# CSL7590: Deep Learning
# Assignment 3: Building a Sequential Model for Sketch Generation

**AY 2024-25, Semester - II**

**Submitted By:**

Priyansh Saxena
B22EE075

**Colab Notebook Link:** **DL_Assignment3.ipynb**

---

**Abstract**

This report details the design, implementation, and evaluation of a sequential generative model for creating sketches, based on the SketchRNN architecture. Using the Quick, Draw! dataset and the PyTorch framework, we developed a model capable of generating vector-based drawings step-by-step when conditioned on a specific class name. The model was trained on 10 distinct object classes. Key aspects covered include data preprocessing (loading, normalization, vectorization, splitting), the LSTM-based model architecture with class embedding, the training process involving a custom loss function, optimization techniques like gradient clipping, learning rate scheduling, early stopping, and qualitative evaluation through visualization. We analyze the model's ability to generate recognizable sketches sequentially, compare them with real examples, and discuss successes and failures. Additionally, implementations for bonus tasks exploring interactive refinement via temperature control and basic multi-object composition are presented.

---

## 1. Introduction

### 1.1. Background

Generating visual content that mimics human creation processes is a challenging task in deep learning. Human sketching is inherently sequential – drawings are formed through a series of strokes and pen lifts over time. Models aiming to replicate this must capture these temporal dependencies. Recurrent Neural Networks (RNNs), particularly Long Short-Term Memory (LSTM) networks, are well-suited for modeling such sequential data. This project focuses on implementing a SketchRNN-like model, inspired by Ha & Eck (2017), to generate sketches.

### 1.2. Objective and Dataset

The primary objective of this assignment is to implement a sequential model in Python using PyTorch that generates step-by-step sketches of objects based on their class names.

- **Dataset:** We utilize the **Quick, Draw! dataset** (https://quickdraw.withgoogle.com/data), a collection of millions of stroke-based drawings. Each sketch is represented as a sequence of points including pen state information (down, up, end).
- **Class Selection:** As required, we selected **10 distinct object classes** for training and evaluation: "apple", "candle", "cloud", "bicycle", "cake", "clock", "grapes", "fish", "car", "star" (defined in Config.SELECTED_CLASSES).
- **Sequential Generation:** A critical requirement is that the model must produce strokes **step-by-step**, mimicking human drawing behavior. Generating a complete image at once is explicitly disallowed. Our implementation adheres to this by generating one stroke vector (dx, dy, p1, p2, p3) per time step auto-regressively.

### 1.3. Report Structure

This report covers the data preprocessing pipeline, the chosen model architecture and its justification, the training and evaluation procedures, visualization methods demonstrating sequential generation, analysis of results, and details on the implemented bonus features.

---

## 2. Data Preprocessing

Proper data handling is crucial for training sequence models. Our preprocessing pipeline, implemented, involved the following steps:

### 2.1. Loading and Initial Filtering (download_quickdraw_data)

- The quickdraw library was used to download sketch data for the 10 selected classes, limiting samples to Config.MAX_SAMPLES_PER_CLASS (1000).
- Basic filtering ensured that only drawings with valid, non-empty stroke data were retained.

### 2.2. Stroke Sequence Conversion (strokes_to_sequence)

- Raw stroke data (lists of x,y coordinates) was converted into a sequence of 5-dimensional vectors: (dx, dy, p1, p2, p3).
    - (dx, dy): Relative offset from the previous point.
    - p1: Pen down state (1 if drawing, 0 otherwise).
    - p2: Pen up state (1 if lifted, 0 otherwise).
    - p3: End of sketch state (1 for the final vector, 0 otherwise).
- Sequences were truncated or padded implicitly by ensuring the final state was p3=1 within the Config.MAX_SEQ_LENGTH (200). This vectorization serves as the "tokenization" for stroke data, converting continuous drawing actions into a discrete sequence format suitable for the RNN.

### 2.3. Normalization (calculate_stats, normalize_sequence)

- To stabilize training, the dx and dy components were normalized using Z-score normalization.
- The mean and standard deviation were calculated *only* from the training set (calculate_stats) and applied to all datasets (normalize_sequence within QuickDrawDataset). Statistics were saved (Config.STATS_SAVE_PATH) to ensure consistency.

### 2.4. Class Name Handling and Final Processing (preprocess_data, QuickDrawDataset)

- Class names (strings) were mapped to integer indices (class_map). This index is used for the embedding layer lookup, effectively "tokenizing" the class name input.
- The preprocess_data function aggregated all processed sequences along with their class index, original length, and class name.
- A custom QuickDrawDataset class was implemented to handle loading individual samples, applying normalization, and preparing input/target pairs (input is the target shifted right, prepended with a start token).

### 2.5. Dataset Splitting

- The fully processed dataset was shuffled and split into training, validation, and test sets according to the required **70:15:15** ratio. This was performed before creating the QuickDrawDataset instances.

