Изображение выглядит как Шрифт, Графика, текст, логотип

Автоматически созданное описание

Software Engineering Department

Capstone Project Phase B

**Digi-Ktav   
A User-Friendly Website Using Deep Learning & LLMs to Digitize Cursive Hebrew Handwritten Documents.**

**Github-website and model usage**: <https://github.com/LiadTssf/FinalProject-hebrew-handwriting-recognition>

Github model train and model itself to download-<https://github.com/LiadTssf/vit-hebrew-character-ocr-DIGIKTAV>

A short video that showcases the site for users- <https://youtu.be/HmeZhrD8rA4?feature=shared>

**Supervisor: Dr. Reuven Cohen**

|  |  |  |
| --- | --- | --- |
| **Name** | Ilya Lazarev | Liad Grabois |
| **E-mail** | ilyalaz01@gmail.com | Liad3010@gmail.com |

**Project ID 25-1-D-20**

**Table of Contents**

|  |  |  |
| --- | --- | --- |
| **Index** | Subject | Page number |
|  | Abstract | 3 |
| 1 | **Introduction** | 3-6 |
| **1**.1 | Background | **3-4** |
| **1**.2 | Challenges in Hebrew Handwriting Recognition | 4 |
| **1**.3 | User Experience and Target Audience | 4 |
| 1.4 | Technical Approach | 4-5 |
| 1.5 | Document Structure | 5-6 |
| 2 | **System Description** | 6-16 |
| **2**.1 | Project Overview | 6-7 |
| **2**.2 | System Architecture | 7-9 |
| **2**.3 | Use Case Diagram | 9-11 |
| **2**.4 | Activity Diagram | 11-13 |
| **2**.5 | Technologies Used | 14-15 |
| **2.6** | Datasets | 16 |
| 3 | **Development Process** | 16-22 |
| **3**.1 | System Overview | 16 |
| **3**.2 | OCR Pipeline (Backend) | 17-21 |
| **3.3** | Frontend | 21 |
| **3.4** | API Communication | 21-22 |
| 4 | **Results** | 22-29 |
| 4.1 | Character Recognition Performance Evaluation | 22-26 |
| **4**.2 | End-to-End System Performance | 26-27 |
| **4**.3 | Evaluation Against Expected Achievements | **27-29** |
| **5** | **Challenges and Solutions** | **29-31** |
| **5.1** | Technical Challenges Encountered | **29-30** |
| **5.2** | Solutions and Approaches | **30-31** |
| **6** | **Conclusions** | **31-33** |
| 6.1 | Key Takeaways | **32-33** |
| 6.2 | Future Improvements | **33** |
| **7** | **References** | **33-34** |
| **8** | **User guide** | **34-41** |
| **8.1** | System Overview | **34** |
| **8.2** | Main Workflow | **35-39** |
| **8.3** | My Documents | **39-40** |
| **8.4** | Processing Options Page | **40-41** |
| **9** | **Maintenance Guide -DIGIKTAV** | **41-45** |
| **9.1** | Hardware and Software Requirements | **41** |
| **9.2** | Installation Instructions | **41-42** |
| 9.3 | How to Extend or Modify the System | **42-43** |
| **9.4** | Package and Architecture Overview | **44-45** |
| **10** | VIT Model Installation Instructions | **46-47** |
| **10.1** | Recommended System Specifications | **46-47** |
| 10.2 | Extending & Modifying the System | **47** |
| 10.3 | Architecture & Package Overview | **47-48** |
|  | Technical Terms | **48-49** |

**Abstract**

This project aims to develop a system for cursive Hebrew handwriting recognition, addressing challenges such as stylistic variability and cursive forms that existing solutions struggle to handle effectively. Current systems often fall short of providing practical usability, and this project seeks to create a reliable tool for digitizing cursive Hebrew handwritten documents.

The system includes a responsive web interface, enabling users to upload documents, view results in real-time, and optionally enhance the recognized text. The architecture follows a three-phase pipeline. First, handwritten images undergo preprocessing using [OpenCV](#OpenCV)1 for noise reduction and binarization. Then, segmentation techniques such as [horizontal projection profiles](#HPP)12 and [morphological filters13](#MorphologicalFiltering) extract lines and characters. Second, each character is classified using a Vision Transformer ([ViT](#VIT)9), a state-of-the-art deep learning model trained on a large, diverse dataset of labeled Hebrew characters. Finally, the recognized text is refined using Google's [Gemini API](#GeminiAPI)7, which improves punctuation, spacing, and overall fluency through contextual correction.

By combining modern vision and language models with a user-friendly interface, this project provides an effective and accessible tool for individuals, researchers, and archivists to digitize Hebrew handwritten documents.

# 1. Introduction

**1.1 Background**

Handwriting recognition technology converts handwritten content into machine-readable formats, supporting preservation of documents and cultural heritage. While systems for widely-used languages like English have achieved high accuracy using machine learning techniques, performance decreases significantly for complex scripts like Hebrew.

Current Hebrew handwriting recognition solutions remain inaccessible to non-technical users. Available systems require programming knowledge to implement and troubleshoot, creating a technological gap for individuals seeking to digitize handwritten Hebrew documents. This project addresses the need for an accurate, user-friendly system that effectively processes Hebrew script.

**1.2 Challenges in Hebrew Handwriting Recognition**

Hebrew handwriting recognition faces unique challenges due to its stylistic variability, diverse character shapes, and diacritical marks. While solutions exist for printed Hebrew text, tools for handwritten Hebrew often struggle with accuracy due to Hebrew's smaller user base compared to more widely-used languages.

. In cursive Hebrew handwriting, characters often blend or connect with one another, creating ambiguity in character boundaries. Our initial trials of using traditional segmentation methods like [contour detection](#ContourDetection)14 and watershed segmentation yielded disappointingly low accuracy rates (approximately 25%).

The lack of comprehensive datasets for Hebrew handwriting and limited support in many machine learning libraries further complicate the development process, explaining the reason why effective Hebrew handwriting recognition solutions remain scarce.

**1.3 User Experience and Target Audience**

The system is designed to simplify the complex task of Hebrew handwriting recognition through an intuitive, accessible web interface. Users can easily upload handwritten documents, view recognition results, and interact with features such as calibration for model personalization.

The platform is intended for a wide range of users, including researchers, archivists, educators, and institutions working with Hebrew manuscripts. By streamlining the digitization process, Digi-Ktav supports the preservation, analysis, and accessibility of handwritten Hebrew texts across various domains.

**1.4 Technical Approach**

Digi-Ktav is built around a modular OCR pipeline designed specifically for the challenges of Hebrew handwriting. The system uses a React-based environment for the frontend, which is a popular JavaScript library for building interactive user interfaces, and a Python [FastAPI](#FastAPI)5 environment for the backend, which is a modern, high-performance web framework for building APIs in Python. This architecture allows a clear division of concerns and supports a system that enables easy maintenance and upgrades.

The frontend system handles image upload, displays the processed text in printing letters alongside the original image, and allows users to correct the output or request to improve their text with spelling correction and structure improvements. The analysis of the handwritten text is done in the backend to keep the interface responsive.

The backend processes the image in three main stages: segmentation, character recognition, and text refinement. Each stage addresses a distinct technical problem.

* Segmentation uses traditional image processing methods like Horizontal Projection Profiles (HPP), which are techniques that count pixels along horizontal lines to identify text line boundaries, and morphological filters, a set of image processing operations that alter image shapes based on their structure, to extract text lines and characters. This rule-based approach avoids the need for labeled bounding box data but struggles with Hebrew’s curved and overlapping characters, which introduces recognition errors.
* Character recognition is performed using a Vision Transformer ([ViT](#VIT)9), a cutting-edge type of neural network that applies the transformer architecture (originally developed for language tasks) directly to image classification, trained on a diverse, augmented dataset of Hebrew letters. , ViTs divide the input image into fixed-size [patches](#VisionTransformerPatch)16 and treat each patch as a “token” in a sequence and using self-attention mechanisms that capture both local and global dependencies within the character sequence.
* Text refinement is handled by the Gemini API, Google's advanced large language model designed to understand and generate human-like text. After initial recognition, the backend assembles the output into full lines and sends it to Gemini with a structured prompt. Gemini corrects spelling, punctuation, and spacing errors, significantly improving word-level accuracy and enabling optional features like summarization or translation.

The ViT model achieved over 93% test accuracy in isolation, so no user calibration was needed. However, segmentation remains the system's main bottleneck. Replacing it with a trained method like [YOLO](#YOLO)10 (You Only Look Once), a popular real-time object detection algorithm that can identify and locate multiple objects in an image quickly, is a clear direction for future improvement.

**1.5 Document Structure**

This document is organized to provide a comprehensive overview of the Digi-Ktav system, from research foundations to technical implementation and user experience. It includes the following sections:

1. **Introduction**  
   Presents the motivation, background, and challenges of Hebrew handwriting recognition, introduces the target audience, and outlines the technical approach.
2. **System Description**  
   Describes the architecture and core components of the system, including the OCR pipeline, use case and activity diagrams, key technologies, and datasets used.
3. Development Process  
   Details the engineering journey from dataset preparation and segmentation to model training, post-processing, frontend design, and backend communication, including key implementation choices.
4. **Results**  
   Evaluates the performance of individual models and the full OCR pipeline, analyzing recognition accuracy, bottlenecks, and end-to-end system effectiveness.
5. **Challenges and Solutions**  
   Reviews the main technical and methodological challenges encountered during development and the approaches taken to overcome them.
6. **Conclusions**  
   Summarizes the outcomes of the project, reflects on key takeaways, and outlines possible directions for future work and improvement.
7. **References**  
   Lists all academic sources, datasets, and external tools referenced throughout the project.
8. **User Guide**  
   Offers step-by-step instructions on how to use the system, including uploading documents, reviewing results, and applying enhancements.
9. **Maintenance Guide**  
   Provides technical instructions for installing, extending, or deploying the system, including hardware requirements and backend/frontend setup.
10. **VIT Model Installation**  
    Includes guidance on running or fine-tuning the ViT character recognition model separately, with detailed notes on architecture and dependencies.
11. **Glossary of Technical Terms**  
    Defines specialized terms and technologies mentioned in the report to support reader understanding.

# 2. System Description

**2.1 Project Overview**

**Digi-Ktav** is a web-based system designed to accurately digitize handwritten Hebrew text. It allows users to upload scanned documents and convert them into editable digital text through a sophisticated, multi-stage pipeline. The system also supports post-recognition editing and offers various text enhancement features, giving users comprehensive control over the final result.

The backend processes each uploaded image through the following distinct stages:

* **Preprocessing:** The uploaded image is first automatically resized, enhanced through noise reduction, and has ruling lines removed. Crucial **morphological operations** are applied to refine character outlines and preserve handwriting structure, ensuring optimal quality for subsequent steps.
* **Segmentation** [**[2]**](#2zfl4cl8bfqm)**:** The processed image is segmented into individual text lines using the **Horizontal Projection Profile (HPP)**, and then into discrete characters and spaces using contour detection. Filters are applied to clean the segments, and the final character bounding boxes are sorted right-to-left for correct text assembly.
* **Character Recognition:** Each segmented character image is passed to a locally hosted **Vision Transformer (ViT)** model, which was specifically fine-tuned for Hebrew characters. The model classifies each character, and these predictions are then assembled into complete text lines, accurately preserving original spacing and line breaks.
* **Contextual Correction:** The assembled raw text is sent to the **Google Gemini API** for advanced contextual refinement. Gemini intelligently corrects OCR errors, spelling, punctuation, and resolves specific issues like letter confusion, while also removing diacritics. This ensures the corrected version accurately preserves the original meaning and structure.

Typically, the integrated pipeline achieves approximately 90% word-level accuracy, providing a highly reliable initial output. Users can then manually correct any remaining errors within the application's editor to achieve precise matching with the original content.

