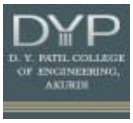


D. Y. Patil college of Engineering, Akurdi, Pune-44

Department of Information Technology

Vision of the Institute:

"Empowerment through Knowledge".

Mission of the Institute:

To educate the students to transform them as professionally competent and quality conscious engineers by providing conducive environment for teaching, learning, and overall personality development, culminating the institute into an international seat of excellence.

Vision of the Department

Developing globally competent IT professional for sustainable growth of humanity.

Mission of the Department

M1 Build a strong foundation and techniques for problem-solving and inculcate communication skills as an integral component of Information Technology

M2 Develop competency skills in the faculty members and students to serve the societal challenges and needs in lieu of its multidisciplinary applications in the field of Information Technology

M3 Encourage development of strong technical skills and knowledge and to encourage studentsto undergo research in the field of Information Technology

M4 Nurture students to become ethical and committed lifelong IT professionals

M5 Empower students with strong decision-making skills and technical competency to accomplish start-up ideas in the field of IT Engineering

Program Educational Objectives (PEOs) defined by Department of Information Technology D Y Patil college of Engineering, Akurdi, Pune :

1) **Core Competency:** To provide graduates with a solid foundation in Mathematics, Science, Engineering fundamentals required to solve complex Software Engineering Problem.
2) **Breadth:** To impart the knowledge and skills in the field of Information Technology; and to comprehend, analyze, design and create novel products and solutions for the real-time and Complex Engineering problems of any domain with innovative approaches.
3) **Professionalism:** To inculcate in graduates, professional and ethical values, effective communication skills, teamwork, multidisciplinary approach, and ability to relate engineering issues in broader social context.
4) **Learning Environment:** To provide graduates with an academic environment that makes them aware of excellence in leadership, presentation, time management and ethics leading them to become responsible and competent professionals prepared to address challenges in the field of IT at global level.
5) **Attainment:** To empower graduates with an attitude and skills of Research, Entrepreneur and Higher education in the field of Information Technology.

Program Specific Outcomes (PSO) defined by Department of Information Technology D Y Patil college of Engineering, Akurdi, Pune :

1) Apply design methodologies, application development tools, engineering skills in Software Engineering Domains and IT Application areas like Cloud Computing, Software Testing, Mobile App Development, etc.
2) Aspire to pursue Higher Education in the specialized fields of IT Engineering and management programs like Data science, Cyber Security, Artificial Intelligence, etc.
3) Formulate decision-making skills, IT Engineering skills, and knowledge to implement start-up ideas as an entrepreneur in the fields such as Cyber Security, Mobile ApplicationDevelopment, etc.
4) Devise to design, implement, and evaluate IT based software systems to serve the needs of society or IT industries at large.

PROGRAM OUTCOME (PO)

Students are expected to know and be able to–		
PO1	**Engineering knowledge**	An ability to apply knowledge of mathematics, computing, science, engineering and technology.
PO2	**Problem analysis**	An ability to define a problem and provide a systematic solution with the help of conducting experiments, analyzing the problem and interpreting the data.
PO3	**Design / Developmentof Solutions**	An ability to design, implement, and evaluate software or a software /hardware system, component, or process to meet desired needs within realistic constraints.
PO4	**Conduct Investigation of Complex Problems**	An ability to identify, formulate, and provide essay schematic solutionsto complex engineering /Technology problems.
PO5	**Modern Tool Usage**	An ability to use the techniques, skills, and modern engineering technology tools, standard processes necessary for practice as a IT professional.
PO6	**The Engineer and Society**	An ability to apply mathematical foundations, algorithmic principles, and computer science theory in the modeling and design of computer- based systems with necessary constraints and assumptions.
PO7	**Environment and Sustainability**	An ability to analyze and provide solution for the local and global impactof information technology on individuals, organizations and society.
PO8	**Ethics**	An ability to understand professional, ethical, legal, security and social issues and responsibilities.
PO9	**Individual and Team Work**	An ability to function effectively as an individual or as a team memberto accomplish a desired goal(s).
PO10	**Communication Skills**	An ability to engage in life-long learning and continuing professional development to cope up with fast changes in the technologies /tools with the help of electives, profession along animations and extra- curricular activities.
PO11	**Project Management and Finance**	An ability to communicate effectively in engineering community at large by means of effective presentations, report writing, paper publications, demonstrations.
PO12	**Life-long Learning**	An ability to understand engineering, management, financial aspects, performance, optimizations and time complexity necessaryfor professional practice.

COURSE OBJECTIVE

CO1	To be able to formulate deep learning problems corresponding to different applications.
CO2	To be able to apply deep learning algorithms to solve problems of moderate complexity.
CO3	To apply the algorithms to a real-world problem, optimize the models learned and report on the expected accuracy that can be achieved by applying the models.

COURSE OUTCOMES

On completion of the course, students will be able to-	
CO1	Learn and Use various Deep Learning tools and packages.
CO2	Build and train a deep Neural Network models for use in various applications.
CO3	Apply Deep Learning techniques like CNN, RNN Auto encoders to solve real word Problems.
CO4	Evaluate the performance of the model build using Deep Learning.

414447: LAB PRACTICE IV

Teaching Scheme: Practical: 2Hours/Week **Credits: 01**

Examination Scheme: Term Work: 25Marks
Practical: 25 Marks

Prerequisites: Python programming language

Sr. No	List of Assignments	Page No
1	Study of Deep learning Packages: Tensorflow, Keras, Theano and PyTorch. Document the distinct features and functionality of the packages. **Note: Use a suitable dataset for the implementation of following assignments.**	
2	Implementing Feedforward neural networks with Keras and TensorFlow a. Import the necessary packages b. Load the training and testing data (MNIST/CIFAR10) c. Define the network architecture using Keras d. Train the model using SGD e. Evaluate the network f. Plot the training loss and accuracy	
3	Build the Image classification model by dividing the model into following 4 stages: a. Loading and preprocessing the image data b. Defining the model's architecture c. Training the model d. Estimating the model's performance	
4	Use Autoencoder to implement anomaly detection. Build the model by using: a. Import required libraries b. Upload / access the dataset c. Encoder converts it into latent representation d. Decoder networks convert it back to the original input e. Compile the models with Optimizer, Loss, and Evaluation Metrics	
5	Implement the Continuous Bag of Words (CBOW) Model. Stages can be: a. Data preparation b. Generate training data c. Train model d. Output	
6	Object detection using Transfer Learning of CNN architectures a. Load in a pre-trained CNN model trained on a large dataset b. Freeze parameters (weights) in model's lower convolutional layers c. Add custom classifier with several layers of trainable parameters to model d. Train classifier layers on training data available for task e. Fine-tune hyper parameters and unfreeze more layers as needed	

REALIZATION OF CO'S and BT Levels

Sr. No	List of Assignments	CO'S Realized	BT Level
1	Study and use of Deep learning Packages: Tensorflow, Keras, Theano and PyTorch. Document the distinct features and functionality of the packages. **Note: Use a suitable dataset for the implementation of following assignments.**	**CO1, CO2, CO3, CO4**	**3**
2	Implementing Feedforward neural networks with Keras and TensorFlow a. Import the necessary packages b. Load the training and testing data (MNIST/CIFAR10) c. Define the network architecture using Keras d. Train the model using SGD e. Evaluate the network f. Plot the training loss and accuracy	**CO1, CO2, CO3, CO4**	**5**
3	Build the Image classification model by dividing the model into following 4 stages: e. Loading and preprocessing the image data f. Defining the model's architecture g. Training the model h. Estimating the model's performance	**CO1, CO2, CO3, CO4**	**6**
4	Use Autoencoder to implement anomaly detection. Build the model by using: a. Import required libraries b. Upload / access the dataset c. Encoder converts it into latent representation d. Decoder networks convert it back to the original input e. Compile the models with Optimizer, Loss, and Evaluation Metrics	**CO1, CO2, CO3, CO4**	**6**
5	Implement the Continuous Bag of Words (CBOW) Model. Stages can be: a. Data preparation b. Generate training data c. Train model d. Output	**CO1, CO2, CO3, CO4**	**5**
6	Object detection using Transfer Learning of CNN architectures f. Load in a pre-trained CNN model trained on a large dataset g. Freeze parameters (weights) in model's lower convolutional layers h. Add custom classifier with several layers of trainable parameters to model i. Train classifier layers on training data available for task j. Fine-tune hyper parameters and unfreeze more layers as needed	**CO1, CO2, CO3, CO4**	**5**

GUIDELINES FOR STUDENTS

Guidelines for Student's Lab Journal

1. Students should submit term work in the form of a handwritten journal based on a specified list of assignments.

2. Practical Examination will be based on the term work.

3. Candidate is expected to know the theory involved in the experiment.

4. The practical examination should be conducted if and only if the journal of the candidate is complete in all respects.

Guidelines for Lab /TW Assessment

1. Examiners will assess the term work based on performance of students considering the parameters such as timely conduction of practical assignment, methodology adopted for implementation of practical assignment, timely submission of assignment in the form of handwritten write-up along with results of implemented assignment, attendance etc.

2. Examiners will judge the understanding of the practical performed in the examination by asking some questions related to theory & implementation of experiments he/she has carried out.

3. Appropriate knowledge of usage of software and hardware related to the respective laboratory should be checked by the concerned faculty member.

Guidelines for Laboratory Conduction

As a conscious effort and little contribution towards Green IT and environment awareness, attaching printed papers of the program in a journal may be avoided. There must be hand-written write-ups for every assignment in the journal. The DVD/CD containing student's programs should be attached to the journal by every student and the same to be maintained by the department/lab In-charge is highly encouraged. For reference one or two journals may be maintained with program prints at Laboratory.

Guidelines for Practical Examination

1. During practical assessment, maximum weightage should be given to satisfactory implementation of the problem statement.

2. Student's understanding of the fundamentals, effective and efficient implementation can be evaluated by asking relevant questions based on implementation of experiments he/she has carried out.

RUBRICS FOR LABORATORY ASSESSMENT

1. Attendance

Assessment Outcome ⇨ / Dimensions ⇩	Poor (1)	Satisfactory(2)	Good (3)	Very Good (4)	Excellent (5)
1.Attendance with Involvement of Student (5M)	Passive observer	Very little involvement	Good Involvement in performing experiment	Individual Involvement in performing experiment	Individual and self - Involvement in performing experiment

2. Viva

Assessment Outcome ⇨ / Dimensions ⇩	Poor (1)	Satisfactory(2)	Good (3)	Very Good (4)	Excellent (5)
1.Preparation and Basic Knowledge (5M)	No preparation	Little Knowledge	Prepared Well	Very well prepared	Advance Knowledge
2.Program development and execution (5M)	Not Executed	Partially executed	Executed	Executed without additional modification	Executed with additional modification
3.Punctuality and Ethics (5M)	Attendance Below 50% and not following the lab instructions	Attendance 50% to 75% And sometimes copies the program	Regular attendance 75-00% and follows the instruction and try to perform on his own	Regular attendance 80-90% and follows the instruction and try to perform on his own	90-100 % attendance, follows all instructions and execute the program on his own

3. Presentation

Assessment Outcome ⇨ / Dimensions ⇩	Poor (1)	Satisfactory(2)	Good (3)	Very Good (4)	Excellent (5)
Journal Presentation (5M)	Not Prepared	Incomplete	Completed documentation	well documented	Very well documented

Outcome: Student will be able to

i. Apply knowledge to real life examples and develop practical approach

ii. Design Basic Application.

Note: Students with poor marks should repeat the assignment

Assignment No.1

Title: Study of Deep learning Packages: Tensorflow, Keras, Theano and PyTorch.

Document the distinct features and functionality of the packages.

Aim: Study and installation of following Deep learning Packages:

- **i.** Tensor Flow
- **ii.** Keras
- **iii.** Theno
- **iv.** PyTorch

Theory:

Installation of Tensorflow On Ubntu:

1. Install the Python Development Environment:

Download Python, the PIP package, and a virtual environment. If these packages are already installed, this step can be skipped. Download and install what is needed by visiting the following links:

https://www.python.org/

https://pip.pypa.io/en/stable/installing/

https://docs.python.org/3/library/venv.html

To install these packages, run the following commands in the terminal:

sudo apt update

sudo apt install python3-dev python3-pip python3-venv

2. Create a Virtual Environment:

Navigate to the directory where you want to store your Python 3.0 virtual environment. It can be in your home directory, or any other directory where your user can read and write permissions.

mkdir tensorflow_files

cd tensorflow_files

Now, you are inside the directory. Run the following command to create a virtual environment:

python3 -m venv virtualenv

The command above creates a directory named virtualenv. It contains a copy of the Python binary, the PIP package manager, the standard Python library, and other supporting files.

3. Activate the Virtual Environment:

source virtualenv/bin/activate

Once the environment is activated, the virtual environment' s bin directory will be added to the beginning of the $PATH variable. Your shell' s prompt will alter, and it will show the name of the virtual environment you are currently using, i.e. virtualenv.

4. Update PIP:

pip install --upgrade pip

5. Install TensorFlow:

The virtual environment is activated, and it' s up and running. Now, it' s time to install the TensorFlow package.

pip install -- upgrade TensorFlow

Installation of Keras on Ubntu :

Prerequisite : Python version 3.5 or above.

