INFO8010: Project Report

Javier Carballal Morgade,[1] **Alyssia Kayembe**,[2] and **Aurélien Juillé**[3]

[1] *javier.carballalmorgade@student.uliege.be (s2404541)*

[2] *aly.kayembe@student.uliege.be (s211023)*

[3] *aurelien.juille@student.uliege.be (s2409448)*

This project explores the application of convolutional neural networks (CNNs) for real-time classification of hand-drawn doodles, inspired by Google's *Quick, Draw!* game. Our goal was to develop a deep learning model capable of interpreting freehand sketches from users through a web interface, despite the inherent abstraction, variability, and noisiness of such inputs. We processed over 3 million images from the QuickDraw dataset using a custom vector-to-raster pipeline, enabling us to efficiently train and evaluate multiple CNN architectures. Among them, our final model, QuickDrawCNN, incorporates residual connections and hierarchical feature extraction, achieving a test accuracy of 70.54% across 345 classes, approaching human-level performance on this task. We also developed a web application that provides real-time predictions based on user input, demonstrating a full deployment pipeline. While the model's accuracy could be further improved with extended training, fine-tuning, or data augmentation, our results highlight the effectiveness of modern CNNs for sketch recognition and the importance of aligning preprocessing with deployment needs. This project emphasizes the feasibility of deploying deep learning models in interactive applications, offering both technical insights and practical experience in end-to-end system development.

I. INTRODUCTION

Humans have used sketching to depict our visual world for thousands of years. Recognising hand-drawn doodles poses unique challenges due to the abstract, inconsistent, and often incomplete nature of sketches, which differ significantly from real images, as humans are more subjective, creative, unpredictable. In this project, we aim to develop a convolutional neural network (CNN)-based system capable of accurately classifying user-provided doodles in real-time. This project is inspired by Google's Quick, Draw! game. The idea was to build an AI model that can understand and interpret freehand sketches provided by a user through an interactive web interface. The model will then attempt to predict the doodle class, even from partial or rapidly drawn input, demonstrating the capabilities of deep learning in image classification tasks. This project offers two challenges: developing an accurate CNN for noisy and varied input, and integrating it into an interactive application. Our goal is to build a minimum viable product that performs real-time doodle recognition for the users.

II. RELATED WORK

Our project is inspired by Google's *Quick, Draw!* game, which uses a neural network trained on millions of doodles to guess what users are drawing in real time. Several academic papers such as:

- David Ha & Douglas Eck (2017). *A Neural Representation of Sketch Drawings.* `https://arxiv.org/abs/1704.03477`
- Adarsh N L, Dr. Arun P V & Aravindh N L (2024). *Enhancing Image Caption Generation Using Reinforcement Learning with Human Feedback.* `https://arxiv.org/abs/2403.06735`
- O. Seddati, S. Dupont and S. Mahmoudi (2015). *DeepSketch: Deep convolutional neural networks for sketch recognition and similarity search.* `https://ieeexplore.ieee.org/document/7153606`
- W. Lu, E. Tran *Free-hand Sketch Recognition Classification* `https://cs231n.stanford.edu/reports/2017/pdfs/420.pdf`

have explored sketch and/or image recognition using CNNs, RNNs or RL, showing strong performance in classification or labelling tasks. The study of M. Eitz, J. Hays, and M. Alexa. (2012). How do humans sketch objects shows that even humans can correctly identify the object category of a sketch 73% of the time, and that even today, sketching is possibly the only rendering technique readily available to all humans. Therefore, if our model reaches at least the same accuracy as human beings, it would be a great sign for us. Similarly to what we do, frameworks such as Sketch-RNN have demonstrated the possibility of generating and understanding human-like sketches, which could be interesting if one would like to go further into this project. The paper *Structured Sparse Convolutional Autoencoder* `https://arxiv.org/abs/1604.04812` Ehsan Hosseini-Asl (2016) showed us an algorithm to improve the feature learning in Convolutional Networks (Convnet) by capturing the structure of objects. This introduced us to the more general concept of Convnet as well as ways to apply some regularisation methods to it. There also exist other similar fields of research which are quite close to our topic and could be worth diving into. For example,

handwriting recognition is similar to doodle recognition for the human subjectivity which is a key element. The paper of C. C. Tappert, C. Y. Suen and T. Wakahara, https://ieeexplore.ieee.org/abstract/document/57669 showed us a good overview of the research in that topic, which could be used to find leads in our own.

