**Cover page**

Text

Description automatically generated with medium confidence

**Submitted to:**  ***Dr/ عبدالمنعم فودة***

**Submitted by:  *عاصم يسري عبدالعظيم نصر***

Student Code: 4190523

level**: 4**

year**: 2018**

**Task title**

**Data Compression Using ANN**

**Ch1- introduction**

1. **Task objective**

The objective of the task is to compress data using artificial neural networks (ANN) as a tool. The ANN can be trained to learn patterns in data and generate compressed versions of it, thus reducing the size of the data without significant loss of information.

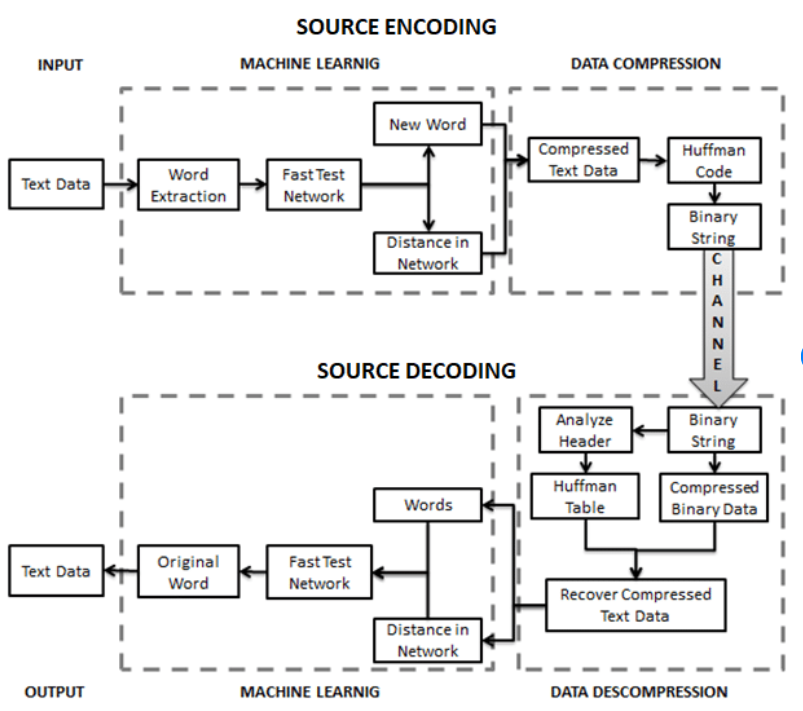
1. **Importance**

Data compression is an important task in various fields such as data storage, transmission, and processing. The compressed data requires less storage space, bandwidth, and processing power, which makes it easier and faster to transfer and process large amounts of data. ANN-based compression techniques can achieve higher compression ratios and better quality compared to traditional compression algorithms, especially for data with complex patterns.

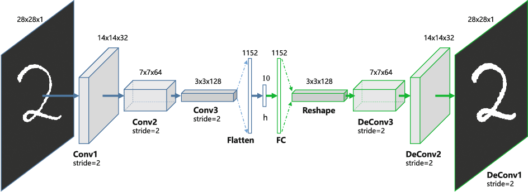
1. **Tool Used**

The tools used to achieve this task can be either software or hardware. In software-based solutions, popular deep learning frameworks such as TensorFlow, PyTorch, or Keras can be used to implement the ANN models and train them on the data. For hardware-based solutions, specialized chips such as Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs) can be used to accelerate the training and inference of ANN models.

**Ch2-Block diagram :**



**Ch-3 Circuit diagram**



**Ch-4 task implementations:**

1. **Algorithm Followed**

1. Load the data to be compressed

2. Preprocess the data (e.g., normalize, standardize, etc.)

3. Split the data into training and validation sets

4. Define the neural network architecture

5. Train the neural network using the training set

6. Evaluate the neural network performance using the validation set

7. If the performance is satisfactory, proceed to step 8, otherwise go back to step 4 and modify the architecture

8. Use the trained neural network to compress new data

9. Save the compressed data and the trained neural network weights for future use

1. **High Level Language Code**

import numpy as np

from keras.layers import Input, Dense

from keras.models import Model

from keras.datasets import mnist

import matplotlib.pyplot as plt

# this is the size of our encoded representations

encoding\_dim = 32 # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# this is our input placeholder

input\_img = Input(shape=(784,))

# "encoded" is the encoded representation of the input

encoded = Dense(encoding\_dim, activation='relu')(input\_img)

# "decoded" is the lossy reconstruction of the input

decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction

autoencoder = Model(input\_img, decoded)

# this model maps an input to its encoded representation

encoder = Model(input\_img, encoded)

# create a placeholder for an encoded (32-dimensional) input

encoded\_input = Input(shape=(encoding\_dim,))

# retrieve the last layer of the autoencoder model

decoder\_layer = autoencoder.layers[-1]

# create the decoder model

decoder = Model(encoded\_input, decoder\_layer(encoded\_input))

# configure our model to use a per-pixel binary crossentropy loss, and the Adadelta optimizer:

autoencoder.compile(optimizer='adadelta', loss='binary\_crossentropy')

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

print (x\_train.shape)

print (x\_test.shape)

autoencoder.fit(x\_train, x\_train,

epochs=50,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test, x\_test))

# encode and decode some digits

# note that we take them from the \*test\* set

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)

n = 20 # how many digits we will display

plt.figure(figsize=(20, 4))

for i in range(n):

# display original

ax = plt.subplot(2, n, i + 1)

plt.imshow(x\_test[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

# display reconstruction

ax = plt.subplot(2, n, i + 1 + n)

plt.imshow(decoded\_imgs[i].reshape(28, 28))

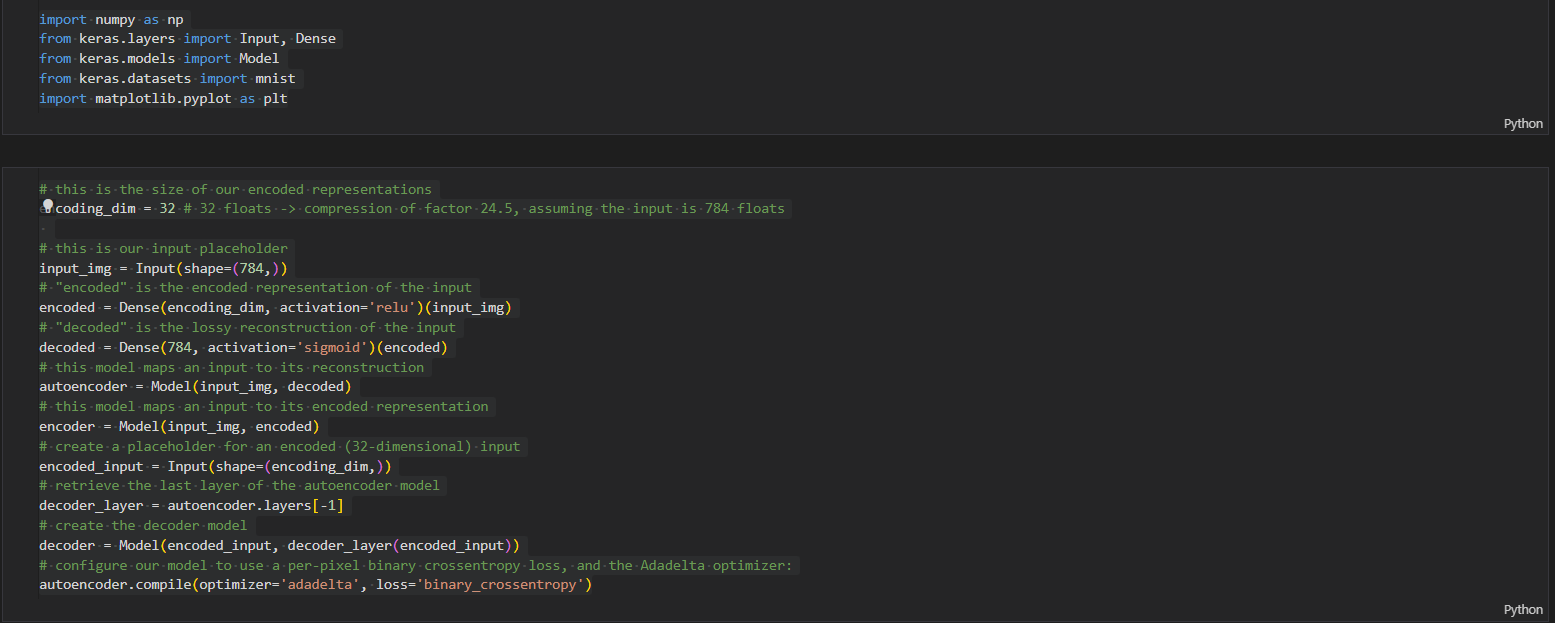
plt.gray()

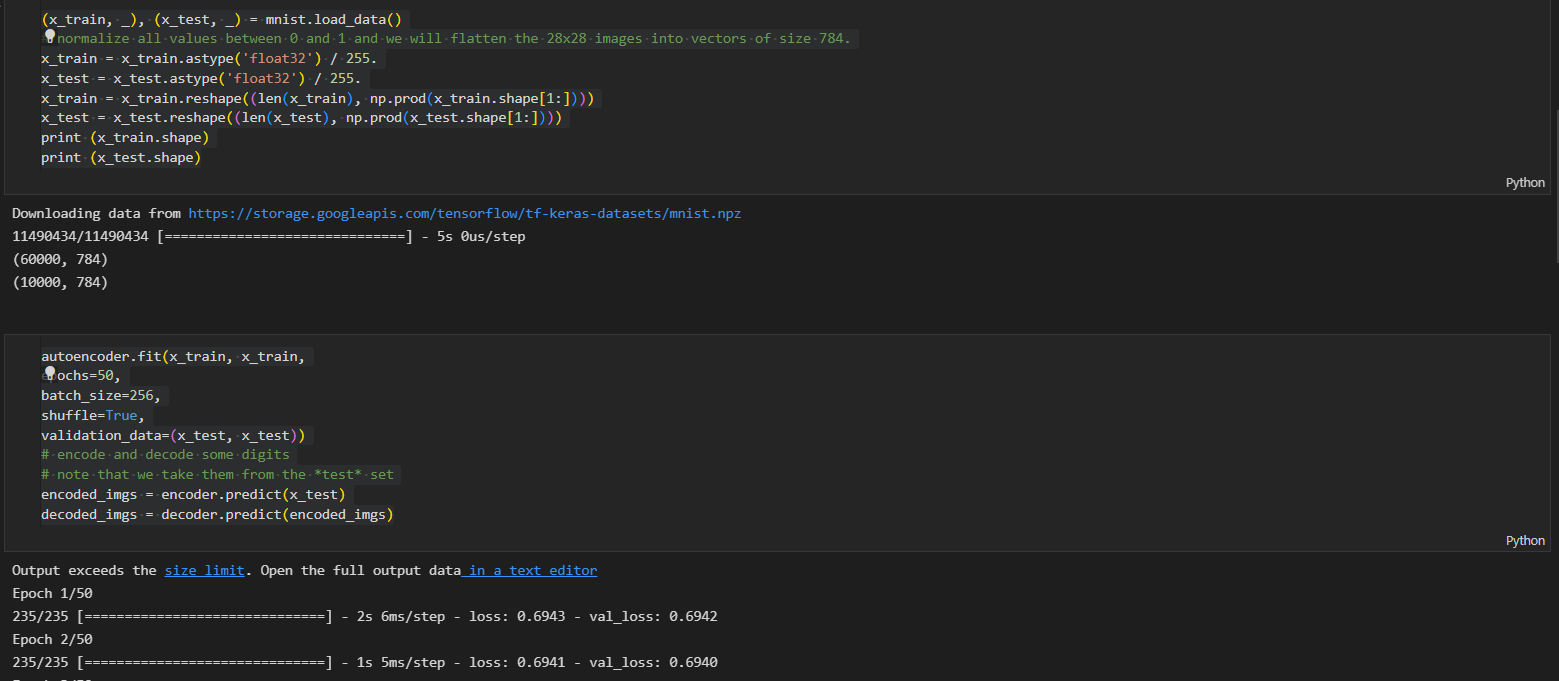
ax.get\_xaxis().set\_visible(False)

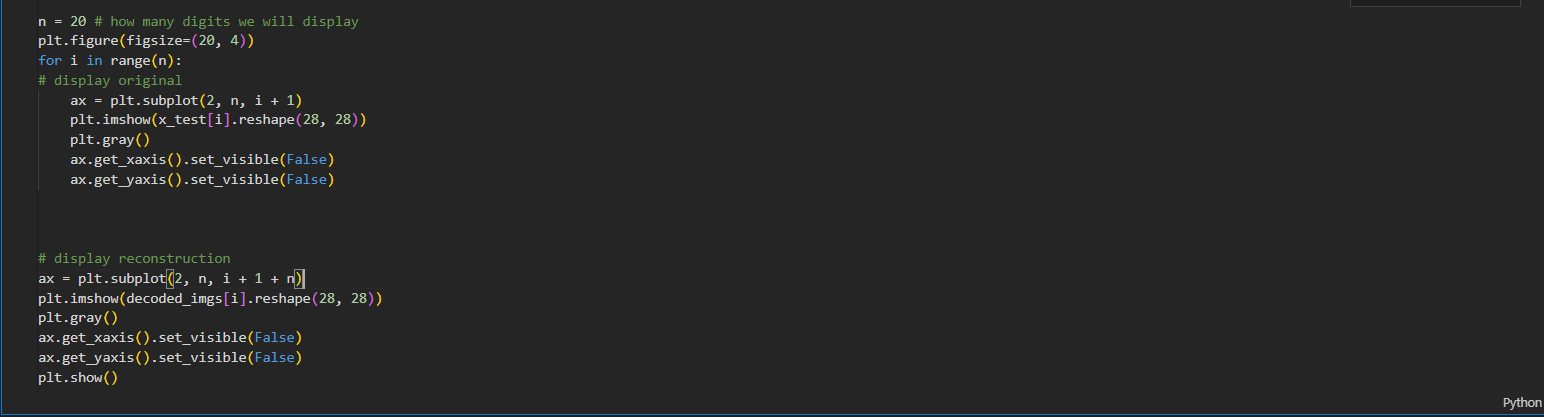
ax.get\_yaxis().set\_visible(False)

plt.show()

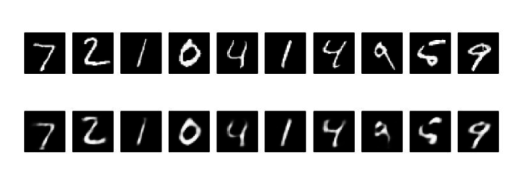
1. **Screenshot of the implementation**







**The Output:**



**Ch5 Conclusion :**

The code implements an autoencoder neural network using Keras and trains it on the MNIST dataset. The autoencoder learns to encode the input data into a lower-dimensional representation and then decode it back to its original form. The network architecture consists of an input layer, a hidden layer with a ReLU activation function, and an output layer with a sigmoid activation function. The input and output layers have the same shape of 784, which corresponds to the flattened 28x28 MNIST images. The hidden layer has a size of 32, which means that the encoded representation has a dimensionality of 32.

