Network Inversion for Uncertainty-Aware Out-of-Distribution Detection

Shaik Rehna Afroz (22B3932)
EE 451 Supervised Research Exposition(SRE)
IIT BOMBAY

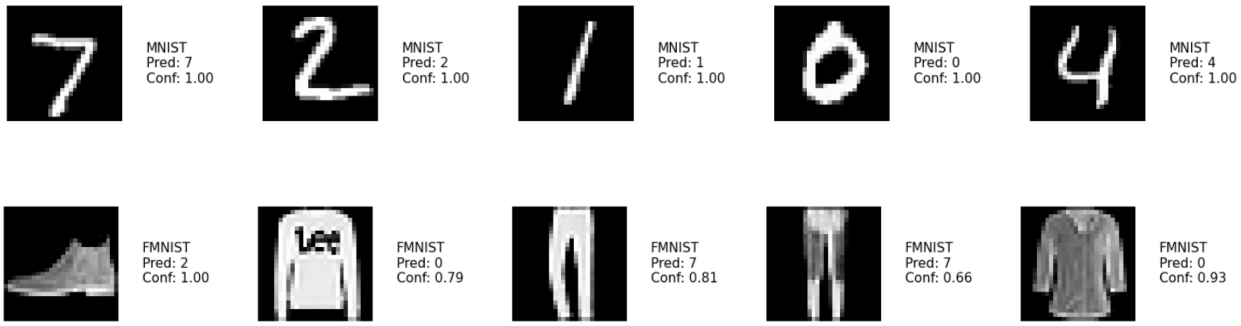
Contents

1 Introduction

1.1 Problem Statement

- Classifiers are excellent at distinguishing between in-distribution classes but often struggle when presented with out-of-distribution (OOD) samples, misclassifying them with high confidence.
- OOD samples are data points that don't belong to the same distribution as the classifier's training data.
- A model trained on MNIST digits (0–9) may incorrectly classify inputs like clothing images (e.g., trousers from FashionMNIST) with high confidence. This misclassification is dangerous in real-world deployments (e.g., medical imaging, autonomous driving) where OOD inputs can lead to catastrophic outcomes.

  Example: A classifier trained on MNIST is shown a FashionMNIST image. It labels the shoe image as a digit "2" with 100% confidence—despite the image being completely unrelated. We need to make the model which flags these kind of inputs as OOD and also if it misclassifies the OOD data it should do it with low confidence



1.2 Why Should This Problem Be Solved

- Solving OOD detection is critical for building safe, interpretable, and deployable AI systems.
- **Safety**: OOD inputs in medical, industrial, or autonomous systems can cause fatal decisions.
- **Generalization**: Robust systems should recognize when an input does not belong to the training distribution.
- **Trust**: Users need interpretable confidence estimates to trust predictions.
- **Performance**: OOD-aware models avoid confidently wrong outputs, leading to better downstream decision-making.

1.3 Why Is It a Challenging Problem

- **Data Distribution Gaps**: Real-world OOD samples can be very different from training data.
- **Confidence Miscalibration**: Softmax outputs are often overconfident, even for unfamiliar inputs.

2 Literature Review

2.1 Network Inversion

- In recent times, the applications of Neural Networks(NN) increased a lot. From image recognition, medical diagnosis to self-driving cars, NN are almost everywhere.
- Despite of their huge usage, they mostly remain like a blackbox.We generally know the output from a NN, but not the internal learnings happened in the NN.This hinders their interpretabiltiy and reliabilty

2.1.1 Paper : Network Inversion and its Applications

- The paper "Network Inversion and its Applications" (https://arxiv.org/pdf/2411.17777) introduces Network Inversion (NI)- a technique to address this issue by determining the input from a known output and system mapping, providing insights into NN decision-making.
- **Methodology:**
- This method involves three key stages:
  First, the classifier is trained and frozen (weights locked).
  Next, an untrained generator (with conditioning mechanisms) learns to synthesize diverse input samples that would produce specific target outputs when fed through the frozen classifier.
  This conditioned generator effectively inverts the classifier's mapping - creating input distributions that reliably trigger desired output classifications while maintaining output diversity.
- Key characteristics:
  Classifier remains fixed during inversion
  Generator training is driven by classifier feedback
  Conditioning ensures label-consistency in generated samples
  Diversity losses prevent mode collapse
- This approach essentially "reverse-engineers" the classifier's decision boundaries by exploring the input space that would lead to particular classifications."

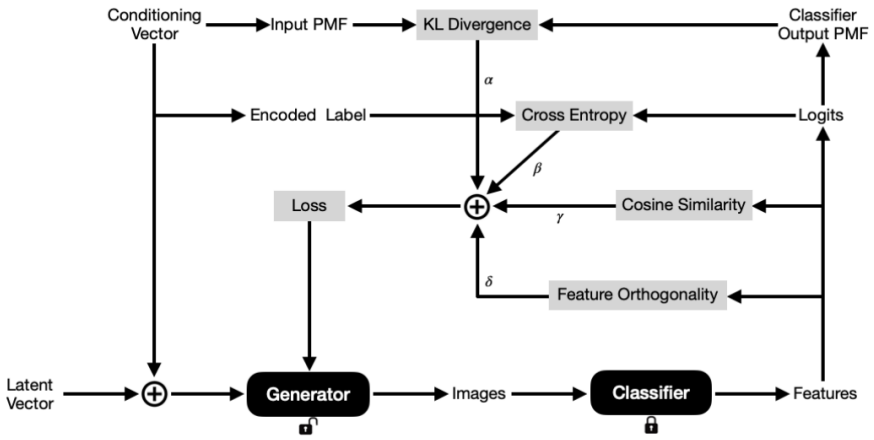


Figure 1. Proposed Approach to Network Inversion

- **Conditioning Mechanisms:**
- **Label conditioning:**
  In generative tasks, generators are conditioned on label embeddings (e.g., in cGANs), where the label encodes class information. However, network inversion requires more than simple embeddings because multiple inputs can lead to the same output. Thus, the generator needs richer guidance to capture input diversity, which is achieved using vectors and matrices
- **Vector conditioning:**
  For an N-class task, N-dimensional vectors are generated from a normal distribution, then softmaxed to represent the input distribution. The argmax index of the softmaxed vector serves as the conditioning label in the loss function, improving the generator's performance
- **Intermediate Matrix Conditioning:**
  Matrix conditioning uses an NXN matrix where the elements in a specific row and column (same index) are set to 1, encoding label information. This conditioning matrix is concatenated with the latent vector intermediately after up-sampling it to NXN spatial dimensions, while the generation upto this point remains unconditioned.
- **Vector-Matrix Conditioning:**
  In Intermediate Matrix Conditioning, the generation starts unconditioned. Both vector and matrix conditioning are combined: vectors are used to condition the generator up to NXN spatial dimensions, after which the conditioning matrix is concatenated for further generation. The argmax index of the vector, corresponding to the row or column set to high in the matrix, serves as the conditioning label.
- Network Inversion (NI) aims to generate images that, when passed through a classifier, produce the same label as the conditioned generator. Using a straightforward cross-entropy loss can cause mode collapse, reducing diversity. To address this, the generator is trained to learn the data distribution of different classes, with a combined loss function L_{inv}:

$$\mathcal{L}_{\text{Inv}} = \alpha \cdot \mathcal{L}_{\text{KL}} + \beta \cdot \mathcal{L}_{\text{CE}} + \gamma \cdot \mathcal{L}_{\text{Cosine}} + \delta \cdot \mathcal{L}_{\text{Ortho}}$$

where $\mathcal{L}_{\text{KL}}$ is the KL Divergence loss, $\mathcal{L}_{\text{CE}}$ is the Cross Entropy loss, $\mathcal{L}_{\text{Cosine}}$ is the Cosine Similarity loss, and $\mathcal{L}_{\text{Ortho}}$ is the Feature Orthogonality loss. The hyperparameters $\alpha, \beta, \gamma, \delta$ control the contribution of each individual loss term defined as:

$$\mathcal{L}_{\text{KL}} = D_{\text{KL}}(P\|Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$$

$$\mathcal{L}_{\text{CE}} = -\sum_i y_i \log(\hat{y}_i)$$

$$\mathcal{L}_{\text{Cosine}} = \frac{1}{N(N-1)} \sum_{i \neq j} \cos(\theta_{ij})$$

$$\mathcal{L}_{\text{Ortho}} = \frac{1}{N^2} \sum_{i,j} (G_{ij} - \delta_{ij})^2$$

- **Loss Functions:**

- **Cross Entropy (L_{CE}):**
  Cross Entropy is used to ensure the generator produces images with the correct labels. When labels are encoded into vectors, they are used in the loss function, allowing the generator to learn the conditioned image distribution. Training with vector conditioning requires the generator to learn the encoding mechanism for better exploration of the classifier's input space.

