

ScratchNet

Neural Network from Scratch for PII Detection

A **from-scratch** multi-layer neural network built with only NumPy to classify whether a text contains Personally Identifiable Information (PII).

This project demonstrates a deep understanding of neural networks by implementing forward propagation, backpropagation, dropout, and training from the ground up — without using PyTorch, TensorFlow, or any high-level "black-box" libraries.

Why This Project?

Most modern ML tutorials use high-level frameworks that hide the math. Here, I built everything manually using only NumPy to truly understand how neural networks work under the hood.

Project Overview

- **Task:** Binary classification — Detect if masked text contains PII (1 = has PII, 0 = no PII)
- **Data:** Mendeley Data Synthetic PII document text data
- **Input:** Pre-trained GloVe word embeddings (300 dimensions)
- **Model:** Custom ScratchNet (MLP) implemented from scratch
- **Key Techniques:** He initialization, ReLU activation, Dropout, Weighted BCE loss, L2 regularization

Final Test Accuracy: 93.3%

Data Source and Transformation

The data was synthetic PII data which you can download in Excel form from [Mendeley Data](#). Once the data was downloaded I used Excel formulas and the PII feature columns to mask the PII in the document text column for roughly 3/4 of the rows for each set.

Why GloVe Embeddings?

Instead of using simple Bag-of-Words or TF-IDF, I chose **GloVe (Global Vectors for Word Representation)** because:

- It captures **semantic meaning** — words with similar meanings have similar vectors

- It handles context better than basic count-based methods
- Averaging word vectors gives a simple yet powerful fixed-size representation for entire documents
- It's a standard, well-studied embedding method from Stanford, making results more meaningful. You can download it [here](#).

This approach is especially useful for PII detection, where understanding word relationships (e.g., names, addresses, IDs) matters.

How the Model Works

1. Data Preparation

Raw text is converted into fixed-size numerical vectors using **average GloVe embeddings** (300 dimensions).

This turns each document into one vector that captures the overall meaning of the words.

2. The Neural Network (ScratchNet)

I built the entire network from scratch using only NumPy. Here's what each major part does:

Initialization - `__init__()`

- **Layer Construction:** Dynamically builds a list of dictionaries, each holding a weight matrix (W) and a bias vector (b).
- **He Initialization (W):**
Before training, the weights are initialized using He initialization:
`W = np.random.randn(prev_size, h) * np.sqrt(2.0 / prev_size)`
 - **What is He Initialization?** A method for setting the initial random weights of a neural network. It was specifically designed to solve the issues that arise when using the ReLU activation function in very deep networks.
 - **Why He Initialization?** Standard random initialization can lead to signals that shrink (vanish) or grow too large (explode). He initialization is specifically designed to keep the signal variance stable when using ReLU activations.
- **Bias Initialization (b):** Biases are set to zero. Why zero? Since the random weights already "break symmetry" (ensuring every neuron learns something different), the biases can safely start at zero and be adjusted via addition during backpropagation.

Forward Pass - `forward()`

- Takes the 300-dimensional input vector and passes it through two hidden layers (512 → 256 neurons).
- Each hidden layer uses **ReLU (Rectified Linear Unit) activation** (`np.maximum(0, Z)`).
 - **What is ReLU?** It is a piecewise linear function that outputs the input directly if it is positive; otherwise, it outputs zero.

- **Why ReLU?** It is simple, fast, and helps prevent the “vanishing gradient” problem. It also makes the model learn faster and perform better on most classification tasks like this one.
- **Inverted Dropout Regularization:** During training, the model randomly "shuts off" a certain amount (here 10%) of neurons in each hidden layer.
 - **Why Dropout?** It prevents the model from becoming too reliant on specific neurons, forcing it to learn more robust, redundant features. This is key to preventing overfitting.
 - **Scaling:** The remaining activations are scaled by $1 / (1 - \text{dropout_rate})$ so that the total "volume" of the signal stays consistent between training and testing.
- **Output Layer (Logits):** The final layer produces "raw" scores (logits) rather than probabilities.
 - This allows for a more numerically stable calculation of the loss during training.

Backward Pass/Backpropagation - backward()

- **Weighted Error Signal:** The process starts at the output layer by calculating the difference between the prediction and the truth, scaled by a `pos_weight=2.0`.
 - **Why a weighted version?** Our dataset is imbalanced (~25% positive). A standard loss would allow the model to "ignore" the rare PII cases to get a decent score. By weighting the positive class 2x, we force the model to prioritize Recall—minimizing the chance of missing actual PII.
- **The ReLU Gradient:** As the error signal moves backward, it is only passed through neurons that were "active" during the forward pass.
 - If the input to a ReLU neuron was ≤ 0 , the gradient is "killed" (set to 0), preventing the model from updating weights for inactive features.
- **Applying Dropout Masks:** The exact same "masks" (the neurons we shut off during the forward pass) are re-applied here.
 - **Why?** You cannot blame a neuron for an error if it wasn't allowed to participate in the "thinking" process during the forward pass.
- **Parameter Gradients (dW and db):** The final gradients are averaged across the batch to ensure a stable "nudge" toward the optimal solution.

