# MACHINE LEARNING-ENHANCED ROOT FINDING METHODS: A COMPARATIVE ANALYSIS

SADIE SHUDDE

Abstract. Root-finding algorithms are fundamental in the field of numerical analysis. There are several different methods commonly used, and Newton-Raphson, also known as Newton's method, is one of them. It is used for its very quick quadratic convergence near the root. However, Newton's performance depends greatly on the initial guess given. Poor initial guesses can cause the method to diverge or converge slowly. Although mathematicians have different methods and reasons for picking an initial guess, machine learning (ML) can be used to provide a quicker and better initial guess by learning patterns. We trained a neural network on 1,000 quadratic equations with random coefficients and compared it to classical Newton-Raphson with random initial guesses. The results showed that the ML-enhanced model reduced iterations by 44.0% (from 4.45 to 2.49 average iterations) and demonstrated 42.09% greater consistency. Although both the classical and ML-enhanced methods showed a 100% success rate, the ML model offers significant computational advantages for repeated root-finding.

## 1. Introduction

Root-finding problems appear in several different fields, including scientific computing, engineering, and applied mathematics. Given a function $f(x)$, the goal is to find the $x$ values such that $f(x) = 0$. These problems arise in differential equations, optimization, physics, and numerous other applications. For example, solving differential equations numerically requires finding roots at each time step. Engineers use root-finding to determine when a projectile hits the ground by solving for the time when $h(t) = 0$.

The Newton-Raphson method is quite popular because of its fast quadratic convergence near a root. The formula is derived using the Taylor series, where we assume $(x - x_0)^2 \approx 0$, eliminating everything beyond the first derivative. This leaves us with the formula

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

for $n = 0, 1, 2, \ldots$.However, this method faces limitations related to the first derivative which makes it highly sensitive to initial guesses. The consequences of a bad guess are divergence, slow convergence, or finding the wrong roots. Currently, the best approaches rely on either trial and error or strong domain knowledge. It also fails dramatically when the $f'(x_n) = 0$. As stated before, there are many different root-finding methods, such as bisection. Bisection is guaranteed to find a root every time, but it has slow linear convergence. Newton's method offers superior efficiency

*Date*: December 1, 2025.
Honors Contract supervised by Dr. Keith Emmert.

for equations where it is viable. For example, Newton's method fails at $f(x) = x^3$ because at the root $x = 0$, the formula becomes unstable due to division by zero.

This study will explore a newer approach to utilizing ML to predict optimal initial guesses for the Newton-Raphson method. Neural networks are extremely capable of learning patterns from data. We trained a neural network on 1,000 quadratic equations with varying coefficients. We hypothesize that this neural network will better approximate root locations, thereby improving convergence efficiency. Machine learning (ML) is increasingly available and powerful. It is a powerful tool with great computational resources, which makes it perfect for data-driven real-world problems. When a neural network is trained, it can learn patterns that are not obvious to researchers. It can take a human quite a while to notice patterns that the model can recognize with much less effort. In this program, the patterns come from coefficient relationships, and the model is very valuable for solving similar root-finding problems repeatedly needing someone to make an initial guess every time.

In this study, we will investigate if ML can help solve the sensitivity issue the Newton-Raphson faces. We specifically focus on quadratic equations, to see if improvement occurs where Newton's method performs best. The neural network is trained to pick the ideal $x_0$ for any given $f(x) = ax^2 + bx + c$. The model is trained based on the coefficients $a, b$, and $c$. Next, we compared the classical and ML-enhanced models over 100 test cases to determine improvements in both efficiency and consistency. Our Contributions are:

(1) Implementation and comparison of three classical methods as a baseline
(2) Development of a neural network model for initial guess prediction
(3) Demonstration of a 44.0% reduction in iterations using the ML-enhanced Newton-Raphson
(4) Analysis of consistency improvements in root-finding performance

## 2. Background

The theory and implementation of these classical methods follow the treatment in *Numerical Analysis I Course Notes* by Dr. Keith Emmert [1].

### 2.1. Classical Root-Finding Methods.

We implemented three classical numerical root-finding methods. Each of these methods are used for various reasons depending on the problem.

**Bisection Method:** This is a bracketing method that utilizes the mid-point to converge to the root.

**Definition of the Bisection Method** The idea behind the Bisection Method is that we:

- Begin with the interval $[a_0, b_0] = [a, b]$.
- Let $p_0 = \frac{a_0+b_0}{2}$.
- Determine where the root lies:
  - If $f(a_0)$ and $f(p_0)$ have the same sign, then the root is in $[a_1, b_1] = [p_0, b_0]$.
  - Otherwise, the root is in $[a_1, b_1] = [a_0, p_0]$.
  - Form $p_1 = \frac{a_1+b_1}{2}$.
- Continue this process. We hope that the sequence of points $p_0, p_1, p_2, \ldots$ converges to a root of $f$.

This method requires $f(a)$ and $f(b)$ to be of opposite signs. This tells the algorithm that there is a root at some point in between the 2 points. The Bisection method is useful because it guarantees a root will be found if there is one between the interval chosen. However, there is a large overhead in comparison to some other methods. It can also take a while to converge to the root depending on the initial interval. Since the convergence type is linear, it requires more iterations to reach the desired tolerance. It is best used when a problem requires a guaranteed convergence and has a bracketed interval.

**Newton-Raphson-Method:** The Newton's method is a root-finding algorithm that utilizes theory from calculus. It has a strong convergence rate when it comes to polynomials specifically quadratics.