- **KL Divergence (L_{KL}):**
  KL Divergence minimizes the difference between the output distribution of generated images and the conditioning distribution, ensuring the generator learns the data distribution effectively.

- **Cosine Similarity (L_{Cosine}):**
  Cosine Similarity encourages diversity by reducing the angular distance between generated image features, ensuring both correct classification and diverse image generation for each label.

- **Feature Orthogonality (L_{Ortho}):**
  In Intermediate Matrix Conditioning, the generation starts unconditioned. Both vector and matrix conditioning are combined: vectors are used to condition the generator up to NXN spatial dimensions, after which the conditioning matrix is concatenated for further generation. The argmax index of the vector, corresponding to the row or column set to high in the matrix, serves as the conditioning label.

- **Applications**:

- **Interpretability** :
  The interpretability of inverted samples is analyzed by their features in the penultimate layer of the classifier. Inversion Accuracy refers to the percentage of images generated with desired labels same as the output labels from the classifier. The generator is trained to achieve an Inversion Accuracy $> 95\%$, meaning that most generated images are classified correctly.

- **Training-Like Data Reconstruction**

- **Out-of-Distribution detection**

2.1.2 Paper: Network Inversion for Training-Like Data Reconstruction

- Through Network Inversion we get a diverse set of images in the input space of the model for different classes. But the inverted samples are completely random

- The paper " Network Inversion for Training-Like Data Reconstruction "
  (https://arxiv.org/pdf/2410.16884) uses Network inversion for reconstructing training-like data entirely from the weights of the classifier without any insight of the training process.

- Reconstruction of training-like data was achieved by exploiting key properties of the training data in relation to the classifier including model confidence, robustness to perturbations gradient behavior along with some prior knowledge about the training data.

2.1.3 Paper: NETWORK INVERSION OF BINARISED NEURAL NETS

- The paper "NETWORK INVERSION OF BINARISED NEURAL NETS " (https://arxiv.org/pdf/2402.11995) introduces a novel approach to invert a BNN by encoding it into a Conjunctive Normal Form (CNF) propositional formula, which captures the network's structure.
- CNF formula: Encodes the computation of each neuron in the network and allows for both inference and inversion using satisfiability (SAT) solvers.
- By setting constraints on output variables, the formula can be used for inversion, generating inputs corresponding to a specific output.
- This method does not require hyperparameter tuning, unlike optimization-based approaches.
- The deterministic nature of CNF encodings ensures fine-grained control over the inverted samples, which helps mitigate issues like mode collapse commonly seen in generative models.

2.2 Uncertainty Estimation

2.2.1 Paper: Autoinverse: Uncertainty Aware Inversion of Neural Networks

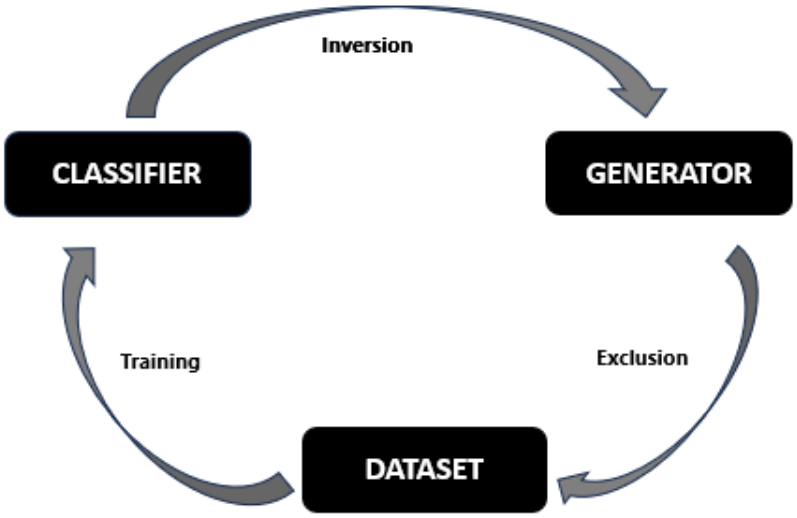
- This paper(https://arxiv.org/abs/2208.13780v2) proposes Autoinverse - a highly automated approach for inverting neural network surrogates. This approach seeks inverse solutions in the vicinity of reliable data by taking into account the predictive uncertainty of the surrogate and minimizing it during inversion.
- Autoinverse improves neural inversion by considering predictive uncertainty of the surrogate model.It finds solutions near reliable training data, minimizing errors from poorly sampled or noisy data regions.Instead of forcing the surrogate to perfectly match the Native Forward Process(NFP), it adapts the inversion process to account for uncertainty.
- Autoinverse achieves this goal by, first by training a surrogate capable of predictive uncertainty. Second, relying on this trained surrogate and using a novel inversion cost function, Autoinverse finds accurate designs with minimal uncertainty. Autoinverse is applied on two inverse methods belonging to the two main neural inversion categories, i.e., optimization- and architecture-based

2.2.2 Paper: Uncertainty-Aware Out-of-Distribution Detection with Gaussian Processes

- This paper (https://arxiv.org/pdf/2412.20918) proposes proposes a Gaussian-process-based method for out-of-distribution (OOD) detection in deep neural networks (DNNs) that eliminate the need for OOD data during training.
- This approach includes multi-class Gaussian processes (GPs) to quantify uncertainty in unconstrained Softmax scores of a DNN and defines a score function based on the predictive distribution of the multi-class GP to distinguish in-distribution(InD) and OOD samples.
- This method is compatible with existing DNN architectures and requires only InD data for training, addressing a key limitation of prior OOD detection techniques.

3 Our Approach:

- To address the problem of OOD detection, Network inversion is used to generate OOD samples from the classifier's input space. These samples, which deviate from the trained distribution, are assigned to a "garbage" class
- This method uses an n+1 approach, where n is the number of classes present in the training data. Extra "garbage" class is added to the classifier's output for ood detection
- To make the classifier familiar with the OOD data, it is trained with an additional "garbage" class, initially populated by samples generated from random noise. This helps the classifier learn to recognize and isolate OOD samples



3.1 Training Process:

- After each training epoch, inverted samples for each class are added to the "garbage" class, further augmenting the dataset with OOD examples.
- This iterative retraining process helps the classifier differentiate between in-distribution and OOD data, improving its robustness.

3.2 Addressing class Imbalance:

- The inclusion of the "garbage" class creates a data imbalance, which is managed using a weighted cross-entropy loss function. This function dynamically adjusts weights to ensure effective learning despite the imbalance.

3.3 Datasets

- The following datasets are used:
  MNIST, FMNIST, CIFAR10, SVHN, CIFAR100, TinyImageNet-200

4 Model Architectures for MNIST, FMNIST

4.1 Classifier Architecture

The classifier consists of three convolutional blocks followed by a global pooling operation and two fully connected stages. The model takes a $[B, nc, 28, 28]$ image as input and predicts one of $n\_classes$ labels.