Parameter Update - update_parameters()

- **Mini-batch Stochastic Gradient Descent (SGD):** Weights and biases are updated using a learning rate of 0.00375 and a `batch_size=128`.
 - **The "Goldilocks" Balance:** Mini-batch SGD sits between Stochastic (one point) and Full-Batch (all data) descent. It provides enough noise to escape "local minima" while maintaining the stability needed for smooth convergence.
- **L2 Regularization (Weight Decay):** We apply a penalty of `l2_lambda=0.005` to the weights during each update.
 - **How it works:** In every step, the weights are slightly "shrunk" toward zero before the gradient is applied ($W = W \times (1 - \text{lr} \cdot \lambda)$).

- **Why L2?** It prevents any single weight from becoming too large and "overpowering" the model. This ensures the network makes decisions based on a broad set of features rather than hyper-focusing on specific noise in the training set.
- **The Step:** $W = W - (\text{Learning Rate} \times \text{Gradient})$:
 - This is where the actual "learning" happens. The learning rate controls the size of the step we take toward the solution. Too large, and we "overshoot"; too small, and the model takes forever to train.

Inference - predict_proba()

- **The "Translator":** Converts the raw scores (logits) from the final layer into a probability between **0 and 1** using the **Sigmoid Activation Function** ($1/(1 + e^{-x})$).
 - **Probability vs. Prediction:** This method doesn't just say "Yes" or "No"; it tells us how confident the model is. For example, a result of 0.92 means the model is 92% sure the text contains PII.
- **Inference Mode Logic: - Training Toggle:** Automatically sets `self.training = False`.
 - **Why this is critical:** During inference, we **must disable Dropout**. We want the full "wisdom" of all neurons working together, not a random 90% subset. This ensures the model's predictions are consistent and reproducible.
- **Thresholding:** While `predict_proba()` provides the decimal, we typically use a threshold of **0.5** to make the final binary decision (PII Detected vs. No PII).

3. Loss Function - stable_bce_loss()

- **Weighted Binary Cross Entropy (BCE):** Measures the "distance" between the model's predicted raw scores (logits) and the true labels (0 or 1).
 - **Why BCE?** It is the standard for binary classification because it heavily penalizes "confident" wrong predictions, forcing the model to become more certain about its decisions.
 - **Class Weighting (pos_weight=2.0):** Since our PII dataset is imbalanced, we assign double the penalty to missed positive cases. This ensures the model's "compass" is pointed toward finding PII, even when it is rare.
- **Numerical Stability:** The implementation uses the Log-Sum-Exp trick to prevent "Overflow" or "Log(0)" errors.
 - **Why it matters:** Computers struggle with extremely small or large numbers. Without this stability, the model might "explode" or return NaN (Not a Number) during training.
- **The "Monitor" Role:** While the `backward()` method handles the actual math of the updates, `stable_bce_loss()` provides the human-readable metric we use to track progress.
 - **The Integral Connection:** The loss value represents the accumulation of error across the dataset. While the `backward()` method calculates the instantaneous 'slope' (gradient) to fix the weights, the loss function gives us the 'total altitude'—telling us how far we are from the bottom of the mountain.

Technologies Used

- **Core:** NumPy (only math library)
- **Embeddings:** GloVe 6B 300d (Stanford)
- **Data:** pandas
- **Visualization/Progress:** matplotlib, tqdm

No deep learning frameworks were used.

Results

Test Accuracy: 0.9333 (93.33%)

Inference time: 0.008 seconds

Classification Report

Class	Precision	Recall	F1-Score	Support
0 (No PII)	0.9545	0.9545	0.9545	22
1 (Has PII)	0.8750	0.8750	0.8750	8
Accuracy			0.9333	30
Macro Avg	0.9148	0.9148	0.9148	30
Weighted Avg	0.9333	0.9333	0.9333	30

Confusion Matrix

	Predicted No PII	Predicted Has PII
Actual No PII	21	1
Actual Has PII	1	7

How to Run

1. Clone the repository:

```
git clone https://github.com/NateBoyle/ScratchNet.git
```

```
cd ScratchNet
```
2. Install the required dependencies (if necessary):
 - `pip install numpy pandas tqdm matplotlib scikit-learn`
3. Download GloVe embeddings:
 - Download glove.6B.300d.zip from Stanford GloVe
 - Extract it and place glove.6B.300d.txt in the project root folder.
4. Open and run the notebook:
 - Launch Jupyter Notebook, VS Code, or Google Colab
 - Open ScratchNet - Demo.ipynb

Important Notes:

A small sample training dataset is included for quick testing and demonstration. The full training set (~45MB) is not included in the repo due to GitHub file size limits. Pre-trained model weights (scratchnet_weights_demo.npz) are provided, so you can skip training and directly run the testing/evaluation cells.