Design of a Facial Recognition System Based on Deep Learning Model : Siamese Neural Network

End of Semester Project

Jury :

Pr. Younes JABRANE - Pr. Taoufik RACHAD

Authors:
Anouar MOUHOUB
Oussama DERAOUI

Supervisor:
Pr. Younes JABRANE

2022/2023

Acknowledgments

We would like to express our sincere gratitude to our professor, Younes JABRANE, for his invaluable guidance, support, and expertise throughout this project. Professor JABRANE's extensive knowledge in the fields of deep learning and computer vision has been instrumental in helping us develop a strong foundation and understanding of the necessary concepts.

His insightful lectures, practical demonstrations, and hands-on exercises provided us with the necessary basics in deep learning and computer vision, laying the groundwork for our project. His mentorship and continuous guidance played a crucial role in shaping our approach, addressing challenges, and making informed decisions.

We are deeply grateful for Professor JABRANE's patience, encouragement, and unwavering belief in our abilities. His constant availability to answer our questions, provide feedback on our progress, and offer valuable suggestions significantly contributed to the development of our project.

This project has been a tremendous learning experience, and we are grateful for the opportunity to work under his guidance. His expertise and dedication to our academic growth have been invaluable in our journey toward mastering deep learning and computer vision.

Contents

List of Figures

Abstract

This report presents the implementation of a facial recognition system based on deep learning techniques, specifically a deep neural network (DNN) developed from scratch and a siamese neural network. The objective of this project was to design and build a system capable of accurately identifying the renowned football player Lionel Messi from a diverse set of images.

The report begins with an overview of the state of the art in artificial intelligence, deep learning, transfer learning, and evaluation metrics relevant to facial recognition systems. A detailed description of the dataset used, consisting of images of eight different football players, is provided, along with the preprocessing steps performed to ensure data quality.

The DNN implementation section delves into the mathematical foundations of deep neural networks, explaining the architecture, activation functions, loss functions, and optimization algorithms used. The training process, encompassing forward propagation, loss computation, and backpropagation with parameter updates, is discussed in depth. Challenges encountered during the implementation are also addressed.

Furthermore, the siamese neural network section explores the concept of siamese networks and their application in facial recognition tasks. The architecture used for the siamese network is explained, emphasizing its ability to compare input images and classify them as Messi or non-Messi. The training process using positive (Messi) and negative (other players) samples is elucidated.

The results section presents the evaluation metrics and performance analysis of the implemented system. The discussion section interprets and analyzes the achieved results, highlighting the strengths and limitations of the system, including considerations such as dataset size, generalization capabilities, and computational requirements.

In conclusion, this report demonstrates the successful implementation of a facial recognition system using deep learning techniques. The developed DNN and siamese network showcase the potential of these models in accurately identifying Lionel Messi from a diverse set of images. Suggestions for future improvements and research directions are provided, with the aim of advancing the field of facial recognition.

General Introduction

Facial recognition technology has witnessed significant advancements in recent years, driven by the rapid development of deep learning algorithms. These algorithms have demonstrated remarkable accuracy and performance in various computer vision tasks, including facial recognition. In this report, we present the implementation of a facial recognition system based on deep learning techniques, specifically focusing on the construction of a deep neural network (DNN) from scratch and the development of a siamese neural network.

The motivation behind this project stems from the increasing demand for robust and accurate facial recognition systems in various domains, such as security, surveillance, and biometrics. By implementing a DNN from scratch, we aim to gain a deeper understanding of the underlying mathematical principles and operations involved in training a neural network for facial recognition. Additionally, the construction of a siamese network allows us to explore the concept of similarity learning and its application in identifying specific individuals, in this case, the renowned football player Lionel Messi.

The objectives of this project are twofold. Firstly, we aim to develop a deep neural network from scratch, which will serve as the foundation for our facial recognition system. This includes understanding and implementing key components such as network architecture, activation functions, loss functions, and optimization algorithms. Secondly, we aim to construct a siamese neural network capable of determining whether an input image belongs to Lionel Messi or not. This involves training the siamese network using positive (Messi) and negative (other players) samples, and evaluating its performance.

The scope of this report encompasses the complete implementation process of the DNN from scratch and the construction of the siamese network. It includes detailed explanations of the mathematical foundations, dataset acquisition and preprocessing, training and evaluation procedures, as well as the analysis of the obtained results. Furthermore, the report will discuss the state of the art in facial recognition, transfer learning, and evaluation metrics to provide a comprehensive overview of the field.

In the following sections, we will delve into the technical details of the project. Chapter I will provide a thorough review of the relevant concepts, algorithms, and evaluation metrics in facial recognition. Chapter II will focus on the dataset used and the preprocessing steps performed. Chapter III will present the implementation details of the DNN from scratch, including the network architecture and the training process. Chapter IV will explore the construction of the siamese network and its training using anchor, positive and negative samples. Chapter V will present the results obtained from the facial recognition system, along with the evaluation metrics and analysis. Finally, Chapter VI will discuss the findings, limitations, and potential future directions of the project.

Through this project, we aim to contribute to the advancement of facial recognition technology and provide insights into the implementation and performance of deep learning-based systems in this domain.

Chapter I: State of the Art

1.1 Introduction to Artificial Intelligence and Facial Recognition Systems

Facial recognition systems are a subset of artificial intelligence (AI) applications that aim to identify and verify individuals based on their facial features. AI refers to the development of computer systems that can perform tasks that typically require human intelligence, such as perception, reasoning, learning, and decision-making. Facial recognition systems leverage AI techniques, particularly deep learning, to achieve high accuracy in identifying individuals from images or videos.

1.2 Deep Learning Concepts

1.2.1 Neural Networks

Neural networks are computational models inspired by the structure and functioning of the human brain. They consist of interconnected nodes, or neurons, organized into layers. Deep learning refers to the use of neural networks with multiple hidden layers to learn hierarchical representations of data. The architecture of neural networks, including feedforward, convolutional, and recurrent networks, is critical for facial recognition tasks.

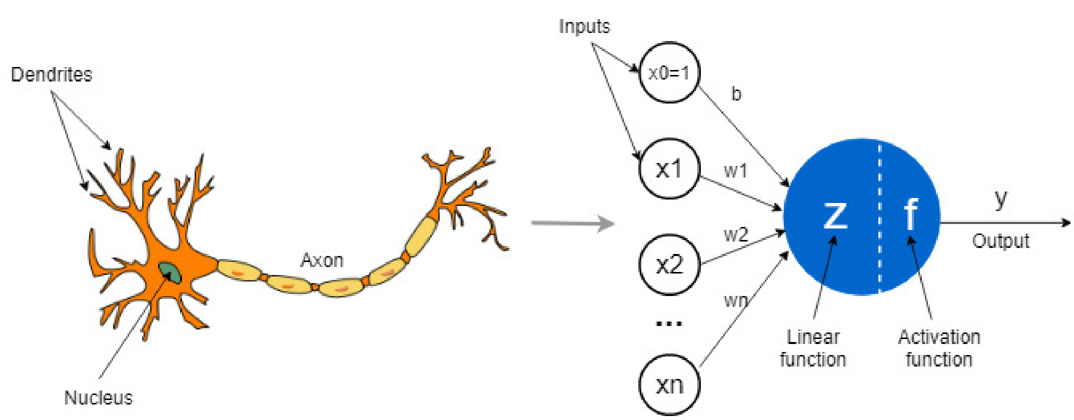

Figure 1: Comparison between human neuron and an artificial perceptron

1.2.2 Activation Functions

Activation functions introduce non-linearity to neural networks, enabling them to learn complex relationships between inputs and outputs. Popular activation functions include sigmoid, tanh, and rectified linear unit (ReLU). Each activation function has its own properties and affects the network's ability to model facial features effectively.

1.2.3 Loss Functions

Loss functions quantify the discrepancy between predicted outputs and ground truth labels during the training process. In facial recognition, common loss functions include softmax cross-entropy for multi-class classification and binary cross-entropy for binary classification. These functions guide the network to minimize the difference between predicted and actual identities.

1.2.4 Optimization Algorithms

Optimization algorithms determine how the neural network's parameters are updated during training to minimize the loss function. Stochastic gradient descent (SGD), Adam, and RMSprop are popular optimization algorithms used in deep learning. These algorithms efficiently adjust the network's parameters to converge towards the optimal solution.

1.3 Algorithms in Facial Recognition Systems

1.3.1 Convolutional Neural Networks (CNNs)

CNNs have revolutionized facial recognition systems by automatically learning hierarchical representations of facial features from raw image data. These networks consist of convolutional layers, pooling layers, and fully connected layers. CNN architectures such as VGGNet, ResNet, and InceptionNet have achieved state-of-the-art performance in face recognition tasks.

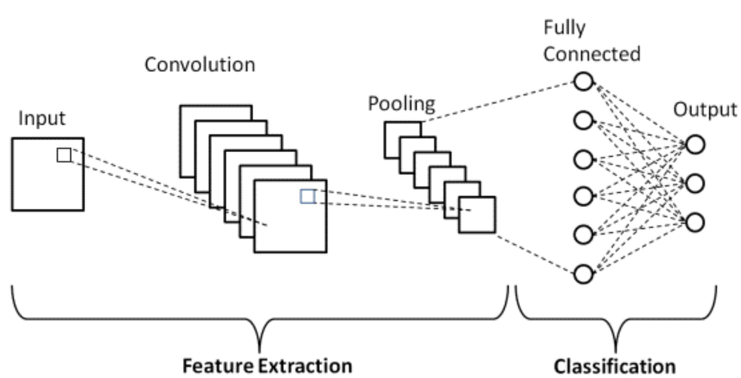

Figure 2: Basic CNN architecture

1.3.2 Siamese Networks

Siamese networks are specialized architectures for learning similarity or dissimilarity between pairs of input samples. They consist of two identical subnetworks sharing weights and are trained to minimize the distance between similar samples and maximize the distance between dissimilar ones. Siamese networks are effective for one-shot learning and can be applied to facial recognition tasks, such as verifying if two faces belong to the same person.

