Artificial Neural Networks with small Datasets. A practical Approach

Masterarbeit

zur Erlangung des akademischen Grades

Master of Science in Engineering (M.Sc.)

Eingereicht bei:

Fachhochschule Kufstein Tirol Bildungs GmbH

Data Science & Intelligent Analytics

Verfasser:

Paul Leitner, BA

1910837299

Erstgutachter	:	Dr. Johannes Luethi
Zweitgutachter	:	Lukas Demetz, PhD

Abgabedatum:

31. October 2021

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und in der Bearbeitung und Abfassung keine anderen als die angegebenen Quellen oder Hilfsmittel benutzt sowie wörtliche und sinngemäße Zitate als solche gekennzeichnet habe. Die vorliegende Masterarbeit wurde noch nicht anderweitig für Prüfungszwecke vorgelegt.

Kufstein, 31. October 2021

Paul Leitner, BA

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

CNN	Convolutional Neural Network
GB	Gradient Boosting
nn	Neural Network
ml	Machine Learning
SMOTE	synthetic minority oversampling technique
GAN	Generative Adversarial Network
DCGAN	Deep Convolutional GAN
EM	Earth Mover's Distance
SVC	Support Vector Machine Classification
KNN	K-Nearest-Neighbor Classification

FH Kufstein Tirol

Data Science & Intelligent Analytics

Abstract of the thesis: **Artificial Neural Networks with small Datasets. A practical Approach**

Author: Paul Leitner, BA

First reviewer: Dr. Johannes Luethi

Second reviewer: Lukas Demetz, PhD

After giving a summary on the literature and history of neural networks, I elucidate the trade-offs between deep learning and other machine learning approaches. I show that machine learning approaches such as Gradient Boosting (GB) mostly trade increased data requirements in favor of data scientist worktime in data preparation and feature engineering. I then investigate whether more complicated Neural Networks (nns) may be used by synthetically enlarging the training data present and thereby achieving comparable accuracy while saving data preparation time, effectively trading processing time (synthetic data enlargement being resource-intensive) for manual feature-engineering time by creating a nn model and benchmarking it against a GB reference model on a standard Machine Learning (ml) dataset with small data, the diabetic retinopathy dataset.

insert result - how much better does this perform? tradeoffs!

note - synthetic data **?**

31. October 2021

1. Introduction

In 2012 Krizhevsky and his colleagues entered and won the ImageNet classification contest with a deep convolutional neural network Convolutional Neural Network (CNN), outperforming other models by a significant margin Krizhevsky et al. (2012). This marked a turning point in machine learning in general, and in perceptual tasks specifically.

Pereira et al. (2009) is often invoked as a shorthand to the core problem of Machine Learning, the fact that a larger training corpus would always be preferable.

Currently, data scientists spend a significant amount (how much? sources!) of their time, when solving 'shallow' machine learning tasks (such as???) in feature engineering / preprocessing. Source! examples! This is due to the fact that shallow approaches such as decision trees, GBM and SVM models require features that 'directly' connect the prepared data to the searched-for outcome. (source)

Deep learning (neural networks) create intermediate representations via **stacked layers** at the cost of increased training data (source). Thereby enabling a more abstract understanding of patterns within the data.

Shearer (2000)

1.1 Problem Situation

1.2 Objectives

1.3 Methods

1.4 Structure

1.5 Tables

Table 1 shows an example table.

Table 1: This is a table

Column 1	Column 2	Column 3
A	B	C
D	E	F
G	H	I

1.6 Source Code

2. Synthetic Data in Privacy

cite –> paper from source, different models on synthetic data!

2.1 Synthetic Data for model performance

When training nns for image classification, (source) a common practice is **data augmentation**, a range of random transformation applied to images in order to synthetically increase the breadth of data that the model is exposed to. Such operations include

- rotation
- shearing
- zoom
- height & width shift

effectively, these operations transform an Image while preserving the underlying signals in the data. However, with other types of data this might be possible. Attributes of another dataset may not be feasibly 'shifted' in one direction or another without fundamentally changing the signal and misleading the model.

note - the infeasibility of pretraining on non-image datasets - representations of the visual world

2.2 Deep Learning

3. Comparison with other solutions to the small data problem

- synthetic minority oversampling technique (SMOTE)
- crossvalidation (k-fold, single holdout)
- transfer learning (word embeddings, image filter layers)
- wholesale synthetic data approaches, **? more sources needed**

3.1 Data Enhancement for image data

4. Technical Application

4.1 Theoretical applicability

In their landmark paper in 2014, Goodfellow et al. (2014) demonstrated the viability of Generative Adversarial Networks (GANs) on creating image data on the classic MNIST dataset (described by Deng (2012)), by generating - among other things - convincing handwritten digits. As mentioned in 3.1, some of the architecture specifics and evaluation are quite specific to image data in that

- the data contains a notion of locality, as neighboring data points (i.e. pixels) are strongly dependent
- dimensionality of the generated data is higher than the **latent space**
- results lend themselves to visual quality inspection by humans (it is easy to see even degrees of quality between different architectures)

specifically the former points are strongly relevant to GAN architecture, as will become obvious shortly.

4.2 Technical implementation steps

Since the goal of this paper is to evaluate whether or not GAN may be used to not only generate more data of a small non-image dataset (which is fairly trivial) but whether or not this data actually serves to **boost model performance** of models trained on the resulting data, a small, well-understood standard dataset was used to develop the initial architecture; Farag and Hassan (2018). Specifically, the iconic titanic dataset constitutes a binary classification problem, which facilitates quick model evaluation and ameliorates some of the more typical difficulties of training GANs - see below.

The first attempts to create a basic, dense GAN actually failed to converge for a significant number of experiments with different amounts of layers, neurons and size of the latent space. Somewhat unsurprisingly, achieving the classic Nash Equilibrium between discriminator and generator was fairly difficult and the initial models all proved unstable. GANs provide several unique challenges, and/or failure modes:

- mode collapse Che et al. (2017)
- oscillation and general instability of the model Liang et al. (2018)
- catastrophic forgetting McCloskey and Cohen (1989)

Mode collapse is especially relevant in a task like MNIST, where there are multiple classes to be generated, and the generator becomes increasingly proficient in generating one class explicitly - thankfully, this is less of an issue in a binary classification task.

4.2.1 Network Architecture

The other failure modes, however **did** all make an appearance at one time or another, after the initial data preparation. It was fairly clear that the initial network, with one layer each for the generator and the discriminator each, and 64 neurons had insufficient representational power to converge on creating convincing samples as can be seen in 1:

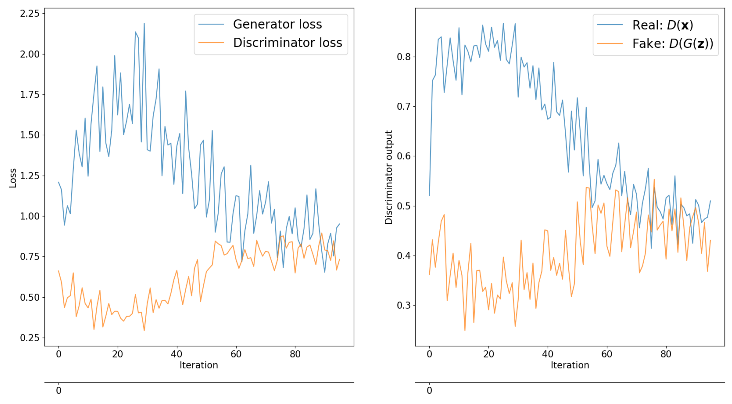

Figure 1: Initial simple dense GAN - left side shows the losses of generator and discriminator, right side shows the probabilities assigned to real and fake samples by the discriminator

Further experiments, with increased numbers of layers and neurons, produced first a very typical oscillation pattern, shown in 2:

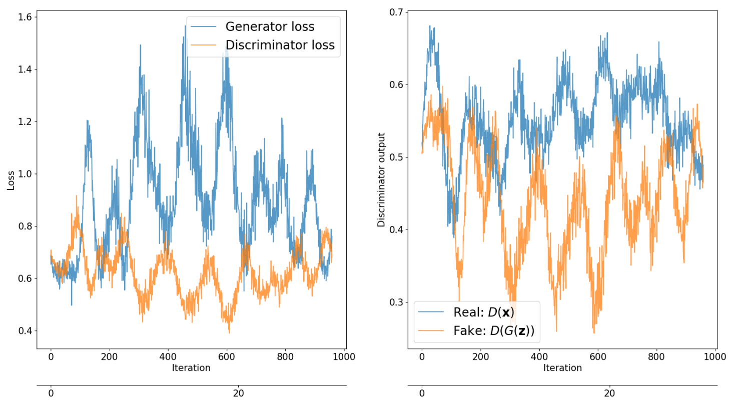

Figure 2: Dense GAN, 3 layers, 64 neurons/layer; left - losses of generator/discriminator right - the probabilities real/fake assigned by the discriminator

Note the oscillations in the early stages, which barely decrease in amplitude at all. Training for an increased number of epochs did not lead to the emergence of a proper equilibrium state, since the network was altogether too unstable.

Finally, it has to be stressed that finding the ideal combination learning rates, dropout in the discriminator and number of training epochs, is really quite difficult, especially since there appears no good substitute to visually examining the pattern that is produced by a given architecture and then to adjust. A process that has to be iterated for quite a while, and is fairly manual and heavy on trial-and-error.

