UNIVERSITY OF CAMBRIDGE

Harry Graham
Christ's College
hg402

Computer Science Tripos - Part II Project

Deep Learning Techniques for Credit Card Fraud Detection

May 18, 2018

Proforma

Name:	**Harry Graham**
College:	**Christ's College**
Project Title:	**Deep Learning Techniques for Credit Card Fraud Detection**
Examination:	**Computer Science Tripos – Part II, June 2018**
Word Count:	**10,709**
Project Originators:	H. Graham & B. Dimanov
Supervisors:	B. Dimanov & Dr M. Jamnik

Original Aims of the Project

The primary aims of the project are to investigate the feasibility and effectiveness of certain deep learning techniques in application to time series data, in particular for credit card fraud detection (CCFD). More specifically, I aim to experiment with two popular types of architecture, namely Convolutional Neural Networks (CNNs) [1] and Generative Adversarial Networks (GANs)[3]. These have been successful in the image classification space and the aim of this project is to investigate their use in the credit card fraud space. This kind of experimentation of predominantly image-based models, on single dimensional, time-series data is a relatively novel approach for CCFD.

Work Completed

All of the core project aims set out in the proposal have been met, meaning results have been collated and evaluated across the three main components of the project: Baseline Models, CNN methods and GAN methods. As extension work, I implemented a Semi-Supervised approach for the GAN. This was an additional investigation, using the models and the data to classify fraudulent and genuine transactions.

Special Difficulties

None.

Declaration

I, Harry Graham of Christ's College, being a candidate for Part II of the Computer Science Tripos, hereby declare that this dissertation and the work described in it are my own work, unaided except as may be specified below, and that the dissertation does not contain material that has already been used to any substantial extent for a comparable purpose.

Signed	Date

Contents

Chapter 1

Introduction

Credit card fraud is a globally significant and increasing problem. According to the Nilson Report [2], annual global fraud losses reached $22.80 billion in 2016, up 4.4% over 2015. Machine learning has contributed a lot to this problem over the years, helping to automatically learn to classify fraudulent transactions. However, this is still somewhat tedious and clearly, the money lost due to fraud is not decreasing. In addition, we still have this difficult business decision of when to draw the cutoff points between classifying fraud but perhaps allowing more benign transactions to be blocked.

A lot of machine learning concepts have been around for decades, but ongoing research into deep learning architectures and their applications, makes for an interesting experimentation space. The main aims of the project are to explore whether certain deep-learning architectures are feasible to use in the CCFD space and also whether they are effective. The types of model that I experiment with are Convolutional Neural Networks (CNNs) [1] and Generative Adversarial Networks (GANs)[3].

In particular, I investigate the use of these architectures that have had success in the image classification/generation space, in the context of non-image data i.e transactional vectors and time series data. This is something that has recently seen some success[4][5] and the application to credit card fraud data is the novel part.

In the project I benchmark a set of baseline classifiers, which are primarily a handful of out-of-the-box supervised learning classifiers, such as Random Forest. The point of these is to set the scene for experimenting with the data and to see what can be achieved with what is more readily understood, in other words without any 'deep' learning components. In addition, I experiment with some resampling techniques using the models, to indicate if these techniques help the fact that the data is highly unbalanced. I go on to implement a unique cross-validation function that handles resampling in a way that ensures validity, given the unbalanced nature of the data.

In the context of CNNs, I introduce two variations for applying this type of model to the data. One in which I keep the data as vectors and a second in which I exploit a temporal order of the data and use a 'sliding window' approach, making use of convolutional features. This includes creating a time-series appropriate form of cross-validation and also the process of tuning key parameters of my model, such as the window size. In the context of the GAN, the primary goal is to see whether the model is useful for learning the data distribution. This is a primary functionality of a GAN and is important because

if we can learn the distribution of what a fraudulent transaction looks like, it could be useful in the problem space (for example, in data augmentation). I outline two variations in which I change the type of the generator network, which I implement from scratch, including further variations on the adversarial training process. Then, in extension, I try a semi-supervised, multitask learning approach for the GAN, in order to test feasibility as a classification model.

In the project I demonstrate that these models can successfully be applied to credit card fraud data with reasonable success, including promising results for the CNN architectures in particular. I conclude from experiments that convolutional techniques are indeed useful for the problem space, given the results and the simplicity of treating the models as classifiers. This contrasts GAN models whereby diminishing returns of the overhead of tweaking the model, to improve performance, is not as useful given the nature of the problem.

Chapter 2

Preparation

2.1 Software Engineering

This section details the project requirements and early design decisions that were made.

2.1.1 Requirements

The main success criteria of the project is outlined as follows:

1. Baseline Models
    - Compare a handful of supervised learning classifiers, from SciKit-Learn.
    - Using metrics described in the Evaluation section of the project.
    - Experiment with resampling techniques.
    - Implement appropriate cross-validation.
2. CNN Models
    - Implement CNN version 1 - Single vector input approach.
    - Implement CNN version 2 - Time series, sliding image window approach.
3. GAN Models
    - Implement GAN version 1 - Dense generator network.
    - Implement GAN version 2 - Using a CNN model from previous work as the generator network.
    - Experiment with how GAN is used and how it performs on the data.

These were all done more or less in order. Some work overlapped, namely work on auxiliary functionality to allow appropriate cross validation or data preparation etc. More details on specifics is outlined in the implementation chapter.

2.1.2 Tools and Technologies Used

Below I describe and justify where necessary the tools and technologies that I used.

Machine Learning

I implemented work predominately making use of Keras[1](with TensorFlow[2] backend) for CNN and GAN work. I made use of SciKit-Learn[3] for baseline models and some general data manipulation/metric functions.

The reasons for these choices were a mixture of good documentation, popularity & ease-of-use. Using a TensorFlow backend meant that I could use GPU acceleration if needed. I used Keras as a TensorFlow wrapper, so I could avoid writing models completely from scratch but still giving me the flexibility to develop around models and customise to a large extent. Similarly with SciKit-Learn, which has a lot of helpful utility functions for evaluating models and processing data.

Version Control and Project Tools

I hosted my project in a repository on GitHub, used Git for version control, and used virtual environments with pip for project package management and requirements.

I made heavy use of Jupyter[4] Notebooks for writing code in an experimental manner, with immediate execution and feedback.

Languages

My project was entirely written in Python, using the libraries and APIs described previously. This is mainly due to the large ecosystem and documentation surrounding these machine learning libraries in python and also for the ease of use of tools such as Jupyter Notebooks for experimentation.

2.1.3 Starting Point

My project codebase was written from scratch, with the assistance of the tools and libraries mentioned above. Apart from a basic knowledge of supervised learning covered by the part IB Artificial Intelligence course, I had to learn about most of the models and best practises myself, through thorough reading around the topics.

In terms of technologies, I had little prior experience with SKLearn and Keras/TensorFlow. I reviewed some documentation of these before-hand but I also took an agile

[1]https://keras.io
[2]https://www.tensorflow.org
[3] http://scikit-learn.org/stable/
[4]http://jupyter.org

approach to the project, whereby I consulted documentation as and when I needed, during my various milestone sprints. I had significant experience in Python, however, from a summer internship in industry as well as experience with Git.

2.2 Convolutional Neural Networks

Convolutional Neural Networks[1] are a class of deep artificial neural networks, that have seen success in image recognition and other computer vision areas. Unlike typical neural networks, CNNs exploit spatial locality by shared weights.

A CNN typically consists of an input layer, multiple hidden layers and an output layer. Hidden layers are usually convolutional layers, pooling layers and normalisation layers.

Convolutional Layer:
A Convolutional Layer has the following important parameters:

1. Input shape
2. Kernel size
3. Number of filters
4. Stride size

The **kernel size** defines the size of the filter that will be moved over the input shape, shifting by an amount defined by **stride size**. The input is from training data and the shape of this is user-defined and depends on the type of data. For example, this could be an image of shape (24,24) or a transactional vector of shape (30,1). The **number of filters** simply defines how many separate filters we initialise and convolve with the input, to give multiple outputs. Convolving a filter with an input is essentially taking the dot product of all overlapping cells between the input and the filter, thus producing a single value in the output shape.

This can be visualised with an example as follows:

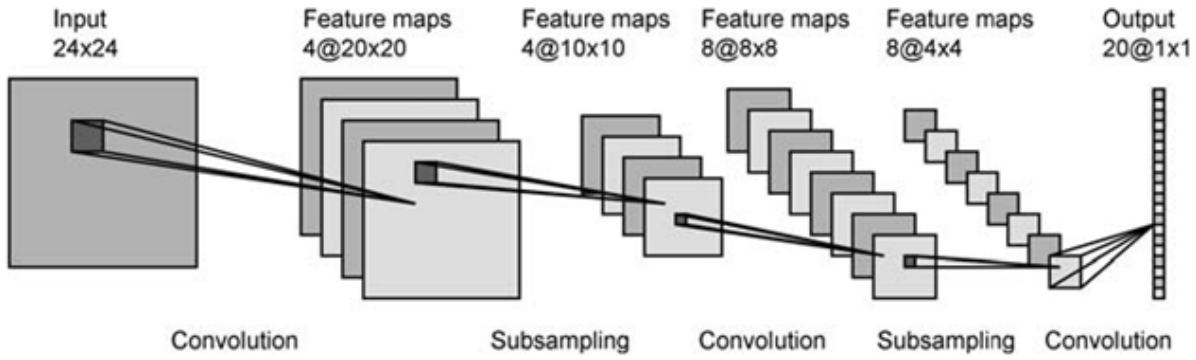


Figure 2.1: Example of a CNN network

We see here that in the first convolutional layer, the input shape is 24x24 and the kernel size is 5x5, which strides over the input shape to produce 4x(20x20) shapes. Here the

number of filters was 4, which is why 4 outputs have been produced. Also, typically the stride size is 1. The number of outputs depends on the number of possible positions of the filter. In general,

$$W = (W - F + 2P)/S + 1$$

Where F is the filter width, P is the padding and S is the stride. In this 24x24 input, with a 5x5 filter we can have $(24 - 5 + 1)$ possible positions of the filter in one direction.

Subsampling / Pooling:
Subsampling is essentially averaging across a group of cells to produce a smaller shape. This therefore produces the same number of 'feature maps' but smaller in size.

Fully Connected Layers:
Fully Connected Layers (FCs) take all the feature maps from a convolutional layer and stack them all together in a traditional neural network, in which every node is connected to every other node (fully connected). This then allows the network to learn the relationships between small parts of the image (or input in general) on a fine-grained level.

