Introducing Kernel Method in In-Context Learning

Research and Development Project
Prof. Bharat B. N. (EECE Department)
IIT Dharwad

Aiswarya M. S.
EE23BT055

What is In-Context Learning (ICL)?

Definition: In-Context Learning is the ability of a language model to learn a new task **using only examples inside the prompt**, with **no parameter updates**.

Example:

$$\begin{aligned} \text{Good movie} &\rightarrow \text{Positive} \\ \text{Bad book} &\rightarrow \text{Negative} \\ \text{Amazing story} &\rightarrow \text{?} \end{aligned}$$

The model infers the task (sentiment classification) directly from the context.

What Problem Are We Solving?

- Language models (like GPT) learn from examples in the prompt.
- **Problems:**
  - Output changes depending on how examples are formatted.
  - Longer prompts have better performance and but higher computations.
  - Shorter prompts have poor performance and but lower computations.
- **Problem Statement:**
  - How can we improve the model performance while reduccing the computational costs?
- **Solution:**
  - Introducing Iterative Vectors or IVs and Modifying the computation by kernel Method

Sentiment Classification Task

Few-Shot Prompt (In-Context Learning):

Prompt Given to the Model

Review: I love this movie!
Sentiment: Positive

Review: The book was terrible.
Sentiment: Negative

Review: It was okay, not great.
Sentiment:

Output: `Neutral`

Prompt Variation (ICL Problem)

Reordered Few-Shot Prompt:

Reordered Prompt

Review: The book was terrible.
Sentiment: Negative

Review: I love this movie!
Sentiment: Positive

Review: It was okay, not great.
Sentiment:

Model Now Predicts: `Positive`
Same examples, same task — but different output.

Real Evidence of ICL Instability

Order Sensitivity:

- *Webson Pavlick (2022)* and *Min et al. (2022)* show that:
  - Reordering examples can significantly change model outputs.
  - Accuracy drops without clear reason.

Prompt Format Sensitivity:

- Changing "Review:" to "Comment:" or "Sentiment:" to "Label:" affects predictions.

IVs Paper (Liu et al. 2025):

- Points out that ICL results are fragile due to:
  - Prompt structure
  - Demonstration ordering
  - Token formatting

Key Insight — Attention = Gradient Descent

First the model processes:

- Tokenization
- Embedding layer
- Transformer layers:
  - Self-Attention (token interactions),Feedforward Network, Residual + LayerNorm

Attention(Attn) reads all examples before the query.It extracts the pattern relating input → label.

$$W = W_0 + \sum_{t=1}^{T} e_t \otimes x_t$$

is mathematically identical to:

$$\mathrm{Attn}(X, E, q) = \sum_{t} (e_t \otimes x_t)\, q$$

Keys (x_t) behave like training inputs, Values (e_t) behave like gradients
Query (q) receives the update

Why This Creates ICL

Because attention behaves like gradient descent:

- Demonstrations in the prompt act like **training examples**.
- Attention layers generate **meta-gradients**.
- These meta-gradients shift the hidden representations of the query.

$$F_{\mathrm{ICL}}(q) = (W_{\mathrm{ZSL}} + \Delta W_{\mathrm{ICL}})\, q$$

Where:

- q = representation of the query token.
- W_{ZSL} = zero-shot transformation (model's original behavior without examples).
- ΔW_{ICL} = meta-gradient update induced by the in-context demonstrations.
- $F_{\mathrm{ICL}}(q)$ = final output representation after applying the in-context update.

Extracting Meta-Gradients (ΔAct)

We run two forward passes:

- **Few-shot** (with demonstrations)
- **Zero-shot** (without demonstrations)
- $\mathbf{Act}_l^{\text{few-shot}}(x_i)$**Activation function of few shot in layer l**
- $\mathbf{Act}_l^{\text{few-shot}}(x_i)$**Activation function of zero shot in layer l**
- **) Activation function can be GELU,RELU ,sigmoid,softmax**

Then compute:

$$\Delta Act_l(x_i) = Act_l^{\text{few-shot}}(x_i) - Act_l^{\text{zero-shot}}(x_q)$$

- Removes the baseline behavior.
- Isolates the ICL-specific update direction.
- Represents the **meta-gradient** at layer l.

