Gait Recognition Using Convolutional Neural Network

Aziza Duisembay, Electrical and Electronics B.Eng

Submitted in fulfilment of the requirements for the degree of Master of Science
in Electrical and Computer Engineering

School of Engineering Department
Electrical and Computer Engineering
Nazarbayev University

53 KabanbayBatyr Avenue,
Nur-Sultan, Kazakhstan, 010000

Supervisor: Grant Ellis

Date of Completion 17.04.2020

DECLARATION

I hereby, declare that this manuscript, entitled “Gait Recognition Using Neural Network”, is the result of my own work except for quotations and citations which have been duly acknowledged. I also declare that, to the best of my knowledge and belief, it has not been previously or concurrently submitted, in whole or in part, for any other degree or diploma at Nazarbayev University or any other national or international institution.

_____

Name: Aziza Duisembay

Date: 17.04.2020

Abstract

This research focuses on building a CNN for gait recognition. It recognizes a person based on his/her walking habits and motion. Because a gait data acquisition can easily be done at any distance from the subject and even with a low camera resolution, gait recognition has become a popular biometric parameter for human recognition. It is difficult to extract gait features manually for machine learning. Hence, in this thesis, a 10-layer convolutional neural network (CNN) that processes and classifies gait energy images is proposed. It learns to extract important distinctive features without human intervention. The main difference of the proposed neural network from the previous studies on applying DNN for gait recognition is that it is robust to view variations and has a high recognition rate.

The networks with the same architecture but different activation functions and optimizers were evaluated and compared based on the achieved accuracies, losses, and computational time. The optimal activation functions are Tanh for intermediate layers, and Softmax for the last layer. The most efficient optimizer for CNN is Adam. The developed CNN was trained on CASIA B dataset and achieved an accuracy rate of 98.58%. Moreover, an LRP technique was applied to verify the network's adequate classification. LRP generates heat maps that show which areas of the input image affect the classifier's decisions the most.

Acknowledgements

First of all, I would like to express my appreciation for my supervisor Professor Grant Ellis for his continuous help and supervision of this thesis work.

I also want to thank my family for being supportive and providing me encouragement and emotional help throughout the master's program.

Thirdly, I want to thank my friend Akzharkyn Izbassarova for giving me so much advice regarding the thesis and education in general.

Table of Contents

List of Figures

List of Abbreviations

Adagrad	Adaptive Gradient
Adam	Adaptive Moment estimation
Adaprop	Adaptive Propagation
CNN	Convolutional Neural Network
DNN	Deep Neural Network
ELU	Exponential Linear Unit
GEI	Gait Energy Image
HMM	Hidden Markov Model
JITN	Joint Intensity Transformer Network
k-NN	k-nearest neighbors
LSTM	Long Short-Term Memory
LRP	Layer-wise Relevance Propagation
MGAN	Multitask Generative Adversarial Network
Nadam	Nesterov-Accelerated Adaptive Moment estimation
NN	Neural Network
PEI	Period Energy Image
ReLU	Rectified Linear Unit
RMSprop	Root Mean Square Propagation
SVM	Support Vector Machine

List of Tables

Chapter 1 - Introduction

1.1 Gait Recognition Overview

Nowadays, a reliable and robust person identification system has become a necessary component of many real-time applications, like disease diagnosis, secure money transactions, building security, etc. To correctly identify an individual, a person recognition system analyzes one or several body parameters such as face, eye iris, voice, hand geometry and others. Those characteristics are called biometric parameters. According to Jain et al., any biological or behavioral characteristics of an individual can be used as a biometric feature for recognition if it is unique, constant, and collectable [1]. The researchers analyzed main human biometric parameters and concluded that it is more possible to circumvent face and voice than fingerprint, gait, iris, or ear.

Thus, recently, gait recognition has become one of the most popular biometric person identification approaches. There are several reasons for this. First of all, gait recognition does not require any cooperation from subjects. Secondly, it works fine even at a distance (usually 10m or more). Thirdly, a video resolution does not affect a network's performance dramatically. And finally, a walking style of a person can hardly be imitated.

In the past 20 years, considerable research has been conducted on gait recognition [2]. Depending on a data acquisition method, gait recognition can be divided into four types: systems with data gathered from cameras, accelerometers, wave radars, and floor sensors. This research is focused on video-based gait recognition using deep learning algorithms. Deep learning is a machine learning method of developing techniques inspired by the construction and operations of a human brain. The crucial advantage of deep learning is that a developer does not have to point to the features of data because the network extracts its distinguishing features by itself. The recent progress of deep learning in computer vision, speech recognition, text processing, and other various fields has encouraged researchers to implement it in gait recognition as well. Some of the existing problems in video-based gait recognition are the following: (1) a clothing covariate, (2) a view covariate, (3) a dynamic background. A dynamic background problem relates to challenges of background subtraction, and is not covered in this research. So, the problem statement of this

research is the development of a robust camera-based gait recognition network based on deep learning algorithms insensitive to a clothing and view covariate.

1.2 Literature Review

In vision-based gait recognition, there are two existing ways to represent the gait data: model-based approach and model-free approach [3]. Model-free approaches are focused on identifying subjects by extracting their shape and geometrical measures, while model-based approaches extract the kinematic characteristics of the gait. 3D body models are a popular way to represent data in model-based gait recognition. They can be obtained using a particle swarm optimization algorithm [4], second-generation Kinect V2 tool [5], [6], etc. For the gait classification, B. Kwolek et al. used an SVM classifier and reached 99.6% accuracy on the dataset composed of 22 subjects. X. Chen et al. [7] proposed to use a k-nearest neighbours (k-NN) algorithm to classify 3D tensor gait features and achieved the accuracy of 80.3% for frontal view and 89.2% for lateral view.

However, this research is focused on the model-free gait representation approach. It is computationally expensive to extract 3D features from the video, since it should be captured by more than two cameras from different angles of view, and the algorithms listed above are more complex and time-consuming than simple background subtraction used for model-free approaches [3]. After the background subtraction, a silhouette on a black background is obtained, which is robust to the color and texture changes. The silhouettes often go through a further preprocessing for a better capture of the gait data. A. Sokolova et al. [8] proposed to use maps of optical flows developed by Simonyan for the human activity recognition task [9]. The optical flow maps are obtained from groups of two consecutive input frames. For long-duration actions, several consecutive flow maps are collected into blocks.

The second way to represent the gait cycle is gait energy images (GEIs) [10], [11]. It is a popular technique that finds the mean matrix of all normalized frames belonging to one gait cycle.

In [12], the researchers use a period energy image (PEI) which is a normalized mean matrix of frames that have amplitudes belonging to the same range of values. They define an amplitude of a frame as a normalized mean width of the leg region. The process of obtaining the PEIs is presented in Figure 1.1.

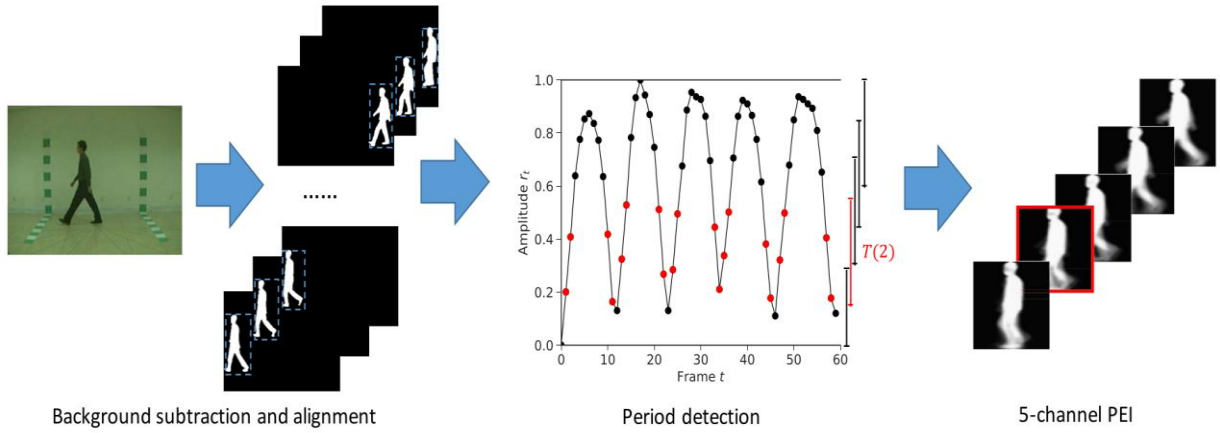

Figure 1.1: Computation of PEI [12]

Despite being a simple and fast way to represent data, a model-free approach is sensitive to clothing conditions, scaling, and view variations. A popular method to overcome these problems is building a neural network trained on a multi-view dataset with different clothing and carrying conditions, e.g. CASIA B gait dataset.

In [10], the authors developed a CNN built of four convolutional layers, each of which is followed by a max-pooling layer, and two densely connected layers. The final classification is performed with the Softmax activation function of the output layer. The number of convolving filters for each layer are 16, 32, 64, and 124 correspondingly. The network is trained separately for each angle of view. Thus, there are 11 CNNs with different weights for each angle. The mean accuracy of the networks is 98.8%. A. Sokolova et al. [8] proposed to apply pre-trained neural networks, VGG-19 and Wide ResNet specifically, to extract features from the gait data and then apply a k-NN classifier to predict the class of the object. Trained on CASIA B, the Wide ResNet model has reached an accuracy of 92.95%. In [11], the silhouette gets split into four parts. Each of them is fed as input into a separate CNN followed by a long short-term memory (LSTM) attention model. The accuracy of the network is equal to 86.5% on CASIA B and 74.3% on OUMVLP.

Y. He et al. [12] developed a multitask generative adversarial network (MGAN) model composed of the following components:

- an encoder that transforms the PEIs into features in a potential space,
- a classifier computing the angles of view of the obtained features,
- a view conversion level that maps the obtained features to a different angle of view,
- a generator computing PEIs from the generated features,

- a discriminator deciding which domains or distributions the PEIs are from.

The accuracy of the MGAN is 74.6% on the CASIA B dataset, 93.2% on OU-ISIR, and 94.7% on USF.

S. Bei et al. [13] extract gait data to a subGEI to acquire temporal gait features. Unlike GEI, subGEI is not a mean matrix of images of one whole gait sequence, but *n* mean matrices for *n* consecutive parts of the gait sequence. The optical flow of the extracted subGEIs and the GEIs are fed to a two-stream CNN model. The best performance was achieved using the Inception-V3 pre-trained model. Next, output scores of the two CNNs were fused. The results of the networks tested on the CASIA B dataset are presented in Figure 1.2. The CNNs are trained individually for the subGEIs that are extracted after dividing a whole gait sequence into 4, 6, and 8 parts. In the figure below, they are denoted as TL1, TL2, and TL3 respectively.

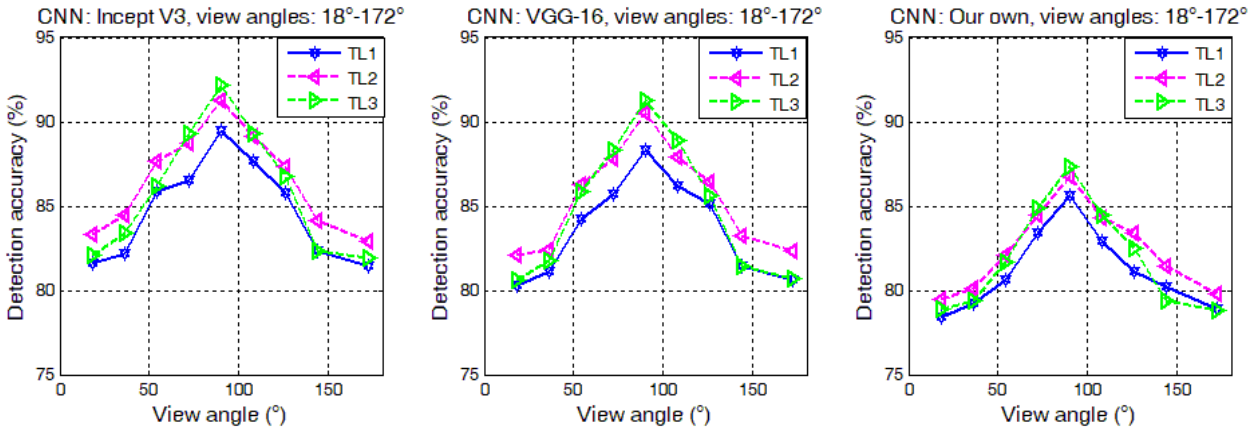

Figure 1.2: Results of the two-stream CNN [13]

In [14], the researchers fuse a support vector machine (SVM) classifier and a hidden Markov model (HMM) using the Bayes combination technique. The SVM classifies dynamic gait features, and the HMM's input data is GEIs. The model's accuracy rate of performance on the OU-ISIR dataset is 96.2%. A resembling method was proposed by X. Li et al. [15] who developed a unified joint intensity transformer network (JITN) for robust classification. It consists of three processes: computing the intensity metric for a given pair of probe and gallery matrices; generating a spatial dissimilarity of the given pair of matrices; next, the computed spatial dissimilarity feature map goes as an input to a DNN which predicts the final dissimilarity scores. After training on the OUTD-B dataset, the model achieved 85.9% accuracy.