Ultimately, a a promising architecture appeared to be dense networks with 3 layers each, but a higher number of neurons, and still these networks diverged rather quickly shown here 3:

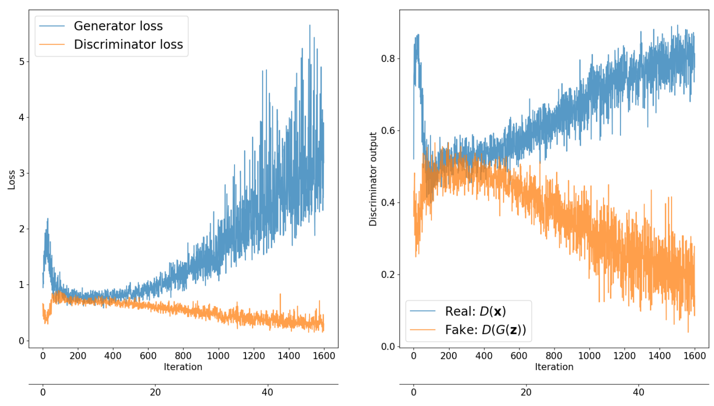

Figure 3: Dense GAN, 3 layers, 128 neurons/layer, reduced learning rate and dropout in discriminator - losses of generator/discriminator right - the probabilities real/fake assigned by the discriminator

Unfortunately, from here on out simply optimizing the number of neurons and learning rate and learning rate scheduling was not enough to mitigate this divergence. Although implementing the popular 1Cycle learning rate decay (described by Smith (2018)) did ameliorate the issue somewhat, it did not fix the network.

What ultimately made the difference is an adaptation of the architecture proposed by Radford et al. (2016). The architecture proposed here for image generation constitutes a **symmetrical** upsampling from the latent space in the generator (in case of images, a **transposed convolution**) and downsampling in the discriminator. As shown by Suh et al. (2019) here:

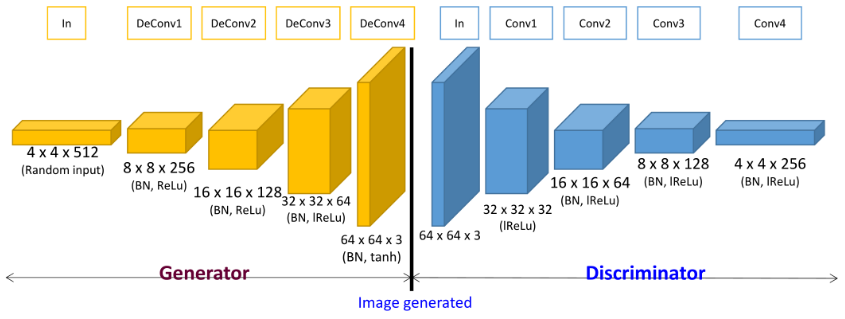

Figure 4: Architecture of Discriminator and Generator

Initially, implementing convolution actually deteriorated performance and completely prevented convergence of the network, a probable explanation would be the fact that convolution and transposed convolution not only upsample the latent space but more fundamentally relate to locality in the data; i.e. multiple convolutional layers over a picture effectively create hierarchical feature extraction. A paper that illustrates the mechanics of this fairly well was Dumoulin and Visin (2018). Effectively, these convolutions would initially find small features in images, subsequent convolutions would assemble these features into feature maps and their presence would indicate the presence of objects in an image. The entire concept of strides and adjacent data points however, does not make sense in the concept of a dataset where an observation consists of a feature vector, in which the order does not convey any information. While 1D convolutions are quite widely used in sequence and time-series processing - which are quite comfortably out of scope of this paper - they fundamentally seem unsuited to a dataset which would not lose any information if the order of its' attributes was permutated.

What **did** make a difference was implementing the symmetry of upsampling and condensing in the generator and discriminator.

Furthermore, Radford et al. (2016) propose other guidelines for building Deep Convolutional GANs (DCGANs) which proved helpful:

- implementing BatchNormalization in the generator and discriminator Ioffe and Szegedy (2015)
- using ReLU activation in all layers in the generator except for the output, which would use tanh
- using LeakyReLU in all layers in the discriminator, except for the output which uses sigmoid

After implementing these guidelines, using Binary Categorical Crossentropy loss, the generator and discriminator actually converged fairly well already, as seen in 5

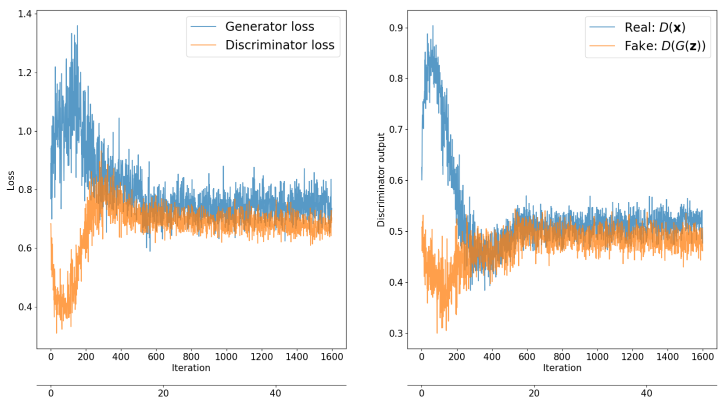

Figure 5: Dense GAN, 2 layers, 32 neurons/layer; left - losses of generator/discriminator right - the probabilities real/fake assigned by the discriminator

Importantly, in this architecture the **generator** model starts with a dense layer containing 32 neurons, which doubles in every layer (once in this case, although this is a variable parameter). The final dense hidden layer is then downsampled again to reflect the original data - like this:

Listing 1: generator network

```python
import tensorflow as tf


def create_generator_network(
    number_hidden_layers: int = 2,
    number_hidden_units_power: int = 5,
    hidden_activation_function: str = 'ReLU',
    use_dropout: bool = False,
    upsampling: bool = True,
    use_batchnorm: bool = True,
    dropout_rate: float = 0.3,
    number_output_units: int = 12,
    output_activation_function: str = 'tanh') -> tf.keras.Model:

    model = tf.keras.Sequential()
    for i in range(number_hidden_layers):

        if upsampling:
            # implements the guideline DCGAN - upsampling layers in the generator
            model.add(tf.keras.layers.Dense(2 ** (number_hidden_units_power + i), use_bias=False))

        else:
            model.add(tf.keras.layers.Dense(2 ** number_hidden_units_power, use_bias=False))

        if use_batchnorm:
            model.add(tf.keras.layers.BatchNormalization())
        else:
            pass

        model.add(tf.keras.layers.Activation(hidden_activation_function))

        if use_dropout:
            model.add(tf.keras.layers.Dropout(dropout_rate))
        else:
            pass

    model.add(tf.keras.layers.Dense(number_output_units))
    model.add(tf.keras.layers.Activation(output_activation_function))

    return model
```

Listing 1 shows the python function which creates the generator network.

The discriminator implements the exact mirror image of this pattern, beginning with the same amount of neurons after the input layer and downsampling by half each layer;

Listing 2: discriminator network

```python
import tensorflow as tf


def create_discriminator_network(
    number_hidden_layers: int = 2,
    number_hidden_units_power: int = 5,
    hidden_activation_function: str = 'LeakyReLU',
    use_dropout: bool = True,
    upsampling: bool = True,
    use_batchnorm: bool = True,
    dropout_rate: float = 0.3,
    number_output_units: int = 1) -> tf.keras.Model:

    model = tf.keras.Sequential()

    for i in range(number_hidden_layers):

        if upsampling:
            # implements the guideline - downsample in the discriminator network
            model.add(tf.keras.layers.Dense(2 ** (number_hidden_units_power + number_hidden_layers - i - 1)))

        else:
            model.add(tf.keras.layers.Dense(2 ** number_hidden_units_power))

        if use_batchnorm:
            model.add(tf.keras.layers.BatchNormalization())
        else:
            pass

        model.add(tf.keras.layers.Activation(hidden_activation_function))

        if use_dropout:
            model.add(tf.keras.layers.Dropout(dropout_rate))
        else:
            pass

    model.add(tf.keras.layers.Dense(number_output_units, activation=None))

    return model
```

Listing 2 shows the python function which creates the discriminator network.

4.2.2 Network Training Implementation

With the basic architecture for the Network in place, the functions are put together into a custom training loop. While there are multiple approaches to training GANs, starting with the seminal paper by Goodfellow et al. (2014), such as training the discriminator and generator in an epoch separately, here a custom training loop with separate optimizers was chosen from the beginning, in order to accommodate more exotic loss functions.

Since a number of experiments on the efficacy of the generated data has to be done, the entire GAN system was set up to be set up with sensible defaults, dynamically adapting to 1D datasets. Specifically Buitinck et al. (2013) suggest design lessons from scikit-learn, one of which is sensible defaults.

In order to automatically create both generators and discriminators dynamically based on input shape (but with strong default settings which ideally do not have to be adjusted during experimentation at all) a small package was created which encapsulates the entire training loop.

Key part here is the training loop shown here:

Listing 3: training loop

```
# lists to store losses and values
all_losses = []
all_d_vals = []

for epoch in range(1, n_epochs+1):
    epoch_losses, epoch_d_vals = [], []
    for i,(input_z,input_real) in enumerate(training_data):

        # generator loss, record gradients
        with tf.GradientTape() as g_tape:
            g_output = generator_model(input_z)
            d_logits_fake = discriminator_model(g_output, training=True)
            labels_real = tf.ones_like(d_logits_fake)
            g_loss = loss_fn(y_true=labels_real, y_pred=d_logits_fake)
        # get loss derivatives from tabe, only for trainable vars, in case of regularization / batchnorm
        g_grads = g_tape.gradient(g_loss, generator_model.trainable_variables)

        # apply optimizer for generator
        g_optimizer.apply_gradients(
            grads_and_vars=zip(g_grads, generator_model.trainable_variables))

        # discriminator loss, gradients
        with tf.GradientTape() as d_tape:
            d_logits_real = discriminator_model(input_real, training=True)

            d_labels_real = tf.ones_like(d_logits_real)

            # loss for the real examples - labeles as 1
            d_loss_real = loss_fn(
                y_true=d_labels_real, y_pred=d_logits_real)

            # loss for the fakes - labeled as 0

            # apply discriminator to generator output like a function
            d_logits_fake = discriminator_model(g_output, training=True)
            d_labels_fake = tf.zeros_like(d_logits_fake)

            # loss function
            d_loss_fake = loss_fn(
                y_true=d_labels_fake, y_pred=d_logits_fake)

            # compute component loss for real & fake
            d_loss = d_loss_real + d_loss_fake

        # get the loss derivatives from the tape
        d_grads = d_tape.gradient(d_loss, discriminator_model.trainable_variables)

        # apply optimizer to discriminator gradients - only trainable :todo: add regularization here
        d_optimizer.apply_gradients(
            grads_and_vars=zip(d_grads, discriminator_model.trainable_variables))

        # add step loss to epoch list
        epoch_losses.append(
            (g_loss.numpy(), d_loss.numpy(),
             d_loss_real.numpy(), d_loss_fake.numpy()))

        # probabilities from logits for predcitions, using tf builtin
        d_probs_real = tf.reduce_mean(tf.sigmoid(d_logits_real))
        d_probs_fake = tf.reduce_mean(tf.sigmoid(d_logits_fake))
        epoch_d_vals.append((d_probs_real.numpy(), d_probs_fake.numpy()))

    # record loss
    all_losses.append(epoch_losses)
    all_d_vals.append(epoch_d_vals)
    print(
        'Epoch {:03d} | ET {:.2f} min | Avg Losses >>'
        ' G/D {:.4f}/{:.4f} [D-Real: {:.4f} D-Fake: {:.4f}]'
        .format(
            epoch, (time.time() - start_time)/60,
            *list(np.mean(all_losses[-1], axis=0))))
result = {
    'all_losses': all_losses,
    'all_d_vals': all_d_vals,
    'generator': generator_model,
    'discriminator': discriminator_model}

if export_generator:

    print()
    print('saving generator model')

    tf.keras.models.save_model(generator_model, f'./models/generator_{model_name}.h5')
    print(f'generator model saved to: ./models/{model_name}.h5')

return result
```

Listing 3 shows the python function which trains the network. Note that this function is **quite strongly simplified**, the actual code used can be found at `https://github.com/PaulBFB/master_thesis/blob/main/train_generator.py` and would not likely be germane to the paper in its' entirety in any case.

The above training loop was developed together with the network architecture and also produced the training graphics shown so far. Before it was used in experimentation however, adjusting its' loss function was tested. Specifically, as proposed by Arjovsky et al. (2017), implementing the Earth Mover's Distance (EM) as a loss function.

$$W(\mathbb{P}_r, \mathbb{P}_g) = \inf_{\gamma \in \Pi(\mathbb{P}_r, \mathbb{P}_g)} \mathbb{E}_{(x,y)\sim\gamma}\big[\, \|x - y\| \,\big]$$

Figure 6: Wasserstein distance formula

As mentioned in the paper, this distance denotes the amount of work that is necessary to transform one distribution in to another, given an optimal transfer plan γ which actually denotes **how** the work is done. Furthermore, and probably most importantly, the EM, in contrast to other loss functions, is actually a function of the parameters θ of the distributions in question, i.e. it can express partial derivatives with respect to the single parameters!

However, finding γ is an optimization problem by itself, since it constitutes an **optimal** solution. As Arjovsky et al. (2017) mention therefore, it is approximated during training. A complete explication of the metric and its' approximation is out of scope of this paper.

What this achieves in practice, is that it enables the discriminator to act as a **critic**, essentially reporting the distance that the generator has yet to move back during training, which the generator then backpropagates to its' parameters. Thereby,the loss during training actually becomes more meaningfully readable.

Arjovsky et al. (2017) recommend in their paper to clip the gradients reported back to the generator to be clipped. This led to substantial instability in the network, as can be seen here:

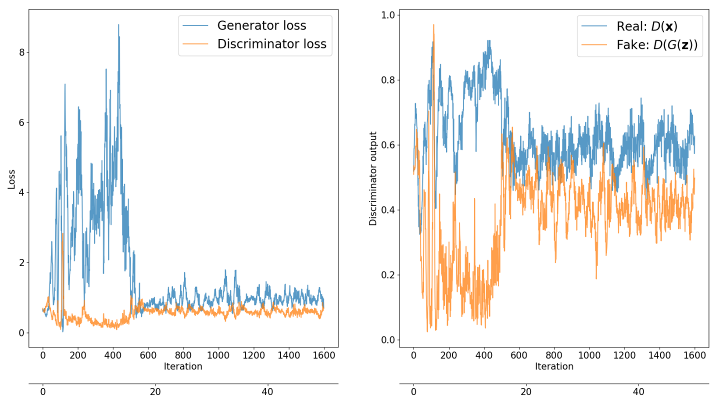

Figure 7: Wasserstein GAN, 3 layers; left - losses of generator/discriminator right - the probabilities real/fake assigned by the discriminator

As can be seen clearly here, the equilibrium between the components is fairly unstable. Gulrajani et al. (2017) pose that gradient clipping here actually leads to exploding and vanishing gradients, which seems to describe this result fairly well. In the actual implementation therefore, the **Gradient Penalty** method they recommend is implemented, namely:

- between real and fake examples in a batch, choose a random number sampled from a uniform distribution
- interpolate between real and fake examples
- calculate discriminator loss for all interpolated examples
- add the gradient penalty based on the interpolations
- remove batch normalization from the discriminator, since it shifts example gradients based on the entire batch

Therefore, the training loop was modified:

Listing 4: training loop

```python
for epoch in range(1, n_epochs+1):
    epoch_losses, epoch_d_vals = [], []
    for i,(input_z,input_real) in enumerate(training_data):


        # set up tapes for both models
        with tf.GradientTape() as d_tape, tf.GradientTape() as g_tape:
            g_output = generator_model(input_z, training=True)

            # real and fake part of the critics output
            d_critics_real = discriminator_model(input_real, training=True)
            d_critics_fake = discriminator_model(g_output, training=True)

            # generator loss - (reverse of discriminator, to avoid vanishing gradient)
            g_loss = -tf.math.reduce_mean(d_critics_fake)

            # discriminator losses
            d_loss_real = -tf.math.reduce_mean(d_critics_real)
            d_loss_fake =  tf.math.reduce_mean(d_critics_fake)
            d_loss = d_loss_real + d_loss_fake

            # INNER LOOP for gradient penalty based on interpolations
            with tf.GradientTape() as gp_tape:
                alpha = rng.uniform(
                    shape=[d_critics_real.shape[0], 1, 1, 1],
                    minval=0.0, maxval=1.0)

                # creating the interpolated examples
                interpolated = (
                    alpha*tf.cast(input_real, dtype=tf.float32) + (1-alpha)*g_output)

                # force recording of gradients of all interpolations (not created by model)
                gp_tape.watch(interpolated)
                d_critics_intp = discriminator_model(interpolated)

            # gradients of the discriminator w. regard to all
            grads_intp = gp_tape.gradient(
                d_critics_intp, [interpolated,])[0]

            # regularization
            grads_intp_l2 = tf.sqrt(
                tf.reduce_sum(tf.square(grads_intp), axis=[1, 2, 3]))

            # compute penalty w. lambda hyperparam
            grad_penalty = tf.reduce_mean(tf.square(grads_intp_l2 - 1.0))

            # add GP to discriminator
            d_loss = d_loss + lambda_gp*grad_penalty
```

Note that according to Gulrajani et al. (2017) a λ value of 10.0 worked well in all examples, which is what was used here. Again, this is a strongly truncated version of the code, the full version can be found at https://github.com/PaulBFB/master_thesis/blob/main/train_wasserstein_generator.py - this also contains modified functions to create a **discriminator** without BatchNormalization.

The resulting training loop with the same layers and upsampling-downsampling symmetry between generator and discriminator resulted in this:

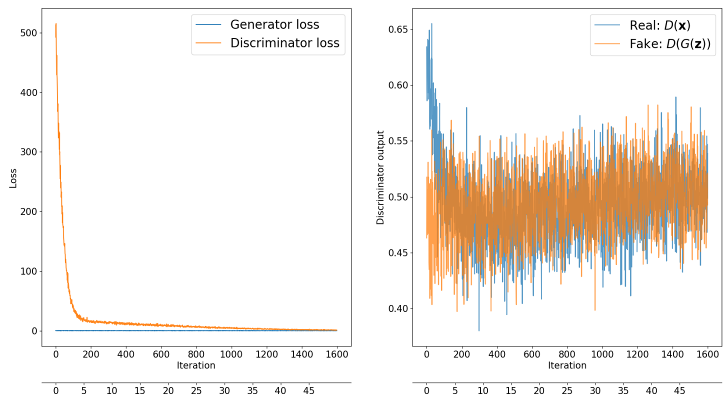

Figure 8: Wasserstein GAN with gradient penalty, 2 layers; left - losses of generator/discriminator right - the probabilities real/fake assigned by the discriminator

The equilibrium is quite tightly clustered around a probability of real and fake samples, which indicates that the discriminator and generator quickly reach a stable equilibrium in which the discriminator is essentially forced to guess between real and fake examples.

Both training loops, the basic DCGAN-adapted including BatchNormalization as well as the Wasserstein loop including penalty were preserved as separate modules. In order to perform efficient experiments on entire datasets, again with sensible default values as recommended by Buitinck et al. (2013) those loops were wrapped into a python module, which enables:

- accept preprocessed data in the form of numpy arrays, keep the original size or take a random sample from it (in order to experiment with even smaller data-subsets)

- retrain either a Wasserstein-GP or DCGAN-adapted generator if the data was decreased (or the generator is forced to be trained)
- create a number of samples based on the amount of original data and mix it into the original data

Especially important here is the second point, since using a generator trained on the **full training data set** as a GAN and then generating data to mix into a reduced subset would effectively constitute **information leakage** from the entire training dataset!

The complete function enhancing data may be found here `https://github.com/PaulBFB/master_thesis/blob/main/enhance_data.py`, also the helper function using the generator to generate data from a distribution may be found here: `https://github.com/PaulBFB/master_thesis/blob/main/generate_data.py` - both functions are fairly basic. Mostly of interest is the fact that they may be used fairly agnostically with a given training dataset, given that is has been processed to be suitable to train standard models on it; i.e. scaled, imputed if necessary and categorically encoded.

