SIX WEEKS SUMMER TRANING

REPORT

On

AI FOR SOFTWARE ENGINEERS

Submitted by

Mohammed Sardar Saajit

Registration. No: 12212969

Programme Name: B. Tech (CSE)

Under the Guidance of

Anoop Garg

School of Computer Science and Engineering

Lovely Professional University, Phagwara

(June-July 2024)

DECLARATION

I hear by declaring that I have completed my Six-week summer training at programming pathshala from 25/05/2024 to 20/07/2024 under the guidance of Anoop Garg. I have by declare that I have worked with full dedication during these Eight weeks of training and my learning outcomes fulfill the requirement of the training for the award of the degree of Bachelor of Technology in Computer Science and Technology Lovely Professional University, Phagwara.

Mohammed Sardar Saajit

12212969

Date: 31/08/2024

ACKNOWLEDEMENT

I would like to express my sincere gratitude to programming pathshala Mr. Anoop Garg (Co-Founder) for providing their invaluable guidance comments and suggestions throughout the course

I would specifically thank Mr. Anoop Garg my training mentor for consistently motivating me to work harder and for encouragement and teaching me course with invaluable assistance and the guidance provided in the summer training.

Also, I would like to thank my parents for giving encouragement and support without their consistent support I may not be able to complete this training properly I would also like to say thanks to my friends for being there for me when I need their help.

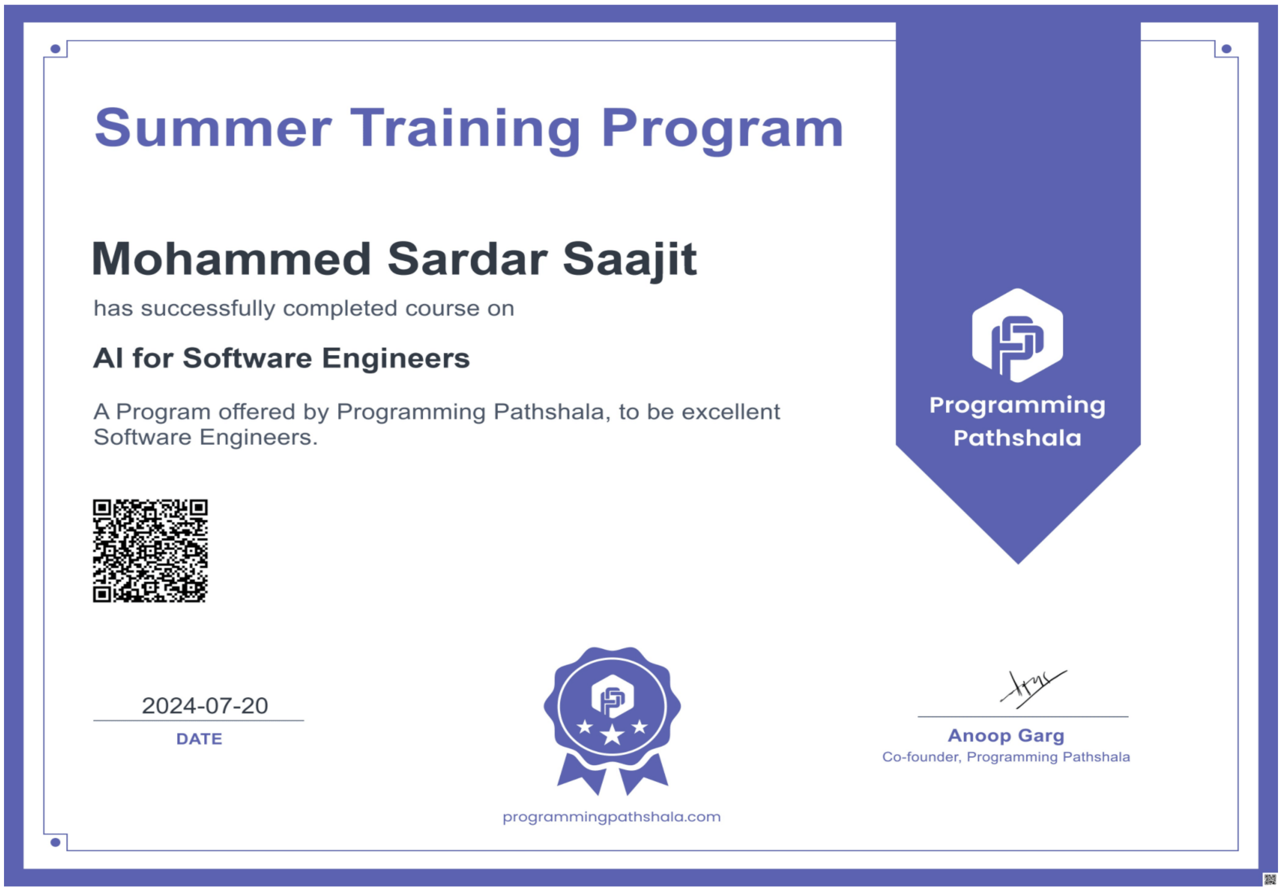
Mohammed Sardar Saajit

12212969

Date: 31/08/2024

TRAINING CERTIFICATE:

Programming Pathshala
669ba48775386515a8255970



Summer Training Program

Mohammed Sardar Saajit

has successfully completed course on

AI for Software Engineers

A Program offered by Programming Pathshala, to be excellent Software Engineers.

Programming Pathshala

2024-07-20
DATE

programmingpathshala.com

[illegible]
Anoop Garg
Co-founder, Programming Pathshala

Kindly verify Programming Pathshala's document on Blockchain by scanning the QR code.
https://certificates.programmingpathshala.com/docs/c703f65c77f6b94d9710d66d2b2b6fa22214a51c390718276967e06e3fbb6b6a

Introduction:

This project focuses on implementing a Convolutional Neural Network (CNN) model for digit classification using the MNIST dataset. The MNIST ("Modified National Institute of Standards and Technology") dataset is a large collection of handwritten digits that has become a standard benchmark in the field of machine learning and computer vision since its release in 1999.

Key aspects of the MNIST dataset:

- Contains 70,000 images in total (60,000 for training, 10,000 for testing)

Each image is a 28x28 grayscale representation of a digit,

- The digits range from 0 to 9.

Each pixel has a single value between 0 and 255, indicating its darkness
The project aims to:

Understand the basic concepts of CNN models:

- Implement a CNN model using real-time MNIST data
- Achieve high accuracy in classifying handwritten digits
- Compare the performance of CNNs with traditional machine learning models

The CNN model developed in this project consists of several key components:

- Convolutional layers for feature extraction
- Pooling layers for dimensionality reduction
- Fully connected layers for classification
- Dropout layers for preventing overfitting
- Technologies Learned

Throughout this project, a wide range of technologies and concepts were explored:

1. Python: The primary programming language used for implementation.
2. NumPy: Used for numerical computations and array manipulations.

Essential for handling the image data as multi-dimensional arrays.

3. Pandas: Used for data manipulation and analysis Employed to read and process the CSV files containing the MNIST data.
4. Matplotlib and Seaborn: Used for data visualization and creating informative plots.
5. Created various visualizations including Bar plots of label distributions
   Sample images from the dataset Training and validation accuracy/loss curves
   Confusion matrices.
6. Scikit-learn: Used for various machine learning utilities.
7. Specifically employed for: train_test_split function for dataset division
   confusion_matrix and classification_report for model evaluation
   Also used to implement and compare traditional ML models: Logistic Regression Support Vector Machines (SVM), Random Forest, K-Nearest Neighbors (KNN).
8. TensorFlow and Keras:
   Used for building and training the CNN model Keras Sequential API used to construct the model layer by layer TensorFlow backend used for efficient computations.
9. Convolutional Neural Networks: Learned the architecture and components
10. CNNs: Convolutional layers, Pooling layers, flatten layer, Dense (fully connected) layers, Dropout layers.
11. Image processing techniques: Normalizing pixel values (dividing by 255 to scale to 0-1 range).
12. Reshaping 2D image data to 3D tensors for CNN input One-hot encoding: Used to convert categorical labels to binary class matrices.
13. Model evaluation techniques: Accuracy, precision, recall, and F1-score
    Confusion matrix visualization.

2. Reason for Choosing AI/ML Project on This Topic:

The choice of an AI/ML project focusing on digit recognition/classifier using CNNs was motivated by several factors:

- Foundational importance: MNIST is often considered the "Hello World" of computer vision Provides a solid starting point for understanding image classification Allows focus on model architecture without the complexities of data preprocessing.
-
- Practical applications: Digit recognition has numerous real-world applications:
  Postal code reading for mail sorting, Check processing in banking, Automatic license plate recognition, Digit recognition in forms and documents.

- Introduction to CNNs: Provides a hands-on opportunity to understand and implement CNNs. CNNs are fundamental to modern computer vision tasks
  Teaches important concepts like convolution, pooling, and feature extraction.

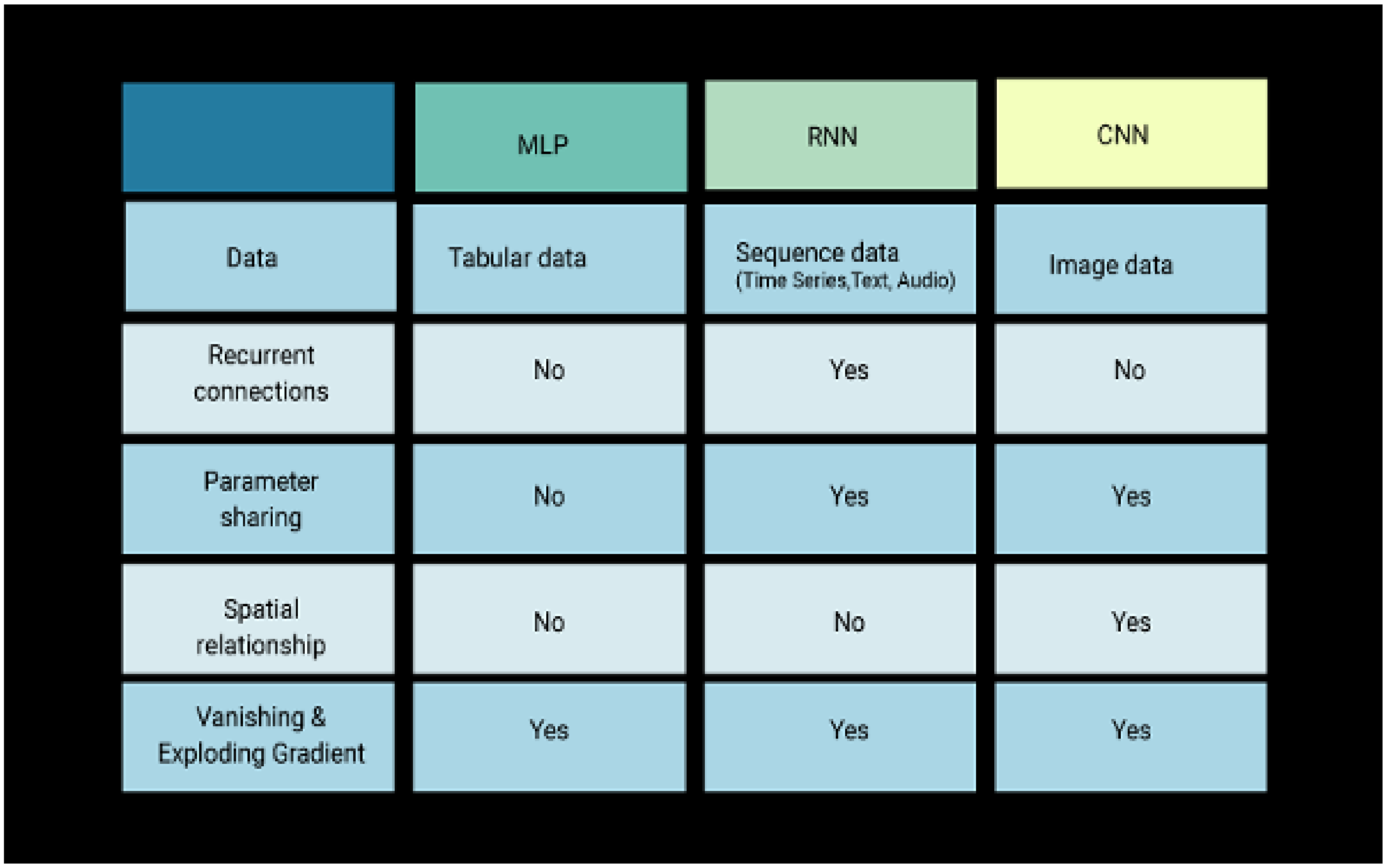
- Benchmark for ML techniques: MNIST remains a reliable resource for validating algorithms. Allows comparison of different models and architectures Provides a standardized dataset for measuring improvements.

