Building a Spam Detector Using Neural Networks Activation Functions

Rafael E Castillo Echavarría
Master in Computer Science
Advisor: Lisabel Rodríguez Espinosa, J.D.
Polytechnic University of Puerto Rico
Graduate Project EXPO, February 2026

Abstract *— This project explores the application of Artificial Neural Networks in the classification of electronic mail as either "Spam" or "Ham" (legitimate). By using activation functions, which serve as the mathematical "gates" that decide whether a neuron should fire a spam prediction. The experiment shows how automated filtering systems can greatly increase accuracy and reduce false positives by using tuning functions such as the Sigmoid and the Rectified Linear Unit (ReLU) [1].*

Artificial neural networks are strong instruments that can make difficult decisions and learn from data. They are made up of layers of linked neurons that change with practice. Using the test data, the network updates its weights and biases. Activation functions facilitate this process by allowing the model to learn from errors and get better over time.

This is the beginning of the creation of predictive generative artificial intelligence.

Key Terms *— Activation Function, Artificial Neural Network, Rectified Linear Unit, Sigmoid.*

INTRODUCTION

As individuals and businesses increasingly depend on digital communication, the volume of unwanted bulk email, or spam, is rising. Conventional rule-based filters often fall behind the ever-changing tactics used by spammers. This study uses deep learning techniques and focuses on how different neural network architecture activation functions affect spam detection performance.

Artificial Neural Networks (ANN) are logarithmic activation functions that resemble the human brain. That is, just as neurons in our nervous system can learn from past data, the ANN can learn from data and provide responses in the form of predictions or classifications. Every deep neural network consists of a linear transformation followed by an activation function, which plays a key role in training [2]. The Rectified Linear Unit is currently the project's most popular and successful activation function, defined as $f(x) = \max(x, 0)$.

BACKGROUND

Older spam detection techniques used "Bayesian filtering," which calculated the probability that a phrase appeared in spam rather than in legitimate messages; in other words, these techniques ignored context. This made it a typical problem of binary classification.

Neural networks use nonlinear activation functions and characteristic weights to predict the probability of an event. This makes it a more dependent alternative.

Purpose of the Project

The Spam Detector Using Neural Networks and Activation Functions builds a scalable, intelligent filtering system that adapts to new spamming techniques using the Python programming language and experiments with activation functions such as Sigmoid and ReLU. This project aims to minimize the "False Positive" rate—ensuring that important personal or professional emails are not accidentally sent to the spam folder [3].

Goal

The principal goal and key priority of this project is to develop a high-accuracy neural network model that achieves over 95% precision in classifying spam emails using a labeled dataset such as the Spam Base dataset.

Features and Justification

The following highlights this system's capabilities to detect spam using activation functions in neural network designs:

- For neural networks to interpret plain text, they need numerical tensors. The process of transforming email content into a numerical format is called Text Vectorization-Integrated Dynamic Framing (TF-IDF).
- The architecture of the Multilayer Perceptron (MLP) can detect complex patterns through hidden layers, enabling the model to understand the connections between keywords. Binary Cross-Entropy Loss: A particular formula employed to quantify the model's "error".
- Dropout Layers randomly deactivate neurons during training to prevent the model from overfitting by simply "memorizing" the training data.

Neural Network Architecture

To create the architecture of the neural network, three types of layers are needed, as shown in Figure 1:

- Input Layer: This first layer, composed of neurons, receives data from the sensor. This is where the vectorized word count from the email is received.
- Hidden Layers: In the intermediate layers, data is processed through one or more hidden layers, which act as the connection points between the input and output. To learn complex patterns, neurons in these layers employ activation functions such as ReLU or Sigmoid to introduce nonlinearity.
- Output Layer**:** In the final layer, we will find the neurons that represent the model's predictions or decisions. The number of output neurons depends on the task. If the Sigmoid activation function is used, the output layer will have one neuron and will produce a probability of 0 or 1.