Step 1: Install and Update Python3 and Pip

Skip this step if you already have Python3 and Pip on your machine**.**

sudo apt install python3 python3.pip

sudo pip3 install – – upgrade pip

Step 2: Upgrade Setuptools

pip3 install – – upgrade setuptools

Step 3: Install TensorFlow

pip3 install tensorflow

Verify the installation was successful by checking the software package information:

pip3 show tensorflow

Step 4: Install Keras

pip3 install keras

Verify the installation by displaying the package information:

pip3 show keras

[https://phoenixnap.com/kb/how-to-install-keras-on-linux]

Installation of Theano on Ubuntu:

Step 1: First of all, we will install Python3 on our Linux Machine. Use the following command in the terminal to install Python3.

sudo apt-get install python3

Step 2: Now, install the pip module

sudo apt install python3-pip

Step 3: Now, install the Theano

Verifying Theano package Installation on Linux using PIP

python3 -m pip show theano

Installation of PyTorch

First, check if you are using python' s latest version or not.Because PyGame requires python 3.7 or a higher version

python3 – version

pip3 – version

pip3 install torch==1.8.1+cpu torchvision==0.9.1+cpu torchaudio==0.8.1 -f

https://download.pytorch.org/whl/torch_stable.html

[Ref : https://www.geeksforgeeks.org/install-pytorch-on-linux/]

Python Libraries and functions required

1. Tensorflow, keras

numpy : NumPy is a Python library used for working with arrays. It also has functions for working in domain of linear algebra, fourier transform, and matrices. NumPy stands for Numerical Python. To import numpy use

import numpy as np

pandas: pandas is a fast, powerful, flexible and easy to use open source data analysis and manipulation tool, built on top of the Python programming language. To import pandas use

import pandas as pd

sklearn : Scikit-learn (Sklearn) is the most useful and robust library for machine learning in Python. It provides a selection of efficient tools for machine learning and statistical modeling including classification, regression, clustering and dimensionality reduction via a consistence interface in Python. This library, which is largely written in Python, is built upon NumPy, SciPy and Matplotlib. For importing train_test_ split use

from sklearn.model_selection import train_test_split

2. For Theaon Requirements:

Python3

Python3-pip

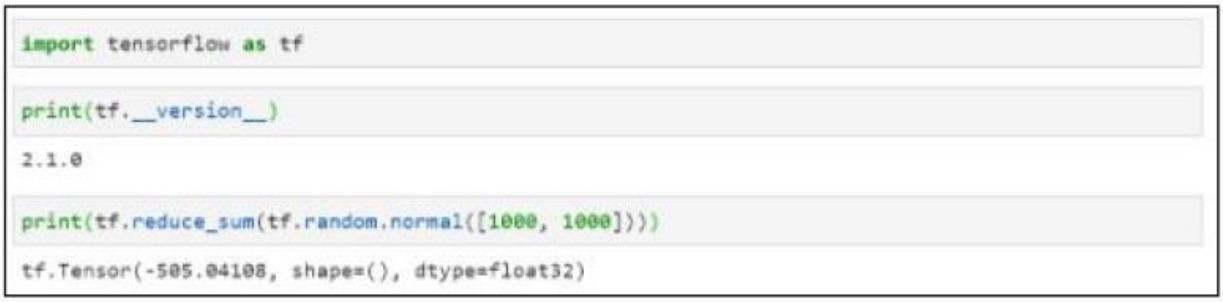
NumPy

SciPy

BLAS

Sample Code with comments

1. Tensorflow Test program:



```
import tensorflow as tf

print(tf.__version__)

2.1.0

print(tf.reduce_sum(tf.random.normal([1000, 1000])))

tf.Tensor(-505.04108, shape=(), dtype=float32)
```

2. Keras Test Program:

```
from tensorflow import keras
from keras import datasets
#
# Load MNIST data
#
(train_images, train_labels), (test_images, test_labels) = datasets.mnist.load_data()
#
# Check the dataset loaded
#
train_images.shape, test_images.shape
```

3. Theano test program

```
# Python program showing
# addition of two scalars
# Addition of two scalars
import numpy
import theano.tensor as T
from theano import function
```

```
# Declaring two variables
x = T.dscalar('x')
y = T.dscalar('y')
# Summing up the two numbers
z = x + y
# Converting it to a callable object
# so that it takes matrix as parameters
f = function([x, y], z)
f(5, 7)
```

4. Test program for PyTorch

```
## The usual imports
import torch
import torch.nn as nn
## print out the pytorch version used
print(torch.__version__)
```

Conclusion:

Tensorflow, PyTorch, Keras and Theano all these packages are installed and ready for Deep learning applications . As per application domain and dataset we can choose the appropriate package and build required type of Neural Network.

Output of Code:

Note: Run the code and attach output of the code here.

Questions:

1)What is Deep learning ?

2)What are various packages in python for supporting Machine Learning libraries and which are mainly used for Deep Learning ?

3)Compare Tensorflow / Keras/Theno and PyTorch on following points(make a table) :

i. Available Functionality

ii. GUI status

iii. Versions.

iv. Features

v. Compatibility with other environments.

vi. Specific Application domains.

4) Enlist the Models Datasets and pretrained models, Libraaries and Extensions , Tools related to Tensorflow also discuss any two casestudies like (PayPal, Intel, Etc.) related to Tensor Flow. [Ref:https://www.tensorflow.org/about]

5) Explain the Keras Ecosystem.(kerastuner,kerasNLP,kerasCV,Autokeras and Modeloptimization.). Also explain following concepts related to keras :

1. Developing sequential Model

2. Training and validation using the inbuilt functions

3. Parameter Optimization. [Ref: https://keras.io/]

6) Explain simple Theano program.

7) Explain PyTorch Tensors . And also explain Uber' s Pyro, Tesala Autopilot.[https://pytorch.org/]

Assignment No. 2

Problem Statement:

Implementing Feed forward Neural Network with Keras and Tensorflow.

1. Import the necessary packages
2. Load the training and testing data(MNIST)
3. Define the network architecture using keras
4. Train the model using SGD
5. Evaluate the network
6. Plot the training loss and accuracy

Objectives:

1. Understand how to use Tensorflow Eager and Keras Layers to build a neural network architecture.
2. Understand how a model is trained and evaluated.
3. Identify digits from images.
4. Our main goal is to train a neural network (using Keras) to obtain > *90%* accuracy on MNIST dataset.
5. Research at least 1 technique that can be used to improve model generalization.

Solution Expected:

Implement and train a feed-forward neural network (also known as an "MLP" for "multi-layer perceptron") on a dataset called MNIST and improve model generalization by achieving increased accuracy and decrease loss where model gains good confidence with the prediction.

Methodology to be used

- Deep Learning
- Feed Forward Neural Network

Theory:

Deep learning has revolutionized the world of machine learning as more and more ML practitioners have adopted deep learning networks to solve real-world problems. Compared to the more traditional ML models, deep learning networks have been shown superior performance for many applications.

The first step toward using deep learning networks is to understand the working of a simple feedforward neural network we get started with how we can build our first neural network model using **Keras** running on top of the **Tensorflow** library.

TensorFlow is an open-source platform for machine learning. Keras is the high-level application programming interface (API) of TensorFlow. Using Keras, we can rapidly develop a prototype system and test it out. This is the first in a three-part series on using TensorFlow for supervised classification tasks.

A Conceptual Diagram of the Neural Network:

we'll build a supervised classification model that learns to identify digits from images. We'll use the well-known MNIST dataset to train and test our model. The MNIST dataset consists of 28-by-28 images of handwritten digits along with their corresponding labels.

We'll construct a neural network with one hidden layer to classify each digit image into its corresponding label. The figure below shows a conceptual diagram of the neural network we are about to build. The output layer consists of 10 units, where each unit corresponds to a different digit. Each unit computes a value that can be interpreted as the confidence value of its respective digit. The final classification is the digit with the maximum confidence value.

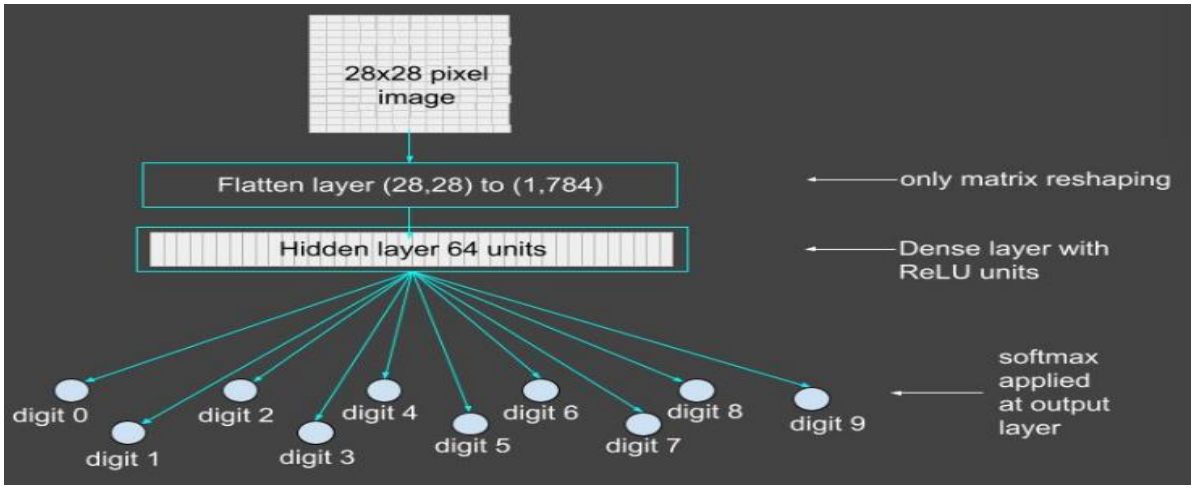


Figure 1: Conceptual diagram of the neural network. Each output unit corresponds to a digit and has its confidence value. The final classification is the digit with the maximum confidence value.

TensorFlow and Keras Libraries:

If Keras and TensorFlow are not installed on your system, you can easily do so using pip or conda depending upon your Python environment.

```
pip install tensorflow
```

In the context of ML, a tensor is a multidimensional array, which in its simplest form is a scalar. Vectors and matrices are special cases of tensors. In TensorFlow, a tensor is a data structure. It is a

multidimensional array composed of elements of the same type. Tensors are used to encapsulate all inputs and outputs to a deep learning network. The training dataset and each test example has to be cast as a tensor. All operations within the layers of the network are also performed on tensors.

Layers in TensorFlow:

You can build a fully connected feedforward neural network by stacking layers sequentially so that the output of one layer becomes the input to the next. In TensorFlow, layers are callable objects, which take tensors as input and generate outputs that are also tensors. Layers can contain weights and biases, which are both tuned during the training phase. We'll create a simple neural network from two layers:

1. Flatten layer
2. Dense layer

The Flatten Layer:

This layer flattens an input tensor without changing its values. Given a tensor of rank n, the Flatten layer reshapes it to a tensor of rank 2. The number of elements on the first axis remains unchanged. The elements of the input tensor's remaining axes are stacked together to form a single axis. We need this layer to create a vectorized version of each image for further input to the next layer.

The Dense Layer

The dense layer is the fully connected, feedforward layer of a neural network. It computes the weighted sum of the inputs, adds a bias, and passes the output through an activation function. We are using the ReLU activation function for this example. This function does not change any value greater than 0. The rest of the values are all set to 0.

The computations of this layer for the parameters shown in the code above are all illustrated in the figure below.

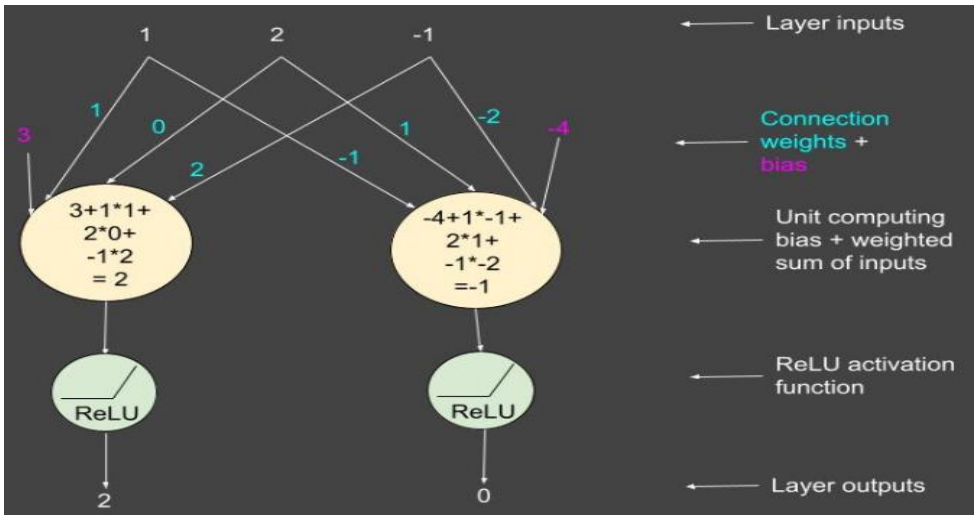


Figure 2: Hidden layer computations.

Creating a Model in TensorFlow:

We are now ready to create a model of a simple neural network with one hidden layer. The simplest method is to use Sequential() with a list of all the layers you want to stack together. The code below creates a model and prints its summary. Note the use of relu as the activation function in the first dense layer and a softmax in the output layer. The softmax function normalizes all outputs to sum to 1 and ensures that they are in the range [0, 1].

```
model = tf.keras.Sequential([
    tf.keras.layers.Flatten(input_shape=(28, 28)),
    tf.keras.layers.Dense(64, activation='relu'),
    tf.keras.layers.Dense(10, activation='softmax')
])
```

Compile the Model

Next we compile the model. Here, we specify the optimizer and loss function of our model. The optimization algorithm determines how the connection weights are updated at each training step with respect to the given loss function. Because we have a multiclass classification problem, the loss function we'll use is categorical_crossentropy, coupled with the adam optimizer. You can experiment with other optimizers too. The value of the metrics argument sets the parameter to monitor and record during the training phase.

```
model.compile(optimizer='sgd',
              loss='sparse_categorical_crossentropy',
              metrics=['accuracy'])
```

Train the Neural Network:

Now that the model is ready, it's time to train it. We'll load the dataset, train the model, and view the training process. Note that the outputs shown here will vary with every run of the program because of the stochastic nature of the algorithms involved.

Load the Dataset:

The following code loads the training set and the test set of the MNIST data. It also prints the statistics of both sets. Because our model has 10 outputs, one for each digit, we need to convert the absolute image labels to categorical ones. The utils module in the Keras library provides the method to_categorical() for this conversion.

Train the Model:

The fit() method of the model object trains the neural network. If you want to use a validation set during training, all you have to do is define the percentage of validation examples to be taken from the training set. The splitting of the training set into a train and validation set is automatically taken care of by the fit() method.

In the code below, the fit() method is called in 10 epochs.

View the Training Process:

The fit() method returns a history object with detailed information regarding model training. The history attribute is a dictionary object.

To view the learning process, we can plot the accuracy and loss corresponding to different epochs for both the training and validation sets. The following code creates two graphs for these two metrics.

The predict() method:

If you want to see the output of the network for one or more train/test examples, use the predict() method. The following example code prints the values of the output layer when the first test image is used as an input. It is a 10-dimensional vector of confidence values corresponding to each digit. The final classification of the image is the argmax() of this vector.

The evaluate() method:

The evaluate() method computes the overall accuracy of the model on the dataset passed as an argument. The code snippet below prints the classification accuracy on both the training set and the test set.

