Assignment 4 - Report

Shahar Oded, 208388918
Nir Rahav, 316275437

הקמה של ה-TabularDataset:

אנחנו מתעסקים פה עם דאטה טבלאי, פשוט יחסית, עבורו אנחנו יודעים לתכנן מראש את הערכים הייחודיים בכל עמודה ואת הסוג שלהם. נציג מידע סיכומי על הדאטה:

```
[Info]: Number of rows: 32561
[Info]: Number of columns: 15

[Info]: Target class distribution:
income
b'<=50K'    0.75919
b'>50K'     0.24081

[Info]: Categorical columns: 9
[Info]: Continuous columns: 6
[Info]: Number of columns after one-hot encoding: 109 == Embedding Vector Dimension (with 1 label column)
```

בסינון זה, ערכים ייחודיים בעמודות שלא מופיעים בדאטה עצמו אינם מיוצגים בווקטור. מכיוון שאנו עוסקים בהתפלגות הנתונים בפועל, זה לא מפריע לנו. דאגנו לסמן get_dummies(...,drop_first=False על מנת שגם עמודות קטגוריאליות בוליאניות יהפכו ל2 עמודות ולא ל-1. זה משפיע לאחר מכן על האופן בו נייצג מידע קטגוריאלי מג'ונרט.

אנחנו רואים פה imbalance בעמודת ה-target אבל לא משהו מאוד חריג. נרצה לבצע stratified split כשנעסוק בחלקים מהדאטה הזה. כמו כן נרצה לאפשר ביצוע אוגמנטציה באמצעות oversampling עם הרעשה רנדומית של המשתנים הנומריים (בעד כדי 0.05) על מנת להוסיף קצת גיוון בערכים בזמן האוגמנטציה ולנסות למנוע מצב של mode collapse בו ה-generator לומד ייצוגים של ה-majority class בלבד.
כמו כן, את הערכים הקטגוריאליים נתרגם באמצעות one-hot-encoder לכניסות נפרדות, ואת הערכים הנומריים ננרמל באמצעות MinMaxScaler. את עמודת ה-target נציג בתור כניסה בודדת עם "1" עבור 'b'>50K' ו"0" עבור 'b'<=50K.
יצרנו בנוסף **TabularDatasetFromArrays** משני שתפקידו לקחת אובייקט מהראשי אחרי ביצוע Stratified Split ולהפוך חזרה לאובייקט אותו ניתן להפוך ל-dataloader.
היות ונשתמש בפונקציית TanH כדי לייצר את הוקטור היוצא מה-generator באקטיבציה הסופית, העברנו את כל הנתונים להיות בטווח של [-1,1].

אחרי שימוש בפונקציית החלוקה אנחנו רואים שימור של היחסים בתוך כל label:

```
[Main]: Class distributions after split:
Train (26048 labels): 0 -> 0.759175
                      1  -> 0.240825

Test (6513 labels): 0  ->  0.759251
                    1  ->  0.240749
```

ואם אנחנו בוחרים להשתמש באוגמנטציה באמצעות SMOTE:

```
[Main]: Class distributions after split:
Train (39550 labels):  1  ->  0.5
                       0  -> 0.5
Test (6513 labels): 0  ->  0.759251
                    1  ->  0.240749
```

הוספנו האפשרות לחלוקה גם ל-val_set מתוך ה-train (לפני אוגמנטציה). בפועל, נדרשנו בזה רק כאשר אמנו AE.

על מנת לטפל בבעיה עתידית, בה המודל לא יוכל לתת ערכים קטגוריאליים (אלא רק רציפים), שמרנו באובייקט גם מיפוי של כל האינדקסים של עמודות one-hot לעמודות הרלוונטיות שלהם, כך שפונקציית generate תוכל לעשות להם עיגול לערך 1 או -1 בהתאמה, עבור הערך המקסימלי בכל קבוצה (עמודה קטגוריאלית לשעבר).
כך נוכל לג'נרט רשומות אמינות יותר, לפחות בהיבט הזה. על מנת שיעבוד, מיפינו ערכים חסרים ('?') לערך 'Unknown', ואלו מהווים עוד אפשרות לערך בקטגוריות הרלוונטיות.

הגדרת מודל GAN:

בדומה לעבודות קודמות, רצינו להשאיר את הקונפיגורציה של השכבות בתור פרמטר חיצוני וברור המגדיר לכל שכבה גדלים, אקטיבציות, batch norm ו-dropout. ניתן לראות את הקונפיגורציות בהן בחרנו ב-config.py:

```
# Model Config
DATA_DIM = 108 # The size of the feature vector
NOISE_DIM = 32  # The size of the initial noise vector

## Generator Configuration
GENERATOR_CONFIG = [
    {"input_dim": NOISE_DIM, "output_dim": 64, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 64, "output_dim": 128, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 128, "output_dim": 256, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 256, "output_dim": 512, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.3},
    {"input_dim": 512, "output_dim": 256, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 256, "output_dim": DATA_DIM, "activation": nn.Tanh(), "batch_norm": False, "dropout": 0.0},  # No dropout on
the last layer
]

## Discriminator Configuration
DISCRIMINATOR_CONFIG = [
    {"input_dim": DATA_DIM, "output_dim": 64, "activation": nn.LeakyReLU(0.2), "batch_norm": False, "dropout": 0.2},
    {"input_dim": 64, "output_dim": 32, "activation": nn.LeakyReLU(0.2), "batch_norm": False, "dropout": 0.2},
    {"input_dim": 32, "output_dim": 1, "activation": nn.Sigmoid(), "batch_norm": False, "dropout": 0.0},  # No dropout on the
output layer
]
```

בדומה לתרגול בחרנו להשתמש בפונקציות אקטיבציה LeakyRelu, כך שה-output של ה-generator יהיה באמצעות TanH, ועבור ה-discriminator בחרנו להשתמש ב-output של sigmoid הרי שהבעיה היא בינארית. רצינו לייצר ארכיטקטורה מורכבת יותר עבור ה-generator שתגדיל בהדרגה את הייצוג של ווקטור הרעש ההתחלתי המוזן לו ולבסוף תתכווץ לגודל DATA_DIM (מבנה פירמידה). עבור ה-discriminator רצינו לייצר ארכיטקטורה פשוטה יותר שתשרת משימת קלאסיפיקציה יחסית "קלה". בהתאם לתוצאות האימון נשקול שינויים במודלים.