During implementation, one potential difficulty is understanding the code itself and the concepts behind the autoencoder neural network. It is important to understand the different layers and activation functions used in the network and how they contribute to the overall architecture.

Another potential difficulty is training the autoencoder on the MNIST dataset, which is a large dataset with many images. This can take a significant amount of time and resources, especially if training is done on a CPU rather than a GPU. One way to overcome this is to use a smaller subset of the data or train for fewer epochs.

Finally, visualizing the reconstructed images can also be challenging, especially if the code is not set up correctly. It is important to ensure that the input and output shapes are correct and that the images are displayed in a clear and organized way.

To overcome these difficulties, it is important to have a good understanding of the concepts behind the autoencoder neural network and to carefully review and debug the code to ensure that it is running correctly. It is also helpful to break down the implementation into smaller parts and test each part separately to ensure that everything is working as expected. Additionally, it is important to have access to a powerful computing resource, such as a GPU, to reduce training time and improve performance. Finally, plotting and displaying the reconstructed images in a clear and organized way can help to identify any issues with the implementation and make it easier to interpret the results.

**REFRENCES**

1. "Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville. MIT Press, 2016.

2. "Hands-On Machine Learning with Scikit-Learn and TensorFlow" by Aurélien Géron. O'Reilly Media, 2017.

3. "Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles" by Mehdi Noroozi and Paolo Favaro. ECCV 2016.

4. "Auto-Encoding Variational Bayes" by Diederik P. Kingma and Max Welling. ICLR 2014.

5. "A Beginner's Guide to Autoencoders" by Adit Deshpande.

<https://www.datacamp.com/community/tutorials/autoencoder-keras-tutorial>

**Task title**

**Data Compression Using ANN**

**Ch1- introduction**

1. **Task objective**

The objective of the task is to compress data using artificial neural networks (ANN) as a tool. The ANN can be trained to learn patterns in data and generate compressed versions of it, thus reducing the size of the data without significant loss of information.

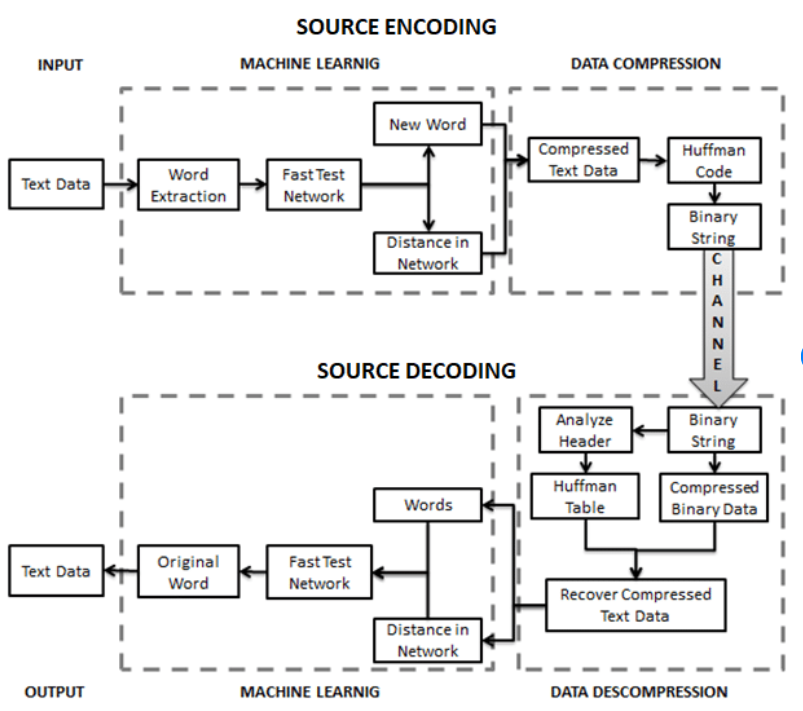
1. **Importance**

Data compression is an important task in various fields such as data storage, transmission, and processing. The compressed data requires less storage space, bandwidth, and processing power, which makes it easier and faster to transfer and process large amounts of data. ANN-based compression techniques can achieve higher compression ratios and better quality compared to traditional compression algorithms, especially for data with complex patterns.

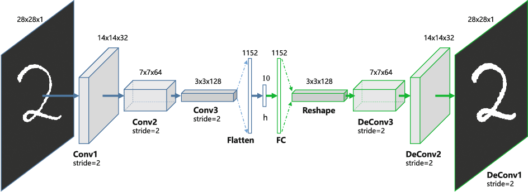
1. **Tool Used**

The tools used to achieve this task can be either software or hardware. In software-based solutions, popular deep learning frameworks such as TensorFlow, PyTorch, or Keras can be used to implement the ANN models and train them on the data. For hardware-based solutions, specialized chips such as Graphics Processing Units (GPUs) or Field-Programmable Gate Arrays (FPGAs) can be used to accelerate the training and inference of ANN models.

**Ch2-Block diagram :**



**Ch-3 Circuit diagram**



**Ch-4 task implementations:**

1. **Algorithm Followed**

1. Load the data to be compressed

2. Preprocess the data (e.g., normalize, standardize, etc.)

3. Split the data into training and validation sets

4. Define the neural network architecture

5. Train the neural network using the training set

6. Evaluate the neural network performance using the validation set

7. If the performance is satisfactory, proceed to step 8, otherwise go back to step 4 and modify the architecture

8. Use the trained neural network to compress new data

9. Save the compressed data and the trained neural network weights for future use

1. **High Level Language Code**

import numpy as np

from keras.layers import Input, Dense

from keras.models import Model

from keras.datasets import mnist

import matplotlib.pyplot as plt

# this is the size of our encoded representations

encoding\_dim = 32 # 32 floats -> compression of factor 24.5, assuming the input is 784 floats

# this is our input placeholder

input\_img = Input(shape=(784,))

# "encoded" is the encoded representation of the input

encoded = Dense(encoding\_dim, activation='relu')(input\_img)

# "decoded" is the lossy reconstruction of the input

decoded = Dense(784, activation='sigmoid')(encoded)

# this model maps an input to its reconstruction

autoencoder = Model(input\_img, decoded)

# this model maps an input to its encoded representation

encoder = Model(input\_img, encoded)

# create a placeholder for an encoded (32-dimensional) input

encoded\_input = Input(shape=(encoding\_dim,))

# retrieve the last layer of the autoencoder model

decoder\_layer = autoencoder.layers[-1]

# create the decoder model

decoder = Model(encoded\_input, decoder\_layer(encoded\_input))

# configure our model to use a per-pixel binary crossentropy loss, and the Adadelta optimizer:

autoencoder.compile(optimizer='adadelta', loss='binary\_crossentropy')

(x\_train, \_), (x\_test, \_) = mnist.load\_data()

# normalize all values between 0 and 1 and we will flatten the 28x28 images into vectors of size 784.

x\_train = x\_train.astype('float32') / 255.

x\_test = x\_test.astype('float32') / 255.

x\_train = x\_train.reshape((len(x\_train), np.prod(x\_train.shape[1:])))

x\_test = x\_test.reshape((len(x\_test), np.prod(x\_test.shape[1:])))

print (x\_train.shape)

print (x\_test.shape)

autoencoder.fit(x\_train, x\_train,

epochs=50,

batch\_size=256,

shuffle=True,

validation\_data=(x\_test, x\_test))

# encode and decode some digits

# note that we take them from the \*test\* set

encoded\_imgs = encoder.predict(x\_test)

decoded\_imgs = decoder.predict(encoded\_imgs)

n = 20 # how many digits we will display

plt.figure(figsize=(20, 4))

for i in range(n):

# display original

ax = plt.subplot(2, n, i + 1)

plt.imshow(x\_test[i].reshape(28, 28))

plt.gray()

ax.get\_xaxis().set\_visible(False)

ax.get\_yaxis().set\_visible(False)

# display reconstruction

ax = plt.subplot(2, n, i + 1 + n)

plt.imshow(decoded\_imgs[i].reshape(28, 28))

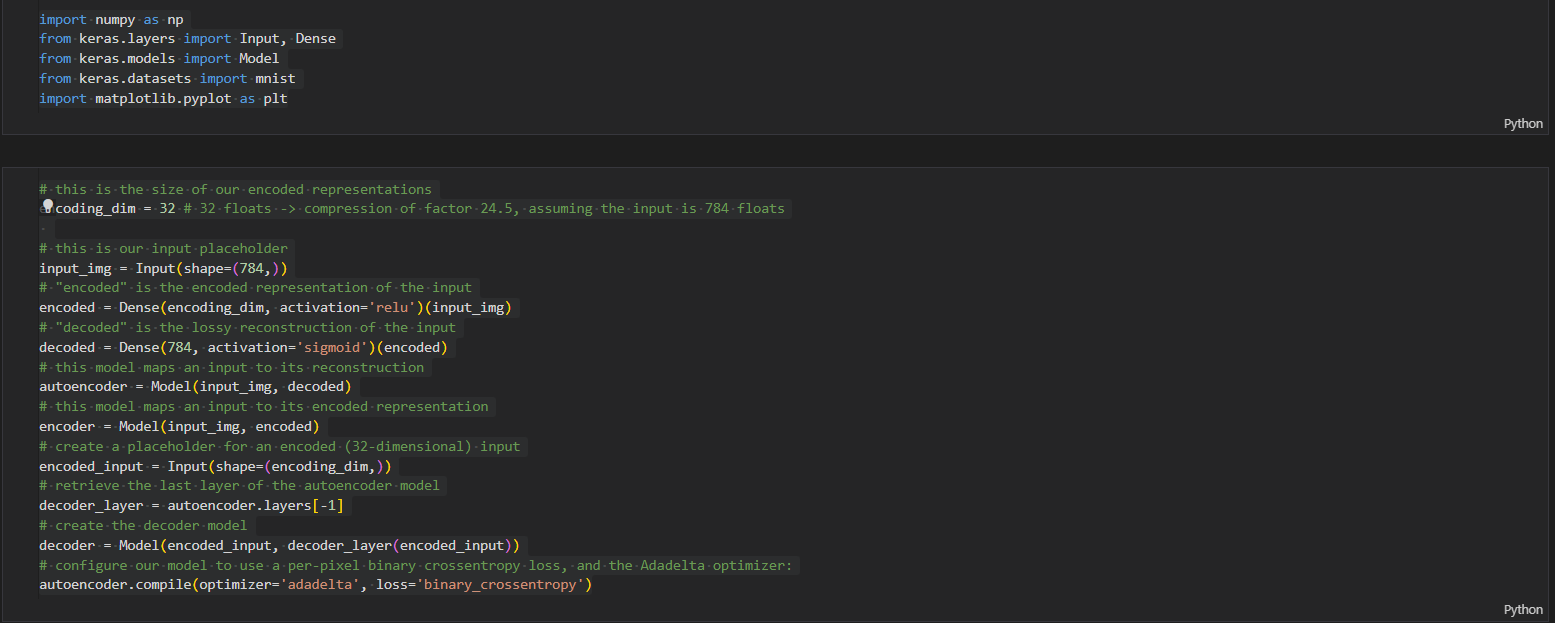
plt.gray()

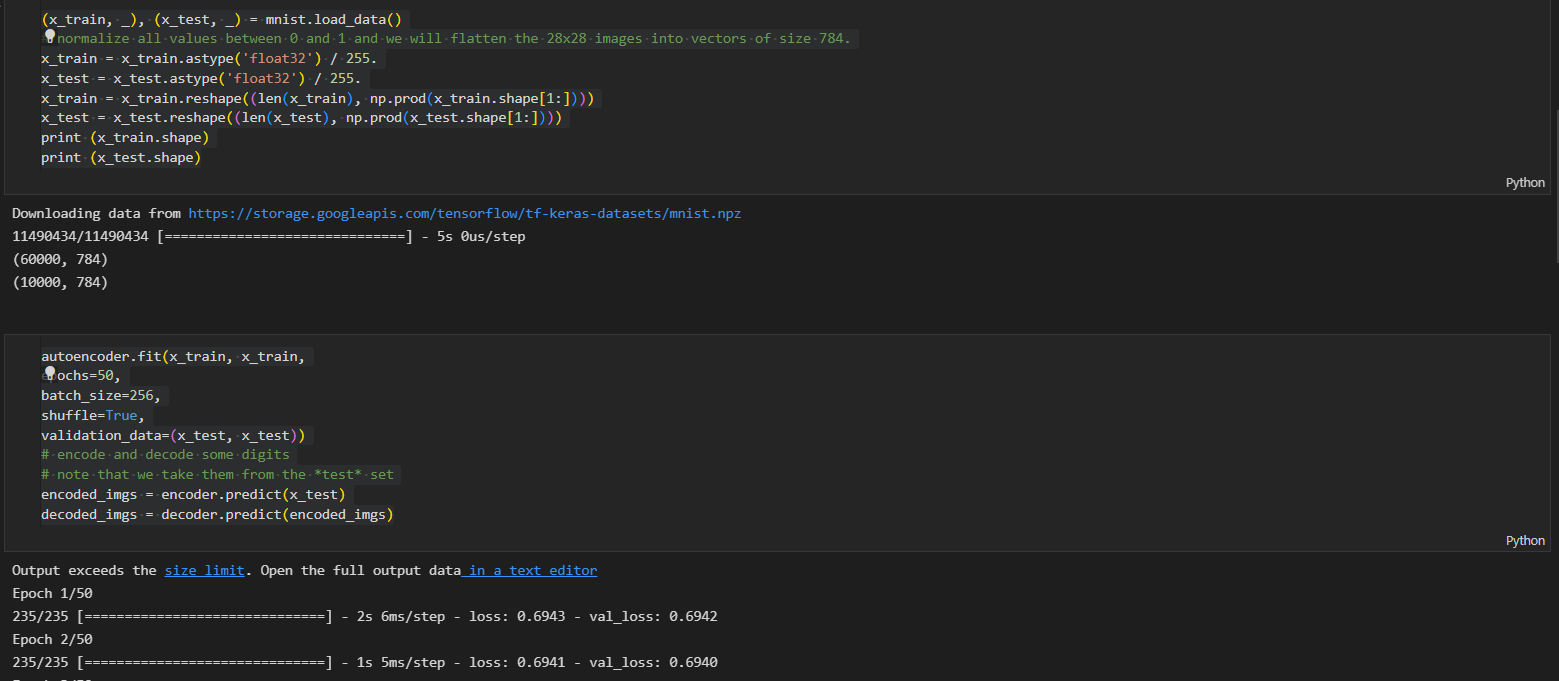
ax.get\_xaxis().set\_visible(False)