- Scalability: Concepts learned can be applied to more complex image classification tasks Serves as a steppingstone to advanced computer vision projects

- Balanced difficulty: Challenging enough to be interesting Simple enough for beginners to grasp and implement

- Rich visualization opportunities: Allows for creation of intuitive visualizations of the data and results Helps in understanding the inner workings of the model

- Comparison with traditional ML models: Demonstrates the superiority of CNNs for image tasks, Provides context for the evolution of machine learning techniques.



	MLP	RNN	CNN
Data	Tabular data	Sequence data (Time Series,Text, Audio)	Image data
Recurrent connections	No	Yes	No
Parameter sharing	No	Yes	Yes
Spatial relationship	No	No	Yes
Vanishing & Exploding Gradient	Yes	Yes	Yes

3. ABOUT:

CNN Model Intro with MNIST Implementation

Brief Introduction

This notebook will cover the following two major topics:

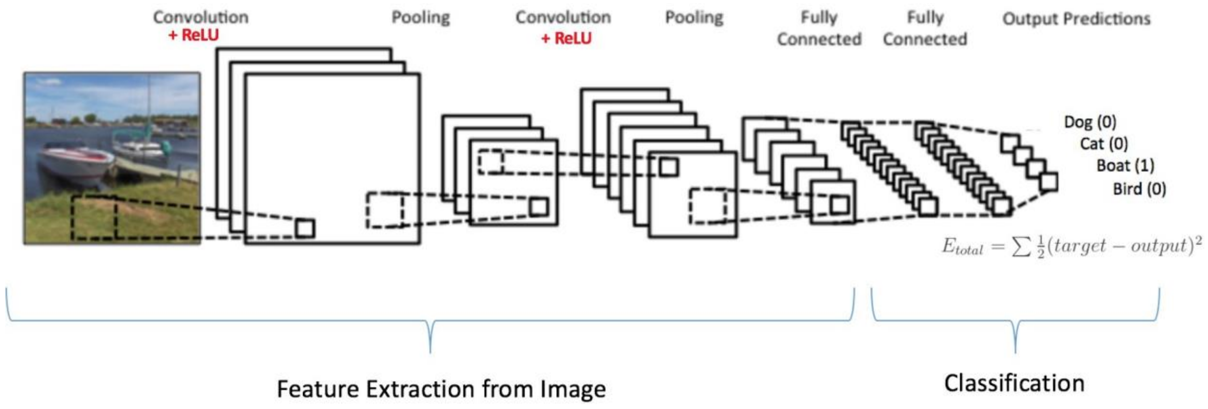
Understand the basic concepts of CNN model Implement CNN model in real-time using MNIST dataset

Understand the basic concepts of CNN model:

Mankind is an awesome natural machine and is capable of looking at multiple images every second and processing them without realizing how the processing is done. But same is not with machines.

The first step in image processing is to understand, how to represent an image so that themachine can read it?

Every image is an cumulative arrangement of dots (a pixel) arranged in a special order. If you change the order or color of a pixel, the image will change as well.



Three basic components to define a basic convolutional neural network.

1.The Convolutional Layer

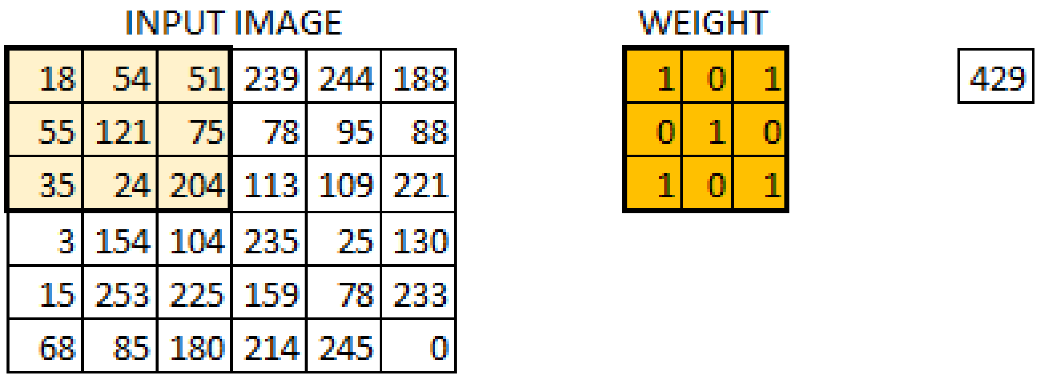
2. The Pooling layer

3.The Output layer

Let's see each of them in detail

3.1 The Convolutional Layer:

In this layer if we have an image of size 66. We define a weight matrix which extracts certain features from the images



INPUT IMAGE

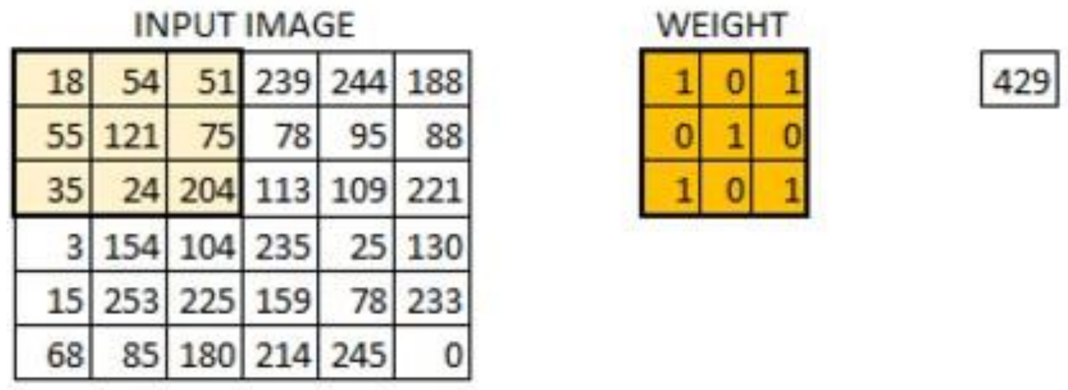
18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

1	0	1
0	1	0
1	0	1

429

We have initialized the weight as a 33 matrix. This weight shall now run across the image such that all the pixels are covered at least once, to give a convolved output. The value 429 above, is obtained by the adding the values obtained by element wise multiplication of the weight matrix and the highlighted 33 part of the input image.



INPUT IMAGE

18	54	51	239	244	188
55	121	75	78	95	88
35	24	204	113	109	221
3	154	104	235	25	130
15	253	225	159	78	233
68	85	180	214	245	0

WEIGHT

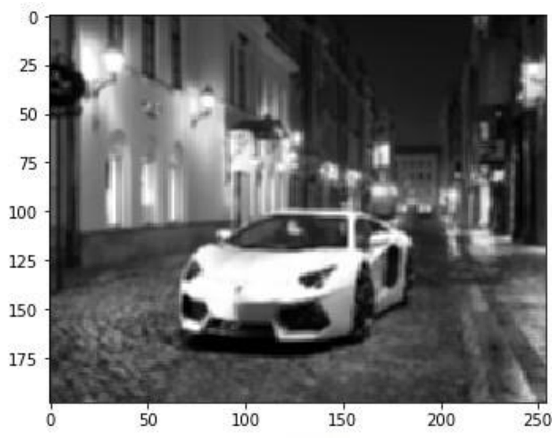
1	0	1
0	1	0
1	0	1

429

The 66 image is now converted into a 44 image. Think of weight matrix like a paint brush painting a wall. The brush first paints the wall horizontally and then comes down and paints thenext row horizontally. Pixel values are used again when the weight matrix moves along the image. This basically enables parameter sharing in a convolutional neural network.

Let's see how this looks like in a real image.

Original Image



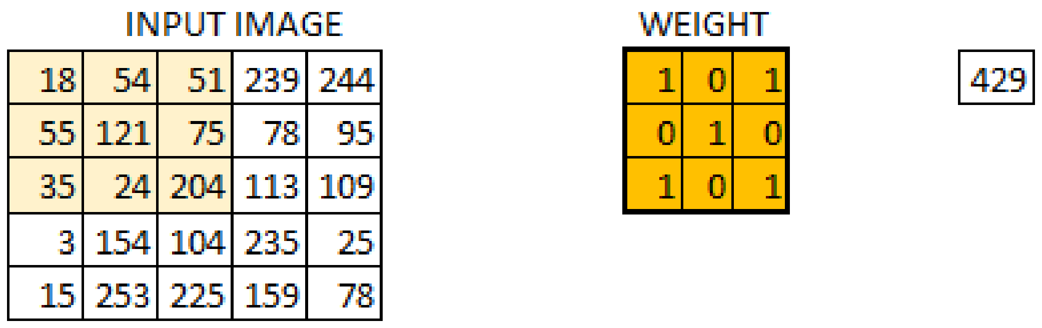
Convoluted image

- The weight matrix behaves like a filter in an image, extracting information from the original image matrix.
- A weight combination might be extracting edges, while another one might a particular color, while another one might just blur the unwanted noise.
- The weights are learnt such that the loss function is minimized and extract features from the original image which help the network in correct prediction.
- When we use multiple convolutional layers, the initial layer extract more generic features,and as network gets deeper the features get complex.

