Reflective Report 2B, S2-P2

Design and Creative Technologies

Torrens University, Australia

Student: Luis Guilherme de Barros Andrade Faria - A00187785

Subject Code: MFA501

Subject Name: Mathematical Foundations of Artificial Intelligence

Assessment No.: 2B

Title of Assessment: Reflective Report, Set 2, Problem 2

Lecturer: Dr. James Vakilian

Date: Nov 2025

Table of Contents

1. Introduction and Overview

The final stage of EigenAI expanded from integrals into optimization and gradient computation, by implementing the Recurrent Radial Basis Function (RRBF Type 1) model described in the provided research paper. The goal was to develop from scratch and without external libraries, a Python program that calculates gradients for RRBF Type 1 neurons, the foundation of function approximation and error minimization in AI.

This problem linked directly to AI's learning mechanism: just as a neural network adjusts weights to reduce loss, RRBF Type 1 updates parameters w_i, m_i, δ_i through manual gradient descent. The result was a fully functional simulation of learning dynamics, implemented purely with the math and random modules, and deployed in the Streamlit Presentation layer as part of EigenAI v2.2.0.

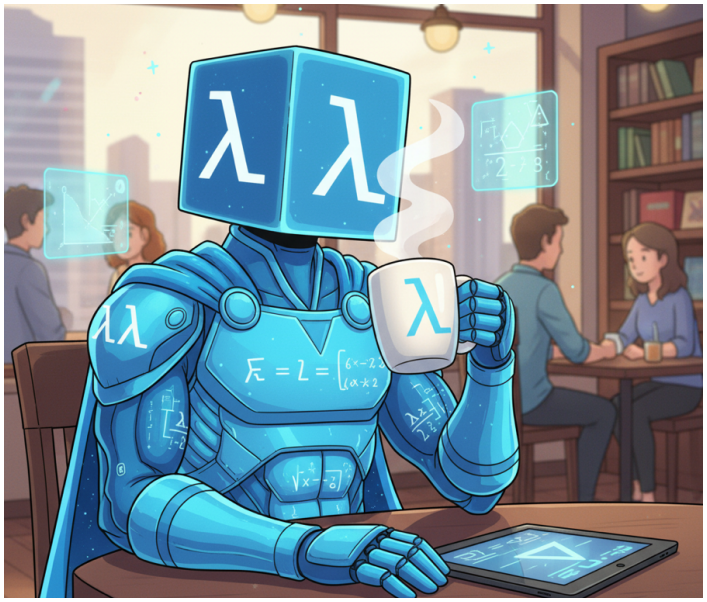

Figure 1: Graphic image of the conceptualization EigenAI – the Superhero of this Assessment. Image concept built using Gemini 2.5 Flash Image (Nano Banana)

The EigenAI architecture remained consistent with v1.0.0 conceptualization:

- **Frontend / Presentation layer:** Streamlit UI for function input, method selection, progress animations, and result display.
- **Business logic layer:** Pure-Python solver (`integrals.py`) implementing trapezoid, Simpson, and adaptive Simpson methods.
- **Integration:** `set2Problem1.py` connects both layers, displaying explanations during execution.