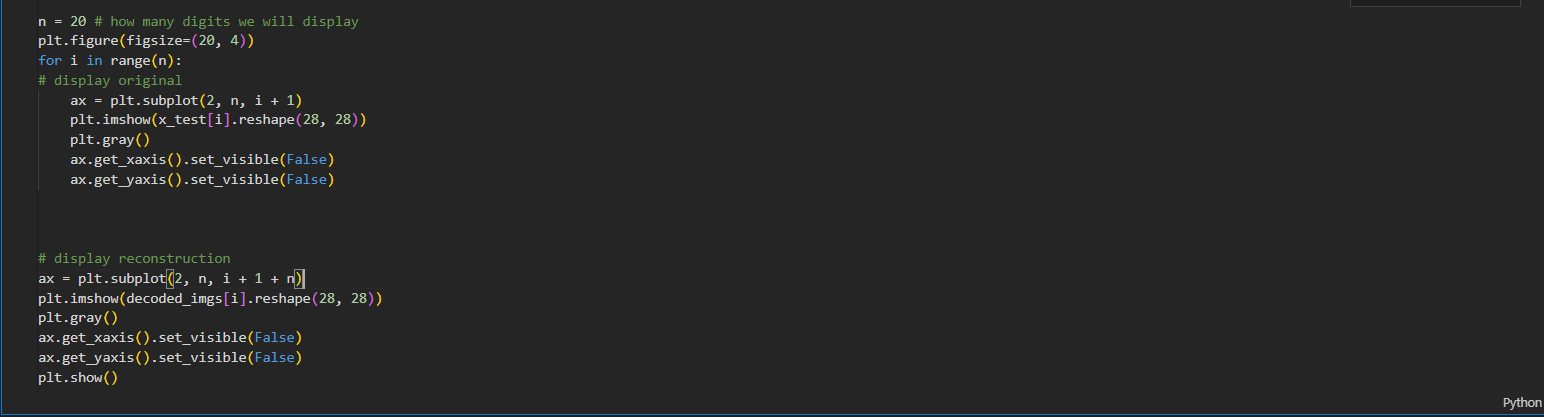
ax.get\_yaxis().set\_visible(False)

plt.show()

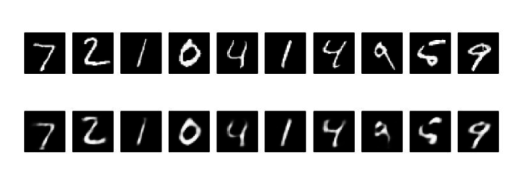
1. **Screenshot of the implementation**







**The Output:**



**Ch5 Conclusion :**

The code implements an autoencoder neural network using Keras and trains it on the MNIST dataset. The autoencoder learns to encode the input data into a lower-dimensional representation and then decode it back to its original form. The network architecture consists of an input layer, a hidden layer with a ReLU activation function, and an output layer with a sigmoid activation function. The input and output layers have the same shape of 784, which corresponds to the flattened 28x28 MNIST images. The hidden layer has a size of 32, which means that the encoded representation has a dimensionality of 32.

During implementation, one potential difficulty is understanding the code itself and the concepts behind the autoencoder neural network. It is important to understand the different layers and activation functions used in the network and how they contribute to the overall architecture.

Another potential difficulty is training the autoencoder on the MNIST dataset, which is a large dataset with many images. This can take a significant amount of time and resources, especially if training is done on a CPU rather than a GPU. One way to overcome this is to use a smaller subset of the data or train for fewer epochs.

Finally, visualizing the reconstructed images can also be challenging, especially if the code is not set up correctly. It is important to ensure that the input and output shapes are correct and that the images are displayed in a clear and organized way.

To overcome these difficulties, it is important to have a good understanding of the concepts behind the autoencoder neural network and to carefully review and debug the code to ensure that it is running correctly. It is also helpful to break down the implementation into smaller parts and test each part separately to ensure that everything is working as expected. Additionally, it is important to have access to a powerful computing resource, such as a GPU, to reduce training time and improve performance. Finally, plotting and displaying the reconstructed images in a clear and organized way can help to identify any issues with the implementation and make it easier to interpret the results.

**REFRENCES**

1. "Deep Learning" by Ian Goodfellow, Yoshua Bengio, and Aaron Courville. MIT Press, 2016.

2. "Hands-On Machine Learning with Scikit-Learn and TensorFlow" by Aurélien Géron. O'Reilly Media, 2017.

3. "Unsupervised Learning of Visual Representations by Solving Jigsaw Puzzles" by Mehdi Noroozi and Paolo Favaro. ECCV 2016.

4. "Auto-Encoding Variational Bayes" by Diederik P. Kingma and Max Welling. ICLR 2014.

5. "A Beginner's Guide to Autoencoders" by Adit Deshpande.

<https://www.datacamp.com/community/tutorials/autoencoder-keras-tutorial>

**Task 1**

**Artificial Neural Networks (ANNs)**

**content**

**Ch1- introduction**

**Ch2- Block diagram**

**Ch3- Circuit diagram**

**Ch4- task implementations**

**Ch5- Conclusion**

**REFRENCES**

1. **introduction**

During the infancy of the development of Neural Networks technology, one thing that excited people's interest was its analogy to biological systems. Even though not all have been understood about the learning processes of human neural systems, Artificial Neural Networks (ANN) have, without a doubt, provided solutions to many problems in different application areas. The brain is a highly complex, nonlinear, and parallel information processing system. It consists of about one hundred billion () neural cells, each connected to about 10,000 neighboring neurons and receiving signals from there. The brain routinely accomplishes perceptual recognition tasks (e.g., recognizing a familiar face in a scene) in about 100-200 msec. The neuron, the basic information processing element (PE) in the central nervous system plays a very important and diverse role in human sensory processing, control, and cognition. The brain can do complex tasks by its ability to learn from experience. An Artificial Neural Network is designed to model the working of the human brain.

The ANN is usually implemented using electronic components (digital or analog) and/or simulated on a digital computer.

It employs massive interconnection of simple computing cells called "neurons" or "processing elements (PE),"

It resembles the brain in two ways:

• knowledge is acquired by the network through learning process

• inter neuron connection strengths (synaptic weights) are responsible for storing the knowledge.

The way the synaptic weights change is what makes the design of ANNs. Such an approach is close to linear adaptive filter theory, which is well established and is used in many diverse fields such as communication, control, sonar, radar, and biomedical engineering.

An ANN works as follows:

A neuron receives inputs from many other neurons or from an external stimulus. A weighted sum of these inputs is fed into a nonlinear activation function. The output of this function is fanned out (distributed) to connections to other neurons. The topology of neurons’ connections defines the flow of information in the network. The way the weights are adjusted in the network constitutes the learning process. Thus, the three essential components of an ANN computational system are - activation function, architecture, and the learning law. Due to the differences in these three components, different ANN structures are explored for various applications and these structures differ in their computational complexities and requirements. The taxonomy along with the interrelationship of neural structures is shown in Figure 1.

The main attributes of neural processing are its nonlinear and adaptive learning capability, which enables machines to recognize possible variations of a same object or pattern and/or to identify unknown functions and mappings based on a finite set of training data, which can be noisy with missing information. Based on this 'Training by example' property with strong support of statistical and optimization theories, neural networks are becoming one of the most powerful and appealing nonlinear and adaptive data analysis tools for a variety of signal processing applications. In the present article we review the recent developments in the field of Neural Networks applications in signal processing which are as applicable in Nuclear Engineering as in any other engineering discipline. The topics covered include pattern recognition and synthesis, control, image analysis and several others. Essentially, neural networks have become a very effective tool in signal processing, particularly in various recognition and/or identification tasks.

* 1. Learning in Neural Networks

Learning is the most important aspect of an ANN. All the knowledge in the ANN is due to interconnection weights between different neurons and it is the learning process that determines the weights. On the basis of learning, ANN's can be classified into two broad classes; supervised and unsupervised learning models. The supervised learning models are trained by exposing them to

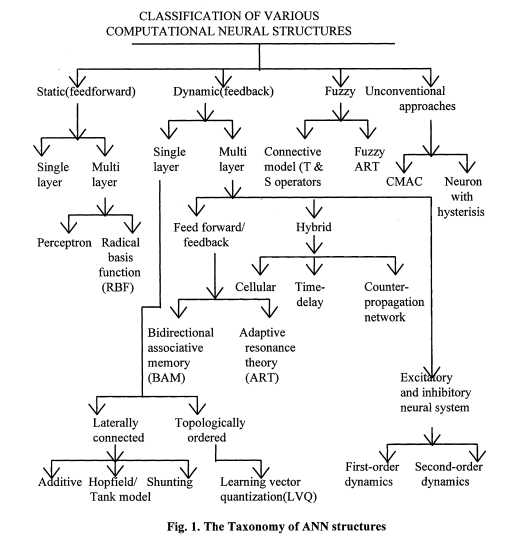
****

Fig. 1. The Taxonomy of ANN structures

example input/output vector pairs to implement mappings that match the examples or underlying equations as closely as possible. The popular examples of supervised learning models are Multi-layer Perceptron neural network (MLP) [1,2] and Radial Basis Function neural network [3,4]. MLP is the most popular although network design and learning and adaptation complexities are faced while implementing it. RBF is relatively faster and is gaining popularity due to its closer relationship to Bayesian theory.

The unsupervised or self organizing models group the input sample into self similar classes based on some specific measure of similarity. The examples of unsupervised ANN's are the Adaptive Resonance Theory (ART) network [5,6] and Kohonen's self organizing feature MAP (SOFM) [7,8]. The SOFM can also be used for regression modelling. Supervised learning ANN's can also be classified as Static or Dynamic Models [9]. Static models perform the mapping from one vector space to another without reference to any time dependent factors whereas dynamic models take into account dynamic/time dependent factors.

These ANN's can be with feed forward, with output feedback or with state feedback. A taxonomy of the learning algorithms used for various applications is given in Figure 2.

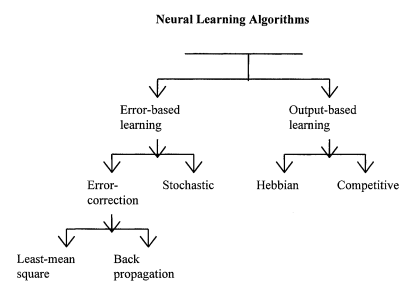
****

Fig. 2. The Taxonomy of ANN algorithm

2. The Use of ANN for Signal Processing

ANN's, because of their inherently non-linear nature are well suited to signal processing applications where the classical assumptions of linearity and second order Gaussian noise statistics cannot be made [10]. Real signals are usually generated by dynamic processes that are non-linear and non-Gaussian, in which case, the classical approach may not always produce an optimal solution [11]. There are two main approaches to engineering problems - parametric and non-parametric. The parametric approach is based on apriori models derived from the scientific knowledge about the problem. The nonparametric approach is based on the use of more general models trained to replicate desired behaviours using sufficiently representative data sets. In practice, the best solution is often desired through a mixture of parametric and nonparametric techniques. In this context, ANN's are useful because they can be treated as a nonparametric technique, which can model an underlying process from example data. They can also adapt their model parameters to statistical change with time.

The general goal for ANN signal processing algorithms in practical application is, given noisy and imprecise non-linear data, to enhance desired responses and reduce irrelevant and unwanted responses. The two extremes of this general goal are:

• Pattern Recognition, and

• Signal Conditioning problems.

In pattern recognition, a discrete decision is made regarding the presence or absence of a desired pattern, whereas in signal conditioning, the pattern is to be recovered. Most signal processing problems are usually related to a time or spatial data series. For example

• pattern recognition and signal detection.

• signal and system modeling and inverse modeling.

• signal filtering and smoothing.

• signal and system prediction.

The application of ANN's to nonlinear signal processing requires considerable judgement to design and signal processing issues. An ANN design involves

• raw data preprocessing,

• feature extraction frolJ1 the preprocessed data,

• selection of network model and type, and

• network testing and evaluation.

The design process is a complex iterative and interactive problem specific task. The choice of the network model and type is based on the precise requirements of the problem. Different ANN models and types have different features that may be typically suitable for particular application. In fact the arithmetic calculating power of a computer has far prevailed over the human capability. But, for complicated signal processing problems, where large quantity of data and sophisticated inference are involved in real time, the performance of electronic hardware is still inferior to human. To perform associatively is essential for many intellectual activities. Therefore, one has to take advantage of parallel processing capability of neural networks in developing intelligent machines, which can have both superior computing speed and profound associative and inference abilities. A configuration of a generic neural network based intelligent system is shown in Figure 3. processing.

A picture containing text, font, line, screenshot

Description automatically generated

Fig. 3. A hybrid scheme of neural network based intelligent system

A high-level symbolic processor can access data from different sources, such as digital data from its cache memory, the main memory and hard disks. This data could also be from the electronic/optical smart sensors with local processing. Inference and decision made by the neural network system can then be used to activate the controller. In what follows we review the fundamentals of Neural Networks in Signal processing and their applications. The topics covered include dynamic modeling, model-based ANN's, statistical learning, eigen structure based processing, and, optical and superconducting signal

1. **Block diagram**

From the optimized ANN model, we designed the CMOS circuit design based on the operational amplifier to design the base cell called perceptron. This cell is commonly used in multi-layer perceptron (MLP) architectures. Figure 1 shows the block diagram of the proposed methodology for the circuit development. Block 1 represents the numerical analysis stage and the simulation of the optimization of the objective function (obtaining synaptic weights and bias) for the ANN, based on gradient. Block 2 focuses on the design of the base neuron (perceptron) at the circuit level. Finally, block 3 shows the development and simulation of the complete MLP circuit to obtain the neural network behavior proposed in the case study.

