Technical Report: Doodle Classifier

Deep Neural Network for QuickDraw Doodle Classification Using Pure Python and NumPy

By: Bilal Ahmad

1. Introduction

This report outlines my mathematical implementation of a feedforward deep neural network (DNN) developed using pure Python and NumPy. The network is trained on a subset of the Quick, Draw! dataset to perform multi-class classification of human-drawn doodles. It discusses the complete architecture, including dense layers, activation functions, dropout, loss function, backpropagation, and optimization using the Adam algorithm. All core components are analyzed mathematically with thorough explanations of their purposes and effects.

2. Dataset Description

The **Quick, Draw!** dataset is a massive collection of human-drawn doodles, containing millions of 28×28 grayscale images across hundreds of object categories. For this project, we selected a subset containing **5 specific classes**:

- Headphones
- Bucket
- Clock
- Pizza
- Hexagon

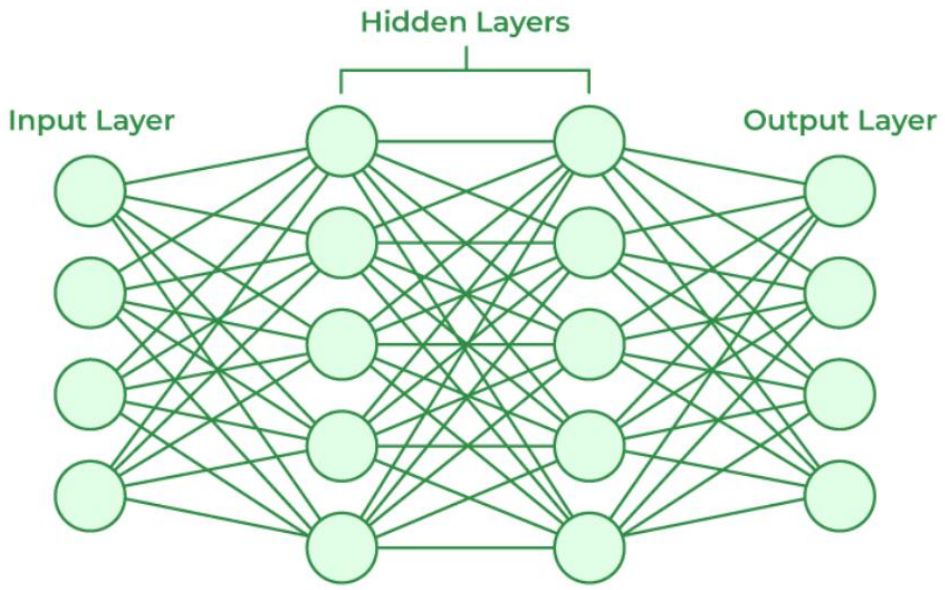
Each sketch is centered and resized to a **28×28 pixel bitmap**, then flattened into a **784-dimensional input vector**. All pixel values are **normalized to the range [0, 1]** by dividing by 255. The complete dataset is divided into training and test sets using an 80:20 split. The diversity and real-world imperfection in human drawings make this a challenging and realistic image classification task.

3. Network Architecture

Input Layer : 784 (28x28 Image)

Neurons in Hidden Layer: 128

Outputs: 5



Neural Networks Architecture

Note: For visual simplicity, the layers are shown with fewer neurons. For instance, the actual input layer has 784 neurons corresponding to the 28×28 pixel image.

Layers	**Neurons**	**Activation**	**Initialization**	**Regularization**	**Dropout**
Input	784	-	-	-	-
Hidden 1	128	ReLU	Xavier-Uniform	L2	0.8
Hidden 2	128	ReLU	Xavier-Uniform	L2	0.8
Output	10	Softmax	-	-	-

4. Mathematical Analysis

4.1 Linear Transformation (Dense layer)

The linear transformation in a dense layer combines inputs with weights and biases:

$$y = Wx + b$$

where x ∈ Rm×n is the input matrix for m samples with n features, W ∈ Rn×k is the weight matrix, b ∈ R1×k is the bias vector, and y ∈ Rm×k is the output. This operation forms the basis of each layer, enabling the network to learn patterns by adjusting W and b.

