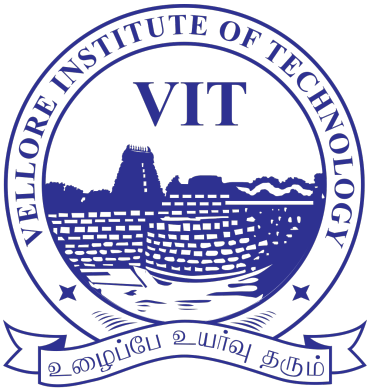
DIGITAL ASSIGNMENT
School of Computer Science and Engineering

Course Code: BCSE318L
Course Name: DATA PRIVACY

AKSHIT GOYAL - 21BCI0152



Under the kind guidance of
DEEPIKA J - SCOPE

School of Computer Science and Engineering
Vellore Institute of Technology, Vellore
MAR, 2024

Contents

1 Introduction to Generative Adversarial Network

Generative Adversarial Networks (GANs) are a class of machine learning models used in unsupervised learning tasks, particularly for generating new data that resembles a given dataset. GANs were introduced by Ian Goodfellow and his colleagues in 2014. Attachment.

1.1 Key Components

1. **Generator:** The generator in a GAN is a neural network that takes random noise as input and attempts to generate data that resembles the target dataset. It learns to produce increasingly convincing outputs through training.

2. **Discriminator:** The discriminator, another neural network, distinguishes between real data from the target dataset and fake data generated by the generator. It aims to improve its ability to discern real from fake samples.

1.2 Working Principle

The generator tries to create data that is indistinguishable from real data, while the discriminator tries to get better at telling real from fake. They engage in a two-player minimax game:

- The generator aims to minimize the probability that the discriminator correctly classifies generated data as fake.

- The discriminator aims to maximize this probability, essentially becoming a more effective detector.

1.3 Training Process

1. **Initialization:** Both the generator and discriminator start with random weights.

2. **Training Iterations:**

    - The generator creates fake data from random noise.
    - The discriminator is trained on a mix of real data from the target dataset and fake data from the generator, adjusting its parameters to improve discrimination.
    - The generator uses the discriminator's feedback to refine its output to appear more genuine.
    - This process continues iteratively, with the models improving and competing until the generator generates data that is convincing and similar to real data.

1.4 Applications

1. **Image Generation:** GANs can generate high-resolution, realistic images, enabling applications in art creation, face generation, and more.

2. **Style Transfer:** GANs can transform images in a specific style, useful for artistic effects or adapting the style of one image to another.

3. **Data Augmentation:** GANs can augment datasets by generating additional data, which helps improve the robustness and performance of machine learning models.

4. **Drug Discovery:** GANs can assist in generating molecular structures for potential drugs, accelerating the drug discovery process.

5. **Anomaly Detection:** GANs can learn the normal patterns of data and identify anomalies, aiding in fraud detection or cybersecurity.

2 Minimax Game

A minimax game is a fundamental concept in game theory, often used in decision-making and strategy analysis. It's a two-player, zero-sum game, meaning what one player gains, the other loses, resulting in a total payoff of zero. The goal of each player is to minimize their maximum possible loss.

In the context of Generative Adversarial Networks (GANs), the training process can be formulated as a minimax game. The generator and discriminator are the two players.

2.1 Generator (Player 1)

- Its objective is to generate data (e.g., images) that closely resemble real data from the target dataset.
- The generator aims to maximize the discriminator's error rate, making it difficult for the discriminator to distinguish between real and generated data.

2.2 Discriminator (Player 2)

- Its objective is to classify real and generated data correctly.
- The discriminator aims to minimize its error rate by effectively distinguishing between real and generated data.

The generator and discriminator are in a constant loop of improvement and competition. The generator gets better at generating realistic data to fool the discriminator, while the discriminator improves its ability to differentiate real and fake data. This iterative process continues until the generator produces data that is indistinguishable from real data. Mathematically, the minimax objective function in this context is:

$$\min_G \max_D V(D,G) = \mathbb{E}_{x\sim p_{\text{data}}(x)}[\log D(x)] + \mathbb{E}_{z\sim p_z(z)}[\log(1 - D(G(z)))]$$

where:

- D is the discriminator's function,
- G is the generator's function,
- $p_{\text{data}}(x)$ is the distribution of real data,

- $p_z(z)$ is the distribution of noise (randomly generated input for the generator).

The generator tries to minimize this value, while the discriminator aims to maximize it. Minimax games are widely used in various domains, from artificial intelligence to economics, and provide a powerful framework for strategic decision-making and competitive scenarios.

2.3 Optimal Discriminator and Generator

In an ideal scenario, the optimal Discriminator (D^*) and Generator (G^*) are as follows:

2.3.1 Best Discriminator (D^*)

The optimal Discriminator is defined by:

$$D^*(x) = \frac{p_{\text{data}}(x)}{p_{\text{data}}(x) + p_g(x)}$$

where:

- $p_{\text{data}}(x)$ is the distribution of real data.
- $p_g(x)$ is the distribution of generated data.

2.3.2 Best Generator (G^*)

The optimal Generator perfectly replicates the true data distribution:

$$G^*(z) = p_{\text{data}}(x)$$

where:

- z is a random noise vector.
- $p_{\text{data}}(x)$ is the distribution of real data.

