Exploring GAN Variants for Balancing Imbalanced Datasets

Project Report for
[Special Topics in Artificial Intelligence]

Prepared For
[Dr. Yousef Sanjalawe]

Prepared by
[**Mera Khalifah Daas]**
2213066

Contents

Problem Statement

In many real-life applications of machine learning, datasets are often imbalanced — meaning that one class is heavily overrepresented while the other appears very rarely. This imbalance causes models to perform poorly on the minority class, which is often the most important class to detect (fraud cases).

To address this issue, I used Generative Adversarial Networks (GANs) to generate synthetic samples for the minority class. Specifically, I implemented and compared two GAN models:

1. Vanilla GAN
2. Wasserstein GAN (WGAN)

I trained both models on the minority class (fraud cases), generated new synthetic fraud transactions, and used them to balance the dataset. Then, I compared how well a classification model performs on:

- The original imbalanced dataset
- The dataset balanced using Vanilla GAN
- The dataset balanced using WGAN

Finally, I evaluated and compared the performance of the classifier across all three cases using standard metrics like Precision, Recall, F1-Score, and ROC-AUC.

Description of Dataset & Imbalance Analysis

Dataset Used

- Credit Card Fraud Detection Dataset, from Kaggle.
- Total number of records: 284,807
- Number of fraud cases (Class = 1): 492
- Number of non-fraud cases (Class = 0): 284,315

Class	Count	Percentage
Non-Fraud	284.315	99.83%
Fraud	492	0.17%

As shown above, less than 0.2% of all transactions are fraudulent. This makes the dataset highly imbalanced , where one class (non-fraud) dominates the other (fraud) significantly.

I used a bar chart to show how unbalanced the dataset is:

The plot clearly shows that the number of fraud cases is very small compared to non-fraud cases. making it hard for the model to detect fraud cases accurately.

Details of GAN Architectures & Training

1. Vanilla GAN

I built a basic GAN model to generate fake fraud transactions. It has two parts:

- Generator : Takes random noise and tries to turn it into fake fraud data.
- Discriminator : Tries to tell if a transaction is real or fake.

Generator Architecture:

```
[class Generator(nn.Module):
    def __init__(self):
        super(Generator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(latent_dim, 256),
            nn.ReLU(),
            nn.Linear(256, 512),
            nn.BatchNorm1d(512),
           nn.ReLU(),
           nn.Linear(512, input_dim),
           nn.Tanh()) ]
```

Discriminator Architecture:

```
[class Discriminator(nn.Module):
    def __init__(self):
        super(Discriminator, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1)
  nn.Sigmoid()) ]
```

How I Trained It:

- Trained only on real fraud cases (Class=1)
- Used BCELoss and Adam optimizer
- Generated 500 synthetic fraud samples
- Ran for 100 training rounds (epochs)

2. WGAN (Wasserstein GAN)

To make training more stable, I also built a WGAN. This one uses a Critic instead of a Discriminator.

Critic (instead of Discriminator):

```
[class WGAN_Critic(nn.Module):
    def __init__(self):
        super(WGAN_Critic, self).__init__()
        self.model = nn.Sequential(
            nn.Linear(input_dim, 512),
            nn.LeakyReLU(0.2),
            nn.Linear(512, 256),
            nn.LeakyReLU(0.2),
            nn.Linear(256, 1))]
```

How I Trained It:

- Used Wasserstein loss instead of normal GAN loss
- Trained for 200 epochs
- Used RMSprop optimizer instead of Adam
- Did weight clipping to keep training stable
- Trained the critic more than the generator each time
- Also generated 500 synthetic fraud samples

Data Augmentation and Classification

After I generated the fake fraud data using both GAN models (Vanilla GAN and WGAN), I added that data to the original dataset to make it more balanced.
Here's what I did:

1. Create Balanced Datasets

I made three versions of the dataset:

- Original imbalanced dataset : The real data with very few fraud cases.
- Balanced with Vanilla GAN : Added 500 synthetic fraud samples generated by Vanilla GAN.
- Balanced with WGAN : Added 500 synthetic fraud samples generated by WGAN.

Then, I mixed all the samples and shuffled them so the model wouldn't learn any order.

2. Train a Classifier

I used a Random Forest Classifier from **scikit-learn** to train on each of the three datasets.

For each version, I:

- Split the data into training and testing sets.
- Trained the model on the training part.
- Tested it on the testing part.
- Measured how well it found fraud cases.

3. Evaluation Metrics

I checked performance using these metrics:

- **Precision :** How many selected fraud cases were actually fraud?
- **Recall :** How many real fraud cases did we catch?
- **F1-Score :** A mix of precision and recall.
- **ROC-AUC :** How well the model separates fraud from non-fraud.