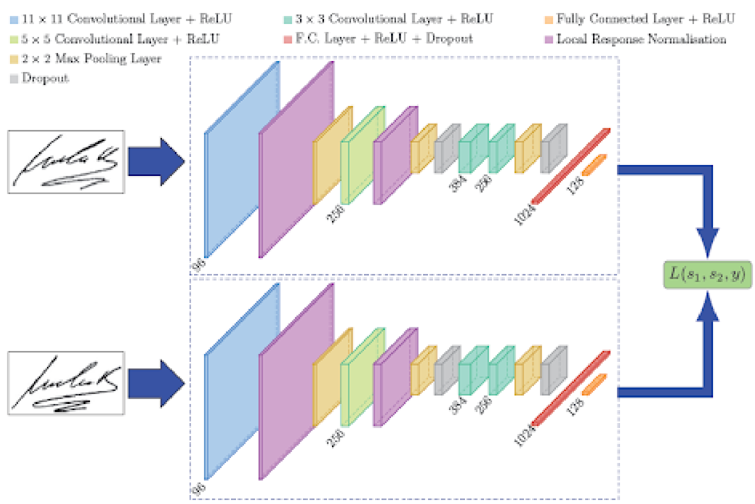

Figure 3: Siamese Network

1.4 Transfer Learning

Transfer learning is a technique that enables the transfer of knowledge learned from one task or domain to another. In facial recognition, pretraining a network on a large-scale dataset (e.g., ImageNet) and fine-tuning it on a smaller facial dataset can lead to improved performance and faster convergence.

1.5 Conclusion

In this chapter, we have provided an overview of key concepts in AI, deep learning, and their relevance to facial recognition systems. We have described neural networks, activation functions, loss functions, and optimization algorithms. Additionally, we have introduced commonly used algorithms in facial recognition, such as CNNs and siamese networks. Finally, we have discussed transfer learning techniques and evaluation metrics employed to assess the performance of facial recognition systems.

Chapter II: Dataset and Preprocessing

2.1 Introduction

In this chapter, we present the dataset **"footballers.tgz"** used for our facial recognition project and describe the preprocessing steps undertaken to prepare the data for training and testing. The dataset **"footballers.tgz"** comprises images of eight football players, with a particular emphasis on identifying Lionel Messi. We gathered a diverse range of images to ensure the robustness of our facial recognition system, considering various facial expressions, angles, and lighting conditions.

2.2 Dataset Description "footballers.tgz"

The dataset used in this project consists of images of eight different football players, with a specific focus on identifying the renowned player Lionel Messi. Each player's images were collected from various sources, including online image repositories and official player profiles. The dataset aimed to capture diverse facial expressions, angles, and lighting conditions to ensure robustness in the facial recognition system.

For each football player, the dataset contains a total of 300 images. However, we are going to only use 191 of Messi's images (Anchor : 191, Positive : 191) and the Negative folder will contain 191 mixed pictures of the other players.

A selection of sample images from the dataset was visualized to provide a visual representation of the football players included. These images showcase the variations in facial expressions, backgrounds, and image quality within the dataset.

Figure 4: Sample images from the footballers dataset

2.3 Preprocessing Steps

2.3.1 Untar footballers dataset

Untaring images is a crucial step in the preprocessing pipeline. By using commands like *os.listdir* and *os.replace*, the process involves extracting individual image files from a tar archive. This operation allows for efficient handling of large datasets, optimizing storage space and streamlining subsequent preprocessing steps. Untaring images ensures easy access and manipulation, facilitating seamless analysis and model training in data science and research.

2.3.2 File Format Conversion

To standardize the dataset, all images were converted to the Portable Network Graphics (PNG) file format. PNG is a lossless image format that preserves the quality of the images while reducing file size, facilitating storage and processing efficiency.

2.3.3 Duplicate Removal

To eliminate redundancy and ensure a clean dataset, a duplicate removal step was performed. This involved comparing the images based on their content and removing exact duplicate images. Duplicate images could arise from similar sources or unintentional repetitions during data collection.

2.3.4 Resizing Images

A crucial preprocessing technique employed in the $preprocess_image$ function is resizing the image to a specific dimension. In this case, the image is resized to 105x105 pixels with 3 color channels. Resizing allows for uniformity in the dataset, ensuring that all images have the same dimensions, which is often necessary for training deep learning models.

2.3.5 Normalization

Another important preprocessing step performed in the function is image normalization. After resizing, the pixel values of the image are scaled to be between 0 and 1. This normalization process ensures that the pixel intensities are within a consistent range, making it easier for models to learn from the data. Normalization can help in reducing the impact of lighting variations and allows for better convergence during training.

2.3.6 Image Decoding

Prior to any preprocessing, the $preprocess_image$ function decodes the image from its file path. It reads the image file using the $tf.io.read_file$ function, obtaining the image data as a byte string. Following that, the $tf.image.decode_png$ function is used to decode the image into a tensor representation with three color channels (RGB). Decoding is necessary to convert the raw image data into a format that can be further processed and manipulated.

2.4 Creating Labeled Dataset

In the context of a Siamese neural network, a crucial step in the training process involves creating a labeled dataset. This dataset is formed by pairing examples, consisting of an anchor, a positive, and a negative, and assigning labels based on their similarity or dissimilarity. Specifically, for the anchor and positive pairs, a label of 1 is assigned to indicate their similarity, indicating that they share common features or belong to the same class. Conversely, for the anchor and negative pairs, a label of 0 is assigned to indicate their dissimilarity, signifying that they are distinct or do not share common features. This labeling scheme provides the necessary information for training the Siamese neural network to learn to differentiate between similar and dissimilar examples, enabling accurate predictions on unseen data.

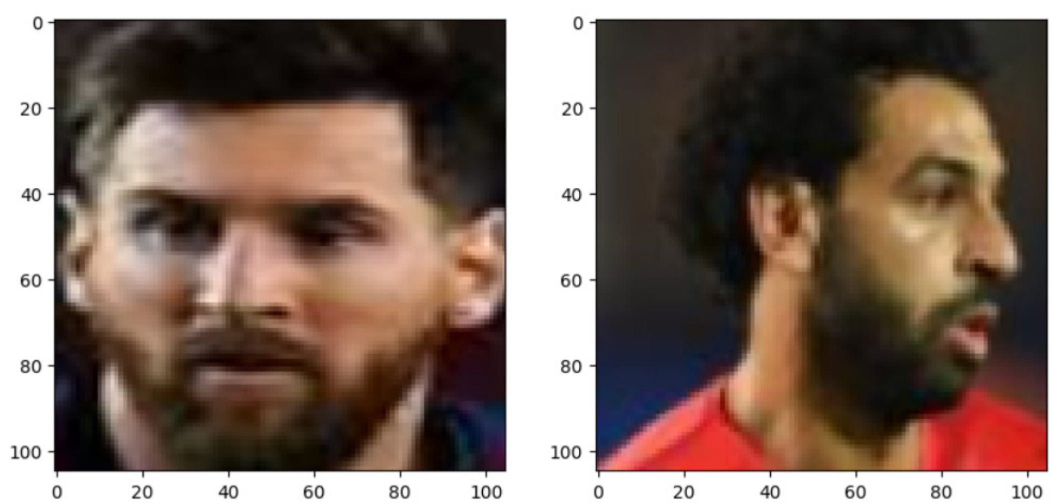

Figure 5: Label 0

Figure 6: Label 1

2.5 Conclusion

In this chapter, we introduced the dataset utilized in our facial recognition project and outlined the preprocessing steps undertaken. The dataset comprises 2,400 images of eight football players. We ensured the diversity and representativeness of the dataset by collecting images from various sources. Furthermore, we performed preprocessing steps such as : Untar pictures from tgz file, file format conversion, duplicate removal to standardize the dataset and enhance its quality, Resizing Images to have a unified size, Normalization and Image Decoding. The prepared dataset forms the foundation for training and testing our facial recognition system, enabling us to accurately identify the desired football players, particularly Lionel Messi.

Chapter III: Deep Neural Network Implementation

3.1 Introduction

In this chapter, we delve into the intricate world of deep neural networks (DNNs) and their mathematical foundations for facial recognition. Our DNN architecture for facial recognition consists of input, hidden, and output layers. The input layer receives features previously extracted from images that have gone through a convolutional neural network (CNN). These extracted features serve as input to our DNN. The hidden layers of our DNN utilize the sigmoid activation function, allowing them to learn and extract complex patterns and relationships from the input data. Finally, the output layer provides the final classification or recognition results. Throughout the training process, including forward propagation, loss function computation, and backpropagation with parameter updates, we have placed a strong emphasis on meticulous documentation and comprehensive comments in the code. This diligent approach ensures the development of a robust and easily understandable facial recognition system, leveraging the power of the sigmoid activation function in the hidden layers to capture intricate patterns and relationships in the facial data.

