Exploration of Data Synthesis with GANs

Matthew Clinton
University of Massachusetts Amherst
mfclinton@umass.edu

Andrew Teeter
University of Massachusetts Amherst
ateeter@umass.edu

Abstract

In this paper we experiment with using a Generative Adversarial Network (GAN) to generate synthetic training data for a multi-class Classifier. We will show how synthetic data is useful for both anonymizing data as well as improving performance on imbalanced data. We prove this by testing both the case of training a Classifier on a fully synthetic data set and also training on imbalanced data that is augmented with synthetic data. In addition, we also test the impact of directing the noise added to the GAN: which results in some improvements in the quality of the generated data. Our experiments found that these methods are most impactful when applied to imbalanced data using Convolutional networks, and even end up yielding large improvements of performance over the original data. We found the best architecture for the GAN was a Deep Convolutional Conditional GAN (DCCGAN) with directed noise.

1. Introduction

At first glance the internet may seem like a limitless source of data; a place where any information is just a click away. Examining closer we know this is not true. For example, if you wanted to look up someone's health insurance information or their education records you will be unable to do so. This is because we have legislation like HIPPA and FERPA preventing this kind of information from being publicly available. While this is done for good reason, by keeping these data sources private it prevents others from finding valuable insights in the data and stunts potential progress. So that begs the question: how do we provide public access to a data source while ensuring the confidentiality of its subjects? This paper explores one possible solution: which is using GANs to create synthetic data that models a real data source and maintains client anonymity.

1.1. Problem With Synthetic Data

There are two key components of synthetic data: utility and privacy assurance. Utility measures "the degree to which our synthetic data is an accurate proxy for real data" [3, p. 16]. On the other hand, "privacy assurance evaluates the extent to which real people can be matched to records in synthetic data" [3, p. 112]. Essentially, these components can be summarized as measures of usefulness and anonymity. The scenarios we chose to examine are the ones where you have a private data set with limited to no publicly available information (ex. private database with possibly few clients opting in to publicize their information). This is because it closely resembles the treatment of confidential data in real life. From there, the problem we are exploring is synthesizing data under these constraints so that we achieve a good level of usefulness while maintaining anonymity. That way, our useful proxy data can be publicized safely without breaking any confidentiality clauses.

2. Background

2.1. GAN and WGAN Explained

The idea behind Generative Adversarial Nets was first published at the Univerity of Montreal [4]. The basic idea behind a GAN is to train two adversarial neural networks called the Generator and the Discriminator. The goal for Discriminator is to determine the likelihood that a given data sample is from the real data set. The goal for the Generator is to maximize the probability of the Discriminator labeling the generated data as from the real data set. In the ideal case, the generated data would be unique and indistinguishable from the real data resulting in the Discriminator being wrong 50% of the time. This results in the Generator being able to create new data that is similar, but not identical, to the training data.

An iteration on the concept of a GAN published by Martin Arjovsky et al. is the idea of a Wasserstien GAN (WGAN)[1]. The main difference between a WGAN and a Vanilla GAN is that a WGAN uses Earth-Mover (EM) distance as a part loss function, while the Vanilla GAN in our case uses binary cross-entropy (BCE) loss. EM distance can be understood as the cost of moving the mass of one distribution to transform it into another [1]. In the case of a WGAN it is used to measure how much we have to move the output distribution of the Generator in order to trick the

Discriminator. The benefit of using EM distance is that it tends to have more linear gradients versus the disappearing gradients that are common in traditional GANs [1]. Another difference between a WGAN and a Vanilla GAN is that the Vanilla Discriminator outputs a probability, while the WGAN outputs a score.

2.2. WGAN Regularized Gradient

When implementing GANs that utilize Wasserstein loss, we encountered several issues with the original weight-clipping variant [1]. These issues with weight-clipping resulted in unstable WGAN training: "weight clipping in WGAN leads to optimization difficulties, and even when optimization succeeds the resulting critic can have a pathological value surface" [5]. We were able to achieve much more consistent and stable results when using an improved technique for WGANs, where the norm of the Discriminator's gradient acts as a regularization term [5]. All future references to WGANs in this paper use this improved version in their implementation.