Let us understand some concepts here before we go further deep

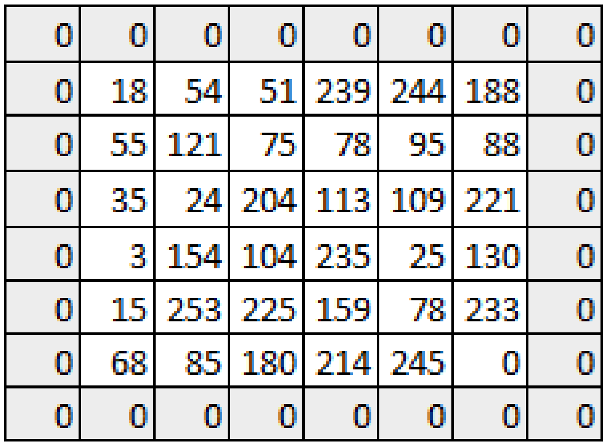
3.2 What is Stride?

As shown above above, the filter or the weight matrix we moved across the entire image moving one pixel at a time.If this is a hyperparameter to move weight matrix 1 pixel at a time across image it is called as stride of 1. Let us see for stride of 2 how it looks.



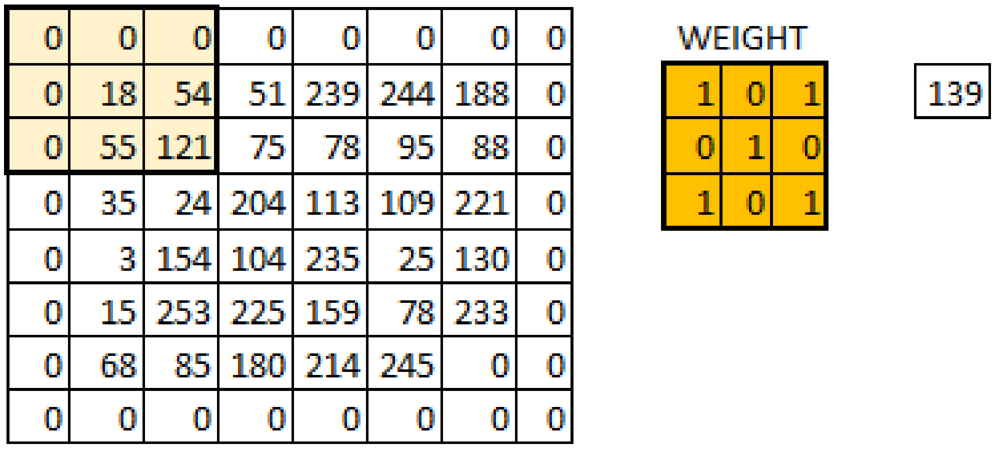
As you can see the size of the image keeps on reducing as we increase the stride value.

Padding the input image with zeros across it solves this problem for us. We can also add morethan one layer of zeros around the image in case of higher stride values.



0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

We can see how the initial shape of the image is retained after we padded the image with a zero.This is known as the same padding since the output image has the same size as the input.



0	0	0	0	0	0	0	0
0	18	54	51	239	244	188	0
0	55	121	75	78	95	88	0
0	35	24	204	113	109	221	0
0	3	154	104	235	25	130	0
0	15	253	225	159	78	233	0
0	68	85	180	214	245	0	0
0	0	0	0	0	0	0	0

WEIGHT

1	0	1
0	1	0
1	0	1

139

This is known as same padding (which means that we considered only the valid pixels of the input image). The middle 4*4 pixels would be the same. Here we have retained more information from the borders and have also preserved the size of the image.

3.4 Having Multiple filters & the Activation Map

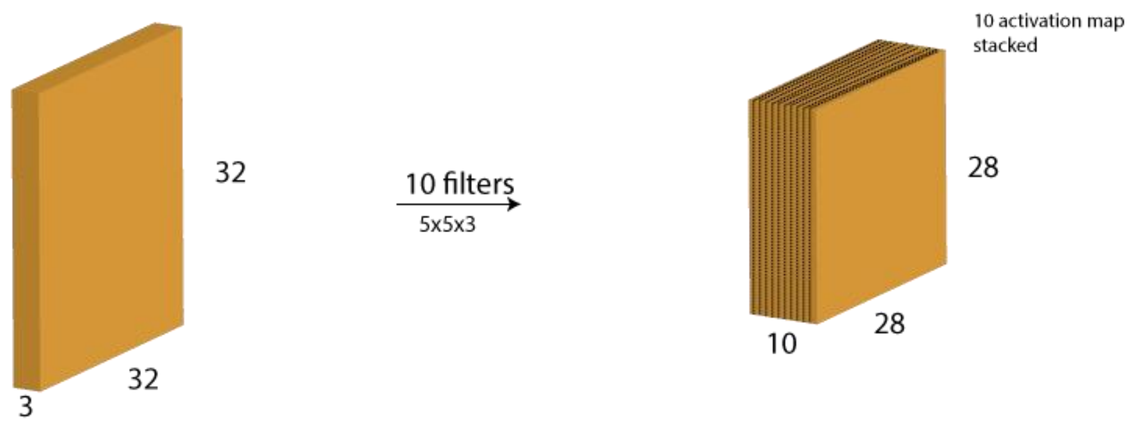
- The depth dimension of the weight would be same as the depth dimension of the input image.
- The weight extends to the entire depth of the input image.
- Convolution with a single weight matrix would result in a convolved output with a single depth dimension. In the case of multiple filters all have same dimensions

appliedtogether.

- The output from each filter is stacked together forming the depth dimension of the convolved image.

Suppose we have an input image of size 32323. And we apply 10 filters of size 553 with valid padding. The output would have the dimensions as 282810.

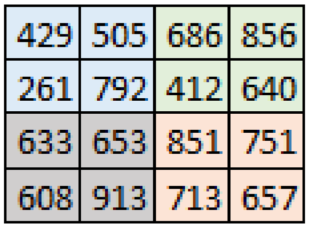
You can visualize it as:



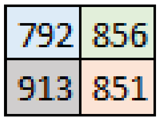
This activation map is the output of the convolution layer.

3.5 The Pooling Layer

If images are big in size, we would need to reduce the no. of trainable parameters. For this we need to use pooling layers between convolution layers. Pooling is used for reducing the spatial size of the image and is implemented independently on each depth dimension resulting in no change in image depth. Max pooling is the most popular form of pooling layer.



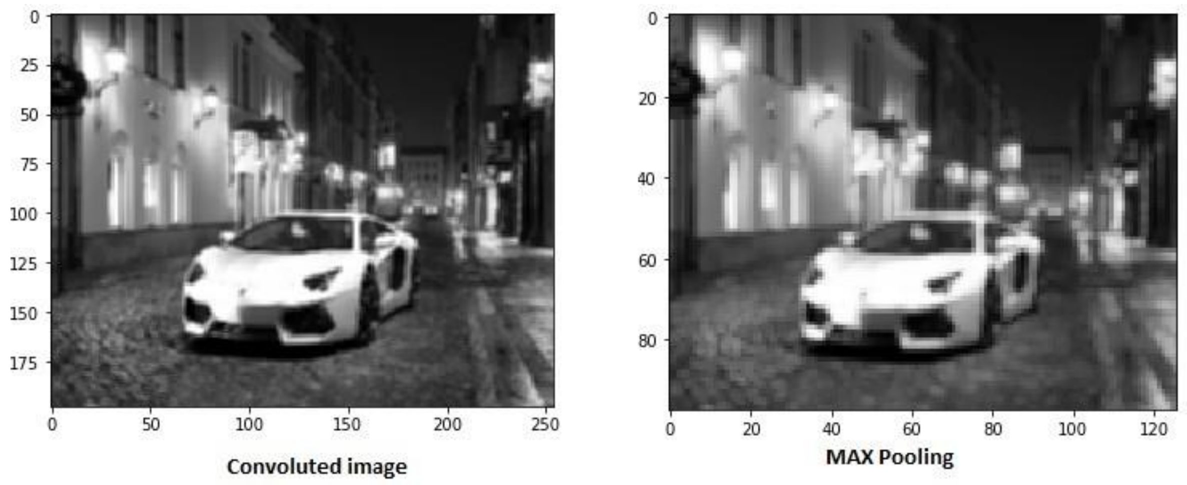
429	505	686	856
261	792	412	640
633	653	851	751
608	913	713	657



792	856
913	851

Here we have taken stride as 2, while pooling size also as 2. The max operation is applied to each depth dimension of the convolved output. As you can see, the 44 convolved output has become 22 after the max pooling operation.

Let's see how max pooling looks on a real image.



In the above image we have taken a convoluted image and applied max pooling on it which resulted in still retaining the image information that is a car but if we closely observe the dimensions of the image is reduced to half which basically means we can reduce the parametersto a great number.

There are other forms of pooling like average pooling, L2 norm pooling.

3.6 Output dimensions

It is tricky at times to understand the input and output dimensions at the end of each convolution layer. For this we will use three hyperparameters that would control the size of output volume.

1. No of Filter: The depth of the output volume will be equal to the number of filter applied.The depth of the activation map will be equal to the number of filters.
2. Stride - When we have a stride of one we move across and down a single pixel. With higher stride values, we move large number of pixels at a time and hence produce smaller output volumes.
3. Zero padding - This helps us to preserve the size of the input image. If a single zero padding is added, a single stride filter movement would retain the size of the original image.

We can apply a simple formula to calculate the output dimensions.

The spatial size of the output image can be calculated as([W-F+2P]/S)+1. where, W is the input volume size, F is the size of the filter, P is the number of padding applied S is the number of strides.

Let us take an example of an input image of size 64643, we apply 10 filters of size 333, with single stride and no zero padding.

Here W=64, F=3, P=0 and S=1. The output depth will be equal to the number of filters applied i.e. 10.

The size of the output volume will be ([64-3+0]/1)+1 = 62. Therefore the output volume will be 626210.

3.7 The Output layer

- With no of layers of convolution and padding, we need the output in the form of a class.

- To generate the final output we need to apply a fully connected layer to generate an output equal to the number of classes we need.
- Convolution layers generate 3D activation maps while we just need the output as whether or not an image belongs to a particular class.
- The Output layer has a loss function like categorical cross-entropy, to compute the error in prediction. Once the forward pass is complete the backpropagation begins to update the weight and biases for error and loss reduction.

4. Summary:

- Pass an input image to the first convolutional layer. The convoluted output is obtained as an activation map. The filters applied in the convolution layer extract relevant features from the input image to pass further.

- Each filter shall give a different feature to aid the correct class prediction. In case we need to retain the size of the image, we use same padding(zero padding), otherwise valid padding is used since it helps to reduce the number of features.

Pooling layers are then added to further reduce the number of parameters

- Several convolution and pooling layers are added before the prediction is made. Convolutional layer help in extracting features. As we go deeper in the network more specific features are extracted as compared to a shallow network where the features extracted are more generic.

- The output layer in a CNN as mentioned previously is a fully connected layer, where the input from the other layers is flattened and sent so as the transform the output into the number of classes as desired by the network.

- The output is then generated through the output layer and is compared to the output layer for error generation. A loss function is defined in the fully connected output layer to compute the mean square loss. The gradient of error is then calculated.
- The error is then backpropagated to update the filter(weights) and bias values.
- One training cycle is completed in a single forward and backward pass.

3.8 Implement CNN model in Realtime using Hand digit MNIST dataset:

Hand digit-MNIST is a dataset of Digits images—consisting of a training set of 20,000 examplesand a test set of 1,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. Members of the AI/ML/Data Science community love this dataset and use it as a benchmark to validate their algorithms. In fact, MNIST is often the first dataset researchers try. "If it doesn't work on MNIST, it won't work at all", they said. "Well, if it does work on MNIST, it may still fail on others."

