The Halt to the Halting Problem: Utilizing Artificial Intelligence to Detect the Halting Behavior of Functions

James Mathis

16 February 2024

Abstract

The Halting Problem, proved unsolvable by Alan Turing in 1936, shows that it is impossible to know with certainty whether a function will halt. The problem stands as a fundamental limitation in the abilities of computers. This research aimed to utilize a neural network to find observable patterns in halting and non-halting functions. The network was trained on 200 sample functions with a known halting behavior and its hyperparameters were optimized. The model reached a mean accuracy of 62.0% based on 100 trials. An accuracy greater than 50% indicates that the model recognizes patterns in the functions and correctly classifies them most of the time.

1 Introduction

The problem of deciding whether a given function will halt or run indefinitely has been a fundamental aspect of theoretical computer science for almost 100 years. This problem, known as the Halting Problem, has been proven to be unsolvable with full certainty. The problem shows that despite being arguably one of mankind's greatest inventions, computers still have limitations on their abilities.

1.1 Background Research

The Halting Problem was originally stated by Alan Turing in his 1936 paper *On Computable Numbers, with an Application to the Entscheidungsproblem.* Turing originally theorized this problem on a conceptual machine called a Turing machine also discussed in the paper that is the precursor to the modern computer. Turing proved that the Halting Problem was undecidable by computable means by discussing the problem on a Turing Machine. Nothing can be determined about the halting behavior of a given Turing machine simply by watching it run, because even if the machine does not halt after a year, it could eventually halt if it is allowed to run[1].

The Halting Problem can be found in many real world situations in the field of computer science. Many software engineers accidentally write programs with non-halting

loops without noticing that they will never halt. Infinite loops have the potential to use a large amount of resources in their execution which wastes energy that accumulates over time. They also waste the time of the engineer, as they have to search for the loop that is causing the issue. This can be tedious for very large or complex programs.

```
while True:
    temperature = read_temperature_sensor()
    if temperature > 100:
        send_alert_email("Temperature too high!")
```

The code snippet above shows a non-halting loop that could be used within a system. Without a proper exit condition, the loop will continue indefinitely until the program is killed. This could cause the program to become unresponsive or consume excessive resources. A program designed to detect these loops could constantly observe the code and alert the developer, decreasing the amount of time and resources that need to be devoted to the routine task of debugging[2].

While it may be impossible to say with certainty whether a given function will halt or run indefinitely, there are ways to approximate the runtime of a function. According to Calude and Stay, if a given function has a size of N bits, the effective maximum time it can run is 2^{N+c} where c is a constant[3]. After this point, the probability of the function halting is so low that it is effectively zero[3]. Using this, it can be concluded that any function that runs longer than several seconds is almost certain to be non-halting.

The undecidability of the problem only accounts for allowing the given function to run. When functions are observed as a whole, patterns may emerge between those which can clearly be classified as either halting or non-halting. This observation prompts an exploration into more sophisticated computing techniques than simply running the function, such as utilizing a neural network. Neural networks, with their capacity to learn intricate relationships in data, present a promising avenue for addressing the task of determining the halting behavior of functions.

1.2 Goal

The goal of this research is to utilize a neural network to detect patterns in halting and non-halting Python functions and determine their halting behavior. To be considered successful, the network needs to correctly classify a function at least 60% of the time. Achieving this benchmark accuracy will demonstrate the model's ability to grasp the relationship between a given function and its halting behavior. The ability to detect functions that run indefinitely will greatly contribute to computer science and software development, as well as the pursuit of a definite solution to the Halting Problem.

1.3 Introduction to Procedures

The main objective of this research is to leverage artificial intelligence for determining the halting behavior of a function, employing a neural network with Python functions as training data to achieve an accuracy of 60% or higher. The training dataset was compiled through three methods: manual creation of Python functions, a crowdsourcing website, and a script generating random functions. The script randomly selected keywords and constructed valid Python functions, determining halting behavior through multithreading and a 3-second wait period. The resulting dataset comprised 200 functions (100 halting and 100 non-halting). The model's topology consisted of a sequential structure with an embedding layer, a flattening layer, and a dense layer with a sigmoid activation function. Hyperparameter optimization, including the model optimizer, epochs, batch size, and output dimensions, was performed. After finding the optimal set of hyperparameters, the model was trained on 150 samples and evaluated over 50 samples for 100 trials to calculate the average performance.

