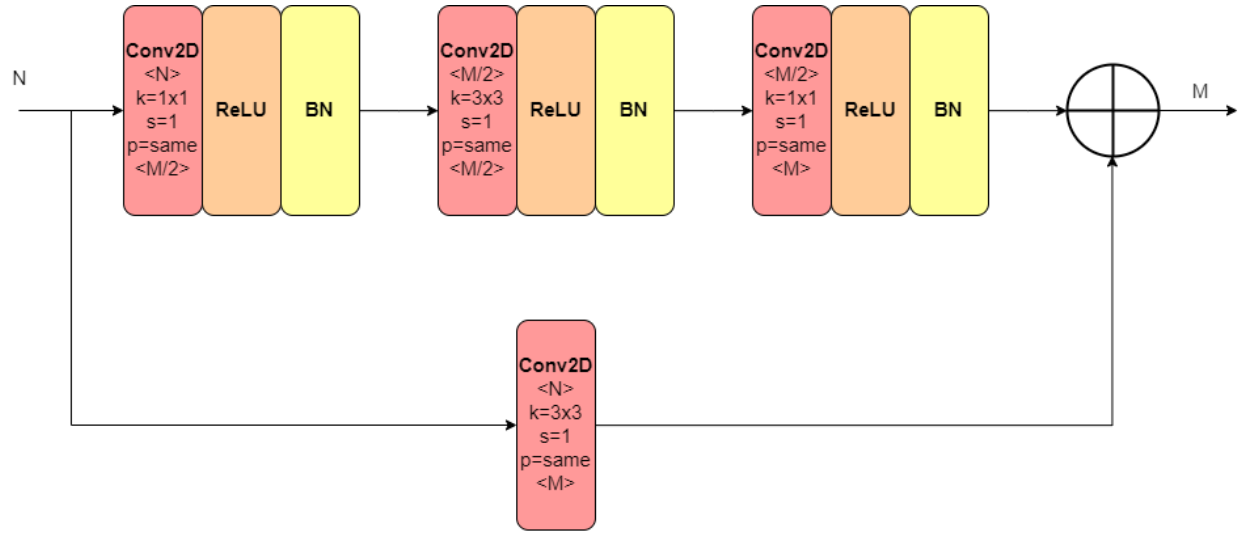


Figure 4.2: Architecture of a residual block.

Figure 4.2 shows there are four distinct operations:

- **Conv2D**: is a 2D convolution operation where $< N >$ is the number of input filters, $< M >$ is the number of output filter, k is kernel size, s is stride and p is type of padding.
- **ReLU:** is ReLU activation layer.
- **BN:** is Batch Normalization. It is a technique for training very deep neural networks that standardizes the inputs to a layer for each batch. This has the effect

of stabilizing the learning process and dramatically reducing the number of training epochs required to train deep networks.

- **Add (⊕):** is an addition operation. It expects all incoming input should be in the same number of filters. For example, input 1 (256,256,256) cannot be added with input 2 (256, 256,128). For that reason, the skip connection (lower branch) has a Conv2D operation to equalize the number of filters of the input and the number of filters of the output.

Resolution of the input is unchanged (width and height), only the number of filters is modified.

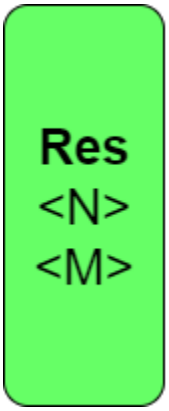


Figure 4.3: Residual block with <N> input filters and <M> output filters.

4.2 Hourglass module

Hourglass module consists of a number of residual models, a few max pooling and up scaling layers. Filters with size greater than 3×3 are never used inside the hourglass module to limit the number of learnable parameters hence reducing memory usage. The architecture is capable of capturing information at every scale. While features like faces or hands are bound to their local spaces, which means local information is essential to pinpoint where they are, a human pose estimation requires a comprehension of the full body. The person's orientation, limbs' arrangement or relationship of adjacent joints (e.g., wrist and elbow) are one of many cues that needed to be captured at different scales of the image. The hourglass architecture is not only able to break down the image into smaller scales and gather local information, but also capable

of assembling the captured features in a later stage with the addition of skip connections which induce information from previous layers.

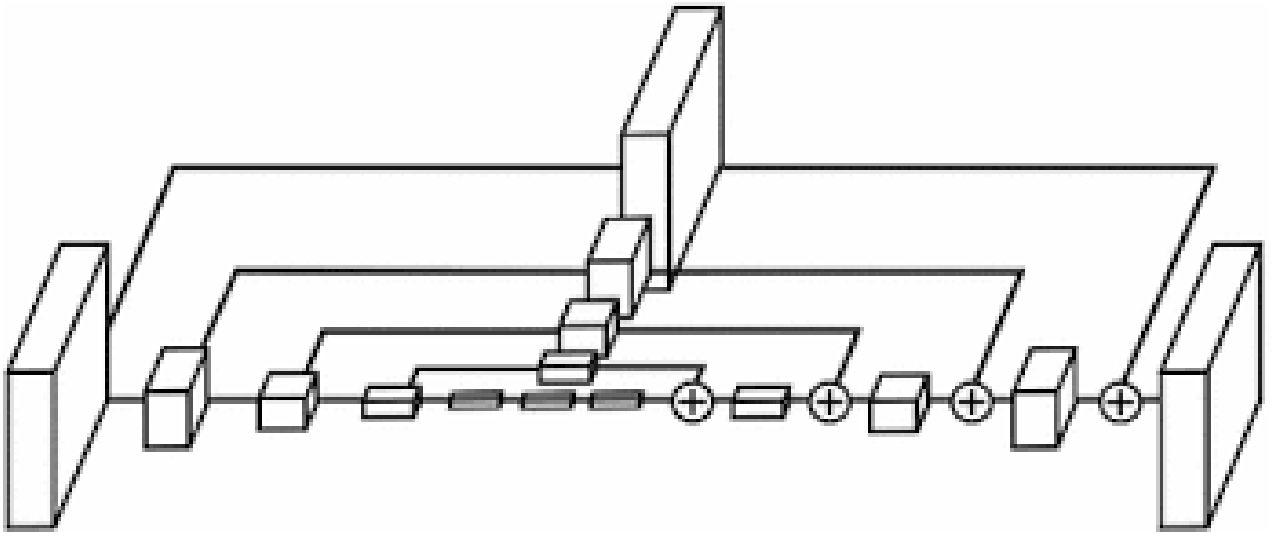


Figure 4.4: An illustration of hourglass module, each box is a residual module [39].

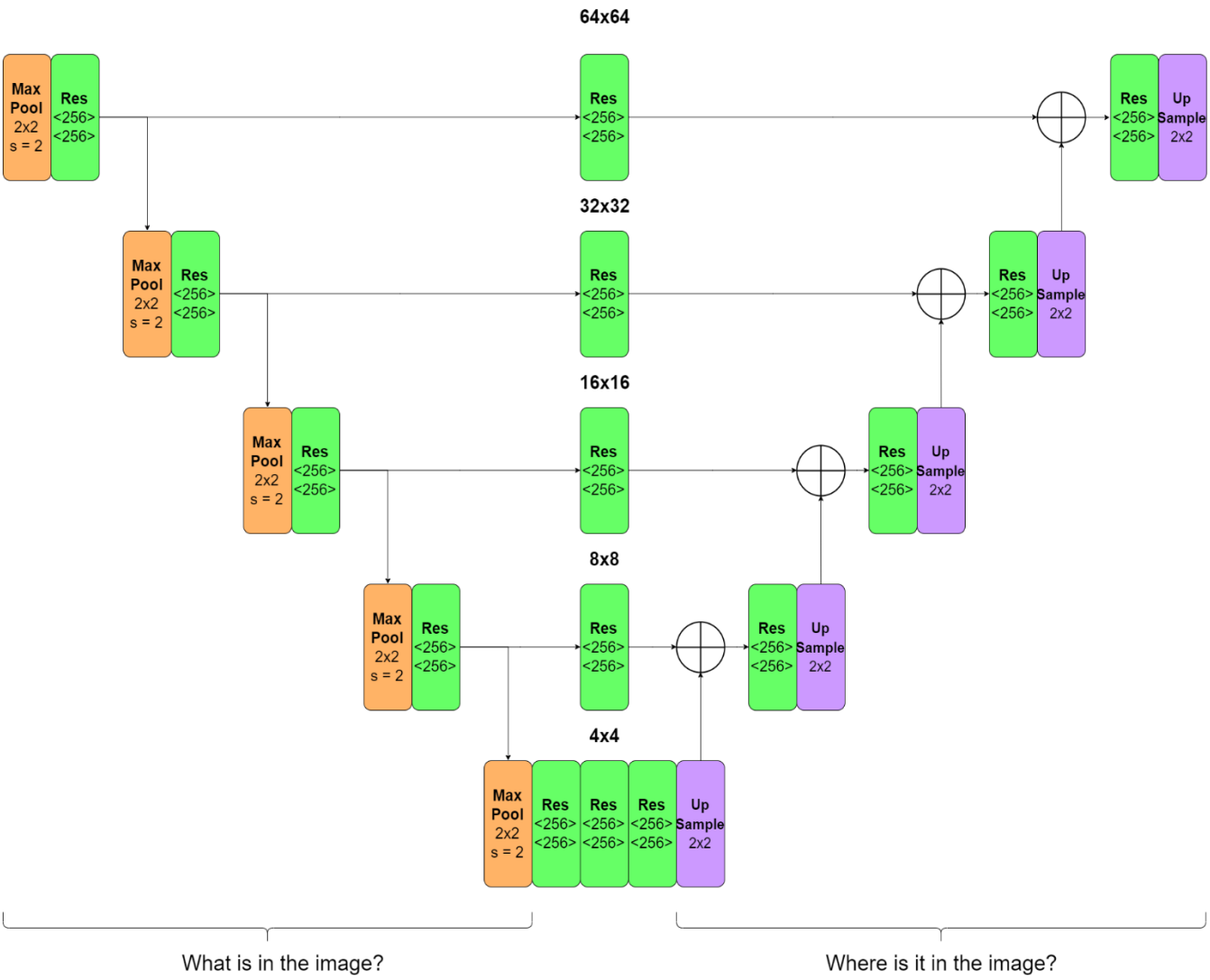


Figure 4.5: Hourglass module in details.

As seen in Fig. 4.5, the number of output and input filters of residual modules (green) stays consistent at 256. The height and width of input is halved after every max pooling (orange) operation with the pool size of 2×2 and stride $s = 2$. At the lowest level the network reaches its lowest resolution of 4×4 pixels. After that a number of 2×2 up scaling (purple) operations are performed to restore the initial height and width. Additionally, at the end of every upscaling, previous information from the corresponding spatial resolution is combined with current information. All upscaling operations use nearest neighbor interpolation.

4.3 Stacked hourglass modules with intermediate supervision

Before images are being fed into the model, they are all resized into 256×256, which still requires a significant amount of memory to process. In order to ease the computational power, the input is passed through a front module prior to the hourglass module. It starts with a 7×7 convolutional layer with stride of 2 to reduce the resolution in half. It is then further downsized to 64×64 by a max pooling layer.

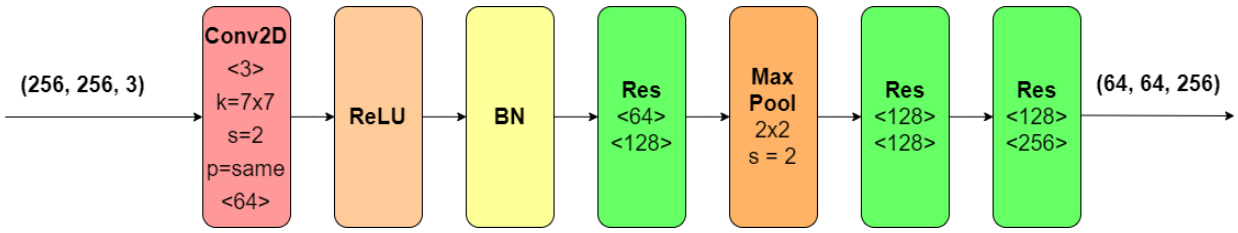


Figure 4.6: Front module in detail

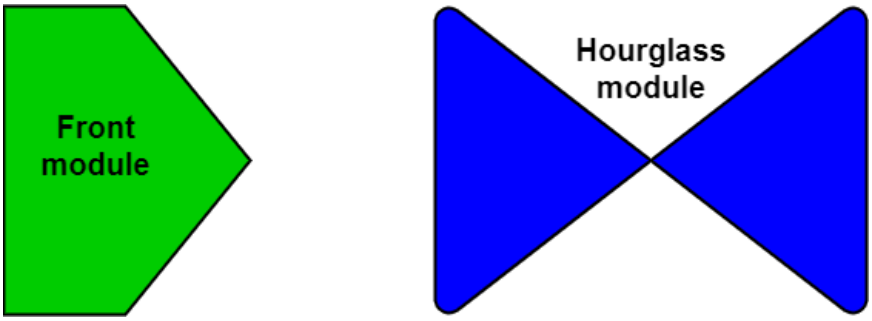


Figure 4.7: Front module and hourglass module.

In order to further refine the result, the hourglass network can be stacked with others. Output of one hourglass is the input for the next one. This provides the network with a mechanism for repeated bottom-up, top-down inference allowing for reevaluation of initial estimates and features across the whole image [39]. Moreover, at each stage before the output of the previous stage is fed into the next hourglass, it can be assessed by applying a loss. In other words, the network makes intermediate predictions at each stage instead of making one prediction at the very end of the network. This is called intermediate supervision. Predictions are generated after each hourglass which allows high level features to be processed again and further evaluated. Improvement in the previous stage leads to a better outcome in the latter stage.

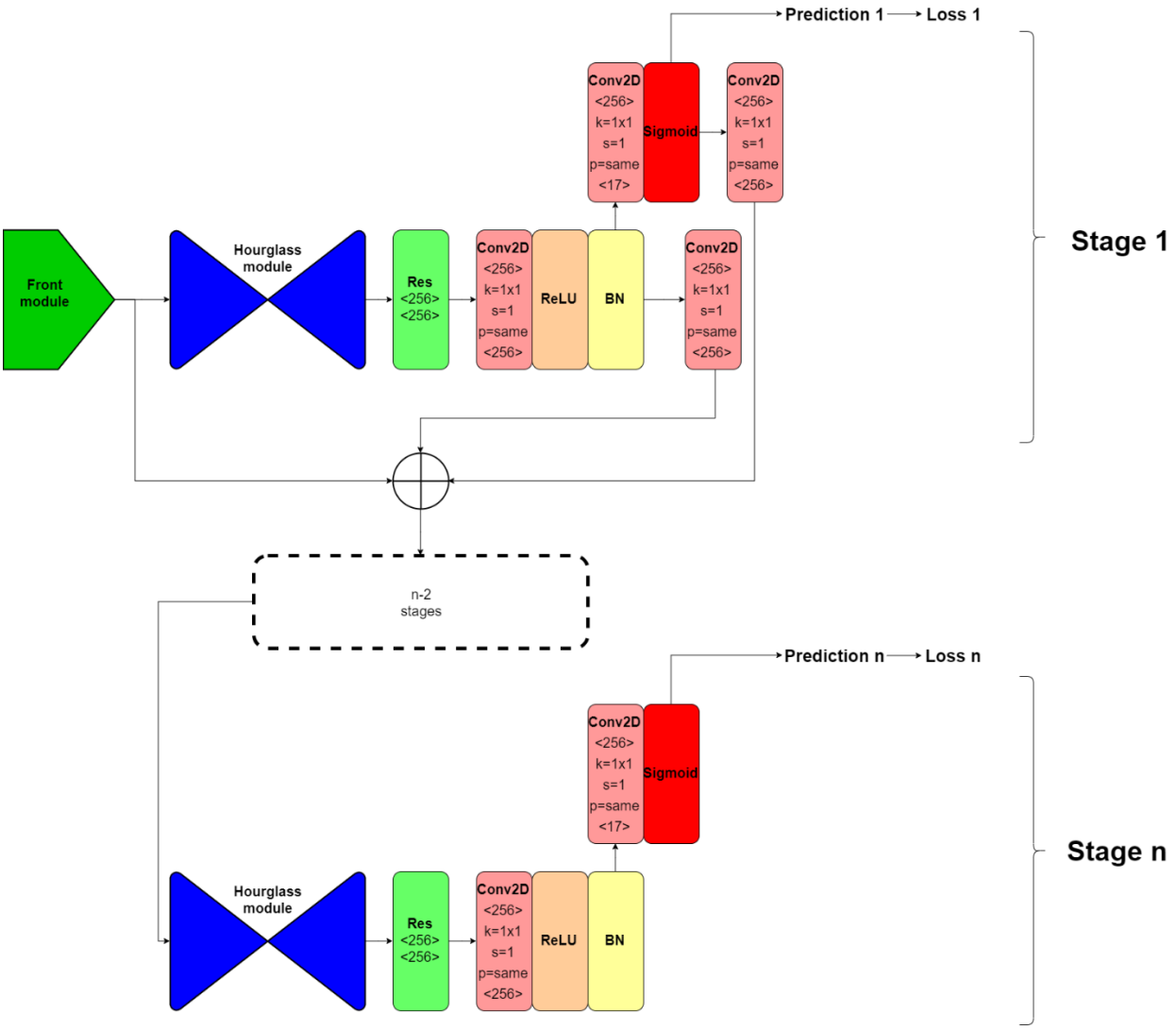


Figure 4.8: Stacked Hourglass Network.

At each stage, prediction is made by a Conv2D block with 17 output filters (COCO dataset has 17 keypoints for each person) followed by a sigmoid activation layer to map all values to $[0,\ 1]$ range. The prediction of each stage has the shape of $(64, 64, 17)$. Losses are computed at each stage by comparing the predictions and ground truth labels. If it is not the final stage, the prediction is then remapped by a 1×1 Conv2D and added back along with the input and output of the stage. This sum is the input of the next hourglass in the next stage.

Chapter 5: Dataset and data preparation

5.1 COCO dataset

5.1.1 Overview

The Common Object in Context (COCO) is one of the most popular large-scale labeled image datasets available for public use [21]. It was published by Microsoft and was used in multiple competitions for various Computer Vision tasks. There have been a few versions over the years but the latest one is the 2017 version which contains image annotations in 80 categories with over 1.5 million instances.

person	fire hydrant	elephant	skis	wine glass	broccoli	dining table	toaster
bicycle	stop sign	bear	snowboard	cup	carrot	toilet	sink
car	parking meter	zebra	sports ball	fork	hot dog	tv	refrigerator
motorcycle	bench	giraffe	kite	knife	pizza	laptop	book
airplane	bird	backpack	baseball bat	spoon	donut	mouse	clock
bus	cat	umbrella	baseball glove	bowl	cake	remote	vase
train	dog	handbag	skateboard	banana	chair	keyboard	scissors
truck	horse	tie	surfboard	apple	couch	cell phone	teddy bear
boat	sheep	suitcase	tennis racket	sandwich	potted plant	microwave	hair drier
traffic light	cow	frisbee	bottle	orange	bed	oven	toothbrush

Figure 5.1: COCO categories [42]

The tasks that COCO dataset is used for are Object Detection, Instance Segmentation, Image Captioning, Panoptic Segmentation, Dense Pose, Stuff Image Segmentation and Keypoints Detections. The annotations are stored using JSON (JavaScript Object Notation) format [43]. All notations regarding the task have the same basic data structure as below:

```
{
   "info"            : info,
   "images"          : [image],
   "annotations"     : [annotation],
   "licenses"        : [license],
}

info{
   "year"            : int,
   "version"         : str,
   "description"     : str,
   "contributor"     : str,
   "url"             : str,
   "date_created"    : datetime,
}

image{
   "id"              : int,
   "width"           : int,
   "height"          : int,
   "file_name"       : str,
   "license"         : int,
   "flickr_url"      : str,
   "coco_url"        : str,
   "date_captured"   : datetime,
}

license{
   "id"              : int,
   "name"            : str,
   "url"             : str,
}
```

Figure 5.2: COCO basic data structure [44]

The fields in basic data structure are not so essential for the chosen task, what are more important are the specific fields given for each task.

