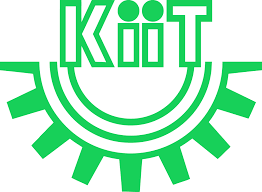
# **Project on** Handwritten Digit Recognition

Project developed by

Under Guidance of

****

**Kalinga Institute of Industrial Technology**

Deemed to be University U/S 3 of the UGC Act, 1956

School of Computer Application

(SCA)

01.05.2023

CERTIFICATE OF ORIGINALITY

# This is to certify that the project report entitled “**Project on** Handwritten Digit Recognition

” Submitted to School of Computer Application, KIIT Deemed to be University in partial fulfilment of the requirement for the award of the degree of Master of Computer Applications (MCA),

is an authentic and original work carried out by Mr. Rohan Pal, with Roll no. 2170108 under my guidance.

The matter embodied in this project is genuine work done by the student and has not been submitted whether to this University or to any other University/Institute for the fulfilment of the requirements of any course of study.

Signature of the Student Signature of the Guide

Date: 02/05/2023 Date:

Name:

School of Computer Applications

KIIT Deemed to be University

Certificate

This is to certify that the project work entitled “**Project on Handwritten Digit Recognition”** submitted by **Rohan Pal** bearing

roll no. **2170108,** is an authentic and original work.

Signature Signature

(Internal Examiner) (External Examiner)

Date: Date:

DECLARATION

I, **Rohan Pal**, with roll no. **2170108** do hereby declare that the project report entitled “**Project on Handwritten Digit Recognition**” submitted to School of Computer Applications, KIIT Deemed to be University, Bhubaneswar for the award of the degree of Master OF Computer Application (MCA), is an authentic and original work carried out by me from 1st January 2023 to 25 th April 2023 at KIIT Deemed to be University

under the guidance of Chinmay Mishra

Signature of the student

Date: 02/05/2023

Acknowledgement

This satisfaction which accompanies the successful completion of any task is incomplete without the mention of those persons whose hands are behind the success. Because the success is the epitome of hard work, prevention, zeal, determination and the most encouraging guidance and advice serving as beacon light and crowing our effort with success. I am grateful to of Chinmay Mishra, the special guide for his endless support and kind cooperation for completion of this project. I also thankful to the faculty members of KIIT Deemed to be University who have constantly strived to keep our normal despite being so far away from us.

**TABLE OF CONTENT**

|  |  |  |
| --- | --- | --- |
| SL.NO. | CONTENTS | PAGE NO. |
| 1. | Chapter 1: Introduction | 8-11 |
|  | 1.1 Introduction to the system | 8 |
|  | 1.2 Project Description | 9-10 |
|  | 1.3 Objectives | 11 |
| 2. | Chapter 2: System Analysis | 12-36 |
|  | 2.1 Identification of need | 12-13 |
|  | 2.2 System Requirement Specification | 14-28 |
|  | 2.2.1 Introduction | 14-16 |
|  | 2.2.2 Software Requirement | 17-28 |
|  | 2.3 Feasibility study | 29-31 |
|  | 2.4 Control flow diagram | 32-34 |
|  | 2.5 Data flow Diagram | 35-36 |
|  | Chapter 3: System Design | 37-38 |
|  | 3.1 Introduction | 37 |
|  | 3.2 Solution Specification | 37 |
|  | Chapter 4: Develop | 39-58 |
|  | 4.1 Implementation | 39-40 |
|  | 4.2 Testing of error | 40-42 |
|  | 4.3 User Interface design | 42-43 |
|  | 4.4 Admin Interface design | 44-50 |
|  | 4.5 Comparative Study | 51-58 |
|  | Chapter 5: System Security | 59-60 |
|  | Chapter 6: Challenges Faced | 61 |
|  | Chapter 7: Future Scope | 62 |
|  | Chapter 8: Conclusion | 63-64 |

1. **Introduction** 
   1. **Introduction to the system**

Handwritten digit recognition is a computer vision task that involves recognizing handwritten digits in an image and interpreting them as numerical values. This task is commonly used in a wide range of applications, such as optical character recognition (OCR), digitizing historical documents, and processing financial forms.

The process of handwritten digit recognition typically involves the following steps:

1. Preprocessing: The input image is preprocessed to enhance its quality, normalize its size and shape, and reduce noise.

2. Feature extraction: Features are extracted from the preprocessed image that represent important characteristics of the digits, such as their shape, size, and orientation.

3. Classification: The extracted features are used to classify the input image into one of several possible classes, each representing a digit from 0 to 9.

Several machine learning algorithms can be used for classification, including support vector machines (SVM), decision trees, and neural networks.

Neural networks, in particular, have shown exceptional performance in handwritten digit recognition tasks. Deep learning models, such as convolutional neural networks (CNNs), have been shown to achieve state-of-the-art results in this task, often outperforming traditional machine learning algorithms.

Handwritten digit recognition has numerous applications in the real world, including automatic mail sorting, signature verification, and digitizing historical documents.

**1.2 Project Description**

Handwritten digit recognition is a computer vision task that involves recognizing digits written by humans in an image. The goal of this project is to develop a machine learning model that can accurately classify handwritten digits into their respective numerical values (0-9).

The project can be broken down into several steps:

1. Dataset Collection: The first step is to collect a dataset of handwritten digits. The MNIST dataset is a popular choice for this task, containing 70,000 images of handwritten digits.

2. Data Preprocessing: The dataset needs to be preprocessed before feeding it to the machine learning model. This involves converting the images into a format that can be easily processed by the model. This may include resizing, normalization, and flattening.

3. Model Selection: There are various machine learning models that can be used for this task, including logistic regression, decision trees, support vector machines, and neural networks. Neural networks have been shown to achieve the highest accuracy for this task, and are commonly used for digit recognition.

4. Model Training: Once a model has been selected, it needs to be trained on the preprocessed data. This involves optimizing the model's parameters to minimize the error between the predicted and actual digit labels.

5. Model Evaluation: After training, the model's accuracy is evaluated on a separate set of test data. This step is important to ensure that the model is not overfitting to the training data.

6. Model Deployment: Finally, the trained model can be deployed in a real-world application. For example, it can be used to recognize handwritten digits on a touchscreen or a scanned document.

Overall, the handwritten digit recognition project is an interesting and challenging application of machine learning and computer vision techniques. With the right approach, it is possible to develop a highly accurate model that can recognize digits with near-human level accuracy.

**1.3 Objective**

The objective of Handwritten Digit Recognition is to develop a computer program or algorithm that can accurately identify and recognize handwritten digits. This is a significant task in the field of machine learning and artificial intelligence, as it involves training a computer to understand and interpret human handwriting.

Handwritten digit recognition has many practical applications, such as in the field of document analysis and processing, where it can be used to automatically digitize and extract information from handwritten forms or documents. It is also used in the field of computer vision and image processing, where it can be used to recognize handwritten digits in images or videos.

The ultimate goal of handwritten digit recognition is to develop algorithms and models that can accurately recognize handwritten digits in a wide range of contexts and applications, with high accuracy and efficiency. This requires the use of advanced machine learning techniques such as deep neural networks and convolutional neural networks, as well as large datasets of labeled handwritten digit images for training and validation.

**2.SYSTEM ANALYSIS**

2.1 Identification of need

Handwritten digit recognition is a common task in the field of machine learning and computer vision. It involves the identification and classification of handwritten digits (usually 0-9) in an image or dataset. This technology has many practical applications, such as:

1. Postal sorting: Handwritten digits are commonly used to represent postal codes on envelopes. Handwritten digit recognition can be used to automatically sort mail according to its destination.

2. Banking: Handwritten digits are used on checks and other financial documents. Digit recognition can be used to read and process these documents more quickly and accurately.

3. Medical diagnosis: Handwritten digits are used to represent numbers in medical records, such as patient IDs or test results. Digit recognition can help automate the process of reading and analyzing these records.

4. Education: Handwritten digit recognition can be used in educational settings to grade tests and quizzes more quickly and accurately.

Overall, the need for handwritten digit recognition arises from the fact that handwritten digits are often used in many important contexts, and automated recognition can save time and improve accuracy compared to manual methods.

**2.2 System Requirement Specification**

**2.2.1 Introduction**

The handwritten digit recognition system is a website application designed to recognize and identify handwritten digits from scanned images or digital input. This system is expected to accurately classify digits ranging from 0 to 9. The system will be used in various applications such as postal automation, signature verification, and financial document processing.

Functional Requirements

1. Image Input: The system should be able to receive scanned images or digital input of handwritten digits for recognition.

2. Pre-processing: The system should have the ability to pre-process the input image, including normalization, filtering, and noise reduction.

3. Segmentation: The system should be able to separate individual digits from the input image if multiple digits are present.

4. Feature Extraction: The system should be able to extract relevant features from the digit image, such as edges, corners, and curves.

5. Classification: The system should be able to classify the digit image based on the extracted features and assign the appropriate label (0 to 9).

6. Output: The system should provide the recognized digit as output.

Non-functional Requirements

1. Accuracy: The system should have a high level of accuracy in digit recognition, with an error rate of less than 1%.

2. Speed: The system should be able to recognize digits quickly, with a processing time of less than 1 second per image.

3. Scalability: The system should be scalable and able to handle a large volume of input images simultaneously.

4. Reliability: The system should be reliable and should not produce false results.

5. User Interface: The system should have a user-friendly interface that allows users to input images easily and view the recognized digits.

6. Compatibility: The system should be compatible with different operating systems and hardware platforms.

7. Security: The system should be secure and protect the privacy of the input images.

Conclusion

The handwritten digit recognition system is an important tool for various applications. The system should have high accuracy, speed, scalability, reliability, and security. The system should also be user-friendly and compatible with different operating systems and hardware platforms. By meeting these requirements, the system will be able to effectively recognize and identify handwritten digits from scanned images or digital input.

**2.2.2 Software Requirement**

**HTML**

The HyperText Markup Language, or HTML is the standard markup language for documents designed to be displayed in a web browser. It can

be assisted by technologies such as Cascading Style Sheets (CSS) and scripting languages such as JavaScript. Web browsers receive HTML documents from a web server or from local storage and render the documents into multimedia web pages. HTML describes the structure of a web page semantically and originally included cues for the appearance of the document. HTML elements are the building blocks of HTML pages. With HTML constructs, images and other objects such as interactive forms may be embedded into the rendered page. HTML provides a means to create structured documents by denoting structural semantics for text such as headings, paragraphs, lists, links, quotes and other items. HTML elements are delineated by *tags*, written using angle brackets. Tags such as <**img** /> and <**input** /> directly introduce content into the page. Other tags such as <**p**> surround and provide information about document text and may include other tags as sub-elements. Browsers do not display the HTML tags, but use them to interpret the content of the page. HTML can embed programs written in a scripting language such as JavaScript, which affects the behavior and content of web pages.

**CSS**

Cascading Style Sheets (CSS) is a style sheet language used for describing the presentation of a document written in a markup language such as HTML. CSS is a cornerstone technology of the World Wide Web, alongside HTML and JavaScript. CSS is designed to enable the separation of presentation and content, including layout, colors, and fonts.[3] This separation can improve content accessibility, provide more flexibility and control in the specification of presentation characteristics, enable multiple web pages to share formatting by specifying the relevant CSS in a separate .css file which reduces complexity and repetition in the structural content as well as enabling the .css file to be cached to improve the page load speed between the pages that share the file and its formatting. Separation of formatting and content also makes it feasible to present the same markup page in different styles for different rendering methods, such as on-screen, in print, by voice (via speech-based browser or screen reader), and on Braille-based tactile devices. CSS also has rules for alternate formatting if the content is accessed on a mobile device. The name *cascading* comes from the specified priority scheme to determine which style rule applies if more than one rule matches a

particular element. This cascading priority scheme is predictable. The CSS specifications are maintained by the World Wide Web Consortium (W3C). Internet media type (MIME type) text/css is registered for use with CSS by RFC 2318 (March 1998). The W3C operates a free CSS validation service for CSS documents.