2 Procedures

The primary goal is to utilize artificial intelligence to determine halting behavior of a function. To reach the target accuracy of 60% or higher, a neural network is employed and given Python functions as text to train on.

The training dataset was collected through 3 different means. First, Python functions were written manually for testing and making a proof of concept. This method was used to collect 15 functions. This approach proved far too slow to get a significant dataset, so a website was developed to crowdsource functions. Only a few people submitted functions, and the functions that were submitted had mostly already been collected manually. Over the course of 3 weeks, only 5 functions were added to the dataset from the website.

The collection method used to populate the vast majority of the dataset was a script to write random functions. The script had a list of the various types of statements and keywords that could be used in Python. It randomly chose a keyword to add to the function and the proper expressions were added to make a valid Python function. For example, if the program chose to add an "if" statement, a valid boolean expression would be placed after it, such as "var_12345 == 80" or "18 != 90". This was repeated between 5 and 25 times, the number of repetitions being randomly selected.

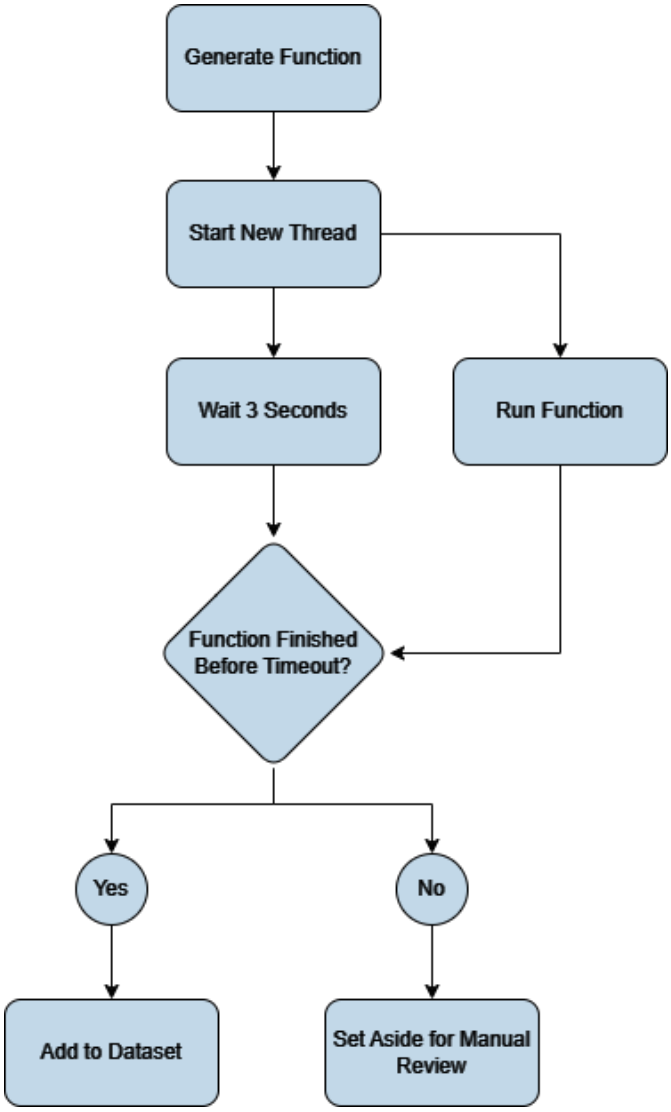

Figure 1: The diagram shows how functions from the function generator are classified.