S. Li et al. [16] proposed a model that takes pairs of gait images as input. The model's processing is divided into several parts. First, a pre-trained ResNet-50 model extracts spatial feature maps. They are fed as input to the spatial attention components that generate identity-related semantic features. LSTM units process the generated features and compute the temporal feature sets. Next, the attentive temporal summary components set various weights to the timestamps and compute sequence-level features. Finally, the generated features are matched to a learned discriminative space using the Null Foley-Sammon Transform method. The results show that the model's accuracy varies between 35.5% and 98% depending on the probe view and corresponding test views.

In terms of the number of people, the state-of-the-art gait recognition system is developed by A. Sokolova et al. [8]. The system recognizes 124 people from the CASIA B dataset with the accuracy rate 92.95%.

Chapter 2 - Description of Research and Work Conducted

2.1 Gait Dataset

The dataset used for the gait recognition task in this project is the CASIA B dataset. It was collected in 2005 by Intelligent Recognition & Digital Security Group formed by Tieniu Tan at National Laboratory of Pattern Recognition. The dataset includes video files that contain gait data collected from 124 people. It is a multi-view dataset that was captured from 11 views at the same time. Moreover, it includes some variations, namely, clothing and carrying condition changes. It also provides the pre-processed videos with extracted silhouettes on the black background. This has been achieved using a background subtraction method. However, for some classes, significant amount of information is lost, i.e. not full silhouettes are provided. Hence, 4 corrupted classes were removed, so that 120 remain. The gait data is gathered from 85 men and 35 women.

Figure 2.1: Images from CASIA B dataset

Storing the frames obtained from the video files requires a lot of memory space, and their processing is computationally expensive. To overcome those problems, gait energy images were introduced by J. Hua and B. Bhanu [17]. Basically, a GEI is a mean vector of all normalized frames belonging to one gait cycle. It represents the gait data in both temporal and spatial domains.

Suppose each $I_n(x, y)$ is a distinct pixel at position (x, y) of frame n, where $n = 0,1, \ldots, N$, and N is a number of frames of one gait cycle. Each frame is normalized along horizontal and vertical axes to a size of 240 x 240 pixels. Here, a normalization of an image means finding and cropping the region that contains the silhouette only, and resizing it so that all silhouettes have the same height.

So, a GEI is defined as following:

$$G = \frac{1}{N}\sum_{n=0}^{N} I_n(x, y) \quad (2.1)$$

Thus, a gait energy image allows to decrease the data size and the computational cost, still preserving the temporal information. The main disadvantage of GEIs is that they lose a part of temporal information of the gait sequence. More specifically, it loses information about the order of body motions [18].

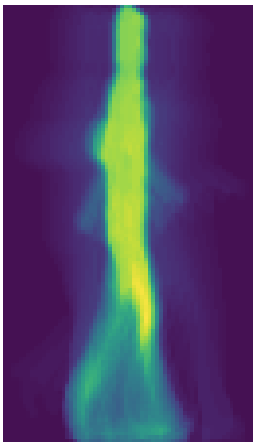

Figure 2.2: Gait energy image

An example of a GEI is presented in Figure 2.2. It contains static and dynamic gait information. Static features do not change their position on the frames after the normalization. So, the static parts are head and body, which are represented as bright areas of the GEI. The dynamic parts are hands and legs, which are rather dim areas of the gait image.

2.2 Convolutional Neural Network

As mentioned in Chapter 1.1, to build a system that can identify a person by his/her gait, it has been decided to use deep learning models. Deep learning is a branch of machine learning that develops different techniques inspired by a construction and operations of a human brain, and are designed to recognize patterns. A set of such algorithms is also known as a neural network. The main goal of a gait recognition neural network is to train on a labeled dataset so that after training it could extract features of each person's gait and classify them.

Neural networks train on a dataset by constantly adjusting the weights and biases of its neurons. Based on the research found in the Literature Review on gait recognition, it was concluded that CNN deals with gait images better than other types of neural networks. Like most of neural

networks, CNN is a set of neurons connected with each other in a directed acyclic graph. Neural networks are commonly divided into distinct layers of neurons, and the output of one layer is fed as input to the next layer. The last layer is an output layer. In classification problems, its neurons contain values of class probabilities. The structure of a 3-layer fully-connected neural network is presented in Figure 2.3 (input layer is not considered as a neural network's layer). A neural network is called fully-connected if it is only composed of dense layers.

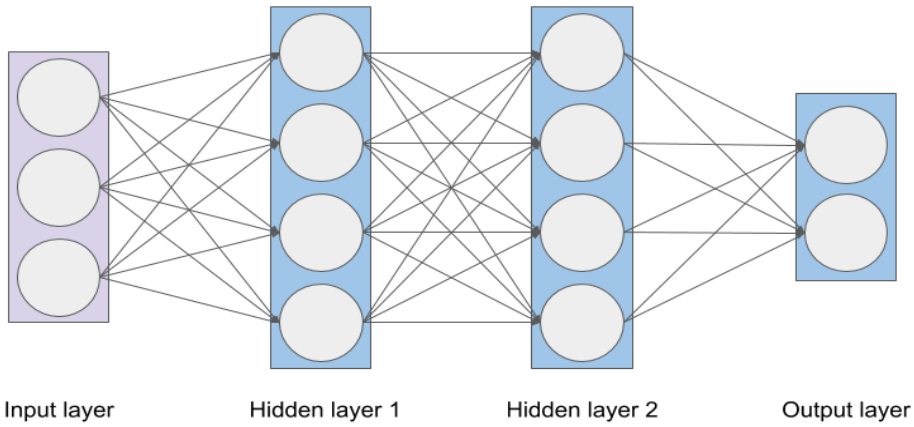

Figure 2.3: Structure of a fully-connected neural network

However, when the input image is large and has more than one channel, a fully-connected neural network becomes too large (in terms of a quantity of neurons) and wasteful. Hence, it is more efficient to use CNN to process image data. Unlike a fully-connected neural network, neurons of convolutional layers are organized in 3 dimensions: width, height, and depth. A filter of a convolutional layer convolves the input activations. The neurons in a CNN layer are only connected to a small number of neurons of the lower layer. In contrast, in dense layers, every neuron of one layer is connected to every neuron of its neighbor layer.

The architecture of a small CNN is presented in Figure 2.4 as an example. The model has two convolutional and two max-pooling layers. When an input matrix passes through a convolutional layer, it transforms into an array of feature maps. The generated maps have identical sizes which are equal to the input's size. Each filter in the convolutional layer produces its feature map. In the given example, max-pooling operation decreases the input's size by the factor of two.

The CNN also has 2 dense layers: a hidden layer and an output layer.

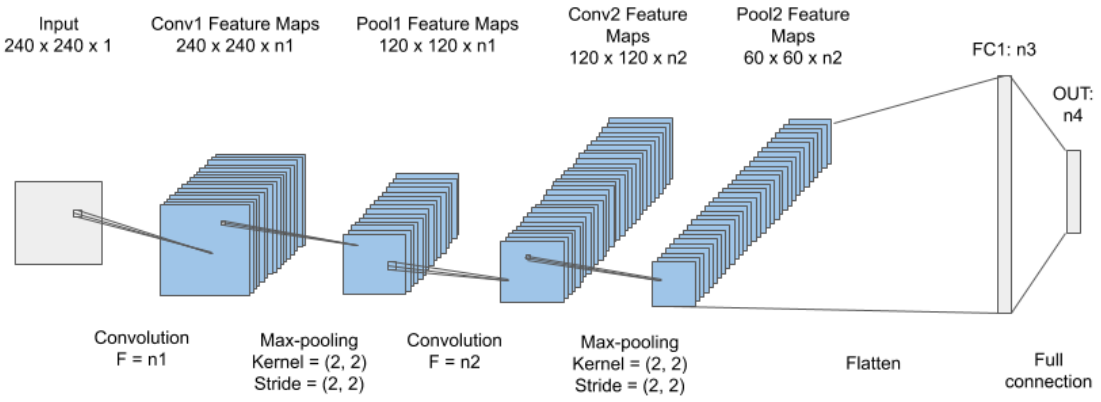

Figure 2.4: Structure of a CNN

So, a dimension of the output layer for a CASIA B image would be 1x1x120. In the final layer, CNN reduces the input matrix to a vector of class probability scores arranged along the depth dimension.

2.2.1 Convolutional Layers & Their Parameters

The name for CNN comes from a convolution operator. Let's introduce its definition. Suppose we have functions f and g. Their convolution is then defined as the integral of their product after one is reserved and shifted by some time τ:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t-\tau)d\tau \tag{2.2}$$

In deep learning, the convolution's aim is to extract distinctive characteristics of the input image. So, there is a 2D matrix, called a filter, that slides, i.e. convolves, around the input matrix. Each filter slides across the input and calculates a dot product of the filter matrix and the corresponding part of the input matrix. As the filter slides over two dimensions of the input matrix, it computes a 2-dimensional activation map. The produced map represents the filter's reactions at every position of the input volume. Figure 2.5 shows an example of a 3x3 filter's operation over an input image presented as a matrix of pixel values. A stride is a step size at which the filter convolves over the input volume. In the given example, the stride is equal to 1.

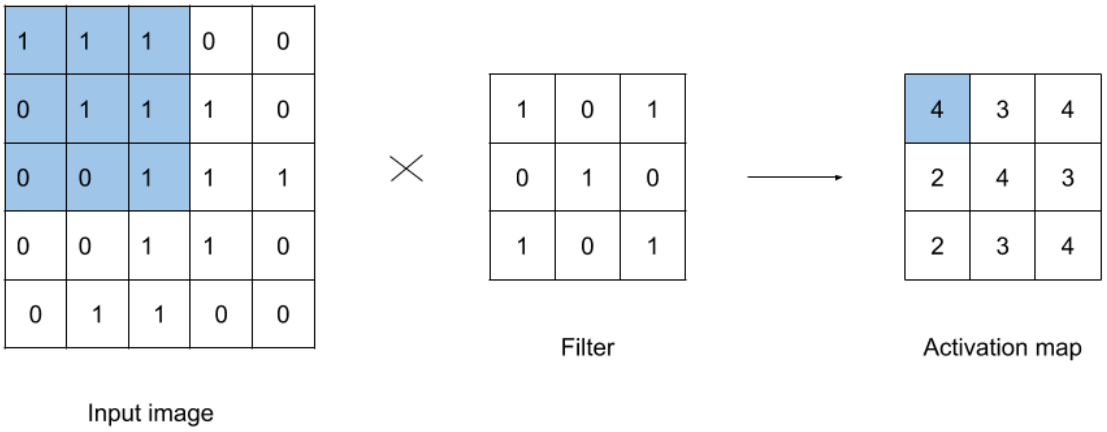

Figure 2.5: Convolution operation

By the end of the training process, the filters will activate when they slide across some visual pattern, like an edge of some object on the first layer, or a whole shape of an object on higher layers of the model.

The CNN developed for the gait classification problem has 10 layers in total. It is not too deep for the sake of fast image processing. 4 convolutional and 4 max pooling layers were included to extract features, and 2 dense layers to classify the features. Each convolutional layer has its hyper-parameters, such as a number of learnable filters, a kernel size of a filter, a stride, a zero padding, and an activation function. Parameters of dense layers are a number of neurons and a dropout rate.