Layer-wise Description

- **Conv Block 1:**
  - `Conv2d(nc → ncf, kernel=4, stride=2, padding=1)` reduces the spatial size from 28×28 to 14×14.
  - `BatchNorm2d` normalizes activations.
  - `LeakyReLU(0.2)` introduces non-linearity.
- **Conv Block 2:**
  - `Conv2d(ncf → 2ncf, kernel=4, stride=2, padding=1)` reduces the feature map to 7×7.
  - `BatchNorm2d`
  - `LeakyReLU(0.2)`
- **Conv Block 3:**
  - `Conv2d(2ncf → 4ncf, kernel=3, stride=1, padding=0)` outputs a 5×5 feature map.
  - `BatchNorm2d`
  - `LeakyReLU(0.2)`
- **Global Pooling and Flattening:**
  - `AdaptiveAvgPool2d(1)` converts $[B, 4ncf, 5, 5]$ to $[B, 4ncf, 1, 1]$.
  - `Flatten()` produces a $[B, 4ncf]$ feature vector.
- **Penultimate Layer:**
  - `Linear(4ncf → 2ncf)`
  - `LeakyReLU(0.2)`
  - `Dropout(0.5)` for regularization.
- **Classification Layer:**
  - `Linear(2ncf → n_classes)` produces logits.

4.2 Generator Architecture

The generator uses class-conditioning, where the label vector is embedded and concatenated with a latent noise vector. It gradually upsamples from 1×1 to 28×28 through transposed convolution layers.

Layer-wise Description

- **Class Embedding:**
  - `Linear(n_classes` $\rightarrow$ `nz)` maps one-hot labels to the latent dimension.
- **Input Fusion:**
  - Concatenate latent vector z and embedded label along channel dimension to obtain a tensor of shape $[B, 2nz, 1, 1]$.
- **Deconv Block 1:**
  - `ConvTranspose2d(2nz` $\rightarrow$ `4ngf, kernel=4, stride=1, padding=0)` produces a 4×4 feature map.
  - `BatchNorm2d`
  - `LeakyReLU(0.2)`
- **Deconv Block 2:**
  - `ConvTranspose2d(4ngf` $\rightarrow$ `2ngf, kernel=4, stride=2, padding=1)` upsamples to 8×8.
  - `BatchNorm2d`
  - `LeakyReLU(0.2)`
- **Deconv Block 3:**
  - `ConvTranspose2d(2ngf` $\rightarrow$ `ngf, kernel=4, stride=2, padding=2)` outputs a 14×14 map.
  - `BatchNorm2d`
  - `LeakyReLU(0.2)`
- **Final Upsampling:**
  - `ConvTranspose2d(ngf` $\rightarrow$ `nc, kernel=4, stride=2, padding=1)` produces a 28×28 grayscale image.
  - `Tanh()` normalizes output to $[-1, 1]$.

5 Model Architectures for CIFAR10, SVHN

5.1 Classifier Architecture

The classifier is a convolutional neural network designed for 32×32 RGB images. It progressively downsamples the input while increasing feature dimensions, followed by a penultimate fully-connected layer and a final classification layer.

Feature Extractor

- **Layer 1:**
  - `Conv2d`(nc $\rightarrow$ ncf, kernel=4, stride=2, padding=1): Downsamples 32×32 to 16×16.
  - `BatchNorm2d`: Stabilizes early training.
  - `LeakyReLU(0.2)`: Prevents dead neurons.
  - `Dropout(0.2)`: Regularization.
- **Layer 2:**
  - `Conv2d`(ncf $\rightarrow$ ncf, kernel=3, stride=1, padding=1): Maintains resolution at 16×16.
  - BatchNorm + LeakyReLU.
- **Layer 3:**
  - `Conv2d`(ncf $\rightarrow$ ncf*2, kernel=4, stride=2): Downsamples 16×16 to 8×8.
  - BatchNorm + LeakyReLU.
  - `Dropout(0.3)`: Stronger regularization.
- **Layer 4:**
  - `Conv2d`(ncf*2 $\rightarrow$ ncf*2, kernel=3): Preserves 8×8 spatial size.
  - BatchNorm + LeakyReLU.
- **Layer 5:**
  - `Conv2d`(ncf*2 $\rightarrow$ ncf*4, kernel=4, stride=2): Downsamples 8×8 to 4×4.
  - BatchNorm + LeakyReLU.
  - `Dropout(0.4)`: Increased dropout at deeper layers.
- **AdaptiveAvgPool2d(1):** Collapses spatial size $4 \times 4 \rightarrow 1 \times 1$.
- **Flatten:** Produces a vector of dimension $ncf \times 4$.

Penultimate Layer

- `Linear`(ncf*4 $\rightarrow$ ncf*4) Dense feature transformation.
- `LeakyReLU(0.2)`
- `Dropout(0.5)` for regularization.

Classification Layer

- `Linear`(ncf*4 → n_classes) Produces logits for all classes.

5.2 Generator Architecture

The generator uses transposed convolutions to synthesize 32×32 images conditioned on class labels.

Class Embedding

- `Linear`(n_classes → nz): Embeds one-hot class vector into latent space.

Main Generator Network

- **Block 1:**
  - `ConvTranspose2d`(nz*2 → ngf*8, kernel=4, stride=1) Expands 1×1 latent code to 4×4.
  - BatchNorm + LeakyReLU.
- **Block 2:**
  - `ConvTranspose2d`(ngf*8 → ngf*4, kernel=4, stride=2) Upsamples $4 \times 4 \rightarrow 8 \times 8$.
  - BatchNorm + LeakyReLU.
- **Block 3:**
  - `ConvTranspose2d`(ngf*4 → ngf*2, kernel=4, stride=2) Upsamples $8 \times 8 \rightarrow 16 \times 16$.
  - BatchNorm + LeakyReLU.
- **Block 4:**
  - `ConvTranspose2d`(ngf*2 → nc, kernel=4, stride=2) Final upsampling: $16 \times 16 \rightarrow 32 \times 32$.
  - `Tanh`: Image normalization to $[-1, 1]$.

Forward Pass

- Class vector is embedded and reshaped to match latent dimensions.
- Latent vector and class embedding are concatenated.
- The combined representation is decoded into an image.

6 Loss Functions

The following loss functions that are mentioned earlier are used for the training of the generator

- Cross-Entropy Loss
- KL Divergence
- Cosine Similarity

7 Uncertainty Estimation Method

To quantify the model's uncertainty on a given input, we use a normalized deviation from the uniform distribution over class probabilities. The uncertainty estimate lies in the range $[0, 1]$, where values closer to 1 indicate higher uncertainty.

7.1 Procedure

Given the classifier logits $z \in \mathbb{R}^{B \times K}$ for a batch of B samples and K classes:

1. Compute softmax probabilities:

$$p = \text{Softmax}(z), \quad p \in \mathbb{R}^{B \times K}$$

2. Define a uniform distribution over K classes:

$$u = \left[\frac{1}{K}, \frac{1}{K}, \ldots, \frac{1}{K}\right] \in \mathbb{R}^K$$

3. Compute squared deviation of the predicted probabilities from the uniform distribution:

$$a = \sum_{i=1}^{K} (p_i - u_i)^2$$

4. Compute squared deviation of the predicted one-hot class (argmax) from the uniform:

$$b = \sum_{i=1}^{K} (\hat{y}_i - u_i)^2, \quad \text{where } \hat{y} = \text{one-hot}(\arg\max(p))$$

5. Final uncertainty estimate:

$$\mathcal{U}(x) = 1 - \frac{a}{b}$$

This normalizes the deviation of the model's predicted distribution from the uniform baseline. A value of $\mathcal{U}(x)$ close to 0 implies high confidence, whereas values near 1 reflect a distribution close to uniform (i.e., maximum uncertainty).

8 Training Strategy and Epoch Flow

- The proposed pipeline performs iterative generator training using a frozen classifier, inversion accuracy estimation, uncertainty quantification, and classifier refinement. The process is designed to gradually improve the generator's ability to produce class-consistent samples while updating the classifier to better generalize when exposed to both inversion-generated synthetic samples and potential out-of-distribution (OOD) data
- One dataset is considered as in-dist and training is done using that dataset and other datasets are OOD datasets used for evaluation

9 Conditioning and Metric Computation

Different types of conditioning can be applied to the generator:

- Vector Conditioning
- Matrix Conditioning
- Vector–Matrix Conditioning

In the experiments, **Vector Conditioning** is used.

9.1 Metric Computation Procedure

- After inversion of the generator, a **fixed noise** and **fixed PDF** are used to generate 10 images per class.
- These generated images are passed through the classifier (CSF) to compute the following metrics:
  - Confidence
  - Uncertainty Estimate (UE)
  - Entropy
  - Margin
- For each metric, the average over 10 samples per class is computed.
- Additionally, the class-wise averages are further averaged to obtain the **overall metric averages**.