**A picture containing text, line, diagram, font

Description automatically generated**

Figure 1. Block diagram of the design and development of the proposed methodolo

All the optimization algorithms described above employ a system of first-order differential equations. Now, we need to apply optimization to a system of second-order differential equations. Therefore, to improve convergence properties, we can use a system of higher-order ordinary differential equations by considering the system of second-order differential equations (1), (3) and (8) as follows:

**A picture containing text, screenshot, font, document

Description automatically generated**

The system ofsecond-orderequationsisinspiredbyclassicalmechanicsandhasthefollowing physical interpretation [6]: (13) represents Newton’s second law (mass × acceleration = force) for a mass particle δ(t) moving in a space Rn subject to a force −∇E(x) givenbythepotentialE(x)andwithforce−γ(t)T(dx).Sinceγ(t)>0,theforce−γ(t)T(dx)is

Dissipative and γ(t) is the friction coefficient. Generally, the mass coefficient of δ(t) and the coefficient of friction γ(t) are constant in time or tend to zero as time approaches infinity. The application of a system of second-order differential equations has several impor-

There are many advantages over a system of first-order differential equations, including:

1. Because of the initial force, the local minimum of the objective function E(x) can be avoided by an appropriate choice of parameters, and the network can find an overall minimum, although this cannot be guaranteed.

2. Second order differential equations have better flexibility.For example, for the sa starting point x(0) different from the selection of (dx)(0), they can lead to adifferentlocalminimum.Thatis,wechangedcoefficientsγandδmakingitpossibleto reach a local minimum from the same initial condition (x(0), x′(0)). Thus, in a system of differential equations of the form given by (13), an additional control of the

Solution is provided.

3. A system of second-order differential equations may have a better property of convergence than a system of first-order differential equations. As a result, its trajectory can be used to obtain responses.

To evaluate the gradient descent algorithm, we performed two simulations. In the first example, we have a single-variable objective function for which the determination of the local minimum depends on the initial condition of the variable. A two-variable objective function (which can fall into a saddle point or a local minimum) is described in the second example. This case can be avoided by an appropriate choice of the initial conditions of the variables. The results obtained from the algorithm and initial conditions are shown in Tables 1 and 2, respectively.

**A picture containing text, diagram, screenshot

Description automatically generated**

**A picture containing text, font, screenshot, receipt

Description automatically generated**

When a search for the local minimum of a particular objective function of two or more variables is performed, it is necessary to avoid falling into the saddle points caused by the initial conditions of the variables when calculating the solution. In Table 1, the results obtained for examples 1 and 2 show how we can improve an example's convergence properties objective function with one or two variables. In general, an n-dimensional objective function will work using the proposed system of higher-order ordinary differential equations. In the following section, we will show how this approach can be applied. We will do this through a case study for its application to an ANN implemented using analog systems and CMOS circuits.

1. **Circuit diagram**

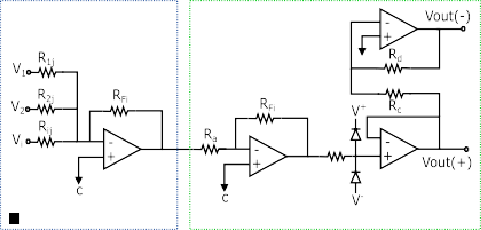
In this section, the application of the proposed system to a neural network [1,4,5] is described. An analog multiplier multiplies the outputs of a voltage adder concerning a constant. The voltage adder generates the sum of the voltages of the matrix *Wij*, which represents the synaptic weight matrix (Figure 2), and the signal of the non-linear function generator. We used operational amplifiers (OP-amps) to design each base neuron that composes the neural architecture. It means that we implemented the Figure 2 neural circuit structure to produce the ANN represented in Figure 3. We note that each neuron node has two parts: a sum function and an activation function (sigmoid). We implemented the first one with an op-amp inverting adder (designed based on Figure 4) and the second one with an array of op-amps and voltage limiters (diodes).

Figure 2 shows the block diagram for the function. It consists of two continuous-time integrators (whose response depends on the feedback network), one analog mul- tiplier, one summing amplifier, and one non-linear function generator for calculating the gradient of the objective function at the circuit level. The optimized parameters *x*(*t*) = [*x*1(*t*), *x*2(*t*), . . . , *xn*(*t*)]*T* are the output signals of the integrator. This circuit is characterized by having a more robust output (insensitive to small perturbations) with respect to parameter variation. The function generator is the only one capable of calculating precisely the gradient of the objective function.

A picture containing diagram, plan, technical drawing, schematic

Description automatically generated

**Figure 2.** Block diagram for applications to ANNs.



**Figure 3.** Complete circuit based on ANN.

A picture containing diagram, plan, technical drawing, text

Description automatically generated

**Figure 4.** Operational amplifier 1.2 µm.

Figure 2 shows the design of the analog neural network, which includes the development of the integrator circuit. As shown in Figure 5, the integrator circuit uses the OP-amp shown in Figure 6 (the block diagram of the 1.2 µm technology operational amplifier in Figure 4). The OP-amp is designed according to Table 3 specifications

-

+

**Figure 5.** Integrator Circuit.

A picture containing black, darkness

Description automatically generated

**Figure 6.** Block diagram of the 1.2 µm operational amplifier.

**Table 3.** Parameters of CMOS op-amp.

|  |  |
| --- | --- |
| **Parameter** | **Value** |
| Open-loop voltage gain | 60 dB |
| Phase margin (*φo* ) | 81◦ |
| Unity gain bandwidth (GB) | 2.30 MHz |
| Slew rate (SR) | 21.06 V/µs |
| CMRR | 47 dB |
| Offset voltage (Vos) | 284 µV |
| Compensation capacitance (CC) | 8.75 pF |
| Capacitive load (CL) | 20 pF |
| Power supply (VDD, VSS)  Power consumption | +/−2.5 V  1.28 mW |

Common mode input voltage range −608.696 mV to 1.98 V

Figure 6 shows the block diagram of the 1.2 µm technology operational amplifier in Figure 4. It shows the operational amplifier with CMOS transistors, formed by the stages of a differential amplifier (M1, M2, M4, and M6), the gain stage (M7, M91, M101, and M8) and the output stage (buffer) (M9 and M10). Compensation network M13 and capacitor Cc, where M13 is made up of a polarization network made up of transistors M14, M15, and M16. The dimensions of the main semiconductors N-channel and P-channel transistors are listed in Table 4. Table 5 describes the integrator's characteristic values [18,19].

The complete circuit of an analog neural network is shown in Figure 3. In this circuit, the inputs Xi, the weights (defined by the resistors Rij = wij), and the activation functions (Σ and F) are represented. The basic circuits that correspond to analog neural network neurons are the inverting amplifiers and the activation functions. The inverting amplifier

The circuit acts as a summation unit. Hence, when many input voltages are connected to the inverting input terminal, the resulting output is the sum of all the input voltages applied, although inverted; this output, combined with the feedback resistor, generates multiplication by a weight. The circuit for implementing a neuron is shown in Figure 7, where the function Σ computes multiplications and the activation function is a Sigmoid [19].

**Table 4.** Widths and lengths for N-channel and P-channel MOS transistors.

**2 µm Technology 1.2 µm Technology**

**N-Channel MOS P-Channel MOS**

**Transistor Width Length Width Length**

Differential Stage

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| M1 | 15 µm | 5 µm | 9 µm | 3 µm |
| M2 | 15 µm | 5 µm | 9 µm | 3 µm |
| M3 | 70 µm | 5 µm | 42 µm | 3 µm |
| M4 | 70 µm | 5 µm | 42 µm | 3 µm |
| M6 | 30 µm | 5 µm | 18 µm | 3 µm |
| M16 | 15 µm | 5 µm | 9 µm | 3 µm |
| M14 | 70 µm | 5 µm | 42 µm | 3 µm |
| M15 | 70 µm | 5 µm | 42 µm | 3 µm |
| Cascode Stage | | | | |
| M13 | 70 µm | 5 µm | 42 µm | 3 µm |
| M7 | 70 µm | 5 µm | 42 µm | 3 µm |
| M91 | 15 µm | 2 µm | 9 µm | 1.2 µm |
| M101 | 70 µm | 2 µm | 42 µm | 1.2 µm |
| M8 | 15 µm | 5 µm | 9 µm | 3 µm |
| Output Stage | | | | |
| M9 | 150 µm | 2 µm | 90 µm | 1.2 µm |
| M10 | 700 µm | 2 µm | 420 µm | 1.2 µm |

**Table 5.** Characteristic Values of the CMOS integrator circuit.

|  |  |
| --- | --- |
| **Characteristic** | **Values** |
| Slew rate | 21.06 V/µs |
| Cutoff frequency | 2.2570 MHz |
| Gain | 60 dB |
| Phase Margin | 83.782◦ |
| A1 gain | 33.33 |
| A2 gain | 55.55 |

**A picture containing diagram, plan, line, technical drawing

Description automatically generated**

Figure 7. Circuit for one neuron.

As shown in Figure 3, the architecture has 9 neurons (conﬁguration 5-3-1), and each neuron has 4 op-amps due to the adder and the sigmoid activation function (see Table 6).

The total power consumption is 46.08 mW, because there are 36 op-amps for the entire circuit, and each of them consumes 1.28 mW [18,20,21].

**Table 6.** Number of op-amps and total power consumption.

|  |  |
| --- | --- |
| **Description** | **Quantity** |
| No. of op amps per adder | 1 |
| No. of op amps per activation function | 3 |
| No. of op amps per neuron | 4 |
| No. of neurons for the 5-3-1 architecture | 9 |
| No. of op amps for the 5-3-1 ANN circuit | 36 |
| Total power consumption | 46.08 mW |

1. **task implementations**

The ANN-based architecture of this work (see Figure 3) is proposed for pattern recognition as an application example, where the recognition task consists of detecting a value of +1 or −1 for Hamming code correction. One of the advantages of the proposed circuit is that It can be integrated as a module in some communication systems in application-speciﬁc Integrated circuits (ASIC).

The electronic simulation of a backpropagation neural network is implemented on the basic circuits designed for a typical neuron (the weighted sum function and the activation function). We simulate the circuit using the Spice program [18], which allows changing the device parameters and stimuli so that different operating conditions are explored.

Once we model the behavior of the cells that make up backpropagation, the complete circuit simulation to study network performance can continue. This is

It is accomplished by connecting the synapses and the activation functions according to Figure 3. The input vector is bipolar (see Table 7).

Figure 8 represents the neural network response designed to classify input patterns with a supervised learning algorithm) to different stimuli, with a sweep of the output signal concerning the input signal. Figure 8a represents a stimulus P = [−1 1 1 1 1] and the corresponding output T = [1]; if we observe from left to right, we have a negative input and a positive posterior one, which is consistent with the input pattern and its positive output:

1. Figure 8b shows the behavior of the architecture with a stimulus P = [−1 −1 −1 −1 −1] and an output T = [1]; if we sweep the input signal from left to right, we can observe that all the input patterns are negative and we have a positive output at 1. For Figure 8 we have an input pattern P = [1 −1 −1 −1 −1] and an output T = [−1], sweeping from left to right the input pattern is negative and then positive; where the ﬁrst position of the input vector will be the least signiﬁcant one for the response of the architecture, resulting in a negative output. Finally, for Figure 8d we have an input pattern P = [1 −1 1 1 1] and an output T = [−1]; if we sweep the output signal from left to right, we observe that the negative value of the pattern is not preponderant for the output of the architecture, which in this case is negative.The proposed methodology is based on the problem analysis, the datasets deﬁnition(inputs and outputs) for a supervised method, the evaluation of ANN conﬁgurations, the selection of the best conﬁguration, the implementation of the conﬁguration through operational ampliﬁers, and the simulation of the ANN-based circuit. In this work, a detection circuit is implemented. The outline training is done through Mathematica (i.e., the training, in which weights and biases are estimated, is a software-based step carried out prior to the simulation of the ANN-based circuit), and the ANN conﬁguration is carried out using OP-amps.

**Table 7.** Bipolar input vector.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | | **Input Vector (P)** |  |  | **Target (T)** |
| −1 | −1 | −1 | −1 | −1 | 1  1 |

−1 −1 −1 −1 1

−1 −1 −1 1 −1 1

−1 −1 −1 1 1 1

−1 −1 1 −1 −1 1

−1 −1 1 −1 1 1

−1 −1 1 1 −1 1

−1 −1 1 1 1 1

−1 1 −1 −1 −1 1

−1 1 −1 −1 1 1

−1 1 −1 1 −1 1

−1 1 −1 1 1 1

−1 1 1 −1 −1 1

−1 1 1 −1 1 1

−1 1 1 1 −1 1

−1 1 1 1 1 1

1 −1 −1 −1 −1 −1

1 −1 −1 −1 1 −1

1 −1 −1 1 −1 −1

1 −1 −1 1 1 −1

1 −1 1 −1 −1 −1

1 −1 1 −1 1 −1

1 −1 1 1 −1 −1

1 −1 1 1 1 −1

1 1 −1 −1 −1 −1

1 1 −1 −1 1 −1

1 1 −1 1 −1 −1

1 1 −1 1 1 −1

1 1 1 −1 −1 −1

1 1 1 −1 1 −1

1 1 1 1 −1 −1

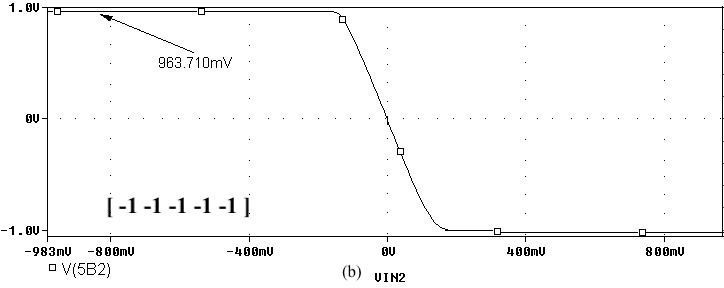
1 1 1 1 1 −1

In [22,23], the authors presented a circuit-level implementation of the backpropagation learning algorithm, exploring the gradient descent method in analog circuit design. They used an ANN in memory crossbar array, simulating in SPICE on TSMC’s 180 nm CMOS technology, reporting output voltages and timing behaviors.

In our case, we presented an analog system simulation based on the solution to opti- mization problems with objective functions of one and two variables using the gradient descent method (ascending slope). ANN configuration behavior (learning algorithm) is shown, where input and processing signals are analog. Our circuit-level implementation was made with a 5-3-1 configuration using the backpropagation training algorithm. We reported the simulation-level architecture results in PSpice level 9.1 on 1.2 µm technology with a lambda of 0.5 µm, with X-channel widths and lengths defined by the technology. Although the learning of the neural network was developed offline, and we consider the optimization results obtained in the mathematical part of the article were satisfactory, we emphasize that its essential purpose was to illustrate the applicability to optimize the learning algorithm of an ANN. We also simulated the optimized ANN model through the design of a circuit with CMOS transistors.

