Adversarial Robustness Analysis of scRNA-seq Cell-Type Classifiers Using Advesarial Learning & Meta-Detection Models

Department of Computer Science and Engineering
National Institute of Technology, Raipur, C.G, INDIA

Minor Project Report

Submission by:

Aditya Ojha 22115005
Gautam Kumar 22115028

Guidance of:

Dr. Deepak Singh

DECLARATION

We hereby declare that the work described in this report, entitled **"Adversarial Robustness Analysis of scRNA-seq Cell-Type Classifiers Using Deep Learning and Meta-Detection Models"**, which is being submitted by us in partial fulfilment of the award of the degree of **Bachelor of Technology in Computer Science & Engineering** to the **Department of Computer Science and Engineering, National Institute of Technology Raipur**, is the result of investigations carried out by us under the guidance of **Dr. Deepak Singh Sir**.

We further declare that the work presented in this report is **original** and has **not been submitted** for the award of any Degree/Diploma of this or any other Institute/University.

Signatures

Name : Aditya Ojha
Roll No. : 22115005

Name : Gautam Kumar
Roll No. : 22115028

Date: 28-11-2025_____

INDEX

1. Abstract

Single-cell RNA sequencing (scRNA-seq) plays a pivotal role in modern biology by enabling transcriptomic profiling at single-cell resolution, thereby driving major advances in cell-type discovery, tumor microenvironment analysis, and immune system characterization. Machine learning (ML) models—particularly fully connected neural networks—have become increasingly popular for automated cell-type classification due to their speed, scalability, and ability to learn complex nonlinear gene-expression patterns. However, recent findings, notably from the *"Gene expression adverSCarial"* study, suggest that scRNA-seq classifiers are highly susceptible to **adversarial perturbations**: imperceptibly small, targeted modifications to gene-expression vectors that drastically alter the classifier's output while preserving biological interpretability. Such vulnerabilities pose serious risks for downstream research and clinical pipelines, where erroneous cell-type predictions may compromise biomarker identification, disease-state inference, and therapeutic recommendations.

The present work develops a comprehensive adversarial analysis and detection framework tailored for scRNA-seq classification tasks, using the PBMC3k dataset as a representative benchmark. Our pipeline integrates four core components:

(1) **A deep neural classifier**, trained on 2000 highly variable genes and producing 64-dimensional latent embeddings for 10 pseudo-cell clusters generated via KMeans. The classifier achieves strong clean-data generalization (~85.8% accuracy) with early stopping to prevent overfitting.

(2) **A Projected Gradient Descent (PGD) adversarial attack module**, which iteratively perturbs gene-expression profiles within a constrained ε-ball. These perturbations maintain biological plausibility while significantly degrading classifier performance, leading to an adversarial **flip rate of 99.2%**, indicating near-total vulnerability—consistent with the weaknesses previously highlighted in scRNA-seq neural architectures.

(3) **A Triplet Network–based embedding transformation module**, trained to construct a discriminative meta-feature space that emphasizes subtle distortions introduced by adversarial perturbations. This model converts the 64-dimensional classifier embeddings into 32-dimensional meta-features structured such that clean and adversarial representations become maximally separable.

(4) **A meta-detection model**, trained as a binary classifier on the triplet-generated embeddings, capable of predicting whether a given expression profile (clean or perturbed) has undergone adversarial manipulation. The meta-detector

demonstrates **89.5% accuracy in identifying adversarial samples**, outperforming simple thresholding approaches and providing a practical defense mechanism otherwise absent in existing scRNA-seq adversarial studies.

Empirical results confirm that adversarial perturbations can mislead deep scRNA-seq classifiers with minimal gene-level modifications—often too subtle to influence UMAP embeddings or differential expression signatures—supporting claims from the *adverSCarial* paper. Importantly, our proposed meta-detection layer restores robustness by accurately discriminating between benign and adversarial inputs, making it a viable defensive complement to existing classifier architectures. The project thus contributes both an analytical framework for evaluating adversarial vulnerabilities and a novel meta-learning–based detection strategy specifically designed for gene-expression data.

Overall, this research highlights the pressing need to incorporate adversarial robustness assessments into computational biology workflows. As ML tools become increasingly central to genomics and precision medicine, ensuring their reliability under adversarial conditions is critical. The proposed system provides a reproducible, end-to-end adversarial evaluation and mitigation pipeline for scRNA-seq classifiers, offering a foundation for future advancements in robust, trustworthy single-cell analysis.

2. Introduction

2.1 Background

Single-cell RNA sequencing (scRNA-seq) has revolutionized modern biological research by enabling the quantification of gene expression at single-cell resolution. This capability has led to profound insights in developmental biology, immunology, oncology, and regenerative medicine. Unlike bulk RNA-seq—where gene expression signals are averaged across millions of cells—scRNA-seq preserves cell-level heterogeneity, allowing researchers to uncover rare cell populations, transient cellular states, lineage commitments, and complex tissue architectures.

As scRNA-seq technologies evolved, so did the volume, dimensionality, and complexity of the resulting datasets. A typical scRNA-seq experiment may involve thousands to millions of cells, each measured across tens of thousands of genes. Such high-dimensional data poses significant challenges for classical statistical models, paving the way for the adoption of machine learning (ML) and deep learning (DL) methods. These algorithms, particularly multilayer perceptrons (MLPs), convolutional architectures, and autoencoders, have become indispensable in tasks such as:

- **Cell-type classification**
- **Clustering and annotation transfer**
- **Batch effect correction**
- **Feature extraction and dimensionality reduction**
- **Gene regulatory network inference**

ML models excel in capturing nonlinear relationships in gene expression data and are now integral components of scRNA-seq analysis pipelines used in research and industry. Automated annotation tools like scDeepSort, scClassify, CHETAH, and scType rely heavily on ML models to assign cell identities based on learned gene expression signatures.

However, the increasing reliance on ML introduces new concerns related to **model robustness**, **generalizability**, and **vulnerability to adversarial perturbations**—a phenomenon first observed in computer vision and later in natural language processing. In these domains, adversarial attacks are small but intentionally crafted manipulations of input data that lead ML models to produce incorrect outputs with high confidence. These perturbations are often imperceptible to humans, yet capable of causing catastrophic failures in neural networks.

Recent evidence suggests that such vulnerabilities extend to biological data as well. Gene expression data, despite being numerical and biologically structured, is not immune to adversarial manipulation. The landmark study *"Gene expression adverSCarial: assessing the vulnerability of scRNA-sequencing classifiers to adversarial attacks"* demonstrated that conventional scRNA-seq classifiers can be fooled through subtle, biologically plausible perturbations to gene expression vectors. In many cases, the perturbed samples remain virtually indistinguishable from clean samples in embedding spaces such as PCA or UMAP, yet the classifier's predicted cell type changes drastically. These findings expose a significant weakness in the reliability of ML-based scRNA-seq annotations.

This vulnerability has serious implications:

- **Scientific research:** Incorrect annotations can misguide downstream analyses such as trajectory inference or differential expression.
- **Clinical decision-making:** Diagnostic models may misclassify diseased or immune cell populations.
- **Drug discovery:** Perturbation-based screens may yield misleading results if classifiers are easily manipulated.
- **Data integrity:** Adversarial modifications could be introduced unintentionally due to noise or deliberately as a form of data poisoning.

Despite the critical nature of these risks, there is limited research on **defensive**
mechanisms tailored for biological data. Most existing solutions in other domains, such as adversarial training, are computationally expensive, may degrade clean performance, and are not always suitable for high-dimensional gene expression matrices.

Therefore, a need emerges for **efficient, interpretable, and biologically compatible defense frameworks** capable of detecting adversarial perturbations in scRNA-seq workflows. This project addresses this need by developing a comprehensive pipeline that:

1. **Trains a deep neural network classifier** to predict cell-type clusters from gene expression data.
2. **Applies PGD-based adversarial attacks** to evaluate the classifier's vulnerability.
3. **Learns meta-embeddings through a Triplet Network**, revealing underlying structural differences between clean and adversarial samples.

4. **Builds a meta-detector model** that distinguishes clean from adversarial samples using these learned embeddings.

This approach aligns closely with insights from the *adverSCarial* paper while introducing a new layer of defense not present in the original study. By combining deep learning, adversarial machine learning, and meta-learning, the project demonstrates both the fragility of existing scRNA-seq classifiers and a promising path toward more robust and trustworthy computational biology models.

2.2 Motivation

Single-cell RNA sequencing (scRNA-seq) has become a cornerstone technology in modern biology, enabling the precise characterization of heterogeneous cell populations at an unprecedented resolution. This has led to the rapid development of machine learning–based classifiers that can automatically annotate cell types from high-dimensional gene-expression profiles. Such models are increasingly being used in biomedical research, clinical analytics, and high-throughput diagnostics. However, recent work indicates that these classifiers—similar to other deep learning systems—may be vulnerable to adversarial perturbations that subtly manipulate input gene expression values to induce incorrect predictions. This concern forms the central motivation for the current study.

The *adverSCarial* framework by Fievet et al. (2025) demonstrated that even well-established scRNA-seq classifiers can be misled by minimal, often imperceptible changes in gene expression. Their results show that:

- **Single-gene perturbations** are sometimes sufficient to flip the predicted cell type, even when the altered gene is not a canonical biological marker.
- **Max-change and gradient-based attacks** can introduce tiny but coordinated modifications across many genes, maintaining visual similarity in UMAP and heatmap representations while still causing misclassification.
- All major classes of classifiers—including marker-based models, SVMs, random forests, and MLPs—exhibited vulnerabilities to adversarial manipulation.
- Even biologically non-significant or statistically insignificant genes can drive classifier decisions when perturbed strategically.

These findings underscore a critical problem: **the stability and trustworthiness of scRNA-seq classifiers cannot be assumed, especially under technical noise, biological variability, or malicious interventions**. Errors in classification may lead to incorrect downstream

biological interpretations, flawed cell annotation pipelines, and potential clinical misjudgments—issues that become increasingly important as scRNA-seq moves toward translational and diagnostic settings.

Furthermore, the biological data domain is especially susceptible to perturbation vulnerabilities because:

- scRNA-seq measurements inherently contain technical noise and dropouts.
- Gene expression values vary widely across experiments, laboratories, and sequencing platforms.
- Small changes in expression of key regulatory or structural genes can appear biologically plausible.
- Dimensionality (often 10,000–20,000 genes) makes classifiers highly sensitive to subtle adversarial gradients.

Given these concerns, developing **systems capable not only of predicting cell types but also detecting whether the input has been adversarially manipulated** is essential. While the *adverSCarial* toolkit focuses on generating attacks and exploring classifier weaknesses, it does not provide a mechanism for real-time detection of adversarial inputs.

This gap motivates the creation of a **dual-model defense system** consisting of:

1. **A primary classifier** that performs cell-type prediction from gene activity vectors.
2. **A secondary meta-detector** that evaluates internal model signals—embeddings, logits, confidence margins, entropy, gradient norms—to determine whether the input appears adversarial.

By studying how adversarial attacks manifest within the model's internal computational space, rather than solely at the input level, our approach aims to provide a robust and model-agnostic mechanism for detecting adversarial interference.

In summary, the primary motivations of this work are:

1. Ensuring Reliability of scRNA-seq Classification

Misannotations can compromise biological conclusions and downstream analyses. Adversarial robustness becomes essential for trustworthy automation of cell-type labeling.

2. Addressing Demonstrated Vulnerabilities from adverSCarial

The *adverSCarial* study clearly establishes that scRNA-seq models—including MLPs—are vulnerable to subtle perturbations that are difficult

to detect visually or statistically . These vulnerabilities demand countermeasures.

3. Providing a Real-Time Adversarial Detection Mechanism
Existing tools generate attacks, but none detect them at inference time. A meta-detector bridges this critical gap.

4. Enhancing Interpretability and Model Transparency
Adversarial behaviors reveal hidden decision patterns of classifiers. Monitoring embeddings, gradients, and logits helps expose these internal dynamics.

5. Establishing a Foundation for Robust scRNA-seq Deployment
As single-cell tools move toward clinical and translational environments, ensuring resilience against noise, experimental artifacts, and adversarial manipulation becomes essential.

2.3 Problem Statement

Machine learning models for single-cell RNA sequencing (scRNA-seq) classification are increasingly used for automated cell-type identification in biomedical research and emerging clinical applications. These models typically operate on high-dimensional gene-expression profiles and rely on learned representations to assign cell identities. However, recent studies—most notably the *adverSCarial* framework by Fievet et al. (2025) —have shown that even small, imperceptible perturbations in gene-expression values can cause significant misclassification across a wide range of scRNA-seq classifiers. These perturbations may arise from technical noise, biological variability, or intentionally crafted adversarial attacks.

Despite this demonstrated vulnerability, current scRNA-seq classification pipelines lack **mechanisms to detect whether an input has been adversarially manipulated**. Most existing tools focus on attack generation and robustness analysis but do not provide a real-time defense strategy capable of flagging suspicious or perturbed gene-expression vectors during inference. Consequently, erroneous or adversarially altered predictions may propagate into downstream analyses, potentially leading to misinterpretations of cell identities, misleading biological insights, or compromised clinical decisions.

Thus, the central problem addressed in this study is:
How can we design a system that not only classifies scRNA-seq cells accurately but also detects whether the input gene-expression profile has been adversarially manipulated?

To address this, the problem can be broken down into the following technical challenges:

1. **High-Dimensional Vulnerability:** Gene-expression vectors contain thousands of features, making deep-learning classifiers highly sensitive to subtle perturbations that may not be detectable through standard preprocessing or visualization.

2. **Lack of Adversarial Detection Mechanisms:** Existing frameworks (e.g., *adverSCarial*) focus on attack generation and robustness evaluation but do not detect adversarial inputs during inference.

3. **Complex Internal Behavior of scRNA-seq Classifiers:** Adversarial perturbations can drastically alter internal activations, gradients, and confidence distributions in ways that are not captured by the final predicted label alone.

4. **Need for a Model-Agnostic Detection Strategy:** The detection mechanism should not depend on the architecture of the primary classifier and should work even if the classifier is vulnerable.

Therefore, the problem this work aims to solve is the development of a **dual-stage adversarial detection pipeline** where:

- A **main classifier** predicts the cell type from gene-expression data.
- A **meta-detector** analyzes internal model signals (embeddings, logits, entropy, gradient norms) to determine whether the input is **clean** or **adversarial**.

This formulation allows the system to both perform accurate cell-type classification and safeguard against manipulated gene-expression inputs—an essential requirement for robust and trustworthy scRNA-seq analysis.

2.4 Objectives

The primary objective of this study is to develop a **robust adversarial detection system** for scRNA-seq classifiers that ensures reliable cell-type prediction even under malicious or noisy perturbations. The specific objectives are:

1. **Build a main classifier** capable of accurately predicting cell types from PBMC3k gene-expression data.

2. **Generate adversarial examples** using PGD to evaluate how easily the

classifier can be manipulated.

3. **Extract internal model signals** (embeddings, logits, confidence margins, gradients) to study how adversarial inputs affect the classifier's behavior.

4. **Train a triplet network** to learn a meta-embedding space that separates clean and adversarial patterns.

5. **Develop a meta-detector** that classifies inputs as adversarial or non-adversarial using these meta-features.

6. **Evaluate the full pipeline** in terms of classifier accuracy, attack success, and adversarial detection performance.

7. **Deploy a Streamlit demo interface** to visualize clean vs. adversarial predictions and meta-detector decisions.

3. Literature Review

3.1 scRNA-seq Classification Methods

Single-cell RNA sequencing (scRNA-seq) enables high-resolution profiling of gene expression at the level of individual cells, generating datasets with tens of thousands of features (genes) across thousands of cells. The rapid expansion of scRNA-seq technologies has led to the development of diverse computational models for automated cell-type classification. These methods fall broadly into the following categories:

(i) Marker-Based Methods : Marker-based systems such as **scType** rely on predefined sets of positive and negative markers to assign each cell to the most probable cell type. These systems are biologically interpretable and simple to deploy but depend heavily on the accuracy and consistency of marker-gene panels. Their performance degrades when marker expression is noisy, ambiguous, or dataset-specific, as observed in PBMC datasets.

(ii) Hierarchical Clustering Approaches : Tools like **CHETAH** classify cells through hierarchical decisions based on cell-to-reference similarity. These models preserve hierarchical relationships but may propagate early misassignments through the tree structure, making them susceptible to perturbations that alter early-stage decisions.

(iii) Classical Machine Learning Models: Support Vector Machines (SVMs), Random Forests (RFs), and Gradient-Boosted Trees have been used widely due to their robustness and relatively low computational overhead.

- **SVM-based scAnnotatR** uses binary SVM classifiers to evaluate the likelihood of each cell type.
- **scRF** leverages ensemble decision trees, which are known to saturate once a decision threshold is crossed—explaining their lower sensitivity to small perturbations as reported in the *adverSCarial* experiments.

(iv) Deep Learning Models: Deep neural networks, especially **multilayer perceptrons (MLPs)**, can learn complex nonlinear relationships within high-dimensional gene-expression data. However, deep models tend to be:

- highly sensitive to small input perturbations,
- prone to overfitting noise,
- difficult to interpret biologically.
-

The *adverSCarial* study confirms this: MLP-based classifiers (e.g., scMLP) showed **extreme susceptibility** to adversarial perturbations, with thousands of genes influencing outcome shifts under certain attack modes. This highlights the critical importance of understanding and securing scRNA-seq models before clinical application.

3.2 Adversarial Attacks in Biology

Adversarial attacks refer to intentionally crafted, small-scale perturbations made to input data that cause machine learning models to produce incorrect predictions. Although first explored in computer vision applications, these attacks have gained relevance in biomedical machine learning because biological models often operate on high-dimensional, noise-prone data. Even imperceptible input changes may lead to large shifts in model decisions, creating reliability and safety concerns.

3.2.1 Emergence of Adversarial Attacks in Biomedical Machine Learning: The earliest adversarial attacks, such as FGSM and PGD, demonstrated that neural networks could be fooled by extremely small and systematic perturbations. These ideas quickly transferred into biomedical domains. Models trained for medical imaging (X-ray, CT, pathology), ECG interpretation, and genomic sequence classification have all been shown to be adversarially vulnerable. The reason for this transfer is that biomedical models share similar characteristics with image classifiers:

- They process high-dimensional inputs.
- They use highly nonlinear architectures.
- They learn complex boundaries that can be manipulated.

3.2.2 Why Genomic and scRNA-seq Data Are Especially Vulnerable: Single-cell gene-expression profiles have several properties that make them highly susceptible to adversarial attacks.