4.1 Data Description

- Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total.
- Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255.
- The training and test data sets have 785 columns.
- The first column consists of the class labels (see above), and represents the digit
- The rest of the columns contain the pixel-values of the associated image.

To locate a pixel on the image, suppose that we have decomposed x as x = i * 28 + j, where i and j are integers between 0 and 27. The pixel is located on row i and column j of a 28 x 28 matrix. For example, pixel31 indicates the pixel that is in the fourth column from the left, and the second row from the top, as in the ascii-diagram below.

4.2 About the Dataset

- MNIST ("Modified National Institute of Standards and Technology") is the de facto “helloworld” dataset of computer vision.

- Since its release in 1999, this classic dataset of handwritten images has served as the basis for benchmarking classification algorithms.
- As new machine learning techniques emerge, MNIST remains a reliable resource for researchers and learners alike.

5. Task (DIGIT CLASIFIER)

- The goal is to accurately identify and classify digits from a large-scale dataset containing tens of thousands of handwritten images. This involves applying advanced machine learning techniques to analyze and interpret the features of each image, ensuring that each digit is correctly recognized and categorized within the test dataset. The process requires the model to be trained on a wide variety of handwriting styles to generalize well and achieve high accuracy in digit identification across the entire dataset.

Libraries

```
# for numerical analysis
import numpy as np
# to store and process in a dataframe
import pandas as pd

# for ploting graphs
import matplotlib.pyplot as plt
# advancec ploting
import seaborn as sns

# image processing
import matplotlib.image as mpimg

# train test split
from sklearn.model_selection import train_test_split
# model performance metrics
from sklearn.metrics import confusion_matrix, classification_report

# utility functions
from tensorflow.keras.utils import to_categorical
```

```
# sequential model
from tensorflow.keras.models import Sequential
# layers
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten,
Dropout
```

5.1 Data

```
# import train and test dataset
train  =  pd.read_csv("/content/sample_data/mnist_train_small.csv")
test = pd.read_csv("/content/sample_data/mnist_test.csv")

# training dataset
train.head()

{"type":"dataframe","variable_name":"train"}

len(test.index)

9999

# test dataset
test.head()

{"type":"dataframe","variable_name":"test"}

train.rename(columns={'6': 'Label'}, inplace=True)
test.rename(columns={'7': 'Label'}, inplace=True)

test.head()

{"type":"dataframe","variable_name":"test"}

# looking for missing values
print(train.isna().sum().sum())
print(test.isna().sum().sum())

0
0
```

5.2 EDA

Label count

```
train['Label'].value_counts()

Label
1      2243
```

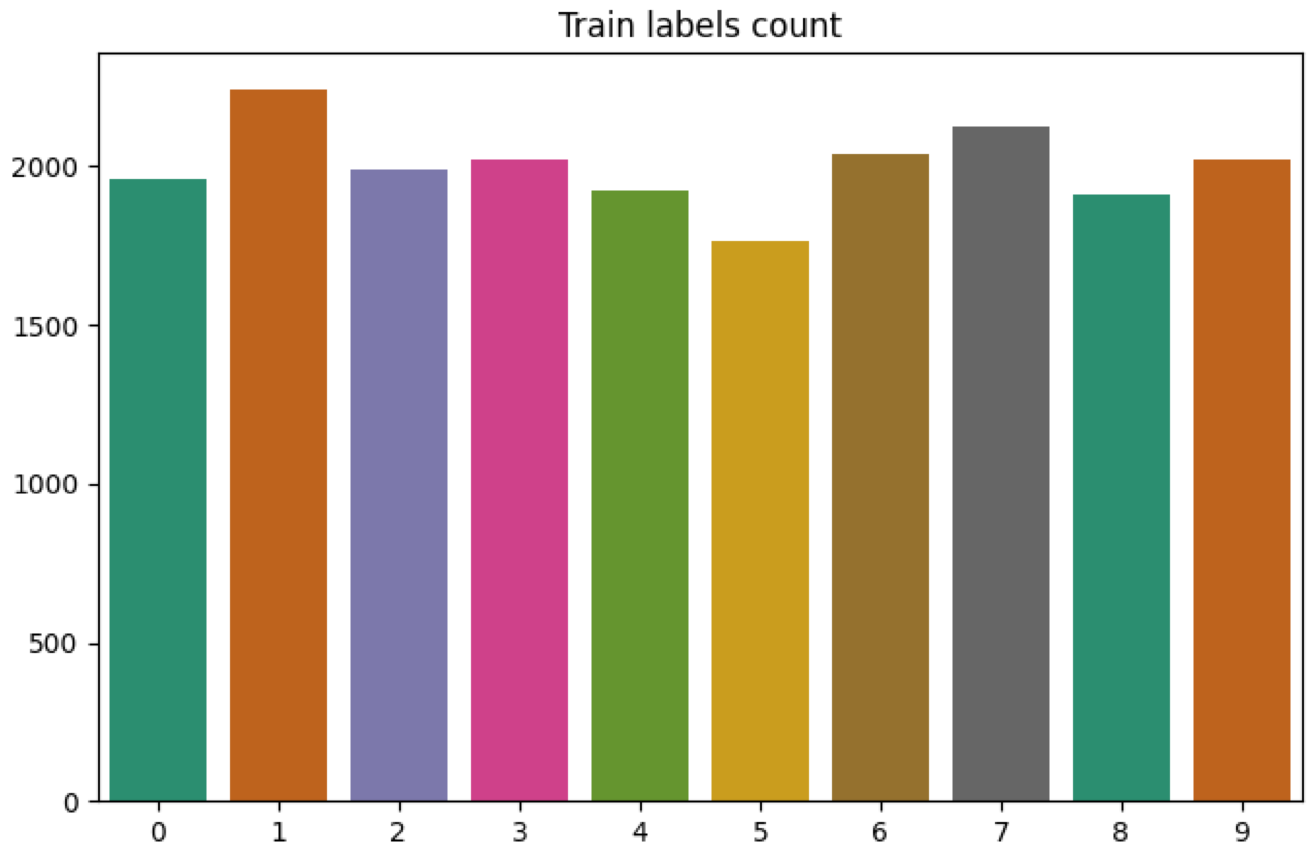
```
7    2126
6    2038
9    2023
3    2021
2    1989
0    1962
4    1924
8    1912
5    1761
Name: count, dtype: int64
```

```
count_num = [x for x in train['Label'].value_counts().sort_index()]
count_num
```

```
[1962, 2243, 1989, 2021, 1924, 1761, 2038, 2126, 1912, 2023]
```

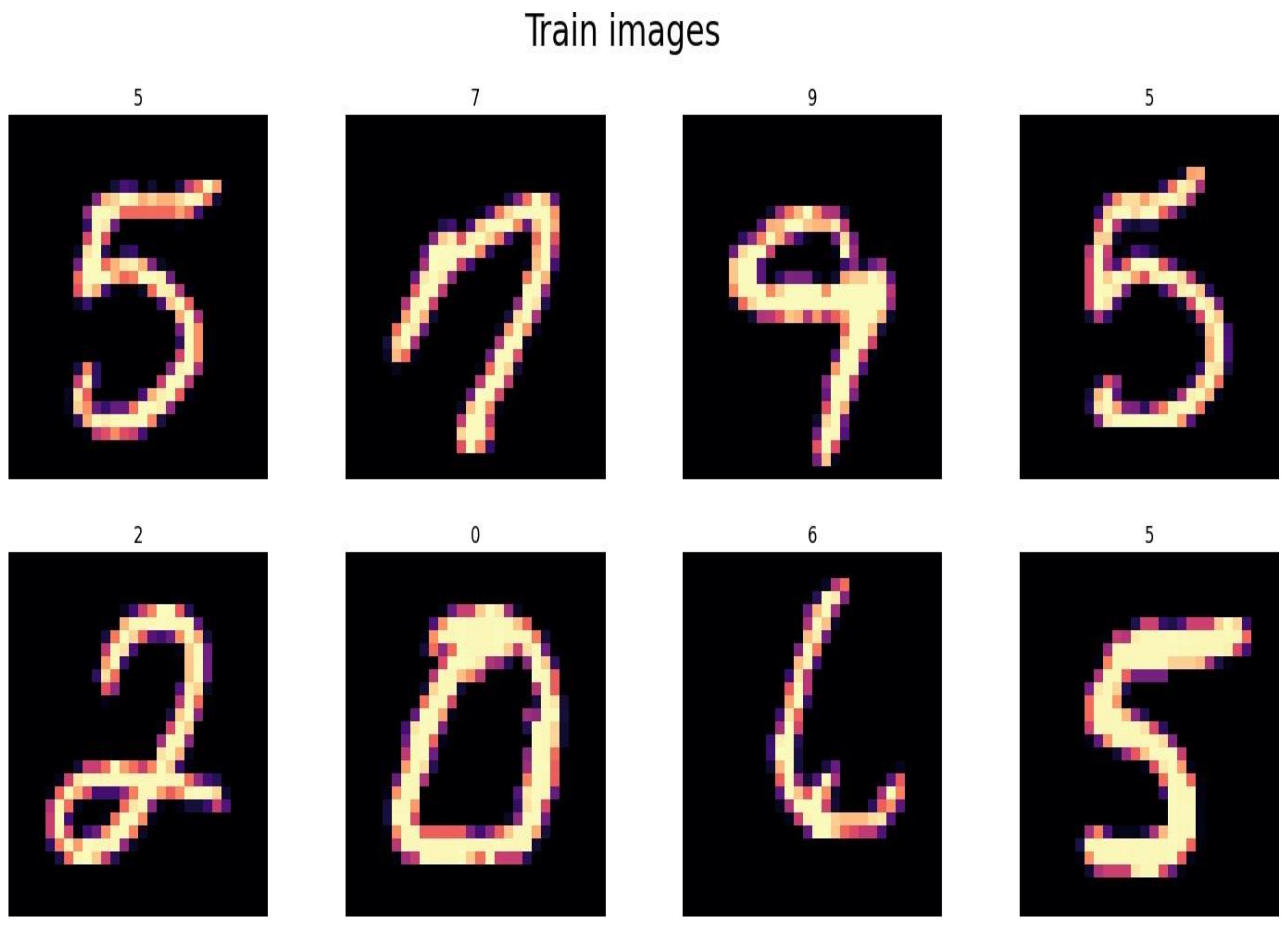
```
plt.figure(figsize=(8, 5))
sns.barplot(count_num,
palette='Dark2') plt.title('Train
labels count') plt.show()
```

```
<ipython-input-11-285a7cc3ef98>:2: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be
removed in v0.14.0. Assign the `x` variable to `hue` and set
`legend=False` for the same effect.

  sns.barplot(count_num,  palette='Dark2')
```



```
# first few train images with labels
fig, ax = plt.subplots(figsize=(18, 8))
for ind, row in train.iloc[:8, :].iterrows():
    plt.subplot(2, 4, ind+1) plt.title(row[0])
    img = row.to_numpy()[1:].reshape(28, 28)
    fig.suptitle('Train images', fontsize=24)
    plt.axis('off')
    plt.imshow(img, cmap='magma')
```

```
<ipython-input-12-1a8796f218a7>:4: MatplotlibDeprecationWarning: Auto-
removal of overlapping axes is deprecated since 3.6 and will be removed two
minor releases later; explicitly call ax.remove() as needed.
  plt.subplot(2, 4, ind+1)
<ipython-input-12-1a8796f218a7>:5: FutureWarning: Series.__getitem
treating keys as positions is deprecated. In a future version, integer keys
will always be treated as labels (consistent with DataFrame behavior). To
access a value by position, use `ser.iloc[pos]`
  plt.title(row[0])
```