A picture containing text, line, plot, diagram

Description automatically generated



**Figure 8.** Response of the neural network to different stimulus. The stimulus means external inputs to the network. (**a**) P = [−1 1 1 1 1]; T = [1], (**b**) P = [−1 −1 −1 −1 −1]; T = [1], (**c**) P = [1 −1 −1 −1

−1]; T = [−1], and (**d**) P = [1 −1 1 1 1]; T = [−1].

1. **Conclusion**

Neural networks are a powerful tool for signal processing applications. They are well-suited to non-linear and time-varying problems, and they can be used for a variety of tasks, including pattern recognition, signal filtering, and system modeling. The application of neural networks to signal processing requires careful design and consideration of the specific problem. However, the potential benefits of using neural networks can be significant, and they are a valuable tool for many signal processing applications.

Here are some additional details from the article that support the conclusion:

* Neural networks are able to learn and adapt to new data, which makes them well-suited for applications where the data is not static.
* Neural networks are able to model complex relationships between data, which makes them useful for tasks such as pattern recognition and signal filtering.
* Neural networks are able to be implemented in hardware, which makes them suitable for real-time applications.

Overall, neural networks are a powerful tool for signal processing applications. They are well-suited to non-linear and time-varying problems, and they can be used for a variety of tasks. The application of neural networks to signal processing requires careful design and consideration of the specific problem. However, the potential benefits of using neural networks can be significant, and they are a valuable tool for many signal processing applications.

**REFRENCES**

1. **D. Ruan (ed.), Fuzzy Systems and Soft Computing in Nuclear Engineering © Springer-Verlag Berlin Heidelberg 2000**
2. **Alejandro Medina-Santiago , Carlos Arturo Hernández-Gracidas , Luis Alberto Morales-Rosales**

**Ignacio Algredo-Badillo , Monica Amador García and Jorge Antonio Orozco Torres “CMOS Implementation of ANNs Based on Analog Optimization of N-Dimensional Objective Functions”** **Department of Computer Science, CONACYT-INAOE (Instituto Nacional de Astrofísica, Óptica y**

**Electrónica) may 2021**

**file:///E:/studying/2022-2023/sem2/pattern/sensors-21-07071-v3.pdf**

1. **Rekha Govil, “Neural Networks in Signal Processing”Department of Computer Science & Electronics Banasthali Vidyapith - 2022, India**

<https://link.springer.com/content/pdf/10.1007/978-3-7908-1866-6_11.pdf>

**Task title**

**Applications of ANN with speech recognition**

**Chapter 1 : introduction**

**the objective and importance of speech recognition system:-**

The objective of speech recognition is to enable machines to understand and interpret human speech. Speech recognition technology has many important applications, including:

1. Accessibility: Speech recognition can make it easier for people with disabilities, such as those with visual impairments or mobility impairments, to interact with computers and mobile devices.
2. Human-machine interaction: Speech recognition can enable more natural and intuitive interactions between humans and machines, such as voice-activated assistants, smart speakers, and voice-controlled appliances.
3. Automation: Speech recognition can be used to automate tasks that would otherwise require manual input, such as transcribing audio recordings, generating captions for video content, or converting spoken commands into text.
4. Language translation: Speech recognition technology can be used to automatically translate speech from one language to another, enabling cross-lingual communication.
5. Medical applications: Speech recognition can be used in medical applications, such as transcribing medical dictation or assisting people with speech impairments.
6. Security: Speech recognition can be used for biometric authentication, enabling secure and convenient access to devices and systems.

The importance of speech recognition lies in its ability to bridge the gap between human communication and machine processing. By enabling machines to understand and interpret human speech, speech recognition technology has the potential to revolutionize how we interact with computers, devices, and systems, and to make our lives easier, more efficient, and more accessible.

**the used tools either software sw or hardware HW to achieve the task:-**

There are a variety of tools and technologies used for speech recognition, both hardware and software-based. Here are some examples:

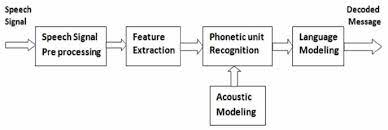
1. Microphones: Speech recognition typically starts with capturing the speech signal using a microphone. There are many types of microphones available, including built-in microphones on devices like smartphones and laptops, as well as external microphones designed for use in specific settings, such as conference rooms or recording studios.
2. Analog-to-digital converters (ADCs): The analog speech signal captured by the microphone needs to be converted into a digital format that can be processed by a computer. ADCs are used to perform this conversion.
3. Speech recognition software: There are many software tools available for speech recognition, ranging from standalone applications to cloud-based services. Some popular examples include Google Cloud Speech-to-Text, Amazon Transcribe, and Microsoft Azure Speech Services.
4. Automatic speech recognition (ASR) engines: ASR engines are software programs that use machine learning algorithms to recognize speech and convert it into text. Some popular ASR engines include Kaldi, Sphinx, and the Google Speech API.
5. Digital signal processors (DSPs): DSPs are specialized microprocessors designed for processing digital signals, such as speech. They are often used in hardware-based speech recognition systems, such as those used in automobiles or mobile devices.
6. Graphics processing units (GPUs): GPUs are specialized hardware devices designed for parallel processing. They are often used in deep learning-based speech recognition systems to accelerate the training and inference processes.

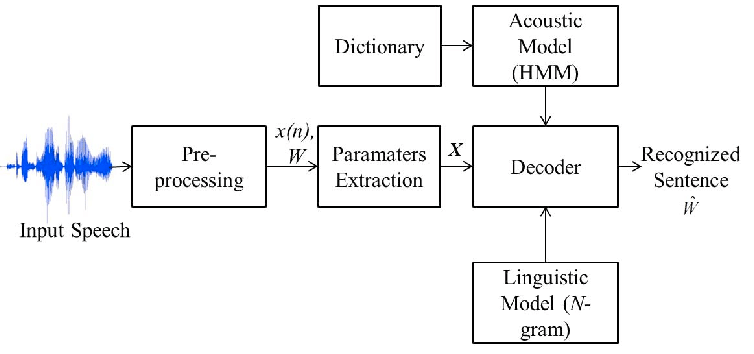
Overall, the choice of tools and technologies for speech recognition depends on the specific application and requirements of the system.

**Chapter 2**

**Block diagram**

**Here's a block diagram that shows the different stages and phases of a typical speech recognition system:-**

****

****

Speech feature extraction is the process of transforming the raw speech signal into a set of features that can be used in subsequent processing stages, such as speech recognition or speaker identification. There are several mathematical models used for speech analysis, but the most commonly used ones are based on the Fourier transform and the short-time Fourier transform.

The Fourier transform is a mathematical technique that decomposes a signal into its constituent frequency components. In speech analysis, the Fourier transform is used to extract spectral information from the speech signal. The resulting spectrum represents the power distribution of the speech signal as a function of frequency, and can be used to identify important features of the speech signal, such as formants, which correspond to the resonant frequencies of the vocal tract.

The short-time Fourier transform (STFT) is a modified version of the Fourier transform that is used to analyze non-stationary signals, such as speech. The STFT divides the speech signal into short overlapping segments, and computes the Fourier transform of each segment. The resulting spectrogram provides a time-frequency representation of the speech signal, which can be used to extract time-varying spectral features, such as the fundamental frequency (pitch) and spectral envelope.

Other mathematical models used for speech analysis include linear predictive coding (LPC), which models the speech signal as a linear combination of its past values, and cepstral analysis, which transforms the speech spectrum into the frequency domain of the logarithm of the spectrum.

Overall, speech feature extraction plays a crucial role in speech processing and analysis, and the choice of mathematical models depends on the specific application and the desired features to be extracted.

**The suitable model for speech recognition :-**

One commonly used model for speech feature extraction is the Mel-Frequency Cepstral Coefficients (MFCC) model. The MFCC model involves several steps, which can be summarized as follows:

1. **Preprocessing**: The speech signal is preprocessed by applying a pre-emphasis filter to boost high-frequency components and reduce low-frequency noise.
2. **Framing**: The preprocessed signal is divided into short overlapping frames of typically 20-30 milliseconds in duration.
3. **Windowing**: Each frame is windowed using a window function, such as the Hamming window, to reduce spectral leakage.
4. **Fourier Transform:** The short-time Fourier transform (STFT) is applied to each frame to obtain its power spectrum.
5. **Mel Filterbank**: The power spectrum is then passed through a bank of Mel-scale filters to extract the spectral envelope of the speech signal.
6. **Logarithm**: The logarithm of the filterbank energies is computed to compress the dynamic range of the spectrum.
7. **Discrete Cosine Transform:** The resulting log-filterbank energies are then transformed using the discrete cosine transform (DCT) to obtain the MFCCs.

**Ch-3Circuit diagram**

**the basic definitions of speech recognition system:-**

here are some basic definitions of speech recognition system:

**Speech Recognition**: The process of converting spoken words into written text or commands using computer algorithms.

Audio Input: The raw sound signal captured by a microphone or other recording device.

**Preprocessing**: The preliminary signal processing techniques used to clean up the audio input and remove noise or distortion that may interfere with the recognition process.

**Feature Extraction**: The process of identifying and extracting meaningful features from the preprocessed audio signal, such as Mel-Frequency Cepstral Coefficients (MFCCs), Linear Predictive Coding (LPC), or other spectral analysis techniques.

**Acoustic Model**: A statistical model that maps the extracted features to phonemes or other basic speech units using techniques such as Hidden Markov Models (HMMs) or Deep Neural Networks (DNNs).

**Language Model**: A statistical model that interprets the phoneme sequence and generates the most likely sequence of words using context and probability-based techniques.

**Decoding**: The process of combining the output of the acoustic and language models to determine the most likely sequence of words that corresponds to the input speech signal.

**Output Text:** The recognized text that is output by the speech recognition system and can be displayed on a screen or output to a speech synthesizer.

Overall, a speech recognition system uses a combination of signal processing, statistical modeling, and machine learning techniques to convert spoken words into written text or commands. The accuracy of the system depends on the quality of the audio input, the effectiveness of the preprocessing and feature extraction techniques, and the accuracy of the acoustic and language models used in the recognition process.

**the number of layers of used ANN:-**

The number of layers in an Artificial Neural Network (ANN) used for speech recognition can vary depending on the specific architecture and task. However, there are some common architectures that are often used for speech recognition, such as deep feedforward neural networks (FFNNs), convolutional neural networks (CNNs), and recurrent neural networks (RNNs).

Deep FFNNs typically consist of multiple hidden layers and have been used for speech recognition tasks such as phoneme classification and speaker identification. The number of hidden layers in a deep FFNN can range from a few layers to dozens of layers, depending on the complexity of the task and the size of the dataset.

CNNs are often used for speech recognition tasks that involve analyzing spectrograms or other types of time-frequency representations of speech signals. CNNs typically consist of multiple convolutional layers followed by one or more fully connected layers. The number of convolutional layers in a CNN can range from a few layers to dozens of layers, again depending on the complexity of the task and the size of the dataset.

RNNs are commonly used in speech recognition tasks that involve modeling the temporal dynamics of speech signals. RNNs can have multiple layers of recurrent units, such as Long Short-Term Memory (LSTM) or Gated Recurrent Unit (GRU) cells. The number of recurrent layers in an RNN can range from a few layers to several layers, depending on the complexity of the task and the size of the dataset.

In summary, the number of layers in an ANN used for speech recognition can vary depending on the specific architecture and task, but it is common for deep architectures to have multiple layers, ranging from a few layers to dozens of layers.

**Ch-4 task implementations:**

**Here is an algorithm for computing MFCCs:-**

Input: Speech signal s, sampling rate fs

Output: MFCC feature vector

**1. Apply pre-emphasis filter to s**

**2. Divide s into overlapping frames of duration T**

**3. Apply a window function to each frame**

**4. Compute the magnitude spectrum using STFT**

**5. Apply a Mel filterbank to the spectrum**

**6. Take the logarithm of the filterbank energies**

**7. Apply DCT to the log-filterbank energies**

**8. Keep the first N coefficients as the MFCC feature vector**

To verify this algorithm, we can implement it in a high-level programming language such as Python and use it to extract MFCC features from a speech signal. We can then use the resulting feature vectors for speech recognition or other speech processing tasks.

Pattern features refer to the distinctive characteristics of a signal or pattern that can be used to identify or classify it. In the context of speech processing, pattern features can include spectral features such as formants, pitch, and MFCCs, as well as temporal features such as speech rate and duration. These features can be used to recognize speech sounds, identify speakers, or detect emotions in speech signals.

**The implementations with python:-**

import numpy as np

from scipy.io import wavfile

from scipy.fftpack import dct

# Define functions for computing the pre-emphasis filter and Mel filterbank

def pre\_emphasis(signal, alpha=0.97):

return np.append(signal[0], signal[1:] - alpha \* signal[:-1])

def mel\_filterbank(nfilt, nfft, fs):

# Compute the Mel scale

low\_freq\_mel = 0

high\_freq\_mel = (2595 \* np.log10(1 + (fs / 2) / 700))

mel\_points = np.linspace(low\_freq\_mel, high\_freq\_mel, nfilt + 2)

hz\_points = (700 \* (10\*\*(mel\_points / 2595) - 1))

bin\_idx = np.floor((nfft + 1) \* hz\_points / fs)

fbank = np.zeros((nfilt, int(np.floor(nfft / 2 + 1))))

for m in range(1, nfilt + 1):

f\_m\_minus = int(bin\_idx[m - 1])

f\_m = int(bin\_idx[m])

f\_m\_plus = int(bin\_idx[m + 1])

for k in range(f\_m\_minus, f\_m):

fbank[m - 1, k] = (k - bin\_idx[m - 1]) / (bin\_idx[m] - bin\_idx[m - 1])

for k in range(f\_m, f\_m\_plus):

fbank[m - 1, k] = (bin\_idx[m + 1] - k) / (bin\_idx[m + 1] - bin\_idx[m])

return fbank

# Define parameters for MFCC computation

n\_mfcc = 13

n\_fft = 2048

hop\_length = int(n\_fft / 2)

n\_filt = 40

# Load the audio file and compute the MFCCs

fs, signal = wavfile.read("audio\_file.wav")

# Step 1: Apply pre-emphasis filter

signal = pre\_emphasis(signal)

# Step 2: Divide signal into overlapping frames

frames = np.lib.stride\_tricks.as\_strided(signal, shape=(len(signal) - n\_fft + 1, n\_fft), strides=(signal.strides[0], signal.strides[0]))

# Step 3: Apply window function to each frame

window = np.hamming(n\_fft)

frames = frames \* window

# Step 4: Compute magnitude spectrum using STFT

mag\_spec = np.abs(np.fft.rfft(frames, n\_fft))

# Step 5: Apply Mel filterbank

mel\_fbank = mel\_filterbank(n\_filt, n\_fft, fs)

mel\_spec = np.dot(mag\_spec, mel\_fbank.T)

# Step 6: Take the logarithm of the filterbank energies

log\_mel\_spec = np.log10(mel\_spec)

# Step 7: Apply DCT to the log-filterbank energies

mfcc = dct(log\_mel\_spec, type=2, axis=1, norm='ortho')

# Step 8: Keep the first N coefficients as the MFCC feature vector

mfcc = mfcc[:, 1:(n\_mfcc + 1)]

# Print the MFCC feature vector

print(mfcc)

Note that this code assumes that the speech signal is stored in a WAV file named "audio\_file.wav" in the same directory as the Python script. You may need to modify the code to read the audio signal from a different source, such as a microphone or a different file format.