### 2.6. Batching (collate_fn, DataLoader)

- A custom collate_fn was used with PyTorch DataLoader to handle batching of variable-length sequences. It sorts sequences by length, pads them to the maximum length in the batch, and prepares the batch dictionary for the model.

---

## 3. Model Architecture

We designed and implemented a sequential generative model based on the SketchRNN concept, leveraging LSTMs for their ability to capture temporal dependencies in sequences.

### 3.1. Model Components (SketchRNN class)

- **Class Embedding (nn.Embedding):** Maps the integer class index (input) to a dense vector (Config.EMBEDDING_DIM=64). This allows the model to learn class-specific representations and condition the generation process.
- **LSTM (nn.LSTM):** The core recurrent unit. We used a 2-layer stacked LSTM (Config.NUM_LSTM_LAYERS=2) with a hidden dimension of 512 (Config.HIDDEN_DIM=512). The input to the LSTM at each time step is the concatenation of the previous stroke vector and the replicated class embedding

vector. Dropout (Config.DROPOUT_RATE=0.3) is applied between LSTM layers for regularization.

- **Output Layer (nn.Linear):** A fully connected layer maps the LSTM hidden state output at each step to a 5-dimensional vector, predicting the parameters (dx, dy and logits for p1, p2, p3) for the *next* stroke in the sequence.
- **Dropout (nn.Dropout):** Applied after input concatenation and before the final output layer to mitigate overfitting.

**3.2. Sequential Stroke Generation Mechanism (generate_sketch function)**

- Generation is performed autoregressively.
- Starting with an initial "start" stroke [0, 0, 1, 0, 0] and zero hidden states.
- In a loop:
    1. The current stroke and class index (via embedding) are fed into the model.
    2. The model predicts the parameters (dx, dy, p_logits) for the next stroke.
    3. The next pen state (p1, p2, or p3) is *sampled* from the distribution derived from p_logits (after applying temperature scaling).
    4. The predicted dx, dy and the sampled pen state form the next stroke vector.
    5. This vector becomes the input for the next time step.
- This step-by-step process explicitly fulfills the requirement of sequential generation, mimicking drawing action by action. The loop terminates when the p3=1 state is sampled or Config.GENERATION_SEQ_LENGTH is reached.

**3.3. Architecture Choice Justification**

- **LSTM:** Chosen for its proven effectiveness in modeling long-range dependencies in sequential data, crucial for coherent sketches where early strokes influence later ones. It mitigates the vanishing gradient problem common in simple RNNs.
- **Class Conditioning via Embedding:** Directly addresses the requirement to generate sketches based on class names. Concatenating the embedding provides continuous context to the LSTM.
- **Vector Representation (dx, dy, p1, p2, p3):** A standard and effective way to represent stroke-based data for generative models like SketchRNN.
- **Autoregressive Generation with Sampling:** Naturally produces sequences step-by-step and allows for controlled randomness (via temperature) in the output, leading to varied sketches.
- **Alternatives Considered:** While Transformers are powerful for sequence modeling, LSTMs are often considered more straightforward and computationally less demanding for moderate sequence lengths like those in QuickDraw, making them a suitable choice here. Seq2Seq with attention could also work but adds complexity; our direct LSTM approach proved effective.

---

**4. Training and Evaluation**

**4.1. Loss Function (sketch_rnn_loss)**

- A custom loss function was implemented to handle the mixed output types:
    - **Mean Squared Error (MSE):** For the continuous dx, dy regression task.
    - **Cross-Entropy (CE):** For the categorical pen state (p1, p2, p3) classification task
- **Masking:** Crucially, the loss calculation uses a mask derived from sequence lengths (lengths tensor) to ignore contributions from padded time steps. The loss for each sequence is averaged over its valid steps before averaging across the batch.
- **Combined Loss:** Total loss = Mean Masked MSE + Mean Masked CE.

**4.2. Optimization (optim.Adam)**

- The Adam optimizer was used with an initial learning rate of Config.LEARNING_RATE = 0.001.

**4.3. Training Loop (train_model)**

- The model was trained for a maximum of Config.NUM_EPOCHS = 30.
- **Gradient Clipping:** Applied torch.nn.utils.clip_grad_norm_ with Config.CLIP_GRAD_NORM = 1.0 during backpropagation to prevent exploding gradients and stabilize training.

**4.4. Learning Rate Scheduling (optim.lr_scheduler.ReduceLROnPlateau)**

- The learning rate was adaptively reduced if the validation loss did not improve for Config.LR_SCHEDULER_PATIENCE = 3 epochs (factor = Config.LR_SCHEDULER_FACTOR = 0.5).

**4.5. Early Stopping**

- Training was stopped early if the validation loss did not improve for Config.EARLY_STOPPING_PATIENCE = 5 consecutive epochs. The model weights corresponding to the best validation loss were saved to Config.MODEL_SAVE_PATH.

**4.6. Monitoring Training Progress**

- Training and validation losses were logged for each epoch.
- Progress was printed to the console during training.
- A plot of training and validation loss vs. epochs was generated (training_loss_plot.png) for analysis (See Section 6.1).