After recognition, the system displays the digital text side-by-side with the original image. Users can further edit the text, save it to their account, or download it. Additionally, the system offers optional enhancement features, powered by the Gemini API, such as summarization, translation to English, or academic rephrasing, providing adaptable tools based on user needs.

**2.2 System Architecture**

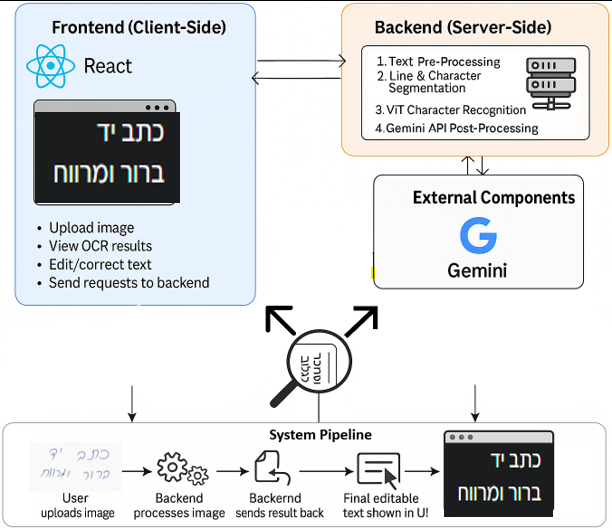


Figure 1: System Architecture Diagram

The **frontend** is implemented as a web application using React and Vite. It allows users to upload handwritten images, view the recognized output next to the original image, correct recognition mistakes in a text editor, and apply optional enhancement features. After processing, users can choose to download the final text or save it to their account if logged in.  
The **backend**, developed with Python and FastAPI, acts as the system's core processing engine. It receives uploaded images from the frontend, passes them through the OCR pipeline, interacts with the Gemini API for text refinement, and returns structured results to the user interface. This architecture supports asynchronous request handling and allows for scalable deployment.  
The **OCR pipeline** itself includes several stages. First, the uploaded image is resized and enhanced through preprocessing techniques. Ruling lines are removed, and morphological operations are applied to maintain character structure. Horizontal projection profiles are then used to segment the image into text lines. These lines are further segmented into characters through contour detection and filtering. Character fragments, such as disconnected parts of letters, are merged, and all components are reordered from right to left while detecting spaces between them.  
Each segmented character image is standardized to appropriate dimensions and then passed to a locally hosted Vision Transformer (ViT) model. This model, meticulously trained on a diverse dataset of Hebrew characters, analyzes the visual features within each image to classify it, predicting the most probable Hebrew letter. These individual character predictions are then sequentially assembled from right-to-left, forming coherent lines of text while diligently preserving the original spacing and newline breaks, thereby reconstructing the handwritten content digitally.  
**Once the text has been assembled**, it is sent to the Gemini API for contextual correction. A carefully designed prompt guides Gemini to fix spelling, punctuation, and structural errors, remove diacritics, and resolve common letter confusions. The user can also choose to apply additional processing options such as summarizing the content, translating it to English, or rephrasing it in a more formal tone. These features are triggered through the same Gemini interface, with prompts adapted to the selected user action.  
**After correction**, the final text is returned to the frontend and shown alongside the original version. Users can edit the text further, download the results, or save them to their account. The system supports typical use cases without requiring any user training or model calibration.  
In addition to these components, the backend is connected to a document-oriented database, which stores user data, uploaded document metadata, and processed results. This allows logged-in users to revisit previous work and retrieve their saved outputs.  
**Digi-Ktav’s** architecture is inherently modular, prioritizing clarity, flexibility, and maintainability in its design. This structured approach means that each component, from image segmentation and character recognition to post-processing and the user interface, can be independently improved, updated, or even replaced without impacting the rest of the system. This foresight allows for continuous internal improvements, such as enhancing the core character recognition models, refining the text processing capabilities for even greater accuracy, or integrating more robust segmentation techniques like YOLO, thereby ensuring the platform's long-term adaptability and growth.

**2.3 Use Case Diagram**

**Use Case Explanation**

The system supports two user types: **Guest Users** and **Logged-in Users**. Both roles have access to the core functionality of the application, including document upload, OCR processing, manual correction, and AI-based text enhancements. Logged-in users can also save and manage their documents for later use.

**Common Functionalities (All Users)**

* **Upload Handwritten Document**  
  Users upload an image of a handwritten Hebrew document. The backend processes the image using the OCR pipeline, which includes preprocessing, segmentation, character recognition with a ViT model, and contextual correction via the Gemini API.
* **View and Edit Recognized Text**  
  The recognized text is shown next to the original scanned image. Users can manually correct any errors in a text box to ensure the result matches the original content.
* **Apply Text Enhancements (Optional)**  
  After editing, users can choose to apply additional AI-based transformations:
  + **Summarize Text**  
    Generates a short summary of the corrected Hebrew text.
  + **Translate to English**  
    Produces an English version of the corrected Hebrew text.
  + **Rephrase to Academic Style**  
    Reformats the text into a more formal and academic tone.
* **Automated Misspelling Corrections:** This option focuses on fixing spelling mistakes. For instance, if the system detects *"אני ימלה את הבקבוק “ani yemale et habakbuk” (I’ll fill the bottle - with spelling mistakes)*, it will correct it to *"אני אמלא את הבקבוק" “ani emale et habakbuk”( I’ll fill the bottle - without spelling mistakes)*, following proper Hebrew grammar and spelling rules.

These enhancements are handled by the Gemini API using adaptive prompts that adjust according to the selected feature.

* **Download Output**  
  Users can download the final version of the text (original or enhanced) as a document file.

**Additional Functionalities (Logged-in Users)**

* **Save Processed Documents**  
  Logged-in users can save their digitized documents in their personal account for later access or download.
* **Access Document History**  
  Enables users to view a list of their previously processed and saved files.

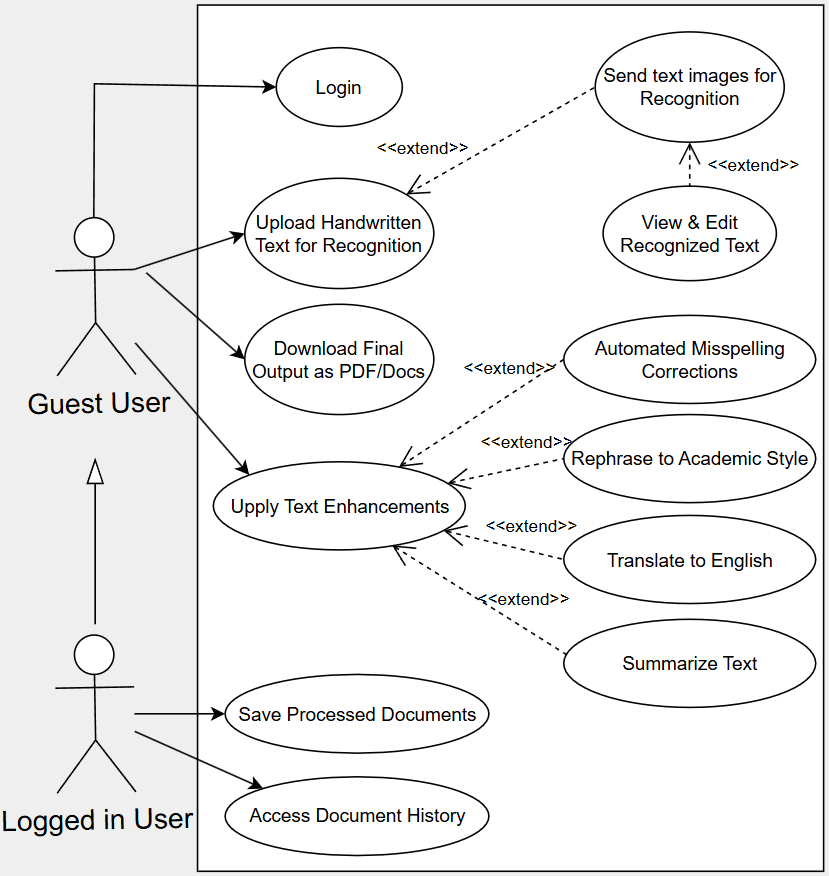


Figure 2: Use Case Diagram of the System

**2.4 Activity Diagram**

The process begins when the user uploads an image of handwritten text through the frontend interface. The frontend sends the image to the backend, where it is first validated and then passed through a preprocessing stage. This stage resizes and normalizes the image and prepares it for segmentation.

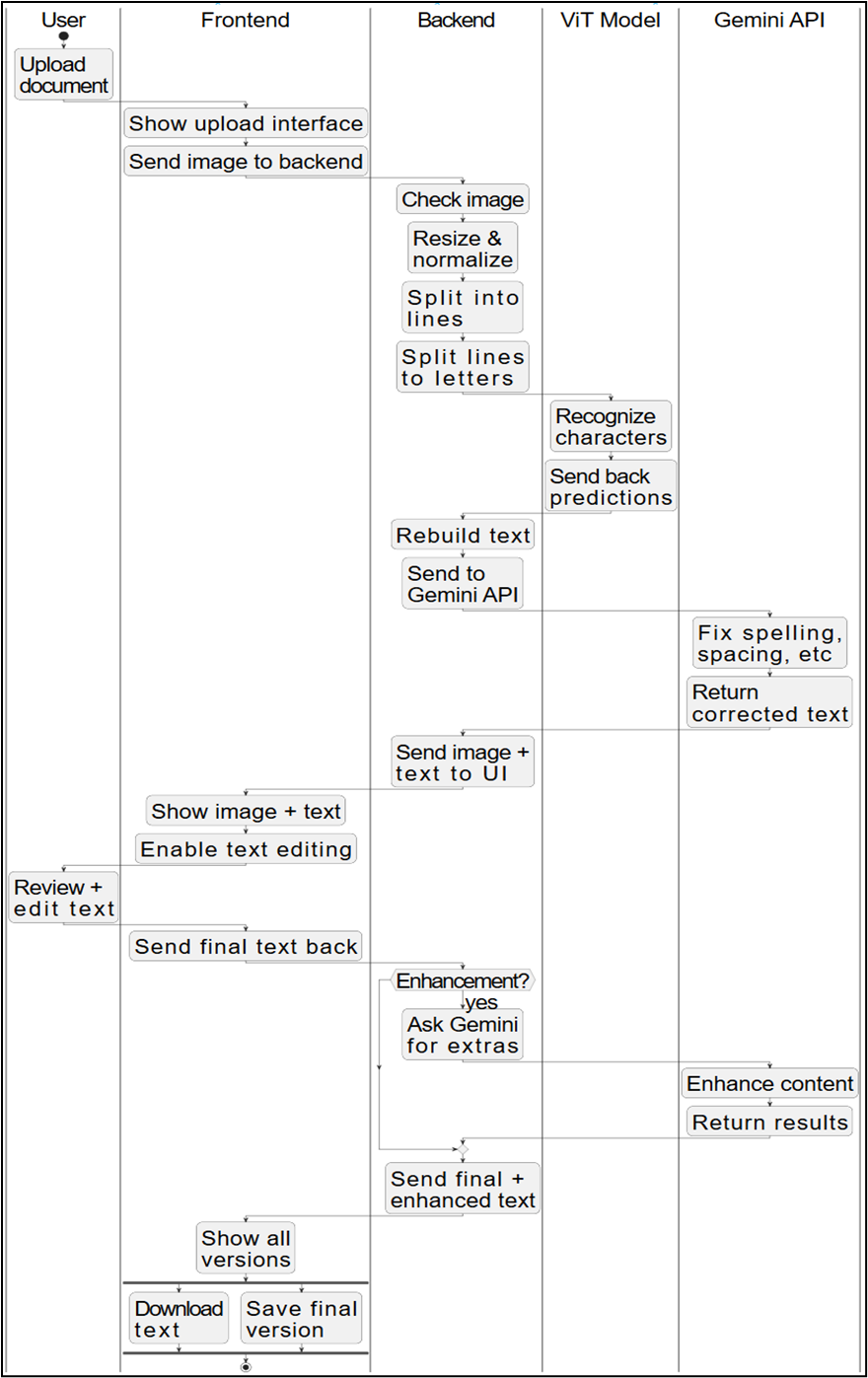
Next, the backend segments the image into individual lines and further splits each line into characters using classical image processing techniques. These character crops are then passed to a locally hosted Vision Transformer model, which predicts the corresponding Hebrew letter for each image.