Figure 1 demonstrates the flow of data in the script after the function is generated. The complete function ran in a separate thread while the main thread waited for 3 seconds. If the thread closed in less than 3 seconds, it was a halting function and was added to the dataset. If the thread hadn't closed, it was likely, but not certainly, a non-halting function. After 3 seconds the thread was killed and the function was put into a text file for manual review. If the function was indeed a non-halting function, it was added to the dataset. This collection method was used to increase the size of the dataset to 200 functions, 100 halting and 100 non-halting.

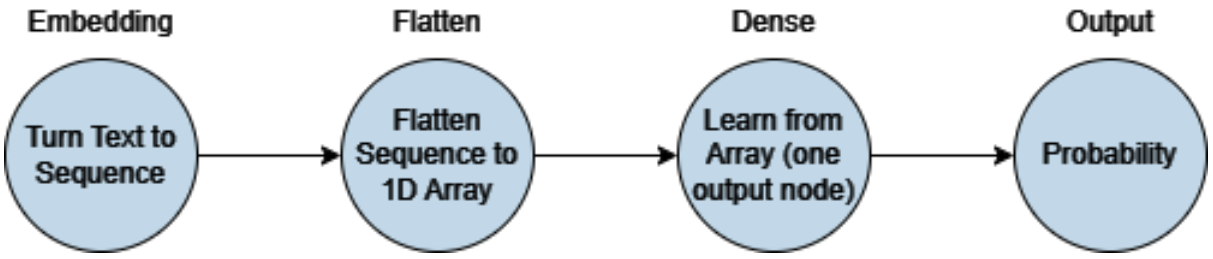

Figure 2: The diagram shows the topology of the neural network. It also shows how the data is transformed in each layer.

The model's topology, shown in Figure 2, was chosen based on the relatively simple complexity needed by the project. The model used a sequential topology with 3 layers. The first layer was an embedding layer, which is commonly used to process sequences of numbers. The input dimensions of this layer represent the maximum number of tokens in a sequence. The output dimensions parameter refers to the dimensionality of the resulting embedding and requires tuning for better accuracy. The model's second layer was a flattening layer used to flatten the embedding from the previous layer to a 1D array that is expected by the third layer and final layer. This layer was a dense layer and is the layer in which learning took place. This layer has a single neuron using the sigmoid activation function.

In the creation, compilation, and training phases of the program, there were a set of hyperparameters that needed to be optimized to improve the accuracy. The hyperparameters that needed to be optimized were the model optimizer, the number of epochs, the size of each batch, and the model's output dimensions. A hyperparameter optimization algorithm was employed to find the hyperparameters that resulted in the highest average accuracy.

	Optimizer	Epochs (step: 100)	Batch Size	Output Dimensions (step: 20)
Possible Values	adam, sgd, rmsprop	[100, 500]	32, 64, 128	[20, 100]
Number of Values	3	4	3	5

Table 1: The table shows the possible values for each hyperparameter. This results in a total search space of 180 combinations.

Each hyperparameter had several possible values as denoted in Table 1. For the optimizer, there are 3 possible values: adam, sgd, and rmsprop. The epoch values ranged from 100 to 500, including both endpoints and stepping by 100. The size of each batch had possible values of 32, 64, and 128. The number of output dimensions ranged from 20 to 100 inclusive, stepping by 20. The ranges and steps for these hyperparameters were chosen semi-arbitrarily while remaining within reasonable bounds for the dataset.

Once the best set of hyperparameters were found, the model was set to use them and trained on 150 random samples. The model then predicted the halting behavior of the other 50 samples while recording the accuracy, loss, precision, and recall. This procedure was repeated 99 times and the metrics' statistics of all 100 trials were calculated.

3 Data and Results

Optimizer	Epochs	Batch Size	Output Dimensions
rmsprop	200	64	20

Table 2: This table shows the results of the hyperparameter optimization algorithm.

Table 2 contains the best set of hyperparameters from the optimization algorithm. The best combination used the rmsprop optimizer with 200 epochs, 64 functions in each batch, and 20 output dimensions.

	Mean	Minimum	Maximum	Standard Deviation
Accuracy	0.620	0.530	0.712	0.0415
Loss	0.802	0.682	0.952	0.0541
Precision	0.653	0.541	0.778	0.0496
Recall	0.566	0.441	0.735	0.0622

Table 3: This table shows the results of training the model 100 times and taking the average of the metrics. The values were rounded to 3 significant digits improve legibility.