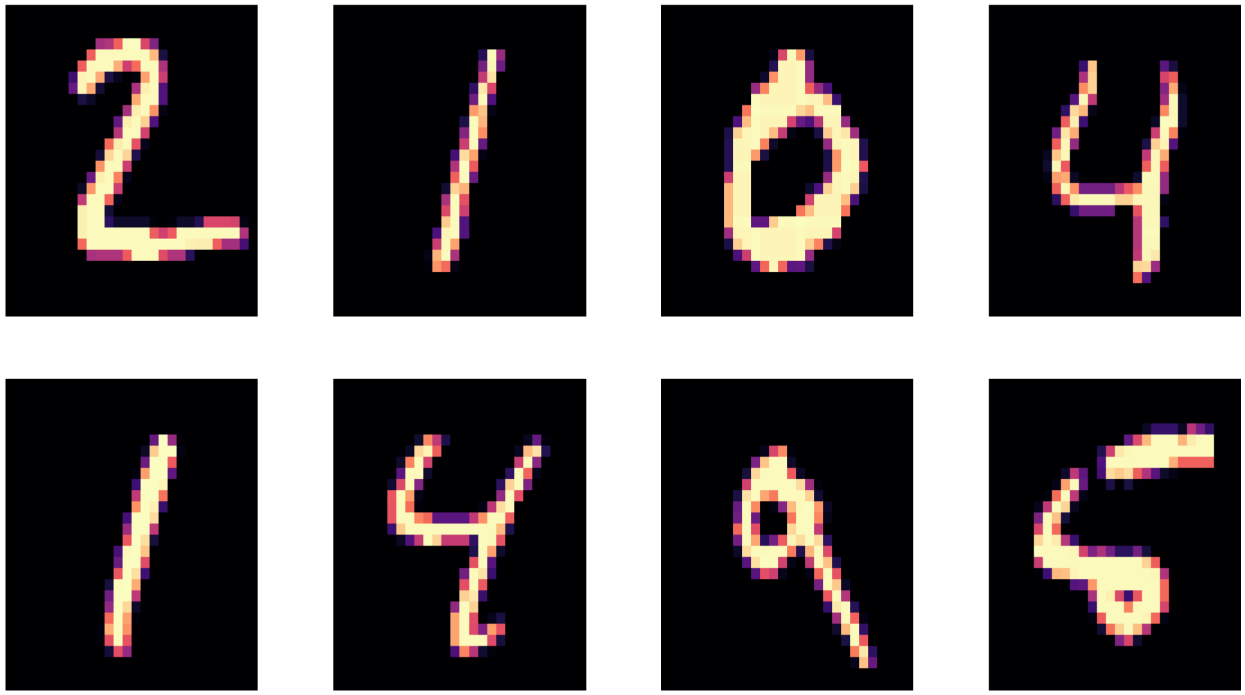
Test images doesn't have labels

```
# first few test images
fig, ax = plt.subplots(figsize=(18, 8))
for ind, row in test.iloc[:8, :].iterrows():
    plt.subplot(2, 4, ind+1)
    img = row.to_numpy()[1:].reshape(28, 28)
    fig.suptitle('Test images', fontsize=24)
    plt.axis('off')
    plt.imshow(img, cmap='magma')
```

```
<ipython-input-13-3566033cd87d>:4: MatplotlibDeprecationWarning: Auto-
removal of overlapping axes is deprecated since 3.6 and will be
removed two minor releases later; explicitly call ax.remove() as
needed.
  plt.subplot(2, 4, ind+1)
```

We need to create a model to predict them

Test images



5.3 Preprocessing

```
# split into image and labels and convert to numpy array
X =  train.iloc[:, 1:].to_numpy()
y  = train['Label'].to_numpy()

# test dataset
X_test = test.iloc[:, 1:].to_numpy()
y_test  = test['Label'].to_numpy()

for i in [X, y, X_test, y_test]:
    print(i.shape)
```

```
(19999, 784)
(19999,)
(9999, 784)
(9999,)
```

```
# normalize the data
# ==================

X  = X  /  255.0
X_test  =  X_test  /  255.0
```

5.4 Checking the predictive ability of various ML Models

```
from sklearn.linear_model import LogisticRegression
from  sklearn.metrics  import  accuracy_score

# Initialize the Logistic Regression model
```

```
logreg = LogisticRegression(max_iter=1000)

# Train the model
logreg.fit(X, y)

# Predict on test data
y_pred_logreg  =  logreg.predict(X_test)

# Evaluate the model
accuracy_logreg      =      accuracy_score(y_test,      y_pred_logreg)
print(f"Logistic Regression Accuracy: {accuracy_logreg}")
```

```
Logistic   Regression   Accuracy:   0.918091809180918
```

```
from
sklearn.svm import SVC

# Initialize the SVM model
svm = SVC()

# Train the model
svm.fit(X, y)

# Predict on test data
y_pred_svm  =  svm.predict(X_test)
```

```
# Evaluate the model
accuracy_svm = accuracy_score(y_test, y_pred_svm)
print(f"SVM Accuracy: {accuracy_svm}")
```

```
SVM Accuracy: 0.96999699969997
```

```
from sklearn.ensemble import RandomForestClassifier

# Initialize the Random Forest model
rf = RandomForestClassifier(n_estimators=100)

# Train the model
rf.fit(X, y)

# Predict on test data
y_pred_rf = rf.predict(X_test)

# Evaluate the model
accuracy_rf = accuracy_score(y_test, y_pred_rf)
print(f"Random Forest Accuracy: {accuracy_rf}")
```

```
Random Forest Accuracy: 0.9593959395939594
```

```
from sklearn.neighbors import KNeighborsClassifier

# Initialize the k-NN model
knn = KNeighborsClassifier(n_neighbors=3)
```

```
# Train the model
knn.fit(X, y)

# Predict on test data
y_pred_knn = knn.predict(X_test)

# Evaluate the model
accuracy_knn = accuracy_score(y_test, y_pred_knn)
print(f"k-NN Accuracy: {accuracy_knn}")
```

```
k-NN Accuracy: 0.9594959495949595
```

SVM model could reach a maximum accuracy of 97%

We will try to improve this accuracy by using CNN and by building a neural network architechture and test our data with this model.

```
# reshape dataset
# ===============

# shape of training and test dataset
print(X.shape)
print(X_test.shape)

# reshape the dataframe to 3x3 matrix with 1 channel grey scale values
X  =  X.reshape(-1,28,28,1)
X_test  =  X_test.reshape(-1,28,28,1)

# shape of training and test dataset
print(X.shape)
print(X_test.shape)
```

```
(19999, 784)
(9999, 784)
(19999, 28, 28, 1)
(9999, 28, 28, 1)
```

```
# utility functions
from tensorflow.keras.utils import to_categorical
# sequential model
from tensorflow.keras.models import Sequential
# layers
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten,
Dropout

# one hot encode target
# ====================

# shape and values of target
print(y.shape)
print(y[0])

# convert Y_train to categorical by one-hot-encoding
y_enc = to_categorical(y, num_classes = 10)

# shape and values of target
print(y_enc.shape)
print(y_enc[0])
```

```
(19999,)
5
(19999, 10)
[0. 0. 0. 0. 0. 1. 0. 0. 0. 0.]
```

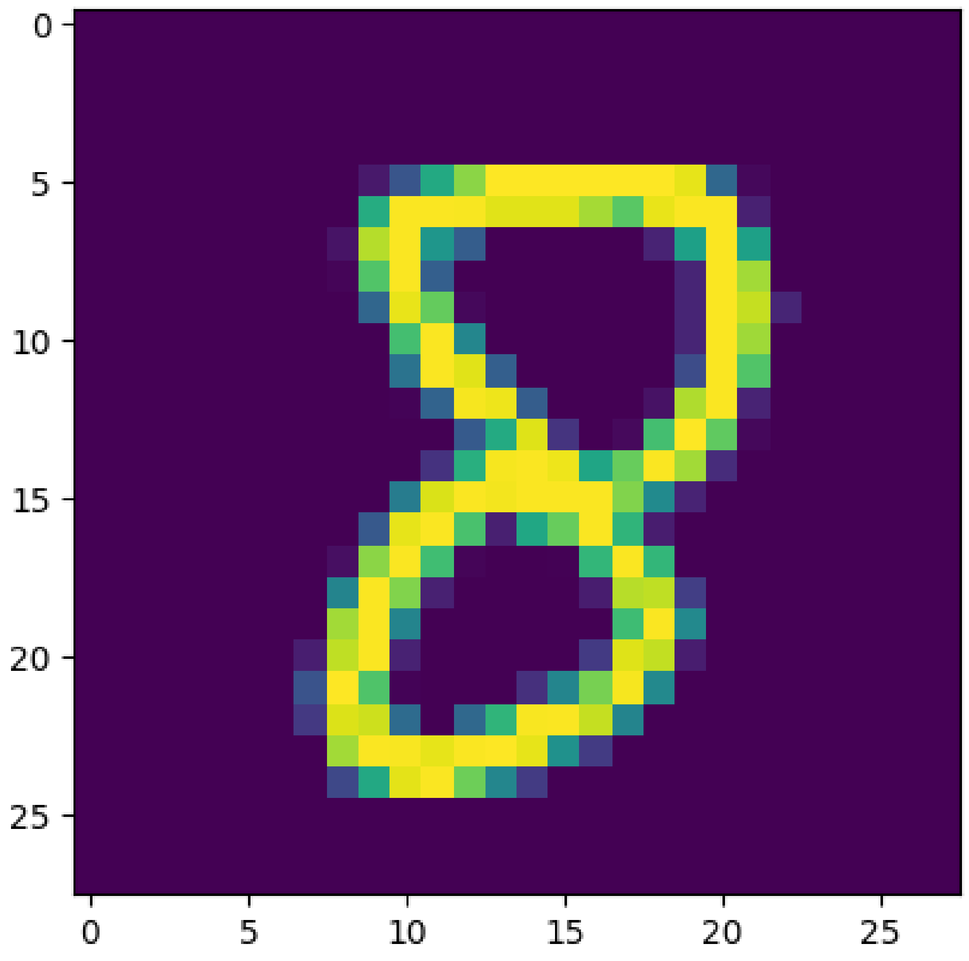
```
# train test split
# ================

# random seed
random_seed = 101

# train validation split
X_train, X_val, y_train_enc, y_val_enc = train_test_split(X, y_enc,
test_size=0.3)

# shape
for i in [X_train, y_train_enc, X_val, y_val_enc]:
    print(i.shape)
```

```
(13999, 28, 28, 1)
(13999, 10)
(6000, 28, 28, 1)
(6000, 10)
```