Code Snippets:

Importing Libraries

```
[1]  #import necessary libraries
     import tensorflow as tf
     from tensorflow import keras
```

```
[2]  import pandas as pd
     import numpy as np
     import matplotlib.pyplot as plt
     import random
     %matplotlib inline
```

Loading & Preparing the data

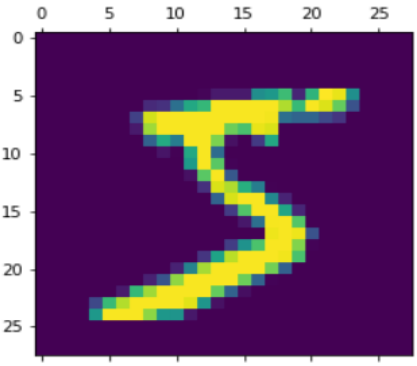
MNIST stands for Modified National Institute of Standards and Technology dataset. It is a dataset of 70,000 handwritten images.Each image is of 28*28 pixel i.e about 784 features. Each feature represent only one pixel intensity i.e from 0(white) to 255(black). This dataset is further devided into 60000 training and 10000 testing images.

```
#import dataset and split into train and test
mnist = tf.keras.datasets.mnist
(x_train,y_train),(x_test,y_test) = mnist.load_data()
```

```
Downloading data from https://storage.googleapis.com/tensorflow/tf-keras-datasets/mnist.npz
11493376/11490434 [==============================] - 0s 0us/step
11501568/11490434 [==============================] - 0s 0us/step
```

```
#to see how first image looks
plt.matshow(x_train[0])
```

```
<matplotlib.image.AxesImage at 0x7f686e32ed10>
```



normalizing the images by scaling the pixel intensities to the range 0 to 1

```
[ ] #normalizing the images by scaling the pixel intensities to the range 0 to 1
    x_train = x_train/255
    x_test = x_test/255
```

```
x_train[0]
```

```
array([[0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.19215686, 0.93333333, 0.99215686,
        0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.99215686,
        0.99215686, 0.99215686, 0.98431373, 0.36470588, 0.32156863,
        0.32156863, 0.21960784, 0.15294118, 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.07058824, 0.85882353, 0.99215686,
        0.99215686, 0.99215686, 0.99215686, 0.99215686, 0.77647059,
        0.71372549, 0.96862745, 0.94509804, 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.        , 0.        ],
       [0.        , 0.        , 0.        , 0.        , 0.        ,
        0.        , 0.        , 0.        , 0.31372549, 0.61176471,
        0.41960784, 0.99215686, 0.99215686, 0.80392157, 0.04313725,
        0.        , 0.16862745, 0.60392157, 0.        , 0.        ,
```

✓ 0s completed at 2:22 PM

Creating the model

The Relu function is one of the most popular activation function. It stands for"Rectified Linear Unit". Mathematically this function is defined as y=max(0,x). The relu function returns 0 if the input is negative and linear if the input is positive.

The softmax function is another activation function. It changes input values into values that reach from 0 to 1.

```
[ ] model = keras.Sequential([
        keras.layers.Flatten(input_shape=(28,28)),
        keras.layers.Dense(128,activation='relu'),
        keras.layers.Dense(10,activation='softmax')
    ])
```

```
[ ]  model.summary()
```

```
Model: "sequential_2"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 flatten_2 (Flatten)         (None, 784)               0

 dense_4 (Dense)             (None, 128)               100480

 dense_5 (Dense)             (None, 10)                1290

=================================================================
Total params: 101,770
Trainable params: 101,770
Non-trainable params: 0
_________________________________________________________________
```

Compile the model

```
[ ]  model.compile(optimizer='sgd',
                   loss='sparse_categorical_crossentropy',
                   metrics=['accuracy'])
```

Train the model

```
[ ] history=model.fit(x_train,y_train,validation_data=(x_test,y_test),epochs=10)
```

```
Epoch 1/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.6254 - accuracy: 0.8444 - val_loss: 0.3584 - val_accuracy: 0.9009
Epoch 2/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.3366 - accuracy: 0.9053 - val_loss: 0.3012 - val_accuracy: 0.9156
Epoch 3/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2889 - accuracy: 0.9188 - val_loss: 0.2641 - val_accuracy: 0.9252
Epoch 4/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.2583 - accuracy: 0.9276 - val_loss: 0.2405 - val_accuracy: 0.9330
Epoch 5/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2352 - accuracy: 0.9345 - val_loss: 0.2215 - val_accuracy: 0.9384
Epoch 6/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.2162 - accuracy: 0.9392 - val_loss: 0.2047 - val_accuracy: 0.9430
Epoch 7/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.2004 - accuracy: 0.9437 - val_loss: 0.1907 - val_accuracy: 0.9455
Epoch 8/10
1875/1875 [==============================] - 5s 3ms/step - loss: 0.1871 - accuracy: 0.9472 - val_loss: 0.1821 - val_accuracy: 0.9476
Epoch 9/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1755 - accuracy: 0.9505 - val_loss: 0.1709 - val_accuracy: 0.9501
Epoch 10/10
1875/1875 [==============================] - 5s 2ms/step - loss: 0.1653 - accuracy: 0.9539 - val_loss: 0.1622 - val_accuracy: 0.9521
```

✓ 0s completed at 2:22 PM

Evaluate the model

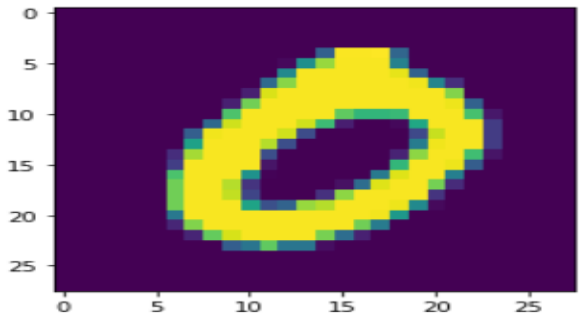
```
[ ]  test_loss,test_acc=model.evaluate(x_test,y_test)
     print("Loss=%.3f" %test_loss)
     print("Accuracy=%.3f" %test_acc)
```

```
313/313 [==============================] - 1s 2ms/step - loss: 0.1622 - accuracy: 0.9521
Loss=0.162
Accuracy=0.952
```

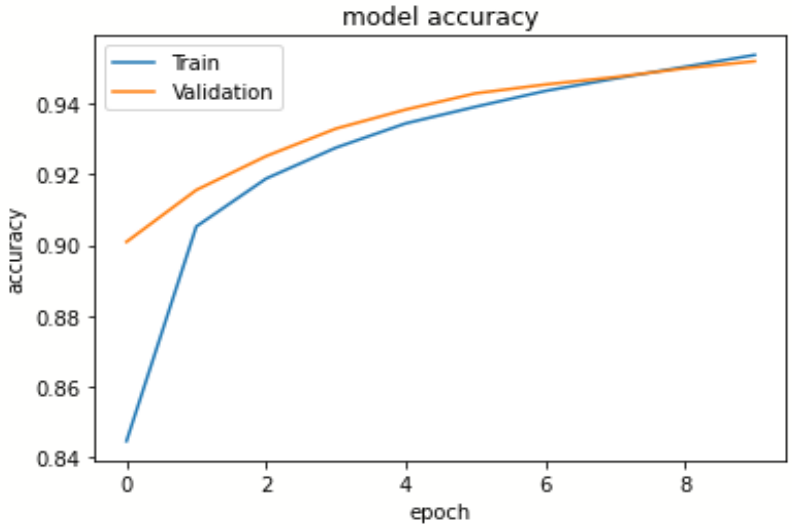
Making prediction on new data

```
n=random.randint(0,9999)
plt.imshow(x_test[n])
plt.show
```

```
<function matplotlib.pyplot.show(*args, **kw)>
```



Plot graph for accuracy and loss



Confusion Matrix

```
test_predict = model.predict(x_test)
# Get the classification labels
test_predict_labels = np.argmax(test_predict, axis=1)
confusion_matrix = tf.math.confusion_matrix(labels=y_test, predictions=test_predict_labels)
print('Confusion matrix of the test set:\n', confusion_matrix)
```

```
Confusion matrix of the test set:
 tf.Tensor(
[[ 965    0    1    1    0    4    6    1    2    0]
 [   0 1117    2    2    1    1    3    2    7    0]
 [   6    2  977   13    5    1    7    8   12    1]
 [   1    0    9  960    1    9    1   10   13    6]
 [   1    1    4    0  939    1    8    3    3   22]
 [   9    1    2   19    3  823   11    2   14    8]
 [   9    3    2    2   10    7  919    1    5    0]
 [   1   10   18    7    5    1    0  968    1   17]
 [   3    1    3   18    7    7    7    9  916    3]
 [   8    8    0   12   22    3    1   12    6  937]], shape=(10, 10), dtype=int32)
```

Conclusion:

With above code we can see that, throughout the epochs, our model accuracy increases and loss decreases that is good since our model gains confidence with our prediction

This indicates the model is trained in a good way:

1. The two loss(loss and val_loss) are decreasing and the accuracy (accuracy and val_accuracy) increasing.
2. The val_accuracy is the measure of how good the model is predicting so, it is observed that the model is well trained after 10 epochs

References

1. S. Arora and M. P. S. Bhatia, "Handwriting recognition using Deep Learning in Keras," *2018 International Conference on Advances in Computing, Communication Control and Networking (ICACCCN)*, 2018, pp. 142-145, doi: 10.1109/ICACCCN.2018.8748540.
2. https://towardsdatascience.com/feed-forward-neural-networks-how-to-successfully-build-them-in-python-74503409d99a
3. https://pyimagesearch.com/2021/05/06/implementing-feedforward-neural-networks-with-keras-and-tensorflow/
4. https://exchange.scale.com/public/blogs/how-to-build-a-fully-connected-feedforward-neural-network-using-keras-and-tensorflow
5. https://www.kaggle.com/code/prashant111/mnist-deep-neural-network-with-keras/notebook
6. https://sanjayasubedi.com.np/deeplearning/tensorflow-2-first-neural-network-for-fashion-mnist/

Assignment No. 3

Problem Statement:

Build the Image classification model by dividing the model into following 4 stages:

1. Loading and preprocessing the image data
2. Defining the model's architecture
3. Training the model
4. Estimating the model's performance

Objective:

1. To be able to apply deep learning algorithms to solve problems of moderate complexity
2. Understand how a model is trained and evaluated.
3. Classifying images from the image dataset.
4. Our main goal is to train a neural network (using Keras) to obtain > 90% accuracy on image dataset..
5. To apply the algorithms to a real-world problem, optimize the models learned and report on the expected accuracy that can be achieved by applying the models

Outcomes: At the end of the assignment the students should able-

1. Learn and Use various Deep Learning tools and packages.
2. Build and train a deep Neural Network models for use in various applications.
3. Apply Deep Learning techniques like CNN, RNN Auto encoders to solve real word Problems.
4. Evaluate the performance of the model build using Deep Learning.

Solution Expected:

Implement and train a Convulutional neural network (CNN) on an hand-written digits image dataset called MNIST and improve model generalisation by achieving increased accuracy and decreased loss where model gains good confidence with the prediction.

Methodology to be used:

- Deep Learning
- Convolutional Neural Network

Infrastructure:

Desktop/ laptop system with Linux /Ubuntu 16.04 or higher (64-bit)/ Windows OS/Mac OS

Software used:

LINUX/ Windows OS/ Virtual Machine/ IOS, Anaconda distribution, Jupyter notebook, python 3.9.12

Theory:

Deep Learning has been proved that its a very powerful tool due to its ability to handle huge amounts of data. The use of hidden layers exceeds traditional techniques, especially for pattern recognition. One of the most popular Deep Neural Networks is Convolutional Neural Networks (CNN).

Convolutional Neural Networks (CNNs):

A convolutional neural network (CNN) is a type of Artificial Neural Network (ANN) used in image recognition and processing which is specially designed for processing data (pixels). The goal of a CNN is to learn higher-order features in the data via convolutions. They are well suited to object recognition with images and consistently top image classification competitions. They can identify faces, individuals, street signs, platypuses, and many other aspects of visual data. CNNs overlap with text analysis via optical character recognition, but they are also useful when analyzing words6 as discrete textual units. They're also good at analyzing sound. The efficacy of CNNs in image recognition is one of the main reasons why the world recognizes the power of deep learning. CNNs are good at building position and (somewhat) rotation invariant features from raw image data. CNNs are powering major advances in machine vision, which has obvious applications for self-driving cars, robotics, drones, and treatments for the visually impaired. The structure of image data allows us to change the architecture of a neural network in a way that we can take advantage of this structure. With CNNs, we can arrange the neurons in a three-dimensional structure for which we have the following:

Width

Height

Depth

These attributes of the input match up to an image structure for which we have:

Image width in pixels

Image height in pixels

RGB channels as the depth

We can consider this structure to be a three-dimensional volume of neurons. A significant aspect to how CNNs evolved from previous feed-forward variants is how they achieved computational efficiency with new layer types.

CNN Architecture Overview:

CNNs transform the input data from the input layer through all connected layers into a set of class scores given by the output layer. There are many variations of the CNN architecture, but they are based on the pattern of layers, as demonstrated in Figure 1 (below).

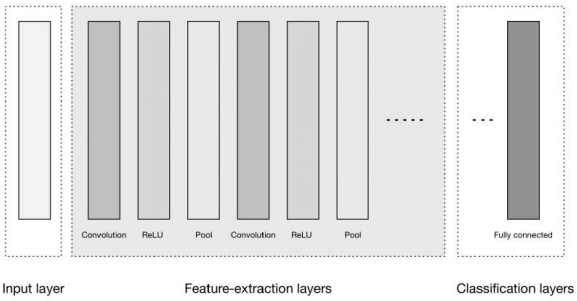


Figure 1: High-level general CNN architecture

Figure 1 depicts three major groups:

1. Input layer

- An image matrix (volume) of dimension **(h x w x d)**
- A filter **(f_h x f_w x d)**
- Outputs a volume dimension **(h - f_h + 1) x (w - f_w + 1) x 1**

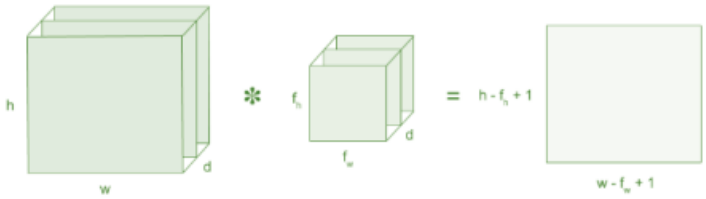


Figure 2: Image matrix multiplies kernel or filter matrix

The input layer accepts three-dimensional input generally in the form spatially of the size (width × height) of the image and has a depth representing the color channels (generally three for RGB color channels) as shown in fig 2 above.

2. Feature-extraction (learning) layers

The feature-extraction layers have a general repeating pattern of the sequence as shown in figure 1:

Convolution layer

We express the Rectified Linear Unit (ReLU) activation function as a layer in the diagram in figure 1. Convolutional layers transform the input data by using a patch of locally connecting neurons from the previous layer.

A. Pooling layer

These layers find a number of features in the images and progressively construct higher-order features. This corresponds directly to the ongoing theme in deep learning by which features are automatically learned as opposed to traditionally hand engineered.

3. Classification layers

Finally we have the classification layers in which we have one or more fully connected layers to take the higher-order features and produce class probabilities or scores. These layers are fully connected to all of the neurons in the previous layer, as their name implies. The output of these layers produces typically a two dimensional output of the dimensions [b × N], where b is the number of examples in the mini-batch and N is the number of classes we're interested in scoring.

Multilayer neural networks vs CNN:

In traditional multilayer neural networks, the layers are fully connected and every neuron in a layer is connected to every neuron in the next layer whereas The neurons in the layers of a CNN are arranged in three dimensions to match the input volumes. Here, depth means the third dimension of the activation volume, not the number of layers, as in a multilayer neural network.

Evolution of the connections between layers:

Another change is how we connect layers in a convolutional architecture. Neurons in a layer are connected to only a small region of neurons in the layer before it. CNNs retain a layer-oriented architecture, as in traditional multilayer networks, but have different types of layers. Each layer transforms the 3D input volume from the previous layer into a 3D output volume of neuron activations with some differentiable function that might or might not have parameters, as demonstrated in Figure 1.