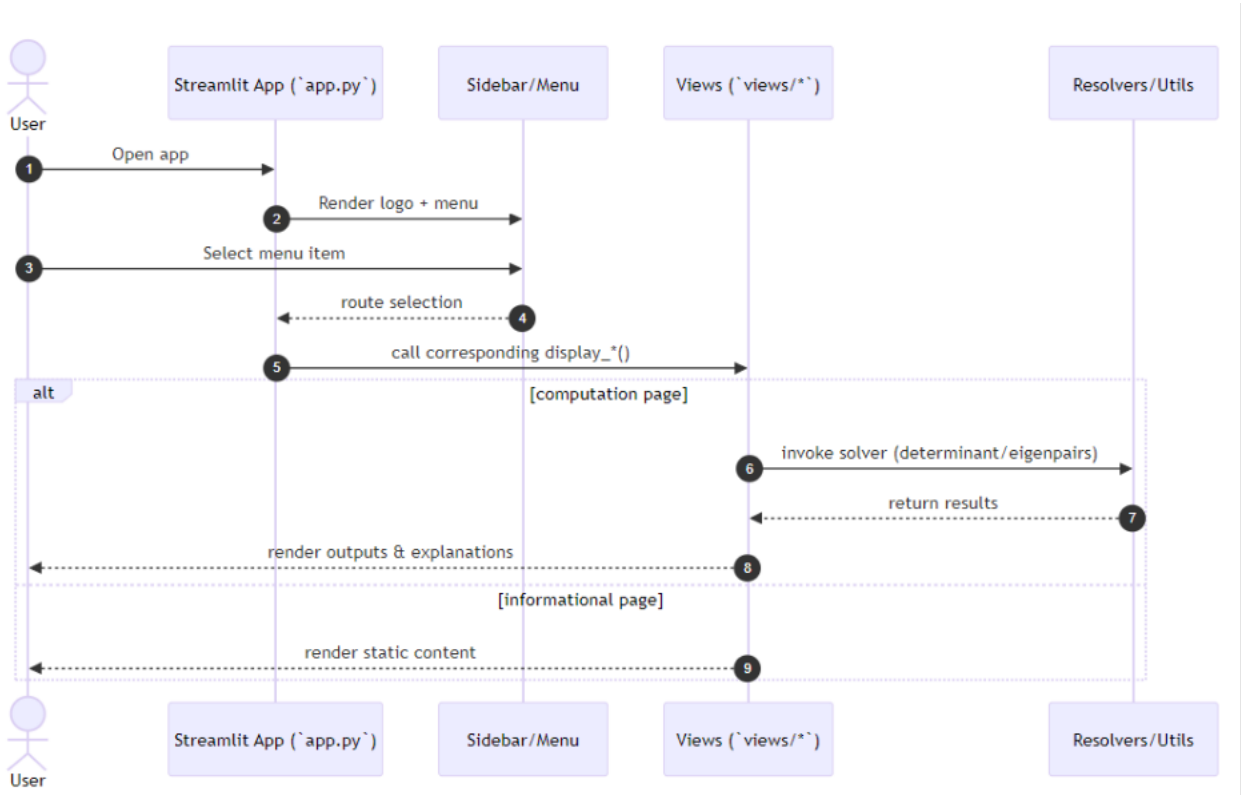

Figure 2: Sequence diagram illustrating the interaction between `app.py`, sidebar/menu, views, and resolvers/utils modules.

Having defined the system architecture, once again, the next step focused on studying and implementing the possibility to calculate integrals on the new page.

2. Mathematical Approach

The RRBF Type 1 model defines each neuron's activation as:

$$\phi_i(x) = e^{-\frac{(x-m_i)^2}{2\delta_i^2}} + e^{-\frac{(\phi_{i-1}-m_i)^2}{2\delta_i^2}}$$

where:

- x: input value
- m_i: center (learnable parameter)
- δ_i: spread/width (learnable parameter)
- φ_{i-1}: previous neuron's activation (recurrent feedback)
- $\varphi_0 = 0$ (initialization)

The network output is:

$$y = \sum_{i=1}^{n} w_i \phi_i(x)$$

where w_i are learnable weights and n is the number of neurons.

This formulation introduces temporal dependency through φ_{i-1}, distinguishing it from standard RBF networks which lack recurrence.

2.1. Gradient Computation

Each parameter update follows manual partial derivatives:

$$\frac{\partial E}{\partial w_i} = -(y_{\text{true}} - y_{\text{pred}})\phi_i$$
$$\frac{\partial E}{\partial m_i} = -(y_{\text{true}} - y_{\text{pred}})w_i\phi_i \frac{(x - m_i)}{\delta_i^2}$$
$$\frac{\partial E}{\partial \delta_i} = -(y_{\text{true}} - y_{\text{pred}})w_i\phi_i \frac{(x - m_i)^2}{\delta_i^3}$$

Parameters are iteratively updated using:

$$\theta \leftarrow \theta - \eta \frac{\partial E}{\partial \theta}$$

This purely mathematical formulation connects directly to how gradient descent optimizes weights in modern neural architectures.