III. DATASET PREPARATION

To train a model capable of recognizing freehand sketches, we used the **Quick, Draw!** dataset provided by Googleprovided by Google.[**?**]. It contains over 50 million drawings across hundreds of classes, each collected from human users in a game-like setting. This makes it both large-scale and highly diverse, but also noisy and inconsistent. [1]

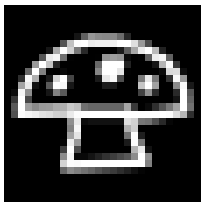

FIG. 1. Example of a rasterized doodle (class: *mushroom*) from the QuickDraw dataset after our preprocessing pipeline. The original drawing was composed of freehand strokes stored as vectors.

A. Available Formats

Google provides multiple formats of the dataset:

- **PNG images** (deprecated): limited and harder to preprocess consistently.
- **NumPy bitmap files**: fixed-size 28×28 grayscale raster images stored as flattened vectors in `.npy` format.
- **Simplified stroke-based NDJSON**: lists of vector strokes (sequences of (x, y) points) describing the full drawing process and adjusted to a 256x256 space.

We experimented with both the **NumPy bitmap format** and the **stroke-based vector format** before selecting the latter for final training.

B. Initial Attempt: NumPy Bitmaps

Our first approach used the rasterized version of the dataset, available on `storage.googleapis`

For each class, we downloaded a `.npy` file, extracted the first $N = 1000$ samples and stored them locally. This method had a few advantages:

- It was simple and required few lines of code to setup.
- The images were already preprocessed in a way that was consistent and ready to use for training.

However, when we wanted to experiment with bigger subsets of the dataset we ended up having issues with this way of fetching the data because it took too much time to download, given that the full .npy files had to be downloaded before saving a subset of the samples.

C. Final Approach: Stroke-Based NDJSON

The NDJSON format has the advantage of being much faster to download, and additionally allows us to only download the portion of the data that we are going to use. For this reason, we switched to this format, available on `storage.googleapis.com`

Each line in a class file is a JSON object describing a doodle as a sequence of strokes, each being a pair of lists (x-coordinates and y-coordinates). This change meant that we were now working with partially processed data, and for that reason we had to implement the code for rasterizing the vector images in a meaningful way. For this purpose, we used the original function used by Google to build the .npy dataset, found at google's documentation.

We implemented a custom pipeline to:

1. Download each `.ndjson` file (up to 10,000 samples per class).
2. Parse and convert strokes into centered and scaled vector images.
3. Rasterize each drawing using the `cairocffi` library with antialiasing and proper stroke thickness.
4. Transform the result into normalized PyTorch tensors.
5. Store each class as a `.pt` file for efficient loading during training.

We processed a total of 345 classes, resulting in over 3 million 28×28 grayscale images.

D. Advantages of the Final Method

This custom preprocessing offers several benefits:

- **Reusability**: allows us to apply the same processing to our own data, narrowing the gap between the dataset images and our custom ones.
- **Scalability**: this approach makes the download times manageable, allowing us to download bigger subsets of the original dataset.
- **Efficiency**: the data is saved in compressed, ready-to-train PyTorch tensor format.

This approach allowed us to train bigger and more accurate models.

E. Parallel Downloading and Processing

To accelerate dataset preparation, we used Python's `ThreadPoolExecutor` to process multiple classes in parallel, exploiting modern multi-core CPUs. Each class file is processed independently and saved separately, greatly improving download speeds and allowing dataset extension in the future.