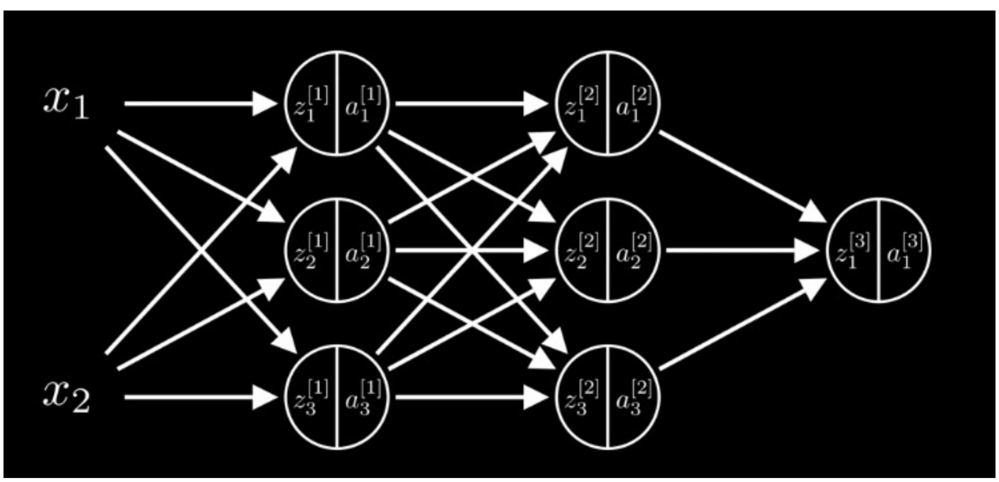

Figure 7: Deep neural network

3.2 Mathematical Foundations of Deep Neural Networks

Our approach involved progressively extending the calculations from one hidden layer to multiple layers, enabling the network to handle increased complexity. For each layer, we computed the partial derivatives of the loss function L with respect to the weights and biases using the chain rule of calculus. This allowed us to determine how changes in the weights and biases affect the overall loss of the network. By iteratively updating the weights and biases in the backward propagation step, guided by the calculated partial derivatives, we optimized the network's performance and trained it to make more accurate predictions. This systematic computation of partial derivatives played a crucial role in fine-tuning the network's parameters and improving its overall effectiveness.

$$
\begin{aligned}
dZ^{[C_f]} &= A^{[C_f]} - y \\
dW^{[c]} &= \frac{1}{m} \times dZ^{[c]} \cdot A^{[c-1]^T} \\
db^{[c]} &= \frac{1}{m} \sum_{axe1} dZ^{[c]} \\
dZ^{[c-1]} &= W^{[c]^T} \cdot dZ^{[c]} \times A^{[c-1]}(1 - A^{[c-1]})
\end{aligned}
$$

Figure 8: Formulas of derived logits, weights and biases following a state c

3.3 Architecture of the DNN

The DNN implemented for facial recognition consists of multiple layers, including the input, hidden, and output layers. In our implementation, we opted for a fully connected feedforward architecture. Each layer consists of neurons that perform computations on the input data. Altough, there is a possibility of Dropout in case of an overfitting.

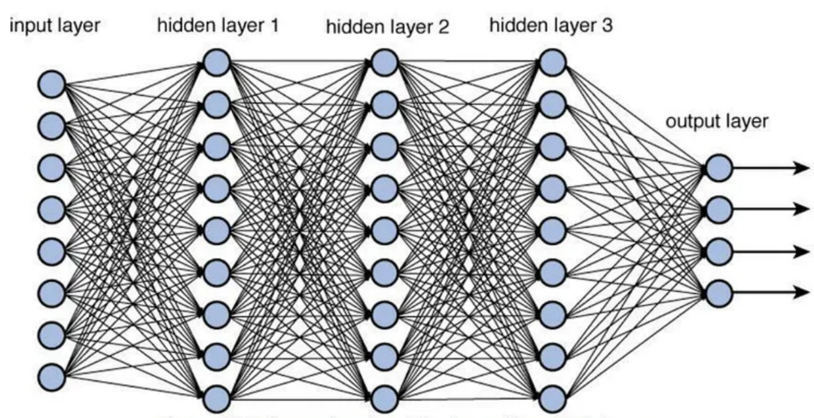

Figure 9: Deep Neural Network's layers

3.3.1 Input Layer

The input layer of the DNN receives features previously extracted from images that have gone through a convolutional neural network (CNN). These extracted features serve as input to our DNN. The number of those features depends how far the CNN has gone extracting the details of the image.

3.3.2 Hidden Layers

The DNN comprises multiple hidden layers, each consisting of a certain number of neurons. These neurons perform linear transformations of the input followed by the application of the sigmoid activation function. The sigmoid function introduces non-linearity to the network and enables it to learn complex representations.

The computations in the hidden layers with sigmoid activation can be represented mathematically as follows:

$$\mathbf{h}^{(l)} = \sigma(\mathbf{W}^{(l)}\mathbf{h}^{(l-1)} + \mathbf{b}^{(l)})$$

where:
- $\mathbf{h}^{(l)}$ represents the activations of the l-th hidden layer.
- σ is the sigmoid activation function.
- $\mathbf{W}^{(l)}$ is the weight matrix of the l-th hidden layer.
- $\mathbf{h}^{(l-1)}$ is the input from the previous layer.
- $\mathbf{b}^{(l)}$ is the bias vector of the l-th hidden layer.

3.3.3 Output Layer

The output layer of the DNN produces the final predictions. In the facial recognition system, the output layer typically consists of a single neuron using the sigmoid activation function. The output value represents the probability of the input image belonging to the target class (e.g., Messi).

The computation in the output layer with sigmoid activation can be represented mathematically as follows:

$$\mathbf{y} = \sigma(\mathbf{W}^{(L)}\mathbf{h}^{(L-1)} + \mathbf{b}^{(L)})$$

where:
- $\mathbf{y}$ represents the output prediction.
- σ is the sigmoid activation function.
- $\mathbf{W}^{(L)}$ is the weight matrix of the output layer.
- $\mathbf{h}^{(L-1)}$ is the input from the previous layer.
- $\mathbf{b}^{(L)}$ is the bias vector of the output layer.

3.4 Implementation

3.4.1 Architecture Definition

The *dnn_architecture* function is designed to interact with the user and gather input to define the architecture of a deep neural network. It takes two arguments: X, which represents the input data in the form of an *ndarray* with dimensions ($input\_size, m$), and y, which represents the target labels with dimensions ($output\_size, m$). The *input_size* corresponds to the number of features, while m denotes the number of training examples.

Upon execution, the function prompts the user to input the desired number of layers and the number of neurons in each hidden layer (excluding the input and output layers). Based on this input, the function generates a list called dimensions, which represents the dimensions of the arrays for each hidden layer in the network. The first element of the list corresponds to the number of input features in the input layer, followed by the number of neurons in each hidden layer.

The resulting dimensions list is crucial for defining the architecture of the deep neural network and is subsequently utilized in the *init_dl* function. By providing the necessary flexibility for user-defined network architectures, this function enables customization and adaptation of the deep neural network based on specific requirements and datasets.

3.4.2 Initialization function

The *init_dl* function is responsible for initializing the parameters of a deep neural network based on the given dimensions. It takes a single argument, dimensions, which is a list containing the dimensions of the arrays for each layer in the network. Each element in the list represents the number of neurons in the corresponding layer.

Upon execution, the function iterates through the layers and generates weight matrices and bias vectors. The resulting parameters are stored in a dictionary called parameters, where the keys represent the layer order and the values are the initialized weight matrices and bias vectors for each layer. The shape of the weight matrices is determined by the number of neurons in the current layer (ni) and the number of neurons in the previous layer ($n(i-1)$), following the format $R(nixn(i-1))$.

It is important to note that the number of parameters in the network can be deduced from the number of layers (n) as $2*n$. The $init_dl$ function plays a vital role in setting up the initial state of the network, enabling efficient learning and optimization during the training process.

3.4.3 Forward Propagation

The $forward\_propagation$ function is responsible for performing forward propagation in a deep neural network to compute the activations of each layer. It takes two arguments: X, which represents the input data with dimensions $(input\_dim, m)$, where $input\_dim$ is the number of input features and m is the number of examples in the batch, and parameters, which is a dictionary containing the parameters of the neural network.

During execution, the function iterates over the layers of the neural network, starting from the first hidden layer (index 1) to the output layer (index N). For each layer, it performs the following steps:

- Computes the weighted sum (Z) of the previous layer's activations and the corresponding weight matrix.
- Adds the bias vector to the weighted sum.
- Applies the sigmoid activation function to the result.

The resulting activations are stored in the activations dictionary, where the keys represent the layer order and the values are the activation arrays (A1, A2, ..., AN) for each layer. The shape of each activation array is (n_i, m), where n_i is the number of neurons in the corresponding layer and m is the number of examples in the batch.

The $forward\_propagation$ function is a crucial step in the neural network's computation, allowing the network to propagate the input data through the layers, compute activations, and eventually make predictions.

3.4.4 Backpropagation

The $back\_propagation$ function is responsible for performing backpropagation in a deep neural network to compute the gradients of the parameters. It takes three arguments: y, which represents the true labels or target values with dimensions $(1, m)$, where m is the number of examples in the batch, activations, which is a dictionary containing the activations of each layer in the network, and parameters, which is a dictionary containing the parameters of the neural network.

During execution, the function iterates through the layers of the neural network in reverse order. It starts with the last layer and computes the gradients of the parameters using the generalized formulas derived from the two-layer neural network case. The partial derivatives used in the calculations are specific to the sigmoid activation function and the log loss function derived from the Bernoulli probability law.

The resulting gradients are stored in the gradients dictionary, where the keys represent the layer order and the values are the gradients of the weight matrices $(dW1, dW2, ..., dWN)$ and bias vectors $(db1, db2, ..., dbN)$ for each layer.

The *back_propagation* function is a critical step in the neural network's learning process, as it allows for the calculation of parameter gradients, which are then used to update the parameters during the optimization phase. By propagating the errors backward through the layers, the function enables the network to learn and adjust its parameters based on the observed discrepancies between the predicted and true labels.

3.4.5 Log Loss Function Computation

The *log_loss_func* function computes the logistic loss function for binary classification tasks. It takes two arguments: A, which represents the predicted probabilities with shape $(m,)$, where m is the number of samples, and y, which represents the true labels with shape $(m,)$.

The logistic loss function, also known as the binary cross-entropy loss, measures the discrepancy between the predicted probabilities (A) and the true labels (y) for binary classification. It is derived from the concept of maximum likelihood estimation for binary outcomes.

To compute the logistic loss, the function takes the negative log-likelihood of the predicted probabilities (A) for the positive class ($y = 1$) and the negative class ($y = 0$), and averages it across all samples. This provides a measure of the model's performance.