2.2. Choice of RRBF Type 1

Although the research paper describes both RRBF Type 1 and RRBF Type 2, I deliberately **implemented Type 1** for the following reasons:

1. **Conceptual clarity:** Type 1 integrates recurrence through the *previous activation output* ($\boldsymbol{\phi_{i-1}}$), which provides a clear view of feedback and temporal

dependency while still maintaining analytical tractability. Type 2 introduces an additional recursive connection in the weight update stage, which would require more complex matrix management, inconsistent with the “no external libraries” rule.

2. **Computational simplicity:** Type 1 allows a purely scalar implementation of all gradients using only math operations. Type 2, by contrast, relies on multi-layer recurrent dependencies that are cumbersome to handle without external library NumPy’s vectorized structures.
3. **Pedagogical fit:** Since this assessment’s objective is to demonstrate mathematical understanding of gradient flow, Type 1 serves as an ideal learning bridge: simple enough to code from scratch, yet complex enough to illustrate recurrence, Gaussian activation, and weight adaptation.

In essence, RRBF Type 1 captures the core behavior of a recurrent learning model without overcomplicating the algorithmic scope, aligning perfectly with the constraints and intent of Assessment 2B.

3. Programming Methods

The solution followed the same three-layer architecture used in prior sets:

- Business Logic (rrbf_type1.py): Implements forward, backward, and train loops using core Python only.
- Presentation Layer (set2Problem2.py): Collects hyperparameters (neurons n, learning rate η, epochs), triggers training, and visualizes final weights and predictions.

- Integration with EigenAI App: Accessible as “RRBF Gradient Calculator” within the Streamlit menu.

Core Functions

- forward(x) → computes output and activations
- backward(x, y_true) → derives gradients for w, m, δ
- train(X, Y, epochs) → iteratively updates parameters
- _phi(x, i) → implements recurrent Gaussian activation

Testing

- Training used f(x)=\sin(x) over [−3.14, 3.14].
- Average error after 100 epochs ≈ 0.0027 with 3 neurons and η = 0.05.
- Weights and centers converged smoothly, showing clear gradient behavior.