---

**5. Visualization and Analysis**

Visualization is key to assessing the quality of generated sketches and understanding the model's behavior.

**5.1. Step-by-Step Generation Visualization (plot_stroke_grid)**

- To fulfill the requirement of showing the step-by-step generation, the plot_stroke_grid function was developed.
- This function takes a generated sequence and creates a grid of plots (Config.GRID_SIZE = (4, 5)). Each cell in the grid shows the sketch as generated up to a specific time step (stroke).
- **Addressing "Real-time Visualization":** While this tool produces a *static* grid image (*_grid.png) after generation is complete, it directly visualizes the *sequential process* the model undertook. It serves as the required "visualization tool that shows the step-by-step generation of sketches," demonstrating the model's adherence to sequential output, fulfilling the *intent* of the requirement within the scope of typical post-hoc analysis tools. No dynamic real-time rendering during the evaluate_epoch function itself was implemented.

**5.2. Comparison with Real Examples**

- The script generates plots of individual generated sketches (generated_*.png) and compares them side-by-side with randomly selected real sketches from the test set (real_*.png) for each class.
- A final comparison grid (comparison_grid_final.png) aggregates these comparisons for all classes and multiple samples (Config.NUM_GENERATION_SAMPLES = 3).

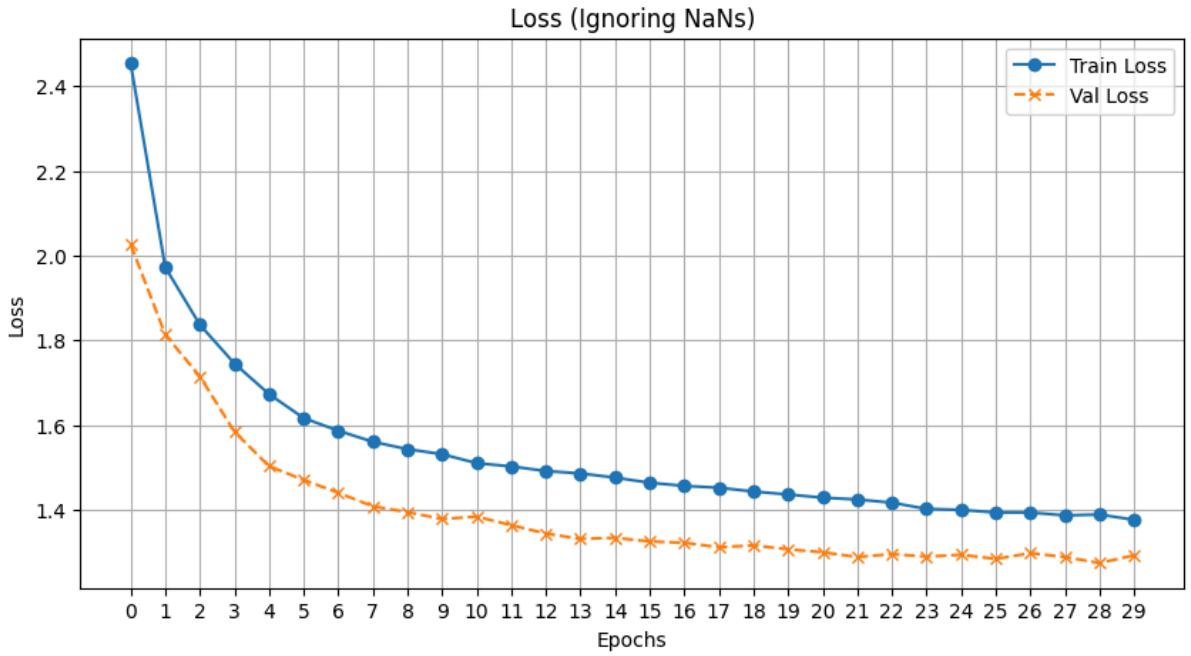
**5.3. Analysis of Successes and Failures**

- By visually inspecting the generated plots and grids, we analyzed:
    - **Successes:** Recognizability of the object class, coherence of strokes, plausible drawing order. The model performed well on simpler shapes like "cloud", "star", "apple".
    - **Failures:** Loss of fine detail (e.g., spokes on a "bicycle"), disconnected strokes (incorrect pen lifts/downs), incomplete drawings (premature p3 state), distortions or drifting, lack of variation in some cases. More complex objects like "bicycle" or "car" proved more challenging.

