26-Aug-2025

Deep Optimal Isolation Forest with Genetic Algorithm for Anomaly Detection

DOIForest Implementation:

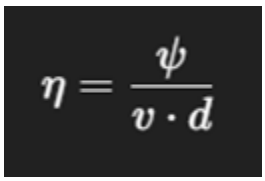
1. Start with a normal isolation forest (iForest/LSHiForest).
2. Apply **genetic algorithm** with two mutations:
    - Outer mutation (new sampling → bigger search space).
    - Inner mutation (local structural tweaks).
3. Use **isolation efficiency (η)** to select the best trees.
4. Repeat layer by layer → evolve toward optimal forest.
5. Result: A **deep optimal isolation forest** that is more accurate and robust for anomaly detection.

Steps of DOIForest Implementation

Step 1: Pre-training (Initial Forest Creation)

- Start with a normal **Isolation Forest** (or LSHiForest).
- Build a set of trees using random subsets of the data.
- This acts as the **initial population** of trees for further optimization.
- Intuition: you need a baseline forest before you can improve it.

Step 2: Define Fitness (Isolation Efficiency)

- For each tree, calculate its **isolation efficiency (η)**:
- 
  where:

    - ψ = number of data points in the tree,
    - v = branching factor,
    - d = average depth.
- This value tells **how well a tree isolates anomalies**.
- Think of η as the “score” of each tree: **higher = better**.

Step 3: Apply Genetic Algorithm

- To improve the forest, use two mutation strategies inspired by **genetic evolution**:

1. **Outer Mutation**
    - Change the training sample → rebuild a tree with a new subset of data.
    - Purpose: Explore **new structures** and avoid getting stuck in local optima.
2. **Inner Mutation**
    - Modify the **internal structure** of an existing tree (like tweaking a subtree split).
    - Purpose: Fine-tune trees by making small adjustments.

- After mutations, evaluate all trees again using **isolation efficiency**.

Step 4: Selection

- From the mutated trees, keep only those with the **highest η**.
- This ensures that the forest gradually **evolves toward more optimal trees**.
- Just like “survival of the fittest” in biology.

Step 5: Layer-by-Layer Evolution

- Repeat **mutation + selection** for several layers (like training epochs).
- Each layer = one “generation” of trees.
- After many layers, the forest becomes a **deep, optimized isolation forest**.

Step 6: Testing/Scoring

- Once training is done, use the optimized forest to score new data points.
- The score is based on the **path length**:
    - Anomalies → short paths (easy to isolate).
    - Normal points → long paths.
- Higher anomaly score = more suspicious data point.

Optimized Deep Isolation Forest

Context:
Anomaly Detection (AD) is vital for tasks like cybersecurity, fraud detection, healthcare monitoring, and industrial fault detection.
Traditional Isolation Forest (IF) is efficient but limited to axis-parallel splits.
Deep Isolation Forest (DIF) improves IF using neural network–based data transformations but is computationally heavy.

Solutions proposed in paper:
Optimized Deep Isolation Forest (ODIF) → an improved version of DIF.
Optimization Idea:
Instead of transforming all training samples, ODIF:

- Samples elements *before* transformation.
- Only transforms those used for building trees.
- Removes subsampling during tree construction.

This leads to **lower computational and memory complexity** while preserving accuracy.

Solution details:

DIF Mechanism:

- Uses a random-weight deep neural network (**CERE**) to generate multiple data representations.Builds isolation forests for each representation.
- Builds isolation forests for each representation.
- Uses **Deviation-Enhanced Anomaly Scoring (DEAS)** combining path length + deviation degree.

ODIF Mechanism:

- Reduces transformation to only `t × n` samples (trees × samples per tree).
- Complexity reduced to:
  Time: O(tnar+rtlogn)O(tna r + rt \log n)O(tnar+rtlogn)
  Memory: O(tnrb+rtlogn)O(tnr b + rt \log n)O(tnrb+rtlogn)

Results:

- **Detection Performance (PR AUC):**
  ODIF ≈ DIF (no statistical difference).
  ODIF > IF, ECOD, and SGAE (statistically significant).

- **Training Time:**
  CPU: ODIF ~1.5× faster than DIF.
  GPU: ODIF ~150× faster than DIF.
  Much faster than SGAE; competitive with IF/ECOD.
- **Memory Usage:**
  RAM: ~18% lower than DIF.
  VRAM: ~55% lower than DIF.
  Stable across dataset sizes → suitable for resource-constrained devices.