**JavaScript**

JavaScript is a scripting or programming language that allows you to implement complex features on web pages — every time a web page does more than just sit there and display static information for you to look at — displaying timely content updates, interactive maps, animated 2D/3D graphics, scrolling video jukeboxes, etc. — you can bet that JavaScript is probably involved. It is the third layer of the layer cake of standard web technologies, two of which ([HTML](https://developer.mozilla.org/en-US/docs/Learn/HTML) and [CSS](https://developer.mozilla.org/en-US/docs/Learn/CSS)) we have covered in much more detail in other parts of the Learning Area. avaScript is applied to your HTML page in a similar manner to CSS. Whereas CSS uses  [<link>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/link)  elements to apply external stylesheets and  [<style>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/style)  elements to apply internal stylesheets to HTML, JavaScript only needs one friend in the world of HTML — the  [<script>](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/script)  element. Incorporating JavaScript improves the user experience of the web page by converting it from a static page into an interactive one. To recap, JavaScript adds **behavior** to web pages.

**Python**

Python is a high-level, general-purpose programming language. Its design philosophy emphasizes code readability with the use of significant indentation via the off-side rule.

Python is dynamically typed and garbage-collected. It supports multiple programming paradigms, including structured (particularly procedural), object-oriented and functional programming. It is often described as a "batteries included" language due to its comprehensive standard library.

Guido van Rossum began working on Python in the late 1980s as a successor to the ABC programming language and first released it in 1991 as Python 0.9.0. Python 2.0 was released in 2000. Python 3.0, released in 2008, was a major revision not completely backward-compatible with earlier versions. Python 2.7.18, released in 2020, was the last release of Python 2. Python consistently ranks as one of the most popular programming languages.

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC programming language, which was inspired by SETL, capable of exception handling and interfacing with the Amoeba operating system. Its implementation began in December 1989. Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018, when he announced his "permanent vacation" from his responsibilities as Python's "benevolent dictator for life", a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker. In January 2019, active Python core developers elected a five-member Steering Council to lead the project.

**NumPy**

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. The predecessor of NumPy, Numeric, was originally created by Jim Hugunin with contributions from several other developers. In 2005, Travis Oliphant created NumPy by incorporating features of the competing Numarray into Numeric, with extensive modifications. NumPy is open-source software and has many contributors. NumPy is a NumFOCUS fiscally sponsored project. NumPy targets the CPython reference implementation of Python, which is a non-optimizing bytecode interpreter. Mathematical algorithms written for this version of Python often run much slower than compiled equivalents due to the absence of compiler optimization. NumPy addresses the slowness problem partly by providing multidimensional arrays and functions and operators that operate efficiently on arrays; using these requires rewriting some code, mostly inner loops, using NumPy.Using NumPy in Python gives functionality comparable to MATLAB since they are both interpreted, and they both allow the user to write fast programs as long as most operations work on arrays or matrices instead of scalars. In comparison, MATLAB boasts a large number of additional toolboxes, notably Simulink, whereas NumPy is intrinsically integrated with Python, a more modern and complete programming language. Moreover, complementary Python packages are available; SciPy is a library that adds more MATLAB-like functionality and Matplotlib is a plotting package that provides MATLAB-like plotting functionality. Internally, both MATLAB and NumPy rely on BLAS and LAPACK for efficient linear algebra computations.

**Matplotlib**

Matplotlib is a plotting library for the Python programming language and its numerical mathematics extension NumPy. It provides an object-oriented API for embedding plots into applications using general-purpose GUI toolkits like Tkinter, wxPython, Qt, or GTK. There is also a procedural "pylab" interface based on a state machine (like OpenGL), designed to closely resemble that of MATLAB, though its use is discouraged. SciPy makes use of Matplotlib.

Matplotlib was originally written by John D. Hunter. Since then it has had an active development community and is distributed under a BSD-style license. Michael Droettboom was nominated as matplotlib's lead developer shortly before John Hunter's death in August 2012 and was further joined by Thomas Caswell. Matplotlib is a NumFOCUS fiscally sponsored project.

Matplotlib 2.0.x supports Python versions 2.7 through 3.10. Python 3 support started with Matplotlib 1.2. Matplotlib 1.4 is the last version to support Python 2.6. Matplotlib has pledged not to support Python 2 past 2020 by signing the Python 3 Statement.

Several toolkits are available which extend Matplotlib functionality. Some are separate downloads, others ship with the Matplotlib source code but have external dependencies.

Basemap: map plotting with various map projections, coastlines, and political boundaries

Cartopy: a mapping library featuring object-oriented map projection definitions, and arbitrary point, line, polygon and image transformation capabilities. (Matplotlib v1.2 and above)

Excel tools: utilities for exchanging data with Microsoft Excel

GTK tools: interface to the GTK library

Qt interface

Mplot3d: 3-D plots

Natgrid: interface to the natgrid library for gridding irregularly spaced data.

tikzplotlib: export to Pgfplots for smooth integration into LaTeX documents (formerly known as matplotlib2tikz)

**Tensorflow**

TensorFlow is an open-source software library developed by Google for building and training machine learning models. It was first released in 2015 and has since become one of the most widely used libraries for machine learning and deep learning.

TensorFlow provides a comprehensive set of tools and APIs for building and training a variety of machine learning models, from simple linear regression to complex deep neural networks. It supports a wide range of platforms, including CPUs, GPUs, and TPUs, and can be used in various programming languages, including Python, C++, Java, and Go.

Some key features of TensorFlow include:

1. Automatic differentiation: TensorFlow can automatically compute gradients for any mathematical expression, which makes it easy to train models using gradient-based optimization algorithms such as stochastic gradient descent.

2. High-level APIs: TensorFlow provides high-level APIs such as Keras and Estimators, which simplify the process of building and training machine learning models.

3. Distributed training: TensorFlow can distribute training across multiple devices, allowing for faster training of large models.

4. Visualization tools: TensorFlow provides tools for visualizing model architectures and monitoring training progress, which can help in debugging and optimizing models.

1. Pre-built models and datasets: TensorFlow provides pre-built models and datasets that can be used for a variety of machine learning tasks, such as image classification and natural language processing.

6. Support for multiple platforms: TensorFlow provides APIs for running machine learning models on a variety of platforms, including mobile devices and the web. For example, TensorFlow Lite is a lightweight version of TensorFlow designed for mobile and embedded devices, while TensorFlow.js allows models to be run in web browsers.

7. Customizable operations: TensorFlow provides a low-level API for defining custom operations, which can be used to create new layers or activation functions for neural networks, or to implement specialized mathematical operations.

8. Model serving: TensorFlow provides tools for serving machine learning models in production environments, such as web services or mobile apps. TensorFlow Serving is a dedicated framework for serving TensorFlow models, with features such as model versioning, load balancing, and distributed serving.

9. Integration with other libraries and frameworks: TensorFlow can be integrated with other popular machine learning libraries and frameworks, such as scikit-learn, PyTorch, and Keras. This allows users to combine the strengths of different libraries and leverage pre-built models and tools.

10. Active development community: TensorFlow has a large and active community of developers, users, and contributors, who provide support, contribute to the development of new features and extensions, and share knowledge and resources through forums, meetups, and online communities.

TensorFlow has been used in a wide range of applications, including computer vision, natural language processing, robotics, and reinforcement learning. Its versatility and powerful features make it a popular choice among machine learning practitioners and researchers.

Overall, TensorFlow provides a rich set of tools and capabilities for building, training, and deploying machine learning models across a wide range of applications and platforms. Its flexibility, scalability, and powerful features make it a popular choice for machine learning practitioners and researchers.

**Keras**

Keras is a high-level open-source software library for building and training neural networks. It was developed by Francois Chollet and first released in 2015. Keras provides a user-friendly and intuitive interface for building neural networks, making it a popular choice for both beginners and experienced machine learning practitioners.

Some key features of Keras include:

1. High-level API: Keras provides a high-level API for building and training neural networks, which allows users to focus on the design of the network architecture and the training process, rather than on low-level implementation details.

2. Compatibility with multiple backends: Keras can be used with multiple backends, including TensorFlow, Theano, and Microsoft Cognitive Toolkit (CNTK), providing flexibility and choice to users.

3. Modular and flexible architecture: Keras provides a modular and flexible architecture for building neural networks, allowing users to easily add or remove layers, change activation functions, and modify other network parameters.

4. Pre-built models and datasets: Keras provides a collection of pre-built models and datasets, which can be used for a variety of machine learning tasks, such as image classification and natural language processing.

5. Visualization tools: Keras provides tools for visualizing model architectures and monitoring training progress, which can help in debugging and optimizing models.

1. Transfer learning: Keras supports transfer learning, which allows users to reuse pre-trained models and fine-tune them for new tasks, thus reducing the need for large amounts of training data.

7. Customizable layers: Keras provides a variety of pre-built layers, such as convolutional, recurrent, and dense layers, but also allows users to define custom layers with specific activation functions, constraints, or regularizers.

8. Callbacks: Keras provides a mechanism called "callbacks" that allows users to customize the training process by monitoring training progress, stopping training if performance does not improve, or saving model checkpoints at certain intervals.

9. Hyperparameter tuning: Keras provides tools for hyperparameter tuning, such as grid search or randomized search, which allow users to optimize the performance of their models by exploring different combinations of hyperparameters.

10. Multi-GPU support: Keras provides support for multi-GPU training, allowing users to accelerate the training process by distributing the workload across multiple GPUs.

11. Support for recurrent neural networks (RNNs): Keras provides a range of pre-built RNN layers, such as LSTM and GRU, which are widely used in natural language processing and sequential data analysis.

12. Integration with other libraries and frameworks: Keras can be integrated with other popular machine learning libraries and frameworks, such as TensorFlow, PyTorch, and scikit-learn. This allows users to combine the strengths of different libraries and leverage pre-built models and tools.

Keras has been widely used in a variety of applications, including computer vision, natural language processing, and reinforcement learning. Its ease of use and flexibility make it a popular choice among machine learning practitioners and researchers. Additionally, Keras has been integrated into TensorFlow as the default high-level API, providing even more flexibility and power to users.

Overall, Keras provides a user-friendly and flexible interface for building and training neural networks, with a rich set of tools and capabilities for customizing and optimizing models. Its compatibility with multiple backends and its integration with other libraries and frameworks make it a versatile and powerful tool for machine learning practitioners and researchers.

**Opencv-Python**

OpenCV (Open Source Computer Vision) is an open-source computer vision and machine learning software library. It was first developed by Intel in the late 1990s and has since been used in a wide range of applications, including robotics, augmented reality, and self-driving cars. OpenCV is written in C++ and has bindings for multiple programming languages, including Python.

The opencv-python library provides Python bindings for OpenCV, allowing users to access OpenCV's functionality in Python. Some key features of opencv-python include:

1. Image and video processing: opencv-python provides a range of functions for processing images and videos, including functions for loading and saving images, resizing, cropping, and transforming images, applying filters and color maps, and detecting edges and features in images.

2. Object detection and recognition: opencv-python provides functions for object detection and recognition, including functions for detecting faces, eyes, and other facial features, and for detecting and tracking objects in videos.

3. Machine learning: opencv-python provides tools for machine learning, including functions for training and using machine learning models, such as support vector machines (SVMs) and neural networks.