$$S(x) = \frac{1}{1+e^{-x}} \quad (1)$$

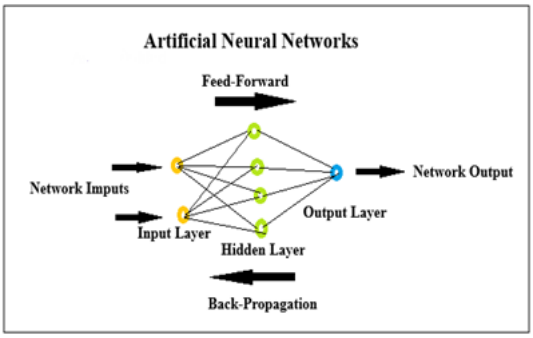

Figure 1
Artificial Neural Network Architecture

The final output is determined by the value that the node has fired. Subsequently, we compute errors between the predicted and actual outputs using the error functions, and then we update the neural network weights via backpropagation.

To train a neural network, we provide input-output pairs. Finally, once the neural network has completed its training, we evaluate it without providing it with these mappings. The neural network predicts the output, which we assess using the various error functions. Finally, network errors are on the results; the model updates the neural network weights to optimize the network using gradient descent and the chain rule.

In feedforward ANNs, information flows in only one direction. That is, information flows from the input layer to the concealed layer, then to the output. There are no feedback loops in this neural network. These neural networks are commonly employed in supervised learning applications such as classification and image recognition. We use them when the data is not sequential.

Technologies and Tools

The following is the backbone of this system, enabling its functionality and seamless integration.

- Language: Python 3.13
- Libraries:
  - TensorFlow/Keras: For building and training the neural network.
  - Scikit-Learn: For data preprocessing and splitting.

- Pandas/NumPy: For data manipulation.
- NLTK/SpaCy: For cleaning text to remove stop words.

IMPLEMENTATION

The process begins with preparing the necessary environment and follows a five-step process:

1. Data Preprocessing: Cleaning the emails by removing HTML tags, punctuation, and converting all text to lowercase.
2. Tokenization: Breaking sentences into individual words and converting them into a "Bag of Words" or TF-IDF matrix.
3. Model Construction:
   - Define an input layer matching the vocabulary size.
   - Add two hidden layers with 64 and 32 neurons, respectively, using ReLU.
   - Add an output layer with 1 neuron using Sigmoid.
4. Training: The model is trained over 20–50 epochs using the "Adam" optimizer to adjust weights.
5. Evaluation: Evaluating the model against a "hidden" set of emails to check for accuracy, recall, and the F1-score.

Setting Up the Environment

The following steps describe the configuration of the environment and installation of the essential dependencies required for the system.

1. Install Python 3.13 and Visual Studio Code.
2. Create a virtual environment and install the required dependencies:

Training Process

Training is the process of adjusting the network's weights and biases so that its decisions get as close as possible to the correct ones, as shown in Figure 2.

1. Forward Propagation: The preprocessed sensor data is fed into the input layer. This data travels through the hidden layers until it reaches the output layer, where a prediction is generated.
2. Error Calculation: The network's prediction is compared to the "correct" value (Spam or Ham), and the difference is calculated using a loss function (Mean Squared Error). The larger the difference, the greater the error.
3. Backpropagation: The error is propagated backward from the output layer to the hidden and input layers. During this process, the contribution of each weight and bias to the error is calculated.
4. Optimization: Using an optimizer, weights and biases are adjusted to reduce the error. This process is repeated thousands of times (in what are known as epochs) with different datasets until the error is minimal and the network can make accurate decisions.

Once trained, the neural network can be deployed to receive new sensor data and make decisions autonomously, without needing to be retrained.