5.5 Plot images

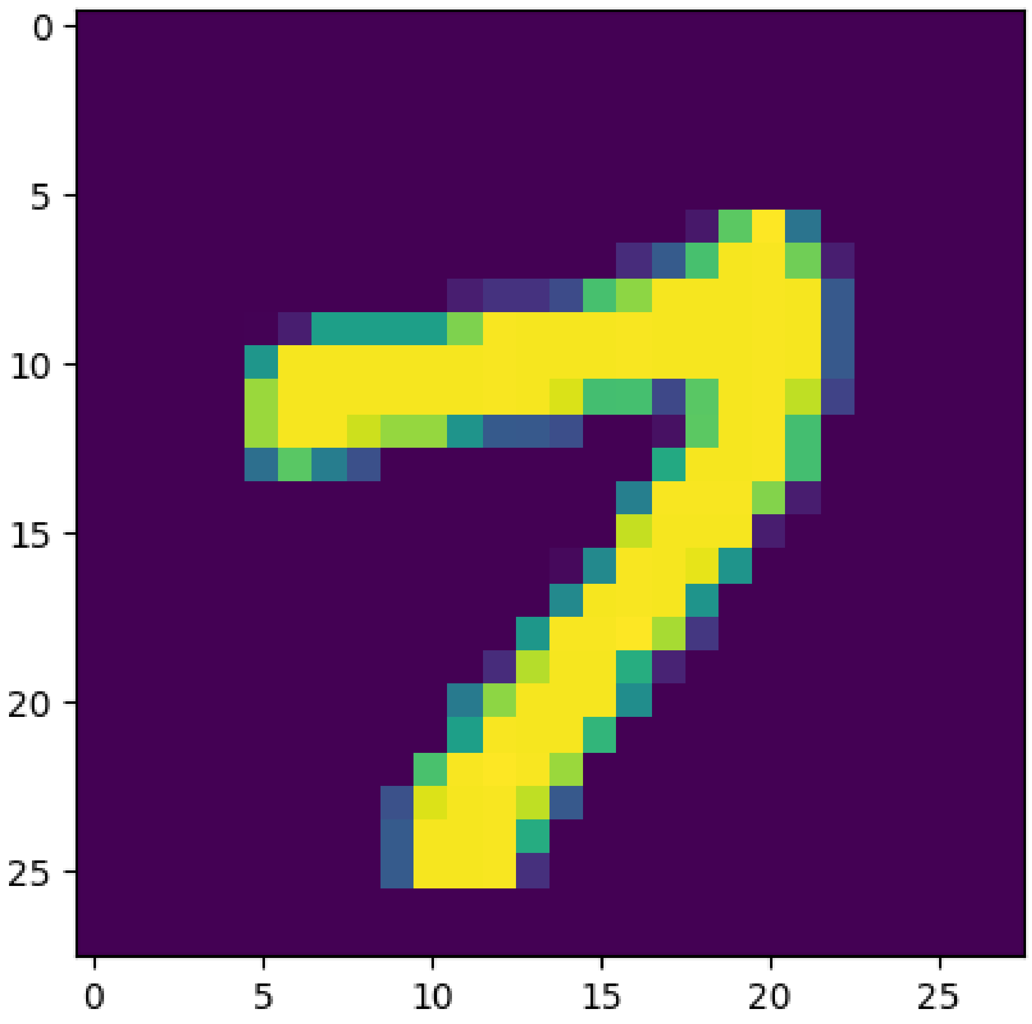
```
g = plt.imshow(X_train[0][:,:,0])
print(y_train_enc[0])
```

```
[0. 0. 0. 0. 0. 0. 0. 0. 1. 0.]
```



```
g  = plt.imshow(X_train[9][:,:,0])
print(y_train_enc[9])
```

```
[0. 0. 0. 0. 0. 0. 0. 1. 0. 0.]
```



5.6 CNN

```
# utility functions
from tensorflow.keras.utils import to_categorical
# sequential model
from tensorflow.keras.models import Sequential
# layers
from tensorflow.keras.layers import Conv2D, MaxPool2D, Dense, Flatten,
Dropout
```

5.7 Model parameters

```
INPUT_SHAPE  =  (28,28,1)
OUTPUT_SHAPE = 10
BATCH_SIZE  =  128
EPOCHS = 10
VERBOSE  =  1
```

5.8 Define CNN Model

```
model  =  Sequential()

model.add(Conv2D(32, kernel_size=(3,3), activation='relu',
input_shape=INPUT_SHAPE))
```

```
model.add(MaxPool2D((2,2)))

model.add(Conv2D(64, kernel_size=(3,3), activation='relu'))
model.add(MaxPool2D((2,2)))

model.add(Flatten())

model.add(Dense(128, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(64, activation='relu'))
model.add(Dropout(0.2))

model.add(Dense(10, activation='softmax'))

/usr/local/lib/python3.10/dist-packages/keras/src/layers/
convolutional/base_conv.py:107: UserWarning: Do not pass an
`input_shape`/`input_dim` argument to a layer. When using Sequential
models, prefer using an `Input(shape)` object as the first layer in
the model instead.
  super().__init__(activity_regularizer=activity_regularizer,
**kwargs)
```

5.9 Compile model

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

5.10 Model summary

```
model.summary()

Model: "sequential"

┌──────────────────────────────────────┬─────────────────────────────┬
────────────────┐
│ Layer (type)                         │ Output Shape                │
Param # │
├──────────────────────────────────────┼─────────────────────────────┼
────────────────┤
│ conv2d (Conv2D)                      │ (None, 26, 26, 32)          │
320 │
├──────────────────────────────────────┼─────────────────────────────┼
────────────────┤
│ max_pooling2d (MaxPooling2D)         │ (None, 13, 13, 32)          │
0 │
├──────────────────────────────────────┼─────────────────────────────┼
────────────────┤
│ conv2d_1 (Conv2D)                    │ (None, 11, 11, 64)          │
```

```
18,496 │
├──────────────────────────────┼───────────────────────┼
───────────────┤
│ max_pooling2d_1 (MaxPooling2D)        │ (None, 5, 5, 64)            │
0 │
├──────────────────────────────┼───────────────────────┼
───────────────┤
│ flatten (Flatten)                     │ (None, 1600)                │
0 │
├──────────────────────────────┼───────────────────────┼
───────────────┤
│ dense (Dense)                         │ (None, 128)                 │
204,928 │
├──────────────────────────────┼───────────────────────┼
───────────────┤
│ dropout (Dropout)                     │ (None, 128)                 │
0 │
├──────────────────────────────┼───────────────────────┼
───────────────┤
│ dense_1 (Dense)                       │ (None, 64)                  │
8,256 │
├──────────────────────────────┼───────────────────────┼
───────────────┤
│ dropout_1 (Dropout)                   │ (None, 64)                  │
0 │
├──────────────────────────────┼───────────────────────┼
───────────────┤
│ dense_2 (Dense)                       │ (None, 10)                  │
650 │
└──────────────────────────────┴───────────────────────┴
───────────────┘
```

```
Total params: 232,650 (908.79 KB)

Trainable params: 232,650 (908.79 KB)

Non-trainable params: 0 (0.00 B)
```

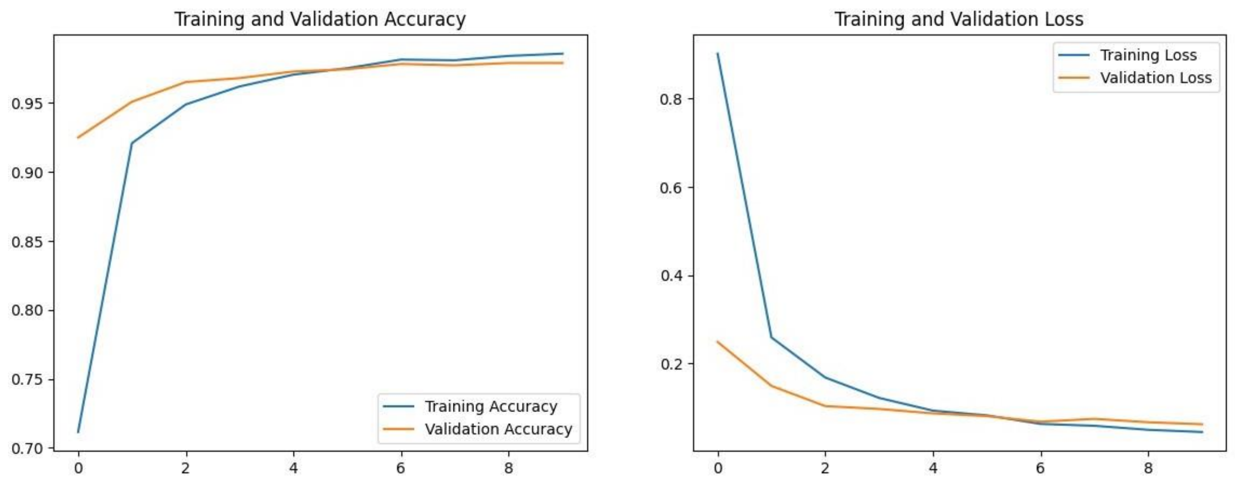
5.11 Model fitting

```
history = model.fit(X_train, y_train_enc,
                    epochs=EPOCHS,
                    batch_size=BATCH_SIZE,
                    verbose=VERBOSE,
                    validation_split=0.3)

Epoch 1/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 10s 102ms/step - accuracy: 0.5162 - loss:
1.4503 - val_accuracy: 0.9250 - val_loss: 0.2487
Epoch 2/10
```

```
77/77 ━━━━━━━━━━━━━━━━━━━━ 9s 110ms/step - accuracy: 0.9107 - loss: 0.2977
- val_accuracy: 0.9510 - val_loss: 0.1491
Epoch 3/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 11s 115ms/step - accuracy: 0.9423 - loss: 0.1824 -
val_accuracy: 0.9652 - val_loss: 0.1033
Epoch 4/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 15s 181ms/step - accuracy: 0.9585 - loss: 0.1286 -
val_accuracy: 0.9681 - val_loss: 0.0969
Epoch 5/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 15s 116ms/step - accuracy: 0.9691 - loss: 0.0983 -
val_accuracy: 0.9729 - val_loss: 0.0868
Epoch 6/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 13s 166ms/step - accuracy: 0.9745 - loss: 0.0877 -
val_accuracy: 0.9745 - val_loss: 0.0805
Epoch 7/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 17s 114ms/step - accuracy: 0.9831 - loss: 0.0648 -
val_accuracy: 0.9783 - val_loss: 0.0680
Epoch 8/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 13s 152ms/step - accuracy: 0.9825 - loss: 0.0554 -
val_accuracy: 0.9774 - val_loss: 0.0744
Epoch 9/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 10s 125ms/step - accuracy: 0.9835 - loss: 0.0496 -
val_accuracy: 0.9790 - val_loss: 0.0667
Epoch 10/10
77/77 ━━━━━━━━━━━━━━━━━━━━ 7s 94ms/step - accuracy: 0.9841 - loss: 0.0486 -
val_accuracy: 0.9790 - val_loss: 0.0622
```