The convolutional layers have the following parameters:

- The number of filters is different for each layer: 16, 32, 64, and 124 filters are used in the 1st, 2nd, 3rd, and 4th convolutional layers correspondingly.
- The filters' kernel size can be individual for each convolutional layer. The filters are small along the width and height axes, but their depths are equal to the depth of the input image. In the proposed model, all filters on the convolutional layers have kernel size of 4x4 (width and height respectively). The output's depth size is identical to the quantity of filters located on the layer.
- The stride defines a step size at which a filter slides an input. In the research, each convolutional layer's stride value is equal to 2.
- Zero padding allows to control the width and height of the output volume by adding zero pixels on the borders of the input. In the proposed CNN, zero padding is set as "same" which

means that it adds so many zero pixels to the output image that it will have the same size as the original input.

- Activation layer always performs a non-linear transformation over the output of the convolutional or dense layer. All operations in those layers are linear, which does not let the network learn complex patterns of the input data.

During the research, different activation functions were tried, like a rectified linear unit (ReLU), exponential linear unit (ELU), Leaky ReLU, and Tanh. The comparison of the results is presented in Chapter 3.2. However, the highest accuracy was achieved using a Tanh activation function. Tanh calculates the scores according to the following formula:

$$f_{tanh}(x) = \frac{2}{1+e^{-2x}} - 1, \tag{2.3}$$

where x is a weight vector of a convolutional layer.

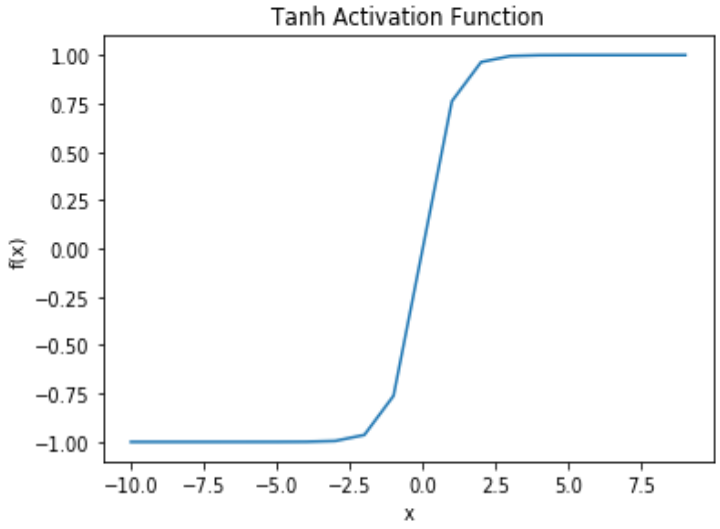

Figure 2.6: Graph of Tanh

Tanh is not a very popular activation layer for CNNs because of a vanishing gradient problem. This problem usually occurs during a training process of a neural network that uses a gradient-based optimization algorithm and an activation function like Sigmoid or Tanh. As can be seen in Figure 2.6, Tanh saturates the input, and even very large (or very small) input scores become equal to 1 (or -1) after passing through the activation function. Sigmoid behaves almost the same way except its output range is [0, 1]. In contrast, ReLU's output is directly proportional to the input.

So, it has large output scores if the input values are large. But, with Tanh and Sigmoid, the final scores of the network become small.

During backpropagation, the partial derivatives of the loss function (gradients) are passed back through the network to update its weights. The issue is that in some cases, the gradients are very small. And as the number of layers increases, the gradients of a loss function approach 0, hence the name of the problem. Such small gradients cause very small change in weights. This slows down the training process, or may even stop it whatsoever.

A common solution of this problem is to use some activation function that does not have saturation properties, e.g. ReLU and its later advanced versions [19].

For shallow networks, though, vanishing gradient is not a big problem. The CNN developed during this research is not very deep: it has only 4 convolutional layers and 1 dense hidden layer. So, the practical results demonstrate that, with Tanh, the network learns better than with a ReLU activation function that was designed to solve the vanishing gradient problem.

ReLU is a piecewise linear function:

$$f_{ReLU}(x) = max(0, x) \tag{2.4}$$

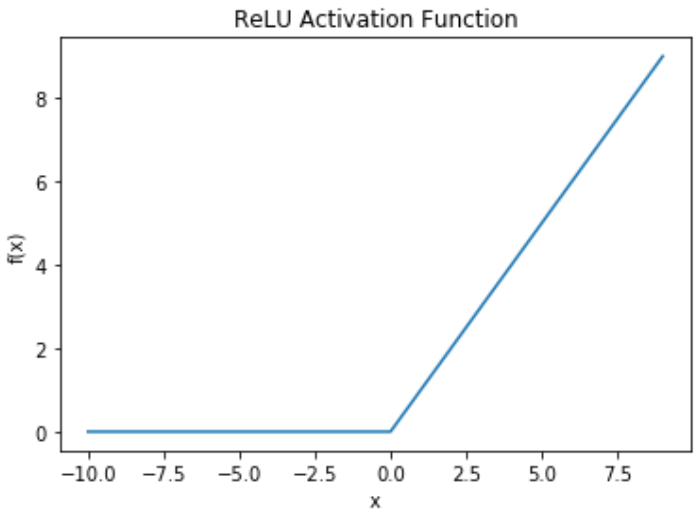

Figure 2.7: Graph of ReLU

Figure 2.7. demonstrates the graph of ReLU activation function. Even though in this case the Tanh beats the ReLU, the CNN with ReLU has shown a good performance with 96.58% accuracy.

Another activation function that has shown even better performance than ReLU is the Leaky ReLU. The function's graph is presented in Figure 2.8, and it is defined as:

$$f_{ReLU}(x) = max(0.01x, x) \tag{2.5}$$

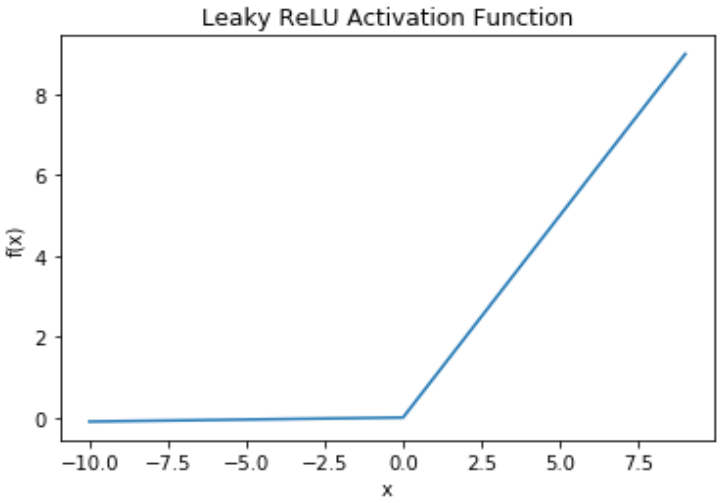

Figure 2.8: Graph of Leaky ReLU

2.2.2 Max-Pooling Operation

Max-pooling is a simple operation over an input vector that results in the input's dimension reduction by summarizing the features of the input. Pooling makes the input's representation become invariant to minor transformations of the input. Invariance to transformations means that if the input changes by a small amount, the values of its pooled version will not change [20]. Max-pooling process can be presented as a window that slides over the input and returns the maximum value that the window contains. Thus, it lowers the computational complexity for further processing. For example, in Figure 2.9, we consider a window of size 2 x 2 with a stride equal to 2. Thus, it contains 4 pixel values, and outputs a single value, reducing the input vector's size by the factor of 4.

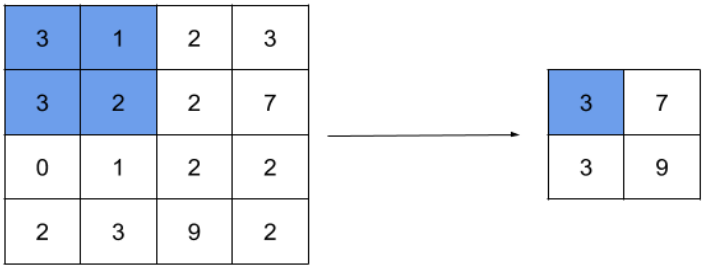

Figure 2.9: Max-pooling operation

The proposed CNN has max-pooling layers with a window size equal to 2 x 2 and stride that's equal to 2, just like in the example above. There are other types of pooling, like average and minimum, but max pooling is the most common, since it extracts the most important features like edges whilst minimum and average give a vague and smooth information. After the pooling, we lose a lot of data about the features of the image. To prevent large data loss, the window size is usually not set to a value larger than 3 x 3.

2.2.3 Dense Layers

Before discussing the dense layers' parameters, we need to note that it is not possible to feed the output of the convolutional layer directly to the dense layer. The reason is that convolutional layers give an output in the form of a 2D matrix whereas the dense layers only accept vectors. That is why in between these two layers it is necessary to place a 'Flatten' layer. It transforms a 2D feature map into a vector. An example of a flattening operation is shown in Figure 2.10.

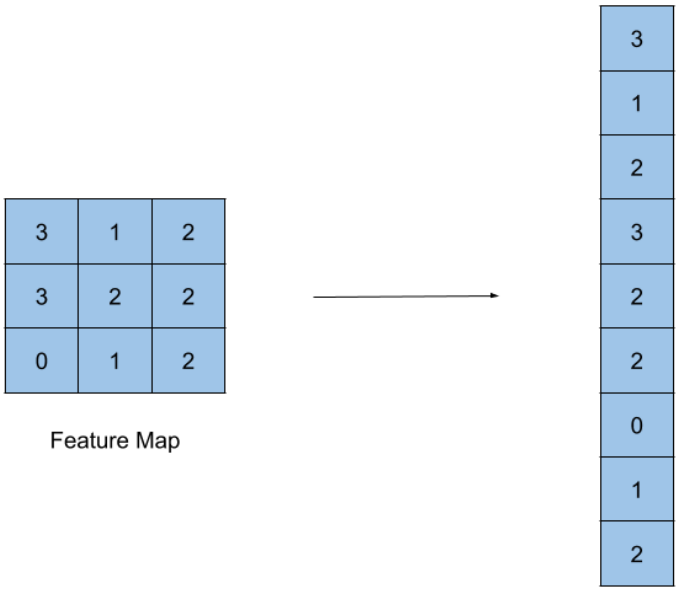

Figure 2.10: Flattening operation

Dense layers serve for classification of the extracted features. As it was mentioned before, the network has two dense layers: the hidden and the output layers.

- Number of neurons in the dense layers: 1024 neurons in the hidden layer, and 120 neurons in the output layer. The number of neurons in the output layer should always be equal to the number of categories in the dataset. Usually, an optimal number of neurons in a hidden layer

should be less than the size of the input layer and bigger than the size of the output layer. Since the input size is 57600 (i.e. 240 x 240) and the output size is 120, 1024 neurons were used. The value is much closer to 120 because a large hidden layer requires a lot of computational time and power.

- Dropout is a technique aimed to reduce overfitting [21]. Overfitting is an error that occurs when a neural model is very well trained on a limited set of data points but shows a poor performance when processing some unseen data. So, dropout randomly removes certain features by setting corresponding weights to 0, thus, not letting the model fit the training data too well. Figure 2.11 demonstrates a dropout with a rate of 0.5 applied to the first hidden layer.

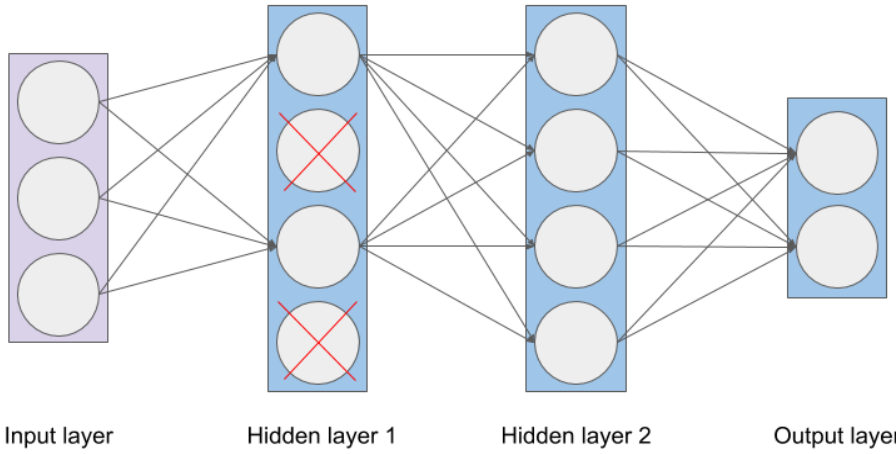

Figure 2.11: Dense neural network with a 50% dropout

A 35% dropout was added to the first dense layer which means that at every training iteration, 35% of its weights are randomly chosen to be set to 0.