**Definition of Newton-Raphson Method** Starting with an initial guess $x_0$, Newton's method iterates:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

We wish to find a root $r$ of a function $f(x)$, that is, $f(r) = 0$. The theory behind Newton's method is derived using *Taylor series*, given an initial $x_0$,

$$f(x) = f(x_0) + f'(x_0)(x - x_0) + \frac{(x - x_0)^2}{2!} f"(\xi_x)'$$

where $\xi_x$ is between $x$ and $x_0$. If we assume $(x - x_0)^2 \approx 0$, then

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0)$$

Then set $f(x) = 0$,

$$x = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

So, we can define

$$x_1 = x_0 - \frac{f(x_0)}{f'(x_0)}.$$

Newton's method is extremely useful because of its fast convergence. It is specifically quick to converge with quadratic equations when the initial guess is near the root. However, it faces many challenges that can force the algorithm to break or perform poorly. First, if the derivative $f'(x_n) = 0$, the program breaks because of division by 0. The second problem can affect the speed of the convergence, and if it converges at all. The initial value guess of $x_0$ affects how the algorithm converges. Newton's method only works if the function is continuous and differentiable within the interval. Convergence issues appear primarily due to a poor initial guess of $x_0$. If the initial guess is to far away from the root, the more likely the second derivative would be needed for the approximation in *Taylor series*; however, based on the theory for this method we assumed that $(x - x_0)^2 \approx 0$, so if it is not 0, then the convergence takes much longer. Newton's method is best used when a problem is quadratic and there is a general knowledge of the domain and where a root might appear. The convergence type is quadratic when near the root, and the theorem for Newton's convergence is as follows.

**Theorem 2.3.1.** *Let $f \in \mathcal{C}^2[a, b]$. If $p \in (a, b)$ is such that $f(p) = 0$ and $f'(p) \neq 0$, then there exists a $\delta > 0$ such that Newton's method generates a sequence $\{x_n\}_{n=1}^{\infty}$ converging to $p$ for any initial approximation $x_0 \in [p - \delta, p + \delta]$.* **2.3.2.1 Failure Cases**

- If $f'(x_n) = 0$, the method fails.

- If $x_0$ is far from the root, divergence may occur.

**Secant Method:** The Secant method is very similar to Newton's method with the main difference of depending on limit definition. This approximates the derivative using the finite difference instead of the explicit derivative. Because of this change Secant will depend on 2 initial guesses $x_0$ and $x_1$. **Theory of Secant Method** Recall that

$$f'(a) = \lim_{h\to 0} \frac{f(a+h) - f(a)}{h} = \lim_{x\to a} \frac{f(a) - f(x)}{a - x}.$$

Therefore,

$$f'(a) \approx \frac{f(a+h) - f(a)}{h} \qquad \text{or} \qquad f'(a) \approx \frac{f(a) - f(x)}{a - x}.$$

When $f'(x)$ is difficult to compute, we can approximate it using finite differences:

$$f'(x_n) \approx \frac{f(x_n) - f(x_{n-1})}{x_n - x_{n-1}}.$$

The, substituting the approximation for $f'(x_n)$ into *Newton's Formula* gives the **secant method**

$$x_{n+1} = x_n - \frac{f(x_n)(x_n - x_{n-1)}}{f(x_n) - f(x_{n-1})}.$$

The secant method is very useful because it is similar to Newton's, but it does not require a derivative. This means it does not inherit the issue of $f'(x) = 0$. While it still requires 2 initial guesses, $x_0$ and $x_1$, it is less sensitive than Newton-Raphson. In terms of convergence, it's type is superlinear. This means it is faster than the bisection method. Secant method should be used when the derivative is hard to compute, but the convergence needs to be faster than bisection. This acts as a practical compromise to the two previous methods.

2.2. **Machine Learning Basics.** Machine learning (ML) is an ever-growing field of research in the 2020s. However, the idea and basic architecture of ML has been around since the 1970s. It works by enabling computers to learn patterns from data without explicit programming to define the pattern. It receives the data, studies the problem, and produces a mathematical model. Then an optimized algorithm will provide *performance guarantees*, as long as the model is a valid portrayal of the problem [4]. In supervised learning, a model is trained on pairs of inputs and their desired output. The goal is to find the pattern between the input and output [4]. This can be utilized in root-finding methods by recognizing patterns between the coefficients and roots that are too complex for a human to find quickly. The human brain has many abilities that would benefit computer algorithms. Human brains are adaptive, generalized, and possess many more desired aspects for computers [5]. When used in computers, they are called artificial neural networks (ANN); these networks excel at learning complex patterns without requiring explicit formulas. A neural network consists of layers of interconnected computational units called neurons [5]. Each neuron receives multiple inputs, computes a weighted sum, and applies a nonlinear activation function to produce an output. The networks is organized into three types of layers.

- **Input layer:** Receives the raw data
- **Hidden layers:** Intermediate layers that learn progressively abstract representations

- **Output layer:** Produces the final prediction

The basic architecture of the ANNs are categorized into feed-forward and recurrent networks.

- **Feed-forward networks:** Data flows in one direction
- **Recurrent networks:** Loops occur meaning outputs feed back as inputs

Our implementation uses a feedforward multilayer perceptron for mapping from the coefficients of the polynomial $(a, b, c)$ to the initial guess $x_0$. Networks can learn through backpropagation [5]. During training, the network will make predictions and compare them to the true values we have with the desired outputs [6] A loss function is used to calculate the difference in the true and predicted values. Backpropagation then computes the gradient of loss with respect to each weight in the network, propagating the error backwards through the layers hence the name. The weights are then adjusted according to the gradient to minimize loss. The training continues until the model converges to an accurate map of inputs to outputs [6]. For our root-finding model, the network learns to associate quadratic coefficients $(a, b, c)$ with optimal initial guesses.

## 3. Methodology

3.1. **Classical Methods Implementation.** To establish a baseline for comparison against the ML model, three classical methods were implemented: Bisection, Newton-Raphson, and Secant. Each method was applied to the same three functions to test and compare each method. These were the test functions:

- $f_1(x) = e^{-x} - x$
- $f_2(x) = x^2 - 4$
- $f_3(x) = x^3 - x - 2$

Each method was coded to return the approximate root, the total number of iterations, and a full history of iterations for an easy way to compare the 3 methods visually. All methods used a tolerance of $10^{-6}$ for stopping and a maximum of 100 iterations. A method was considered to have converged if either the update size fell below tolerance or the function value became sufficiently small. If 100 iterations were reached with out fulfilling these conditions, the attempt was marked as a failure.

**Bisection Method** For the Bisection method, an interval $[a, b]$ was chosen for each function such that $f(a) \cdot f(b) < 0$, this guarantees that a root lies inside. As stated before Bisection method guarantees a root, but it has slow convergence. For all the test functions, the chosen intervals were:

- $f_1(x) : [0, 1]$
- $f_2(x) : [0, 3]$
- $f_3(x) : [1, 2]$

As promised by the method, Bisection successfully converged for all three tests.