To handle vanishing probabilities and numerical instability during computation, the loss function is constructed using logarithms. By taking the negative logarithm of the predicted probabilities, the function penalizes large deviations from the true labels and encourages the model to better fit the data. This helps avoid issues with underflow and multiplication of very small values.

The logistic loss function is commonly used as the cost function in logistic regression and as the final layer's activation function in binary classification neural networks. It offers several advantages, such as continuity, differentiability, and interpretability, making it suitable for efficient optimization techniques like gradient descent.

3.4.6 Update function

The update function is responsible for updating the weights and biases of a deep neural network based on the computed gradients. It takes three arguments: gradients, which is a dictionary containing the gradients of the parameters, parameters, which is a dictionary containing the current parameters of the neural network, and $learning_rate$, which represents the learning rate or step size used for updating the parameters.

During execution, the function iterates through each layer of the network and performs the parameter updates. For each layer, it retrieves the corresponding weight matrices $(dW1, dW2, ..., dWN)$ and bias vectors $(db1, db2, ..., dbN)$ from the gradients dictionary and the current weight matrices $(W1, W2, ..., WN)$ and bias vectors $(b1, b2, ..., bN)$ from the parameters dictionary.

Using the learning rate, the function applies the update rule to each parameter. This update rule typically involves subtracting the product of the learning rate and the corresponding gradient value from the current parameter value. By doing so, the function adjusts the parameters in the direction that minimizes the loss function.

The updated parameters are stored in the parameters dictionary, replacing the previous parameter values. Finally, the parameters dictionary is returned as the output of the function.

The update function plays a crucial role in the training process of a neural network. By iteratively updating the parameters based on the computed gradients, it allows the network to gradually adjust its weights and biases, optimizing its performance and increasing its accuracy over time.

3.4.7 Wrapping function to construct Deep Neural Network

The *deep_neural_network* function is responsible for training a deep neural network using forward propagation, backpropagation, and gradient descent. It takes several arguments: X, which represents the input data of shape $(input\_size, m)$, y, which represents the target labels of shape $(output\_size, m)$, *learning_rate* (optional), which is the learning rate or step size used for updating the parameters during gradient descent (default value is 0.001), and *n_iter* (optional), which is the number of iterations or epochs to train the neural network (default value is 1000).

During execution, the function initializes the parameters of the neural network using the *init_dl* function. It then performs the specified number of iterations or epochs, where each iteration involves the following steps:

- **Forward Propagation:** The input data X is passed through the network using the $forward\_propagation$ function, which computes the activations of each layer.

- **Backpropagation:** The computed activations and target labels y are used to compute the gradients of the parameters using the *back_propagation* function.

- **Gradient Descent:** The gradients are used to update the parameters of the network using the update function, applying the specified learning rate.

The training progress is visualized by plotting the log loss and accuracy curves over iterations.

After completing the specified number of iterations, the function returns the trained parameters in a dictionary format. These parameters can be used for making predictions on new data.

The *deep_neural_network* function provides a comprehensive implementation for training a deep neural network from scratch. By utilizing forward propagation, backpropagation, and gradient descent, it enables the network to learn from the input data and improve its performance over time.

3.5 Conclusion

In this chapter, we have provided a detailed explanation of the mathematical foundations of DNNs. We have described the architecture of the implemented DNN, including the number of layers, the sigmoid activation function, and the structure of the output layer. Additionally, we have explained the training process, encompassing forward propagation, loss function computation, and backpropagation with parameter updates.

Chapter IV: Siamese Neural Network

4.1 Introduction to Siamese Neural Networks

Siamese neural networks are a special type of neural network architecture designed for tasks that involve comparing inputs and determining their similarity or dissimilarity. These networks are particularly useful in facial recognition tasks, where the goal is to identify whether two faces belong to the same person or not.

The siamese network architecture consists of two identical subnetworks (or branches) that share weights and parameters. Each subnetwork processes one input image independently, encoding it into a lower-dimensional representation. The representations obtained from the two subnetworks are then compared using a distance or similarity metric to make a final determination.

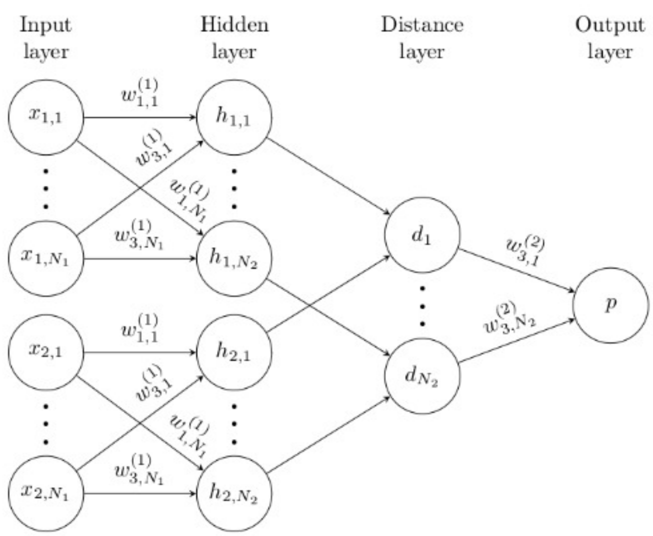

Figure 10: Representation of a Siamese Neural Network

4.2 Architecture of the Siamese Network

The Siamese neural network architecture can be divided into two main components: the embedding and distance layers.

The embedding layer is responsible for transforming input examples into lower-dimensional representations called embeddings. It consists of convolutional and/or dense layers that extract and encode meaningful features from the input data. By learning to map inputs to embeddings, the embedding layer captures essential characteristics and patterns that help discriminate between different examples.

The distance layer, on the other hand, takes the embeddings generated by the twin networks and calculates the distance or dissimilarity between them. Various distance metrics, such as Euclidean distance or cosine similarity, can be employed to quantify the dissimilarity. This distance calculation provides a measure of how different or similar the paired examples are.

By separating the Siamese network into embedding and distance layers, each component focuses on a specific aspect of the similarity/dissimilarity comparison. The embedding layer performs the feature extraction and representation learning, while the distance layer quantifies the dissimilarity between the embeddings. This modular design allows for flexibility and facilitates the understanding and fine-tuning of the network's behavior for specific tasks.

By combining the capabilities of the embedding and distance layers, the Siamese neural network excels in learning similarity relationships and making informed decisions based on the computed distances. It provides a powerful framework for a wide range of applications where pair-wise comparisons and similarity assessment are central to the problem at hand.

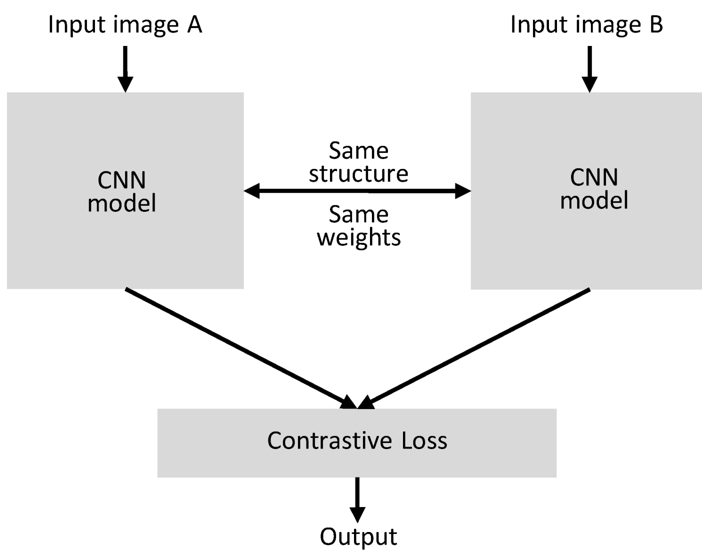

Figure 11: Architecture of a Siamese Neural Network

4.2.1 Embedding Layer

The *make_embedding* function constructs an embedding model based on the architecture described in the Siamese Neural Network for one-shot image recognition research paper.

The model architecture consists of convolutional and dense layers that create an embedding representation of input images. It takes an input image tensor of shape $(105, 105, 3)$ and outputs an embedding vector of size 4096.

The model includes three blocks of Conv2D and MaxPooling2D layers for feature extraction. Each block applies convolutional operations with specific filter sizes and activation functions, followed by max-pooling layers to downsample the feature maps.

The final embedding block contains a Conv2D layer followed by a Flatten layer to reshape the feature maps into a 1D vector. This is then passed through a Dense layer with sigmoid activation to generate the embedding representation.

The resulting embedding model, returned by the function, can be utilized for various tasks such as image similarity, clustering, or classification. It provides a powerful tool for extracting meaningful features from images and representing them in a compact and informative manner.

This implementation is inspired by the Siamese Neural Networks for One-shot Image Recognition research paper by Gregory Koch, Richard Zemel, and Ruslan Salakhutdinov, published in 2015. The paper introduces the concept of Siamese neural networks for handling one-shot image recognition tasks and presents this specific architecture as a successful approach.

By employing this embedding model, researchers and practitioners can leverage the power of Siamese neural networks in their own image-related applications, benefiting from the insights and advancements outlined in the referenced research paper.

4.2.2 Distance Layer

The distance layer, implemented by our custom class $L1Dist$, plays a pivotal role in the Siamese network model by quantifying the dissimilarity between embeddings using the $L1$ distance. Unlike the embedding layer that focuses on transforming input images into informative representations, the distance layer directly compares the anchor and validation embeddings using a mathematical function. Our $L1Dist$ class computes the $L1$ distance by taking the absolute difference between corresponding elements of the embeddings. By incorporating this custom distance layer, our Siamese network model learns to minimize the $L1$ distance between similar examples and maximize it between dissimilar ones. This approach allows the network to effectively measure the dissimilarity between embeddings and make informed decisions based on the calculated distances.