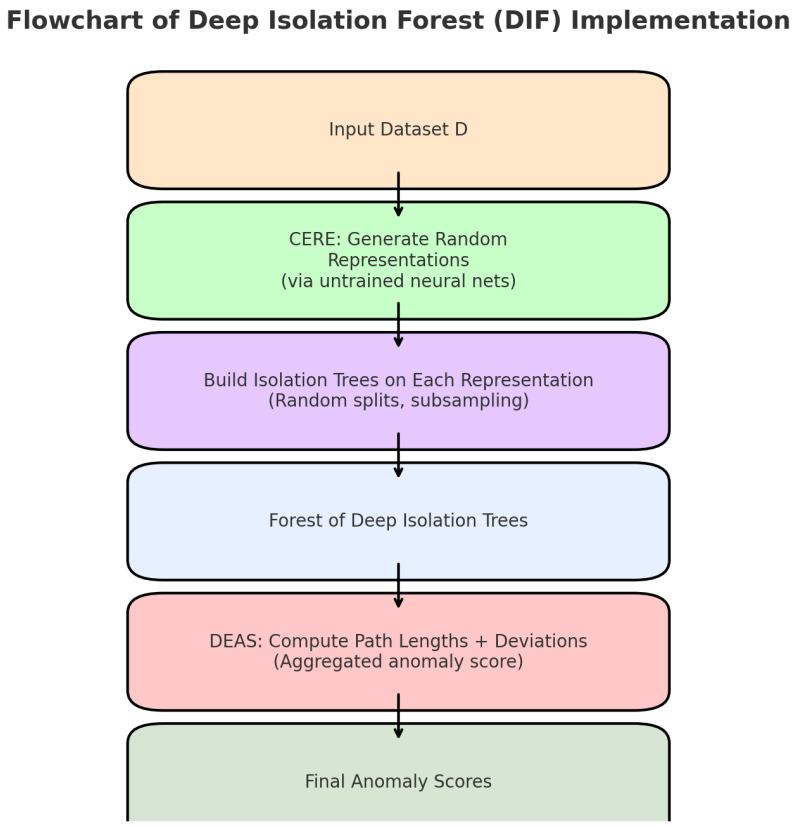
Conclusions:

- **ODIF matches DIF in accuracy** but with **much lower execution time and memory usage.**
- Outperforms other baselines (IF, ECOD, SGAE).
- More practical for large-scale or limited-resource anomaly detection applications.

Future add ons:

- Instead of random weights, we can use trainable representation networks.
- Extending ODIF idea to other anomaly detection algorithms.

Deep Isolation Forest for Anomaly Detection



Perfect — here's a crisp, copy-pasteable **step-by-step explanation of the DIF flowchart**, aligned to the paper.

Phase 1: Training (build the forest)

Step 1 — Input data.
Start with the training set D\mathcal D.

Step 2 — Generate random representations with CERE.
Pass D\mathcal D through the CERE module to produce rr different representation spaces {X1,…,Xr}\{\mathcal X_1,\ldots,\mathcal X_r\}. CERE vectorises the forward pass so all ensemble members are computed in one

batched operation (no training/optimisation), enabling efficient parallel generation of representations.

Step 3 — Build isolation trees on each representation.
For every representation Xu\mathcal X_u, build tt iTrees exactly like iForest: (a) take a random subsample P1P_1 (size nn) as the root; (b) while a node can split and the depth <J< J, choose a random feature jkj_k and a random split ηk\eta_k between the min/max of that feature in the node; (c) send points left/right by testing x(jk)≤ηkx^{(j_k)} \le \eta_k; (d) recurse until a single point is isolated or depth limit is hit. The forest then contains T=r×tT=r\times t trees.

Typical paper settings (for reference): r=50r=50, t=6t=6, n=256n=256.

Phase 2: Scoring (anomaly detection)

Step 4 — New point arrives.
Given a new object o\mathbf o, compute its rr representations {x1,…,xr}\{\mathbf x_1,\ldots,\mathbf x_r\} using the *same* CERE parameters as in training.

Step 5 — Traverse all trees & record path statistics.
For each representation xu\mathbf x_u and each of its tt trees τi\tau_i:

- Record **path length** ∣p(xu∣τi)∣|p(\mathbf x_u\!\mid\!\tau_i)| (number of splits from root to the terminal node).

- Accumulate **deviation** along the path: β+=∣xu(jk)−ηk∣\beta \mathrel{+}= \big|x_u^{(j_k)} - \eta_k\big| at every visited node kk.

- Convert to **average deviation** for that path:
  g(xu∣τi)=1∣p(xu∣τi)∣∑k∈p(xu∣τi)∣xu(jk)−ηk∣g(\mathbf x_u\!\mid\!\tau_i)=\frac{1}{|p(\mathbf x_u\!\mid\!\tau_i)|}\sum_{k\in p(\mathbf x_u\!\mid\!\tau_i)}\big|x_u^{(j_k)}-\eta_k\big|
  (captures local density/hardness of isolation).

Step 6 — Average over the forest.
Aggregate **mean path length** and **mean deviation** across all TT trees for the point. (This is implemented directly in the paper's Algorithm 2.)

Step 7 — Compute the DEAS score.
Combine both signals to get the final anomaly score:

FDEAS(o∣T)=2− E[∣p∣]/C(T)×E[g(xu∣τi)]F_{\text{DEAS}}(\mathbf o\!\mid\!\mathcal T) = 2^{-\;\mathbb E[\,|p|\,]/C(T)} \times \mathbb E\big[\,g(\mathbf x_u\!\mid\!\tau_i)\,\big]

Here C(T)C(T) is the iForest normalisation constant; the first term is the classic path-length score, and the second term amplifies anomalies by their average deviation to split thresholds. Higher is more anomalous.

Output.
Return a single scalar score per point; rank or threshold as needed.

Why this works (one-liner)

Random (untrained) neural-net representations + random partitioning let simple axis-parallel cuts act like non-linear separators in the original space, reducing false negatives and removing "ghost regions."

Want this formatted as **pseudocode** for your report (Algorithm block with inputs/outputs and variables r,t,n,Jr,t,n,J)?