4. Camera calibration and 3D reconstruction: opencv-python provides functions for camera calibration, which is essential for obtaining accurate measurements from images and videos, and for 3D reconstruction, which involves reconstructing 3D models from multiple 2D images.

5. Graphical user interfaces: opencv-python provides tools for creating graphical user interfaces (GUIs) for image and video processing applications, including functions for displaying images and videos and for handling user input.

6. Integration with other libraries and frameworks: opencv-python can be integrated with other popular machine learning libraries and frameworks, such as TensorFlow and PyTorch, and with libraries for scientific computing and data analysis, such as NumPy and Pandas.

7. Real-time image and video processing: opencv-python provides tools for real-time image and video processing, allowing users to process and analyze images and videos in real-time using a webcam or other camera source.

8. Stereo vision: opencv-python provides functions for stereo vision, which involves capturing and analyzing images from two or more cameras to obtain depth information and create 3D models.

9. Optical flow: opencv-python provides functions for computing optical flow, which involves tracking the motion of objects in a video over time.

10. Feature detection and matching: opencv-python provides functions for detecting and matching features in images, which is useful for tasks such as image registration, object recognition, and 3D reconstruction.

11. Image segmentation: opencv-python provides functions for image segmentation, which involves dividing an image into multiple segments or regions based on characteristics such as color, texture, or shape.

12. High-level computer vision tasks: opencv-python provides tools for high-level computer vision tasks, such as scene recognition, image classification, and object tracking.

Opencv-python provides a powerful set of tools and capabilities for image and video processing, object detection and recognition, machine learning, camera calibration, and 3D reconstruction. Its Python bindings make it accessible and easy to use for Python programmers, and its integration with other libraries and frameworks makes it a versatile tool for a wide range of applications.

Overall, opencv-python is a powerful and versatile library for computer vision and machine learning applications, with a wide range of tools and capabilities for image and video processing, object detection and recognition, machine learning, camera calibration, and 3D reconstruction. Its Python bindings and integration with other libraries and frameworks make it accessible and easy to use, and its support for real-time processing and high-level computer vision tasks make it a valuable tool for a variety of applications.

**Anaconda**

Anaconda is a distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.), that aims to simplify package management and deployment. The distribution includes data-science packages suitable for Windows, Linux, and macOS. It is developed and maintained by Anaconda, Inc., which was founded by Peter Wang and Travis Oliphant in 2012. As an Anaconda, Inc. product, it is also known as Anaconda Distribution or Anaconda Individual Edition, while other products from the company are Anaconda Team Edition and Anaconda Enterprise Edition, both of which are not free. Package versions in Anaconda are managed by the package management system conda. This package manager was spun out as a separate open-source package as it ended up being useful on its own and for things other than Python. There is also a small, bootstrap version of Anaconda called Miniconda, which includes only conda, Python, the packages they depend on, and a small number of other packages. Anaconda distribution comes with over 250 packages automatically installed, and over 7,500 additional open-source packages can be installed from PyPI as well as the conda package and virtual environment manager. It also includes a GUI, Anaconda Navigator,[12] as a graphical alternative to the command-line interface (CLI). The big difference between conda and the pip package manager is in how package dependencies are managed, which is a significant challenge for Python data science and the reason conda exists. Anaconda distribution comes with over 250 packages automatically installed, and over 7,500 additional open-source packages can be installed from PyPI as well as the conda package and virtual environment manager. It also includes a GUI, Anaconda Navigator, as a graphical alternative to the command-line interface (CLI). The big difference between conda and the pip package manager is in how package dependencies are managed, which is a significant challenge for Python data science and the reason conda exists.Anaconda distribution comes with over 250 packages automatically installed, and over 7,500 additional open-source packages can be installed from PyPI as well as the conda package and virtual environment manager. It also includes a GUI, Anaconda Navigator, as a graphical alternative to the command-line interface (CLI). The big difference between conda and the pip package manager is in how package dependencies are managed, which is a significant challenge for Python data science and the reason conda exists.

**Flask Server**

Flask is a web application framework written in Python. It was developed by Armin Ronacher, who led a team of international Python enthusiasts called Poocco. Flask is based on the Werkzeg WSGI toolkit and the Jinja2 template engine.Both are Pocco projects. The Web Server Gateway Interface (Web Server Gateway Interface, WSGI) has been used as a standard for Python web application development. WSGI is the specification of a common interface between web servers and web applications. Werkzeug is a WSGI toolkit that implements requests, response objects, and utility functions. This enables a web frame to be built on it. The Flask framework uses Werkzeg as one of its bases. jinja2 is a popular template engine for Python.A web template system combines a template with a specific data source to render a dynamic web page. Flask is often referred to as a microframework. It is designed to keep the core of the application simple and scalable. Instead of an abstraction layer for database support, Flask supports extensions to add such capabilities to the application. Unlike the Django framework, Flask is very Pythonic. It’s easy to get started with Flask, because it doesn’t have a huge learning curve. On top of that it’s very explicit, which increases readability. To create the “Hello World” app, you only need a few lines of code.

**2.3 Feasibility study**

Handwritten digit recognition is a popular application of machine learning and computer vision. It involves training a computer program to recognize handwritten digits and classify them into one of the ten possible categories (0 through 9). The feasibility of a handwritten digit recognition system depends on several factors, including the accuracy of the recognition, the speed of processing, and the ease of use.

Accuracy:

The accuracy of a handwritten digit recognition system is crucial for its feasibility. A high level of accuracy is necessary to ensure that the system can reliably recognize and classify handwritten digits. The accuracy of a system can be measured using various metrics, including precision, recall, and F1 score. The higher the accuracy, the more feasible the system is for practical use.

Data Availability:

To train a machine learning model, a sufficient amount of data is required. A feasible handwritten digit recognition system requires a significant amount of labeled training data. The training data should be representative of the actual data that the system will encounter in the real world. If there is not enough data, the system may not be accurate enough to be considered feasible.

Model Complexity:

The complexity of the machine learning model used for handwritten digit recognition can also impact its feasibility. A complex model may have higher accuracy but may require more processing power and time to train and make predictions. Simpler models may be less accurate, but they are easier to train and may be more practical for some applications.

Processing Time:

The processing time of a handwritten digit recognition system is another important factor in determining its feasibility. The system should be able to process inputs quickly enough to be practical for real-time applications. If the processing time is too slow, the system may not be useful in practical settings.

Ease of Use:

The ease of use of a handwritten digit recognition system is also a key factor in determining its feasibility. The system should be easy to use and accessible to people with different levels of technical expertise. Ideally, the system should require minimal user input and provide accurate results quickly and easily.

Conclusion:

Overall, a handwritten digit recognition system is feasible if it can achieve a high level of accuracy, process inputs quickly, and be easy to use. These factors depend on several variables, including the availability of data, the complexity of the model, and the processing power of the system. With advances in machine learning and computer vision, handwritten digit recognition systems have become increasingly feasible and are now used in various practical applications.

**2.4 Control flow diagram**

In more detail, the process would involve:

1. Start: The system is initialized and ready to process an input image.

2. Load image of handwritten digit: The user inputs an image of a handwritten digit, either by drawing it directly or by uploading a digital image file.

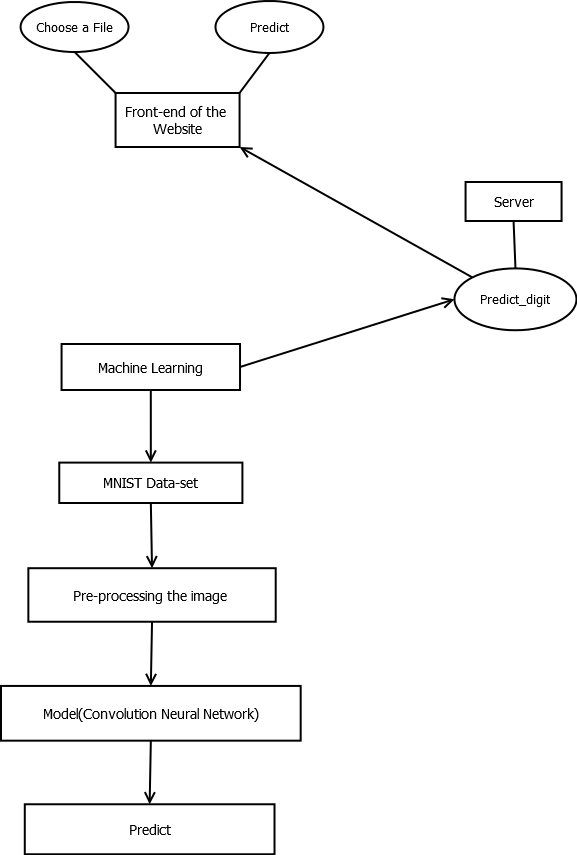
3. Preprocess image: The image is preprocessed to remove noise, resize it to a standard size, convert it to grayscale, and normalize the pixel values to a common scale.

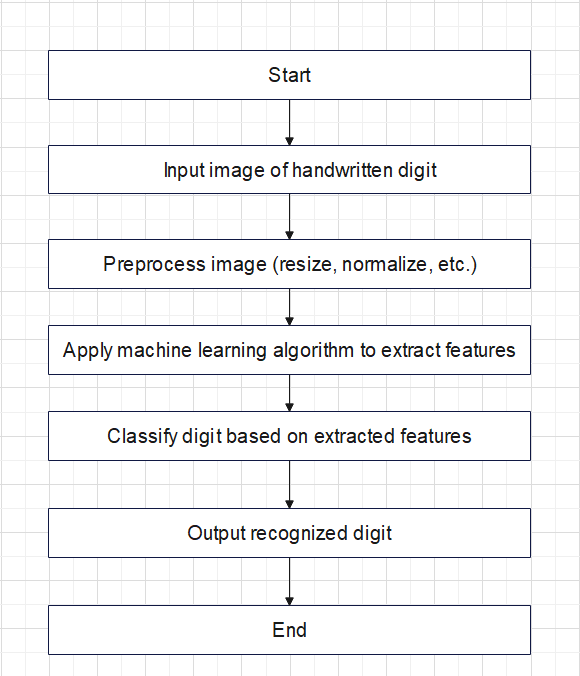
4. Extract features: The image is analyzed to extract relevant features that can help distinguish different digits. This might involve using edge detection, corner detection, or more sophisticated techniques like Histogram of Oriented Gradients (HOG).

5. Classify digit using a machine learning model: A machine learning model is trained on a large dataset of labeled digit images, and can predict which digit a new image corresponds to. This project is done using a Convolution Neural Network Model.

6. Display predicted digit: The system outputs the predicted digit to the user, either as a textual label or as an image of the digit.

1. End: The process is complete and the system is ready to process a new input image.





1. Receive input image

2. Preprocess image:

a. Convert to grayscale

b. Normalize image size

c. Apply noise reduction techniques (if necessary)

3. Extract features:

a. Divide image into smaller sections

b.Calculate features for each section (e.g., edge detection, intensity, etc.)

c. Combine features into a feature vector

4. Train model:

a. Divide dataset into training and validation sets

b. Train model on training set

c. Evaluate model on validation set

5. Test model:

a. Receive new input image

b. Preprocess image

c. Extract features

d. Use trained model to predict digit

6. Display output:

a. Show input image

b. Show predicted digit

7. Loop back to step 5 if user says prediction is incorrect

**2.5 Data flow diagram**

Handwritten digit recognition is a common application of machine learning and computer vision. The data flow diagram for handwritten digit recognition can be divided into three main components: input data, processing, and output.