After evaluating the model on 100 trials, the metrics displayed in Table 3 were calculated. The values of interest are the accuracy, loss, precision, and recall of the model.

The first metric that was recorded was accuracy, which represents the probability that the network will correctly classify the testing samples.

$$\text{Accuracy} = \frac{\text{tp} + \text{tn}}{\text{tp} + \text{fp} + \text{tn} + \text{fn}}$$

The equation above shows the calculation for accuracy. In this equation and all subsequent equations, `tp` means `true positive`, `tn` means `true negative`, `fp` means `false positive`, and `fn` means `false negative`. Accuracy is the most basic and common metric used to evaluate the performance of a neural network. The recorded accuracy had a mean of 0.620, minimum of 0.530, maximum of 0.712, and a standard deviation of 0.0415.

The next metric was the value of the loss function. The loss function quantifies how well the model recognizes patterns from the training phase within the testing samples. A low loss score indicates that the model is able to generalize and apply the relationships in the training samples to the testing samples. The loss metric averaged at 0.802, with the lowest value being 0.682 and the highest 0.952. There was a standard deviation of 0.0541 across all 100 trials.

The statistics of the model's precision was then calculated using the following equation:

$$\text{Precision} = \frac{tp}{tp + fp}$$

Precision is defined as the ratio of true positives to predicted positives. The precision measures the probability that the model is correct when predicting that a function halts. The recorded precision had a mean of 0.653, minimum of 0.541, maximum of 0.778, and a standard deviation of 0.0496.

The final metric recorded over the 100 trials was recall. The recall value is similar to precision, and is often combined with it to form the F1 score.

$$\text{Recall} = \frac{tp}{tp + fn}$$

This metric, calculated with the above equation, is the ratio of true positives to all positive cases. A high recall score indicates that the model effectively identifies most of the positive cases. The recorded recall had a mean of 0.566, minimum of 0.441, maximum of 0.735, and a standard deviation of 0.0622.