המודל מקבל וקטור רעש בגודל NOISE_DIM בגודל 32 וממנו מג'נרט את הדגימה. בחרנו ברעש הקטן מ-DATA_DIM לאור המלצה, על מנת שזה יתרחב בהדרגה לאורך שכבות המודל.

הפרמטרים של המודל מוגדרים גם הם חיצונית:

```
APPLY_AUGMENTATION = False # Apply augmentation on minority classes when stratified split is called.
BATCH_SIZE = 128
VAL_RATIO = 0.0     # Ratio out of the training dataset
TEST_RATIO = 0.2    # Ratio out of the full dataset
SEED = 42   # Change the seed and check influence on the model

# Training Config
LEARNING_RATE = 5e-4    # Initial learning rate
WEIGHT_DECAY = 1e-5
EARLY_STOP = 50     # Stop after |EARLY_STOP| epochs with no improvement in the total loss
WARMUP_EPOCHS = 50  # Define a number of warmup iterations in which the model won't count towards an early stop.
EPOCHS = 500    #   A high number of epochs, hoping for an early stopping
GENERATOR_UPDATE_FREQ = 1   # Number of G updates per D updates, to balance their losses.
```

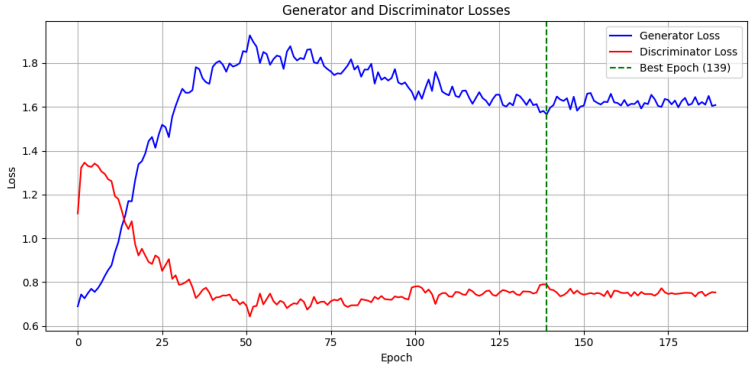
המודל מאמן את שני חלקיו יחד בתוך פונקציית train, כאשר הגדרנו פרמטר GENERATOR_UPDATE_FREQ המאפשר לאמן את הגנרטור יותר מאשר את הדיסקרימינטור במקרה של שיפור מואץ שלו על פניו. התנסנו עם פרמטר זה אך ללא המון הצלחה ניכרת. לתחושתנו, ניסוי זה גרם בכלל לפערים יותר משמעותיים בין השניים.

הגדרנו בתהליך האימון לשמור בכל נקודת שיפור את המשקלים של המודל הטוב ביותר על מנת שנוכל לשחזר המודלים מבלי לאמן אותם מחדש. המודל הטוב ביותר נקבע ע"י ערכי ה-loss של ה-Generator, היות והמטרה היא לייצר דגימות מזויפות משכנעות, ולכן נקבע שיפור ע"י ירידה ב-loss שלו. עם זאת, בתהליך האימון נרצה להיות ערניים לירידת ה-loss של ה-discriminator כדי לוודא שהוא בעצמו מצליח ללמוד להבחין בין הווקטורים, ונשאר יחסית מאוזן, ולכן לפני שהוא מאומן דיו, לא לקחנו את המודל הטוב ביותר כ-checkpoint (לפחות עד סוף ה-warm_up). הגדרנו גם עצירה מוקדמת אם אין שיפור בערך זה אחרי 50 אפוקים, כך שבפועל כל מודל יתאמן לפחות 100 epochs, אבל יתכן וימצא את האופטימום שלו אחרי האיטרציה ה-50, או מתישהו בהמשך. עצירה כפויה אחרי 500 איטרציות.

המודל עובד עם אופטימייזר ADAM עבור שני המודלים, עם LR ו-W2 כמופיע בתמונה ועם פונקציית loss מסוג BCELoss לאור האופי הבינארי של הבעיה. מספר האפוקים הסופי של כל מודל יופיע בתיעוד האימון שלו. לאור ירידת ה-loss הלא יציבה לא ראינו לנכון לממש scheduler בארכיטקטורה זו. בניסיון נוסף לאזן עוד יותר בין המודלים חקרנו W2 רק עבור הדיסקרימינטור, עם LR הקטן פי 10 מזה של הג'נרטור, אבל לא הצלחנו להצביע באמצעות זה על שיפור.

