Using a Machine Learning Model for Handwritten Digit Recognition in GeoGebra

Mitsushi Fujimoto

University of Teacher Education Fukuoka,
1-1 Akamabunkyo-machi, Munakata 811-4192, Japan
fujimoto@fukuoka-edu.ac.jp
https://staff.fukuoka-edu.ac.jp/fujimoto/

Abstract. We describe methods for recognizing handwritten digits in GeoGebra using a PyTorch-based machine learning model. We present two inference approaches: server-side inference with FastAPI and browser-based inference with ONNX Runtime Web. A performance comparison across three configurations—FastAPI, ONNX with WebAssembly, and ONNX with WebGPU—showed that ONNX with WebAssembly achieved the fastest inference for the small-scale model used. As an application, we implemented a feature that adjusts a triangle so that its side ratios match handwritten digit inputs.

Keywords: GeoGebra · Handwritten digit recognition · ONNX · Mathematics education.

1 Introduction

Handwritten input interfaces play an important role in the educational use of mathematical software. The dynamic geometry software GeoGebra [1] provides a Freehand Shape feature that converts hand-drawn strokes into geometric objects such as line segments, triangles, quadrilaterals, and circles, which is effective for construction tasks when learning Euclidean geometry proofs. However, it does not support handwritten input of mathematical expressions. Since figures corresponding to geometric propositions often include numerical values such as side ratios, the addition of a handwritten expression input feature is desirable.

GeoGebra can be extended using JavaScript. By entering JavaScript code in the Scripting tab of an object's Properties Dialog, one can assign scripts to GeoGebra objects. When more complex functionality is needed, it is more practical to embed GeoGebra into a web application and use JavaScript within that context, as this enables integration with other useful web technologies.

We previously proposed a method for building GeoGebra as a web application using React for the user interface and C with WebAssembly for the computational engine [2].

In this paper, building on this approach, we propose methods for using a machine learning model for handwritten digit recognition in GeoGebra. Specifically, we implement two inference approaches—server-side inference using FastAPI and browser-based inference using ONNX Runtime Web—and compare their characteristics and inference speeds. Furthermore, as an application, we implement a feature that adjusts triangle shapes by inputting digits for side ratios via handwriting, demonstrating the potential for mathematics education.

2 Machine Learning Model

The handwritten digit recognition addressed in this paper is a classification task into 10 classes (digits 0–9). The model was implemented using PyTorch [3] and trained on the MNIST dataset [4], which consists of 60,000 training and 10,000 test images of 28×28 pixel 8-bit grayscale handwritten digits.

Since this is a straightforward classification problem, we adopted a fully connected neural network with two hidden layers: an input layer (784 nodes), two hidden layers (128 nodes each, ReLU activation), and an output layer (10 nodes), totaling 118,282 parameters (Fig. 1).

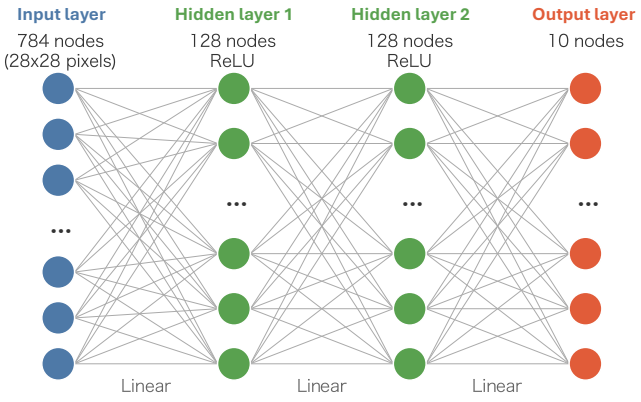

Fig. 1. Fully connected neural network with two hidden layers

Training was performed with batch size 500 for 200 epochs using cross-entropy loss and SGD (learning rate 0.01). The test accuracy was 97.10%. The trained state was saved to a checkpoint file `mnist_net2.pth`.

3 Server-Side Inference

This section describes server-side inference using FastAPI [5], a Python framework for building Web APIs with low latency and asynchronous I/O. FastAPI exposes Python-implemented processing as HTTP endpoints callable from various clients.

3.1 Inference Server with FastAPI

The following code shows the FastAPI instance creation and model loading, both of which are executed at server startup.