2.3. Noise Explanation

A previous study done at The University of Copenhagen found that adding noise to the input samples of our GAN resulted better convergence during training [7, Appendix C]. The reason for this is because the noise acts as a form regularization and reduces the number of possible distribution fits, meaning: "we don't have to worry about over-training the Discriminator, we can train it until convergence" [7, Appendix C]. Inspired by these findings, we came up with the idea of adding class-specific directed noise to our data. We hoped that by doing so we would extenuate the distribution of our classes, thus making the data trend more prominent and lead to faster convergence. In addition, we believed that this technique would encourage our GAN to generate more diverse samples for each class, including more outliers, as well as discouraging GAN mode collapse. Where mode collapse just means little to no variation in the GAN's output. Figure 1 shows an illustration of this idea: where the Vanilla distribution (blue) is a more condensed diagonal line, but after adding directed noise (red) the data becomes slightly more spread out diagonally.

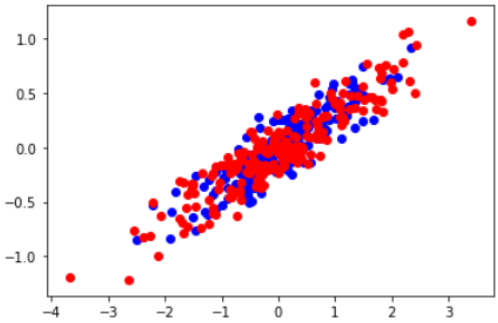


Figure 1: Vanilla Data Distribution (Blue) VS Data with Directed Noise Distribution (Red)

3. Approach

We chose to tackle our problem by breaking it up into two steps: the GAN training step and the classification data evaluation step. We outline the architectures deployed for both steps below.

3.1. GAN Architectures

In our experiments, we chose to evaluate several different GAN implementations: cGAN, cWGAN, cDCGAN, and cWDCGAN. First, the lowercase "c" in front of each of each GAN stands for "conditional," meaning our synthetic data is label-dependent. By setting our GANs up this way, we are able to directly control what class the data is being synthesized for. The ability to condition our synthetic data on the passed in labels is achieved by using various label-embedding techniques. Second, the "DC" in two of our GAN names stands for "Deep Convolutional," meaning those GANs utilize multiple Convolutional layers. Lastly, the "W" refers to "Wasserstein," meaning those GANs use Wasserstein loss as opposed to BCE loss: which is used by the "Vanilla" GANs. The cGan and the cWGAN use the Vanilla Generator and Discriminator; while the cDCGAN and the cWDCGAN use the Convolutional Generator and Discriminator.

3.1.1 Vanilla Generator

First, the Vanilla Generator takes in, as an input, a label and a noise vector. Then, the Generator creates a vector that represents a one-hot embedding of the label. Afterwards, the Generator appends the one-hot embedding to the noise vector and passes the resulting concatenation to the first layer of the neural network. Figure 2 shows the general structure of the network.

[Noise, one-hot class embedding]

↓

(Linear → BatchNorm → LeakyReLU)^3

↓

Linear

↓

Tanh

Figure 2: Vanilla Generator Architecture

In more detail, there are three sections of our network consisting of a Linear layer, followed by Batch Normalization, and followed by Leaky ReLU. The input and output sizes for each section are as follows: **(length of noise vector + number of classes, 256), (256, 512), (512, 1024)**. Each Batch Norm uses a momentum of 0.8, and each Leaky ReLU uses a hinge slope of 0.2. The last Linear layer has a input and output size of **(1024, number of sample features)**, with a Tanh activation layer to keep pixel values in the range [-1, 1].

3.1.2 Vanilla Discriminator

The Vanilla Discriminator takes in a label and the associated data sample. The sample is flattened and appended with a one-hot embedding of its label; the resulting vector is then fed into the network. Figure 3 shows the general structure of the network.