4.2 ReLU Activation

The Rectified Linear Unit (ReLU) activation introduces non-linearity:

$$f(x) = \max(0, x)$$

Its derivative, used in backpropagation, is:

$$f'(x) = \begin{cases} 1 & \text{if } x > 0 \\ 0 & \text{if } x \leq 0 \end{cases}$$

ReLU helps the network model complex relationships and mitigates vanishing gradient issues.

4.3 Softmax Activation

The softmax function converts raw scores into probabilities:

$$\sigma(z_i) = \frac{e^{z_i}}{\sum_{j=1}^{k} e^{z_j}}$$

where z ∈ Rm×k represents scores for m samples and k classes. This is used in the output layer for multi-class classification, ensuring probabilities sum to 1.

4.4 Loss (Categorical Cross-Entropy)

Categorical cross-entropy loss measures prediction accuracy:

$$L = -\sum_{c=1}^{k} y_c \log(\hat{y}_c)$$

For a batch, the average loss is:

$$L = -\frac{1}{m}\sum_{i=1}^{m}\sum_{c=1}^{k} y_{i,c} \log(\hat{y}_{i,c})$$

where yc is the true label (1 for the correct class, 0 otherwise), and yˆc is the predicted probability. This loss guides the model to improve predictions.

4.5 Backpropagation

Backpropagation computes gradients using the chain rule:

- Weights:

$$\frac{\partial L}{\partial W} = x^T \cdot \frac{\partial L}{\partial y}$$

- Biases:

$$\frac{\partial L}{\partial b} = \sum_{i=1}^{m} \frac{\partial L}{\partial y_i}$$

- Inputs:

$$\frac{\partial L}{\partial x} = \frac{\partial L}{\partial y} \cdot W^T$$

4.6 Regularization

L1 and L2 regularization prevent overfitting:

$$\text{L1: } \lambda_1 \sum |w|$$

$$\text{L2: } \lambda_2 \sum w^2$$

The total loss includes:

$$L_{\text{total}} = L_{\text{data}} + \lambda_1 \sum |w| + \lambda_2 \sum w^2$$

Gradients are adjusted:

- Weights: $\frac{\partial L}{\partial w} += \lambda_1 \cdot \text{sign}(w) + 2\lambda_2 w$
- Biases: $\frac{\partial L}{\partial b} += \lambda_1 \cdot \text{sign}(b) + 2\lambda_2 b$

4.7 Dropout

Dropout randomly sets a fraction p of inputs to 0:

$$x_{\text{dropout}} = x \cdot m/(1-p)$$

where m is a binary mask with probability 1 − p.

This improves generalization by reducing reliance on specific neurons.

4.8 Gradient Descent

Gradient descent updates parameters:

$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\partial L}{\partial \theta}$$

where η is the learning rate.

My code uses the Adam optimizer, an advanced variant.

4.9 Learning Rate Decay

Learning rate decay adjusts the learning rate:

$$\eta_t = \frac{\eta_0}{1 + \text{decay} \cdot t}$$

This helps fine-tune parameters in later training stages.

4.10 Momentum

Momentum maintains a moving average of gradients:

$$m_t = \beta m_{t-1} + (1 - \beta) g_t$$

This is incorporated into the Adam optimizer.

4.11 Adam Optimizer

The Adam optimizer uses adaptive learning rates:

- **First Moment**:
$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
- **Second Moment**:
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$
- **Bias-Corrected Moments**:
$$\hat{m}_t = \frac{m_t}{1 - \beta_1^t}, \quad \hat{v}_t = \frac{v_t}{1 - \beta_2^t}$$
- **Update**:
$$\theta_t = \theta_{t-1} - \eta \cdot \frac{\hat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Adam ensures faster and more stable convergence.