- Activation function. The hidden dense layer has a Tanh activation function, just like hidden convolutional layers. But the activation function of intermediate layers is not suitable for the output layer, since the last activation function should produce probability scores of the classes. The probability score defines how likely the input image belongs to the corresponding class. The sum of all probability scores is usually equal to 1.

Because it is a multi-class classification task, Softmax is an optimal activation function for the output layers. This function is designed to be an activation function of the last neural layer. It is the only activation function that computes the chances that the input data belongs to a particular class by computing probability scores in the range [0, 1]. So, it performs a probabilistic explanation of the output.

As mentioned before, the output layer is as large as is the number of categories in the dataset. Softmax takes its output scores from those neurons and computes the probability scores assigned to each class. Figure 2.12 shows the graph of Softmax. Assuming that we have C classes, and $f_{Softmax}(x)_i$ is a probability score for each of them, i.e. $i = 1, 2, \ldots, C$, and s_i is an output score of the i^{th} neuron of the last dense layer, Softmax function is defined as:

$$f_{Softmax}(x)_i = \frac{e^{s_i}}{\sum_{j=1}^{C} e^{s_j}} \tag{2.6}$$

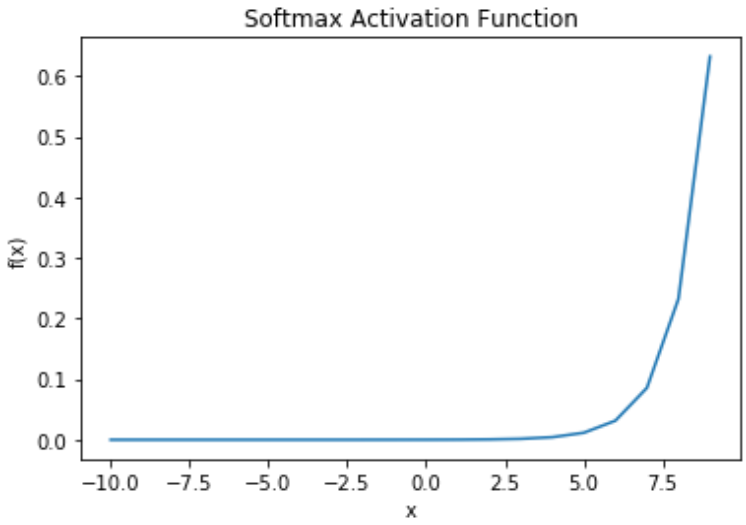

Figure 2.12: Graph of Softmax

2.2.4 Loss Functions

After the model makes its predictions, it compares its output with the actual class labels using a loss function. Loss function is a minimization function that calculates the error of CNN. The more the model's outputs deviate from the actual data, the larger a loss function's output would be. Loss, i.e. the output of the loss function, is used to update the model's weights to reduce the loss on the next iteration.

Loss functions can be divided into two main groups: classification losses and regression losses. Classification losses are used when the task of the neural network is to classify data into a finite set of categories. But if the network needs to predict a label of the input data from a set of infinite values, the regression losses are commonly used. For example, defining the cost of a house or the length of a plant belongs to regression problems. Some of the classification losses are Kullback-Leibler divergence loss, cross-entropy loss, and multiclass hinge/SVM loss.

Kullback-Leibler divergence loss calculates how much the model's output probability distribution differs from a true output distribution. It can be formulated as:

$$L_{KL} = \frac{1}{M}\sum_{m=1}^{M}\sum_{j=1}^{C} y_{mj}(log\ y_{mj} - log\ s_{mj}), \quad (2.7)$$

where M is the number of training samples; C is the number of classes; s_{mj} is a predicted probability score that the input m belongs to class j; andy_{mj} is a true probability score of sample m belonging to class j.

Categorical cross-entropy loss is a special case of Kullback-Leibler divergence loss where the labels are one-hot encoded into vectors of 0's and 1's. In a single-label classification, 1 denotes the target class, and other 0's indicate all other classes. Cross-entropy loss computes the sum of the average differences between the actual and predicted output scores for all classes in the problem. Mathematically, it is defined as:

$$L_{CE} = -\frac{1}{M}\sum_{m=1}^{M}\sum_{j=1}^{C} y_{mj} log\ s_{mj}, \quad (2.8)$$

where s_{mj} is a predicted probability score and y_{mj} is a true probability that the input belongs to class j.

Hinge/SVM loss works according to the following principle: the difference between the probability score of the correct class and each probability score of all incorrect classes should be positive and larger than some threshold value (usually equal to 1). It is expressed as:

$$L_{hinge} = \frac{1}{M}\sum_{m=1}^{M}\sum_{j=1,j\neq i}^{C} max(0, s_{mj} - s_{mi} + 1) \quad (2.9)$$

Here, all notations are the same as in the previous equations, and s_{mi} is a predicted probability score that the input m belongs to its true class i.

The hinge loss finds not only incorrect predictions, but even correct but not certain ones. The penalties for wrong predictions are large, for correct but not certain predictions, slightly less, and only certain and correct predictions are not fined at all. However, once the hinge loss recognizes that the true class score already exceeds the others by the threshold value or more, it will become equal to 0. Hence, the network will no longer change the weights in an attempt to increase the accuracy than it is already.

In contrast to the hinge loss, the categorical cross entropy targets a maximum likelihood estimate of the network's parameters. So, the proposed CNN computes the loss using a categorical cross-entropy function.

2.2.5 Optimizer

After the network computes the loss, it fixes the weights of neurons using some optimization algorithm. Those algorithms are called optimizers - they define how weights change based on the change of a loss function. Most modern optimizers are based on the gradient descent method such as an adaptive gradient (Adagrad), adaptive propagation (Adaprop), root mean square propagation (RMSprop), adaptive moment estimation (Adam), etc. The experimental results of this research show that for training of the proposed CNN, it is optimal to use Adam optimizer that has become the most popular optimizer nowadays [22].

Adam is an advanced version of a stochastic gradient descent algorithm that can be applied on non-convex loss functions. Let's assume that W_t is a vector of weights, M_t is a 1st-moment estimate, R_t is a 2nd-moment estimate, and $\underline{M_t}$ and $\underline{R_t}$ are 1st and 2nd-momentum bias corrections correspondingly at t^{th} iteration. The algorithm is carried out as follows:

1. Initialization: $M_t = 0, R_t = 0$.
2. For $t = 1, \ldots, T$:

$$M_t = \beta_1 M_{t-1} + (1 - \beta_1) \ \nabla L_t(W_{t-1}) \tag{2.10}$$

$$R_t = \beta_2 R_{t-1} + (1 - \beta_2) \ \nabla L_t(W_{t-1})^2 \tag{2.11}$$

$$\underline{M_t} = \frac{M_t}{1-(\beta_1)^t} \tag{2.12}$$

$$\underline{R_t} = \frac{R_t}{1-(\beta_2)^t} \tag{2.13}$$

$$\triangle_t = \alpha \frac{\underline{M_t}}{\sqrt{\underline{R_t}+\epsilon}} \tag{2.14}$$

$$W_t = W_{t-1} - \triangle_t \tag{2.15}$$

Here, ∇L_t is a change in a loss function, α is a learning rate, $\beta_1 \in [0, 1)$ is a 1st-momentum decay rate, $\beta_2 \in [0, 1)$ is a 2nd-momentum decay rate, and ϵ is numerical term that is typically set as 10^{-8}. These variables are the hyper-parameters of Adam optimizer. Properties of Adam:

1. Scale invariance:

$$L(W) \to cL(W) \Rightarrow \underline{M_t} \to c\underline{M_t} \ \wedge \ \underline{R_t} \to c\underline{R_t}$$

$$\Rightarrow \triangle_t \text{ does not change}$$

2. Bounded norm:

$$\|\Delta_t\|_\infty = \begin{cases} \frac{\alpha(1-\beta_1)}{\sqrt{1-\beta_2}}, & if\ 1-\beta_1 > \sqrt{1-\beta_2} \\ \alpha, & otherwise \end{cases} \tag{2.16}$$

Adam was designed as a combination of Adagrad and RMSprop and to have the advantages of both of the optimizers. So, the advantages of using the Adam optimizer are the following:

- Straightforward implementation;
- Low memory consumption;
- The optimizer works well with sparse gradients, which are feedback signals too weak to optimize the weights;
- The learning rates are scaled based on the moving average of the gradients for the weight. Hence, the optimizer works well on online problems.

2.2.6 Structure of Proposed CNN

To summarize, the CNN consists of 4 convolutional layers that extract distinctive features of the input, and 4 max-pooling layers that make the network invariant to slight translations of the input matrix and decrease the number of the features, and 2 dense layers to classify the output of the neural network. Max-pooling helps to decrease the computational cost and time that it takes to train and test the network. Each convolutional layer has the Tanh activation function. In Chapter 3.2, the results of applying other activation functions, like ReLU, Leaky ReLU, and ELU, are presented.

The extracted features are classified into the categories in the dense layers. Like all other hidden layers, the hidden dense layer's output is activated with the Tanh function. The output dense layer contains the network's predictions: each of 120 neurons contains a probability score that the input belongs to the corresponding class. Its activation function is Softmax. The implementation of the CNN in Python is presented in Appendix A.

The proposed network's architecture is presented in Figure 2.13.

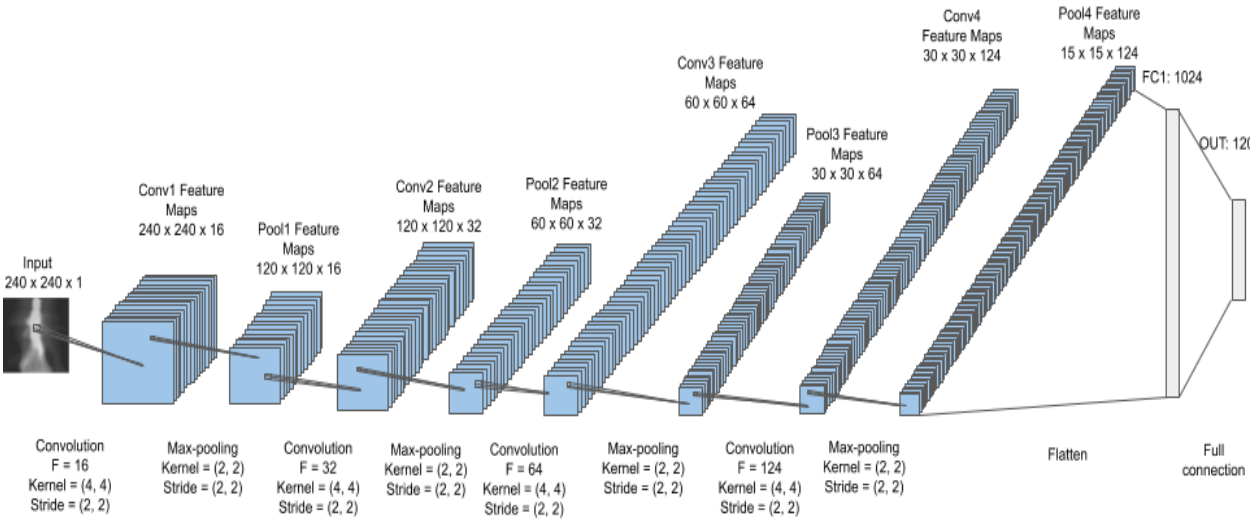

Figure 2.13: The structure of the proposed CNN

2.3 Layer-wise Relevance Propagation

Deep learning models have been treated as 'black boxes' despite their increased popularity in recent years. To make classification processes more understandable and transparent, several techniques, like layer-wise relevance propagation (LRP), have been developed. LRP decomposes a neural network's output $f(x)$ into a heat map that indicates each input data point's relevance to the final decision of the network [23]. So, the network's output is decomposed into a sum of relevances:

$$f(x) = \sum_{i=1}^{N} r_i \tag{2.17}$$

where N is the number of pixels of the input image, and r_i is a relevance score of the i^{th} pixel.

Technically, this technique executes a backward pass of the output probability score of a certain class through the network back to the first layer. So, it passes the relevances of upper layer neurons to the neurons of an adjacent lower layer. Thus, each pixel of the input layer obtains a relevance score proportional to its contribution to the final decision of the network [24].