9.2 Example Output

```
Confidence : [0.7997, 0.9037, 0.3358, 0.8905, 0.7673, 0.573,
              0.868, 0.8418, 0.847, 0.3709, 0.8798]

Margin     : [0.6819, 0.8617, 0.1322, 0.8465, 0.6732, 0.2993,
              0.8004, 0.7809, 0.7455, 0.0781, 0.8312]

UE         : [0.3794, 0.1987, 0.8919, 0.2239, 0.4335, 0.6516,
              0.2637, 0.3139, 0.2984, 0.8145, 0.2436]

Entropy    : [0.7657, 0.4681, 1.9575, 0.5076, 0.9141, 1.2666,
              0.5483, 0.7079, 0.6028, 1.6117, 0.535]
```

Average Metrics:

- Avg_Confidence = 0.7343
- Avg_Margin = 0.6119
- Avg_UE = 0.4285
- Avg_Entropy = 0.8987

9.3 Notes on Training and Inversion

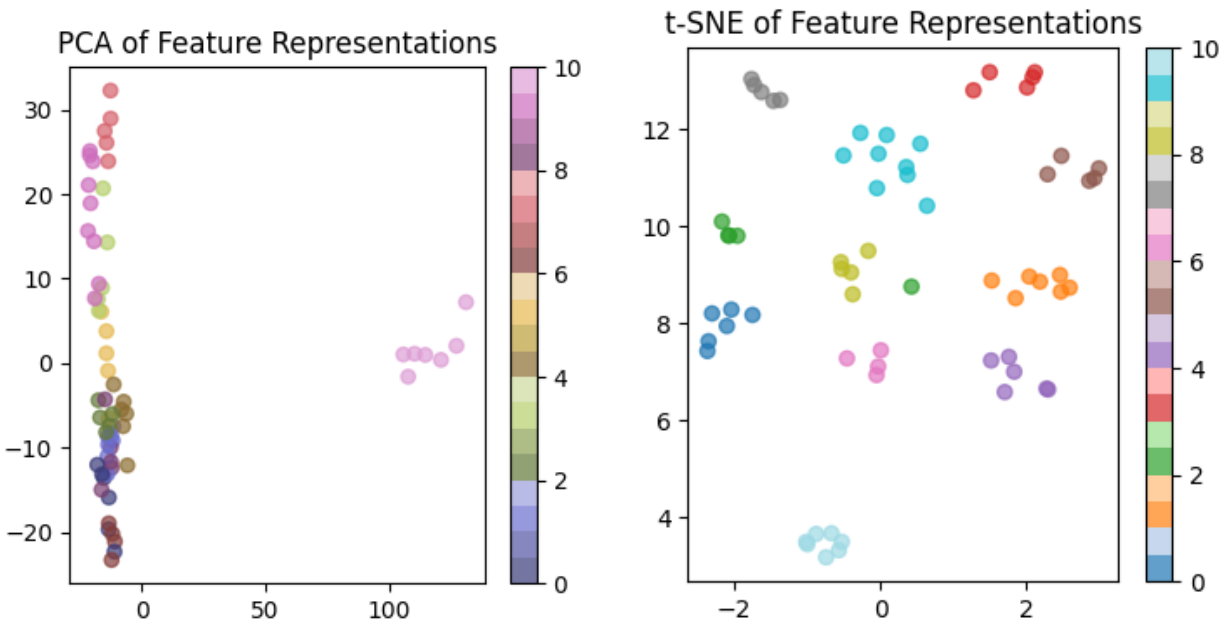
- A fixed noise and PDF are used throughout training to maintain a **fair comparison** of metrics across epochs.
- Inversion accuracy is computed using the images generated during the **final inversion step** of an epoch.
- For each epoch:
  - If 100 inversion steps are performed
  - and 11000 images are generated per step,
  - then the 11000 images from the **100th step** are added to the **garbage class** of the training set.

9.4 Hyperparameter Tuning

- Hyperparameters are tuned based on the behaviour of the four metrics (Confidence, Margin, UE, Entropy).
- In general OOD evaluation uses a held-out validation OOD dataset and reports:
  - AUROC
  - AUPR
  - FPR@95TPR
- Typically, early in training:
  - Confidence and Margin are **low**.
  - UE and Entropy are **high** (poorly generated samples).
- With training:
  - Confidence and Margin **increase toward 1**.
  - UE and Entropy **decrease toward 0**.

9.5 Pre-training the Classifier for 1 epoch

- At the very beginning the classifier is pre-trained on the (mnist-train + guassian nosie) dataset for 1 epoch.
- Below figures show the PCA, t-SNE plot of this features generated for the 11 classses by this trained classifer
- The isolated dots in the PCA, t-SNE plots represent the garbage class



9.6 Overview of Iterative Process

1. **Initialize and Load Pretrained Classifier**:
   Load the pretrained classifier trained for 1 epoch on (MNIST train + Gaussian noise) dataset. Freeze its weights during generator updates.

2. **Generator Training** :

   - Sample latent vectors $z \sim \mathcal{N}(0, I)$ and random class-conditioning vectors $\mathbf{c}$
   - Generate synthetic images $\hat{x} = G(z, \mathbf{c})$.
   - Compute inversion loss , backpropagate and update generator $\mathcal{L}_{\text{inv}}$:

$$\mathcal{L}_{\text{inv}} = \alpha \cdot \text{KL} + \beta \cdot \text{CE} + \gamma \cdot \text{CosSim}$$

3. **Uncertainty Estimation**:

   - Pass generated images through the classifier.
   - Metrics are calculated as explained in "9.1 Metric Computation Procedure"

4. **Save Generated Samples**:

   - Save generated images and label-wise sample grids.

5. **Classifier Retraining**:

   - Add generated samples to the training set.
   - Compute class-balanced loss weights and train classifier for 1 epoch with updated data.
   - Repeat steps 2,3,4,5 for 'n' epochs

- This iterative scheme strengthens the generator's conditioning capability and encourages the classifier to adapt to hard or uncertain regions in the data space.

Standard OOD Metrics

- AUROC : Probability a random OOD score > random ID score. It is independent of any specific threshold
- AUPR: Precision–recall curve with OOD treated as positive class
- FPR@95TPR: False positive rate when 95% of ID samples are correctly detected. Lower FPR gives better OOD rejection.

OOD Detection methods

Scores for out-of-distribution (OOD) detection and classification confidence: Maximum Softmax Probability (MSP), ODIN (temperature scaling + input perturbation), Energy score, and the Mahalanobis-based detector.

Notation

Let $x \in \mathcal{X}$ be an input and let a classifier produce logits $\mathbf{z}(x) = [z_1(x), \ldots, z_C(x)]^\top \in \mathbb{R}^C$ for C classes. The softmax probabilities are

$$p_i(x) = \text{softmax} * i(\mathbf{z}(x)) = \frac{e^{z_i(x)}}{\sum * j = 1^C e^{z_j(x)}}, . \tag{1}$$

Let $f_\ell(x) \in \mathbb{R}^d$ be the feature vector extracted from layer ℓ of the network (often the penultimate layer).

Maximum Softmax Probability (MSP)

The MSP score is the simplest confidence score:

$$\text{MSP}(x) = \max_{i \in 1,\ldots,C} p_i(x) = \max_i \text{softmax}_i(\mathbf{z}(x)). \tag{2}$$

A detector flags x as OOD if $\text{MSP}(x) < \tau$ for some threshold τ. MSP is cheap and often a strong baseline, but it can be overconfident for OOD inputs.

ODIN

ODIN (Liang et al., 2018) improves MSP by applying temperature scaling and a small input perturbation designed to increase separation between in- and out-of-distribution scores.

Temperature-scaled softmax

For temperature $T > 0$ define

$$p_i^{(T)}(x) = \frac{e^{z_i(x)/T}}{\sum_{j=1}^C e^{z_j(x)/T}}, . \tag{3}$$

Higher T (e.g. $T \gg 1$) makes the softmax softer and changes relative ordering of MSP magnitudes.