**Ch5 Conclusion:**

**the different mathematical model for speech Analysis:-**

There are several mathematical models used for speech analysis, some of which are described below:

1. Fourier Transform (FT): The Fourier transform is a mathematical technique used to analyze signals in the frequency domain. In speech analysis, the Fourier transform can be used to transform a speech signal from the time domain to the frequency domain, allowing for analysis of the spectral content of the signal.
2. Short-time Fourier Transform (STFT): The STFT is a modification of the Fourier transform that allows for analysis of the frequency content of a signal over short time intervals. In speech analysis, the STFT is often used to analyze the spectral content of speech frames, which are short segments of speech signals.
3. Linear Predictive Coding (LPC): LPC is a mathematical technique used to model the spectral envelope of a speech signal. LPC models the speech signal as a linear combination of past speech samples, and uses the resulting coefficients to estimate the spectral envelope of the signal.
4. Mel-Frequency Cepstral Coefficients (MFCCs): MFCCs are a feature extraction technique used in speech analysis. MFCCs are computed by taking the logarithm of the magnitude of the Mel filterbank energies computed from the STFT of a speech signal, followed by a Discrete Cosine Transform (DCT) applied to the resulting coefficients.
5. Hidden Markov Models (HMMs): HMMs are a statistical model used to represent the temporal dynamics of speech signals. In speech analysis, HMMs are often used for speech recognition tasks by modeling the probability distribution of speech sounds and using the resulting model to recognize speech.
6. Deep Neural Networks (DNNs): DNNs are a type of artificial neural network used for speech analysis tasks such as speech recognition and speaker identification. DNNs consist of multiple layers of interconnected neurons and are trained on large datasets of speech signals to learn patterns in the data.

These are just a few examples of the mathematical models used in speech analysis. The choice of model depends on the specific task and the characteristics of the speech data, and often involves a combination of different models to achieve the desired analysis results.

In conclusion, speech recognition using Artificial Neural Networks (ANNs) is a rapidly advancing field with many promising applications. ANNs have been successfully applied to a variety of speech recognition tasks, including speech-to-text transcription, speaker identification, and emotion recognition.

One of the key advantages of ANNs for speech recognition is their ability to learn complex patterns in the speech data, which allows them to achieve high levels of accuracy and robustness across different speakers, accents, and environmental conditions. Deep neural networks, in particular, have been shown to be highly effective for speech recognition tasks, due to their ability to learn hierarchical representations of the speech data.

However, speech recognition using ANNs also presents several challenges, such as the need for large amounts of labeled training data, the difficulty of handling speech variability, and the computational resources required for training and inference. Addressing these challenges will be important for further advancing the field of speech recognition using ANNs.

Despite these challenges, speech recognition using ANNs holds great promise for improving human-computer interaction, enabling new applications in fields such as healthcare, education, and entertainment, and enhancing accessibility for people with disabilities. As ANNs continue to evolve and improve, we can expect to see continued progress in the field of speech recognition and its many applications.

**REFRENCES**

1. "Speech and Language Processing" by Daniel Jurafsky and James H. Martin. This textbook provides a comprehensive introduction to natural language processing and covers topics such as speech recognition, language modeling, and deep learning for NLP.
2. "Deep Learning for Speech Recognition" by Joseph Keshet. This book provides an overview of deep learning techniques for speech recognition, including convolutional neural networks, recurrent neural networks, and attention-based models.
3. "Automatic Speech Recognition: A Deep Learning Approach" by Dong Yu and Li Deng. This book provides an in-depth description of deep learning techniques for speech recognition, including hybrid DNN-HMM systems, sequence discriminative training, and unsupervised pre-training.
4. "Speech Recognition with Deep Recurrent Neural Networks" by Alex Graves and Navdeep Jaitly. This paper introduces a deep recurrent neural network architecture for speech recognition that includes both forward and backward connections, as well as a variational inference approach for modeling uncertainty.
5. "Connectionist Temporal Classification: Labelling Unsegmented Sequence Data with Recurrent Neural Networks" by Alex Graves, Santiago Fernandez, Faustino Gomez, and Jurgen Schmidhuber. This paper introduces the Connectionist Temporal Classification (CTC) algorithm, which is a popular method for training recurrent neural networks for speech recognition without requiring explicit alignment between the input and output sequences.
6. "End-to-End Speech Recognition with Recurrent Neural Networks" by Alex Graves and Navdeep Jaitly. This paper introduces an end-to-end approach to speech recognition using recurrent neural networks, which eliminates the need for separate acoustic and language models and can be trained using only speech transcription data.

**Task Name: Speaker Recognition**

**Ch1- introduction**

* **Objective**

**The objective of speaker recognition is to identify or verify the identity of an individual based on their voice. This is a biometric technology that uses the unique characteristics of a person's speech, such as their voice pitch, tone, accent, and pronunciation, to distinguish them from others.**

**Speaker recognition systems can be used in various applications, such as security systems, access control, and forensic investigations. For example, in security systems, a speaker recognition system can be used to grant access to a secure facility by verifying the identity of the person speaking. In forensic investigations, speaker recognition can be used to identify suspects based on recorded speech samples.**

**There are two main types of speaker recognition systems:**

**verification and identification.**

**Verification systems compare the voice of an individual against a pre-registered voiceprint to determine if they are the same person.**

**Identification systems, on the other hand, compare the voice of an individual against a database of voiceprints to determine their identity.**

* **Importance**

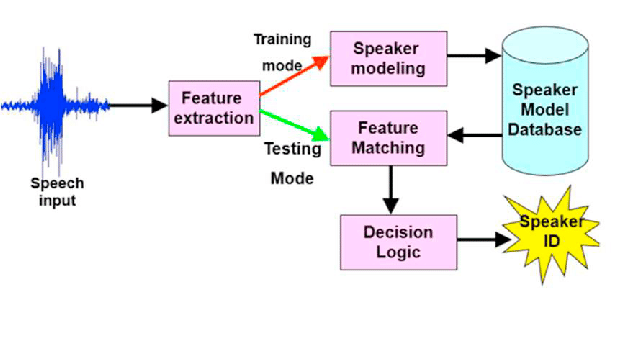
1. **Security: One of the main applications of speaker recognition is in security systems. By verifying the identity of individuals based on their voice, speaker recognition systems can help prevent unauthorized access to secure facilities and protect sensitive information.**
2. **Access Control: Speaker recognition can be used for access control in a variety of settings, such as office buildings, data centers, and hospitals. By verifying the identity of individuals before granting them access, speaker recognition can help ensure that only authorized personnel are allowed in.**
3. **Forensic Investigations: Speaker recognition is also used in forensic investigations to identify suspects based on recorded speech samples. This can be especially useful in cases where other forms of evidence are lacking or inconclusive.**
4. **Personalization: Speaker recognition can also be used for personalization in a variety of applications, such as smart homes and virtual assistants. By recognizing the voice of a specific individual, these systems can tailor their responses and actions to their preferences and needs.**
5. **Accessibility: Speaker recognition can also be used to make technology more accessible to individuals with disabilities. For example, speech recognition technology can be used to allow individuals with mobility impairments to interact with computers and other devices using their voice.**

* **The used tools**

**By using MATLAB, train three convolutional neural networks (CNNs) to perform speaker verification and then compare the performances of the architectures. The architectures of the three CNNs are all equivalent except for the first convolutional layer in each:**

1. **In the first architecture, the first convolutional layer is a "standard" convolutional layer, implemented using convolution2dLayer.**
2. **In the second architecture, the first convolutional layer is a constant sinc filter bank, implemented using a custom layer.**
3. **In the third architecture, the first convolutional layer is a trainable sinc filter bank, implemented using a custom layer. This architecture is referred to as *SincNet***

**Ch2-Block diagram**

****

**Fig. 1. Block Diagram of Speaker Recognition System**

**Fig.1 shows the block diagram of a speaker recognition system. The main blocks include feature extraction and speaker modelling.**

**The feature extraction process aims to extract a compact, efficient set of parameters that represent the acoustic properties observed from input speech signal, for subsequent utilization.**

**Actually, Feature extraction is the method of reducing the dimension of data of the speech signal while retaining the required information.**

**Speech signal contains several information of which not all are necessary for the identification of the speaker.**

**Ideal features must be robust against noise and distortion, occur frequently and naturally in speech, be easy to measure from speech signal, and be difficult to mimic etc.**

**The features extracted may be categorized into short term spectral features, voice source features, Spectro-temporal features, prosodic features etc.**

**Short term spectral features are extracted from speech signals by dividing them into short frames of 20-30 MS duration.**

**Voice source features make use of the features of the vocal tract. Here we make use of MFCC and IHC which are short term features and pitch and formants which are voice source features.**

**Ch-3 task implementations**

**Algorithm for Speaker Recognition using CNN and MFCCs**

**1. Load the audio dataset and extract MFCCs for each audio sample.**

**2. Split the dataset into training and testing sets.**

**3. Build a CNN architecture with several convolutional layers followed by one or more fully connected layers.**

**4. Compile the CNN model with an appropriate loss function and optimizer for classification.**

**5. Train the CNN using the training set and monitor the performance on the validation set.**

**6. Evaluate the trained CNN on the testing set and calculate the accuracy, precision, recall, and F1 score using the confusion matrix.**

**7. Fine-tune the CNN architecture and training parameters to improve performance if necessary.**

**8. Save the trained CNN model for future use.**

**The meaning of pattern features in speaker recognition refers to the extracted features from the audio signals that are used to identify different speakers. MFCCs are a common pattern feature used in speaker recognition, which involves breaking down the audio signal into small frames and applying a filter bank to each frame to extract relevant frequency components. The resulting MFCCs can be used as input to a machine learning model such as a CNN to identify different speakers based on their unique voice characteristics.**

* **MATLAB code**

**% Load audio dataset and extract MFCCs**

**audio\_data = load('audio\_dataset.mat');**

**mfccs = extract\_mfccs(audio\_data);**

**% Split dataset into training and testing sets**

**[train\_data,test\_data] = split\_data(mfccs,0.7);**

**% Build CNN architecture**

**model = build\_cnn();**

**% Compile the CNN model**

**model = compile\_cnn(model);**

**% Train the CNN model**

**model = train\_cnn(model,train\_data);**

**% Evaluate the trained CNN on the testing set**

**predictions = predict\_cnn(model,test\_data);**

**metrics = evaluate(predictions,test\_data);**

**% Fine-tune the CNN architecture and training parameters**

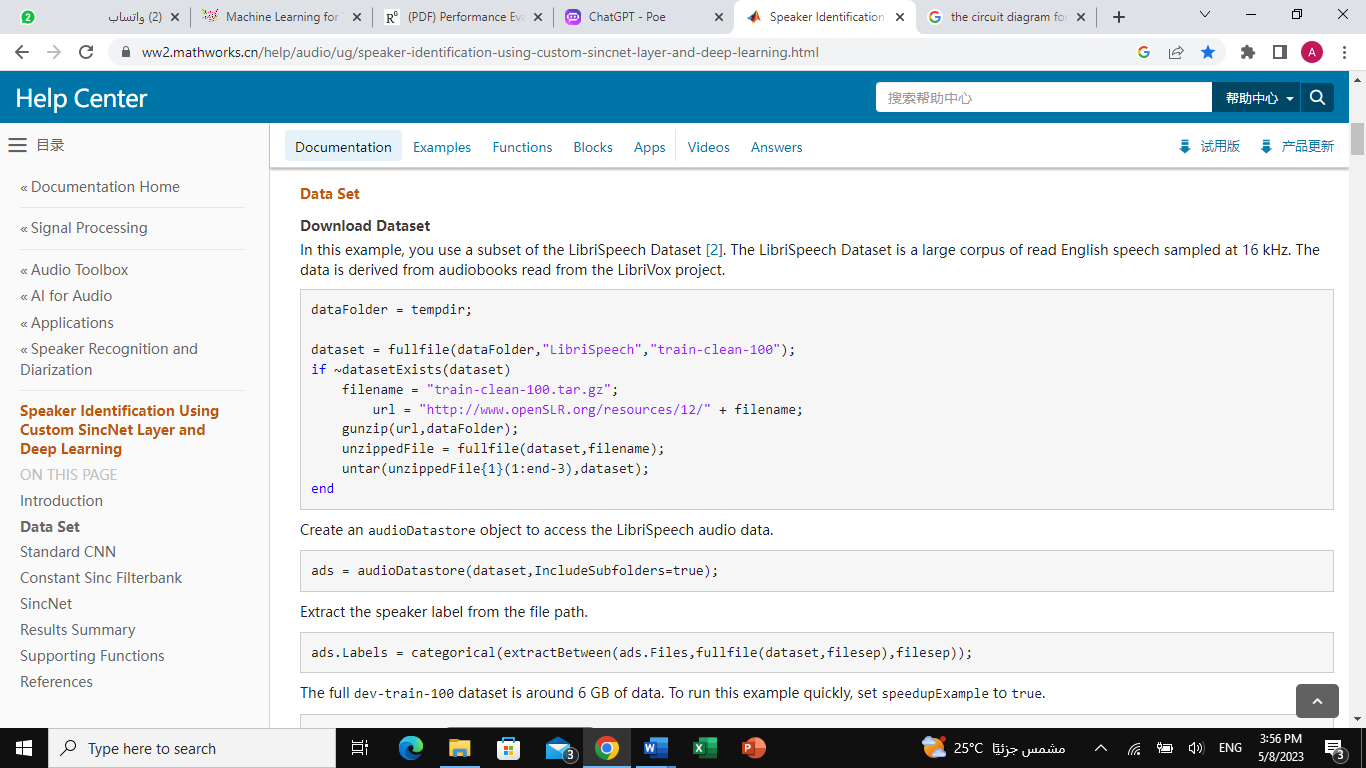
**model = fine\_tune\_cnn(model,train\_data,validation\_data);**