תהליך האימון:

אפשר לראות שבהתחלה קל מאוד ל-generator להטל ב-discriminator ולכן ערכי ה-loss שלו טובים יותר. לאחר מספר אפוקים ה-discriminator מבין את בעיית הסיווג ומתחיל להשתפר בה. אנחנו רצינו לאפשר למודל לעצור בנקודת מינימום לוקאלית של ה-generator אחרי ששני המודלים כבר למדו להגיב אחד לשני ושה-loss יחסית יציב. למרות ההתחלה היחסית קשה, נראה כאילו אחרי כ-100 אפוקים ה-loss ההדדי שלהם נכנס למגמת התייצבות.




```
[Training Status]: Epoch 1: Generator Loss: 0.7351, Discriminator Loss: 1.1036
[Training Status]: Epoch 2: Generator Loss: 0.7505, Discriminator Loss: 1.3043
[Training Status]: Epoch 3: Generator Loss: 0.7176, Discriminator Loss: 1.3592
[Training Status]: Epoch 4: Generator Loss: 0.7282, Discriminator Loss: 1.3425
[Training Status]: Epoch 5: Generator Loss: 0.7483, Discriminator Loss: 1.3446
[Training Status]: Epoch 6: Generator Loss: 0.7587, Discriminator Loss: 1.3431
[Training Status]: Epoch 7: Generator Loss: 0.7593, Discriminator Loss: 1.3415
[Training Status]: Epoch 8: Generator Loss: 0.7724, Discriminator Loss: 1.3283
[Training Status]: Epoch 9: Generator Loss: 0.7996, Discriminator Loss: 1.3119
[Training Status]: Epoch 10: Generator Loss: 0.8027, Discriminator Loss: 1.3086
.
.
.
.
.
[Training Status]: Epoch 140: Generator Loss: 1.5841, Discriminator Loss: 0.8315
Model saved to Trained Models\gan\best_model.pth
.
.
.
.
[Training Status]: Epoch 190: Generator Loss: 1.6173, Discriminator Loss: 0.7763
[Training Status]: Early stopping triggered!
```

הגדרת מודל cGAN:

על מנת לעשות הסבה של המודל הקודם הגדרנו מחלקה יורשת בה יש פרמטר נוסף, NUM_CLASSES. האופי המודולרי של הקוד אפשר לנו באמצעות שינויים מינימליים (למשל הגדלת ה-input dimention של השכבה הראשונה) להשתמש באותו המבנה. בפונקציית ה-train היינו צריכים לדאוג להעביר לשני חלקי המודל לצד הוקטור z את ה-label האמיתי של הדגימה, המועברת ע"י ה-dataloader (ומקודם פשוט לא נעשה בה שימוש).

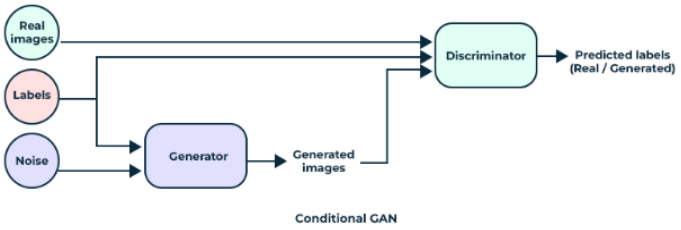
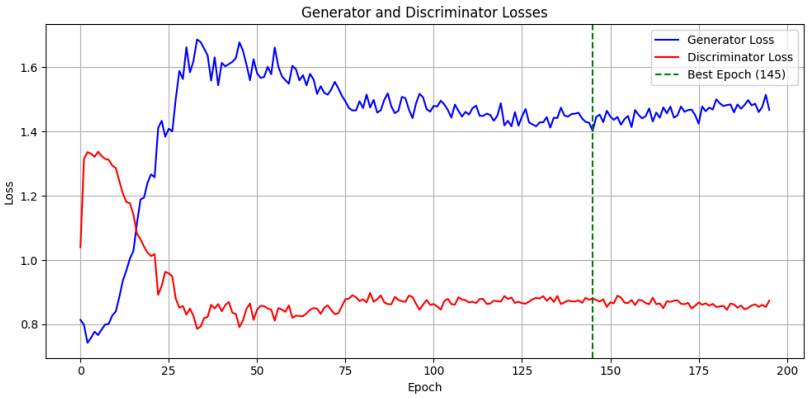


Figure 1: the conditional GAN architecture

בניגוד למודל הקודם, שאם היה מג'נרט דגימות לפי מחלקה 1 או 0 זה היה כתלות בלבד בהתפלגות הפיצ'רים שלהם במרחב, המודל הזה מנסה ללמוד את ההתפלגות של כל מחלקה בנפרד, ואז מייצר דגימות יותר מפוקסות למחלקה זו, לפי דרישה.

תהליך האימון:

שמנו לב שמודל זה לוקח יותר איטרציות עד שמגיע ל-early stop, מפה אנחנו מניחים שהוספת האינפורמציה עוזרת לו בביצועים, ומייצרת מצב "פינג-פונג" יותר מעניין בין ביצועי ה-generator לביצועי ה-discriminator. גם פה נשמר הפער בביצועי ה-loss של שניהם, אך נראה שהוא יחסית מתון יותר מה-GAN ונשאר יציב לאורך זמן - סממן חיובי.