1. Input Data:

The first component of the data flow diagram for handwritten digit recognition is the input data. This can be a scanned image of a handwritten digit or a live feed from a digital pen or touch screen. The input data is usually in the form of a grayscale or RGB image.

2. Processing:

The second component of the data flow diagram is processing. The processing involves several steps that transform the input data into a format that can be analyzed by a machine learning algorithm. These steps may include:

- Image pre-processing: This step involves applying filters, thresholding, and other techniques to enhance the image quality and remove noise.

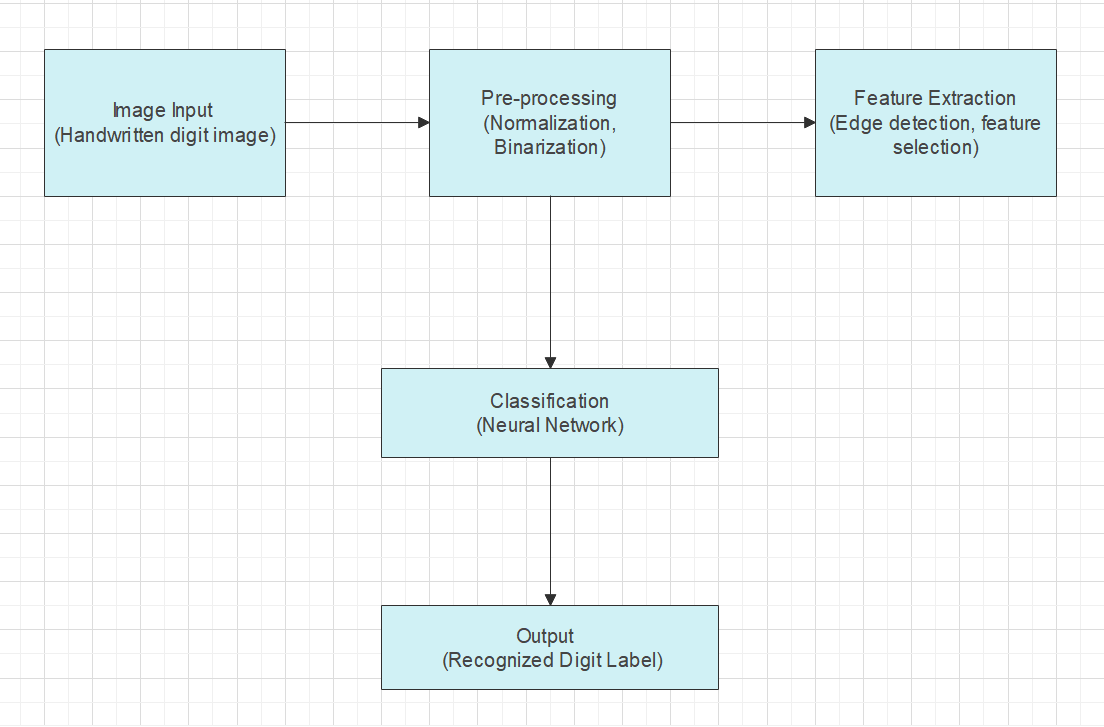
- Feature extraction: This step involves extracting relevant features from the image, such as edges, corners, and texture, that can be used by the machine learning algorithm.

- Classification: This step involves using a machine learning algorithm to classify the digit based on the extracted features. The algorithm may use techniques such as neural networks, support vector machines, or decision trees.

3. Output:

The final component of the data flow diagram is output. The output is usually the recognized digit, along with a confidence score or probability of the classification. The output can be displayed on a screen, saved to a file, or sent to another system for further processing.

Overall, the data flow diagram for handwritten digit recognition involves inputting an image of a handwritten digit, processing the image to extract relevant features, classifying the digit using a machine learning algorithm, and outputting the recognized digit and confidence score.



**3.0 Design**

## 3.1 Introduction

Handwritten digit recognition is a technology that allows computers to recognize and interpret handwritten digits. This technology has numerous applications, including postal automation, banking, and document analysis. The design of a handwritten digit recognition system involves several stages, including data collection, pre-processing, feature extraction, and classification.

In the data collection stage, a dataset of handwritten digits is collected. The dataset should include a diverse range of digits, written by different people and with different writing styles, to ensure the system can recognize digits accurately. In the pre-processing stage, the images of the handwritten digits are normalized, filtered, and enhanced to improve the quality of the data.

The feature extraction stage involves selecting the relevant features from the pre-processed data that can be used to distinguish between different digits. Several methods can be used for feature extraction, including pixel intensity, edge detection, and texture analysis. The choice of feature extraction method depends on the specific application and the dataset.

Finally, in the classification stage, the extracted features are used to classify the digits into their corresponding classes. Different classification algorithms can be used, such as k-nearest neighbors, support vector machines, and neural networks. The performance of the system is evaluated using metrics such as accuracy, precision, and recall.

Overall, the design of a handwritten digit recognition system involves a combination of image processing, machine learning, and pattern recognition techniques. The accuracy and reliability of the system depend on the quality of the data, the feature extraction method, and the classification algorithm used.

## 3.1Problem definition

Handwritten digit recognition is a problem in the field of computer vision and pattern recognition, where the goal is to develop algorithms and models that can automatically identify and classify handwritten digits from images.

In this problem, the input is an image of a handwritten digit, and the output is the corresponding digit (0-9). The images are usually grayscale and of a fixed size, and they may have some noise or variation in the handwriting style.

The handwritten digit recognition problem is commonly used as a benchmark task for evaluating the performance of machine learning algorithms, especially in the context of deep learning and neural networks. It has many applications, including digit recognition in postal and banking systems, automatic recognition of handwritten forms, and even medical diagnosis.

## 3.2 Solution specification

### Handwritten digit recognition refers to the task of identifying digits that are written by humans in natural handwriting. This task is a well-known problem in the field of machine learning and computer vision. The following is a specification for a handwritten digit recognition solution:

### 1. Data collection and preparation:

### - Collect a dataset of handwritten digits. The MNIST dataset is a widely used benchmark dataset for this task, but other datasets such as SVHN and EMNIST can also be used.

### - Preprocess the data by normalizing the images and converting them to grayscale or binary format.

### - Split the dataset into training, validation, and testing sets.

### 2. Model selection and training:

### - Select a suitable model architecture for the task, such as a convolutional neural network (CNN).

### - Train the model on the training set, using an appropriate optimization algorithm and loss function.

### - Use the validation set to tune the hyperparameters of the model, such as the learning rate and regularization strength.

### 3. Evaluation:

### - Evaluate the trained model on the testing set to obtain a final accuracy score.

### - Calculate other metrics such as precision, recall, and F1 score.

### - Analyze the results and identify any areas for improvement.

### 4. Deployment:

### - Deploy the trained model as an API or web service.

### - Provide a user interface for users to input handwritten digits and receive predictions from the model.

### - Monitor the performance of the deployed model and make updates as necessary.

### Overall, a handwritten digit recognition solution involves data collection and preparation, model selection and training, evaluation, and deployment. By following this specification, a high-performing and user-friendly solution can be developed for this task.

### Screen Design

##### This is the design of the developed website.

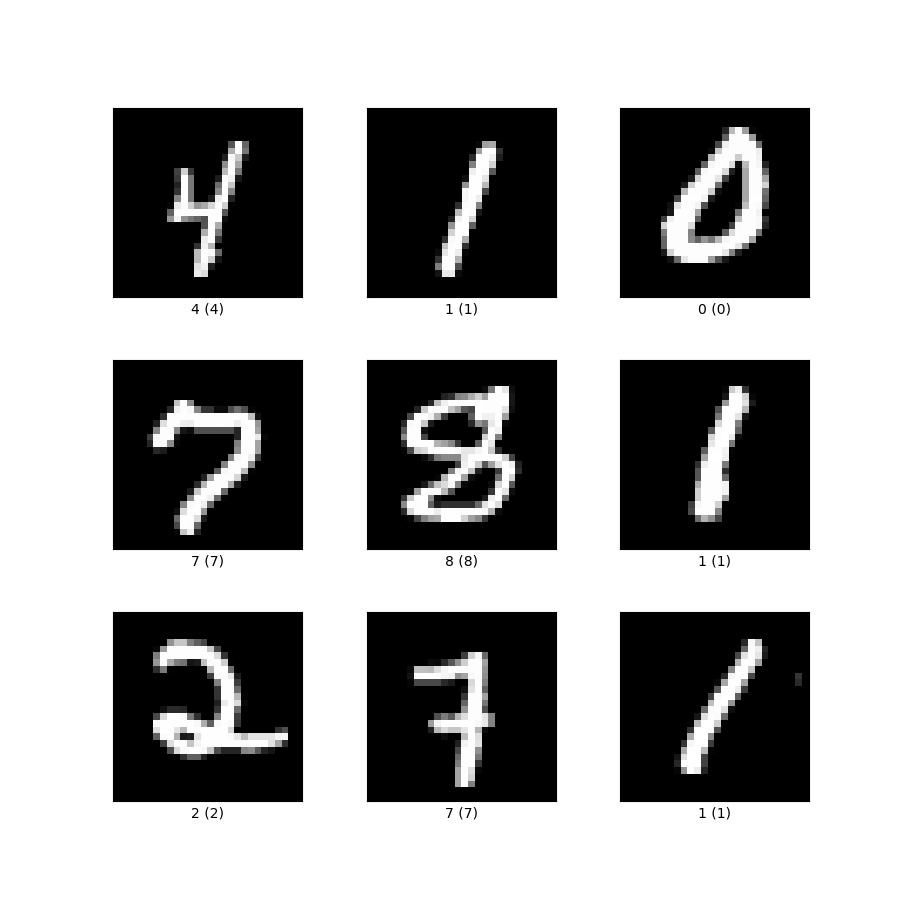
# 

# ss_hwd Figure 1

# 4.0 Develop

I have trained a convolution neural network model which can predict digits when image is given as an input. Based on this deep learning model, I have also developed a website which takes input from the user and predict the digit. The model is fetched by the website using a backend flask server, which I have created using python.

## 4.1 Implementation

I have created a Convolution Neural Network Model for accurate prediction of hand-written digits inputted by the user. I have used the MNIST data-set of Handwritten digits for training the model. The MNIST data-set has all the images pre-processed and has 70,000 examples of which 60,000 are of training set and 10,000 are of test set. The images are re-sized into 28 x 28 grayscale images and noises are also removed. The model was trained and it gave a accuracy of 99.86% on the training set and 99.13% on the test set. The model was then saved in a ‘mnist\_model.h5’ file. The complete model was then connected through a flask server in the back-end for drawing predictions in the front-end. The image files inputted in the website is pre-processed in the ‘server.py’ file and is then inputted in the model. The response is then fetched by the java-script in the back-end. The response is then displayed in the front-end.

## 

After coding the model, I have coded the server.py and util.py files. This two files creates the flask server from which data is sent to the front-end. In the server file, I have created a link for fetching the model and connecting it to the front-end

As the flask server is created, I coded the complete website. The website contains a front-end model coded using html and css, and there is a back-end model coded using javascript. In the front-end of the website, I have given a choose file option for the user to input a image file. If the user happen to input a non-image file it will show a error. Once the user has inputted the image accordingly, they can press the predict button and it will predict the digit. The back-end of the model created using javascript fetches the CNN model.

We first have to ensure that the process carried out in the last step corresponds to the task which was originally defined, if this is not the case than all the previous steps have to be retraced. Once we are satisfied that the task has been performed satisfactorily we may then go about looking for ways of improving the previous steps. And so this cycle may be repeated several times before we are finally satisfied. In this context this cycle is referred to as the software development cycle.