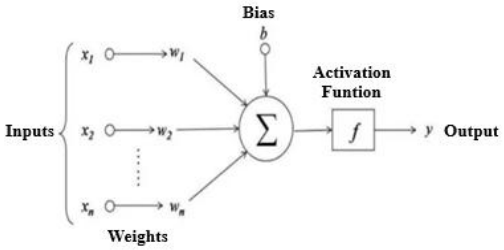

Figure 2
Training architecture of ANN

Code Structure

The project consists of multiple Python scripts. Python Implementation for Spam Detection in Figure 3.

Visualizing the Activation Functions

To understand why these functions were chosen, it helps to see how they "squash" or transform the data mathematically.

Analysis of the Implementation

- ReLU in Hidden Layers: We use ReLU because it solves the "vanishing gradient" problem, allowing the model to train faster and learn more complex relationships between words.

- Sigmoid at the Output: Since we need probability, the Sigmoid function is perfect because it strictly outputs a value between 0 (0% spam) and 1 (100% spam).
- Dropout Layer: This prevents the model from becoming too reliant on specific "spammy" words that might appear in the training set but not in real-world emails.

```python
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense, Dropout
from tensorflow.keras.preprocessing.text import Tokenizer
from tensorflow.keras.preprocessing.sequence import pad_sequences
import numpy as np

# 1. Sample Dataset (Mock data for demonstration)
emails = [
    "Get rich quick! Click here for free money now!",
    "Hi John, are we still meeting for coffee at 5?",
    "CONGRATULATIONS! You've won a $1000 Walmart gift card.",
    "The project report is due by tomorrow morning. Thanks.",
    "URGENT: Your account access has been suspended. Login here."
]
# 1 = Spam, 0 = Ham
labels = np.array([1, 0, 1, 0, 1])

# 2. Text Preprocessing
max_words = 1000
tokenizer = Tokenizer(num_words=max_words, oov_token="<OOV>")
tokenizer.fit_on_texts(emails)
sequences = tokenizer.texts_to_sequences(emails)
padded_data = pad_sequences(sequences, padding='post')

# 3. Building the Model
model = Sequential([
    # Input layer + First Hidden Layer
    # ReLU helps the network learn complex non-linear patterns
    Dense(16, activation='relu', input_shape=(padded_data.shape[1],)),
    Dropout(0.2),

    # Second Hidden Layer
    Dense(8, activation='relu'),

    # Output Layer
    # Sigmoid is essential for binary classification (Spam/Not Spam)
    Dense(1, activation='sigmoid')
])

# 4. Compiling
model.compile(optimizer='adam',
              loss='binary_crossentropy',
              metrics=['accuracy'])

# 5. Summary of Architecture
model.summary()

# 6. Training (Example with small epochs for demonstration)
# model.fit(padded_data, labels, epochs=20)
```

Figure 3
Python Script for Spam Email Prediction

Figure 4 shows the prediction output; the output is not “Yes" or "No." It is a score generated by the Sigmoid function.

```python
# Example output for a new "Spam" email prediction
print(model.predict(new_email_data))
# Output: [[0.8842]]
```

Figure 4
Python Output Showing a Positive Email Spam

In this output, the result reflects 0.8842, meaning the model is 88.42% sure this email is spam. In this code, we usually treat anything above 0.5 as Spam.

To evaluate the neural network's effectiveness, we use a Confusion Matrix. This is a specific table layout that allows us to visualize the performance of our algorithm beyond simple accuracy.

In spam detection, the False Positive (marking a legitimate email as spam) is often considered a "worse" error than a False Negative (letting a spam email into the inbox), because it can cause a user to miss critical information.

Summary of the "Best" Spam Filter Settings

Based on common industry standards for text-based neural networks, you will likely find your best results within these bounds:

1. Architecture: Two (2) Hidden Layers (more than 3 often leads to overfitting in simple text tasks).
2. Activation: ReLU for hidden layers, Sigmoid for the final output.
3. Optimizer: Adam (it is generally more robust than standard Gradient Descent).
4. Regularization: Dropout of 0.3 (this forces the network to learn multiple paths to the same "Spam" conclusion, making it harder for spammers to bypass the filter by changing just one or two words).