Input perturbation

Compute the gradient of the (negative) log-likelihood of the predicted class w.r.t. the input and take a small step:

$$\hat{y} = \arg\max_i z_i(x),\ \tilde{x} \qquad = x - \varepsilon \cdot \operatorname{sign}\left(-\nabla_x \log p_{\hat{y}}^{(T)}(x)\right), . \tag{4}$$

ODIN score uses the MSP (or max temperature-scaled softmax) evaluated at $\tilde{x}$:

$$\mathrm{ODIN}(x) = \max_i p_i^{(T)}(\tilde{x}), . \tag{5}$$

Typical hyperparameters: $T \in 1, 10, 1000$ and ε small (e.g. 10^{-3} to 10^{-1} depending on input scale). Values are tuned on a held-out validation set.

Energy Score

The free-energy score (Liu et al., 2020) associates an energy to logits; lower energy implies higher likelihood under the model.

$$E(x) = -T \cdot \log\left(\sum_{i=1}^{C} e^{z_i(x)/T}\right) = -T, \mathrm{LSE}\left(\frac{\mathbf{z}(x)}{T}\right) \tag{6}$$

Where $\mathrm{LSE}(\mathbf{v}) = \log \sum_i e^{v_i}$. Note that

$$E(x) = -T \log\left(\sum_i e^{z_i/T}\right) = -T \log\left(e^{z_k/T}(1 + \sum_{i \neq k} e^{(z_i - z_k)/T})\right), , \tag{7}$$

so energy is linked to the log-sum-exp of logits. For detection, we often threshold $E(x)$: OOD if $E(x) > \tau_E$ (higher energy = less likely).

Energy has some advantages over raw MSP because it uses the full logits vector (not only the maximum) and can be tuned by temperature T.

Mahalanobis Distance-based Detector

This method (Lee et al., 2018) fits class-conditional Gaussian distributions on intermediate features $f_\ell(x)$. For each class c estimate the empirical mean $\boldsymbol{\mu}_c$ and a shared covariance matrix $\boldsymbol{\Sigma}$ (pooled over all classes):

$$\boldsymbol{\mu}_c = \frac{1}{N_c} \sum_{x \in \mathcal{D}_c} f_\ell(x),\ \boldsymbol{\Sigma} \qquad = \frac{1}{N} \sum_{c=1}^{C} \sum_{x \in \mathcal{D}_c} \big(f_\ell(x) - \boldsymbol{\mu}_{y(x)}\big)\big(f_\ell(x) - \boldsymbol{\mu}_{y(x)}\big)^\top. \tag{8}$$

The (squared) Mahalanobis distance of a test feature $f = f_\ell(x)$ to class c is

$$D_c(f) = (f - \boldsymbol{\mu}_c)^\top \boldsymbol{\Sigma}^{-1} (f - \boldsymbol{\mu}_c). \tag{9}$$

The detector score is the negative of the minimum Mahalanobis distance across classes (or a calibrated mapping):

$$\mathrm{Mahalanobis}(x) = -\min_c ; D_c\big(f_\ell(x)\big). \tag{10}$$

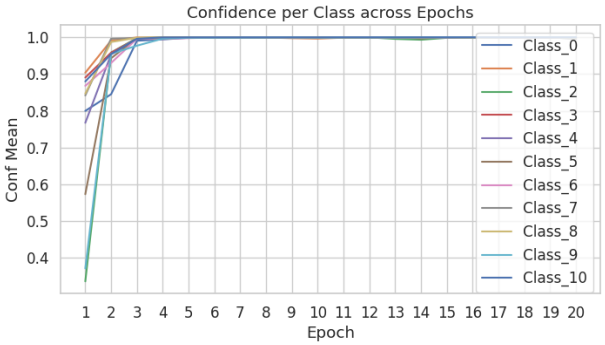
Larger (less negative) values indicate OOD. In practice, using multiple layers ℓ, and optionally adding small input perturbations (similar in spirit to ODIN) and a logistic regression calibration step, improves performance.

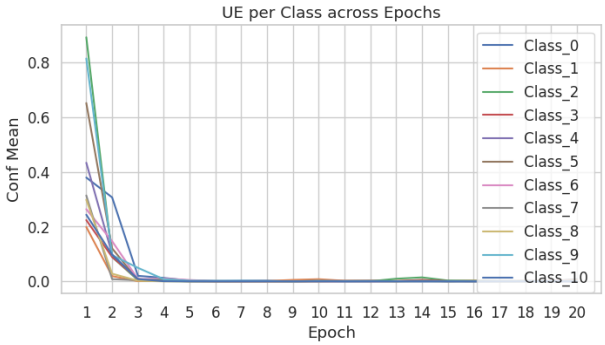
10 Experiments and Results

10.1 MNIST as In-dist

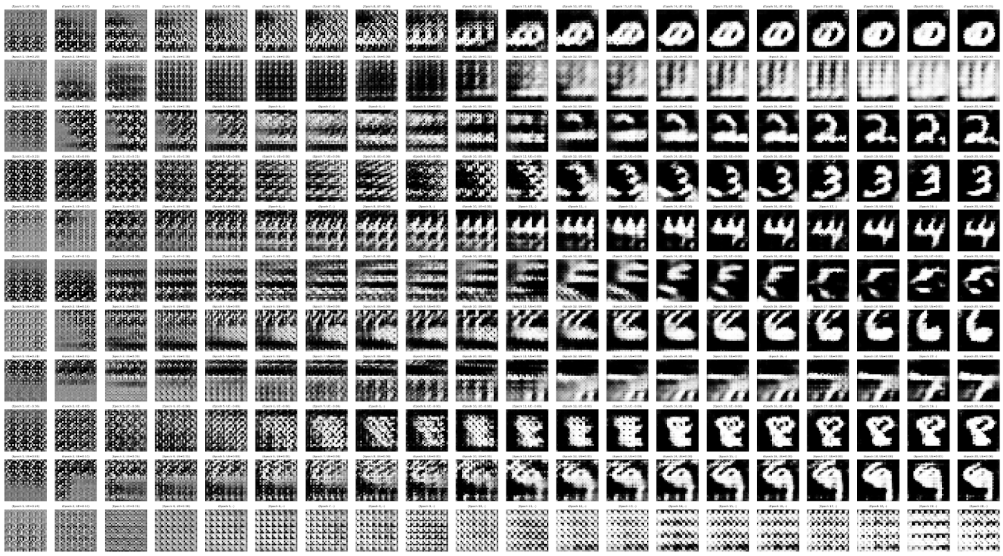
10.1.1 Hyperparameters and Settings

- Inversion Loss Weights: $\alpha = 0$, $\beta = 1$, $\gamma = 0$
- Latent Dimension: 100, Batch Size: 32
- Inversion Steps per Epoch: 100
- Number of samples generated in each step = 11000
- Number of Epochs: 20
- Learning Rate:
  Classifier: 1e-4, Generator: 1e-3





10.1.2 Evolution of generated samples across 20 epochs



Uncertainty Estimate

Before Inversion
(Classifier trained only for 1 epoch)

(Test Accuracy: 0.9687)

Metric: UE

	Dataset	ID Min	ID Max	ID Mean	ID Std	OOD Min	OOD Max	OOD Mean	OOD Std
0	MNIST	0.000205	0.928912	0.100253	0.173317	NaN	NaN	NaN	NaN
1	FashionMNIST	0.000004	0.945538	0.530332	0.226740	0.002357	0.941319	0.381060	0.258903
2	CIFAR-10	0.066922	0.907345	0.606462	0.168031	0.001533	0.883389	0.092518	0.123436
3	SVHN	0.079920	0.851894	0.489411	0.107215	0.004392	0.841092	0.145559	0.143081
4	CIFAR-100	0.085097	0.912813	0.560186	0.170852	0.001078	0.919983	0.108546	0.142902
5	TinyImageNet200	0.017234	0.922731	0.579638	0.191451	0.000751	0.899822	0.093800	0.138340

After Inversion
(Test Accuracy: 0.9915)