Two types of Averaging

Averaging into Class Vectors

For each class j:

$$v_{j,l} = \frac{1}{|C_j|} \sum_{i \in C_j} \Delta Act_l(x_i)$$

- Produces a stable class-specific meta-gradient.
- Reduces noise from individual samples.
- One vector per class per layer.

Episode-Averaged Task Vectors

Across many episodes:

$$V_l = \frac{1}{m} \sum_{i=1}^{m} V_l^i$$

- Aggregates meta-gradients across episodes.
- Produces a stable task-level update direction.
- Much more reliable than a single episode.

Iterative Vectors (IVs) = Multi-Step Refinement

During extraction of batch i:

$$Act \leftarrow Act + \alpha_1 \cdot \bar{V}_{i-1}$$

Where:

- Act = activation at the input–output separator token during extraction.
- α_1 = extraction strength (how strongly previous vectors are applied).
- $\bar{V}_{i-1}$ = cumulative average of IVs from all previous batches:

$$\bar{V}_{i-1} = \frac{1}{i-1} \sum_{j=1}^{i-1} V_l^j$$

- V_l^i = vectors extracted from batch i at layer l.

Final refined vectors:

$$V_l = \frac{1}{m} \sum_{i=1}^{m} V_l^i$$

Where: V_l = final Iterative Vector at layer l, m = total number of extraction batches.

Final Takeaway

(1) Demonstrations in prompt

$\downarrow$

(2) Attention processes them $\rightarrow$ gradient-like updates

$\downarrow$

(3) These updates change hidden activations (ICL)

$\downarrow$

(4) ΔAct = (few-shot − zero-shot) captures this update

$\downarrow$

(5) Averaging ΔAct $\rightarrow$ class-specific meta-gradients

$\downarrow$

(6) Averaging across episodes $\rightarrow$ stable task meta-gradients

$\downarrow$

(7) Iterative refinement simulates gradient descent steps $\rightarrow$ IVs

Model and Verbalizer

Model

- We have used pretrained language model:

$$\text{GPT-J -6B}$$

- **No gradient updates** anywhere.
- Only **forward pass** activations are used.

Verbalizer

- Converts class labels into natural language tokens.
- Example:

$$\text{Positive} \rightarrow \text{“positive”}, \quad \text{Negative} \rightarrow \text{“negative”}$$

- Used to compute probabilities via answer-token logits.

Extraction Process

- Run model on:

$$\text{Few-shot prompt}, \quad \text{Zero-shot prompt}$$

- Collect relevant activations at separator tokens.
- Compute deltas:

$$\Delta\text{Act} = \text{Act}_{FS} - \text{Act}_{ZS}$$

- Compute class-wise averages:

$$v_{\text{pos}}, \quad v_{\text{neg}}, \quad v_q$$

Part 1: What is an Episode?

A 2-shot (2 examples per class), 2-way (Positive/Negative) classification task.
Episode prompt E:

$$\text{great movie} \to \text{Positive}$$

$$\text{terrible book} \to \text{Negative}$$

$$\text{hated it} \to \text{Negative}$$

$$\text{amazing film} \to \text{Positive}$$

$$\text{loved that movie} \to ?$$

Part 1: contd...

Contains:

Positive:	great movie, amazing film
Negative:	terrible book, hated it
Query:	loved that movie

Zero-shot version E_0 only contains the query:

loved that movie

Part 2: Collecting Activations

We run the model on:

1) E(Few-shot prompt) 2) E_0(Zero-shot prompt)

For each run, we collect attention activation at the separator token.
Stored activations: Act_l(Activation function of Few-shot examples)
Act_l^0(Activation function of zero shot examples)

$$\begin{aligned} \text{Act}_l(\text{great movie}) \\ \text{Act}_l(\text{terrible book}) \\ \text{Act}_l(\text{hated it}) \\ \text{Act}_l(\text{amazing film}) \\ \text{Act}_l(\text{loved that movie}) \\ \text{Act}_l^0(\text{loved that movie}) \end{aligned}$$

These are high-dimensional vectors (e.g., 768, 1024).

Part 3: Computing ΔAct

For every example x (including query):

$$\Delta \mathrm{Act}_l(x) = \mathrm{Act}_l(x) - \mathrm{Act}_l^0(q)$$

This isolates the change caused by demonstrations.
For the episode, 5 vectors:

$$\begin{aligned} &\Delta\mathrm{Act}(\text{great movie}) \\ &\Delta\mathrm{Act}(\text{amazing film}) \\ &\Delta\mathrm{Act}(\text{terrible book}) \\ &\Delta\mathrm{Act}(\text{hated it}) \\ &\Delta\mathrm{Act}(\text{loved that movie}) \end{aligned}$$

Each represents the demo-induced shift.

