

# Project Report: A Browser-Based System for Real-Time Handwritten Digit Recognition

**Author:** Md Mahbubul Haque Ripon **Date:** 03 november **Course/Context:** Independent Portfolio  
Project - Machine Learning Engineering

## Abstract

This report details the design, methodology, and analysis of a self-contained, browser-based application for real-time handwritten digit recognition. The primary objective was to implement and deploy a complete machine learning pipeline—from model definition and in-browser training to interactive prediction—using purely client-side technologies. The system leverages TensorFlow.js to construct, train, and execute a Convolutional Neural Network (CNN) that classifies digits (0-9) drawn by a user on an HTML canvas. The application successfully demonstrates the viability of on-device machine learning, achieving high validation accuracy (approx. 98%) after training on a subset of the MNIST dataset. The project serves as a practical demonstration of spatial feature extraction via CNNs and the accessibility of modern web-based ML frameworks.

## 1. Introduction

The task of handwritten digit recognition is a foundational problem in the field of computer vision and machine learning. Its solution has practical applications in areas such as automated postal code sorting, bank check processing, and form digitization. The MNIST dataset, comprising 70,000 28x28 grayscale images of handwritten digits, is widely regarded as the "Hello, World!" of the domain, providing a standardized benchmark for model performance.

Traditionally, machine learning models are trained on powerful servers and accessed via an API. This project challenges that paradigm by exploring the capabilities of modern web browsers as platforms for machine learning. The primary aim is to **develop an end-to-end, interactive system where a user can train a model and receive predictions without any server-side computation.**

This report outlines the system's architecture, the technological stack employed, the deep learning methodology, and an analysis of its performance and interactive capabilities.

## 2. System Design and Technology Stack

The application is architected as a single-page application (SPA), with all assets, logic, and computation handled by the client's web browser.

### 2.1. Technology Stack

- **Front-End Structure (HTML5):** Provides the semantic structure for the application, including the primary 280x280 `canvas` for user input, a hidden 28x28 `canvas` for preprocessing, and DOM elements for controls and results.
- **Styling (Tailwind CSS):** A utility-first CSS framework used to rapidly build a responsive and modern user interface.
- **Core Logic (JavaScript - ES6+):** Manages all application state, UI event handling (e.g., button clicks, canvas drawing), and coordinates the machine learning pipeline.

- **Machine Learning Framework (TensorFlow.js):** The core technology for this project. TensorFlow.js is a Google-developed library for training and deploying ML models in the browser and on Node.js. It is used here for:
  - **Model Definition:** Building the CNN layer-by-layer.
  - **In-Browser Training:** Running the `model.fit()` training loop directly on the client's machine, leveraging their GPU via WebGL for acceleration.
  - **Prediction:** Executing `model.predict()` on user-generated data.
- **Data (MNIST):** A hosted helper script ( `data.js` ) is used to load a pre-partitioned and shuffled subset of the MNIST dataset.

## 2.2. Application Flow

1. **Initialization:** The UI is rendered, event listeners are attached, and the probability chart is dynamically generated.
2. **Training Trigger:** The user initiates the "Load Data & Train Model" button.
3. **Data Loading:** The `MnistData` class asynchronously fetches 8,000 training samples and 2,000 test samples.
4. **Model Training:** The CNN model is compiled and trained on the 8,000 samples, validating against the 2,000 test samples. UI status is updated live via the `onEpochEnd` callback.
5. **Ready State:** Once training is complete, the "Predict" button is enabled.
6. **User Input:** The user draws a digit on the 280x280 canvas.
7. **Prediction Trigger:** The user clicks "Predict."
8. **Preprocessing:** The 280x280 image is downsampled, converted to a grayscale tensor, and normalized.
9. **Inference:** The processed tensor is fed into the trained model.
10. **Post-processing:** The output `softmax` vector is analyzed to find the highest-probability digit and update the results UI.
11. **Clear:** The user can clear the canvas to repeat the process.

## 3. Methodology

The core of the project is the machine learning methodology, which is divided into data preprocessing, model architecture, and the training process.

### 3.1. Data Preprocessing

Two distinct preprocessing pipelines are employed:

1. **Training/Test Data:** The `MnistData` library provides data as flat 784-element arrays (28\*28). These are reshaped by TensorFlow.js into 4D tensors of shape `[batch_size, 28, 28, 1]`, which is the required input shape for a 2D convolutional layer. The labels are provided in a one-hot encoded format (e.g., '3' is `[0, 0, 0, 1, 0, 0, 0, 0, 0, 0]`).
2. **User Input (Canvas) Data:** This is a critical engineering step. The raw 280x280 pixel canvas with a white background and black stroke must be transformed to match the MNIST format (28x28, black background, white digit, normalized 0-1).

- **Downsampling:** The 280x280 canvas is drawn onto the hidden 28x28 canvas using `hiddenCtx.drawImage(canvas, 0, 0, 28, 28)`. This performs a bilinear interpolation, effectively anti-aliasing the result.
- **Pixel Extraction:** `hiddenCtx.getImageData(0, 0, 28, 28)` retrieves a `Uint8ClampedArray` of RGBA values.
- **Grayscale Conversion & Inversion:** The MNIST data is white-on-black (pixel value 255 is the digit). The canvas drawing is black-on-white. The `alpha` channel (`imageData.data[i * 4 + 3]`) is used as it correctly represents the "ink" density (0 for white, 255 for black).
- **Normalization:** The alpha value is divided by 255.0 to normalize the data into a `[0, 1]` float range.
- **Tensor Reshaping:** The resulting 784-element `Float32Array` is wrapped in a tensor and reshaped to `[1, 28, 28, 1]`, representing a single-image batch.