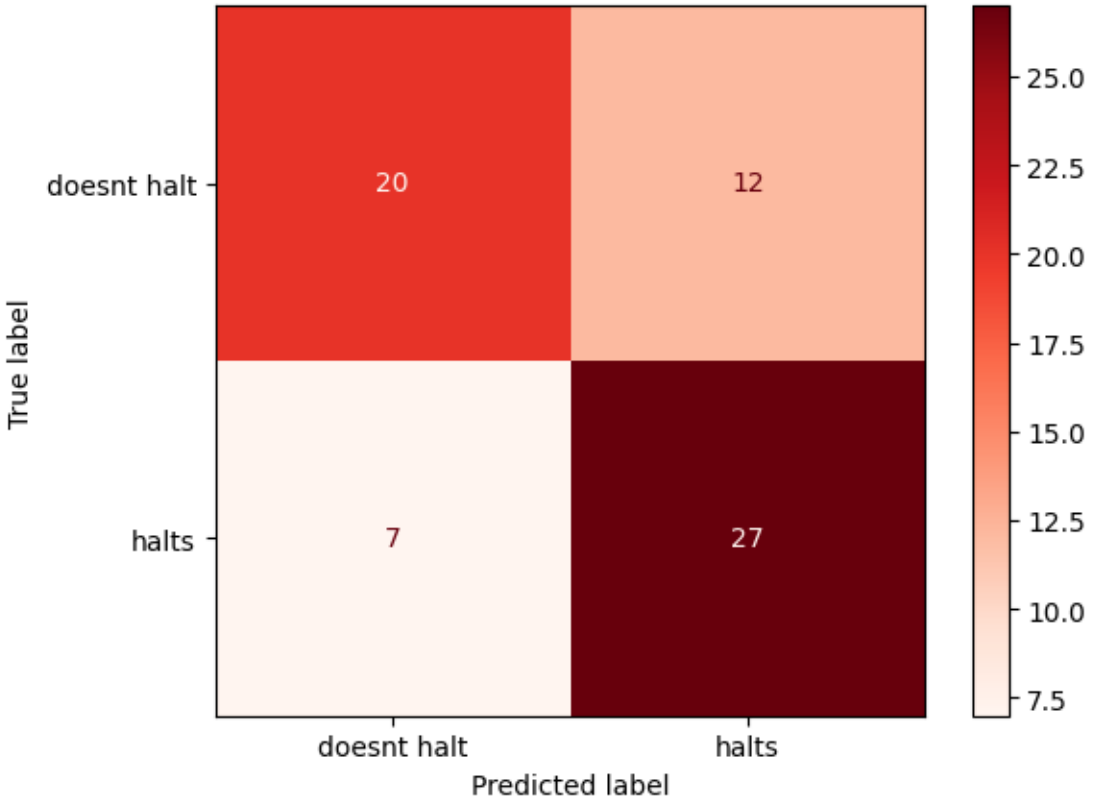

Figure 3: The confusion matrix above shows how the model performed at classifying 66 functions.

The confusion matrix in Figure 3 shows the classification performance of the model in the trial with the highest accuracy. The figure shows that in the best trial of the 100, the model correctly predicted the halting behavior of 47 functions and incorrectly classified 19 functions.

Metric	Value
Accuracy	0.712
Precision	0.733
Recall	0.667
F1 Score	0.698
Geometric Mean	0.699

Table 4: This table shows the metrics for best run out of 100 trials.

Table 4, the best performing version of the model performs very well, with most of the models reaching at least 70

4 Conclusion

4.1 Summary

This research explored the longstanding Halting Problem, a fundamental challenge in theoretical computer science for almost a century, demonstrating the inherent limitations of computers despite their significant advancements. Originating from Alan Turing's seminal work in 1936, the Halting Problem was proven to be undecidable with certainty, showcasing the complexities in determining whether a given function will halt or run indefinitely. The study aims to leverage artificial intelligence, specifically a neural network, to detect patterns in the halting behavior of Python functions. The dataset, compiled through manual creation, crowdsourcing, and random function generation, consisted of 200 functions. The neural network's goal was to achieve a classification accuracy of at least 60%, with hyperparameter optimization and performance evaluation conducted over 100 trials. If successful, this research has the potential to contribute significantly to computer science and software development, offering insights into the Halting Problem and advancing the understanding of function behavior.

The training dataset consisted of 100 halting and 100 non-halting functions, collected through three methods. Initially, 15 functions were manually written to establish a proof of concept, but due to the slow pace, a crowdsourcing website was developed, contributing only 5 additional functions over three weeks. The majority of the dataset was generated using a script that randomly created Python functions, with the halting behavior determined through multithreading and a 10-second wait period. The neural network's topology consisted of an embedding layer, a flattening layer, and a dense layer with a sigmoid activation function. Hyperparameter optimization, involving the model optimizer, epochs, batch size, and output dimensions, was performed. The resulting model was trained on 150 samples, evaluated on 50 samples for 100 trials, and the statistics of the accuracy, loss, precision, and recall was calculated.

After evaluating the model through 100 trials, key metrics were recorded to assess its performance. Accuracy, the primary metric, signifies the probability of the network

correctly classifying testing samples. The recorded accuracy exhibited a mean of 0.620, minimum of 0.530, maximum of 0.712, and a standard deviation of 0.0415. The evaluation loss, a measure of how well the model recognizes patterns from the training phase in testing samples, was also recorded. A lower loss indicates the model's ability to generalize and apply relationships from training to testing. The mean loss was 0.801, with a minimum of 0.682, maximum of 0.952, and a standard deviation of 0.0547.. Precision, measuring the ratio of true positives to predicted positives, was defined with halting functions considered positive cases. The precision's mean was 0.653, with a minimum of 0.541, maximum of 0.778, and a standard deviation of 0.0496. Finally, recall, indicating the ratio of true positives to all positive cases, showed a mean of 0.566, minimum of 0.441, maximum of 0.735, and a standard deviation of 0.0622.. The confusion matrix in Figure 3, representing the best trial, illustrates the model's classification performance, correctly predicting the halting behavior of 47 functions and misclassifying 19 functions.