Input Layers

Input layers are where we load and store the raw input data of the image for processing in the network. This input data specifies the width, height, and number of channels. Typically, the number of channels is three, for the RGB values for each pixel.

Convolutional Layers

Convolutional layers are considered the core building blocks of CNN architectures. As Figure 2 illustrates, convolutional layers transform the input data by using a patch of locally connecting neurons from the previous layer. The layer will compute a dot product between the region of the neurons in the input layer and the weights to which they are locally connected in the output layer.

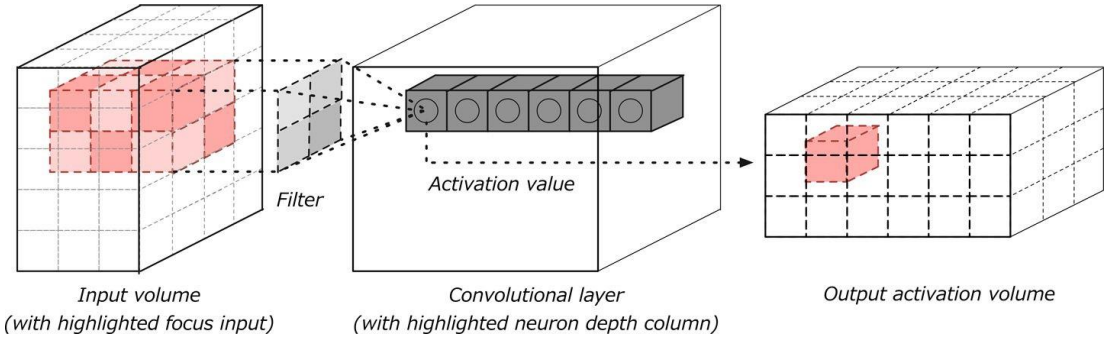


Figure 3: Convolution layer with input and output volumes

The resulting output generally has the same spatial dimensions (or smaller spatial dimensions) but sometimes increases the number of elements in the third dimension of the output (depth dimension).

TensorFlow is an open-source platform for machine learning. Keras is the high-level application programming interface (API) of TensorFlow. Using Keras, we can rapidly develop a prototype system and test it out. This is the first in a three-part series on using TensorFlow for supervised classification tasks.

STEPS: To implement Convolutional Neural Network for image classification

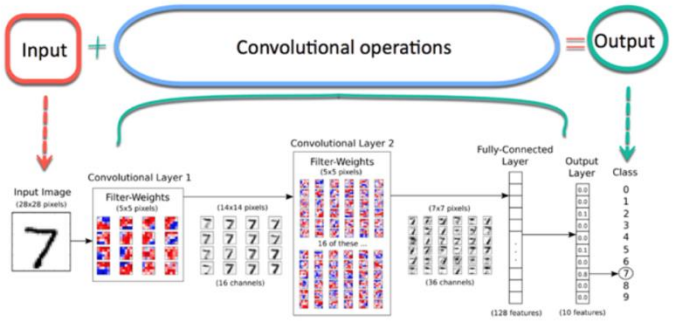


Figure 4: Image Classification model using CNN with python

CNN:

Now imagine there is an image of a bird, and you want to identify it whether it is really a bird or something other. The first thing you should do is feed the pixels of the image in the form of arrays to the input layer of the neural network (MLP networks used to classify such things). The hidden layers carry Feature Extraction by performing various calculations and operations. There are multiple hidden layers like the convolution, the ReLU, and the pooling layer that performs feature extraction from your image. So finally, there is a fully connected layer that you can see which identifies the exact object in the image. You can understand very easily from the following figure:

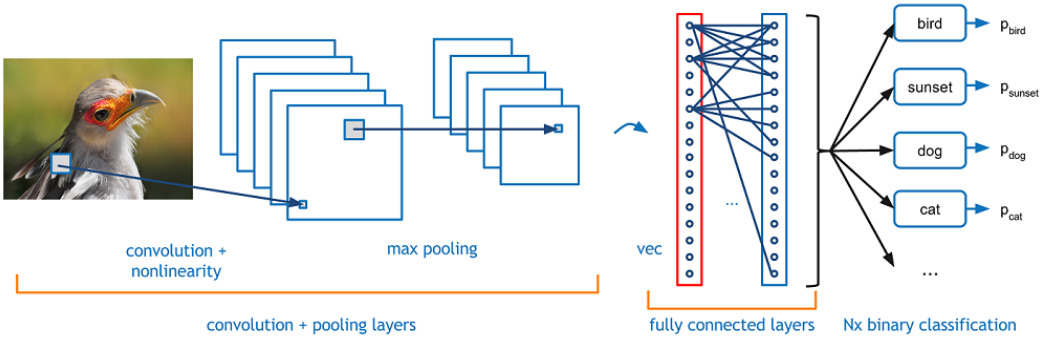


Figure 5: Convolution and pooling layers in CNN

Convolution:

Convolution Operation involves matrix arithmetic operations and every image is represented in the form of an array of values(pixels).

Let us understand example:

a = [2,5,8,4,7,9]

b = [1,2,3]

In Convolution Operation, the arrays are multiplied one by one element-wise, and the product is grouped or summed to create a new array that represents a*b.

The first three elements of matrix a are now multiplied by the elements of matrix b. The product is summed to get the result and stored in a new array of a*b.

This process remains continuous until the operation gets completed.

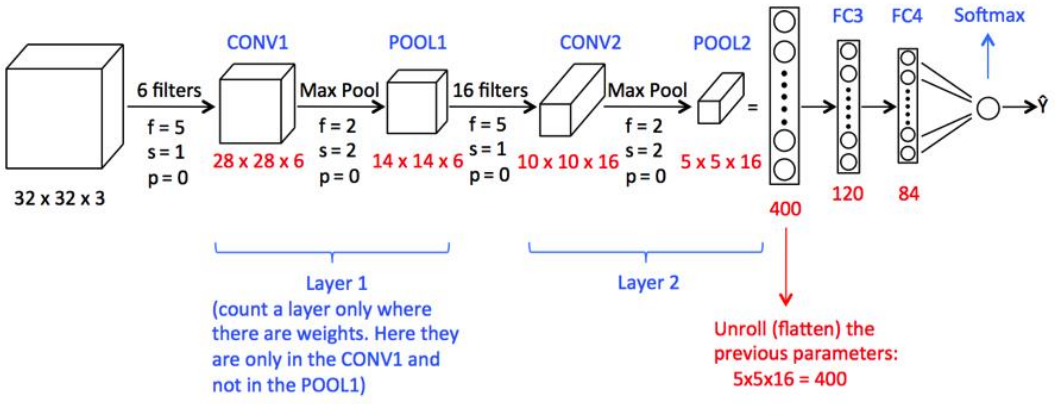


Figure 6: Sequence of Convolution and pooling layers in CNN

Pooling:

After the convolution, there is another operation called pooling. So, in the chain, convolution and pooling are applied sequentially on the data in the interest of extracting some features from the data. After the sequential convolutional and pooling layers, the data is flattened into a feed-forward neural network which is also called a Multi-Layer Perceptron.

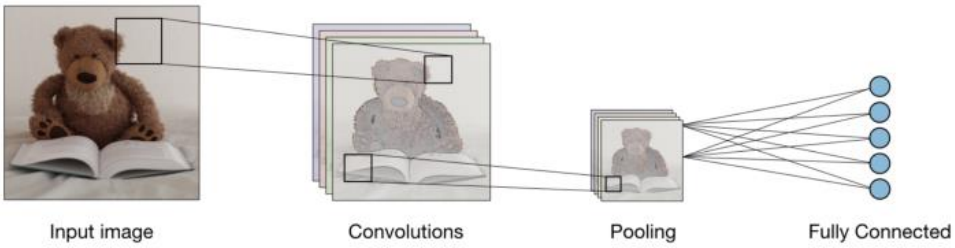


Figure 7: Data flattening into a feed-forward neural network

Thus, we have seen steps that are important for building CNN model.

Code Snippets:

#Importing Libraries

```
import numpy as np
import pandas as pd
import random
import tensorflow as tf
import matplotlib.pyplot as plt
from sklearn.metrics import accuracy_score
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Flatten, Conv2D, Dense, MaxPooling2D
from tensorflow.keras.optimizers import SGD
from tensorflow.keras.utils import to_categorical
from tensorflow.keras.datasets import mnist
```

 # Loading and preprocessing the image data

Loading the MNIST dataset is very simple using Keras. The dataset is so popular, that you can use access it through datasets.mnist and use the load_data() function to get a train and a test set.

The dataset contains 28x28 images showing handwritten digits.

(X_train, y_train), (X_test, y_test) = mnist.load_data()

If you print the shape the any of the sets you will see more information about the samples.

For example, printing the shape of the train set will get you (60000, 28, 28):

60000: This is the number of samples in the set.

28: This is the height of each image.

28: This is the width of each image.

So we have 60,000 28x28 images.

```
print(X_train.shape)
```

(60000, 28, 28)

Anytime you are using a neural network, you should pay special attention to the range of the input values you will be feeding it. In our case, each image is a matrix of 28x28 pixels.

Let's print the range of these values to understand what's the scale we are working with:

```
X_train[0].min(), X_train[0].max()
```

(0, 255)

The pixel values in our images are between 0 (black) to 255 (white).

Neural networks have a much easier time converging when they work with values that don't vary a lot in scale. It's a common practice to scale every input to fit a small range like 0 to 1 or -1 to 1.

We should do that here, and scale our pixels to a range that goes from 0 (black) to 1 (white).

Here is the formula to scale a value: scaled_value = (original_value – min) / (max – min). In our case, the minimum value is 0 and the maximum value is 255.

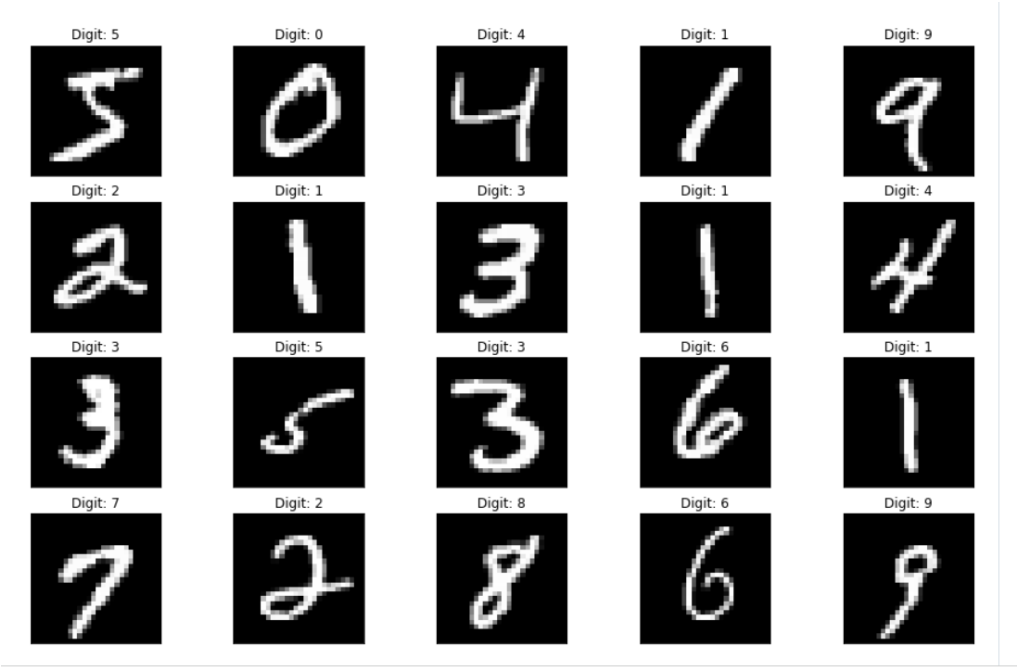
Let's use this to scale our train and test sets (notice that you can get rid of the 0.0 in the formula below but I'm leaving them there for clarity purposes):

```
X_train = (X_train - 0.0) / (255.0 - 0.0)
X_test = (X_test - 0.0) / (255.0 - 0.0)
X_train[0].min(), X_train[0].max()
```

(0.0, 1.0)

We can now plot the first 20 images on the train set:

```
def plot_digit(image, digit, plt, i):
    plt.subplot(4, 5, i + 1)
    plt.imshow(image, cmap=plt.get_cmap('gray'))
    plt.title(f"Digit: {digit}")
    plt.xticks([])
    plt.yticks([])
plt.figure(figsize=(16, 10))
for i in range(20):
    plot_digit(X_train[i], y_train[i], plt, i)
plt.show()
```



In Computer Vision, we usually use 4 dimensions to represent a set of images:

The total number of images (we call this "batch size")

The width of each image

The height of each image

The number of channels of each image

As you saw before, our train set has 3 dimensions only; we are missing the number of channels. We need to transform our data by adding that fourth dimension. Since these images are grayscale, that fourth dimension will be 1.

We can use numpy's reshape() function to reshape all of the data by adding that extra dimension.

```
X_train = X_train.reshape((X_train.shape + (1,)))
X_test = X_test.reshape((X_test.shape + (1,)))
```

Finally, let's take a look at the format of our target values (y_train). Let's print the first 20 samples in our train set:

```
y_train[0:20]
array([5, 0, 4, 1, 9, 2, 1, 3, 1, 4, 3, 5, 3, 6, 1, 7, 2, 8, 6, 9],
    dtype=uint8)
```

We are representing the target digits using integers (the digit 5 is represented with value 5, the digit 0 with value 0, etc.) This is important because it determines which loss function we should use to optimize our neural network.

We have two options:

Use integers for our target values (as they are now), and make sure we use the "Sparse Categorical Cross-Entropy" loss function.

One-hot encode the targets and use the "Categorical Cross-Entropy" loss function.

The easiest solution is to leave the targets as they are, so let's move on to creating the model.

Defining the model's architecture

There are several ways to create a model in Keras. In this example, we are going to use Kera's Sequential API because it's very simple to use.

Let's break down the definition of model below step by step:

First, we are going to define the first hidden layer of our network: A convolutional layer with 32 filters and a 3x3 kernel. This layer will use a ReLU activation function. The goal of this layer is to generate 32 different representations of an image, each one of 26x26. The 3x3 kernel will discard a pixel on each side of the original image and that's way we get 26x26 squares instead of 28x28.

Notice how we also need to define the input shape of the network as part of that first layer. Remember that our images are 28x28 with a single color channel, so that leads to the (28, 28, 1) shape.

Right after that first layer, we are going to do a 2x2 max pooling to downsample the amount of information generated by the convolutional layer. This operation will half the size of the filters. Remember we start with 32 filters of 26x26, so after this operation will have 32 filters of 13x13.

We then take the (13, 13, 32) vector and flatten it to a (5408,) vector. Notice that 13 x 13 x 32 = 5408.