2.3 Generative Adversarial Networks

GAN is a framework proposed by Ian Goodfellow, Yoshua Bengio and others in 2014[3]. The idea is that we have two networks: a generator and a discriminator. The objective of the generator network is to produce a specific form of data from random noise. This could be an image, or in this case a fraudulent transaction vector. The discriminator takes in real-life data (from the training set) and also the generated data from the generator network. The objective of the discriminator is to determine whether the current input is real or generated. Effectively, these two networks play a game with each other: the goal of the generator is to fool the discriminator whilst the goal of the discriminator is to catch out the generator. This continues until each network does not improve any more and the GAN stabilises. In practice, however, an alternative outcome is simply that one network is inherently stronger and will overpower the other. Figure 2.1 represents an overview of a GAN network.

Mathematically, we can define the following quantities:

$$X_{x \sim p_{\text{data}}(x)} = \text{Sample from distribution of real data}$$

$$Z_{z \sim p_z(x)} = \text{Sample from distribution of generated data}$$

$$G(z) = \text{Generator Network}$$

$$D(x) = \text{Discriminator Network}$$

The process of training for a GAN is like a min-max game between the two networks, and can thus be represented by the following value cost function:

$$\min_G \max_D V(D, G)$$

where

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

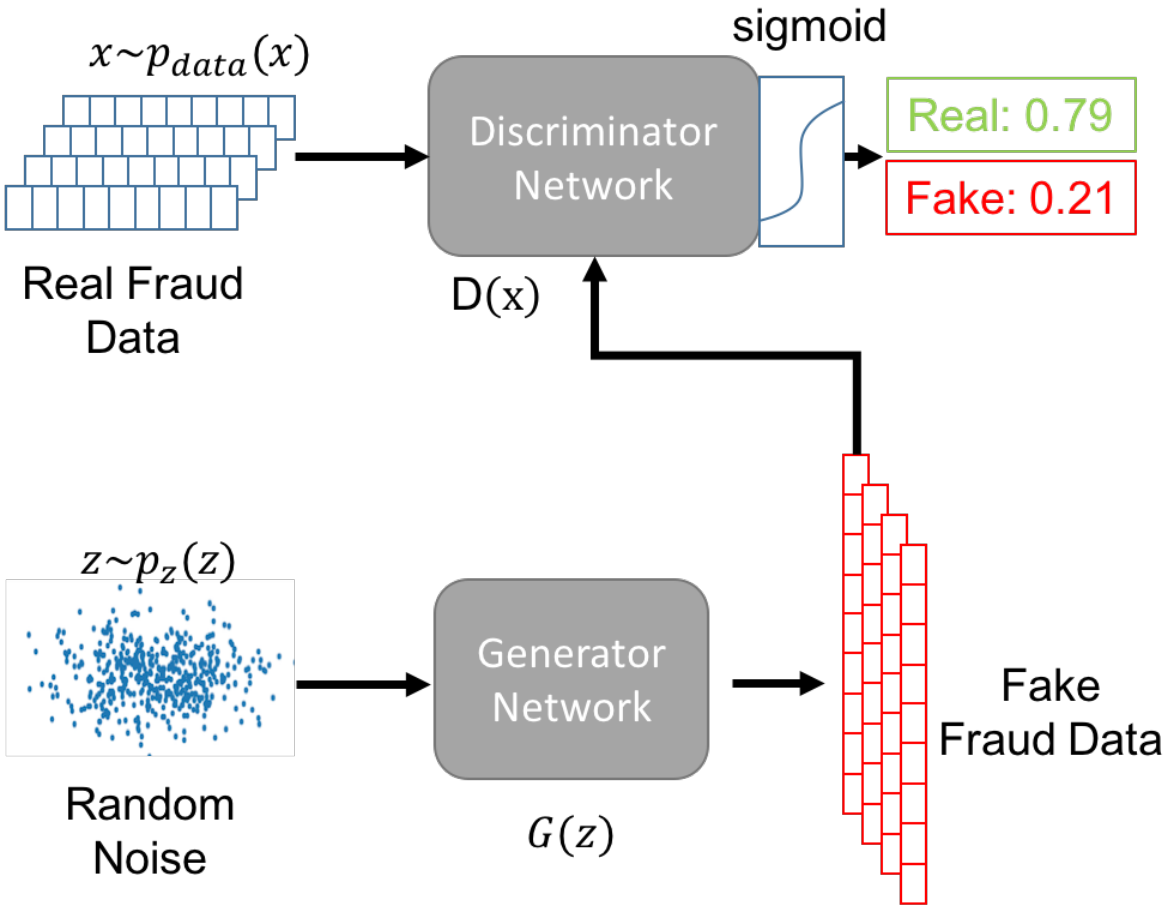


Figure 2.2: Overview of a GAN network

The first term in this equation represents the quantity of the real-distributed data passed through the discriminator network. The discriminator tries to maximise this such that $D(x) \rightarrow 1$. The second term represents generated data passed through the discriminator. The generator tries to minimise such that $D(G(z)) \rightarrow 1$ (i.e the discriminator is fooled by the generated sample).

The steps for training a GAN can be outlined as followed:

Step 1: (a) Take a batch of real data and train discriminator to correctly predict them as real

(b) Take a batch of generated data and train discriminator to correctly predict them as fake

Step 2: Freeze the training of the discriminator network

Step 3: Generate a batch of fake data and use the frozen discriminator to train the generator

Step 4: Repeat the above for n epochs until neither network makes any further improvements

In summary, we alternate between training of the discriminator; to correctly determine real or fake data, and training the generator; on fooling the discriminator. The reason for freezing the weights of the discriminator while we train the generator is exactly so that we don't alter the weights during this process and the generator can use the current state of the discriminator to become better.

2.4 Machine Learning Evaluation Practices

Here I describe some of the main machine learning evaluation concepts that underpin the implementation of this project.

2.4.1 Train and Test Splits

When training a machine learning model, it will often report some metrics or loss function to reflect how the performance increased during the training process. The problem with this, is that our model may not perform well on **unseen** data. In other words, it might not **generalise** well. To overcome this, it is good practise to preserve a testing portion of the data, called the test set, that is not used in the training process. The test set is then used to evaluate how the model performs after training.

Let's say we have $\boldsymbol{m}$ examples $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_m$ each in $\mathbb{R}^n$. In a supervised learning problem, we also have $\boldsymbol{m}$ labels $\{y_1, y_2, \ldots, y_m\}$ in a set $\boldsymbol{Y}$.
In machine learning we wish to find a hypothesis $h : \mathbb{R}^n \rightarrow Y$ that is defined by a vector of weights $\boldsymbol{w}$. It is then common to write $h_w(x)$.

We define a training set as $\mathbf{s} = [(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_m, \mathbf{y}_m)]$. This dataset should then be split, to preserve a testing set.

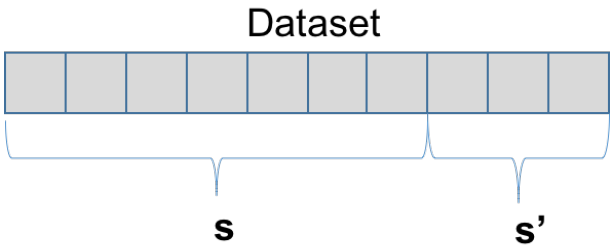


Figure 2.3: A 70:30 train test split of data.

2.4.2 Cross-validation

Cross-validation takes train-test splits a step further. The hypothesis of our model may in fact generalise and perform better on some portions of the data over others and thus performing just one split, may not give the most confident results. Also, when performing these train-test splits, the element of randomness in the split can work against us as even if we average this process say 3 times, it is likely that there is some overlap in the runs and we are not really using the data available to its full capacity. Figure 2.4 represents a type of cross-validation called K-Fold. This is where the the data is split into K sections and with every iteration, the model trains on $K-1$ folds and tests on the remaining fold. This ensures that all the data is beings used to its full extent and that we test on every portion of the data to get the best estimation for the generalisability of the model.

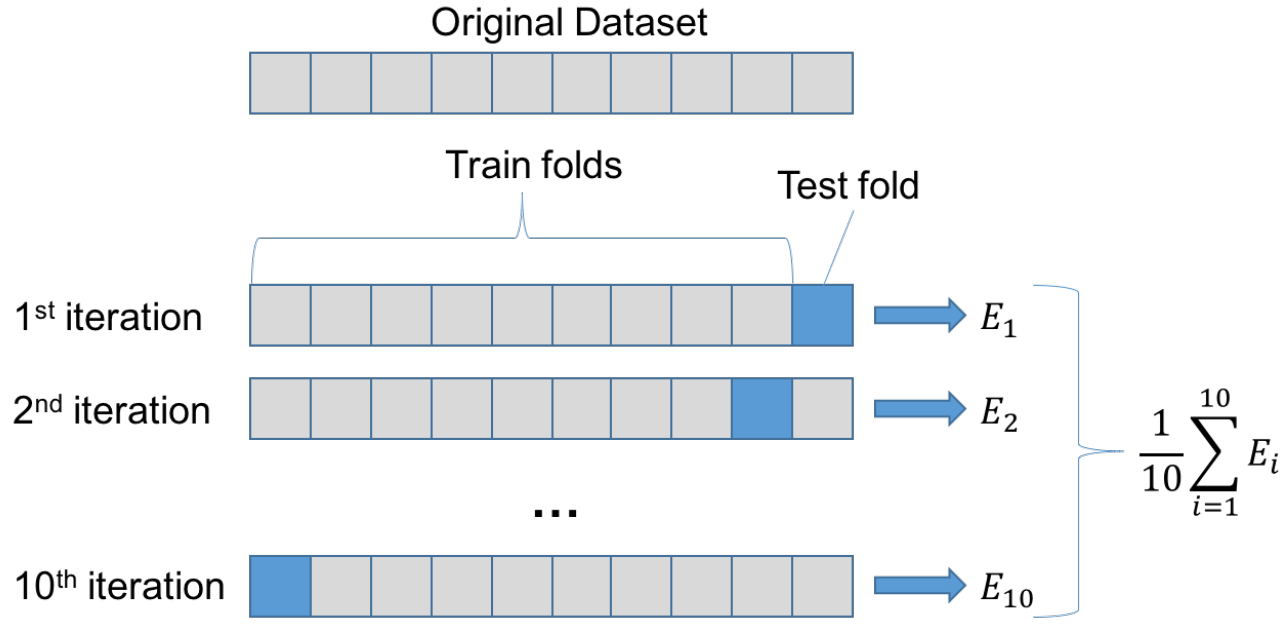


Figure 2.4: K-Fold Cross Validation.

2.4.3 Resampling Methods

When using an unbalanced dataset, like credit card transactions, models will naturally learn to discriminate the majority class better. Therefore, in the baseline models work, I

also experiment with resampling methods. These are ways of balancing the dataset such that the ratio of positive class to negative class is brought to 50:50. I do this to give an example of how these methods affect our results before moving onto the deep learning architectures.

The three main methods I explore are:

1. Undersampling
2. Duplicate Oversampling
3. Synthetic Minority Oversampling Technique (SMOTE)

Undersampling

Undersampling is the process of reducing the majority class down to a lower amount, to balance more with the minority class. This can be achieved by randomly removing samples until the ratio is 50:50 or another specified amount.