**Newton-Raphson Method** Newton's method requires only a single initial guess, along with the derivative of the function. In these tests, the initial guess was intentionally kept simple rather than optimized for each function. Although this was not originally intended to be a "bad guess" experiment, it ended up highlighting one of Newton's well-known weaknesses, its sensitivity to initial guesses. The initial guesses for each function were:

- $f_1(x) : x_0 = 0.5$

- $f_2(x) : x_0 = 1.0$
- $f_3(x) : x_0 = 1.0$

All three initial guesses led to Newton's method reaching the maximum iteration limit without converging. All three function were picked because they have "well behaved" derivatives to not fully break the Newton method. Despite the functions being picked well, the initial guesses still caused the method to diverge. This reinforces the motivation for the ML potion of the project. Better initial guesses can be the deciding factor in whether Newton's method converges or not.

**Secant Method** The Secant method as mentioned before is similar to Newton's method but does not require the derivative. It requires two initial guesses; in these three tests, the guesses were automatically chosen where whatever was entered as $x_0$ then $x_1 = x_0 + 1.0$. With the initial guesses being:

- $f_1(x) : x_0 = 0.5$ and $x_1 = 1.5$
- $f_2(x) : x_0 = 1.0$ and $x_1 = 2.0$
- $f_3(x) : x_0 = 1.0$ and $x_1 = 2.0$

Even with simple starting points, the Secant method successfully converged for all three functions, usually between 3-6 iterations.

3.2. **Machine Learning Approach.** The goal of the ML portion of the project was to not approximate the roots directly, but instead train a model to produce a good initial guess for Newton's method. Since Newton's method is sensitive to that initial guess, training a ML model to generate good guesses would lead to improvements in speed and reliability.

3.2.1. *Training Data Generation.* To train the model, a large dataset of randomly generated quadratic equations of the form

$$ax^2 + bx + c = 0$$

was created. The coefficients $a$,$b$, and $c$ were sampled uniformly from specified ranges.

- $a \sim U(0.5, 5)$
- $b \sim U(-10, 10)$
- $c \sim U(-10, 10)$

The coefficient $a$ was restricted to a positive value to ensure proper quadratic behavior, while $b$ and $c$ were allowed to vary more. After the coefficients were calculated the discriminant $\Delta = b^2 - 4ac$ and only the quadratics with real roots were kept by discarding anytime $\Delta < 0$. Then the real root were calculated using the quadratic formula, since two roots were a possibility the root closest to zero was kept as the target. This is because roots near the origin are often numerically more stable. The training data had the input features of $X = [a, b, c]$ and the target output of y = best-root. The reason for using the synthetic data approach held several advantages. First, it produces a large diverse dataset without requiring manual labeling. Second, the actual root are calculated using the quadratic formula providing perfect training labels. Third, the variety in coefficients ensures the models learns general patterns rater than memorizing specific cases which can lead to overfitting.

3.2.2. *Neural Network Architecture.* We implemented a feedforward multiplayer perceptron(MLP) using scikit-learn's MLPRegressor in Python. The networks architecture consists of

- **Input layer:** 3 neurons corresponding to coefficients $(a, b, c)$.
- **Hidden layers:** Three hidden layers with 50, 30, and 20, neurons respectively.
- **Output layer:** 1 neuron predicting the optimal guess $x_0$.
- **Activation function:** ReLU (Rectified Linear Unit) for all hidden layers.
- **Training parameters:**
  - Maximum iterations: 500
  - Early Stopping: enabled to prevent overfitting
  - Train/Test split: 80/20 with random-state = 72 for reproducibility

The complete architecture is illustrated in Figure 1

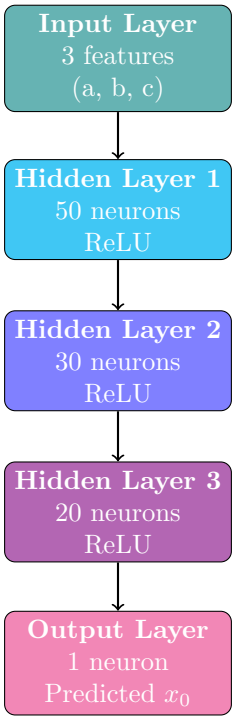


FIGURE 1. Neural Network Architecture

The choice of a three-layer architecture with decreasing neuron counts helps create a funnel structure that progressively extracts more abstract features from the coefficients. ReLU is an activation function that is the input is positive that same number is output while if the input is negative the output is zero. ReLU was chosen for it computational efficiency and ability to mitigate the vanishing gradient problem, which occurs when weights are updated and become extremely small slowing training, during backpropagation.

Training terminates when validation performance stopped improving, helping prevent overfitting to the training data. The model achieve an $R^2$ score of 0.92 on the training set and 0.84 on the test set. An $R^2$ score explains the percentage of the variance in optimal initial guesses based on the coefficients.

3.3. **Comparison Methodology.** To evaluate the ML-enhanced approach against classical Newton-Raphson, we conducted 100 independent test cases using quadratic equations not seen during training. For each test:

(1) A random quadratic equation was generated using the same coefficient distributions as training.
(2) Equations with imaginary roots were excluded.
(3) For classical method: A random initial guess was generated by taking the true root and adding uniform noise in the range [-3,3], simulating realistic uncertainty in initial guesses.
(4) For ML method: The trained neural network predicted the initial guess from the coefficients.
(5) Both methods applied Newton's method with tolerance $10^-6$ and maximum 100 iterations.

A success was defined if convergence happened within 50 iterations to avoid counting methods that barely converged before the iteration limit. The following was tracked for the comparison:

- Success rate
- Average number of iterations for successful convergences
- Standard deviation of iterations
- Minimum and maximum iterations required

This experimental design isolate the impact of initial guess quality while controlling for all other factors. By using the same random equations for both methods, we ensure a fair comparison where differences in performance directly reflect the quality of initial guesses. However, it does need to be noted that for the classical the initial guesses were random whereas if this was compared to an initial guess made by someone with a strong domain knowledge the outcomes could favor the classical methods.