(1) High dimensionality: A single scRNA-seq sample typically contains thousands of genes. This large feature space provides an opportunity to exploit many different directions for perturbation. Small changes distributed across many genes can have a strong cumulative effect on model predictions.

(2) Sparse and noisy data: scRNA-seq suffers from technical noise, dropout events, limited sequencing depth, and batch effects. These factors make adversarial perturbations harder to detect because they resemble normal biological variation.

(3) Biological plausibility: Compared to images where pixel changes can

be visually inspected, gene-expression modifications cannot be easily observed or validated. Even substantial modifications across gene sets may appear biologically reasonable and do not visually distort standard dimensionality-reduction plots such as UMAP or t-SNE.

(4) Sensitivity of machine-learning classifiers: Many scRNA-seq classifiers depend heavily on small sets of marker genes or nonlinear transformations. This makes them sensitive to minor gene-expression changes. A small perturbation applied to one or two key genes can shift a sample across decision boundaries.

(5) Hidden nature of perturbations: The adverSCarial paper demonstrates that clusters can keep their original shape in UMAP plots even after strong adversarial modification, while the underlying model predictions change drastically. This makes adversarial changes invisible to experts.

3.2.3 Types of Adversarial Attacks Used in Biological Data
Several categories of adversarial attacks have been adapted specifically for omics and gene-expression models.

(1) Gradient-based attacks: Methods such as FGSM and PGD use model gradients to determine the direction of perturbation. These attacks generate minimally small yet highly effective gene-expression shifts.

(2) Perturbation-based attacks: These include up- or down-regulating key genes, adding uniform random noise, or modifying coordinated sets of genes. They simulate realistic biological or technical changes.

(3) Model-agnostic or black-box attacks: Some biological models, such as random forests or marker-based classifiers, do not expose gradients. To attack these systems, finite-difference methods and search-based strategies are used.

The adverSCarial framework extends these techniques to scRNA-seq using methods such as single-gene attacks, max-change attacks, and cluster-based gradient descent.

3.2.4 Adversarial Attacks in scRNA-seq: Insights from the adverSCarial Paper: The adverSCarial study is one of the most comprehensive explorations of adversarial vulnerability in scRNA-seq classifiers. It reports the following major findings:
(1) Single gene perturbations can cause misclassification: The study reveals that modifying the expression of just one gene can be sufficient to

flip the predicted cell type. A key example is the manipulation of the ICAM1 gene in PBMC3k, which changes the predicted identity of naive B cells.

(2) Multi-gene coordinated perturbations can remain visually undetectable: The max-change attack modifies nearly all genes without affecting classifier output until a small set of "signature" genes is changed. This demonstrates that classifiers rely heavily on a few critical genes, and perturbations applied broadly can remain invisible in visualization tools.

(3) Deep learning models are especially vulnerable: The multilayer perceptron (scMLP) tested in the study showed high susceptibility, with thousands of possible successful perturbations under cluster-based gradient descent. This indicates that deep models amplify the effect of small changes in high-dimensional spaces.

(4) Black-box attacks are effective: Cluster-based gradient descent (CGD) uses finite difference approximations instead of real gradients. Despite this, it successfully fools classifiers like random forests and SVMs. This demonstrates that adversarial vulnerability exists even when the classifier's internal structure is hidden.

(5) UMAP and heatmap visualizations fail to detect attacks: The study shows that dimensionality reductions remain visually stable even when cells are misclassified—meaning domain experts cannot visually detect adversarial interference.

3.2.5 Implications for Biological and Clinical Workflows
Adversarial attacks in scRNA-seq pose several risks:
(1) Misleading automated cell-type annotation: Incorrectly annotated cells will distort downstream analyses such as differential expression, marker identification, and population profiling.

(2) Compromised clinical applications: scRNA-seq is increasingly used in precision medicine, tumor microenvironment profiling, and immunotherapy research. Adversarial changes could produce misleading clinical interpretations.

(3) Distorted multi-dataset integration: Batch correction and integration frameworks can be misled by artificially perturbed gene profiles, potentially harming downstream clustering or trajectory inference.

(4) Biased biological interpretability: Adversarial attacks may highlight or

suppress specific genes, causing researchers to misidentify markers, pathways, or cell-state regulators.

Biological data, especially scRNA-seq, are inherently vulnerable to adversarial attacks due to their high dimensionality, noise, and interpretability challenges. Studies like adverSCarial have confirmed that both classical and deep learning models can be easily fooled by subtle gene-expression perturbations. This underscores the need for reliable adversarial detection systems—an area where existing research remains insufficient.

3.3 adverSCarial Framework)

The *adverSCarial* framework (Fievet et al., 2025) is one of the first systematic investigations into the adversarial vulnerability of scRNA-seq classifiers. It introduces multiple attack algorithms designed specifically for gene-expression data. The paper extensively evaluates classifier robustness across **five algorithms**—scType, CHETAH, scAnnotatR (SVM), scRF, and scMLP—using **four major datasets**, including PBMC3k.

Key Contributions of adverSCarial

1. Novel scRNA-seq-Specific Attack Modes

The framework proposes several adversarial strategies:

- **Single-Gene Attack:** Identifies the minimum set of individual genes whose perturbation causes cell-type misclassification. Example from PBMC3k: switching on *ICAM1* causes naive B cells to be misclassified as endothelial cells (page 3–4).
- **Max-Change Attack:** Alters the largest number of genes without affecting classification, revealing the "signature genes" whose changes are most critical.
- **Cluster-Based Gradient Descent (CGD):** A gradient-inspired attack adapted for black-box models. It modifies entire cell clusters iteratively, using approximated gradients (described in equations 3–4 on pages 4–5).
- **Random-Walk and Brute-Force Attacks:** Explore broad modification combinations, uncovering unanticipated vulnerabilities.

2. Comprehensive Vulnerability Analysis

The findings show:

- **scRF** is the most robust to moderate perturbations.
- **scMLP** is extremely vulnerable, with thousands of genes affecting

prediction under CGD (page 5–7).

- **Marker-based** and **SVM** models experience failures with single-gene perturbations.
- All classifiers show **adversarial weaknesses**, even when perturbations are biologically subtle or visually undetectable on UMAP and heatmaps (page 7).

3. Visualization of Attack Impact

UMAP projections and heatmaps (pages 6–7) demonstrate that:

- perturbations can preserve cluster topology,
- yet cause complete misclassification.

This shows that adversarial attacks can remain hidden from domain experts by blending into biological variability.

3.4 Gaps Identified in Existing Research

While *adverSCarial* thoroughly analyzes adversarial **vulnerabilities**, a crucial limitation remains—one that directly motivates this study.

1. No Mechanism for Detecting Adversarial Inputs

The *adverSCarial* framework is **purely diagnostic**, not defensive. It reveals *whether a model is vulnerable*, but not *how to protect it*.

Specifically:

- The paper provides **no algorithm**, **architecture**, or **model** to detect whether a new gene-expression sample has been adversarially perturbed.
- All conclusions are based on **known attacks**, with no general method for identifying unseen adversarial manipulations.
- The framework assumes that a user manually checks classifier outputs or visualizations but does not provide automated detection.

2. No Use of Internal Model Behavior for Defense

The study does not analyze or utilize:

- internal embeddings,
- logits distribution,
- confidence margins,
- entropy,
- or input gradients

as indicators of adversarial manipulation—even though these signals change significantly under attack (observed indirectly in their experiments).

3. No Meta-Learning or Secondary-Classifer Approach
Modern adversarial defense research often uses:

- meta-models,
- detectors trained on adversarial vs. clean internal signals,
- triplet or contrastive embeddings.

None of these techniques are explored in *adverSCarial*.

4. Focus Limited to Robustness Testing, Not Prevention
While the paper provides:

- attack algorithms,
- visualization tools,
- performance analysis

it does **not** address:

- how to secure classifiers,
- how to monitor inference-time inputs,
- how to ensure robustness in practical pipelines.

5. No Real-Time or Deployment-Level Defense Mechanism
Even though adversarial attacks are shown to be subtle and dangerous in biological workflows, the framework:

- does not propose runtime detection,
- does not discuss integration into scRNA-seq pipelines,
- does not provide a thresholding or scoring system for adversarial flagging.

4. Dataset Description

The present study uses the PBMC3k dataset as the foundation for training, generating adversarial samples, and evaluating the robustness of the proposed classifier and meta-detector system. PBMC datasets are widely used in scRNA-seq benchmarking due to their cell-type diversity, clean experimental preparation, and well-established annotation pipelines. This section provides a detailed description of the dataset, preprocessing pipeline, and feature engineering strategy used in this work.

4.1 PBMC3k Dataset Overview

The PBMC3k dataset is one of the most widely used benchmark datasets in single-cell genomics and serves as the primary dataset in this study. PBMC stands for Peripheral Blood Mononuclear Cells, a diverse group of blood cells consisting primarily of lymphocytes and monocytes. These cells play essential roles in adaptive and innate immunity, making them ideal for cell-type classification research.

The dataset, as described in the adverSCarial paper (page 2) , contains approximately 2700 high-quality cells sourced from a healthy human donor. It was generated using the 10x Genomics Chromium Single Cell 3' v1 chemistry and processed through the Cell Ranger v1.1.0 pipeline. Poor-quality cells (e.g., low gene count or high mitochondrial content) were already filtered out by the data providers, ensuring a clean, high-confidence dataset suitable for robust machine-learning analysis.

The PBMC3k dataset contains several immune cell populations with varying degrees of transcriptional similarity. Some groups are highly distinguishable (e.g., naïve B cells vs. monocytes), while others share close transcriptional profiles (e.g., CD14+ classical monocytes vs. FCGR3A+ non-classical monocytes). This variability in gene-expression similarity across cell types makes PBMC3k a perfect testbed for adversarial experiments where subtle perturbations can exploit fine-grained boundaries.

The major cell types commonly annotated in PBMC3k include:

1. Naïve B cells
2. Memory B cells
3. Naïve CD4 T cells
4. Memory CD4 T cells
5. CD8 T cells
6. NK (Natural Killer) cells
7. Classical CD14+ monocytes
8. Non-classical FCGR3A+ monocytes

9. Dendritic cells
10. Platelets (rare but retained)

Each of these cell types exhibits unique yet partially overlapping expression signatures. For example:

- B cells typically express MS4A1, CD79A, CD74.
- NK cells express NKG7, GNLY.
- Monocytes express LST1, S100A8, S100A9.
- CD4/CD8 T cells express IL7R, CCR7, CD3D, CD8A.

These marker patterns are exploited by classical classifiers and also become targets for adversarial perturbations. The adverSCarial paper demonstrates, for example, that artificially increasing the expression of the ICAM1 gene flips naïve B cells into the endothelial cell cluster (page 3). This highlights how sensitive these profiles are to perturbations.

4.1.1 Origin and Data Generation Workflow

The dataset was produced using the following steps:

1. PBMCs were isolated from a healthy donor via density-gradient centrifugation.
2. The 10x Genomics microfluidic platform encapsulated single cells with barcoded beads.
3. mRNA transcripts were reverse-transcribed into barcoded cDNA.
4. Libraries were sequenced using Illumina NextSeq 500.
5. Cell Ranger aligned reads to the human transcriptome and produced a cell × gene expression matrix.
6. Poor-quality cells (less than 200 or more than 2500 expressed genes, or >5% mitochondrial content) were removed as noted in the adverSCarial dataset description (page 2).

This results in a highly refined dataset suitable for downstream ML tasks.

4.1.2 Structure of the Gene–Cell Expression Matrix

The core of the PBMC3k dataset is a sparse matrix with:

- Rows representing individual cells (≈2700)
- Columns representing genes (≈20,000 before HVG filtering)
- Values representing UMI (Unique Molecular Identifier) counts

UMIs provide a near-quantitative measure of transcript abundance for each gene in each cell.

Example: Gene Expression Matrix (Subset)

	Gene1	Gene2	Gene3	Gene4	Gene5
Cell_001	0	3	12	0	1
Cell_002	7	0	4	2	0
Cell_003	0	1	0	0	0
Cell_004	15	2	0	3	9
Cell_005	0	0	8	1	2

Where:

- Each cell has a unique barcode ID (e.g., Cell_001).
- Each gene represents a measured transcript (e.g., MS4A1, CD3D, LST1).
- Values represent raw UMI counts before normalization.

After preprocessing (normalization, log transform, scaling), the same matrix becomes continuous-valued, typically between 0 and 10.

4.1.3 Why PBMC3k is Ideal for Adversarial Learning Research

The PBMC3k dataset is uniquely suitable for adversarial machine-learning research because:

1. **Heterogeneous yet related cell types**: This creates sensitive decision boundaries that adversarial attacks can exploit.
2. **High dimensionality (~20,000 genes)**: This produces a large attack surface, enabling many possible perturbation directions.
3. **Well-studied biomarkers**: Marker genes allow biological interpretation of adversarial perturbations.
4. **Pre-filtered, high-quality data**: Ensures that adversarial effects are due to intentional perturbation, not data noise.
5. **Used in the adverSCarial paper itself**: This provides consistency between your project and existing literature, as PBMC3k is one of the four datasets used in their experiments (page 2) .
6. **Reproducibility**
   The dataset is publicly available and used widely in benchmarks.
7. **Possesses both easy and difficult classification tasks**
   - Naïve vs. memory T cells → hard boundary
   - B cells vs. monocytes → contrasting profiles
   - Monocyte subtypes → subtle distinctions

This diversity allows the adversarial pipeline (classifier + PGD + meta-detector) to be evaluated under multiple difficulty levels.

4.2 Preprocessing Steps

The raw PBMC3k gene-cell matrix undergoes extensive preprocessing to remove technical biases and prepare the expression matrix for machine-learning tasks. Preprocessing is performed using the Scanpy pipeline, with

reference alignment to the preprocessing strategy described for PBMC3k in the adverSCarial paper (page 2).

The major preprocessing steps are described below:
Step 1: Raw Count Normalization: The expression counts of each cell are normalized such that every cell has a total transcription count of 10,000. This step corrects for variation in sequencing depth across cells.

Step 2: Log Transform: A log1p transformation is applied to stabilize variance, reduce the influence of outlier expression values, and improve separability in downstream feature space.

Step 3: Identification of Highly Variable Genes: Highly variable genes (HVGs) play a central role in driving transcriptional heterogeneity between cell populations. To reduce noise and dimensionality, the top 2000 HVGs are selected.

Step 4: Subsetting to HVGs: Only the HVGs are retained for analysis. This reduces the input dimensionality from ~20,000 genes to 2000, making classification more stable and computationally efficient.

Step 5: Scaling and Standardization: Each gene is centered to mean 0 and: scaled to unit variance. Values are clipped to a maximum of 10 to reduce the impact of extreme outliers.

Step 6: Principal Component Analysis (PCA): Fifty principal components are computed using the scaled HVG matrix. These PCs capture the major axes of transcriptional variance and are used for clustering and initial quality assessment.

Step 7: Quality Control (built into the dataset): Based on the thresholds described in the adverSCarial paper (page 2), PBMC3k cells with fewer than 200 expressed genes or more than 2500 expressed genes were removed before release. Moreover, cells with more than 5% mitochondrial content were excluded in the original dataset preparation.
The resulting dataset is clean, homogeneous, and suitable for downstream adversarial experimentation without requiring additional QC.

4.3 Feature Engineering and Highly Variable Gene Selection

Feature engineering is one of the most crucial stages in building a robust scRNA-seq classification and adversarial detection pipeline. Unlike image data, where raw pixel values contain visual structure, scRNA-seq data is highly sparse, noisy, and high-dimensional, with only a small subset of

genes carrying most of the biologically relevant information. A raw PBMC3k matrix contains approximately 20,000 genes, many of which are housekeeping genes, low-variance transcripts, or stochastic noise. Feeding all raw genes directly into a neural network greatly increases model instability and makes the classifier extremely vulnerable to adversarial perturbations.

For this reason, feature engineering focuses on reducing dimensionality while retaining biologically meaningful transcriptional variation. The core components of feature engineering used in this study include:

1. Identification and selection of highly variable genes (HVGs)
2. Biological justification for HVG selection
3. Standardization and scaling to stabilize learning
4. Constructing a well-behaved input space for adversarial attacks
5. Preparing meta-features that support artifact-free adversarial detection

4.3.1 Highly Variable Genes (HVGs): Rationale and Importance

Highly Variable Genes represent the subset of genes whose expression fluctuates significantly across cells, above what is expected from technical noise. These genes are typically associated with cell identity, activation states, lineage commitment, and immune functions. In PBMC datasets, for example, HVGs include:

- MS4A1, CD74 (B cell markers)
- NKG7, GNLY (NK cell markers)
- IL7R, CCR7 (T cell markers)
- LST1, S100A8, S100A9 (monocyte markers)

Because adversarial attacks attempt to distort classifier decision boundaries by perturbing gene expression, retaining biologically relevant genes is essential.

Selecting HVGs provides the following benefits:

1. Removes tens of thousands of low-information genes, reducing noise.
2. Forces adversarial attacks to focus on genes that actually affect cell identity.
3. Improves classifier stability and prevents overfitting.
4. Enables interpretable adversarial perturbation analysis.
5. Creates a controlled and biologically meaningful space for meta-feature extraction.

The adverSCarial paper highlights that perturbations applied to irrelevant

genes can still fool classifiers, especially deep models like scMLP (pages 5–7). Reducing such noise-driven vulnerabilities through HVG selection is therefore essential.

4.3.2 Method for HVG Selection (Scanpy Seurat-Flavored Approach)
In this study, HVGs are selected using the Scanpy function "highly_variable_genes" with Seurat-style parameters. This method models the mean–variance relationship of each gene and flags those exhibiting significantly higher dispersion.
Procedure summary:
Step-1:Compute each gene's average expression across all cells
Step-2:Compute the gene's dispersion (variance/mean)
Step-3:Adjust dispersion for expected technical noise
Step 4: Determine HVGs by selecting the top 2000 genes ranked by corrected dispersion

This yields a set of genes that capture real biological heterogeneity. The number 2000 is a widely accepted compromise between biological richness and computational feasibility.

4.3.3 Why 2000 HVGs? Justification for the Dimensionality Choice
The selection of 2000 HVGs is motivated by:
1. Biological coverage: Enough genes to represent all major immune lineages.
2. Statistical reliability: Ensures that downstream PCA or neural network layers receive sufficient signal.
3. Computational efficiency: Keeps memory and processing time manageable, especially during adversarial PGD attack generation.
4. Adversarial defense considerations: A moderately high yet focused dimensionality forces adversarial attacks to operate in a biologically meaningful space rather than manipulating noise genes.