From here on out, experiments were performed on the dataset with different types of models.

5. Data Boosting Experiment Results

5.1 Experiments on the original dataset

The technical implementation that has been described in 4 has been entirely developed on the titanic dataset described by Farag and Hassan (2018). Therefore the first experiments were also performed on this dataset.

It is important to **note** here, that due to the fact that this dataset is extremely widely used, there have been significantly higher performances in accuracy achieved. These performances are mostly due to extensive feature engineering, since some of the features of the dataset contain implicit information. Take for example the cabin number, which contains information on where the passenger was staying, which would logically have bearing on their odds of survival, if it was mapped to the cabin's distance from the deck and/or lifeboats.

This is, however, explicitly **not** the purpose of this experiment, a notebook that does this fairly well can be found here: `https://www.kaggle.com/vinothan/titanic-model-with-90-accuracy`

The purpose of this experiment however, is to see if gains in performance can be achieved by simply applying larger compute power to the dataset in an agnostic fashio (more details in the discussion).

5.1.1 Experiments with decreased amounts of data

Initially, the question posed had been whether or not increasing the amount of training data available using a GAN could increase neural network performance. To examine this in detail, the data was systematically decreased in steps, and neural networks were then trained in parallel

- on the shrunken data
- on progressively boosted data

using parameter gridsearch. Gridsearch on neural networks is not yet well automated, therefore in order to do this, some helper modules were created:

A model creation function

To ensure that the models that were trained on the shrunken and boosted data partitions, these models had to be created using the same parameters. This was done with a simple model creation function that encapsulated all the necessary defaults;

Listing 5: training loop

```
def make_model(
    input_shape: tuple = (11, ),
    number_hidden_layers: int = 8,
    activation: str = 'elu',
    alpha: float = .2,
    neurons: int = 32,
    loss: str = 'binary_crossentropy',
    learning_rate: float = .003,
    dropout_rate: float = .5) -> Model:

    model = models.Sequential()
    model.add(layers.InputLayer(input_shape=input_shape))

    for i in range(number_hidden_layers):
        model.add(layers.Dense(
            neurons,
            kernel_initializer='he_normal',
            name=f'hidden_layer_{i}_relu_alpha_{alpha}'))

        if number_hidden_layers >= 3:
            model.add(layers.BatchNormalization())

        model.add(layers.Activation(activation))
        model.add(layers.Dropout(dropout_rate, name=f'dropout_{i}_{round(dropout_rate * 100)}'))

    model.add(layers.Dense(1, activation='sigmoid'))

    optimizer = Adam(learning_rate=learning_rate)

    model.compile(loss=loss, optimizer=optimizer, metrics=['accuracy', Precision(), Recall(), AUC()])

    return model
```

The module mostly creates a standard Sequential-class Keras model with minimal dynamic changes (such as BatchNormalization based on the number of layers).

Sklearn-style Gridsearch

Using the KerasClassifier wrapper https://www.tensorflow.org/api_docs/python/tf/keras/wrappers/scikit_learn/KerasClassifier, a basic grid-search function was created to be applied to all models.

Listing 6: training loop

```python
def nn_gridsearch(
    make_model_function,
    x_train: np.ndarray = None,
    y_train: np.ndarray = None,
    params: dict = None,
    epochs: int = 100,
    validation_split: float = .1,
    patience: int = 10,
    batch_size: int = 16,
    n_iterations: int = 10,
    early_stop: bool = True,
    save_logs: bool = False,
    verbose: int = 1):

    keras_cl = KerasClassifier(
        make_model_function,
        batch_size=batch_size,
        shuffle=True,
        verbose=verbose)

    rnd_search_cv = RandomizedSearchCV(
        keras_cl,
        params,
        n_iter=n_iterations,
        cv=3,
        verbose=2,
        n_jobs=-1)

    callbacks = []
    if early_stop:
        callbacks.append(EarlyStopping(patience=patience, monitor='val_loss', mode='min'))
    if save_logs:
        callbacks.append(TensorBoard(logdir()))

    rnd_search_cv.fit(
        x_train, y_train,
        epochs=epochs,
        validation_split=validation_split,
        callbacks=callbacks)

    return rnd_search_cv


if __name__ == '__main__':

    grid_parameters = {
        'number_hidden_layers': list(range(1, 8)),
        'neurons': np.arange(1, 100).tolist(),
        'learning_rate': reciprocal(3e-4, 3e-2).rvs(1000).tolist(),
        'dropout_rate': np.arange(.2, .6, .1).tolist(),
        'alpha': np.arange(.2, .35, .05).tolist(),
        'activation': ['elu', 'selu', 'relu']}

    grid = nn_gridsearch(
        make_model,
        data['x_train_processed'], data['y_train'],
        grid_parameters,
        n_iterations=3)

    best_model = grid.best_estimator_.model
    best_model.save(model_path)
```

Note that both the gridsearch and model function are slightly simplified here, the original is again found at https://github.com/PaulBFB/master_thesis/blob/main/nn_gridsearch.py. Also, to quickly apply the gridsearch function to progressively larger segments of the original data, a script https://github.com/PaulBFB/master_thesis/blob/main/boost_experiments.py was used; which is nothing more than an iteration over the different sized data parts, applying gridsearch to all of them, in order to run in the background (or on remote machines, as some of these tests may take an exceedingly long time, based on the size of the grid and the hardware they run on).

As for the results with decreased data, the following steps were automatically performed:

- shuffle the data randomly, take a small subset from it
- train a generator on it, either a Wasserstein-GP or DCGAN-adapted style generator
- perform gridsearch on it, from the base model creation function
- record the results

All these steps were performed on a subset of the entire data, the training set, the remaining 20% of the data, the test set, was only used for evaluation. Where Gridsearch was applied, a subset of the training data was withheld for Crossvalidation.

The resulting graphic is always formatted in the same way; height of the bar denotes accuracy of the best model, color of the bar denotes the type of boosting that was applied, and the bar position on the x-axis denotes by how much the data was boosted.

Boosting factor ranges are (somewhat arbitrarily chosen, after multiple experiments):

- +20%
- +30%
- +50%
- +200%

For clarity, the accuracy of the unboosted data is marked by a line with its' exact value, to clearly denote performance differences.