5.1.2 Keypoints Detection data format

Besides basic data structure as mentioned above, a keypoint annotation contains the following fields:

```
annotation{
  "keypoints": [x1,y1,v1,...],
  "num_keypoints": int,
  "id": int,
  "image_id": int,
  "category_id": int,
  "segmentation": RLE or [polygon],
  "area": float,
  "bbox": [x, y, width, height],
  "iscrowd": 0 or 1,
}

categories[{
  "keypoints": [str],
  "skeleton": [edge],
  "id": int,
  "name": str,
  "supercategory": str,
}]
```

Figure 5.3: Annotation format for Keypoint Detection

The fields in annotation are as follow:

1. **"keypoints"**: an array with $3k$ length where k is the maximum number of keypoints one person has (for COCO dataset $k = 17$). Each keypoint has the coordinates of (x_i, y_i) and a visibility flag denoted as v_i. If a keypoint is not labeled (in which case $x_i = y_i = 0$), v_i will be 0. If a keypoint is labeled but not visible (occluded), $v_i = 1$. Lastly $v_i = 2$ if the keypoint is labeled and visible.
2. **"num_keypoints"**: number of labeled keypoints ($v_i > 0$).
3. **"id"**: annotation id, a unique integer for each instance. An instance can be a person or a crowd of people.
4. **"image_id"**: image id, a unique integer for each image. One image may contain multiple instances, hence multiple annotations.
5. **"category_id"**: for Keypoints task it is 1 as for "person" category.
6. **"segmentation"**: not used.
7. **"area"**: not used.

8. **“bbox”**: bounding box around the person/instance. It consists of four floating point numbers. (x, y) are the top left corner coordinates and $(width, height)$ of the box.
9. **“is_crowd”**: either 0 or 1. It indicates if this annotation is a crowd which also means there are no keypoints if it is.

The fields in categories as follow:

1. **“keypoints”**: a list of keypoint names including “nose”, “left_eye”, “right_eye”, “left_ear”, “right_ear”, “left_shoulder”, “right_shoulder”, “left_elbow”, “right_elbow”, “left_wrist”, “right_wrist”, “left_hip”, “right_hip”, “left_knee”, “right_knee”, “left_ankle”, “right_ankle”. A total of 17 keypoints and all annotations have this same list and are in the same order.
2. **“skeleton”**: a list of connections or bones between keypoint indices (from 1 to 17 of the above list). For example, $[16, 14]$ means “left_ankle” connects with “left_knee”. The full list is: $[16, 14]$, $[14, 12]$, $[17, 15]$, $[15, 13]$, $[12, 13]$, $[6, 12]$, $[7, 13]$, $[6, 7]$, $[6, 8]$, $[7, 9]$, $[8, 10]$, $[9, 11]$, $[2, 3]$, $[1, 2]$, $[1, 3]$, $[2, 4]$, $[3, 5]$, $[4, 6]$, $[5, 7]$.
3. **“id”**: category id, for Keypoints Detection it is 1.
4. **“name”**: category name “person”.
5. **“supercategory”**: “person”.

Index	Key point
0	Nose
1	Left-eye
2	Right-eye
3	Left-ear
4	Right-ear
5	Left-shoulder
6	Right-shoulder
7	Left-elbow
8	Right-elbow
9	Left-wrist
10	Right-wrist
11	Left-hip
12	Right-hip
13	Left-knee
14	Right-knee
15	Left-ankle
16	Right-ankle

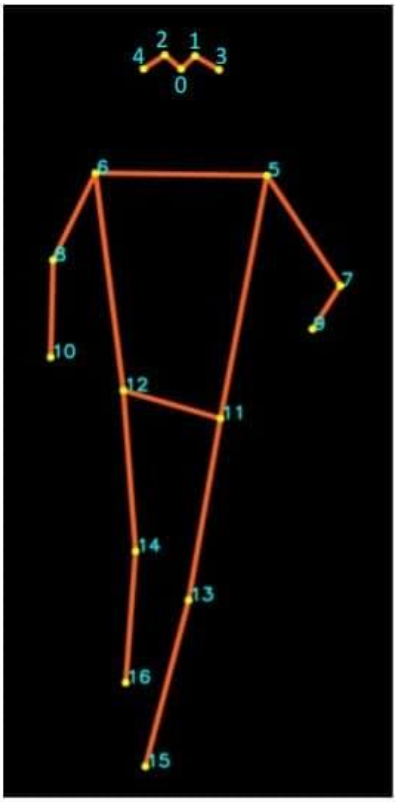


Figure 5.4: COCO keypoint annotations (zero-indexed) [45].

5.1.3 Deep dive into COCO dataset

For the 2017 COCO dataset there are several annotation files for different tasks but only two are needed:

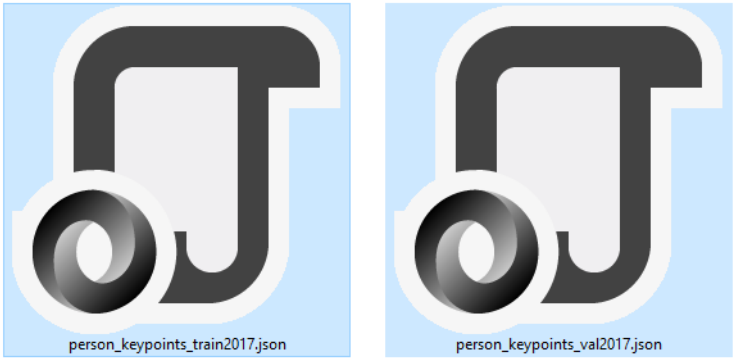


Figure 5.5: Annotation files

These annotation files are specifically used for Keypoints Detection task. By using the built-in python json library, annotations can be read from these files. All annotations have the same format as mentioned in Section 5.1.2. From the extracted data, further processing can be conducted before using for training, testing and evalution.

5.1.3.1 Number of instances/annotations per image

For both train2017 and val2017 there are a total of 273,469 annotations/instances and 66,808 images. Since the COCO dataset contains images with multiple annotations, it is good to examine the distribution of images with a specific number of annotations.

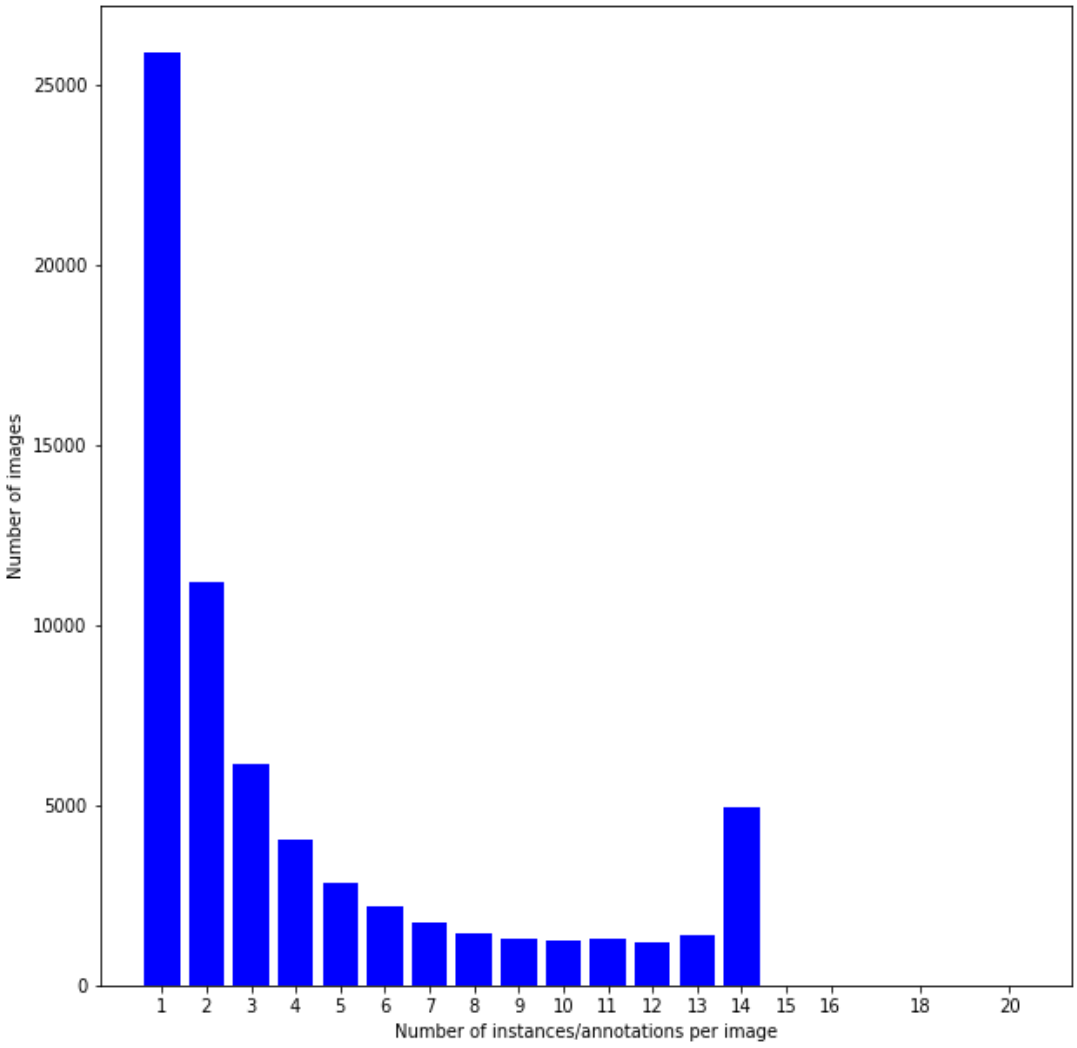


Figure 5.6: Distribution of images with specific numbers of annotations.

As seen in Fig. 5.6, a large number of images contain only one annotation. It takes up approximately 38,73% of the total number of images. However, not every image that has only one annotation or even more annotations has people with keypoints. One of the reasons is that some annotations come with a **"is_crowd"** flag that indicates a crowd annotation which has no keypoints. The other reason is that the person just has no keypoints annotated.

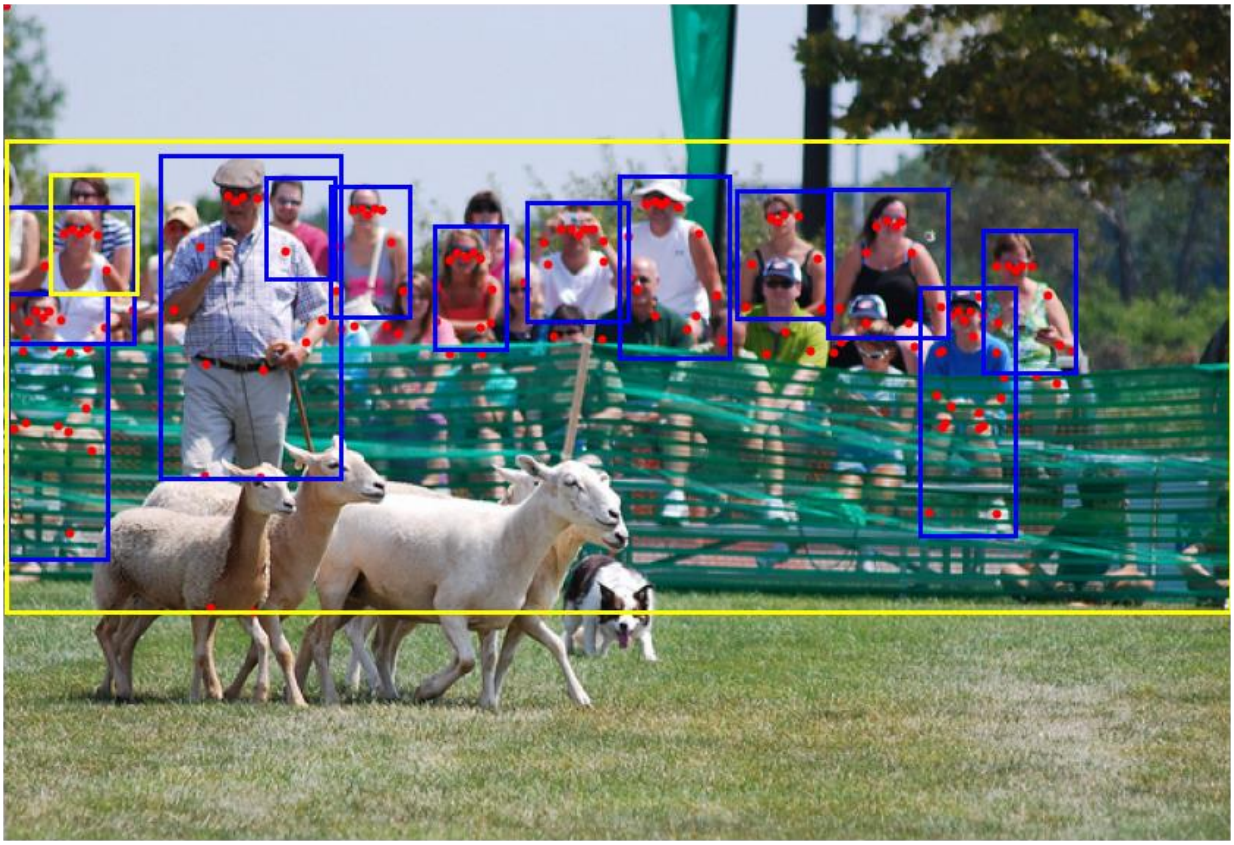


Figure 5.7: Visualize bounding boxes and keypoints.

In Fig. 5.7, two yellow boxes have no keypoints but the bigger one is a crowd annotation, and the other is not. The blue boxes all contain keypoints yet some of the keypoints are out of bound or invisible ($x = y = 0$, topleft corner). Moreover, the boxes are in different sizes and aspect ratios. Hence, before it can be used the annotation needs to be preprocessed.

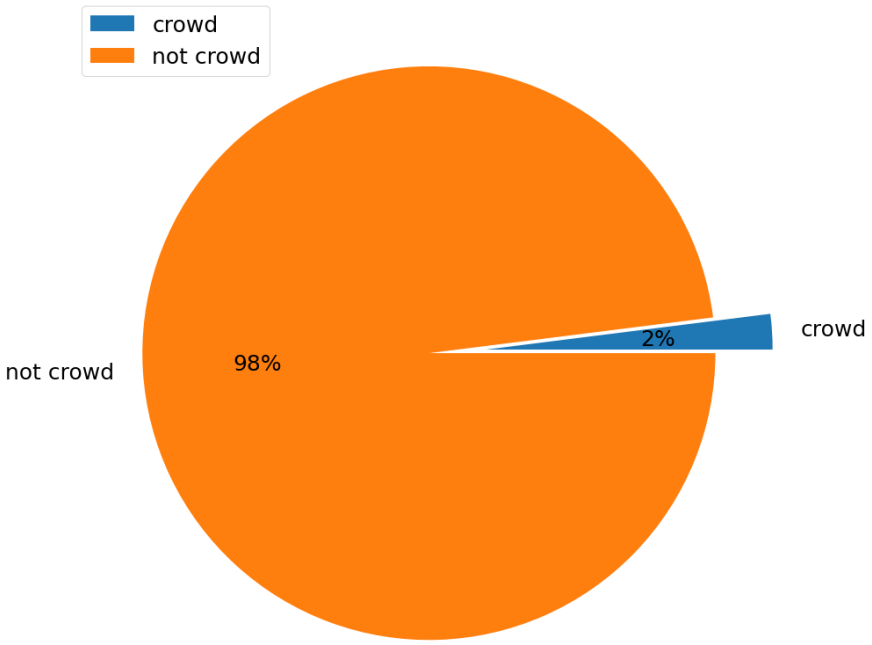


Figure 5.8: Number of crowd vs non crowd annotations.

5.1.3.2 Number of keypoints

Another metric that is good to look at is the number of keypoints per bounding box. Only bounding boxes that are not marked as crowd are considered since crowd is just a group of people with no keypoints.

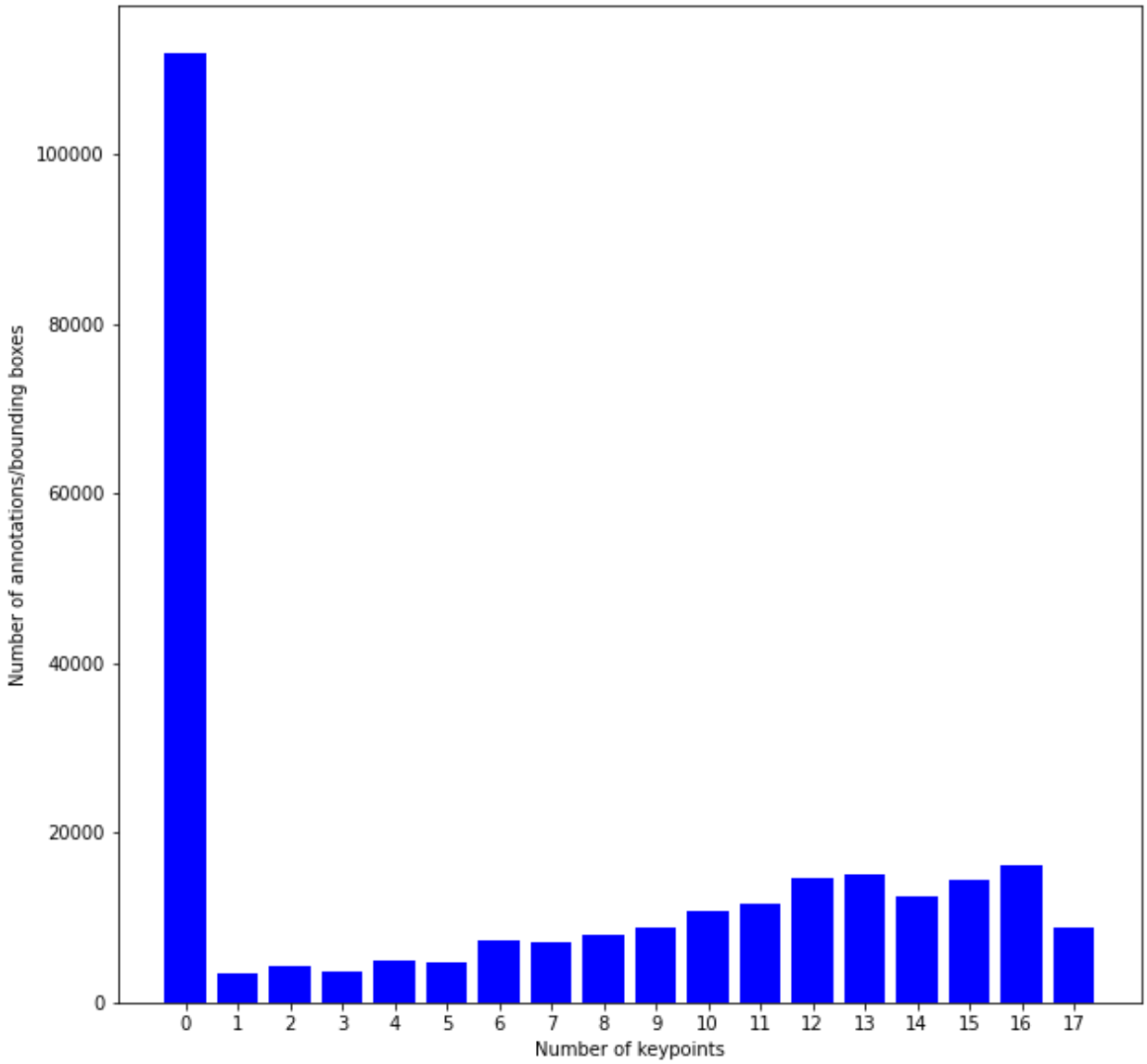


Figure 5.9: Number of keypoints distribution.