Oversampling

Oversampling is the process of increasing the minority class by adding data-points, until a satisfactory ratio of classes is achieved. Machine learning models often benefit from more training data and so oversampling mitigates a problem of undersampling whereby the amount of data our models have to learn on, is restricted. I explore two types of oversampling and address the problems that can occur.

Duplicate Oversampling Duplicate Oversampling is a naive method of increasing the amount of the minority data class, to match that of the majority. This works by taking existing minority data points and simply duplicating them. The resulting dataset therefore includes a lot of duplicated data, of the minority class.

SMOTE Synthetic Minority Oversampling Technique (SMOTE) is another method of increasing the amount of the minority data but not by duplication as before. SMOTE takes the k nearest neighbours of a data point, randomly selects one of these k neighbours and creates a vector between the two. Then a new, synthetic data point is created some random distance along this vector line, by an amount in the range $[0, 1]$.

Cross-validating Correctly

Oversampling techniques affect the validity of the cross-validation process. It no longer is valid to simply oversample the dataset and then perform cross-validation because of the potential of overfitting (depicted by Figure 2.5).

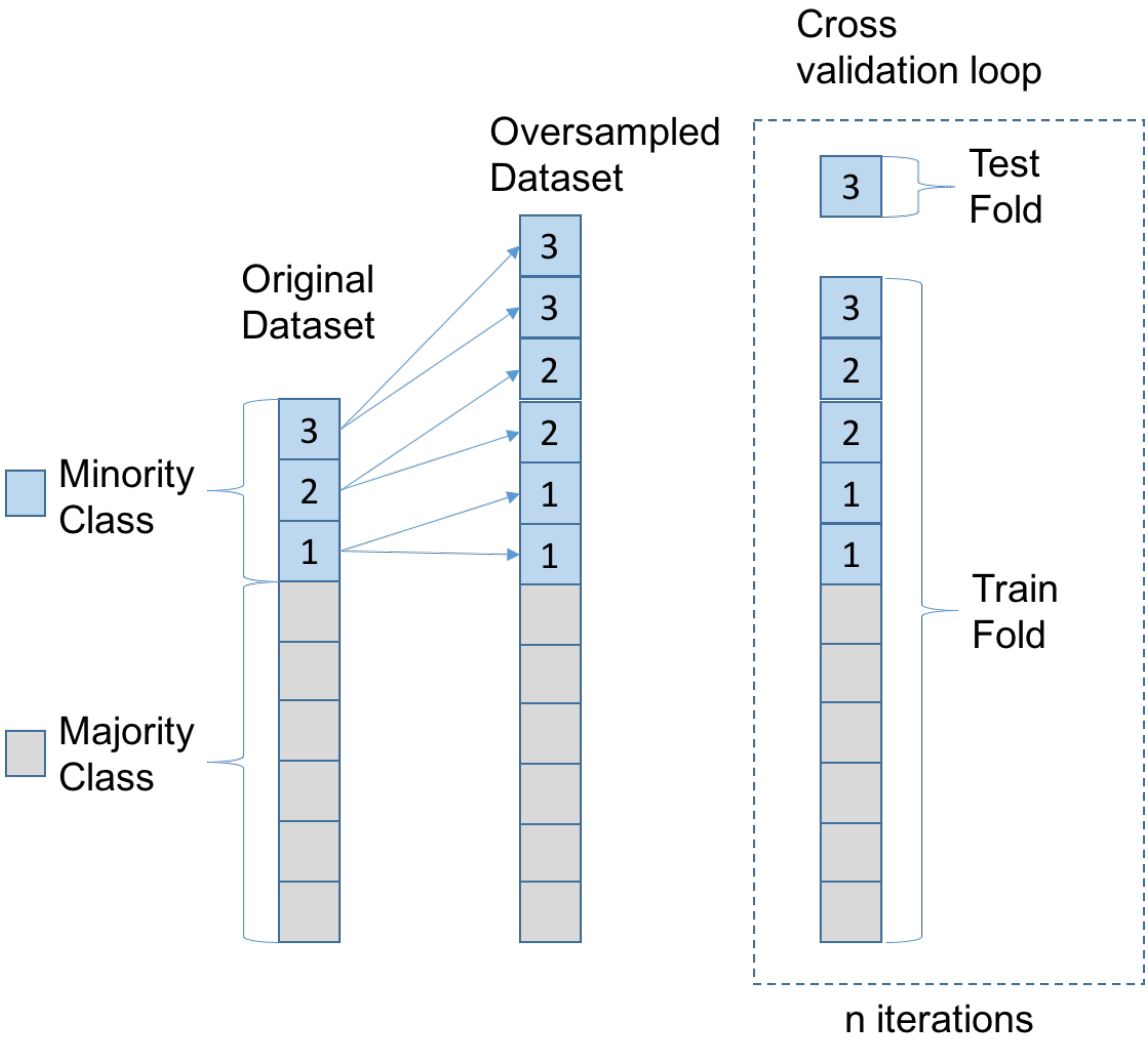


Figure 2.5: Oversampling OUTSIDE the cross-val loop.

Looking at Figure 2.5, the dataset is oversampled and passed to cross-validation (left-to-right). Notice that inside the cross-validation loop, the current train-test split means that some of the oversampled minority class (node 3) is in both the training and testing portions. This means that our model would have seen this data during the training process and will therefore have a bias during the testing and evaluation phase as it already knows how to classify this data.

It is therefore crucial that this is adapted such that any oversampling occurs inside the cross-validation loop, and not before/outside it. Figure 2.6 and Figure 2.7 represent the correct way to handle this, for $n = 1$ and $n = 2$ respectively. For every iteration of the loop, we **first** split the data into train-test first and then oversample the training data fold only. This preserves the testing fold, while keeping a portion of the original dataset untouched for evaluation. Using this methodology is crucial and a lot more effective in testing the generalisability of a model.

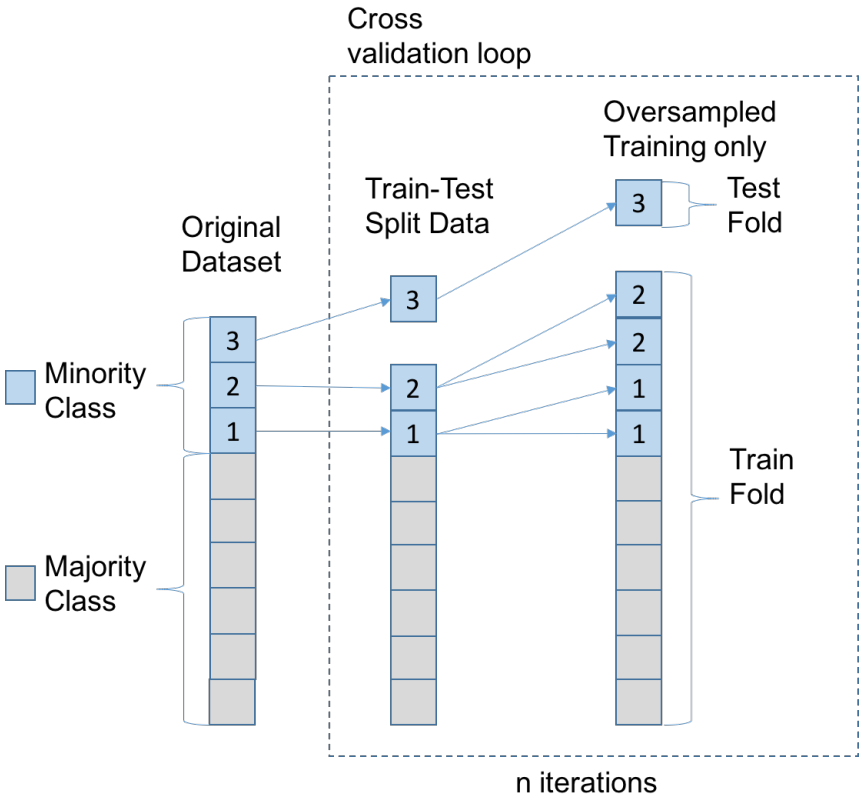


Figure 2.6: Oversampling INSIDE the cross-val loop [n=1].

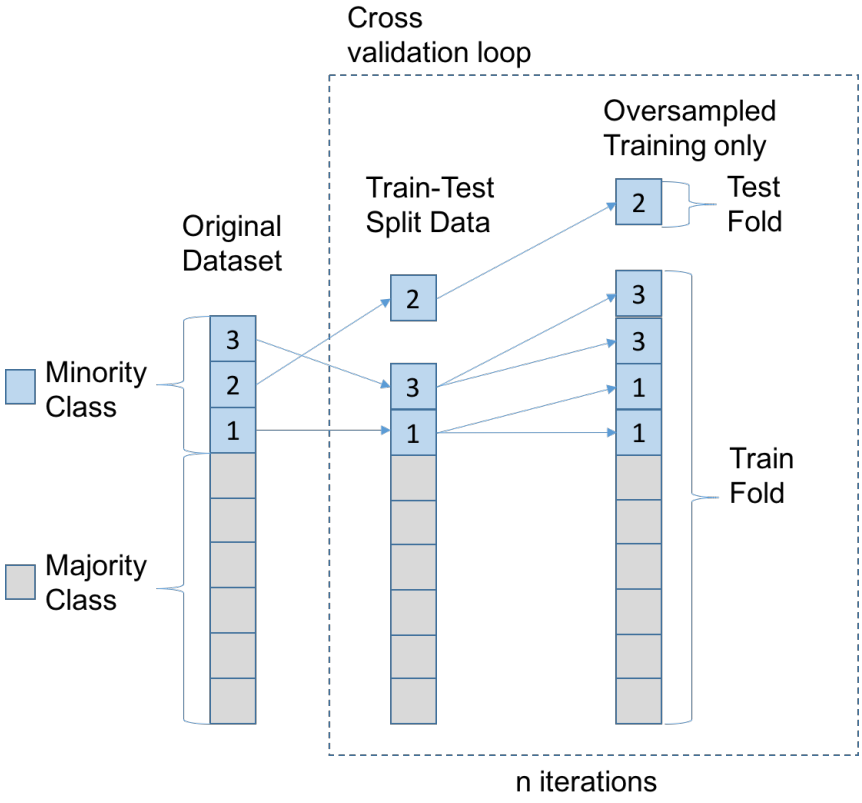


Figure 2.7: Oversampling INSIDE the cross-val loop [n=2].