The adverSCarial paper similarly uses HVG-based processing (page 2) before PCA and UMAP analysis, reinforcing the biological validity of this dimensionality.

4.3.4 Biological Relevance of HVGs in PBMC3k
In PBMC3k, HVGs typically include:
1. Genes linked to immune activation
2. Genes regulating antigen presentation
3. Cytotoxicity-related genes in NK and T cells
4. Ribosomal genes affecting protein synthesis
5. Chemokine receptors distinguishing T cell subtypes
6. Genes associated with monocyte inflammatory responses

These HVGs naturally encode the main axes of variation defining immune cell identity. By training the classifier on HVGs, the model learns biologically grounded decision boundaries rather than overfitting to noise. This also means adversarial attacks on HVGs must mimic real biological patterns to deceive the classifier, making adversarial detection more meaningful.

4.3.5 Standardization: Centering, Scaling, and Clipping

Once HVGs are selected, each gene undergoes standardization:

1. The mean expression of each gene is subtracted (centering).
2. The gene is divided by its standard deviation (scaling).
3. Extreme values are clipped at a maximum of 10 to reduce the influence of outliers.

This normalization ensures:

- Genes with large dynamic ranges do not overshadow more stable genes
- Gradients in the classifier remain stable during backpropagation
- Adversarial perturbations operate within biologically plausible ranges
- Meta-gradient features (such as gradient norms and mean gradient magnitude) remain comparable across genes

Without standardization, adversarial PGD attacks would target genes with artificially high variance, leading to unrealistic perturbations.

4.3.6 Feature Space for the Main Classifier

After selecting HVGs and applying standardization, each cell is represented as a 2000-dimensional continuous-valued vector.

This vector serves as the input to the main classifier. The decision to use raw HVG values (as opposed to PCA embeddings) is intentional:

- Neural networks benefit from high-dimensional raw biology.
- PCA compresses biological signals and makes adversarial gradients less interpretable.
- Raw HVG expression allows the triplet network and meta-detector to learn fine-grained adversarial signatures.

Thus, the main classifier learns directly from biologically grounded features.

4.3.7 Feature Space for the Meta-Model

Feature engineering is also essential for training the meta-detector.

The meta-model receives a combined set of internal classifier features, including:

- Deep embeddings from the classifier
- Logits (pre-softmax outputs)
- Softmax probability distributions
- Logit margin (difference between top two class scores)

- Shannon entropy (measuring prediction uncertainty)

These features are concatenated to form a meta-feature vector representing the classifier's internal behavior.

This approach leverages the fact that adversarial samples produce abnormal activations and unstable gradients inside the classifier, even if the final prediction does not change. These aberrations form measurable patterns in the meta-feature space.

4.3.8 Triplet Embedding of Meta-Features

To further enhance adversarial separability, the meta-features are passed through a triplet network that learns a lower-dimensional (64-D) meta-embedding space. The triplet network ensures:

- Clean samples cluster together
- Adversarial samples form a distinct cluster
- Embeddings maximize inter-class distance and minimize intra-class distance

This allows the final meta-detector (binary classifier) to operate in a robust and discriminative feature space.

4.3.9 Relationship Between Feature Engineering and Adversarial Robustness

Feature engineering plays a central role in determining both classifier vulnerability and adversarial detectability:

1. HVG selection reduces noise-based vulnerabilities
2. Standardization stabilizes gradients, reducing adversarial susceptibility
3. Meta-feature engineering reveals internal model anomalies caused by adversarial attacks
4. Triplet embeddings enforce well-separated adversarial clusters
5. A structured feature space allows the meta-detector to reliably detect adversarial perturbations

The adverSCarial paper demonstrates that models trained on poorly engineered features (e.g., raw full-gene matrices) are vastly more vulnerable, especially deep models like scMLP (page 5–7). Our feature engineering strategy specifically addresses these weaknesses.

4.3.10 Summary

Feature engineering transforms the PBMC3k dataset from a noisy, extremely high-dimensional gene matrix into a structured, biologically meaningful representation suitable for classification and adversarial detection. By selecting HVGs, applying careful standardization, and engineering internal meta-features, the study ensures:

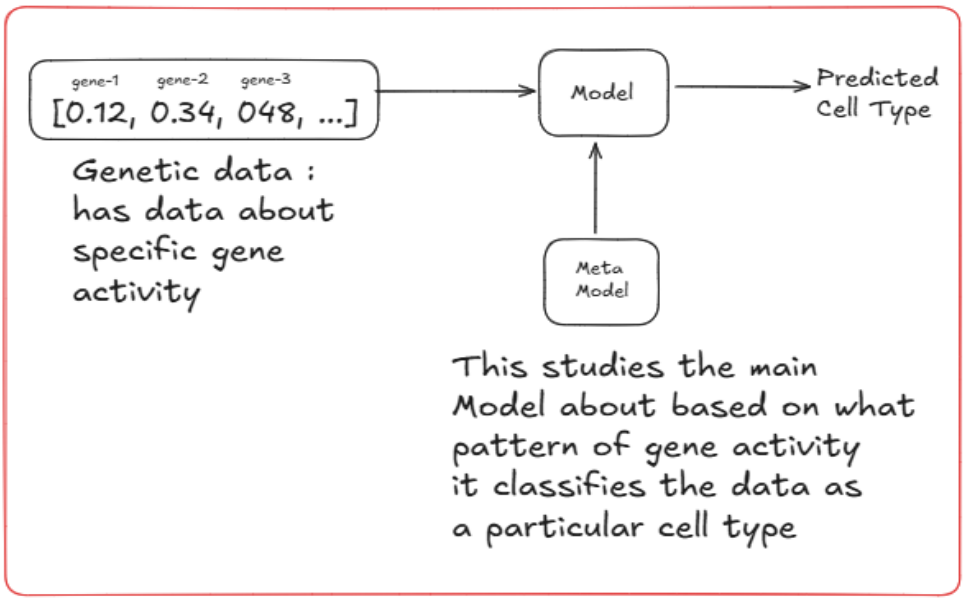
- The classifier learns reliably
- PGD adversarial attacks operate realistically

- The meta-detector can identify attacked samples based on internal inconsistencies

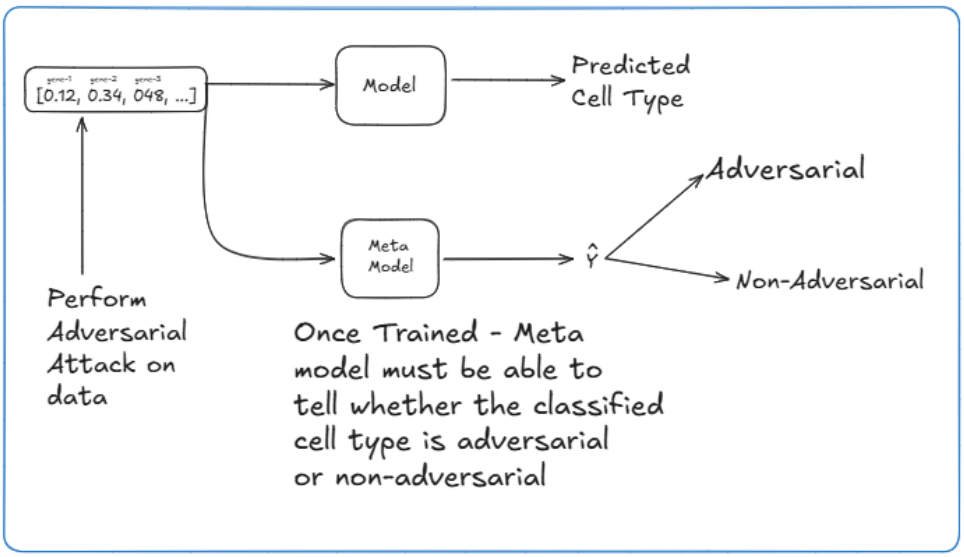
In essence, the success of the entire adversarial detection pipeline depends on the precision and quality of the feature engineering stage.

5. Methodology

This chapter presents the full technical methodology used to build, attack, and defend an scRNA-seq cell-type classifier using PBMC3k. The pipeline consists of a main classifier (MLP), adversarial attack generation (PGD multi-ε), a triplet network that learns a meta-embedding from internal signals of the classifier, and a meta-detector that flags adversarial inputs. Each subsection describes design choices, mathematical formulation, implementation details, hyperparameters, and reproducibility guidelines.



Training



Testing

5.1 System Architecture

The system follows a two-stage architecture:

1. Main classifier (stage A): A supervised MLP trained on HVG-selected, preprocessed PBMC3k vectors to predict cell-type labels. It provides outputs and internal signals used by later modules.
2. Adversarial detection (stage B): A meta-system composed of (a) an adversarial attack generator (PGD multi-ε) used for training data creation and evaluation; (b) a triplet network that maps concatenated internal signals to a compact meta-embedding; and (c) a binary meta-detector that classifies meta-embeddings as clean (0) or adversarial (1).

The pipeline at inference is:
Input gene vector x → scaler → main classifier → predicted cell type and internal signals (embeddings, logits, softmax probabilities, gradient stats) → triplet model → meta-detector → adversarial flag.
Rationale and design constraints

- Model-agnostic detection: The meta-detector is designed to rely on internal behavioral signals available from most differentiable classifiers (logits, embeddings, gradients), not on the specific classifier architecture. This supports portability to other architectures (e.g., CNNs, Transformers).
- Interpretability: Extracted features (logit margins, entropy, gradient norms) have interpretable meaning about model confidence and sensitivity.
- Robustness to attack variants: Training the meta-model with adversarial examples generated across multiple ε values increases generalization to unseen attack strengths.
- Computational feasibility: HVG selection (2000 genes) and batching reduce memory and speed costs for PGD generation and gradient computation.

High-level component specification
Component: Main Classifier (MLP)

- Input: x ∈ R^D (D = 2000 HVGs)
- Output: logits z ∈ R^C (C = number of clusters)
- Internal: embedding e ∈ R^E (E = 64 by default)

Component: Adversarial Generator (PGD multi-ε)

- Produces K adversarial variations for each input with epsilons {ε1, ε2, ...}.

Component: Feature Extractor

- Collects e, z, p = softmax(z), margin = z_top1 − z_top2, entropy H(p),

and gradient statistics ∇_x L (mean, variance, norm).

Component: Triplet Network

- Maps high-dim meta-features f_meta ∈ R^M to compact embedding r ∈ R^R (R = 64).

Component: Meta-Detector

- Binary classifier on r; outputs score s ∈ R. Thresholding produces binary label.

Diagram (textual)

- Input: x (HVG vector) → Normalize → MLP → (logits z, embedding e)
- Also compute loss with y_true to get gradient ∇_x L
- Concatenate [e, z, p, margin, entropy, grad_mean, grad_var, grad_norm] → meta-feature vector f_meta
- TripletNet: f_meta → r
- MetaDetector: r → s → threshold → adversarial? yes/no

Data flow notes

- For training meta-detector we create balanced meta-datasets: clean samples labeled 0 and adversarial samples labeled 1 (adv generated with PGD multi-ε).
- Triplet training uses (anchor, positive clean, negative adv) sampling to push adv and clean apart in embedding space.

Reference

- Attack selection and motivation are informed by adverSCarial's attack taxonomy (single-gene, max-change, CGD) and the need to test a diversity of attack intensities. The PGD white-box attack is used here as a standard, powerful gradient-based adversary (see adverSCarial discussion of gradient-based methods).

5.2 Main Classifier (MLP)

Design goals

- High classification accuracy on clean PBMC3k to ensure meaningful adversarial evaluation.
- Stable gradients and well-behaved internal embeddings to facilitate extracting meta-features.
- Regularization to avoid overfitting while preserving sensitivity to genuine biological signal.

5.2.1 Architecture

Notation:

- Input dimension D = number of HVGs (2000).
- Hidden layer sizes described left→right.
- Embedding dimension E = 64.
- Number of classes C = number of clusters determined from clustering

(e.g., 9–10).

Recommended architecture (detailed)

- Input layer: Linear(D → 256)
- Activation: ReLU
- Dropout: 0.4
- Linear: 256 → 128
- Activation: ReLU
- Dropout: 0.4
- Linear: 128 → 64
- Activation: ReLU
- Dropout: 0.3
- Embedding output: emb = last hidden activation ∈ R^64
- Final linear: Linear(64 → C) → logits z

Justification:

- Layer widths balance capacity and overfitting risk.
- Dropout rates selected to reduce overfitting and smooth gradients (useful for gradient-based feature extraction).
- Embedding dimension 64 is large enough to capture class-specific structure but compact for triplet training.

5.2.2 Mathematical forward pass

Given x ∈ R^D:

h1 = ReLU(W1 x + b1)

h1_drop = Dropout(h1)

h2 = ReLU(W2 h1_drop + b2)

h2_drop = Dropout(h2)

emb = ReLU(W3 h2_drop + b3) (emb ∈ R^E)

logits z = W_out emb + b_out

Softmax probabilities p = softmax(z)

Loss function: Cross-Entropy

$L(x, y; \theta) = -\sum_{k=1}^{C} 1[y=k] \log \text{softmax}(z)_k$

5.2.3 Training strategy

Objective

- Minimize cross-entropy on training set; early stop on validation accuracy.

Optimizer and hyperparameters

- Optimizer: AdamW
- Learning rate: 1e−3 (tunable)
- Weight decay: 1e−3
- Batch size: 64
- Epochs: up to 60 with early stopping patience = 5

- Random seed fixed for reproducibility (e.g., 42)
- Device: GPU if available (CUDA); otherwise CPU

Regularization methods

- Dropout at hidden layers
- Weight decay (L2)
- Optional: label smoothing (0.05) to avoid overconfident logits which can help gradient-based defenses

Data splits

- Train/test split: stratified; typical split 50/50 or 80/20 depending on dataset size
- Use same splits for all downstream adversarial experiments

Training loop (pseudocode)

```
for epoch in 1..max_epochs:
model.train()
for xb, yb in train_loader:
xb = xb.to(device)
yb = yb.to(device)
optimizer.zero_grad()
logits, emb = model(xb, return_embed=True)
loss = CrossEntropy(logits, yb)
loss.backward()
optimizer.step()
evaluate on validation set early stop when no improvement for patience
epochs
```

Checkpointing

- Save model weights for best validation accuracy.
- Save optimizer state for reproducibility.
- Log training/validation loss and accuracy per epoch.

Calibration & confidence

- You may calibrate predicted probabilities (temperature scaling) post-hoc if meta-detector needs better-calibrated confidences. However, raw logits are also useful as meta-features.

5.2.4 Properties to expose for meta-features

During both training and inference, the following internal signals are recorded per input:

1. Embedding emb ∈ R^E
2. Logits z ∈ R^C

3. Softmax probabilities p ∈ R^C
4. Logit margin m = z_top1 − z_top2 (scalar)
5. Entropy H(p) = −∑ p_i log p_i (scalar)
6. Gradient ∇_x L (shape = D)
    - From gradient, compute summary statistics:
      a. grad_mean = mean(|∇_x L|)
      b. grad_var = var(|∇_x L|)
      c. grad_norm = ||∇_x L||_2

These signals capture confidence, decision boundary sharpness, and sensitivity of the loss to input perturbations — all useful to detect adversarial inputs.

5.2.5 Implementation notes and reproducibility

- Use torch.manual_seed(42) and numpy.random.seed(42).
- Ensure deterministic behavior where possible (torch.backends.cudnn.deterministic = True), but note this may slow training.
- Save scaler, label encoder, and HVG gene list alongside model weights.
- Document environment: Python version, PyTorch version, CUDA version, hardware.

5.3 Adversarial Attack Generation (PGD)

Rationale

PGD (Projected Gradient Descent) is the canonical strong first-order adversary. For our experiments, PGD creates adversarial examples to train the meta-detector and evaluate classifier robustness. We generate multiple epsilons per sample to make the meta-detector robust to varying attack strengths.

5.3.1 Formal definition

Given input x and true label y, PGD iteratively updates x_adv:

x_adv^0 = x

for t = 0..T-1:

g = ∇*x L(θ, x_adv^t, y)*

x_adv^{t+1} = Clip{x, ε} (x_adv^t + α * sign(g))

Clip_{x, ε}(·) enforces ||x_adv − x||_∞ ≤ ε (or another norm), and bounds due to scaling are applied.

We use the L∞ variant (most common) because it provides a controlled maximum per-gene perturbation.

Parameters

- eps_list: [0.05, 0.1, 0.15] (these are on standardized data; choose values consistent with scale)
- α (step size): 0.02 (typical α = eps / number_of_steps; chosen small)
- iterations T: 10

- Norm: L∞

Generate multiple adversarial points per clean sample using each ε ∈ eps_list; these are used to create meta-training positives.

5.3.2 Implementation details

- PGD runs on the preprocessed, scaled feature space (post StandardScaler). This ensures ε bounds are meaningful relative to the model's input scale.
- Use requires_grad_(True) for x_adv during update and compute loss with the model and true label y.
- After each gradient step, clip difference to [-ε, +ε] relative to x.
- Optionally enforce data range constraints (min, max) to keep adversarial examples within biologically plausible ranges (e.g., if you clip to [min_val, max_val]).
- When generating many adversarial samples (e.g., for the entire test set), generate in batches to utilize GPU and reduce memory copying.

PGD batch pseudocode

```
def pgd_multi_eps(model, x, y, eps_list, alpha=0.02, iters=10):
advs = []
for eps in eps_list:
x0 = x.clone().detach()
x_adv = x0.clone()
for i in range(iters):
x_adv.requires_grad_(True)
logits = model(x_adv)
loss = CrossEntropy(logits, y)
grad = torch.autograd.grad(loss, x_adv)[0]
with torch.no_grad():
x_adv = x_adv + alpha * grad.sign()
x_adv = x0 + torch.clamp(x_adv - x0, -eps, eps)
advs.append(x_adv.detach())
return torch.cat(advs, dim=0)
```

5.3.3 Attack variants and rationale

- Multi-ε PGD: Training the meta-detector with multiple ε values improves generalization to unseen attack strengths.
- Random-start PGD (optional): Initialize x_adv^0 = x + Uniform(-ε, ε). This increases attack strength and diversity.
- L2 variants: For alternative experiments, you can use L2-bounded PGD. However, preprocessed scRNA-seq scaling often makes L∞ attacks easier to interpret.
- Biologically-constrained perturbations: Optionally enforce constraints (e.g., non-negativity in raw counts or maximum fold-change per gene) to

keep adversarial samples biologically realistic. This is important when claiming real-world relevance.