Part 4: Averaging by Class

Positive class (2 examples):

$$v^l_{\text{positive}} = \frac{1}{2}\Big(\Delta\text{Act}(\text{great movie}) + \Delta\text{Act}(\text{amazing film})\Big)$$

Negative class:

$$v^l_{\text{negative}} = \frac{1}{2}\Big(\Delta\text{Act}(\text{terrible book}) + \Delta\text{Act}(\text{hated it})\Big)$$

Query vector:

$$v^l_q = \Delta\text{Act}(\text{loved that movie})$$

Episode vector:

$$V^l_E = \{v^l_{\text{positive}},\ v^l_{\text{negative}},\ v^l_q\}$$

Iterative Refinement Across Batches

Task setup:

2-way: Positive / Negative, 2-shot per class, 1 query

10 episodes total, Batch size = 2

Batches:

$$
\begin{aligned}
B_1 &= (E_1, E_2) \\
B_2 &= (E_3, E_4) \\
B_3 &= (E_5, E_6) \\
B_4 &= (E_7, E_8) \\
B_5 &= (E_9, E_{10})
\end{aligned}
$$

Each episode provides:

$$V_E^l = \{v_{\text{pos}}^l,\ v_{\text{neg}}^l,\ v_q^l\}$$

These vectors are derived from class-wise averages of ΔAct.

Batch 1 (Episodes 1 and 2)

Step 1: Extract V_1^l

$$V_1^l = \frac{1}{2}\left(V_{E_1}^l + V_{E_2}^l\right)$$

No activation editing in Batch 1.

Step 2: Running average

$$\bar{V}_1^l = V_1^l$$

This is the first rough task vector.

Batch 2 (Episodes 3 and 4)

Activation editing before extraction:

$$\text{Act}_{\text{edited}} = \text{Act} + \alpha_1 \cdot \bar{V}_1^l$$

Step 1: Extract V_2^l

$$V_2^l = \frac{1}{2}\left(V_{E_3}^l + V_{E_4}^l\right)$$

Step 2: Running average

$$\bar{V}_2^l = \frac{1}{2}\left(V_1^l + V_2^l\right)$$

Now Batch 1 and Batch 2 contribute to the representation.

Batch 3 (Episodes 5 and 6)

Activation editing:

$$\text{Act}_{edited} = \text{Act} + \alpha_1 \cdot \bar{V}_2^l$$

Step 1: Extract V_3^l

$$V_3^l = \frac{1}{2}\left(V_{E_5}^l + V_{E_6}^l\right)$$

Step 2: Running average

$$\bar{V}_3^l = \frac{1}{3}\left(V_1^l + V_2^l + V_3^l\right)$$

Now 3 batches contribute to meta-learning.

Batch 4 (Episodes 7 and 8)

Activation editing:

$$\text{Act}_{edited} = \text{Act} + \alpha_1 \cdot \bar{V}_3^l$$

Step 1: Extract V_4^l

$$V_4^l = \frac{1}{2}(V_{E_7}^l + V_{E_8}^l)$$

Step 2: Running average

$$\bar{V}_4^l = \frac{1}{4}\left(V_1^l + V_2^l + V_3^l + V_4^l\right)$$

The representation becomes more refined.

Batch 5 (Episodes 9 and 10)

Activation editing:

$$\mathrm{Act}_{edited} = \mathrm{Act} + \alpha_1 \cdot \bar{V}_4^l$$

Step 1: Extract V_5^l

$$V_5^l = \frac{1}{2}\left(V_{E_9}^l + V_{E_{10}}^l\right)$$

Step 2: Running average

$$\bar{V}_5^l = \frac{1}{5}\left(V_1^l + V_2^l + V_3^l + V_4^l + V_5^l\right)$$

$\bar{V}_5^l$ is the final refined representation.