Chapter 3

Implementation

3.1 Models Overview

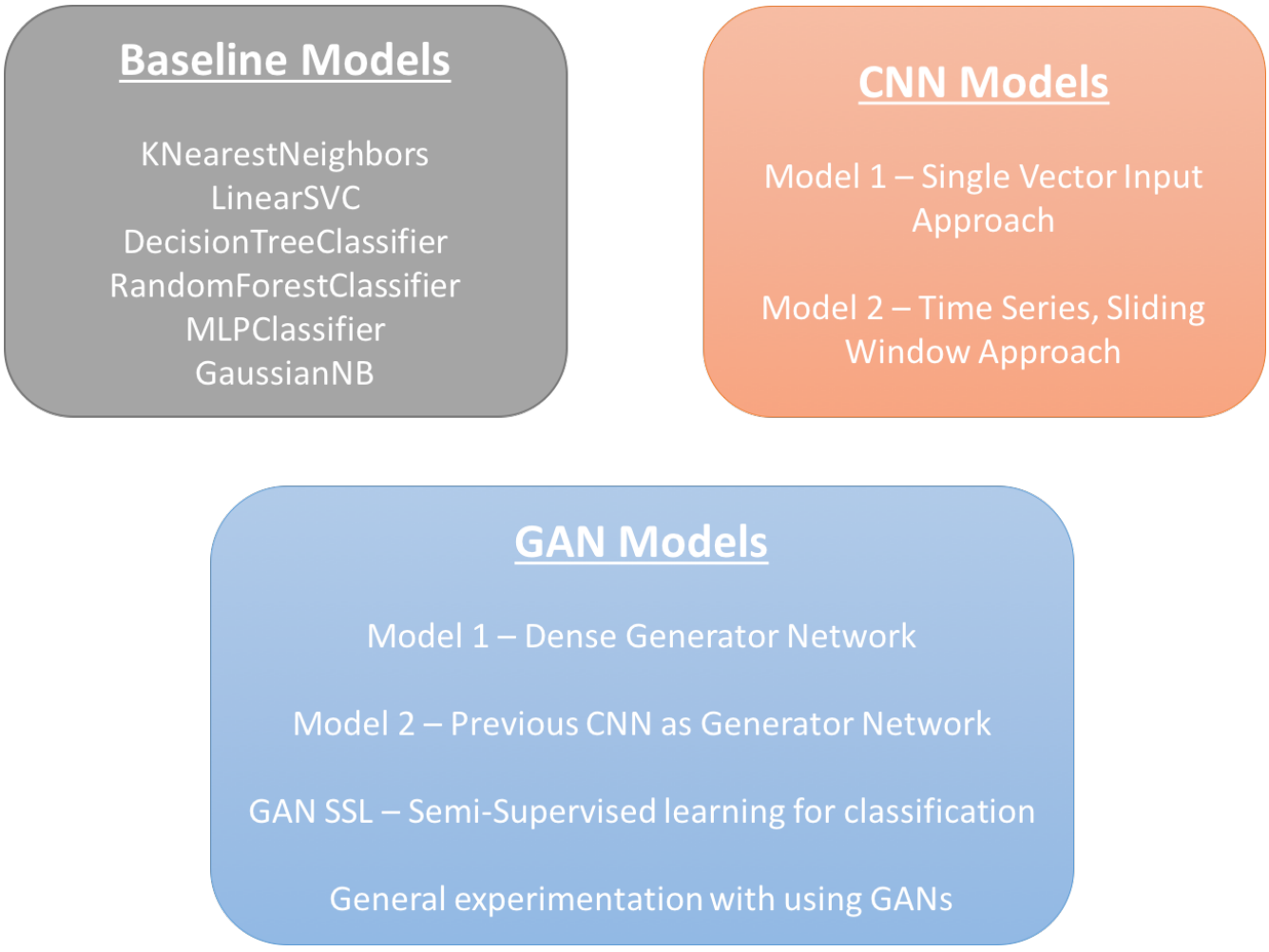


Figure 3.1: Overview of models experimented with.

3.2 Baseline Models

3.2.1 Data Preparation

The Data

The data I use in this project is a dataset from a research collaboration of Worldline and the Machine Learning Group (http://mlg.ulb.ac.be) of ULB (University of Brussels) on big data mining and fraud detection. This was also a popular dataset on Kaggle[1], for some time.
The datasets contains transactions made by credit cards in September 2013 by european cardholders. The time span for transactions are over in two days and there are 492 frauds out of 284,807 transactions. This makes the set highly unbalanced, the positive class (frauds) account for 0.172% of all transactions.

The transactions are in the form of single vectors, with 30 features and the associated class labels (1 or 0). Twenty-eight of these features are results from PCA (Principal Component Analysis) and the other two are Time and Amount.

Feature Scaling

Standardisation involves rescaling the features such that they have the properties of a standard normal distribution with a mean of zero and a standard deviation of one. This is very important in machine learning when features can be a lot larger in magnitude than others. For example, if one feature is age and another is salary in USD then these two quantities will be in different ranges. Therefore, the salary feature may bias classifier algorithms, by deeming that feature to be more important and confusing the weights assignment. Scaling solves this problem.

Therefore one of the first steps was to scale the features, which I did using StandardScaler from SKLearn.

Feature Engineering

In many machine learning and data science applications, feature engineering is something used to add additional features based on calculations and domain knowledge of the data. In this case, I decided not to do any feature engineering. The data is already post-PCA (Principal Component Analysis), which means information about the features is already lost and so it would be difficult to reason about their effectiveness. Therefore, I just employ algorithms on the original dataset features.

Cleaning and Preprocessing

I checked the data to see if any values were missing, or null. This was to ensure that the data was clean to be passed through machine learning networks. There were no missing data in this case.

[1] kaggle.com

3.2.2 Cross Validation Function

Defining my own cross-validation function early on was a crucial step in this project. As explained briefly in the preparation chapter, the way in which cross-validation is performed is very important to ensure no overfitting occurs. As far as implementation is concerned, the built in SKLearn cross-validation function does not provide the ability for changing the evaluation metrics or preserving the validity of the sampling. I wanted very specific metrics and the ability to output in a form convenient to me. Also, I needed to resample data in the correct place which meant I needed to define my own function, based on the standard library, which uses Stratified KFold. This essentially means that the splitting of the data in each fold of the KFold procedure, contains at least some portion of the underrepresented class. This is important as it would be very common that some folds do not contain any of the minority class.

The function uses a utility function from SKLearn, **precision_recall_f1**, which simply reports the precision, recall and f1-score for predicted values. The input is the set of predicted values and the set of true values. The output is the calculated metrics of interest. I use this inside my function to report the metrics we are interested in; something I could not do originally.

This is something I could then use for varying **Resample** methods, seen on Line 15 of Algorithm 1.

3.2.3 Cross Validation the Wrong Way

Test_Train_Split vs Custom cross_val_score using KFold
This concerns the underlying approach for the cross validation loop. SKLearn's test_train_split function splits the data into train and test portions. We can then oversample our training data, fit the model, and make predictions using the test set. One possibility is to simply do this multiple times and average the results. I showed in my experiments that this approach will give better results than expected, in comparison with KFold methods. The reason for this is due to the nature of test_train_split and its randomness, which introduces a slight bias.

Test_train_split allows you to randomly split your data, by giving a parameter that specifies the ratio. However as the split is random, it is likely that there will be overlap in the CV iterations as to which data points are put in the test set. In other words, values selected during one iteration, could be selected again during another iteration. The consequences of this means that the model may not be exposed to particular portions of the data whereby it does not generalise well and these are not captured in the results. Also, the way that test_train_split operates, doe not make maximal use of the data we have.

I ran my classifiers using this approach too, as a means of comparison with what can go wrong and give invalid results. The correct way, as detailed above, uses KFold cross validation to split the data during a loop and to do any customisations or resampling inside this loop.

Algorithm 1 Cross-Validation function

```
1: procedure CUSTOM-CROSS-VAL(X, Y, CLF, N)
2:     skfolds ← StratifiedKFold(n_splits = N)
3:
4:     precisions ← []
5:     recalls ← []
6:     f1scores ← []
7:     elapsedtimes ← []
8:
9:     for train_indices, test_indices in skfolds.split(X,Y) do
10:
11:        X_train_folds ← X[train_indices]
12:        Y_train_folds ← Y[train_indices]
13:        X_test_folds ← X[test_indices]
14:        Y_test_folds ← Y[test_indices]

15:        X_res, Y_res ← Resample(X_train_folds, Y_train_folds)

16:        start ← currenttime
17:        CLF.fit(X_res, Y_res)
18:        end ← currenttime
19:        elapsed ← end − start
20:        elapsedtimes.append(elapsed)
21:        y_pred ← CLF.predict(X_test_folds)
22:        stats ← precision_recall_f1(y_pred, Y_test_folds)

23:        precisions.append(stats[0])
24:        recalls.append(stats[1])
25:        f1scores.append(stats[2])
26:    end for
27:    return [ mean(precisions), mean(recalls), mean(f1scores), mean(elapsedtimes) ]
28: end procedure
```

3.2.4 Resampling functions

To implement resampling functions I made use of the **imbalanced-learn**[6] library. This is an API that focused on methods for unbalanced datasets, including oversampling and SMOTE techniques. They have written functions that do exactly what I needed so I made use of their work in this small part of my project.

I created simple functions in the form of:
Input: X data, Y data
Output: X-res, Y-res
Where X-res and Y-res are the resampled X and Y data.

For **Duplicate Oversampling** I used **imblearn.over_sampling.RandomOverSampler.** This does as described Chapter 2. The method randomly selects a datapoint and duplicates it, or in other words 'selecting random data points, with replacement'. For **SMOTE**, I used **imblearn.over_sampling.SMOTE**. The default parameter of k_neighbors was used, which is a value of 5 and it suitable for this use. For **Under-Sampling**, I created a small function myself that makes use of NumPy random to randomly select indexes from the majority class, in order to remove data points and bring the ratio down.

3.2.5 Collecting Results for All Classifiers

By using the carefully constructed cross-validation algorithms defined above for the various resampling methods, I constructed a general algorithm to loop through the classifiers I am interested in. Then, I run cross-validation for each of them, for each resampling method and collating results in a Python DataFrame log.

The outline is described by Algorithm 2.

I used Python DataFrames as they were a convenient method of storing the results and meant I could construct tables of results in a very organised way. Also, DataFrames have nice methods for plotting data and saving the results to disk.

3.3 Convolutional Neural Network Models

3.3.1 Overview

In this section, I describe the work I completed in constructing the CNN models and retrieving results using these models. I experimented with two main approaches that were briefly designed and outlined in the initial stages of the project. The main goal here is to see if it is suitable and indeed effective to use these types of deep learning models on time series data, specifically with the credit card fraud data.