3 Neural Networks and GANs

The backward pass is a critical step in training neural networks and Generative Adversarial Networks (GANs). It involves calculating gradients of the loss function with respect to the model parameters using backpropagation. These gradients are crucial for updating the weights and biases to minimize the loss function and improve the performance of the models.

3.1 Steps of the Backward Pass

The steps of the backward pass in a neural network, including GANs, typically involve:

1. **Loss Computation:** Compute the loss (L) between the model's predictions (y) and the ground truth (y_{true}) using an appropriate loss function (e.g., mean squared error for regression, binary cross-entropy for GANs):

$$L = \text{Loss}(y, y_{\text{true}})$$

2. **Gradient Calculation:** Calculate the gradient of the loss with respect to the model's output ($\frac{\partial L}{\partial y}$) using calculus and the chosen loss function.

3. **Backpropagation:** Use the chain rule of calculus to propagate the gradients backward through the network, computing the gradient of the loss with respect to each parameter at each layer.

4. **Parameter Updates:** Use an optimization algorithm (e.g., stochastic gradient descent, Adam) to update the weights (W) and biases (b) of the model in the opposite direction of the computed gradients, aiming to minimize the loss:

$$W_{\text{new}} = W_{\text{old}} - \alpha \frac{\partial L}{\partial W}, \quad b_{\text{new}} = b_{\text{old}} - \alpha \frac{\partial L}{\partial b}$$

where α is the learning rate.

3.2 Weight and Bias Updates in GANs

In GANs, the generator and discriminator are trained using adversarial learning. During the backward pass:

- **Generator Update:** Gradients for the generator are obtained by backpropagating the loss from the discriminator through the generator. The generator's weights (W_{gen}) and biases (b_{gen}) are updated to produce data that better fools the discriminator.

- **Discriminator Update:** Gradients for the discriminator are obtained by backpropagating the loss from the real and generated data. The discriminator's weights (W_{disc}) and biases (b_{disc}) are updated to improve its ability to distinguish between real and generated data.

The iterative process of forward and backward passes, along with weight and bias updates, continues until the models converge and achieve the desired performance.

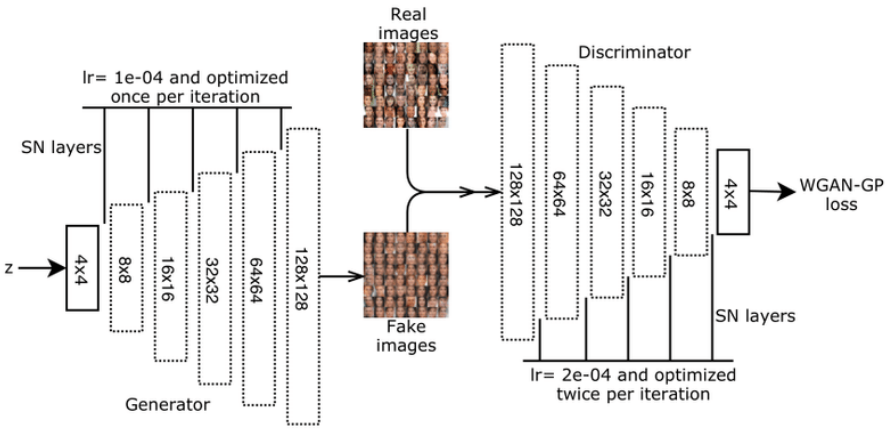


Figure 1: *GAN Architecture*

The figure 1 shows the architecture of Generative Adversarial Network (GAN).

3.3 First toy model in Pytorch to learn the framework

Linear regression is a fundamental technique in machine learning and statistics. It aims to model the relationship between a dependent variable (target) and one or more independent variables (features). In this documentation, we explore the principles and concepts of linear regression using PyTorch.
Attachment to the complete implementation.

3.3.1 Linear Regression

Linear regression models the relationship between the target variable y and one or more input features x. In its simplest form, it is represented as:

$$y = wx + b$$

Where:

- y: the target variable.
- x: the input feature.
- w: the weight or coefficient.
- b: the bias or intercept.

In multivariate linear regression, there are multiple input features, and the equation becomes more complex.

3.3.2 Linear Regression with PyTorch

To implement linear regression in PyTorch, we follow a series of key steps, including data preparation, model definition, making predictions, training the model, and model evaluation. These steps are explained in detail in the following sections.

3.3.3 Loss Function and Optimizer

Set up a loss function to quantify the error between predicted and actual values (e.g., mean squared error or L1 loss). Create an optimizer (e.g., SGD) to update the model's parameters during training.