The character predictions are assembled into complete lines of text, preserving spaces and line breaks. This raw text is then sent to the Gemini API, which corrects spelling, punctuation, and formatting errors while removing diacritics. The corrected version is returned to the backend and sent back to the frontend, along with the original image.

At this point, the user sees the recognized text alongside the image and is given the option to manually correct any remaining recognition mistakes in a text editor. Once the user finishes editing, the final corrected text is sent to the backend.

If the user chooses an enhancement feature such as summarization, translation, or academic rephrasing, the backend sends the corrected text to the Gemini API with an adaptive prompt matching the selected action. Gemini returns the enhanced version, which is added to the output.

Finally, the frontend displays both the corrected original and enhanced versions of the text. The user can then download the result or save it to their account.

Figure 3: Activity Diagram of the System

**2.5 Technologies Used**

Digi-Ktav integrates multiple technologies across both the frontend and backend to support Hebrew handwriting recognition and text enhancement.

**Frontend**

**React with Vite**

We used React, a popular JavaScript library for building dynamic and interactive user interfaces. React enables the creation of reusable UI components, leading to a modular and scalable frontend architecture for the application. Vite was chosen as the build tool for its exceptional speed during development, offering features like instant server start and hot module replacement, which significantly improves developer experience. For production, Vite ensures efficient bundling of assets, optimizing load times for users. The frontend, powered by these tools, allows users to intuitively upload handwritten documents, view and interactively edit the OCR results, and access various text processing features.[[7]](#mf0x07wqgsed)

**Axios**

[Axios6](#Axios) is a widely used, promise-based HTTP client for JavaScript. In Digi-Ktav, it is essential for making asynchronous HTTP requests from the frontend to the backend FastAPI server. This includes securely handling file uploads, retrieving processed text outputs, and managing all other data exchanges between the client and server. Its robust error handling and request cancellation capabilities make it a reliable choice for efficient communication.

**Tailwind CSS (optional)**

Tailwind CSS is a utility-first CSS framework that facilitates rapid UI development by providing a vast collection of pre-defined utility classes. If styling was customized beyond the default, Tailwind was instrumental in maintaining a consistent and highly adaptable design across the entire application without needing to write custom CSS from scratch, promoting cleaner code and faster iteration.

**Backend**

**Python with FastAPI**Python serves as the primary programming language for the backend, offering a rich ecosystem of libraries for machine learning and data processing. FastAPI acts as the core web framework, specifically chosen for its outstanding performance (comparable to NodeJS and Go) and its modern features. It leverages standard Python type hints for automatic data validation, serialization, and generates interactive API documentation (Swagger UI/ReDoc) out-of-the-box, making it incredibly easy to structure the OCR pipeline as clear, callable API endpoints for seamless integration with the frontend.  
**NumPy and OpenCV**[NumPy](#NumPy)2 is the fundamental package for numerical computing in Python, providing powerful array objects and tools for mathematical operations. It is indispensable for handling and manipulating image data, which is essentially represented as multi-dimensional arrays of pixels. OpenCV (Open Source Computer Vision Library) is a comprehensive library dedicated to real-time computer vision tasks. These libraries are extensively used for all image preprocessing operations within the backend, including resizing images to standard dimensions, applying thresholding for binarization, performing contour detection to identify distinct characters, and conducting projection profile analysis for line segmentation.  
**Transformers (Hugging Face)** [**[1]**](#ed2x1apjyc9b)[**[4]**](#viomedc3gas8)We utilized Hugging Face’s Transformers library, a leading open-source library that provides state-of-the-art pre-trained models for various machine learning tasks, including computer vision. This library allowed us to easily load and fine-tune a pre-trained Vision Transformer (ViT) model, specifically the google/vit-base-patch16-224-in21k base model, to work effectively for Hebrew character recognition, rather than building a model from scratch. The associated ViTImageProcessor component is crucial for standardizing the input image format (e.g., resizing, normalization) to the exact specifications required by the ViT model before classification, ensuring optimal model performance.**Google Gemini API**After the initial character recognition generates raw OCR output, the backend transmits this text to the Google Gemini API. Gemini is a powerful, multimodal large language model capable of understanding and generating human-like text with exceptional nuance. This API plays a critical role in the post-processing phase, performing advanced corrections for spelling, punctuation, spacing, and resolving known OCR-specific errors. Its ability to understand context significantly refines the accuracy and readability of the final text.

**Model & Dataset Tools**

[**PyTorch**](#PyTorch)**3**The Vision Transformer (ViT) character classification model was rigorously trained and evaluated using PyTorch, an open-source machine learning framework known for its flexibility and intuitive interface. PyTorch’s dynamic computation graph allows for easy debugging and experimentation with complex neural network architectures. It enabled precise control over training loops, efficient loading of large datasets, and seamless utilization of GPU acceleration for faster model training and inference.[**TorchVision4**](#TorchVision)TorchVision is a package within the PyTorch ecosystem specifically designed for computer vision tasks. It provides a rich collection of common datasets, pre-trained models, and essential image transformations. In our project, TorchVision was invaluable for efficient dataset handling (e.g., loading and preparing image datasets) and applying various transformations (e.g., resizing, normalization, [data augmentation15](#DataAugmentation)) during both model training and evaluation phases, ensuring the data is in the optimal format for the ViT model. **Custom Dataset Loader**Given the unique characteristics of our Hebrew handwritten character data, we implemented a custom folder-based dataset loader. This loader was specifically structured to organize and access image crops categorized per character class. This tailored approach ensured that the ViT model was fed with correctly formatted and labeled data efficiently during the training process, which is crucial for achieving high recognition accuracy.

**2.6 Datasets**

To train a robust recognition model for handwritten Hebrew characters, we used a composite dataset built from multiple existing sources. We focused on visual diversity and consistency across samples to help the model generalize to different handwriting styles.[[3]](#xsft1mrhqg47)

**2.6.1 HHD Dataset (Hebrew Handwritten Dataset)**[[3]](#xsft1mrhqg47)

One of the main datasets used in this project is the HHD dataset from Ben Gurion University. It contains thousands of grayscale images of handwritten Hebrew characters and short words, collected from various writers. This dataset provides a diverse sample of real-world handwriting styles and has been used in prior academic research.

**2.6.2 Moranzargari Letters Dataset**[[3]](#xsft1mrhqg47)

This dataset, publicly available on GitHub, contains individual handwritten Hebrew letters organized by class. Although more limited in structure compared to HHD, it complements the main dataset by providing clean, isolated characters. The samples vary in thickness, spacing, and curvature, which supports training a character-level classifier.

**2.6.3 Combined Custom Dataset**[[3]](#xsft1mrhqg47)

We created a unified dataset by merging the HHD and Moranzargari datasets and incorporating additional handwritten Hebrew data from public OCR repositories. This combination provided wide visual variability in character appearance, contrast, and background.

Key features of the final dataset:

* 27 character classes (each class represents a letter).
* Over 6,000 images per class in the training set after augmentation.
* Mixed image styles, including white-on-black and black-on-white, blue-on-white.
* Standardized to 72×72 resolution before training.

This mixed-source dataset provided a good base for training a general-purpose recognition model without needing to generate synthetic data manually.

**2.6.4 Dataset Augmentation**

To further improve generalization, we applied data augmentation to every image in the training set, generating **three augmented versions** of each source image. This process effectively **tripled the size** of the dataset.

Each original image underwent the following pre-processing:

* **Auto-orientation** (removal of [EXIF](#EXIF)20 rotation metadata)
* **Resizing** to 72×72 pixels (stretch applied)

Then, for each of the three augmented copies, we applied **random transformations**:

* **Cropping**: up to 10% randomly cropped from the image edges
* **Rotation**: random angle between -3° and +3°
* **Shear**: up to ±6° horizontally and vertically
* **Brightness**: adjusted randomly within ±13%
* **Exposure**: adjusted randomly within ±10%
* **Gaussian blur**: applied with strength between 0 and 1.1 pixels
* [**Salt-and-pepper noise**](#SaltandPepperNoise)**21**: added to 0.06% of pixels

This augmentation process introduced realistic variations in character shape, alignment, and noise. It helped the model become more robust to real-world inputs, especially when dealing with slight distortions, inconsistent lighting, or common scan artifacts.

# 3. Development Process

**3.1 System Overview**

The development of Digi-Ktav was shaped by the goal of creating a practical and modular system for recognizing handwritten Hebrew text. The project combines a React-based frontend for user interaction with a FastAPI backend that performs image processing. The OCR pipeline includes character recognition using a locally integrated Vision Transformer (ViT) model, followed by language-level correction using the Gemini API. The system reflects a working integration of visual recognition and contextual refinement tailored for Hebrew handwriting.

**3.2 OCR Pipeline (Backend)**

**3.2.1 Data Preparation**

To prepare training data for the character recognition model, we merged several datasets containing handwritten Hebrew characters, including [HHD](https://cris.bgu.ac.il/en/publications/the-hhd-dataset-2), [Moranzargari’s letter collection](https://github.com/moranzargari/Handwriting-detection-recognition/blob/master/Letters%20Dataset%20Collection/dataset.zip), and an open-source [repository by Sofia Naer](https://github.com/SofiaNaer/Hebrew-Handwriting/tree/master/dataset). These sources varied in quality, background, and style, which helped introduce useful diversity.

Images were organized into 27 classes, one per Hebrew letter. We applied basic preprocessing to ensure consistent size and orientation across the dataset.

To expand the dataset and improve robustness, we applied data augmentation during preprocessing. This step is detailed in Section 2.6.4 and was applied before training, resulting in over 6,000 images per class, or more than 160,000 labeled character images.

The final dataset, combining real-world variability with controlled augmentation, enabled the ViT model to learn from a wide range of handwriting styles and distortions.

**3.2.2 Segmentation**[**[2]**](#2zfl4cl8bfqm)

The initial step in the OCR pipeline focused on preparing scanned handwritten Hebrew documents for character-level recognition. To accelerate development and avoid reinventing well-established techniques, we adopted and adapted an open-source implementation of image segmentation and morphological preprocessing available on [*GitHub repository*](https://github.com/kasbekarameya/Morphology-Image-Processing-Point-Detection-Image-Segmentation-and-Thresholding-Hough-Transform).

This repository provided a foundation for:

* Horizontal projection profile (HPP) based line segmentation
* Morphological filtering for noise reduction
* Contour-based letter detection using OpenCV

We modified the codebase to suit the unique characteristics of Hebrew handwriting and our specific model input requirements. Key adjustments included:

* Custom height and width thresholds tuned for Hebrew scripts
* RTL (Right-to-Left) bounding box sorting
* Additional logic to preserve characters with overhanging elements (e.g. ל)
* Introduction of space detection by analyzing distances between adjacent contours

Each input image was converted to grayscale, resized, and binarized. Lines were extracted using HPP peaks, and individual characters were segmented using contour analysis with aspect ratio and area filtering. All character crops were padded, centered, and resized to 72×72 pixels before passing into the recognition model.

While the core segmentation code was taken from an open-source project, we made several adjustments to make it suitable for Hebrew text. These included right-to-left character ordering, detecting spaces between words based on gap sizes, and improving the handling of Hebrew letter shapes—especially in cases where characters with overhanging or disconnected strokes were incorrectly split or merged due to scan quality.