IV. METHODS

A. Exploratory Data Analysis

The Google Quick, Draw! dataset comes with clear and detailed documentation, which made it easy for us to understand how the data is structured and formatted. Most of the information we needed came directly from reading these documents. Despite this, we still created a notebook to inspect the data more closely. We noticed that all the images follow a very consistent format in terms of size and structure, which is helpful when it comes to processing them. However, the drawings themselves are quite varied, showing a lot of different styles and interpretations from the people who created them all around the world.

B. Data Loading and Hyperparameters

Before starting the training, we configured the key hyperparameters: batch size, number of epochs, initial learning rate and a learning rate scheduler that reduced the learning rate by a factor of 0.5 if no improvement was observed for 3 consecutive epochs. We used a GPU to accelerate the process whenever it was available.

We loaded the data directly from `.pt` files, one per class, as tensors. After verifying the format and ensuring each tensor included the appropriate channel and labels, we combined them into a single `TensorDataset`. We then split this dataset into training, validation, and test sets using a 70/10/20 ratio. To handle data loading efficiently, we employed a `DataLoader` with multiple worker threads and pinned memory.

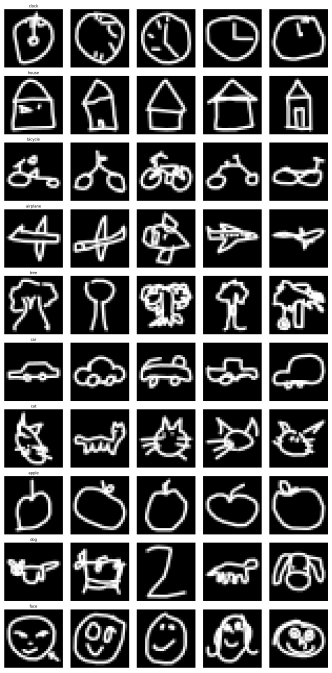

FIG. 2. Visualization of 5 sample images for each of our 10 initial categories

C. The Models

1. *SimpleCNN*

This initial model was designed as a baseline to validate our training and prediction pipeline. It consists of two simple convolutional layers with ReLU activations and MaxPooling, followed by two fully connected layers. The goal was to start with an architecture that is easy to implement, quick to train, and easy to debug.

We chose this structure for its simplicity: each convolutional layer gradually extracts more complex patterns from the input (e.g., edges and simple textures), and the pooling operations reduce the spatial dimensions, limiting the number of parameters in the fully connected layers.

It was initially used as a baseline when our project focused on only 10 doodle categories, where it achieved up to 95% accuracy. However, as we expanded to the full set of 345 categories, its performance dropped significantly to around 45.3% validation accuracy (a result that was expected given the model's limited capacity).

While this model works well for small datasets or simple tasks, it quickly reaches its limits with more complex

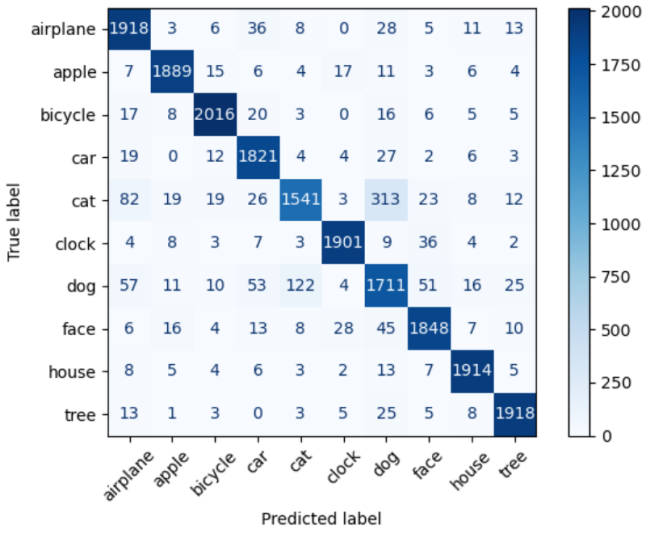

FIG. 3. Comparison between predicted and true labels by the SimpleCNN model.

data or large numbers of classes, like in our case with 345 categories.