Data Size 0.3 - 0.8

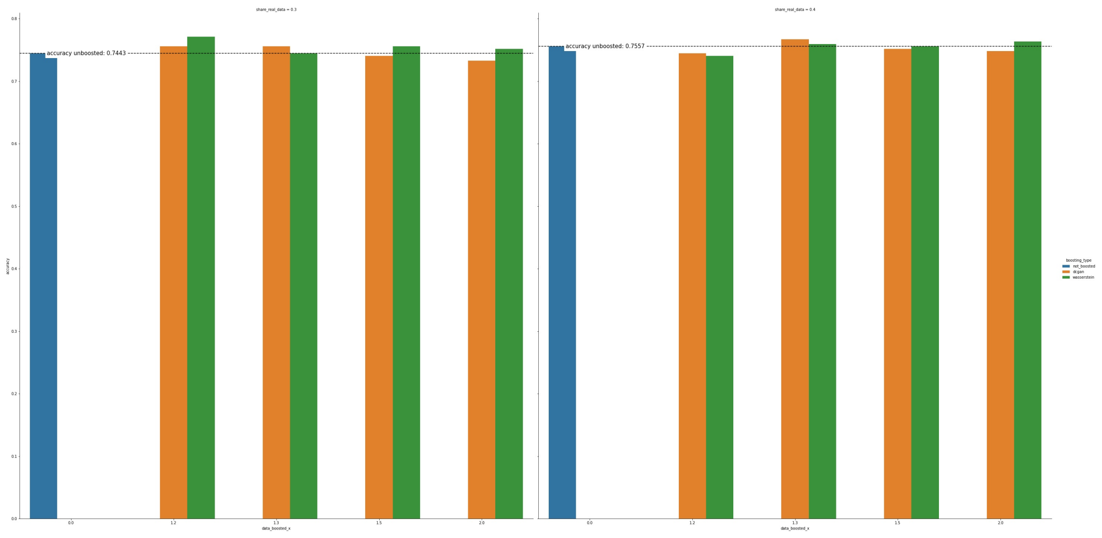

Figure 9: Experiment with data sizes 30%, 40% of original data

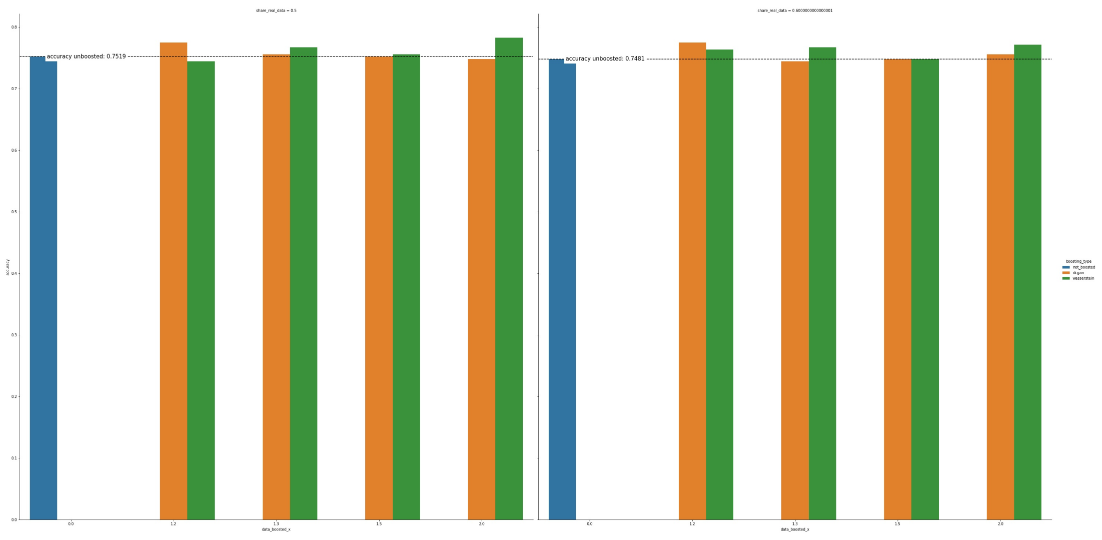

Figure 10: Experiment with data sizes 50%, 60% of original data

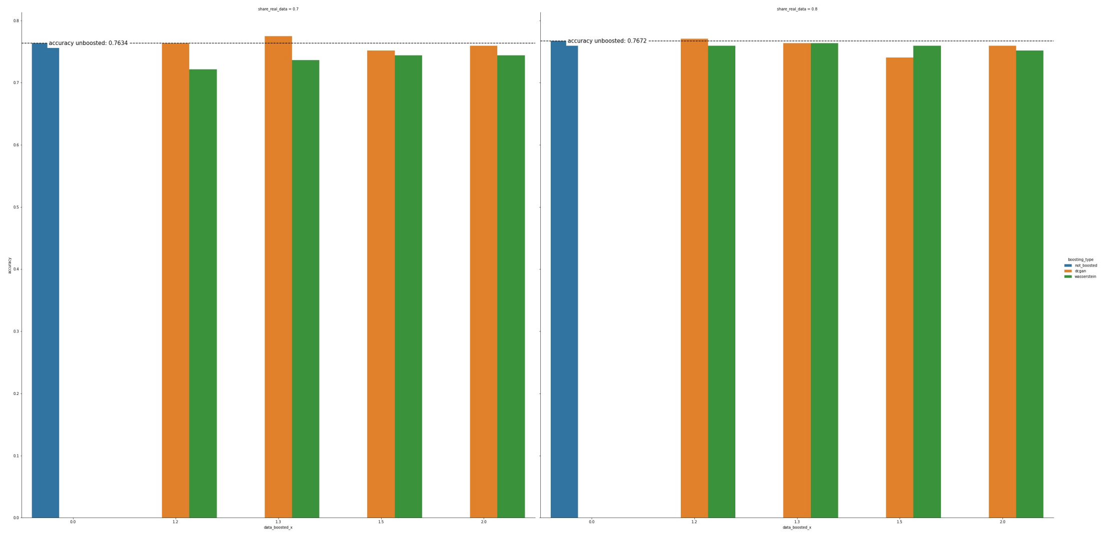

Figure 11: Experiment with data sizes 70%, 80% of original data

As can be seen quite plainly, while at some sizes there appears to be a (very very slight) but static gain in performance, the variation is well within the range of simple random fluctuations due to the random grid search performed on the networks.

Data Size 1

Performing the same test on the entire data:

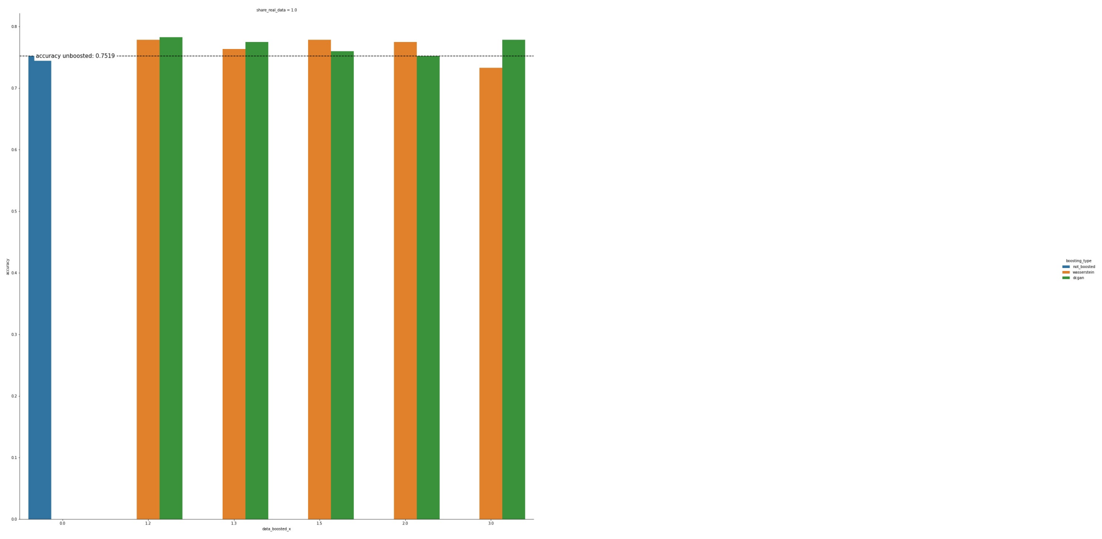

Figure 12: Experiment with data size 100% of original data

appears to yield a performance gain, which is not significant either, but seems more consistent.

Larger Data Sizes

Finally, it was tested whether or not boosting the data to a more significant size would deteriorate performance on these standard models - which would stand to reason. Any patterns in the training data that do not represent the entire dataset well, would be strongly magnified and therefore skew the result, effectively magnifying the model's generalization error.

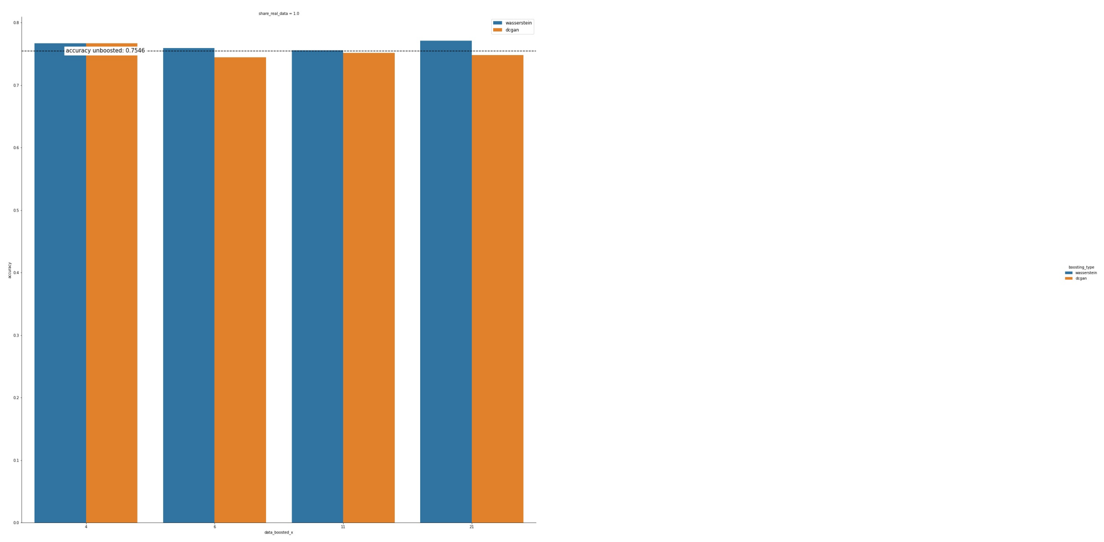

Figure 13: Experiment with data sizes +300% - +2000% original data

The format of the experiment remains the same, the only change being that only models trained on the boosted data are recorded here. The line representing "unboosted" performance takes its' value from the model that was trained on the entire training set.

While the fluctuation in performance remains on the same scale, which quite probably reflects nothing more than random noise, it seems remarkable here, that model performance does not, in fact, degrade.

Elaboration on this is found in the final chapter, it just seems prudent to note here that the axiom that should come to mind here is:

"if you torture the data long enough, it will confess"

since creating and tuning the entire model architecture and all the helper modules as well as the gridsearch itself was done on the same dataset (so far), to say that this data was tortured is probably quite the understatement.

Therefore, it seemed to be only prudent to use this methodology to test performance on different data sets, as well was with different types of models.

This accomplished multiple things simultaneously:

- it tested the degree to which fitting the generator and boosting the training data was portable to other classification datasets
- it tests to what extent the impact on model performance by the boosted data is simply due to an extreme amount of overfitting on the data used so far, as well as information leakage from the training to the test set, due to extensive gridsearch runs.

5.2 Different Model Types and Datasets

In order to compare the impact of this type of boosting on different types of models, a number of standard models were selected from scikit-learn:

- Logistic Regression Classification
- Support Vector Machine Classification
- Random Forest Classification
- Decision Tree Classification
- K-Nearest-Neighbor Classification
- Gaussian Naive Bayes Classification
- Dense Neural Network, for reference

Just a very brief summary of each type of classification (except for neural networks) follows here. All these models were implemented with their **default values** from scikit-learn version 1.0. As noted before, the very sensible default values in scikit-learn are described also by Buitinck et al. (2013) and have served as guidelines of code implemented here. Deep descriptions of these basic models are out of scope of this paper, as they only serve to test the performance of the approach for generation of synthetic training data on new datasets.

The relevant documentation is linked in all sections.

Logistic Regression

Logistic Regression classification, based on scikit-learn. using a liblinear solver as described by Fan et al. (2008).

Logistic Regression calculates a log-probability from the input vector for each observation based on a parameter vector θ. The cost function for misclassification of observations is then normalized by a solver, which constitutes a stochastic gradient descent. Therefore, Logistic Regression in this case behaves similarly to a single layer neural network with one neuron per input parameter.

The regularization that was applied (by scikit-learn default) was ℓ_2 regularization, penalizing parameters.

Documentation can be found here: https://scikit-learn.org/stable/modules/generated/sklearn.linear_model.LogisticRegression.html,

Support Vector Machine Classification

Support Vector Machine Classification (SVC) attempts to formulate a decision boundary around training instances. The goal is to formulate a hyperplane, n-1 dimensional space (where n is the number of attributes in the training data) which maximizes the distance to the nearest training instance. Originally formulated by Vapnik et al. (1995) it is a mathematically exceedingly elegant classification solution. In cases where the problem lends itself to projection or dimensionality reduction, SVCs are known to perform very well.