## 4.2 Testing for errors

Even though the model has a error rate less than 1%, it has a weak relation with unprocessed data. To minimize the prediction error with user inputted data, I have also done some pre-processing on the image file inputted. I have calculated the loss function using categorical cross-entropy formula.

The categorical cross-entropy is a loss function used in machine learning and deep learning for multi-class classification problems. It measures the dissimilarity between the predicted probability distribution and the true probability distribution of the target class.

The formula for categorical cross-entropy is as follows:

```

CE(y, ŷ) = - Σ yᵢ \* log(ŷᵢ)

```

where:

- `y` is the true probability distribution of the target class, represented as a one-hot encoded vector.

- `ŷ` is the predicted probability distribution of the target class, represented as a vector of probabilities.

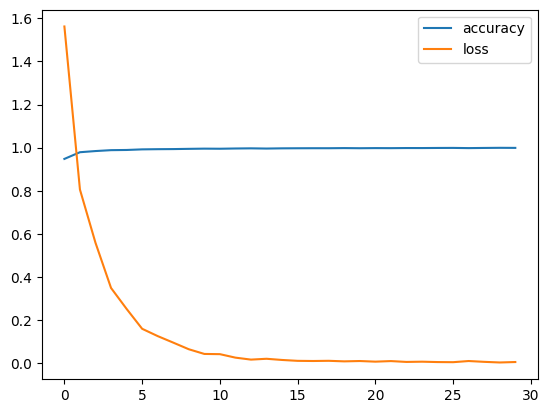
- `yᵢ` is the i-th element of `y`, which represents the probability of the i-th class being the true class.

- `ŷᵢ` is the i-th element of `ŷ`, which represents the predicted probability of the i-th class being the true class.

- The sum is taken over all classes.

The categorical cross-entropy penalizes the model more severely for predicting low probabilities for the true class, which helps the model to converge to a better solution. The goal of training a model with categorical cross-entropy is to minimize the loss function by adjusting the model's parameters using gradient descent or other optimization techniques.

The graph of the accuracy and loss function :

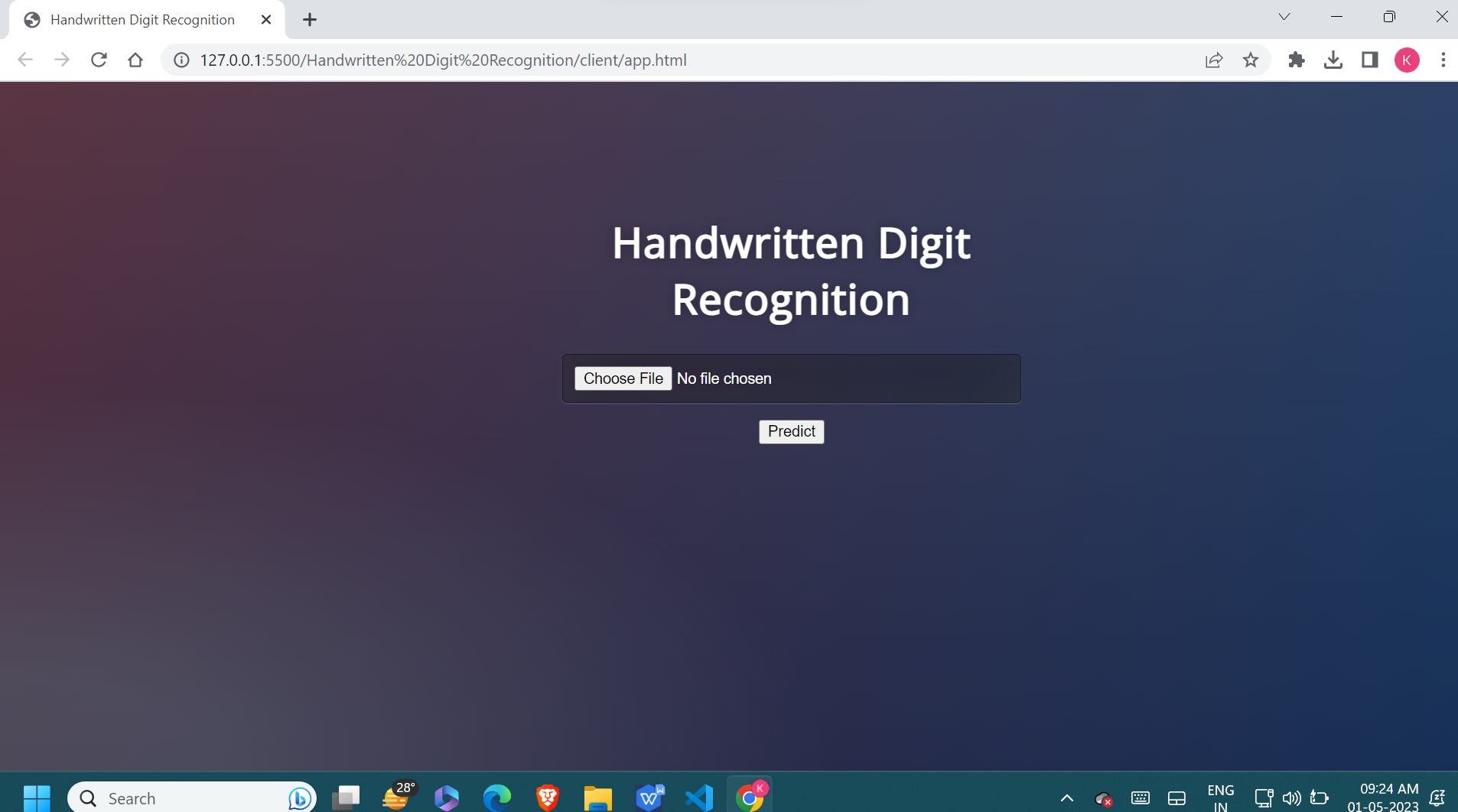


**4.3 User Interface design**

**Deployment Website:**

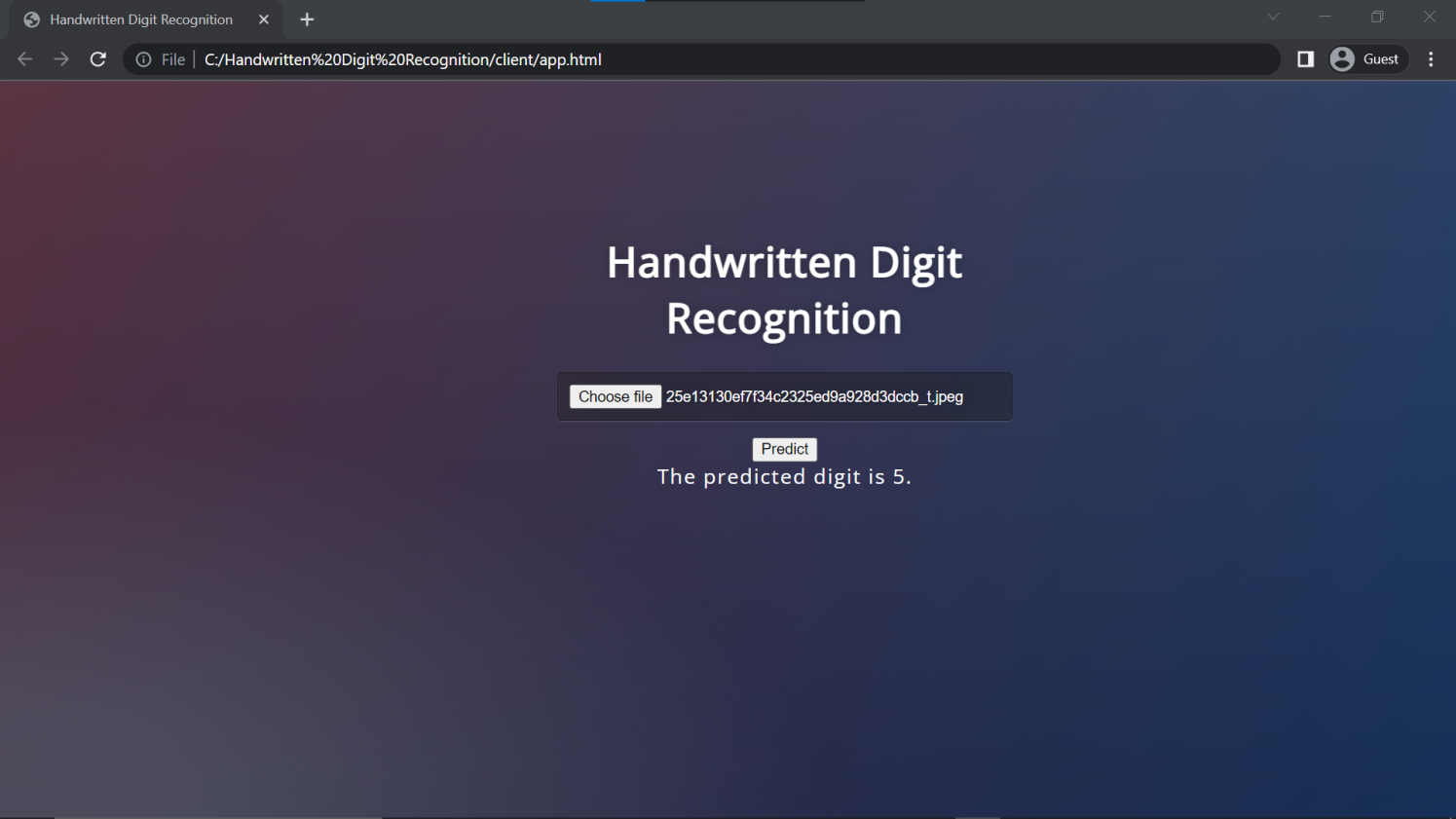
I have developed a simple user interactive web-page which takes the input from the user. User has to upload the image file in .png, .jpg or .jpeg file format. And then there is Predict button for predicting the the digit.

The Model is deployed through Python Web server and using of Postman API Flask in collaboration with HTML and CSS.