**% Save the trained CNN model**

**save('speaker\_recognition\_model.mat','model');**

**This is just a rough example of the code, and the specific implementation details may vary depending on the specific use case and dataset. The extract\_mfccs, split\_data, build\_cnn, compile\_cnn, train\_cnn, predict\_cnn, evaluate, and fine\_tune\_cnn functions would need to be defined and implemented separately. These functions would typically involve applying appropriate signal processing and machine learning techniques to preprocess the audio data, build and train the CNN model, and evaluate its performance.**

* **screens after implementation to show numerical results**

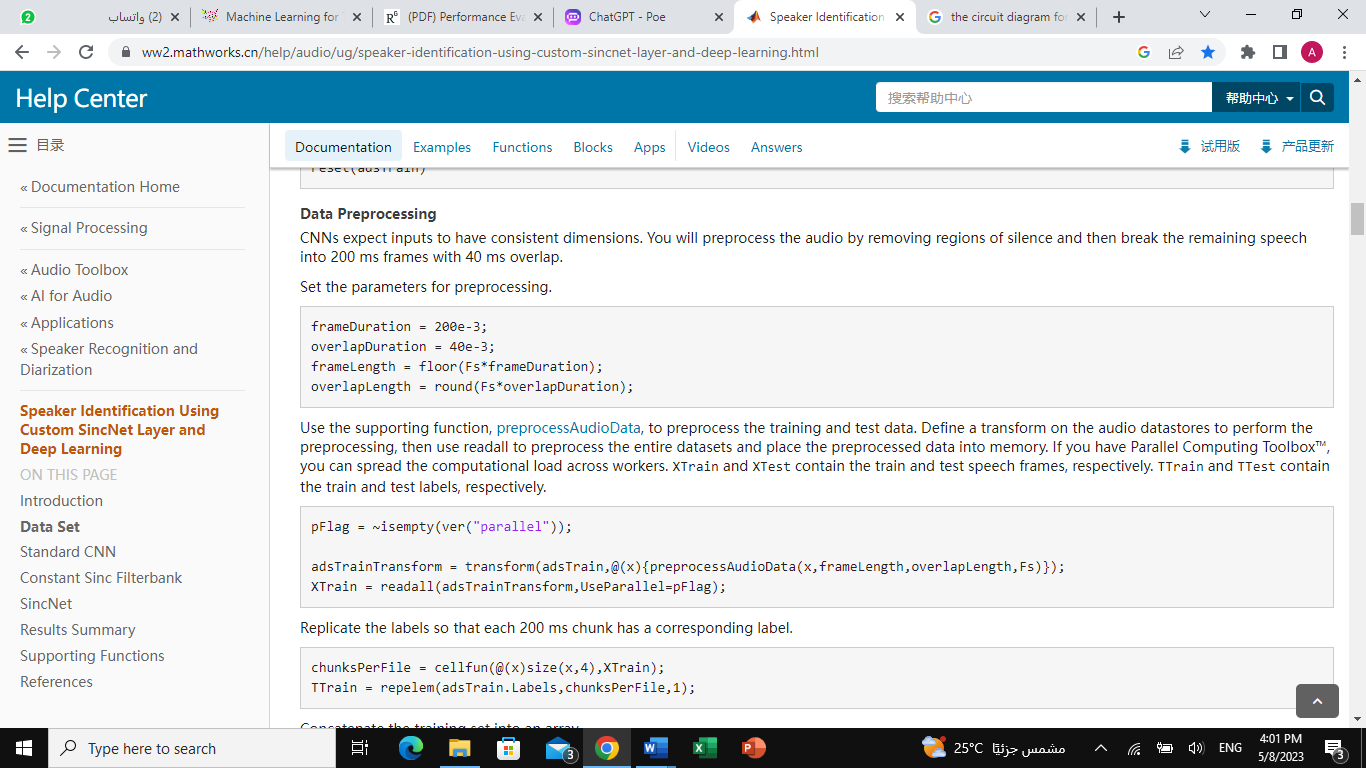


A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated



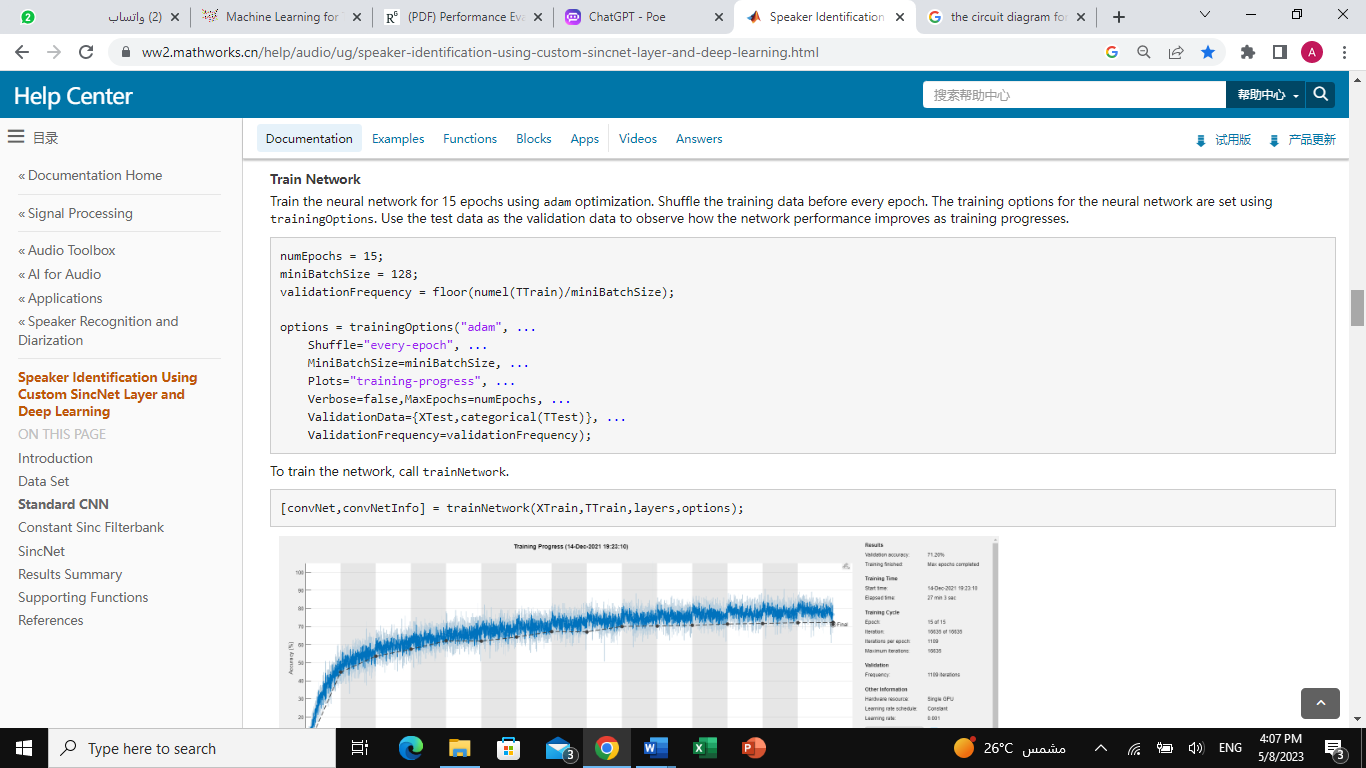
A screenshot of a computer

Description automatically generated



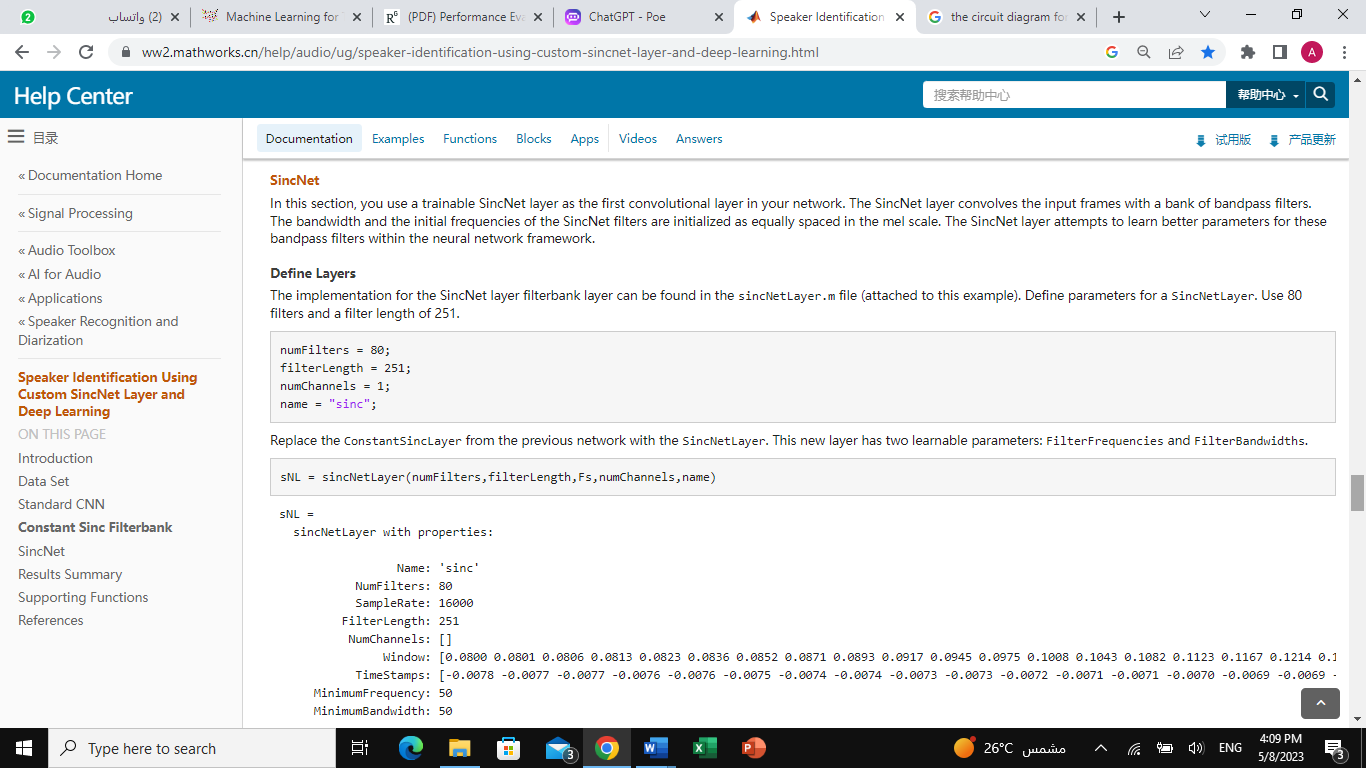
A screenshot of a computer

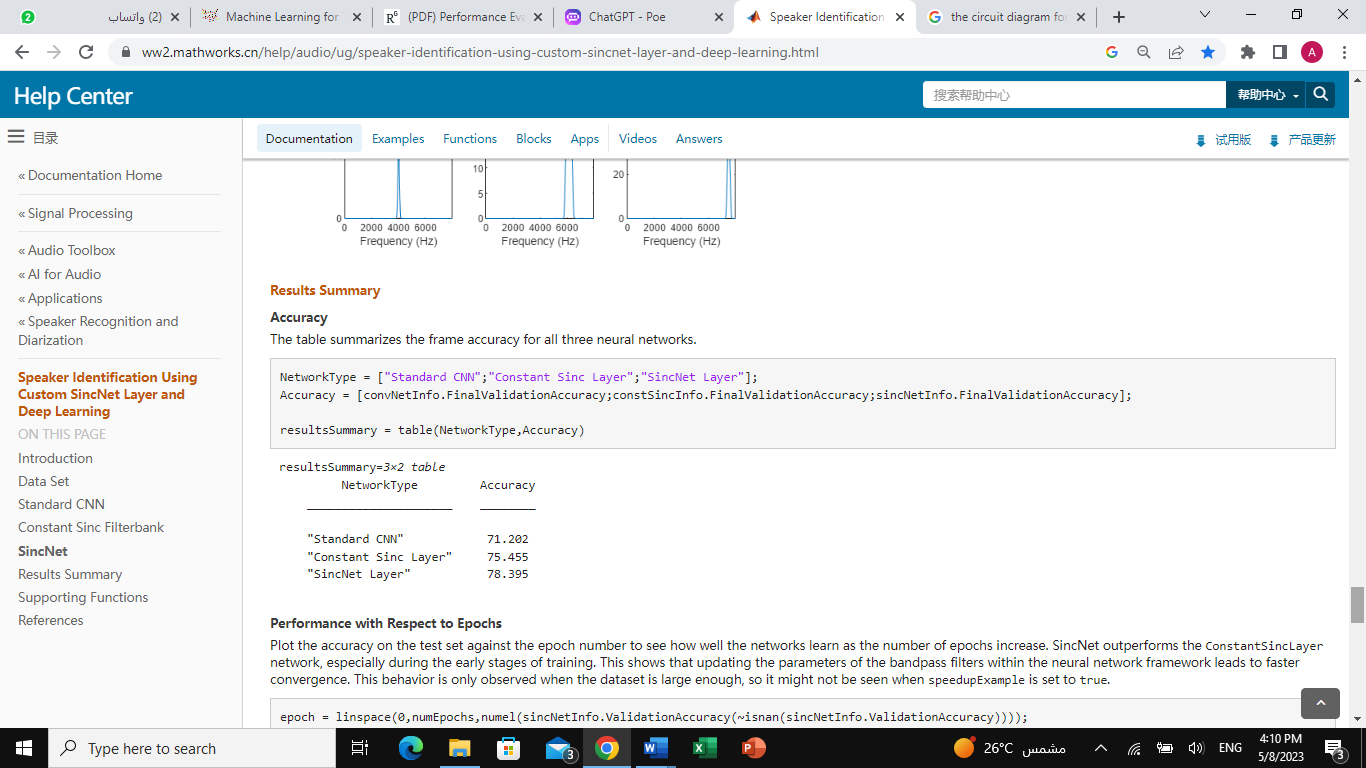
Description automatically generated



A screenshot of a computer

Description automatically generated





**Ch4 Conclusion**

**speaker recognition is an important application of artificial neural networks (ANNs) that involves identifying a person based on their voice. A common approach is to use a convolutional neural network (CNN) architecture with Mel-frequency cepstral coefficients (MFCCs) as pattern features.**