Metric: UE

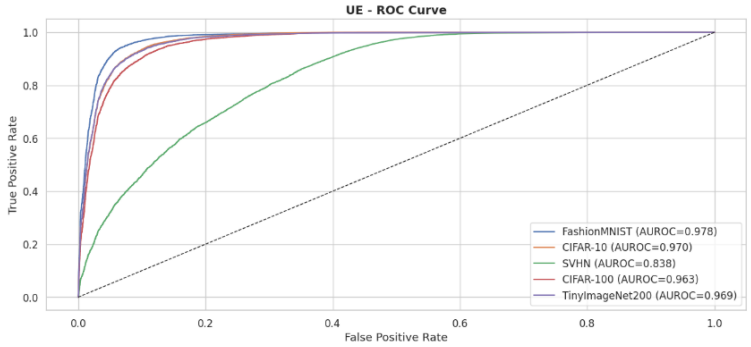
	Dataset	ID Min	ID Max	ID Mean	ID Std	OOD Min	OOD Max	OOD Mean	OOD Std
0	MNIST	0.000000	0.853623	0.010711	0.063468	0.468219	0.672305	0.570262	0.102043
1	FashionMNIST	0.000024	0.898632	0.374476	0.243180	0.000000	0.837257	0.086875	0.170573
2	CIFAR-10	0.000017	0.815992	0.178606	0.201556	0.000000	0.739022	0.018194	0.079734
3	SVHN	0.000002	0.857726	0.074587	0.148292	0.000000	0.732551	0.045126	0.121348
4	CIFAR-100	0.000010	0.811448	0.187837	0.207085	0.000000	0.749334	0.026406	0.094324
5	TinyImageNet200	0.000002	0.835493	0.195724	0.217809	0.000000	0.752626	0.021974	0.085779

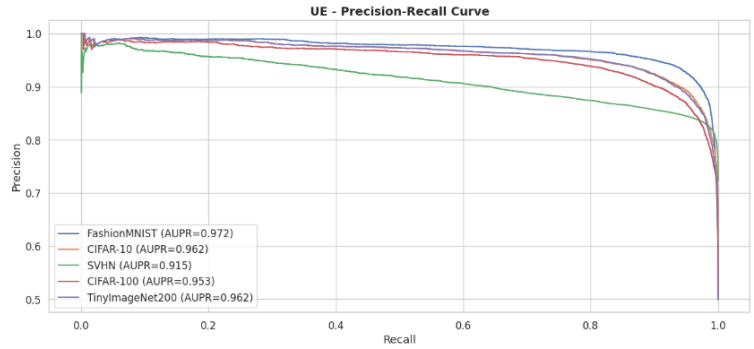
10.1.3 AUPR (↑) , AUPR (↑) , FPRTPR(↓)

COMPREHENSIVE OOD ANALYSIS (Garbage class excluded) - UE

	OOD Dataset	AUROC	AUPR	FPR@95TPR	ID ue Mean	OOD ue Mean	Separation
0	FashionMNIST	0.9784	0.9719	0.0747	0.0108	0.3754	0.3646
1	CIFAR-10	0.9704	0.9620	0.1137	0.0108	0.3242	0.3134
2	SVHN	0.8381	0.9152	0.4550	0.0108	0.0981	0.0873
3	CIFAR-100	0.9626	0.9526	0.1430	0.0108	0.2858	0.2750
4	TinyImageNet200	0.9692	0.9615	0.1204	0.0108	0.3266	0.3158

Avg AUROC = 0.9437 | Avg AUPR = 0.9526 | Avg FPR@95TPR = 0.1814





Comparison of AUROC, AUPR and FPR@95TPR for different OOD detection methods

=== AUROC Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	FashionMNIST	0.9784	0.9780	0.9829	0.9929	0.9982
1	CIFAR-10	0.9704	0.9701	0.9845	0.9961	0.9999
2	SVHN	0.8381	0.8381	0.8890	0.9407	0.9998
3	CIFAR-100	0.9626	0.9623	0.9770	0.9935	0.9998
4	TinyImageNet200	0.9692	0.9689	0.9814	0.9947	0.9998

=== AUPR Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	FashionMNIST	0.9719	0.9706	0.9790	0.9922	0.9983
1	CIFAR-10	0.9620	0.9607	0.9798	0.9960	0.9999
2	SVHN	0.9152	0.9148	0.9409	0.9732	0.9999
3	CIFAR-100	0.9526	0.9515	0.9710	0.9935	0.9998
4	TinyImageNet200	0.9615	0.9601	0.9776	0.9949	0.9998

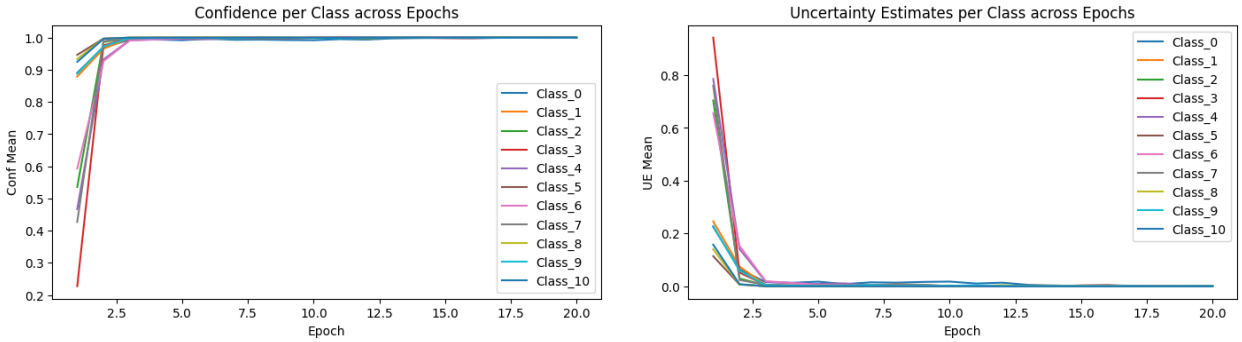
=== FPR@95TPR Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	FashionMNIST	0.0747	0.0747	0.0599	0.0286	0.0083
1	CIFAR-10	0.1137	0.1137	0.0658	0.0199	0.0000
2	SVHN	0.4550	0.4550	0.3375	0.2247	0.0004
3	CIFAR-100	0.1430	0.1430	0.0967	0.0343	0.0000
4	TinyImageNet200	0.1204	0.1205	0.0788	0.0254	0.0001

10.2 FMNIST as In-dist

10.2.1 Hyperparameters and Settings

- Inversion Loss Weights: $\alpha = 0$, $\beta = 1$, $\gamma = 0$
- Latent Dimension: 100, Batch Size: 32
- Inversion Steps per Epoch: 100
- Number of samples generated in each step = 11000
- Number of Epochs: 20
- Learning Rate:
  Classifier: 1e-4, Generator: 1e-3



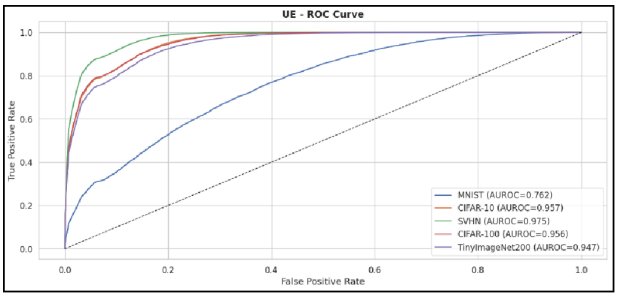
10.2.2 Evolution of generated samples across 20 epochs

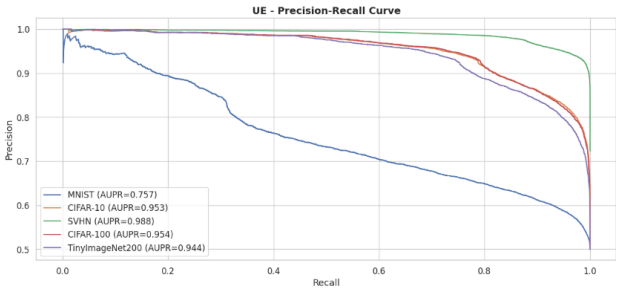
10.2.3 AUPR (↑) , AUPR (↑) , FPRTPR(↓)