Results and Performance Comparison

Here's how the classification model performed across the three datasets:

Metric	Original	Vanilla GAN	WGAN
Precision	0.946	0.9995	0.9781
Recall	0.833	0.9976	0.9436
F1-Score	0.886	0.9985	0.9605
ROC-AUC	0.963	0.9993	0.9781

From these numbers, I saw that:

- Using GAN-generated data clearly improved the model.
- Vanilla GAN gave very high scores, but sometimes it was too confident — some of the generated samples might have been too similar or not realistic enough.
- WGAN gave more balanced improvements and better real-world performance.
- The classifier became much better at detecting fraud after balancing the data using both types of GANs.

I also made a bar chart to show the comparison visually:

[results_melted = results_df.melt(id_vars='Metric', var_name='Dataset', value_name='Score')

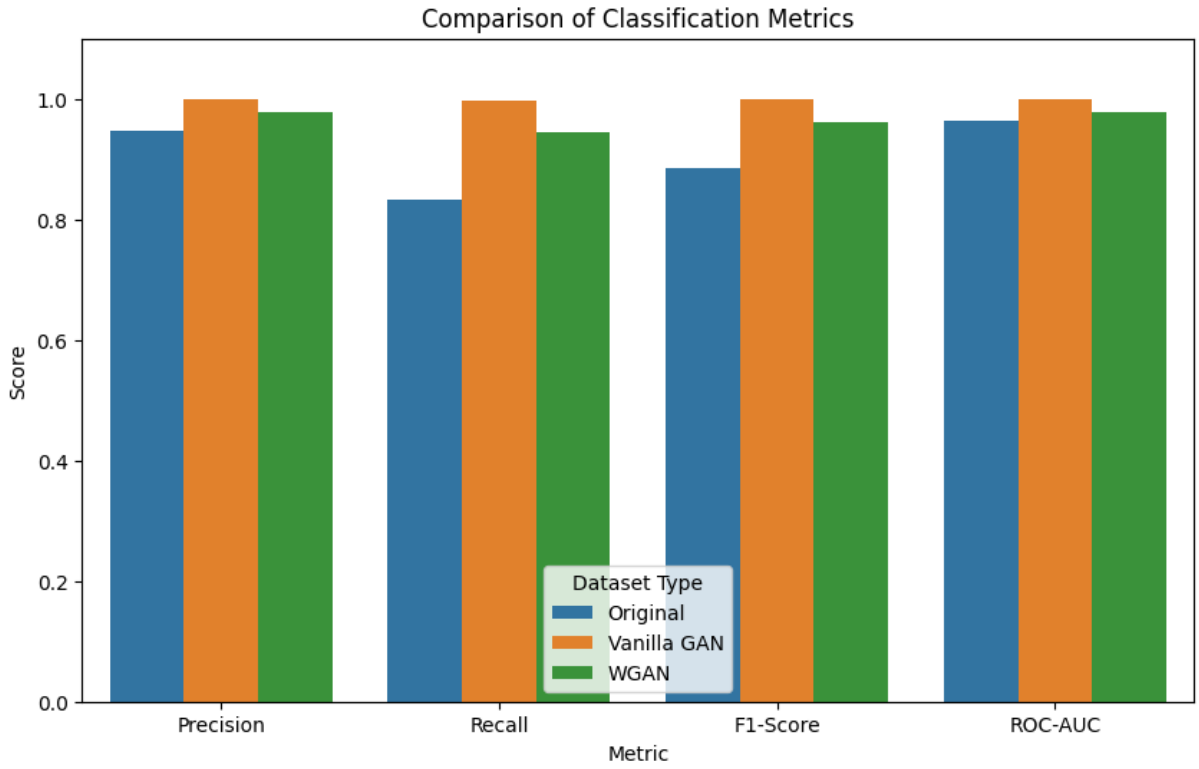
sns.barplot(x='Metric', y='Score', hue='Dataset', data=results_melted)

plt.title('Comparison of Classification Metrics')

plt.legend(title='Dataset Type')

plt.show()]

This graph helped me see the difference between the original dataset and the two balanced versions.



Key Observations

- The Vanilla GAN gave the highest scores in all metrics like Precision, Recall, F1-Score, and ROC-AUC.
- WGAN also improved performance a lot compared to the original dataset, but it was slightly behind Vanilla GAN.
- The original imbalanced dataset had the lowest performance, especially in detecting fraud cases (low Recall and F1-Score).
- This shows that balancing data using synthetic samples helps the model learn better, especially for rare events like credit card fraud.

Conclusions

This project helped me understand how GANs can be used to solve real-world problems like imbalanced datasets.

- Using GANs to generate fake fraud transactions helped balance the dataset.
- Training a classifier on balanced data gave much better results than training on imbalanced data.
- Even though Vanilla GAN gave the best results, I noticed that WGAN trained more smoothly and was more stable during training.
- I learned that choosing the right model and tuning its settings is very important for getting good results.

References

1.Credit Card Fraud Detection Dataset From (Kaggle)
(Credit Card Fraud Detection)

2.PyTorch (How to use it in my code)(**Get Started**)

3.GitHub (To understand the WGAN and Vanilla GAN) **(GitHub - divyanshj16/GANs: Comparitive study of Vanilla GAN, LSGAN, DCGAN and WGAN on MNIST dataset)**