Final Iterative Vector

Final IV for layer l:

$$V_{\mathrm{IV}}^{l} = \bar{V}_{5}^{l}$$

Usage during inference:

$$\mathrm{Attn}_{l}(x) \leftarrow \mathrm{Attn}_{l}(x) + \alpha_2 \cdot V_{\mathrm{IV}}^{l}$$

Summary

- Each episode $\rightarrow \Delta\text{Act} \rightarrow (v_{\text{pos}}, v_{\text{neg}}, v_q)$.
- Batch 1 $\rightarrow V^1$.
- Batch 2 $\rightarrow V^2$ with activation editing.
- Batch 3 $\rightarrow V^3$ with editing.
- Batch 4 $\rightarrow V^4$ with editing.
- Batch 5 $\rightarrow V^5$ with editing.
- Final IV:

$$V_{\text{IV}} = \frac{1}{5}(V^1 + V^2 + V^3 + V^4 + V^5)$$

Final Summary

1. Run few-shot & zero-shot $\rightarrow$ collect activations.
2. Subtract $\rightarrow$ ΔAct vectors.
3. Average by class $\rightarrow$ v_{pos}, v_{neg}, v_q.
4. Each episode gives V_E^l.
5. Process episodes in batches $\rightarrow$ V_i^l.
6. Iterative refinement using running averages.
7. Final IV:

$$V_{\mathrm{IV}}^l = \frac{1}{5}\sum_{i=1}^{5} V_i^l$$

Result: A small set of vectors encoding the entire task.

Introducing Batch level Kernel-Based Averaging

Without kernel (paper default):

$$V_i^l = \frac{1}{|B_i|} \sum_{E \in B_i} V_E^l$$

To improve the performance ,My code replaces this with a kernel-weighted average:

$\mathrm{ivs}[l] = \mathrm{attention\_aggregate}(\mathrm{iv\_list},\, \sigma = 0.2)$(function used in improved code)

instead of:

$$\mathrm{ivs}[l] = \mathrm{mean}(\mathrm{iv\_list})$$

Thus, the aggregation becomes:

$$V_i^l = \mathrm{KernelWeightedAverage}(V_E^l)$$

What is `iv_list`?

Inside the function `attention_aggregate`, the input has shape:

$$\text{iv\_list} \in \mathbb{R}^{(N_{\text{episodes}}) \times (C+1) \times d}$$

For a 2-way task (Positive/Negative):

$$\text{iv\_list}[i][c] = \begin{cases} v_{\text{pos}}^{(i)} \\ v_{\text{neg}}^{(i)} \\ v_{q}^{(i)} \end{cases}$$

Therefore, it aggregates:

ALL episodes within a batch

for each of the class/query vectors.

What the Kernel Function Does

The function:

$$\text{attention\_aggregate}(\text{iv\_list}, \sigma)$$

performs:

1. computes a reference vector (mean)
2. normalizes embeddings
3. computes cosine similarities
4. turns similarity into Gaussian RBF weights
5. normalizes these weights
6. returns a weighted average

Instead of:

$$\text{simple mean}$$

it computes:

$$\text{RBF-weighted mean}$$

Step 1: Reference Vector and Step 2: L2 Normalization

Reference Vector

$$\text{ref} = \text{mean}(\text{iv\_list},\ \text{dim} = 0)$$

This yields one "central" vector per class:

$$\text{ref}[c] = \frac{1}{N}\sum_{i=1}^{N} \text{iv\_list}[i][c]$$

This is the anchor for similarity measurement.

L2 Normalization

$$\text{iv\_norm} = \frac{\text{iv\_list}}{\|\text{iv\_list}\|} \qquad \text{ref\_norm} = \frac{\text{ref}}{\|\text{ref}\|}$$

Purpose:

$$\text{Cosine similarity becomes dot product}$$

Step 3: Cosine Similarity Computation

Per episode and per class:

$$\text{sim}_{ic}(\text{Cosine similarity}) = \text{iv\_norm}_{ic} \cdot \text{ref\_norm}_{c}$$

Shape:

$$\text{sim} \in \mathbb{R}^{N_{\text{episodes}} \times C}$$

Interpretation:

- sim $\approx 1 \rightarrow$ very similar episode
- sim $\approx 0 \rightarrow$ unrelated
- sim $< 0 \rightarrow$ contradictory or noisy episode

Step 4: Gaussian RBF Weight

Weights computed as:

$$w_{ic} = \exp\left(-\frac{1 - \mathrm{sim}_{ic}}{2\sigma^2}\right)$$

If sim ≈ 1 (good episode):

$$w_{ic} \approx 1$$

If low similarity (noisy episode):

$$w_{ic} \to 0$$

Thus, the kernel:

- boosts good episodes
- suppresses bad/noisy episodes

Step 5: Normalize Weights

Per class c:

$$w_{ic} \leftarrow \frac{w_{ic}}{\sum_j w_{jc}}$$

This ensures:

$$\sum_i w_{ic} = 1$$

for each class c.