Algorithm 2 All Classifiers Run

```
procedure ALL-CLASSIFIERS-RUN

    classifiers ← [KNeighborsClassifier, LinearSVC, DecisionTreeClassifier,
RandomForestClassifier, MLPClassifier, GaussianNB]
    columns ← ["precision","recall","f1","auc","training − time"]
    log_original ← DataFrame(columns = columns)
    log_under ← DataFrame(columns = columns)
    log_over ← DataFrame(columns = columns)
    log_smote ← DataFrame(columns = columns)
    for clf in classifiers do
        log_entry_original ← cross_val_original(data, clf, 3)
        log_entry_under ← cross_val_under(data, clf, 3)
        log_entry_over ← cross_val_over(data, clf, 3)
        log_entry_smote ← cross_val_smote(data, clf, 3)
        log_original ← log_original.append(log_entry_original
    end for
    return [ mean(precisions), mean(recalls), mean(f1scores), mean(elapsedtimes) ]
end procedure
```

3.3.2 CNN Version 1

Version 1-1

In the first proposed approach for my CNN models, the network bridges from our baseline models by keeping single vector data as inputs. In other words, the input to the first CNN is still a 30 x 1 transaction vector. I then introduce convolutional aspects to the network and aim to construct a model appropriately based on the output of each layer.

Figure 3.2 shows the outline of the first CNNv1 network.

The input layer takes a transactional vector. Then, a convolutional layer performs convolutions on the input data, using 30x1 kernels, of which there are 256, differently initialised. This creates 256 1x1 outputs. By flattening these outputs, I then introduce a fully connected layer, which essentially means that every node is connected to every other node, between the flattened layer and the subsequent dense layer of which I used 300 nodes. Then there is a softmax output layer which takes the weighted sum of the fully connected layer and also has an activation, as with usual neural network architectures and gives us our predictions, corresponding to Fraud or Genuine.

Version 1-2

In an attempt to take this model a little further, I tried a variation whereby I added an extra convolutional layer and also an extra dense layer in the fully connected section. This was to give an indication of effect of additional layers would have.

The results of this version were also reported and compared with Version 1-1.

Listing 3.1 is a code snippet that shows the short function that creates CNNv1-1.

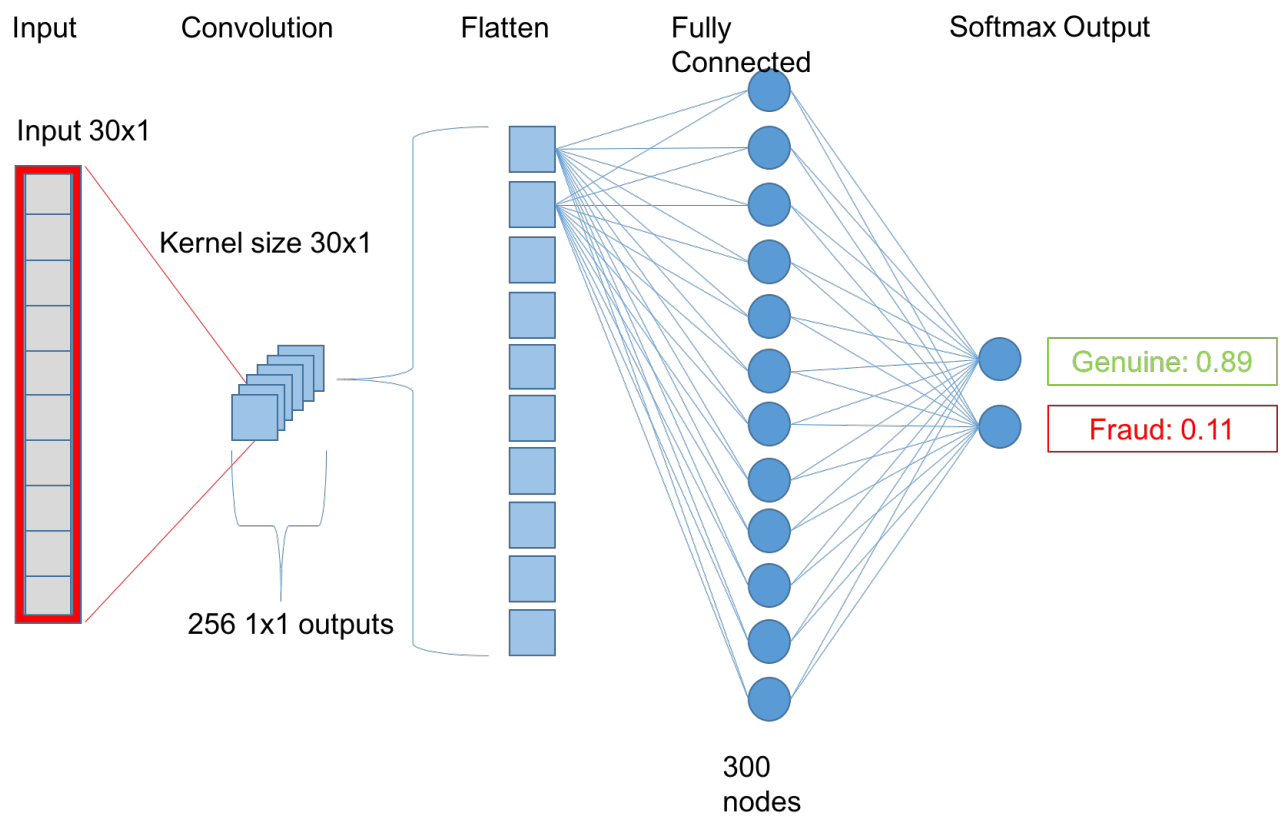


Figure 3.2: Overview of CNN Model 1.

```
# Function to create model
def create_model():
    # create model
    seed(2017)
    conv = Sequential()
    conv.add(Conv1D(256, 30, input_shape=(30, 1), activation='relu'))
    conv.add(Flatten())
    conv.add(Dense(300, activation = 'relu'))
    conv.add(Dense(2, activation = 'softmax'))

    sgd = SGD(lr = 0.1, momentum = 0.9, decay = 0, nesterov = False)

    # Compile model
    conv.compile(loss='categorical_crossentropy', optimizer=sgd)
    return conv
```

Listing 3.1: CNNv1-1 create_model function.

Listing 3.2 is a code snippet that shows the short function that creates CNNv1-2.

```
def create_model_2():
    # create model
    seed(2017)
    conv = Sequential()
    conv.add(Conv1D(256, 30, input_shape=(30, 1), activation='relu'))
    conv.add(Conv1D(256, 1, activation='relu'))
    conv.add(Flatten())

    conv.add(Dense(300, activation = 'relu'))
```

```
10      conv.add(Dense(100, activation = 'relu'))
11      conv.add(Dense(2, activation = 'softmax'))
12
13      sgd = SGD(lr = 0.1, momentum = 0.9, decay = 0, nesterov = False)
14
15      # Compile model
16      conv.compile(loss='categorical_crossentropy', optimizer=sgd)
17      return conv
```

Listing 3.2: CNNv1-2 create_model function.

ReLU Activations

On each hidden layer, I use the Rectified Linear Unit (ReLU). In modern deep learning applications, the use of the ReLU activation is very popular, prominently because it mitigates the vanishing gradient problem.

The ReLU function is defined mathematically as:

$$f(x) = x^+ = \max(0, x)$$

The Sigmoid Activation is defined as:

$$S(t) = \frac{1}{1 + e^{-t}}$$

The gradient of sigmoid is:

$$S'(t) = S(t)(1.0 - S(t))$$

Vanishing Gradient Problem During gradient based learning methods, a weight update is back-propagated through a network. The gradient of a sigmoid activation will be close to zero if the output is close to the tail ends of 0 or 1 (saturated neurons). This is a problem when there are multiple layers to the network as multiplication of near-zero quantities at each layer results in a 'vanishing gradient'. This means that even large changes, will attenuate through the network and result in affecting the output less. The hyperbolic tangent function activation also has this saturation problem.

The ReLU activation however, has a constant gradient. This means there is no attenuation in the back propagated signal and actually training is faster. The mathematical calculations are easier too, as the ReLU does not use exponentials. In the ImageNet[7] paper by Krizhevsky et al, they show that networks with ReLUs consistently learn several times faster than equivalents with saturating neurons (for example, with a tanh layer).

3.3.3 CNN Version 2

Overview

Version 2 aims to explore the idea of arranging the transactions temporally and trying to use the image metaphor, to unravel relationships in the data. In this setup, a bit more

preprocessing is needed in order to suit the network. The model itself takes an input batch of 100 vectors, so we have a 100 x 30 block. Then here our kernel (or filter) is a 5 x 30 block which slides down the temporally ordered block and convolves with the features, to give the output.

A window of height 5, can be positioned in 96 unique positions over the batch. Again, the model uses a number of different filters which are initialised. Figure 3.3 shows this process, using 32 different kernels. This gives an output of 96 (from the convolution) x 32 (the number of different kernels).

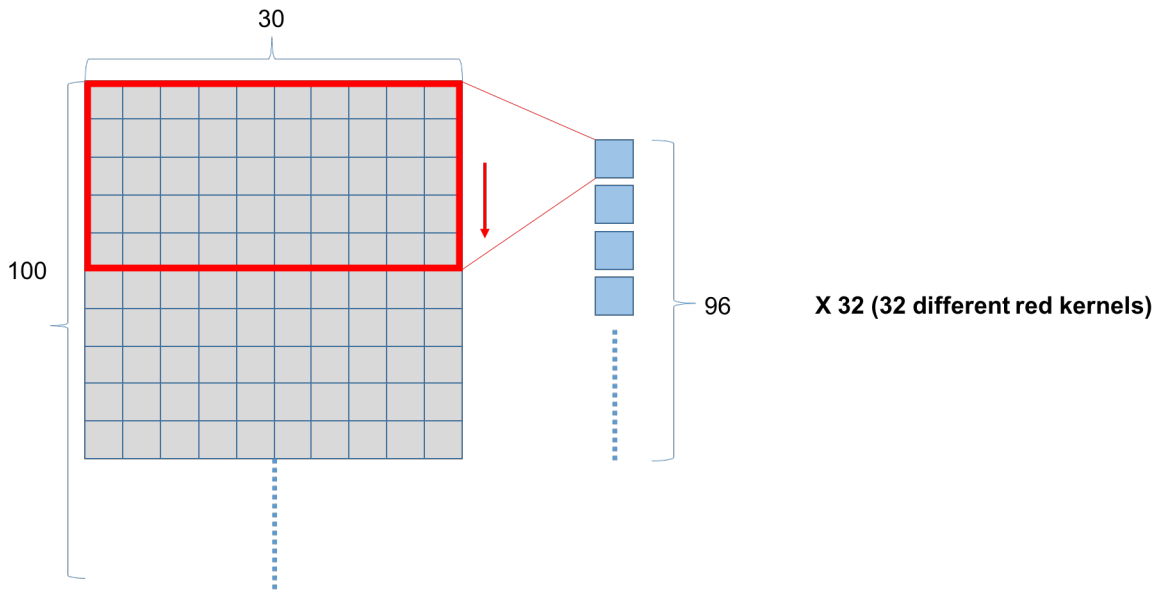


Figure 3.3: Overview of CNN Model 2 - Part 1.