**The speaker recognition algorithm typically involves loading the audio dataset, extracting MFCCs for each audio sample, splitting the dataset into training and testing sets, building a CNN architecture with several convolutional layers followed by one or more fully connected layers, compiling the CNN model with an appropriate loss function and optimizer for classification, training the CNN using the training set, monitoring the performance on the validation set, evaluating the trained CNN on the testing set, calculating the accuracy, precision, recall, and F1 score using the confusion matrix, fine-tuning the CNN architecture and training parameters to improve performance if necessary, and saving the trained CNN model for future use.**

**Implementing this algorithm in MATLAB or another programming language involves defining and implementing the necessary functions for loading and processing the audio data, building and training the CNN model, and evaluating its performance. Careful tuning and evaluation using appropriate figures and curves can help to ensure the accuracy and reliability of the model.**

**REFRENCES**

1. [**https://www.researchgate.net/publication/320436764\_Performance\_Evaluation\_of\_Different\_Modeling\_Methods\_and\_Classifiers\_with\_MFCC\_and\_IHC\_Features\_for\_Speaker\_Recognition**](https://www.researchgate.net/publication/320436764_Performance_Evaluation_of_Different_Modeling_Methods_and_Classifiers_with_MFCC_and_IHC_Features_for_Speaker_Recognition)
2. **7th International Conference on Advances in Computing & Communications, ICACC-2017, 22-24 August 2017, Cochin, India**
3. [**https://patents.google.com/patent/EP1222506A1**](https://patents.google.com/patent/EP1222506A1)

**Image Feature Extraction using DCT**

**Ch1- introduction**

[Discrete cosine transform](https://www.sciencedirect.com/topics/computer-science/discrete-cosine-transform) (DCT) is a powerful transform to extract proper features for face recognition. After applying DCT to the entire face images, some of the coefficients are selected to construct feature vectors. Most of the conventional approaches select coefficients in a zigzag manner or by zonal masking. In some cases, the low-frequency coefficients are discarded in order to compensate [illumination variations](https://www.sciencedirect.com/topics/computer-science/illumination-variation). Since the discrimination power of all the coefficients is not the same and some of them are discriminant than others, so we can achieve a higher true recognition rate by using discriminant coefficients (DCs) as feature vectors. Discrimination power analysis (DPA) is a statistical analysis based on the DCT coefficients properties and discrimination concept. It searches for the coefficients which have more power to discriminate different classes better than others. The proposed approach, against the conventional approaches, is data-dependent and is able to find DCs on each database. The simulations results of the various coefficient selection (CS) approaches on ORL and Yale databases confirm the success of the proposed approach. The DPA-based approaches achieve the performance of PCA/LDA or better with less complexity. The proposed method can be implemented for any feature selection problem as well as DCT coefficients. Also, a new modification of PCA and [LDA](https://www.sciencedirect.com/topics/computer-science/linear-discriminant-analysis) is proposed namely, DPA–PCA and DPA–LDA. In these modifications DCs which are selected by DPA are used as the input of these transforms. Simulation results of DPA–PCA and DPA–LDA on the ORL and Yale database verify the improvement of the results by using these new modifications.

**To work with DCT in image processing, you will need the following tools:**

1. Image processing software: You will need software that supports image processing and can perform DCT on images. Examples of such software include MATLAB, Python with OpenCV, and ImageJ.

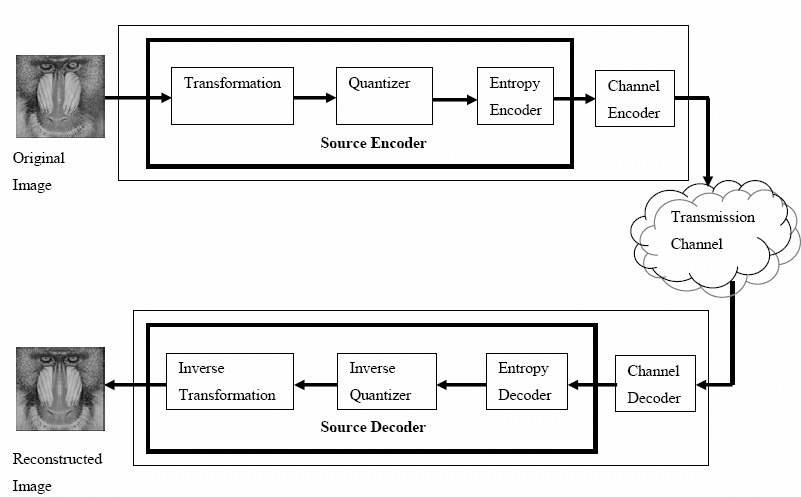
2. Programming language: To implement DCT-based applications, you will need to write code in a programming language such as MATLAB, Python, or C/C++. These languages have libraries and frameworks that support DCT computation and image processing.

3. Image dataset: You will need a dataset of images to test your DCT-based application. The dataset should contain images that are representative of the problem you are trying to solve. For example, if you are developing a face recognition system, you will need a dataset of face images.

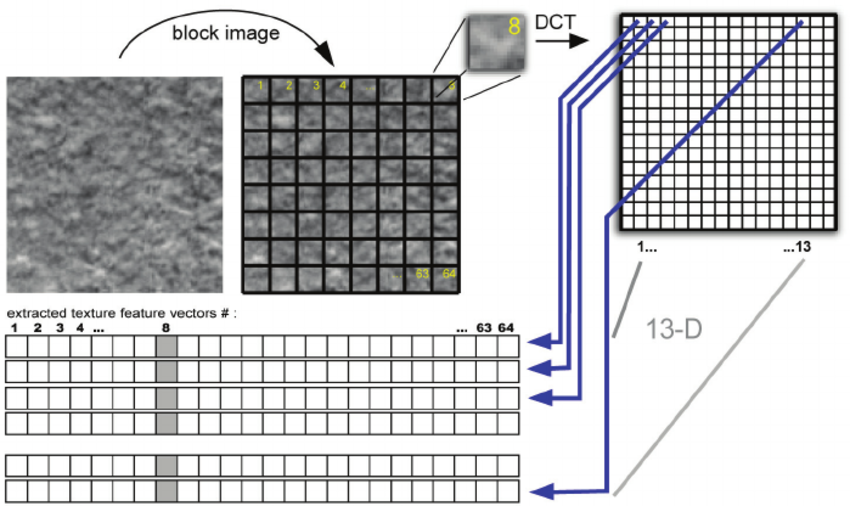
4. Hardware: The hardware requirements for DCT-based applications depend on the size of the images and the complexity of the processing algorithms. For small images, a standard laptop or desktop computer may be sufficient. However, for larger images or real-time processing, you may need a more powerful computer or specialized hardware such as a graphics processing unit (GPU).

5. Knowledge of DCT and image processing: To develop DCT-based applications, you will need to have a good understanding of the DCT transform and its properties, as well as image processing techniques such as filtering, segmentation, and feature extraction.

**Ch2-Block diagram :**

****

* **Image Acquisition:** The first step is to acquire an image using a camera or other imaging device. The image can be in any common format, such as JPEG, PNG, or BMP.
* **Color Space Conversion:** In order to apply DCT, the image must be converted to a color space that is better suited for frequency analysis. The most commonly used color space for DCT is YCbCr, which separates the image into luminance (Y) and chrominance (Cb and Cr) components.
* **Image Segmentation:** The image is divided into small blocks of typically 8x8 or 16x16 pixels. These blocks are processed independently using DCT.
* **DCT Calculation:** For each block, DCT is applied to the luminance (Y) component, resulting in a set of DCT coefficients. The DCT algorithm involves taking the cosine transform of each block and computing the frequency coefficients.
* **Coefficient Quantization:** Since most of the energy of the image is concentrated in a small number of coefficients, the remaining coefficients can be discarded or quantized to reduce the size of the feature vector. The quantization step involves dividing each coefficient by a quantization factor and rounding to the nearest integer.
* **Feature Extraction:** The resulting set of quantized DCT coefficients from all blocks is concatenated to form a feature vector that represents the image. This feature vector can be used for various tasks such as classification, object detection, or image retrieval.
* **Inverse DCT Transformation:** If required, the feature vector can be transformed back to the spatial domain by applying the inverse DCT transformation. This step is optional and usually not required for feature extraction.
* **Image Reconstruction:** Finally, the reconstructed image can be obtained by combining the processed blocks. This step is optional and usually not required for feature extraction.

**Ch-3Circuit diagram**

The Discrete Cosine Transform (DCT) has been widely used in the literature for efficient texture feature selection. It uses cosines of varying spatial frequencies as basis functions and is commonly known for its implementation in the JPEG compression standard (Bhaskaran & Konstantinides, 1995). The DCT coefficients are obtained covering different spectral bands. For texture images, much of the signal energy lies at low-frequency components, which appear in the upper left corner of the DCT. Knowing that DCT converts the spatial information into the frequency domain, texture features can be defined as the spectrum energies in different localizations of a local block. Since the DC coefficient represents (almost) the average grayscale value of each N × N block, it is not considered to carry any texture information. The remaining AC coefficients capture the details - or frequency and directionality properties - within the pixel-block and therefore can be considered to characterize image texture and be utilized as texture features.

In the figure above, Texture features extraction using DCT. DCT is applied on each block of the texture image and a feature vector is creating via summing up the square values of the coefficients in the diagonals of the block. In order to extract textural attributes, the images are initially partitioned into N × N pixelblocks, with 16 N = in our case. The block size was selected in order to reduce the number of extracted feature vectors and also try to effectively capture the texture information using 336 Tools in Artificial Intelligence a larger image patch. In addition, it was experimentally verified to produce enhanced classification results compared to a smaller pixel-block (e.g., N=8 ). Then, the DCT is applied to each distinct block, as illustrated in Fig. 2. From each DCT block, texture can be now represented by a feature vector Vm , with m∈[1, 2N − 2], the elements of which are the square sums of coefficients of the corresponding diagonals (i.e., zig-zag traversal lines). The vector resulting from the zig-zag ordering contain all the AC coefficients starting from the upper left location (i.e., (0, 1) ) to the bottom right (i.e., (N − 1, N − 1) ). Assuming that a given image is initially divided into M blocks of 16 16 × pixels, then a set of M feature vectors can be extracted that best describes the texture image content of the particular image. The specific indexing scheme was found to be robust, when similarity-based image rotation is considered.

**Ch-4 implementation:**

**// Define the size of the image block**

**BLOCK\_SIZE = 8**

**// Define the quantization matrix**

**QUANTIZATION\_MATRIX = np.array(**

**[[16, 11, 10, 16, 24, 40, 51, 61],**

**[12, 12, 14, 19, 26, 58, 60, 55],**

**[14, 13, 16, 24, 40, 57, 69, 56],**

**[14, 17, 22, 29, 51, 87, 80, 62],**

**[18, 22, 37, 56, 68, 109, 103, 77],**

**[24, 35, 55, 64, 81, 104, 113, 92],**

**[49, 64, 78, 87, 103, 121, 120, 101],**

**[72, 92, 95, 98, 112, 100, 103, 99]]**

**)**

**// Define the function for DCT calculation**

**def dct(block):**

**// Apply the DCT to the block using the OpenCV function**

**dct\_block = cv2.dct(block.astype(np.float32))**

**// Divide the DCT coefficients by the quantization matrix**

**quantized\_block = np.round(dct\_block / QUANTIZATION\_MATRIX)**

**return quantized\_block**

**// Load an image and convert it to grayscale**

**image = cv2.imread('image.jpg')**

**gray\_image = cv2.cvtColor(image, cv2.COLOR\_BGR2GRAY)**

**// Divide the image into blocks of size BLOCK\_SIZE x BLOCK\_SIZE**

**blocks = [gray\_image[i:i+BLOCK\_SIZE, j:j+BLOCK\_SIZE] for i in range(0, gray\_image.shape[0], BLOCK\_SIZE) for j in range(0, gray\_image.shape[1], BLOCK\_SIZE)]**

**// Apply DCT to each block**

**dct\_blocks = [dct(block) for block in blocks]**

**// Flatten the DCT coefficients for each block into a feature vector**

**feature\_vectors = [block.flatten() for block in dct\_blocks]**

In this algorithm, we define the size of the image block and the quantization matrix. We then define the dct() function, which applies DCT to a block using the OpenCV cv2.dct() function. The DCT coefficients are then divided by the quantization matrix and rounded to the nearest integer.

We then load an image and convert it to grayscale. The image is divided into blocks of size BLOCK\_SIZE x BLOCK\_SIZE, and DCT is applied to each block using the dct() function. Finally, the DCT coefficients for each block are flattened into a feature vector.

**Ch5 Conclusion :**

the Discrete Cosine Transform (DCT) is a widely used mathematical transform in image processing and compression. It transforms an image from the spatial domain to the frequency domain, where it can be represented as a set of frequency coefficients. The DCT is particularly effective for compressing images because it can remove redundancy in the image data by representing it in fewer coefficients, without significant loss of image quality.

DCT is used in a variety of applications, including image and video compression, feature extraction, and pattern recognition. It is commonly used in popular image and video compression standards such as JPEG, MPEG, and H.264. In addition, DCT is used in various machine learning applications for image recognition and classification.

The DCT algorithm is relatively simple and can be implemented efficiently in both software and hardware. There are many libraries available for implementing DCT in various programming languages, including Python, MATLAB, and C/C++. The choice of library depends on the specific application and the desired level of performance.

Overall, the DCT is a powerful tool in image processing and compression that has been widely adopted in various applications.

**REFRENCES**

[**https://www.sciencedirect.com/science/article/abs/pii/S0031320309004142**](https://www.sciencedirect.com/science/article/abs/pii/S0031320309004142)

[**https://www.researchgate.net/publication/4343765\_Feature\_extraction\_using\_discrete\_cosine\_transform\_for\_face\_recognition**](https://www.researchgate.net/publication/4343765_Feature_extraction_using_discrete_cosine_transform_for_face_recognition)

[**https://www.researchgate.net/figure/Texture-features-extraction-using-DCT-DCT-is-applied-on-each-block-of-the-texture-image\_fig2\_221787121**](https://www.researchgate.net/figure/Texture-features-extraction-using-DCT-DCT-is-applied-on-each-block-of-the-texture-image_fig2_221787121)

[**https://www.researchgate.net/figure/Figure4-Basic-block-diagram-for-DCT-conversion\_fig4\_46105836**](https://www.researchgate.net/figure/Figure4-Basic-block-diagram-for-DCT-conversion_fig4_46105836)