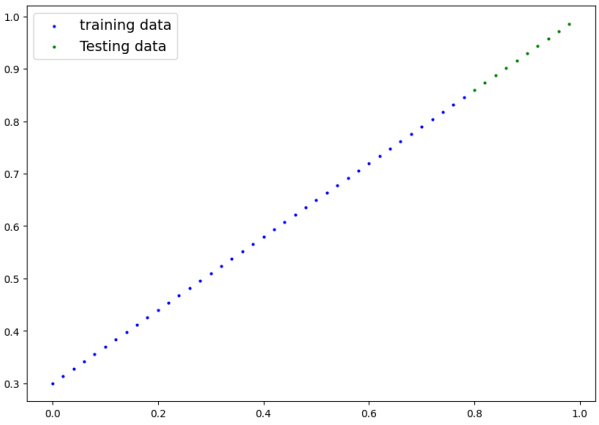


Figure 2: *Linear Regression*

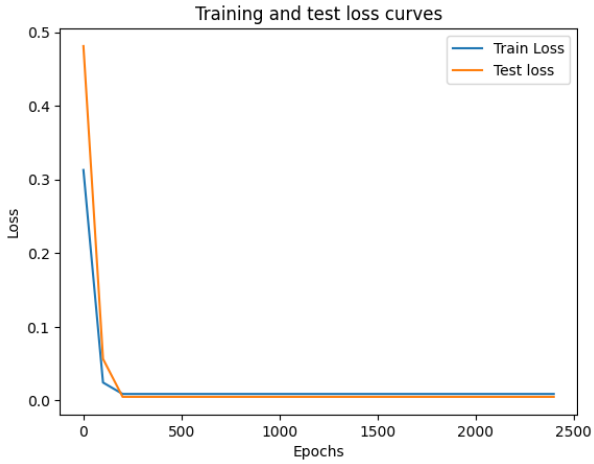


Figure 3: *Linear Regression Epochs*

3.3.4 Applications of Linear Regression

1. **Prediction**: Linear regression is commonly used for prediction tasks. It can be applied to predict stock prices, weather conditions, sales, and other future outcomes based on historical data.

2. **Understanding Relationships**: Linear regression helps in understanding the relationships between variables. By examining the model's coefficients, one can identify which factors have the most significant impact on the target variable.

3. **Risk Assessment**: In finance and insurance, linear regression is used to assess and quantify risk. For instance, it can determine the risk associated with lending money to individuals based on their financial characteristics.

4. **Quality Control**: Linear regression is employed in quality control to analyze and improve manufacturing processes. It helps identify factors affecting product quality.

5. **Economics and Social Sciences**: Economists and social scientists use linear regression to model economic relationships, such as the impact of GDP on unemployment rates or the relationship between education and income.

6. **Medical Research**: Linear regression is used to analyze the impact of variables like age, weight, and lifestyle on health outcomes. For example, it can help understand the correlation between smoking and lung cancer.

7. **Marketing and Sales**: Businesses use linear regression to assess the effectiveness of marketing campaigns and pricing strategies. It helps in understanding how changes in advertising budgets, pricing, or product features affect sales.

8. **Real Estate**: In the real estate industry, linear regression is used to estimate property values based on factors like location, size, and the number of bedrooms and bathrooms.

9. **Environmental Science**: Linear regression models can be applied to understand environmental phenomena, such as the relationship between temperature and energy consumption or pollution levels and health outcomes.

10. **Education**: Linear regression is used to analyze educational data to identify factors that influence student performance. For example, it can help determine the impact of class size on student achievement.

11. **Sports Analytics**: Linear regression is used in sports analytics to analyze player performance, team dynamics, and the impact of various factors on game outcomes.

12. **Resource Allocation**: Linear regression is used in resource allocation decisions, such as budget allocation in project management or supply chain optimization.

3.4 Second Toy Model: Sine Curve Training Pytorch

Attachment to the complete implementation.

3.4.1 Neural Network Architecture

The code defines a simple feedforward neural network with three layers:

- The input layer has a single neuron, corresponding to the input feature (independent variable), representing positions on the x-axis.

- The hidden layer consists of 64 neurons and employs the Rectified Linear Unit (ReLU) activation function. This layer captures complex patterns in the data.

- The output layer contains a single neuron, responsible for providing the predicted value, which represents the output of the sine function.

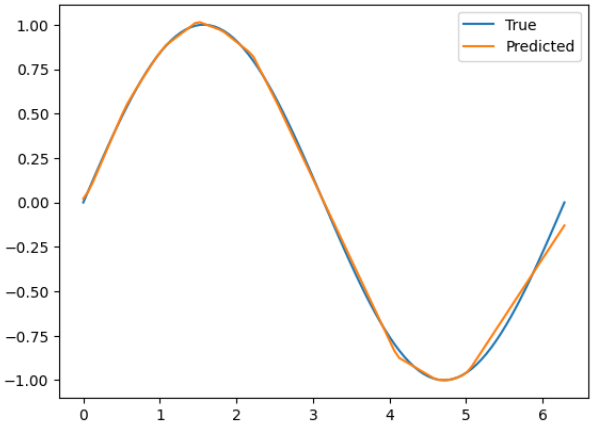


Figure 4: *Sine curve after training*

3.4.2 Loss Function and Optimizer

The Mean Squared Error (MSE) loss function is used to measure the difference between the predicted values and the ground truth (actual sine values). MSE is a common choice for regression tasks, aiming to minimize the squared differences between predictions and actual values. Stochastic Gradient Descent (SGD) serves as the optimization algorithm, with a learning rate set to 0.001. SGD adjusts the model's parameters during training to minimize the loss function.