5. Inference: Predicting on Custom Doodle Images

Once the model is trained and its parameters are saved, it can be used for **inference** — i.e., making predictions on unseen hand-drawn doodles provided by users. This process requires only the **forward pass** of the network and does not involve backpropagation.

The inference pipeline consists of the following steps:

5.1 Model Loading

The learned **weights and biases** for each dense layer are loaded from the saved .pkl file (model_weights.pkl). These parameters were obtained after training the network and are essential for reproducing the model's behavior during inference. Each dense layer restores its weight matrix and bias vector to perform the same forward computations as during training.

5.2 Image Preprocessing

Before an input sketch can be classified, it must be processed into the correct format expected by the network:

- The **input image is loaded** and converted to **grayscale**, ensuring a single-channel 2D array
- It is **resized to 28×28 pixels** to match the input shape of the network (784 input features).
- If the sketch has a **white background with black strokes**, it is kept as-is. If inverted (e.g., black background), the colors are automatically inverted to match the training data format.
- The image is then **normalized to the range [0, 1]** by dividing each pixel by 255.
- Finally, the 28×28 matrix is **flattened into a 784-dimensional vector**, ready for input into the neural network.

This preprocessing ensures that any hand-drawn doodle—scanned, drawn on a canvas, or uploaded—can be properly classified by the trained model.

5.3 Forward Propagation

The input is passed sequentially through:

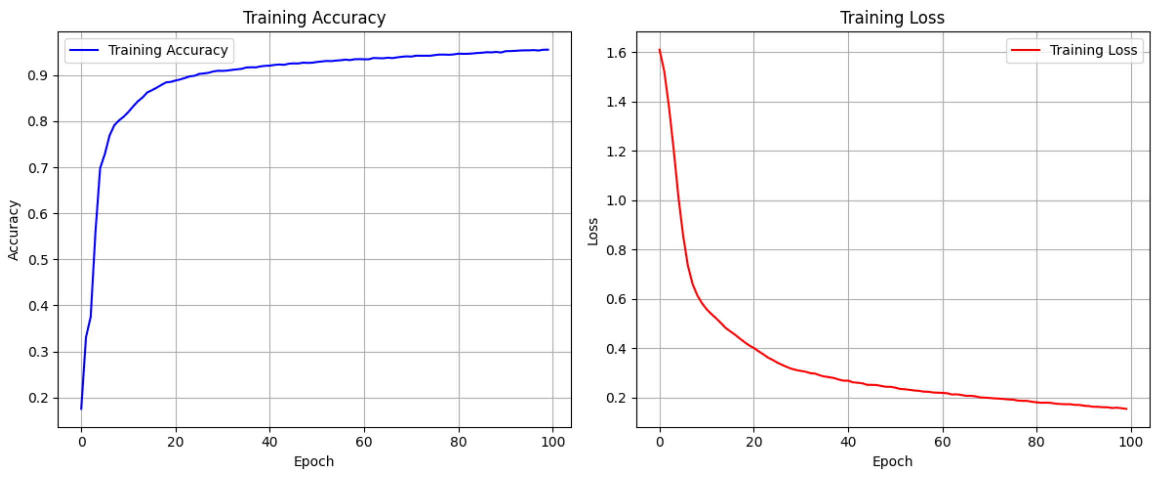
Dense → ReLU → Dense → ReLU → Dense → Softmax

The output of the softmax layer is a probability distribution over the 10 fashion categories. The predicted class corresponds to the maximum probability value.

6. Accuracy Upon Training

Testing Accuracy: **95.5%**

Training Accuracy: **93.7%**



7. References and Learning Resources

- NNs from Scratch in Python - By Santdex
- Building NNs from Scratch - By Vizuara