**3.2.3 ViT Model Integration**

As we began evaluating alternatives to our originally planned CNN approach, we encountered a dataset [[3]](#xsft1mrhqg47) containing approximately 800 scanned images of handwritten Hebrew text, with each image averaging around 60 words. These samples were collected from roughly 100 different writers. The idea of training a Vision Transformer (ViT) model to recognize full lines of text — instead of individual characters — seemed promising and potentially more robust.

We labeled the dataset accordingly and attempted to train a ViT model to directly predict text at the line level. However, the experiment did not yield usable results. In hindsight, the dataset was too small and unbalanced for this type of task. More importantly, our system architecture, dataset format, and OCR pipeline were all originally designed around letter-level recognition, as planned in Phase A.

Because we were better prepared — both in terms of data quantity and preprocessing tools — for character classification, we shifted our focus back to the letter-level approach and began training the ViT model using the dataset we had curated and augmented for that purpose.

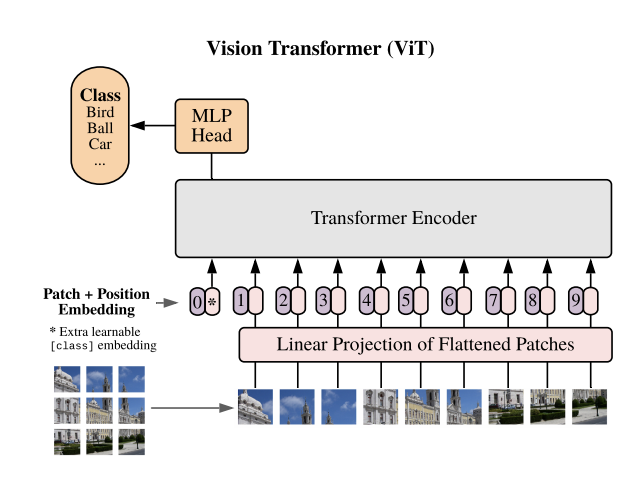
We implemented and evaluated two versions of the ViT model for this task. First, we trained our own Vision Transformer model from scratch using images resized to 72×72 pixels. The model was implemented using PyTorch and HuggingFace Transformers and trained with the [Adam18 optimizer](#AdamOptimizer) and [cross-entropy loss17](#CrossEntropyLoss). This custom model achieved a validation accuracy of approximately 84.6%, which provided a solid starting point.

Following this, we tested a pretrained model from HuggingFace — google/vit-base-patch16-224 — which was originally trained on large-scale natural image datasets. Since this model expects RGB input at 224×224 resolution, we adapted our character images:

* Converted grayscale images into 3-channel RGB format
* Resized to 224×224
* Normalized using ViTImageProcessor from the Transformers [**[1]**](#ed2x1apjyc9b)[**[4]**](#viomedc3gas8)library

Training the pretrained model required high computation resources due to the large input size and the model complexity, but it achieved better results — around 94% accuracy on the same validation set. Thus, we integrated this model into our final system.

Unlike CNN algorithms, which use local filters to extract spatial features, the ViT model partitions every image into small patches and processes them in a sequence. Each patch is encoded, and the model then uses a self-attention system to learn the relationships between these patches across the entire image. This self-attention mechanism allows the model to dynamically weigh the importance of different parts of the input image relative to each other when processing each patch. By considering global dependencies, rather than just local ones, this global processing allows the model to better recognize characters that contain visually distant parts — like tall strokes, dots, or disconnected lines — which are common in Hebrew handwriting. This enables a more comprehensive understanding of the character's overall structure, even when its components are spatially separated.The ViT architecture proved to be well-suited for this task, offering improved accuracy and robustness over the CNN-based models. It helped to improve recognition accuracy in cases where traditional CNN-based models often struggled.

Figure 4: Vision Transformer Achitecture diagram [**[1]**](#ed2x1apjyc9b)[**[4]**](#viomedc3gas8) 

**3.2.4 Post-Processing with Gemini API**

Following the character-level prediction stage, the model outputs a raw Hebrew string composed of reconstructed lines from segmented characters. Due to the inherent challenges of OCR for handwritten Hebrew, this text may contain various recognition errors, including:

* Incorrect letter recognition
* Missing or extra characters
* Incorrect spacing between words
* Incomplete words due to segmentation issues

To improve the final output, we integrated Google’s Gemini API to act as a post-correction language model. The OCR result is sent to Gemini along with a carefully engineered prompt that guides it to make targeted corrections while preserving the structure and intent of the original text.

The prompt:

* Informs Gemini that the input is a noisy OCR result with ~75% character-level accuracy
* Lists common recognition mistakes observed during testing such as ו→י, נ→כ, ר↔כ, ט↔ל
* Includes a dedicated section describing common OCR issues. For example, the Hebrew letter ל (Lamed), particularly overhang errors where the upper stroke of Lamed extends far to the right and overlaps the previous character. This causes the segmentation step to group both letters into a single wide bounding box. When this happens with a small adjacent letter, the combined shape is often misrecognized as the letter ט (Tet).(Figure 4)
* Specifies strict rules for correction:
  + Do not paraphrase, reword, or summarize
  + Do not add or invent new words
  + Do not include any Nikkud (vowel marks)
  + Preserve every line and word unless clearly unreadable
  + Only correct individual characters, punctuation, or spacing based on context

This detailed guidance ensures that Gemini acts as a focused corrector rather than a generative rewriter.

Gemini’s output is a cleaned, semantically accurate Hebrew text that reflects the original intent of the handwritten input while correcting likely OCR mistakes. This stage significantly improves readability and usability, especially for edge cases such as merged characters, ambiguous letters, or inconsistent spacing. Without this correction step, raw OCR output typically achieves only 65 to 75 percent word-level accuracy due to segmentation errors. After Gemini refinement, accuracy improves significantly, often reaching between 80 and 95 percent. This detailed guidance ensures that Gemini acts as a focused corrector rather than a generative rewriter.

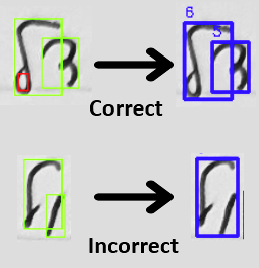


Figure 5: Example of Overhang Error in Lamed (ל) Character Segmentation and Recognition  
*Correct segmentation separates ל and the following character. Incorrect segmentation merges both into one box.*

3.3 Frontend

The client interface was implemented using the React framework with the Vite build tool for fast development and responsiveness. It allows users to interact with the OCR system and manage the full recognition and enhancement workflow. The full set of supported actions is described in the Use Case Diagram (Section 2.3). All requests are sent asynchronously to the FastAPI backend, including [base64-encoded](#Base64EncodedImage)19 image uploads and user-selected enhancement options. The interface includes visual feedback, error handling, and progress indicators throughout the process.

3.4 API Communication

The Digi-Ktav system uses a modular client-server architecture, where the React frontend communicates with the FastAPI backend via RESTful HTTP endpoints. This separation allows for independent development, testing, and deployment of both components.

The backend exposes the following primary endpoints:

* **POST /process-image/**Receives an uploaded image (PNG, JPG, etc.), processes it through the OCR pipeline—including segmentation and character recognition—and returns the raw predicted Hebrew text.
* **POST /enhance-text/**  
  Accepts the raw OCR output and enhancement options, forwards it to the Gemini API via a structured prompt, and returns a cleaned and linguistically corrected version of the text.
* **POST /compress-image/**  
  Compresses base64-encoded image data to meet size constraints for frontend display or storage. This is used to optimize large images before OCR processing.

Additionally, the backend includes:

* **GET /health/**  
  Returns basic server and component availability status, including Gemini integration health.
* **GET /debug/**  
  Provides in-depth diagnostic information, such as environment variable status and package availability.

Each API response typically includes:

* The recognized Hebrew text (raw OCR output)
* The enhanced or corrected version (if applicable)
* Metadata such as processing time, file name, and error details if any

This flexible API design supports future expansion—for example, integrating new model types, formats, or user-specific personalization—all without requiring major changes to the frontend.

# 4. Results

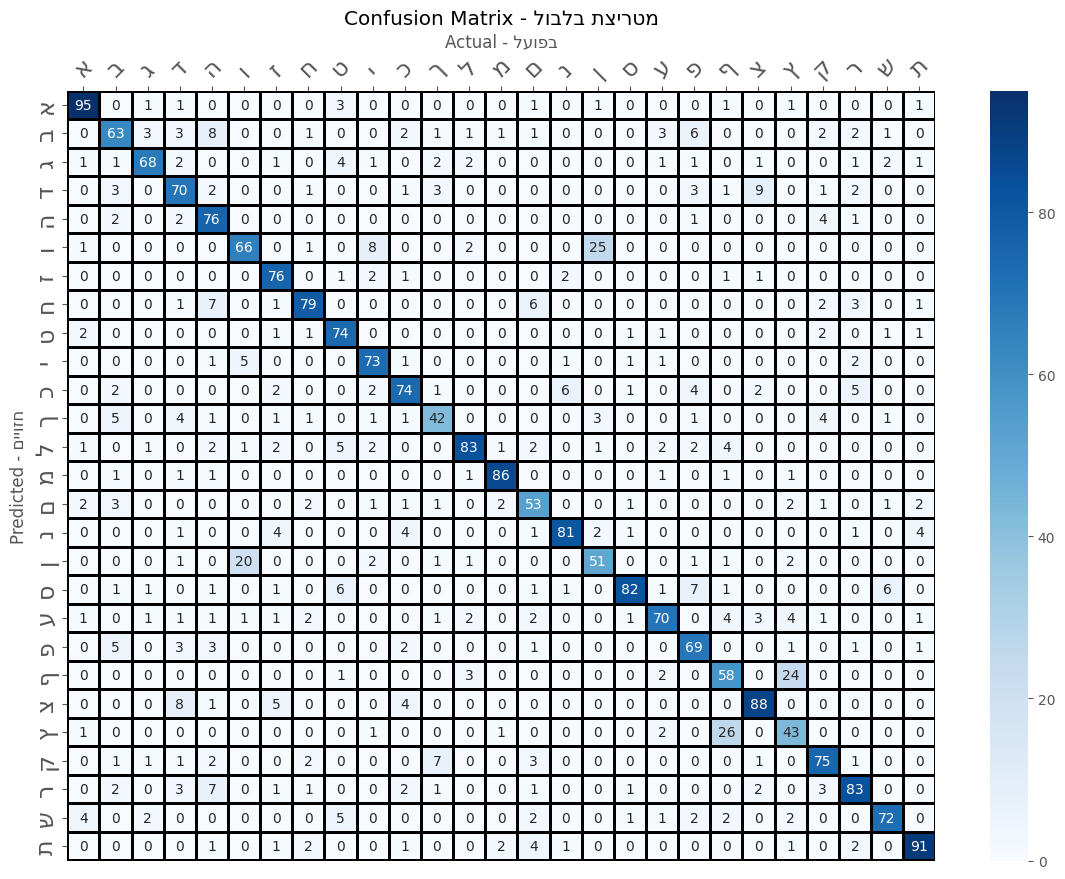
**4.1 Character Recognition Performance Evaluation**To assess the performance of our Hebrew handwriting recognition system, we conducted a comprehensive evaluation of three distinct model configurations: a Convolutional Neural Network (CNN), a Vision Transformer (ViT) trained from scratch, and a ViT model fine-tuned from pretrained weights. All three models were trained and evaluated on the same dataset described in Section 3.2.1, ensuring consistent comparisons across architectures. Performance was measured using validation accuracy, validation loss, and detailed confusion matrices.

Model 1: Convolutional Neural Network (CNN)

As a baseline, we implemented a simple convolutional neural network trained for 10 epochs on grayscale images resized to 72×72. The model was trained using the Adam optimizer with a sparse categorical cross-entropy loss function. Due to limited gains observed beyond the early epochs we limited training to 10 epochs.

