Real-Time Sketch Classification using Convolutional Neural Networks

Ethan Sterling

CS 399, December 2024

1 Introduction

1.1 Google's Quick, Draw

Quick, Draw! is an engaging online game developed by Google. It challenges users to sketch objects or concepts within 20 seconds, with an artificial intelligence (AI) neural network attempting to guess the subject of the drawing. Each user's drawing contributes to the AI's learning, enhancing its prediction capabilities. Quick, Draw! exemplifies AI's potential in creating intuitive, interactive experiences, combining simple gameplay akin to Pictionary with cutting-edge AI capabilities.

The aim of this project was to recreate the essence of Quick, Draw! by building an AI-driven application that predicts drawings in real time.

1.2 Why Machine Learning

Machine Learning (ML) is an indispensable tool for tackling tasks that involve complex, high-dimensional datasets such as those comprising freehand sketches. Traditional programming paradigms rely heavily on predefined rules and explicit instructions to process data. While this approach works well for structured and predictable data, it is ill-suited for the inherent variability and ambiguity present in hand-drawn sketches. Human sketches are diverse, reflecting variations in style, precision, and interpretation, which makes it challenging to capture their essence using rule-based approaches.

ML, particularly data-driven learning models, is well-equipped to manage such complexities. By learning patterns directly from the data, ML algorithms can develop representations that are robust to noise and variations. This ability makes ML a compelling choice for interpreting user-drawn sketches where features such as stroke order, pressure, and proportions can vary widely between individuals.

2 Dataset and Preprocessing

2.1 Dataset Overview

The Quick, Draw! Dataset is a vast and diverse collection of over 50 million hand-drawn sketches contributed by players worldwide as part of Google's Quick, Draw! game. Spanning 345 categories, the dataset captures the creative and interpretative ways individuals represent common objects and ideas through drawings. Each sketch is vectorized, meaning the dataset stores the stroke-by-stroke sequence used to create the drawing. This vectorized format retains rich temporal and spatial information, enabling detailed analysis of the drawing process. Additionally, metadata accompanies each drawing, including the category label (what the user was attempting to draw) and the geographical region of the contributor, offering insights into cultural and stylistic variations.

For this project, we focused on a subset of the dataset to streamline the scope and computational requirements. Fourteen categories were selected based on their recognizability, diversity, and relevance to everyday objects. These categories include: airplane, hot air balloon, ice cream, flower, pizza, bicycle, star, camera, dog, radio, envelope, lighthouse, mailbox, and windmill. By narrowing the dataset to these categories, the project balances complexity with feasibility, ensuring the model is trained on a manageable yet representative subset of the original dataset.

This curated subset not only simplifies the computational requirements but also provides a meaningful challenge for the model, given the variability in how contributors depict objects like "star" or "windmill." The selected categories also reflect a mix of simple shapes (e.g., "star") and more intricate patterns (e.g., "bicycle"), enabling the model to learn features across varying levels of complexity. This approach ensures the resulting system is both versatile and practical in its ability to recognize diverse sketches.

2.2 Preprocessing Steps

The Quick, Draw! Dataset and the preprocessing applied to it form the foundation of this project. Effective preprocessing is essential for reducing noise, standardizing inputs, and optimizing data for model training. Preprocessing was conducted in two stages: initially by the dataset creators and subsequently through custom preprocessing implemented in this project. Each step was designed to ensure the data was in a format suitable for efficient learning by machine learning algorithms.

2.2.1 Preprocessing by the Dataset Creators

The Quick, Draw! Dataset was initially preprocessed by its creators to simplify the raw input and facilitate efficient exploration. These steps included the following:

1. Vector Simplification: The dataset simplified the vectorized stroke data by removing timing information, retaining only the spatial representation of each drawing. This ensures the focus remains on the static representation of the sketches rather than temporal drawing patterns, which might add unnecessary complexity for this use case.

2. Alignment and Scaling: Drawings were aligned to the top-left corner, ensuring the minimum values for both x and y coordinates were 0. The sketches were then uniformly scaled to fit within a 256x256 region, with the maximum value set to 255. This step normalized the dimensions of the drawings, making them comparable regardless of the original size or position of the sketch.

3. Resampling: Each stroke was resampled with 1-pixel spacing. This process regularizes the representation of strokes, ensuring consistency in the density of the points and removing irregularities caused by varying stroke speeds or drawing precision.