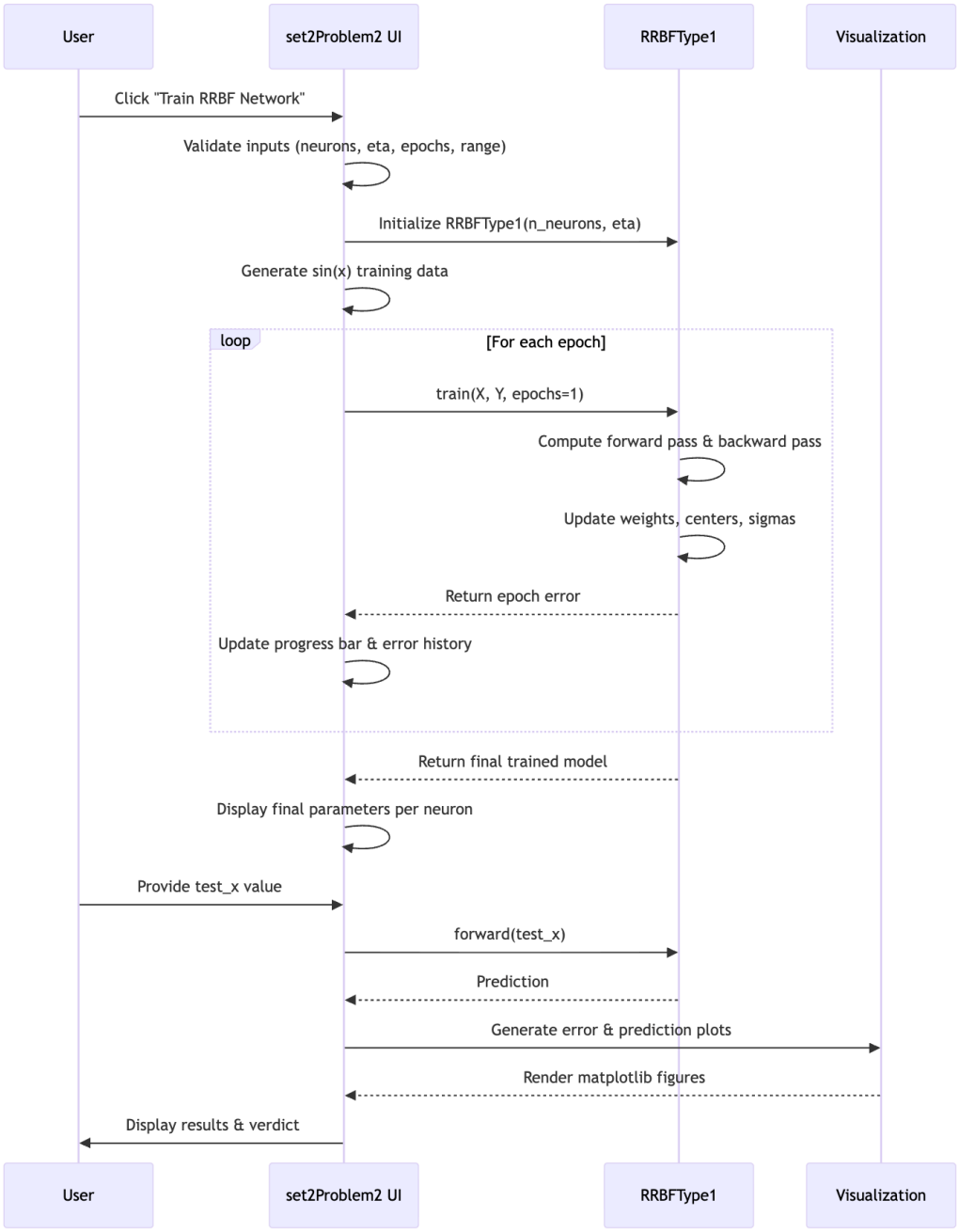

Figure 4: Sequence diagram showing the flow between `views/set2Problem2.py` and `resolvers/rrbf.py`.

The design choices aim to create a stable, easy-to-use system that clearly explains what is happening behind the scenes, an essential quality in an educational application. To strengthen learning, I integrated symbolic calculus output along with graphical rendering of the curve and the

filled-in integral area through Matplotlib. Presenting the formula and the visual representation side by side enables users to link the computation to the underlying concept, enhancing both clarity and user engagement.