* Final validation accuracy: 77.4%
* Final validation loss: 2.12

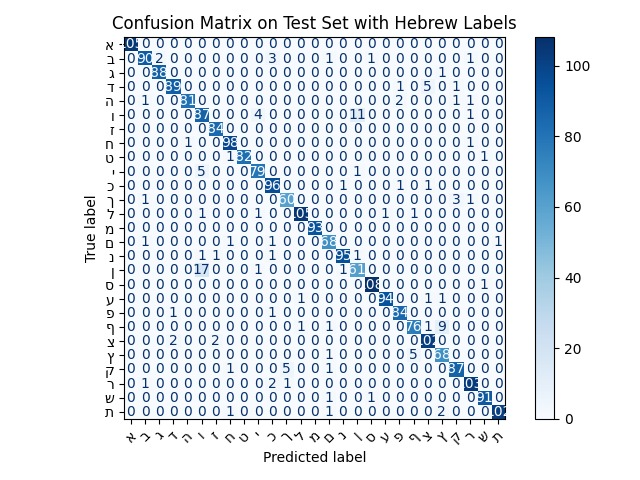
This model demonstrated fast initial convergence, reaching its peak performance around epoch 3. Subsequent epochs showed continued improvement in training accuracy but stagnation in validation accuracy, indicative of overfitting. The confusion matrix (Figure X) shows notable misclassifications between visually similar Hebrew characters for example: ן/י, ף/ץ, and ר/ד.

Figure 6: confusion matrix of cnn

**Model 2: Vision Transformer (ViT) – Trained from Scratch**  
This configuration employed the ViT-Base architecture trained from scratch on RGB images resized to 224×224. The training process spanned 71 epochs, with performance improving steadily throughout.

* Final validation accuracy: 87.7%
* Maximum validation accuracy: 88.0%
* Validation loss range: 0.45 to 0.50
* Test accuracy: 91.9%

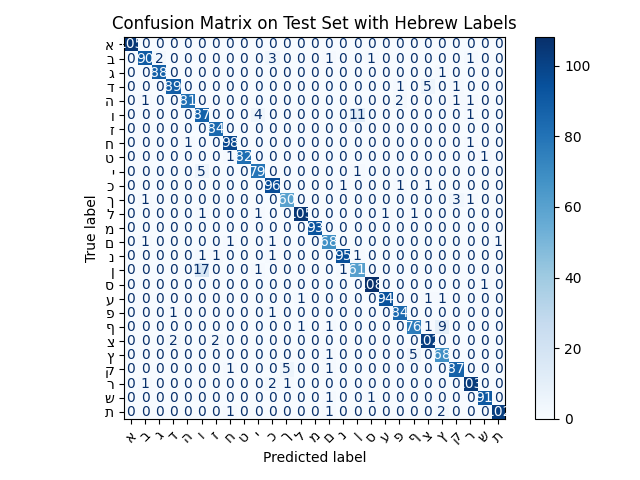
The scratch-trained ViT significantly outperformed the CNN in accuracy and robustness. It required considerably more epochs to converge and exhibited fluctuations in validation performance. The confusion matrix (Figure Y) and the corresponding classification report revealed strong performance across most classes, with especially high precision for characters such as א, מ, and ש. Nevertheless, some minor confusion remained, especially for visually ambiguous characters like ו and ן.

****Figure 7: confusion matrix of model 2

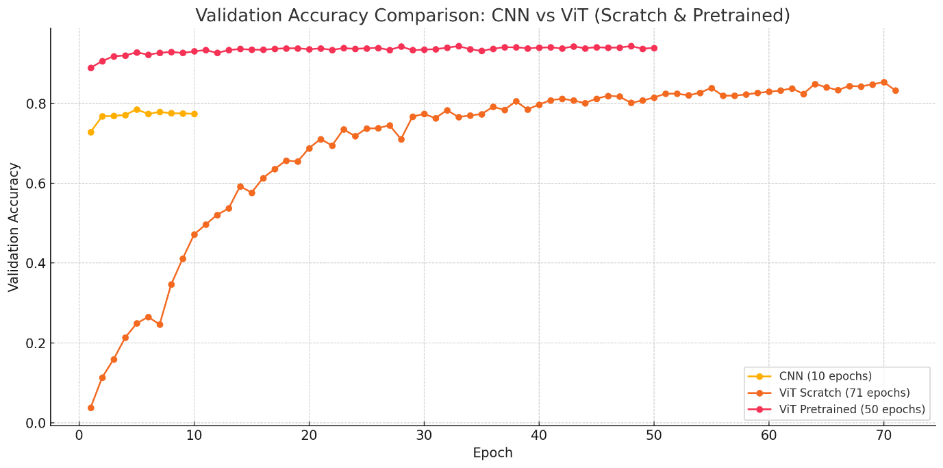
**Model 3: Vision Transformer (ViT) – Fine-Tuned from Pretrained Weights**  
Our best-performing configuration leveraged transfer learning. The google/vit-base-patch16-224 model was fine-tuned on the same dataset as previous models, using identical preprocessing.

* Final validation accuracy: 94.4%
* Peak accuracy: 95.19% (epoch ~49.7)
* Minimum validation loss: 0.4496
* Test accuracy: 93.0%

This model achieved the highest overall performance. Validation accuracy remained above 94% during the last 15 epochs, reflecting a strong learning plateau. The confusion matrix (Figure Z) and classification report confirmed improved generalization across the entire label set, reducing confusion for difficult character pairs seen in earlier models. Though the confusion matrix may initially appear more balanced in the scratch-trained model, the fine-tuned model achieves more consistent per-class precision and recall, especially in problematic classes such as ף, ץ, and ן. This consistency, combined with higher average accuracy and more stable training behavior, led us to select the fine-tuned ViT as our final model.

  
Figure 8: confusion matrix of model 3

**Comparative Analysis**  
Figure W presents a comparative plot of validation accuracy across all three models. The CNN reached a performance ceiling quickly, whereas the ViT trained from scratch required extensive training to outperform it. The fine-tuned ViT achieved the highest overall and per-class metrics, illustrating the substantial benefits of pretraining on large-scale datasets before fine-tuning for Hebrew handwriting.

  
Figure 9: validation accuracy comparison

**4.2 End-to-End System Performance**

While the ViT recognition model reached high accuracy in isolation, evaluating the full pipeline (from segmentation to final output), revealed the real performance limits of the system in a real-world setting. The primary bottleneck is the character segmentation stage, which struggles with curved and non-uniform shapes typical in handwritten Hebrew. The segmentation algorithm is rule-based and mathematical, relying on contour detection and morphological filtering. This approach is sensitive to overlapping characters, irregular spacing, and overhanging strokes. Since Hebrew includes many curved letters, such as ק or ל, and writing styles vary widely, the segmentation often fails to isolate characters correctly, resulting in merged or fragmented bounding boxes.

Due to time and resource constraints, we did not implement a machine learning-based segmentation method such as YOLO[[5]](#rlh66pyfbeaf). Instead, we used deterministic methods, which are easier to implement but less flexible. While not ideal, this approach allowed us to test the rest of the pipeline effectively and left room for future improvement.

To estimate the real-world performance of the full system, we conducted a rough but informative test. We used three different handwritten Hebrew texts and ran each through the full pipeline. We recorded results at two points: the raw OCR output (after segmentation and ViT recognition) and the final output (after Gemini correction and minor manual edits as supported by the system). We then compared these outputs to the original texts using character and word accuracy metrics.

The average results across the three texts were:

|  |  |  |
| --- | --- | --- |
| Stage | Character Accuracy | Word Accuracy |
| Segmentation + OCR | 65–75% | ~55% |
| After Gemini Corrections | ~90% | ~85% |

These results confirm that segmentation is the weakest point in the pipeline. The ViT model itself performs well, achieving 94% test accuracy in a controlled dataset. However, poor segmentation leads to broken or merged characters, which undermines the recognition stage. For example, when two letters are connected into a single bounding box or a single letter is cut in half, even the best classifier cannot produce accurate results.

Despite these challenges, the Gemini post-processing step significantly improves the final output. By using context to correct probable errors, it boosts overall accuracy by about 25–30 percentage points in both character and word-level metrics. This makes the final system usable for practical transcription, especially when combined with final user corrections through the interface.

**4.**3 **Evaluation Against Expected Achievements**

Our project set out with clearly defined technical and user-oriented goals. Below is a structured evaluation of our final system’s performance against these expectations, as outlined in Phase A.

**Primary Objectives**

|  |  |  |
| --- | --- | --- |
| Objective | Status | Explanation |
| High accuracy in Hebrew handwriting recognition | ✅ Partially Achieved | The ViT-based character recognition model reached 94.4% validation accuracy and 93.0% test accuracy, significantly exceeding the 85% baseline target. However, due to limitations in the segmentation stage, overall accuracy across the full text recognition pipeline was lower. The recognition model performed well in isolation, but the end-to-end system performance remains limited by segmentation quality. |
| Fast and reliable OCR pipeline | ❌  Not Achieved | The average inference time per character was approximately 0.13 seconds, with full image processing completed in under 2-4 minutes depends on amounts of letters in image. |
| Accessible and easy-to-use platform | ✅ Achieved | The web interface supports image upload, result visualization, and editing with minimal interaction, targeting users with various levels of technical expertise. |

**Expected System Performance**

|  |  |  |
| --- | --- | --- |
| Component | Status | Notes |
| Integrated pipeline (preprocessing + recognition + correction) | ✅ Achieved | The final pipeline integrates OpenCV-based preprocessing, ViT-based recognition, and Gemini-based language correction. |
| Image enhancement (noise reduction) | ✅ Achieved | Custom preprocessing routines using OpenCV ensured high-quality character segmentation and denoising. |
| Gemini API for post-recognition correction | ✅ Achieved | The Gemini API was integrated into the backend to refine OCR output. In testing, it improved word accuracy by 20 to 25 percentage points on average, raising results from around 40% to 65%. |
| Target system accuracy of ~85% | ✅ Achieved | System accuracy reached ~80-90%. As explained earlier, overall system accuracy did meet the target but under strict guidelines in order to get maximum accuracy. |

**Success Criteria**

|  |  |  |
| --- | --- | --- |
| Criterion | Achieved? | Details |
| Document digitization within 10 seconds | ❌ | Average end-to-end processing time: ~40 seconds per full text page which means 2-4 minutes depends on the amount of letters in the image. |
| ≥80% user satisfaction (ease of use) | ✅ | showed mostly close circle people and got verbal feedback which was really satisfied with a bit of comments about expending processing explanations which later we added more because of that. |
| Support for PNG, JPG, and PDF | ✅ *Partially* | PNG and JPG formats are fully supported. PDF support was planned but not implemented. |
| Simultaneous processing of ≥5 documents | ✅ | The backend supports asynchronous processing and batch uploads in separate requests. |
| Compatibility with Chrome, Firefox, Safari, Edge | ✅ | Cross-browser testing confirmed reliable performance across all major modern browsers. |

# 5. Challenges and Solutions

Throughout the development of the Digi-Ktav system, we encountered several technical and methodological challenges. Addressing these obstacles required iterative experimentation, creative problem-solving, and strategic decision-making grounded in both theoretical research and practical constraints.

**5.1 Technical Challenges Encountered**

1. **Segmentation of Hebrew Handwriting**  
Hebrew handwriting exhibits high variability in character shape, spacing, and alignment. Segmenting text into lines, words, and individual letters was particularly challenging due to overlapping strokes and inconsistent baselines.

2. **Character Ambiguity and Visual Similarity**Characters such as ו/י/ן ף/ץ ,and ר/ד/כ are often written in similar forms. Even with high-quality preprocessing, these similarities led to frequent misclassifications, especially in early CNN models.