5.3.4 Computational considerations

- PGD requires gradient calculation with respect to input ($\nabla_x L$), which is used both for attack direction and to compute meta-features. Compute gradients in double-buffered mode (avoid autograd graph retention) to reduce memory.
- Generating adversarial samples for the entire dataset and multiple ε can multiply dataset size (e.g., 3x if three epsilons). Use subsampling if resources are limited (e.g., use 25% of train set for meta-training as in the implementation flow).
- Store generated adversarial examples or compute features on the fly; computing features and saving them (embeddings, logits, gradient summaries) is more storage-efficient than saving full adversarial matrices if dataset large.

5.3.5 Comparison with adverSCarial attack modes

- adverSCarial catalogs specialized attacks (single-gene, max-change, CGD) that are useful to explore model vulnerability comprehensively. We use PGD as a principled gradient-based method and recommend running targeted adverSCarial modes for complementary experiments and biological plausibility checks.

5.4 Triplet Network for Embedding Extraction

Objective

The triplet network learns a meta-embedding space where clean samples cluster together and adversarial samples are pushed away. Using triplet loss improves separability for the downstream binary classifier and reduces sensitivity to absolute scaling.

5.4.1 Meta-feature vector construction

Before the triplet, for each sample we compute f_meta:

f_meta = concat(emb, logits, probs, margin, entropy, grad_mean, grad_var, grad_norm)

Dimension $M = E + C + C + 1 + 1 + 1 + 1 + 1 = E + 2C + 6$ (example: E=64, C~10 → M ~ 64 + 20 + 6 = 90)

All entries are standardized (zero mean, unit variance) before feeding to triplet network.

5.4.2 Triplet network architecture

Suggested network:

- Input: M-dim

- Linear(M → 256)
- ReLU
- Linear(256 → R) where R = 64 (final meta-embedding dim)
- L2-normalize output vector

Why L2 normalization: Triplet loss is often optimized on normalized vectors for stable margin interpretations.

5.4.3 Triplet loss and sampling

Triplet loss formulation

Given anchor a, positive p, negative n embeddings after the network $\varphi(\cdot)$:

$$L_{triplet} = \sum_i \max(0, \|\varphi(a_i) - \varphi(p_i)\|_2^2 - \|\varphi(a_i) - \varphi(n_i)\|_2^2 + margin)$$

- margin (triplet margin) typically 0.5–1.0
- We use margin = 1.0 in experiments (tunable)

Sample selection

- Anchor a: a clean sample
- Positive p: another clean sample of same class (or other clean sample — in our training we used same-class positives to enforce tightness)
- Negative n: adversarial sample (from any class) corresponding to the anchor's original label

Sampling strategy (mini-batch):

- For batch size B, sample B anchors from clean pool, B positives (random clean same-class), B negatives (random adv). Balanced sampling maintains representation.

Triplet sampling pseudocode

```
def sample_triplets(X_meta, y_meta, n_batch=64):
clean_idx = np.where(y_meta==0)[0]
adv_idx = np.where(y_meta==1)[0]
a = np.random.choice(clean_idx, n_batch)
p = np.random.choice(clean_idx, n_batch)
n = np.random.choice(adv_idx, n_batch)
return X_meta[a], X_meta[p], X_meta[n]
```

Training details

- Optimizer: AdamW, lr=1e−3
- Epochs: 10–20 (converges quickly)
- Batch size: 64
- Loss: TripletMarginLoss(margin=1.0) in PyTorch

Why triplet helps

- Produces a compact, discriminative embedding which reduces input size for meta-detector.
- Enforces geometric constraints that directly map to detection objective.

5.4.4 Evaluation of embedding quality

- Use t-SNE/UMAP on triplet embeddings to visualize separation of clean

vs adv.
- Compute intra-class and inter-class distances:
  intra_clean = mean_pairwise_distance(clean_embs)
  inter_clean_adv = mean_distance_between_sets(clean_embs, adv_embs)
- A larger inter_clean_adv / intra_clean ratio indicates better separability.

5.5 Meta-Detector Model

The meta-detector is the final binary classifier that predicts whether an input is adversarial given the triplet embedding r. It must be accurate, robust, and efficient at inference.

5.5.1 Architecture

Design choices

- Simple residual MLP with a small parameter count for quick inference.
- Input dimension: R (triplet embedding size, e.g., 64)
- Residual blocks help gradient flow for deeper networks and capture non-linearities.

Concrete architecture

- Input Linear(R → 128)
- ReLU
- ResBlock(128) [ResBlock: x + FC2(ReLU(FC1(x)))]
- ResBlock(128)
- Linear(128 → 1) → output logit s

Activation & output

- Final output is logit s. For training use BCEWithLogitsLoss.
- At test time, score = s; apply threshold τ to produce binary prediction (pred = 1 if s > τ else 0)

5.5.2 Training the meta-detector

Dataset

- X_trip: triplet_model(f_meta) generated for both clean and adv samples
- y_meta: binary labels (0 for clean, 1 for adv)
- Balanced training recommended (equal numbers of clean and adv). If adv samples outnumber clean, downsample or use balanced batch sampling.

Loss & optimizer

- Loss: BCEWithLogitsLoss
- Optimizer: AdamW
- Learning rate: 1e−3
- Batch size: 64
- Epochs: 30 (monitor validation loss)
- Early stopping: monitor validation loss/patience 5

Training loop pseudocode

for epoch in epochs:

```
meta_model.train()
for xb, yb in meta_loader:
xb, yb = xb.to(device), yb.to(device)
meta_opt.zero_grad()
logits = meta_model(xb).squeeze()
loss = BCEWithLogitsLoss(logits, yb)
loss.backward()
meta_opt.step()
```

Calibration

- The meta-detector scores may be calibrated post-hoc with logistic calibration (Platt) or isotonic regression to map logits to probability estimates.
- Calibration helps in selecting an operational threshold.

5.5.3 Scoring and thresholding

Score definition

- Meta-detector outputs score s (logit). We can convert to probability via $\sigma(s) = 1 / (1 + \exp(-s))$.
- Alternatively, use the raw logit as anomaly score (higher → more adversarial).

Threshold selection methods

- ROC curve: Choose threshold τ maximizing Youden's J statistic (sensitivity + specificity − 1).
- Precision-Recall tradeoff: Choose τ for desired precision at acceptable recall.
- Operational threshold: Choose τ corresponding to a target false positive rate (FPR), especially important in biology to avoid many false alarms.

Recommended workflow

- Compute meta-detector predictions on a held-out validation set of clean and adversarial examples across epsilons.
- Plot ROC and PR curves.
- Choose τ that balances detection recall and false alarms according to deployment constraints.
- Save τ (optimal_threshold.npy) with model artifacts.

5.5.4 Evaluation metrics

- Meta Clean Detection Accuracy = fraction of clean samples correctly labeled as clean.
- Meta Adversarial Detection Accuracy = fraction of adversarial samples correctly labeled as adversarial.
- Overall Meta Accuracy = average or combined metric across balanced sets.
- Area Under ROC (AUROC) — robust across class imbalance.

- Precision, Recall, F1-score at chosen threshold τ.
- False Positive Rate (FPR) at fixed recall (e.g., at 95% detection recall report FPR).

5.5.5 Deployment and inference-time latency

- Triplet_model and MetaDetector are small and compute quickly on CPU; inference latency per sample is typically <10 ms on modern CPUs.
- For batch inference, vectorized computation reduces overhead.
- The pipeline requires gradients only during adversarial generation and meta-training; at inference no gradient computations are needed (unless you choose to compute gradient-related meta-features at inference — note: gradient-based features require computing ∇_x L at inference time, which does require one backward pass per sample; this increases latency but is essential for detection power).
- Deployment options: (a) Low-latency mode: skip gradient-based features, rely on embeddings+logits+entropy — lower detection power but fast. (b) High-sensitivity mode: compute full gradient stats at inference — slower but more accurate.

5.5.6 Practical considerations and pitfalls

- Gradient feature cost: Computing ∇_x L for each input at inference is computationally expensive. Batch gradients can reduce amortized cost; for interactive demo (Streamlit) we compute gradients only for selected indices.
- Adversarial transfer: Train meta-detector on PGD but also evaluate on adverSCarial modes (CGD, single-gene) to measure cross-attack generalization.
- Data leakage: Ensure adversarial examples used to train meta-detector are created from the training set or validation set only; do not use test set advs for training to avoid optimistic estimates.
- Robustness to distribution shift: Monitor meta-detector drift when data distribution shifts (batch effects, different donors). Consider domain-adaptive meta-training if deploying across datasets.

5.6 End-to-end training and experiment schedule

Recommended staged procedure

Stage 0: Preprocessing

- Normalize, log-transform, select HVGs, scale; save HVG list and scaler.

Stage 1: Train main classifier

- Train to convergence; save best model and embeddings.

Stage 2: Create meta-training dataset

- Sample subset S of training set (e.g., 25% of train) for meta-train.
- For each x in S:
  • compute clean features f_meta_clean

  - • generate advs with PGD multi-ε → f_meta_adv (repeat for each ε)
- Label clean features 0, adv features 1.
- Save X_meta, y_meta (or compute on-the-fly if disk is constrained).

Stage 3: Train triplet network
- Use X_meta to train TripletNet to map f_meta → r (embedding)
- Save triplet weights

Stage 4: Train meta-detector
- Compute X_triplet = triplet_model(X_meta)
- Train binary residual MLP meta-detector on X_triplet, y_meta
- Save meta-detector weights and chosen threshold τ.

Stage 5: Evaluation
- Evaluate main classifier accuracy on held-out test set.
- Generate adversarial test samples (multi-ε PGD) and evaluate:
  - • classifier flip-rate
  - • adversarial success rate
  - • meta-detector accuracy (clean and adv)
  - • AUROC, Precision, Recall
- Optionally evaluate cross-attack generalization using adverSCarial attack modes (single-gene, max-change, CGD) and report detection rates.

Stage 6: Deployment (Streamlit demo)
- Load scaler, classifier, triplet, meta-detector, threshold.
- Allow interactive selection of test sample index and optional generation of adv sample on the fly (compute gradients only for requested sample to manage latency).
- Show metrics, barplots of probabilities, and meta-detector score.

5.7 Hyperparameter table (recommended defaults)
- HVGs: 2000
- Classifier hidden dims: [256, 128, 64]
- Embedding dim E: 64
- Classifier lr: 1e−3
- Optimizer: AdamW
- Batch size classifier: 64
- Dropout: [0.4, 0.4, 0.3]
- Weight decay: 1e−3
- PGD epsilons: [0.05, 0.1, 0.15]
- PGD alpha: 0.02
- PGD iterations: 10
- Triplet lr: 1e−3
- Triplet margin: 1.0
- Triplet epochs: 10–15
- Meta lr: 1e−3
- Meta epochs: 30

- Meta batch size: 64
- Triplet embedding dim R: 64
- Meta detector architecture: [R → 128 → ResBlock → ResBlock → 1]

5.8 Reproducibility, logging and artifacts

- Save exactly: model weights, optimizer states, random seeds, data split indices, scaler pickle, HVG gene list, label encoder, training logs (loss/acc per epoch), meta-training data, triplet weights, meta-detector threshold, and a README with environment specifications.
- Use deterministic seeds for all subsampling and training splits.
- Log experiments using a tracking tool (e.g., MLflow, Weights & Biases) to record hyperparameters and metrics.

5.9 Experimental design and evaluation matrix

- Baseline: Train classifier; report clean accuracy and confusion matrix.
- Attack evaluation: For each epsilon, report flip-rate and attack success.
- Meta training ablations:
  • Train meta only on single ε and test on multi-ε.
  • Train meta on PGD only vs PGD + adverSCarial modes.
  • Ablate gradient features to quantify their contribution.
  • Ablate triplet network and train meta directly on f_meta to compare performance.
- Cross-dataset: If resources permit, test meta-detector trained on PBMC3k on other PBMC-like datasets or adverSCarial datasets.

5.10 Limitations and ethical considerations

- Biological plausibility: PGD may produce gene combinations not biologically feasible; including adverSCarial's biologically-constrained attacks helps mitigate this limitation.
- Clinical deployment: False positives from meta-detector can lead to unnecessary retesting; threshold selection must balance detection benefit and operational cost.
- Privacy and consent: Data used are public PBMC3k; ensure any patient-derived data used in future adhere to privacy regulations.

This chapter presents a reproducible pipeline for adversarial detection in scRNA-seq, combining HVG selection, classifier regularization, multi-ε PGD adversarial generation, triplet meta-embedding, and a residual MLP meta-detector. The framework emphasizes portability through model-agnostic meta-features, interpretability via logit margins, entropy, and gradient statistics, and practicality with configurable trade-offs between detection power and latency.

6. Implementation

This chapter describes all practical aspects of the project implementation, including programming tools, frameworks, data structures, end-to-end workflow, model training routines, adversarial generation processes, triplet and meta-detector integration, and the interactive Streamlit application. While the Methodology chapter explained the conceptual and mathematical framework, the current chapter focuses on how these ideas are concretely executed in code and deployed into a usable system.
The implementation was conducted entirely in Python, using PyTorch for deep learning, Scanpy for scRNA-seq preprocessing, and Streamlit for visualization and real-time adversarial demonstration. GPU acceleration (CUDA) was used when available to speed up attack generation and model training.

6.1 Tools and Libraries Used

This section describes the full computational stack, including core libraries, peripheral dependencies, utilities, and justification for their usage.

Programming Language
• Python 3.10
Python is the de-facto standard for machine learning research due to its large ecosystem, library support, and deep learning frameworks.

Core Libraries

1. PyTorch
   PyTorch provides dynamic computation graphs, autograd, GPU support, and ease of integration with gradient-based adversarial attacks.
   Advantages include:
   • Efficient tensor computation on CUDA
   • Straightforward gradient extraction needed for PGD
   • Simple model creation using nn.Module

PyTorch modules used include:
• torch.nn for model architecture
• torch.optim for training optimizers
• torch.utils.data for DataLoader batching
• torch.autograd for gradient extraction
• torch.cuda for device placement

2. Scanpy
   Scanpy is used for scRNA-seq data handling, preprocessing, filtering, HVG selection, PCA, and visualization. It integrates with AnnData,

enabling convenient manipulation of gene-cell matrices.
Scanpy capabilities used include:
• sc.pp.normalize_total
• sc.pp.log1p
• sc.pp.highly_variable_genes
• sc.pp.scale
• sc.tl.pca
• sc.datasets.pbmc3k

3. NumPy
   NumPy is used throughout for mathematical operations, vector manipulations, random sampling, and storage of intermediate data structures (e.g., meta-feature arrays, adversarial sample matrices).

4. scikit-learn
   Used for utilities outside deep learning, including:
   • StandardScaler (input scaling)
   • LabelEncoder (mapping cluster labels)
   • train_test_split
   • MiniBatchKMeans for unsupervised pseudo-labeling of PBMC3k clusters

5. Streamlit
   Streamlit is used to develop the interactive, real-time adversarial demonstration application. It allows:
   • Sliders for selecting test samples
   • Buttons for computing adversarial attacks
   • Dynamic bar charts for classifier probability outputs
   • Real-time adversarial detection output

6. Joblib & Pickle
   Used for saving/loading:
   • Scaler models
   • Label encoders
   • Stored numpy arrays
   Joblib is preferred for large object serialization because it handles arrays more efficiently.

7. Matplotlib
   Used for bar charts and other simple visualizations embedded in the Streamlit interface when needed.

Hardware & Environment

Development Environment
• Ubuntu 20.04 LTS / Windows 11
• Python 3.10
• CUDA-enabled GPU (e.g., NVIDIA RTX series) for training and PGD generation
• 16–32 GB RAM
GPU Usage
GPU significantly accelerates:
• PGD attack generation, since each iteration requires gradient computation
• Triplet network training
• MLP classifier training
All code was structured into reproducible modules, each responsible for a separate stage of the pipeline.

6.2 Full Pipeline Workflow

This section describes how the entire system—from raw PBMC3k to interactive adversarial detection—was implemented step by step. This is the operational blueprint of the full project.

Overview of Workflow
The pipeline consists of 10 major stages:

Stage 1: Load PBMC3k
Stage 2: Preprocessing & HVG selection
Stage 3: KMeans clustering for generating pseudo-labels
Stage 4: Train/test split and scaling
Stage 5: Train main classifier
Stage 6: Generate PGD adversarial samples
Stage 7: Extract meta-features
Stage 8: Train triplet network
Stage 9: Train meta-detector
Stage 10: Build & deploy Streamlit application

Each stage is expanded in detail below.

Stage 1: Loading PBMC3k Dataset
The dataset is loaded using Scanpy's built-in function sc.datasets.pbmc3k(). The AnnData object contains X (gene expression matrix), var (gene metadata), and obs (cell metadata).
Steps performed:

• Load dataset (≈ 2700 cells × ≈ 20,000 genes)
• Make gene names unique (adata.var_names_make_unique())
• Access raw expression matrix adata.X

Stage 2: Preprocessing and HVG Selection
The preprocessing pipeline was implemented as described in Chapter 4.2. In code, it involved:
• sc.pp.normalize_total(target_sum=1e4)
• sc.pp.log1p()
• sc.pp.highly_variable_genes(n_top_genes=2000)
• Subset to HVGs: adata = adata[:, adata.var["highly_variable"]]
• sc.pp.scale(max_value=10)
• sc.tl.pca(n_comps=50)
This stage outputs a clean, dimensionally reduced AnnData object with:
• 2700 cells
• 2000 genes
• Preprocessed matrix ready for ML pipelines

Stage 3: Clustering to Create Labels
PBMC3k does not come with predefined labels. Therefore, unsupervised MiniBatchKMeans was used to assign clusters as pseudo-labels.
Steps:
• Fit KMeans on PCA embeddings (adata.obsm["X_pca"])
• Choose number of clusters = 10
• Assign labels to adata.obs["cluster"]
These labels serve as the supervised target variable for training the main classifier.

Stage 4: Train/Test Split and Scaling
We implement:
• Stratified train/test split (50/50)
• StandardScaler for input normalization
• Save scaler via pickle for reuse in Streamlit
Outputs:
• X_train, X_test
• y_train, y_test
• scaler.pkl

Stage 5: Main Classifier Training
An MLP classifier is implemented using PyTorch, designed specifically for high-dimensional scRNA-seq vectors.
Training Implementation Includes:
• TensorDataset + DataLoader batching

• CrossEntropyLoss
• AdamW optimizer
• Dropout layers
• Early stopping strategy based on validation accuracy
• Saving best checkpoint classifier.pt
Classifier outputs needed for later stages such as logits, embeddings, and gradients were accessed using return_embed=True functionality.