The input data, denoted as x, is generated by uniformly sampling 100 points between 0 and 2π. These points represent positions on the x-axis. The ground truth values, denoted as y, are calculated by applying the sine function to the input data. These are the target values that the model should learn to predict.

3.4.3 Training Loop

The training loop iterates for a specified number of epochs (200,000 in this case). In each epoch, the following steps are performed:

- Forward Pass: The input data x is passed through the neural network to obtain predictions.
- Loss Computation: The Mean Squared Error (MSE) loss is computed by comparing the predicted values to the ground truth y.
- Backward Pass and Optimization: The optimizer backpropagates the loss through the network, computing gradients and updating the model's parameters using SGD.

3.4.4 Outcome

The neural network architecture consists of an input layer, a hidden layer with ReLU activation, and an output layer. Using the Mean Squared Error (MSE) loss function and Stochastic Gradient Descent (SGD) optimization, the network learns to predict the sine function's values from the input data. As the training progresses through 200,000 epochs, the loss decreases, indicating that the network is increasingly approximating the sine wave. The final outcome is visualized with a plot showing the true sine curve and the predicted curve, with the latter expected to closely match the former, demonstrating the network's ability to capture the underlying sine pattern.

3.5 Third Toy Model: Classification Pytorch Model

Attachment to the complete implementation.
We will explore the theoretical aspects of creating a PyTorch model for classification. We will discuss the key concepts and steps involved in building, training, and evaluating a classification model.

3.5.1 Data Generation and Preprocessing

To begin the process of building a classification model, we often start with data generation and preprocessing. This involves obtaining or generating a dataset suitable for classification tasks. The dataset typically consists of input features and corresponding labels. In our case, we have generated synthetic data using the `make_circles` function from the `sklearn.datasets` module.

3.5.2 Data Visualization

Before building a model, it is essential to visualize the dataset. Data visualization helps in gaining insights into the distribution of data points and the relationships between features. In our example, we create a scatter plot using matplotlib to visualize the data points, with colors representing class labels.

3.5.3 Data Transformation and Splitting

Once the data is visualized, we need to transform it into a suitable format for machine learning models. In PyTorch, data is often represented as tensors. We convert our data into PyTorch tensors and split it into training and test sets using `train_test_split` from `sklearn.model_selection`.

3.5.4 Building a Neural Network Model

The core of our classification model is a neural network. In PyTorch, we can define neural network architectures using either a subclass of `nn.Module` or `nn.Sequential`. The model architecture typically consists of input layers, hidden layers, and output layers. The choice of architecture depends on the complexity of the classification problem.

3.5.5 Loss and Optimizer

In order to train a classification model, we need to define a loss function and an optimizer. The loss function quantifies the model's performance by measuring the error between predicted and actual labels. For binary classification, `nn.BCEWithLogitsLoss` is commonly used. The optimizer, such as stochastic gradient descent (SGD), is responsible for updating the model's parameters to minimize the loss.

3.5.6 Training the Model

Training the model involves iterative steps of forward pass, loss calculation, backpropagation, and parameter optimization. During each epoch, the model makes predictions on the training data, computes the loss, and updates its parameters. This process continues for multiple epochs to improve the model's performance.

3.5.7 Visualizing the Model

To assess the model's performance and visualize its decision boundaries, we can create visualizations of the model's predictions. Visualizing the decision boundaries can help us understand how well the model separates different classes in the input space.

3.5.8 Model Improvement Strategies

To enhance the model's performance, various strategies can be employed. These include adjusting the neural network architecture, increasing the number of hidden layers, training for a longer duration, altering activation functions, and experimenting with different hyperparameters. The goal is to iteratively improve the model's accuracy in classifying data. These theoretical aspects provide an overview of the steps and considerations involved in building a classification model in PyTorch.

4 Updated Process to Run unmodified CT-GAN

4.1 Prerequisites

To successfully run Bézier-GAN, ensure you have the following prerequisites:

- Python 3.6
- Virtual environment
- TensorFlow version 1.15.5
- NumPy
- Protocol Buffers (protobuf) version 3.20.0
- Matplotlib
- scikit-learn

4.2 Steps

Follow these steps to run CT-GAN:

4.2.1 Create a Virtual Environment

To create a virtual environment with Python 3.6, use the following command:

```
python3.6 -m venv venv
```

Activate the virtual environment based on your platform:
On Windows:

```
.\Activate.ps1
```

On other platforms:

```
<venv_directory>/bin/activate
```

4.2.2 Install Required Packages

Install the necessary packages within the virtual environment:

```
pip install tensorflow==1.15.5
pip install numpy
pip install protobuf==3.20.0
pip install matplotlib
pip install scikit-learn
```

4.2.3 Navigate to the CT-GAN Directory

Change your current directory to the CT-GAN directory:

```
cd CTgan
```

4.2.4 Create a New Directory

Create a new directory named ”CTgan2”:

```
mkdir CTgan2
```

4.2.5 Change Directory

Change your current directory to ”CTgan2”:

```
cd CTgan2
```

4.2.6 Run the GAN Training Script

Execute the GAN training script with the specified parameters:

```
python train_gan.py
```

This command will train a CT-GAN. The trained model and synthesized shape plots will be saved under the directory trainedgan.”