As can be seen from Fig. 5.9, the large portion of annotations do not have any keypoints. This can give rise to some issues when training the model since these annotations can act as noises and confuse the NN model. Not only that, but if the number of keypoints is too low, it can still be difficult for the model to learn.

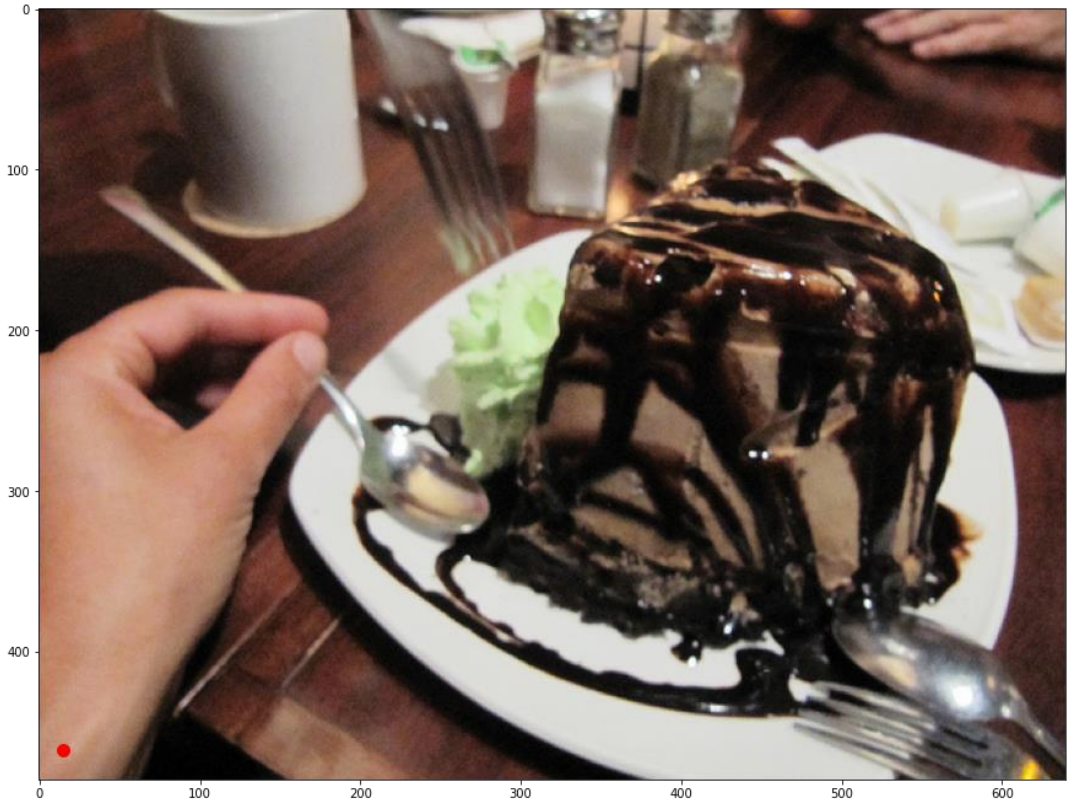


Figure 5.10: Annotation with only one wrist annotation (red).

5.1.3.3 Bounding box scales

Information about the scale of a bounding box is very useful since the occurrences that are too small may be discarded or upscaling needs to be performed.

Scale	*Bounding box area*	*Percentage (%)*
S (Small)	$0 < area < 32^2$	30.75
M (Medium)	$32^2 \leq area < 64^2$	21.59
L (Large)	$64^2 \leq area < 96^2$	11.96
XL (Extra-large)	$96^2 \leq area < 128^2$	8.08
XXL (Extra extra-large)	$area \geq 128^2$	27.61

Table 5.1: Scales of bounding box.

5.2 Data preprocessing

5.2.1 Filtering annotations

Before the data can be used for either training, testing or validation, a portion of the data is sieved so as to improve accuracy in the long run. Firstly, all annotations with crowd flag are eliminated since crowd carries no keypoints. Secondly, only annotations with the minimum of 5 keypoints are used. Even though the NN model should be expected to produce output with a dynamic number of keypoints, a human with only 0 to 4 keypoints carries not enough information to clearly make out a person. Furthermore, 5 is chosen intuitively because it is the smallest cluster of keypoints that form the human head (see Fig. 5.4). As a result, the bounding box with a number of keypoints greater than 5 will always has an area bigger than the person's head area. Lastly, all S-scale bounding boxes are dropped too.

	Train	**Valid**	**Total**
Before filtering	262,465	11,004	273,469
After filtering	134,214	5,647	139,861
Reduced by (%)	48,86%	48,68%	48,86%

Table 5.2: Before and after filerting.

5.2.2 Images

5.2.2.1 Cropping

Images from the COCO dataset contains multiple people, but the main focus of the thesis is single Human Pose Estimation. Therefore, an image of a person is cropped out from the original image using bounding box information. Nonetheless, there are some problems.

Firstly, because the bounding boxes are in different scales (see Section 5.1.3.3), an upscaling needs to be performed in order to make all cropped images the same size.

Secondly, despite the need for upscaling, the width and height ratio of each box is very different to one another. Subsequently, the bounding boxes need to be modified so as to achieve uniform upscaling.

Thirdly, for the provided bounding box annotations, some of the keypoints are actually out of bound (see Fig. 5.7).

So as to tackle all the above issues, some adjustments are used to transform the original annotations. Before cropping, the bounding box of an annotation is converted to a square if it is not. This is done by using the longer side of the box as the side of the new square. Then, the square is increased by 25% to minimize the scenario where the keypoints are out of bound.

Figure 5.11: Old (blue) vs new (black) bounding box.

5.2.2.2 Padding

There also exists situations where the person is too close to the border of the image. As a result, only part of the bounding box is actually cropped. This can lead to inconsistent cropping results.

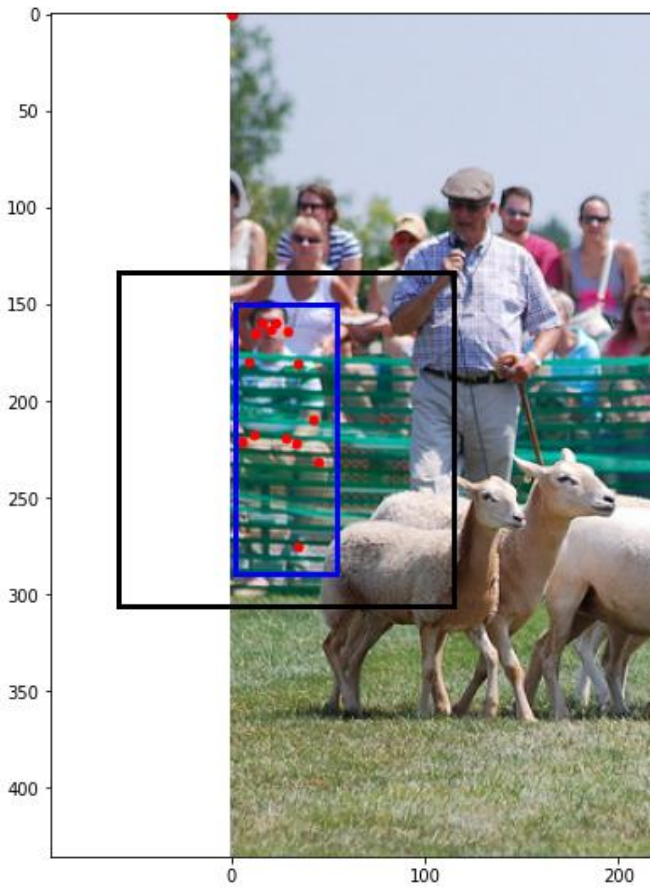


Figure 5.12: The adjusted bounding box (black) is outside of the image.

Moreover, if the annotation is only cropped to the border of the image, the person in Fig. 5.12 is actually off center. For the purpose of keeping the human in the frame in the center and with consistent scaling, padding is needed. In a word, for cases where the new bounding box is actually outside of the image, the image is cropped within the original width and height, then it is padded with zeros.

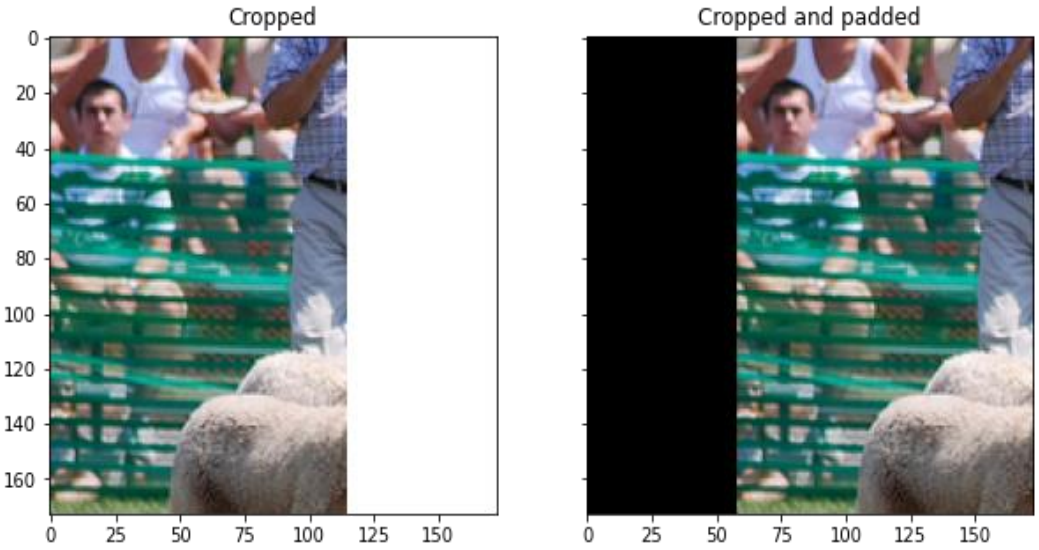


Figure 5.13: Only cropping vs cropping and padding

5.2.2.3 Scaling/Resizing

After being cropped and padded, all new images are resized to $(256, 256)$. Resizing an image can be tricky since in case of upscaling, the image needs to be filled with new pixels and in the case of downscaling, the information of the initial image is reduced. Either way, an image size can be changed in several ways by utilizing different pixel interpolation algorithms including nearest-neighbor, bilinear, bicubic, sinc (Lanczos), etc. [46]

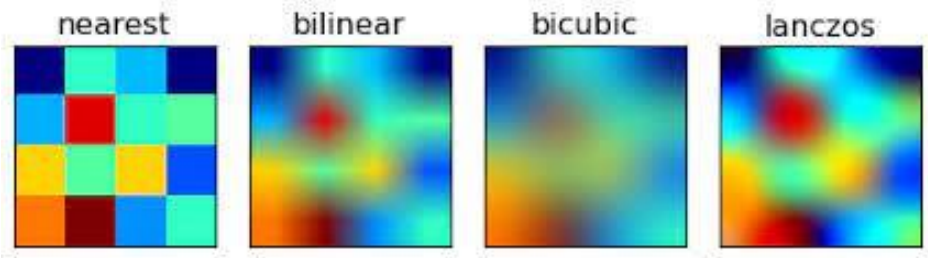


Figure 5.14: Comparision between differen interpolation algorithms.

However, for HPE geometric information of the human in the frame is actually more essential, hence the interpolation methods do not have much effect on the outcome or at least there has not a paper that implies such improvement. For this thesis, only the nearest-neighbor method is used since it is the default method of most resizing functions of different libraries.

Nearest-neighbor interpolation, sometimes known as proximal interpolation, is the most straightforward and common approach to enlarge an image. The algorithm simply replaces every pixel with the nearest pixel in the output, which means for upsizing multiple pixels will have the same value or color. This allows fast computation and sharper details, however, also results in jagged edges in previously smooth images. It is worth noting that "nearest" does not necessarily refer to the mathematical nearest. A common approach is to round towards zero so as to reduce artifacts in the final image.

The algorithm can be demonstrated as follows:

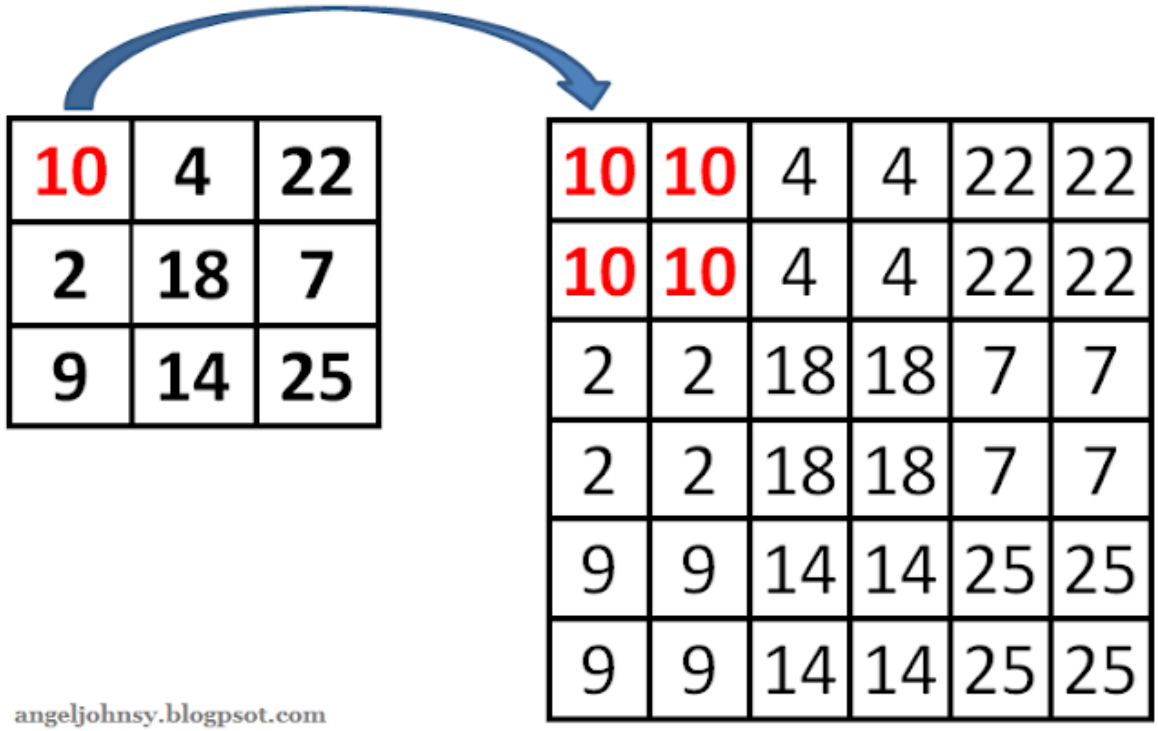


Figure 5.15: Nearest-neighbor interpolation [47].

5.2.2.4 Pixel scaling

All images provided from COCO dataset are stored in JPG format and are RGB images. JPG or JPEG (Joint Photographic Experts Group) is a digital image format which contains compressed image data. Images with this format are very compact due to the 10:1 compression ratio. It is the most common image format for sharing photos on the internet.

RGB (Red Green Blue) images are represented as a 3D array $(width, height, 3)$. Each pixel's color of an RGB image is determined by the combination of the red, green and blue intensities stored in each respective color plane (channel) at the pixel's location. RGB images have 24-bit format, where red, green and blue components are 8 bits each. Therefore, a single channel's value ranges from 0 to 255. For example: pure red is (255, 0, 0), pure green is (0, 255, 0), pure blue is (0, 0, 255), white is (255, 255, 255), black is (0, 0, 0), etc. Due to this format, there are over 16 million (2^{24}) potential colors for an individual pixel [48].

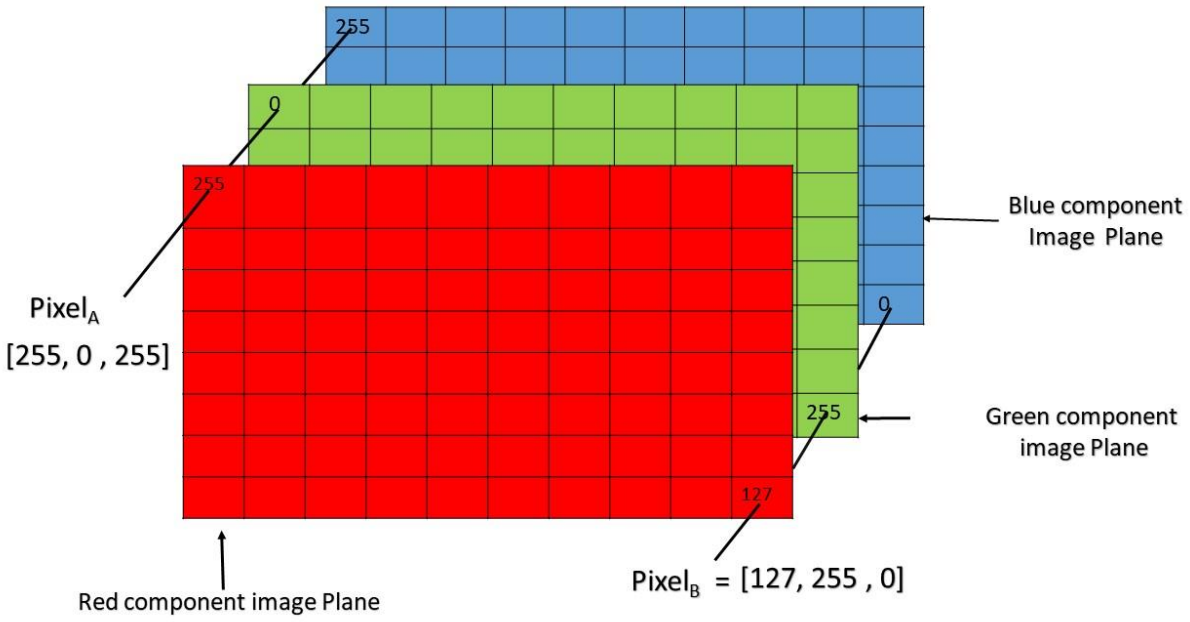


Figure 5.16: Color channels of an RGB image.

Pixel's values in the range of 0 to 255 can result in some troublesome issues. This range is not significant by itself, but a single picture potentially contains a lot of pixels and when we perform calculations with these values, the number will soon grow exponentially. As a result, either the NaN (Not a Number) error arises or more problematic errors such as slow convergence or gradient explosion occur. In essence, it is more beneficial if the pixel values are scaled or prepared rather than using the values in raw format.

There are three main techniques: Pixel Normalization, Pixel Centering and Pixel Standardizing [49]. Only Pixel Normalization is used since it is the most straightforward, easy to compute and does not require extra calculations. This technique transforms the range of pixel

values from $[0, 255]$ to $[0.0, 1.0]$ by dividing all pixel values by the largest possible value of a pixel, which is 255. It is carried out across all channels. Nonetheless, it may be less accurate in comparison with other methods.

5.2.3 Keypoints

After cropping and padding the image, the original keypoint annotation of that image is no longer correct, hence, they need to be recalculated.

