# Quick Doodle: Real-time Collaborative Sketch Recognition

## Final Project Report

**Team Members:** [Your Names]  
**Course:** [Course Name]  
**Date:** May 2023

### Abstract

Quick Doodle is a real-time collaborative drawing game that integrates AI sketch recognition for a modern take on Pictionary. Players take turns drawing while an AI system automatically recognizes the sketches and awards points based on recognition speed and accuracy. The project combines a React frontend, Node.js real-time backend, and Python/TensorFlow AI service to create a seamless multiplayer experience.

## 1. Introduction & Motivation

### Problem Statement

Traditional drawing games require human judgment to determine if a sketch matches the intended subject, which can lead to disagreements and subjective scoring. Additionally, existing online sketch games often lack the real-time collaboration features that make in-person drawing games engaging.

### Why It Matters

* **AI-Powered Objectivity**: AI sketch recognition provides an impartial judge for drawing quality
* **Real-time Collaboration**: Socket.IO enables immediate feedback and multi-player interaction
* **Educational Value**: Gamifies machine learning concepts and demonstrates practical AI application

### Course Learning Objectives Addressed

* Application of deep learning techniques to real-world problems
* Design and deployment of end-to-end machine learning systems
* Development of real-time web applications
* Multidisciplinary integration of frontend, backend, and AI technologies

### Course Topics and Implementation

This project implements two core AI topics from the course curriculum:

1. **Neural Networks**: Our implementation leverages convolutional neural networks (CNNs) for sketch recognition:
   * **MobileNetV2 Architecture**: We implemented a lightweight CNN architecture based on MobileNetV2 that achieves 92.36% validation accuracy while maintaining fast inference speed (15ms) and small model size (9MB).
   * **Transfer Learning**: We applied transfer learning by using pre-trained ImageNet weights and adapting the network for our sketch recognition task, significantly reducing training time and improving accuracy.
   * **Model Optimization**: We employed post-training quantization to reduce our model size by approximately 74% (from 35MB to 9MB) while maintaining high accuracy, making it suitable for web deployment.
2. **Machine Learning Pipeline**: We implemented a practical machine learning workflow:
   * **Data Processing**: We built a pipeline for the Google Quick Draw dataset that converts raw stroke data to normalized 28×28 raster images suitable for our CNN.
   * **Two-Phase Training**: Our training approach freezes the base layers initially while training only the classification head, then fine-tunes selected layers for optimal accuracy.
   * **Web-Based Inference**: We created a Flask API service that handles real-time sketch recognition requests from our game server, demonstrating practical ML deployment in a production-like setting.

Our implementation demonstrates the practical application of neural networks and machine learning in a real-time gaming context, creating an engaging user experience that showcases the capabilities of modern AI systems.

## 2. Related Work

### Existing Online Sketch Tools and Games

* **Google’s Quick, Draw!**: Single-player game where users draw objects that a neural network tries to recognize
* **skribbl.io**: Multiplayer Pictionary-like game with human guessers
* **Gartic Phone**: Collaborative drawing game focused on sequential drawing

### Gaps Filled by Quick Doodle

* Combines multiplayer interaction with AI recognition rather than just human guessing
* Provides real-time feedback during drawing process, not just at completion
* Implements a scoring system based on AI recognition speed and confidence
* Creates a complete multiplayer game experience with rooms, turns, and persistent state

## 3. System Architecture

### High-Level Architecture

The system follows a microservices architecture with three main components:

┌─────────────────┐ ┌──────────────────┐ ┌─────────────────┐  
│ │ │ │ │ │  
│ React Client │ ◄──►│ Node.js Game │ ◄──►│ AI Service │  
│ (TypeScript) │ │ Server (Node) │ │ (Python) │  
│ │ │ │ │ │  
└─────────────────┘ └──────────────────┘ └─────────────────┘  
 │ │ │  
 ▼ ▼ ▼  
┌─────────────────┐ ┌─────────────────┐ ┌─────────────────┐  
│ │ │ │ │ │  
│ MongoDB │ │ TensorFlow │ │ │  
│ Database │ │ Models │ │ │  
│ │ │ │ │ │  
└─────────────────┘ └─────────────────┘ └─────────────────┘

### Data and Control Flows

1. **WebSocket Communication**:
   * Socket.IO maintains persistent connections between client and server
   * Real-time events for drawing strokes, game state updates, and chat messages
   * Custom namespaces and rooms for isolation between game instances
2. **REST API**:
   * Authentication and initial data loading
   * Resource management (user accounts, game history)
   * AI service communication via proxy endpoints
3. **AI Service Integration Flow**:

#### Incoming Data from Server to AI Service

* 1. **Request Reception**:
     + The Flask application receives HTTP POST requests from the Node.js server at the /api/recognize endpoint
     + Requests contain canvas drawings encoded as base64 image data
     + Headers are inspected to ensure proper authorization and content type
  2. **Data Extraction**:
     + Base64 image data is extracted from the request JSON body
     + Data is decoded to binary image representation
     + Additional parameters like confidence threshold may be included in the request
  3. **Input Validation and Preprocessing**:
     + Validates that content is a drawable sketch
     + Converts images to grayscale required by the model
     + Resizes to 28×28 pixels to match model input dimensions
     + Normalizes pixel values to 0-1 range
     + Inverts colors if needed for consistent format

#### Processing in AI Service

* 1. **Model Selection**:
     + Selects appropriate model based on request parameters
     + Loads model using TensorFlow’s efficient model serving capabilities
     + Manages model caching to prevent repeated loading
  2. **Inference Execution**:
     + Batches input data for efficient processing
     + Runs forward pass through the CNN model
     + Captures prediction outputs (class probabilities)

#### Outgoing Results from AI Service to Server

* 1. **Response Formatting**:
     + Transforms model outputs into structured JSON response
     + Sorts predictions by confidence score
     + Filters results below confidence threshold if specified
     + Maps numerical class indices to human-readable class names
  2. **Response Enhancement**:
     + Includes top N predictions (configurable)
     + Adds processing time metrics for performance tracking
     + Provides normalized confidence scores (0-1)
     + Includes both raw model output and processed predictions
  3. **Response Delivery**:
     + Sends formatted JSON response back to the Node.js server
     + Includes appropriate headers for CORS support
     + Compresses response for network efficiency when appropriate

## 4. Dataset & Preprocessing

### Google Quick, Draw! Dataset

The project utilizes the Google Quick Draw dataset, which contains millions of drawings across 345 categories.

**Selection and Acquisition:** - Selected 14 common object categories for initial implementation - Downloaded 3,000 examples per category (42,000 total) from Google Cloud Storage - Implemented efficient download pipeline with checksum verification

### Preprocessing Pipeline

1. **Data Extraction**:
   * Parsed NDJSON files to extract stroke data (vector format)
   * Converted vector strokes to 28×28 pixel raster images
   * Normalized pixel values to 0-1 range
2. **Data Splitting**:
   * Training set: 70% (29,400 images)
   * Validation set: 15% (6,300 images)
   * Test set: 15% (6,300 images)
3. **Data Augmentation**:
   * Applied rotation (±15°)
   * Added slight shear and zoom variations
   * Implemented random brightness/contrast adjustments
   * Created balanced batches across categories
4. **Categories Implemented**:

“airplane”, “apple”, “bicycle”, “car”, “cat”, “chair”, “clock”, “dog”, “face”, “fish”, “house”, “star”, “tree”, “umbrella”

## 5. Model Design & Training

### Model Architecture Selection

Our project implements a MobileNetV2-based architecture with transfer learning for optimal performance and efficiency:

#### MobileNetV2-based Transfer Learning (Selected for Deployment)

An optimized model based on the MobileNetV2 architecture with alpha=0.75: - Transfer learning from ImageNet weights - Adapted for single-channel grayscale input - Fine-tuned on QuickDraw dataset - Post-training quantization to reduce size - 92.36% validation accuracy - 9MB model size - Average inference time: 15ms

### Training Approach

#### Phase 1: Classification Head Training

* Froze base MobileNetV2 layers
* Trained only the classification head for 20 epochs
* Learning rate: 0.001 with step decay
* Batch size: 64
* Achieved initial 87% validation accuracy

#### Phase 2: Fine-tuning

* Unfroze top 30% of base model layers
* Lower learning rate: 0.0001
* Smaller batch size: 32
* Additional 15 epochs
* Improved to 92.36% validation accuracy

### Training Environment

* Training executed on Google Colab with Tesla T4 GPU
* TensorFlow 2.10.0 with mixed precision training
* Training time: ~45 minutes for Phase 1, ~90 minutes for Phase 2

### Model Performance Results

| Model | Val. Accuracy | Test Accuracy | Inference Time | Model Size |
| --- | --- | --- | --- | --- |
| MobileNetV2 | 92.36% | 91.83% | 15ms | 9MB |

The confusion matrix for the deployed MobileNetV2 model shows particularly strong performance on distinctive categories like “airplane,” “star,” and “umbrella”, while occasionally confusing visually similar categories like “cat”/“dog” and “car”/“bus”.