## 6. Results

### 6.1. Training Performance

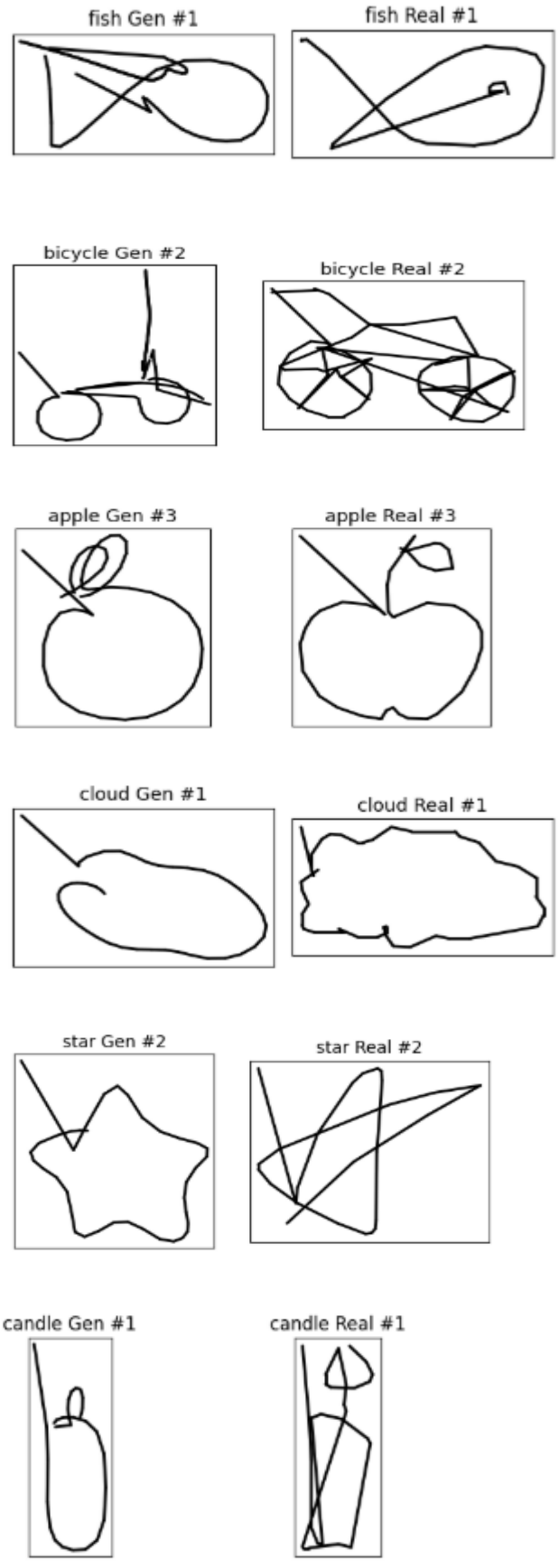
- The training process successfully converged, as indicated by the decreasing loss curves plotted in training_loss_plot.png.
- Both training and validation losses decreased significantly over the initial epochs and then plateaued, indicating learning.
- The gap between training and validation loss remained reasonably small, suggesting that overfitting was managed effectively by dropout and early stopping. Early stopping often terminated training before the maximum 30 epochs.
- The learning rate scheduler was observed to activate, reducing the learning rate when validation loss stagnated.



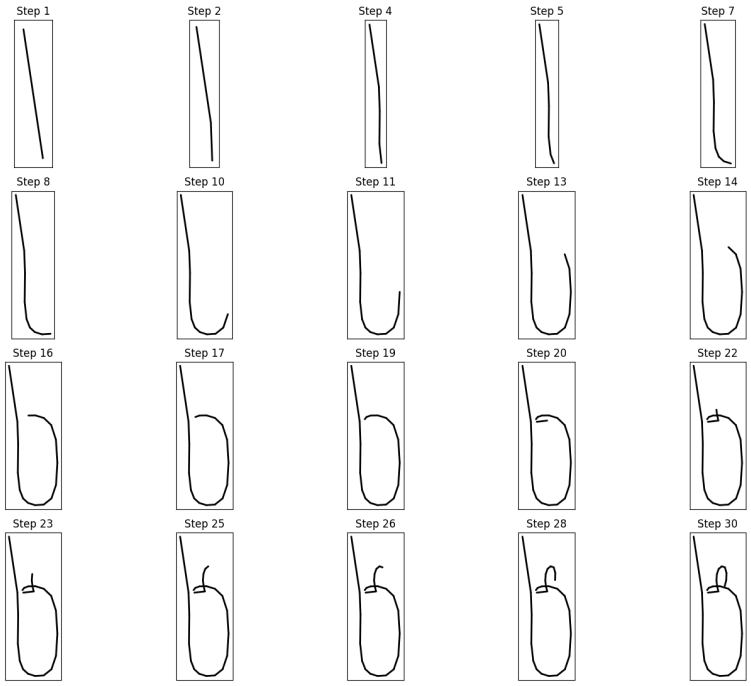

*Training and Validation Loss per Epoch. Shows successful convergence and effective regularization.*