COMPREHENSIVE OOD ANALYSIS (Garbage class excluded) - UE

	OOD Dataset	AUROC	AUPR	FPR@95TPR	ID ue Mean	OOD ue Mean	Separation
0	MNIST	0.7623	0.7575	0.6677	0.1247	0.3441	0.2194
1	CIFAR-10	0.9566	0.9533	0.1971	0.1247	0.6800	0.5553
2	SVHN	0.9747	0.9884	0.1322	0.1247	0.7291	0.6044
3	CIFAR-100	0.9562	0.9535	0.2053	0.1247	0.6813	0.5567
4	TinyImageNet200	0.9465	0.9444	0.2424	0.1247	0.6597	0.5350

Avg AUROC = 0.9193 | Avg AUPR = 0.9194 | Avg FPR@95TPR = 0.2889





Comparison of AUROC, AUPR and FPR@95TPR for different OOD detection methods

=== AUROC Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.7623	0.7553	0.9039	0.8836	0.9769
1	CIFAR-10	0.9566	0.9454	0.9880	0.9967	0.9981
2	SVHN	0.9747	0.9619	0.9946	0.9983	0.9901
3	CIFAR-100	0.9562	0.9448	0.9880	0.9963	0.9963
4	TinyImageNet200	0.9465	0.9353	0.9867	0.9937	0.9943

=== AUPR Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.7575	0.7368	0.9066	0.8876	0.9713
1	CIFAR-10	0.9533	0.9372	0.9854	0.9968	0.9983
2	SVHN	0.9884	0.9814	0.9971	0.9993	0.9964
3	CIFAR-100	0.9535	0.9371	0.9859	0.9965	0.9968
4	TinyImageNet200	0.9444	0.9278	0.9849	0.9943	0.9950

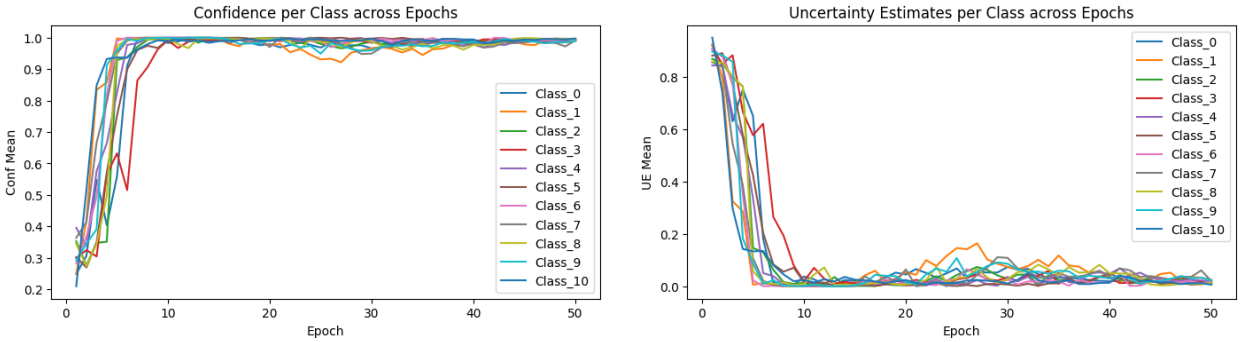
=== FPR@95TPR Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.6677	0.6677	0.4565	0.5121	0.0881
1	CIFAR-10	0.1971	0.2010	0.0489	0.0144	0.0039
2	SVHN	0.1322	0.1415	0.0195	0.0090	0.0398
3	CIFAR-100	0.2053	0.2080	0.0505	0.0137	0.0052
4	TinyImageNet200	0.2424	0.2441	0.0574	0.0244	0.0141

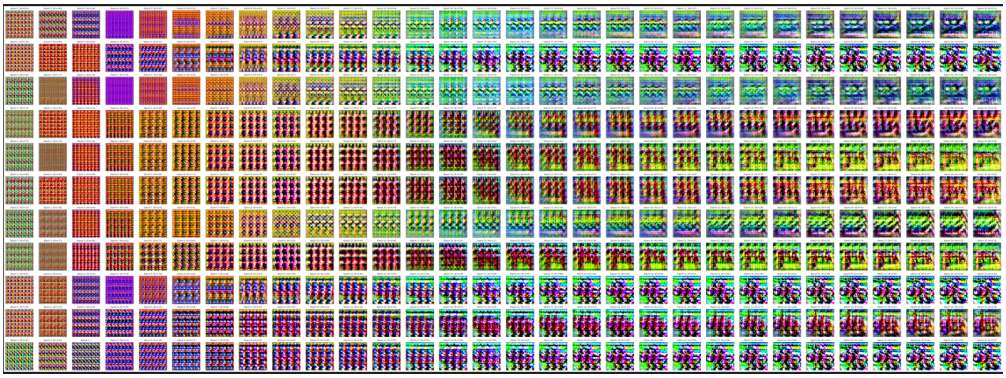
10.3 CIFAR10 as In-dist

10.3.1 Hyperparameters and Settings

- Inversion Loss Weights: $\alpha = 0$, $\beta = 1$, $\gamma = 100$
- Latent Dimension: 100, Batch Size: 32
- Inversion Steps per Epoch: 100
- Number of samples generated in each step = 11000
- Number of Epochs: 41
- Learning Rate:
  Classifier: 1e-4, Generator: 1e-3



10.3.2 Evolution of generated samples across 30 epochs

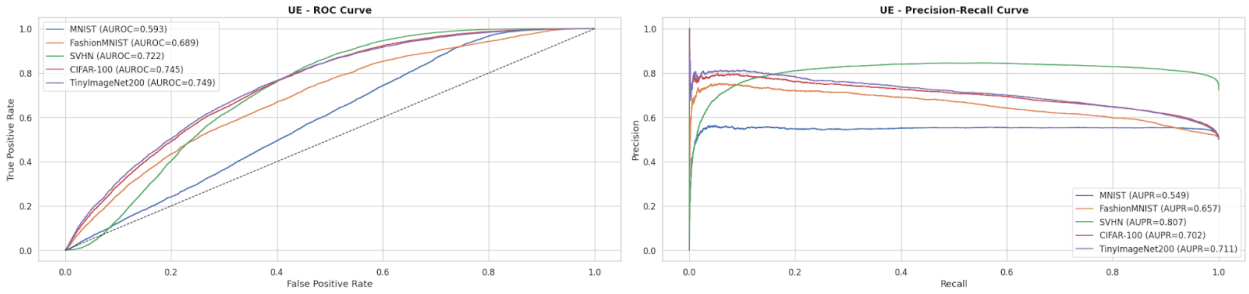


10.3.3 AUPR (↑) , AUPR (↑) , FPRTPR(↓)

COMPREHENSIVE OOD ANALYSIS (Garbage class excluded) - UE

	OOD Dataset	AUROC	AUPR	FPR@95TPR	ID ue Mean	OOD ue Mean	Separation
0	MNIST	0.5925	0.5489	0.7795	0.3521	0.4314	0.0794
1	FashionMNIST	0.6893	0.6574	0.8196	0.3521	0.5437	0.1916
2	SVHN	0.7221	0.8071	0.6079	0.3521	0.5752	0.2232
3	CIFAR-100	0.7451	0.7023	0.6632	0.3521	0.6023	0.2502
4	TinyImageNet200	0.7488	0.7110	0.6774	0.3521	0.6069	0.2549

Avg AUROC = 0.6996 | Avg AUPR = 0.6853 | Avg FPR@95TPR = 0.7095



Comparison of AUROC, AUPR and FPR@95TPR for different OOD detection methods

=== AUROC Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.5925	0.5936	0.6109	0.5417	0.9729
1	FashionMNIST	0.6893	0.6789	0.7461	0.7228	0.8527
2	SVHN	0.7221	0.7247	0.6819	0.7481	0.4810
3	CIFAR-100	0.7451	0.7383	0.7602	0.7539	0.5190
4	TinyImageNet200	0.7488	0.7410	0.7712	0.7670	0.4948

=== AUPR Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.3873	0.3933	0.2738	0.1922	0.9690
1	FashionMNIST	0.7098	0.7051	0.7034	0.4640	0.6412
2	CIFAR10	0.8769	0.8646	0.8924	0.7433	0.3229
3	CIFAR-100	0.8608	0.8493	0.8741	0.7357	0.3309
4	TinyImageNet200	0.8713	0.8598	0.8863	0.7589	0.3131