In the figure3, we show the classification results obtained by this model with the small 10 class dataset. It is interesting to notice that similar classes like cat and dog are the ones where the model struggles the most.

2. *BiggerCNN*

To overcome the limitations of the baseline model, we developed `BiggerCNN`, a deeper architecture with more convolutional layers and filters. The idea was to enhance the network's capacity to learn richer features while still keeping the structure relatively straightforward.

We increased the depth (from 64 to 128 channels), stacked more convolutions with padding to preserve spatial resolution, and added a dropout layer before the final fully connected layer to reduce overfitting. Pooling is used to gradually reduce spatial dimensions and keep the computation manageable.

This intermediate design helped us understand the trade-offs between depth, feature extraction, and regularization before moving toward more advanced models inspired by modern architectures like ResNet.

However, despite these improvements, the model still struggled to exceed 59% validation accuracy on the full dataset. This limitation motivated us to explore a more modern and powerful architecture, which led to the development of the final model described in the next section.

3. *QuickDrawCNN*

We wanted to experiment with a more complex CNN that resembles more a modern approach to this problem. We built this model to better understand how modern CNNs are structured. It combines many key ideas from the theory we saw in class, especially those related to convolutional layers, residual connections, and the hierarchical nature of visual processing.

The model starts with a basic **stem block**, which uses a 3×3 convolution followed by batch normalization and a ReLU activation. This is the typical pattern for the initial layer in a CNN, where the goal is to begin extracting low-level features like edges and textures. Using small kernels (like 3×3) and stacking them helps the network build up complex representations in deeper layers — an idea we saw when discussing how CNNs mimic the visual system's feature hierarchy.

What makes this model more interesting is the use of **residual blocks**, which are inspired by ResNet. In each of these blocks, the input is added to the output of a small sequence of convolutional layers. This "skip connection" helps solve the vanishing gradients problem in deep networks, making it easier for the model to learn by preserving information across layers. When the number of channels changes, the skip path includes a 1×1 convolution to align dimensions — something we learned is often necessary in deeper networks.

Instead of using pooling layers, the model performs **downsampling** by setting the stride of some convolutions to 2. This reduces the spatial resolution while increasing the number of channels, allowing the network to capture more abstract features. This relates to the concept of **increasing receptive fields**, which grows more efficiently when downsampling is used alongside convolutions.

At the end, the network uses **adaptive average pooling** to compress the spatial dimensions to 1×1, summarizing each channel's information regardless of the original image size. This is followed by **dropout** for regularization and a fully connected layer that outputs the 345 class scores.

D. Training

To optimize the model, we used the Adam optimizer with a weight decay of 0.0001. For the loss function, we chose `CrossEntropyLoss`, which is appropriate for multiclass classification tasks. We relied on the learning rate scheduler to dynamically reduce the learning rate when the validation loss plateaued, although the small number of epochs that we performed limited its usefulness.

The training process iteratively updated the model's parameters via backpropagation using the training data.

At the end of each epoch, we evaluated the model on the validation set, recording both loss and accuracy metrics. This feedback guided adjustments to the learning rate and helped us monitor the model's generalization.