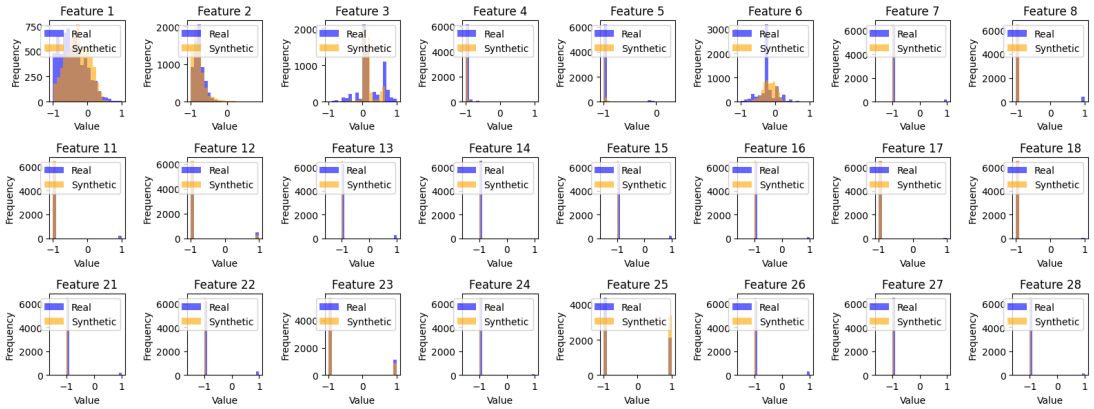

```
Epoch 1: Generator Loss: 0.8140, Discriminator Loss: 1.0400
Epoch 2: Generator Loss: 0.7986, Discriminator Loss: 1.3144
Epoch 3: Generator Loss: 0.7426, Discriminator Loss: 1.3361
Epoch 4: Generator Loss: 0.7584, Discriminator Loss: 1.3308
Epoch 5: Generator Loss: 0.7772, Discriminator Loss: 1.3216
Epoch 6: Generator Loss: 0.7668, Discriminator Loss: 1.3375
Epoch 7: Generator Loss: 0.7841, Discriminator Loss: 1.3240
Epoch 8: Generator Loss: 0.7996, Discriminator Loss: 1.3150
Epoch 9: Generator Loss: 0.8016, Discriminator Loss: 1.3123
Epoch 10: Generator Loss: 0.8273, Discriminator Loss: 1.2942
.
.
.
Epoch 49: Generator Loss: 1.5597, Discriminator Loss: 0.8650
Epoch 50: Generator Loss: 1.6252, Discriminator Loss: 0.8139
Epoch 51: Generator Loss: 1.5815, Discriminator Loss: 0.8455
Model saved to Trained Models\cgan\best_model.pth
Epoch 52: Generator Loss: 1.5672, Discriminator Loss: 0.8577
.
.
.
Epoch 95: Generator Loss: 1.4421, Discriminator Loss: 0.8851
Model saved to Trained Models\cgan\best_model.pth
Epoch 96: Generator Loss: 1.4879, Discriminator Loss: 0.8641
Epoch 97: Generator Loss: 1.5178, Discriminator Loss: 0.8456
.
.
Epoch 146: Generator Loss: 1.4023, Discriminator Loss: 0.8809
Model saved to Trained Models\cgan\best_model.pth
Epoch 147: Generator Loss: 1.4458, Discriminator Loss: 0.8760
.
.
Epoch 196: Generator Loss: 1.4675, Discriminator Loss: 0.8738
[Training Status]: Early stopping triggered!
```

את המימוש המלא לשני המודלים ניתן למצוא במודול gan.py.

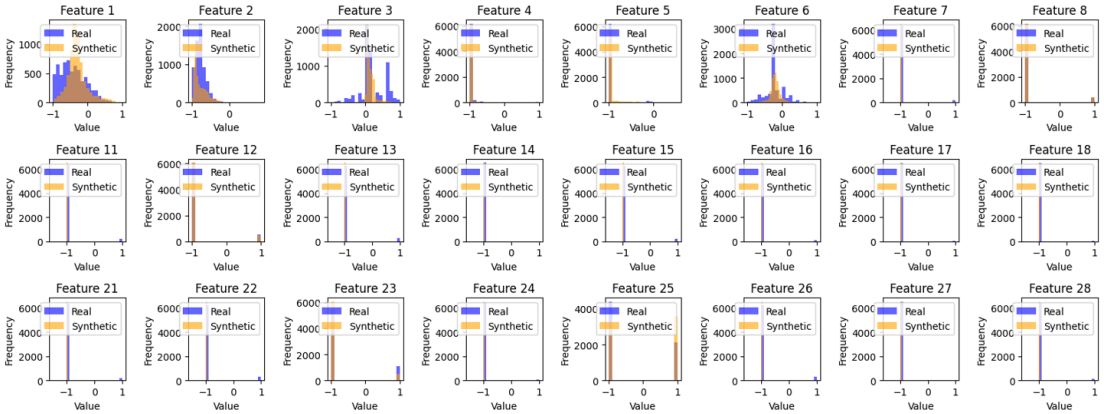
ביצוע Feature Analysis:

בשלב ראשון רצינו לבחון את התפלגויות הנתונים הנוצרים מה-GAN ומה-cGAN. בתמונות 6 הפיצ'רים הראשונים הם נומריים והיתר קטגוריאליים (מוצגים 28 מתוך 108 הפיצ'רים, מטעמי מקום). אפשר לראות דוגמה לפילוח מלא במחברת.

GAN (top few):



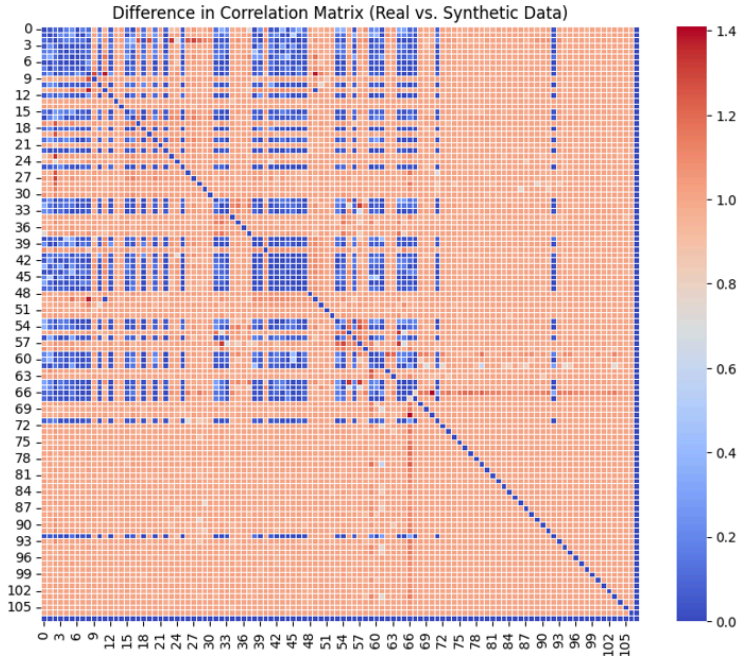
cGAN (top few):