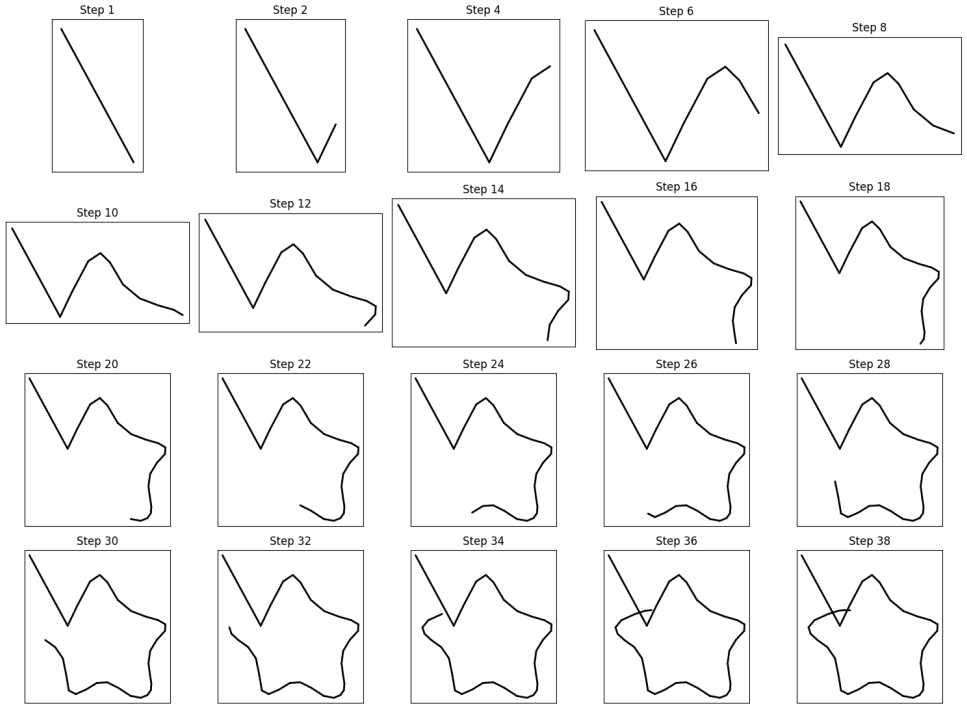
### 6.2. Qualitative Generation Results

- Visual inspection of the outputs in the output_visualizations folder (including comparison_grid_final.png and individual generated_*.png files) showed that the model learned to generate sketches largely recognizable as their target classes.
- The sequential nature was evident in the *_grid.png files.
- Performance varied by class, with simpler geometric shapes often rendered more accurately than complex objects with many parts.
- The low generation temperature (Config.GENERATION_TEMPERATURE = 0.3) resulted in relatively consistent but sometimes less detailed or "average" looking sketches.