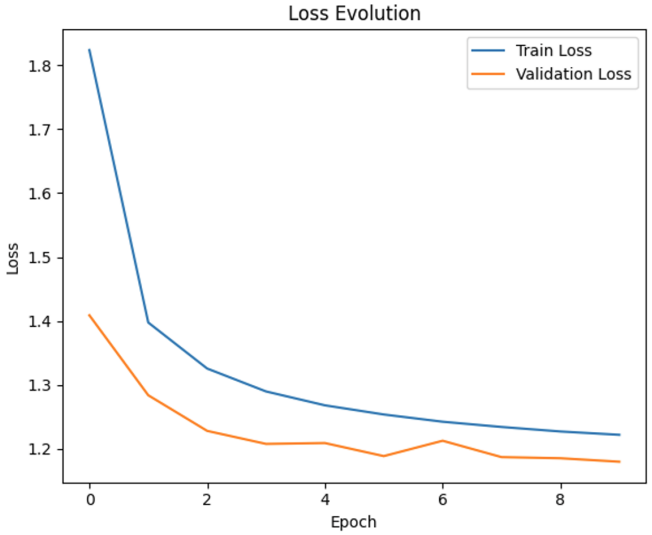

FIG. 4. Training and validation loss over 10 epochs. Both losses decrease, indicating successful learning. The validation loss flattens early, suggesting a need for further regularization or early stopping to avoid overfitting.

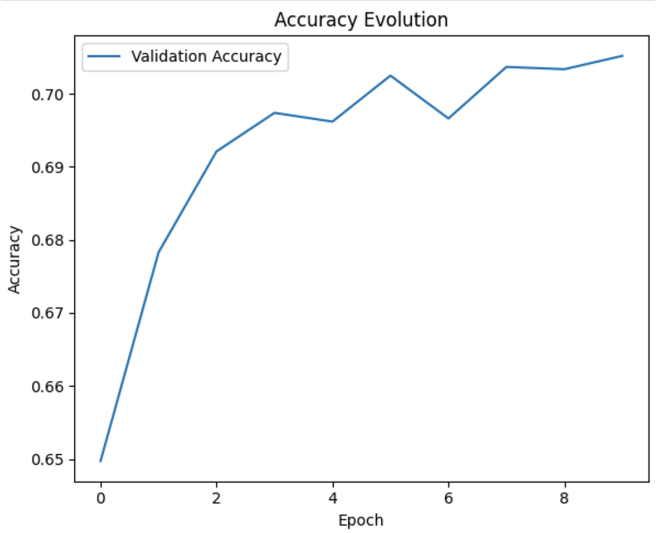

FIG. 5. Evolution of validation accuracy over 10 epochs. We observe a steady increase, showing that the model improves its generalization performance with each epoch.

Over the span of 10 epochs, we observed steady improvements in performance: the validation accuracy rose from 64.97% in the first epoch to 70.52% in the last, while the validation loss decreased from 1.40 to 1.17. These results indicated that the model was learning to generalize effectively.

Finally, we saved the trained model weights to a `.pt` file, enabling us to reuse the model for inference tasks in our web application.

V. RESULTS

A. Quantitative Analysis

We evaluated the performance of our best model, **QuickDrawCNN**, on the test dataset and achieved an overall accuracy of **70.54%**. This is a strong result considering the abstract and inconsistent nature of hand-drawn sketches. It's important to note that even humans typically achieve around 73% accuracy when identifying abstract doodles in this dataset, as reported by M. Eitz et al. (2012) (see related work). This places our model's performance near human level.

a. Final Training Metrics:

- **Validation accuracy:** Improved from 64.97% to 70.52% over 10 epochs.
- **Validation loss:** Decreased from 1.40 to 1.17.
- **Test accuracy:** Reached 70.54%.

b. Model Comparison: See the table I to visualise the comparison of our different model architectures and their performance.

B. Qualitative Analysis

To further evaluate the model behavior, we examined a variety of correct and incorrect predictions.

Correct Predictions: The model performed well **even with minimalistic or abstract** sketches. For example:

- A "mushroom" sketch with partial strokes was correctly predicted.
- A simple "umbrella" outline was classified accurately.

Incorrect Predictions: Misclassifications often occurred in cases of:

- **Visually similar classes** (e.g., "donut" vs. "cookie").
- **Ambiguous or overly simplistic drawings**.

Model	Parameters	Val. Accuracy (%)	Test Accuracy (%)
SimpleCNN	∼0.5M	45.30	47.83
BiggerCNN	∼2.1M	62.53	65.40
QuickDrawCNN	∼5.4M	**70.52**	**70.54**

TABLE I. Comparison of model architectures and performance.