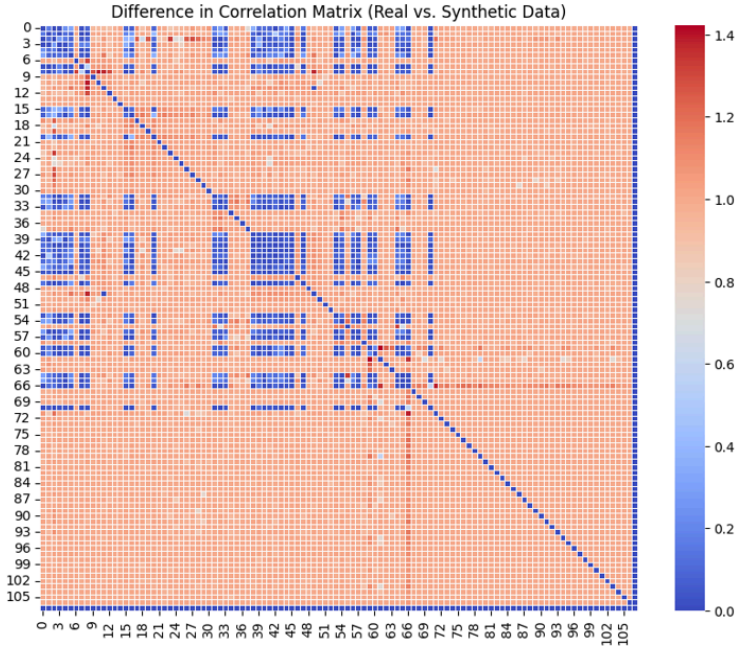
בבדיקה זו, לדעתנו, מודל ה-GAN מציג התפלגויות מעט יותר קרובות למציאות, עם פחות מרכוז סביב ערכים ספציפיים. התרשימים מראים לנו שהמודלים מצליחים בסך הכל לייצר גם את הערכים הנומריים וגם הערכים הקטגוריאליים, בהתפלגויות יחסית דומות למקור. אם למשל היינו רואים "פעמון" באיזור שונה בסקאלה מזה בו הנתונים האמיתיים נמצאים - היינו דואגים. זה לא ממש המצב, למרות שכן ניתן לראות סטייה מסוימת עבור feature 1, למשל. בדיקה זו היא ראשונית ונחמדה, אבל בשלבים הבאים נרצה לבחון האם הנתונים אמינים. האם רשומה מסוימת משכנעת אותנו שהיא אמיתית.

לשם כך בשלב ראשון נבדוק את הקורלציות בין הפיצ'רים, והאם דומות לקורלציות בין הפיצ'רים בנתונים האמיתיים. על מנת לפרשן יותר בקלות, יצרנו מטריצת קורלציות המראה את ההפרשים בקורלציות בתוך הנתונים המזויפים למול בתוך הנתונים האמיתיים.
בפשטות יחסית, תמונה בה הצבע הכחול שולט יותר מציגה לנו למעשה שההבדלים בקורלציות בין הפיצ'רים שלו יותר קטנים, ז"א שהוא יחסית מייצג יותר את היחסים בין הפיצ'רים. לדעתנו מודל ה-GAN יצר זאת מעט יותר טוב:

GAN:



cGAN:



לסיכום של בדיקה זו - הנתונים המיוצרים חד משמעית אינם מושלמים, ויש איזורי חולשה משמעותיים בהם נראה שגם אם התפלגויות הנתונים יחסית נשמרות, הקורלציות בין הנתונים לא. סביר להניח שכשננסה להשתמש בנתונים אלו לאימון מודל, הוא יוכל להבחין שאינם קוהרנטיים עם עצמם, ושהקורלציות לא נשמרות.

קיימת הבעיה של הערכים הקטגוריאליים, בהם בפרט נראה ששני המודלים מתקשים (לאור כך שהאינדקסים ה"אדומים" הם לא של הערכים הרציפים - אינדקס 6 ומעלה הם קטגוריאליים). ייתכן שהפתרון שלנו של עיגול הערכים עבור עמודות מסומנות מראש הוא אגרסיבי מידי, ומייצר "הרעבה" יחסית של ערכי פיצ'רים שנמצאים בייצוג נמוך יותר בנתונים האמיתיים. חשוב לומר שבהתבוננות מעמיקה בגרפים הקודמים, לא ראינו פיצ'רים כאלו שלא מופיעים כלל - סממן חיובי. פתרון אפשרי לדבר כזה הוא אוגמנטציה המאזנת את ערכי כל הפיצ'רים בדאטה.