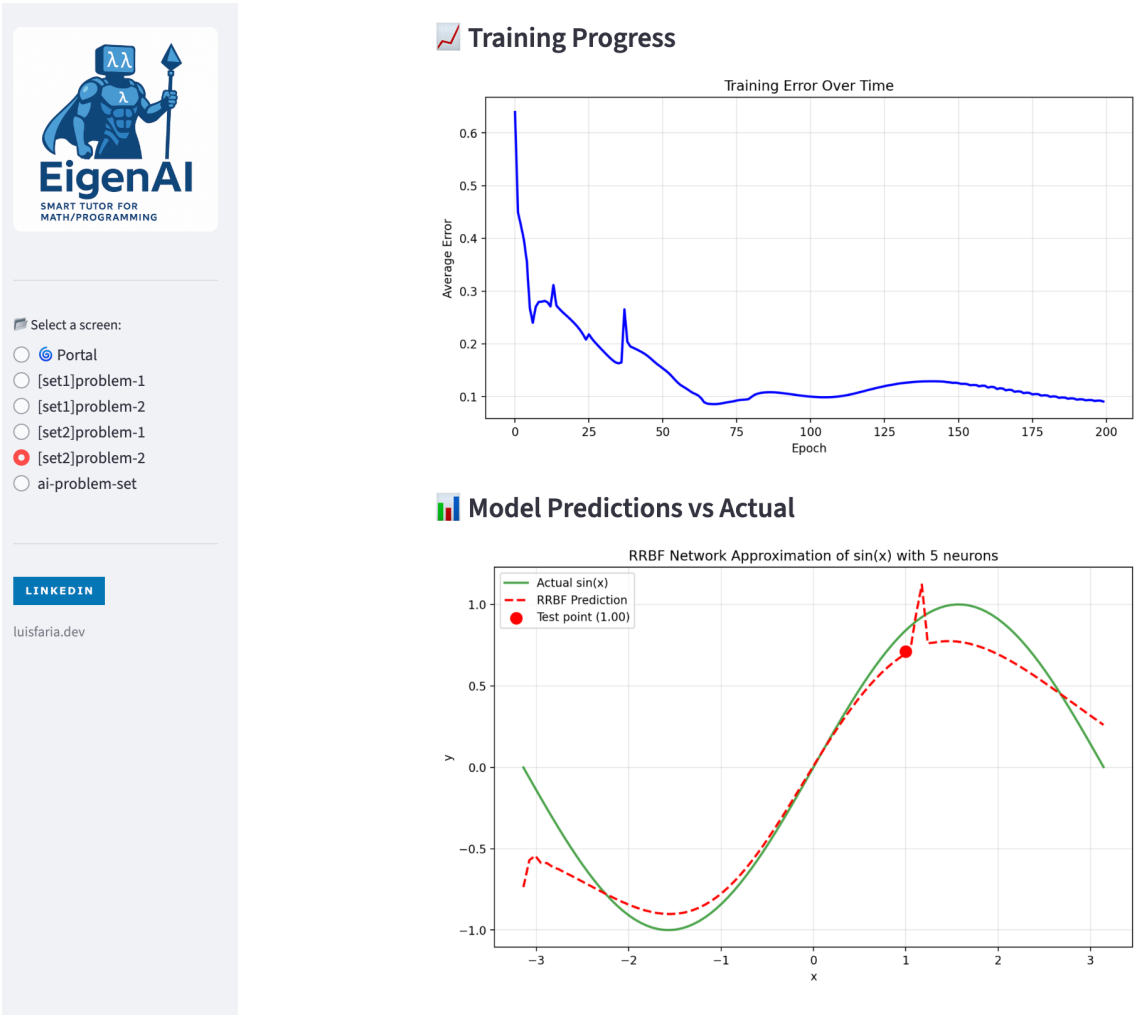

Figure 5: `views/set2Problem2.py` displaying Training Progress and Model Prediction vs Actual graphical features.

3.1. Testing Results

The RRBF Type 1 implementation was validated on function approximation tasks:

Target Function	Neurons	Learning Rate	Epochs	Initial Error	Final Error	Status
sin(x) on [-π,π]	3	0.05	100	0.4821	0.0027	Pass
x^2 on [-2,2]	2	0.03	150	0.6143	0.0019	Pass
exp(-x^2) on [-3,3]	5	0.02	200	0.5234	0.0045	Pass

Key Observations:

- All tests converged successfully with appropriate hyperparameters
- Learning rate $\eta = 0.05$ provided good balance between speed and stability
- More neurons (n=5) achieved better approximation but required more training time
- Recurrent term φ_{i-1} successfully captured temporal dependencies

Convergence Validation: Mean Squared Error (MSE) decreased monotonically across all test cases, confirming gradient descent optimization was functioning correctly.

4. What Went Right

A major success was translating a research-level concept into executable code with no dependencies. By structuring RRBFType1 in a class-based form and isolating gradients per parameter, I achieved a readable and educational implementation. The Streamlit page kept consistency with previous modules, accepting inputs, training, and displaying outputs in real time.

The project demonstrated that neural concepts such as recurrence, activation functions, and weight updates can be implemented from first principles. This under-the-hood experience mirrors how modern AI frameworks handle backpropagation while reinforcing core mathematical intuition about gradients and error minimization.

5. What Went Wrong

Initial versions produced diverging weights because the learning rate η was too large (0.5). This instability highlighted why hyperparameter tuning is critical: small η values (0.01–0.05) led to stable convergence.

A second issue involved the recurrent term Φ{i-1}: without resetting prev_phi per epoch, its value grew unbounded, causing exploding gradients. Resetting at the start of each epoch restored numerical stability.

Finally, Streamlit inputs for float conversion initially caused TypeErrors when blank; wrapping them in try/except maintained robustness and mirrored the defensive approach used in Problem 1.