For cases where bounding box (red) fits inside the image (black):

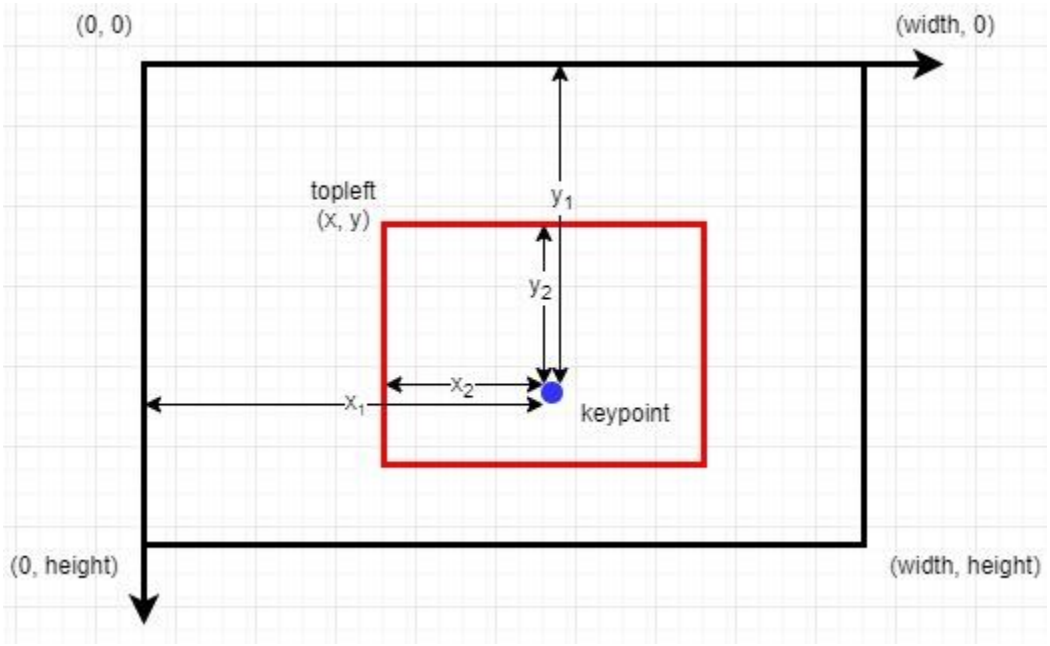


Figure 5.17: Bounding box is inside the image.

Let us denote (x, y) are the coordinates of the top left corner of the bounding box, (x_1, y_1) are the original coordinates of the keypoint (blue) and (x_2, y_2) are the new coordinates of the keypoint. (x_2, y_2) can be found as follow:

$$\begin{cases} x_2 = x_1 - x \\ y_2 = y_1 - y \end{cases} \quad (5.1)$$

The above formula stays true for cases where the bounding box is out of bound on the right or/and bottom of the image.

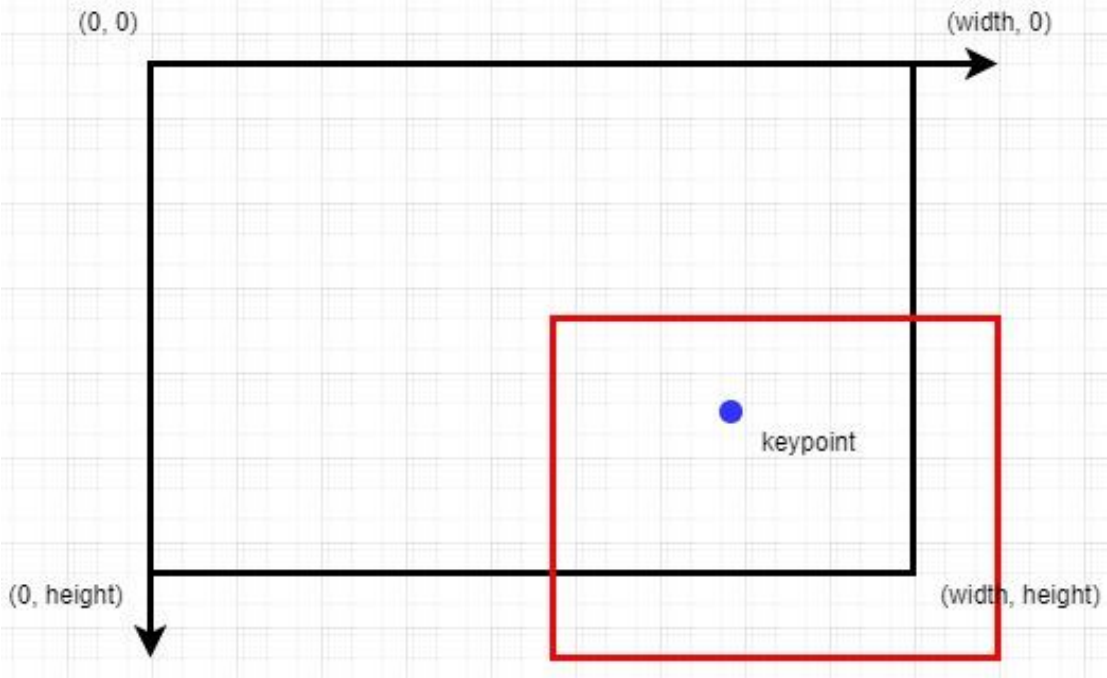


Figure 5.18: Bounding box (red) is outside of the image (black) 1.

However, if the bounding boxes are outside of the left or/and top of the image, a different calculation is needed.

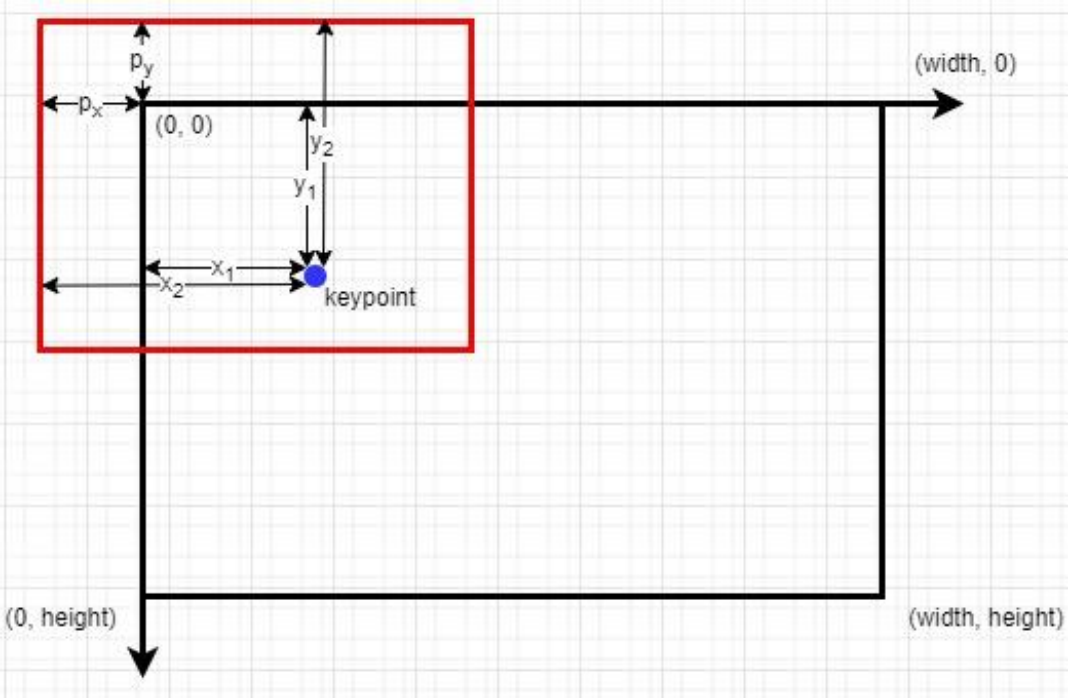


Figure 5.19: Bounding box (red) is outside of the image (black) 2.

With the same denotation as above and (p_x, p_y) are the amount of padding on x- and y-axis respectively. x_2 and y_2 can be calculated as follow:

$$\begin{cases} x_2 = x_1 + p_x \\ y_2 = y_2 + p_y \end{cases}. \tag{5.2}$$

Finally, keypoint coordinates need to be scaled in order to be in $(256, 256)$ space (Section 5.2.2.3).

$$\begin{cases} x_{final} = x_2 \cdot \dfrac{256}{bbox_w} \\ y_{final} = y_2 \cdot \dfrac{256}{bbox_h} \end{cases}, \tag{5.3}$$

where x_2 and y_2 are the result from above, $bbox_w$ is the bounding box's width and $bbox_h$ is the bounding box's height. The ideal is to normalize both x_2 and y_2 to $[0.0, 1.0]$ range and then multiply them with 256.

5.3 Storing preprocessed data

After being preprocessed both images and annotations are stored in TFRecord format [50] for later usages. TFRecord is a storage format of Tensorflow, which is a ML framework and is used in this thesis for the implementation of the CNN model. It uses Protocol Buffers [51], a cross-platform, cross-language library for efficient serialization of structured data.

Many benefits of using this format includes but are not limited to:

- Efficiency: TFRecord files take less space than storing data in JPG format for images and JSON format for annotations.
- Fast I/O (Input/Output): TFRecord is native to Tensorflow and can be read with parallel I/O operations. When combined with the use of TPU (Tensor Processing Unit) or GPU (Graphical Processing Unit), it is very useful.
- Self-contained files: Everything that is needed is stored in the same TFRecord file, both the image or any kind of data and its metadata.

- Compatibility with other Tensorflow's APIs (Application Programming Interfaces): Since the format is native to Tensorflow, it can be combined with other APIs to serve different purposes.

5.4 Summary

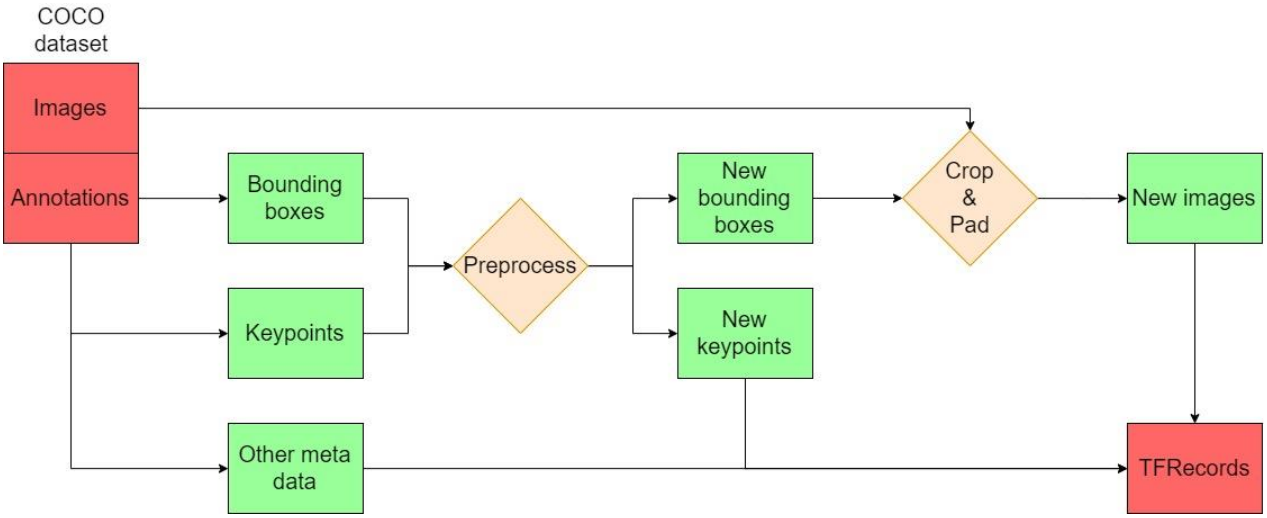


Figure 5.20: Preprocessing pipeline.

From the COCO dataset, bounding boxes and keypoints data are extracted and preprocessed. Depending on the number of annotations an image carries, new images are cropped using the new bounding boxes from the original image. New images, new keypoints and some other metadata are packed and stored in TFRecord files. It is recommended that each TFRecord file should be approximately 100 Mb [52]. Thus, a TFRecord file contains 2042 instances in this case.

	Number of instances	**TFRecord files**
Train	134,214	66
Valid	5,647	3
Total	139,861	69

Table 5.3: Number of TFRecord files for training and validation.

Chapter 6: Implementation and training details of the CNN model

6.1 Model implementation

The Stacked Hourglass Network is implemented entirely using the Tensorflow framework with Keras backend [53]. Primarily used layers are Conv2D, MaxPool2D, BatchNormalization, UpSampling2D and Add.

```
import tensorflow as tf
tf.keras.layers.Conv2D(filters, kernel_size, strides, padding, activation)
tf.keras.layers.MaxPool2D(pool_size, strides)
tf.keras.layers.BatchNormalization()
tf.keras.layers.Add()
tf.keras.layers.UpSampling2D(size, interpolation='nearest')
```

Figure 6.1: Layers of Tensorflow used for the CNN model.

Layers are constructed using Tensorflow Functional API as it enables branching and multiple outputs.

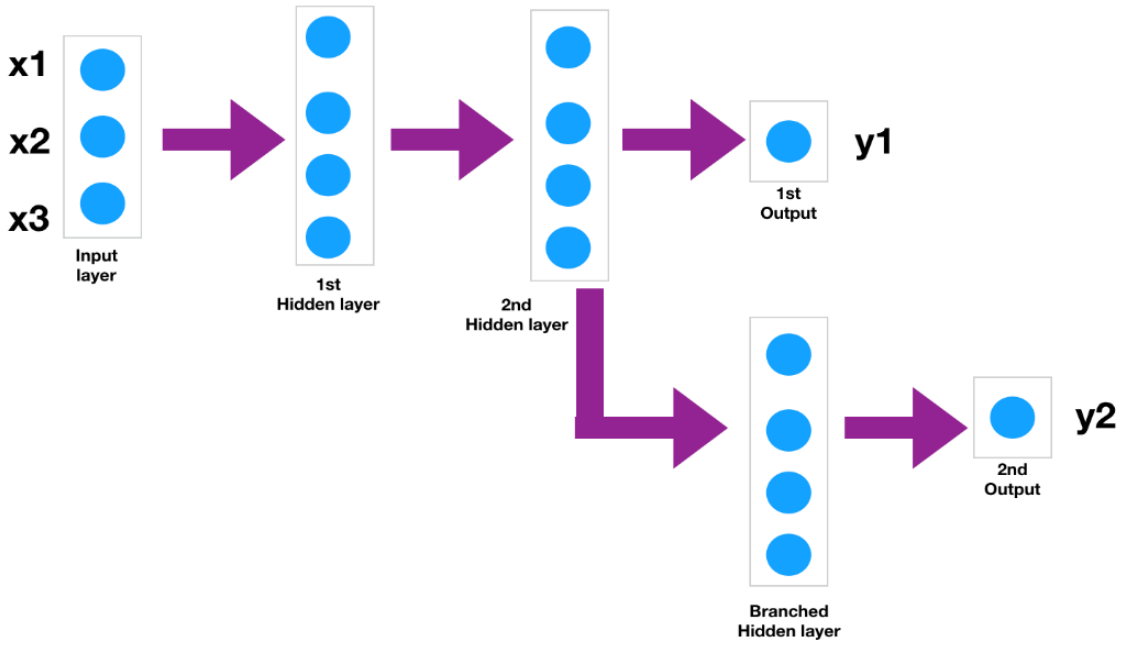


Figure 6.2: Example of Functional API [54].

Lastly, when all layers are properly set up with reference to Chapter 4, a Tensorflow model need to be defined as follows:

```
tf.keras.Model(inputs, outputs)
```

Figure 6.3: Tensorflow model

6.2 Training details

6.2.1 Hardware

The entire training process is conducted in the Google Colaboratory environment which provides free cloud CPUs, GPUs and TPUs.

“Colaboratory, or “Colab” for short, is a product from Google Research. Colab allows anybody to write and execute arbitrary python code through the browser, and is especially well suited to machine learning, data analysis and education. More technically, Colab is a hosted

Jupyter notebook service that requires no setup to use, while providing access free of charge to computing resources including GPUs.” - Colaboratory: Frequently Asked Questions [55]

The training is done using GPU; however, it is not possible to choose a specific GPU type. The GPU is arbitrarily assigned to the user at run time. The available types are Nvidia K80s, T4s, P4s and P100s. Around 12 GB of RAM and 150 GB of storage is also provided. Additionally, Colab allows the user to “mount” Google Drive for more storage, but it is slower to write and read to. In order to use Colab, the user must keep a stable internet connection via browser and when disconnected all data in the current environment will be wiped. Therefore, all progress of the training is saved on Google Drive at the end.

6.2.2 Input pipeline

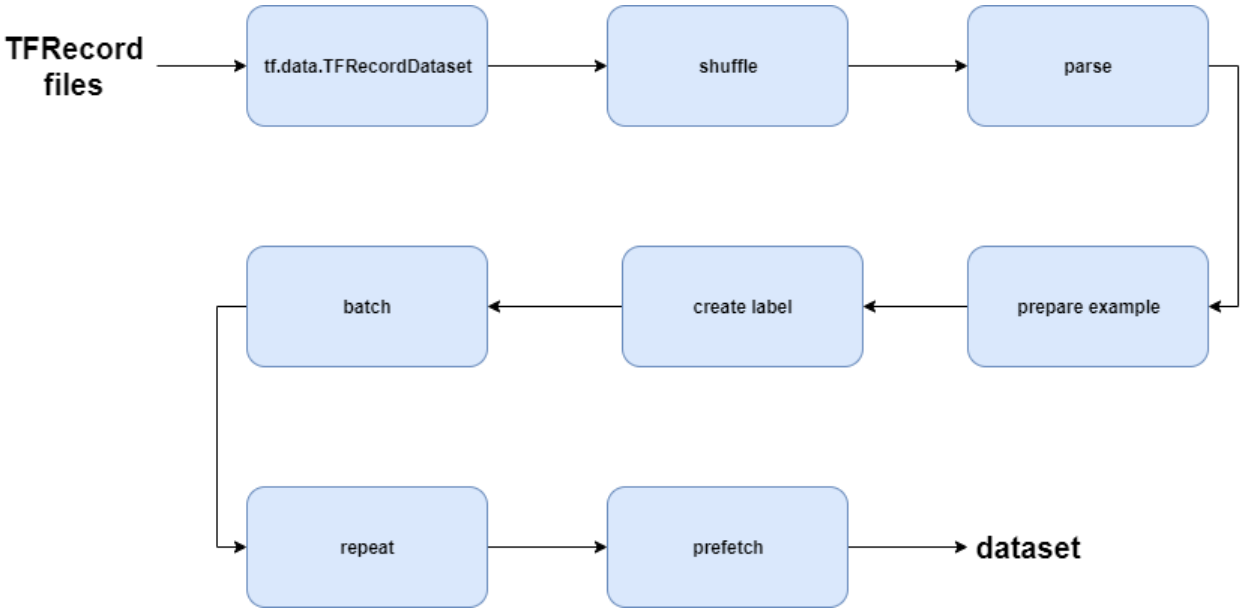


Figure 6.4: Input pipeline.

6.2.2.1 Reading TFRecord files

At the end of Chapter 5, the entire filtered dataset is converted and stored in TFRecord format. Therefore, before it can be used, the data needs to be interpreted using tf.data API [56]. The API allows the user to build a complex input pipeline from simple, reusable pieces such as TFRrecord files.

```
tf.data.TFRecordDataset(filenames)
```

Figure 6.5: Tensorflow dataset.

6.2.2.2 Shuffling

After the dataset is created, its examples are shuffled to help the training converge faster, reduce bias during training and prevent the model from learning the order of training. Only the dataset used for training is shuffled, there is no need to shuffle the validation dataset.

```
dataset.shuffle(buffer_size)
```

Figure 6.6: Shuffle dataset.

Buffer size represents the number of examples which are randomized and then returned to the dataset. In an ideal scenario, it is best to set the buffer size to the length of the dataset. Nonetheless, the bigger the buffer size is, the more RAM is needed. Therefore, a buffer with the size of 1000 is chosen.