Finally, we add a couple more fully-connected layers (also called Dense layers.) Notice how the output layer has size 10 (one for each of our possible digit values) and a softmax activation. Softmax ensures we get a probability distribution indicating the most likely digit in the image.

```
model = Sequential([
    Conv2D(32, (3, 3), activation="relu", input_shape=(28, 28, 1)),
    MaxPooling2D((2, 2)),
    Flatten(),
    Dense(100, activation="relu"),
    Dense(10, activation="softmax")
])
```

We now have our model. The next step is to define how we want to train it:

Let's use an SGD optimizer (Stochastic Gradient Descent) with 0.01 as the learning rate.

As we discussed before, we need to use the sparse_categorical_crossentropy loss because our target values

are represented as integers.

And we are going to compute accuracy of our model as we train it.

Notice in the summary of the model the shape of the vectors as they move through the layers we defined. They should look familiar after reading the explanation of our model above.

```
optimizer = SGD(learning_rate=0.01, momentum=0.9)
model.compile(
    optimizer=optimizer,
    loss="sparse_categorical_crossentropy",
    metrics=["accuracy"]
)
model.summary()
```

```
Model: "sequential_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
conv2d_1 (Conv2D)            (None, 26, 26, 32)        320
_________________________________________________________________
max_pooling2d_1 (MaxPooling2 (None, 13, 13, 32)        0
_________________________________________________________________
flatten_1 (Flatten)          (None, 5408)              0
_________________________________________________________________
dense_2 (Dense)              (None, 100)               540900
_________________________________________________________________
dense_3 (Dense)              (None, 10)                1010
=================================================================
Total params: 542,230
Trainable params: 542,230
Non-trainable params: 0
_________________________________________________________________
```

#Training and testing the model

At this point we are ready to fit our model on the train set.

For this example, we are going to run batches of 32 samples through our model for 10 iterations (epochs.)

This should be enough to get a model with good predictive capabilities.

Note: This network is fairly shallow so it shouldn't take a long time to train it on a CPU. If you have access to a GPU it should be much faster.

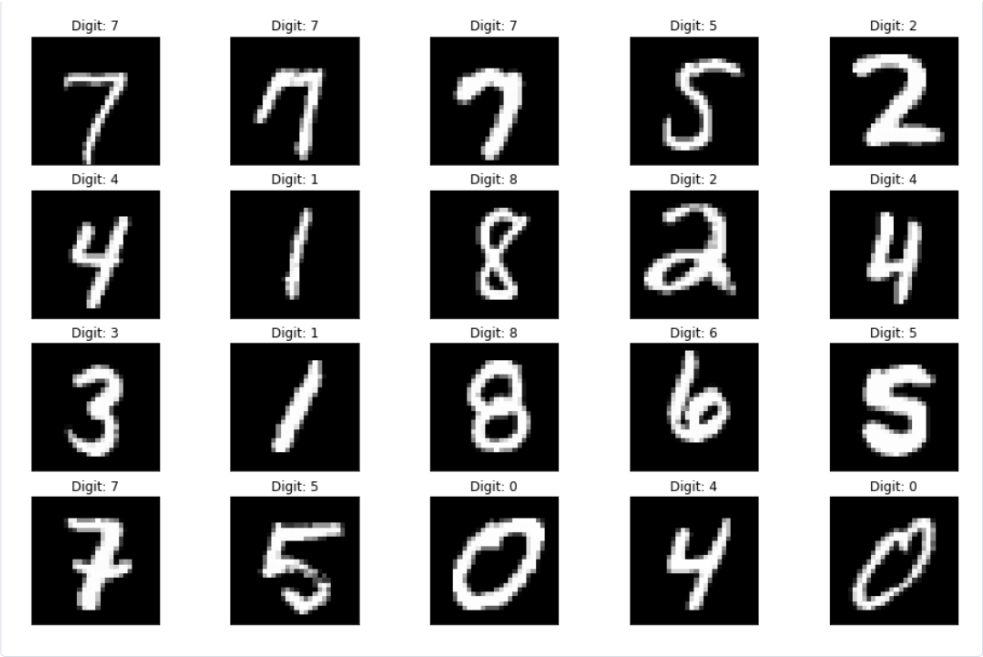
```
model.fit(X_train, y_train, epochs=10, batch_size=32)
Epoch 1/10
1875/1875 [==============================] - 7s 3ms/step - loss: 0.2412 - accuracy: 0.9252
```

```
Epoch 2/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0847 - accuracy: 0.9748
Epoch 3/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0535 - accuracy: 0.9831
Epoch 4/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0383 - accuracy: 0.9886
Epoch 5/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0288 - accuracy: 0.9909
Epoch 6/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0212 - accuracy: 0.9936
Epoch 7/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0163 - accuracy: 0.9951
Epoch 8/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0129 - accuracy: 0.9961
Epoch 9/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0092 - accuracy: 0.9973
Epoch 10/10
1875/1875 [==============================] - 6s 3ms/step - loss: 0.0072 - accuracy: 0.9983
<keras.callbacks.History at 0x7ff9e24c1a20>
```

At this point you should have a model that scored above 99% accuracy on the train set.

Now it's time to test it with a few of the images that we set aside on our test set. Let's run 20 random images through the model and display them together with the predicted digit:

```
plt.figure(figsize=(16, 10))
for i in range(20):
    image = random.choice(X_test).squeeze()
    digit = np.argmax(model.predict(image.reshape((1, 28, 28, 1)))[0], axis=-1)
    plot_digit(image, digit, plt, i)
        plt.show()
```



The results look pretty good!

To get a much better idea about the quality of the predictions, we can run the entire test set (10,000 images) through the model and compute the final accuracy. To do this we can use the accuracy_score() function from SciKit-Learn passing a couple of arguments:

True values: The correct digit expected for each image. These are the values we have stored in y_test.

Predicted values: The predictions that our model made. These are the results of our model.

The final accuracy will be the value printed after running the cell.

predictions = np.argmax(model.predict(X_test), axis=-1)

accuracy_score(y_test, predictions)

0.9855

 # Estimating the model's performance

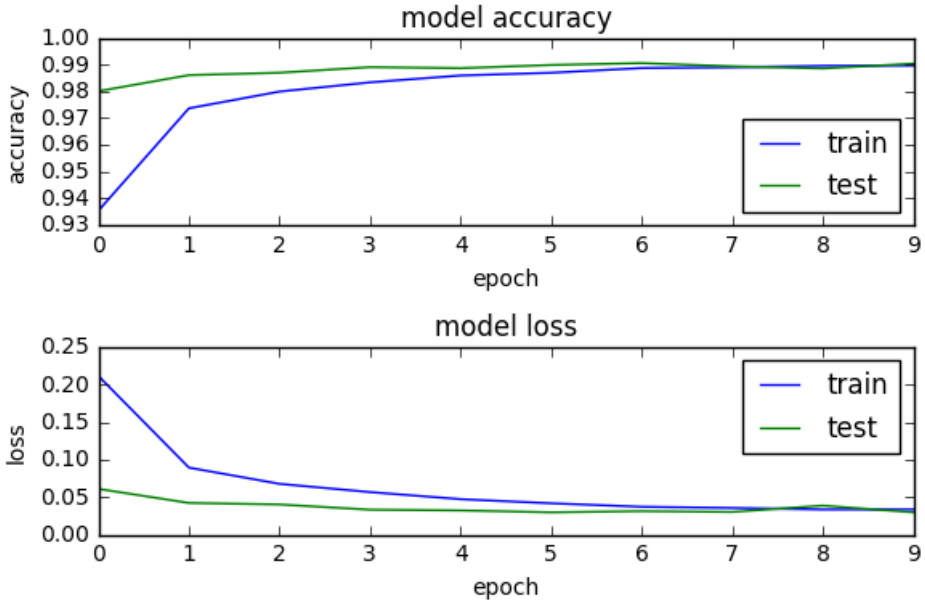
Now the trained model needs to be evaluated in terms of performance.

```
score = model.evaluate(X_test, y_test, verbose=0)
print('Test loss:', score[0]) #Test loss: 0.0296396646054
print('Test accuracy:', score[1]) #Test accuracy: 0.9904
```

Test accuracy 99%+ implies the model is trained well for prediction. If we visualize the whole training log, then with more number of epochs the loss and accuracy of the model on training and testing data converged

thus making the model a stable one.

```
import os

# plotting the metrics

fig = plt.figure()

plt.subplot(2,1,1)

plt.plot(model_log.history['acc'])

plt.plot(model_log.history['val_acc'])

plt.title('model accuracy')

plt.ylabel('accuracy')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='lower right')plt.subplot(2,1,2)

plt.plot(model_log.history['loss'])

plt.plot(model_log.history['val_loss'])

.title('model loss')

plt.ylabel('loss')

plt.xlabel('epoch')

plt.legend(['train', 'test'], loc='upper right')plt.tight_layout()fig
```



Saving the model to disk for reuse

Now, the trained model needs to be serialized. The architecture or structure of the model will be stored in a json file and the weights will be stored in hdf5 file format.

```
#Save the model
# serialize model to JSON
model_digit_json = model.to_json()
with open("model_digit.json", "w") as json_file:
    json_file.write(model_digit_json)
# serialize weights to HDF5
model.save_weights("model_digit.h5")
print("Saved model to disk")
```

Hence the saved model can be reused later or easily ported to other environments too

Conclusion:

Thus, we have implemented the Image classification model using CNN. With above code we can see that sufficient accuracy has been met. Throughout the epochs, our model accuracy increases and loss decreases that is good since our model gains confidence with our prediction

This indicates the model is trained in a good way

1. The loss is decreasing and the accuracy is increasing with every epoch.
2. The test accuracy is the measure of how good the model is predicting so, it is observed that the model is well trained after 10 epochs

References:

1. https://www.analyticsvidhya.com/blog/2021/06/image-classification-using-convolutional-neural-network-with-python/
2. Josh Patterson, Adam Gibson "Deep Learning: A Practitioner's Approach", O'ReiIIy Media, 2017
3. https://deepnote.com/@svpino/MNIST-with-Convolutional-Neural-Networks-a4b3c412-b802-4185-9806-02640fbba02e
4. https://towardsdatascience.com/a-simple-2d-cnn-for-mnist-digit-recognition-a998dbc1e79a

Assignment No. 4

Problem Statement:

Use Autoencoder to implement anomaly detection.
Build the model by using

a. Import required libraries

b. Upload/access the dataset

c. Encoder converts it into latent representation

d. Decoder networks convert it back to the original input

e. Compile the models with Optimizer, Loss, and Evaluation

Solution Expected:

AutoEncoders are widely used in anomaly detection. The reconstruction errors are used as the anomaly scores. Let us look at how we can use AutoEncoder for anomaly detection using TensorFlow.

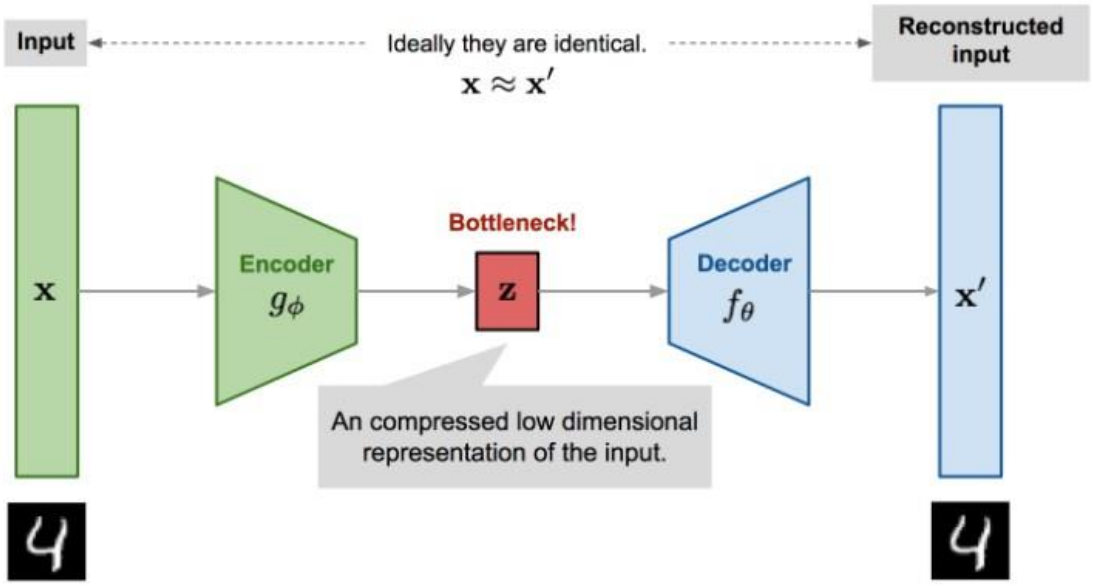
Import the required libraries and load the data. Here we are using the ECG data which consists of labels 0 and 1. Label 0 denotes the observation as an anomaly and label 1 denotes the observation as normal.

Objectives to be achieved:

1)Use Autoencoder to implement anomaly detection.

Methodology to be used:

AutoEncoder is a generative unsupervised deep learning algorithm used for reconstructing high-dimensional input data using a neural network with a narrow bottleneck layer in the middle which contains the latent representation of the input data.