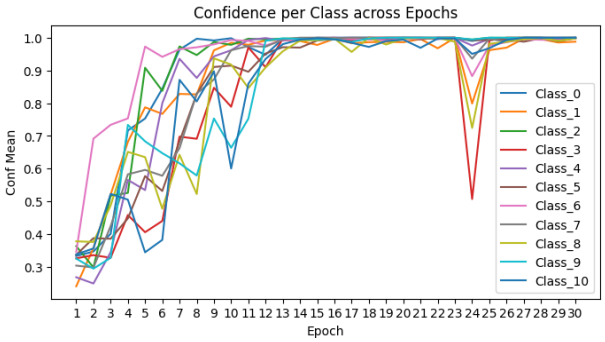
=== FPR@95TPR Comparison ===

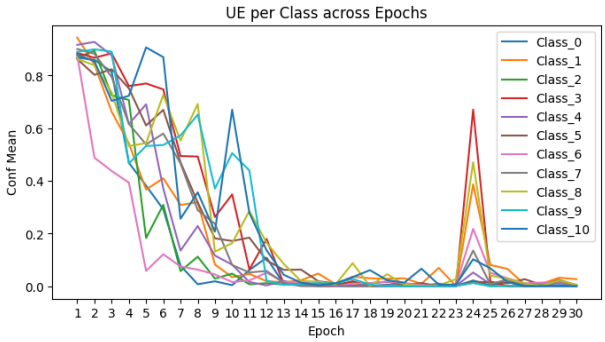
	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.9174	0.9178	0.9715	0.9889	0.0719
1	FashionMNIST	0.5218	0.5218	0.5232	0.6524	0.5760
2	CIFAR10	0.1865	0.1870	0.1532	0.2284	0.8646
3	CIFAR-100	0.2147	0.2151	0.2012	0.2839	0.8734
4	TinyImageNet200	0.2001	0.2006	0.1756	0.2583	0.8843

10.4 SVHN as In-dist

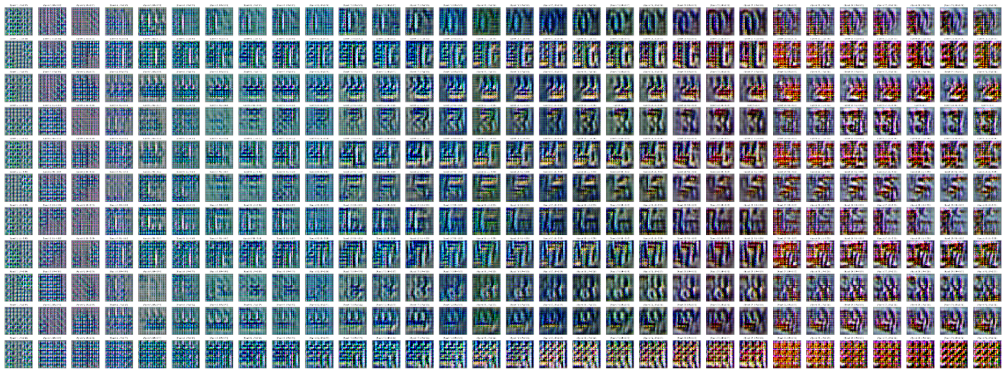
10.4.1 Hyperparameters and Settings

- Inversion Loss Weights: $\alpha = 0$, $\beta = 1$, $\gamma = 10$
- Latent Dimension: 100, Batch Size: 32
- Inversion Steps per Epoch: 90
- Number of samples generated in each step = 11000
- Number of Epochs: 30
- Learning Rate:
  Classifier: 1e-3, Generator: 1e-3





10.4.2 Evolution of generated samples across 30 epochs

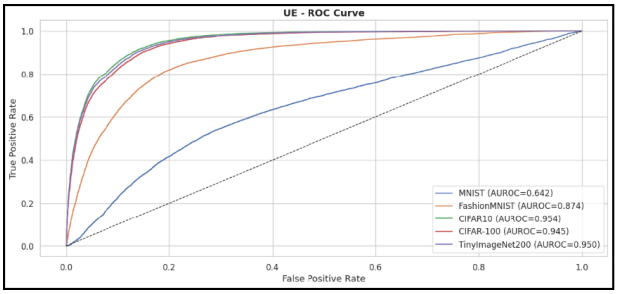


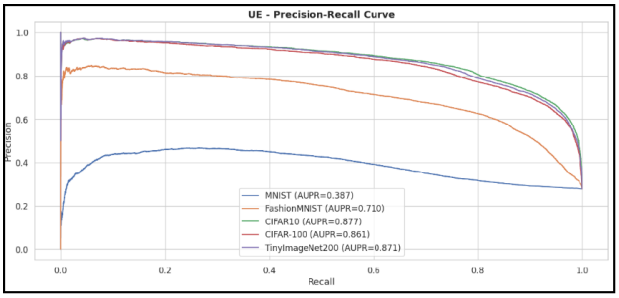
10.4.3 AUPR (↑) , AUPR (↑) , FPRTPR(↓)

COMPREHENSIVE OOD ANALYSIS (Garbage class excluded) - UE

	OOD Dataset	AUROC	AUPR	FPR@95TPR	ID ue Mean	OOD ue Mean	Separation
0	MNIST	0.6425	0.3873	0.9174	0.1044	0.2110	0.1066
1	FashionMNIST	0.8738	0.7098	0.5218	0.1044	0.5187	0.4143
2	CIFAR10	0.9536	0.8769	0.1865	0.1044	0.7237	0.6194
3	CIFAR-100	0.9453	0.8608	0.2147	0.1044	0.6984	0.5940
4	TinyImageNet200	0.9500	0.8713	0.2001	0.1044	0.7144	0.6100

Avg AUROC = 0.8730 | Avg AUPR = 0.7412 | Avg FPR@95TPR = 0.4081





Comparison of AUROC, AUPR and FPR@95TPR for different OOD detection methods

=== AUROC Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.6425	0.6436	0.4794	0.3007	0.9868
1	FashionMNIST	0.8738	0.8725	0.8702	0.7724	0.8383
2	CIFAR10	0.9536	0.9502	0.9611	0.9214	0.5983
3	CIFAR-100	0.9453	0.9422	0.9512	0.9115	0.5952
4	TinyImageNet200	0.9500	0.9469	0.9569	0.9212	0.5758

=== AUPR Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.3873	0.3933	0.2738	0.1922	0.9690
1	FashionMNIST	0.7098	0.7051	0.7034	0.4640	0.6412
2	CIFAR10	0.8769	0.8646	0.8924	0.7433	0.3229
3	CIFAR-100	0.8608	0.8493	0.8741	0.7357	0.3309
4	TinyImageNet200	0.8713	0.8598	0.8863	0.7589	0.3131

=== FPR@95TPR Comparison ===

	OOD Dataset	UE	MSP	ODIN	Energy	Mahalanobis
0	MNIST	0.9174	0.9178	0.9715	0.9889	0.0719
1	FashionMNIST	0.5218	0.5218	0.5232	0.6524	0.5760
2	CIFAR10	0.1865	0.1870	0.1532	0.2284	0.8646
3	CIFAR-100	0.2147	0.2151	0.2012	0.2839	0.8734
4	TinyImageNet200	0.2001	0.2006	0.1756	0.2583	0.8843

11 Conclusion & Future Work

- Across all experiments, network inversion consistently achieved high AUROC, AUPR on OOD datasets
- Future work can also consider the use n garbage classes—one for each of the in-distribution classes—for fine-grained separation of OOD samples and weighted individual OOD sample contribution to the loss while retraining the classifier based on uncertainty

12 References

- Liang, S., Li, Y., & Srikant, R. (2018). Enhancing The Reliability of Out-of-distribution Image Detection in Neural Networks. *ICLR*.
- Liu, W., et al. (2020). Energy-based Out-of-distribution Detection. *NeurIPS*.
- Lee, K., et al. (2018). A Simple Unified Framework for Detecting Out-of-Distribution Samples and Adversarial Attacks. *NeurIPS*.