With a mean accuracy of 0.620, the model successfully reached the target accuracy of 60%. The success of the neural network demonstrated that there are observable patterns within a function that can determine whether or not it will halt. A standard deviation of 0.0415 shows that the model reliably achieves the mean accuracy. This implies that when the model is used to classify a function, the average of only 2 or 3 trials is needed to produce a trustworthy prediction. The mean loss of 0.802 signifies that the model effectively identifies patterns and relationships that influence a function's halting behavior. The mean precision of 0.653 shows that, on average, when the model predicts a function to be a halting function, it is correct 65.3% of the time. Finally, with a mean recall score of 0.566, the model is able to recognize a halting function 56.5% of the time on average.

4.2 Discussion

The recorded metrics hold significant implications for the use of artificial intelligence in attempting to solve Turing's Halting Problem. The model's success at classifying functions as halting and non-halting illustrates that there may be common patterns in all non-halting functions. To increase the reliability and efficiency of the neural network,

there are two main avenues of future research: increasing the size of the dataset and the implementation of an ensemble learning algorithm.

A sample size of 200 functions is large enough to get some results when using a neural network, but more data is always better. Ramezan et al performed a study in which the accuracy of various machine learning algorithms was recorded after being trained on datasets of various sizes[4]. They found that the accuracy of neural networks trained on the data in their specific problem performed best with a training set of 10,000 samples[4]. This shows that it is best to use as much data as possible when training neural networks.

For this research, a dataset with a size on the same order of magnitude as in Ramezan's study would significantly improve the performance of the model. The easiest ways to increase the size of the dataset is generating more random functions, larger scale crowdsourcing, and scraping loops from Github. The dataset also doesn't necessarily need to be balanced between the 2 classes, as long as a metric other than accuracy is used to evaluate the model.

On top of increasing the dataset size, an ensemble learning algorithm can be employed to increase reliability. In ensemble learning, multiple models are combined and the output of each is merged when predicting to reduce variance[5]. One such algorithm that has the potential to boost the model's accuracy is the Random Forest algorithm. In Random Forest, multiple models, known as trees, are trained on random subsets of the data to form a forest, according to IBM[6]. To classify a function using the Random Forest algorithm, each tree would predict the halting behavior of the function. The prediction merge algorithm most commonly used with Random Forest is a simple majority vote. This ensemble learning algorithm has multiple benefits, including a reduced risk of overfitting, greater flexibility, and more reliable predictions[6].

Although Alan Turing proved the Halting Problem to be unsolvable through conventional computing, the ability to predict whether a program will halt or run indefinitely has the potential to advance the field of computer science and improve the lives of many developers. By employing a neural network to analyze patterns between halting and non-halting Python functions, this study demonstrated that there are observable pat-

terns within these functions that can be used to predict their behavior. Future research on this subject includes modifying the model to use more meaningful discriminator metrics and utilizing algorithms such as ensemble learning to increase reliability.

References

(1) C. BJ, *The Essential Turing : Seminal Writings in computing, logic, philosophy, Artificial intelligence, and Artificial Life plus the Secrets of Enigma*, Clarendon Press, 2004.

(2) M. Barenkamp, J. Rebstadt and O. Thomas, *AI Perspectives*, 2020, **2**, DOI: 10.1186/s42467-020-00005-4.

(3) C. S. Calude and M. A. Stay, *Advances in Applied Mathematics*, 2008, **40**, 295–308.

(4) C. A. Ramezan, T. A. Warner, A. E. Maxwell and B. S. Price, *Remote Sensing*, 2021, **13**, 368.

(5) T. N. Rincy and R. Gupta, *Ensemble Learning Techniques and its Efficiency in Machine Learning: A Survey*, IEEE Xplore, 2020.

(6) IBM, *What is Random Forest? — IBM*, www.ibm.com, 2023.