4. Stroke Simplification: The Ramer–Douglas–Peucker algorithm was applied with an epsilon value of 2.0. This algorithm reduces the number of points in each stroke while preserving its overall shape. Simplification minimizes redundancy in the data while retaining critical features.

These preprocessing steps made the dataset more compact, efficient, and suitable for large-scale machine learning tasks.

2.2.2 Custom Preprocessing

Building on the dataset's preprocessing, additional steps were implemented to prepare the data for Convolutional Neural Network (CNN) training. These steps were tailored to match the model's input requirements and to optimize computational efficiency:

1. Reshaping to (28, 28, 1): Each drawing was resized to 28x28 pixels and converted to grayscale, resulting in a shape of (28, 28, 1) for each sample. This dimensionality reduction significantly lowers computational requirements while retaining the essential features of the drawings. The grayscale format simplifies the input, as color information is unnecessary for this classification task.

2. Normalization of Pixel Values: Pixel values were normalized to the range [0, 1] by dividing by 255. This ensures uniform scaling of features and prevents dominance by higher pixel intensity values during training. Normalization also improves model convergence by standardizing input distributions.

3. Label Assignment: Labels were assigned based on the order of .npy files in the dataset folder. This consistent mapping ensures that the relationship between sketches and their respective categories is preserved throughout the training process.

4. Class Name Extraction: Class names were extracted from the filenames of the .npy files, excluding their extensions. These names provide a

human-readable mapping between numerical labels and their corresponding categories, aiding in evaluation and interpretation.

5. Data Splitting: The dataset was split into training and validation sets to enable model evaluation during training. A standard split ratio (e.g., 80:20) was used to ensure sufficient data for both learning and validation. This step mitigates overfitting and provides an unbiased assessment of model performance.

6. Saving Preprocessed Data: The processed data splits (X_train, X_val, y_train, y_val) and class_names were saved to disk in .npy format. This allows for easy reuse without repeating the preprocessing pipeline, saving computational resources and ensuring consistency across experiments.

2.2.3 Rationale for Preprocessing Steps

Each preprocessing step served a specific purpose in ensuring the data's suitability for machine learning:

- Vector Simplification and Stroke Processing: Reduced the complexity and size of the raw dataset while retaining essential features for classification.
- Reshaping and Normalization: Standardized the input dimensions and feature scales, enabling the CNN to learn efficiently.
- Label and Class Name Management: Maintained consistency in category mappings, facilitating training and evaluation.
- Data Splitting: Supported the development of a robust model by providing separate datasets for training and validation.

These preprocessing steps collectively transformed raw sketches into a format optimized for CNN training, balancing computational efficiency with the retention of critical features. This pipeline ensures that the model is trained on standardized, high-quality input, enabling it to generalize effectively to unseen data.

3 Machine Learning Techniques

3.1 Convolutional Neural Network

Convolutional Neural Networks (CNNs) are a class of deep learning models designed specifically for processing and analyzing visual data. Inspired by the organization of the animal visual cortex, CNNs are highly effective at recognizing patterns and structures in images, making them a natural fit for tasks like sketch recognition.

Unlike traditional neural networks, which treat all input data as a single vector, CNNs maintain the spatial structure of the data. This allows them to extract and learn hierarchical features—starting from simple patterns like edges and progressing to more complex structures such as shapes or object outlines. This hierarchical processing mimics the way humans perceive and interpret visual information.

3.1.1 Structure of a CNN

1. Convolutional Layers:

    - The convolutional layer is the core building block of a CNN. It applies a series of filters (or kernels) to the input image, producing feature maps that highlight specific patterns or features in the image.
    - Filters are small matrices (e.g., 3x3 or 5x5) that slide over the input image, computing dot products at each step. This process captures local patterns such as edges, corners, or textures.
    - Each filter learns to detect a specific feature, such as a horizontal edge or a curve. The depth of the feature map increases with the number of filters, allowing the network to represent a diverse set of patterns.
    - Example: A convolutional layer might detect the edges of a "bicycle" in the first layer and its wheels or frame in subsequent layers.

2. Activation Functions:

    - After convolution, activation functions such as ReLU (Rectified Linear Unit) are applied to introduce non-linearity. This enables the network to learn complex relationships in the data.

- ReLU replaces all negative values in the feature map with zero, making computations efficient and ensuring the model focuses only on significant features.