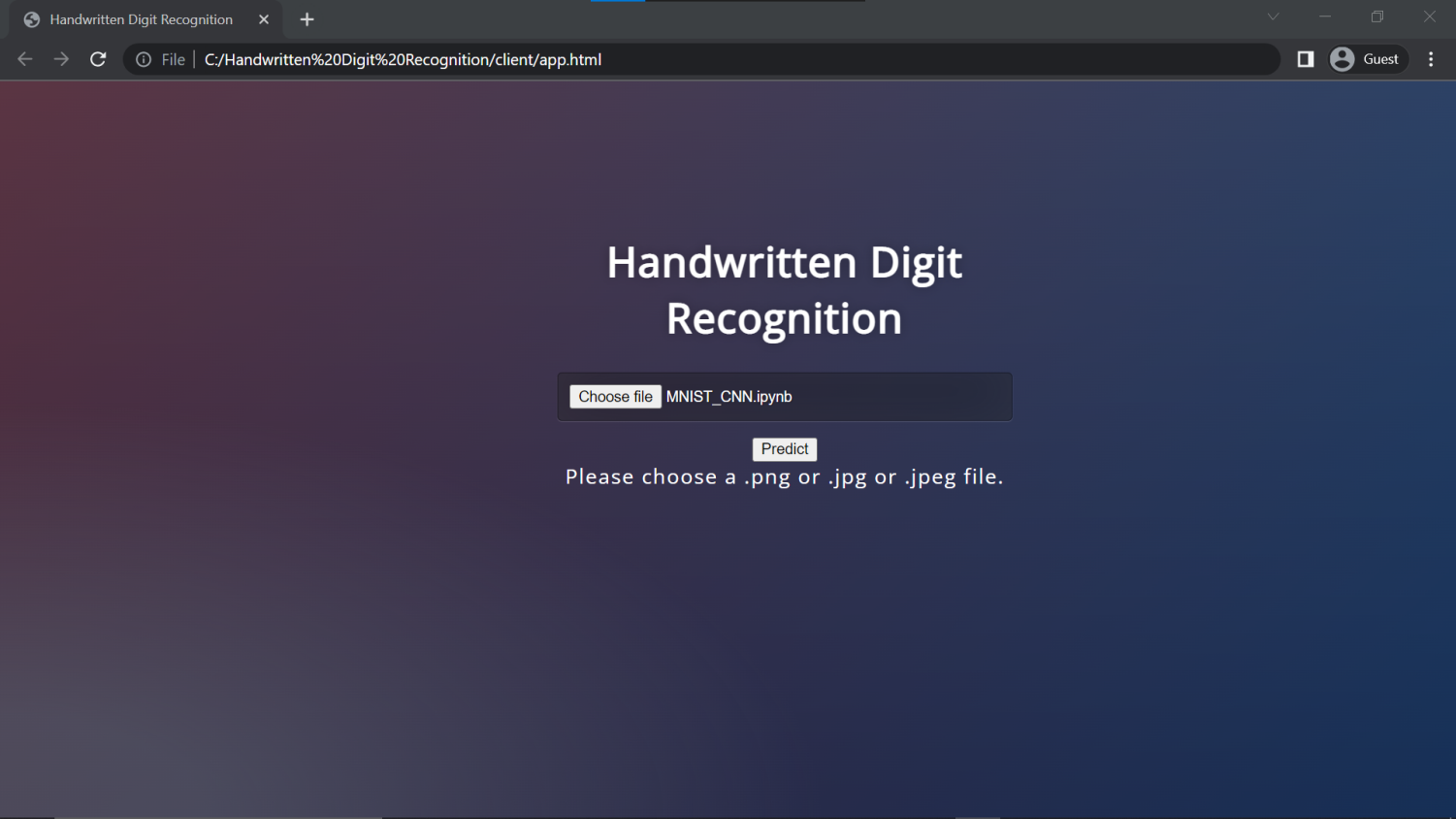


**Drawing Predictions**

The website has a choose file option. The user has to input a image of a digit and the file will be send to the back-end for inputting it to the model. The model will give a prediction and the response will be fetched and displayed in the front-end. It will be as follows :



Another thing is that the user has to input a image file or else it will show a error as follows :



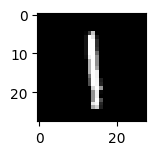
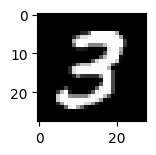
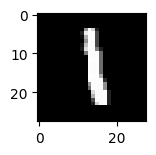
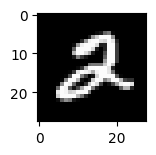
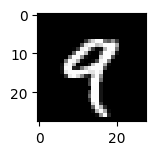
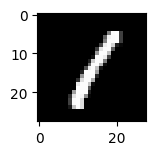
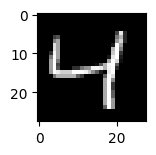
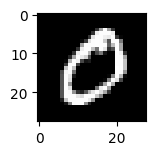
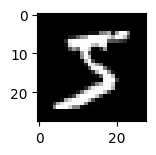
**4.4 Admin Interface design**

**Data-set**

Here we have imported the MNIST data-set from the tensorflow library. It is available directly in tensorflow package because it is a common data-set used in deep learning world. The thing about MNIST data-set is that the images are already preprocessed. All the noises in the image are removed.

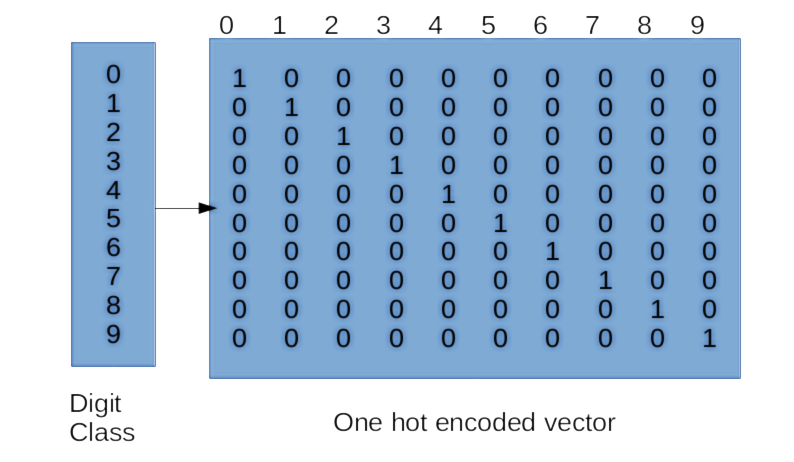
**Data-set Overview**

The images used are in gray-scale and are resized in 28 x 28 pixels. Other than that noise is also removed from the image. The images look like as follows :



**Data Labelling**

The data is classified into 10 digits that is 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9. I have converted the Y labels in a one-hot vector of shape (1 , 10). The Y vectors of all the images are merged in a single matrix and is none as a one-hot matrix



**Data Division**

The MNIST data-set has 70,000 , 28 x 28 pre-processed images of the digits. I have divided the data-set into two parts mainly training set and test set. The training set contains 60,000 images and the test set contains 10,000 images.

**MODELS USED**

**Convolution Neural Network Model**

I have trained a machine learning model which can predict digits according to the image given as input. I have trained a Convolution Neural Network model for the predictions. The model gave us a accuracy of 99.86% on the training set and 99.13% accuracy on the test set.

* Convolution Neural Networks are a sub division of Deep Neural Networks , specifically used for image related problems.
* As we know , Nowadays most of the image data we get are in higher resolutions , Hence , we have much more pixel information to process at a time.
* Let’s say we have a 1000 X 1000 res image, with traditional neural networks we have to densely connect all the pixel values of the given image with the neurons of the next layer. If the next layer have 16 neurons , then the no. of trainable parameters will add up to 1,60,00,001 including 1 bias parameter.
* Even though we have advanced hardwares for fast training of such large amount of parameters , it is not a efficient model.
* CNNs are the most efficient way for image related problems , as here , no. of trainable parameters doesn’t depend upon the image size.

Convolutional Neural Networks (CNNs) are a type of deep learning neural network used for image processing tasks. They are inspired by the biological visual cortex in animals and are designed to recognize patterns in images.

CNNs are made up of multiple layers that process images in a hierarchical manner. The first layer is the input layer, which takes in raw image data. The next few layers are convolutional layers, which apply a set of learnable filters to the input image. These filters effectively scan the image for certain features such as edges, corners, or textures. The output of each filter is a feature map that represents the presence of that feature in the image.

Each filter in a convolutional layer is small in size and slides over the entire input image. The filter computes a dot product between the filter values and the corresponding pixel values in the input image. The output of this computation is a single value in the feature map. Each filter can produce a different feature map, capturing different features in the input image.

After the convolutional layers, a pooling layer is often added to downsample the feature maps. This reduces the size of the feature maps and the number of parameters in the network, making it more computationally efficient. The most common pooling operation is max pooling, which takes the maximum value within a small region of the feature map. This operation retains the most important information and discards the less important information.

The final layers of a CNN are typically fully connected layers, which take the flattened feature maps and use them to classify the image into one of several classes. The output layer usually consists of a softmax function, which produces a probability distribution over the different classes.

Training a CNN involves iteratively adjusting the weights of the filters and fully connected layers to minimize the difference between the predicted output and the ground truth label. This is done using backpropagation, which calculates the gradient of the loss function with respect to the weights and biases of the network.

One advantage of CNNs is their ability to learn hierarchical representations of features, where lower layers detect basic features such as edges and corners, and higher layers detect more complex features such as facial features or object parts. This makes them highly effective for image classification tasks and has led to state-of-the-art performance on many benchmark datasets.

In addition to image classification, CNNs can also be used for object detection, semantic segmentation, and image generation tasks. For object detection, a region proposal network (RPN) can be added to the network to identify potential regions of interest in an image, and then a classifier can be applied to each of these regions. For semantic segmentation, each pixel in the input image is assigned a class label, enabling the network to identify objects and their boundaries in the image. For image generation, generative adversarial networks (GANs) can be used to generate new images that are similar to a given dataset.

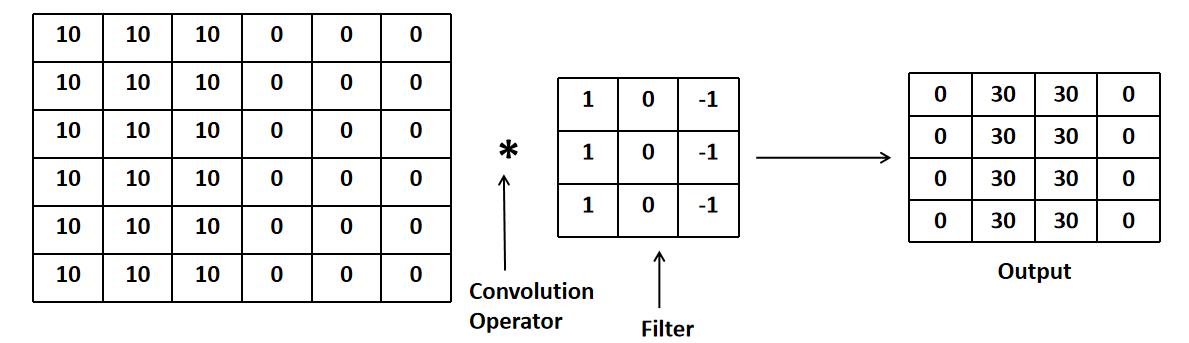
However, CNNs are computationally expensive and require a large amount of training data and computational resources. There is ongoing research on improving the efficiency and interpretability of CNNs, as well as developing new architectures that can better handle more complex tasks.

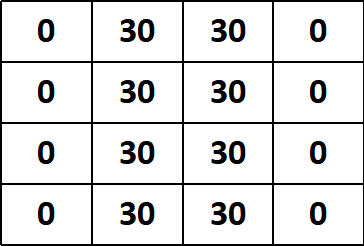
One such architecture is the ResNet (Residual Network), which uses skip connections to enable training of deeper networks. These connections allow the network to learn residual functions, which can be used to improve the accuracy of the network. Another architecture is the Inception Network, which uses a combination of 1x1, 3x3, and 5x5 convolutions to capture features at different scales.

Overall, CNNs have revolutionized the field of computer vision and have enabled the development of many applications such as autonomous vehicles, facial recognition, and medical image analysis. With ongoing research and development, CNNs are expected to continue to advance and be applied to even more challenging image processing

**Convolution Operator(\*)**

* Let’s consider a gray-scale image of dimension 6 X 6 and pixel values as follows.
* The Convolution Operator is denoted as “\*” and shouldn’t be confused with Python multiplication operator.
* In this operation , the filter is placed element-wise at the start of the image . We calculate the element-wise operation and add all the elements into one pixel output. We shift the filter one side at a time and calculate the pixels until the complete image is covered. The output size is (n - f + 1) X (n - f + 1).





**Learnable parameters**

**Output**

* As I have shown in the last slide a convolution operation between a given image and filter , In a CNN , the filter values are trained throughout the dataset for the best value.
* As the filter shape is a hyper-parameter and has no relation to the actual input shape , we comparatively have less parameters to train than a dense high dimensional neural network.