Step 6: Weighted Aggregation (Final IV)

Final aggregated vector:

$$V^l_{\text{agg}}[c] = \sum_{i=1}^{N} w_{ic} \cdot \text{iv\_list}[i][c]$$

Or using tensor notation:

$$V^l_{\text{agg}} = \text{einsum}\ \left(''ic,\ icd \to cd'',\ w,\ \text{iv\_list}\right)$$

This replaces:

$$V^l_i = \text{mean}(V^l_E)$$

with a smarter, RBF-weighted version.

Naive Averaging vs Weighted Averaging

Naive averaging:

$$V_i{}^l = \frac{1}{N}\left(V_{E_1}^l + V_{E_2}^l + \cdots + V_{E_N}^l\right)$$

method Introduced (weighted averaging):

$$V_i{}^l = w_1\, V_{E_1}^l + w_2\, V_{E_2}^l + \cdots + w_N\, V_{E_N}^l$$

Final Intuition and Summary

Simple averaging (old):

$$V = \frac{1}{N} \sum_i V_i$$

treats all episodes equally.

Kernel averaging (in code):

$$V = \sum_i w_i \, V_i, \qquad w_i \text{ from cosine + RBF}$$

- **Good episodes** get high weight
- **Noisy/poor episodes** get low weight
- Produces **smooth, stable, robust** IVs
- Mimics **attention mechanism**

Super Simple Summary:

$$\text{KernelAggregate} = \text{WeightedAverage}\,(\text{weights from cosine similarity})$$

Example Setup + Original IV Issue

Toy 2D vectors (Positive class):

$$v_1 = [1, 1], \quad v_2 = [2, 2], \quad v_3 = [8, -2] \text{ (noisy)}$$

Simple mean (Original IV):

$$V_{\text{mean}} = \frac{1}{3}(v_1 + v_2 + v_3) = [3.67,\ 0.33]$$

Problem: Outlier v_3 strongly biases the mean.

Kernel Step 1–2: Reference + Cosine

Reference vector (same mean):

$$R = [3.67,\ 0.33]$$

Cosine similarity:

$$\mathrm{sim}_1 = \mathrm{sim}_2 \approx 0.71, \qquad \mathrm{sim}_3 \approx 0.28$$

Interpretation:

- $v_1, v_2 \rightarrow$ aligned with reference
- $v_3 \rightarrow$ wrong direction (noise)

Kernel Step 3–4: RBF Weights + Batch 1 IV

RBF weights:

$$w_1 = w_2 \approx 0.49, \qquad w_3 \approx 0.002$$

Kernel batch vector:

$$V^1_{\text{kernel}} \approx [1.49,\ 1.47]$$

Outlier suppressed → clean class signal.

Batch 2 + First Refinement

$$v_4 = [3,3], \qquad v_5 = [4,4], \qquad v_6 = [10,-5] \text{ (noisy)}$$

Batch 2 kernel result:

$$V^2 \approx [4.95,\ 4.91]$$

Refinement after 2 batches:

$$\bar{V}^2 = \frac{1}{2}(V^1 + V^2) = [3.22,\ 3.19]$$

This becomes the new editing vector for Batch 3.

$$E_7 : v_7 = [2,\ -1]$$
$$E_8 : v_8 = [3,\ -2]$$

$$E_9 : v_9 = [20,\ -10] \quad \text{(outlier)}$$
$$E_{10} : v_{10} = [-20,\ 10] \quad \text{(opposing outlier)}$$