Code 1. `mnist/server.py`

```
def load_model() -> Net2:
  ckpt = torch.load("./mnist_net2.pth", map_location="cpu")
  m = Net2(int(ckpt["n_input"]), int(ckpt["n_output"]),
    int(ckpt["n_hidden"]))
  m.load_state_dict(ckpt["state_dict"])
  m.eval()
  return m

@asynccontextmanager
async def lifespan(app: FastAPI):
  global net;  net = load_model();  yield
app = FastAPI(lifespan=lifespan)

@app.post("/predict", response_model=PredictResponse)
def predict(req: PredictRequest) -> PredictResponse:
  x = preprocess(req.pixels)
  with torch.no_grad():
    logits = net(x)
    probs = torch.softmax(logits, dim=1)[0]
    pred = int(torch.argmax(probs).item())
  top2 = topk_from_probs(probs, k=2)
  return PredictResponse(pred=pred, topk=top2)
```

The checkpoint stores the model weights (`state_dict`) and architecture parameters (`n_input`, `n_output`, `n_hidden`). The `load_model()` function creates and restores the `Net2` instance, and is invoked through the `lifespan()` context manager at startup. The endpoint `/predict` preprocesses the input (784 pixels) into a tensor, runs inference with `torch.no_grad()`, applies softmax, and returns the predicted class and top-2 results as JSON. Since FastAPI itself does not listen for requests, the server is launched with the ASGI server Uvicorn [6]: `uvicorn server:app --port 8001`.

3.2 Using the ML Model from GeoGebra (Server-Side)

The following code is executed when the user clicks the **Predict** button in GeoGebra.

Code 2. `src/App.jsx`

```
async function handlePredict() {
  const ggbApplet = window.ggbApplet;
  const pixels = strokeToMnistPixels(ggbApplet);
  // Inference
  try {
    const res = await fetch("http://localhost:8001/predict", {
      method: "POST",
      headers: {"Content-Type":"application/json"},
      body: JSON.stringify({ pixels }) });
    const json = await res.json();
    const pred = json.pred;  const t2 = json.topk;
    const p1 = (t2[0].p * 100).toFixed(1);
    const p2 = (t2[1].p * 100).toFixed(1);
    const msg = `Top-1: ${pred}(${p1}\\%), Top-2: ${t2[1].i}(${p2}\\%)`;
    ggbApplet.evalCommand(`result = FormulaText("${msg}")`);
  } catch (e) { console.error(e); }
}
```

This function uses `strokeToMnistPixels()` to extract pen strokes and generate a 28×28 pixel array (784 elements) in MNIST format, sends it via `fetch()` as a POST request to the server (shown in blue), and displays the recognition result on GeoGebra (Fig. 2).

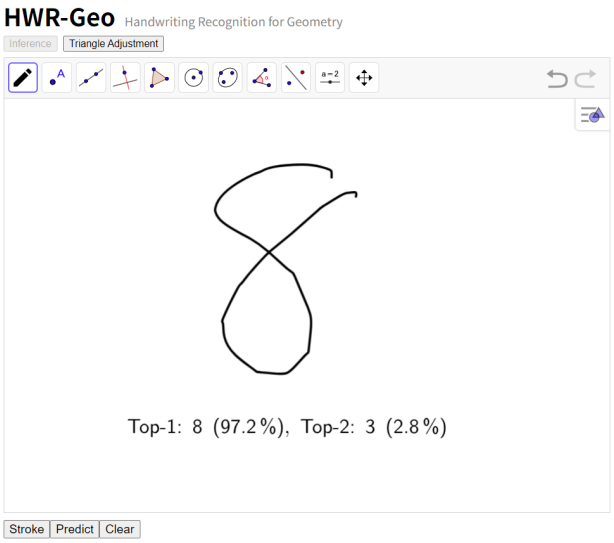

Fig. 2. Handwritten digit recognition result via server-side inference

4 Browser-Based Inference

This section describes the migration from server-side inference to browser-based inference using ONNX Runtime Web, eliminating the need for a server and allowing the application to run entirely as static files.

4.1 Converting a PyTorch Model to ONNX Format

ONNX (Open Neural Network Exchange) [7] is a format designed for exchanging and reusing trained models across different machine learning frameworks such as TensorFlow, Keras, and PyTorch. Models converted to ONNX can be executed for inference on various platforms using ONNX Runtime [8]. For web browsers, ONNX Runtime Web is provided, which uses WebAssembly and other technologies to run inference without a Python environment. This enables a workflow where models are built and trained on large-scale hardware equipped with GPUs, then converted to ONNX and deployed for inference in web browsers.