3. **Model Overfitting and Generalization**The CNN model rapidly overfit the training data due to limited variability and training epochs. The ViT trained from scratch showed better generalization, but fluctuated during training and required extensive resources and tuning.

4**. Designing Gemini Post-Correction Evaluation**  
While Gemini proved effective at correcting frequent OCR mistakes, integrating it meaningfully into the pipeline required defining how and when to apply corrections. We had to decide whether Gemini should process raw predictions or structured lines, and how to measure improvement — as the model sometimes introduced unexpected changes.

5**. Limitations of Traditional Segmentation:**We used Horizontal Projection Profile (HPP) for line segmentation. It worked on simple layouts but struggled with multi-column or irregular structures. YOLO could detect text lines or words directly, handling complex pages more robustly and much faster, while our current OCR pipeline can take up to a minute per page. However, YOLO requires a large annotated dataset, which was beyond our scope.[[5]](#rlh66pyfbeaf)

6. **Low Accuracy from** [**TrOCR11**](#TrOCR) **with Full-Line Dataset**We experimented with fine-tuning a TrOCR-style encoder-decoder model using full-line image-text pairs, without segmentation. Despite building a proper dataset (image path + full text), the model performed poorly. We suspect this was due to limited training data (~1000 samples), high sequence complexity (long Hebrew sentences), and a lack of pretraining on right-to-left Hebrew. This highlighted the limitations of applying end-to-end OCR without sufficient data volume or RTL-specific augmentation.[[8]](#xfx79qa1my5s)

**5.2 Solutions and Approaches**

1**. Custom Segmentation Pipeline**We designed a three-stage segmentation system (lines → words → characters) with heuristic filtering and projection-based techniques. This allowed us to isolate characters even in cases of connected writing or inconsistent spacing.

2. Ensemble Evaluation and Fine-Tuning Strategy  
To address misclassification, we fine-tuned a pretrained ViT model that leveraged large-scale features learned on general image datasets. This significantly improved precision and recall across most characters.

3. Data Inspection and Manual Label Review  
To resolve test inconsistencies, we manually reviewed a subset of the dataset, corrected mislabeled samples, and ensured balanced class representation in validation and test sets.

4. Postprocessing Layer with Gemini  
We added a postprocessing module that reformats OCR predictions into contextual input for Gemini. This improved readability, grammar correction, and resolved logical inconsistencies in the recognized output.

5. Hybrid Approach:  
To stay efficient without manual labeling, we combined Horizontal Projection Profile with rule-based post-processing: removing lines, aligning splits to whitespace, and refining boundaries. This gave us a practical solution with acceptable accuracy and speed, without the overhead of training a detection model.

6. Deferred TrOCR Training – Focus Shifted to Segmentation + Classification:  
Due to TrOCR’s poor performance and high resource demands, we chose to pivot to a segmentation-based OCR pipeline using ViT for character classification. While TrOCR could still be a powerful direction with better data, it was beyond the project’s scope at this stage.[[8]](#xfx79qa1my5s)

# 6.Summary and Conclusions

**Meeting Project Criteria**

In Phase A of the project, we defined ambitious yet measurable goals for Digi-Ktav, our Hebrew handwriting OCR system. These included an expected character-level recognition accuracy of at least 85%, support for common image formats (PNG, JPG, PDF), processing times under 10 seconds per document, and a functional, user-friendly interface.

By the end of development, we achieved:

* Final character-level accuracy of 93.0% using a fine-tuned Vision Transformer (ViT), surpassing our original goal.
* Full end-to-end processing (including segmentation, model inference, and Gemini correction) within an average of 40 seconds per image.
* Cross-platform, user friendly web application with support for all major image formats and real-time OCR results.
* Gemini-based post-correction which improved output readability and logical consistency.

One exception was the adaptive learning module, which we had originally intended to implement to personalize recognition for individual handwriting styles. In practice, this turned out to be unnecessary — our model performed well across a wide range of inputs without user-specific fine-tuning. Given the size and quality of our dataset, additional per-user adaptation or dataset expansion wasn’t needed.

**Constraint Management**  
Training Vision Transformers from scratch required a lot of computing power and time, so we chose to fine-tune a pretrained ViT model instead, which saved time and improved accuracy. Our Hebrew dataset had uneven class distribution and similar-looking characters, so we carefully prepared and checked the data across all 27 classes. When deploying the system, we dealt with issues like supporting different image formats, managing asynchronous communication between backend and frontend, and ensuring the system stayed stable. We solved these problems through repeated testing and fixing.

**Considerations in Decision-Making**  
Our development followed key steps:

1. Creating a reliable segmentation pipeline, which was important for accurate character recognition.
2. Starting with CNN models with about 77% accuracy, then training a ViT from scratch 91.9%, and finally choosing a fine-tuned pretrained ViT - 93.0% - as the best model.
3. Adding Gemini post-processing to correct mistakes and improve the final text.
4. Building a responsive web interface with a Python backend and asynchronous API calls.

Due to limited time and team size, we worked on multiple parts simultaneously—like developing segmentation while training models—to move faster.

**6.1 Key Takeaways**

One of the most important findings was the significant performance gain from using a fine-tuned Vision Transformer compared to CNNs, especially for recognizing handwritten Hebrew with high variability. At the same time, we saw that segmentation quality was often the main bottleneck — many recognition errors were caused not by the model itself, but by inaccurate or fragmented character boundaries. Adding Gemini as a post-processing step proved useful for fixing common mistakes and improving the structure of the output.

Another key takeaway is the importance of combining early development with testing during the research phase. In our case, we relied on traditional segmentation methods throughout Phase A and only realized their limitations too late — when we were already in the implementation phase. Had we evaluated and tested our segmentation more thoroughly in the first semester, we could have planned a more robust, learning-based approach like YOLO and had time to prepare a suitable dataset. This would have saved time and allowed for better results in the long run.

Finally, building the full system made it clear that high model accuracy alone isn't enough — practical aspects like processing time, user interface, and system stability are just as important when developing real-world machine learning tools.

**6.2 Future Improvements**

Several areas were identified for continued development:

* **Transition to detection-based recognition**: Replace the current segmentation-dependent pipeline with an object detection model like YOLO to directly detect characters or words. This would remove segmentation as a bottleneck, improve recognition accuracy on complex layouts, and greatly speed up processing.
* **User personalization**: Incorporate a calibration module allowing users to upload handwriting samples for handwritten style-specific fine-tuning.
* **Accessibility enhancements**: Improve accessibility for visually impaired users.
* **Interactive feedback loop**: Create a feedback mechanism where user corrections are stored and used to improve future recognition through semi-supervised learning.
* **Dataset expansion**: Extend the training dataset to include more handwriting styles (e.g., children, elderly, different writing instruments) for better generalization.
* **User-consented dataset expansion:** To support sequence-level modeling and improve generalization, we propose developing a mechanism that allows users to optionally consent to sharing their processed handwriting data. This would involve storing uploaded images and their corrected OCR outputs in an anonymized, secure database. Over time, this growing dataset could be used to retrain or fine-tune models, particularly for diverse handwriting styles and natural sentence structures.

**6.3 Summary**

Digi-Ktav successfully met and exceeded the core objectives set forth in Phase A, delivering a high-performing, user-accessible Hebrew handwriting recognition system. Through a structured and iterative development process—spanning segmentation, model experimentation, post-processing, and interface design—we achieved a final character-level accuracy of 93% using a fine-tuned Vision Transformer. While we faced challenges related to resource constraints, segmentation complexity, and system integration, strategic decisions such as leveraging transfer learning and language model correction helped overcome these limitations. Our experience highlighted the importance of precise segmentation, the advantages of transfer learning for Hebrew OCR, and the effectiveness of combining vision-based recognition with language-based correction to improve accuracy and clarity. Looking ahead, the project opens pathways for future improvements including replacing the current segmentation-based pipeline with a YOLO-style detection model to improve both speed and accuracy, user personalization, sequence-level modeling, and user-consented dataset expansion to sustain long-term accuracy and scalability. Digi-Ktav demonstrates a strong foundation for advancing handwritten Hebrew OCR and serves as a promising platform for continued innovation in low-resource language digitization.

# 7. References

1. “An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale”  [arXiv:2010.11929](https://arxiv.org/abs/2010.11929)**[cs.CV]**
2. “Morphology-Image-Processing-Point-Detection-Image-Segmentation-and-Thresholding-Hough-Transform” <https://github.com/kasbekarameya/Morphology-Image-Processing-Point-Detection-Image-Segmentation-and-Thresholding-Hough-Transform>
3. Datasets used: [HHD](https://cris.bgu.ac.il/en/publications/the-hhd-dataset-2) , [moranzargari Handwriting-detection-recognition](https://github.com/moranzargari/Handwriting-detection-recognition/blob/master/Letters%20Dataset%20Collection/dataset.zip) , [SofiaNaer Hebrew-Handwriting](https://github.com/SofiaNaer/Hebrew-Handwriting/tree/master/dataset)
4. Wolf, T., et al. (2020). "Transformers: State-of-the-Art Natural Language Processing." EMNLP 2020.<https://huggingface.co>
5. Redmon, J., Divvala, S., Girshick, R., & Farhadi, A. (2016). "You Only Look Once: Unified, Real-Time Object Detection." In CVPR.
6. Google DeepMind. "Gemini: Google’s Multimodal LLM Family." <https://arxiv.org/abs/2312.11805>
7. Jordan Walke et al. "React – A JavaScript Library for Building User Interfaces."<https://reactjs.org>
8. Li, M., Cui, Y., Huang, Y., Zhang, C., Yu, L., Zhang, Y., Liu, Z., Hu, H., Cui, L., & Zhang, T. (2021). TrOCR: Transformer-based Optical Character Recognition with Pre-trained Models. *arXiv preprint arXiv:2109.10282*.<https://arxiv.org/abs/2109.10282>

# 8. User guide

A short video that showcases the site for users- <https://youtu.be/HmeZhrD8rA4?feature=shared>

**8.1 System Overview**

**Digi-Ktav** is a web-based tool that allows users to convert scanned Hebrew handwriting into editable digital text using artificial intelligence. The system works on both **PC and mobile devices**.

The standard user workflow includes four main steps:

1. **Uploading a handwritten Hebrew document**.
2. **Viewing and correcting** the automatically recognized text.
3. **Applying optional AI-based enhancements**, such as spelling correction, summarization, English translation, or academic rephrasing.
4. **Saving or downloading** the result.

You can use the system as:

* **Guest mode -** Full access to features, but saving is disabled
* **Logged-in mode**: Enables saving, adding descriptions, and accessing "My Documents"

The interface includes **light/dark modes** and a **sidebar** with guides and tech info.

A screenshot of a computer

AI-generated content may be incorrect.A screenshot of a computer

AI-generated content may be incorrect.

Figure 10: Sidebar Tabs

**8.2 Main Workflow**

**Step 1: Uploading a Document**

The first step in using the system is uploading an image of a handwritten Hebrew document. This is done directly from the homepage.

**Accepted formats:** JPG or PNG, up to 10MB. PDF not supported.  
**To upload:**

1. Go to the homepage.
2. Upload your image by dragging it into the dashed box or clicking "Browse Files."
3. The file name and size will appear below.
4. Click “Process Document” to start OCR, or “Remove” to cancel.  
   You’ll be redirected to the “Review & Correct” screen after processing ends.

**Photo Quality Guidelines**

For best results, follow these **3 simple rules**:

1. **Full A4 Page** – Photograph the entire A4 paper, don't crop
2. **Normal Writing** – Use clear handwriting with proper spacing
3. **Good Lighting** – Take photo from directly above, no shadows

Full instructions available inHelp & Documentation > Photo Instructions on the website.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 11: Upload Document (main) Page

**Step 2: Review & Edit**  
After clicking “Process Document,” the system processes the image and takes you to the “Review & Correct” screen.