Documentation found here: https://scikit-learn.org/stable/modules/generated/sklearn.svm.SVC.html.

Random Forest & Decision Tree Classification

Decision trees, first described by Quinlan (1986), attempt to iteratively bisect the data in a fashion which increases the resulting halves in terms of their "purity". Purity in this case is described as how "unbalanced" the classes of the target variable are in the resulting halves, the more unbalanced the better (usually tracked in terms of gini impurity or entropy). By recording these split points, decision trees effectively learn how to halve the data until all resulting observations are in pure subsets.

Decision trees are intuitively well understandable, and are robust against outliers and feature scale, but are prone to overfit training data depending on their depth (the amount of times the classifier is allowed to split the data in training).

Documentation can be found here: `https://scikit-learn.org/stable/modules/generated/sklearn.tree.DecisionTreeClassifier.html`

Related to decision trees, **Random Forect Classifiers** train multiple decision trees on different subsets of the data and aggregate their results, and constitute a uniquely adaptable and robust ensemble method.

Documentation can be found here: `https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html`

K-Nearest-Neighbor Classification

K-Nearest-Neighbor Classification (KNN) calculates all distances between training observations based on their attributes in n-dimensional space (where n is the number of attributes in the training data) and classifies each observation by its' K nearest other observations. The default value of neighbors that is used here is 3 (in order to be less prone to overfitting the training data).

Documentation can be found here: https://scikit-learn.org/stable/modules/generated/sklearn.neighbors.KNeighborsClassifier.html

Gaussian Naive Bayes Classification

The Naive Bayes Algorithm estimates the probability of a given outcome (1/0 in this case) given the value of a given attribute; as per Bayes' Theorem of conditional probability. Here, the key assumption is that all attributes are normally distributed (hence gaussian) and furthermore that the features are all independent of each other (a strong assumption which makes this model fairly brittle in practice, hence naive).

Treating all attributes independently of one another eliminates the requirement of observing all possible permutations of features for all outcomes.

Documentation can be found here: https://scikit-learn.org/stable/modules/generated/sklearn.naive_bayes.GaussianNB.html

5.2.1 Reference - model fitting on the titanic dataset

All these models have been fit on the titanic dataset for reference. In order to make these models comparable, they have all been fit on all datasets with the same parameters (as mentioned in the sections) as well as using 80% of their data as training data, withholding 20% for performance testing.

All training data have then been used to fit a Wasserstein-GP generator and increase their training data size by 20% (since this seemed to be the most consistent and promising configuration).

For clarity, the baseline for a "naive" prediction (that is, always simply predicting the most common class) is also shown, in order to demonstrate whether any of the models exhibit predictive power beyond simply making the safest guess (i.e. a static model).

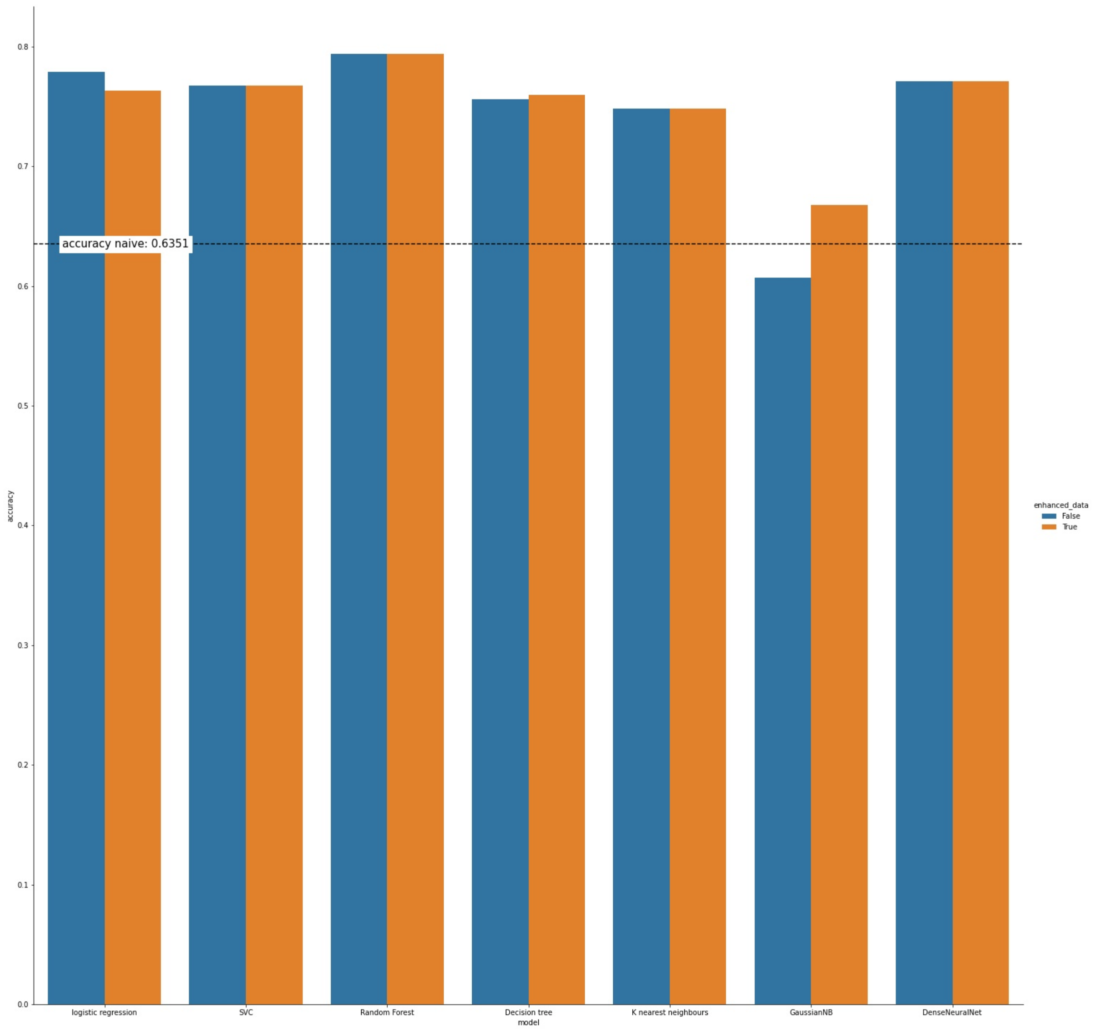

Figure 14: Comparing all model type performances, training data size boosted by 20% with Wasserstein-GP generator

The outcome here seems quite consistent, with no change in most models, deterioration of performance does not actually occur, which is promising. Given the fact that this dataset is the one the Wasserstein-GP architecture was built on any further deductions seem spurious.

5.3 Fitting the Generator and models on completely new data

Two new datasets were selected as test sets for the architecture that was used:

5.3.1 The Wine Quality Dataset

Described by Cortez et al. (2009). The dataset contains 6000+ observations of wine quality with 11 attributes and one binary target variable (quality). The attributes describe characteristics such as citric acid, chlorides, residual sugar, alcohol...

Data preparation steps (using sklearn Pipelines):

- target variable (quality) changed to binary (any quality over 6 is considered quality wine)
- imputation of median value into columns with missing data ("fixed acidity", "pH", "volatile acidity", "sulphates", "citric acid", "residual sugar", "chlorides", number of missing values <10)
- OneHotEncoding of categorical attribute "wine type"
- MinMaxScaling of numerical attributes

Again, feature engineering of the data was deliberately excluded, just as with the titanic dataset. The steps taken are the bare minimum in order to enable machine learning at all by design. As a final note, the dataset used here is fairly unbalanced, roughly 80% falling into the more common class (low quality wine). More on this in the discussion.

Fitting the Wasserstein-GP model on this data, after preprocessing but without changing any of the default values (layers, number of epochs, architecture, loss function, dynamically building the models from the input shape) yielded the following training log:

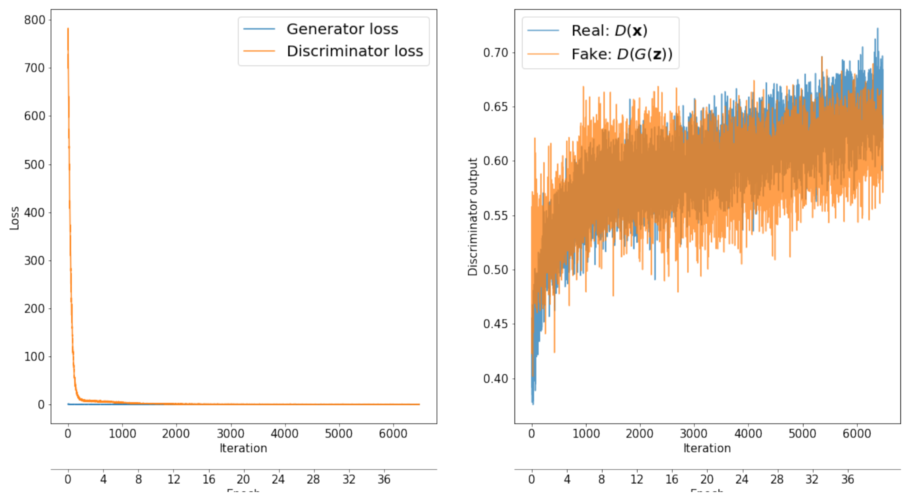

Figure 15: Wasserstein GAN with gradient penalty, 2 layers; left - losses of generator/discriminator right - the probabilities real/fake assigned by the discriminator

Encouragingly, the loss for both modules seems to decrease in much the same way as on the original dataset, and the equilibrium denoted by real and generated data clustering together, each at around 0.5 probability by the discriminator seems to hold fairly stable as well, even though the dataset has different size and shape from the original.