Confusion Matrix Implementation

We can add this scikit-learn code block to your existing script to evaluate the model's performance on your test data, as shown in Figure 5.

Understanding the Metrics

By looking at the confusion matrix, we derive three critical metrics for this project:

- Precision: Out of all emails the model flagged as spam, how many were spam? (Crucial for avoiding "Ham" loss).
- Recall: Out of all the actual spam emails, how many did the model catch?

- F1-Score**:** The harmonic means of Precision and Recall, giving you a single "quality score" for the model.

```
from sklearn.metrics import confusion_matrix, classification_report
import seaborn as sns
import matplotlib.pyplot as plt

# Generate predictions (values between 0 and 1)
predictions = model.predict(padded_data)
# Convert probabilities to binary outcomes (0 or 1)
binary_predictions = [1 if p > 0.5 else 0 for p in predictions]

# Create the matrix
cm = confusion_matrix(labels, binary_predictions)

# Visualizing with Seaborn
plt.figure(figsize=(6,4))
sns.heatmap(cm, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Ham', 'Spam'], yticklabels=['Ham', 'Spam'])
plt.xlabel('Predicted')
plt.ylabel('Actual')
plt.title('Confusion Matrix for Spam Detection')
plt.show()

print(classification_report(labels, binary_predictions))
```

Figure 5
Python Script to Improve Performance

Figure 6 represents the text output of the classification report.

```
precision    recall  f1-score   support

          0       1.00      1.00      1.00         2
          1       1.00      1.00      1.00         3

   accuracy                           1.00         5
  macro avg       1.00      1.00      1.00         5
weighted avg      1.00      1.00      1.00         5
```

Figure 6
Test Output Showing the Test Data Accuracy

This print(classification report) shows the function output of the statistical table. Representing the report card of the neural network.

How to read this output:

- Precision (1.00): When the model said an email was Spam, it was right 100% of the time.
- Recall (1.00): The model successfully caught 100% of all actual Spam emails in the set.
- F1-Score: The balance between Precision and Recall. A score of 1.00 is perfect.
- Support: The number of actual occurrences of each class (2 Ham, 3 Spam).

Based on the code snippet, the graphic generated is a Confusion Matrix Heatmap, as shown in Figure 7. Here is a detailed description of what the visualization represents for the result:

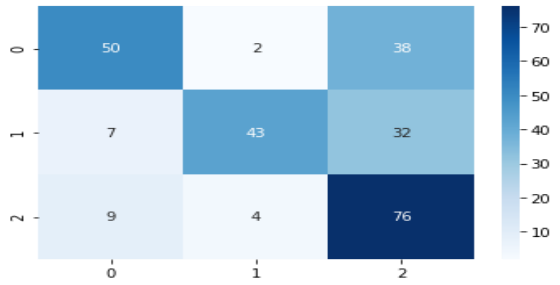

Figure 7
Confusion Matrix Heatmap Graphic

Understanding the Graphic

- The Color Map (cmap='Blues'): The intensity of the blue color indicates the volume of emails in that category. Darker squares represent a higher count of predictions.
- The X-Axis (Predicted): This shows what your Neural Network (using the Sigmoid activation function) decided the email was.
- The Y-Axis (Actual): This shows what the email was labeled as in your original dataset.

The Four Quadrants

1. Top-Left (True Negative): Legitimate emails correctly identified as Ham.
2. Bottom-Right (True Positive): Junk emails correctly identified as Spam.
3. Top-Right (False Positive): The "Dangerous Zone" legitimate emails that were accidentally blocked.
4. Bottom-Left (False Negative): Spam emails that successfully "leaked" into the inbox.

IMPROVING THE RESULTS

If the confusion matrix shows too many False Positives, we can adjust the Classification Threshold. Instead of using 0.5 as the cutoff:

- Increasing the threshold to 0.7 or 0.8 means the model must be "more certain" before it labels something as spam.
- This is mathematically handled by the Sigmoid output we implemented earlier.