* The handwritten image appears on the left.
* Editable recognized Hebrew text appears on the right.

You can freely correct any errors—letters, spelling, spacing, punctuation—before continuing.  
Once done, proceed to **Text Enhancements**, **Download**, or **Save**.

\

Figure 12: Review & Correct Page

**Step 3: Apply Enhancements**  
After reviewing the text, click “Processing Options” to open a window with enhancement checkboxes.  
You can choose one or more of the following:

* **Spelling Correction** – Fix common Hebrew spelling errors
* **Text Restructuring** – Improve formatting and readability
* **Text Summarization** – Create a short summary of the content
* **Hebrew to English Translation** – Translate using adaptive styles

Click “Apply Enhancements” to process, or “Cancel” to skip.  
Enhanced results will appear in a new section, ready to download or save.

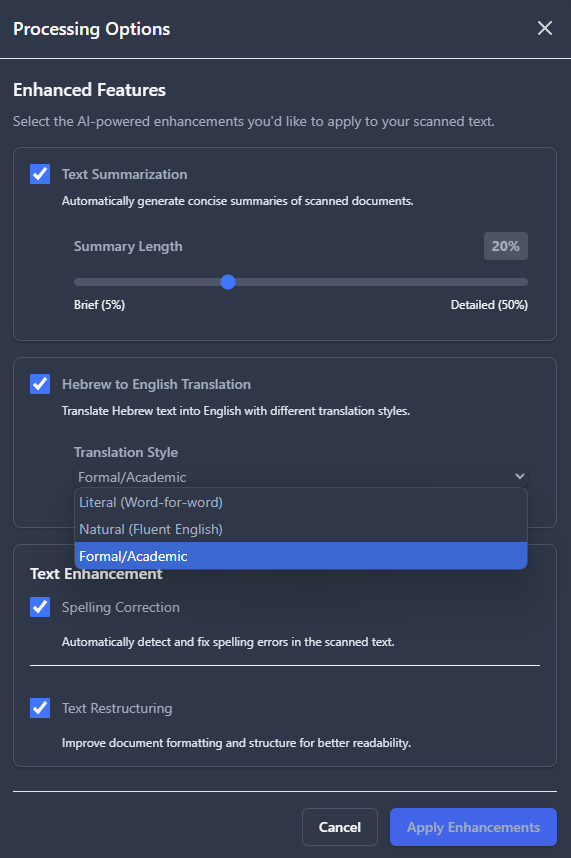


Figure 13: Processing Options / Enhancement Features

**Step 4: Review & Finalize**

After selecting enhancements and clicking **“Apply Enhancements,”** the results appear in the **Enhancement Results** screen.

You can now [4]:

* **View the Final Result**  – A combined version with all selected enhancements applied.
* **Check Individual Sections** – See each enhancement’s effect separately.
* **Compare with Your Edited Text** – View the original text you submitted before enhancement

Also available [1]:

* **Download** – Export the final enhanced text
* **Save to My Documents** – (Logged-in users only) Save the result to your account
* **Upload New File** – Start a new upload
* **Back to Text Editor** [5] – Return to manual editing if further changes are needed

🛈 Applied enhancements are displayed as colored tags “Spelling Correction,” “Text Restructuring” for clarity. [2]

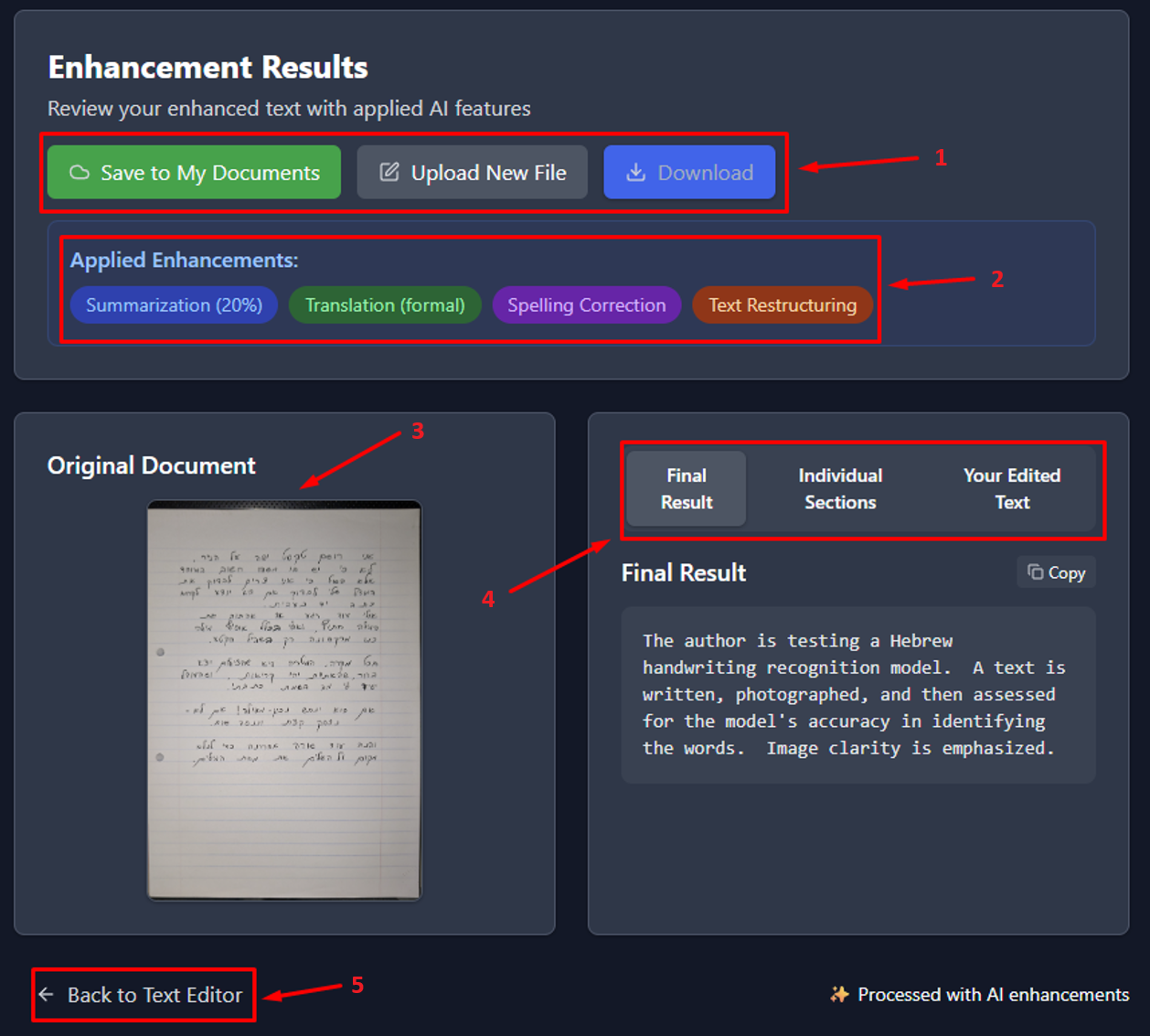


Figure 14: Enhancement Results Page

**8.3 My Documents**

The **My Documents** page is available to logged-in users and displays all the documents you've processed and saved.

For each saved document, you can:

* **View** – Reopen the document with all associated data and enhancements
* **Download** – Export the final version
* **Delete** – Permanently remove the file
* **Edit** – Rename the document title
* See **creation and update dates**
* Identify applied enhancements via colored tags
* View the original image

This page makes it easy to organize and revisit previous work, including school assignments, personal notes, or scanned archives

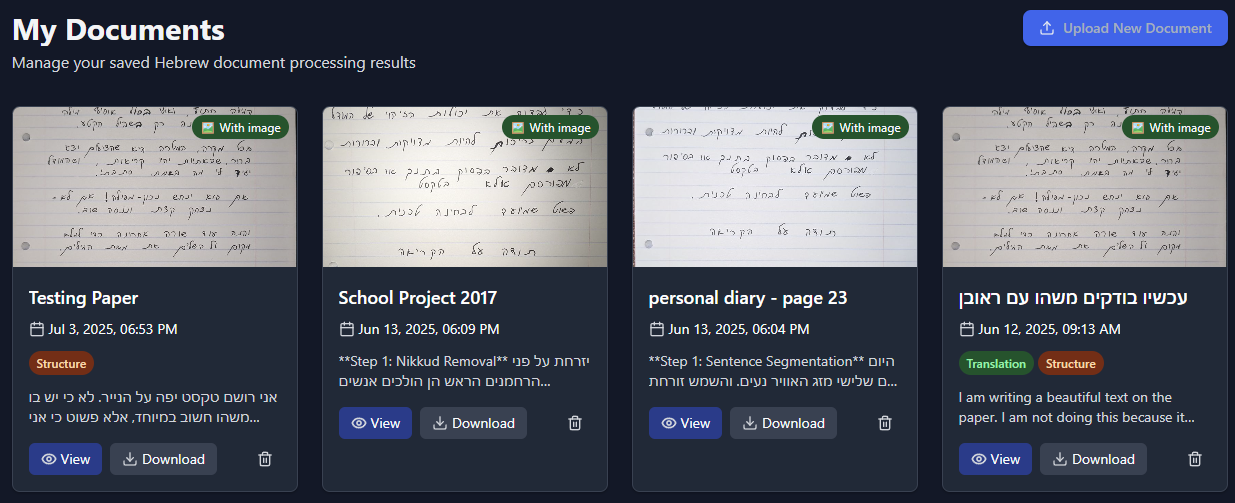


Figure 15: My Documents Page With Examples of Processed Documents

**8.4 Processing Options Page**  
The Processing Options Page is a **standalone** settings screen where users can explore and configure available enhancement features. It is not tied to the recognition pipeline and can be accessed independently at any time — even without uploading a document

Each enhancement option is listed with a short explanation. Users can:

* Preview all available processing features
* **Permanently select** default options to appear pre-checked during future document processing
* Customize the behavior of the enhancement step to match their personal workflow

Preferences are saved per user (when logged in) and applied automatically during processing, making repeated use faster and easier.