## 4. Results

4.1. **Classical Methods Comparison.** The three classical root-finding methods were tested on the three different functions mentioned in section 3.1 $f_1(x), f_2(x)$ and $f_3(x)$. The convergence behavior and efficiency varied significantly across methods and functions, as shown in Figures 2, 3, and 4. All of these show on the left panel iteration values approaching the true root while the right panel displays the absolute error on a logarithmic scale.

**Test 1:** $e^{-x} - x$ (Figure 2)

- **Bisection:**
  - Converged in 20 iterations
  - Root: 0.5671434402
  - f(root): $-2.35 \cdot 10^{-7}$
- **Newton-Raphson:**
  - Maximum iterations reached
  - Root: 0.5663110032
  - f(root): $1.30 \cdot 10^{-3}$
- **Secant:**
  - Converged in 4 iterations
  - Root: 0.5671432914

– f(root): $-1.57 \cdot 10^{-9}$

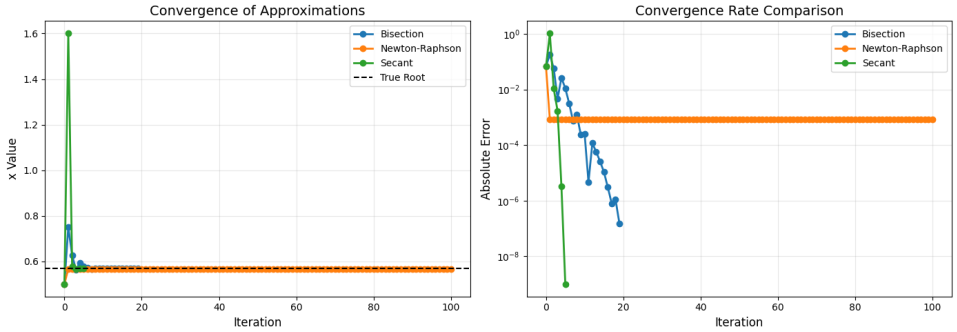


FIGURE 2. Convergence comparison for $f_1(x) = e^{-x} - x$

The convergence plots reveal differences in the method behavior. The Secant method demonstrated the fastest convergence. Newton's method showcases one of its weaknesses where it will oscillate without approaching the true root demonstrated by the orange lines in the graphs. Bisection provided guaranteed convergence with steady linear progression, requiring 20 iterations.

**Test 2:** $x^2 - 4$ (Figure 3)

- **Bisection:**
  - Converged in 22 iterations
  - Root: 2.0000002384
  - f(root): $9.54 \cdot 10^{-7}$
- **Newton-Raphson:**
  - Maximum iterations reached
  - Root: 2.5000000000
  - f(root): $2.25 \cdot 10^{0}$
- **Secant:**
  - Converged in 4 iterations
  - Root: 1.9999999987
  - f(root): $-5.11 \cdot 10^{-9}$

This demonstration is very similar to the last except with Newton's method diverging away from the root. Secant still appears to be the fastest, and Bisection still maintains its guaranteed convergence with a linear speed taking 18 iterations longer than Secant.

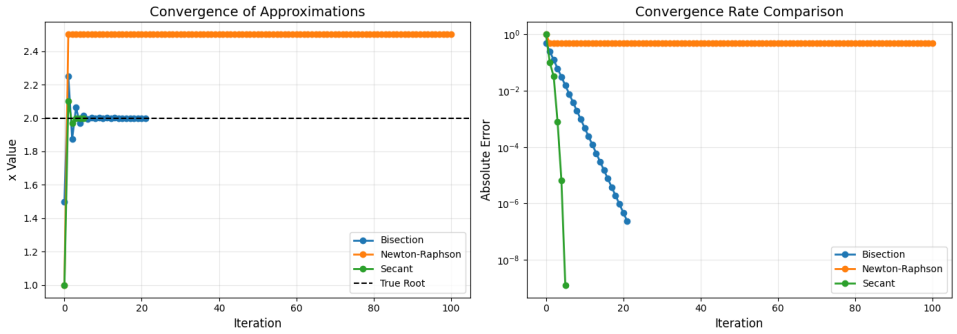


FIGURE 3. Convergence comparison for $f_2(x) = x^2 - 4$

**Test 3:** $x^3 - x - 2$ (Figure 4)

- **Bisection:**
  - Converged in 20 iterations
  - Root: 1.5213804245
  - f(root): $4.27 \cdot 10^{-6}$
- **Newton-Raphson:**
  - Maximum iterations reached
  - Root: 2.0000000000
  - f(root): $4.00 \cdot 10^{0}$
- **Secant:**
  - Converged in 6 iterations
  - Root: 1.5213797146
  - f(root): $4.64 \cdot 10^{-8}$

Once again Newton's method diverges away from the root. Showing how a poor initial guess can cause the algorithm to go away from the root. Secant and Bisection remain the same with both methods converging but at different speeds.

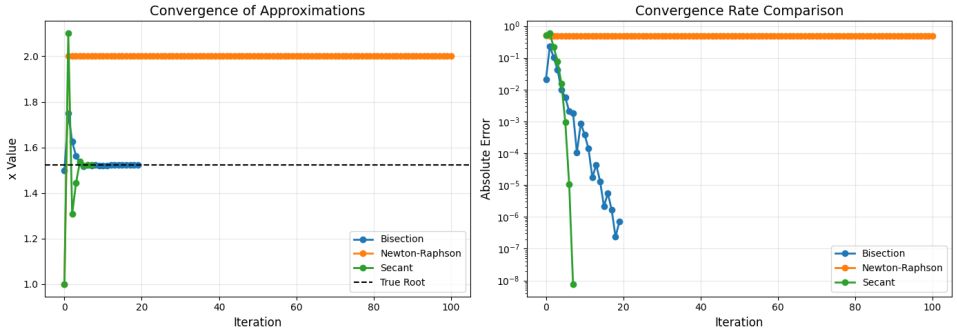


FIGURE 4. Convergence comparison for $f_3(x) = x^3 - x - 2$

In Summary: Across all three tests, the Secant method proved most reliable, converging successfully in 4-6 iterations for every function. Bisection guaranteed convergence but required 20-22 iterations due to its linear convergence. Newton-Raphson shows how a poor initial guess can destroy the convergence of the method.

In Test 1 the method converged to the correct root but oscillated never fully converging. In Tests 2 and 3 the method diverged completely. This evidence strongly motivates the need for better initial guess selection, which we address through machine learning.