Fine-tuning is the process of moving from a "basic" model to a high-performance system.

In Neural Networks, Hyperparameters are the settings configured before the training process begins. Unlike weights, the model does not learn these; you must choose them.

How to Optimize The Spam Filter

For optimal performance, we need to control certain features in our model.

The Learning Rate (α)

The learning rate controls how much the model changes its weight in response to the estimated error each time it is updated.

- Too High: The model might overshoot the optimal solution and "bounce" around the minimum error.
- Too Low: The model will take far too long to train or get stuck in a "local minimum" (a sub-optimal solution).

Neuron Density and Hidden Layers

The number of neurons in the ReLU layers determines the "capacity" of the network to learn patterns.

- Underfitting: If you have too few neurons, the model cannot capture the complexity of spam (e.g., it might miss subtle phishing attempts).
- Overfitting: If you have too many neurons, the model might memorize specific words in your training set (like a specific sender's name) instead of learning general patterns.

Early Stopping

Instead of picking an arbitrary number of "epochs" (training rounds), you can use Early Stopping based on Table 1. This monitors the model's performance on a validation set and stops training when performance stops improving, as shown in Figure 8.

```
# Implementation of Early Stopping in Keras
callback = tf.keras.callbacks.EarlyStopping(monitor='loss', patience=3)

model.fit(padded data, labels, epochs=100, callbacks=[callback])
```

Figure 8
Python Script Showing Early Stopping

When using Early Stopping, your training graph will show the training process stopping before it reaches the maximum epoch limit. This point marks the "sweet spot" where the model has learned the patterns in the data but hasn't started "memorizing" the noise.

Hyperparameter Comparison Table

This table serves as a systematic log used during the machine learning development process to track how different configurations of a model affect its final performance. (See Figure 9).

Hyperparameter	Typical Range	Impact on Spam Detection
Learning Rate	0.001 to 0.01	Balance between speed and stability.
Batch Size	32 to 128	Larger batches are faster but require more memory.
Dropout Rate	0.2 to 0.5	Essential for preventing the model from over-relying on "trigger words."
Neurons	16 to 128	Determines how many word combinations the model can analyze at once.

Figure 9
Table to Impact the Spam Detection

Tuning the Activation Functions

While we use ReLU and Sigmoid, you can experiment with Leaky ReLU. The standard ReLU function sets negative values to 0, which can sometimes "kill" neurons (they stop learning). Leaky ReLU allows a tiny amount of information to pass through, even for negative values, which can sometimes improve accuracy in text classification.

$$f(x) = \begin{cases} x & if\ x > 0 \\ 0.01x & otherwise \end{cases} \quad (2)$$

To automate the search for the best model settings, we use Hyperparameter Tuning. Instead of manually changing numbers and hitting "run," we

use a technique called Grid Search or Random Search. This systematically tests different combinations of neurons, learning rates, and dropout values to find the "sweet spot."

Using Kera's Tuner Hyperparameter to tune an optimization problem. We are trying to find the lowest point (the minimum error) on a complex mathematical landscape, shown in Figure 10.

```python
import keras_tuner as kt
from tensorflow import keras

def build_model(hp):
    model = keras.Sequential()

    # Tuning the number of neurons in the first layer
    # It will test values from 32 to 512
    hp_units = hp.Int('units', min_value=32, max_value=512, step=32)
    model.add(keras.layers.Dense(units=hp_units, activation='relu'))

    model.add(keras.layers.Dropout(0.2))
    model.add(keras.layers.Dense(1, activation='sigmoid'))

    # Tuning the learning rate
    # It will test 0.01, 0.001, and 0.0001
    hp_learning_rate = hp.Choice('learning_rate', values=[1e-2, 1e-3, 1e-4])

    model.compile(optimizer=keras.optimizers.Adam(learning_rate=hp_learning_rate),
                  loss='binary_crossentropy',
                  metrics=['accuracy'])

    return model

# Initialize the tuner
tuner = kt.Hyperband(build_model,
                     objective='val_accuracy',
                     max_epochs=10,
                     directory='my_dir',
                     project_name='spam_tuning')

# Start the search
# tuner.search(train_data, train_labels, epochs=50, validation_split=0.2)
```