3. Pooling Layers:
    - Pooling layers follow convolutional layers and downsample the feature maps, reducing their spatial dimensions (e.g., height and width) while retaining the most important information.
    - Common pooling techniques include max pooling (which selects the maximum value in a region) and average pooling (which computes the average value in a region).
    - By reducing the size of feature maps, pooling layers decrease computational requirements, mitigate overfitting, and make the model more robust to minor variations like translations or distortions in the input image.

4. Fully Connected Layers:
    - After feature extraction, the output of the convolutional and pooling layers is flattened into a one-dimensional vector. This vector is passed through fully connected (dense) layers to combine features and make predictions.
    - Fully connected layers serve as the decision-making component of the network, assigning probabilities to different categories based on the extracted features.

5. Dropout Layers:
    - Dropout is a regularization technique used in CNNs to prevent overfitting. During training, dropout layers randomly deactivate a subset of neurons in the network. This forces the model to learn robust features that are not overly reliant on specific neurons.

3.1.2 How CNNs Process Images

When a sketch is input into a CNN, the following steps occur:

- The first convolutional layers identify basic features such as lines, edges, and blobs.

- Subsequent layers build on these low-level features to detect more complex shapes or patterns, such as the circular shape of a ”pizza” or the spokes of a ”bicycle wheel.”
- Pooling layers ensure that only the most critical information is retained, reducing the sensitivity to variations in the input image (e.g., scale or orientation changes).
- Fully connected layers combine the features and assign probabilities to each class, providing the final prediction.

3.1.3 Advantages of CNNs

1. Parameter Sharing: CNNs utilize filters that share weights across different regions of the input. This reduces the number of parameters, making the model computationally efficient and less prone to overfitting.
2. Translation Invariance: The spatial hierarchy in CNNs ensures that patterns are recognized regardless of their location in the image. This is particularly useful for sketch recognition, where the position of strokes may vary.
3. Scalability: CNNs can process high-dimensional data, enabling them to scale effectively for large and complex datasets like Quick, Draw!.
4. Automation of Feature Extraction: Traditional approaches to image recognition require handcrafted features, which can be labor-intensive and domain-specific. CNNs learn these features automatically during training, adapting to the specific characteristics of the dataset.

3.2 Quick, Draw! Model

3.2.1 Starting Model

The initial model was designed to provide a foundational framework for sketch classification. This model leveraged three convolutional layers, followed by pooling layers, and concluded with fully connected dense layers. The architecture is outlined below:

1. Convolutional Layers: The model uses three convolutional layers with 32, 64, and 128 filters, respectively, and a kernel size of (3, 3). This configuration aims to extract features progressively, starting from simple patterns like edges to more complex patterns like shapes.

2. Pooling Layers: Each convolutional layer is followed by a MaxPooling layer with a pooling size of (2, 2). This layer reduces the spatial dimensions of the feature maps by half, preserving the most important features while reducing computational complexity.

3. Dense Layers: The Flatten layer converts the 3D feature maps into a 1D vector, which is passed to a fully connected dense layer with 128 units. This layer combines the learned features to classify the sketches.

4. Dropout: A dropout rate of 0.5 is applied before the final output layer to prevent overfitting by randomly deactivating half the neurons during training.

5. Output Layer: The final dense layer uses the softmax activation function to assign probabilities across the num_classes categories.

This model performed adequately during training and validation, achieving reasonable accuracy. However, limitations such as overfitting and insufficient capacity for learning complex features became apparent during evaluation, prompting a need for refinement.

3.2.2 Refined Model

Based on the insights gained from the initial model, several architectural changes were made to enhance performance. The refined model architecture is as follows:

1. Larger Filter Sizes: The kernel size was increased from (3, 3) to (5, 5) for both convolutional layers. Larger filters allow the network to capture more contextual information within each convolutional step, which is beneficial for recognizing more intricate patterns in the sketches.

2. Padding: The MaxPooling layers now use padding='same', ensuring that the output size remains consistent and important features near the edges are not lost during pooling.

3. Increased Dense Layer Capacity: The number of neurons in the first dense layer was increased from 128 to 512, significantly expanding the model's capacity to learn complex features and relationships within the data.

4. Higher Dropout Rate: The dropout rate was increased from 0.5 to 0.6 in both dropout layers, further reducing the likelihood of overfitting by forcing the model to rely on a more distributed set of features.