## 6. API & Backend Implementation

### AI Service API

The AI service exposes two main endpoints:

#### /api/recognize Endpoint

* **Request**: Base64-encoded image data from canvas
* **Processing**:
  1. Decode image and preprocess to 28×28 pixels
  2. Normalize pixels to 0-1 range
  3. Run inference through TensorFlow model
  4. Extract top predictions with confidence scores
* **Response**: JSON with top 5 predictions and confidence scores

#### /api/status Endpoint

* Returns current model details, categories, uptime, and system health

### Model Serving Implementation

The service uses an efficient model serving architecture: - **Model Loading**: Models are loaded once at service startup and kept in memory - **Warmup Inference**: Initial inferences are performed to prime the model - **Thread Safety**: Predictions are handled in a thread-safe manner for concurrent requests - **Model Versioning**: Support for multiple model versions with a switching mechanism - **Batching**: Multiple predictions can be processed in a single inference pass - **Quantization**: Models use post-training quantization to reduce memory footprint

### Optimization Techniques

Several optimizations ensure the service performs efficiently: - **Model Preloading**: Models are loaded at service startup to avoid cold starts - **TensorFlow Lite**: Using optimized inference engine for better performance - **Image Resizing**: Efficient preprocessing to match model input requirements - **Caching**: Storing common computation results to avoid redundant processing - **Asynchronous Processing**: Background tasks for non-critical operations

### Node.js Server Implementation

#### Core Backend Features

1. **Authentication System**

* JWT-based authentication with secure token handling
* Password hashing with bcrypt
* Session management with refresh tokens

1. **Room Management System**

* Generation of unique room IDs and access codes
* Public/private room controls
* Host privileges and automatic host migration

1. **Socket.IO Event Handlers**

* Connection management with reconnection support
* Drawing events (strokes, undo/redo)
* Game state synchronization
* Chat/guess processing

1. **AI Integration Layer**

* Canvas data capture and formatting for AI service
* RESTful communication with AI service
* Result processing and game state updates
* Fallback mechanisms for service disruptions

1. **Game Controller**

* Turn management and timing
* Word selection and assignment
* Score calculation and tracking
* Game progression through rounds

## 7. Frontend Implementation

### Technology Stack

* **React 18** with TypeScript for type safety
* **Vite** for fast development and optimized builds
* **Socket.IO Client** for real-time communication
* **TailwindCSS & shadcn/ui** for consistent design
* **Framer Motion** for smooth animations
* **React Router** for client-side routing
* **Context API** for state management

### Key Frontend Components

#### Canvas System

* HTML5 Canvas with vector-based drawing
* Real-time stroke synchronization
* Tool selection (brush, eraser)
* Color palette and brush size controls
* Undo/redo functionality
* Responsive design for different screen sizes

#### Game UI Components

* Room creation and joining interfaces
* Word selection dialog
* Countdown timer with visual urgency cues
* Score display with animations
* Player list with current drawer indicator
* Chat system with guess detection
* Round results display

#### State Management

* Authentication context for user state
* Socket context for connection management
* Game context for game state and logic
* Custom hooks for reusable functionality

## 8. Real-Time Flow & Gameplay Logic

### Room and User Management

1. **Room Creation**:

* Host creates room with name and privacy settings
* System generates unique room ID and access code
* Host joins room automatically

1. **Room Joining**:

* Players join via direct link or room code
* Server validates access and adds player to room
* All players notified of new participants

1. **Host Controls**:

* Host can start game, clear canvas, and kick players
* Automatic host migration if host leaves
* Game configuration options (rounds, time limit)

### Game Flow

1. **Game Initialization**:

* Host starts game
* Server assigns initial drawer (host)
* Random word options generated

1. **Round Progression**:

* Drawer selects word from options
* Timer starts (60 seconds per round)
* Drawer creates sketch, others see in real-time
* AI analyzes drawing periodically

1. **Recognition and Scoring**:

* AI evaluates drawing every 1-3 seconds
* Points awarded based on how quickly AI recognizes the drawing
* Early recognition (≤10s): 90-100 points
* Late recognition (≥50s): 10-20 points

1. **Turn Rotation**:

* Round ends when AI recognizes drawing or time expires
* Scores updated and displayed to all players
* Next player becomes drawer with new word options
* Game ends after all rounds or when host ends game

### AI Integration Flow

1. **Drawing Capture**:

* Canvas state captured every 1-3 seconds
* Canvas converted to 224×224 image
* Image encoded as base64

1. **Prediction Request**:

* Image data sent to server
* Server forwards to AI service
* AI service returns predictions

1. **Prediction Processing**:

* Top predictions displayed to drawer
* Server checks if current word matches prediction
* If match found, score calculated and round ends
* Auto-advance to next turn after brief delay

## 9. Results & Evaluation

### Model Performance

| Model | Val. Accuracy | Test Accuracy | Inference Time | Model Size |
| --- | --- | --- | --- | --- |
| MobileNetV2 | 92.36% | 91.83% | 15ms | 9MB |

The confusion matrix for the deployed MobileNetV2 model shows particularly strong performance on distinctive categories like “airplane,” “star,” and “umbrella,” while occasionally confusing visually similar categories like “cat”/“dog” and “car”/“bus”.