C. Summary

Our final model achieved strong quantitative results with **70.54%** test accuracy and showed steady improvement during training. The qualitative analysis demonstrated that the model performs well on a wide variety of inputs but may struggle with ambiguous or overly abstract sketches. The validation loss continued to decrease steadily without signs of overfitting, indicating that the model had not yet plateaued. This suggests that further improvements were possible with extended training and more fine-tuning. However, due to the computational cost and long training times required, especially given the model size and dataset scale with the amount of categories, we had to settle for an earlier stopping point. Nonetheless, the model shows robust generalization and behavior close to human-level performance, which was our main goal for this challenge. Some visual proofs of the results can be seen directly in the Web Application section next.

VI. WEB APPLICATION

Despite not being the main goal of the project, we wanted to provide an interactive and accessible interface where users can draw freehand doodles and receive real-time predictions from the trained neural network, showcasing a real world application for our model. For this, we developed a minimalistic web application using the `Flask` framework, which serves as a bridge between the deep learning model and the front-end interface.

A. Architecture Overview

The application follows a classical client-server architecture:

- The **front-end** is a lightweight web page (HTML/CSS/JavaScript) that allows the user to draw sketches on a canvas.
- The **back-end**, implemented in Python with Flask, processes the sketch data, runs it through the CNN model, and returns a prediction.

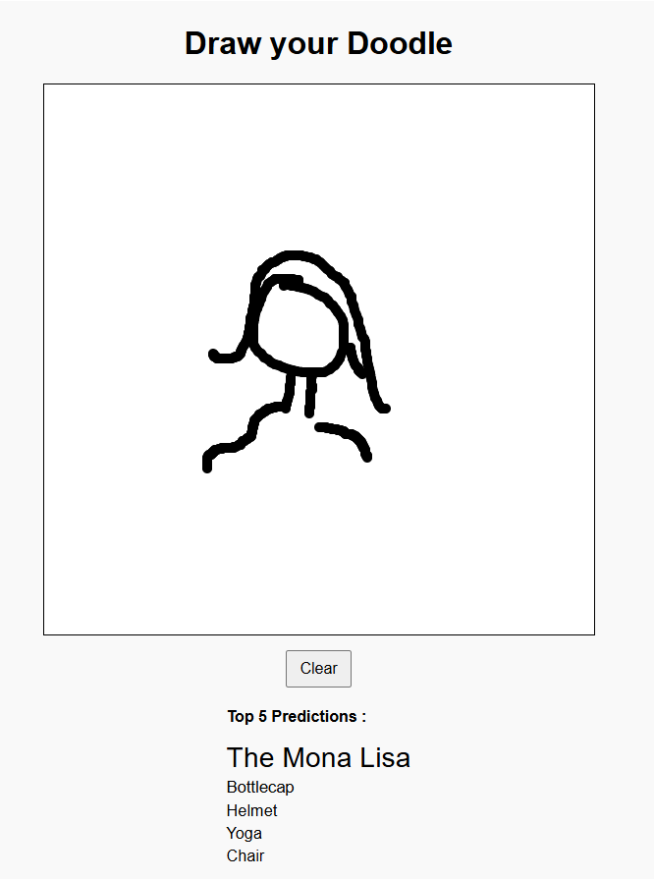

FIG. 6. Screen shot of the web app displaying the top 5 predictions for a Mushroom doodle.

B. Front-End Functionality

The front-end consists of a simple HTML canvas where users can draw using a mouse (or finger on touch devices). As the user draws, the canvas records the strokes as sequences of `(x, y)` coordinates grouped by stroke. Once a stroke is completed (on mouse up or out), the current list of strokes is sent to the server via an asynchronous `POST` request (AJAX).

To ensure a good user experience, the canvas includes a "Clear" button to reset the drawing, and the prediction result is dynamically updated on the page.

The client does not send an image, but rather the raw vector data of the strokes, which preserves resolution independence and reduces payload size.