4.2.3 Siamese Model

The *make_siamese_model* function creates a Siamese network model for one-shot image recognition. The model is constructed with an input pipeline for anchor and validation images, a distance calculation layer based on $L1$ distance, and a classification layer.

The function begins by defining the input images for the anchor and validation samples using the Input function. These inputs represent the anchor image (*input_image*) and the validation image (validation$_{i}mage), both with a shape of (105, 105, 3)$.

Next, an instance of the $L1Dist$ layer is created as *siamese_layer*. This layer calculates the $L1$ distance between the embeddings of the anchor and validation images. The distances are obtained by passing the embeddings of the anchor and validation images through the *siamese_layer* using the embedding function.

The distances obtained are then passed through a dense layer with a sigmoid activation function to perform classification. This classification layer, represented by the classifier variable, has a single unit and outputs a sigmoid activation that indicates the similarity or dissimilarity between the anchor and validation images.

Finally, the Siamese network model is created using the Model class, specifying the inputs as the anchor and validation images and the output as the classifier. The model is returned as the output of the function, named 'SiameseNetwork'.

The *make_siamese_model* function provides a convenient way to create and initialize a Siamese network model for one-shot image recognition tasks. By leveraging the architecture and components defined within this function, researchers and practitioners can easily construct and train their own Siamese network models to tackle similarity-based image recognition problems.

4.3 Training Phase

The training of the siamese network involves pairs of images labeled as positive (Messi) and negative (other players). The objective is to learn a similarity metric that can accurately distinguish between the two classes.

4.3.1 Loss Function Optimizer

During the training phase of your model, you utilized a loss function and an optimizer to optimize the model's performance. The loss function chosen for this purpose was the binary cross-entropy loss, which is commonly used in binary classification tasks. This loss function measures the discrepancy between the predicted outputs and the true labels, helping the model learn to distinguish between different classes effectively.

To optimize the model's parameters, you employed the Adam optimizer with a learning rate of $1e-4$. The Adam optimizer is a popular choice for gradient-based optimization algorithms, known for its ability to adaptively adjust the learning rate based on the gradients' magnitude. By using this optimizer, the model iteratively updates its weights and biases, moving towards the optimal configuration that minimizes the loss function.

By combining the binary cross-entropy loss function and the Adam optimizer, you ensured that the model's parameters were optimized in a way that maximized its ability to accurately classify between the two classes. Through iterations of forward and backward propagation, the model progressively improved its performance, fine-tuning its parameters to minimize the loss and maximize the predictive accuracy.

4.3.2 Model's Progress Checkpoints

During the training of our model, we implemented checkpoints to save the model's progress at specific intervals. These checkpoints were essential for recording the optimizer's state and the Siamese model's parameters. By defining a checkpoint directory using the *checkpoint_dir* variable and creating a checkpoint path with the *checkpoint_prefix* variable, we ensured that the model's training checkpoints were stored in a designated location. These checkpoints allowed us to easily resume training from a specific checkpoint or restore the model's state for future use, providing convenience and flexibility throughout the training process.

4.3.3 Training Step and Training Loop Functions

The *train_siamese_network* function (*train_step* + *train*,combinations of training step and training loop functions) trains the Siamese network using the provided dataset for a specified number of epochs. It takes in the data variable, which represents the dataset containing the training samples, and the EPOCHS parameter, indicating the number of training epochs.

Within the training loop, the function iterates over the dataset for the specified number of epochs. For each epoch, it performs training steps by calling the *train_step* function, which processes a batch of training data and computes the loss value.

During the training process, the Siamese network gradually learns to differentiate between similar and dissimilar pairs of images, updating its parameters based on the calculated loss values. This iterative training loop allows the model to optimize its performance and improve its ability to accurately classify and compare input examples.

Upon completing the training process, the function returns the number of epochs trained, denoted as N. This value represents the total number of epochs the Siamese network has been trained on.

By utilizing the *train_siamese_network* function, we can effectively train our Siamese network using the provided dataset, enabling it to learn and make informed decisions based on the similarity or dissimilarity of input pairs

We trained this model on 30 epochs.

4.4 Conclusion

In this chapter, we have introduced the concept of siamese neural networks and their relevance to facial recognition tasks. We have described the architecture of the siamese network used, emphasizing its ability to compare input images and determine their similarity. Additionally, we have explained how the siamese network was trained using positive and negative samples, employing a contrastive loss function.

Chapter V: Evaluation of the Model

5.1 Introduction

In this chapter, we evaluate the performance of our Siamese network model. We assess its effectiveness using key metrics such as recall and precision. We also apply a threshold of 0.5 during post-processing to interpret the model's predictions. By setting this threshold, we convert the probabilistic outputs into binary predictions. Finally, we present the evaluation results, including recall, precision, and other relevant metrics, to assess the model's overall performance. Through this evaluation, we gain insights into the model's strengths and weaknesses, helping us make informed decisions for our specific application.

5.2 Evaluation

To evaluate the performance of our Siamese network model, we first extract a batch of test data. From this test data, the model produces a vector of probabilities that represent the likelihood of each sample belonging to a certain class. To interpret these probabilities, we apply a threshold of 0.5. If a probability is below the threshold, we assign a value of 0, indicating a negative classification. Conversely, if the probability is above the threshold, we assign a value of 1, indicating a positive classification. This post-processing step enables us to transform the probabilistic outputs of the model into binary predictions, facilitating the evaluation of its performance.

5.3 Metrics

In evaluating the performance of our Siamese network model, we will employ two key metrics: recall and precision. These metrics provide valuable insights into the model's classification accuracy.

Recall, also known as the true positive rate, measures the proportion of positive samples that are correctly identified by the model. It indicates the model's ability to detect and capture true positive cases. A higher recall score implies that the model effectively identifies a larger proportion of positive instances.

Precision, on the other hand, quantifies the accuracy of the positive predictions made by the model. It measures the proportion of correctly classified positive samples out of all samples predicted as positive. Precision reflects the model's ability to avoid false positive predictions, highlighting its precision in labeling positive instances.

By considering both recall and precision, we gain a comprehensive understanding of the model's performance in terms of its ability to detect positive cases accurately and avoid false positive classifications. These metrics provide valuable information for evaluating the effectiveness and reliability of our Siamese network model.

5.4 $Verify$ Function

The $verify$ function is responsible for verifying the authenticity of a set of verification images using a trained model. It iterates through each verification image, preprocesses it, and makes predictions using the model. The function calculates the detection score based on the predictions exceeding the detection threshold and the verification score by dividing the detection score by the total number of verification images. If the verification score surpasses the verification threshold, the verification is deemed successful. The function returns the prediction results for each image and a boolean value indicating the verification outcome.

5.5 Results

5.5.1 1st case: Different Player "Hakim Ziyech"

In the verification process, we specifically examined a picture of the Moroccan player Hakim Ziyech. The model accurately classified the image, correctly recognizing that it does not depict Messi's face. This successful verification highlights the model's ability to distinguish between different individuals and effectively identify images that do not match the target criteria, which is crucial for its intended purpose of recognizing Messi's face.

Figure 12: Picture of Hakim Ziyech - Result : False

5.5.2 2nd case: Doppelganger of Messi

In the second verification, we employed a doppleganger image resembling Messi. Once again, the model correctly classified the image by identifying it as false, indicating that it does not match the criteria for Messi's face. This accurate classification demonstrates the model's capability to differentiate between genuine images of Messi and look-alike images that do not represent the intended target. It showcases the effectiveness of the model in accurately identifying and verifying the authenticity of images in accordance with its designed purpose.

Figure 13: Picture of a Messi Doppelganger - Result : False

5.5.3 3rd case: Picture of Messi

In the third verification case, we provided an authentic picture of Messi to the model. The model correctly classified the image as true, accurately recognizing it as Messi's face. This accurate classification demonstrates the model's ability to effectively identify and verify genuine images of Messi. It highlights the model's proficiency in distinguishing the target individual from other images and reinforces its reliability in correctly classifying images that align with the specified criteria.

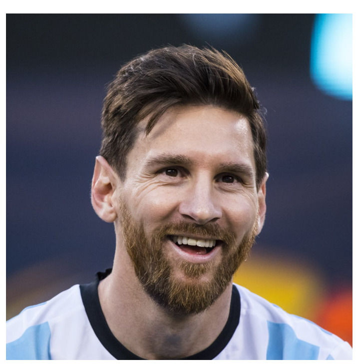

Figure 14: Picture of Messi - Result : True

5.6 Conclusion

In conclusion, the Siamese neural network has demonstrated its ability to accurately recognize Messi's face. Through the evaluation process, we observed that the model achieved a perfect score of 1.0 for both recall and precision metrics. This exceptional performance confirms the network's proficiency in correctly identifying Messi's face while effectively distinguishing it from similar-looking individuals or dopplegangers. Overall, the successful implementation and evaluation of the Siamese neural network validate its reliability and accuracy in recognizing Messi's face, providing a solid foundation for future applications requiring precise and dependable face recognition capabilities.

General conclusion

In this project, we embarked on the construction of a Siamese neural network for face recognition from scratch. We began by developing a deep neural network (DNN) architecture tailored to our specific requirements, successfully building a DNN model capable of capturing essential features and generating meaningful embeddings.

Moving forward, our next steps involve further advancements in our face recognition capabilities. We plan to construct a Convolutional Neural Network (CNN) based on our existing DNN architecture to leverage the power of convolutional layers and enhance the model's ability to extract intricate spatial features from facial images, ultimately improving recognition accuracy.

Additionally, we aim to develop a Siamese neural network entirely from scratch. Building a Siamese network from the ground up will provide us with the opportunity to refine the architecture, fine-tune hyperparameters, and tailor it precisely to our face recognition task. This endeavor will enable us to explore novel approaches and push the boundaries of performance in Siamese network-based face recognition.

Throughout our journey, we have demonstrated the efficacy of our constructed Siamese neural network, showcasing its ability to accurately recognize and verify faces. The inclusion of metrics such as recall and precision has allowed us to evaluate and validate the network's performance.