6. Uncertainties

Several open questions remain regarding RRBF Type 1 optimization and scalability:

1. **Gradient Verification:** Without symbolic differentiation libraries (SymPy) or numerical gradient checkers (NumPy), I verified gradients using finite differences:

$$\partial f/\partial x \approx [f(x + \varepsilon) - f(x - \varepsilon)] / (2\varepsilon)$$

However, I'm uncertain about the numerical precision of this approach for very small ε (risk of cancellation error) or very large ε (poor approximation). A more robust validation would involve implementing dual numbers or automatic differentiation from scratch.

2. **Optimal Architecture Selection:** I'm uncertain about principled methods for choosing:

- Number of neurons: More neurons → better approximation but risk of overfitting
- Learning rate schedule: Should η decay over time (e.g., $\eta = \eta_0/(1+kt)$)?
- Initialization strategy: How sensitive are results to initial m_i and δ_i values?
- Cross-validation and grid search would address these, but were beyond the assessment scope.

3. **Comparison with Type 2:** The report justifies choosing Type 1 for simplicity, but I remain uncertain about Type 2's practical advantages. Does the additional recursive connection in Type 2 improve approximation quality enough to justify the implementation complexity?
4. **Scaling to Multi-Input Functions:** This implementation handles f(x) where $x \in \mathbb{R}$. Extending to $f(x_1, x_2, ..., x_n)$ requires:
   - Distance calculations in n-dimensional space
   - Vectorized gradient computation
   - Efficient memory management for large input dimensions

   I'm uncertain whether pure-Python implementation remains feasible for $n > 3$, or whether C extensions or GPU acceleration becomes necessary.
5. **Connection to Modern RNNs:** RRBF Type 1's recurrence (φ_{i-1}) is conceptually similar to RNN hidden states. However, I'm uncertain about formal equivalences: - Can RRBF Type 1 approximate LSTMs or GRUs? - What's the expressive power compared to standard recurrent architectures? - How does training stability compare (vanishing/exploding gradients)?

These uncertainties motivate future work comparing RRBF networks with contemporary deep learning approaches.

7. Personal Insight

This stage was a turning point in connecting mathematical theory to AI learning practice. By deriving gradients manually, I grasped how loss propagation drives adaptation. It felt like peeling back the black box of neural networks, the RRBF model is essentially a miniature, transparent neural layer.

The process strengthened my intuitions on learning rate, overfitting, and gradient stability, and confirmed that mathematical rigor and computational implementation must co-exist for reliable AI systems.

This assessment introduced a range of unfamiliar mathematical and computational ideas. The table below served as a structured way for me to synthesize, clarify, and retain these concepts throughout my development process:

Concept	Description (My understanding)	Role in AI
RRBF – Recurrent Radial Basis Function Network	A hybrid model that combines RBF kernels with temporal feedback (φ_{i-1}), giving the system short-term memory. I learned how recurrence affects stability and how previous outputs influence the next computation	Forms the conceptual bridge between classic RBF networks and recurrent architectures (RNN/LSTM), used in time-series prediction, sequence modelling, and dynamic systems
Neurons	Each neuron corresponds to a radial basis function centered at m_i. I learned how neurons activate based on distance from the input and	Equivalent to feature detectors in neural networks. More neurons = higher expressive power, enabling universal approximation

	how multiple neurons combine to approximate a function	
Weights (w_i)	Learned how weights modulate each neuron's contribution to the final output. Updating them via gradients changed the shape of the approximated function	Core to all neural models—weights store learned patterns. Training = optimizing weights to reduce error
Learning Rate	Small η slowed learning, large η caused divergence. I had to experiment to find a stable value	Critical hyperparameter in gradient descent. Too high = exploding gradients; too low = slow convergence
Epochs	Repeated passes over the dataset improved accuracy. I learned how iterative refinement moves the model toward the true function	Central in all ML training—more epochs allow deeper convergence (if not overfitting)
Error Minimization	I learned how the cost function measures how far the current output is from the target. Minimizing it requires computing gradients and making incremental parameter updates	This is the foundation of backpropagation and optimization in all neural networks
Function Approximation	Saw how multiple Gaussian activations combine to mimic the target curve. RRBF acts as a nonlinear interpolator capable of fitting smooth functions	Key to neural networks' ability to model any continuous function (universal approximation theorem)
η (Learning Rate Symbol)	Understood η as the scaling factor for every gradient update. It controls the "step size" in parameter space	Appears in every gradient-based learning algorithm; governs training stability
Gradient Descent	Learned how to move parameters opposite to the gradient to reduce error. Implementing it manually clarified how optimization happens	Core optimization technique behind deep learning—SGD, Adam, RMSProp all stem from it