Batch 3 kernel result:

$$V^3 \approx [6.0,\ 5.8]$$

Refined across first 3 batches:

$$\bar{V}^3 = [4.15,\ 4.06]$$

Final Batches + Summary

$$E_9(Batch-4) : v_9 = [20,\ -10] \quad \text{(outlier)}$$
$$E_{10}(Batch-4) : v_{10} = [-20,\ 10] \quad \text{(opposing outlier)}$$

$$V^4 = [7.1, 7.0], \qquad V^5 = [6.8, 6.6]$$

Final Kernel-IV vector:

$$\bar{V}^5 = \frac{1}{5}(V^1 + V^2 + V^3 + V^4 + V^5) = [5.27,\ 5.16]$$

Batch	Kernel IV
1	[1.49,1.47]
2	[4.95,4.91]
3	[6.00,5.80]
4	[7.10,7.00]
5	[6.80,6.60]
Final	$\boxed{[5.27,\ 5.16]}$

Why Kernel-Based Averaging is Better Than Naive Averaging

1. Naive Averaging Treats All Episodes Equally

$$V^i_{\text{mean}} = \frac{1}{|B_i|} \sum_{E \in B_i} V^E$$

- noisy or contradictory episodes distort the mean
- no mechanism to prefer high-quality episodes
- direction of V^E is lost when outliers dominate

2. Kernel Method Uses Similarity-Based Weighting

$$V^i_c = \sum_E w_{E,c}\, V^E_c \qquad w_{E,c} = \frac{\exp\left(-\frac{1-\cos(V^E_c, R_c)}{2\sigma^2}\right)}{\sum_{E'} \exp\left(-\frac{1-\cos(V^{E'}_c, R_c)}{2\sigma^2}\right)}$$

- episodes aligned with reference R_c get higher weight
- noisy/outlier episodes get very low weight
- acts as "attention over episodes"

Naive Averaging vs Kernel Method (Results + Explanation)

Performance Comparison

Metric	Naive Avg	Kernel Method	Gain
Micro F1	63.79	64.26	+0.47
Macro F1	59.97	61.04	+1.07
Weighted F1	59.84	60.96	+1.12

Metrics Explanation

- **Micro F1:** Global F1 over all samples (majority-class heavy).
- **Macro F1:** F1 averaged per class (treats all classes equally).
- **Weighted F1:** F1 per class weighted by class size.

Why Kernel is Better

- Naive averaging treats all episodes equally $\rightarrow$ noisy episodes distort V^i.
- Kernel method uses similarity-based weights (cosine + RBF).
- Good episodes get high weight noisy episodes get very low weight.

Problem Statement & How Kernel Methods Help

Goal: Improve model performance in In-Context Learning while reducing computational cost.

Challenges:

- Long few-shot prompts $\rightarrow$ expensive attention ($O(n^2)$).
- Iterative Vector extraction requires multiple forward passes.
- Memory + latency increase with number of examples.

Question: How do we obtain the performance benefits of ICL **without** the heavy computational overhead?

How Kernel Methods Help

Kernel methods offer a low-cost approximation of transformer behavior.

- Approximate attention using kernel feature maps (reduces $O(n^2)$ $\rightarrow$ $O(n)$).
- Compute ICL meta-gradients without full forward passes.
- Kernel regression matches ICL behavior in hidden space.

Future Improvements

- **Adaptive Kernel Bandwidth** Instead of a fixed σ, learn or tune σ dynamically per batch or per class.
- **Layer-Wise Weight Learning** Assign different strengths to IVs at different layers, instead of uniform α.
- **Magnitude-Aware Weighting** Combine cosine similarity with vector magnitude to suppress large-magnitude outliers.
- **Better Episode Sampling** Use curriculum-based or difficulty-aware sampling instead of random episode selection.
- **Cross-Batch Attention** Replace running average with attention over all previous batches for richer refinement.
- **Task-Adaptive Strength (α)** Learn or tune α per task rather than using fixed values.
- **Robustness to Noisy Demonstrations** Explore more robust kernels (e.g., Laplacian, Cauchy) for episode weighting.

References

- Liu, Z., Li, X., Zhou, K., Leike, J. (2025). *Iterative Vectors: In-Context Gradient Steering without Backpropagation*. arXiv:2501.XXXX.
- Webson, A. Pavlick, E. (2022). *Do Prompt-Based Models Really Understand the Task?* arXiv:2109.01247.
- Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer. (Gaussian RBF Kernel Definition)
- ArkciaTheDragon. (2024). *Iterative Vectors GitHub Repository*. `https://github.com/ArkciaTheDragon/iterative-vectors`
- Vaswani, A. et al. (2017). *Attention is All You Need*. NeurIPS.