Accurayc and loss

```
plt.figure(figsize=(14, 5))

plt.subplot(1, 2, 1)
plt.plot(history.history['accuracy'], label='Training Accuracy')
plt.plot(history.history['val_accuracy'], label='Validation Accuracy')
plt.legend(loc='lower right')
plt.title('Training and Validation Accuracy')

plt.subplot(1, 2, 2)
plt.plot(history.history['loss'], label='Training Loss')
plt.plot(history.history['val_loss'], label='Validation Loss')
plt.legend(loc='upper right')
plt.title('Training and Validation Loss')

plt.savefig('./loss.png')
plt.show()
```



5.12 Evaluating on validation dataset

```
# model loss and accuracy on validation set
model.evaluate(X_val, y_val_enc, verbose=1)

188/188 ━━━━━━━━━━━━━━━━━━━━ 4s 19ms/step - accuracy: 0.9778 - loss:0.0811

[0.06753452867269516, 0.9810000061988831]

# predicted values
y_pred_enc = model.predict(X_val)

# actual
y_act = [np.argmax(i) for i in y_val_enc]
```

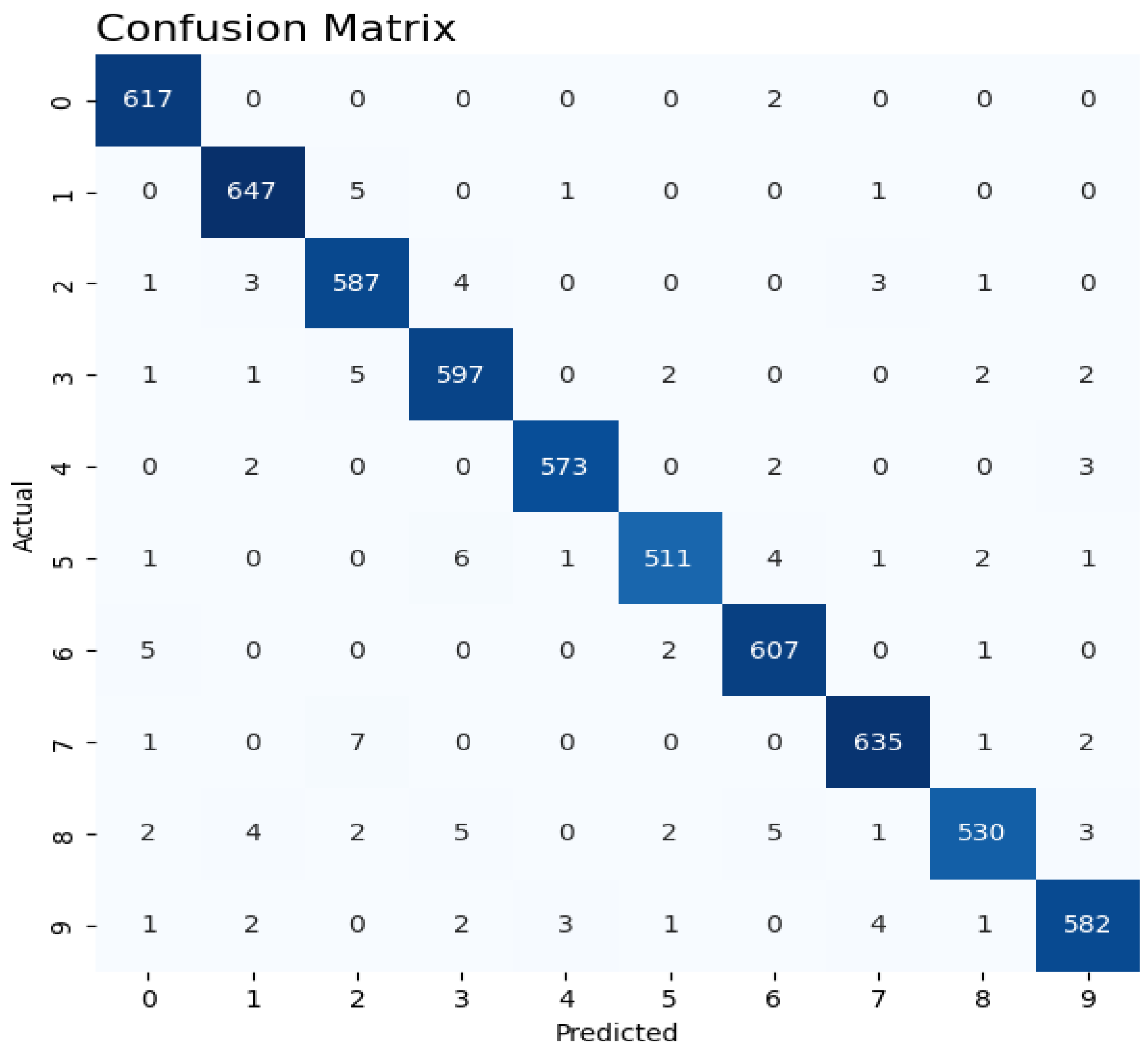
```
# decoding predicted values
y_pred = [np.argmax(i) for i in y_pred_enc]

print(y_pred_enc[0])
print(y_pred[0])
```

```
188/188 ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 5s  22ms/step
[1.0198731e-04  1.0614760e-07  2.6230978e-07  1.5416976e-08  4.3098404e-05
  7.4584415e-07 9.9985337e-01 1.9907452e-08 1.6794039e-07 3.4765679e-07]
6
```

```
print(classification_report(y_act, y_pred))
```

```
              precision    recall  f1-score   support

0                 0.98      1.00      0.99       619
1                 0.98      0.99      0.99       654
2                 0.97      0.98      0.97       599
3                 0.97      0.98      0.98       610
4                 0.99      0.99      0.99       580
5                 0.99      0.97      0.98       527
6                 0.98      0.99      0.98       615
7                 0.98      0.98      0.98       646
8                 0.99      0.96      0.97       554
9                 0.98      0.98      0.98       596

accuracy                              0.98      6000
macro avg         0.98      0.98      0.98      6000
weighted avg      0.98      0.98      0.98      6000
```

```python
fig, ax = plt.subplots(figsize=(7, 7))
sns.heatmap(confusion_matrix(y_act, y_pred), annot=True,
            cbar=False, fmt='1d', cmap='Blues', ax=ax)
ax.set_title('Confusion Matrix', loc='left', fontsize=16)
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
plt.show()
```



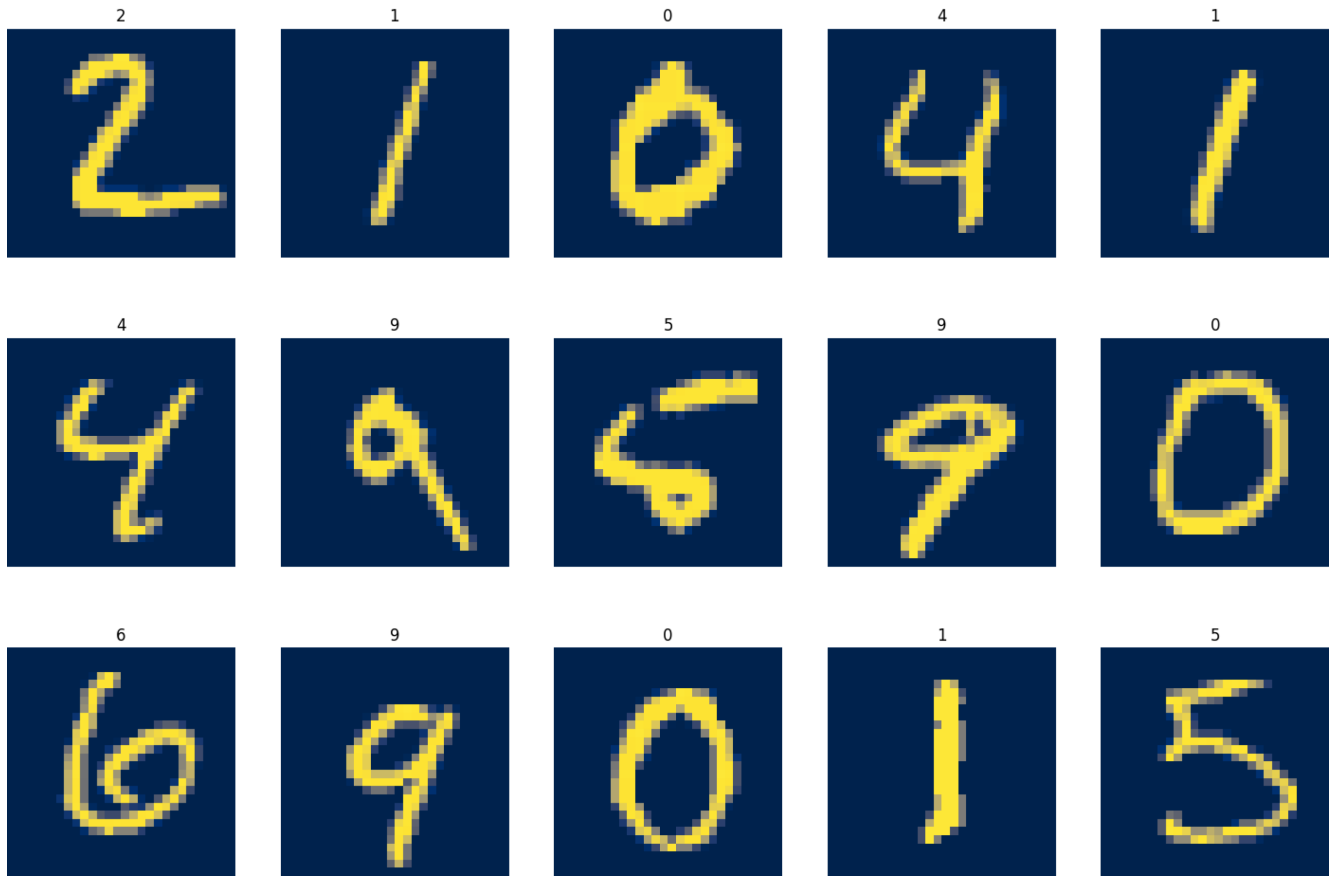
	0	1	2	3	4	5	6	7	8	9
0	617	0	0	0	0	0	2	0	0	0
1	0	647	5	0	1	0	0	1	0	0
2	1	3	587	4	0	0	0	3	1	0
3	1	1	5	597	0	2	0	0	2	2
4	0	2	0	0	573	0	2	0	0	3
5	1	0	0	6	1	511	4	1	2	1
6	5	0	0	0	0	2	607	0	1	0
7	1	0	7	0	0	0	0	635	1	2
8	2	4	2	5	0	2	5	1	530	3
9	1	2	0	2	3	1	0	4	1	582