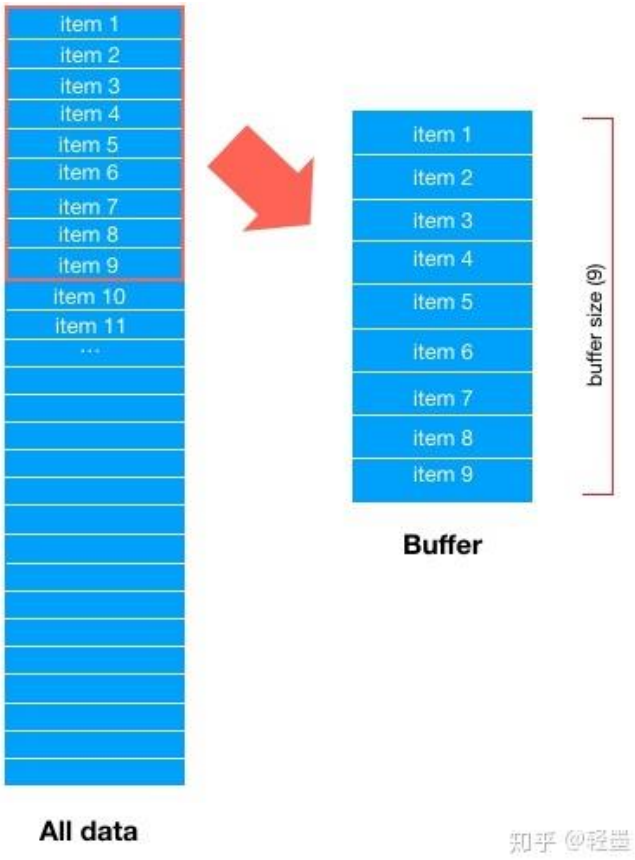


Figure 6.7: Buffer size example [57].

6.2.2.3 Parsing

Each example of the dataset contains both image and metadata. As a result, the image data and necessary metadata needs to be extracted for latter operations. The metadata fields used during the pipeline are width and height of the image, x-, y-coordinates and visible flags of the keypoints.

6.2.2.4 Preparing

Typically, in this step images are resized into the desired size, but all images already are 256×256. Hence, no resizing is needed. For keypoints, their coordinates are recalculated as follows:

$$\begin{cases} x_{new} = x_{old} * \dfrac{64}{width} \\ y_{new} = y_{old} * \dfrac{64}{height} \end{cases}, \tag{6.1}$$

where $width$ and $height$ are obtained in the previous step and 64×64 are the dimensions of the label.

6.2.2.5 Creating label

First, for the training dataset, each example is augmented to increase the diversity of training data by applying random transformations. Dataset used for validation does not need to be augmented. Augmentation in combination with shuffling guarantees better randomness for each example. The transformations include:

- Radom flipping left right with 50% chance
- Random scaling in range of $[0.75, 1.25]$
- Random rotation with an angle from -30° to 30°
- Random brightness
- Random contrast
- Random hue
- Random saturation

The first three transformations affect both the image and its keypoints, therefore they are implemented using imgaug library [58]. The others affect only the image and are implemented using Tensorflow image module [59].

After being transformed, keypoints are now ready to be used for creating label. There are two ways to use keypoint coordinates as label which are direct and indirect methods. The direct method regresses keypoints as they are, numerical values (x and y). Regressing coordinates directly is highly non-linear and more difficult to learn for the network. For instance, an image of size 256×256 would have 65536 pixels, so it would be exceedingly challenging to locate a point in 65536 pixels $(1: 65536)$. In an indirect method, a prediction by regressing a heatmap over the image for each keypoint and then the keypoint can be extracted by various ways but the

easiest way is to find the pixel with maximum value. A heatmap is a 2D Gaussian distribution with small deviation centered at the keypoint coordinate, that means there are more pixels which denpend on the kernel size. For example, instead of regressing 1 point out of 65536 pixels, the rate can be $200: 65536$, which is significantly better. Therefore, the second method is selected.

Heatmaps are generated using 2D Gaussian function:

$$G(x, y) = e^{-\frac{(x-x_0)^2+(y-y_0)^2}{2\sigma^2}}. \quad (6.2)$$

For each keypoint that is visible ($v > 0$), a patch that represents Gaussian distribution is added to an empty 64×64 matrix, where the center of the patch is located at the keypoint's coordinates. The size of the map can be found use the following formula:

$$size = 6 * \sigma + 1. \quad (6.3)$$

With $\sigma = 1$, the patch has the size of $7x7$. The center of the path $(3,3)$ will have the value of $G = e^0 = 1$. The pixel to the right of the center will have the value $G = e^{-\frac{1+0}{2}} = 0.61$ and so on.

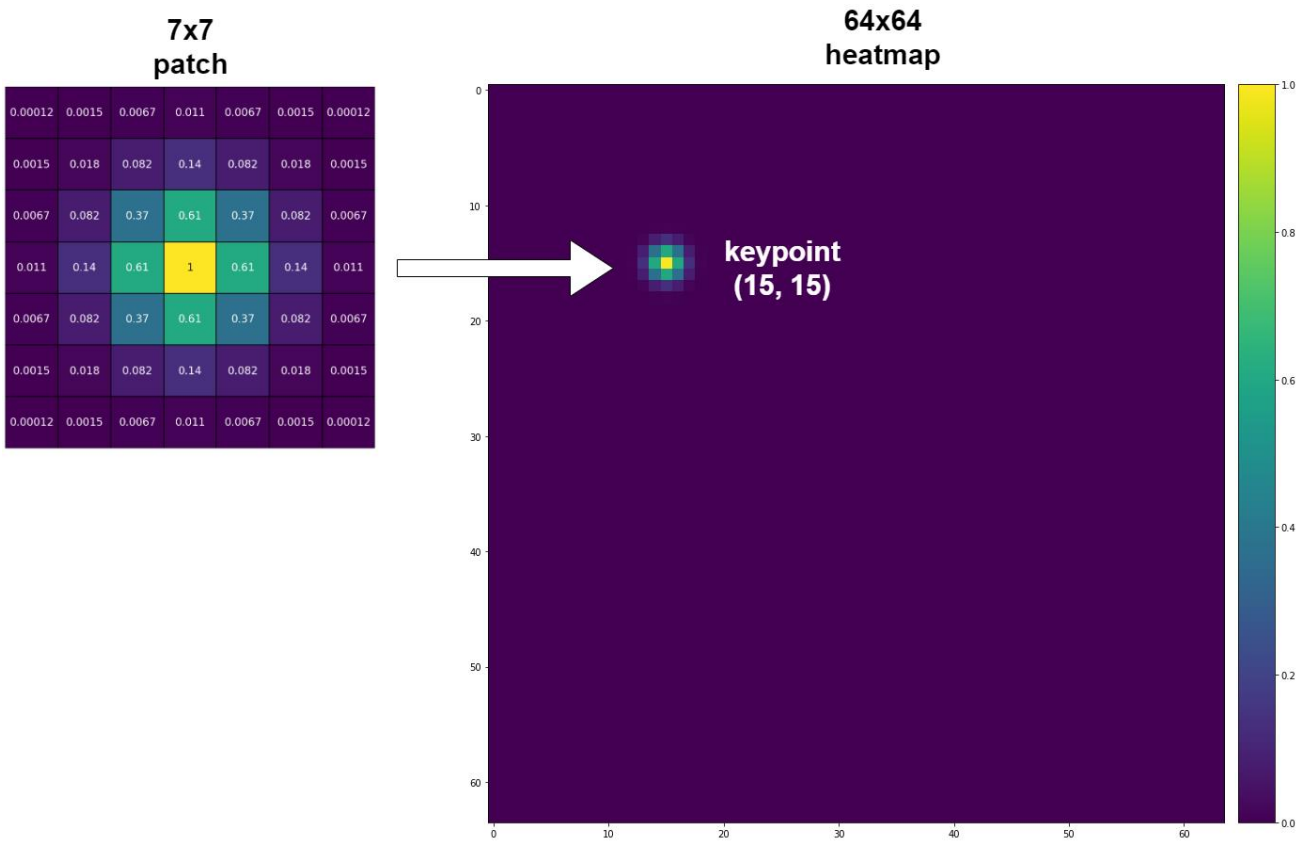


Figure 6.8: Heatmap of a keypoint with x = 15 and y = 15.

Since COCO dataset has 17 keypoints for one person, the label has the shape of $(64, 64, 17)$, the same shape of the SHG's outputs (see Section 4.3). In short, every example has a 256×256 image as input and seventeen 64×64 heatmaps as a label.

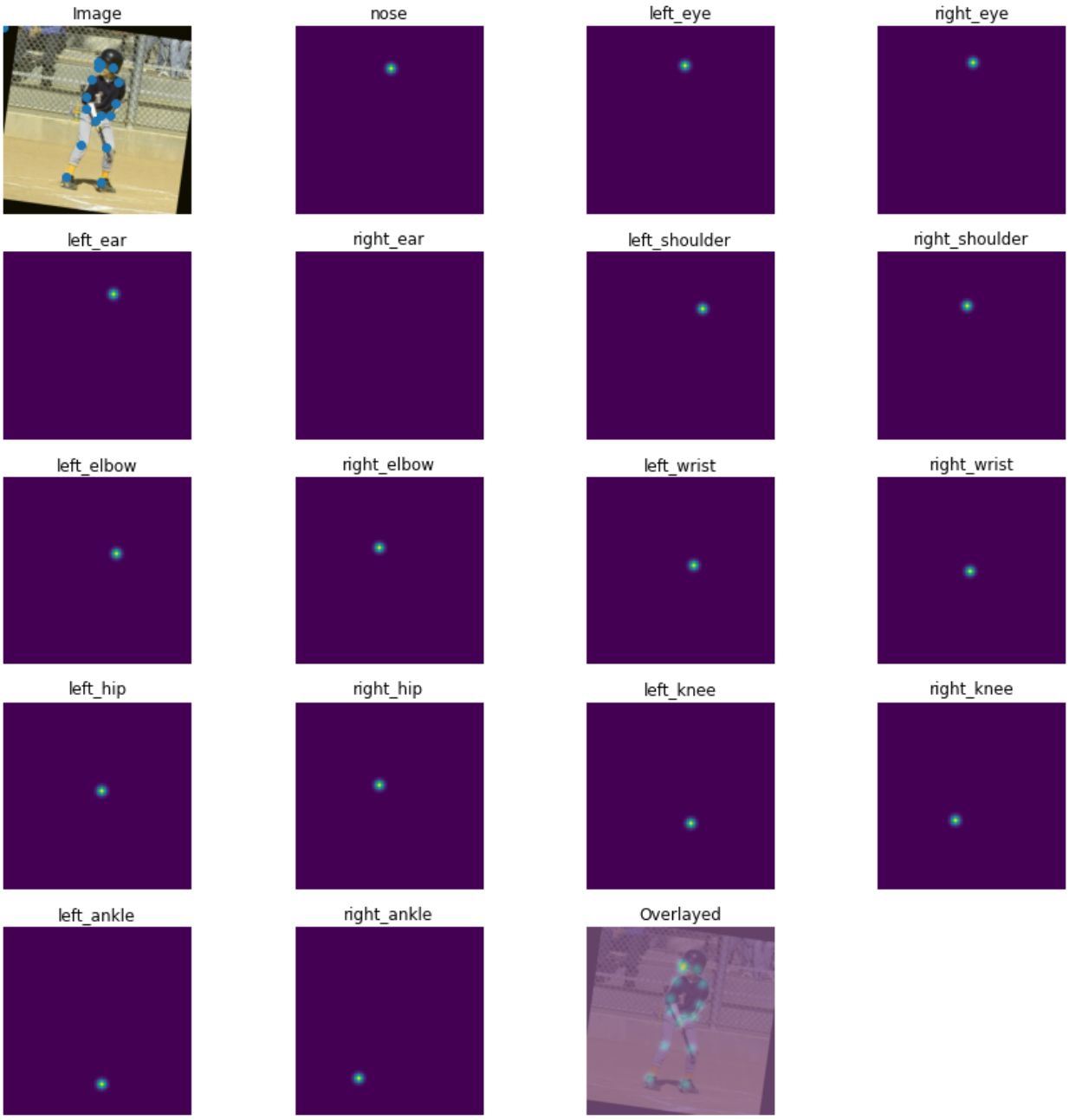


Figure 6.9: Augmented image and heatmaps of train dataset.

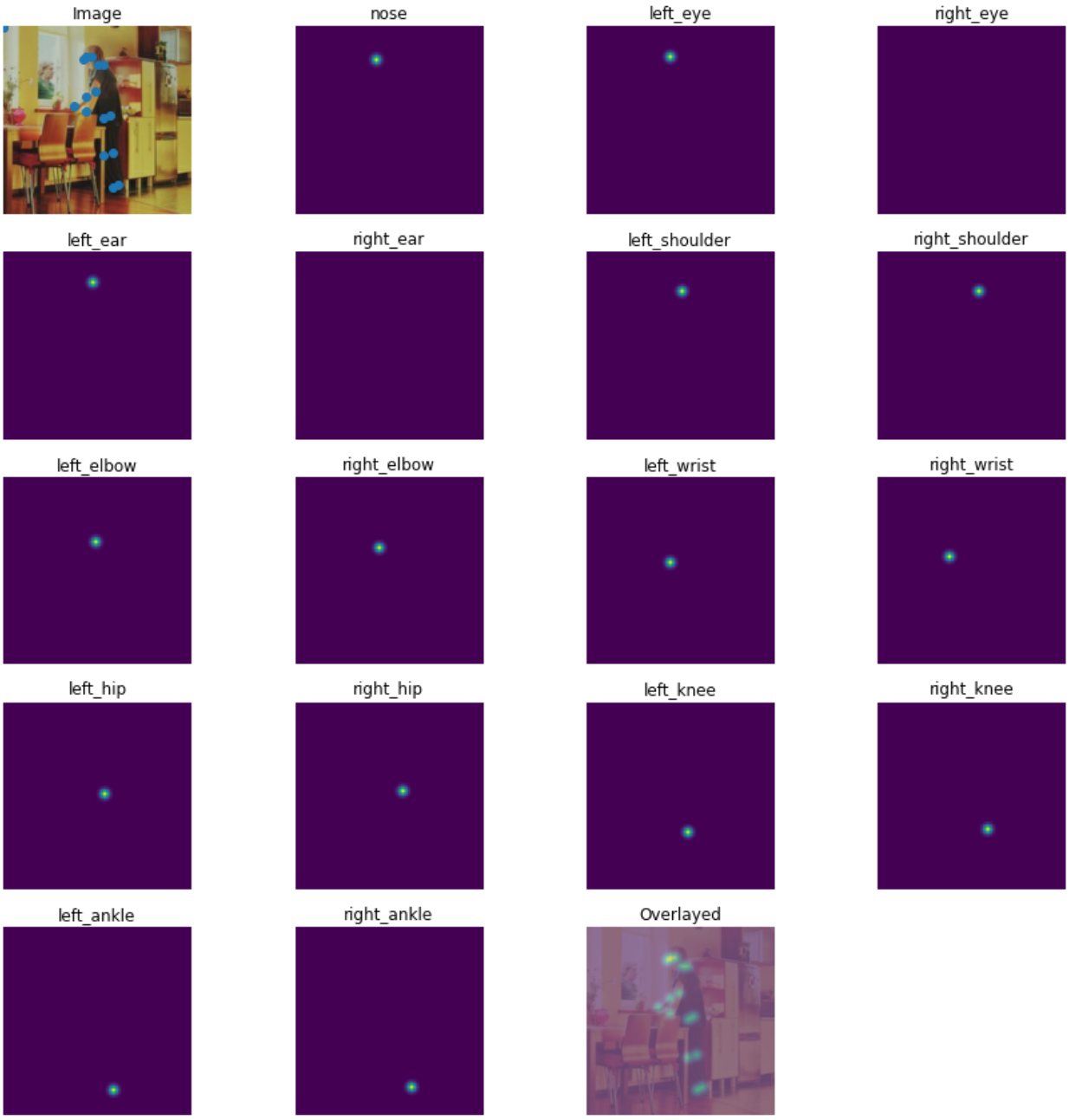


Figure 6.10: Image and heatmaps of validation dataset.

6.2.2.6 Batching

As discussed in Section 2.5.4, batching the data helps training progress faster, however, due to memory constraints only batch size of 16 is used.

6.2.2.7 Repeating

As soon as all the entries of the dataset are read, if the repeat method is not called, an error will be thrown.

```
dataset.repeat()
```

Figure 6.11: Repeating dataset.

Repeat method signifies that, when the end of the dataset is reached, it will be reinitialized and start at the beginning again. This makes training for multiple epochs possible.

6.2.2.8 Prefetching

Prefetching overlaps the data processing and model execution of a training step. Instead of staying idle and waiting for the next batch to be processed and fed into the GPU, data can be loaded and processed by the CPU asynchronously while the model is being trained on the GPU.

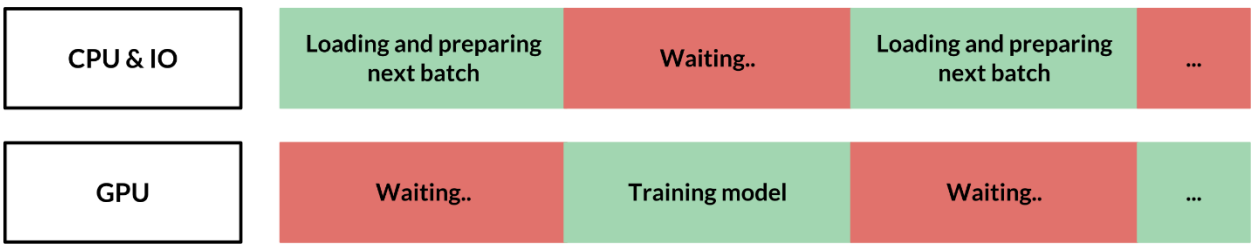
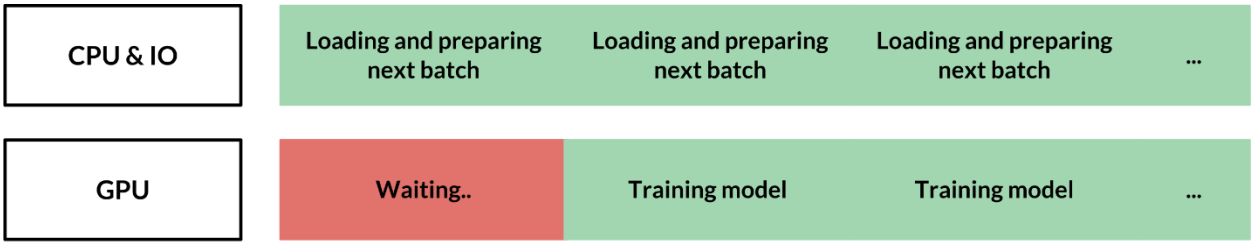


Figure 6.12: Pipeline without prefetching [60]



(Simplified, in reality there is always *some* idle time on both devices)

Figure 6.13: Pipeline with prefetching [60].

Figure 6.13 shows the reduction of training time using prefetching, so further improvement can be achieved if multiple CPUs parallel the loading and preprocessing data.

6.2.3 Choosing training hyperparameters

6.2.3.1 Optimizer

A majority of HPE papers train using RMSprop optimizer, but more recent projects have started adopting Adam optimizer since it provides better results than its counterpart. “Adaptive Moment Estimation is an algorithm for optimization technique for gradient descent. The method is really efficient when working with large problems involving a lot of data or parameters. It requires less memory and is efficient. Intuitively, it is a combination of the ‘gradient descent with momentum’ algorithm and the ‘RMSP’ algorithm.” [61] For this paper, Adam optimizer with initial learning rate of 10^{-3} is used.

6.2.3.2 Losses

Most commonly used loss for HPE task is MSE (see Section 2.4.2). Besides using MSE, two more loss functions are also adopted in order to experiment which one gives the best outcome for training.

The first one is weighted MSE. The main intuition is that except for the pixels that surround keypoints, all background pixels have zero value. There are around 82 times more background pixels than foreground pixels. This can theoretically hinder the convergence of the model since the model can cheat by predicting all zeros to reach local minima. The solution is to create a weight matrix for each pixel of the 64×64 heatmap. For the $7x7$ patch (see Section 6.2.2.5), a factor of 82 is multiplied with all pixel values of the patch.