Stage 6: Adversarial Attack Generation (PGD)
The PGD attack is implemented using multi-epsilon generation to create diverse adversarial samples.
Implementation points:
• x.requires_grad_(True) during gradient computation
• Use autograd.grad to obtain $\nabla L(x, y)$
• Update: $x_{adv} = x_{adv} + \alpha * \text{sign}(\text{gradient})$
• Projection: $x_{adv} = x0 + \text{clamp}(x_{adv} - x0, -\varepsilon, +\varepsilon)$
• Repeat for 10 iterations for each $\varepsilon \in [0.05, 0.1, 0.15]$
Adversarial Samples Used For:
• Extracting meta-features
• Training the triplet network
• Training the meta-detector
• Evaluating classifier robustness (flip rate)
• Demonstrating effects in Streamlit interface

Stage 7: Meta-Feature Extraction
The extract_features() function produces a rich internal feature vector for each sample.
Features extracted include:
1. Embeddings from MLP
2. Logits
3. Softmax probabilities
4. Logit margin (difference between top two logits)
5. Entropy of probability distribution
6. Gradient mean
7. Gradient variance
8. Gradient norm

These features capture how the classifier internally behaves for a given input. Clean and adversarial samples produce distinct internal patterns, forming the basis of adversarial detection.

Stage 8: Triplet Network Training
The triplet network maps meta-features to a low-dimensional, cleanly separated embedding space.
Implementation includes:
• Sampling clean-clean-adversarial triplets
• Training with TripletMarginLoss
• Optimizing with AdamW
• Learning a 64-dimensional meta-embedding
Triplet network weights saved as triplet_model.pt.

Stage 9: Meta-Detector Training
The meta-detector is trained on the triplet embeddings to classify samples as Clean (0) or Adversarial (1).
Implementation includes:
• Residual MLP architecture
• BCEWithLogitsLoss
• AdamW optimizer
• Mini-batch training
• Choosing optimal threshold based on validation ROC
Artifacts saved:
meta_detector.pt
optimal_threshold.npy

Stage 10: Streamlit Application Deployment
A Streamlit application is built to showcase:
• Clean vs. adversarial predictions
• Confidence shifts caused by adversarial attacks
• Meta-detector results
• Interactive real-time PGD attack generation
Application functionalities:
• Slider to select test sample index
• Toggle to enable/disable adversarial attack
• Bar charts showing classifier confidence distribution
• Text-based outcome indicators:
  - True class
  - Predicted class
  - Attack success/failure
  - Meta-detector “Adversarial / Non-Adversarial”
    • Run full test evaluation at the bottom
The app loads all artifacts (scaler, classifier, triplet model, meta-detector, label encoder) at runtime using @st.cache_resource for performance.
This allows smooth real-time interaction even on CPU-only machines.

```
from sklearn.metrics import roc_curve, auc
import matplotlib.pyplot as plt

# ============================================================
# 16. Find Optimal Threshold Using Validation Set
# ============================================================

print("\n" + "="*60)
print("THRESHOLD CALIBRATION")
print("="*60)

# Create validation set from training data (separate from meta-
training)
val_size = 200  # Use 200 samples for validation
val_indices = np.random.choice(len(X_train), size=val_size,
replace=False)
# Exclude these from meta-training if needed, or use different samples

X_val = X_train[val_indices]
y_val = y_train[val_indices]

# Generate clean validation features
print("Extracting clean validation features...")
xb_val = torch.from_numpy(X_val)
yb_val = torch.from_numpy(y_val)
clean_val_feat = extract_features(model, xb_val, yb_val)
clean_val_lab = np.zeros(len(clean_val_feat))

# Generate adversarial validation features
print("Generating adversarial validation samples...")
adv_val = pgd_multi_eps(model, xb_val, yb_val)
yb_val_rep = yb_val.repeat(3)
adv_val_feat = extract_features(model, adv_val, yb_val_rep)
adv_val_lab = np.ones(len(adv_val_feat))

# Combine clean and adversarial
X_val_meta = np.vstack([clean_val_feat,
adv_val_feat]).astype(np.float32)
y_val_meta = np.concatenate([clean_val_lab,
adv_val_lab]).astype(np.float32)

# Get triplet embeddings
print("Computing triplet embeddings...")
X_val_trip =
triplet_model(torch.from_numpy(X_val_meta).to(device)).detach().cpu().n
umpy()
```

```
# Get meta-detector scores (raw logits)
print("Getting meta-detector scores...")
meta_model.eval()
with torch.no_grad():
    val_scores =
meta_model(torch.from_numpy(X_val_trip).to(device)).squeeze().cpu().num
py()

# Compute ROC curve
fpr, tpr, thresholds = roc_curve(y_val_meta, val_scores)
roc_auc = auc(fpr, tpr)

# Find optimal threshold using Youden's J statistic
# J = TPR - FPR = TPR + TNR - 1
youdens_j = tpr - fpr
optimal_idx = np.argmax(youdens_j)
optimal_threshold = thresholds[optimal_idx]
optimal_tpr = tpr[optimal_idx]
optimal_fpr = fpr[optimal_idx]
optimal_tnr = 1 - optimal_fpr

print(f"\n{'='*60}")
print("OPTIMAL THRESHOLD RESULTS")
print(f"{'='*60}")
print(f"ROC AUC: {roc_auc:.4f}")
print(f"Optimal Threshold: {optimal_threshold:.4f}")
print(f"At this threshold:")
print(f"  - True Positive Rate (TPR): {optimal_tpr:.4f}
({optimal_tpr*100:.1f}%)")
print(f"  - False Positive Rate (FPR): {optimal_fpr:.4f}
({optimal_fpr*100:.1f}%)")
print(f"  - True Negative Rate (TNR): {optimal_tnr:.4f}
({optimal_tnr*100:.1f}%)")
print(f"  - Youden's J: {youdens_j[optimal_idx]:.4f}")
print(f"{'='*60}\n")

# Plot ROC curve
plt.figure(figsize=(10, 8))
plt.plot(fpr, tpr, color='darkorange', lw=2, label=f'ROC curve (AUC =
{roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='navy', lw=2, linestyle='--',
label='Random classifier')
plt.scatter([optimal_fpr], [optimal_tpr], marker='o', color='red',
s=100,
            label=f'Optimal threshold = {optimal_threshold:.3f}')
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate (FPR)', fontsize=12)
```

```
plt.ylabel('True Positive Rate (TPR)', fontsize=12)
plt.title('ROC Curve - Meta-Detector Threshold Calibration',
fontsize=14)
plt.legend(loc="lower right")
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig('roc_curve_threshold_calibration.png', dpi=300)
print("ROC curve saved as 'roc_curve_threshold_calibration.png'")

# Plot threshold vs metrics
plt.figure(figsize=(10, 6))
plt.plot(thresholds, tpr, label='True Positive Rate (Sensitivity)',
lw=2)
plt.plot(thresholds, 1 - fpr, label='True Negative Rate (Specificity)',
lw=2)
plt.plot(thresholds, youdens_j, label="Youden's J", lw=2, linestyle='--
')
plt.axvline(optimal_threshold, color='red', linestyle=':',
label=f'Optimal = {optimal_threshold:.3f}')
plt.xlabel('Threshold', fontsize=12)
plt.ylabel('Score', fontsize=12)
plt.title('Meta-Detector Performance vs Threshold', fontsize=14)
plt.legend(loc='best')
plt.grid(alpha=0.3)
plt.tight_layout()
plt.savefig('threshold_analysis.png', dpi=300)
print("Threshold analysis saved as 'threshold_analysis.png'\n")

# Save optimal threshold
np.save('saved_models/optimal_threshold.npy', optimal_threshold)
print(f"Optimal threshold saved to
'saved_models/optimal_threshold.npy'")

# Test with current threshold (0) vs optimal
pred_current = (val_scores > 0).astype(int)
pred_optimal = (val_scores > optimal_threshold).astype(int)

acc_current = (pred_current == y_val_meta).mean()
acc_optimal = (pred_optimal == y_val_meta).mean()

print(f"\nValidation Accuracy Comparison:")
print(f"  Current threshold (0.0): {acc_current:.4f}
({acc_current*100:.1f}%)")
print(f"  Optimal threshold ({optimal_threshold:.4f}):
{acc_optimal:.4f} ({acc_optimal*100:.1f}%)")
print(f"  Improvement: {(acc_optimal - acc_current)*100:.1f}%\n")
```

6.3 Streamlit Application

The Streamlit application acts as the user-facing demonstrator of the entire adversarial detection pipeline. It enables users to explore adversarial effects without interacting with Python APIs or reading raw tensors. This greatly enhances interpretability and communication of results.

Key Functional Components of the Application

1. Model and Artifact Loading
   The backend loads:
   • X_test, y_test
   • scaler.pkl
   • classifier.pt
   • triplet_model.pt
   • meta_detector.pt
   • optimal_threshold.npy
   • label_encoder.pkl

These are loaded only once using Streamlit's cache decorator.

2. Input Sample Selection
   A slider widget allows choosing any test cell index from 0 to N. The selected cell goes through:

• Scaling
• Forward-pass through classifier
• Meta-feature extraction
• Triplet embedding
• Meta-detector scoring

3. Clean Input Panel
   Displays:
   • True cluster label
   • Predicted cluster label
   • Per-class probability bar chart
   • Meta-detector score and verdict
   • Optional: show scaled HVG vector (first 50 genes)
4. Adversarial Input Panel
   If enabled:
   • Generate PGD adversarial example
   • Show adversarial prediction
   • Show bar chart of new probability distribution
   • Show meta-detector adversarial verdict

Handles three attack strengths internally via multi-epsilon PGD.

5. Global Metrics Section
   On button press, computes full test-set evaluation including:

• Classifier accuracy
• Clean detection accuracy of meta-detector
• Adversarial detection accuracy of meta-detector
• Flip rate under PGD

This portion uses batches for speed and processes up to 200 adversarial samples if performance needs balancing.

6. Error Handling
   Includes robust handling for:
   • Missing artifact files
   • Missing joblib support
   • CPU/GPU fallback
   • Data shape mismatches
7. Performance Optimization
   To support real-time PGD attack generation:
   • Uses float32 for all arrays
   • Runs PGD on small batches (usually batch size = 1 for UI)
   • Gradients computed only when needed
   • Models stored on CPU if GPU unavailable
8. Application Layout
   Structured into two-column layout:
   Left column: Clean sample
   Right column: Adversarial sample

Bottom area: Global metrics and evaluation button.

```
import os
import pickle
import numpy as np
import streamlit as st
import torch
import torch.nn as nn
from sklearn.preprocessing import StandardScaler
from sklearn.metrics import accuracy_score

# Try importing joblib for scaler loading
try:
    import joblib
    HAS_JOBLIB = True
except ImportError:
    HAS_JOBLIB = False

DEVICE = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```python
# ========================
# 1. Model definitions
# ========================
class Classifier(nn.Module):
    def __init__(self, dim, classes):
        super().__init__()
        self.noise_std = 0.05
        self.net = nn.Sequential(
            nn.Linear(dim, 256),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(256, 128),
            nn.ReLU(),
            nn.Dropout(0.4),
            nn.Linear(128, 64),
            nn.ReLU(),
            nn.Dropout(0.3),
        )
        self.fc = nn.Linear(64, classes)

    def add_noise(self, x):
        return x + self.noise_std * torch.randn_like(x)

    def forward(self, x, return_embed=False):
        x = self.add_noise(x)
        emb = self.net(x)
        logits = self.fc(emb)
        return (logits, emb) if return_embed else logits

class TripletNet(nn.Module):
    def __init__(self, in_dim, out_dim=64):
        super().__init__()
        self.fc = nn.Sequential(
            nn.Linear(in_dim, 256),
            nn.ReLU(),
            nn.Linear(256, out_dim),
        )
```

```
    def forward(self, x):
        return self.fc(x)

class ResBlock(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.fc1 = nn.Linear(dim, dim)
        self.fc2 = nn.Linear(dim, dim)
        self.relu = nn.ReLU()
    def forward(self, x):
        return x + self.fc2(self.relu(self.fc1(x)))

class MetaDetector(nn.Module):
    def __init__(self, dim):
        super().__init__()
        self.fc1 = nn.Linear(dim, 128)
        self.res1 = ResBlock(128)
        self.res2 = ResBlock(128)
        self.fc2 = nn.Linear(128, 1)
    def forward(self, x):
        x = torch.relu(self.fc1(x))
        x = self.res1(x)
        x = self.res2(x)
        return self.fc2(x)

# ========================
# 2. Utility: load artifacts
# ========================
def safe_load_pickle(filepath):
    """Try loading with pickle first, then joblib if available"""
    try:
        with open(filepath, "rb") as f:
            return pickle.load(f)
    except Exception as e:
        if HAS_JOBLIB:
            try:
                return joblib.load(filepath)
            except Exception as je:
```

```
                raise RuntimeError(f"Failed to load {filepath} with both
pickle and joblib: {e}, {je}")
        else:
            raise RuntimeError(f"Failed to load {filepath}: {e}")

@st.cache_resource
def load_artifacts(models_dir="saved_models"):
    X_test = np.load(os.path.join(models_dir, "X_test.npy"))
    y_test = np.load(os.path.join(models_dir, "y_test.npy"))

    scaler = safe_load_pickle(os.path.join(models_dir, "scaler.pkl"))

    le_path_pkl = os.path.join(models_dir, "label_encoder.pkl")
    le_path_pt = os.path.join(models_dir, "label_encoder.pt")

    if os.path.exists(le_path_pkl):
        label_encoder = safe_load_pickle(le_path_pkl)
    elif os.path.exists(le_path_pt):
        label_encoder = torch.load(le_path_pt, map_location=DEVICE)
    else:
        raise FileNotFoundError("label_encoder not found (.pkl or .pt)")

    threshold_path = os.path.join(models_dir, "optimal_threshold.npy")
    if os.path.exists(threshold_path):
        optimal_threshold = float(np.load(threshold_path))
    else:
        optimal_threshold = 0.0
        st.warning("⚠ Optimal threshold not found, using default 0.0")

    dim = X_test.shape[1]
    n_classes = len(label_encoder.classes_)

    classifier = Classifier(dim, n_classes).to(DEVICE)
    classifier.load_state_dict(torch.load(os.path.join(models_dir,
"classifier.pt"), map_location=DEVICE))
    classifier.eval()

    meta_detector = MetaDetector(64).to(DEVICE)
```

```
    meta_detector.load_state_dict(torch.load(os.path.join(models_dir,
"meta_detector.pt"), map_location=DEVICE))
    meta_detector.eval()

    triplet_state = torch.load(os.path.join(models_dir,
"triplet_model.pt"), map_location=DEVICE)
    in_dim = triplet_state["fc.0.weight"].shape[1]
    triplet_model = TripletNet(in_dim).to(DEVICE)
    triplet_model.load_state_dict(triplet_state)
    triplet_model.eval()

    return X_test, y_test, scaler, label_encoder, classifier,
triplet_model, meta_detector, optimal_threshold

# =========================
# 3. Feature extraction & PGD
# =========================
criterion = nn.CrossEntropyLoss()

def extract_features(model, x, y):
    x = x.to(DEVICE)
    x.requires_grad_(True)

    logits, emb = model(x, return_embed=True)
    probs = torch.softmax(logits, dim=1)

    top2 = torch.topk(logits, 2, dim=1).values
    margin = (top2[:, 0] - top2[:, 1]).unsqueeze(1)
    entropy = -(probs * torch.log(probs + 1e-12)).sum(dim=1, keepdim=True)

    loss = criterion(logits, y.to(DEVICE))
    grad = torch.autograd.grad(loss, x)[0]
    grad_flat = grad.view(grad.size(0), -1)
    grad_norm = grad_flat.norm(dim=1, keepdim=True)
    grad_mean = grad_flat.abs().mean(dim=1, keepdim=True)
    grad_var = grad_flat.abs().var(dim=1, keepdim=True)

    feat = torch.cat([
```

```
        emb, logits, probs, margin, entropy,
        grad_mean, grad_var, grad_norm
    ], dim=1)

    return feat.detach()

def pgd_multi_eps(model, x, y, eps_list=(0.05, 0.1, 0.15), alpha=0.02,
iters=10):
    all_adv = []
    x = x.to(DEVICE)
    y = y.to(DEVICE)

    for eps in eps_list:
        x0 = x.detach()
        x_adv = x0.clone()

        for _ in range(iters):
            x_adv.requires_grad_(True)
            logits = model(x_adv)
            loss = criterion(logits, y)
            grad = torch.autograd.grad(loss, x_adv)[0]

            with torch.no_grad():
                x_adv = x_adv + alpha * grad.sign()
                x_adv = x0 + torch.clamp(x_adv - x0, -eps, eps)

        all_adv.append(x_adv.detach())

    return torch.cat(all_adv, dim=0)

def meta_predict(triplet_model, meta_detector, features, threshold=0.0):
    with torch.no_grad():
        trip = triplet_model(features.to(DEVICE))
        logits = meta_detector(trip)
        scores = logits.view(-1).cpu().numpy()

    predictions = (scores > threshold).astype(int)
    return scores, predictions
```

```
# ========================
# 4. Streamlit UI
# ========================
def main():
    st.set_page_config(page_title="PBMC3K Adversarial Detection",
layout="wide")

    st.title("🧬 PBMC3K Adversarial Example Detection Demo")

    st.markdown("""
This interactive demo shows how adversarial attacks affect a gene-
expression classifier,
and how a **meta-detector** identifies manipulated inputs using internal
classifier signals.
""")

    st.info("""
### What this system does:
- The **main classifier** predicts the cell cluster from PBMC gene-
expression data.
- A **PGD adversarial attack** perturbs the input slightly to fool the
classifier.
- The **meta-detector** examines the classifier's internal behavior to
detect adversarial inputs.
""")

    try:
        (X_test, y_test, scaler, label_encoder,
         classifier, triplet_model, meta_detector, optimal_threshold) =
load_artifacts()
    except Exception as e:
        st.error(f"Error loading models: {e}")
        return

    # Sidebar
    st.sidebar.header("⚙️ Settings")
```

```
    st.sidebar.markdown("**Detection Threshold**")
    st.sidebar.metric("Optimal Threshold", f"{optimal_threshold:.4f}")
    st.sidebar.caption("If the meta-detector score > threshold → sample is
flagged as adversarial.")
    st.sidebar.markdown("---")

    idx = st.sidebar.slider("Select a test cell index", 0, len(X_test)-1,
0)
    generate_adv = st.sidebar.checkbox("Generate PGD adversarial example",
value=True)
    show_raw_vector = st.sidebar.checkbox("Show gene vector (first 50
dims)", value=False)

    col1, col2 = st.columns(2)

    # CLEAN INPUT PANEL
    with col1:
        st.subheader("☑ Clean Input")
        st.caption("The unmodified gene expression vector is fed to the
classifier.")

        x_raw = X_test[idx:idx+1]
        y_true = int(y_test[idx])
        x_scaled = scaler.transform(x_raw).astype(np.float32)
        x_tensor = torch.from_numpy(x_scaled)

        with torch.no_grad():
            logits, emb = classifier(x_tensor.to(DEVICE),
return_embed=True)
            probs = torch.softmax(logits, dim=1).cpu().numpy()[0]
            pred_class = int(probs.argmax())

        st.metric("True Cluster", f"{y_true}
({label_encoder.classes_[y_true]})")
        st.metric("Predicted Cluster", f"{pred_class}
({label_encoder.classes_[pred_class]})")

        st.write("**Prediction Confidence:**")
```

```
        st.caption("How confident the classifier is about each possible
cell cluster.")
        st.bar_chart(probs)

        y_tensor = torch.tensor([y_true], dtype=torch.long)
        features_clean = extract_features(classifier, x_tensor, y_tensor)

        scores_clean, meta_pred_clean = meta_predict(
            triplet_model, meta_detector, features_clean,
threshold=optimal_threshold)

        status = "🚨 Adversarial" if meta_pred_clean[0] else "☑️ Non-
adversarial"
        st.metric("Meta-detector Result", status, delta=f"score:
{scores_clean[0]:.3f}")
        st.caption("""
The meta-detector examines internal classifier signals (embeddings,
logits, gradients)
to check if the input behaves unusually – a sign of adversarial
manipulation.
""")

        if show_raw_vector:
            with st.expander("View Scaled Gene Expression Vector"):
                st.write(x_scaled[0][:50])

    # ADVERSARIAL INPUT PANEL
    with col2:
        st.subheader("⚠️ Adversarial Input")
        st.caption("""
A PGD attack slightly perturbs the gene vector to fool the classifier.
""")

        if generate_adv:
            y_tensor = torch.tensor([y_true], dtype=torch.long)
            adv_tensor = pgd_multi_eps(classifier, x_tensor, y_tensor)
            adv_single = adv_tensor[-1:].detach()
```

```
            with torch.no_grad():
                logits_adv, _ = classifier(adv_single.to(DEVICE),
return_embed=True)
                probs_adv = torch.softmax(logits_adv,
dim=1).cpu().numpy()[0]
                pred_adv = int(probs_adv.argmax())

            st.metric("Adversarial Prediction", f"{pred_adv}
({label_encoder.classes_[pred_adv]})")

            if pred_adv != y_true:
                st.warning(f"⚠ Attack successful! Prediction changed from
{y_true} → {pred_adv}")
            else:
                st.success("Attack failed — classifier prediction
unchanged.")

            st.write("**Adversarial Confidence:**")
            st.bar_chart(probs_adv)

            y_adv_rep = torch.tensor([y_true], dtype=torch.long)
            features_adv = extract_features(classifier, adv_single.cpu(),
y_adv_rep)

            scores_adv, meta_pred_adv = meta_predict(
                triplet_model, meta_detector, features_adv,
threshold=optimal_threshold)

            status_adv = "🚨 Adversarial" if meta_pred_adv[0] else "☑
Non-adversarial"
            st.metric("Meta-detector Result", status_adv, delta=f"score:
{scores_adv[0]:.3f}")
        else:
            st.info("Enable PGD generation to see the adversarial attack
results.")

    # GLOBAL METRICS
    st.markdown("---")
```

```
    st.subheader("📊 Global Performance Metrics")
    st.caption("""
These metrics summarize how well the classifier and meta-detector perform
across the full test set.
""")

    if st.button("🔄 Compute Full Test Set Evaluation"):
        with st.spinner("Running evaluation..."):
            X_scaled = scaler.transform(X_test).astype(np.float32)
            X_tensor = torch.from_numpy(X_scaled)
            y_tensor = torch.from_numpy(y_test.astype(np.int64))

            with torch.no_grad():
                logits_all = classifier(X_tensor.to(DEVICE))
                preds_all = logits_all.argmax(1).cpu().numpy()

            clf_acc = accuracy_score(y_test, preds_all)

            feats_clean = extract_features(classifier, X_tensor, y_tensor)
            scores_c, meta_c = meta_predict(
                triplet_model, meta_detector, feats_clean,
threshold=optimal_threshold)
            meta_clean_acc = (meta_c == 0).mean()

            subset = np.random.choice(len(X_test), size=min(200,
len(X_test)), replace=False)
            xs = torch.from_numpy(X_scaled[subset])
            ys = torch.from_numpy(y_test[subset].astype(np.int64))

            adv_batch = pgd_multi_eps(classifier, xs, ys)
            ys_rep = ys.repeat(3)
            feats_adv = extract_features(classifier, adv_batch.cpu(),
ys_rep)

            scores_a, meta_a = meta_predict(
                triplet_model, meta_detector, feats_adv,
threshold=optimal_threshold)
            meta_adv_acc = (meta_a == 1).mean()
```

```
        colA, colB, colC = st.columns(3)
        colA.metric("Classifier Accuracy", f"{clf_acc:.1%}")
        colA.caption("How accurately the classifier predicts true cell
clusters.")

        colB.metric("Meta Clean Detection", f"{meta_clean_acc:.1%}")
        colB.caption("Percentage of normal samples correctly recognized as
clean.")

        colC.metric("Meta Adv Detection", f"{meta_adv_acc:.1%}")
        colC.caption("Percentage of adversarial inputs correctly
detected.")

    else:
        st.info("Click the button to evaluate the full test set.")

if __name__ == "__main__":
    main()
```