Suppose that a neural network has the input vector $\underline{x} = [x_1, x_2, \dots, x_N]$, and each element of $\underline{x}$ is a neuron of the network's first layer, then $\underline{x}^{(1)} = \underline{x}$. The neurons of the next layer are computed as follows:

$$\underline{x}_j{}^{(l+1)} = g\left(\sum_i \; x_i^l w_{ij}^{(l,\; l+1)} + b_j^{(l+1)}\right), \tag{2.18}$$

where j is the index of a neuron at layer $l+1$; i indexes the neurons at lower layer l; $w_{ij}^{(l,\ l+1)}$ is a learned weight of the connection between a pair of two adjacent neurons; $b_j^{(l+1)}$ is a bias of the j^{th} neuron at level $l+1$; and $g(.)$ is an activation function that follows the layer l.

To reduce the LRP formulas, the following variable is introduced:

$$z_{ij} = x_i^l w_{ij}^{(l,\ l+1)} \tag{2.19}$$

The total relevance that equals to the network's probability score is passed to the lower adjacent layer, then the relevances computed at the lower layer are passed to the next lower layer, and so on. Thus, the formula of LRP is the following:

$$r_i^l = \sum_j \frac{z_{ij}}{\sum_{i'} z_{i'j}} r_j^{(l+1)}, \tag{2.20}$$

where r_i^l is a relevance score of i^{th} pixel of the l^{th} layer, and z_{ij} is a product of the i^{th} pixel value and j^{th} neuron's weight. Figure 2.14 demonstrates the LRP process.

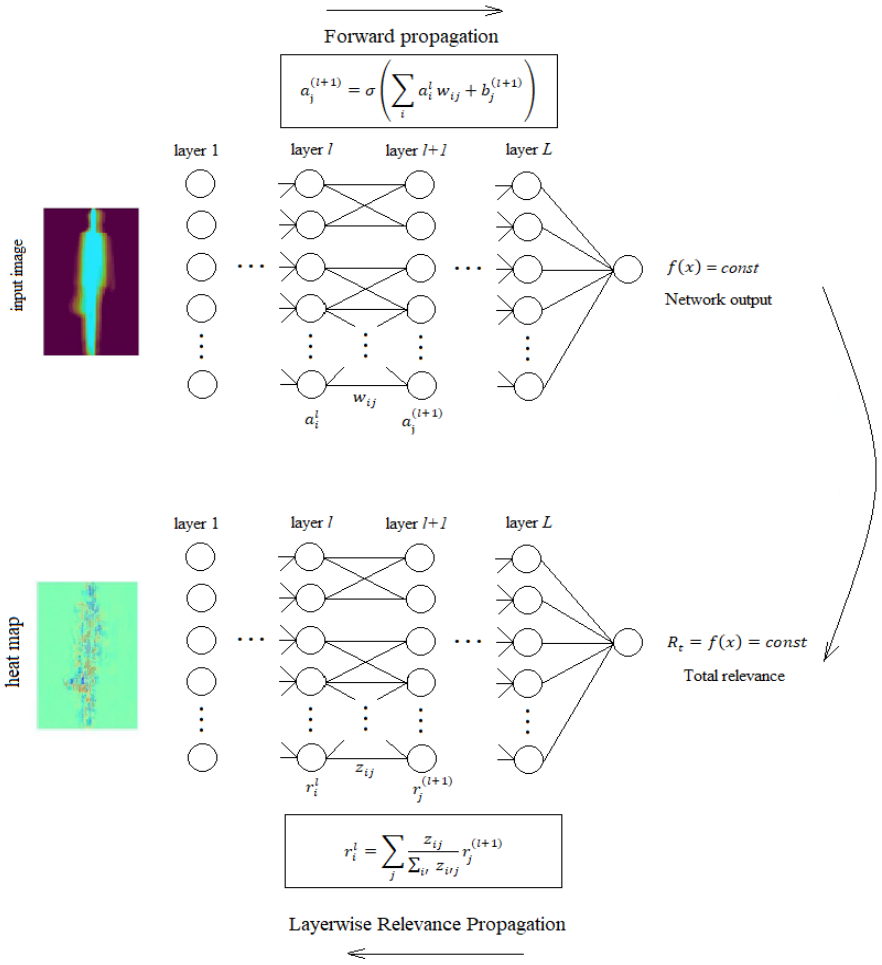

Figure 2.14: Heat map generation. Retrieved from [25]

Figure 2.15 shows the heat map of the input image. The red areas indicate the highest relevance scores whilst the blue ones indicate negative scores. As we can see, the network classifies the image based on its watermark. So, the heat map exposes an example of the classifier's incorrect response.

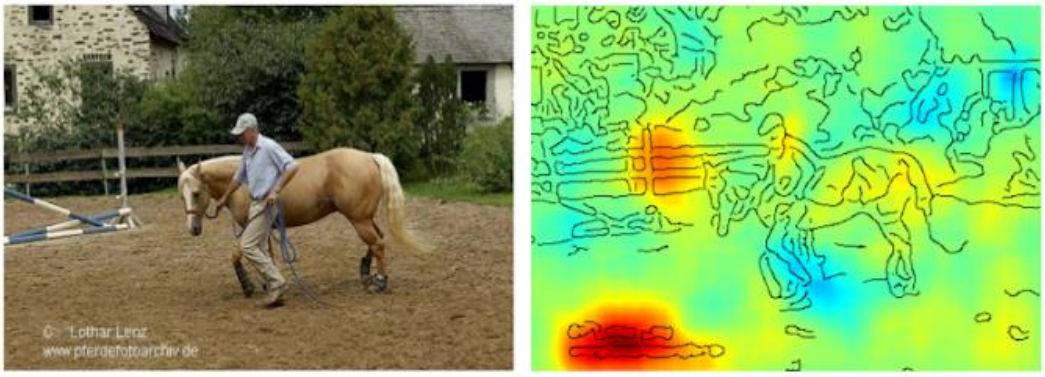

Figure 2.15: Input image and1 LRP's heat map [23]

The aim of LRP is to provide a visual explanation or description of any classifier's output in the domain of its input. It brings some intuitive clarity, but not any quantitative information.

2.4 Experimental Setup

An experimental setup programmed in Python and using Keras/Tensorflow was used to generate results provided in the next section. This setup consists of the pre-processed dataset of gait images, and the proposed CNN model.

The dataset consists of 120 classes. Each class contains 110 gait energy images randomly split up into training and testing images with ratio 4:1. The input images have a size of 240 x 240 x 1.

The neural network has been built in Python using Tensorflow and Keras libraries. It was trained and tested on GEIs extracted from the CASIA B dataset. Several different optimizers and activation functions were applied to find the most optimal ones. Thus, the applied activation functions of the hidden layers are ReLU, Leaky ReLU, ELU, and Tanh. The output layer's activation function used is Softmax.

Next, the following optimizers were tested:

- Adam: the learning rate is 0.0001;
- Adadelta: the initial learning rate is 1.0 and the decay factor is 0.95;

- Nesterov-accelerated adaptive moment estimation (Nadam) with a learning rate of 0.0005.

The applied loss function is a categorical cross-entropy. The CNN was trained with a batch size of 12 for 40 epochs. The training was carried out on NVIDIA GeForce RTX 2080 Ti GPU with 11GB RAM.

Chapter 3 - Presentation of Results

3.1 Optimizer Selection

Table 3.1 below shows the performances of the networks with different optimizers. The Leaky ReLU activation function is applied to all the hidden layers. Time for training and testing is measured when running a neural network on a workstation CPU 3.2 GHz with NVIDIA GeForce RTX 2080 Ti GPU with 11GB RAM.

Table 3.1: Comparison of optimizers

Optimizer	Accuracy (%)	Loss	Time for training (s)	Time for testing (s)
Adam	97.39	0.1488	563	0.058
Adadelta	97.27	0.2081	523	0.052
Nadam	96.68	0.2517	642	0.051

The graphs of the accuracy and loss change during the training process of the model with Adam, Adadelta, and Nadam are presented in Figures 3.1 - 3.6.