The following code converts the trained PyTorch model to an ONNX model file `mnist_net2.onnx`.

Code 3. `mnist/export_onnx.py`

```
ckpt = torch.load("./mnist_net2.pth", map_location="cpu",
  weights_only=False)
model = Net2(784, 10, 128)
model.load_state_dict(ckpt["state_dict"])
model.eval()
dummy_input = torch.randn(1, 784)
torch.onnx.export(
  model, dummy_input, "mnist_net2.onnx",
  export_params=True, opset_version=18, do_constant_folding=True,
  input_names=["input"], output_names=["output"], external_data=False,
  dynamic_axes={"input": {0: "batch_size"},"output": {0: "batch_size"}},
)
```

The `torch.onnx.export()` function traces the forward pass using a dummy input of shape 1×784 and records the computation graph in ONNX format. The `input_names` and `output_names` parameters assign tensor names for use during inference.

4.2 Implementing the ONNX Inference Module

The following code loads the ONNX model file `mnist_net2.onnx` and performs inference.

Code 4. `src/onnxInference.js`

```
import * as ort from 'onnxruntime-web';
const MODEL_URL = `${import.meta.env.BASE_URL}mnist_net2.onnx`;
let sessionPromise = null;
export async function initSession() {
  if (!sessionPromise) {
    sessionPromise = ort.InferenceSession.create(MODEL_URL, {
      executionProviders: ['webgpu', 'wasm'],  });
  }
  return sessionPromise;
}
export async function onnxPredict(pixels, k = 2) {
  const session = await initSession();
  const input = preprocess(pixels);
  const tensor = new ort.Tensor('float32', input, [1, input.length]);
  const feeds = { input: tensor };
  const results = await session.run(feeds);
```

```
  const outputTensor = results[session.outputNames[0]];
  const probs = softmax(outputTensor.data);
  const topk = topK(probs, k);
  return { pred: topk[0].i, topk };
}
```

The `initSession()` function loads the model on the first call and caches the session, specifying `['webgpu', 'wasm']` as execution providers (WebGPU if available, otherwise WebAssembly). The `onnxPredict()` function preprocesses the pixels, creates an ONNX tensor of shape $[1, 784]$, runs inference, applies softmax, and returns the top-k predictions. The key `input` in the feeds object must match the name specified in `export_onnx.py`.

4.3 Using the ML Model from GeoGebra (Browser-Based)

The initialization function is modified to preload the ONNX model by calling `initSession()`. Furthermore, the inference call inside `handlePredict()` (Code 2) is changed from `fetch` to `onnxPredict()`:

Code 5. `src/App.jsx` (modified)

```
  // Inference
  try {
    const json = await onnxPredict(pixels, 2);
    const pred = json.pred;  const t2 = json.topk;
    ...
```

With this change (shown in red), the inference runs entirely within the browser.

5 Performance Comparison

We conducted a comparative experiment on inference speed between server-side inference using FastAPI (Sect. 3) and browser-based inference using ONNX Runtime Web (Sect. 4). Since ONNX Runtime Web offers two execution providers—WebAssembly (hereafter WASM) and WebGPU—we compare a total of three configurations:

1. **FastAPI**: Server-side inference implemented in Sect. 3. Measured values include HTTP round-trip time within localhost.
2. **ONNX (WASM)**: Browser-based CPU inference using the WebAssembly backend of ONNX Runtime Web.
3. **ONNX (WebGPU)**: Browser-based GPU inference using the WebGPU backend of ONNX Runtime Web.

We created 100 test samples by handwriting each digit 0–9 ten times in GeoGebra, obtaining 784-element pixel arrays through stroke extraction, 28×28 rasterization, and center-of-mass normalization. These 100 pixel arrays were used as common input for all three configurations. Since all three configurations use the same trained model, there is no difference in recognition accuracy. Note that these 100 handwritten samples were created solely for comparing inference speed; the model accuracy of 97.10% was evaluated on the full MNIST test set (10,000 images) as described in Sect. 2. After one warm-up inference, response times for the 100 samples were measured using JavaScript's `performance.now()` function. The experimental environment is shown in Table 1.