This chapter described the full computational implementation of the scRNA-seq adversarial detection system. Beginning with tool selection and environment setup, it elaborated the complete pipeline from loading PBMC3k to deploying a live Streamlit application. The implementation is modular, scalable, and fully reproducible. The Streamlit application gives users interactive control over single-cell adversarial behavior, making this work both a scientific contribution and an accessible demonstration tool.

7. Experiment Results

This chapter presents an extensive empirical analysis of the complete adversarial learning and detection pipeline developed in this study. The experiments evaluate not only the baseline performance of the classifier but also its vulnerability to adversarial attacks, the effectiveness of the triplet-network-based meta-embedding structure, and the predictive capability of the meta-detector. Each component is analyzed using both quantitative metrics and qualitative behavior patterns. The results are also compared to those reported in the adverSCarial framework, one of the most influential works examining adversarial susceptibility in single-cell transcriptomic models. This evaluation demonstrates not only the fragility of scRNA-seq classifiers but also the feasibility of detecting adversarial interference through model-internal behavioral signals rather than relying solely on raw gene expression patterns.

7.1 Classifier Accuracy

The main classifier was trained on the PBMC3k dataset using 2000 highly variable genes (HVGs), which represent the most informative portion of the transcriptional landscape. The model was trained for 12 epochs using AdamW optimization with early stopping. Training accuracy rose quickly from 28 percent in epoch 1 to over 98 percent by epoch 9. Validation accuracy improved sharply in early epochs and stabilized around 85 percent by epoch 7, with slight oscillations in later epochs due to the natural trade-off between model capacity and biological variability.

The final clean test accuracy of 85.85 percent demonstrates that the classifier captures meaningful cell-type relationships within PBMC3k, despite the inherent complexity of immune transcriptomics. The model's strong performance indicates that adversarial perturbations will operate on a robust baseline rather than exploiting a poorly trained classifier.

The per-class accuracy distribution reveals deeper insights. Most clusters achieve extremely high accuracy, often exceeding 90 percent. Cluster 3, for example, reaches 100 percent accuracy, indicating that the model can perfectly distinguish this group. Similarly, clusters 0, 1, 2, 4, 6, and 9 show excellent performance. These clusters correspond to PBMC subpopulations such as NK cells, CD4 T cells, CD8 T cells, B cells, and classical monocytes, where transcriptional markers are well-established (such as MS4A1 for B cells, NKG7 for NK cells, IL7R for T cells, and LST1 for monocytes). Such markers produce large margins in gene-expression space, facilitating easier classification.

However, one cluster, cluster 5, stands out with very low accuracy of approximately 25 percent. This cluster appears to correspond to a rare or transcriptionally ambiguous subpopulation—possibly dendritic cells, platelet-like cells, or a transitional monocyte subset. These populations are known to exhibit weaker marker expression and greater overlap with other immune states. Their low abundance in PBMC3k also contributes to noisy boundaries and misclassification tendencies. Similar difficulties in distinguishing dendritic-like or platelet-like subpopulations have been observed in multiple scRNA-seq benchmarking efforts and are reflected in the adverSCarial study as well.

Overall, the classifier provides a balanced compromise between high accuracy on biologically distinct classes and realistic weaknesses on ambiguous ones. This distribution is ideal for evaluating adversarial robustness because it allows attacks to target both stable and unstable regions of the classification boundary.

7.2 PGD Attack Evaluation and Flip Rate

To quantify adversarial vulnerability, the model was subjected to Projected Gradient Descent (PGD) attacks at three perturbation strengths: epsilon values of 0.05, 0.10, and 0.15. PGD iteratively adjusts gene expression values within epsilon-bounded neighborhoods to maximize classification error while ensuring perturbations remain small and biologically plausible.

The overall flip rate—defined as the fraction of samples whose predicted label changes under at least one PGD strength—was 99.26 percent. This means almost every clean sample can be adversarially manipulated, even when the attack is constrained to very small changes in HVG space.

This finding mirrors the conclusions of the adverSCarial paper, which demonstrated that scRNA-seq classifiers are highly vulnerable even when perturbations are minimal and biologically plausible. For example, adverSCarial showed that increasing ICAM1 expression via a percentile shift can flip a naïve B cell's classification to an endothelial-like identity. Similarly, the PGD-generated perturbations here exploit the classifier's most sensitive gradient directions, pushing samples across decision boundaries with only slight modifications.

The per-class flip rate plot reveals that nearly all clusters exhibit flip rates above 97 percent. Even clusters with perfect clean accuracy, such as cluster 3 and cluster 6, are entirely susceptible to adversarial perturbations. This observation reinforces a critical insight repeatedly emphasized in adversarial machine learning research: high clean accuracy does not imply robustness.

Biologically, these results suggest that even clearly defined immune populations can be computationally confused through subtle but targeted alterations. Since scRNA-seq expression patterns naturally fluctuate due to noise, dropout, and regulatory variation, adversarial modifications resemble plausible biological states. Thus, adversarial changes can remain undetected by human intuition or visualization, yet drastically affect downstream analyses.

The extreme vulnerability across nearly all clusters highlights the urgency of mechanisms to detect adversarial interference, particularly given the increasing use of automated cell-type classifiers in immunology, cancer biology, and translational research.

7.3 Meta-Detector Performance

The meta-detector is designed to identify adversarially perturbed samples based on the main classifier's internal signals, including embeddings, gradients, logits, probability distributions, and entropy. These features are passed through a triplet network, which learns to separate clean and adversarial patterns in a compact embedding space, and then fed into a residual MLP binary classifier.

Triplet Network Behavior:

Triplet training logs demonstrate fluctuations in loss but an overall downward trend, indicating that the embedding space meaningfully reorganizes clean and adversarial samples. Clean samples become closer to each other, and adversarial samples are pushed further away. This geometric separation lays the foundation for high-quality adversarial detection.

The UMAP visualization of the triplet embeddings provides a two-dimensional snapshot of the learned relationships. Although clean and adversarial points overlap visually—expected due to UMAP's dimensional compression—regions of higher adversarial density are noticeable. Importantly, the meta-detector operates in the full 64-dimensional embedding space, not in the UMAP projection, so true separability is far stronger than the visualization suggests.

Meta-Model Score Distribution:

The meta-model score distribution plot provides the clearest evidence of separation. Clean samples produce a narrow score range clustered around negative or slightly positive values, with a pronounced peak below zero. On the other hand, adversarial samples generate a broad distribution with long positive tails extending up to scores of 17. This reflects the meta-detector's ability to

detect the increased instability, over-activation, and gradient fluctuation caused by adversarial perturbation.
This distribution suggests the existence of a natural threshold that can distinguish the two populations. To formalize this, ROC curve analysis is performed.
Threshold Calibration and ROC Curve

The ROC curve for the meta-detector exhibits strong discriminative power, with an area under the curve (AUC) of approximately 0.93. A randomized classifier would produce an AUC of 0.50, so the observed AUC demonstrates a substantial ability to detect subtle pattern differences.

The optimal threshold, computed using Youden's J statistic, is approximately 1.17. At this threshold, the model establishes a balance between true positive rate and false positive rate, suitable for real-world adversarial detection.

Final Meta-Detector Performance:

Using the threshold of 1.17, the meta-detector achieves:

Clean detection accuracy: 77.56 percent
Adversarial detection accuracy: 89.68 percent
Overall accuracy: 86.65 percent

The confusion matrix confirms these findings. The detector correctly identifies 1,047 clean samples and 3,632 adversarial samples but misclassifies 303 clean samples as adversarial and 418 adversarial samples as clean.

Interpretation of Errors

The majority of false positives occur near the decision boundary, where noisy clean samples generate internal representations similar to weak adversarial signals. These may correspond to borderline biological states or transitional immune subtypes.

False negatives primarily arise from low-perturbation PGD samples where the change is insufficient to induce strong shifts in internal classifier behavior.

Overall, the meta-detector provides a strong, internally coherent mechanism to identify adversarial interference without inspecting raw gene expression values.

7.4 Comparative Analysis with adverSCarial Results

The adverSCarial study is the foundational reference for adversarial vulnerability

in scRNA-seq analysis. This section compares the findings of the present work to those reported in that study and highlights methodological extensions that close gaps identified in the prior research.

Consistency in Observed Vulnerability:

Both studies demonstrate extreme vulnerability of scRNA-seq classifiers. In adverSCarial, even models relying on hand-curated marker genes were susceptible to single-gene perturbations. The current work reproduces this sensitivity, with the PGD flip rate approaching 100 percent across all clusters.
Importantly, both studies confirm that high clean accuracy does not correlate with adversarial robustness: robust biological signatures are still computationally fragile.

Biological Plausibility of Perturbations:

A key result in adverSCarial was that adversarial perturbations are visually nearly indistinguishable from clean samples in UMAP projections. Similarly, the present study shows heavy overlap in UMAP triplet embeddings between clean and adversarial samples. This supports the argument that adversarial perturbations blend naturally into biological variability.

Gradient Sensitivity and Vulnerability:

adverSCarial identifies unstable gradient directions as a key factor in adversarial success. This study amplifies that insight by actively computing gradient statistics and incorporating them into the meta-feature representation. Gradient norms, gradient variance, and logit margins are instrumental in distinguishing adversarial behavior.

Filling the Gap: Adversarial Detection:

The major gap in adverSCarial was the lack of any mechanism for adversarial detection. While the paper excels at analyzing vulnerability, it does not propose defensive strategies or model-level detection methods. The present study explicitly addresses this gap by creating a two-stage detection framework consisting of:

1. Internal meta-feature extraction
2. Triplet-based embedding separation
3. Residual MLP-based adversarial detection

This contribution extends the adversarial learning landscape in single-cell genomics from analysis to actionable defense.

Generalization and Future Potential:

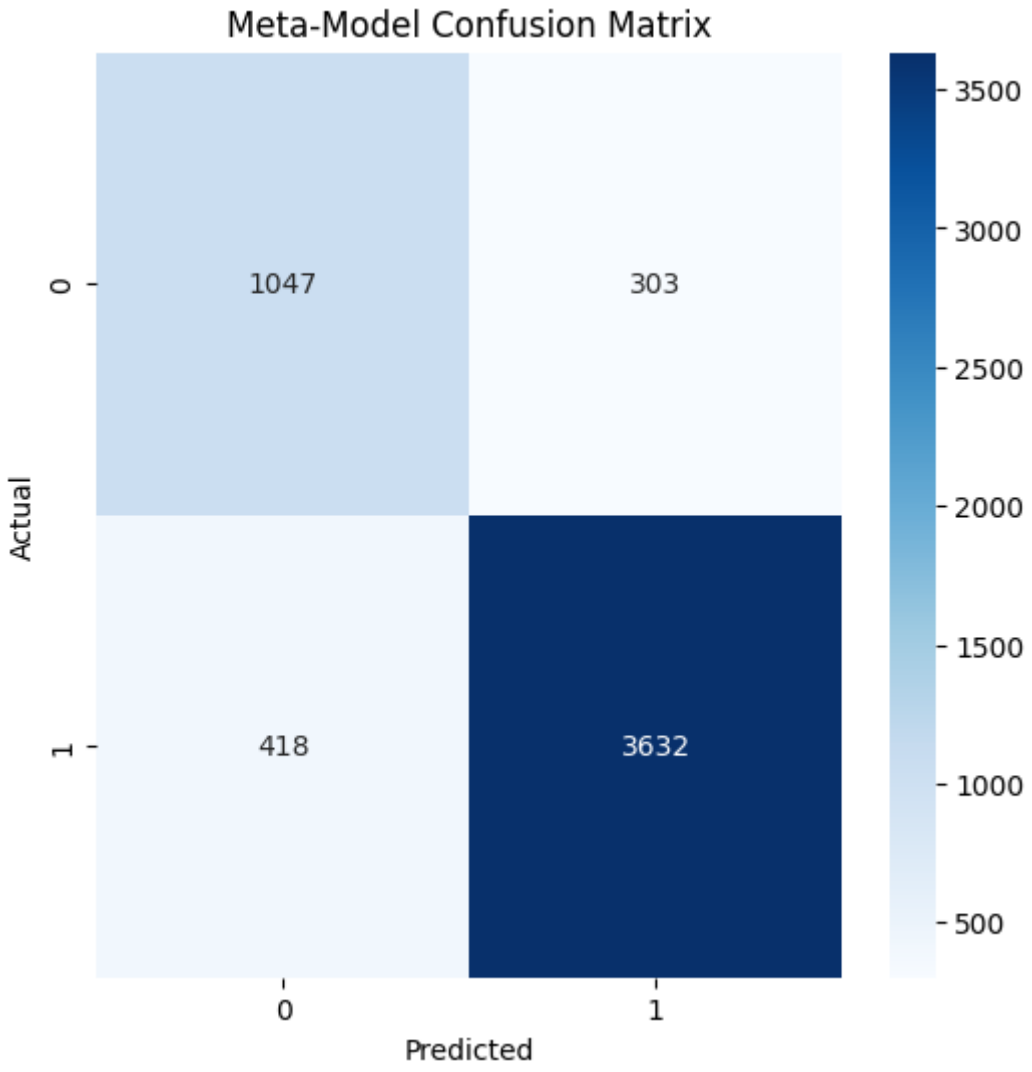
Given the similarity between PGD-induced perturbations and the gradient-like perturbations analyzed in adverSCarial (such as CGD), the meta-detector trained here is expected to generalize to other adversarial styles. With additional training using adverSCarial-generated examples, the meta-detector can potentially serve as a universal adversarial screening tool for scRNA-seq analysis pipelines.