Figure 3.4 then shows the remainder of the model. The output from the convolutions are flattened and hooked up to a fully connected layer. Then the softmax output layer, as before, to output our classification probabilities.

Causal Padding

The choice between the type of padding to have on the model, is an important one. By nature of time-series prediction, the prediction at time step t depends on the time steps previous to it. Therefore depending on the size of the kernel being used, it can be the case that the vectors at the beginning are not fully utilised. In order to work around this, causal padding is used. This causal padding approach is similar to section 2.1 on dilated causal convolutions in the WaveNet[4] paper, differing in application.

Effectively, this means that although theoretically the number in the output layer of the convolutions should be 96, as described, it is actually 100. This is due to padding the start of the batch to ensure that the first 4 transactions are handled appropriately.

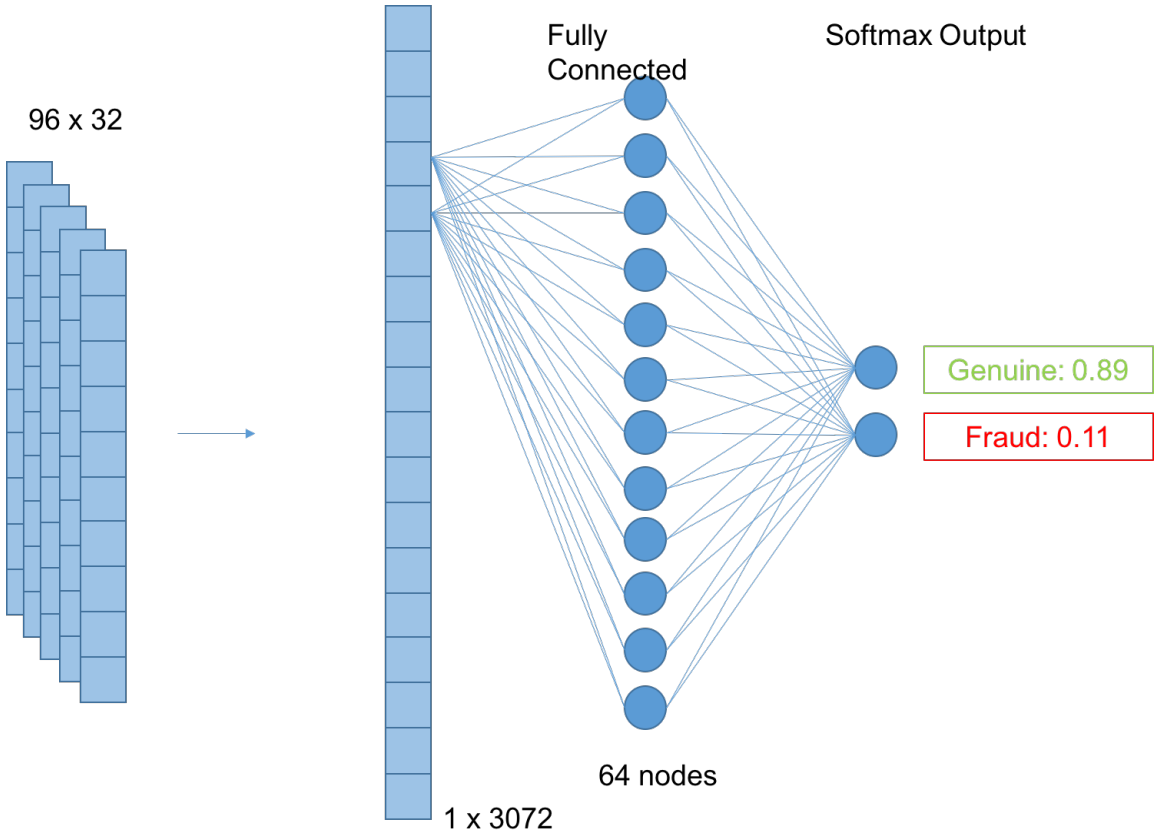


Figure 3.4: Overview of CNN Model 2 - Part 2.

Preprocessing

Due to the nature of this network, a lot more care has to go into preprocessing the data so that we can pass it through the model. One important thing is to batch together the dataset into chunks of size 100 (in this case), as this is the input of the model. So if the original size of the training data is (199365, 30), then we want to transform this to (1994, 100, 30). However, if the length of the training set is not a whole multiple of 100 then there will be a problem with the final batch. In order to fix this, the final batch is padded with zeros.

Listing 3.3 outlines a function that packs up this transformation into a function that can be reused for varying values of this hyper-parameter.

```
def reshape_to_batches(a, batch_size):
    # pad with zeros if the length is not divisible by the batch_size
    batch_num = np.ceil((float)(a.shape[0]) / batch_size)
    modulo = batch_num * batch_size - a.shape[0]
    if modulo != 0:
        pad = np.zeros((int(modulo), a.shape[1]))
        a = np.vstack((a, pad))
    return np.array(np.split(a, batch_num))
```

Listing 3.3: CNNv2 batch reshape function.

Time-Series Cross-Validation

Due to the fact that this CNN model specifically exploits the temporal ordering of the transactions, the usual method of cross-validating becomes insufficient. This is because during the usual KFold procedure, the data will be split randomly and of course this will violate temporal ordering. In general, cross-validating for predictive modelling like this is tricky, but there are a ways of doing it. I outline one here in my implementation.

The approach is to split the data as usual (perhaps a 70:30 split), but keeping the temporal ordering such that every transaction after the cutoff point (i.e in the testing set) has a transaction with time $T > T_n$, where T_n is the last transaction of the training set. When transactions are ordered by time, in the real world we often predict only the next one or two time steps (think of stock market fluctuations). We can use this therefore to average results across the size of the testing set, progressively predicting less test data.

This can be visualised as follows:

$$
\begin{aligned}
&[1][2][3][4] - - - [5][6][7], \quad && n = 1 \\
&[1][2][3][4][5] - - - [6][7], \quad && n = 2 \\
&[1][2][3][4][5][6] - - - [7], \quad && n = 3
\end{aligned}
$$

The implementation of this is through wrapping the model creation into it's own function, like previously, but also taking parameters to make the batch-size and window-height dynamic. Inside the cross-validation function itself, the preprocessing, model creation and model predictions are all wrapped up together. There is also another auxiliary function that generates the time-series split of the data by ordering it and splitting it at a specified ratio, given to the function as a parameter. Inside the cross-validation, this function is called with decreasing value, to return a progressively larger training set and smaller test set with every iteration.

3.4 Generative Adversarial Network Models

3.4.1 Overview

As described in the preparation chapter, GANs are essentially two networks that play a game with each other, each trying to reduce their own loss function.

The three major components of a GAN are the generator network, the discriminator network and the combined network training process. Below, I outline the steps needed in the adversarial training stage of the GAN, as well as various utilities that were important to the setup and running of the models.

Then this section outlines the three major variations in which I experimented; two variations on the generator network and then an attempt at incorporating a semi-supervised learning approach to treat the GAN as a classifier. This came about due to the fact that GANs in their common use case, display how the networks perform with regard to their loss functions and also can display the intermediate generations (like in the classic

MNIST dataset) to visually inspect how the GAN is performing. Seeing as the data I am working with, is inherently used in classification problems, it is of interest to experiment with using the GAN as so. However, simply investigating how the generator performs at constructing fraud-like data is also very interesting, this could have potential of helping the domain create more data on which to train with.

Below, listing 3.4 shows the abstract definition of the GAN class, in which I wrapped up all of the main functions of the model. This definition shows the prominent functions with short descriptions.

```
class GAN():

    def init_v17_v10_plot(self):
        '''Functions to initalise figure/plot variables for specific features,
        by setting up figure and plotting real fraud data on the left hand side for comparisons. '''
    def init_v17_v14_plot(self):

    def load_fraud_data(self):
        ''' Read in the dataset and do all of the necessary preprocessing (Feature scaling etc)
        and store in accessible class variables.'''

    def __init__(self):
        '''Calls any initalisation functions, builds generator and discriminator networks,
        compile the networks, set up combined model.'''

    def build_generator(self):
        '''Uses Keras functional API to create the generator network.'''

    def build_discriminator(self):
        '''Uses Keras functional API to create the discriminator network.
    '''

    def train(self, epochs, batch_size=128, save_interval=200):
        '''The adversarial training function for the GAN.'''

    def save_loss_plot(self):
        '''Utility to save the plot of losses for the GAN.'''

    def save_imgs(self, epoch, img, gen_imgs):
        '''Utility to save the plot of generated output for specified save intervals,
        during the training process.'''

    def test_as_classifier(self):
        '''For use in GAN-SSL. Wraps up all the evaluation procedures for predictions and retrieving results metrics.'''

if __name__ == '__main__':
    gan = GAN()
    gan.train(epochs=6000, batch_size=32, save_interval=1000)
    gan.test_as_classifier()
```

Listing 3.4: GAN Class Definition.

3.4.2 Adversarial Training

The training procedure of the GAN is a crucial component. Unlike other models in Keras, a GAN is not nicely wrapped up in one library function and so we have to connect up networks ourselves and implement the training process from scratch. This is what I did in the GAN class. The training process must do a number of things, which I outline in a high-level pseudo-code algorithm described by Algorithm 3 below.

Algorithm 3 Adversarial Training

```
1: procedure GAN-TRAIN(self, epochs, batch_size = 128, save_interval = 200)
2:     Get X and Y train data
3:     Reshape training data into tensors
4:
5:     for epoch in epochs do
6:                                                  ▷ Discriminator Training
7:         Take a random half batch of training data
8:         Generate, using the generator, a half batch of fake data
9:
10:        Set discriminator.trainable = True
11:
12:        Create half batch number of labels for the real and fake data
13:        Format labels into categorical shape
14:        Train discriminator separately on real and fake data
15:        Set discriminator.trainable = False
16:
17:                                                  ▷ Generator Training
18:        Create random noise
19:        Create associated positive labels for the random noise
20:        Train the Combined model using this noise and labels
21:
22:        Save and Append losses of both networks
23:                                                  ▷ Save Intervals
24:        if epoch mod save_interval == 0 then
25:            Save current generated output
26:        end if
27:    end for
28: end procedure
```

The Adversarial Training algorithm shown is a very high level outline of the procedure. Some lines in particular contain a lot under the hood:

Line 3: Reshape training data into tensors. This line takes the training data and expands the dimensions in order to satisfy the 3D tensor requirement of the models. This is due to TensorFlow being the underlying engine.

Line 13: Format labels into categorical shape. Because I used categorical cross-entropy loss in the combined GAN model, the labels of the data needs to be in a binary matrix representation. Targets are in categorical format (e.g. if you have 2 classes, the

target for each sample should be a 2-dimensional vector that is all-zeros except for a 1 at the index corresponding to the class of the sample).