Gaussian Activation	Activation Each neuron outputs $\exp(-\delta\|x - m_i\|^2)$. I learned how δ controls spread and how distance sharply affects activation strength	Basis of kernel methods, SVM with RBF kernels, and probabilistic models
Activation Output (φ_i)	The output of each Gaussian neuron. The recurrent version φ_i = previous + current improved model expressiveness but added instability if mishandled	Mirrors activation flows in deep nets. Understanding this helps in designing neural architectures
Initial Error	First computation before training showed how far the untrained model was from the target. This baseline guided my early tuning	Used to measure model readiness and determine if learning is progressing correctly
Final Error	Observed how repeated updates reduced error toward zero. A small final error indicated successful convergence	Key metric for evaluating training success in all ML workflows
Converge Validation	Validation I learned to stop training when improvement became negligible	$E_i - E_{i-1}$

8. Conclusion

Set 2 Problem 2 brought *EigenAI* closer to simulating a true AI engine. Through manual gradient computation and recurrent feedback, the system learned to adjust parameters and approximate a continuous function without any external help.

Key Takeaways:

1. Manual gradients deepen understanding of learning mechanics.
2. Pure Python teaches discipline in numerical stability.
3. Streamlit maintains an educational bridge between math and AI concepts.
4. Recurrent feedback introduces temporal dynamics within static inputs..

Future Enhancements:

- Compare RRBF Type 1 and Type 2 implementations.
- Experiment with stochastic weight updates (batch sampling)

Ultimately, this project reinforced that learning in AI is nothing more than controlled error correction guided by mathematical precision, and that mastering those fundamentals makes one not just a coder, but an engineer of intelligence.

Statement of Acknowledgment

I acknowledge that I have used the following AI tool(s) in the creation of this report:

- OpenAI ChatGPT (GPT-5): Used to assist with outlining, refining structure, improving clarity of academic language, and supporting APA 7th referencing conventions.

I confirm that the use of the AI tool has been in accordance with the Torrens University Australia Academic Integrity Policy and TUA, Think and MDS's Position Paper on the Use of AI. I confirm that the final output is authored by me and represents my own critical thinking, analysis, and synthesis of sources. I take full responsibility for the final content of this report.

9. References

Dash, R. B., & Dalai, D. K. (2008). *Fundamentals of linear algebra*. ProQuest Ebook Central.

Burden, R. L., & Faires, J. D. *Numerical analysis* (any ed.), sections on quadrature.

Golub, G. H., & Van Loan, C. F. (2013). *Matrix computations* (4th ed.). Johns Hopkins University Press.

Goodfellow, I., Bengio, Y., Courville, A., & Bengio, Y. (2016). *Deep learning* (Vol. 1, No. 2). MIT press.

Lay, D. C., S. R., & McDonald, J.J (2015). *Linear algebra and its applications* (5th ed.). Pearson.

Quarteroni, A., Saleri, F., & Gervasio, P. *Scientific computing with matlab and octave* (for theory; not used as toolbox).

Strang, G. (2016). *Introduction to linear algebra* (5th ed.). Wellesley-Cambridge Press.

Streamlit, Inc. (2025). *Streamlit documentation.* Retrieved from https://docs.streamlit.io/

SymPy Documentation (2024). *Symbolic computation and integration*. https://docs.sympy.org

Nocedal, J., & Wright, S. (2006). *Numerical optimization.* Springer.

Torrens University Australia (2025). *MFA501 Module notes – linear transformations and matrix operations.*

Vakilian, J. (2025). *MFA501 Mathematical foundations of artificial intelligence.* Torrens University Australia.