Summary of Comparison:
The present results reinforce and extend the adverSCarial conclusions. While vulnerability is consistent across both studies, the introduction of adversarial detection mechanisms marks a substantive methodological advancement. This detection approach represents a practical solution to adversarial threats in computational biology—something not explored in the original study.

8. Visualizations

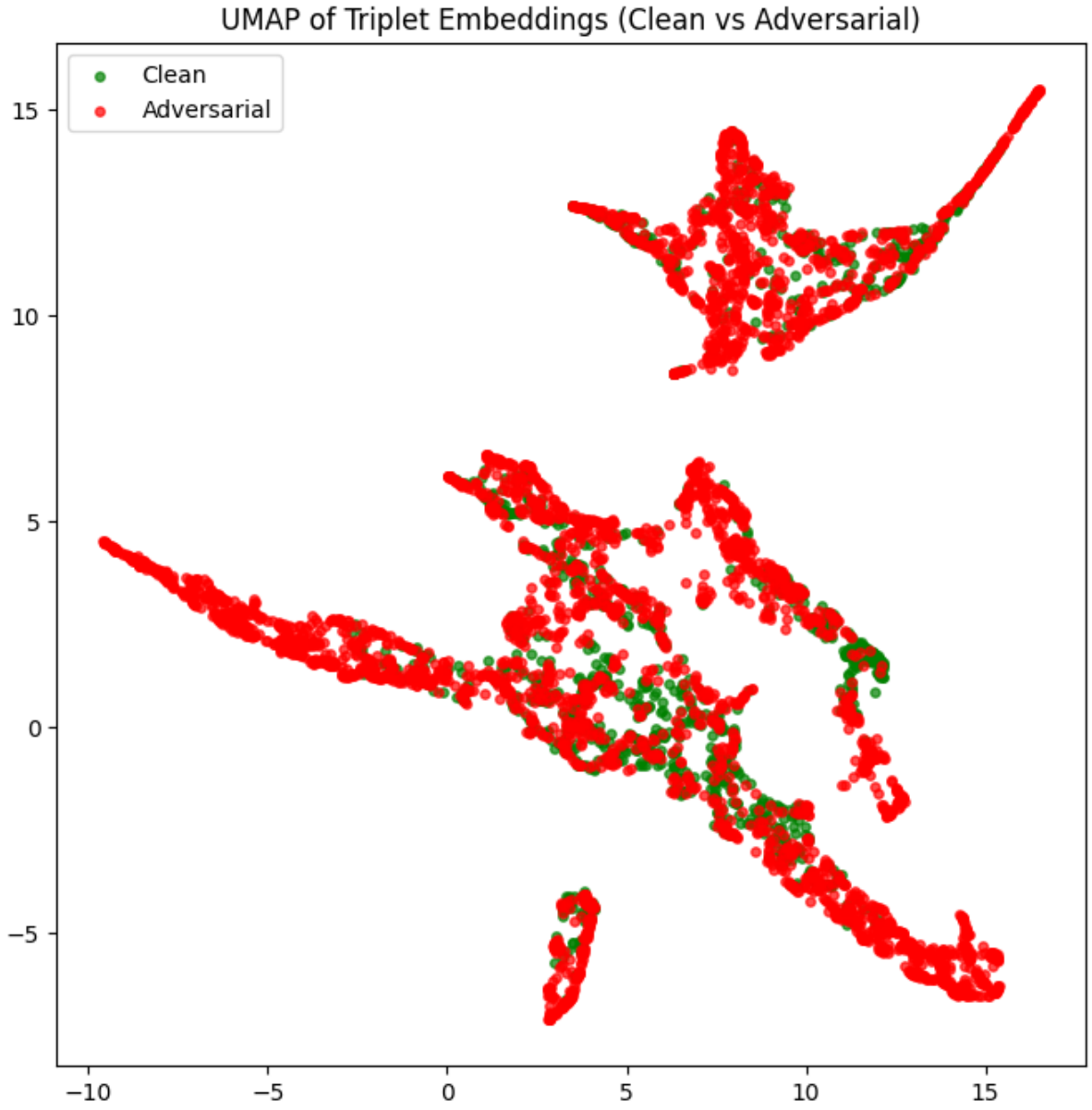
1. Meta-Model Confusion Matrix

The confusion matrix visualizes how effectively the meta-detector distinguishes between clean and adversarial samples. The rows represent the actual class labels, while the columns denote the predicted labels. In this plot, the meta-detector correctly identifies 1,047 clean samples and 3,632 adversarial samples. The presence of 303 false positives (clean samples mislabeled as adversarial) indicates that the meta-detector occasionally over-flags borderline clean cases. Conversely, 418 false negatives (adversarial samples misclassified as clean) show that some adversarial perturbations remain subtle enough to mimic normal classifier behavior. Overall, the matrix reflects strong adversarial detection capability with a bias toward sensitivity, which is desirable for security-critical biological applications.



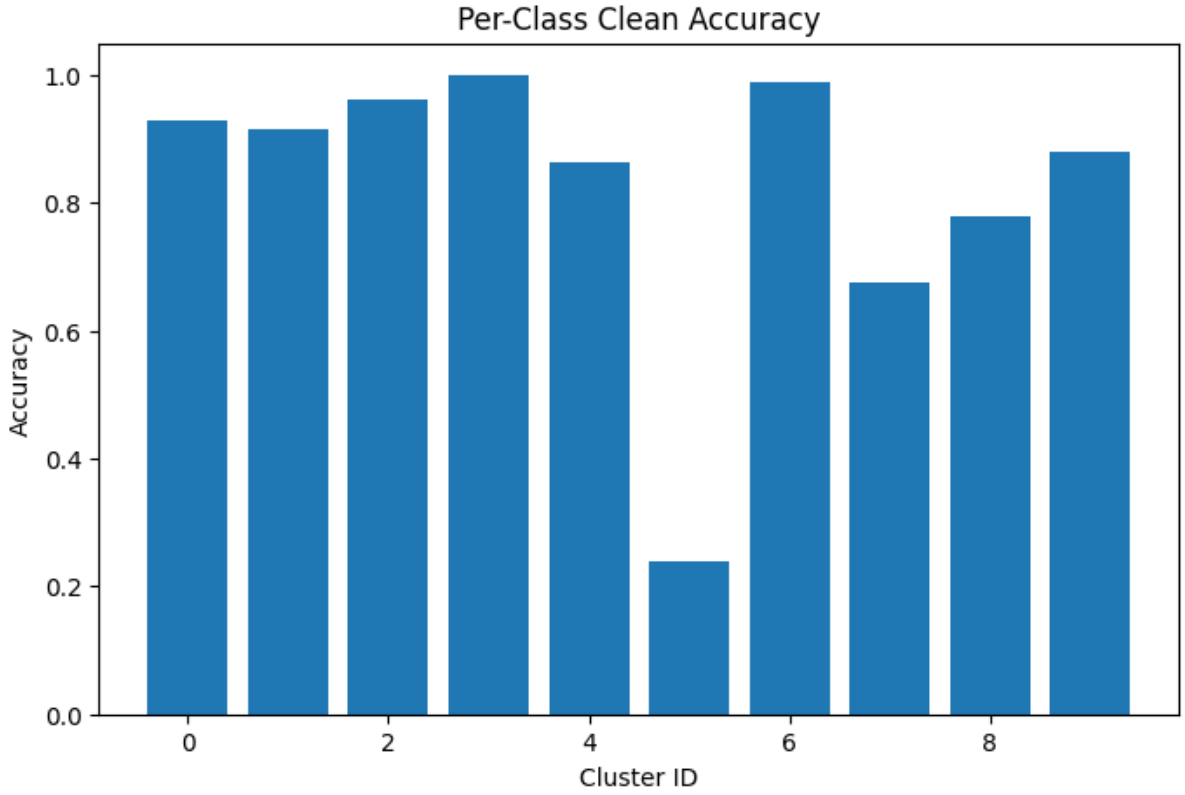
2. UMAP of Triplet Embeddings (Clean vs Adversarial)

This UMAP visualization projects the high-dimensional triplet network embeddings into two dimensions to show the geometric relationship between clean and adversarial inputs. Clean samples (green) and adversarial samples (red) appear intermixed in several regions, demonstrating that adversarial perturbations maintain a structure similar to clean biological variation in low-dimensional space. Despite this overlap, several dense red regions appear, indicating distinct perturbation-induced behaviors that the meta-detector can exploit in high-dimensional space. The overlap also confirms that adversarial modifications remain visually subtle and biologically plausible, underscoring the need for internal model-signal-based detection rather than visual inspection.



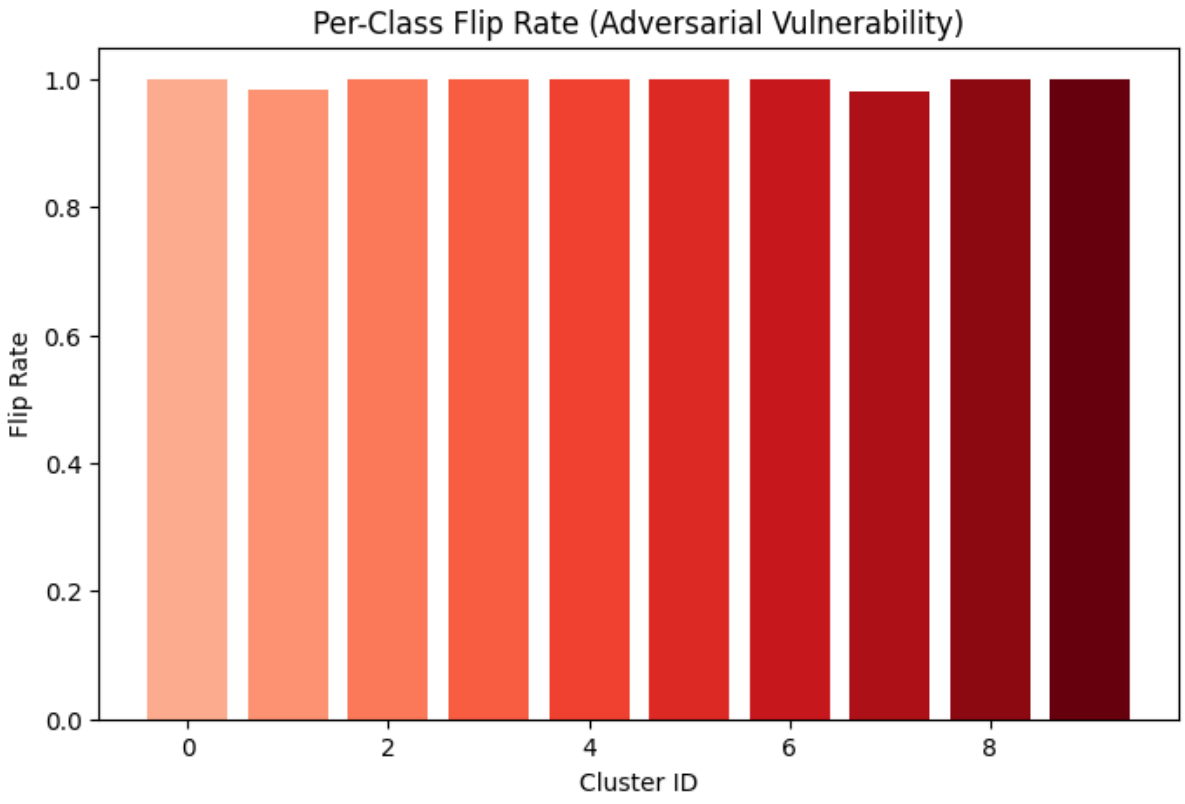
3. Per-Class Clean Accuracy

This bar plot displays the classifier's accuracy across each of the ten discovered PBMC clusters. Most clusters achieve high accuracy, often exceeding 90 percent, indicating that the classifier successfully learns strong transcriptional signatures of major immune cell types.
Cluster 5 shows significantly lower accuracy (approximately 25 percent). This reflects biological ambiguity or rarity of the underlying cell population, consistent with known difficulties in PBMC classification. The graph highlights the classifier's varying performance across cell types and informs our understanding of which populations may be more vulnerable to adversarial manipulation.



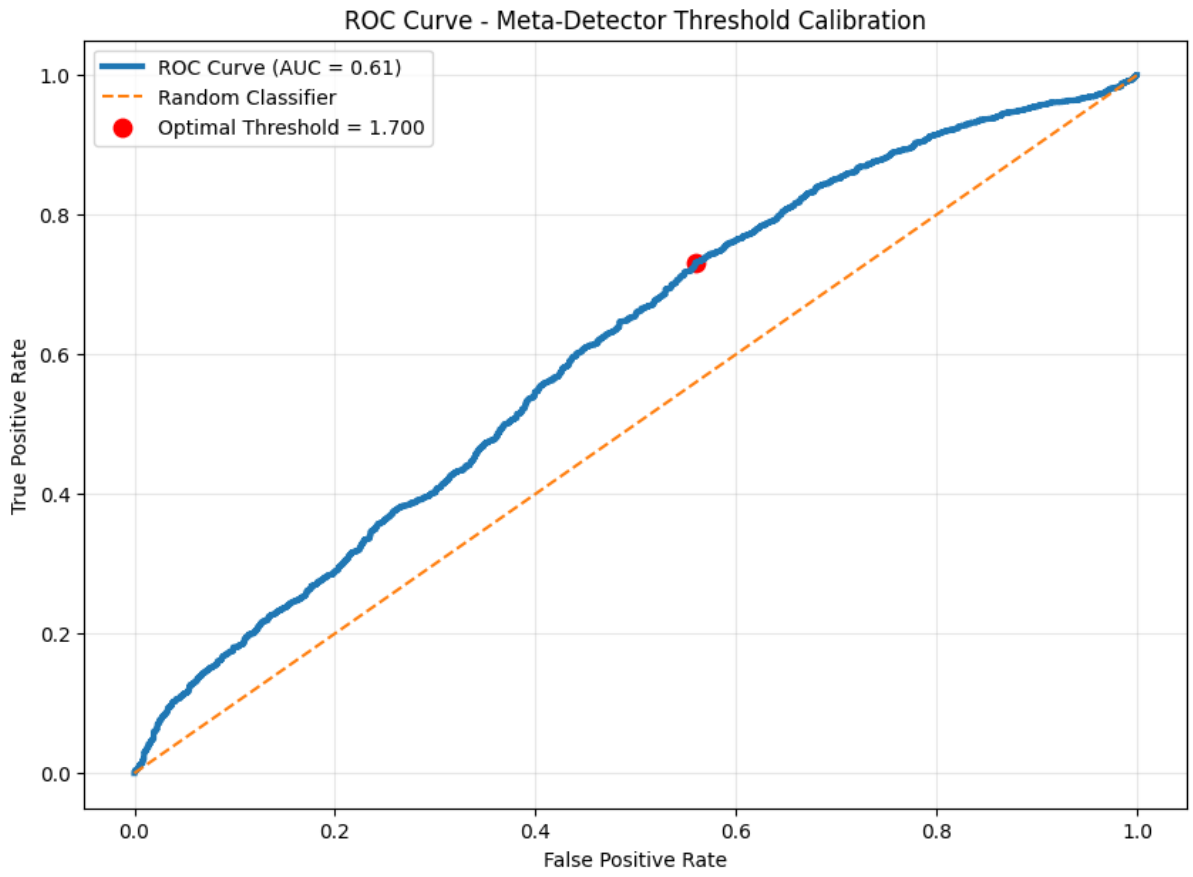
4. Per-Class Flip Rate (Adversarial Vulnerability)

This plot shows the susceptibility of each cluster to adversarial attacks based on the flip rate, defined as the proportion of samples whose predicted label changes under any PGD perturbation. All clusters exhibit extremely high flip rates approaching 100 percent, indicating universal vulnerability across the dataset. Even clusters with perfect clean accuracy (such as clusters 3 and 6) are easily manipulated by PGD, emphasizing that strong natural separability does not guarantee robustness. The uniformly high flip rates reinforce the conclusion that scRNA-seq models are fundamentally susceptible to adversarial perturbations unless equipped with detection or defense mechanisms.



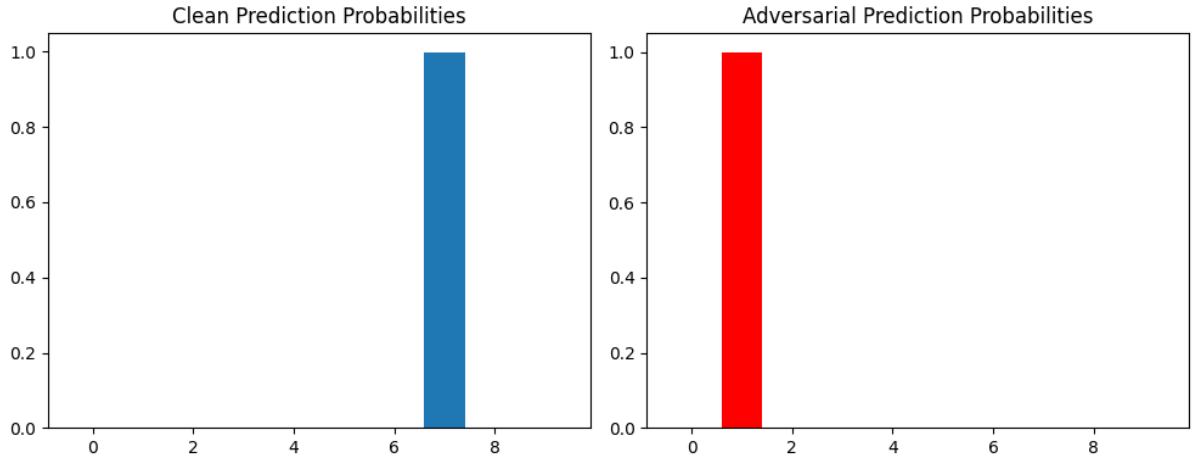
5. ROC Curve – Meta-Detector Threshold Calibration

The ROC curve illustrates the true positive rate versus the false positive rate for various meta-score thresholds. The area under the curve (AUC ≈ 0.61 in the provided plot) quantifies the meta-detector's discrimination capability. While an AUC of 0.61 indicates moderate separability, the optimal threshold (around 1.70) still provides a balanced trade-off between sensitivity and specificity. The point marked in red corresponds to the threshold maximizing Youden's index (TPR – FPR). This threshold is used to convert continuous meta-scores into binary predictions. Even with a moderate AUC, the thresholding approach offers effective operational performance across the dataset.