Table 2 shows the results. ONNX (WASM) was remarkably fast at 0.022 ms mean, with most values below the timer resolution of 0.1 ms, because the small model ($784 \to 128 \to 128 \to 10$) completes matrix operations in sub-millisecond time on WASM. ONNX (WebGPU) averaged 4.070 ms. Despite using the GPU, it was slower than WASM because kernel launch and CPU–GPU data transfer overhead are relatively large compared to the computation time of this small

Table 1. Experimental environment

Item	Specification
CPU	AMD Ryzen AI 9 HX 370 5.1GHz (Base clock 2.0GHz)
RAM	96GB (DDR5-5600)
GPU	AMD Radeon 890M
VRAM	32GB (DDR5-5600)
OS	Debian 13.3 (WSL2/Windows 11 25H2)
Web Browser	Google Chrome (Version 145.0.7632.110)
FastAPI Server	Running on localhost:8001 on the same machine

Table 2. Response time by inference method (unit: ms, $n = 100$)

Method	Mean	Median	Min	Max	Std. Dev.
ONNX (WASM)	0.022	0.000	0.0	0.1	0.041
ONNX (WebGPU)	4.070	4.100	1.6	5.5	0.575
FastAPI	4.754	4.150	3.2	48.2	4.420

model. FastAPI averaged 4.754 ms (median 4.150 ms), which is relatively fast even including HTTP communication over localhost, but spikes up to 48.2 ms were observed, indicating that jitter from network communication affects stability.

For small-scale models such as the one used here, ONNX (WASM) is the most suitable choice due to its overwhelmingly fast and stable performance. ONNX (WebGPU) is inferior to WASM for small models due to overhead, but is expected to excel for large-scale models through GPU parallelism. FastAPI showed the largest variance due to network communication jitter.

6 Application: Triangle Adjustment

In the preceding sections, we built a system that recognizes handwritten digits in GeoGebra using an MNIST model. In this section, as an application of this handwritten digit recognition, we implement a feature that adjusts a triangle's shape so that its side ratios match handwritten digit inputs. This illustrates what kinds of functionality become possible by introducing machine learning models into GeoGebra.

6.1 Feature Overview

The usage procedure is as follows (Fig. 3): (1) draw a triangle freehand using GeoGebra's Pen Tool; (2) the handwritten triangle is automatically detected and displayed; (3) enter digits near each side representing the desired length ratio; (4) click the **Adjust** button to trigger digit recognition and adjust the triangle's shape based on the recognized ratios.

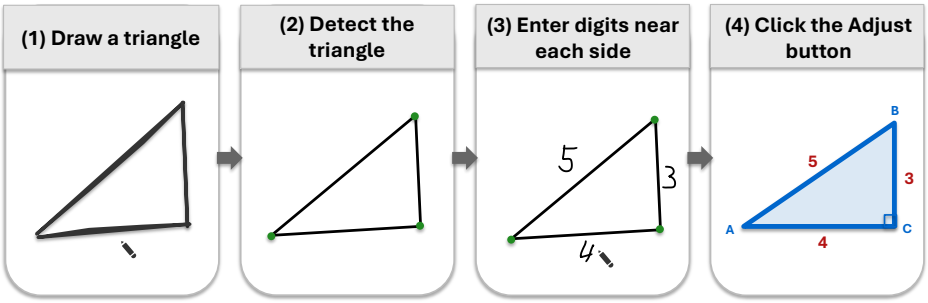

Fig. 3. Operation flow for triangle adjustment

6.2 Processing Steps

The feature consists of five processing steps.

(1) Stroke merging and triangle detection: GeoGebra's Freehand Shape tool can recognize hand-drawn geometric shapes, but it is a UI-level feature not accessible via the JavaScript API. Therefore, triangle detection is implemented independently in our system. Since users may draw a triangle using one, two, or three separate strokes, the endpoints of all sub-strokes are collected and nearby endpoints are clustered by proximity using Union-Find. If exactly three clusters are found, their centroids are taken as the triangle vertices; if more than three, the combination forming the largest triangle area is selected. When fewer than three clusters exist (e.g., when the triangle is drawn in a single continuous stroke), the algorithm falls back to detecting corners from abrupt changes in stroke direction.

(2) Separating side and digit strokes: After the triangle vertices are identified, each sub-stroke is classified as either a side stroke or a digit stroke: if both endpoints of a sub-stroke lie near distinct vertices, it is a side stroke; otherwise, it is treated as a digit candidate.