**Padding**

* Padding is a process used to eliminate the loss of information in the training process.
* In a convolution operation , the centre pixels of a image has more contribution to the result compared to the edge pixels , and we certainly lose information in the training process due to this problem.
* In padding , we add layers of pixels on all the sides of the image , and run convolution operation on the padded image. This allows us to prevent loss of information as the pixels are pushed towards the centre.
* A padded image has a shape of (n + 2p - f + 1) X (n + 2p - f + 1).
* Types of Convolutions :-
* “Valid” Convolutions :- No padding.
* “Same” Convolutions :- Pad so that output size is same as input size

**Strided Convolutions**

* Strided convolution is normal convolution operation with a added hyper-parameter strides.
* Strides are the no. of steps to be taken while doing the convolution operation.
* The output size of the operation is :-

(((n + 2p - f) / s) + 1) X (((n + 2p - f) / s) + 1)

**Convolutions over Volume**

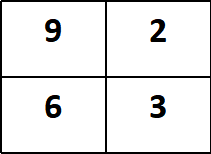
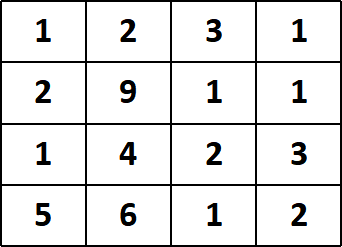
* Most of the time , the input image of the CNN will be RGB , hence having 3 channels.
* The Convolution operation is done over all the channels present and then they are all added.
* The number of filters applied on a layer , determine the no. of channels of the output image. If the no. of filters applied on a layer is nc[l] then the no. of channels of the output is nc[l] .

**Types of Layer in CNN**

* There are three types of layers which build up a convolutional neural network :-
* Convolution Layer(Conv.)
* Pooling Layer(POL)
* Fully Connected Dense Layer(FC)

**Pooling**

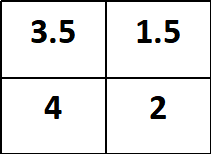
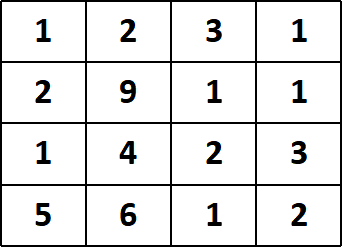
Max Pooling :- Lets say we have a 4 X 4 dimensional image. In this process we just take the max value of the chunk of area we select.



**f = 2**

**s = 2**

Average pooling :- Average pooling does the same task as max pooling except now it takes average of all the chunk of pixels.



**f = 2**

**s = 2**

**Fully connected Dense Layer**

* This layer share the same properties as a traditional neural network.
* The output values of the previous layer gets flattened into a single vector in this layer.
* This flattened vector is connected with the next layer in the way same as traditional neural networks.
* Then the last layer output is operated , usually softnmax classifier.

**Backward Propagation in Convolution Neural Network**

* **Convolution Layer Backward Pass** : Backward propagation in convolutional layers involves calculating the gradients of the loss function with respect to the weights and biases of the layer, and then propagating these gradients back through the layer to update the parameters using an optimization algorithm such as stochastic gradient descent (SGD).

Here are the numerical formulas for computing the gradients of the loss function with respect to the weights and biases of a convolutional layer:

- Gradient of the loss with respect to the weights:

```

**dL/dW[k,l,c] = Σ\_i Σ\_j Σ\_p Δ[k+i, l+j, p] \* X[i,j,c]**

```

where:

- `dL/dW[k,l,c]` is the gradient of the loss with respect to the weight at position `(k,l)` and channel `c`.

- `Δ` is the error tensor of the layer's output. The error tensor is the element-wise product of the next layer's error tensor and the derivative of the activation function of the current layer's output.

- `X` is the input tensor to the layer.

- The outer summations are taken over the positions `(i,j)` in the input tensor that overlap with the weight at `(k,l)`.

- Gradient of the loss with respect to the biases:

```

**dL/db[c] = Σ\_i Σ\_j Σ\_p Δ[i,j,p]**

```

where:

- `dL/db[c]` is the gradient of the loss with respect to the bias for channel `c`.

- `Δ` is the error tensor of the layer's output.

- The outer summations are taken over all positions `(i,j)` in the input tensor.

In practice, these formulas are implemented using vectorized operations to improve efficiency. The gradients are then used to update the weights and biases of the convolutional layer during the optimization process.

* **Pooling Layers Backward Pass**

Backward propagation in pooling layers involves computing the gradients of the loss function with respect to the input tensor of the layer. The pooling operation reduces the spatial dimensionality of the input tensor by down-sampling it, which makes it challenging to compute the gradients analytically. Therefore, the most common approach is to perform backward propagation using a technique called "max unpooling" or "average unpooling," which involves storing the locations of the maximum or average pooling operations during the forward pass and using them to reconstruct the input tensor during the backward pass.

* **Max Pooling Backward Pass :-** Here are the numerical formulas for computing the gradients of the loss function with respect to the input tensor of a max pooling layer:

- Gradient of the loss with respect to the input tensor:

```

**Δ' = zeros\_like(X)** # create a tensor of the same shape as X, filled with zeros

**for i in range(output\_shape[0]):**

**for j in range(output\_shape[1]):**

**for k in range(output\_shape[2]):**

**for l in range(output\_shape[3]):**

**window = X[batch, i\*stride:i\*stride+pool\_size, j\*stride:j\*stride+pool\_size, k]**

**max\_idx = np.argmax(window)**

**max\_pos = np.unravel\_index(max\_idx, window.shape)**

**Δ'[batch, i\*stride+max\_pos[0], j\*stride+max\_pos[1], k] = Δ[batch, i, j, k, l]**

```

where:

- `Δ` is the error tensor of the layer's output.

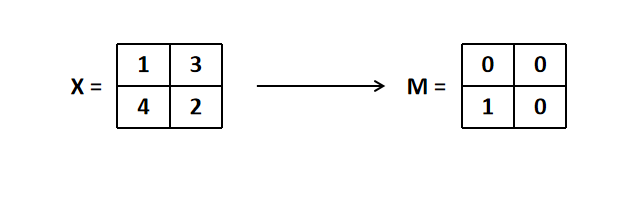
- `X` is the input tensor to the layer.

- `output\_shape` is the shape of the layer's output tensor.

- `pool\_size` is the size of the pooling window.

- `stride` is the stride of the pooling window.

- The nested loops iterate over the positions of the pooling windows in the output tensor, and for each window, find the location of the maximum value in the corresponding window of the input tensor and set the corresponding element of `Δ'` to the corresponding element of `Δ`.



A mask is created from the max pool layer

* **Average Pooling Backward Pass :** The numerical formulas for computing the gradients of the loss function with respect to the input tensor of an average pooling layer are similar, but the values of the error tensor `Δ` are distributed uniformly across the pooling window during the forward pass. Therefore, the gradient of the loss with respect to each element of the input tensor is simply the average of the corresponding values in the error tensor:

- Gradient of the loss with respect to the input tensor:

```

**Δ' = zeros\_like(X)** # create a tensor of the same shape as X, filled with zeros

**for i in range(output\_shape[0]):**

**for j in range(output\_shape[1]):**

**for k in range(output\_shape[2]):**

**for l in range(output\_shape[3]):**

**window = X[batch, i\*stride:i\*stride+pool\_size, j\*stride:j\*stride+pool\_size, k]**

**Δ'[batch, i\*stride:i\*stride+pool\_size, j\*stride:j\*stride+pool\_size, k] = Δ[batch, i, j, k, l] / pool\_size\*\*2**

```

where:

- `Δ` is the error tensor of the layer's output.

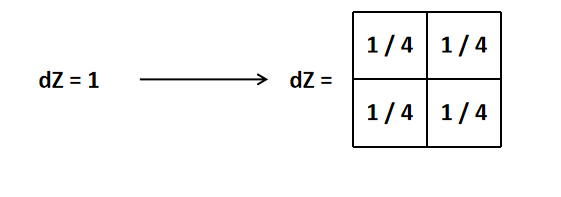
- `X` is the input tensor to the layer.

- `output\_shape` is the shape of the layer's output tensor.

- `pool\_size` is the size of the pooling window.

- `stride` is the stride of the pooling window.

- The nested loops iterate over the positions of the pooling windows in the output tensor, and for each window, set the corresponding



The value gets distributed over the layer-dimensions

**Fully Connected Layer Backward Pass**

Backward propagation in a fully connected dense layer involves computing the gradients of the loss function with respect to the layer's weights and biases, as well as the gradients of the loss with respect to the input tensor of the layer, which are then passed back to the previous layer. Here are the numerical formulas for computing these gradients:

- Gradient of the loss with respect to the weights:

```

**dW = np.dot(X.T, delta) / batch\_size**

```

where:

- `X` is the input tensor to the layer, with shape `(batch\_size, input\_size)`.

- `delta` is the error tensor of the layer's output, with shape `(batch\_size, output\_size)`.

- `batch\_size` is the number of samples in the batch.

- The formula computes the dot product of the transpose of `X` with `delta`, and divides the result by the batch size to obtain the average gradient across the batch.

- Gradient of the loss with respect to the biases:

```

**db = np.mean(delta, axis=0)**

```

where:

- `delta` is the error tensor of the layer's output, with shape `(batch\_size, output\_size)`.

- The formula computes the mean of `delta` along the batch dimension (axis 0) to obtain the average gradient across the batch.

- Gradient of the loss with respect to the input tensor:

```

**delta\_ = np.dot(delta, W.T)**

```

where:

- `delta` is the error tensor of the layer's output, with shape `(batch\_size, output\_size)`.

- `W` is the weight tensor of the layer, with shape `(input\_size, output\_size)`.

- The formula computes the dot product of `delta` with the transpose of `W` to obtain the gradient of the loss with respect to the input tensor of the layer. This gradient is then passed back to the previous layer in the neural network.

**Challenges Faced**

Developing a handwritten digit recognition system can be a complex and challenging task due to several factors. Some of the challenges faced during the development of a home price detection system include:

1. **Data quality and quantity**: A handwritten digit recognition system relies heavily on the quality and quantity of the data used to train the machine learning model. The dataset used should be comprehensive, diverse, and accurate, and must be pre-processed to remove any missing or incorrect data. Obtaining a suitable dataset can be challenging as it may require considerable resources and access to large and reliable sources of data.

2. **Overfitting and underfitting**: Overfitting occurs when the model is too complex and fits the training data too closely, resulting in poor performance on new data. Underfitting, on the other hand, occurs when the model is too simple and does not capture the complexity of the data, resulting in poor performance on both training and new data. Finding the right balance between overfitting and underfitting can be a challenge.

3. **Model complexity**: The complexity of the machine learning model used for handwritten digit recognition can also pose a challenge. The model must be able to capture the complexity of the data while being interpretable and easy to understand. Choosing the right model architecture and hyperparameters can be a challenging task, requiring significant expertise and knowledge.

4. **Performance and scalability**: A handwritten digit recognition system must be able to handle large datasets and provide accurate predictions in real-time. Ensuring that the system is performant and scalable can be a challenge, requiring careful optimization of the system architecture, algorithms, and data management.

5. **Interpretability**: Interpreting the results of a machine learning model used in a handwritten digit recognition system can be challenging. The system must be able to explain the reasoning behind its predictions and provide transparent and interpretable results. Ensuring that the model is interpretable and understandable can be a significant challenge.

Overall, developing a handwritten digit recognition system can be a challenging task that requires significant expertise, resources, and domain knowledge. Overcoming these challenges requires careful consideration of the system's requirements and constraints, as well as a deep understanding of the machine learning algorithms and techniques used.