4.2. **ML Model Performance.** The neural network was trained on $1,000$ randomly generated quadratic equations to learn the relationship between coefficients (a,b,c) and optimal initial guesses. The training process achieved strong performance metrics:

- **Training $R^2$ score:** 0.9236 (92.4% of variance explained)
- **Testing $R^2$ score:** 0.8403 (84% of variance explained)

The high test score indicates the model generalizes well to unseen equations rather than simply memorizing the training data. A high $R^2$ means the neural network can predict optimal initial guesses based solely on the quadratic coefficients, showcasing how ANN can capture patterns humans struggle to see. The small gap between the testing and training scores approximately 0.084 suggests minimal overfitting, validating the use of early stopping and the 80\20 train-test split.

4.3. **Classical vs ML-Enhanced Newton-Raphson.** To evaluate the practical impact of ML-predicted initial guesses, we generated 100 random quadratic equations and tested both the classical and ML-enhanced version of Newton-Raphson. After filtering out equations with imaginary roots 73 valid test cases remained. Figure 5 presents the comparison through both distribution and box plot visuals.

**Performance Metrics:**

- *Classical Newton-Raphson with random initial guess*
  - Success rate: 100%
  - Average iterations: 4.45
  - Standard Deviation: 1.22
  - Range: 2-7 iterations
- *ML-Enhanced Newton-Raphson with predicted initial guess*
  - Success rate: 100%
  - Average iterations: 2.49
  - Standard Deviation: 0.70
  - Range: 1-5 iterations

**Analysis of Results:** The histogram in the left panel of Figure 5 shows a sharp shift in the distribution of iteration counts. The classical method shows a broad distribution spread across 2-7 with the most taking 4 iterations. While the ML-enhanced model resided heavily on the left side between 1-5 with the most taking 2 iterations and very few requiring 7 iterations. This demonstrates the neural network consistently predicting initial guesses very close to the true roots.

The box plot in the right panel of Figure 5 provides a clear visual comparison of convergence speed and consistency. The classical method's box spans from 4 to 5 iterations with a median at 4, and shows considerable spread with outliers reaching 7 iterations. The ML-enhanced method's box is compressed between 2 and 3 iterations with a median at 2, and its spread only has outliers to 5 iterations. Both cases have outliers represented by the tiny circles seen in Figure 5.

**Key Improvements:**

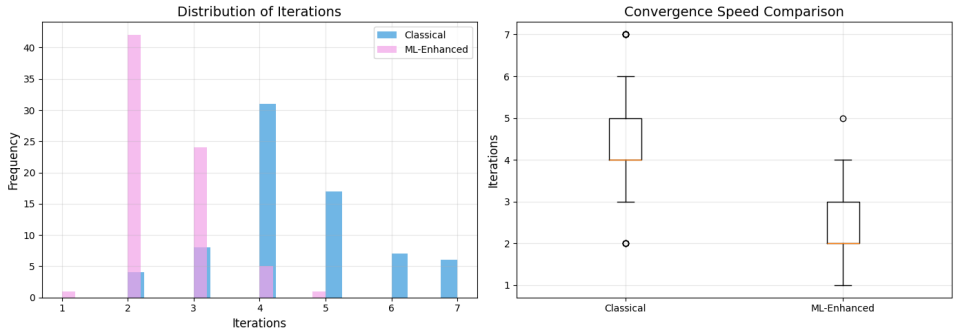


FIGURE 5. Classical vs ML-enhanced

(1) **Iteration Reduction:** The ML-enhanced method achieve a 44.0% reduction in average iterations. This would result in substantial computational savings, especially for applications requiring repeated root-finding.
(2) **Consistency:** The standard deviation decreased by 42.6% indicating the ML prediction provide more reliable performance across diverse equations. Lower variance means more predictable runtime, which is valuable in real-time or resource constrained applications.
(3) **Success Rate:** Both methods achieved 100% success rate because of how the code tracked successes. It could only be counted successful if the iterations were less than 50 and if the roots were real numbers. Since the random initial guesses used for the classical implementation were close to the real root, and the ML-enhanced was trained to predicted well the iterations side was never an issue.

The results demonstrate how machine learning can successfully address Newton-Raphson's primary weakness, sensitivity to initial guesses, without introducing new failure modes. The ML model learned to recognize coefficient patterns that indicate root locations, effectively automating the domain expertise typically required for choosing good initial guesses.

## 5. Discussion

5.1. **Interpretation of Results.** The experimental results confirm our central hypothesis that machine learning can effectively address Newton's methods sensitivity to initial guesses. The 44% reduction in average iterations demonstrate that a neural network trained on coefficient patterns can help automate the selection of good starting points, a task which can require a lot of trial and error or domain knowledge. The improved consistency, 43% reduction in standard deviation, is equally important. It ensures predictable performances across diverse problems. Once again it has to be kept in mind that for the classical method the initial guesses were random, but since they were kept uniformly close to the root it helps negate any error from the random guess being extremely poor.

5.2. **Why Machine Learning Works.** Neural networks excel at pattern recognition in high dimensional spaces. Our model learned implicit relationships between quadratic coefficients (a,b,c) and root locations. These patterns are usually mathematically defined by the quadratic formulas, but the network numerically

optimized them through training, not a formula. This means you do not need to explicitly program initial guess selections formulas or rules. The network discovers through gradient descent which combinations of coefficients predict initial guesses near roots. The architecture used funnels progressively abstracts these patterns which enables the generalization to unseen patterns and equations. This is useful because these patterns are minute and difficult for humans to recognize and tedious to code manually.

5.3. **Limitations.** This study focused exclusively on quadratic equations, which represent a simple class of functions with known analytic solutions for the roots. The approach may not generalize directly to high-degree polynomials or non polynomial functions. The neural network also requires sufficient training data and computational resources for initial training phase; however, once trained the model's inference is very fast. Also our test cases used uniform random coefficient distributions; real-world problems may have different coefficient patterns that could affect how the model performs. The classical method's random initial guesses, while also uniform, may not accurately reflect how an expert in the domain or with the problem would select guesses.