Line 20: Train the Combined model using this noise and labels. As outlined in the subsections below, in order to train the generator, we need the discriminator too and thus this is a combined model. The model must be carefully defined so that the loss of the combined model incorporates the loss of the generator. Essentially the networks are stacked together. This is also why in **Line 15**, we set the discriminator to untrainable (i.e frozen) so that the current state of the network can be used when training the generator in order to allow it to improve.

Line 22: Save and Append losses of both networks. This line covers the work of saving the loss of each network with every epoch, in order to do things like plotting the loss curves for visual evaluation.

Line 25: Save current generated output. When the save interval is reached, a utility function is called that I implemented to do a number of things. One of which was to use the current state of the generator to take some random noise and generate an output. This is so that we can visualise what the current output looks like at this epoch. This function also includes the various plotting utilities that builds up the final figure which shows generated output for regular training intervals.

Notice also, how the discriminator is trained on real and fake data separately, to give a stable and clear separation of the data.

3.4.3 GAN Version 1 - Dense Generator

In the first GAN version I focus on creating the generator network from scratch, instead of importing one of the previous CNN models. The dense generator network takes in random noise as input, has a series of hidden dense layers and includes regularisation layers such as Leaky ReLU activations and Batch Normalisation.

An outline of the dense generator network can be viewed as:

```
1  def build_generator(self):
2
3          noise_shape = (100,)
4
5          model = Sequential()
6
7          model.add(Dense(256, input_shape=noise_shape))
8          model.add(LeakyReLU(alpha=0.2))
9          model.add(BatchNormalization(momentum=0.8))
10         model.add(Dense(512))
11         model.add(LeakyReLU(alpha=0.2))
12         model.add(BatchNormalization(momentum=0.8))
13         model.add(Dense(1024))
14         model.add(LeakyReLU(alpha=0.2))
15         model.add(BatchNormalization(momentum=0.8))
16         model.add(Dense(np.prod(self.img_shape))) # , activation='tanh'
17         model.add(Reshape(self.img_shape))
18
19         model.summary()
20
```

```
21          noise = Input(shape=noise_shape)
22          img = model(noise)
23
24          return Model(noise, img)
```

Listing 3.5: GAN v1 - Dense Generator.

Leaky ReLUs

I use leaky ReLU to allow gradients to flow backwards through the layer unimpeded. TensorFlow does not provide an operation for leaky ReLUs, However the Keras API has a nice wrapper for adding this as a layer. The alpha parameter is essentially how 'leaky' we want the negative gradient to be and in a lot of cases we just want this to be slight, to avoid it being completely zero and so a value of 0.2 is employed here. This is something that is quite common amongst GAN papers, in particular in DCGAN(deep convolutional GAN)[8]. One of the main problems Leaky ReLUs solve is the dying ReLU problem, whereby a large gradient flowing through a ReLU node could cause the weights to update in such a way that the neuron will never activate again. It is common on large scale networks that a large proportion is 'dead'. I care about this here in the GAN, as the network is very dense in terms of the amount of epochs we train on the data and so I want to mitigate this problem.

Batch Normalisation layers aim to increase stability of the networks (which in GANs is something we really care about) by normalising node outputs, just like we do to data in the original preprocessing stages.

The Adam Optimiser

The Adam optimiser is one which is often seen as the default to use in modern deep learning applications, due to empirical evidence reported in the original paper[9] whereby it is stated "Using large models and datasets, we demonstrate Adam can efficiently solve practical deep learning problems" and showing its performance over that of other optimiser functions.

To this end, I used the Adam optimiser as the primary algorithm in my models. Adam is not like classical stochastic gradient descent whereby a single, constant learning rate is maintained across all weights, but instead the method computes individual adaptive learning rates for different parameters.

By altering the learning rate in the Adam optimiser, I was able to see better convergence in my GAN loss graphs.

Training Variations

Given that this is very experimental, I also tried variations of the models in order to see what effect they had on the loss curves and generated outputs. The idea being that these may combat the main problem of one network overpowering the other. I outline some of the variations I tried below.

- Standard adversarial training: with each epoch, the discriminator and generator are training independantly.
- Pre-training the discriminator for 500 epochs and then letting the generator learn for the remainder.
- Alternating between training the discriminator for 500 epochs and then the generator for 500 epochs.
- Using SMOTE on the data and using standard adversarial training.

3.4.4 GAN Version 2 - Previous CNN Generator

The second GAN model was set up with the aim that the generator network would be convolutional and also, be the same as one of our previous CNN experiments. I used the model from CNNv1. This meant that I had to add some functionality to save the model to disk and load the model into the GAN program, conserving the weights. An extra layer of complexity is that I needed to be able to adapt the layers, the output layer needs to be changed to one which is of the shape we want. To do this, I popped the last layer of the model and constructed a dense and reshape layer to model the generated shape we need (which in this case is a 1 by 30 vector, but in 3D tensor form). Then, using the Keras functional API to wrap layers around each other, I constructed the generator by hooking up the input of the loaded CNN model, with the adapted layers.

3.4.5 Semi-Supervised Learning for Classification

Semi-Supervised learning[10] is an approach in which the output of the discriminator network is altered such that the network splits the real probability into N classes and thus giving us means of classifying with the model.

A difficulty that lies with trying to use the GAN as a classifier, is in the decision of what to base the generative model on. In the previous models, I was experimenting mainly on how well we can learn and generate fraudulent data and with some parameter tweaking, a stable GAN could be achieved with interesting results. However, if we want to classify between Fraud and Genuine as well as Fake and Generated, then the network needs to be trained on the genuine examples too.

Some problems that can occur here are fairly logical. If we only train to generate fraud then the network might learn that all fraud-like data is simply fake. On the other hand, if we train the generator to generate a transaction in general (i.e both fraud and genuine) then the imbalance in the dataset may have a hard time to pick up fraud. I experimented with a few variations to see how classification results compare.
These variations were: **generating fraud only, providing the discriminator with half fraud and half genuine examples from the real data and just trying to generate transactions in general from providing the discriminator with original dataset examples**.

One important key factor with GANs is the number of epochs. Letting the training process go on for a longer time can greatly affect the results obtained. This was very much part

of the experimentation process. In the case of the GAN variations, more training epochs proved to decrease the number of misclassified benign transactions, significantly.

Figure 3.5 below shows a diagram that outlines the change that the semi-supervised approach has on the output of the discriminator network. Effectively this is a form of multi-task learning; one output is a sigmoid layer that gives us the probability of an example being fake or generated and the other is a softmax output which gives us probabilities of class labels, as discussed. The dynamic of the GAN changes here, the discriminator becoming the predominant network that learns to classify.

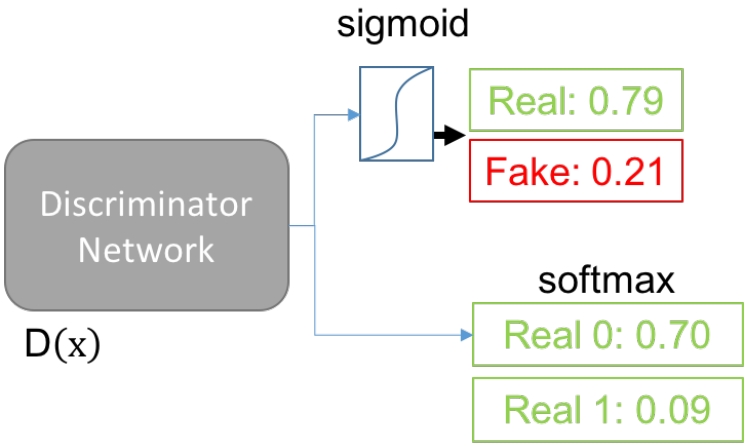


Figure 3.5: Overview of GAN-SSL discriminator output.

Problems with GANs

Mode Collapse: One problem when training the GAN for a large number of epochs, is a problem called Mode Collapse. This can be visualised in the generated output when the data distributed 'collapses' and suddenly becomes a lot more refined. This is the network learning a certain range and shape of data particularly well and disregarding the rest. In attempt to mitigate this problem, one can try adjusting the number of epochs for which you let the network run, to try and best capture the data distribution.

Chapter 4

Evaluation

4.1 Evaluation Methodology

4.1.1 Metrics

In this project, the metrics that were of interest were not just simply the accuracy, as with many machine learning applications. Accuracy simply measures how many correct classifications are made. This, in the context of credit card fraud, is useless. This is because, even if we classified all fraud as benign, the classifier would still be over 99% accurate, due to the high imbalance of the fraud/non-fraud classes. In the dataset being used, fraud accounts for only 0.17% of the total.

Precision and Recall

Therefore we need to look at metrics that tell us more about what we're interested in. Two main measures of interest are *Precision* and *Recall*.

$$Precision = \frac{T_p}{T_p + F_p} \quad , \quad Recall = \frac{T_p}{T_p + F_n}$$

$$T_p = TruePositive \quad , \quad F_n = FalseNegative$$

Precision is intuitively the ability of the classifier to not misclassify. Recall is intuitively how well to catch fraudulent examples.

Of course the threshold by which we allow benign transactions to be misclassified as fraud, is a business decision. In the context of a bank, of course the number of fraudulent transactions caught is very important but at the same time we care about precision as this amounts to freezing customers accounts, sending text messages even though they are not subject to fraud, which ultimately costs the bank time and money and potentially customers.

F1-Score

F1-score is the harmonic mean of both precision and recall. In general, F-measure can be defined as:

$$F = \frac{1}{\alpha\frac{1}{P} + (1-\alpha)\frac{1}{R}} = \frac{(\beta^2+1)\,PR}{\beta^2 P + R} \quad \text{where} \quad \beta^2 = \frac{1-\alpha}{\alpha}$$

This is a weighted harmonic mean and allows us to use the parameters to pay more attention to precision or recall. The most common case, however, is the balanced F_1 score which uses $\beta = 1$ or $\alpha = 0.5$. Thus giving:

$$F_1 = \frac{2PR}{P+R}$$

The reason why we us this harmonic approach, rather than a standard average is so that extreme cases are penalised more. Of course, reported values of F1-Scores are actually just snapshots of single precision and single recall scores, AUC captures a summary of these as described below.

Confusion Matrix

A confusion matrix summarises the number of examples that were classified into either:

1. **TP - True Positive - Predict fraud and it is fraud.**
2. **TN - True Negative - Predict benign and it is benign.**
3. **FP - False Positive - Predict fraud but it is actually benign.**
4. **FN - False Negative - Predict benign but it is actually fraud.**

These statistics are shown in a matrix representation, that can be printed out using plotting utilities, to easily inspect how a classifier has performed. Figure 4.1 below shows an example confusion matrix plot, for a single run of a logistic regression classifier.