הגדרת מודל AutoEncoder:

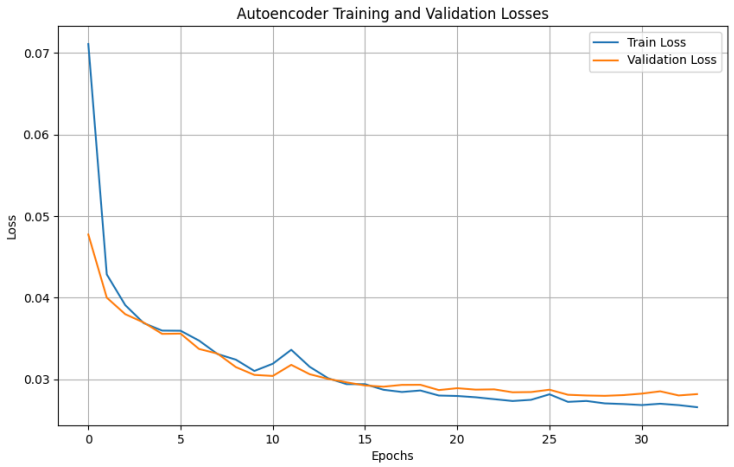
מתוך תקווה לשיפור בביצועים ע"י הקלת המשימה של ה-Generator והיפטרות מהערכים הקטגוריאליים, ניסינו להעביר את כל הבעיה למרחב לטנטי קטן יותר משמעותית. לשם כך בנינו מודל AE. בתחילה, היה למודל קשה להתכנס והוא היה ב-underfit משמעותי, אז התאמנו את ה-learning rate שלו באמצעות scheduler לצמצום כאשר הירידה ב-val_loss יורדת. הגדרנו לו גם עצירה מוקדמת משמעותית פחות מחמירה משל ה-GAN, ללא warm up. ברגע שהעברנו את המודל לפונקציית loss מסוג nn.SmoothL1Loss ראינו שיפור משמעותי ואימון שהיה נראה טוב. ההשערה שלנו היא שזה נובע מכיוון שפונקציה זו מותאמת יותר לעבוד עם טווח הערכים בין [1, -1] אותו הגדרנו ב-dataset. פונקציית האקטיבציה נבחרה גם פה לאור טווחי הערכים בבעיה. הפרמטרים של המודל היו:

```
NOISE_DIM = 8  # The size of the initial noise vector
LATENT_DIM = 16 # The dimension of the latent (encoding) dimension

# Encoder Configuration
ENCODER_CONFIG = [
    {"input_dim": DATA_DIM, "output_dim": 512, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 512, "output_dim": 256, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 256, "output_dim": 128, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 128, "output_dim": 64, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 64, "output_dim": 32, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 32, "output_dim": LATENT_DIM, "activation": None, "batch_norm": False, "dropout": 0.0},  # Latent space, no
activation
]

# Decoder Configuration
DECODER_CONFIG = [
    {"input_dim": LATENT_DIM, "output_dim": 32, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 32, "output_dim": 64, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 64, "output_dim": 128, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 128, "output_dim": 256, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 256, "output_dim": 512, "activation": nn.ReLU(), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 512, "output_dim": DATA_DIM, "activation": nn.Tanh(), "batch_norm": False, "dropout": 0.0},  # Reconstructed
output
]
```

המודל הצליח להתאמן יפה, הראה ירידה הדרגתית ב-loss והגיע ל-reconstructions דיי משכנעים כשהופעל, עם test_loss מזערי של 0.0288. מתוך תהליך הלמידה שלו:



בתור POC שהמודל אכן טוב, בנינו מחדש דגימה שלו. וירא כי טוב (שומר על ערכים קרוב לקטגוריאליים בעמודות cat):

```
Reconstructed: tensor([-0.6594, -0.7627,  0.6474, -0.9781, -0.9589, -0.1745, -0.9276, -0.9996,
    -0.9986, -1.0000, -0.9975, -1.0000,  0.9150, -0.9998, -1.0000, -0.9961,
    -0.9992, -0.9685, -0.0239, -0.9972, -0.9994, -0.9964, -0.9982, -0.9981,
    -0.9916, -0.9638, -0.9991, -0.9895, -0.9995, -0.9988, -0.9945,  0.9956,
    -1.0000, -0.9962, -0.9994, -0.9989, -1.0000, -0.9998, -1.0000, -1.0000,
    -1.0000,  0.9983, -1.0000, -1.0000, -0.9999, -0.9985, -0.9995, -1.0000,
    -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -0.9996, -0.9999, -1.0000,
     0.9936, -0.9982, -1.0000,  0.8829, -0.9521, -0.9619, -0.9981, -0.9990,
     0.9999, -0.9999,  0.9455, -0.9995, -1.0000, -0.9829, -0.9849, -0.9983,
    -1.0000, -0.9999, -0.9999, -0.9997, -0.9998, -1.0000, -0.9994, -0.9999,
    -1.0000, -1.0000, -0.9998, -1.0000, -1.0000, -1.0000, -0.9996, -0.9966,
    -1.0000, -1.0000, -0.9992, -1.0000, -0.9979, -0.9997, -0.9999, -0.9997,
    -0.9998, -1.0000, -1.0000, -1.0000, -1.0000, -1.0000, -0.9999, -1.0000,
    -0.9999, -0.9999, -0.9999, -1.0000])
```