[Flatten(Data Sample), one-hot class embedding]

↓

(Linear → LeakyReLU → Dropout)^3

↓

Linear

↓

Sigmoid*

Figure 3: Vanilla Discriminator Architecture,
***Note:** Sigmoid layer removed when using Wasserstein loss

Specifically, the network is composed of three sections: each containing a Linear layer, then Leaky ReLU (hinge slope = 0.3), then Dropout (p = 0.2). The input and output sizes for each section are the following: **(number of sample features + number of classes, 1024), (1024, 512), (512, 256)**. Then, if we are using BCE loss, the output layer consists of a Linear layer with an input and output size of **(256, 1)** respectively, and the Sigmoid activation function. If, however, we are using Wasserstein loss: then we remove the Sigmoid activation since the output of the Discriminator is no longer a probability.

3.1.3 Convolutional Generator

The Convolutional Generator takes in a label and a noise vector as its input. The original approach for the label embedding was to perform a linear pass on both the noise and a learned embedding of our label such that they can both be reshaped to 2d arrays with the same width and height. Then, these results would be stacked on top of each other as different channels and fed into the network. When run on the MNIST digit dataset, this resulted in learning lower resolution versions of the digits as the label-embedding as shown in Figure 4.

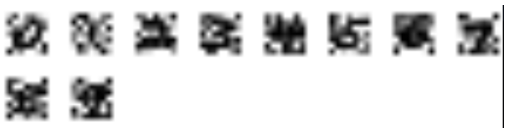


Figure 4: Learned Embedding with 2d Stacking Technique

While this embedding technique usually led to decent results, we occasionally ran into issues where the learned embeddings would be too similar leading to inconsistent results. Better and more consistent results were achieved using a different embedding technique, where the Generator learns a label-embedding the same length as the noise vector, then multiplies the label-embedding element-wise by the noise vector to get the processed input. For these reasons, the final Generators ended up using this multiplication embedding technique. Afterwards, the resulting processed input is fed into the rest of the network, as illustrated in 5.

[preprocess(Noise, label)]

↓

(Linear → LeakyReLU → BatchNorm)

↓

Reshape

↓

(ConvT2d → LeakyReLU → BatchNorm → Dropout)

↓

(ConvT2d → LeakyReLU)

↓

Conv2d

↓

Tanh

Figure 5: Convolutional Generator Architecture

The first section of our network, consisting of a Linear layer followed by Leaky ReLU (hinge slope = 0.2) and Batch Norm, is dedicated to reshaping the processed inputs from the processed input vector to a 14 x 14 array with 128 channels. The Linear layer has an input and output size of **(length of noise vector, 128 * 14 * 14)** respectively.

Afterwards, two different sections of the network, each with a ConvT2d layer, thin out the channels and spread out the width and height. The ConvT2d layers had the following hyper-parameters respectively: **(input channels = 128, output channels = 64, filter size = 4, stride = 2, padding = 1)** and **(input channels = 64, output channels = 16, filter size = 4, stride = 2, padding = 1)**. Each Leaky ReLU in these two sections use a hinge slope of 0.1; and the Dropout in the first section has a probability of .4.

Lastly, the output contains a Convolutional layer, with hyper-parameters **(input channels = 16, output channels = 1, filter size = 4, stride = 2, padding = 1)**, that reduces the number of channels to 1, and brings the width and height to 28 by 28. We used these specific convolution hyper-parameters for our experiments, but you can slightly modify them to generate data of different feature lengths. The final Tanh activation layer just keeps pixel values in the range [-1, 1].

3.1.4 Convolutional Discriminator

The Convolutional Discriminator uses a similar embedding technique as outlined in the **Convolutional Generator** section. The Discriminator learns a label embedding of the same shape as the inputted data sample, and multiplies it element-wise by the data sample. Same as before, the resulting processed input is fed into the Discriminator network shown in Figure 6.

[preprocess(Input Image, label)]
↓
(Flatten → Linear → LeakyReLU → Reshape)
↓
(Conv2d → LeakyReLU → MaxPool2d)^2
↓
Flatten
↓
Linear
↓
Sigmoid*

Figure 6: Convolutional Discriminator Architecture, ***Note:** Sigmoid layer removed when using Wasserstein loss

The first section of our network flattens out the processed input, then puts it through a Linear layer with an input and output size of **(number of sample features, 128 * 7 * 7)** respectively. The result is put through a LeakyReLU (hinge slope = 0.2), and reshaped into an array of shape **(128, 7, 7)**.