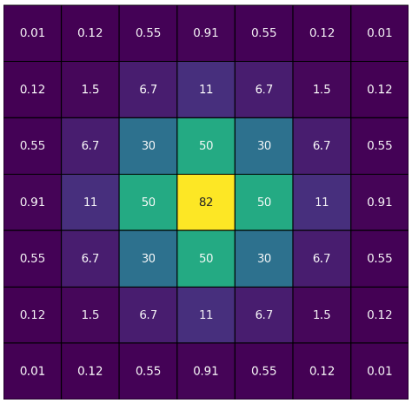


Figure 6.14: Weights as a 7x7 patch.

For all other pixels, their values simply are one. The weighted MSE can then be calculated as follows:

$$Weighed\ MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 * weights, \tag{6.4}$$

where $weights$ is the weight matrix. Since the weights of all foreground pixels is one, only weights surrounding the keypoint affect the final calculation. The weight matrix adds more significance or importance to the area around the keypoint, as a result, it is hard for the model to ignore non-zero values.

The second loss is Intersection Over Union (IoU). IoU loss is rarely used in HPE yet shows great possibilities. This loss is rather widely adopted in object detection or segmentation problems. IoU is defined as the intersection of ground truth and prediction regions over their union. When the intersection perfectly concurs, IoU is 1 and if there is no intersection IoU is 0. This is suitable for heatmap regression since an area is regressed, not just a single point. The following formulas are slightly modified for heatmaps and were used to predict keypoints of hands [62]:

$$I = \sum_i (y_i * \hat{y}_i) \tag{6.5}$$

$$U = \sum_i (y_i * y_i) + \sum_i (\hat{y}_i * \hat{y}_i) - \sum_i (y_i * \hat{y}_i) \tag{6.6}$$

$$IoU = 1 - \frac{I}{U}, \tag{6.7}$$

where y_i and $\hat{y}_i$ are predicted and target pixel values of a heatmap.

6.2.4 Training pipeline

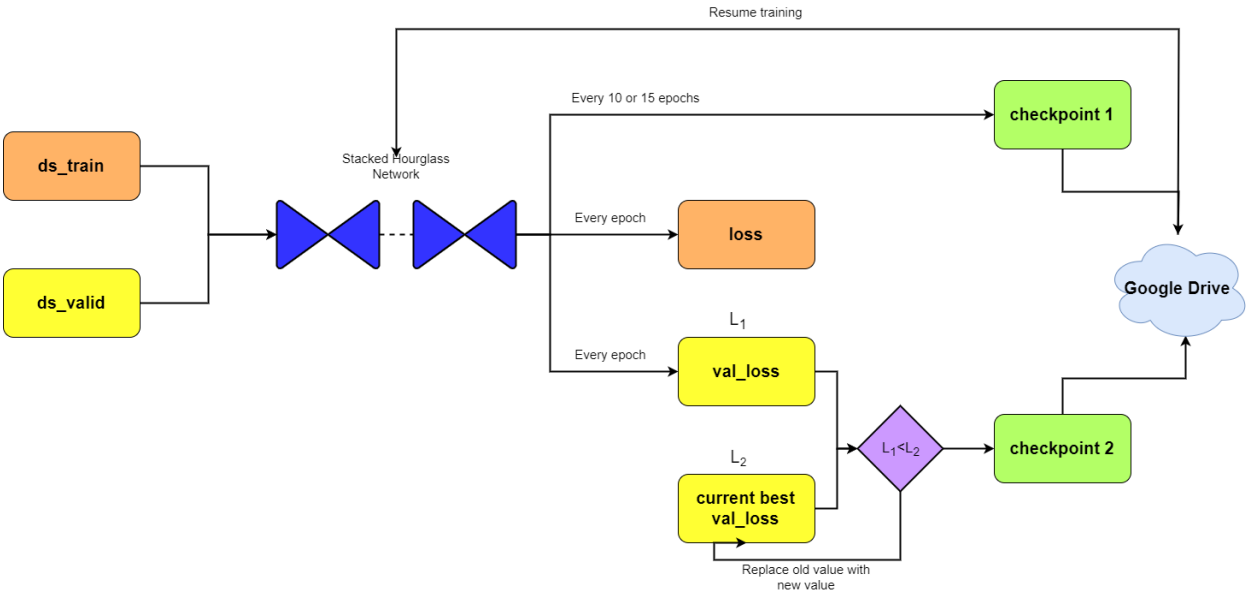


Figure 6.15: Training pipeline.

Training and validation datasets, which are obtained from the input pipe, are fed into the CNN model (SHN). From training dataset at the end of one epoch $loss = sum(loss_1, \dots, loss_n)$ is calculated where n is the number stacks/stages of the SHN. Similarly, for validation dataset $val\_loss = sum(val\_loss_1, \dots, val\_loss_n)$. Afterwards, val_loss which is denoted as L_1 is compared with L_2 (current best/lowest val_loss). If $L_1 < L_2$, a checkpoint is saved to Google Drive. A checkpoint only contains weights of the model since it takes less storage space than saving the whole model.

Training the model with COCO dataset potentially takes days. Nonetheless, Colab only keeps the connection from user to the environment for up to 24 hours. It is, however, not recommended to use GPU for training for over 12 hours, otherwise the runtime will be disconnected, and the progress will be lost. Therefore, training is split into sessions with each taking 10 to 12 hours. At the end of each session, another checkpoint will be saved to the drive. For a 2 stacks hourglass network, one session has 15 epochs and for a 4-stack one, it is 10 epochs.

To resume training from previous session, $checkpoint$ 2 is loaded from Google Drive to the model. $checkpoint$ 1 is only used for evaluation.

Chapter 7: Results, evaluation conclusion and suggestion

7.1 Heatmaps postprocessing

Since CNN model outputs heatmaps instead of x and y coordinates, the output needs to be converted before being used for evaluation.

The first method is simple. For each 64×64 heatmap, which represents one keypoint, the brightest pixel is the location of the keypoint. The value of the pixel is the confidence score, which indicates how confident the network thinks this pixel contains the keypoint.

However, the first method may not be that accurate. The input of the SHN is 256×256 image and its output is 64×64 heatmaps to represent keypoint locations. It is four times down-scaling. However, the image is initially resized to 256×256 from a bigger resolution such as 720×480. As a result, a 64×64 heatmap would be viewed as too coarse. Therefore, instead of just using the pixel with maximum value, the second method also takes into consideration the neighbor pixels. It infers that the actual keypoint location might be slightly towards the direction of the max-valued pixel and its neighbor. This method mimics the behavior of gradient descent, which also points to the optimal solution. If we denote (x_0, y_0) is the point with maximum value of the heatmap, (x_1, y_1) is the point with second highest pixel value of a 3×3 patch, which (x_0, y_0) is the center, then the coordinates of desired keypoint can be calculated as follows:

$$\begin{cases} x = x_0 + \frac{x_1}{4} \\ y = y_0 + \frac{y_1}{4} \end{cases}. \tag{7.1}$$

Finally, for both methods, a confidence threshold of 0.1 is set. Any keypoint with confidence score less than threshold is considered occluded $(0, 0)$.

7.2 Evaluation metrics

7.2.1 Object Keypoint Similarity (OKS)

OKS is an evaluation metric used by the COCO dataset. Recently, this metric has become more popular than other metrics in keypoints detection task. Its usages and details are discussed on the COCO official website [63].

For each instance, the OKS value is computed according to the following formula:

$$OKS = \frac{\sum_i \left(e^{-\frac{d_i^2}{2s^2k_i^2}} \right) \delta(v_i > 0)}{\sum_i \delta(v_i > 0)}, \tag{7.2}$$

where d_i is the distance between predicted keypoint and ground truth keypoint. Visible flag v_i indicates if a keypoint is not labeled $v = 0$, labeled but not visible $v = 1$ and labeled and visible $v = 2$. As the above equation shows, unlabeled keypoints do not affect the OKS value. Scale s is defined as the square root of the object segment area. The value of k_i is a per keypoint constant that controls falloff. These constants have been calculated by the COCO team and they come from the labeling error.

KEYPOINT	k_i
HIPS	0.107
ANKLES	0.089
KNEES	0.087
SHOULDERS	0.079
ELBOWS	0.072
WRISTS	0.062
EARS	0.035
NOSE	0.026
EYES	0.025

Table 7.1: Keypoint constants for OKS.

Once the OKS value for one example is obtained, it needs to be categorized to determine if it is a True Positive (TP) or a False Positive (FP). There are several thresholds that COCO dataset uses ranging from 0.5 (loose) to 0.95 (very strict) with a step size of 0.05. A value is considered a TP if it is over the threshold and if it is under, it is an FP. It is considered as a False Negative (FN), if a person within an image has not been detected. Precision and Recall can be calculated using TP, FP and FN.

$$Precision = \frac{TP}{TP + FP}. \tag{7.3}$$

$$Recall = \frac{TP}{TP + FN}. \tag{7.4}$$

Using 0.5 and 0.95 as example threshold, Precision-Recall (PR) curves can be drawn as follows:

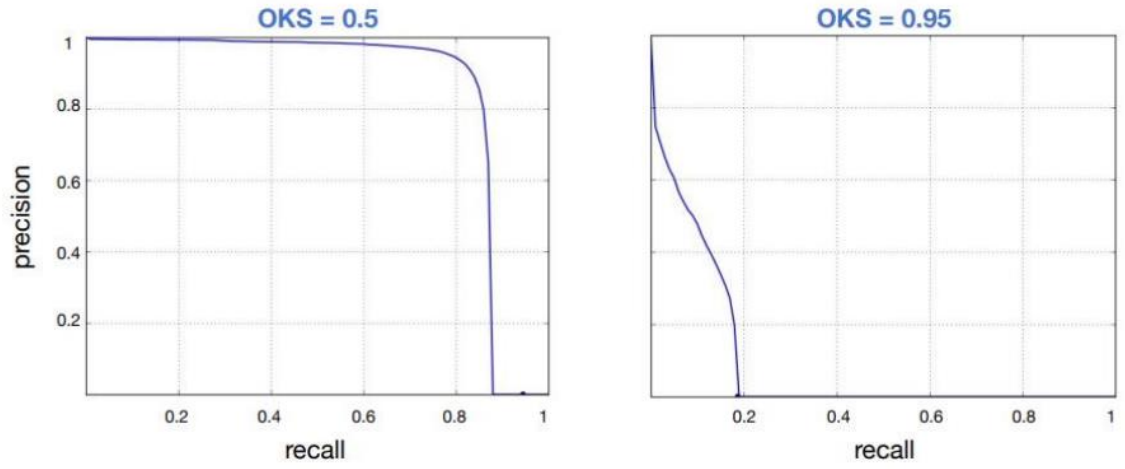


Figure 7.1: PR curves with different thresholds [64].

A good PR curve has greater AUC (area under curve). Figure 7.1 shows that, the lower the threshold is, the more tolerant it is. Hence, more TPs are found with 0.5 threshold. Average Precision (AP) for a specific threshold is the AUC. The average of all AP values from 0.5 to 0.95 is called mAP. This metric value is the primary metric to decide the winner for the COCO Keypoints Challenge.

```
Average Precision (AP):
  AP                  % AP at OKS=.50:.05:.95 (primary challenge metric)
  AP^OKS=.50          % AP at OKS=.50 (loose metric)
  AP^OKS=.75          % AP at OKS=.75 (strict metric)
AP Across Scales:
  AP^medium           % AP for medium objects: 32^2 < area < 96^2
  AP^large            % AP for large objects: area > 96^2
Average Recall (AR):
  AR                  % AR at OKS=.50:.05:.95
  AR^OKS=.50          % AR at OKS=.50
  AR^OKS=.75          % AR at OKS=.75
AR Across Scales:
  AR^medium           % AR for medium objects: 32^2 < area < 96^2
  AR^large            % AR for large objects: area > 96^2
```

Figure 7.2: COCO evaluation metrics.

The implementation of the metric is provided through COCO API [65].

7.2.2 Percentage of Correct Keypoints (PCK)

For PCK, a keypoint is considered detected if its distance to the ground truth keypoint is less or equal to a certain threshold. The threshold can either be:

- PCKh@0.5: threshold equals 50% of the head bone link (right ear to left ear)
- PCK@0.2: threshold equals 20% of the torso diameter (right hip to left shoulder)
- Sometimes a fixed distance is used such as 150mm

The problem with the first two thresholds is that the two joints they use are not always available. Furthermore, the distance between two keypoints is extremely sensitive with the body's rotation; the threshold would be too small if the person turns sideways. Using a fixed distance is not better either, since people can come in different scales. In an attempt to address those issues, the threshold is set to 5% of the human bounding box diagonal which is provided by the COCO dataset. Only labeled keypoints ($v > 0$) are considered.

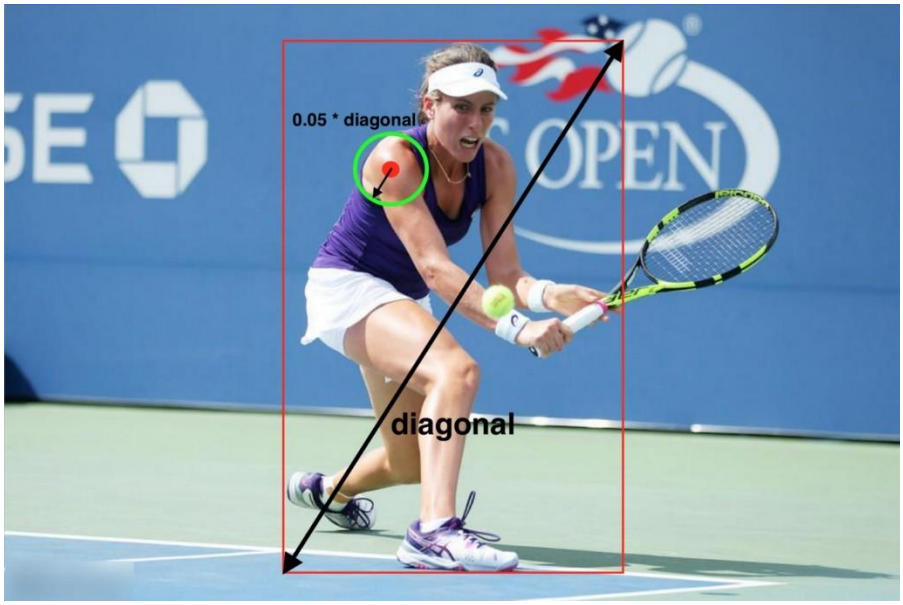


Figure 7.3: Visualization how PCK works [66].

$$PCK = \frac{\sum_i (0.05 * diagonal)\delta(v_i > 0)}{\sum_i \delta(v_i > 0)}. \quad (7.5)$$

7.3 Evaluation

7.3.1 Different losses

Three different CNN models were trained with three different losses including MSE, weighted MSE and IOU (Section 6.2.3.2). Each model has the same architecture of 2 stacks hourglasses and is trained for 3 sessions. The first and second sessions are both 15 epochs and the third one is 10 epochs. Therefore, the total number of epochs is 40 epochs for each model. The learning rate is 10^{-3} for all epochs.

OKS evaluation with heatmaps to keypoints method 1 and 2 is as follows:

	IOU		Weighed MSE		MSE	
	Method 1	Method 2	Method 1	Method 2	Method 1	Method 2
AP	0.45	0.52	0.41	0.47	0.45	0.52
AP $^{OKS=.50}$	0.79	0.81	0.76	0.78	0.78	0.79
AP $^{OKS=.75}$	0.48	0.58	0.41	0.50	0.48	0.58
AP ^{medium}	0.42	0.48	0.39	0.45	0.41	0.48
AP ^{large}	0.50	0.58	0.44	0.51	0.51	0.58
AR	0.51	0.57	0.47	0.54	0.49	0.56
AR $^{OKS=.50}$	0.81	0.82	0.79	0.81	0.80	0.81
AR $^{OKS=.75}$	0.56	0.64	0.51	0.58	0.55	0.61
AR ^{medium}	0.46	0.52	0.43	0.49	0.44	0.50
AR ^{large}	0.58	0.65	0.54	0.61	0.57	0.64

Table 7.2: OKS evaluation with different losses.

PCK evaluation with heatmaps to keypoints method 1 and 2 is as follows:

	IOU		Weighted MSE		MSE	
	Method 1	Method 2	Method 1	Method 2	Method 1	Method 2
Head	0.94	0.95	0.93	0.93	0.93	0.94
Shoulder	0.82	0.83	0.81	0.82	0.81	0.82
Elbow	0.79	0.80	0.76	0.77	0.78	0.78
Wrist	0.76	0.77	0.71	0.72	0.75	0.76
Hip	0.65	0.66	0.65	0.66	0.65	0.66
Knee	0.75	0.76	0.72	0.72	0.74	0.75
Ankle	0.75	0.77	0.72	0.73	0.76	0.76
Average	0.78	0.79	0.76	0.76	0.77	0.78

Table 7.3: PCK evaluation with different losses.

As Table 7.2 and Table 7.3 show, method 2 clearly gives better outcomes. Therefore, latter predictions will use the second post processing method as default. It also can be inferred that IOU loss performs slightly better than other losses. Weighted MSE is actually not better than MSE. One can hypothesize that the models were not trained long enough. For PCK, head

detection is the average detection rates of the left and right eyes, left and right eyes and nose. Others are simply of the average left keypoint and its right counterpart.

During training, loss values can go up and down and sometimes the loss at the end is not actually the best.

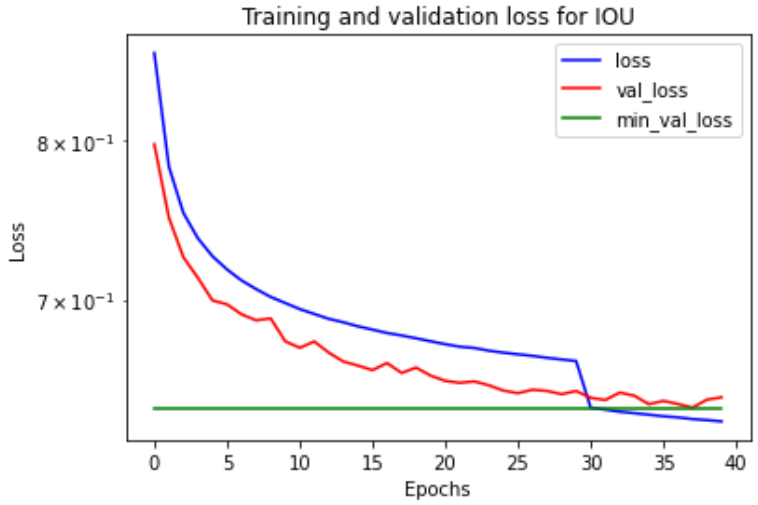


Figure 7.4: Training and validation losses for IOU.

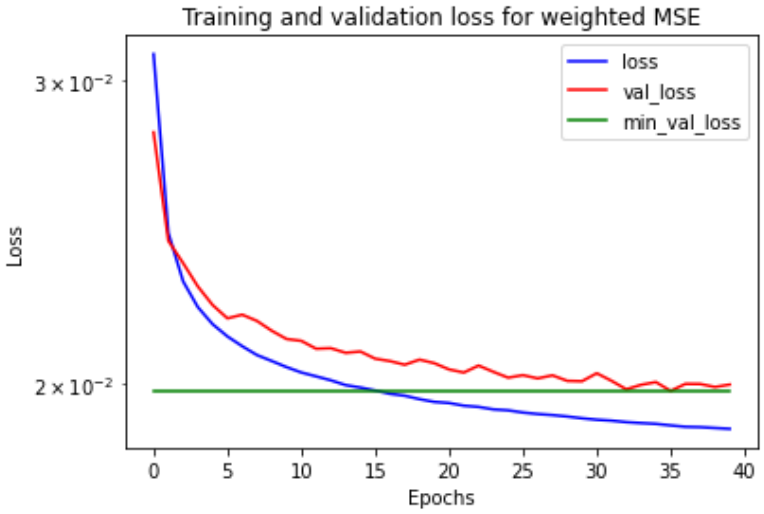


Figure 7.5: Training and validation losses for weighted MSE.