4.3 Additional Notes

Here are some additional considerations:

- If you are using a different development environment, such as VSCode, make sure you have selected the correct Python kernel (Python 3.6).
- You can experiment with different latent and noise dimensions to achieve different results.

5 Translation to run in latest version of TensorFlow 2.x

The transition from TensorFlow 1 to TensorFlow 2 marked a substantial shift in its usage and functionality, particularly evident in the implementation of generative models like CT-GANs.

1. **Eager Execution by Default**
   One of the most notable changes in TensorFlow 2 is the introduction of eager execution as the default mode. TensorFlow 1 relied heavily on the use of sessions and placeholders to create a computation graph that was compiled and then run. This approach, while efficient for certain types of computation, made the code verbose and somewhat difficult to debug due to its static nature.

   In TensorFlow 2, eager execution allows operations to be evaluated directly and immediately reflects the results. This change is particularly beneficial for developing generative models like CTGANs, as it simplifies the debugging process and makes the code more intuitive. Eager execution enables a more interactive and dynamic coding environment, akin to standard Python programming.

2. **Gradient Tape for Custom Training Loops**
   The introduction of the GradientTape API in TensorFlow 2 is a significant shift from the way gradients were handled in TensorFlow 1. In TensorFlow 1, gradients were typically computed using static computation graphs, which could be cumbersome for complex models. GradientTape, however, offers a more flexible approach to automatic differentiation, which is crucial for training generative models like CTGANs.

   With GradientTape, you can record operations for automatic differentiation. When training a CTGAN, which involves a generator and a discriminator with complex loss functions, GradientTape makes it easier to customize and experiment with different training loops and gradient updates. This flexibility is particularly useful for balancing the training of the generator and discriminator, a common challenge in GAN training.

3. **Keras API Integration**
   TensorFlow 2 has integrated the Keras API as its official high-level API, replacing the tf.layers module from TensorFlow 1. This integration has significantly streamlined the process of building and training models. In the context of CTGANs, this means that you can leverage Keras's user-friendly and modular API to construct both the generator and discriminator networks with ease.

   Keras provides a wealth of pre-built layers, optimizers, and utilities, making it more straightforward to implement complex architectures required for CTGANs. Additionally, Keras's Sequential and Functional APIs offer a more intuitive approach to model building, allowing for clear and concise code when designing the intricate layers of a generative network.

4. **Simplified Model Saving and Loading**
   TensorFlow 2 has simplified the process of saving and loading models, which is particularly useful for generative models like CTGANs, where the training process can be time-consuming, and preserving intermediate states is crucial. TensorFlow 2's tf.saved_model provides a unified format for saving a model's architecture, weights, and configuration, both for TensorFlow and Keras models.

   This change means that you can easily save the entire state of your CTGAN at any point during training and resume from that state later. This is particularly useful for iterative

and experimental development processes common in GAN training, where you might need to fine-tune or adjust models based on intermediate results.

This improvement means that feeding data into a CTGAN for training becomes more streamlined and efficient, allowing for better utilization of hardware resources and potentially speeding up the training process.

Overall, TensorFlow 2 provides a more pythonic, user-friendly approach to deep learning, with specific benefits for building and training advanced generative models like CTGANs. Its focus on eager execution, integration with Keras, and improved APIs for gradients, model saving, and data handling significantly enhance the development experience and open up new possibilities for innovation and experimentation in the field of generative adversarial networks.

6 CTGAN: A Solution for Data Privacy

6.1 Introduction

CTGAN, or Conditional Tabular Generative Adversarial Network, is a powerful deep learning technique used to generate synthetic tabular data. It operates similarly to Generative Adversarial Networks (GANs), learning the statistical properties of real data to create new, realistic data points that mimic these characteristics.

6.2 Use Cases for Data Privacy

1. **Sharing Data for Research and Collaboration:**
    - CTGAN enables the creation of anonymized versions of sensitive datasets, facilitating sharing among researchers and institutions without compromising individual privacy.
    - Valuable in sectors like healthcare and finance where data sensitivity is critical.
2. **Training Machine Learning Models:**
    - When real data is scarce or unavailable, CTGAN can generate synthetic data for training and testing machine learning models.
    - Essential for developing models in areas such as fraud detection and risk assessment.

6.3 Advantages of CTGAN

1. **Preserves Statistical Properties:**
    - CTGAN effectively captures relationships between different data points, ensuring generated data maintains the same statistical properties as the original dataset.
2. **Handles Diverse Data Formats:**
    - Versatile tool capable of working with various data types, including continuous, discrete, and categorical variables.
3. **Improves Data Efficiency:**

- Alleviates data scarcity issues by generating synthetic data, enabling more efficient training and testing of machine learning models.

6.4 Drawbacks of CTGAN

1. **Model Complexity:**
    - Implementation and training of CTGAN models require expertise in deep learning and can be computationally expensive, especially for large datasets.
2. **Potential for Bias:**
    - If training data is biased, the generated synthetic data may also inherit those biases, impacting fairness and validity in downstream applications.
3. **Explainability Challenges:**
    - Understanding the rationale behind CTGAN's generated data can be difficult, raising concerns about interpretability and potential bias in the synthetic data.