The next two sections of our network contain Conv2d layers followed by a LeakyReLU and MaxPool2d layer. The Conv2d layers have the following hyper-parameters respectively: **(input channels = 128, output channels = 64, filter size = 3, stride = 1, padding = 1)** and **(input channels = 64, output channels = 32, filter size = 3, stride = 1, padding = 1)**. Both Leaky ReLUs have a hinge slope of 0.1, and both MaxPool2d layers have a pooling size of 2 with a stride of 1. The original Convolutional Discriminator had no MaxPool2d layers, but this led to a too scrupulous Discriminator; so these layers are essential for best performance.

Finally, the output section of the network consists of flattening the result from the previous layer and feeding it through a Linear layer. The Linear layer has a input size of 800 and an output size of 1. If Wasserstein loss is being used then the last Sigmoid activation is removed since the result of the Discriminator is not a probability in this case.

3.2. Directed Noise Implementation

In order to evaluate the impact of adding directed noise to training data for each GAN architecture, we first have to derive the principal components for subsets of the training data with respect to each label. These principal components point in the direction of maximal variance within our data, so we end up using them to figure out the directional vectors that best capture each class distribution. In addition, we also extract the explained variance ratio for each of the principal components. When adding directed random noise to a sample, we pick several of the top principal components: each acting directional axis. The amount of noise added in that direction is scaled based off a random number in range [-c, c], where c is a chosen scale factor. In addition, the noise on each axis is also scaled by the respective explained variance ration for that principal component; this gives more weight to more important directions for capturing the distribution, as shown in Figure 1.

3.3. Classifier Architecture

As described in the abstract, a Classifier's performance is used as our utility metric to compare the usefulness of the data. The Classifier architecture shown in Figure 7 is a Convolutional Neural Network that uses Leaky ReLU as an activation function. It takes the input sample and then puts it through three sequences of a Conv2d layer followed by a MaxPool2d layer, then the result is flattened into a linear output. Both Convolutional layers have a filter size of 5 and a stride of 1, while the MaxPool2d layers have a pooling size of 2 and a stride of 2. The first Convolutional layer has 32 filters while the second Convolutional layer has 64 filters. The following Linear layer takes takes in the previous flattened result and outputs a vector of size of 1024. The last layer is a Linear output layer has an output size of the number of classes. We chose to use cross entropy loss for the Classifier.

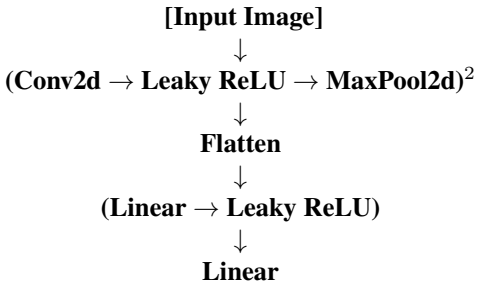


Figure 7: Classifier Architecture

4. Experiment

The optimizers used for both the Generator and Discriminator during training are shown in 1. For our implementation of improved Wasserstein loss [5], we achieved best results with a penalty coefficient of 5. In addition, we also used a noise vector of length 50 for all of our tests: where each element in the vector is pulled from a gaussian distribution with zero mean and unit variance [8]. All GANs are trained on the data set that they are supposed to create data to emulate. This is to avoid having access to more information than the models trained by raw data.

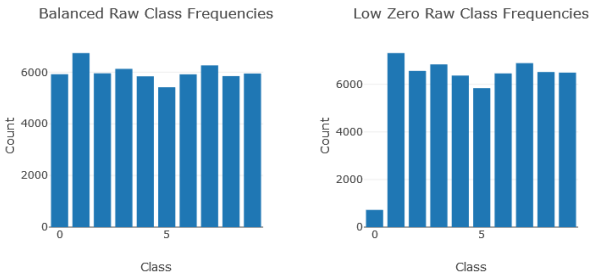
GAN Type	Optimizer
cGAN	Adam(lr=$1e^{-3}$)
cDCGAN	Adam(lr=$1e^{-4}$, weight-decay=$1e^{-5}$)
cWGAN	RMSprop(lr=$1e^{-5}$)
cWDCGAN	Adam(lr=$1e^{-4}$, weight-decay=$1e^{-5}$)