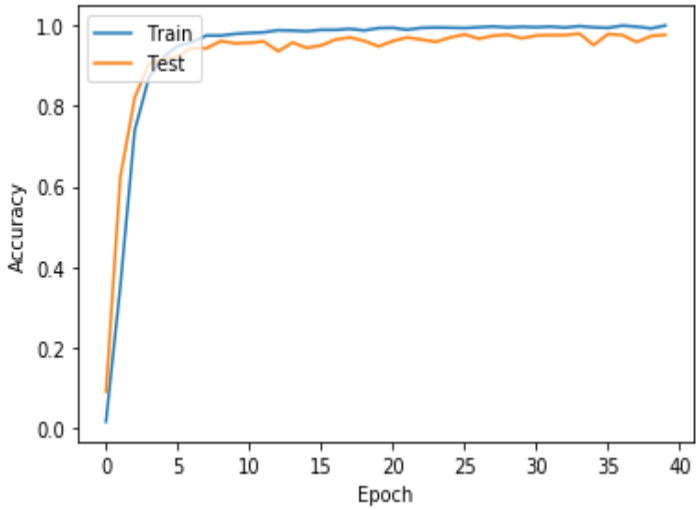

Figure 3.1: CNN model accuracy with Adam optimizer and Leaky ReLU

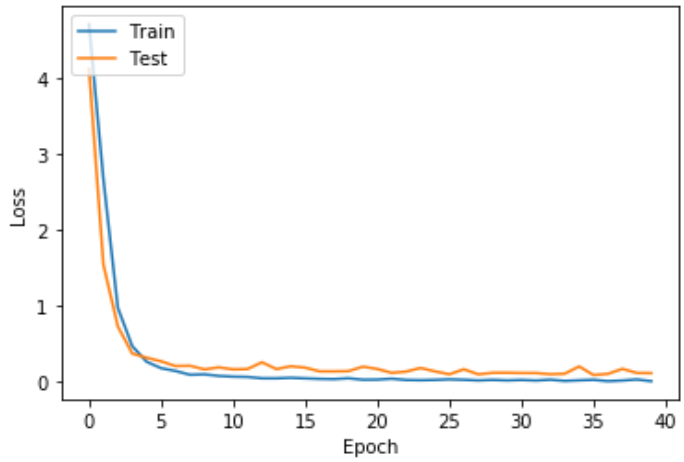

Figure 3.2: CNN model loss with Adam optimizer and Leaky ReLU

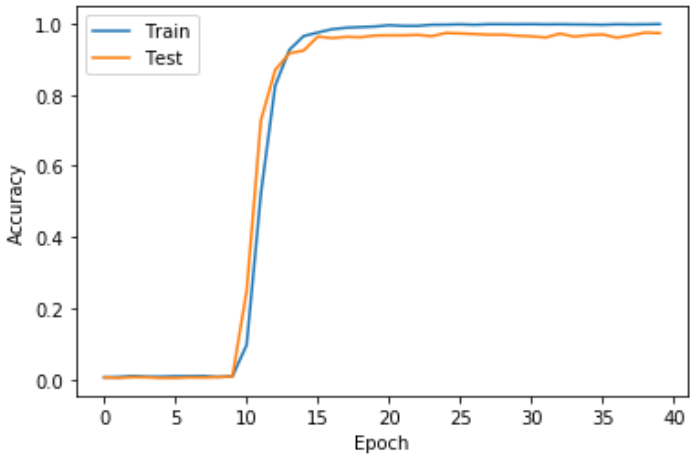

Figure 3.3: CNN model accuracy with Adadelta optimizer and Leaky ReLU

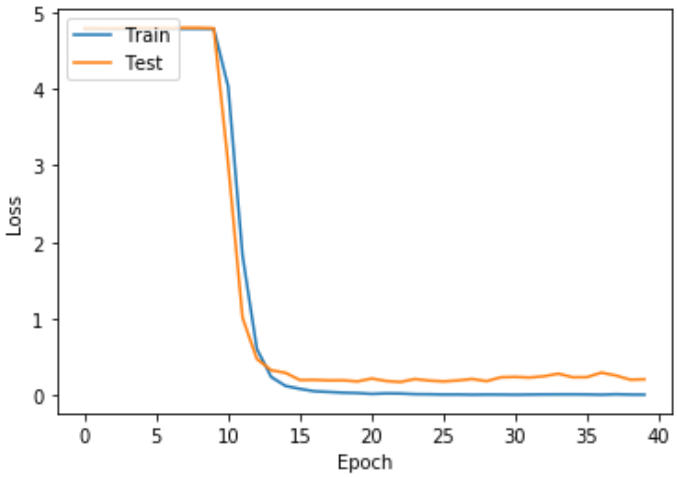

Figure 3.4: CNN model loss with Adadelta optimizer and Leaky ReLU

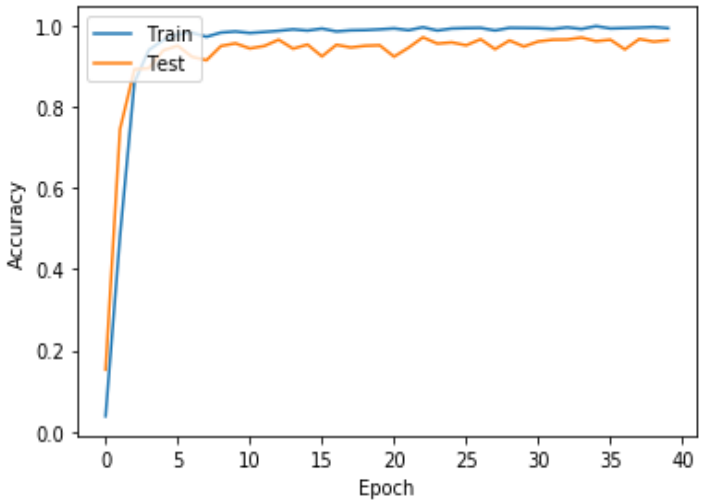

Figure 3.5: CNN model accuracy with Nadam optimizer and Leaky ReLU

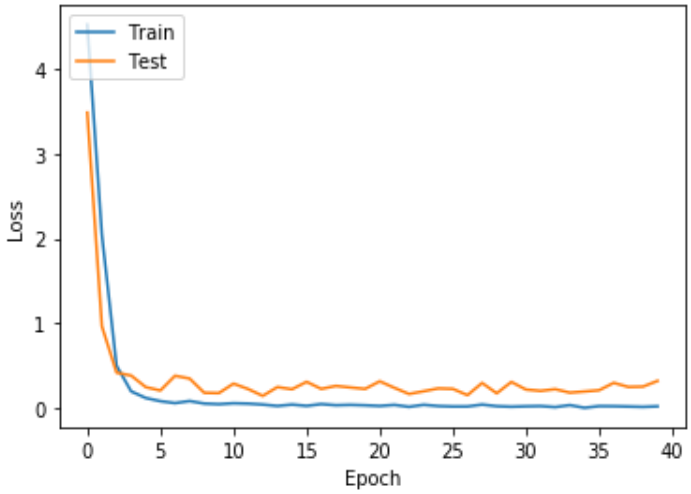

Figure 3.6: CNN model loss with Nadam optimizer and Leaky ReLU

The test loss shows the performance of the weights on the unseen data at a given iteration. As the weights change during training, they achieve different accuracies and losses when classifying the test data. However, only the train loss helps to update the weights, while the test loss helps the developer to see the model's actual performance and change its parameters based on the test accuracy and loss values.

The losses using the Adadelta and Nadam optimization increase for increasing number of iterations. The learning rate of Adadelta monotonically decreases [26], while Adam's learning rate goes up and down based on the loss of the network. It means that Adam adapts better to different inputs fed into the CNN. Nadam combines two optimization methods: Nesterov accelerated gradient and Adam [27]. It applies NAG instead of a gradient descent in Adam, because, according

to the experiments in [27], it makes a more accurate step by applying the momentum to the weights before computing the gradients. However, the results of this research show that the best performance was achieved with Adam optimizer. Its description is presented in Chapter 2.2.4.

3.2 Activation Function Selection

So, several activation functions were tried with the Adam optimizer. Again, these include ReLU, Leaky ReLU, ELU, and Tanh. The results of the model's performance are presented in Table 3.2.

Table 3.2: Comparison of activation functions

Activation function	Accuracy (%)	Loss	Time for training (s)	Time for testing (s)
ReLU	96.41	0.1167	524	0.065
Leaky ReLU	97.39	0.1488	563	0.058
ELU	96.26	0.1970	524	0.057
Tanh	98.58	0.0584	527	0.058

The graphs of the accuracy and loss change during the training process of the model with the ReLU, ELU, and Tanh activation functions are presented in Figures 3.7 – 3.12. The performance of the model with Leaky ReLU is presented in Figures 3.1 and 3.2.

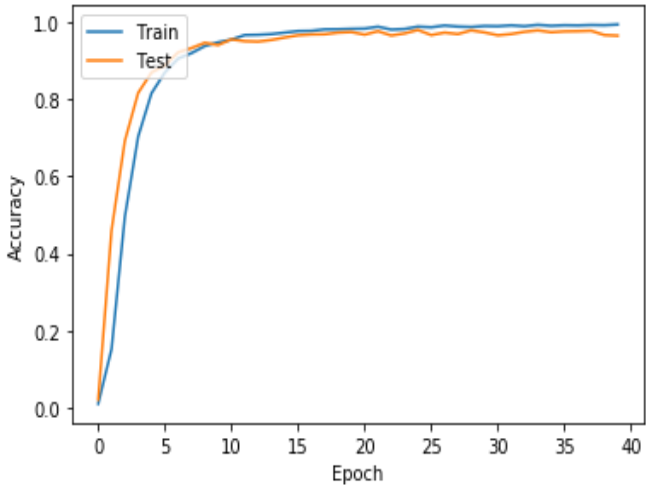

Figure 3.7: CNN model accuracy with Adam optimizer and ReLU

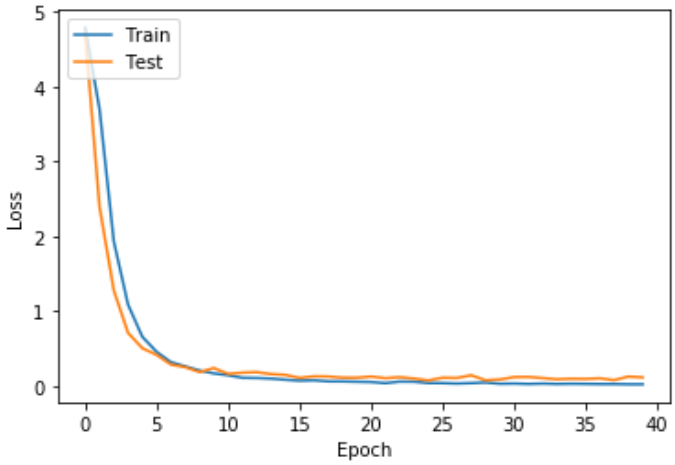

Figure 3.8: CNN model loss with Adam optimizer and ReLU

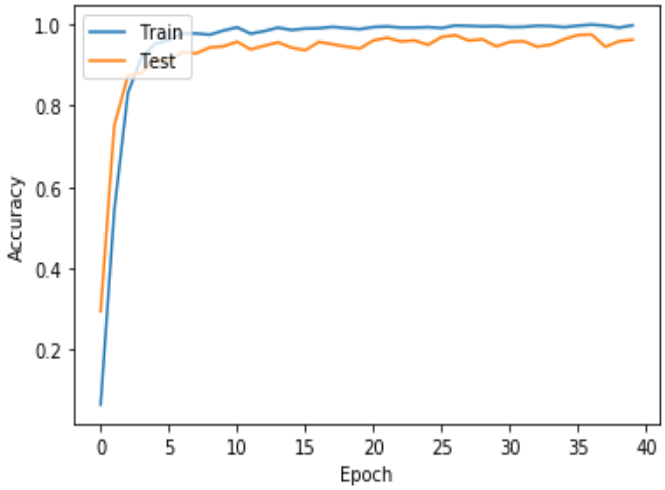

Figure 3.9: CNN model accuracy with Adam optimizer and ELU

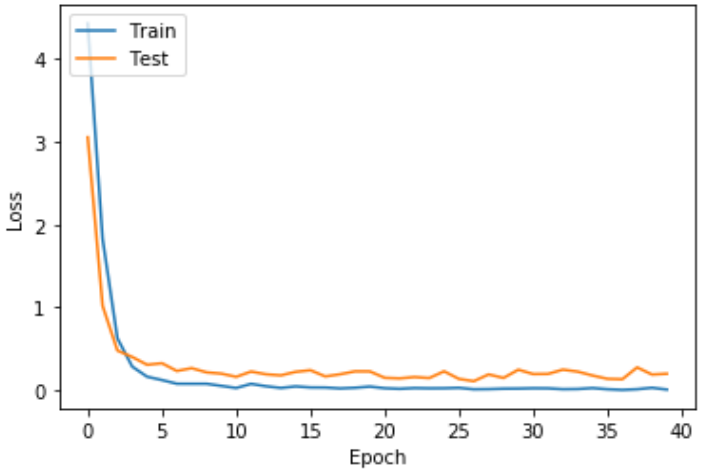

Figure 3.10: CNN model loss with Adam optimizer and ELU

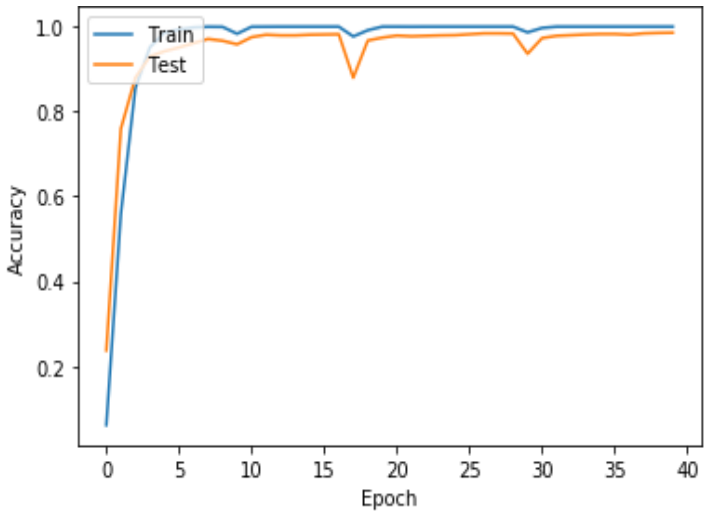

Figure 3.11: CNN model accuracy with Adam optimizer and Tanh

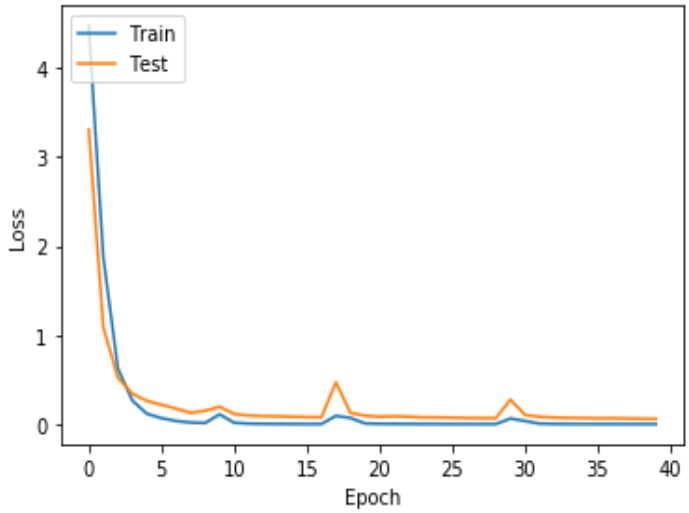

Figure 3.12: CNN model loss with Adam optimizer and Tanh

The model's performance with different activation functions is nearly identical. The main conclusion from the experimental results is that Tanh is more robust compared to ReLU and its kindred functions. Using the Tanh activation function, the model's test loss reduces to 0.0584. The second smallest loss is achieved using ReLU, and is almost twice bigger than that of Tanh – 0.1167.

As it was mentioned in Chapter 2.2, Tanh tends to be exposed to the vanishing gradient problem. Indeed, Figures 3.11 and 3.12 demonstrate that there were two points when the gradients were so small that the loss dropped significantly. But, as the model keeps training, it learns weights that diminish the effect of the vanishing gradient problem. It can be observed from the graphs: the peaks of the loss are diminishing.

Thus, the experimental results show that the best combination of an optimizer and an activation function for the given task is Adam and Tanh.

3.3 Results Using LRP

The tools used to implement the LRP are Tensorflow and Keras libraries. Appendix C contains a brief description of these two libraries. The code of LRP implementation is attached in Appendix B. The results of the LRP applied to the developed CNN is presented in Figure 3.13 where the right image is an original input gait energy image, and the left image is a heat map generated by the LRP.