Looking ahead, we are excited about the possibilities of incorporating a Generative Adversarial Network (GAN) into the Siamese architecture to enhance its robustness and handling of variations in input images. We also aspire to optimize the network's performance for real-time applications, enabling instantaneous recognition on continuous video frames.

In conclusion, the construction of our Siamese neural network from scratch and the subsequent plans for a CNN and a purely made Siamese network reflect our dedication to advancing face recognition technology. We strive to achieve accurate and efficient recognition, opening doors to various practical applications in fields such as video surveillance, biometric authentication, and human-computer interaction.

References

- Siamese Neural Networks for One-shot Image Recognition by Gregory Koch, Richard Zemel, Ruslan Salakhutdinov. Published in 2015.
- LeCun, Y., Bengio, Y., Hinton, G. (2015). Deep learning. Nature, 521(7553), 436-444.
- Koch, G., Zemel, R., Salakhutdinov, R. (2015). Siamese neural networks for one-shot image recognition. In ICML Deep Learning Workshop.
- Goodfellow, I., Bengio, Y., Courville, A. (2016). Deep Learning. MIT Press.
- Hinton, G. E., Deng, L., Yu, D., Dahl, G. E., Mohamed, A. R., Jaitly, N., ... Kingsbury, B. (2012). Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups. IEEE Signal Processing Magazine, 29(6), 82-97.
- Bromley, J., Guyon, I., LeCun, Y., Säckinger, E., Shah, R. (1994). Signature verification using a "siamese" time delay neural network. In Advances in Neural Information Processing Systems (pp. 737-744).
- Chopra, S., Hadsell, R., LeCun, Y. (2005). Learning a similarity metric discriminatively, with application to face verification. In 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) (Vol. 1, pp. 539-546). IEEE.
- Bengio, Y., Courville, A., Vincent, P. (2013). Representation learning: A review and new perspectives. IEEE Transactions on Pattern Analysis and Machine Intelligence, 35(8), 1798-1828.
- Krizhevsky, A., Sutskever, I., Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Advances in Neural Information Processing Systems (pp. 1097-1105).
- Simonyan, K., Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.
- Schmidhuber, J. (2015). Deep learning in neural networks: An overview. Neural Networks, 61, 85-117.

8 Appendix

8.1 Our own DNN implementation from scratch

```
def dnn_architecture(X, y):

    """
    Asks for user input to define the architecture of a deep neural network.

    Args:
    X (ndarray): The input data of shape (input_size, m), where input_size is the number of features
                        and m is the number of training examples.
    y (ndarray): The target labels of shape (output_size, m), where output_size is the number of classes
                        and m is the number of training examples.

    Returns:
    dimensions (list): A list containing the dimensions of the arrays for each hidden layer in the network.
    Each element in the list represents the number of neurons in the corresponding layer.
    The first element represents the number of input features in the input layer, followed by
    the number of neurons in each hidden layer.

    Note:
    This function prompts the user to enter the desired number of layers and the number of neurons in each layer
    (excluding the input and output layer). The function then returns a list of dimensions, which can be used
    to define the architecture of the deep neural network. It will also be used later in the init_dl function.
    """

    # number of total layers excluding input  and output layer

    nb_layers = int(input("enter the number of hidden layers you desire to put in your deep neural network"))

    # dimensions array is to store the number of neurons of each layer also number of input features

    dimensions =[]
```

```python

    # for loop on hidden layers only

    for i in range(nb_layers) :

        # for hidden layers

            nb_hidden_layers = int(input("enter the number of lneurals you desire to put in the {} th layer".format(i+1)))
            dimensions.append(nb_hidden_layers)


        # wih each iteration and in each case we append the dimensions list respectively to keep the order of layers


    # insert the input layer in position 0

    dimensions.insert(0, X.shape[0])

    # append the output layer to dimensions

    dimensions.append(y.shape[0])

    return dimensions

def init_dl(dimensions):

    """
    Initializes the parameters of a deep neural network based on the given dimensions.

    Args:
    dimensions (list): A list containing the dimensions of the arrays for each layer in the network.
    Each element in the list represents the number of neurons in the corresponding layer.
    For example, for a network with n layers, the dimensions list will have n+2 elements,
    where the first element is the input dimension and the last element is the output dimension.

    Returns:
    parameters (dict): A dictionary containing the initialized parameters of the network.
```

```
    The keys of the dictionary represent the layer order, and the values are the initialized
    weight matrices and bias vectors for each layer. The shape of the weight matrices will
    be R(ni x n(i-1)), where ni is the number of neurons in the current layer and n(i-1) is
    the number of neurons in the previous layer.

    Note:
    The number of parameters in the network can be deduced from the number of layers (n) as 2*n.
    """

    # we initialize N as the len of dimensions

    N = len(dimensions)
    parameters = {}

    for i in range(1,N):

        # storing value Wi in a dictionnary with shape(ith layer,"i-1" th layer)
        # dimensions(i) represent the number of neural in current layer and dimensions(i-1) number
        # of neural in previous layer. NB: if i=1 dimensions(i-1) represents number of input features

        parameters["W"+str(i)] = np.random.randn(dimensions[i], dimensions[i-1])

        # same logic is applicable to the biais that is in all cases a vector

        parameters["b"+str(i)] = np.random.randn(dimensions[i],1)

    return parameters

# we must not forget that we will apply a transpose to our Wi matrix later
# like what we did in 2layers code, in order to make the dot product possible


def forward_propagation(X, parameters):

```

```
    """
    Performs forward propagation in a deep neural network to compute the activations of each layer.

    Args:
        X (array-like): Input data of shape (input_dim, m), where input_dim is the number of input features
            and m is the number of examples in the batch.
        parameters (dict): A dictionary containing the parameters of the neural network.
            The keys of the dictionary represent the layer order, and the values are dictionaries
            containing the weight matrices (keys: "W1", "W2", ..., "WN") and bias vectors
            (keys: "b1", "b2", ..., "bN") for each layer.

    Returns:
        activations (dict): A dictionary containing the activations of each layer in the network.
            The keys of the dictionary represent the layer order, and the values are the activation
            arrays (A1, A2, ..., AN) for each layer. The shape of each activation array is
            (n_i, m), where n_i is the number of neurons in the corresponding layer and m is the
            number of examples in the batch.

    Note:
        This function iterates over the layers of the neural network, starting from the first hidden layer (index 1)
        to the output layer (index N). For each layer, it computes the weighted sum (Z) of the previous layer's
        activations and the corresponding weight matrix, adds the bias vector, and applies the sigmoid activation
        function. The resulting activations are stored in the `activations` dictionary and returned as the output.
    """

    # first we initialize a dictionnary that will store activations Ai
    # We will also give this dictionnary a key A0 with value X to solve the first layer problem where Z1 = W1.X + b1

    activations = { "A0" : X }

    # N-1 is the total number of layers that we will obtain using
```

```
integer division

    N = len(parameters)//2

    # this loop will start from 1 since we have already determined W0 to be X and go to N

    for i in range(1,N+1) :

        # Each loop will compute a function Zi , and the next loop will overwrite the previous Zi to compute Z(i+1)

        Z = parameters["W"+str(i)].dot(activations["A"+str(i-1)]) + parameters["b"+str(i)]

        # Here we store the activation output using sigmoid function in the activations dict

        activations["A"+str(i)] = 1 + (1+np.exp(-Z))

    return activations


    def back_propagation (y, activations, parameters):

    """
    Performs backpropagation in a deep neural network to compute the gradients of the parameters.

    Args:
        y (array-like): The true labels or target values of shape (1, m), where m is the number of examples in the batch.
        activations (dict): A dictionary containing the activations of each layer in the network.
                              The keys of the dictionary represent the layer order, and the values are the activation
                              arrays (A1, A2, ..., AN) for each layer.
        parameters (dict): A dictionary containing the parameters of the neural network.
                             The keys of the dictionary represent the layer order, and the values are dictionaries
                             containing the weight matrices (keys: "W1", "W2", ..., "WN") and bias vectors
                             (keys: "b1", "b2", ..., "bN") for each layer.
```

```

    Returns:
        gradients (dict): A dictionary containing the gradients of the parameters.
                          The keys of the dictionary represent the layer order, and the values are the gradients
                          of the weight matrices (dW1, dW2, ..., dWN) and bias vectors (db1, db2, ..., dbN) for each layer.

    Note:
        This function performs backpropagation by iterating through the layers of the neural network in reverse order.
        It starts with the last layer and computes the gradients of the parameters using the generalized formulas
        obtained from the two-layer neural network case. The partial derivatives are specific to the sigmoid activation
        function and the log loss function derived from the Bernoulli probability law. The gradients are then stored
        in the `gradients` dictionary and returned as the output.
    """

    # initializing the an empty dictionnary gradients

    gradients = {}

    # this line will help us get the dimensions of the output vector, in other terms size of the data we trained
    # we do it to get 1/m, the scaling factor "1/m" is used to normalize the average gradient, not the individual
    # gradient updates, and it helps to ensure consistent and stable learning across different dataset sizes.
    # The actual scaling of the gradient update step is done by the learning rate.

    m = y.shape[1]

    N = len(parameters)//2

    # this line here is used to initialize the partial derivative of the last layer of our neural network
    # wich is in fact the first step of our back propagation since we go in reversed path to generate partial derivatives

    dZ = activations["A"+str(N)] - y

```

```
# Proceeding with reversed for loop in order to go throught the NN from last to first layer

for i in reversed(range(1,N+1)) :

    # we intialize A to represent A(i-1), because we noticed that apart from dZN that represents the last layer
    # all of the other equations will use only A(i-1)

    A = activations["A" + str(i-1)]

    # we generate W to represent Wi because we will use it to compute dZ(i-1)

    W = parameters["W" + str(i)]

    # we apply the formulas we reached from generalizing the computations we got previously on the 2 layers NN

    gradients["dW" + str(i)] =  1/m * np.dot(dZ,A.T)

    # same with biais formula we apply the generalized form we got from the 2 layers NN

    gradients["db" + str(i)] = 1/m * np.sum(dZ, axis=1, keepdims=True)

    # this line is to update the value of dZ in order to jump to next layer wich is in fact and order of forward propagation
    # the precedent layer of the actual layer. Each iteration new dZ will overwrite the old dZ
    # and we must not forget to apply the condicion on i > 1 beacuse dZ0 doesnt exist, because in iteration i we
    # compute dZ(i-1)

    if i > 0 :
        dZ =  np.dot(W.T,dZ) * A * (1-A)

return gradients


def log_loss_func(A, y):
"""
Compute the logistic loss function for binary classification.