Table 1: Optimizers for GAN Training
Note: Same optimizer used for both Generator and Discriminator

In order to test the effectiveness of our synthetic data, we tested the performance of the Classifier when trained on the data in three different scenarios: raw, augmented, and synthetic. For the raw data run, we train the Classifier normally as if all we have is the original data set. For the augmented run, we use one of our trained GANs to generate additional synthetic data, on top of the existing data, to ensure that every class has as many samples as the majority class. For the synthetic run, we generate 6000 examples for each class and use that as our training set. Each model is trained for 30 epochs with a batch size of 512 in order to allow for fast training times. More robust models can be achieved by using smaller batch sizes, but the relative difference in techniques should remain constant regardless of batch size. All Classifiers used an Adam optimizer with a learning rate of $1e-5$, and all Classifiers are tested on the same test set to allow for meaningful comparison of results.

4.1. Data

All of our data is based off of the MNIST [6] data set, which is a collection of hand written digits from 0 to 9. We decided to use two variants of this data set: the Vanilla one where the classes are relatively balanced, and also a modified one where the zero class is severely under-sampled. These constraints were enforced by using a class-weighted data sampler during training. The class frequencies for each can be seen in Figure 8.



(a) (b)

Figure 8: Class Frequencies for the two data set variants

The Vanilla MNIST data set is chosen to represent data sets that have a relatively "balanced" class distribution. The Low Zero variant is chosen to represent data sets where there is one class that is an outlier and has a much lower frequency. This type of an imbalanced class distribution can be seen in data sets such as the Credit Card Fraud [2]. We had plans to use the Credit Card Fraud DB but it proved to be difficult to train a GAN that produced quality results, so we chose to use the MNIST data set as a proof of concept. We chose to use a balanced test set from the original MNIST data set. This will allow us to use accuracy as a valid metric for measuring the effectiveness of the model. This also ensures that the test examples are representative of the effectiveness of the model on real data and not our generated data. Additionally, in some tests, we evaluated the impact of directed noise on the GANs by temporarily adding noise to the samples during each iteration of training.

4.2. Results

4.2.1 Added Noise

The addition of guided noise to the training of the GANs had positive results on the Convolutional architecture and negative results on the Vanilla GAN architecture when trained on the Vanilla MNIST data set as can be seen in Table 2. The highest performing model was a Convolutional GAN using BCE loss, with added directional noise to its training samples. That model achieved an accuracy of 84% which was an increase of 25% over the model trained without noise. We believe that the directed noise worked

better on the Convolutional architecture because it leads to an exaggeration of our data distribution. This can result in class-defining features becoming more prominent, leading to the Convolutional layers learning filters with stronger activations for these features: thus improving its classification ability. Additionally, in the Convolutional architecture, the image gets smoothed over thus reducing the impact of individual noise, while the Vanilla architecture works on individual pixels and is more prone to the negative effects of noise.

Architecture	Loss	w/o Noise	w/ Noise
Vanilla	BCE	74	57
Vanilla	Wass	79	65
Convolutional	BCE	59	84
Convolutional	Wass	51	67

Table 2: Accuracy of Model Architectures Trained on Vanilla MNIST with and without Noise

4.2.2 Proving Anonymity

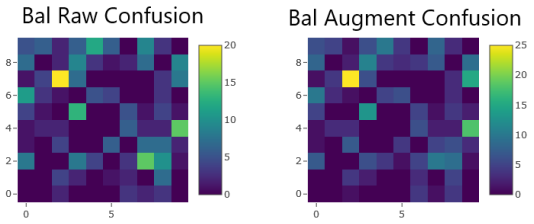
Since privacy is one of the factors we care about, we want a provable way to ensure that each GAN is not just memorizing and outputting the training data. We accomplish this with high confidence by generating a large amount of samples and finding its nearest neighbor within the training data. From evaluating 1000 samples for each class across the different architectures, we found zero matches between the synthetic data and the training data. The average minkowski distance between a generated sample and a real training sample across the tests was roughly 9.2.