5. Streamlined Architecture: The architecture was simplified by removing one convolutional layer, making the model more computationally efficient while maintaining its feature extraction capacity.

The refined model achieved higher validation accuracy and better generalization to unseen data compared to the initial model. These improvements were reflected in evaluation metrics such as precision, recall, and F1-scores.

3.2.3 Comparison and Justification for Changes

1. Model Depth vs. Complexity: The starting model, with three convolutional layers, provided sufficient depth but lacked the capacity to learn from complex patterns. The refined model balanced this by focusing on larger filters and enhanced dense layers.

2. Overfitting Mitigation: The increased dropout rate and simplified architecture in the refined model helped mitigate overfitting, which was observed in the initial model during validation.

3. Enhanced Feature Representation: Larger filters and more neurons in the dense layer enabled the refined model to capture more nuanced features in sketches like "bicycle" or "flower," which require a higher degree of abstraction.

4. Computational Efficiency: Despite having larger filters and a denser network, the refined model maintained reasonable training times due to the reduction in convolutional layers and efficient pooling strategies.

In summary, the progression from the initial to the refined model illustrates a systematic approach to improving both the learning capacity and generalization ability of the network, resulting in a more robust classifier for the Quick, Draw! task.

4 Results

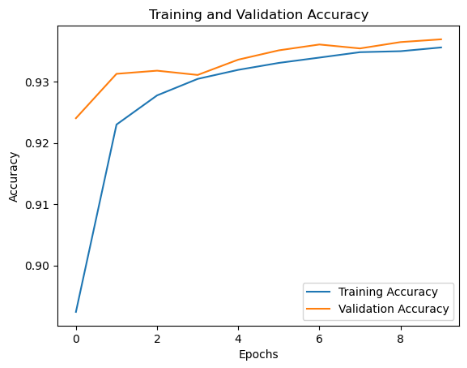
4.1 Training Performance

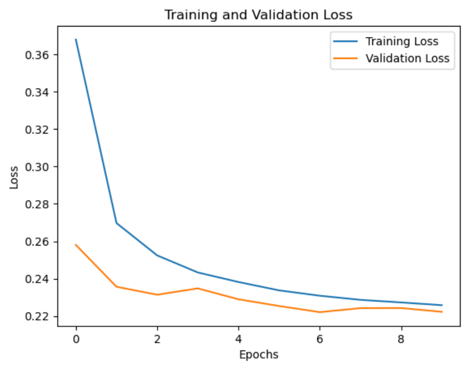
Text

4.2 Evaluation Metrics

The training and validation performance of the starting and final models are visualized in the plots below. These metrics highlight the evolution of the models over the training epochs and provide insights into their ability to generalize to unseen data.

4.2.1 Starting Model



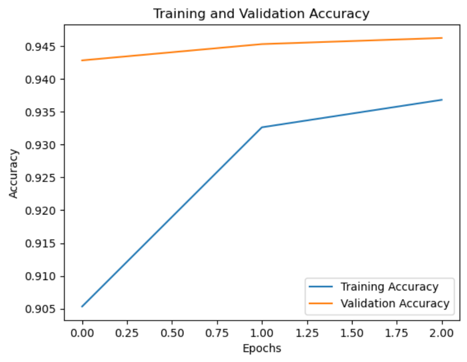


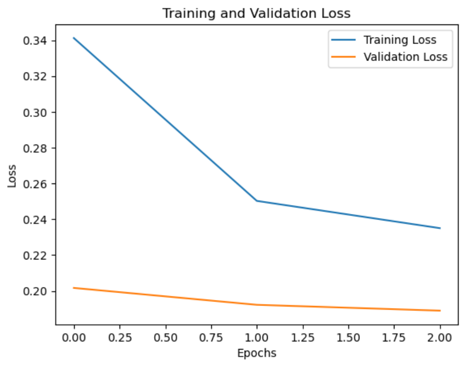
1. Training and Validation Accuracy:
   - The starting model exhibits steady improvement in training accuracy, eventually plateauing near 93% by the 9th epoch.
   - Validation accuracy starts higher than training accuracy, likely due to the regularization effects of dropout. By the end of training, validation accuracy reaches approximately 94%.
   - The close alignment between training and validation accuracy curves suggests that the model generalizes reasonably well without severe overfitting.
2. Training and Validation Loss:

- Training loss decreases consistently across epochs, indicating that the model is effectively learning to minimize the error on the training set.
- Validation loss also decreases in tandem with training loss, stabilizing around 0.22. The smooth convergence of both losses suggests that the model's capacity is well-suited to the complexity of the task, though additional improvement might be achievable with a deeper architecture.