```

```
    The logistic loss function, also known as the binary cross-entropy loss, is a common
    cost function used in binary classification tasks. It measures the discrepancy between
    the predicted probabilities (A) and the true labels (y) for binary classification.

    Parameters:
    A (numpy.ndarray): Predicted probabilities of shape (m,) where m is the number of samples.
    y (numpy.ndarray): True labels of shape (m,) where m is the number of samples.

    Returns:
    float: The computed logistic loss.

    Explanation:
    The logistic loss function is derived from the concept of maximum likelihood estimation
    for binary outcomes. By taking the negative log-likelihood of the predicted probabilities
    (A) for the positive class (y=1) and the negative class (y=0), and averaging it across
    all samples, we obtain a measure of the model's performance.

    To avoid vanishing probabilities and numerical instability during computation, the loss
    function is constructed using logarithms. The logarithm helps to compress the range of
    probabilities and prevents multiplication of very small values, which could lead to
    underflow issues. By taking the negative logarithm of the predicted probabilities, we
    penalize large deviations from the true labels and encourage the model to better fit the
    data.

    The logistic loss function is commonly used as the cost function in logistic regression
    and as the final layer's activation function in binary classification neural networks.
    It provides a continuous, differentiable, and interpretable measure of the model's
    performance, allowing for efficient optimization through techniques like gradient
```

```
    descent.

    References:
    - Logistic regression: https://en.wikipedia.org/wiki/Logistic_regression
    - Binary cross-entropy loss: https://en.wikipedia.org/wiki/Cross_entropy#Cross-entropy_loss_function_and_logistic_regression
    """
    return (1 / len(y)) * np.sum(-y * np.log(A) - (1 - y) * np.log(1 - A))


def update(gradients, parameters, learning_rate):
    """
    Updates the weights and biases of a deep neural network based on the computed gradients.

    Args:
        gradients (dict): A dictionary containing the gradients of the parameters.
                              The keys of the dictionary represent the layer order, and the values are the gradients
                              of the weight matrices (dW1, dW2, ..., dWN) and bias vectors (db1, db2, ..., dbN) for each layer.
        parameters (dict): A dictionary containing the parameters of the neural network.
                              The keys of the dictionary represent the layer order, and the values are dictionaries
                              containing the weight matrices (keys: "W1", "W2", ..., "WN") and bias vectors
                              (keys: "b1", "b2", ..., "bN") for each layer.
        learning_rate (float): The learning rate or step size used for updating the parameters.

    Returns:
        parameters (dict): A dictionary containing the updated parameters of the neural network.
                              The keys of the dictionary represent the layer order, and the values are dictionaries
                              containing the updated weight matrices and bias vectors for each layer.

    Note:
        This function updates the weights and biases of the neural
```

```
network using the computed gradients.
        It iterates through each layer of the network and performs the parameter updates based on the learning rate
        and corresponding gradient values. The updated parameters are stored in the `parameters` dictionary and
        returned as the output.
    """

    N = len(parameters)//2

    # again a for loop to update every weight and bias in the whole network respectively to his gradients

    for i in range(1,N+1) :

        # update weight Wi following gradient dWi

        parameters["W" + str(i)] = parameters["W" + str(i)] - learning_rate * gradients["dW" + str(i)]

        # update bias bi following gradient dbi

        parameters["b" + str(i)] = parameters["b" + str(i)] - learning_rate * gradients["db" + str(i)]

    return parameters

def visualization(X, y, parameters, ax):
    """
    Visualizes the training progress of the deep neural network.

    Args:
        X (ndarray): The input data of shape (input_size, m), where input_size is the number of features
                     and m is the number of training examples.
        y (ndarray): The target labels of shape (output_size, m), where output_size is the number of classes
                     and m is the number of training examples.
        parameters (dict): A dictionary containing the trained parameters of the neural network.
                           The keys of the dictionary represent the layer order, and the values are dictionaries
                           containing the weight matrices and bias vectors for each layer.
        ax (AxesSubplot): The axes to plot the visualization.
```

```
    Returns:
        None

    """

    # Perform forward propagation to get the final activations
    activations = forward_propagation(X, parameters)
    A_final = activations['A' + str(len(parameters) // 2)]

    # Scatter plot of the input data
    ax[2].scatter(X[0, :], X[1, :], c=y.flatten(), cmap='coolwarm', edgecolors='k')
    ax[2].set_xlabel('X1')
    ax[2].set_ylabel('X2')
    ax[2].set_title('Input Data')

    # Plot the decision boundary
    x1_min, x1_max = X[0, :].min() - 1, X[0, :].max() + 1
    x2_min, x2_max = X[1, :].min() - 1, X[1, :].max() + 1
    xx1, xx2 = np.meshgrid(np.arange(x1_min, x1_max, 0.1), np.arange(x2_min, x2_max, 0.1))
    input_data = np.vstack((xx1.ravel(), xx2.ravel()))
    Z = predict(input_data, parameters)
    Z = Z.reshape(xx1.shape)
    ax[2].contourf(xx1, xx2, Z, alpha=0.4, cmap='coolwarm')
    ax[2].set_xlim(xx1.min(), xx1.max())
    ax[2].set_ylim(xx2.min(), xx2.max())
    ax[2].set_title('Decision Boundary')

    # Plot the loss curve
    ax[0].plot(loss_list, label="Log Loss training curve")
    ax[0].set_xlabel('Iteration')
    ax[0].set_ylabel('Log Loss')
    ax[0].set_title('Training Loss')

    # Plot the accuracy curve
    ax[1].plot(acc_list, label="Accuracy training curve")
    ax[1].set_xlabel('Iteration')
    ax[1].set_ylabel('Accuracy')
    ax[1].set_title('Training Accuracy')

    # Show legend for all plots
    ax[0].legend()
    ax[1].legend()
```

```python

    plt.tight_layout()

    return None


def deep_neural_network(X, y, learning_rate = 0.001, n_iter = 1000):
    """
    Trains a deep neural network using forward propagation, backpropagation, and gradient descent.

    Args:
        X (ndarray): The input data of shape (input_size, m), where input_size is the number of features
                        and m is the number of training examples.
        y (ndarray): The target labels of shape (output_size, m), where output_size is the number of classes
                        and m is the number of training examples.
        learning_rate (float, optional): The learning rate or step size used for updating the parameters during gradient descent.
                                            Defaults to 0.001.
        n_iter (int, optional): The number of iterations or epochs to train the neural network. Defaults to 1000.

    Returns:
        parameters (dict): A dictionary containing the trained parameters of the neural network.
                            The keys of the dictionary represent the layer order, and the values are dictionaries
                            containing the weight matrices and bias vectors for each layer.

    Note:
        This function trains a deep neural network by performing forward propagation, backpropagation, and gradient descent.
        It initializes the parameters, performs the specified number of iterations, and updates the parameters
        in each iteration based on the computed gradients. The training progress is visualized by plotting the
        log loss and accuracy curves over iterations.

        The function returns the trained parameters that can be used for making predictions.

    """
```

```
    np.random.seed(1)

    # initializing the parameters

    dimensions = dnn_architecture()

    # generate parameters using initialization function

    parameters = init_dl(X, y, dimensions)

    # initialize two empty lists that will store values of loss
   function and accuracy

    loss_list = []
    acc_list = []

    # gradient descent algorithm where in each iteration we do a
   forwad prop, a back prop and an update of the parameters
    # tqdm is a library that put a progress bar on the progression ,
   it's derived from arabic "taqadom" wich means progressoin

    for i in tqdm(range(n_iter)):

        activations = forward_propagation(X, parameters)
        gradients = back_propagation(y, parameters, activations)
        parameters = update(gradients, parameters, learning_rate)

        # compute the final value of output layer in activations so we
    use it later in comparison of log_loss

        N = len(parameters)//2
        A_final = activations['A' + str(N)]

        # each ten iterations we will compute the log_loss between y
   and activations of output layer and in parallel
        # we also analyze the accuracy between predicted values and y

        if i%10 == 0 :

            # compute log_loss to see the difference of activations
   from labeled dataset

            loss_list.append(log_loss(y,A_final))
            y_pred = predict(X,parameters)
```

```

            # compute the current accuracy of the model

            accuracy = accuracy_score(y.flatten(),y_pred.flatten())
            acc_list.append(accuracy)


    # plotting training curves

    fig , ax = plt.subplots(nrows=1 , ncols=3 , figsize=(18,4))
    ax[0].plot(loss_list, label="Log Loss training curve")
    ax[0].legend

    ax[1].plot(acc_list, label="Accuracy training curve")
    ax[1].legend

    visualization(X, y, parameters, ax)
    plt.show()

    return parameters
```