4.2.3 Vanilla MNIST

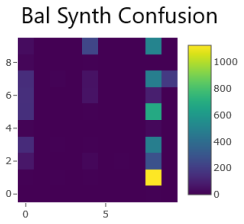
Data	Accuracy
Raw	.97
Augmented	.97
Synthetic	.84

Table 3: Results of the Vanilla data set

As could be expected, the Classifier performs very well on the raw Vanilla data. The augmented results are similar to the raw because the class frequencies are already relatively close to begin with. This tells us that the synthetic data is not strongly swaying the performance of the model. The best model was able to achieve 84% accuracy when trained on the fully synthetic data. This is much better than the random benchmark of 10%, which tells us that a lot of the information from the training data is still carried over to the synthetic data.



(a) Raw Confusion Matrix (b) Augmented Confusion Matrix



(c) Synthetic Confusion Matrix

Figure 9: Confusion matrices when trained on the different amounts of synthetic data for the Vanilla data set.

The confusion matrices for the trained models can be seen in Figure 9. As can be expected, the raw and augmented confusion matrices are very similar. The synthetic confusion matrix shows that the generated data in the 8 class may not be very strong because the Classifier confuses 8 with many other classes- but not with one class in particular. This implies that the generated 8 is a weakness in the synthetic data. Other than that outlier, the synthetic confusion matrix has roughly the same classification levels as the raw confusion matrix.

The comparison of the loss during training shown in Figure 10 shows that the model had a steeper loss curve on the synthetic data which implies that the patterns are more readily apparent in the generated data. This could be why the model has slightly worse performance when trained on the synthetic data because the synthetic data is missing some of the more intricate patterns that can be seen in the raw data.

4.2.4 Low Zero MNIST

The under-sampled 0 class caused the Classifier to have a drastically reduced accuracy of only 56%. Both the augmented and synthetic data were able to boost this accuracy to 75% in the best case as can be seen in Table 4. The augmented data has a much bigger impact in this data variation

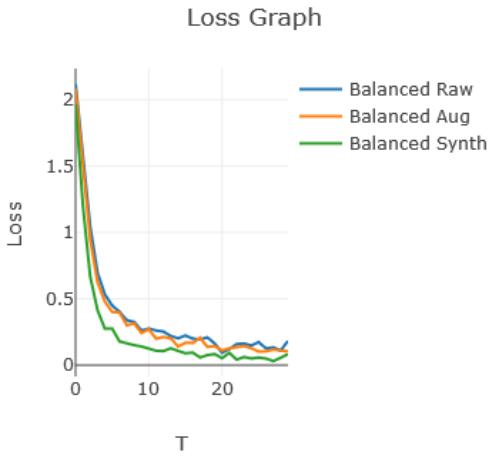


Figure 10: Loss over time for when the Classifier was trained on the Vanilla data

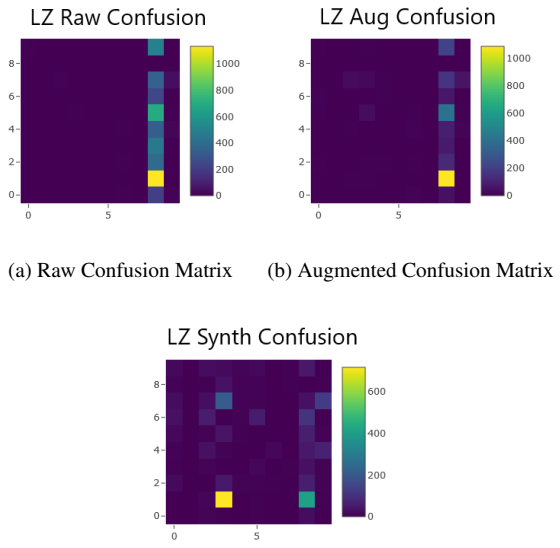
Data	Accuracy
Raw	.56
Augmented	.75
Synthetic	.75

Table 4: Results of the Low Zero data set

than it does in the Vanilla variation because the class imbalance is much bigger. The augmented data consists of around 6000 synthetic examples for the 0 class meaning that the majority of the zero examples are synthetic. The improved performance from the augmented data means that the generated examples of the under-sampled class were able to compensate for the lack of examples in the raw data set.