Note, details on the training module found here: 4.2.2

The result of all boosted models, again using default values, analogous to the comparison on the titanic dataset:

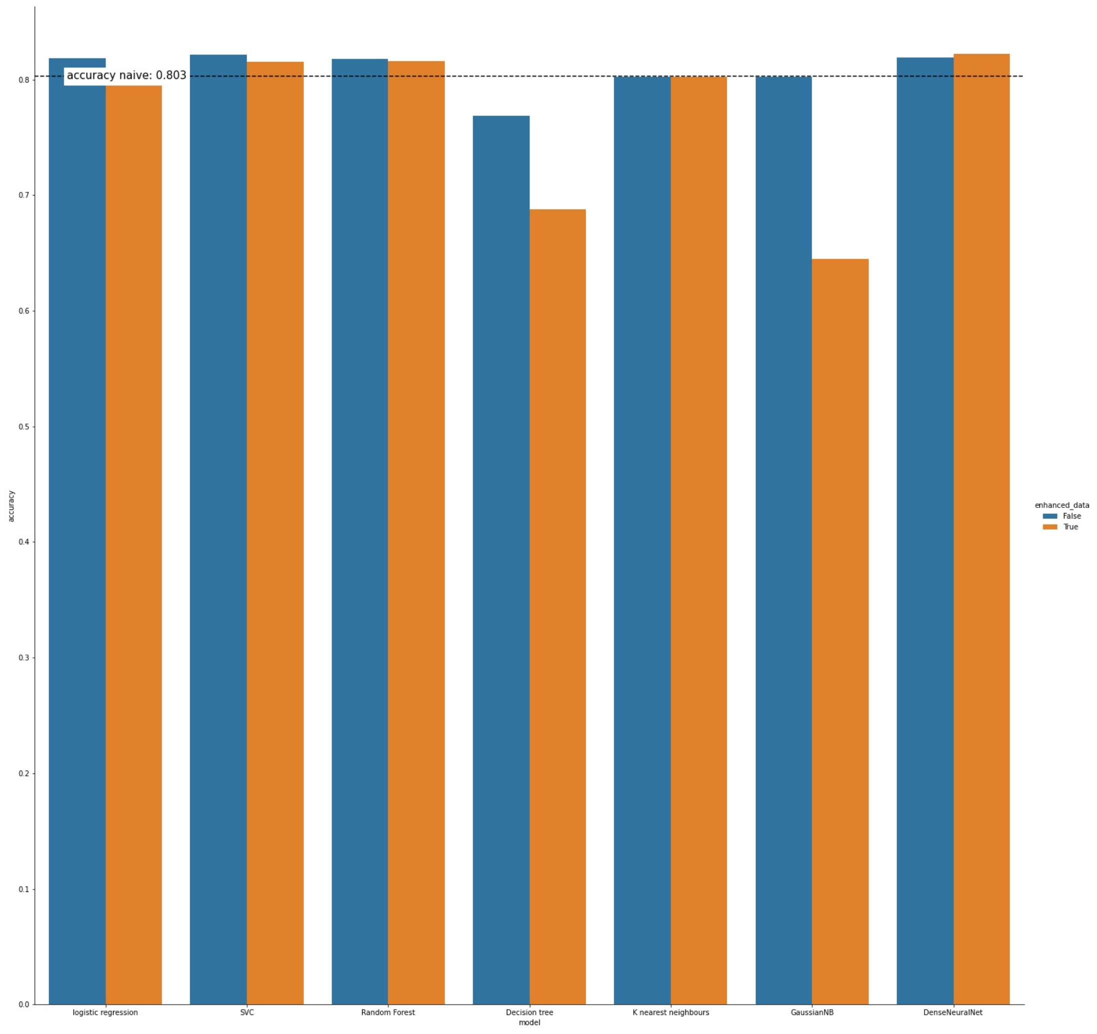

Figure 16: Comparing all model type performances, training data size boosted by 20% with Wasserstein-GP generator

After the generator was trained on the entire training data (withholding a test set completely to avoid information leakage), in the interest of consistency with the original model, the training data was boosted by 20%.

Now immediately apparent here seems the fact that most models barely exhibit any predictive power "out of the box" at all - which seems fair, given that there appears to be a rare class problem. The nn seems to be fairly robust against this, even gaining some predictive power from boosting the data. Also, the models with the biggest (negative) impact in performance appear to be the most brittle models, Decision Tree and Naive Bayes. Promisingly, the other models (while not gaining performance) appear to barely be affected at all, which seems to indicate that the generated data overall reflects the pattern in the training data.

A more detailed analysis follows in the discussion.

5.3.2 The Pima Indians Diabetes Dataset

A dataset originally created by the National Institute of Diabetes and Digestive and Kidney Diseases in order to classify diabetes in female patients.

The dataset includes 8 diagnostic markers:

- Pregnancies
- Glucose
- BloodPressure
- SkinThickness
- Insulin
- BMI
- DiabetesPedigreeFunction
- Age

As well as an outcome column.

Initially described by Smith et al. (1988), classifying diabetes using neural networks.

The dataset is fairly balanced, with around 65% of outcomes in the more common class, making it fairly similar to the titanic dataset in this regard (63%, for comparison).

Data preparation steps (using sklearn Pipelines):

- Imputation of missing values
- smoothing of extreme outliers (.05 - .95 percentiles)

- MinMaxScaling of numerical attributes

Again, extensive preprocessing or feature engineering was excluded. The steps taken are the bare minimum in order to enable machine learning at all by design.

The Wasserstein-GP model was fit automatically on the data, after preprocessing but without changing any of the default values (layers, number of epochs, architecture, loss function, dynamically building the models from the input shape) yielded the following training log:

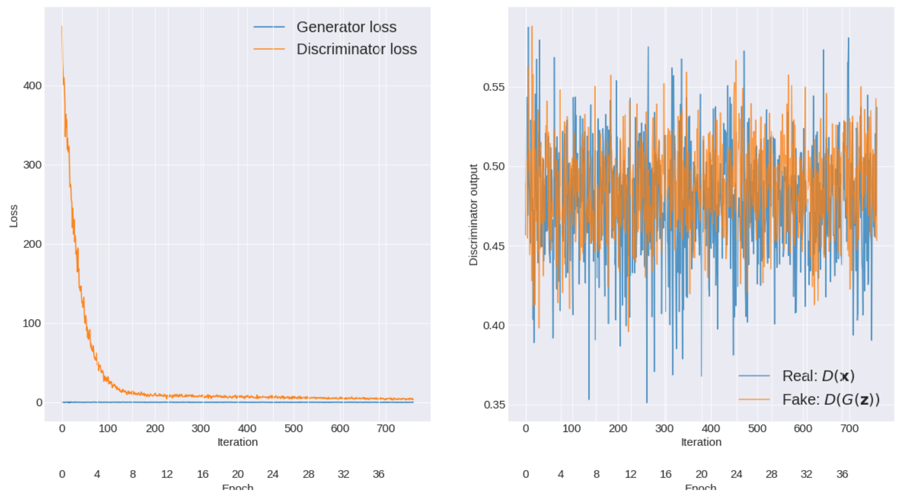

Figure 17: Wasserstein GAN with gradient penalty, 2 layers; left - losses of generator/discriminator right - the probabilities real/fake assigned by the discriminator

Once again, even with the data being quite similar to the original dataset the architecture was built on, the components seem to establish an acceptable equilibrium quickly, which also appears to be fairly stable. With more details in the discussion, this seems to point towards the architecture being fundamentally sensible.

After training the generator agnostically i.e. out of the box only using the default values, the training data was boosted by 20% and the standard models were trained on the data for comparison:

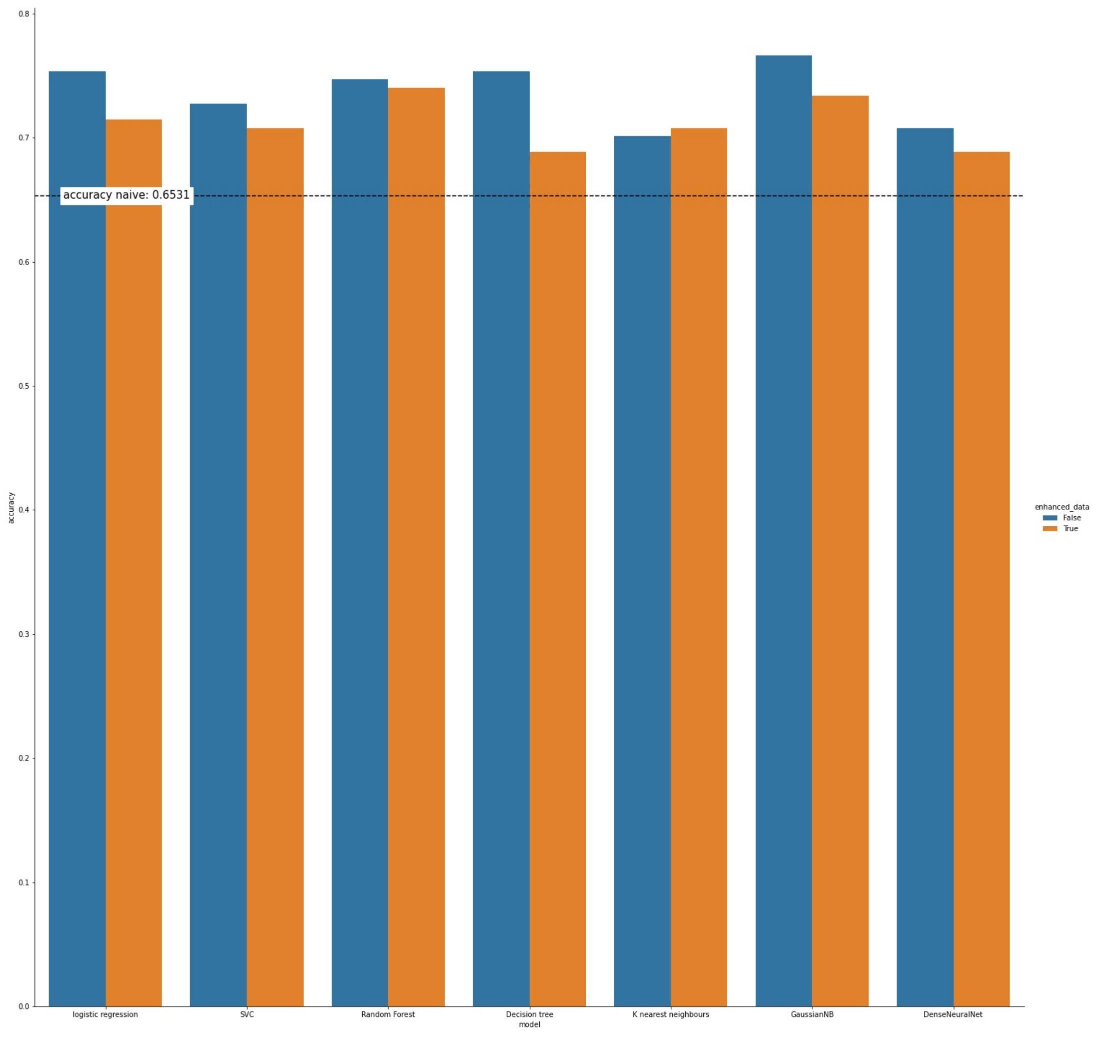

Figure 18: Comparing all model type performances, training data size boosted by 20% with Wasserstein-GP generator