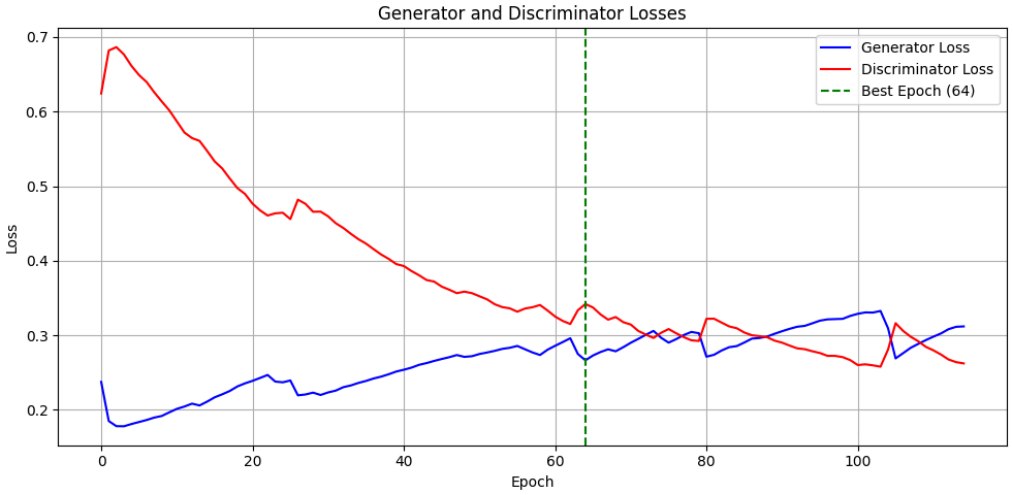
המודל השפיע באופן דרסטי על הארכיטקטורה הנדרשת עבור מודלי ה-GAN, ודרש מאיתנו generator שיהיה משמעותית יותר חזק מה-discriminator (יותר מבארכיטקטורה הקודמת) עם רגולריזציות כבדות על D כדי לאפשר ל-G אימון יותר משמעותי. כמו כן, מאחר וכעת עבדנו עם מרחב לטנטי float וללא עמודות קטגוריאליות, שינינו את פונקציית ה-loss של המודל להיות nn.MSELoss. כמו כן, בתהליך האימון פה כן מצאנו לנכון להגדיר W2 רק עבור D, ו-LR קטן פי 100, על מנת לאפשר ל-G "לעמוד בקצב". את המימוש המלא ניתן לראות במודול ae_gan.py.

הקונפיגורציות החדשות של שני המודלים היו:

```
## Generator Configuration
GENERATOR_CONFIG = [
    {"input_dim": NOISE_DIM, "output_dim": 32, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 32, "output_dim": 64, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 64, "output_dim": 128, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.3},
    {"input_dim": 128, "output_dim": 64, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.3},
    {"input_dim": 64, "output_dim": 32, "activation": nn.LeakyReLU(0.2), "batch_norm": True, "dropout": 0.2},
    {"input_dim": 32, "output_dim": LATENT_DIM, "activation": nn.Tanh(), "batch_norm": False, "dropout": 0.0},  # No dropout
on the last layer
]

## Discriminator Configuration
DISCRIMINATOR_CONFIG = [
    {"input_dim": LATENT_DIM, "output_dim": 8, "activation": nn.LeakyReLU(0.2), "batch_norm": False, "dropout": 0.5},
    {"input_dim": 8, "output_dim": 1, "activation": nn.Sigmoid(), "batch_norm": False, "dropout": 0.0},  # No dropout on the
output layer
]
```

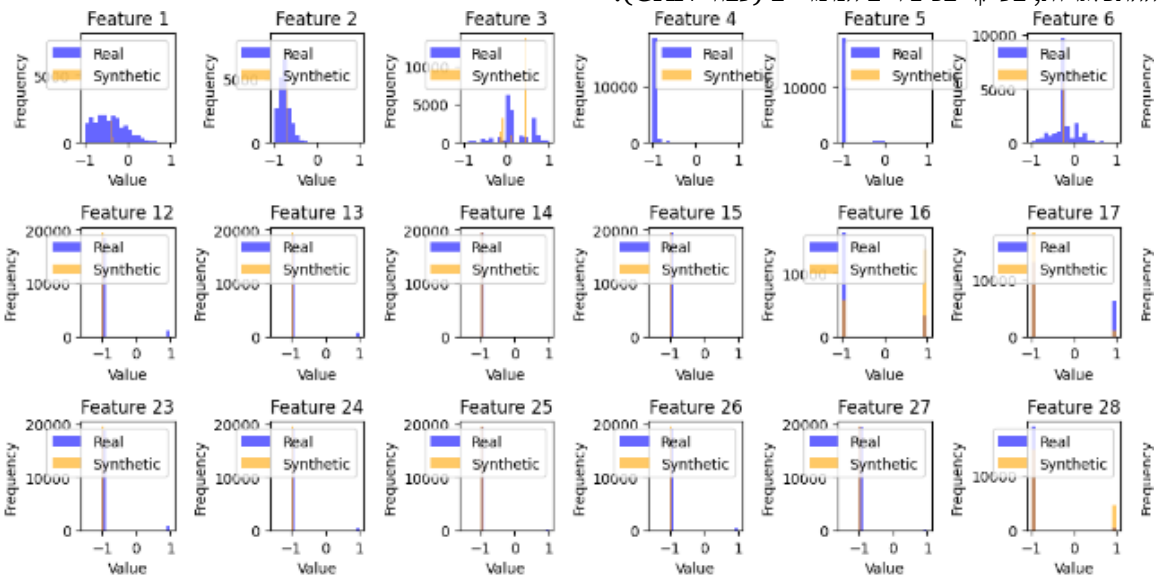
כאשר האימון של GAN היה נראה באופן הבא:



אנו מרוצים במיוחד מגרף loss זה מכיוון שהוא מראה את התכונה לה קיווינו בין 2 חלקי המודל - שיפור של אחד על חשבון השני ואיזון של ה-loss שלהם.

איפה הבאסה?

למרות הציפיות הגבוהות, עוד בשלב התפלגויות הפיצ'רים ראינו שהמודל הזה לא מבצע טוב, ונראה שלא מציג היטב את ההתפלגויות, בעיקר בפיצ'רים הנומריים (עבור GAN):