The confusion matrices shown in Figure 8b show that 8 is commonly misclassified as a variety of other classes. The augmentation of the data allowed for the Classifier to be more confident in differentiating 8s from classes other than 1. Training on fully synthetic data surprisingly improved the performance of the classification of 8 by a lot, but brought the performance of the other classes down: especially focusing on the 3 class. The synthetic data performed better overall than the raw data but this was mainly due to performance increases in the 8 class.

The loss comparison for the Low Zero variant shown in Figure 12 shows an even larger difference in loss curve slope than in the Vanilla variant of the data. This could be due to the fact that there were fewer examples for the zero class leading to even fewer overall patterns being picked up. This would explain why the model trained on the Vanilla synthetic data performed better than the model trained on the Low Zero synthetic data. It would make sense that the GAN, given access to higher quality data, would be able to produce higher quality synthetic data as a result.



(a) Raw Confusion Matrix (b) Augmented Confusion Matrix

(c) Synthetic Confusion Matrix

Figure 11: Confusion matrices when trained on the different amounts of synthetic data for the Low Zero data set.

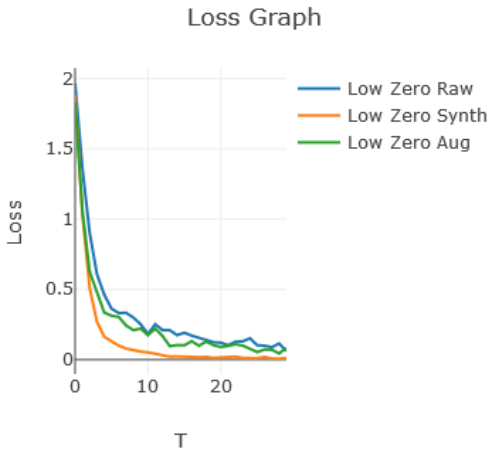


Figure 12: Loss over time for when the Classifier was trained on the Low Zero data

5. Conclusion

Our findings conclude that while the use of synthetic data was most effective in the case of augmenting the imbalanced data set where we saw utility gains, but we were also able to achieve minimal loss of utility in comparison to the vanilla data set. A future study might involve Credit Card Fraud detection [2], since that data set represents potentially anonymous data and is imbalanced. It must be noted that despite the varying accuracy results in some cases, each GAN ended up generating visually pleasing MNIST results. Reasons for this disconnect could be that the GAN collapsed into a specific version of a class and did not have enough variation in its output, or just did not capture all the original data's complexities. Aside from utility, we establish high confidence in data anonymity by showing no generated samples is equivalent to an existing real sample. Finally, we found that directed random noise holds the potential to improve the results of Convolutional GANs, but does not yield the same effect for Vanilla architectures. Overall, these experiments show the kind of results you should expect when dealing with synthetic data in different scenarios, and suggests different architectures and techniques for data synthesis.

References

[1] M. Arjovsky, S. Chintala, and L. Bottou. Wasserstein gan, 2017.

[2] A. Dal Pozzolo, G. Boracchi, O. Caelen, C. Alippi, and G. Bontempi. Credit card fraud detection: A realistic modeling and a novel learning strategy. *IEEE Transactions on Neural Networks and Learning Systems*, PP:1–14, 09 2017.

[3] K. Emam, L. Mosquera, and R. Hoptroff. *Practical Synthetic Data Generation: Balancing Privacy and the Broad Availability of Data*. O'Reilly Media, Incorporated, 2020.

[4] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative adversarial networks, 2014.

[5] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. C. Courville. Improved training of wasserstein gans. *CoRR*, abs/1704.00028, 2017.

[6] Y. LeCun and C. Cortes. MNIST handwritten digit database. 2010.

[7] C. K. Sønderby, J. Caballero, L. Theis, W. Shi, and F. Huszár. Amortised MAP inference for image super-resolution. *CoRR*, abs/1610.04490, 2016.

[8] T. White. Sampling generative networks, 2016.