C. Back-End Pipeline

When the server receives a new set of strokes, it follows these main steps:

1. **Preprocessing**: We process the strokes in the same way as the original dataset, to reproduce the format of the training images as faithfully as possible. For that, we made a custom function to:
    - Clean the strokes (e.g., empty or consecutive identical strokes are removed).
    - Align the drawing to the top-left corner, to have minimum values of 0.
    - Uniformly scale the drawing, to have a maximum value of 255.
    - Resample all strokes with a 1 pixel spacing.
    - Simplify all strokes using the Ramer–Douglas–Peucker algorithm with an epsilon value of 2.0.
2. **Rasterization**: To convert the simplified strokes into raster images, we apply the original Quick-Draw drawing process to scale the images down to 28x28 and align to the center, among other adjustments such as padding and antialiasing.
3. **Model Inference**: The rasterized image is converted into a normalized PyTorch tensor, passed through the trained CNN model, and the class with the highest probability is returned.

This inference process is fast enough to return predictions in real-time for user interaction.

D. Model Integration and Deployment

The web application loads a PyTorch model checkpoint at startup using:

```
model = OurModel()
model.load_state_dict(torch.load(...))
model.eval()
```

This ensures the model is ready to perform inference without retraining. The prediction logic is entirely encapsulated in the back-end, which allows flexibility to update the model or preprocessing pipeline without touching the front-end.

During development, the app was run locally using Flask's development server, but it could easily be containerized and deployed on a remote server with GPU support using services like Heroku, AWS, or Render.

E. Challenges and Considerations

A few challenges emerged while implementing the application:

- **Stroke preprocessing**: Raw mouse/touch data is often noisy, requiring resampling and simplification to match the training data format.
- **Rasterization**: Reproducing Google's vector-to-raster process required using the Cairo graphics library, which has native dependencies that can be difficult to install on some platforms.
- **Prediction delay**: Ensuring low latency between drawing and prediction response was key to providing a smooth UX.

F. Conclusion

This minimalist web app demonstrates how a deep learning model can be effectively deployed and interacted with in real time through a web interface. It also shows the importance of carefully designing the data pipeline (from strokes to tensors) to match training conditions and ensure robust predictions in practice.

VII. DISCUSSION

Our project successfully demonstrated the viability of training a convolutional neural network for real-time sketch recognition, achieving a test accuracy of 70.54% across 345 different categories. This performance is close to human-level accuracy for this task, suggesting that our preprocessing and modelling choices were effective for dealing with the abstract and variable nature of doodles.

Nonetheless, the model has several limitations. First, the training was constrained to just 10 epochs due to limited computational resources. The learning curves indicated that further training could have improved performance. Second, the model lacked extensive hyperparameter tuning, data augmentation, or ensembling—all of which are known to improve accuracy in classification tasks.

Future work could focus on several improvements:

- **Extended training and tuning:** Training the model for more epochs, using more sophisticated learning rate schedules, and fine-tuning architecture-specific hyperparameters could lead to better accuracy.

- **Data augmentation:** Introducing random distortions, rotations, or elastic deformations could improve robustness to drawing variability.
- **Attention mechanisms or transformers:** These architectures have shown promise in vision tasks and could help the model focus on key regions of the drawing.
- **Improved web interface:** Enriching the web application with features like class feedback, model explanation, or drawing hints could enhance usability and model interpretability.

In conclusion, beyond the technical achievements, this project provided us with a valuable learning experience across multiple aspects of machine learning and software engineering. We deepened our understanding of convolutional neural networks, data preprocessing, and the practical challenges of working with real-world, noisy datasets. Additionally, integrating the trained model into a real-time web application taught us how to bridge the gap between research and deployment. While there is still room for improvement, the project gave us hands-on insight into the full pipeline of developing, evaluating, and deploying a deep learning system—from raw data to end-user interaction.

---

[1] `https://quickdraw.withgoogle.com/data`.