(3) Multi-digit recognition: Digit strokes whose centroids are close to each other are grouped together. Within each group, strokes are sorted in ascending order of x-coordinate, and the optimal digit segmentation is found via dynamic programming.

(4) Assigning digits to sides: Each recognized digit group is assigned to the nearest side based on the distance between the group's centroid and each side's midpoint.

(5) Triangle adjustment and rendering: First, a target triangle with the specified side ratios is constructed at a canonical position using the law of cosines, and two mirror-image candidates are generated. Next, each candidate is scaled to match the perimeter of the handwritten triangle, and aligned to its centroid position and longest-side direction. Finally, the candidate with the smaller total vertex distance error is selected and drawn using GeoGebra's `Polygon` command. The recognized numerical values for each side are displayed as labels and also stored as GeoGebra numerical objects, making them accessible from other GeoGebra features.

6.3 Example

Fig. 4 shows an adjustment example with the side ratio 5 : 12 : 13. Through the multi-digit recognition described above, "12" and "13", which consist of multiple strokes, are correctly recognized and reflected in the side ratios.

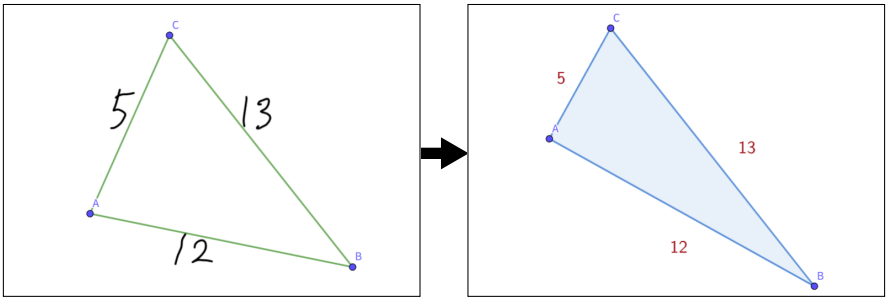

Fig. 4. Adjustment example with side ratio 5 : 12 : 13.

7 Conclusion

We described methods for utilizing a machine learning model for handwritten digit recognition in a React-based GeoGebra web application. The FastAPI approach converts the training environment directly into a Web API, suitable for prototyping. The ONNX approach requires

neither a server nor a Python environment, operating entirely within the web browser; it also supports GPU acceleration via WebGPU.

Unlike native GPU stacks such as CUDA and ROCm, WebGPU runs on supported browsers, eliminating the need for users to set up individual GPU environments. This enables the ONNX approach to reduce deployment and distribution costs for both users and developers.

This approach makes it possible to add advanced machine-learning-based functionality to GeoGebra. As an application, we implemented a triangle adjustment feature that reuses the existing MNIST inference module and rasterization pipeline, demonstrating that new functionality can be added without modifying the model itself. This feature is expected to be useful in mathematics education, such as quickly creating triangles with specific side ratios for learning about trigonometric ratios and similarity.

As future work, since the current MNIST model only recognizes digits 0–9, it cannot handle inputs containing mathematical symbols such as radicals. The current multi-digit segmentation assumes that each digit consists of 1–3 strokes and does not handle continuously written digit sequences; adopting a more robust segmentation method or an end-to-end sequence recognition model would improve practical usability. We envision introducing models with expanded recognition targets and developing automated proving of Euclidean geometry propositions from handwritten figures and auxiliary symbols.

The complete source code, including all code discussed in this paper, is available from [9].

Acknowledgments. This work was supported by JSPS KAKENHI Grant Numbers JP21K12157 and JP25K15373.

References

1. GeoGebra, https://www.geogebra.org/
2. Fujimoto, M.: Integrating GeoGebra with React and WebAssembly: A Web-Based Approach for Mathematical Software Development. In: Buzzard, K. et al. (Eds.) Mathematical Software – ICMS 2024, LNCS, vol. 14749, pp. 343–353. Springer (2024)
3. PyTorch, https://pytorch.org/
4. MNIST, http://yann.lecun.com/exdb/mnist/
5. FastAPI, https://fastapi.tiangolo.com/
6. Uvicorn, https://uvicorn.dev/
7. ONNX, https://onnx.ai/
8. ONNX Runtime, https://onnxruntime.ai/
9. Fujimoto, M.: HWR-Geo, https://github.com/mitsushi-fujimoto/hwr-geo/