5.4. **Future Work.** There are several directions this project could explore further. First, extending the machine learning approach to high degree polynomials and non-polynomial functions. Second, developing models for other classical methods such as the Secant method that requires two initial guesses. Third, exploring how the model generalizes from simple functions to complex ones. Finally fine-tuning the models to optimize performance. This includes exploring tolerance, maximum iterations, and changing what is considered a successful rate.

## 6. Conclusion

This study demonstrates machine learning can substantially improve Newton-Raphson method's efficiency by automating initial guess selection. Our neural network, trained on $1,000$ random quadratic equations, achieved $R^2 = 0.84$ on test data and reduced average iterations by 44% (4.45 to 2.49) while improving consistency by 43% across 73 test cases. Both classical and ML-enhanced methods maintained 100% success rates, confirming the ML approach enhances performance without compromising reliability. These results also validate the central premise that neural networks can learn coefficient-to-root patterns that traditionally require domain expertise. Combining machine learning and root-finding methods can lead to great computational savings which can help in the many fields where root-finding is a necessity.

## 7. Appendix

### 7.1. Code for Classical methods.

### 7.2. Bisection Method.

```
def bisection(f, a, b, tol, max_iter=100):
  """
  Finds a root of f(x) in the interval [a, b] using the bisection method.
  Parameters:
  f: function - the function to find the root of
```

```
    a, b: float
    Interval endpoints with f(a) * f(b) < 0 (root must lie inside).
    tol: float
    Tolerance for stopping (default: 1e-6).
    max_iter: int
    Maximum number of iterations (default: 100).
    Returns:
    Approximate root of f(x), interations, and history
    """
    history = []
    #Check to see if the endpoints are roots
    if abs(f(a)) < tol:
        print(f"a is already a root")
        return a
    if abs(f(b)) < tol:
        print(f"b is already a root")
        return b
    # Check that the root exists in [a, b]
    if f(a) * f(b) >= 0:
        print("Bisection method fails: f(a) and f(b) must have opposite signs.")
        return None
    for i in range(max_iter):

        #Check to see if the endpoints are roots
        if abs(f(a)) < tol:
            print(f"Converged in {i+1} iterations.")
            return a
        if abs(f(b)) < tol:
            print(f"Converged in {i+1} iterations.")
            return b
        # Compute midpoint
        c = (a + b) / 2.0
        history.append(c)
        # Check if the midpoint is a root or close enough
        if abs(f(c)) < tol or (b - a) / 2.0 < tol:
            print(f"Converged in {i+1} iterations.")
            return c, i+1, history
    # Decide which subinterval contains the root
        if f(a) * f(c) < 0:
            b = c # Root is in [a, c]
        else:
            a = c # Root is in [c, b]

    print("Maximum iterations reached.")
    return (a + b) / 2.0, max_iter, history # Return best estimate
```

### 7.3. **Newton-Raphson Method.**

```
def newton_raphson(f, df, x0, tol, max_iter = 100):
  """
  Finds a root of f(x) using Newton-Raphson method.
  Parameters:
  f: function - the function to find the root of
  df: function - the derivative of f(x)
  x0: float initial guess
  tol: float - Tolerance for stopping
  max_iter: int - mmaximum number of iterations

  returns:
  root, iterations, history
  """

  x = x0;
  history = [x0]

  for i in range(max_iter):
    fx = f(x)
    dfx = df(x)

    #Check if derivative is too small
    if abs(dfx) <1e-12:
      print("Newton-Raphson method fails: derivative is too small.")
      return None, i, history

    #Newton Raphson iteration
    x_new = x - fx/dfx
    history.append(x_new)

    #Check if root is close enough
    if abs(x_new - x) < tol or abs(f(x_new)) < tol:
      print(f"Converged in {i+1} iterations.")
      return x_new, i+1, history

  x = x_new
  print("Maximum iterations reached.")
  return x, max_iter, history
```

### 7.4. **Secant Method.**

```
def secant (f, x0, x1, tol = 1e-6, max_iter = 100):
  """
  Finds a root of f(x) using the secant method

  Parameters;
  f: function - the function to find the root of
  x0, x1: float - two initial guesses
  tol: float - tolerace for stopping
  max_iter: int - maximum number of iterations
```

```
  Returns:
  root, iterations, history
  """

  history = [x0, x1]

  for i in range(max_iter):
    fx0 = f(x0)
    fx1 = f(x1)

    #Check if denominator is too small
    if abs(fx1 - fx0) < 1e-12:
      print("Secant method fails: denominator is too small.")
      return None, i, history

    #Secant method iteration
    x2 = x1-fx1 * (x1-x0)/(fx1-fx0)
    history.append(x2)

    #Check if root is close enough
    if abs(x2 - x1) < tol or abs(f(x2)) < tol:
      print(f"Converged in {i+1} iterations.")
      return x2, i+1, history

    x0 = x1
    x1 = x2

  print("Maximum iterations reached.")
  return x1, max_iter, history
```