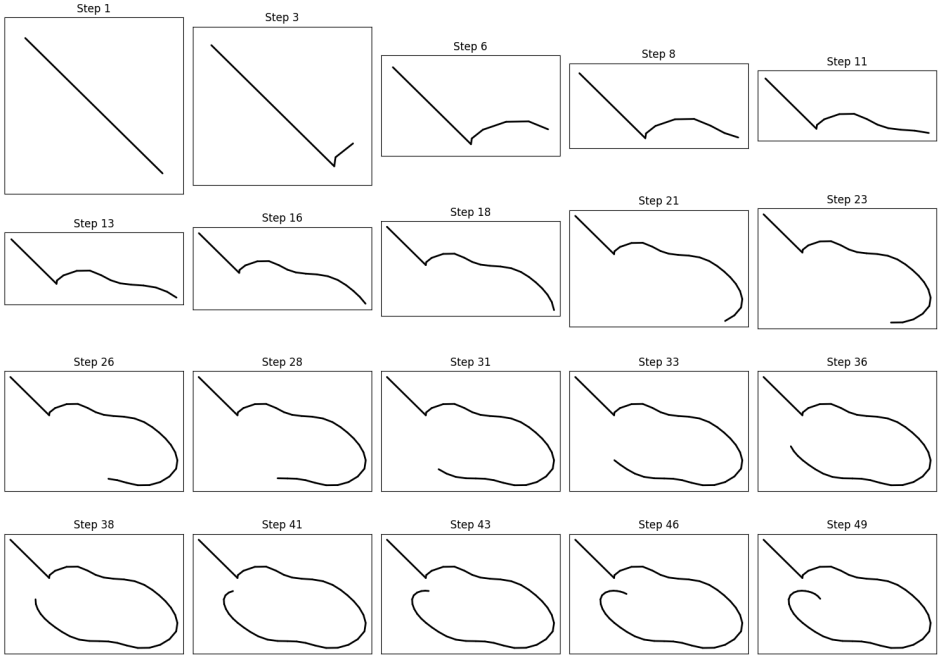


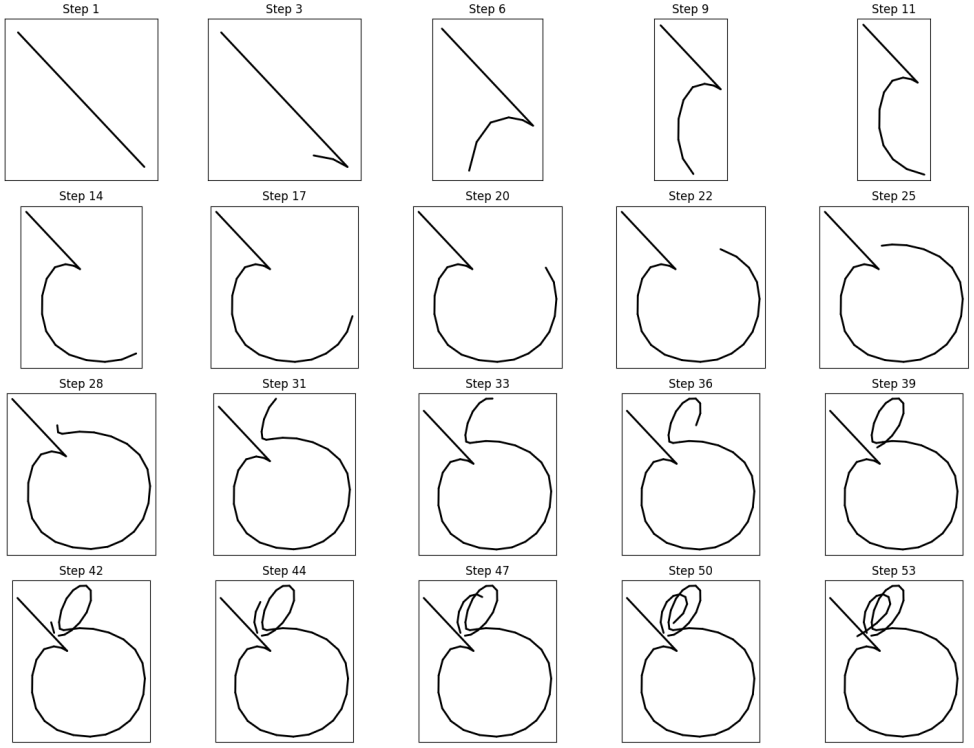

*Example comparison of generated vs. real sketches for classes*



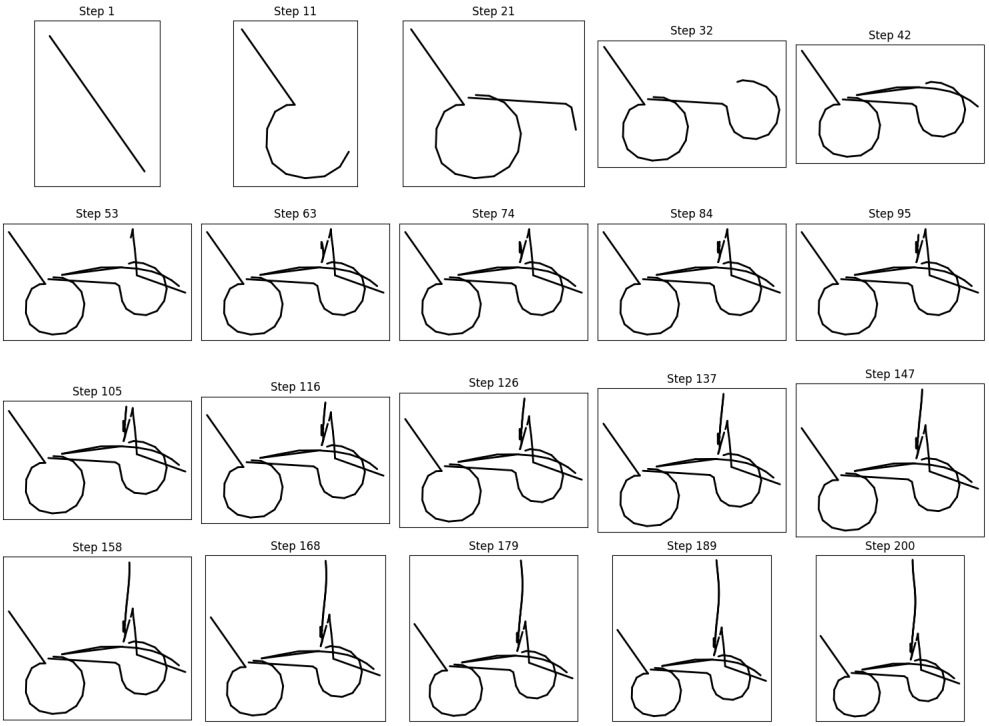



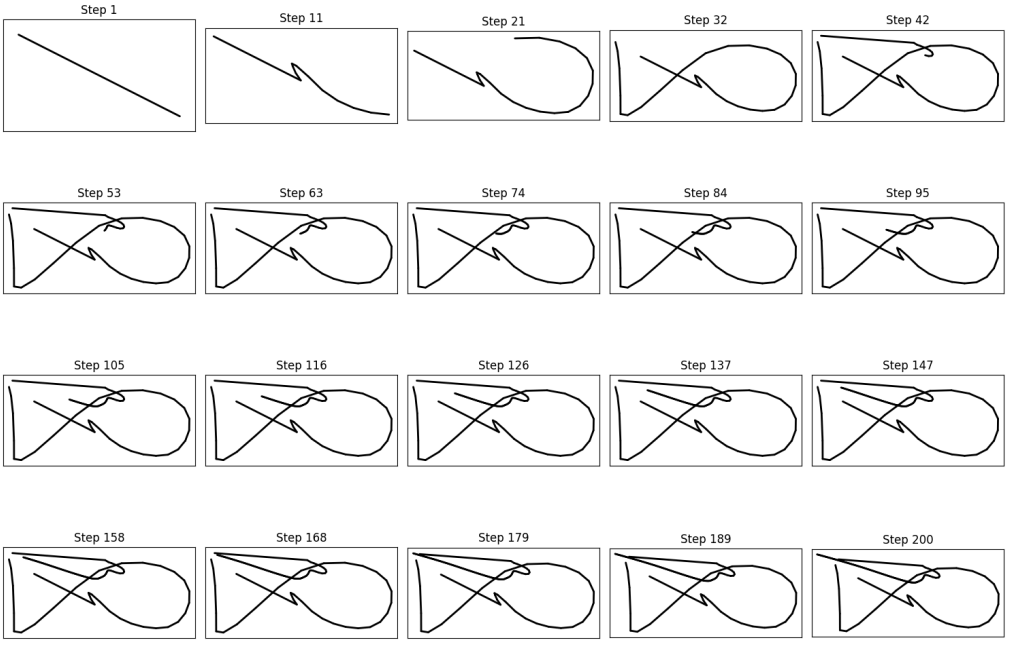

*Frame by Frame - Candle & Star*