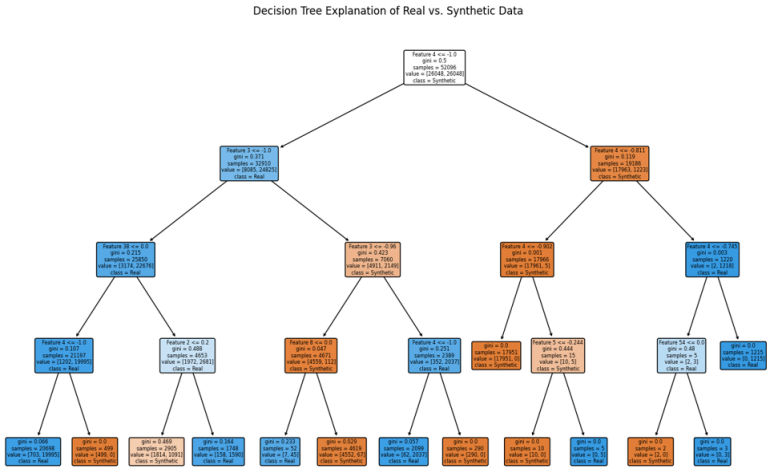
גם מטריצת הפרשי הקורלציות הציגה הפרשים גדולים משמעותית ממה שראינו במודל הישן, מה שגרם לנו להבין שהשינוי הזה, למרות המרחב הלטנטי המאומן היטב וגרף ה-loss המשכנע, לא פותרים את הבעיה.

ביצוע הערכה לדאטה הסינטטי באמצעות detection & efficiency:

התוצאות בחלק זה מתייחסות למודלי GAN ו-cGAN ללא תוספת ה-AE. בדקנו התוצאות גם איתו, אך הן היו יותר גרועות. הניסויים בוצעו על חלוקות train-test באמצעות 3 ערכי SEED שונים (42, 76, 2005) והתוצאות המופיעות פה הן ממוצע של ציוני הביצוע שהתקבלו ע"י 2 המודלים. בשלב הראשון, בהתאם לבקשה, ניקח את ה-train_set ו generated set באותו הגודל (מושפעים מערך ה-SEED) ובאמצעות CV 4 folds נאמן מודל RF על 75%, ועבור ה-25% הנוספים נבדוק האם המודל מצליח להבחין בין הדגימות האמיתיות והמזויפות. כל הדגימות האמיתיות יסומנו ב-1 וכל המזויפות ב-0.

לצערנו, בשלב זה, לא הצלחנו בכלל לעבוד על מודל ה-RF, והוא הצליח להבחין באופן מלא בין הדגימות האמיתיות לבין המזויפות (Average AUC for detection: 1.0000) גם עם מודל ה-GAN וגם עם ה-cGAN.

על מנת לנסות ולהבין מה יש בדגימות המזויפות שגורם למודל לגלות אותן ניסינו לבנות מודל שניתן להסביר את בחירותיו. החשש המרכזי שלנו היה שנראה שיש פיצ'ר מסוים / ערך מסוים / כניסה מסוימת בוקטור שמסגירה בוודאות שמדובר בדגימה מזויפת. ל"מזלנו", זה לא המצב, וכדי להגיע לחיזוי גם המודל הפשוט ביותר היה צריך לעבור דרך מספר צמתי החלטה:



בהמשך לזה ניסינו לבדוק האם יש פיצ'רים ספציפיים שאנחנו פשוט מג'נרטים ממש ממש גרוע, וקיבלנו את הרשימה הבאה של פיצ'רים בעלי חשיבות לקבלת ההחלטה של המודל (בסדר יורד). כנראה שבתחום של הפיצ'רים הנומריים דווקא המודל לא למד טוב את התפלגות הערכים המאפיינת רשומות באופן "אמין", היות ופיצ'ר 4 (capital-gain) הוא באופן משמעותי המסגיר ביותר:

Feature Importances (Top features that separate real and synthetic data):
Feature 4: 0.6687
Feature 3: 0.2551
Feature 38: 0.0452
Feature 2: 0.0273
Feature 8: 0.0033
Feature 5: 0.0003
Feature 54: 0.0001

בדיקה זו בוצעה על ה-cGAN, היות והוא כן הראה ביצועים מעט יותר טובים לפי efficiency (אך עדיין גרועים) בחלק זה.

בשלב השני נרצה לבדוק האם הדגימות המג'ונרטות מהוות חליף טוב לדגימות האמיתיות ע"י אימון מודל RF פעם אחת לדאטה האמיתי ופעם אחת למזויף. אם אותו המודל מצליח להגיע לציון AUC דומה בין שני ה-settings על ה-test set סימן שהדאטה שימושי. נבחן ע"י היחס בין שני הציונים: Real AUC\Fake AUC.

עבור בדיקה זו קיבלנו AVG Efficacy Score: 0.63 עבור מודל ה-cGAN, לאורך 3 ניסויי SEED שונים, ציון שאינו טוב במיוחד, היות והבעיה היא בעיית סיווג בינארית. מודל ה-GAN קיבל ציון ממוצע נמוך יותר - 0.56 - פער שאינו מפתיע מבחינה תיאורטית, שכן אנחנו מצפים שמודל המג'נרט דגימות לפי לייבל יצליח לשכנע יותר טוב מודל ML.

הביצועים הגרועים בשני המדדים מראים שכנראה למידע הסינטטי יש סממנים מסגירים הקשורים ביחסים בין הפיצ'רים השונים. בגדול, זה אומר שה-GAN לא למד מספיק טוב את היחסים בין הפיצ'רים ולכן מייצר דגימות שאינן אמינות.

פתרונות אפשריים יכולים להיות מודל יותר מורכב, שימוש בארכיטקטורת WGAN, שימוש בקשרים residual , יצירת תהליך אימון יותר מורכב הלומד תוך שילוב ב-loss של המסקנות על feature importance או שיטות אוגמנטציה יותר מורכבות (שיטת האוגמנטציה בה השתמשנו לא הספיקה כדי לשפר המדדים), אבל הדברים האלו אינם ב-scope של העבודה. כמו כן, ייתכן שאופטימיזציה יותר משמעותית של שילוב המודל עם ה-AE ותחקור ספציפי של הפיצ'רים בהם הוא לא מצליח לייצר ערכים טובים הייתה פותרת את הבעיה.