6.5 Other Uses of CTGAN

1. **Data Augmentation:**
    - Increases the size and diversity of existing datasets, improving machine learning model performance by addressing issues like class imbalances.
2. **Data Simulation:**
    - Simulates various data scenarios for testing and evaluating machine learning models in controlled environments.

6.6 Industry Use Cases

1. **Healthcare:**
    - CTGAN creates synthetic patient data for research and development of new medical treatments while protecting patient privacy.
2. **Finance:**
    - Generates synthetic financial data for fraud detection and risk assessment systems, safeguarding sensitive customer information.
3. **Retail:**
    - Creates synthetic customer data for analyzing buying patterns and developing personalized marketing strategies while safeguarding customer privacy.

6.7 Algorithm

Algorithm 1: Train CTGAN on step.

Input: Training data $\mathbf{T}_{train}$, Conditional generator and Critic parameters Φ_G and Φ_C respectively, batch size m, pac size pac.
Result: Conditional generator and Critic parameters Φ_G, Φ_C updated.

1 Create masks $\{\mathbf{m}_1, \ldots, \mathbf{m}_{i^*}, \ldots, \mathbf{m}_{N_d}\}_j$, for $1 \leq j \leq m$
2 Create condition vectors $cond_j$, for $1 \leq j \leq m$ from masks ▷ Create m conditional vectors
3 Sample $\{z_j\} \sim \texttt{MVN}(0, \mathbf{I})$, for $1 \leq j \leq m$
4 $\hat{\mathbf{r}}_j \leftarrow \texttt{Generator}(z_j, cond_j)$, for $1 \leq j \leq m$ ▷ Generate fake data
5 Sample $\mathbf{r}_j \sim \texttt{Uniform}(\mathbf{T}_{train}|cond_j)$, for $1 \leq j \leq m$ ▷ Get real data

6 $cond_k^{(pac)} \leftarrow cond_{k\times pac+1} \oplus \ldots \oplus cond_{k\times pac+pac}$, for $1 \leq k \leq m/pac$ ▷ Conditional vector pacs
7 $\hat{\mathbf{r}}_k^{(pac)} \leftarrow \hat{\mathbf{r}}_{k\times pac+1} \oplus \ldots \oplus \hat{\mathbf{r}}_{k\times pac+pac}$, for $1 \leq k \leq m/pac$ ▷ Fake data pacs
8 $\mathbf{r}_k^{(pac)} \leftarrow \mathbf{r}_{k\times pac+1} \oplus \ldots \oplus \mathbf{r}_{k\times pac+pac}$, for $1 \leq k \leq m/pac$ ▷ Real data pacs
9 $\mathcal{L}_C \leftarrow \frac{1}{m/pac}\sum_{k=1}^{m/pac} \texttt{Critic}(\hat{\mathbf{r}}_k^{(pac)}, cond_k^{(pac)}) - \frac{1}{m/pac}\sum_{k=1}^{m/pac} \texttt{Critic}(\mathbf{r}_k^{(pac)}, cond_k^{(pac)})$

10 Sample $\rho_1, \ldots, \rho_{m/pac} \sim \texttt{Uniform}(0, 1)$
11 $\tilde{\mathbf{r}}_k^{(pac)} \leftarrow \rho_k \hat{\mathbf{r}}_k^{(pac)} + (1 - \rho_k)\mathbf{r}_k^{(pac)}$, for $1 \leq k \leq m/pac$
12 $\mathcal{L}_{GP} \leftarrow \frac{1}{m/pac}\sum_{k=1}^{m/pac}(||\nabla_{\tilde{\mathbf{r}}_k^{(pac)}} \texttt{Critic}(\tilde{\mathbf{r}}_k^{(pac)}, cond_k^{(pac)})||_2 - 1)^2$ ▷ Gradient Penalty
13 $\Phi_C \leftarrow \Phi_C - 0.0002 \times \texttt{Adam}(\nabla_{\Phi_C}(\mathcal{L}_C + 10\mathcal{L}_{GP}))$

14 Regenerate $\hat{\mathbf{r}}_j$ following lines 1 to 7
15 $\mathcal{L}_G \leftarrow -\frac{1}{m/pac}\sum_{k=1}^{m/pac} \texttt{Critic}(\hat{\mathbf{r}}_k^{(pac)}, cond_k^{(pac)}) + \frac{1}{m}\sum_{j=1}^{m} \texttt{CrossEntropy}(\hat{\mathbf{d}}_{i^*,j}, \mathbf{m}_{i^*})$
16 $\Phi_G \leftarrow \Phi_G - 0.0002 \times \texttt{Adam}(\nabla_{\Phi_G}\mathcal{L}_G)$