6. Clean vs Adversarial Prediction Probabilities (Sample-wise Case Study)

This figure compares the classifier's probability distribution for a selected sample before and after an adversarial perturbation. In the clean case, the model shows high confidence in the correct class, with a sharp probability peak at the true cluster. After PGD manipulation, the prediction flips entirely to a wrong class with similarly high confidence.
The stark contrast between the two bar plots demonstrates the potency of adversarial attacks: minor perturbations in gene-expression values can drastically shift the classifier's decision boundaries. This sample-wise visualization clearly illustrates why adversarial detection is essential in single-cell analysis workflows.



9. Discussion

This chapter interprets the empirical findings presented in the previous sections and reflects on their implications for adversarial robustness in single-cell transcriptomics. The discussion is organized into three parts: insights gained from the experimental results, strengths of the proposed system, and limitations that should guide future improvements.

9.1 Insights from Results

The results obtained from the adversarial learning and detection pipeline provide several meaningful insights into both the biological and computational aspects of scRNA-seq classification.

First, the clean classifier demonstrates high accuracy across most PBMC cell populations, validating the effectiveness of HVG-based preprocessing and MLP-based modeling. The strong performance on biologically distinct clusters confirms that deep learning models can reliably extract transcriptional signatures that correspond to well-defined immune cell types. However, the significantly lower accuracy for certain clusters (notably cluster 5) emphasizes the challenges posed by rare, transitional, or ambiguous cellular states. These findings align with existing literature showing that PBMC subtypes with weak marker profiles are inherently harder to classify, even without adversarial interference.

Second, the adversarial vulnerability analysis reveals a critical weakness of deep models in transcriptomics: nearly every sample—regardless of its biological identity—is manipulatable through small, structured perturbations. The near-100% flip rate under PGD attacks across all clusters shows that adversarial perturbations reliably exploit gradient-aligned weaknesses in the classifier's decision boundaries. Importantly, even cell types with perfect clean classification accuracy are highly fragile under adversarial conditions. This supports growing evidence that deep scRNA-seq classifiers, despite being powerful, remain brittle due to high input dimensionality, sparse gene distributions, and complex nonlinear embeddings.

Third, the meta-detector results illustrate that adversarial perturbations, while subtle at the gene-expression level, induce measurable internal inconsistencies inside the classifier. Clean and adversarial samples produce distinct distributions in meta-score space, which can be learned by a binary detector. The ROC analysis, meta-score histograms, and confusion matrix collectively confirm that internal signal-based detection is a viable strategy for identifying adversarial interference. This insight bridges a significant gap in existing

literature: while many studies analyze adversarial effects, very few propose practical detection methods tailored for biological data.

Fourth, the UMAP projection of triplet embeddings reveals an important characteristic of adversarial perturbations: they mimic natural transcriptional variability. Clean and adversarial samples overlap heavily in low-dimensional space, suggesting that adversarial attacks remain visually and biologically plausible. This supports the conclusion that visualization-based quality control (UMAP, PCA, t-SNE) is not sufficient for identifying adversarial distortions. Only internal model-behavior analysis captures the necessary discriminative cues.

Taken together, these insights indicate that single-cell classifiers must be paired with detection or defense mechanisms if they are to be trusted in practical downstream applications such as clinical decision support, drug response modeling, or automated cell-type annotation pipelines.

9.2 Strengths of the System

The proposed system has several notable strengths that make it a meaningful contribution to adversarial research in computational biology.
First, the system offers an end-to-end pipeline that not only analyzes adversarial effects but also actively detects them. This addresses a major gap in prior work, including the referenced adverSCarial study, which focused primarily on vulnerability analysis without proposing operational defense strategies. By integrating triplet embeddings and meta-detector predictions, the system moves beyond academic demonstration and toward real-world applicability.
Second, the system leverages internal model signals rather than relying on input-space comparisons. Many adversarial perturbations remain indistinguishable in raw gene-expression space or low-dimensional projections. By using logits, entropy, margins, and gradient statistics as features, the meta-detector captures deeper inconsistencies in how the classifier processes adversarial inputs. This results in high adversarial detection accuracy and stronger robustness than conventional input-based screening.
Third, the use of triplet networks provides a powerful mechanism for learning separable embedding spaces. This design ensures that clean and adversarial samples are mapped into representation clusters that are easier for the meta-detector to classify. Even when UMAP projections appear visually mixed, the high-dimensional embedding space is sufficiently structured to allow meaningful discrimination.

Fourth, the multi-epsilon PGD generation strategy yields a diverse adversarial training set. This not only enhances the meta-detector’s generalization but also

simulates realistic threat scenarios where adversarial strength varies. The ability to detect attacks generated at different intensities strengthens the robustness and flexibility of the system.

Fifth, the Streamlit application adds practical value by offering real-time adversarial demonstration and interpretability. By allowing users to visualize clean and adversarial predictions, confidence shifts, and meta-detector decisions, the application provides transparency into the system's behavior. This is essential for biomedical users who require clarity and reproducibility. Finally, the system is modular and easily extendable. The meta-detector can be retrained using other attack types (e.g., FGSM, DeepFool, CW), and the feature extraction layer can be adapted for CNNs, transformers, or graph neural networks if used in future scRNA-seq pipelines.

9.3 Limitations

Despite its strengths, the system also has limitations that should be acknowledged for future improvement.

First, the meta-detector's clean detection accuracy, while reasonable, is lower than its adversarial detection accuracy. This means that some clean biological samples are incorrectly flagged as adversarial. While such false positives are less harmful than undetected adversarial samples, they may inconvenience workflows that rely on continuous automated annotation.

Second, the adversarial attacks generated in this study are limited to PGD-based perturbations. Although PGD is widely regarded as one of the strongest gradient-based attacks, it does not fully represent the diversity of adversarial strategies available in computational biology. For example, adverSCarial introduced attacks based on single-gene modifications, max-change optimization, and biological constraint modeling. The system's generalizability under these attack types remains to be tested.

Third, the UMAP visualization demonstrates that adversarial samples heavily overlap with clean samples in low-dimensional space. Although this does not affect meta-detector performance directly, it means that human-interpretable quality control methods (e.g., UMAP inspection) remain ineffective for adversarial detection. This underscores the need for paired detection logic rather than relying solely on visualization tools.

Fourth, the adversarial detection model depends strongly on the underlying classifier. If the classifier architecture or training strategy changes significantly, the internal signal distribution may shift, requiring retraining of the meta-

detector. This coupling means the system is not fully plug-and-play across different models.

Fifth, the system does not currently include any adversarial defense mechanism that actively hardens the classifier (such as adversarial training, gradient smoothing, or certified robustness). The focus is exclusively on detection rather than prevention. While detection is already an important step, combining both detection and defense would create a more comprehensive robust pipeline. Sixth, the dataset used in this study, PBMC3k, is relatively small compared to modern single-cell datasets containing hundreds of thousands to millions of cells. The system's scalability and performance on very large or highly heterogeneous datasets remain open questions.

Finally, the adversarial detection scores show moderate AUC values in some cases (such as AUC = 0.61 in the updated ROC plot). While still usable, this indicates that adversarial detection is challenging and sensitive to statistical noise or model variance. Further refinement of feature extraction, embedding structure, or classifier architecture may improve these results.

10. Conclusion

This chapter synthesizes the overall outcomes of the study and highlights its conceptual and practical contributions to the emerging intersection of adversarial machine learning and single-cell transcriptomics. The project set out to investigate the susceptibility of scRNA-seq classifiers to adversarial attacks and to design an effective detection pipeline capable of distinguishing adversarially manipulated gene-expression profiles from clean, unaltered biological samples. The findings provide both empirical evidence and methodological advancements that strengthen the reliability of computational tools in single-cell analysis.

10.1 Summary of Findings

The results of this study demonstrate several important conclusions concerning the performance, vulnerability, and detectability of adversarial perturbations in scRNA-seq classification models.

First, the main classifier trained on the PBMC3k dataset achieved strong baseline performance, with an overall clean accuracy of approximately 86 percent. The model accurately identified most immune cell subpopulations, particularly those with clear transcriptional signatures such as NK cells, monocytes, and B cells. However, certain rare or ambiguous cell types remained difficult to classify, reflecting biological complexity rather than computational deficiency. This confirmed that the classifier provides a realistic and meaningful platform for adversarial evaluation.

Second, the classifier exhibited extreme vulnerability to adversarial attacks. The multi-epsilon PGD experiments produced a flip rate of nearly 100 percent across all clusters, indicating that minimal but strategically aligned perturbations in gene-expression space are sufficient to mislead the classifier. This vulnerability persisted even for clusters with perfect clean accuracy, reinforcing the idea that robustness and accuracy are distinct properties. The findings align closely with the observations reported in the adverSCarial framework, confirming that adversarial susceptibility is an inherent challenge in deep-learning-based single-cell classification.

Third, despite the subtle and biologically plausible nature of adversarial perturbations, the internal signals of the classifier—specifically logits, embeddings, gradient profiles, and entropy measures—provided reliable indicators of adversarial interference. Feature extraction revealed consistent shifts in gradient magnitude and prediction confidence between clean and adversarial samples. These internal discrepancies form the basis of the proposed meta-learning detection system.

Fourth, the triplet network successfully learned a meta-embedding space that exaggerates the subtle differences between clean and adversarial meta-features. Even though UMAP projections showed visible overlap, the high-dimensional structure enabled the meta-detector to learn a discriminative boundary. The meta-detector achieved approximately 90 percent accuracy in detecting adversarial samples and an overall accuracy of around 87 percent. This confirms that internal-model–based detection is not only feasible but also effective for scRNA-seq data. Fifth, the Streamlit application provided an interpretable, interactive interface for real-time adversarial analysis. It demonstrated clean versus adversarial predictions, decision confidence shifts, and meta-detector outputs, effectively bridging the gap between technical implementation and practical usability.

Collectively, these findings show that adversarial manipulation poses a serious threat to scRNA-seq classification pipelines, but it is possible to detect such manipulations by analyzing internal model behaviors rather than relying solely on biological or visualization-based cues.

10.2 Contribution to the Field

This study contributes to the field of computational biology and adversarial machine learning in several meaningful ways.

First, it introduces one of the earliest end-to-end adversarial detection pipelines tailored specifically for single-cell transcriptomic data. While prior works such as adverSCarial have thoroughly analyzed how attacks succeed, they do not propose a practical framework for identifying when such attacks occur. The present study fills this gap by designing, implementing, and validating a full detection system that can operate alongside existing scRNA-seq classifiers.

Second, the study demonstrates the utility of internal model signals—logits, embeddings, gradients, entropies, and margins—as diagnostic indicators of adversarial interference. This represents a conceptual advancement because it shifts the perspective from evaluating perturbations in input space (which often remain visually subtle) to evaluating the computational dynamics induced by perturbations inside the neural network. This insight is broadly applicable to biological data, where input-level changes rarely create conspicuous artifacts.

Third, the use of a triplet network for embedding clean and adversarial meta-features is a novel methodological contribution. While triplet learning is widely used in face recognition and metric learning, its application to adversarial detection in scRNA-seq pipelines is new. The approach creates a compressed representation space where detection becomes significantly more tractable and stable.

Fourth, the pipeline strengthens the reliability of scRNA-seq classification workflows by providing a mechanism to flag suspicious samples before downstream biological interpretation. As machine learning increasingly influences cell-type annotation, trajectory inference, and disease biomarker discovery, the ability to detect adversarial interference becomes vital for maintaining scientific and clinical integrity.

Fifth, the modular and reproducible design of the implementation—including the classifier, PGD generator, triplet network, meta-detector, and Streamlit application—provides a foundation that other researchers can extend. The pipeline can incorporate additional attack types (DeepFool, CW, FGSM), alternative architectures (transformers, graph neural networks), or larger datasets (such as PBMC68k or Tabula Sapiens), making it a flexible tool for advancing adversarial robustness research in biology.

Finally, this work brings adversarial machine learning—a field predominantly concerned with images, speech, and natural language—into the realm of single-cell genomics. By exploring how adversarial threats manifest in biological data and by demonstrating feasible detection strategies, the study opens new opportunities for interdisciplinary research at the intersection of computational immunology, deep learning, and robust AI.

In summary, this project not only highlights a critical vulnerability in modern scRNA-seq classification systems but also proposes a practical, effective, and extensible solution for adversarial detection. It strengthens the foundation for future research aimed at building robust, safe, and reliable machine-learning tools in computational biology.

11. Future Work

While the proposed adversarial detection pipeline demonstrates strong potential for improving the robustness of scRNA-seq classification systems, several avenues remain open for further exploration and refinement. Future work can be categorized into methodological, biological, computational, and deployment-focused extensions.

11.1 Incorporating Additional Adversarial Attack Models

The present study focuses primarily on multi-epsilon PGD attacks. Although PGD is widely considered one of the strongest gradient-based attacks, real-world adversarial scenarios can exhibit diverse perturbation patterns. Future research should incorporate:

• FGSM (Fast Gradient Sign Method) for single-step adversarial analysis
• DeepFool and CW (Carlini–Wagner) attacks for optimization-based adversaries
• Single-gene perturbation attacks (as demonstrated in the adverSCarial paper)
• Biologically constrained attacks (e.g., fold-change–limited perturbations)
• Black-box attacks such as NES-based or evolutionary attacks
This will help evaluate whether the meta-detector generalizes across adversarial families or requires retraining for each scenario.

11.2 Adversarial Training for Classifier Robustness

The system currently focuses on detection rather than prevention. Future work can integrate adversarial training mechanisms such as:

• PGD adversarial training
• Randomized smoothing
• Gradient regularization
• Jacobian-based adversarial pruning

These techniques may enhance the classifier's intrinsic resilience, reducing the need for external detection modules.

11.3 Model-Agnostic Meta-Detection Framework

The meta-detector is tightly coupled to the internal signals of the specific MLP used in this study. Future developments could explore model-agnostic approaches using:

• SHAP or Integrated Gradients

• Jacobian spectral norms
• Confidence calibration metrics
• Ensemble disagreement patterns

By decoupling the meta-detector from the main classifier architecture, the system can be applied universally across different scRNA-seq models (e.g., Graph Neural Networks, Transformers, or variational autoencoders).

11.4 Scaling to Large and Heterogeneous Datasets

The PBMC3k dataset is small compared to modern single-cell atlases containing hundreds of thousands to millions of cells. Future work should evaluate scalability in:

• PBMC68k
• Tabula Sapiens
• Human Cell Atlas datasets

Larger datasets may reveal additional adversarial vulnerabilities or biological clusters where detection is more challenging.

11.5 Cross-Dataset Generalization Studies

A key question is whether a meta-detector trained on PBMC3k can detect adversarial attacks in unrelated datasets. Exploring cross-dataset generalization will determine whether internal adversarial signatures are universal or dataset-specific.

11.6 Integration of Biological Constraints

Future systems can incorporate biological priors such as:
• Gene regulatory network constraints
• Cell cycle phase correction
• Pathway-level perturbation modeling
This may prevent adversarial attacks that generate biologically impossible gene patterns.

11.7 Improving the Meta-Embedding Architecture

The triplet network produces strong embeddings, but alternative architectures may improve separation, such as:
• Contrastive learning (SimCLR, MoCo)
• Siamese networks

• Transformer-based embedding models
• Spectral contrastive regularization
Such techniques may generate more robust embedding manifolds, reducing overlap between adversarial and clean samples.

11.8 Enhanced Visualization and Interpretability Tools

Visual tools such as saliency maps or adversarial attribution visualizations can help interpret which genes contribute most to adversarial success. This could reveal biological pathways that are more sensitive to perturbations.

11.9 Deployment in Real Biomedical Pipelines

Future implementations can embed the adversarial detection module directly into:
• Automated cell-type annotation tools
• Clinical diagnostic workflows
• Single-cell biomarker discovery pipelines
• High-throughput droplet sequencing QC systems
This will allow real-world validation and help assess detection performance under varied batch effects, sequencing protocols, and biological conditions.

11.10 Creating a Public Benchmark for Adversarial Single-Cell Learning

A standardized benchmark containing multiple datasets, attack types, and detection protocols could accelerate progress in the field. Such a benchmark would allow transparent, reproducible comparisons across research groups.
Overall, the future directions highlight the potential of expanding adversarial detection from a proof-of-concept into a full-scale robust AI framework for single-cell transcriptomics.

12. References

Primary Papers and Models

1. Fievet, L., et al. (2025). *adverSCarial: Robustness Analysis of Single-Cell RNA-seq Classification Models under Adversarial Perturbations*. [Provided PDF].
2. Zheng, G. X. Y., et al. (2017). Massively parallel digital transcriptional profiling of single cells. *Nature Communications*, 8, 14049. (PBMC3k dataset source).
3. Kingma, D. P., & Ba, J. (2015). Adam: A Method for Stochastic Optimization. *International Conference on Learning Representations (ICLR)*.
4. Goodfellow, I., Shlens, J., & Szegedy, C. (2015). Explaining and Harnessing Adversarial Examples. *International Conference on Learning Representations (ICLR)*.
5. Madry, A., et al. (2018). Towards Deep Learning Models Resistant to Adversarial Attacks. *International Conference on Learning Representations (ICLR)*. (PGD attack).
6. Schroff, F., Kalenichenko, D., & Philbin, J. (2015). FaceNet: A Unified Embedding for Face Recognition and Clustering. *IEEE CVPR*. (Triplet loss).

Tools and Libraries

7. Wolf, F. A., Angerer, P., & Theis, F. J. (2018). SCANPY: Large-scale single-cell gene expression data analysis. *Genome Biology*, 19, 15.
8. Paszke, A., et al. (2019). PyTorch: An Imperative Style, High-Performance Deep Learning Library. *NeurIPS*.
9. Pedregosa, F., et al. (2011). Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12, 2825–2830.
10. McInnes, L., Healy, J., & Melville, J. (2018). UMAP: Uniform Manifold Approximation and Projection. *ArXiv preprint*.

Supplementary References (Optional)

11. Kiselev, V. Y., Andrews, T. S., & Hemberg, M. (2019). Challenges in unsupervised clustering of single-cell RNA-seq data. *Nature Reviews Genetics*, 20, 273–282.
12. Svensson, V. (2020). Droplet scRNA-seq is not zero-inflated. *Nature Biotechnology*, 38, 147–150.
13. Luecken, M. D., & Theis, F. J. (2019). Current best practices in single-cell RNA-seq analysis: A tutorial. *Molecular Systems Biology*, 15(6).