Import required libraries

```
import pandas as pd
import numpy as np
import tensorflow as tf
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler
from sklearn.metrics import confusion_matrix, recall_score,
accuracy_score, precision_score

RANDOM_SEED = 2021
TEST_PCT = 0.3
LABELS = ["Normal","Fraud"]
```

Read the dataset

I had downloaded the data from Kaggle and stored it in the local directory.

```
dataset = pd.read_csv("creditcard.csv")
```

Exploratory Data Analysis

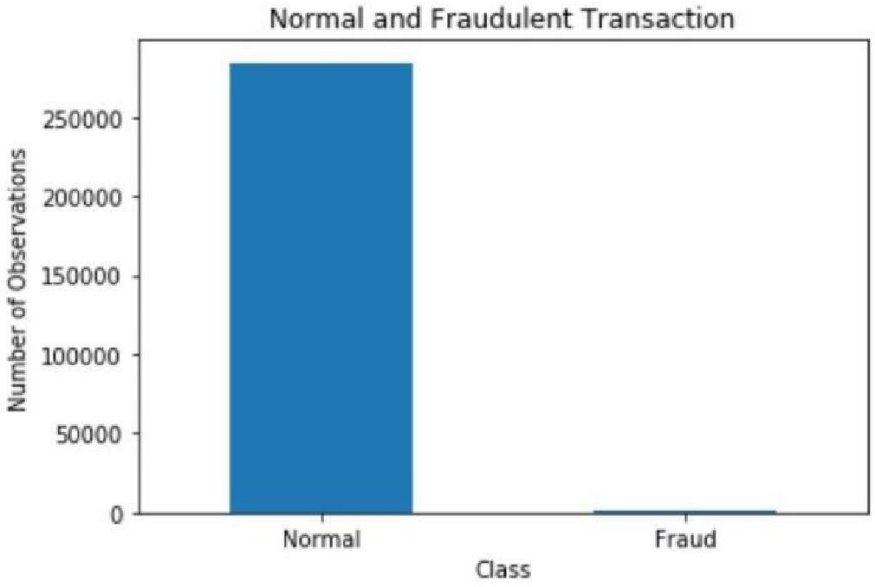
```
#check for any  nullvalues
print("Any nulls in the dataset
",dataset.isnull().values.any() )
print('-------')
print("No. of unique labels ",
len(dataset['Class'].unique()))
print("Label values
",dataset.Class.unique())

#0 is for normal credit card transaction
#1 is for fraudulent credit card
transaction
print('-------')
print("Break down of the Normal and Fraud
Transactions")
print(pd.value_counts(dataset['Class'],
sort = True) )
```

```
Any nulls in the dataset  False
-------
No. of unique labels  2
Label values  [0 1]
-------
Break down of the Normal and Fraud Transactions
0    284315
1       492
Name: Class, dtype: int64
```

Visualize the dataset

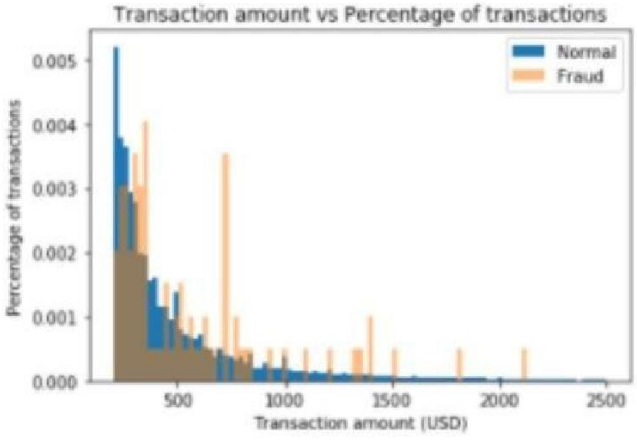
plotting the number of normal and fraud transactions in the dataset.

```
#Visualizing the imbalanced dataset
count_classes =
pd.value_counts(dataset['Class'], sort =
True)
count_classes.plot(kind = 'bar', rot=0)
plt.xticks(range(len(dataset['Class'].uni
que())), dataset.Class.unique())
plt.title("Frequency by observation
number")
plt.xlabel("Class")
plt.ylabel("Number of Observations");
```



Visualizing the amount for normal and fraud transactions.

```
# Save the normal and fradulent
transactions in separate dataframe
normal_dataset = dataset[dataset.Class ==
0]
fraud_dataset = dataset[dataset.Class ==
1]

#Visualize transactionamounts for normal
and fraudulent transactions
bins = np.linspace(200, 2500, 100)
plt.hist(normal_dataset.Amount,
bins=bins, alpha=1, density=True,
label='Normal')
plt.hist(fraud_dataset.Amount, bins=bins,
alpha=0.5, density=True, label='Fraud')
plt.legend(loc='upper right')
plt.title("Transaction amount vs
Percentage of transactions")
plt.xlabel("Transaction amount (USD)")
plt.ylabel("Percentage of transactions");
plt.show()
```



Create train and test dataset

Checking on the dataset

V6	V7	V8	V9	V10	...	V22	V23	V24	V25	V26	V27	V28	Time	Amount	Class
0.462388	0.239599	0.098698	0.363787	0.090794	...	0.277838	-0.110474	0.066928	0.128539	-0.189115	0.133558	-0.021053	0.0	149.62	0
-0.082361	-0.078803	0.085102	-0.255425	-0.166974	...	-0.638672	0.101288	-0.339846	0.167170	0.125895	-0.008983	0.014724	0.0	2.69	0
1.800499	0.791461	0.247676	-1.514654	0.207643	...	0.771679	0.909412	-0.689281	-0.327642	-0.139097	-0.055353	-0.059752	1.0	378.66	0
1.247203	0.237609	0.377436	-1.387024	-0.054952	...	0.005274	-0.190321	-1.175575	0.647376	-0.221929	0.062723	0.061458	1.0	123.50	0
0.095921	0.592941	-0.270533	0.817739	0.753074	...	0.798278	-0.137458	0.141267	-0.206010	0.502292	0.219422	0.215153	2.0	69.99	0

Time and Amount are the columns that are not scaled, so applying StandardScaler to only Amount and Time columns. Normalizing the values between 0 and 1 did not work great for the dataset.

```
sc=StandardScaler()
dataset['Time'] =
sc.fit_transform(dataset['Time'].values.r
eshape(-1, 1))
dataset['Amount'] =
sc.fit_transform(dataset['Amount'].values
.reshape(-1, 1))
```

The last column in the dataset is our target variable.

The last column in the dataset is our target variable.

```
raw_data = dataset.values
# The last element contains if the
transaction is normal which is
represented by a 0 and if fraud then 1
labels = raw_data[:, -1]

# The other data points are the
electrocadriogram data
data = raw_data[:, 0:-1]

train_data, test_data, train_labels,
test_labels = train_test_split(
    data, labels, test_size=0.2,
random_state=2021
)
```

Normalize the data to have a value between 0 and 1

Normalize the data to have a value between 0 and 1

```
min_val = tf.reduce_min(train_data)
max_val = tf.reduce_max(train_data)

train_data = (train_data - min_val) /
(max_val - min_val)
test_data = (test_data - min_val) /
(max_val - min_val)

train_data = tf.cast(train_data,
tf.float32)
test_data = tf.cast(test_data,
tf.float32)
```

Use only normal transactions to train the Autoencoder.

Normal data has a value of 0 in the target variable. Using the target variable to create a normal and fraud dataset.

```
train_labels = train_labels.astype(bool)
test_labels = test_labels.astype(bool)

#creating normal and fraud datasets
normal_train_data =
train_data[~train_labels]
normal_test_data =
test_data[~test_labels]

fraud_train_data =
train_data[train_labels]
fraud_test_data = test_data[test_labels]
print(" No. of records in Fraud Train
Data=",len(fraud_train_data))
print(" No. of records in Normal Train
data=",len(normal_train_data))
print(" No. of records in Fraud Test
Data=",len(fraud_test_data))
print(" No. of records in Normal Test
data=",len(normal_test_data))
```

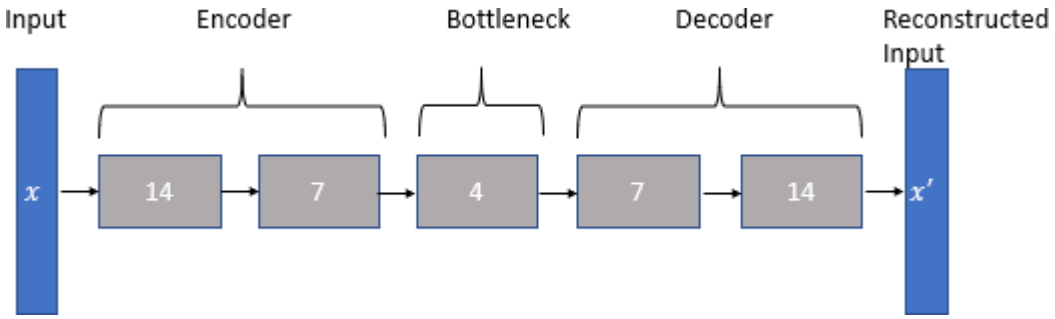
Set the training parameter values

Set the training parameter values

```
nb_epoch = 50
batch_size = 64
input_dim = normal_train_data.shape[1]
#num of columns, 30
encoding_dim = 14
hidden_dim_1 = int(encoding_dim / 2) #
hidden_dim_2=4
learning_rate = 1e-7
```

Create the Autoencoder

The architecture of the autoencoder is shown below.




```
#input Layer
input_layer =
tf.keras.layers.Input(shape=(input_dim,
))

#Encoder

encoder =
tf.keras.layers.Dense(encoding_dim,
activation="tanh",
activity_regularizer=tf.keras.regularizer
s.l2(learning_rate))(input_layer)
encoder=tf.keras.layers.Dropout(0.2)
(encoder)
encoder =
tf.keras.layers.Dense(hidden_dim_1,
activation='relu')(encoder)
encoder =
tf.keras.layers.Dense(hidden_dim_2,
activation=tf.nn.leaky_relu)(encoder)

# Decoder
decoder =
tf.keras.layers.Dense(hidden_dim_1,
activation='relu')(encoder)
decoder=tf.keras.layers.Dropout(0.2)
(decoder)
decoder =
tf.keras.layers.Dense(encoding_dim,
activation='relu')(decoder)
decoder =
tf.keras.layers.Dense(input_dim,
activation='tanh')(decoder)

#Autoencoder
autoencoder =
tf.keras.Model(inputs=input_layer,
outputs=decoder)
autoencoder.summary()
```

```
Model: "functional_1"
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
input_1 (InputLayer)         [(None, 30)]              0
_________________________________________________________________
dense (Dense)                (None, 14)                434
_________________________________________________________________
dropout (Dropout)            (None, 14)                0
_________________________________________________________________
dense_1 (Dense)              (None, 7)                 105
_________________________________________________________________
dense_2 (Dense)              (None, 4)                 32
_________________________________________________________________
dense_3 (Dense)              (None, 7)                 35
_________________________________________________________________
dropout_1 (Dropout)          (None, 7)                 0
_________________________________________________________________
dense_4 (Dense)              (None, 14)                112
_________________________________________________________________
dense_5 (Dense)              (None, 30)                450
=================================================================
Total params: 1,168
Trainable params: 1,168
Non-trainable params: 0
```

Define the callbacks for checkpoints and early stopping

```
cp =
tf.keras.callbacks.ModelCheckpoint(filepa
th="autoencoder_fraud.h5",

mode='min', monitor='val_loss',
verbose=2, save_best_only=True)
# define our early stopping
early_stop =
tf.keras.callbacks.EarlyStopping(
    monitor='val_loss',
    min_delta=0.0001,
    patience=10,
    verbose=1,
    mode='min',
    restore_best_weights=True
```

Plot training and test loss

Compile the Autoencoder

```
autoencoder.compile(metrics=['accuracy'],

loss='mean_squared_error',
                    optimizer='adam')
```

Train the Autoencoder

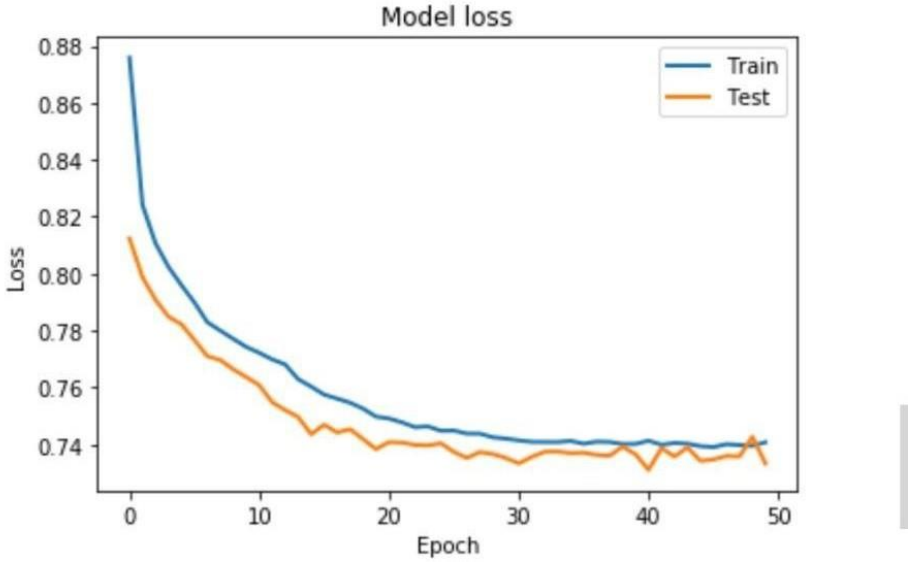
```
history =
autoencoder.fit(normal_train_data,
normal_train_data,
                    epochs=nb_epoch,

batch_size=batch_size,
                    shuffle=True,
                    validation_data=
(test_data, test_data),
                    verbose=1,
                    callbacks=[cp,
early_stop]
                    ).history
```

```
Epoch 11/25
7085/7108 [============================>.] - ETA: 0s - loss: 5.3796e-04
Epoch 00011: val_loss did not improve from 0.00053
Restoring model weights from the end of the best epoch.
7108/7108 [==============================] - 8s 1ms/step - loss: 5.3799e-04 - val_loss: 5.3531e-04
Epoch 00011: early stopping
```

Plot training and test loss

Plot training and test loss

```
plt.plot(history['loss'], linewidth=2,
label='Train')
plt.plot(history['val_loss'],
linewidth=2, label='Test')
plt.legend(loc='upper right')
plt.title('Model loss')
plt.ylabel('Loss')
plt.xlabel('Epoch')
#plt.ylim(ymin=0.70,ymax=1)
plt.show()
```



Detect Anomalies on test data

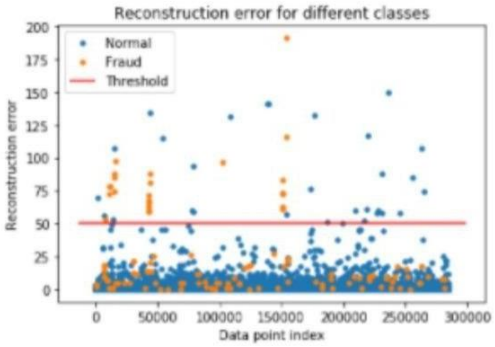
Anomalies are data points where the reconstruction loss is higher.

To calculate the reconstruction loss on test data, predict the test data and calculate the mean square error between the test data and the reconstructed test data.

```
test_x_predictions =
autoencoder.predict(test_data)
mse = np.mean(np.power(test_data -
test_x_predictions, 2), axis=1)
error_df =
pd.DataFrame({'Reconstruction_error':
mse,
                          'True_class':
test_labels})
```

Plotting the test data points and their respective reconstruction error sets a threshold value to visualize if the threshold value needs to be adjusted.

```
threshold_fixed = 50
groups = error_df.groupby('True_class')
fig, ax = plt.subplots()

for name, group in groups:
    ax.plot(group.index,
group.Reconstruction_error, marker='o',
ms=3.5, linestyle='',
            label= "Fraud" if name == 1
else "Normal")
ax.hlines(threshold_fixed, ax.get_xlim()
[0], ax.get_xlim()[1], colors="r",
zorder=100, label='Threshold')
ax.legend()
plt.title("Reconstruction error for
normal and fraud data")
plt.ylabel("Reconstruction error")
plt.xlabel("Data point index")
plt.show();
```



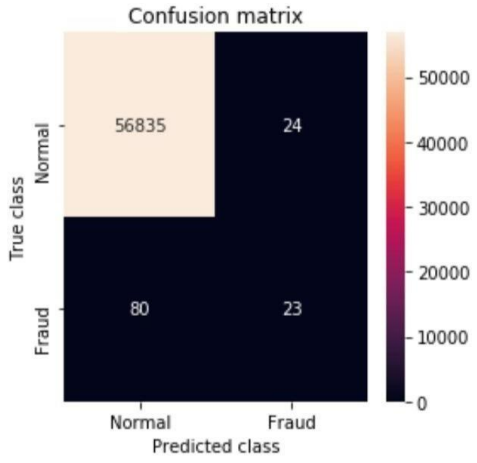
Detect anomalies as points where the reconstruction loss is greater than a fixed threshold. Here we see that a value of 52 for the threshold will be good.