6.8 Python Code

```
# -*- coding: utf-8 -*-
"""CTGAN-Insurance.ipynb
# Setup
"""

!pip install ctgan

!pip install table_evaluator

!gdown 1CFLXcella3VFDzPJRGrfkGlYNOhf1RCQ

import pandas as pd
data = pd.read_csv('./insurance.csv')

"""Next, we define a list with column names for categorical variables. This
    list will be passed to the model so that the model can decide how to
   process these fields."""

categorical_features = ['age','sex','children','smoker','region']

"""#Model training

Next, we simply define an instance of CTGANSynthesizer and call the fit
   method with the dataframe and the list of categorical variables.
```

```
22
23 We train the model for 300 epochs only as the discriminator and generator
       loss becomes quite low after these many epochs.
24 """
25
26 from ctgan import CTGAN
27
28 ctgan = CTGAN(verbose=True)
29 ctgan.fit(data, categorical_features, epochs = 300)
30
31 """#Synthetic data generation"""
32
33 samples = ctgan.sample(1000)
34
35 samples
36
37 """#Evaluation"""
38
39 from table_evaluator import TableEvaluator
40
41 print(data.shape, samples.shape)
42 table_evaluator =  TableEvaluator(data, samples, cat_cols=
       categorical_features)
43
44 table_evaluator.visual_evaluation()
```