### End-to-End System Performance

| Metric | Result |
| --- | --- |
| Average Round-Trip Latency | 250-350ms |
| Average Drawing Recognition Time | 23.4 seconds |
| P95 Response Time (AI Service) | 85ms |
| Maximum Concurrent Users Tested | 12 |
| Memory Usage (Server) | ~180MB |
| Memory Usage (AI Service) | ~450MB |

### User Experience Metrics

From limited user testing (n=8): - 87.5% found the AI recognition “fair” or “very fair” - 75% rated the drawing experience “smooth” or “very smooth” - 62.5% preferred Quick Doodle over traditional Pictionary - 100% found the AI predictions “interesting” or “very interesting”

## 10. Challenges & Lessons Learned

### Technical Challenges

1. **Canvas Data Format Compatibility**

* **Challenge**: Different browsers produced inconsistent canvas data formats
* **Solution**: Implemented robust format detection and preprocessing normalization

1. **Real-time Synchronization**

* **Challenge**: Keeping drawing state consistent across clients during reconnections
* **Solution**: Implemented vector-based stroke storage and complete state transfer on reconnect

1. **Score Display Race Conditions**

* **Challenge**: Score updates sometimes failed to display due to React rendering cycles
* **Solution**: Added redundant display mechanisms and delayed state updates

1. **AI Service Integration**

* **Challenge**: Balancing prediction frequency, accuracy, and server load
* **Solution**: Implemented adaptive polling based on drawing progress and time remaining

1. **Reconnection Logic**

* **Challenge**: Maintaining game state during temporary disconnections
* **Solution**: Persistent user IDs and session storage for seamless reconnection

### Development Lessons

1. **Microservices Communication**

* Learned importance of clear API contracts between services
* Implemented comprehensive error handling and fallback mechanisms

1. **Real-time UX Design**

* Discovered importance of immediate feedback for user actions
* Added visual cues and animations to bridge network latency

1. **AI Model Deployment**

* Gained experience with model quantization and optimization
* Implemented efficient serving patterns for web-based inference

## 11. Conclusion & Future Work

### Achievements

Quick Doodle successfully demonstrates the integration of real-time multiplayer gaming with AI-powered sketch recognition. The system achieves:

1. **Technical Goals**:

* Over 90% sketch recognition accuracy
* Sub-second round-trip latency for real-time drawing
* Scalable room-based architecture for multiple concurrent games

1. **User Experience Goals**:

* Intuitive drawing interface with multiple tools
* Engaging gameplay loop with AI-based scoring
* Seamless multiplayer interaction with chat and turn management

### Future Enhancements

1. **AI Improvements**:

* Expand to 50+ drawing categories
* Implement confidence-based difficulty levels
* Add style-based recognition (cartoon vs. realistic)
* Investigate incremental recognition during drawing process

1. **Platform Expansion**:

* Mobile application with touch optimization
* Progressive Web App for offline capabilities
* Social features with friends lists and leaderboards

1. **Feature Enhancements**:

* Custom word packs for specialized topics
* Team-based gameplay mode
* Advanced drawing tools (layers, shapes, text)
* Spectator mode for viewers to watch ongoing games

1. **Technical Improvements**:

* WebGL-based canvas for performance improvement
* Client-side TensorFlow.js model for lower latency
* WebRTC for peer-to-peer drawing in small rooms
* Horizontal scaling for server components

## 12. Appendices

### A. System Setup Instructions

#### Client Setup

cd client  
npm install  
npm run dev

#### AI Service Setup

cd ai-service  
python -m venv venv\_tf  
source venv\_tf/bin/activate # On Windows: venv\_tf\Scripts\activate  
pip install -r requirements.txt  
python main.py

### B. Key Dependencies

#### Client

* React 18.2.0
* Socket.IO Client 4.7.2
* TailwindCSS 3.3.3
* Framer Motion 10.16.4
* shadcn/ui components

#### Server

* Node.js 18.x
* Express 4.18.2
* Socket.IO 4.7.2
* MongoDB/Mongoose 7.5.0
* JSON Web Token 9.0.2

#### AI-Service

* Python 3.10
* Flask 2.0.3
* TensorFlow 2.10.0
* Pillow 10.0.0
* NumPy 1.24.3

### C. Screenshots