8.2 Implementation of the Siamese Network based on the referred paper

```
    # setting up paths

# sub-network within the Siamese neural network that processes another
    image of person whome we need to identify
Pos_path = os.path.join('data','positive')

# sub-network within the Siamese neural network that processes
   different image of person whome we need to identify
Neg_path = os.path.join('data','negative')

# sub-network within the Siamese neural network that processes image
   of person whome we need to identify
Anc_path = os.path.join('data','anchor')


# create the directories

os.makedirs(Pos_path)
os.makedirs(Neg_path)
os.makedirs(Anc_path)
```

```
# Uncompress the tar file Fz labeled faces in the wild dataset

!tar -xf footballers.tgz


for directory in os.listdir('footballers'):
    for file in os.listdir(os.path.join('footballers', directory)):
        Ex_path = os.path.join('footballers', directory, file)
        New_path = os.path.join(Neg_path, file)
        os.replace(Ex_path, New_path)


# Example parameters
image_size = (224, 224)

def preprocess_image(image_path):
    # Read in image from file path
    byte_img = tf.io.read_file(image_path)
    # Load in the image
    image = tf.image.decode_png(byte_img, channels=3)

    # Preprocessing steps - resizing the image to be 105x105x3
    image = tf.image.resize(image, (105, 105))
    # Scale image to be between 0 and 1
    image = image / 255.0

    # Return image
    return image


# Get file paths using glob
file_pattern = os.path.join(Anc_path, '*.png')
file_list = glob.glob(file_pattern)

# Create anchor dataset
anchor = tf.data.Dataset.from_tensor_slices(file_list[:191])
anchor = anchor.map(preprocess_image)


# Get file paths using glob
file_pattern = os.path.join(Neg_path, '*.png')
file_list = glob.glob(file_pattern)

```

```python
# Create negative dataset
negative = tf.data.Dataset.from_tensor_slices(file_list[:191])
negative = negative.map(preprocess_image)


# Get file paths using glob
file_pattern = os.path.join(Pos_path, '*.png')
file_list = glob.glob(file_pattern)

# Create positive dataset
positive = tf.data.Dataset.from_tensor_slices(file_list[:191])
positive = positive.map(preprocess_image)


# Function to display a grid of images from each repository
# To make sure that our previous block was well done executed
# as you can see it is accurate

def display_images(images, labels):
    fig, axes = plt.subplots(nrows=2, ncols=5, figsize=(12, 6))
    for i, ax in enumerate(axes.flat):
        ax.imshow(images[i])
        ax.axis('off')
        ax.set_title(labels[i])
    plt.tight_layout()
    plt.show()

# Convert datasets to lists


anchor_list = list(anchor.as_numpy_iterator())
negative_list = list(negative.as_numpy_iterator())
positive_list = list(positive.as_numpy_iterator())

# Randomly select 10 images from each directory for display


random.shuffle(anchor_list)
random.shuffle(negative_list)
random.shuffle(positive_list)

display_images(anchor_list[:10], ["Anchor"] * 10)
display_images(negative_list[:10], ["Negative"] * 10)
display_images(positive_list[:10], ["Positive"] * 10)

```

```python

def make_embedding():
    """
    Constructs an embedding model based on the architecture from the Siamese Neural Network for one-shot image recognition research paper.

    Returns:
    - A Keras Model object representing the embedding model.

    The architecture of the model consists of convolutional and dense layers to create an embedding representation of input images. It follows the design presented in the Siamese Neural Network for one-shot image recognition research paper. The model takes an input image tensor of shape (105, 105, 3) and outputs an embedding vector of size 4096.

    The model architecture:
    - Three blocks of Conv2D and MaxPooling2D layers for feature extraction.
    - A final embedding block with Conv2D layer.
    - A Flatten layer to reshape the feature maps.
    - A Dense layer with sigmoid activation for generating the embedding representation.

    This embedding model can be used for various tasks such as image similarity, clustering, or classification.

    Reference:
    Siamese Neural Networks for One-shot Image Recognition by Gregory Koch, Richard Zemel, Ruslan Salakhutdinov. Published in 2015.

    """

    # Define the input layer
    inp = Input(shape=(105,105,3), name='input_image')

    # First block
    # First Convolutional layer with 64 filters and ReLU activation,
    # followed by Max Pooling layer with 2x2 window and 'same' padding

    c1 = Conv2D(64, (10,10), activation='relu')(inp)
    m1 = MaxPooling2D(64, (2,2), padding='same')(c1)

    # Second block
```

```
    # Second Convolutional layer with 128 filters and ReLU activation,
    # followed by Max Pooling layer with 2x2 window and 'same' padding
    c2 = Conv2D(128, (7,7), activation='relu')(m1)
    m2 = MaxPooling2D(64, (2,2), padding='same')(c2)

    # Third block
    # Third Convolutional layer with 128 filters and ReLU activation,
    # followed by Max Pooling layer with 2x2 window and 'same' padding
    c3 = Conv2D(128, (4,4), activation='relu')(m2)
    m3 = MaxPooling2D(64, (2,2), padding='same')(c3)

    # Final embedding block
    # Fourth Convolutional layer with 256 filters and ReLU activation
    c4 = Conv2D(256, (4,4), activation='relu')(m3)
    # Flatten layer to convert 3D feature maps into 1D vector
    f1 = Flatten()(c4)
    # Fully connected Dense layer with 4096 units and sigmoid activation
    d1 = Dense(4096, activation='sigmoid')(f1)

    # Returns the embedding model
    return Model(inputs=[inp], outputs=[d1], name='embedding')

    # Siamese L1 Distance class
class L1Dist(Layer):

    # Init method - inheritance
    def __init__(self, **kwargs):
        super().__init__()

    # This is the functin that computes similarity calculation between pictures
    def call(self, input_embedding, validation_embedding):
        return tf.math.abs(input_embedding - validation_embedding)


def make_siamese_model():

    """
    Creates a Siamese network model for one-shot image recognition.

    This function constructs a Siamese network model with an input pipeline for anchor and validation images,
    a distance calculation layer based on L1 distance, and a classification layer.
```

```
    Returns:
        A `Model` object representing the Siamese network model.

    """

    # Anchor image input in the network

    input_image = Input(name='input_img', shape=(105,105,3))

    # Validation image in the network

    validation_image = Input(name='validation_img', shape=(105,105,3))

    # Combine siamese distance components
    # Create an instance of the L1Dist layer

    siamese_layer = L1Dist()

    # Rename the siamese layer for clarity

    siamese_layer._name = 'distance'

    # Calculate the distances between the embeddings of anchor and
    validation images

    distances = siamese_layer(embedding(input_image), embedding(
    validation_image))

    # Classification layer
    # Pass the distances through a dense layer with sigmoid activation

    classifier = Dense(1, activation='sigmoid')(distances)

    # Create the Siamese network model

    return Model(inputs=[input_image, validation_image], outputs=
    classifier, name='SiameseNetwork')

# used only to optimize the perf of the func by converting it to TF
    graph
@tf.function
def train_step(batch):

    """
```

```
    Performs a single training step for the siamese model.

    Args:
        batch: A batch of training data containing anchor image,
positive/negative image, and label.

    Returns:
        loss: The calculated loss value for the batch.
    """

    # The tf.GradientTape() context manager records the operations
executed within its scope, storing them in a tape
    # for later use during backpropagation to compute gradients with
respect to the recorded operations. This allows
    # for efficient differentiation and gradient calculation.

    with tf.GradientTape() as tape:

        # Extract anchor and positive/negative images

        X = batch[:2]

        # extract labels

        y = batch[2]

        # Forward pass in the siamese model

        yhat = siamese_model(X, training=True)

        # Calculate binary cross-entropy loss

        loss = binary_cross_loss(y, yhat)

    print(loss)

    # Calculate gradients of the loss with respect to the variables

    grad = tape.gradient(loss, siamese_model.trainable_variables)

    # Classical update of weights like in DNN

    opt.apply_gradients(zip(grad, siamese_model.trainable_variables))

    # Return loss
```

```
    return loss


    # care there is a diff betweeen capital EPOCHS and lower epochs one is the
# variable name the other is an iteration variable

def train(data, EPOCHS):
    """
    Trains the Siamese network using the provided data for a specified number of epochs.

    Args:
        data (tf.data.Dataset): The dataset containing the training samples.
        EPOCHS (int): The number of training epochs.

    Returns:
        None
    """

    # Loop through epochs

    for epoch in range(1, EPOCHS+1):
        print('\n Epoch {}/{}'.format(epoch, EPOCHS))

        # creating tf progress bar similar to taqadom tqdm in python

        progbar = tf.keras.utils.Progbar(len(data))

        # Loop through each batch in training data

        for idx, batch in enumerate(data):

            # Run train step here, single step by single step

            train_step(batch)

            # Then update the progbar to show completion of current batch

            progbar.update(idx+1)

        # Save checkpoints

```

```
        if epoch % 10 == 0:
            checkpoint.save(file_prefix=checkpoint_prefix)


    def verify(model, detection_threshold, verification_threshold):
    # Build results array
    results = []
    for image in os.listdir(os.path.join('application_data', 'verification_images')):
        input_img = preprocess_image(os.path.join('application_data', 'input_image', 'input_image.jpg'))
        validation_img = preprocess_image(os.path.join('application_data', 'verification_images', image))

        # Make Predictions
        result = model.predict(list(np.expand_dims([input_img, validation_img], axis=1)))
        results.append(result)

    # Detection Threshold: Metric above which a prediciton is considered positive
    detection = np.sum(np.array(results) > detection_threshold)

    # Verification Threshold: Proportion of positive predictions / total positive samples
    verification = detection / len(os.listdir(os.path.join('application_data', 'verification_images')))
    verified = verification > verification_threshold

    return results, verified
```

8.3 Real time verification using Opencv and cam

```
    cap = cv2.VideoCapture(0)
while cap.isOpened():

    ret, frame = cap.read()
    frame = frame[120:120+250,200:200+250, :]

    cv2.imshow('Verification', frame)

    # Verification trigger

    if cv2.waitKey(10) & 0xFF == ord('v'):

```

```
        # Save input image to application_data/input_image folder

        cv2.imwrite(os.path.join('application_data', 'input_image', '
    input_image.jpg'), frame)

        # Run verification

        results, verified = verify(model, 0.5, 0.5)
        print(verified)

    if cv2.waitKey(10) & 0xFF == ord('q'):
        break
cap.release()
cv2.destroyAllWindows()
```