### 7.5. Comparison and Analysis.

```
def compare_all_methods(f, df, a, b, x0, tol = 1e-6):
  """
  Compare all classical methods on the current function f(x)

  Parameters:
  f: function - the function to find the root of
  df: function - the derivative of f(x)
  a, b: float - interval endpoints
  x0: float - initial guess for Newton-Raphson and Secant
  tol: floar - tolerance
  """

  print("-"*70)
  print("Comparison of Classical Root-Finding Methods")
  # print(f"Function: f(x) = e^(-x) - x") # This will now be dynamic or removed
  print(f"Tolerance: {tol}")
  print("-"*70)
```

```
  results = {}

  #Bisection
  print("\n1. Bisection Method")
  print("-"*70)
  # Pass f to bisection
  root, iters, hist = bisection(f, a, b, tol)
  if root is not None:
    results['Bisection'] = {'root': root, 'iterations': iters, 'history': hist}
    print(f"Root: {root:.10f}")
    print(f"f(root): {f(root):.2e}")


  #Newton-Raphson
  print("\n2. Newton-Raphson Method")
  print("-"*70)
  # Pass f and df to newton_raphson
  root, iters, hist = newton_raphson(f, df, x0, tol)
  if root is not None:
    results['Newton-Raphson'] = {'root': root, 'iterations': iters, 'history': hist}
    print(f"Root: {root:.10f}")
    print(f"f(root): {f(root):.2e}")

  #Secant
  print("\n3. Secant Method")
  print("-"*70)
  # Pass f to secant
  root, iters, hist = secant(f, x0, x0+1.0, tol)
  # Added x1=x0+1.0 as secant needs two initial guesses
  if root is not None:
    results['Secant'] = {'root': root, 'iterations': iters, 'history': hist}
    print(f"Root: {root:.10f}")
    print(f"f(root): {f(root):.2e}")

  print("\n"+"="*70)
  print("Summary")
  print("-"*70)
  print(f"{'Method':<20}{'Root':<15}{'Iterations':<12}{'f(root)'}")
  print("-"*70)
  for method, data in results.items():
    print(f"{method:<20}{data['root']:15.10f}{data['iterations']:<12}{f(data['root']):.2e}")

  return results


"Plot results"
def plot_convergence(results, true_root=None):
```

```
  """Plot Convergence history for all methods"""
  plt.figure(figsize=(14, 5))


  #Plot 1: Iteration values
  plt.subplot(1, 2, 1)
  for method, data in results.items():
    plt.plot(data['history'],marker = 'o', label=method, linewidth = 2)

  if true_root:
    plt.axhline(y=true_root, color='black', linestyle='--', label='True Root')

  plt.xlabel('Iteration', fontsize = 12)
  plt.ylabel('x Value', fontsize = 12)
  plt.title('Convergence of Approximations', fontsize = 14)
  plt.legend()
  plt.grid(True, alpha = 0.3)
  #Plot 2: Error if the true root cannot be provided
  if true_root:
    plt.subplot(1, 2, 2)
    for method, data in results.items():
      errors = [abs(x-true_root) for x in data['history']]
      plt.semilogy(errors, marker = 'o', label=method, linewidth = 2)

      plt.xlabel('Iteration', fontsize = 12)
      plt.ylabel('Absolute Error', fontsize = 12)
      plt.title('Convergence Rate Comparison', fontsize = 14)
      plt.legend()
      plt.grid(True, alpha = 0.3)

  plt.tight_layout()
  plt.show()
```

### 7.6. **Testing multiple Functions.**

```
def test_multiple_functions():
  print("\n"+"*"*70)
  print("Testing Classical Root Finding Methods")
  print("*"*70)

  print("\n\nTest 1: f(x) = e^(-x)-x")
  print("="*70)

  def f(x):
    return math.exp(-x)-x
  def df(x):
    return -math.exp(-x)-1
  results = compare_all_methods(f, df, a=0, b=1, x0=0.5, tol=1e-6)
  plot_convergence(results, true_root = 0.567143290409784)
```

```
  #Redfine f and df temporarily
  print("\n\nTest 2: f(x) = x^2-4")
  print("="*70)
  def f(x):
    return x**2-4
  def df(x):
    return 2*x

  results2 = compare_all_methods(f, df, a=0, b=3, x0=1, tol = 1e-6)
  plot_convergence(results2, true_root = 2.0)

  print("\n\nTest 3: f(x) = x^3 - x - 2")
  print("="*70)

  #Redfine f and df temporarily
  def f(x):
    return x**3 - x - 2
  def df(x):
    return 3*x**2 - 1

  results3 = compare_all_methods(f, df, a=1,b=2,x0=1, tol=1e-6)
  plot_convergence(results3, true_root=1.521379706804568)


#Run it
test_multiple_functions()
```

### 7.7. Code for Machine Learning.

### 7.8. Generate Training Data.

```
def generate_training_data(n_samples = 1000):
  """
  Generates training data for neural network
  for quadratic equations ax^2 + bx + c = 0
  """
  np.random.seed(72)

  training_data = []

  for _ in range(n_samples):
    #Random quadratic
    a = np.random.uniform(0.5, 5)
    b = np.random.uniform(-10, 10)
    c = np.random.uniform(-10, 10)

    #Calculate the actual roots using quadratic formula
    discriminant = b**2 - 4*a*c

    if discriminant >= 0:
```

```
      root1 = (-b + np.sqrt(discriminant)) / (2*a)
      root2 = (-b - np.sqrt(discriminant)) / (2*a)

      #Use the root closest to 0
      if abs(root1) < abs(root2):
        best_root = root1
      else:
        best_root = root2


      #Store the coefficients and best guess
      training_data.append([a, b, c, best_root])

  training_data = np.array(training_data)
  X = training_data[:, :3] #Features a, b, and c
  y = training_data[:, 3] #Target good initial guess

  return X, y
```

7.9. **Train the model.**

```
def train_initial_guess_predictor():
  """ Train the neural network to predict good initial guesses"""
  print("Training Neural Network")

  X, y = generate_training_data()
  X_train, X_test, y_train, y_test = train_test_split(X,y,
  test_size = 0.2, random_state = 72)
  model = MLPRegressor(
      hidden_layer_sizes = (50, 30, 20),
      activation = 'relu',
      max_iter = 500,
      random_state = 72,
      early_stopping = True,
      verbose = False
  )

  model.fit(X_train, y_train)

  train_score = model.score(X_train, y_train)
  test_score = model.score(X_test, y_test)

  print("Training complete")
  print(f"Train R^2 Score: {train_score}")
  print(f"Test R^2 Score: {test_score}")

  return model
model = train_initial_guess_predictor()
```

7.10. **ML Enhanced Newton.**

```
def ml_enhanced_newton(a, b, c, tol = 1e-6, max_iter = 100):
  """Newton-Raphson with ML predicted initial guess"""
  #Define a function
  def f(x):
    return a*x**2 + b*x + c
  def df(x):
    return 2*a*x + b

  features = np.array([a, b, c]).reshape(1, -1)
  # Reshape to 2D array
  x0_predicted = model.predict(features)[0]

  root, iterations = newton_raphson_standard(f, df,
  x0_predicted, tol, max_iter)
  return root, iterations, x0_predicted
```