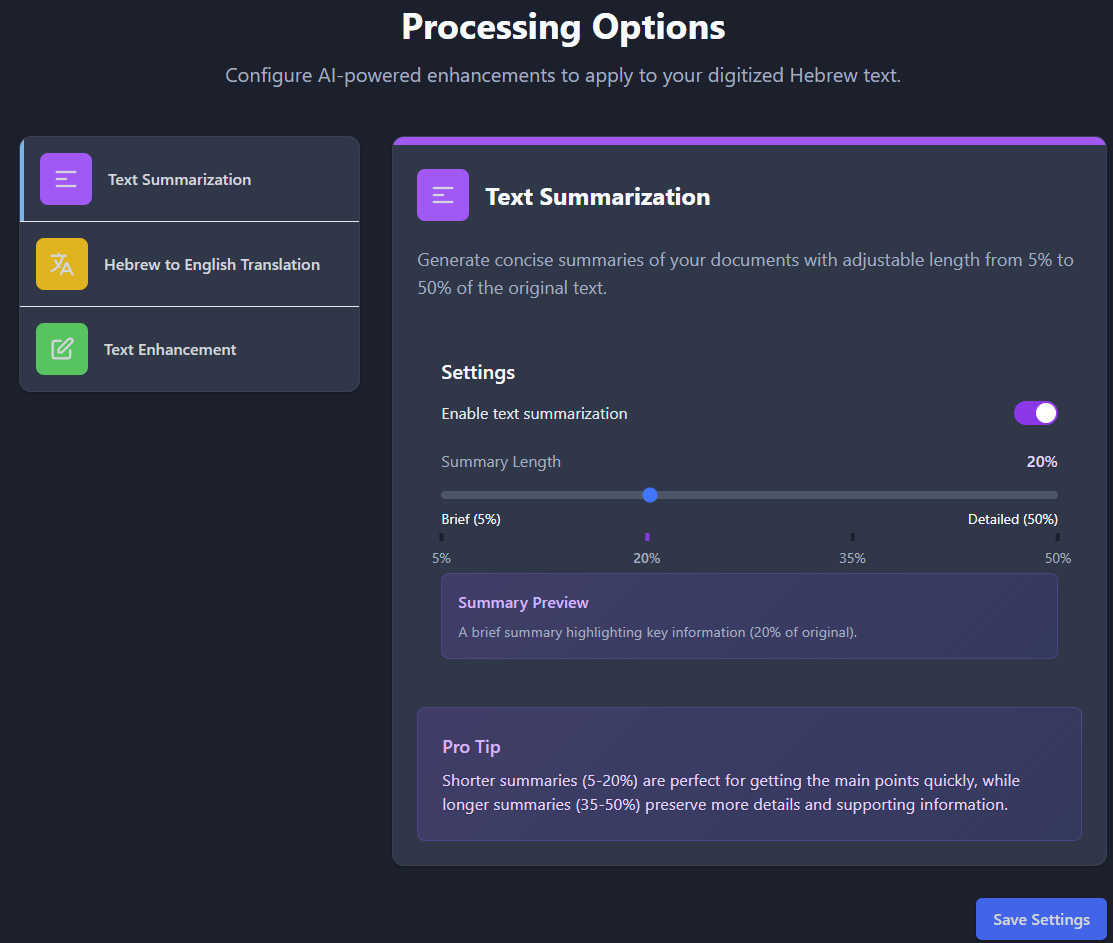


Figure 16: Processing Options / Enhancement Features Page

# 9. Maintenance Guide -DIGIKTAV

**9.1 Hardware and Software Requirements**

To run Digi-Ktav locally or on a server, the following software components are recommended:

* OS: Linux (Ubuntu 20.04+) or Windows 10+
* Python 3.10+
* Node.js (for frontend development)
* Git
* Modern web browser (Chrome, Firefox, Edge)

**9.2 Installation Instructions**

1. Clone the repo:

git clone https://github.com/LiadTssf/FinalProject-hebrew-handwriting-recognition.git

cd FinalProject-hebrew-handwriting-recognition

1. Backend setup:

pip install -r backend/requirements.txt

1. Frontend setup:

cd src

npm install

1. Start backend server:

cd ..

uvicorn backend.main:app --reload

1. Start frontend:

cd src

npm run dev

**9.3 How to Extend or Modify the System**

**Add a New OCR Model:**

* **Place model files** inside the backend/ directory. For example: backend/my\_new\_model/ with model weights, config.
* **Update model path** in backend/ocr\_pipeline.py:
  + VIT\_MODEL\_PATH = "./my\_new\_model"
* **Adjust preprocessing** in load\_models() to match the model’s expected input size and normalization.
* **Rebuild and redeploy** if you use Docker or Google Cloud (see 9.5).

**To improve segmentation:**

* **Edit segmentation logic** in ocr\_pipeline.py, particularly the segment\_image\_to\_lines() and segment\_line\_to\_items() functions.
* **Tune parameters** like spacing thresholds or character merge rules.
* **Optional**: Replace the projection/contour-based method with a learned model like **CRAFT**, **MMOCR**, or **PaddleOCR**.

**To expand the Gemini post-processing logic:**

* **Modify the prompt** in text\_enhancement.py or ocr\_pipeline.py under the function correct\_text\_gemini().
* **Add** an API route in main.py
* **Create** prompt logic per feature in text\_enhancement.py

**To personalize handwriting styles:**

* Add a calibration flow in the frontend UI.
* Store user-specific samples and fine-tune models via a training service.

**To support additional formats (e.g., DOCX, TIFF):**

* Update the upload handler and file-type parsing logic in both the frontend and backend.

**9.4 Rebuild and Redeploy Guide (Docker & Google Cloud)**

**A. Rebuild & Run Locally with Docker**

**Step 1:** Build the Docker image

Open terminal in the backend/ folder and run:

* docker build -t digiktav-local .

**Step 2:** Run the container

* docker run -p 8000:8000 -e GEMINI\_API\_KEY="your\_api\_key" digiktav-local

**B. Deploy to Google Cloud Run**

**Step 1**: Submit build to Artifact Registry

Run this from the root project folder:

* gcloud builds submit --tag me-west1-docker.pkg.dev/digi-ktav-ocr-project/digiktav-repo/digiktav-backend:latest .

**Step 2**: Deploy to Cloud Run:

gcloud run deploy digiktav-backend-service `

--image me-west1-docker.pkg.dev/digi-ktav-ocr-project/digiktav-repo/digiktav-backend:latest `

--platform managed `

--region me-west1 `

--allow-unauthenticated `

--port 8000 `

--cpu 1 `

--memory 2Gi `

--min-instances 0 `

--set-env-vars GEMINI\_API\_KEY="your\_api\_key" `

--timeout=900

**When Should I Rebuild & Redeploy?**

After changing:

* ocr\_pipeline.py
* text\_enhancement.py
* main.py
* Model files
* requirements.txt
* Dockerfile

**9.5 Package and Architecture Overview**

**Backend Packages:**

* uvicorn, fastapi - API server
* torch, transformers, Pillow - Model loading and inference
* opencv-python - Image preprocessing
* google-generativeai- Post-processing with Gemini API

**Frontend Packages:**

* react, tailwindcss- UI rendering and styling
* axios- Client-server communication

**High-Level Structure:**

FinalProject-hebrew-handwriting-recognition/ // Main project folder

├── backend/ // Backend logic and OCR pipeline

│ ├── main.py // FastAPI entry point

│ ├── ocr\_pipeline.py // Segmentation and recognition logic

│ ├── text\_enhancement.py // Gemini post-processing logic

│ ├── requirements.txt // Backend dependencies

│ └── vit-hebrew-final/ // Pretrained ViT model weights

├── src/ // Frontend - React app

│ ├── components/ // Reusable UI components

│ ├── pages/ // Main UI pages

│ ├── services/ // API utilities

│ │ ├── enhancementService.js // Sends enhancement requests to backend

│ │ └── firebaseService.js // Manages document saving and Firebase auth

│ └── main.jsx // Entry point for the React app

├── public/ // Static frontend assets

├── index.html // HTML template for the React app

├── vite.config.js // Frontend build configuration

├── package.json // Frontend dependencies and scripts

└── README.md // Project instructions and documentation

# 10. VIT Model Installation Instructions

1. Clone the repo:

git clone https://github.com/LiadTssf/vit-hebrew-character-ocr-DIGIKTAV.git

cd vit-hebrew-character-ocr-DIGIKTAV

1. (Optional) Create a Python virtual environment:

python -m venv venv

source venv/bin/activate # or .\venv\Scripts\activate on Windows

1. Install dependencies:

pip install -r requirements.txt

1. Run inference/demo script:

python predict.py # or perdict\_ViT.py / ViT.py as needed

1. (Optional) Retrain/fine-tune:
   * Place dataset folder in dataset fore cnn hebrew letters.v1i.folder/
   * Adjust parameters in ViT.py or ViT2.py
   * Execute:

python ViT.py # or ViT2.py

**10.1 Recommended System Specifications**

**Operating System**:

* Ubuntu 20.04+ or Windows 10/11 (Linux preferred for better GPU support and performance)

**CPU**:

* Intel Core i7 (10th gen or later) or AMD Ryzen 7 (5000 series or later)

**RAM**:

* Minimum: 16 GB
* Recommended: 32 GB (for smoother training and multitasking)

**GPU (Critical for ViT Training)**:

* Minimum: NVIDIA RTX 3060 with 12GB VRAM
* Recommended: NVIDIA RTX 3080 / 4070 / 4090 or equivalent (with at least 12–24GB VRAM)
* Note: Ensure CUDA 11.x or newer is compatible with the GPU driver and PyTorch version

**10.2 Extending & Modifying the System**

**Adding New Models:**

* Add model weights to the repo root or create a models/ folder.
* Update predict.py or training scripts to load the new model.
* Adjust preprocessing (image size, normalization) in ViT.py/ocr scripts.

**Segmentation:**

Works only on letters and segmentation is based on <https://github.com/LiadTssf/FinalProject-hebrew-handwriting-recognition/tree/main> like explained in its Maintenance Guide

**predict:**

* To check the models after train you can run perdict\_ViT.py/ predict.py make sure to load the right model name you trained

**Training Tweaks:**

* Edit augmentation, learning rate, epochs directly in ViT.py or ViT2.py.
* Monitor logs in logs/ and review performance JSONs or CSVs.

**10.3 Architecture & Package Overview**

**Project structure highlights:**

vit-hebrew-character-ocr-DIGIKTAV/

├── ViT.py # Training from scratch

├── ViT2.py # Fine-tuning variant

├── predict.py

├── perdict\_ViT.py # Inference script

├── dataset fore cnn .../

├── predictions\_log/ # Inference outputs

├── vit-hebrew-final/ # Final weights directory

├── confusion matrix PNGs

└── wrong\_predictions.csv

**Dependencies:**  
See requirements.txt (includes):

* torch, transformers
* Pillow, opencv-python
* roboflow (if used), numpy, etc.

# Technical Terms

1. **OpenCV**: A library for image processing, used to clean and enhance images of handwritten text for better recognition.
2. **NumPy**: A core Python library for numerical operations, used to process image data in the form of pixel arrays.
3. **PyTorch**: An open-source machine learning framework known for flexibility and speed, used here to train deep learning models for character recognition.
4. **TorchVision**: A package in PyTorch providing tools and datasets for computer vision tasks.
5. **FastAPI**: A modern Python framework for building APIs quickly and efficiently. It powers the backend of the system.
6. **Axios**: A JavaScript library used for making HTTP requests between the frontend and backend.
7. **Gemini API (Google)**: A powerful AI model that processes and improves text by fixing grammar, punctuation, and spelling. Used here for OCR output correction and enhancements.
8. **CNN (Convolutional Neural Network)**: A type of deep learning model commonly used for image-related tasks. Processes images with local filters to detect shapes and patterns.
9. **ViT (Vision Transformer)**: A modern image classification model that treats an image like a sequence of patches (like words in text), using self-attention to learn relationships between parts of the image.
10. **YOLO (You Only Look Once)**: A real-time object detection model that identifies and locates multiple objects in a single pass through the image. Proposed as a future replacement for the current segmentation system.
11. **TrOCR**: A Transformer-based OCR system designed to read full lines of text without segmentation. It requires large datasets and struggles with Hebrew without special training.
12. **Horizontal Projection Profile (****HPP)**: A segmentation method that counts the number of black pixels along horizontal lines to detect where lines of text are located.
13. **Morphological Filtering**: Image processing operations that clean up shapes, reduce noise, or fill gaps—useful for identifying and isolating letters in handwriting.
14. **Contour Detection**: An image analysis technique that finds outlines or borders of shapes—used here to locate handwritten characters.
15. **Data Augmentation**: A process of artificially increasing the training dataset size by applying transformations (like rotation or noise) to the original images, helping the model generalize better.
16. **Vision Transformer Patch**: Each image is broken into small square patches (like 16x16 pixels), then treated as input tokens to a Transformer.
17. **Cross-Entropy Loss**: A function used during training to measure how far off a model’s predictions are from the true labels.
18. **Adam Optimizer**: A commonly used method for updating model weights during training. It combines the advantages of other optimizers for faster convergence.
19. **Base64-Encoded Image**: A way of representing image data as text so it can be sent through APIs or stored in databases.
20. **EXIF Rotation Metadata**: Information embedded in image files that can cause images to appear rotated; removed during preprocessing for consistency.
21. **Salt-and-Pepper Noise**: A type of image distortion that adds random black and white pixels; often simulated during training for robustness.