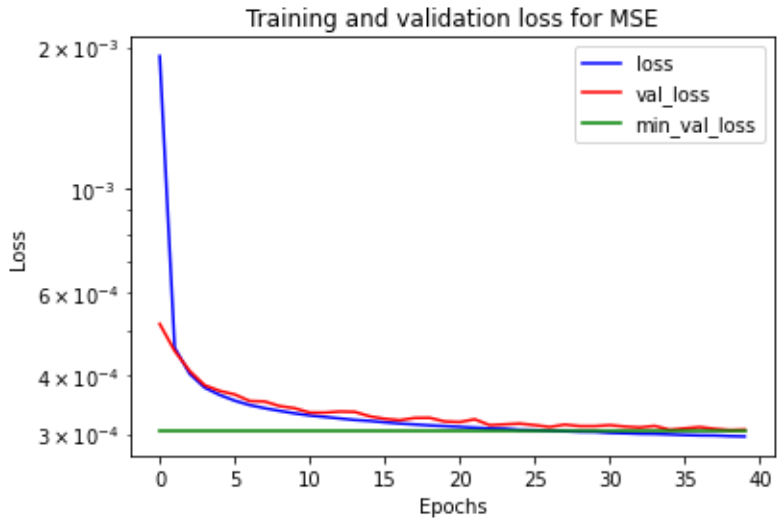


Figure 7.6: Training and validation losses for MSE.

The following figures show the evaluation of the models where validation loss is the smallest:

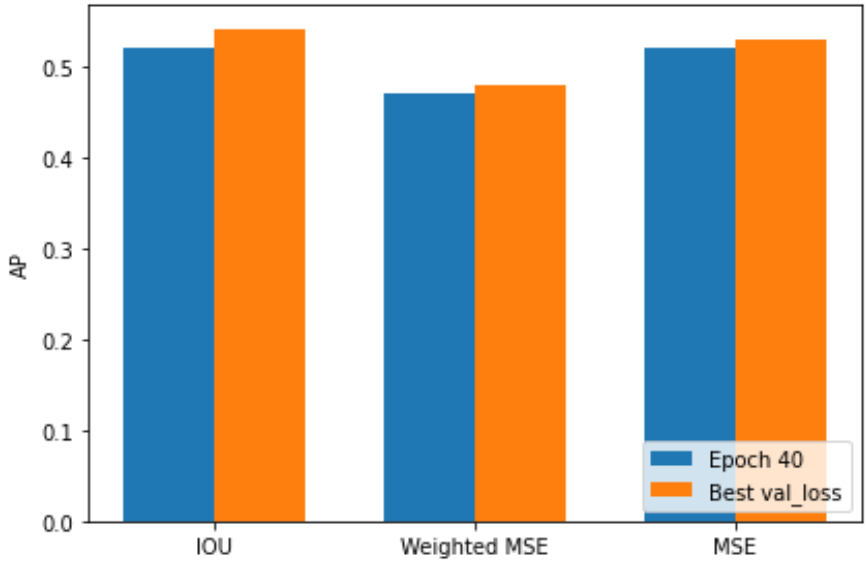


Figure 7.7: Compare OKS evaluations of different losses at epoch 40 and epoch where val_loss is smallest.

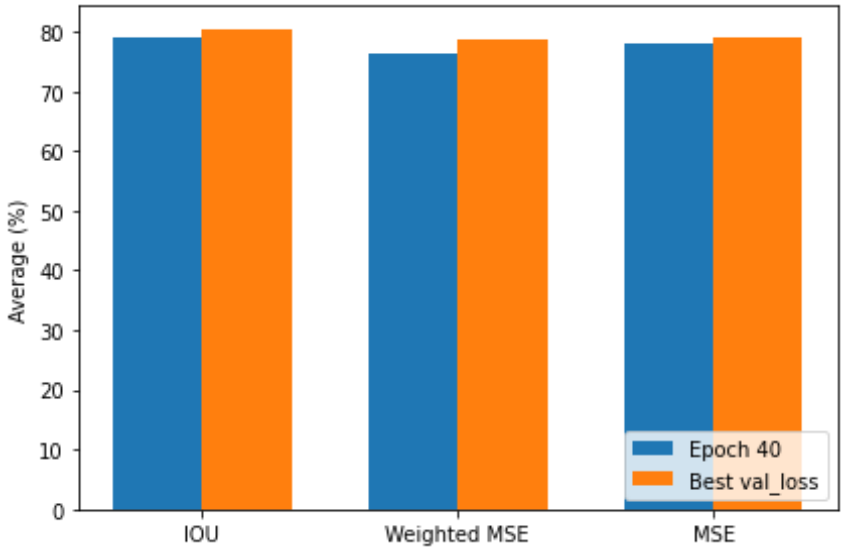


Figure 7.8: Compare PCK evaluations of different losses at epoch 40 and epoch where val_loss is smallest.

7.3.2 Four stacks vs two stacks Hourglass Network

A model with four stacks was trained for 80 epochs with IOU. From epoch 1 to 30 the learning rate is 10^{-3}, from epoch 31 to 60 the learning rate is 5×10^{-4} and from epoch 61 to 80 the learning rate is 2.5×10^{-4}. The following table shows the comparison between this model and a 2 stacks model with the same loss.

Model	Params	OKS	PCK
2 stacks (40 epochs)	7,034,530	0.52	0.79
4 stacks (40 epochs)	13,784,260	0.58	0.83
4 stacks (80 epochs)	13,784,260	0.59	0.85

Table 7.4: Compare a 2 stacks model vs a 4 stacks model.

For Table 7.4, Params is the number of parameters, OKS uses primary metric (mAP) and PCK uses average detection rates of all joints. The table also shows that at epoch 40, the 4-stack model has higher accuracy than the 2 stacks model. Further training can increase accuracy.

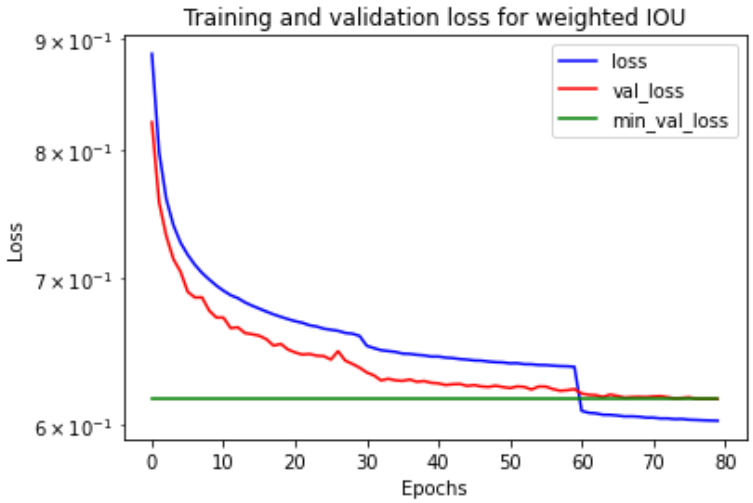


Figure 7.9: Training and validation losses of 4 stacks model.

Method	Backbone	Input size	Params	Pretrain	AP
8-stack Hourglass [39]	8-stack Hourglass	256 x 192	25.1M	N	66.9
CPN [67]	ResNet-50	256 x 192	27.0M	Y	68.6
CPN + OHKM [67]	ResNet-50	256 x 192	27.0M	Y	69.4
SimpleBaseline [68]	ResNet-50	256 x 192	34.0M	Y	70.4
HRNetV1 [69]	HRNetV1-W32	256 x 192	28.5M	N	73.4
Ours	4-stack Hourglass	256 x 256	13.8M	N	59.1

Table 7.5: Comparisons from different methods on COCO val2017 dataset.

Even though our model has the lowest score, it has less than half of the number of parameters than other architectures. Furthermore, most of the above models were trained for much longer periods of time and with vigorous testing. Therefore, this result is acceptable.

7.4 Results

Since the 4-stack model has the highest accuracy, only its outputs are used.

Figure 7.10: Ground truth vs predicted heatmaps.

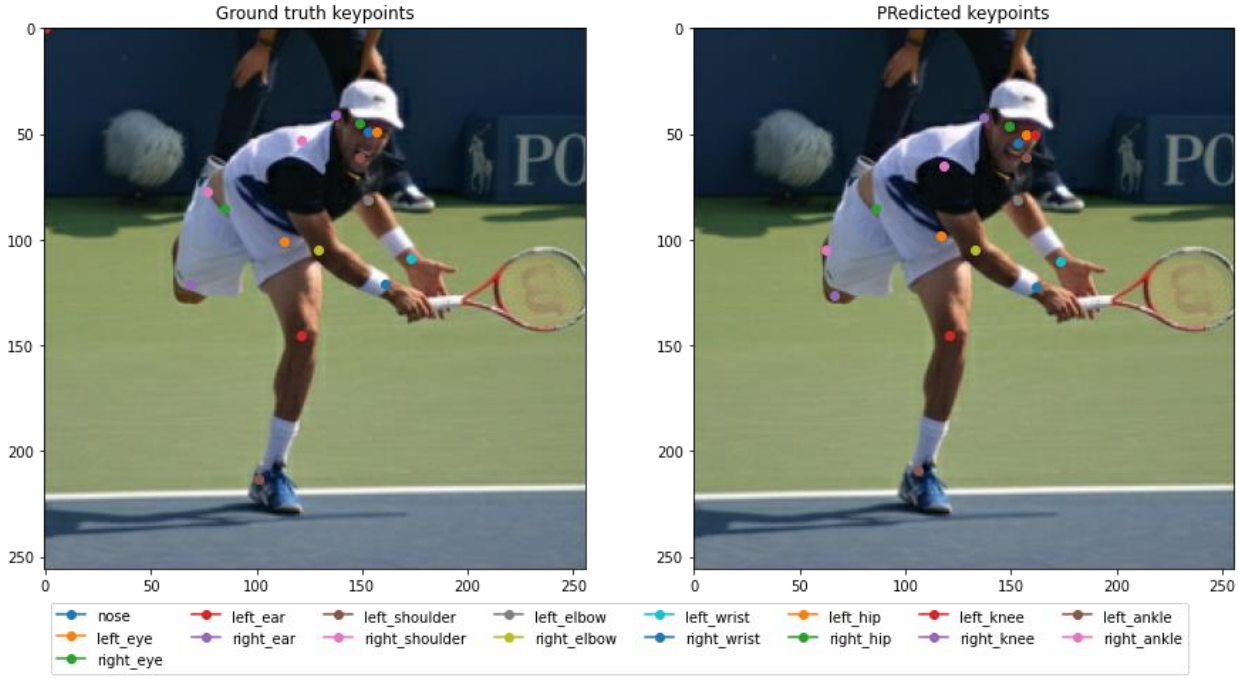


Figure 7.11: Ground truth vs predicted keypoints.

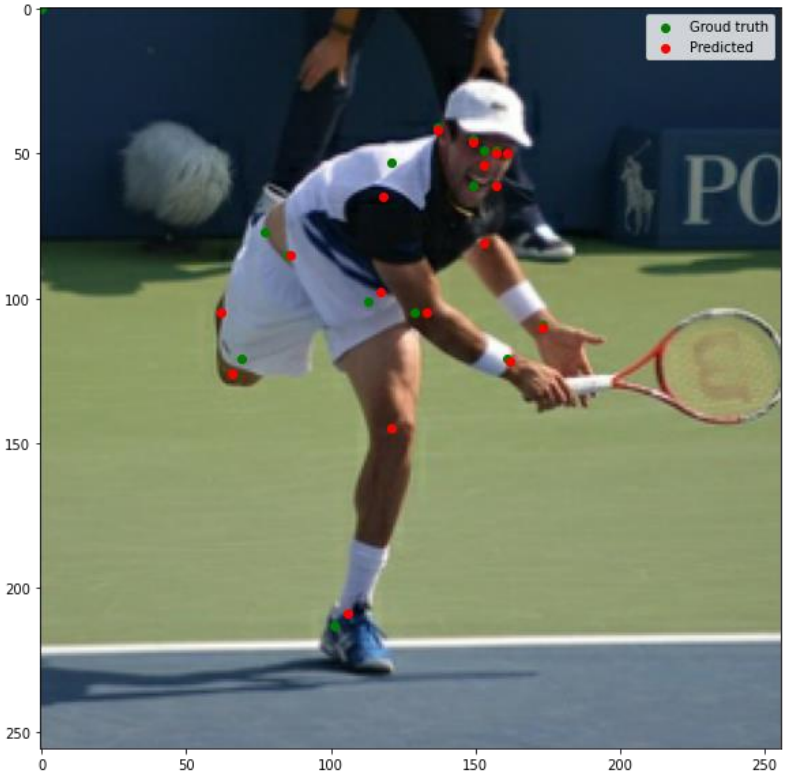


Figure 7.12: Overlapped predicted and ground truth keypoints.

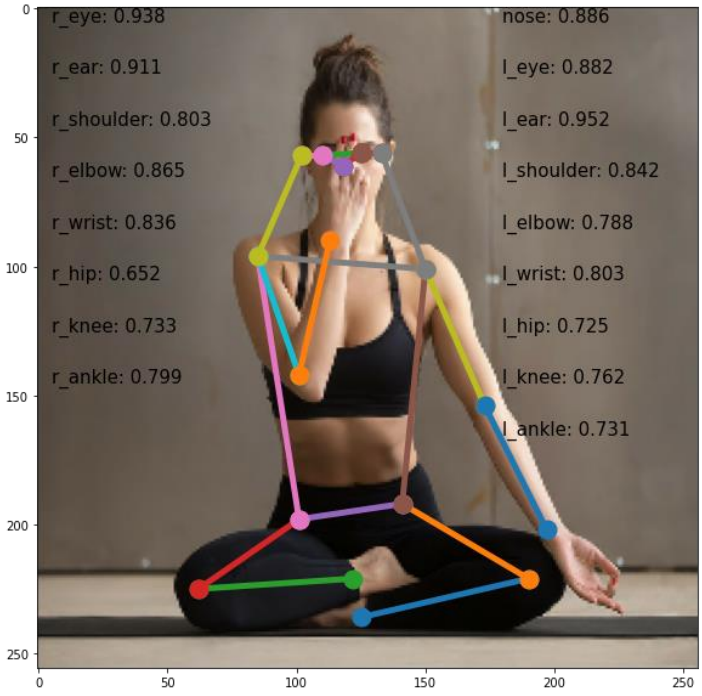


Figure 7.13: Prediction in skeleton format and confidence scores.

Figure 7.14: Predictions of one batch.

Figure 7.15: Failed cases.

As illustrated in Fig. 7.15, the model is certainly mistaking the person of interest with the nearby person. Single Pose Estimation is extremely sensitive to off center error. This error can come from various sources. For example, mislabeling of the bounding boxes, rounding errors during cropping, etc.

By reverting the operations in Section 5.2.2, the predicted keypoints can be converted back to the original image size, though not required.

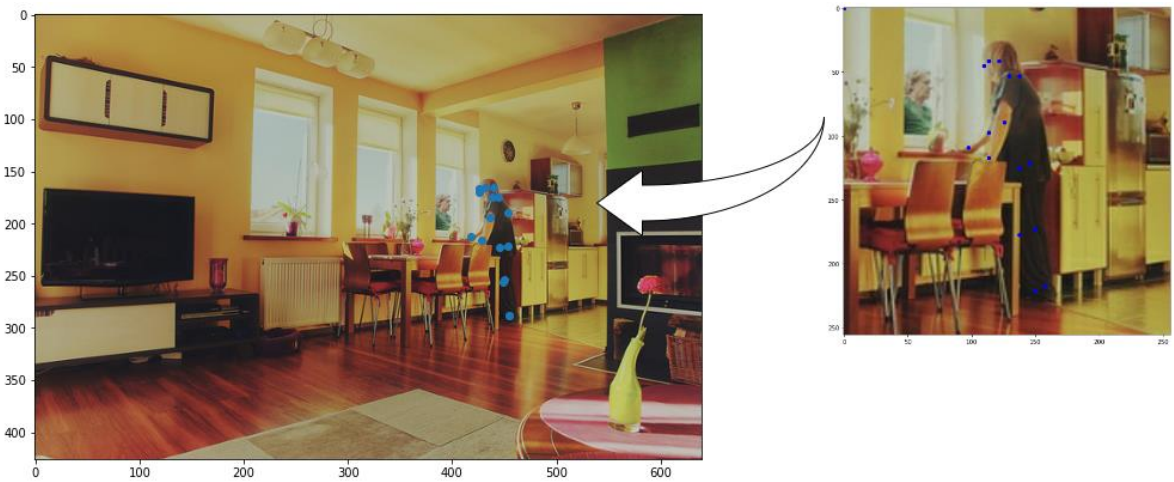


Figure 7.16: Reverted keypoints.

7.5 Stacked Hourglass Networks in multi-person pose estimation

Even though the focal point of this thesis is single-person pose estimation, it is a good benchmark to observe how trained models perform in a multi-person pose estimation context.

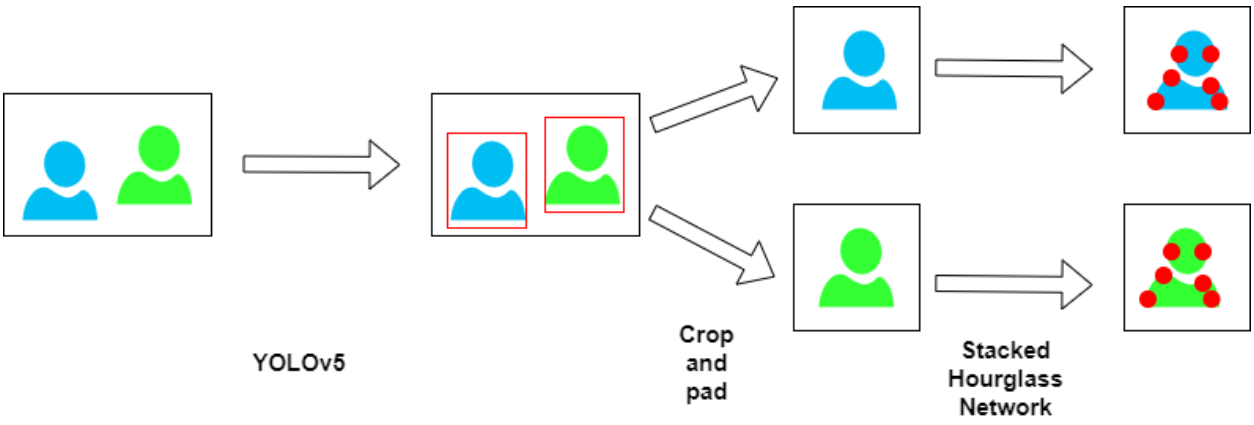


Figure 7.17: Multi-person pose estimation pipeline.

Figure 7.17 shows the top-down approach for this problem (see Section 1.1.5). Initially, an input image is passed through a person-detector, namely YOLOv5 (You Only Look Once version 5) [70]. The detector predicts human bounding boxes in the given image and using this data each person will be cropped and padded accordingly to produce instances for keypoint-detector. Each instance is processed by a 4-stack Hourglass Network to output keypoints. Lastly, these keypoints can be reverted and put back into the original image for visualization purposes.