Evaluating the performance of the anomaly detection

```
threshold_fixed =52
pred_y = [1 if e > threshold_fixed else 0
for e in
error_df.Reconstruction_error.values]
error_df['pred'] =pred_y
conf_matrix =
confusion_matrix(error_df.True_class,
pred_y)

plt.figure(figsize=(4, 4))
sns.heatmap(conf_matrix,
xticklabels=LABELS, yticklabels=LABELS,
annot=True, fmt="d");
plt.title("Confusion matrix")
plt.ylabel('True class')
plt.xlabel('Predicted class')
plt.show()

# print Accuracy, precision and recall
print(" Accuracy:
",accuracy_score(error_df['True_class'],
error_df['pred']))
print(" Recall:
",recall_score(error_df['True_class'],
error_df['pred']))
print(" Precision:
",precision_score(error_df['True_class'],
error_df['pred']))
```

Evaluating the performance of the anomaly detection

```
threshold_fixed =52
pred_y = [1 if e > threshold_fixed else 0
for e in
error_df.Reconstruction_error.values]
error_df['pred'] =pred_y
conf_matrix =
confusion_matrix(error_df.True_class,
pred_y)

plt.figure(figsize=(4, 4))
sns.heatmap(conf_matrix,
xticklabels=LABELS, yticklabels=LABELS,
annot=True, fmt="d");
plt.title("Confusion matrix")
plt.ylabel('True class')
plt.xlabel('Predicted class')
plt.show()

# print Accuracy, precision and recall
print(" Accuracy:
",accuracy_score(error_df['True_class'],
error_df['pred']))
print(" Recall:
",recall_score(error_df['True_class'],
error_df['pred']))
print(" Precision:
",precision_score(error_df['True_class'],
error_df['pred']))
```




```
Accuracy:  0.9981742214107651
Recall:  0.22330097087378642
Precision:  0.48936170212765956
```

As our dataset is highly imbalanced, we see a high accuracy but a low recall and precision. Things to further improve precision and recall would add more relevant features, different architecture

for autoencoder, different hyperparameters, or a different algorithm.

Conclusion:

Autoencoders can be used as an anomaly detection algorithm when we have an unbalanced dataset where we have a lot of good examples and only a few anomalies. Autoencoders are trained to minimize reconstruction error. When we train the autoencoders on normal data or good data, we can hypothesize that the anomalies will have higher reconstruction errors than the good or normal data.

Assignment No.5

Title: Implement the Continuous Bag of Words (CBOW) Model

Aim: Implement the Continuous Bag of Words (CBOW) Model. Stages can be:

a. Data preparation

b. Generate training data

c. Train model

d. Output

Theory :

1)What is NLP ?

2) What is Word embedding related to NLP ?

3) Explain Word2Vec techniques.

4) Enlist applications of Word embedding in NLP.

5) Explain CBOW architecture.

6) What will be input to CBOW model and Output to CBW model.

7) What is Tokenizer .

8) Explain window size parameter in detail for CBOW model.

9) Explain Embedding and Lmbda layer from keras

10) What is yield()

Steps/ Algorithm

1. Dataset link and libraries :

Create any English 5 to 10 sententece paragraph

as inputImport following data from keras :

keras.models import Sequential

keras.layers import Dense, Embedding, Lambda

keras.utils import np_utils

keras.preprocessing import sequence

keras.preprocessing.text import Tokenizer

Import Gensim for NLP operations : requirements :

Gensim runs on Linux, Windows and Mac OS X, and should run on any other platform that supports Python 3.6+ and NumPy. Gensim depends on the following software: Python, testedwith versions 3.6, 3.7 and 3.8. NumPy for number crunching.

Ref: https://analyticsindiamag.com/the-continuous-bag-of-words-cbow-model-in-nlp-hands-on-implementation-with-codes/

a) Import following libraries gemsim and numpy set i.e. text file created . It should bepreprocessed.

b) Tokenize the every word from the paragraph . You can call in built tokenizer present inGensim

c) Fit the data to tokenizer

d) Find total no of words and total no of sentences.

e) Generate the pairs of Context words and target words :

```
e.g. cbow_model(data, window_size,
   total_vocab):total_length =
   window_size*2
   for text in data:
      text_len =
      len(text)
      for idx, word in
         enumerate(text):
         context_word = []
         target  = []
         begin = idx -
  window_size end = idx +
  window_size + 1
context_word.append([text[i] for i in range(begin, end) if 0 <= i < text_len and i
 != idx])
            target.append(word)
            contextual = sequence.pad_sequences(context_word, total_length=total_length)
            final_target = np_utils.to_categorical(target, total_vocab)
            yield(contextual, final_target)
```

f) Create Neural Network model with following parameters . Model type : sequential

Layers : Dense , Lambda , embedding. Compile Options : (loss='categorical_crossentropy', optimizer='adam')

g) Create vector file of some word for

testinge.g.:dimensions=100

```
vect_file = open('/content/gdrive/My Drive/vectors.txt' ,'w')

vect_file.write('{} {}\n'.format(total_vocab,dimensions)
```

h) Assign weights to your trained model

```
e.g. weights = model.get_weights()[0]

for text, i in vectorize.word_index.items():

final_vec = ' '.join(map(str, list(weights[i, :])))

vect_file.write('{} {}\n'.format(text, final_vec)

Close()
```

i) Use the vectors created in Gemsim :

e.g. cbow_output = gensim.models.KeyedVectors.load_word2vec_format('/content/gdrive/My Drive/vectors.txt', binary=False)

j) choose the word to get similar type of

words:

```
cbow_output.most_similar(positive=['Your

word'])
```

Conclusion: In this experiment, we saw what a CBOW model is and how it works. We also implemented the model on a custom dataset and got good output. We learnt what word embeddings are and how CBOW is useful. These can be used for text recognition, speech to text conversion etc.

Sample Code with comments: Attach Printout with Output .

Assignment No.6

Title: Object detection using Transfer Learning of CNN architectures

Aim: Object detection using Transfer Learning of CNN architectures

a. Load in a pre-trained CNN model trained on a large dataset

b. Freeze parameters (weights) in model's lower convolutional layers

c. Add custom classifier with several layers of trainable parameters to model

d. Train classifier layers on training data available for task

e. Fine-tune hyper parameters and unfreeze more layers as needed

Theory:

1)What is Transfer learning ?

2) What are pretrained Neural Network models ?

3) Explain Pytorch library in short.

4) What are advantages of Transfer learning.

5) What are applications of Transfer learning.

6) Explain Caltech 101 images dataset.

7) Explain Imagenet dataset **.**

8) List down basic steps for transfer learning.

9) What is Data augmentation?

10) How and why Data augmentation is done related to transfer learning?

11) Why preprocessing is needed on inputdata in Transfer learning.

12) What is PyTorch Transforms module.Explain following commands w.r.t it :

Compose([RandomResizedCrop(size=256, scale=(0.8, 1.0)),RandomRotation(degrees=15),

ColorJitter(),

RandomHorizontalFlip(),

CenterCrop(size=224), # Image net standards

.ToTensor(),

Normalize

13) Explain the Validation Transforms steps with Pytorch Transforms.

14) Explain VGG-16 model from Pytorch

Steps/ Algorithm

1. Dataset link and libraries :

https://data.caltech.edu/records/mzrjq-6wc02

separate the data into training, validation, and testing sets with a 50%, 25%, 25% split

andthen structured the directories as follows:

/datadir

/train

/class1

/class2

.

.

/valid

/class1

/class2

.

.

/test

/class1

/class2

.

Libraries required :

PyTorch

torchvision import transforms

torchvision import datasets

torch.utils.data import

DataLoadertorchvision

import models torch.nn as nn

torch import optim

Ref: https://towardsdatascience.com/transfer-learning-with-convolutional-neural-networks-in-pytorch-dd09190245ce

1) Prepare the dataset in splitting in three directories Train , alidation and test with 50 25 25
2) Do pre-processing on data with transform from PytorchTraining dataset transformation as follows : transforms.Compose([

```
transforms.RandomResizedCrop(size=256,
scale=(0.8, 1.0)),
transforms.RandomRotation(degrees=15),
transforms.ColorJitter(),
transforms.RandomHorizontalFlip(),
transforms.CenterCrop(size=224), # Image net
standards transforms.ToTensor(),
transforms.Normalize([0.485, 0.456, 0.406],
                     [0.229, 0.224, 0.225]) # Imagenet
```

standardsValidation Dataset transform as follows :

```
transforms.Compose([
transforms.Resize(size=256)
,
transforms.CenterCrop(size=
224),transforms.ToTensor(),
transforms.Normalize([0.485, 0.456, 0.406], [0.229, 0.224, 0.225])
```

3) Create Datasets and Loaders :

```
data = {
  'train':(Our name given to train data set dir created )
  datasets.ImageFolder(root=traindir,
  transform=image_transforms['train']),'valid':
  datasets.ImageFolder(root=validdir, transform=image_transforms['valid']),
}
 dataloaders = {
  'train': DataLoader(data['train'], batch_size=batch_size,
  shuffle=True),'val': DataLoader(data['valid'],
  batch_size=batch_size, shuffle=True)
```

}

4) Load Pretrain Model : from torchvision import models

model = model.vgg16(pretrained=True)

5) Freez all the Models Weight

for param in

model.parameters():

param.requires_grad =

False

6) Add our own custom classifier with following parameters :

Fully connected with ReLU activation, shape = (n_inputs,

256)Dropout with 40% chance of dropping

Fully connected with log softmax output, shape = (256,

n_classes)import torch.nn as nn

Add on classifier

model.classifier[6] =

nn.Sequential(

nn.Linear(n_inputs,

256),nn.ReLU(),

nn.Dropout(0.4),

nn.Linear(256,

n_classes),

nn.LogSoftmax(dim

=1))

7) Only train the sixth layer of classifier keep remaining layers

off .Sequential(

(0): Linear(in_features=25088, out_features=4096,

bias=True)(1): ReLU(inplace)

(2): Dropout(p=0.5)

(3): Linear(in_features=4096, out_features=4096,

bias=True)(4): ReLU(inplace)

```
(5): Dropout(p=0.5)
(6): Sequential(
  (0): Linear(in_features=4096, out_features=256,
  bias=True)(1): ReLU()
  (2): Dropout(p=0.4)
  (3): Linear(in_features=256, out_features=100,
  bias=True)(4): LogSoftmax()
 )
)
```

8) Initialize the loss and optimizercriteration = nn.NLLLoss()
   optimizer = optim.Adam(model.parameters())
9) Train the model using Pytorch for epoch in range(n_epochs): for data, targets in trainloader:

```
    # Generate
    predictionsout =
    model(data)
    # Calculate loss
    loss = criterion(out,
    targets)#
    Backpropagation
    loss.backward()
    # Update model
    parameters
    optimizer.step()
```

10) Perform Early stopping
11) Draw performance curve

12) Calculate Accuracy

```
pred = torch.max(ps,
dim=1)equals = pred
== targets
# Calculate accuracy
accuracy = torch.mean(equals)
```

Conclusion: In this experiment, we were able to see the basics of using PyTorch as well as the concept of transfer learning, an effective method for object recognition. Instead of training a model from scratch, we can use existing architectures that have been trained on a large dataset and then tune them for our task. This reduces the time to train and often results in better overall performance. The outcome of this experiment is knowledge of transfer learning and PyTorch that we can build on to build more complex applications.

https://www.google.com/url?q=https://towardsdatascience.com/transfer-learning-with- convolutional-neural-networks-in-pytorch-dd0

Sample Code with comments: Attach Printout with Output.

```
# example of using a pre-trained model as a classifier
from tensorflow.keras.utils import load_img
from tensorflow.keras.utils import img_to_array
from keras.applications.vgg16 import preprocess_input
from keras.applications.vgg16 import decode_predictions
from keras.applications.vgg16 import VGG16
# load an image from file
image = load_img('dog.jpg', target_size=(224, 224))
# convert the image pixels to a numpy array
image = img_to_array(image)
# reshape data for the model
```

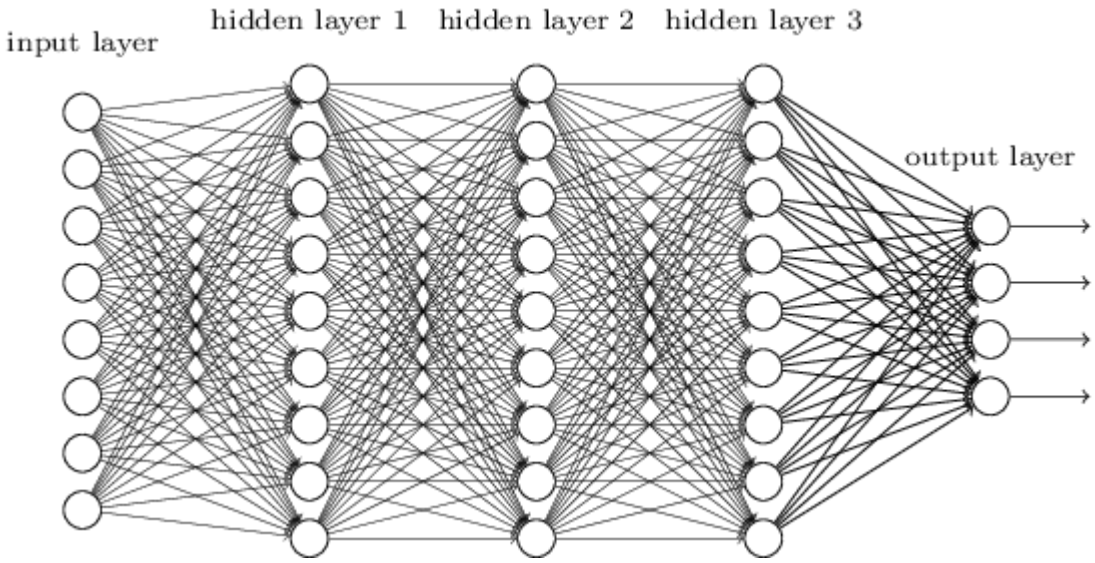
```
image = image.reshape((1, image.shape[0], image.shape[1], image.shape[2]))
# prepare the image for the VGG model
image = preprocess_input(image)
# load the model
model = VGG16()
# predict the probability across all output classes
yhat = model.predict(image)
# convert the probabilities to class labels
label = decode_predictions(yhat)
# retrieve the most likely result, e.g. highest probability
label = label[0][0]
# print the classification
print('%s (%.2f%%)' % (label[1], label[2]*100))
```

Extra Assignment

Title: Calculate output in a multi-layer feed forward network

Aim: To design layers required to build both a feed-forward neural network (also popularly known as a Multilayer Perceptron classifier) and the Convolution Neural Networks.