Listing 1: *Implementation Python code*

6.9 Visualizing and comparing the generated data and the real data for similarities

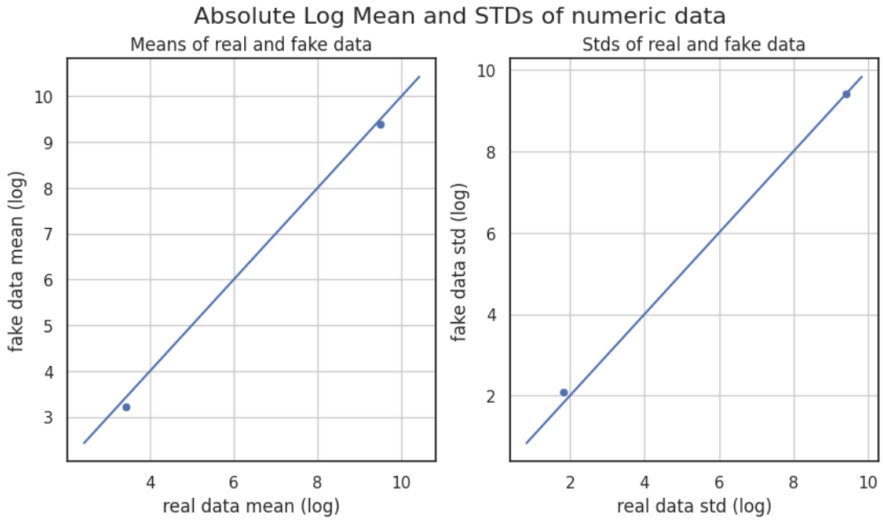


Figure 5: *Absolute Log Mean and STDs of numeric data*

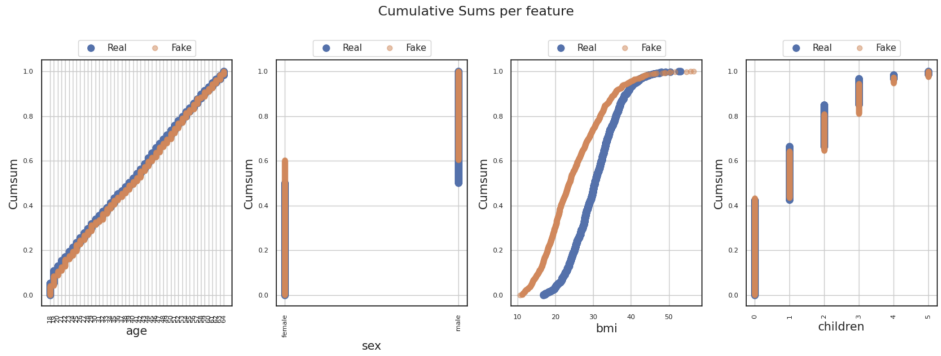


Figure 6: *Cumulative Sums per feature*

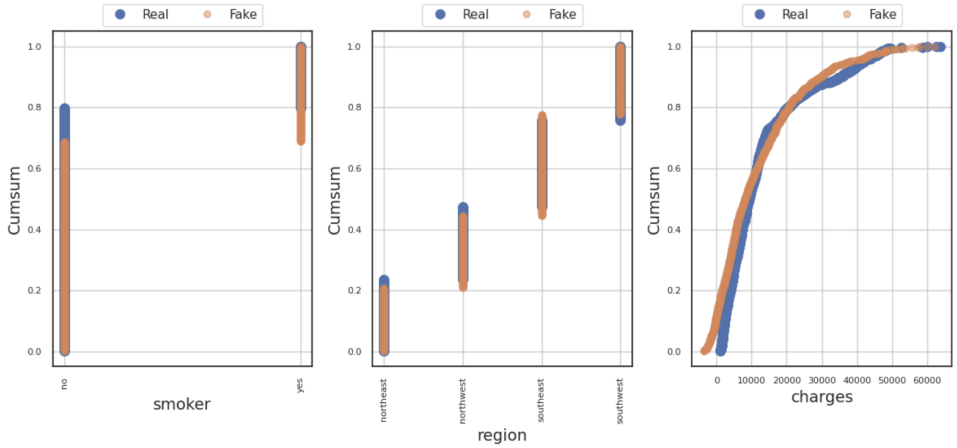


Figure 7: *Similarity comparison*

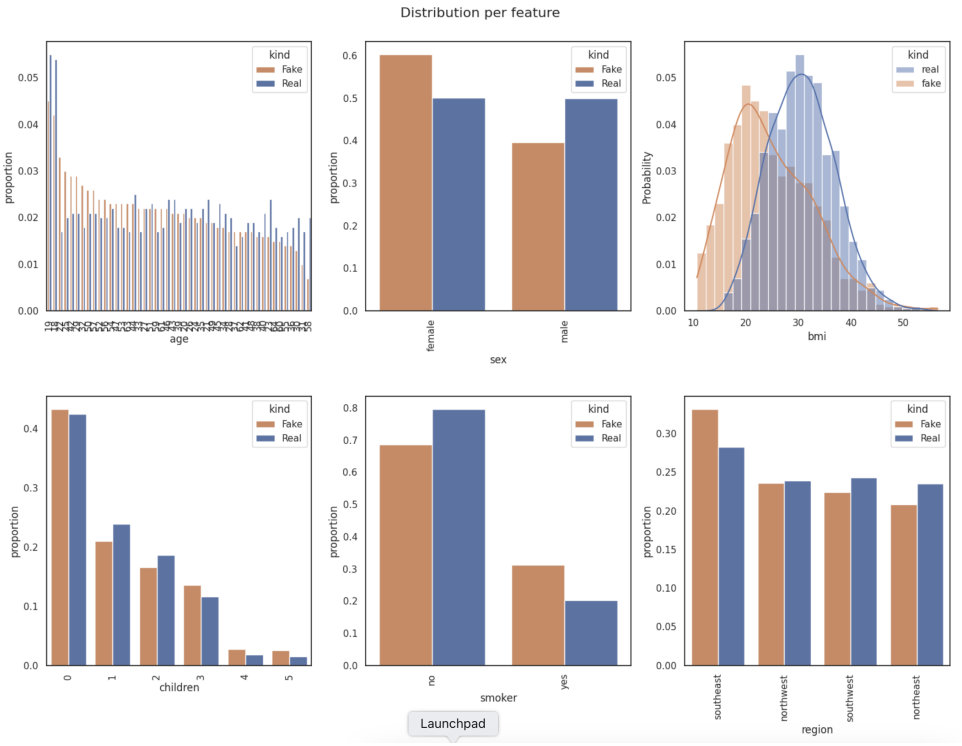


Figure 8: *Distribution per feature*

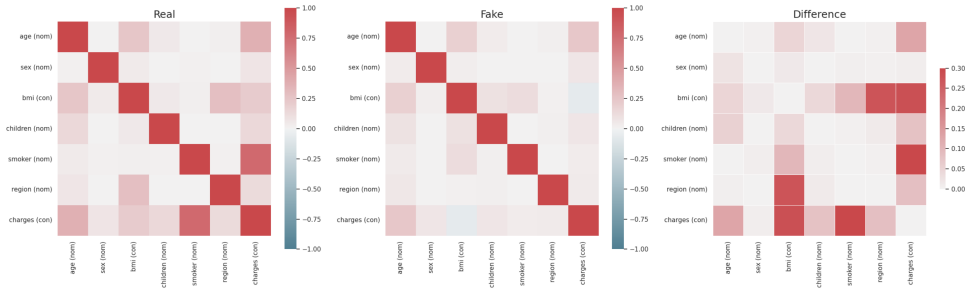


Figure 9: *Confusion matrix*

6.10 Conclusion

CTGAN emerges as a promising solution tailored to address the intricate challenges inherent in tabular data. Despite its limitations in model complexity, potential bias, and explainability, CTGAN's prowess in generating realistic and diverse synthetic tabular data holds invaluable potential for data anonymization, machine learning advancement, and various other applications within the realm of structured data analysis.

In our pursuit of a flexible and robust model capable of learning the intricate distributions prevalent in tabular data, we uncovered a significant observation: existing deep generative models faltered in comparison to Bayesian networks, particularly those employing a greedy approach to discretize continuous values. Recognizing the unique properties of tabular data and the shortcomings of existing methodologies. Through empirical evaluations, we substantiated that CTGAN outperforms Bayesian networks in learning tabular data distributions.

7 Related Work

In this section, we discuss related work. Some notable works include those by (1), (2), (3), (4), and (5).

References

[1] Lei Xu, Maria Skoularidou, Alfredo Cuesta-Infante, and Kalyan Veeramachaneni, *Modeling Tabular data using Conditional GAN*, In Proceedings of Advances in Neural Information Processing Systems, 2019.

[2] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu, *Seqgan: Sequence generative adversarial nets with policy gradient*, In AAAI Conference on Artificial Intelligence, 2017.

[3] Jun Zhang, Xiaokui Xiao, and Xing Xie, *Privtree: A differentially private algorithm for hierarchical decompositions*, In International Conference on Management of Data. ACM, 2016.

[4] Jun Zhang, Graham Cormode, Cecilia M Procopiuc, Divesh Srivastava, and Xiaokui Xiao, *Privbayes: Private data release via bayesian networks*, ACM Transactions on Database Systems, 42(4):25, 2017.

[5] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros, *Unpaired image-to-image translation using cycle-consistent adversarial networks*, In International Conference on Computer Vision, pages 2223–2232. IEEE, 2017.