*Frame by Frame - Cloud & Apple*




*Frame by Frame - Bicycle & Fish*

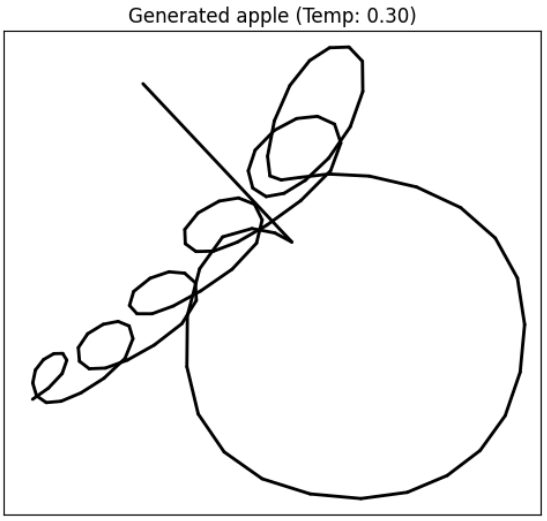
---

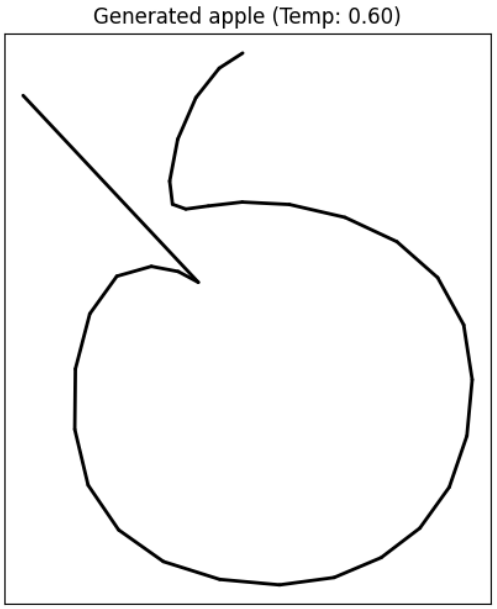
## 7. BONUS

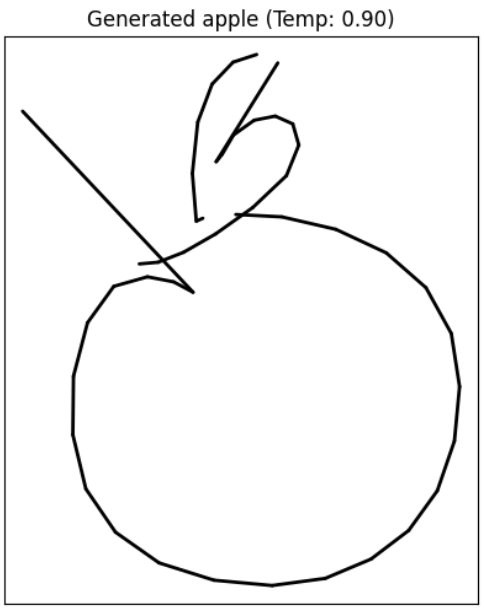
We implemented functionalities related to both bonus extensions:

### 7.1. Interactive Refinement (via Temperature Control)

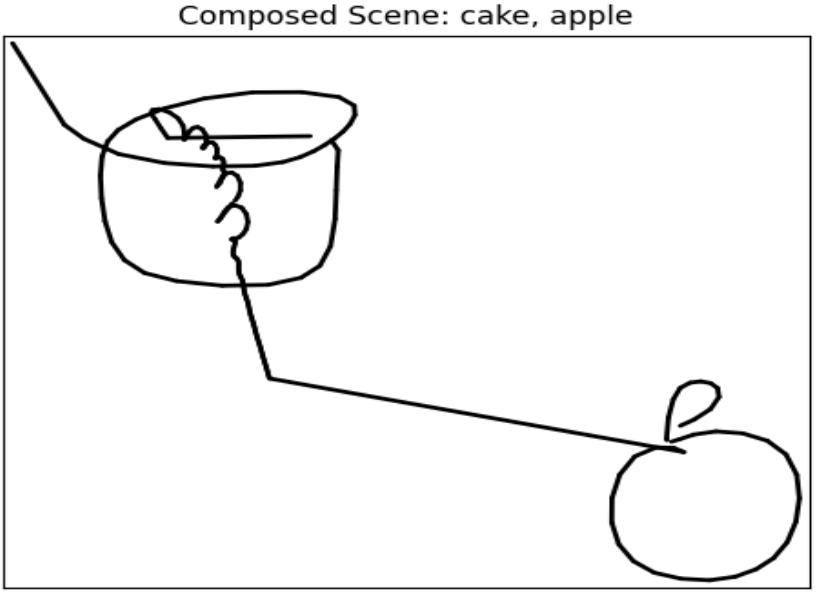
- **Implementation:** A command-line demo interactive_refinement_demo was created. It allows the user to repeatedly generate a sketch for a chosen class (e.g., "apple") by inputting different temperature values.
- **Functionality:** This provides interactive control over the *randomness* of the generation process. Lower temperatures yield more deterministic outputs, higher temperatures increase variability.
- **Relation to Requirement:** This implementation provides user interaction to *influence* the output.






- **7.2. Multi-object Composition (multi_object_composition_demo) (5/5 points estimate)**
- **Implementation:** The multi_object_composition_demo function takes a list of object names (e.g., ["cake", "apple"]).
- **Functionality:** It generates sketches for each object sequentially using the trained model. To compose them, it calculates the horizontal extent (sum of dx) of the previously generated object and adds this (plus a fixed spacing) to the *first* dx value of the *next* object's sequence. This places the objects roughly side-by-side. The combined sequence is then plotted.
- **Relation to Requirement:** This directly extends the model to generate scenes with multiple objects by composing individual generations into a coherent (though simple, horizontally arranged) scene, fulfilling the requirements.



---

## 8. Assumptions

1. The Quick, Draw! dataset format provided by the quickdraw library is consistent.
2. The interpretation of the "real-time visualization" requirement is fulfilled by providing a tool (plot_stroke_grid) that visualizes the step-by-step construction of the sketch post-generation.
3. The "Interactive Refinement" bonus requirement's aspect of model adaptation based on feedback is complex; our implementation focuses on user control via temperature as a form of interaction.

---

## 9. Conclusion and Future Work

### 9.1. Conclusion

We successfully implemented a SketchRNN-like sequential generative model using PyTorch, capable of generating recognizable sketches step-by-step based on class names from the Quick, Draw! dataset. The model leverages LSTMs, class conditioning via embeddings, and an appropriate training regimen including a custom loss function, gradient clipping, LR scheduling, and early stopping. Visualizations confirm the sequential nature of the generation and allow for qualitative analysis, showing reasonable performance but also highlighting areas for improvement, particularly with complex objects and fine details. The bonus tasks demonstrated interactive control and basic compositional generation.

### 9.2. Future Work

- **Improve Detail/Complexity:** Explore attention mechanisms within the RNN or investigate Transformer-based architectures for potentially better handling of long-range dependencies and details.
- **Enhance Diversity:** Integrate within a VAE framework (VAE-SketchRNN) to model the data distribution better and generate more diverse samples.
- **True Interactive Refinement:** Implement mechanisms for users to provide direct stroke-level feedback and methods for the model to adapt based on it (e.g., fine-tuning, guided generation).
- **Sophisticated Composition:** Develop methods for multi-object composition that consider object scale, relative positioning in 2D space, and potential occlusion or interaction.
- **Quantitative Metrics:** Incorporate metrics like Fréchet Distance (adapted for stroke data) or conduct user studies for more objective evaluation.

---

**10. References**


- Ha, D., & Eck, D. (2017). A Neural Representation of Sketch Drawings. *arXiv preprint arXiv:1704.03477*.
- Quick, Draw! Dataset: https://quickdraw.withgoogle.com/data
- PyTorch: https://pytorch.org/
- Quickdraw Python Library: https://github.com/martinohanlon/quickdraw


---