Theory :



An artificial neural network with 3 hidden layers

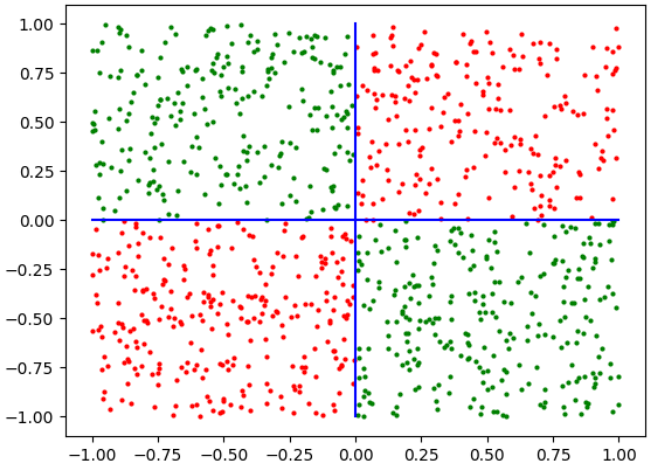
Data Sets

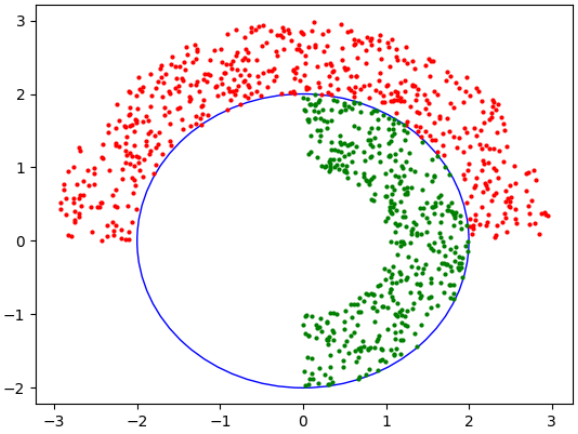
Link to data- https://www.cse.iitb.ac.in/~shivaram/teaching/old/cs337+335-s2019/resources/la-2/Lab2_base.tar.gz

To test code, provided a set of toy examples (1 and 2 below), each a 2-dimensional, 2-class problem and tools to visualise the performance of the neural networks which will train on these problems. The plots show the true separating boundary in blue. Each data set has a total of 10,000 examples, divided into a training set of 8000 examples, validation set of 1000 examples and test set of 1000 examples.

Data set 3 is the MNIST data set collection of images handwritten digits.

Data set 4 is the CIFAR-10 data set collection of images of commonly seen things segregated into 10 classes.





XOR Data (Left)

The input X is a list of 2-dimensional vectors. Every example X_i is represented by a 2-dimensional vector [x,y]. The output y_i corresponding to the i^{th} example is either 0 or 1. The labels follow XOR-like distribution. That is, the first and third quadrant has same label ($y_i = 1$) and the second and fourth quadrant has same label ($y_i = 0$) as shown in Figure 1.

Semicircle Data (Right)

The input X is a list of 2-dimensional vectors. Every example X_i is represented by a 2-dimensional vector [x,y]. The output y_i corresponding to the i^{th} example is either 0 or 1. Each set of label ($y_i = 1$ and $y_i = 0$) is arranged approximately on the periphery of a semi circle of unknown radius R with some spread W as shown in Figure 2.

MNIST Data

MNIST data set is used which contains a collection of handwritten numerical digits (0-9) as 28x28-sized binary images. Therefore, input X is represented as a vector of size 784. These images have been size-normalised and centred in a fixed-size image. MNIST provides a total 70,000 examples, divided into a test set of 10,000 images and a training set of 60,000 images. It will carve out a validation set of 10,000 images from the MNIST training set, and use the remaining 50,000 examples for training.

CIFAR 10 Data

CIFAR-10 data set contains 60,000 32x32 color(RGB) images in 10 different classes. The 10 different classes represent **airplanes, cars, birds, cats, deer, dogs, frogs, horses, ships, and trucks**. There are 6,000 images of each class. These 60,000 images are divided into a test set of 10,000 images and a training set of 50,000 images. It will carve out a validation set of 10,000 images from the CIFAR-10 training set, and use the remaining 40,000 examples for training.

Here, we will code a feed-forward neural network and convolution neural network from scratch using python3 and the numpy library.

Code

The base code for this assignment is available on drive link shared with students. Below is the list of files:

File Name	Description
layers.py	This file contains base code for the various layers of neural network
nn.py	This file contains base code for the neural network implementation.
test_feedforward.py	This file contains code to test feedforward() implementation for various layers.
visualize.py	This file contains the code for visualisation of the neural network's predictions on toy data sets.
visualizeTruth.py	This file contains the code to visualise actual data distribution of toy data sets.
tasks.py	This files contains base code for the tasks that need to implement.
test.py	This file contains base code for testing neural network on different data sets.
util.py	This file contains some of the methods used by above code files.
autograder.py	This is used for testing Task 1 and Task 2

All data sets are provided in the datasets directory.

Understanding the code

We start with building the neural network framework. The base code for the same is provided in nn.py and layer.py.

nn.py. The base code consists of the NeuralNetwork class which has the following methods.

- __init__ : Constructor method of the NeuralNetwork class. The various parameters are initialised and set using this method. The description of various parameters can be seen in file nn.py.
- addLayer : Method to add layers to the neural network.

- train: This method is used to train the neural network on a training data set. Optionally, can provide the validation data set as well, to compute validation accuracy while training. Further description on various input parameters to the method are available in the base code.

- computeLoss: This method takes as input the neural network activations of the final layer predictions and the actual labels Y and computes the squared error loss for the same.

- computeAccuracy: This method takes as input the predicted labels in one hot encoded form, predLabels, and the actual labels Y and computes the accuracy of the prediction. The two toy data sets will require neural nets with two outputs, and MNIST and CIFAR neural nets would have 10 outputs. In each case the output with the highest activation will provide the label.

- validate: This method takes as input the validation data set, validX and validY, and computes the validation set accuracy using the currently trained neural network model.

- feedforward: This method takes as input the training data features, X, and does one forward pass across all the layers of neural network.

- backpropagate: This method takes as input the training labels and activations produced by forward pass, and does one backward pass across all the layers of neural network.

layers.py. The base code consists of various Layer classes, each of which have the following methods.

- __init__: This method takes as input the various parameters need to construct a particular layer of neural net (for example, a fully connected layer would required input and output nodes as parameters). Some layers contain self.weights and self.biases (numpy multi dimensional arrays) which are initialized using a normal distribution with mean 0 and standard deviation 0.1. The specific dimensions of these are given in layers.py.

- forwardpass: Needs to be implemented.

- backwardpass: Needs to be implemented.

Methods need to be implemented

We need to implement these functions for all the layers in the layer.py.

- forwardpass: This method takes as input activation from previous layer X, and returns new activations w.r.t. this layer. The activations must be returned as a list of numpy arrays. **NOTE:** Remember to follow same convention as specified in the comments of the layers.py file. Also, It might need to save the outgoing summation of weights * feautres + biases calculated during the forward pass to be used in backpropagation

- backwardpass: This method takes as input the activations of previous layer (activation_prev) and partial derivatives of the error w.r.t. current layer activations(delta), and updates the weights and biases of the current layer. It returns as output the partial derivatives of the error w.r.t previous layer activations.
  NOTE: It must implement backpropagation based on sigmoid activation everywhere. Recall that sigmoid(x) = $1 / (1 + e^{-x})$.

Layers: Explanation and References

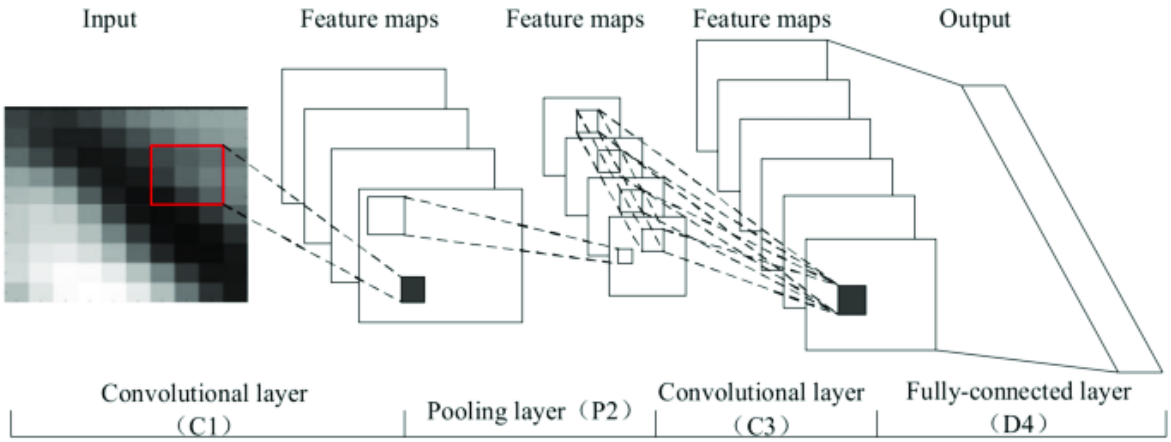
- Fully Connected Layer: This layer contains weights in form of in_nodes X out_nodes . Forward pass in such a layer is simply matrix multiplication of activations from previous layer and its weights and addition of bias to it. A_new = A_old X Curr_Wts + Bias. Similarly, backpropagation updates can be obtained by simple matrix multiplications.

- Convolution Layer: This layer contains weights in form of numpy array of form (out_depth, in_depth, filter_rows, filter_cols). Forward pass in such layer can be obtained from correlation of filter with the previous layer for all its depths and addition of biases to this sum. Backwardpass for such a system of weights results in another convolution that need to figure out.

- AvgPooling Layer: This layer is essentially used to reduce the weights used in convolution. It downsamples the previous layer by a factor stride and filter (**in our case both can be assumed to the same**). While downsampling one could have used many operations, including taking mean, median, max, min, etc,. Here we take the **Average** operation for downsampling. This doesn't effect the depth of the input. Forward pass in such layer can be obtained by taking Mean in a grid of elements of size = filter_size. Backwardpass for such a system just upsamples the numpy multidimensional array, scaling them by same factor that was used in downsampling.

- Flatten Layer: This layer is inserted to convert 3d structure of convolution layers to 1d nodes required for fully connected layers.

Following are some key numpy functions that will speed up the computation: tile, transpose, max, argmax, repeat, dot, matmul. To perform element-wise product between two vectors or matrices of the same dimension, simply use the * operator.



A CNN with 2 Convolution layers, 1 Pooling Layer and 1 Fullyconnected layer. Flatten Layer b/w C3 and D4 is implicit in the figure

---

Task 1: Forward Pass

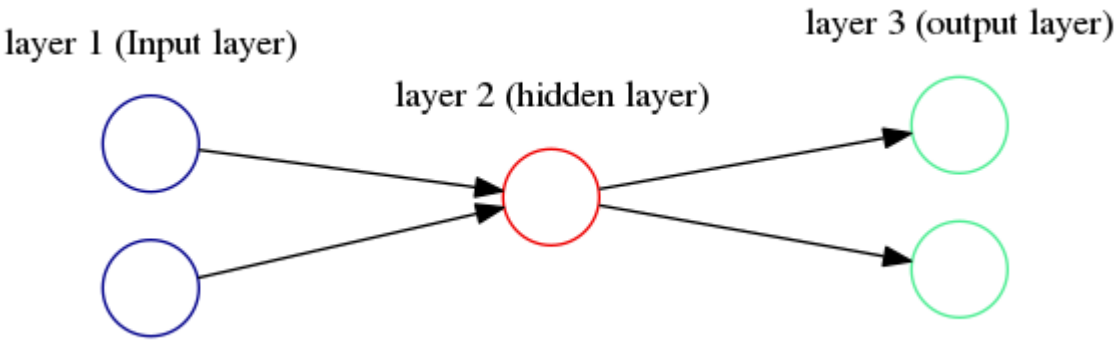
In this task, complete the feedforward methods in layers.py file.

Testing feed-forward network

To test feed forward network implementation, run the following code.

python3 test_feedforward.py

There are three test cases. The first corresponds to a small neural network (with 2 fully connected layers (2 X 1 and 1 X 2) shown below, while the second test is based on results of the MNIST data set, and the third test is based on results of the CIFAR-10 data set (see test_feedforward.py).



Task 2: Backpropagation implementation

In this task, complete the backwardpass() methods in layers.py file. Use the matrix form of the backpropagation algorithm for quicker and more efficient computation. Also, see Task 2.5 before tweaking models for remaining tasks, report neural networks with *minimal* topologies.

Evaluation over Data Sets

Backpropagation code will be tested on toy data sets (tasks 2.1 and 2.2)

Task 2.1: XOR Data set

Complete taskSquare() in tasks.py. To test backpropagation code on this data set, run the following command

python3 test.py 1 seedValue

To visualise ground truth, run the following command.
python3 visualizeTruth.py 1

Task 2.2: SemiCircle Data set

Complete taskSemiCircle() in tasks.py. To test backpropagation code on this data set, run the following command.

python3 test.py 2 seedValue

To visualise ground truth, run the following command.
python3 visualizeTruth.py 2

Task 2.3: Compute Accuracy over MNIST (using Feed-Forward Neural Network)

The MNIST data set is given as mnist.pkl.gz file. The data is read using the readMNIST() method implemented in util.py file.

Instantiate a NeuralNetwork class object (that uses FullyConnectedLayers) and train the neural network using the training data from MNIST data set. Choose appropriate hyper parameters for the training of the neural network.

Complete taskMnist() in tasks.py. To test backpropagation code on this data set, run the following command.

python3 test.py 3 seedValue

Here seedValue is a suitable integer value to initialise the seed for random-generator.

Task 2.4: Compute Accuracy over CIFAR (using Convolution Neural Network)

The CIFAR-10 data set is given in cifar-10 directory. The data is read using the readCIFAR10() method implemented in util.py file.

Instantiate a NeuralNetwork class object (with at least one convolutional layer) and train the neural network using the training data from CIFAR data. Choose appropriate hyperparameters for the training of the neural network. Choose the size of data to use for training, testing and validation. Save the weights is the file model.npy (look nn.py train() function for more details) for trained model and also **uncomment line 90 in tasks.py**. Also list all hyperparameters and the random seed used (seedValue) for training on this task (in a file named observations.txt), so as to reproduce results exactly.
NOTE: It will require lot of time to train (at least 4-5 hrs) in order to get the accuracies.

Complete taskCifar10() in tasks.py. To test backpropagation code on this data set, run the following command.

python3 test.py 4 seedValue

Task 2.5: Describe the results

In this task, It is required to report hyperparameters (learning rate, number of hidden layers, number of nodes in each hidden layer, batchsize and number of epochs) of Neural Network for each of the above 4 tasks that lead to minimal-topology neural networks that pass the test: that is, with the minimal number of nodes needed for each task. For example, in case of a linearly separable data set, should be able to get very high accuracy with just a single hidden node. Write observations in observations.txt. Also, explain the observations.

Conclusion: In this experiment, we have designed layers required to build both a feed-forward neural network (also popularly known as a Multilayer Perceptron classifier) and the Convolution Neural Networks.