The starting model demonstrated solid performance but lacked the capacity to learn more complex patterns in the data. Its reliance on a moderate number of filters and smaller dense layers limited its ability to distinguish between visually similar categories, such as "airplane" and "hot air balloon." This limitation became a motivating factor for refining the model.

4.2.2 Refined Model





1. Training and Validation Accuracy:
   - The final model achieves a higher starting accuracy on both training and validation sets, suggesting that the adjustments (e.g., larger filters and increased dense layer capacity) allowed it to learn useful patterns earlier in training.
   - Validation accuracy exceeds 94% within the first epoch and stabilizes at around 95%, indicating improved generalization.

- The close alignment of the curves highlights the model's robustness and suggests reduced overfitting compared to the starting model.

2. Training and Validation Loss:
    - Training loss decreases significantly over the three epochs, reaching approximately 0.19 by the end.
    - Validation loss remains consistently lower than training loss, stabilizing around 0.20. This indicates that the regularization techniques, such as increased dropout rates, effectively mitigated overfitting.

The refined model demonstrates superior performance compared to the starting model, with higher accuracy and lower loss on both training and validation sets. The increased filter sizes, larger dense layers, and adjusted dropout rates likely contributed to this improvement by enabling the model to learn more intricate features without overfitting.

4.2.3 Comparison

The starting model required 10 epochs to achieve 94% validation accuracy, while the refined model surpassed this benchmark within 2 epochs and reached 95% validation accuracy by the end of training. This improvement not only highlights the refined model's ability to learn complex patterns but also its efficiency in doing so. Moreover, the refined model's lower validation loss indicates greater confidence in its predictions, making it better suited for deployment in real-world applications like the Quick, Draw! recreation.

4.3 Execution Times

The execution times for the starting and final models provide valuable insights into the computational efficiency and scalability of each architecture. While the final model achieved better performance in fewer epochs, its increased complexity resulted in longer training times per epoch.

4.3.1 Starting Model

- Execution Time: 10 epochs at approximately 5 minutes each, totaling 50 minutes of training time.

- Architecture Complexity: The starting model used three convolutional layers with smaller kernel sizes (3x3) and fewer dense layer neurons. This lightweight design allowed for faster per-epoch training times, making it computationally efficient.

- Training Behavior: Despite the shorter per-epoch time, the model required all 10 epochs to converge to a validation accuracy of approximately 94%. The relatively shallow architecture and smaller feature extraction capacity limited the model's ability to learn more complex patterns efficiently.

4.3.2 Refined Model

- Execution Time: 3 epochs at approximately 8 minutes each, totaling 24 minutes of training time.

- Architecture Complexity: The refined model incorporated larger kernel sizes (5x5), more neurons in the dense layers (up to 512), and a higher dropout rate. These modifications increased the model's ability to learn intricate features but required more computations per epoch.

- Training Behavior: Despite the increased per-epoch time, the final model converged to a validation accuracy of 95% within just 3 epochs. This demonstrates the efficiency of the refined architecture in extracting and learning patterns from the dataset with fewer training iterations.

4.3.3 Comparison

1. Total Training Time: The starting model required twice the total training time (50 minutes) compared to the final model (24 minutes) due to the need for more epochs to converge. While the starting model was computationally lighter, it was less effective in learning complex features, resulting in longer overall training.

2. Efficiency per Epoch: Although the final model took 60% longer per epoch (8 minutes compared to 5 minutes), its ability to achieve higher validation accuracy in fewer epochs highlights its superior efficiency in terms of learning capacity.

3. Trade-offs: The refined model's increased per-epoch time is a reasonable trade-off given the reduction in total training time and the significant improvement in accuracy and loss metrics.rate

The final model demonstrates a clear advantage in terms of overall training efficiency. By converging in fewer epochs, it reduced the computational overhead associated with prolonged training while achieving superior performance. These results underline the importance of optimizing architectural complexity to balance per-epoch time with the total training duration.

5 Analysis

5.1 Model Evolution

The evolution from the starting model to the final model marked a significant leap in performance and efficiency, driven by targeted architectural improvements. While the starting model provided a solid foundation, it lacked the complexity required to effectively capture intricate features in the dataset. The refined model addressed these shortcomings, resulting in better accuracy, faster convergence, and improved generalization.