**Model Designed**

I have designed a medium complex Convolution Neural Network Model for the task. It consists of 2 layers with each layer consisting of 2 Convolution Layers, 1 Max-pool layer and other than that the end layer of the model is densely connected with the soft-max layer of 10 neurons. Every layer is batch-normalized. The model architecture is as follows :

Layer 1

7

28 x 28 x 64

28 x 28 x 64

28 x 28 x 32

28 x 28 x 1

**f = 32**

**s = 1**

**p = ‘same’**

Image Input

**f = 64**

**s = 1**

**p = ‘same’**

Conv1

Conv2

14 x 14 x 6

Maxpool1

**f = 64**

**s = 1**

**p = ‘same’**

Layer 2

28 x 28 x 96

28 x 28 x 128

28 x 28 x 128

**f = 32**

**s = 1**

**p = ‘same’**

10 x 1

5 x 5 x 16

Maxpool2

Conv4

**f = 128**

**s = 1**

**p = ‘same’**

**f = 96**

**s = 1**

**p = ‘same’**

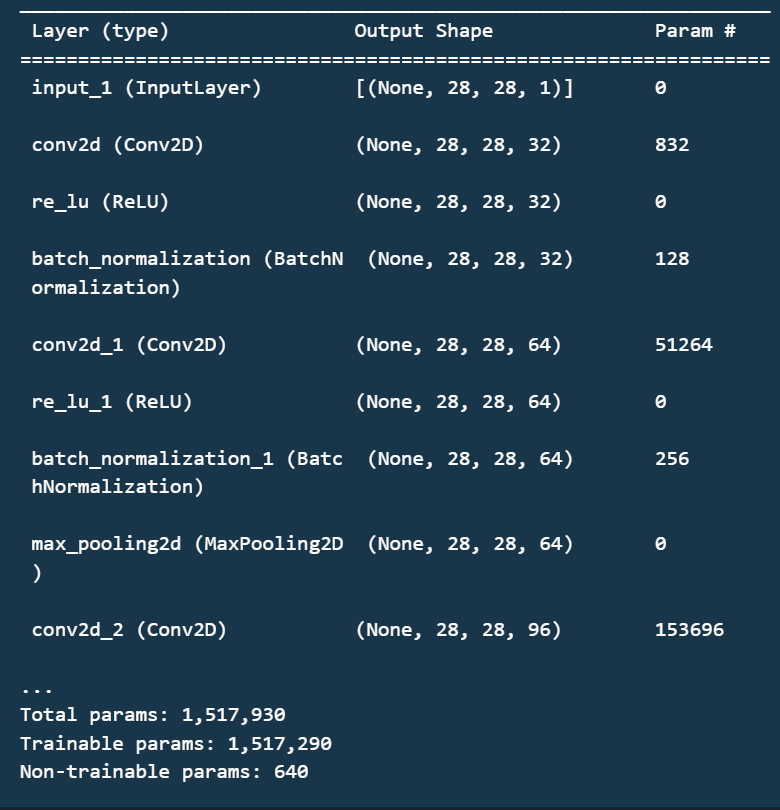
Conv3

100352 x 1

Softmax

FC1

Overview of the layers and the no. of parameters used :



**Model Parameters**

The algorithm used to train the model is adam-optimization algorithm. The batch size used is 64 and the number of epochs on which the model is trained is 30.

The parameters of a handwritten digit recognition model depend on the specific architecture of the model, but generally include:

1. Convolutional Layers: These layers have a set of learnable filters (also called kernels) that convolve over the input image to extract features. The parameters of each filter include the filter size, the number of filters, the stride, and the padding. These parameters control the size of the output feature maps and the receptive field of the filters. The kernel size, I used is (5 , 5). The number of filters used in two conv2d in 1st layer are 32 and 64 respectively. The number of filters used in two conv2d in 2nd layer are 96 and 128 respectively. The stride is set to 1 and the padding is ‘same’.

2. Pooling Layers: These layers downsample the feature maps obtained from the convolutional layers. The parameters of the pooling layer include the pool size and the stride. We have done max-pooling on each layer. The pool size set was (2 , 2) for both and the stride was set to 1. I used ‘same’ padding.

3. Activation Functions: The activation function is applied to the output of each neuron in the network. Popular choices include ReLU, sigmoid, and tanh. The parameters of the activation function depend on the specific function used. ReLU function was used for activation and the prediction was drawn on a softmax classifier.

4. Dropout: Dropout is a regularization technique that randomly sets some neurons to zero during training to prevent overfitting. The dropout rate is a hyperparameter that determines the probability of setting a neuron to zero. Dropout was not used in our model but it can be used afterwards.

5. Fully Connected Layers: These layers connect all neurons in the previous layer to all neurons in the current layer. The parameters of the fully connected layer include the number of neurons and the weight and bias parameters for each neuron. The end layer which is used for classification was densely connected with the flattened layer. The flattened layer had a shape of (100352 , 1).

6. Loss Function: The loss function measures the difference between the predicted and actual labels. Popular choices include categorical cross-entropy for classification problems and mean squared error for regression problems. As I wanted to classify the images into digits , I used categorical cross-entopy loss function for the same.

7. Optimization Algorithm: The optimization algorithm updates the model parameters during training to minimize the loss function. Popular choices include stochastic gradient descent (SGD), Adam, and RMSprop. The parameters of the optimization algorithm include the learning rate, the momentum, and the decay rate. Adam Optimization Algorithm was used for best convergence of the loss function

8. Batch Size: The batch size is the number of samples processed by the model in one forward/backward pass. The batch size is a hyper-parameter that affects the training time and memory usage. Batch size was set to 64 for this model while training.

9. Number of Epochs: The number of epochs is the number of times the model iterates over the entire training dataset. The number of epochs is a hyperparameter that controls the duration of the training process and the generalization of the model. My model was trained in 30 epochs.

These parameters were tuned through a combination of experimentation and hyperparameter optimization to achieve the best performance.

1. **SYSTEM SECURITY**

Handwritten digit recognition systems are vulnerable to various security threats, including adversarial attacks, data poisoning attacks, and model extraction attacks. Here are some ways to enhance the security of a handwritten digit recognition system:

1. Adversarial training: Adversarial training involves adding perturbations to the training data to improve the system's robustness against adversarial attacks. This can be done by generating adversarial examples using techniques such as FGSM or PGD and including them in the training dataset.

2. Input validation: Handwritten digit recognition systems should include input validation checks to prevent attacks such as SQL injection or buffer overflow. This can be done by checking the input format and size to ensure that it conforms to the expected input specifications.

3. Model encryption: Model encryption can be used to protect the intellectual property of the model and prevent model extraction attacks. This can be done using techniques such as homomorphic encryption, differential privacy, or secure multi-party computation.

4. Data authentication: Data authentication techniques such as digital signatures can be used to ensure that the data used to train the model has not been tampered with or modified.

5. Model monitoring: Regular monitoring of the model's performance and behavior can help detect any anomalies or attacks. This can be done by analyzing the model's output distribution or by comparing the model's predictions with ground-truth labels.

6. Access control: Access to the model and its data should be restricted to authorized users only. This can be done by implementing proper authentication and authorization mechanisms and using secure communication protocols.

7. Regular updates: Regular updates to the system's software and security mechanisms can help keep it secure against newly discovered vulnerabilities and threats.

Overall, enhancing the security of a handwritten digit recognition system involves a combination of technical, organizational, and procedural measures.

1. **CHALLENGES FACED**

Handwritten digit recognition is a challenging task because of the following reasons:

1. Variations in handwriting: Handwriting can vary widely from person to person, making it difficult to create a generalized model that can recognize all types of handwriting.

2. Variations in writing style: Even within the same person's handwriting, there can be variations in the writing style due to factors such as the writing instrument used, writing speed, and writing angle.

3. Overlapping digits: In some cases, digits may overlap or touch each other, making it difficult for the model to differentiate between them.

4. Noise in the image: Handwritten digit images may contain noise, such as smudges, blots, or other artifacts, which can make it difficult for the model to accurately recognize the digits.

5. Skewed or distorted images: Handwritten digit images may be skewed or distorted, which can affect the model's ability to recognize the digits.

6. Limited amount of training data: Handwritten digit recognition models require a large amount of training data to achieve high accuracy, but obtaining such data can be challenging due to the time and effort required to collect and label the data.

1. Computational complexity: Training a high-accuracy handwritten digit recognition model can be computationally expensive, requiring significant computational resources and time.
2. **FUTURE SCOPE**

Handwritten digit recognition technology has come a long way over the years, and there are still several avenues for future development and improvement. Some potential future directions for this technology include:

1. Improved Accuracy: While current systems have achieved high accuracy rates, there is still room for improvement. Researchers can continue to develop new algorithms and improve existing ones to achieve higher levels of accuracy.

2. Enhanced Real-Time Performance: Real-time performance is an important factor for many applications, such as recognizing digits written on a whiteboard during a lecture or identifying handwritten addresses on mail envelopes. Researchers can focus on improving the speed of the system to meet these demands.

3. Handling Variability: Handwritten digits vary greatly in style, size, and shape. Therefore, digit recognition systems should be able to handle these variations effectively. Future research can explore ways to make the system more robust to variability.

1. Multilingual Recognition: Digit recognition technology can be extended to recognize handwritten digits in different languages. This could be useful for applications such as recognizing postal codes or phone numbers on documents from different parts of the world.

5. Integration with Other Technologies: Handwritten digit recognition technology can be integrated with other technologies such as speech recognition, natural language processing, and machine translation to create more sophisticated systems that can recognize and interpret complex information.

6. Application in Various Fields: Handwritten digit recognition technology can be applied in various fields such as healthcare, education, and finance to automate various processes and improve efficiency. For example, digit recognition technology can be used to automatically read and interpret medical prescriptions or to process checks and bank statements.

Overall, the future of handwritten digit recognition technology is bright, with many exciting opportunities for development and growth.

1. **CONCLUSIONS**

Handwritten digit recognition using Convolutional Neural Networks (CNNs) has become one of the most popular computer vision tasks in recent years. CNNs have demonstrated excellent performance in identifying and classifying handwritten digits with high accuracy rates. Here are some conclusions that can be drawn from the use of CNNs for handwritten digit recognition:

1. CNNs are highly effective for handwritten digit recognition: CNNs have shown exceptional performance in recognizing and classifying handwritten digits, achieving accuracy rates above 99% on various datasets.

2. CNNs can handle large-scale datasets: CNNs can handle large-scale datasets with millions of training examples, enabling them to learn complex features from the data and achieve high accuracy rates.

3. Preprocessing can improve accuracy: Preprocessing techniques such as normalization and image augmentation can improve the accuracy of the CNN model by reducing the variability in the input images.

4. Regularization techniques can prevent overfitting: Regularization techniques such as dropout and weight decay can prevent overfitting and improve the generalization performance of the CNN model.

5. Hyperparameter tuning is essential: Proper selection of hyperparameters such as learning rate, batch size, and number of filters is essential for achieving the best performance of the CNN model.

6. Transfer learning can speed up training: Transfer learning can speed up the training of CNNs by using pre-trained models on large datasets to initialize the weights of the network.

7. CNNs can be used for other image recognition tasks: CNNs can be used for other image recognition tasks such as object detection, semantic segmentation, and image captioning.

In conclusion, CNNs have proven to be highly effective for handwritten digit recognition, achieving outstanding performance on various datasets. Preprocessing, regularization, hyperparameter tuning, and transfer learning are essential techniques to improve the accuracy and generalization performance of the CNN model. Furthermore, CNNs can be used for other image recognition tasks, demonstrating their versatility and potential for various applications in computer vision.