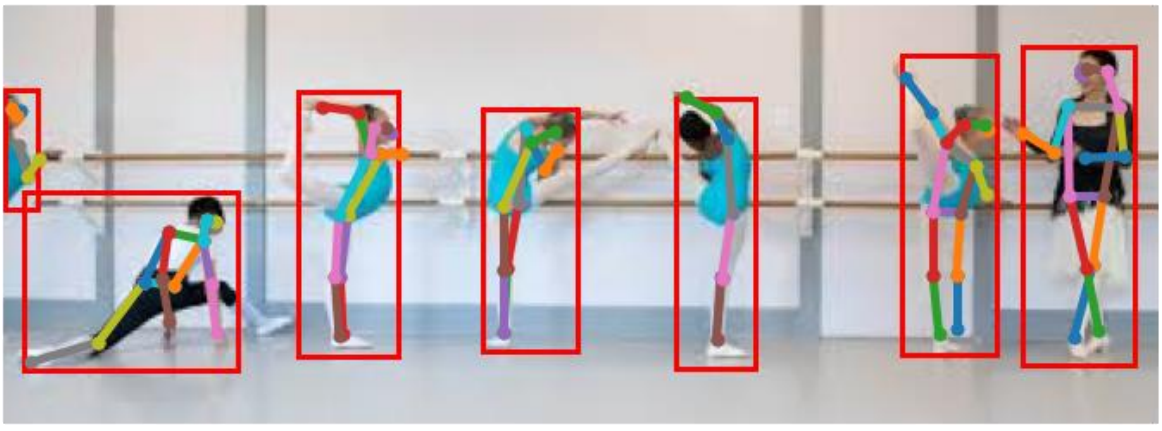


Figure 7.18: Output of the multi-person pose estimation pipeline.

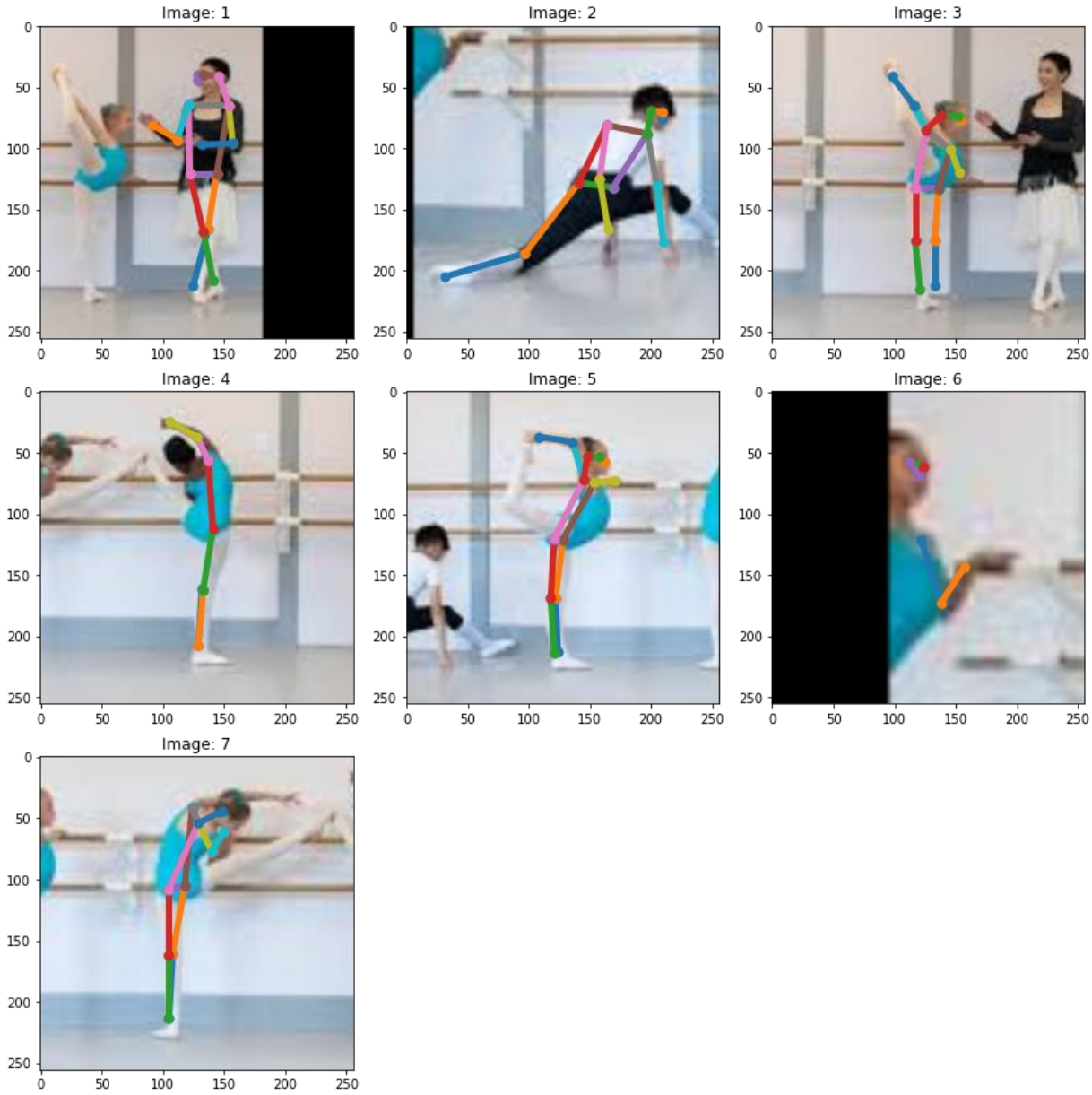


Figure 7.19: Stacked Hourglass Network's predictions separately.

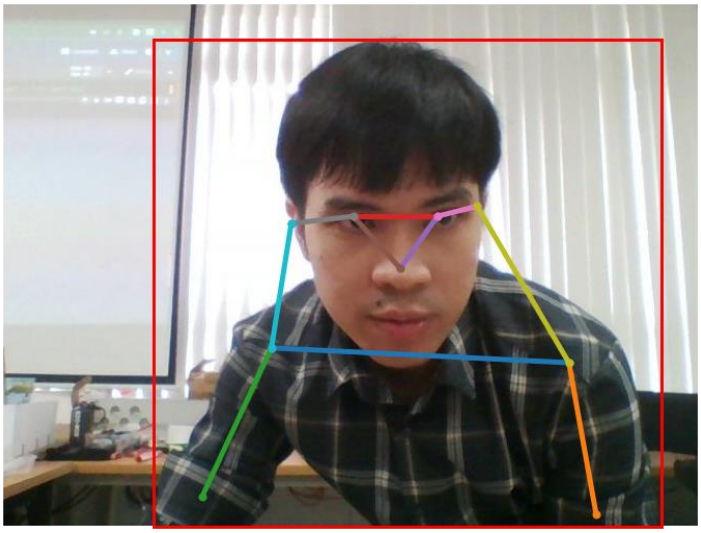


Figure 7.20: Prediction using input from webcam.

Despite of acceptable output, there are a few drawbacks with this approach. One of them is its high dependency on the person-detector. If the person-detector fails to detect the human instance, the entire pipeline fails. Secondly, the used model is not fine-tuned nor is it the light-weight version to be deployed in real-time environment. Therefore, there is a significant delay when using the video stream from a webcam as input

7.6 Conclusion and discussion

During the development of this project, I have learnt a considerate number of concepts, terminologies and bases in both Computer Vision and Deep Learning fields. Moreover, it helps me deepening my knowledge and practice with programming, particularly with Python. It is also a tremendous experience working with virtual environment.

The Stacked Hourglass Network architecture achieves satisfactory results on the COCO dataset despite the constraints of time and resources. Further improvement can be achieved by increasing training time, combining different datasets, better augmentation pipeline, adopting more training strategies, etc.

Human Pose Estimation is a very challenging task since it is highly non-linear and unconstrained. However, its potential applications are substantial. A good base line for single

HPE can catapult the project further. A good case in point is multi-instances Pose Estimation. Using a person detector, the instances can be extracted and the processed individually by the keypoint detector. Additionally, from keypoints estimation, a system which can categorize activities can be developed. The Pose Estimation problem can also be expanded into the third dimension using only 2D data. Poses from other objects should always be taken into consideration such as animals, robots or 3D models.

Chapter 8: References

[1] Gupta and Ayush, "Human Pose Estimation Using Machine Learning in Python," 26 October 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/10/human-pose-estimation-using-machine-learning-in-python/.

[2] hickeys and mobajemu-msft, "Kinect for Windows," 22 May 2022. [Online]. Available: https://docs.microsoft.com/en-us/windows/apps/design/devices/kinect-for-windows.

[3] C. ZHENG, W. WU, C. CHEN, T. YANG, S. ZHU, J. SHEN, N. KEHTARNAVAZ and M. SHAH, "Deep Learning-Based Human Pose Estimation: A Survey," *ArXiv*, vol. abs/2012.13392, 2019.

[4] M. Dantone, J. Gall, C. Leistner and L. V. Gool, "Human Pose Estimation Using Body Parts Dependent Joint Regressors," *2013 IEEE Conference on Computer Vision and Pattern Recognition,* pp. 3041-3048, 2013.

[5] Y. Yang and D. Ramanan, "Articulated pose estimation with flexible mixtures-of-parts," *CVPR 2011,* pp. 1385-1392, 2011.

[6] Z. Hu, Y. Hu, B. Wu and J. Liu, "Hand Pose Estimation with CNN-RNN," *2017 European Conference on Electrical Engineering and Computer Science (EECS),* pp. 458-463, 2017.

[7] A. Toshev and C. Szegedy, "DeepPose: Human Pose Estimation via Deep Neural Networks," *ArXiv,* vol. 1312.4659, 2013.

[8] R. A. Güler, N. Neverova and I. Kokkinos, "DensePose: Dense Human Pose Estimation In The Wild," *ArXiv,* vol. 1802.00434, 2018.

[9] S.-E. Wei, V. Ramakrishna, T. Kanad and Y. Sheikh, "Convolutional Pose Machines,"

ArXiv, vol. 1602.00134, 2016.

[10] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, W. Liu and B. Xiao, "Deep High-Resolution Representation Learning for Visual Recognition," *ArXiv*, vol. 1908.07919, 2019.

[11] R. A. Güler, N. Neverova, V. Khalidov and V. Khalidov, "Densepose: Dense human pose estimation in the wild," *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pp. 7297-7306, 2018.

[12] C. Lassner, J. Romero, M. Kiefel, F. Bogo, M. J. Black and P. V. Gehler, "Unite the People: Closing the Loop Between 3D and 2D Human Representations," *ArXiv*, vol. 1701.02468, 2017.

[13] C. Ionescu, D. Papava, V. O. Sminchisescu and Cristian, "Human3.6M: Large Scale Datasets and Predictive Methods for 3D Human Sensing in Natural Environments," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 36, 2014.

[14] T. v. Marcard, R. Henschel, M. Black, B. Rosenhahn and G. Pons-Mol, "Recovering Accurate 3D Human Pose in The Wild Using IMUs and a Moving Camera," *European Conference on Computer Vision (ECCV)*, 2018.

[15] M. Trumble, A. Gilbert, C. Malleson, A. Hilton and J. Collomosse, "Total Capture: 3D Human Pose Estimation Fusing Video and Inertial Sensors," *2017 British Machine Vision Conference (BMVC)*, 2017.

[16] M. Andriluka, L. Pishchulin, P. Gehler and B. Schiele, "2D Human Pose Estimation: New Benchmark and State of the Art Analysis," *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2014.

[17] S. Johnson and M. Everingham, "Clustered Pose and Nonlinear Appearance Models for Human Pose Estimation," *Proceedings of the British Machine Vision Conference*, 2010.

[18] B. Sapp and B. Taskar, "MODEC: Multimodal Decomposable Models for Human Pose Estimation," *In Proc. CVPR*, 2013.

[19] J. Tompson, A. Jain, Y. Lecun and C. Bregler, "Joint Training of a Convolutional Network and a Graphical Model for Human Pose Estimation," *NIPS*, 2014.

[20] J. Charles, T. Pfister, D. Magee, D. Hogg and A. Zisserman, "Personalizing Human Video Pose Estimation," *Computer Vision and Pattern Recognition*, 2016.

[21] T.-Y. Lin, M. Maire, S. Belongie, L. Bourdev, R. Girshick, J. Hays, P. Perona, D. Ramanan, C. L. Zitnick and P. Dollár, "Microsoft COCO: Common Objects in Context," *ArXiv*, vol. 1405.0312, 2014.

[22] "What is Perceptron: A Beginners Guide for Perceptron," Simplilearn Solutions, 18 September 2021. [Online]. Available: https://www.simplilearn.com/tutorials/deep-learning-tutorial/perceptron.

[23] W. McCulloh and W. Pitts, "A Logical Calculus of Ideas Immanent in Nervous Activity," *Bulletin of Mathematical Biophysics,* pp. 115-133, 1943.

[24] F. Rosenblatt, "The Perceptron—a perceiving and recognizing automaton," Cornell Aeronautical Laboratory, 1957.

[25] F. V. Veen, "THE NEURAL NETWORK ZOO," The Asimov Institue, 14 September 2016. [Online]. Available: Available: https://www.asimovinstitute.org/neural-network-zoo.

[26] S. Kostadinov, "Understanding Backpropagation algorithm: Introducing the math behind neural networks (Part 1)," 13 August 2019. [Online]. Available: https://www.linkedin.com/pulse/understanding-backpropagation-algorithm-introducing-math-kostadinov.

[27] J. Jordan, "Setting the learning rate of your neural network," 1 March 2018. [Online]. Available: https://www.jeremyjordan.me/nn-learning-rate.

[28] J. Mockus, "On Bayesian Methods for Seeking the Extremum," *Optimization Techniques,* pp. 400-404, 1974.

[29] M. R. Bonyadi and Z. Michalewicz, "Particle swarm optimization for single objective continuous space problems: a review," *Evolutionary Computation,* p. 1–54, 2017.

[30] A. Krizhevsky, V. Nair and G. Hinton, "CIFAR-10 (Canadian Institute for Advanced Research)," [Online]. Available: http://www.cs.toronto.edu/~kriz/cifar.html.

[31] D. H. Hubel and T. Wiesel, "Receptive fields and functional architecture of monkey striate," *The Journal of Physiology,* vol. 195, no. 1, pp. 215-243, 1968.

[32] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biol. Cybernetics,* vol. 36, p. 193–202, 1980.

[33] K. O'Shea and R. Nash, "An Introduction to Convolutional Neural Networks," *ArXiv,* vol. 511.08458, 2015.

[34] J. Brownlee, "How Do Convolutional Layers Work in Deep Learning Neural Networks?," Machine Learning Mastery," 17 April 2019. [Online]. Available: https://machinelearningmastery.com/convolutional-layers-for-deep-learning-neural-networks.

[35] A. D. Nishad, "Convolution Neural Network(Very Basic)," 2020. [Online]. Available:

https://www.kaggle.com/general/171197.

[36] T. Sharmin, F. Di Troia, K. Potika and M. Stamp, "Convolutional Neural Networks for Image Spam Detection," 2022.

[37] Algorithmia, "onvolutional Neural Nets in PyTorch," 10 April 2018. [Online]. Available: https://algorithmia.com/blog/convolutional-neural-nets-in-pytorch.

[38] Y. Li, X. Zhang and D. Chen, "CSRNet: Dilated Convolutional Neural Networks for Understanding the Highly Congested Scenes," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1091-1100, 2018.

[39] A. Newell, K. Yang and J. Deng, "Stacked Hourglass Networks for Human Pose Estimation," *ArXiv*, vol. 1603.06937, 2016.

[40] O. Ronneberger, P. Fischer and T. Brox, "U-Net: Convolutional Networks for Biomedical Image Segmentation," *ArXiv*, vol. 1505.04597, 2015.

[41] K. He, X. Zhang, S. Ren and J. Sun, "Deep Residual Learning for Image Recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770-778, 2016.

[42] T. Petrosyan, "OpenCV," 12 October 2021. [Online]. Available: https://opencv.org/introduction-to-the-coco-dataset/.

[43] "Introducing JSON," [Online]. Available: https://www.json.org/json-en.html.

[44] "COCO: Common Objects in Context," [Online]. Available: https://cocodataset.org.

[45] C. Patil and V. Gupta, "LearnOpenCV," 21 June 2021. [Online]. Available: https://learnopencv.com/human-pose-estimation-using-keypoint-rcnn-in-pytorch/.

[46] P. S. Parsania and P. V. Virparia, "A Comparative Analysis of Image Interpolation," *International Journal of Advanced Research in Computer and Communication Engineering*, vol. 5, no. 1, pp. 29-34, 2016.

[47] A. C. Johnsy, "Nearest Neighbor Interpolation," 2017. [Online]. Available: https://www.imageeprocessing.com/2017/11/nearest-neighbor-interpolation.html.

[48] "RGB Images, Image Processing Toolbox," Electrical and Computer Engineering, Northwestern University, [Online]. Available: http://www.ece.northwestern.edu/local-apps/matlabhelp/toolbox/images/intro8.html.

[49] J. Brownlee, "How to Manually Scale Image Pixel Data for Deep Learning," Machine Learning Mastery, 25 March 2019. [Online]. Available: https://machinelearningmastery.com/how-to-manually-scale-image-pixel-data-for-deep-

learning/.

[50] "TFRecord and tf.train.Example," Google, [Online]. Available: https://www.tensorflow.org/tutorials/load_data/tfrecord.

[51] "Protocol Buffers," Google, [Online]. Available: https://developers.google.com/protocol-buffers/.

[52] "Performance Guide," [Online]. Available: https://docs.w3cub.com/tensorflow~guide/performance/performance_guide.

[53] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro,...and Xiaoqiang Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015. [Online]. Available: https://www.tensorflow.org/.

[54] S. SWAIN, "Understanding Sequential Vs Functional API in Keras," 13 July 2021. [Online]. Available: https://www.analyticsvidhya.com/blog/2021/07/understanding-sequential-vs-functional-api-in-keras/.

[55] "Colaboratory: Frequently Asked Questions," Google, [Online]. Available: https://research.google.com/colaboratory/faq.html.

[56] "tf.data: Build TensorFlow input pipelines," [Online]. Available: https://www.tensorflow.org/guide/data.

[57] "Understanding of buffer_size in the dataset shuffle method," 2019. [Online]. Available: https://zhuanlan.zhihu.com/p/42417456.

[58] A. B. Jung, "imgaug," 2018. [Online]. Available: ttps://github.com/aleju/imgaug.

[59] "Module: tf.image," [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/image.

[60] D. Schidt, "Why tf.data is much better than feed_dict and how to build a simple data pipeline in 5 minutes," 27 August 2018. [Online]. Available: https://dominikschmidt.xyz/tensorflow-data-pipeline/.

[61] prakharr0y, "Intuition of Adam Optimizer," 24 October 2020. [Online]. Available: https://www.geeksforgeeks.org/intuition-of-adam-optimizer/#:~:text=Adam%20optimizer%20involves%20a%20combination,minima%20in%20a%20faster%20pace..

[62] O. Chernytska, "3D Hand Pose Estimation," *Ukrainian Catholic University,* 2019.

[63] [Online]. Available: https://cocodataset.org/#keypoints-eval.

[64] S. M. Ribera, "Computer Vision Analysis of the body-pose similarity from two different subjects with the aim of the correct development of physical exercises," UNIVERSITAT POLITECNICA DE CATALUNYA, 2020.

[65] "cocoapi," [Online]. Available: https://github.com/cocodataset/cocoapi.

[66] E. Y. Li, "Human Pose Estimation with Stacked Hourglass Network and TensorFlow," 15 March 2020. [Online]. Available: https://towardsdatascience.com/human-pose-estimation-with-stacked-hourglass-network-and-tensorflow-c4e9f84fd3ce.

[67] Y. Chen, Z. Wang, Y. Peng, Z. Zhang, G. Yu and J. Sun, "Cascaded pyramid network for multi-person pose estimation," *CoRR,* vol. 1711.07319, 2017.

[68] B. Xiao, H. Wu and Y. Wei, "Simple baselines for human pose," *ECCV*, pp. 472-487, 2018.

[69] J. Wang, K. Sun, T. Cheng, B. Jiang, C. Deng, Y. Zhao, D. Liu, Y. Mu, M. Tan, X. Wang, W. Liu and B. Xiao, "Deep High-Resolution Representation Learning for Visual Recognition," *ArXiv*, vol. 1908.07919, 2019.

[70] R. Couturier, H. N. Noura and O. S. a. A. Sider, "A DEEP LEARNING OBJECT DETECTION METHOD FOR AN," *ArXiv*, vol. 2104.13634, 2021.