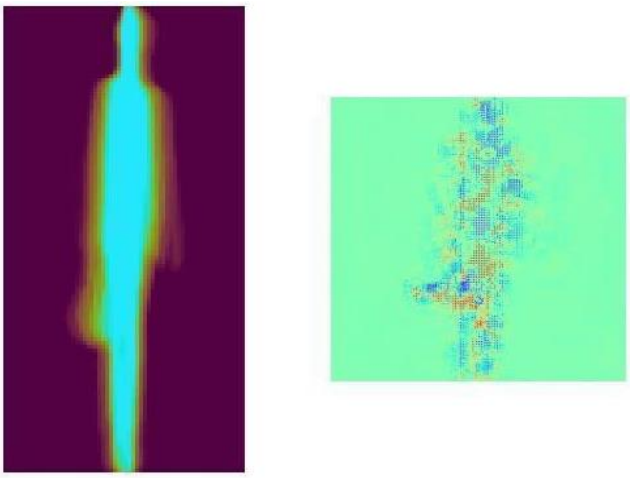

Figure 3.13: LRP applied to class '0' at angle 180º

The heat map shows that the classifier is focused on the edges of the silhouettes. The heat map was built using the 'Rainbow' color map. It means that the positive relevance values correspond to orange and red points, values around 0 correspond to green points, and negative values correspond to blue points. The red areas are the pixels that were positive and classify the input as class '0', and the blue areas correspond to negative values which means that the blue area fits some other class better than class '0'. The green (neutral) area does not play any role in classifying the image. The results of LRP are subjective, since there is no quantitative way to assess them. But they are useful for explaining a neural network's classification decisions.

Another example is shown in Figure 3.14.

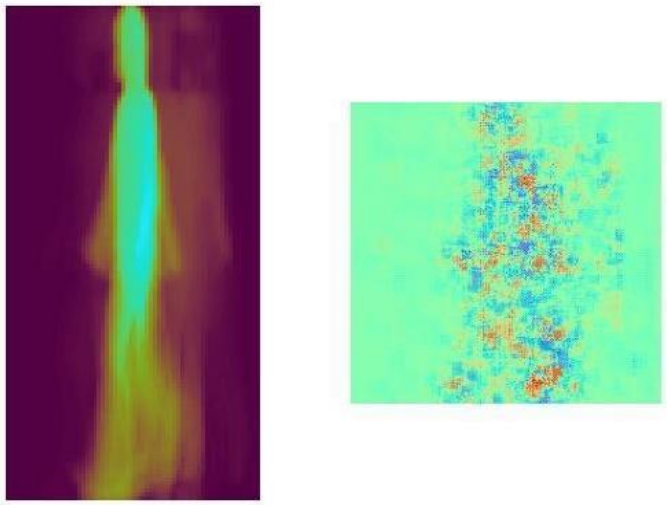

Figure 3.14: LRP applied to class '8' at angle 90º

Chapter 4 – Conclusion

This research focuses on building a CNN for gait recognition. The network's input are gait energy images. The proposed CNN is composed of 4 convolutional layers that extract distinctive features of the input, and 4 max-pooling layers that make the network invariant to slight translations of the input matrix and decrease the number of the features, and 2 dense layers for classification. The neural network extracts important gait features without human intervention. Several networks with the same architecture but different activation functions and optimizers were developed and compared with each other in terms of accuracy, loss, and computational time. The optimal activation functions are Tanh for hidden layers and Softmax for the output layer. The most suitable optimizer is Adam. The CNN developed in this project was trained on the CASIA B dataset and achieved an accuracy rate of 98.58%. This is the highest accuracy rate achieved for the entire CASIA B dataset regardless of the view angle and clothing covariate, although earlier researches' results are competitive, too. The state-of-the-art CNN for gait recognition achieved 92.95% accuracy trained and tested on the entire CASIA B dataset. Thus, the accuracy achieved by CNN proposed in this work is consistent with the best accuracy found in the literature.

Moreover, the LRP technique was applied to the network's output to verify the network's adequate classification. The LRP generates heat maps that show which areas of the input image affect the classifier's decisions the most. It has shown that the network focuses on the edge features like a distance between legs, height of the steps, position of hands in reference to other body parts, etc. However, the heat maps only give visual information that helps to slightly clarify the network's decision-making process. LRP information is useful at the data collection stage. Since it shows that the network's attention is focused on limbs of a person, it is necessary to have non-corrupted information about these areas. So, a silhouette should not be overlapped with other objects.

This research differs from the previous studies on gait recognition by its robustness to view variations and its high recognition rate. The papers studied during the literature review mostly propose separate networks for each angle of view, while others propose multi-view recognition networks with a relatively low accuracy rate compared to the results of this research.

The following issues should be addressed for practical implementation of the proposed CNN:

- For real-time applications, it is necessary to select and implement a fast background subtraction algorithm;
- In real-time applications, the camera should be mounted at approximately the same height as it was during data collection;
- The background on the video should be relatively static for effective silhouette extraction;
- The silhouettes of several people captured on a frame should not overlap each other.

The future work needs to seek solutions of the following problems:

- Robustness to shoes variations (heels and flats);
- Robustness to weight variation, i.e. a network should be able to recognize a person even after one's significant weight loss/gain;
- Gait recognition when several persons' silhouettes overlap each other on the video.

Also, to make the CNN recognize more persons than those collected in the dataset, there are two potential solutions to be explored:

- If the number of new labels is much smaller than the initial quantity, i.e. $n \ll 120$, then it would be optimal to use the CNN's pre-trained weights, and change the output Softmax layer from 120 neurons to $120 + n$. It is still necessary to retrain the network, but transfer learning make it faster and more efficient;
- If the number of new labels is significant, then the structure of the CNN itself needs to be partially changed to increase its capacity. To develop a network that grows its capacity as the new classes arrive, an incremental learning technique was proposed by T. Xiao et al [28]. The proposed algorithm groups classes according to their resemblances and divides them into levels. As a new batch of labels arrives, it clones the existing network to classify them. The crucial moment is that all models can be trained in parallel. The new models copy hallmarks from existing ones which allows to accelerate the learning process.

References

[1] A. K. Jain, A. Ross, and S. Prabhakar, ‘‘An introduction to biometric recognition,’’ IEEE Trans. Circuits Syst. Video Technol., vol. 14, no. 1, pp. 4–20, Jan. 2004

[2] J. P. Singh, S. Jain, S. Arora and U. P. Singh, "Vision-Based Gait Recognition: A Survey," in IEEE Access, vol. 6, pp. 70497-70527, 2018.

[3] J. Singh, S. Jain, S. Arora and U. Singh, "Vision-Based Gait Recognition: A Survey", *IEEE Access*, vol. 6, pp. 70497-70527, 2018.

[4] B. Kwolek, T. Krzeszowski, A. Michalczuk, and H. Josinski, “3D gait recognition using spatio-temporal motion descriptors,” in Proc. Asian Conf. Intell. Inf. Database Syst. (ACIIDS), 2014, pp. 595–604.

[5] Y. Wang, J. Sun, J. Li, and D. Zhao, “Gait recognition based on 3D skeleton joints captured by Kinect,” in Proc. IEEE Int. Conf. Image Process. (ICIP), Sep. 2016, pp. 3151–3155.

[6] D. Lopez-Fernandez, F. J. Madrid-Cuevas, A. Carmona-Poyato, R. Munoz-Salinas, and R. Medina Carnicer, “A new approach for multi-view gait recognition on unconstrained paths,” J. Vis. Commun. Image Represent., vol. 38, pp. 396–406, Jul. 2016.

[7] X. Chen, J. Xu, and J. Weng, “Multi-gait recognition using hypergraph partition,” Mach. Vis. Appl., vol. 28, nos. 1–2, pp. 117–127, Feb. 2017.

[8] A. Sokolova and A. Konushin, "Pose-based deep gait recognition", *IET Biometrics*, vol. 8, no. 2, pp. 134-143, 2019.

[9] K. Simonyan and A. Zisserman, “Two-stream convolutional networks for action recognition in videos,” NIPS, pp. 568–576, 2014.

[10] P. P. Min, S. Sayeed and T. S. Ong, "Gait Recognition Using Deep Convolutional Features," 2019 7th International Conference on Information and Communication Technology (ICoICT), Kuala Lumpur, Malaysia, 2019, pp. 1-5.

[11] Y. Zhang, Y. Huang, S. Yu and L. Wang, "Cross-View Gait Recognition by Discriminative Feature Learning," in IEEE Transactions on Image Processing, vol. 29, pp. 1001-1015, 2020.

[12] Y. He, J. Zhang, H. Shan and L. Wang, "Multi-Task GANs for View-Specific Feature Learning in Gait Recognition," in IEEE Transactions on Information Forensics and Security, vol. 14, no. 1, pp. 102-113, Jan. 2019.

[13] S. Bei, J. Deng, Z. Zhen and S. Shaojing, "Gender Recognition via Fused Silhouette Features Based on Visual Sensors," in IEEE Sensors Journal, vol. 19, no. 20, pp. 9496-9503, 15 Oct.15, 2019.

[14] X. Wang and S. Feng, "Multi-perspective gait recognition based on classifier fusion," in IET Image Processing, vol. 13, no. 11, pp. 1885-1891, 19 9 2019.

[15] X. Li, Y. Makihara, C. Xu, Y. Yagi and M. Ren, "Joint Intensity Transformer Network for Gait Recognition Robust Against Clothing and Carrying Status," in IEEE Transactions on Information Forensics and Security, vol. 14, no. 12, pp. 3102-3115, Dec. 2019.

[16] S. Li, W. Liu and H. Ma, "Attentive Spatial–Temporal Summary Networks for Feature Learning in Irregular Gait Recognition," in IEEE Transactions on Multimedia, vol. 21, no. 9, pp. 2361-2375, Sept. 2019.

[17] J.Hua and B.Bhanu, “Individual recognition using gait energy image,” IEEE Trans. on PAMI, 2006.

[18] C. Wang, J. Zhang, L. Wang, J. Pu and X. Yuan, "Human Identification Using Temporal Information Preserving Gait Template," in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 34, no. 11, pp. 2164-2176, Nov. 2012.

[19] D. A. Clevert, T. Unterthiner, S. Hochreiter, “Fast and accurate deep network learning by exponential linear units (elus),” in Proceedings of the International Conference on Learning Representations (ICLR), 2016.

[20] I. Goodfellow, Y. Bengio and A. Courville, Deep Learning (Adaptive Computation and Machine Learning series). Cambridge, Massachusetts: The MIT Press, 2016, p. 342.

[21] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever and R. Salakhutdinov, "Dropout: A Simple Way to Prevent Neural Networks from Overfitting", Journal of Machine Learning Research, vol. 15, no. 56, pp. 1929-1958, 2014.

[22] D. Kingma, J. Ba, "Adam: A Method for Stochastic Optimization", in International Conference on Learning Representations, 2014.

[23] G. Montavon, W. Samek and K. Müller, "Methods for interpreting and understanding deep neural networks", Digital Signal Processing, vol. 73, pp. 1-15, 2018.

[24] S. Bach, A. Binder, G. Montavon, F. Klauschen, K. Müller and W. Samek, "On Pixel-Wise Explanations for Non-Linear Classifier Decisions by Layer-Wise Relevance Propagation", PLOS ONE, vol. 10, no. 7, p. 130-140, 2015.

[25] A. Binder, S. Bach, G. Montavon, K. Müller and W. Samek, "Layer-Wise Relevance Propagation for Deep Neural Network Architectures", Lecture Notes in Electrical Engineering, pp. 913-922, 2016.

[26] M. D. Zeiler, “ADADELTA: an adaptive learning rate method”, Computer Science, 2012.

[27] T. Dozat, “Incorporating Nesterov Momentum into Adam”, ICLR Workshop, (1):2013–2016, 2016.

[28] T.Xiao, J. Zhang, K. Yang, Y. Peng и Z. Zhang, “Error-Driven Incremental Learning in Deep Convolutional Neural Network for Large-Scale Image Classification”, in ACM Conference on Multimedia, 2014.