One of the primary differences between the two models lies in their convolutional layers. The starting model utilized three convolutional layers with smaller kernel sizes of 3×3. This design, while effective for capturing simple patterns, limited the model's ability to extract more contextual information from the sketches. In contrast, the final model reduced the number of convolutional layers to two but increased the kernel size to 5×5. This change allowed the final model to capture broader patterns within each layer, enabling it to better recognize intricate structures, such as the details in categories like "bicycle" or "lighthouse." Additionally, the final model incorporated padding in its pooling layers, ensuring that features at the edges of the feature maps were preserved. This adjustment improved the robustness of the extracted features, which was particularly beneficial for sketches that were drawn near the edges of the input canvas.

Another critical enhancement was in the dense layers. The starting model employed a single dense layer with 128 neurons, which limited its capacity to combine and learn complex feature representations. The final model addressed this limitation by adding a second dense layer and increasing the number of neurons in the first dense layer to 512. This expansion provided the model with a significantly greater capacity to process and interpret the rich features extracted from the convolutional layers. Furthermore, the dropout rate was increased from 0.5 to 0.6 in the final model, improving regularization and mitigating overfitting despite the model's increased complexity.

These architectural refinements translated into tangible improvements in performance. The starting model achieved a validation accuracy of approximately 94% after 10 epochs, with a validation loss stabilizing around 0.22. In contrast, the final model reached a higher validation accuracy of 95% in just three epochs and reduced the validation loss to 0.20. The reduced training time and faster convergence of the final model underscored its enhanced learning efficiency. While the starting model required a total training time of 50 minutes across 10 epochs, the final model completed training in just 24 minutes, highlighting the effectiveness of the refined architecture in learning complex patterns more efficiently.

In summary, the final model represents a significant improvement over the starting model in terms of both accuracy and computational efficiency. By incorporating larger filters, optimizing pooling strategies, and expanding dense layers, the refined model demonstrated superior generalization and faster convergence. These improvements underscore the importance of iterative refinement in deep learning, where careful adjustments to architecture can result in substantial gains in performance and applicability to real-world tasks like sketch recognition.

5.2 Real-Time Predictions

As part of this project, a real-time prediction interface was developed to allow users to draw their own sketches and receive predictions from the trained model. This interface was built using Tkinter, a Python library for creating graphical user interfaces. The application featured an interactive canvas where users could draw using a brush tool, with options to clear the canvas or trigger a prediction. When a user clicked the "Predict" button, the drawn image was preprocessed and fed into the model, which returned its best guess along with the confidence score. This user-friendly interface aimed to recreate the core experience of Google's Quick, Draw! game.

Despite the functional interface, the model often struggled to correctly identify the user-drawn sketches. For many drawings, the predictions were either incorrect or lacked sufficient confidence, highlighting a significant gap between the performance observed during model evaluation and the real-world results. This disparity is most likely due to differences in the input conditions between the training data and the live-drawn sketches.

The Quick, Draw! dataset underwent substantial preprocessing during its creation, including alignment, scaling, and vector simplification, which standardized the input for the model. However, the real-time interface used a distinct preprocessing pipeline, where user drawings were resized to 28×28, converted to grayscale, and normalized. This difference in preprocessing likely resulted in a mismatch between the characteristics of the input images seen by the model during training and those generated by the drawing interface. Additionally, user-drawn sketches often contained noise, uneven strokes, and variations in line thickness, further diverging from the clean and standardized dataset the model was trained on.

These challenges highlight the limitations of the current implementation in real-time applications. While the model performed well on the preprocessed validation set, its poor performance on live input underscores the importance of aligning training data with real-world use cases. Addressing these issues could involve augmenting the dataset with more varied and realistic inputs, refining the preprocessing steps in the real-time interface to better match those used during training, or employing advanced techniques like transfer learning to improve robustness.

In conclusion, while the real-time prediction interface successfully demonstrated the application of the trained model in an interactive setting, it exposed critical gaps in the model's ability to generalize to live-drawn inputs. These findings provide valuable insights into areas for improvement in future iterations of the project.

6 References

- Google Creative Lab. Quick, Draw! Data . GitHub, `https://github.com/googlecreativelab/quickdraw-dataset/tree/master?tab=readme-ov-file#preprocessed-dataset` Accessed 3 Dec. 2024.