The impact of inserting synthetic data seems to be less pronounced than on the performance of classifiers in the wine quality dataset (16) which may be due to the structural similarities between this dataset and the titanic dataset (specifically its' class balance). A detailed discussion in the final chapter.

5.4 Replacing Training Data with Synthetic Data

Finally, since it seems pertinent, it was tested what the effect of **entirely replacing the training data** would have on the performance of models trained on entirely synthetic data.

To test this, the following steps were performed on the titanic, diabetes and wine datasets:

- splitting the data into training and test data (20% withheld)
- fitting a a Wasserstein-GP GAN on the training data
- completely discard the real training data and replacing it with synthetic data of identical size

Note that both the attributes of the data as well as the target attribute was replaced with synthetic data.

5.4.1 Models on purely synthetic Titanic Data

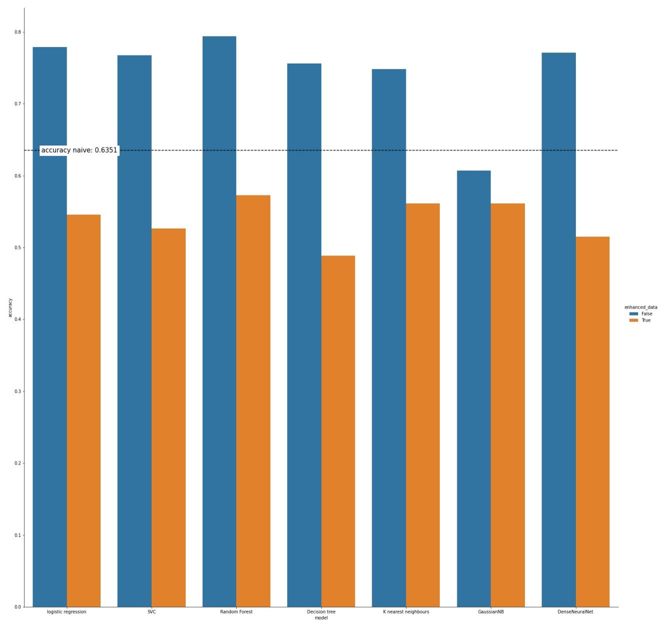

Figure 19: Comparing all model type performances, training data entirely replaced by synthetic data of same size with Wasserstein-GP generator

5.4.2 Models on purely synthetic Wine Data

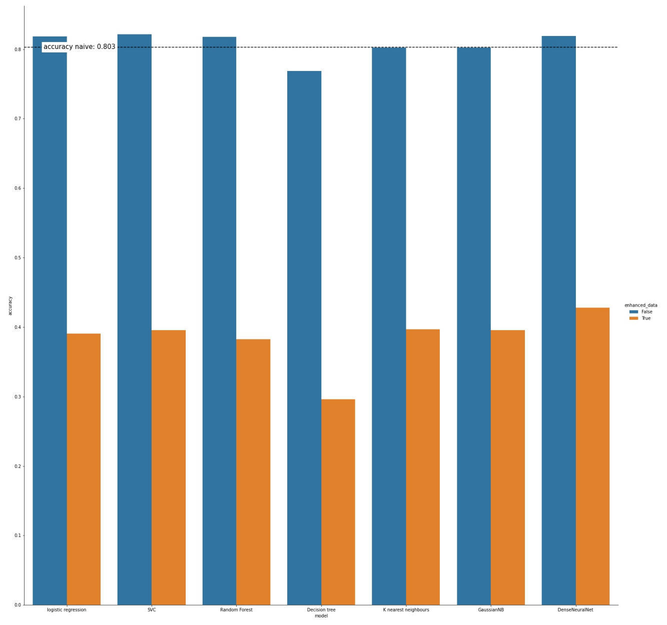

Figure 20: Comparing all model type performances, training data entirely replaced by synthetic data of same size with Wasserstein-GP generator

5.4.3 Models on purely synthetic Diabetes Data

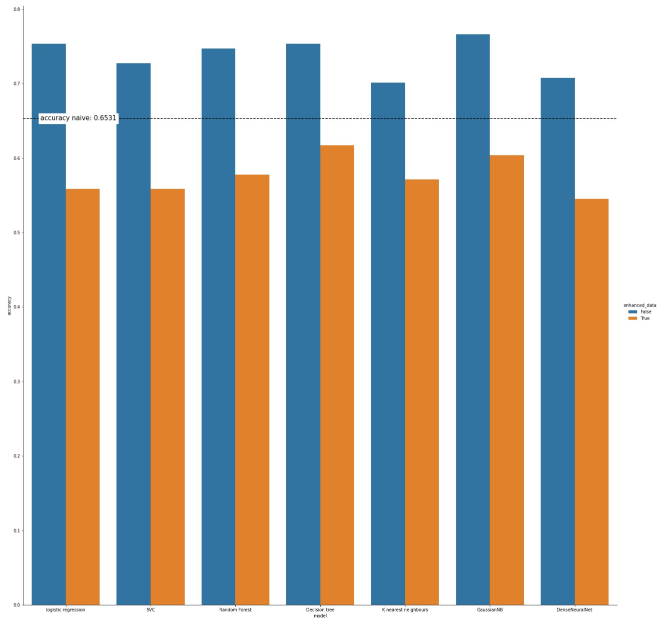

Figure 21: Comparing all model type performances, training data entirely replaced by synthetic data of same size with Wasserstein-GP generator

6. Discussion

Bibliography

Arjovsky, M., Chintala, S., and Bottou, L. (2017). Wasserstein gan.

Buitinck, L., Louppe, G., Blondel, M., Pedregosa, F., Mueller, A., Grisel, O., Niculae, V., Prettenhofer, P., Gramfort, A., Grobler, J., Layton, R., Vanderplas, J., Joly, A., Holt, B., and Varoquaux, G. (2013). Api design for machine learning software: experiences from the scikit-learn project.

Che, T., Li, Y., Jacob, A. P., Bengio, Y., and Li, W. (2017). Mode regularized generative adversarial networks.

Cortez, P., Cerdeira, A., Almeida, F., Matos, T., and Reis, J. (2009). Modeling wine preferences by data mining from physicochemical properties. *Decision Support Systems*, 47(4):547–553. Smart Business Networks: Concepts and Empirical Evidence.

Deng, L. (2012). The mnist database of handwritten digit images for machine learning research. *IEEE Signal Processing Magazine*, 29(6):141–142.

Dumoulin, V. and Visin, F. (2018). A guide to convolution arithmetic for deep learning.

Fan, R.-E., Chang, K.-W., Hsieh, C.-J., Wang, X.-R., and Lin, C.-J. (2008). Liblinear: A library for large linear classification. *the Journal of machine Learning research*, 9:1871–1874.

Farag, N. and Hassan, G. (2018). Predicting the survivors of the titanic kaggle, machine learning from disaster. In *Proceedings of the 7th International Con-*

ference on Software and Information Engineering, ICSIE '18, page 32–37, New York, NY, USA. Association for Computing Machinery.

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

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). Improved training of wasserstein gans.

Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems*, 25:1097–1105.

Liang, K. J., Li, C., Wang, G., and Carin, L. (2018). Generative adversarial network training is a continual learning problem.

McCloskey, M. and Cohen, N. J. (1989). Catastrophic interference in connectionist networks: The sequential learning problem. In *Psychology of learning and motivation*, volume 24, pages 109–165. Elsevier.

Pereira, F., Norvig, P., and Halevy, A. (2009). The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(02):8–12.

Quinlan, J. R. (1986). Induction of decision trees. *Machine learning*, 1(1):81–106.

Radford, A., Metz, L., and Chintala, S. (2016). Unsupervised representation learning with deep convolutional generative adversarial networks.

Shearer, C. (2000). The crisp-dm model: the new blueprint for data mining. *Journal of data warehousing*, 5(4):13–22.

Smith, J. W., Everhart, J. E., Dickson, W., Knowler, W. C., and Johannes, R. S. (1988). Using the adap learning algorithm to forecast the onset of diabetes mellitus. In *Proceedings of the annual symposium on computer application in medical care*, page 261. American Medical Informatics Association.

Smith, L. N. (2018). A disciplined approach to neural network hyper-parameters: Part 1 – learning rate, batch size, momentum, and weight decay.

Suh, S., Lee, H., Jo, J., Lukowicz, P., and Lee, Y. (2019). Generative oversampling method for imbalanced data on bearing fault detection and diagnosis. *Applied Sciences*, 9:746.

Vapnik, V., Guyon, I., and Hastie, T. (1995). Support vector machines. *Mach. Learn*, 20(3):273–297.

A. List of Interview Partners

B. Code Table