Appendices

Appendix A

This code loads the gait dataset, builds the proposed CNN, and performs its training and testing.

```
import cv2
import numpy as np
import glob
import keras
import matplotlib.pyplot as plt
from keras.utils import to_categorical, np_utils
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Conv2D, Flatten, Dropout, MaxPooling2D
from sklearn.model_selection import train_test_split
from tensorflow.keras import optimizers

# 1 - Load the data
imageData = []
imageLabels = []
path = '/home/user/Documents/Aziza/CASIA B GEI/Train'
for class_folder in glob.glob(path + "/*"):
    for image_path in glob.glob(class_folder + '/*'):
        image = cv2.imread(image_path)
        image = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
        dim = (240, 240)
        image = cv2.resize(image, dim, interpolation = cv2.INTER_AREA)
        imageData.append(image)
        label = int(class_folder[-3:])
        imageLabels.append(label)

# 2 - Split into training & testing data
x_train, x_test, y_train, y_test = train_test_split(np.array(imageData) / 255.0,
                                                    np.array(imageLabels), train_size=0.9, random_state = 13)

# 3 - Transform the data into a format acceptable for the CNN
nb_classes = 120
X_train = np.array(x_train)
X_test = np.array(x_test)
y_train = np.array(y_train)
y_test = np.array(y_test)
Y_train = np_utils.to_categorical(y_train, nb_classes)
Y_test = np_utils.to_categorical(y_test, nb_classes)
X_train.shape = (9477, 240, 240, 1)
X_test.shape = (1054, 240, 240, 1)

# 4 - Build and compile the CNN model
model = Sequential()
model.add(Conv2D(16, kernel_size=4,input_shape=(240, 240, 1),strides=(1, 1),padding='same', activation='tanh'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(32, kernel_size=4, padding = 'same', activation='tanh'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(64, kernel_size=4, padding = 'same', activation='tanh'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Conv2D(124, kernel_size=4, padding = 'same', activation='tanh'))
model.add(MaxPooling2D(pool_size=(2, 2), strides=(2, 2)))
model.add(Flatten())
model.add(Dense(1024, activation='tanh'))
model.add(Dropout(rate = 0.3))
model.add(Dense(120, activation='softmax'))

adam = optimizers.Adam(lr=0.0001, beta_1=0.9, beta_2=0.999, amsgrad=False)

model.compile(optimizer=adam, loss = 'categorical_crossentropy', metrics = ['accuracy'])

# 5 - Train the model
history = model.fit(x=X_train, y=Y_train, batch_size=12, epochs=40, verbose=1, callbacks=None,
                    validation_split=0.2)

# 6 - Plot training & validation accuracy values
plt.plot(history.history['acc'])
plt.plot(history.history['val_acc'])
plt.title('Model accuracy')
plt.ylabel('Accuracy')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# 7 - Plot training & validation loss values
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
plt.legend(['Train', 'Test'], loc='upper left')
plt.show()

# 8 - Save the model
model_json = model.to_json()
with open("model_GR_tanh.json", "w") as json_file:
    json_file.write(model_json)
model.save_weights("model_GR_kernel_4_tanh.h5")
```

Figure A.1: Code from 'CNN.ipynb'

Appendix B

File 'lrp.py': In this program, a LayerwiseRelevancePropopagation class is created that applies LRP to every layer of the model. They can be applied to different kinds of layers: dense, convolutional, etc. It computes and passes relevances from the output to the input layer where it generates a heatmap.

```python
import sys
import cv2
import numpy as np
from tensorflow.keras.models import model_from_json
from tensorflow.keras import backend as K
from tensorflow.python.ops import gen_nn_ops
from utils import (get_model_params,get_gammas,get_heatmaps,load_images,predict_labels,visualize_heatmap)
import tensorflow as tf

images_dir, results_dir = '/home/user/Documents/Aziza/CASIA B GEI/Validation/060/', './results/'

class LayerwiseRelevancePropagation:
    def __init__(self, alpha=2, epsilon=1e-7):
        json_file = open('/home/user/Documents/Aziza/model_GR_tanh.json','r')
        model_json = json_file.read()
        json_file.close()
        self.model = model_from_json(model_json)
        self.model.load_weights('/home/user/Documents/Aziza/model_GR_tanh.h5')
        self.alpha = alpha
        self.beta = 1 - alpha
        self.epsilon = epsilon
        self.names, self.activations, self.weights = get_model_params(self.model)
        self.num_layers = len(self.names)
        self.relevance = self.compute_relevances()
        self.lrp_runner = K.function(inputs=[self.model.input, ],outputs=[self.relevance, ])
    def compute_relevances(self):
        r = self.model.output
        for i in range(self.num_layers-2, -1, -1):
            if 'dense' in self.names[i + 1]:
                r=self.backprop_fc(self.weights[i+1][0],self.weights[i+1][1], self.activations[i], r)
            elif 'flatten' in self.names[i + 1]:
                r = self.backprop_flatten(self.activations[i], r)
            elif 'pool' in self.names[i + 1]:
                r = self.backprop_max_pool2d(self.activations[i], r)
            elif 'conv' in self.names[i + 1]:
                r=self.backprop_conv2d(self.weights[i+1][0], self.weights[i+1][1], self.activations[i], r)
            else:
                pass
        return r
    def backprop_fc(self, w, b, a, r):
        w_p, w_n = K.maximum(w, 0.), K.minimum(w, 0.)
        b_p, b_n = K.maximum(b, 0.), K.minimum(b, 0.)
        z_p = K.dot(a, w_p) + b_p + self.epsilon, K.dot(a, w_n) + b_n - self.epsilon
        s_p, s_n = r / z_p, r / z_n
        c_p = K.dot(s_p, K.transpose(w_p)), K.dot(s_n, K.transpose(w_n))
        return a * (self.alpha * c_p + self.beta * c_n)
    def backprop_flatten(self, a, r):
        shape = a.get_shape().as_list()
        shape[0] = -1
        return K.reshape(r, shape)
    def backprop_max_pool2d(self, a, r, ksize=(1, 2, 2, 1), strides=(1, 2, 2, 1)):
        z = K.pool2d(a, pool_size=ksize[1:-1], strides=strides[1:-1], padding='valid', pool_mode='max')
        z_p, z_n = K.maximum(z, 0.) + self.epsilon, K.minimum(z, 0.) - self.epsilon
        s_p, s_n = r / z_p, r / z_n
        c_p=gen_nn_ops.max_pool_grad_v2(a, z_p, s_p, ksize, strides, padding='VALID')
        c_n=gen_nn_ops.max_pool_grad_v2(a, z_n, s_n, ksize, strides, padding='VALID')
        return a * (self.alpha * c_p + self.beta * c_n)
    def backprop_conv2d(self, w, b, a, r, strides=(1, 1, 1, 1)):
        w_p, w_n = K.maximum(w, 0.), K.minimum(w, 0.)
        b_p, b_n = K.maximum(b, 0.), K.minimum(b, 0.)
        z_p = K.conv2d(a,kernel=w_p,strides=strides[1:-1],padding='same')+b_p+self.epsilon
        z_n = K.conv2d(a, kernel=w_n, strides=strides[1:-1], padding='same') + b_n - self.epsilon
        s_p, s_n = r / z_p, r / z_n
        c_p = tf.compat.v1.nn.conv2d_backprop_input(K.shape(a), w_p, s_p, strides, padding='SAME')
        c_n = tf.compat.v1.nn.conv2d_backprop_input(K.shape(a),w_n,s_n,strides, padding='SAME')
        return a * (self.alpha * c_p + self.beta * c_n)
    def predict_labels(self, images):
        return predict_labels(self.model, images)
    def run_lrp(self, images):
        return self.lrp_runner([images, ])[0]
    def compute_heatmaps(self, images, g=0.2, cmap_type='rainbow', **kwargs):
        lrps = self.run_lrp(images)
        gammas = get_gammas(lrps, g=g, **kwargs)
        heatmaps = get_heatmaps(gammas, cmap_type=cmap_type, **kwargs)
        return heatmaps
if __name__ == '__main__':
    image_names = ['nm-04-072.png']
    image_names += sys.argv[1:]
    image_paths = [images_dir + name for name in image_names]
    image_names = [name.split('.')[0] for name in image_names]
    raw_images, processed_images = load_images(image_paths)
    test_image = processed_images[0]
    lrp = LayerwiseRelevancePropagation()
    labels = lrp.predict_labels(test_image)
    heatmaps = lrp.compute_heatmaps(test_image)
    for img, hmap, label, name in zip(raw_images, heatmaps, labels, image_names):
        visualize_heatmap(img, hmap, label, results_dir + name + '.jpg')
```

Figure B.1: Code from 'lrp.py'

File 'utils.py': this code contains auxiliary functions used in 'lrp.py'. E.g. it implements a function that pre-processes input data.

```python
import numpy as np
import matplotlib.pyplot as plt
import cv2
from matplotlib.cm import get_cmap
from keras.preprocessing.image import img_to_array, load_img

def get_model_params(model):
    names, activations, weights = [], [], []
    for layer in model.layers:
        name = layer.name if layer.name != 'predictions' else 'fc_out'
        names.append(name)
        activations.append(layer.output)
        weights.append(layer.get_weights())
return names, activations, weights
def load_images(image_paths, target_size=(240, 240)):
    raw_images, processed_images = [], []
    for path in image_paths:
        test_image = cv2.imread(path)
        raw_images.append(test_image)
        test_image = cv2.cvtColor(test_image, cv2.COLOR_BGR2GRAY)
        test_image = cv2.resize(test_image, (240, 240))
        test_image = np.array(test_image) / 255.0
        test_image.shape = (1, 240, 240, 1)
        processed_images.append(test_image)
    return raw_images, np.array(processed_images)
def predict_labels(model, images):
    labels = []
    preds = model.predict(images)
    predicted_class = np.where(preds[0] == preds[0].max())[0][0]
    labels.append(predicted_class)
    return labels
def gamma_correction(image, gamma=0.4, minamp=0, maxamp=None):
    eps = 1e-7
    c_image = np.zeros_like(image)
    image -= minamp
    if maxamp is None:
        maxamp = np.abs(image).max() + eps
        image /= maxamp
        pos_mask = (image > 0)
        neg_mask = (image < 0)
        c_image[pos_mask] = np.power(image[pos_mask], gamma)
        c_image[neg_mask] = -np.power(-image[neg_mask], gamma)
    return c_image * maxamp + minamp
def project_image(image, output_range=(0, 1), absmax=None, input_is_positive_only=False):
    if absmax is None:
        absmax = np.max(np.abs(image), axis=tuple(range(1, len(image.shape))))
    absmax = np.asarray(absmax)
    mask = (absmax != 0)
    if mask.sum() > 0:
        image[mask] /= absmax[mask]
    if not input_is_positive_only:
        image = (image + 1) / 2
    image = image.clip(0, 1)
    projection = output_range[0] + image * (output_range[1] - output_range[0])
    return projection
def reduce_channels(image, axis=-1, op='sum'):
    if op == 'sum':
        return image.sum(axis=axis)
    elif op == 'mean':
        return image.mean(axis=axis)
    elif op == 'absmax':
        pos_max, neg_max = image.max(axis=axis), -((-image).max(axis=axis))
        return np.select([pos_max >= neg_max, pos_max < neg_max], [pos_max, neg_max])
def heatmap(image, cmap_type='rainbow', reduce_op='sum', reduce_axis=-1, **kwargs):
    cmap = get_cmap(cmap_type)
    shape = list(image.shape)
    reduced_image = reduce_channels(image, axis=reduce_axis, op=reduce_op)
    projected_image = project_image(reduced_image, **kwargs)
    heatmap = cmap(projected_image.flatten())[:, :3]
    shape[reduce_axis] = 3
    return heatmap.reshape(shape)
def get_gammas(images, g=0.4, **kwargs):
    return [gamma_correction(img, gamma=g, **kwargs) for img in images]
def get_heatmaps(gammas, cmap_type='rainbow', **kwargs):
    return [heatmap(g, cmap_type=cmap_type, **kwargs) for g in gammas]
def visualize_heatmap(image, heatmap, label, savepath=None):
    fig = plt.figure()
    fig.suptitle(label)
    ax0 = fig.add_subplot(131)
    ax0.imshow(image)
    ax1 = fig.add_subplot(132)
    ax1.imshow(heatmap, interpolation='bilinear')
    if savepath is not None:
        fig.savefig(savepath)

if __name__ == '__main__':
    pass
```

Figure B.2: Code from 'utils.py'

Appendix C

TensorFlow is a powerful library developed by Google for numerous problems in machine learning. It provides both high-level and low-level API. It is best supported on Python, but some of its interfaces are also available on C, C++, Go, and Java. TensorFlow programs can be run on a CPU or GPU, on local computer or in the cloud, on iOS and Android devices.

Keras is a Python library for deep learning tasks. It is written in Python; hence, it is an intuitive and user-friendly tool. It works on top of low-level libraries like TensorFlow or Theano.