5.13 Predicting on test

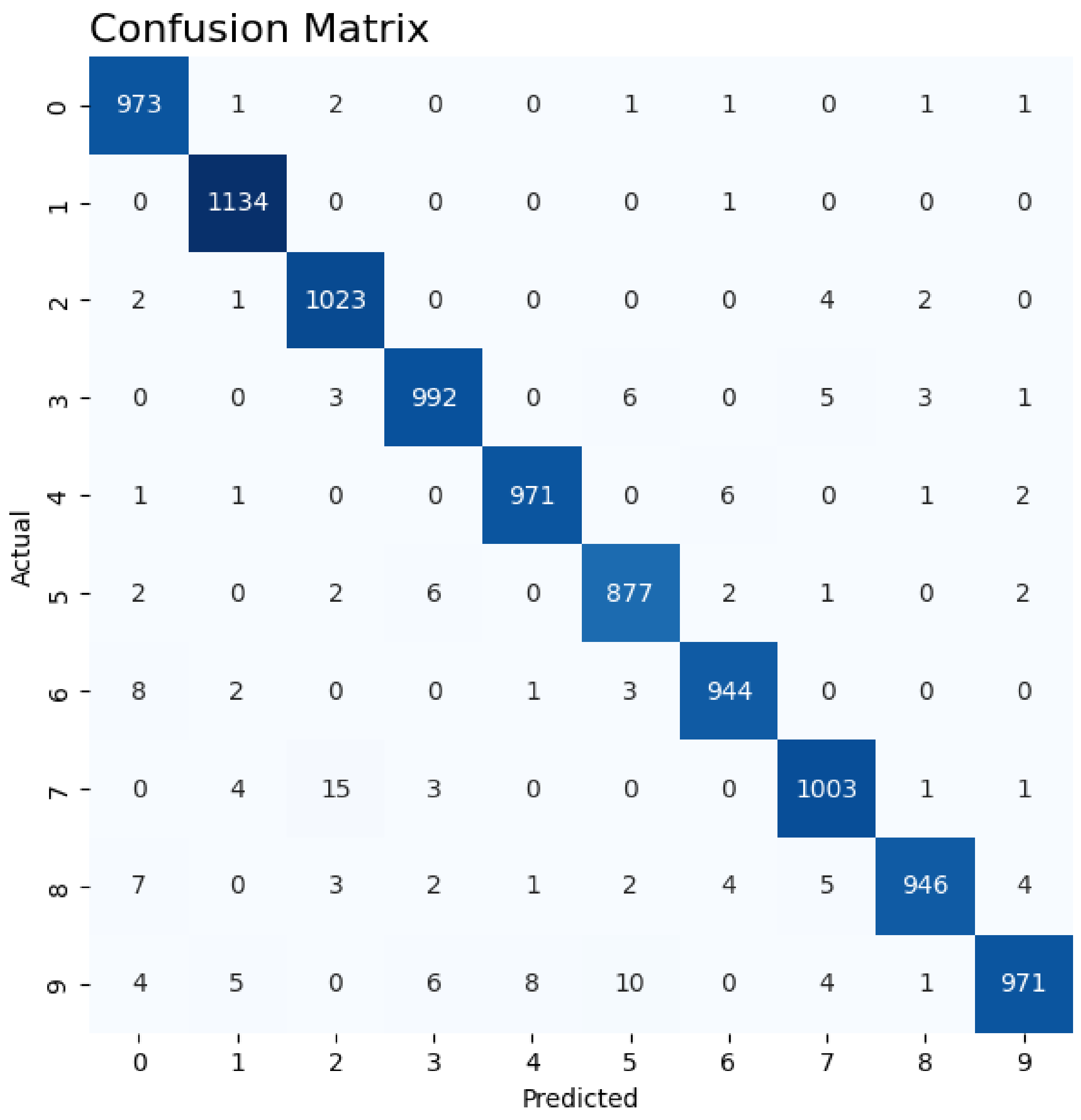
```
# predicted values
y_pred_enc  =  model.predict(X_test)

# decoding predicted values
y_pred = [np.argmax(i) for i in y_pred_enc]

print(y_pred_enc[0])
print(y_pred[0])

313/313 ──────────────────────── 4s  13ms/step
[2.4079909e-09 2.3551547e-07  9.9999970e-01  4.6285633e-09  3.4621056e-10
 8.6640005e-13  8.5211538e-10  1.3785962e-11  6.5881103e-12  3.5365303e-
```

```
13]
2

# predicted targets of each images
# (labels above the images are predicted labels)
fig, ax = plt.subplots(figsize=(18, 12))
for ind, row in enumerate(X_test[:15]):
    plt.subplot(3, 5, ind+1)
    plt.title(y_pred[ind])    img
    = row.reshape(28, 28)
    fig.suptitle('Predicted values', fontsize=24)
    plt.axis('off')
    plt.imshow(img, cmap='cividis')

<ipython-input-39-644aa9b41e7e>:5: MatplotlibDeprecationWarning: Auto-
removal of overlapping axes is deprecated since 3.6 and will be
removed two minor releases later; explicitly call ax.remove() as
needed.
  plt.subplot(3, 5, ind+1)
```

Predicted values



```
fig, ax = plt.subplots(figsize=(7, 7))
sns.heatmap(confusion_matrix(y_test, y_pred), annot=True,
            cbar=False, fmt='1d', cmap='Blues', ax=ax)
ax.set_title('Confusion Matrix', loc='left', fontsize=16)
ax.set_xlabel('Predicted')
ax.set_ylabel('Actual')
plt.show()
```


Confusion Matrix

Actual \ Predicted	0	1	2	3	4	5	6	7	8	9
0	973	1	2	0	0	1	1	0	1	1
1	0	1134	0	0	0	0	1	0	0	0
2	2	1	1023	0	0	0	0	4	2	0
3	0	0	3	992	0	6	0	5	3	1
4	1	1	0	0	971	0	6	0	1	2
5	2	0	2	6	0	877	2	1	0	2
6	8	2	0	0	1	3	944	0	0	0
7	0	4	15	3	0	0	0	1003	1	1
8	7	0	3	2	1	2	4	5	946	4
9	4	5	0	6	8	10	0	4	1	971

```
print(classification_report(y_test, y_pred))
              precision    recall  f1-score   support

           0       0.98      0.99      0.98       980
```

```
1               0.99      1.00      0.99    1135
2               0.98      0.99      0.98    1032
3               0.98      0.98      0.98    1010
4               0.99      0.99      0.99    982
5               0.98      0.98      0.98    892
6               0.99      0.99      0.99    958
7               0.98      0.98      0.98    1027
8               0.99      0.97      0.98    974
9               0.99      0.96      0.98    1009

accuracy                            0.98    9999
macro avg       0.98      0.98      0.98    9999
weighted avg    0.98      0.98      0.98    9999
```

Final output of the test: “Test accuracy is 98%”.

6. Conclusion

This work has aimed to implement and evaluate a CNN for classifying handwritten digits using the MNIST database. This paper follows through in a step-by-step process from data preprocessing, model development, training, and evaluation to attain our main objective: developing a high-performance digit recognition system.

6.1 Key Achievements

Model Performance: The CNN model was really impressive, producing an accuracy of 98% on the MNIST test dataset. It was very effective at recognizing handwritten digits. Performance compared to the traditional machine learning methods tested, which were at Logistic Regression-91.8%, Support Vector Machines at 97%, Random Forest at 95.9%, and k-NN at 95.9%, represents a major improvement.

Robust Architecture: The developed CNN architecture with convolutional layers, max pooling, dropout for regularization, and dense layers proved very powerful for the given image classification task. High digit class accuracy reflects the robustness and generalization capability of the model.

Deep Performance Evaluation: A deeper understanding of performance characteristics and learning dynamics was gained through the use of confusion matrices, classification reports, and learning curves.

6.2 Importance of Work

Benchmark Performance: Achieving an accuracy of 98% on the MNIST dataset places our model in an almost top-tier performance bracket-indeed a testament to our approach being quite successful.

Educational Value: The project proved to be a brilliant hands-on opportunity for each and every stage of the machine learning pipeline, right from preprocessing to deployment of the model in computer vision.

Practical Applications: The developed model will find potential applications in many digit recognition areas, including automated form processing, reading of postal codes, and license plate recognition.

6.3 Learnings and Insights

Superiority of CNN: In this project, the superiority of CNNs as compared to traditional machine learning methods for image classification problems, specifically related to accuracy and capability in feature learning.

The success of the model showed how important it is to do an appropriate architecture design, including choosing the right type of layer, activation function, and regularization technique.

Preprocessing of Data: The project has underlined the importance of proper preprocessing of data in order to obtain the best model results.

Visualization Techniques: We learned various visualization techniques that are helpful for understanding model behavior, interpreting the results, and presenting the findings effectively.

6.4 Limitation and Further Work

Even though this project has been successful in most of the primary aims that were proposed, there are a few directions in which further improvements could be explored and work continued:

Further Model Optimization: Extensive experimentation with hyperparameters, architecture variations, and advanced techniques such as data augmentation should be able to yield higher accuracy than found in this study.

Testing Generalization: The model needs to be thoroughly tested on some other datasets of handwritten digit data, or in general on real-world tasks.

Deployment Considerations: Much is to be explored in model compression techniques and deployment strategies for resource-constrained environments in order to let this system be of practical use. Recognizing multi-digit numbers: A natural extension, given this base, would be extending toward the recognition of multi-digit numbers or perhaps handwritten text.

Eventually, it I have achieved the result of giving rise to a most accurate digit recognition system and provided valuable experience in applying deep learning techniques for solving real-world problems. Knowledge and skills acquired during the process will provide a good background for more complex computer vision and machine learning tasks.

7. Bibliography:

Based on the content and technologies used in the project, the following resources are used: Guided and taught by Programming Parthashla.

Chollet, F. (2018). Deep Learning with Python. Manning Publications.

- Provides a comprehensive guide to deep learning with Keras and TensorFlow

Géron, A. (2019). Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow. O'Reilly Media.

- Offers practical implementations of machine learning algorithms

LeCun, Y., Cortes, C., & Burges, C. J. (2010). MNIST handwritten digit database. Available at: http://yann.lecun.com/exdb/mnist/

- The original source of the MNIST dataset

Goodfellow, I., Bengio, Y., & Courville, A. (2016). Deep Learning. MIT Press.

- Provides in-depth theoretical background on deep learning concepts

TensorFlow documentation. Available at: https://www.tensorflow.org/api_docs

- Official documentation for TensorFlow library

Keras documentation. Available at: https://keras.io/api/

- Official documentation for Keras API

Scikit-learn documentation. Available at:

- Comprehensive guide to using scikit-learn for machine learning

Matplotlib documentation. Available at: https://matplotlib.org/stable/contents.html

- Detailed guide on creating various types of plots in Python

Seaborn documentation. Available at: https://seaborn.pydata.org/

- Tutorial and API reference for the Seaborn data visualization library

Krizhevsky, A., Sutskever, I., & Hinton, G. E. (2012). ImageNet classification with deep convolutional neural networks. In Advances in neural information processing systems (pp. 1097-1105).

- Seminal paper on CNNs for image classification

Simonyan, K., & Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. arXiv preprint arXiv:1409.1556.

- Introduces the concept of deep CNN architectures

He, K., Zhang, X., Ren, S., & Sun, J. (2016). Deep residual learning for image recognition. In Proceedings of the IEEE conference on computer vision and pattern recognition (pp. 770-778).

- Presents the ResNet architecture, advancing the state of the art in CNNs