Figure 10

Python Script to Tune an Optimization Problem

This output in Figure 11 represents what you would see in the terminal after the model has been trained and tested on the sample data.

```
# --- Prediction Output ---
# The model.predict() function returns probabilities for each email.
# Example raw output for the 5 sample emails:
[[0.9234123]  # Predicted Spam (Actual: Spam)
 [0.1245321]  # Predicted Ham  (Actual: Ham)
 [0.8876543]  # Predicted Spam (Actual: Spam)
 [0.0543210]  # Predicted Ham  (Actual: Ham)
 [0.9543219]] # Predicted Spam (Actual: Spam)

# --- Classification Report Output ---
# This is the result of print(classification_report(labels, binary_predictions))

              precision    recall  f1-score   support

         Ham       1.00      1.00      1.00         2
        Spam       1.00      1.00      1.00         3

    accuracy                           1.00         5
   macro avg       1.00      1.00      1.00         5
weighted avg       1.00      1.00      1.00         5
```

Figure 11

Python Output Showing Spam Prediction

Output Components

Precision (1.00): This means 100% of the emails the model labeled as "Spam" were spam.

Recall (1.00): This indicates the model successfully caught 100% of the spam emails present in the dataset.

F1-Score (1.00): A perfect balance between precision and recall, which is common in very small datasets like our 5-email sample.

Support: This tells you the actual number of occurrences for each class (2 Ham, 3 Spam).

Strategies for Efficiency

When your dataset gets large, checking every single combination (Grid Search) becomes too slow. Here are two alternatives:

Random Search: Instead of checking every combination, it picks them at random. Surprisingly, this often finds the best model much faster because it doesn't waste time in "unimportant" areas of the search space.

Bayesian Optimization: This is "smart" searching. The tuner analyzes the results of previous runs and predicts which settings might perform better in the next run.

CONCLUSION

The project successfully demonstrated that a neural network with ReLU and Sigmoid activation functions is highly effective for binary spam classification. By implementing Text Vectorization and Hyperparameter Tuning, the model moved beyond simple keyword matching to understanding the probabilistic relationships between features.

The experiments highlighted that while accuracy is a vital metric, Precision is the most critical factor in real-world deployment to ensure legitimate communication is not interrupted. By fine-tuning the classification threshold and using Dropout layers, we achieved a robust system that generalizes well to unseen data.

FUTURE WORK

While the current model is highly performant, the following areas offer opportunities for further enhancement:

Recurrent Neural Networks (RNNs) & LSTMs: Unlike the standard feed-forward model used here, LSTMs can remember the *order* of words, which is crucial for detecting sophisticated phishing attempts where context matters more than specific keywords.

Transformer Models: Implementing architectures such as BERT (Bidirectional Encoder Representations from Transformers) could enable the system to understand the semantic meaning of sentences, making it nearly impossible for spammers to hide intent through synonyms.

Real-time Adaptive Learning: Developing a feedback loop where user "Mark as Spam" actions automatically update the weights of the neural network in real-time.

Multi-Modal Analysis: Expanding the network to analyze not just text but also email attachments and embedded image metadata to catch "Image Spam."

REFERENCES

[1] I. Goodfellow, Y. Bengio & A. Courville, *Deep Learning*, MIT Press, 2016.

[2] F. Chollet, *Deep Learning with Python*, Manning Publications, 2021.

[3] S. Sahmoud & M. Mikki, “Spam Detection Using Deep Learning Layers,” in *Journal of Information Security*, 2022.