7.11. **Compare Classical vs ML.**

```
def compare_classical_vs_ml(model, n_tests = 100):
  """Compare classical vs ML-enhanced Newton-Raphson"""
  np.random.seed(413)

  classical_iterations = []
  ml_iterations = []
  classical_successes = 0
  ml_successes = 0
  classical_failures = 0
  ml_failures = 0

  print("="*70)
  print("Comparing Classical vs ML-enhanced Newton-Raphson")
  print("="*70)
  tests_with_both_success = 0
  for i in range(n_tests):
    a = np.random.uniform(0.5, 5)
    b = np.random.uniform(-10, 10)
    c = np.random.uniform(-10, 10)

    discriminant = b**2-4*a*c
    if discriminant < 0:
      continue

    def f(x):
      return a*x**2 + b*x + c
    def df(x):
      return 2*a*x + b

    #Classical with random guess
    true_root = (-b + np.sqrt(discriminant))/(2*a)
    x0_random = true_root + np.random.uniform(-3, 3)
```

```
    root_classical, iterations_classical =
    newton_raphson_standard(f, df, x0_random, tol = 1e-6,
    max_iter = 100)

    #ML-enhanced with predicted
    root_ml, iterations_ml, x0_predicted = ml_enhanced_newton(a,
    b, c, tol = 1e-6, max_iter=100)

    #Track individual success rates
    classical_success = (root_classical is not None and
    iterations_classical<50)
    ml_success = (root_ml is not None and iterations_ml<50)

    if classical_success:
      classical_iterations.append(iterations_classical)
      classical_successes += 1
    else:
      classical_failures += 1

    if ml_success:
      ml_iterations.append(iterations_ml)
      ml_successes += 1
    else:
      ml_failures += 1

    #Track cases were both succeeded
    if classical_success and ml_success:
      tests_with_both_success += 1

  total_tests = classical_successes + classical_failures
  #Print Results
  print(f"Total tests: {total_tests}")
  print(f"Tests with both success: {tests_with_both_success}")

  print(f"\nClassical Newton-Raphson(random initial guess):")
  print(f"  Success rate: {classical_successes/total_tests*100:.2f}%")
  if classical_iterations:
    print(f"  Average iterations: {np.mean(classical_iterations):.2f}")
    print(f"  Std dev: {np.std(classical_iterations):.2f}")
    print(f"  Min/Max: {np.min(classical_iterations)}/
    {np.max(classical_iterations)}")
  print(f"\nML-Enhanced Newton-Raphson(random initial guess):")
  print(f"  Success rate: {ml_successes/total_tests*100:.2f}%")
  if ml_iterations:
    print(f"  Average iterations: {np.mean(ml_iterations):.2f}")
    print(f"  Std dev: {np.std(ml_iterations):.2f}")
    print(f"  Min/Max: {np.min(ml_iterations)}/{np.max(ml_iterations)}")
```

```
  print("Key Improvements")
  if classical_successes > 0 and ml_successes > classical_successes:
    improvement_rate = (ml_successes - classical_successes) /
    classical_successes * 100
    print(f"  ML-Enhanced Newton-Raphson is
    {improvement_rate:.2f}% more efficient.")
  elif ml_successes > classical_successes:
    print(f"  ML succeeded {ml_successes} times vs Classical
    {classical_successes}")

  if classical_iterations and ml_iterations:
    iter_improvement_rate = (np.mean(classical_iterations) -
    np.mean(ml_iterations)) / np.mean(classical_iterations)*100
    print(f"  ML-Enhanced Newton-Raphson uses {iter_improvement_rate:.2f}%
    less iterations on average.")

    if np.std(ml_iterations) < np.std(classical_iterations):
      consistency = (np.std(classical_iterations)
      np.std(ml_iterations))
      /np.std(classical_iterations)*100
      print(f"  ML-Enhanced Newton-Raphson is {consistency:.2f}%
      more consistent.")

  print("="*70)
  return classical_iterations, ml_iterations
```

7.12. **Plot Comparison.**

```
def plot_comparison(classical_iter, ml_iter):
  """Visualize the comparison"""
  fig, axes = plt.subplots(1, 2, figsize=(14,5))

  #Histogram
  axes[0].hist(classical_iter, bins = 20, alpha = 0.7,
  label = 'Classical', color = '#3498db')
  axes[0].hist(ml_iter, bins = 20, alpha = 0.7, label =
  'ML-Enhanced', color = '#f2a2e8')
  axes[0].set_xlabel('Iterations', fontsize = 12)
  axes[0].set_ylabel('Frequency', fontsize = 12)
  axes[0].set_title('Distribution of Iterations', fontsize = 14)
  axes[0].legend()
  axes[0].grid(True, alpha = 0.3)

  #Box plot
  axes[1].boxplot([classical_iter, ml_iter],
  tick_labels=['Classical', 'ML-Enhanced'])
  axes[1].set_ylabel('Iterations', fontsize = 12)
  axes[1].set_title('Convergence Speed Comparison', fontsize = 14)
  axes[1].grid(True, alpha = 0.3)
```

```
  plt.tight_layout()
  plt.show()
#Run the comparison
classical_iters, ml_iters = compare_classical_vs_ml(model, n_tests = 100)

#Plot results
plot_comparison(classical_iters, ml_iters)
```

## References


[1] K. E. Emmert and P. White, "Solutions of Equations in One Variable," Chapter 2, *Numerical Analysis Course Notes*, Tarleton State University, Fall 2025.

[2] R. L. Burden, J. D. Faires, and A. M. Burden, *Numerical Analysis*, 10th ed. Boston, MA: Cengage Learning, 2016.

[3] J. Shawe-Taylor and C. Cristianini, "Machine Learning," *IEEE Software*, vol. 33, no. 4, pp. 99–101, July-Aug. 2016.

[4] N. Shlezinger, N. Farsad, Y. C. Eldar, and A. J. Goldsmith, "A Very Brief Introduction to Machine Learning With Applications to Communication Systems," *IEEE Transactions on Cognitive Communications and Networking*, vol. 5, no. 2, pp. 210–233, June 2019.

[5] A. K. Jain, J. Mao, and K. M. Mohiuddin, "Artificial Neural Networks: A Tutorial," *Computer*, vol. 29, no. 3, pp. 31–44, Mar. 1996.

[6] A. Sayal *et al.*, "Neural Networks And Machine Learning," in *2023 IEEE 5th International Conference on Cybernetics, Cognition and Machine Learning Applications (ICCCMLA)*, Hamburg, Germany, 2023, pp. 58–63, doi: 10.1109/ICCCMLA58983.2023.10346612.



Tarleton State University, Department of Mathematics
*Email address*: sadie.shudde@go.tarleton.edu