### 3.2. Model Architecture: Convolutional Neural Network (CNN)

A sequential CNN is defined, as this architecture is specialized for learning spatial hierarchies from grid-like data (i.e., images).

- **Layer 1 (Conv2D):**
  - `inputShape` : `[28, 28, 1]`
  - `filters` : 8
  - `kernelSize` : 5x5
  - `activation` : `relu` (Rectified Linear Unit)
  - *Purpose:* Detects low-level features (e.g., edges, curves). `ReLU` introduces non-linearity.
- **Layer 2 (MaxPooling2D):**
  - `poolSize` : 2x2
  - *Purpose:* Downsamples the feature maps, reducing computational load and providing basic translational invariance.
- **Layer 3 (Conv2D):**
  - `filters` : 16
  - `kernelSize` : 5x5
  - `activation` : `relu`
  - *Purpose:* Detects more complex features by combining the low-level features from the previous layer.
- **Layer 4 (MaxPooling2D):**
  - `poolSize` : 2x2
  - *Purpose:* Further downsampling.
- **Layer 5 (Flatten):**
  - *Purpose:* Converts the final 2D feature maps into a 1D vector, preparing the data for the final classification layer.
- **Layer 6 (Dense - Output Layer):**

- `units : 10` (one for each digit class)
- `activation : softmax`
- **Purpose:** A fully connected layer that performs the final classification. `Softmax` converts the layer's raw output (logits) into a probability distribution, where the sum of all 10 unit activations is 1.0.

### 3.3. Model Training & Compilation

- **Compiler Configuration:**
  - `optimizer : tf.train.adam()` : The Adam optimizer is chosen for its adaptive learning rate and efficiency, making it well-suited for a wide range of problems.
  - `loss : categoricalCrossentropy` : This is the standard loss function for multi-class classification problems where labels are one-hot encoded.
  - `metrics : ['accuracy']` : The model's performance is monitored using the accuracy metric.
- **Training Process ( `model.fit()` ):**
  - **Dataset:** `TRAIN_DATA_SIZE` (8,000) samples.
  - **Validation:** `TEST_DATA_SIZE` (2,000) samples.
  - **Epochs:** 10.
  - **Batch Size:** 512.
  - **Rationale:** A smaller-than-standard dataset (8k vs 60k) and a larger batch size (512) are chosen as a trade-off. This significantly accelerates in-browser training time while still being sufficient to achieve high accuracy on this simple task.

## 4. Analysis and Results

- **Training Performance:** The model consistently achieves a validation accuracy ( `val_acc` ) of ~97-98.5% after 10 epochs. The training process completes in approximately 30-60 seconds on a modern consumer laptop, confirming the viability of in-browser training.
- **Predictive Analysis:** The application provides two forms of output for analysis:
  1. **Quantitative (Highest-Probability Guess):** The digit with the highest activation in the `softmax` output layer is presented as the final prediction. In testing, this proves highly accurate for clearly drawn, centered digits.
  2. **Qualitative (Probability Distribution):** The bar chart displays the full `softmax` output, showing the model's "confidence" for each class. This is a powerful analytical tool. For example, a poorly drawn "4" might show a 60% probability for "4" and a 35% probability for "9," offering insight into the model's confusion.
- **Error Sources:** The primary sources of prediction error are (a) poor user drawings (e.g., digits are not connected) and (b) variance from the MNIST training data (e.g., digits drawn off-center or with a different stroke thickness than the 20px line used).

## 5. Discussion and Future Work

This project successfully demonstrates a complete, on-device machine learning workflow. The primary achievement is not just the high accuracy, but the interactive and educational nature of the tool. A user

can observe the training process and then immediately test the resulting model, providing a tangible link between "training" and "inference."

### 5.1. Limitations

1. **Training Data:** Using only 8,000 of 60,000 available training samples limits the model's ability to generalize compared to a model trained on the full dataset.
2. **Preprocessing:** The current canvas preprocessing is simple (direct downsampling). It does not, for example, calculate the drawn digit's center of mass to re-center it within the 28x28 grid, which is a common technique for improving MNIST accuracy.
3. **Ephemeral Model:** The model is trained from scratch on every page load and exists only in memory.

### 5.2. Future Work

1. **Pre-trained Model:** The application could be modified to load a pre-trained model (e.g., from `tfjs-models` or one trained offline). This would eliminate the in-browser training step, providing instant predictive capabilities.
2. **Advanced Preprocessing:** Implement a center-of-mass calculation to find the bounding box of the drawn digit, resize it, and center it in the 28x28 tensor to better match the MNIST data format.
3. **Model Persistence:** Use `model.save()` with `indexedDB` to allow a user to save their trained model in the browser, persisting it across sessions.

## 6. Conclusion

This project serves as a comprehensive case study in client-side machine learning. By successfully implementing a handwritten digit recognizer using TensorFlow.js, it demonstrates that complex computational tasks, including deep learning model training, are now feasible within the standard web browser. The resulting application is not only a functional classification tool but also a powerful educational platform for visualizing the end-to-end machine learning pipeline in an interactive environment.