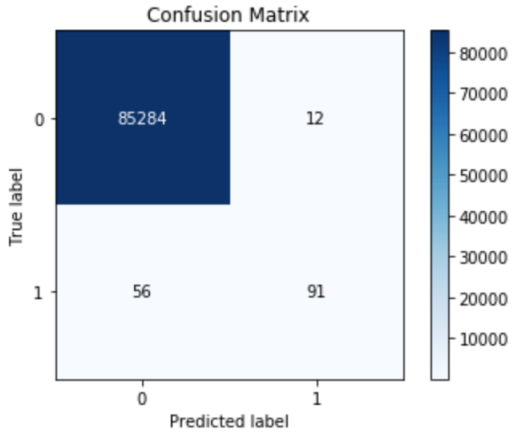


Figure 4.1: Example Confusion Matrix plot.

From these, precision, recall and F measure scores can therefore be calculated and a classification report summary can be shown.

4.1.2 Visual Inspections

Precision-Recall Curves and ROC
When classifying, the true positive rate (TPR) and false positive rate (FPR) changes for increasing values of precision and recall. This can be visualised by looking at TPR-FPR curves. This gives an indication of how the trade-off varies for differing values of TPR and FPR.

Taking the area under this curve is known as AUC, therefore gives an overall statistic for comparing classifiers. In the results that follow, I simply use a 0.5 probability threshold for classification. The ROC-AUC scores are useful in shedding light on the effectiveness of the deep learning models as this metric will summarise the varying thresholds.

GAN loss function graphs and generated data visualising
With GANs, the way in which we can visualise performance is different to others due to it's specific adversarial nature. Two prominent use cases for GANs is to view the loss function graph for the generator and discriminator networks to see how the two networks fight with each other. In an ideal, theoretical world, we would expect to see the two loss curves converging to some stable value, which indicates the two networks can no longer improve or better themselves.

Also, seeing as a large part of the GAN architecture is generating data, we can inspect this to see if it looks appropriate. A common example with the MNIST dataset is to see if the generated images look like handwritten digits. In this case, we can inspect the shape and distribution of the generated fraud transactions and view them side-by-side with real fraud transactions to see if any similarities have been picked up by the network. To do this, I selected two of the most prominent features from the data in which fraud is very clearly disparate from that of the genuine examples. This can easily be observed by plotting the features on histogram plots to see the variations. I did so and I chose to inspect a plot of V17-V10 (features 17 and 10).

4.2 Results Overview

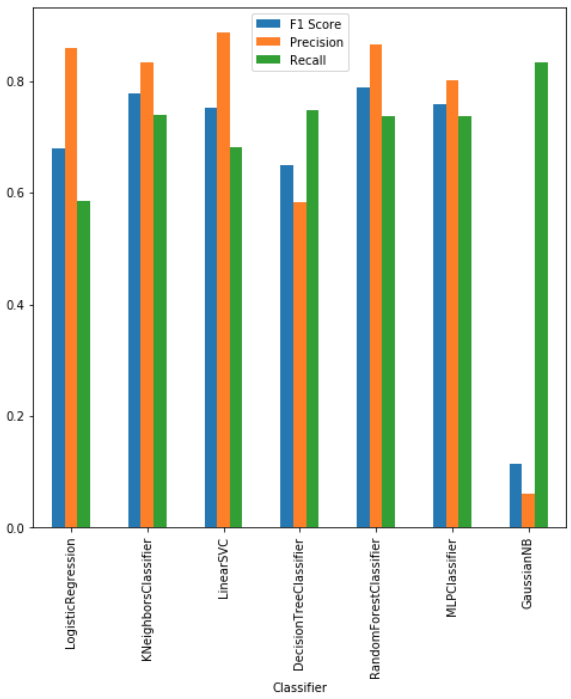
4.2.1 Baseline Models

Results Table

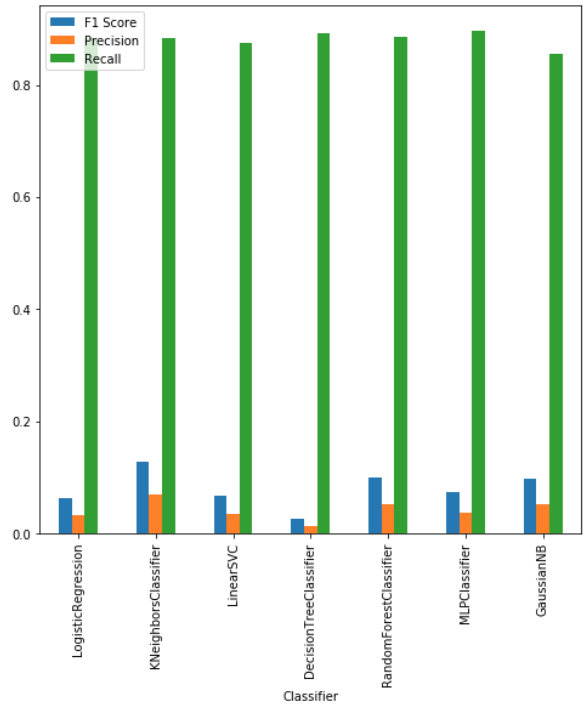
Table 4.1 below demonstrates the results for the baseline classifiers, for every type of re-sampling method. These results came from the cross-validation described in the previous two chapters. Figure 4.2 visually depicts the performance of the classifiers across the resampling methods. All results reported were based upon a default class classification threshold of 0.5. AUC-ROC scores are given to provide an indication of potential performance across other threshold values, however this is more prominent in the deep learning work of the project.

Classifier	F1	Precision	Recall	AUC-ROC
Original Dataset				
LogisticRegression	0.679526	0.858680	0.585366	0.792588
KNeighborsClassifier	0.777306	0.834855	0.739837	0.869780
LinearSVC	0.752517	0.887192	0.680894	0.840350
DecisionTreeClassifier	0.648748	0.584061	0.747967	0.873483
RandomForestClassifier	**0.789572**	0.867006	0.737805	0.868790
MLPClassifier	0.759672	0.801885	0.737805	0.868720
GaussianNB	0.114077	0.061255	0.833333	0.905594
Under-Sampled Dataset				
LogisticRegression	0.062811	0.032562	0.884146	0.919338
KNeighborsClassifier	0.128275	0.069294	0.884146	0.931555
LinearSVC	0.068514	0.035956	0.876016	0.912220
DecisionTreeClassifier	0.026292	0.013374	0.892276	0.882144
RandomForestClassifier	0.100256	0.053308	0.886179	0.928646
MLPClassifier	0.073166	0.038147	0.896341	0.928560
GaussianNB	0.099017	0.052607	0.855691	0.914186
Over-Sampled Dataset				
LogisticRegression	0.111965	0.059809	0.892276	0.933744
KNeighborsClassifier	0.625977	0.553314	0.786585	0.892538
LinearSVC	0.118897	0.063827	0.871951	0.924837
DecisionTreeClassifier	0.649036	0.661330	0.664634	0.831985
RandomForestClassifier	0.801992	0.884864	0.745935	0.872869
MLPClassifier	0.673009	0.631834	0.752033	0.875552
GaussianNB	0.100591	0.053461	0.855691	0.914668
SMOTE-Sampled Dataset				
LogisticRegression	0.107417	0.057296	0.873984	0.924311
KNeighborsClassifier	0.481362	0.358277	0.829268	0.913002
LinearSVC	0.118380	0.063560	0.865854	0.921825
DecisionTreeClassifier	0.459940	0.336584	0.747967	0.872652
RandomForestClassifier	**0.805452**	0.844639	0.782520	0.891119
MLPClassifier	0.678626	0.649623	0.731707	0.865467
GaussianNB	0.107921	0.057627	0.855691	0.915657

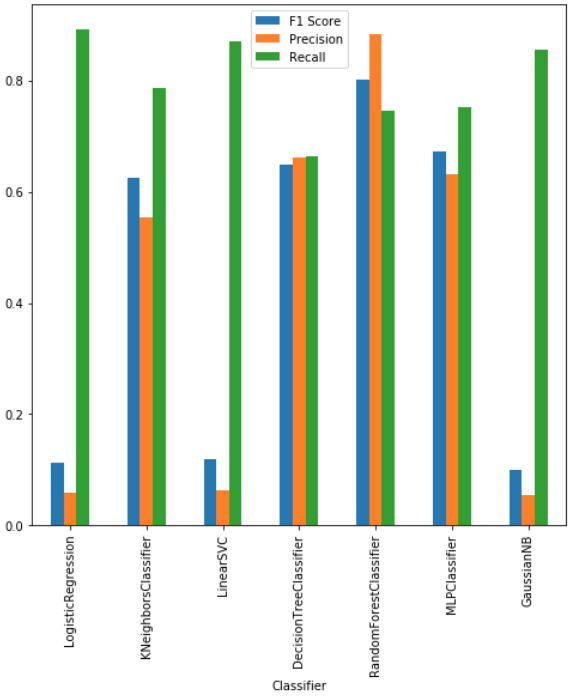
Table 4.1: Baseline Models Results.



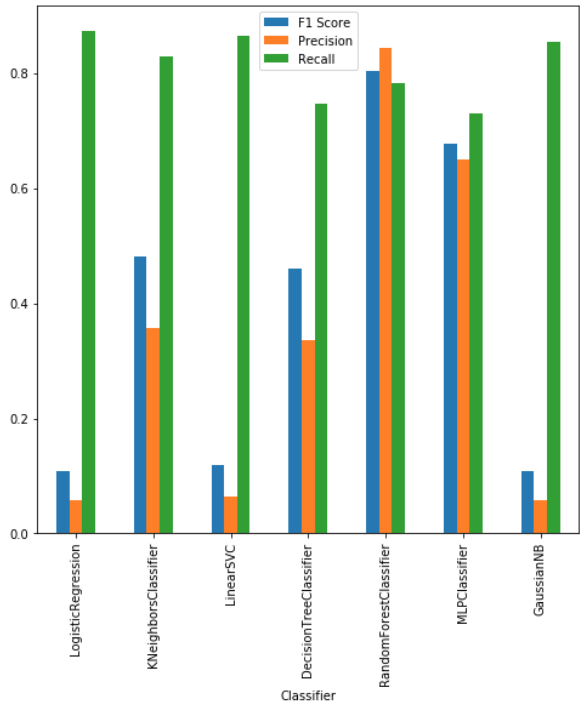
(a) Original



(b) Undersampled



(c) Oversampled



(d) SMOTE

Figure 4.2: Baseline Models Results - Bar Charts.

Key Observations

1. **Original dataset results**: The results on the original dataset are fairly standard across the board, with certain classifiers achieving better F1 than others. Random Forest reports the best F1 score.

2. **Under-Sampled dataset results**: When the data has been under-sampled, that is, a very large proportion of data has been taken away to balance the classes, we see that precision is dramatically affected. Hence, so are the F1 scores. This observation is perhaps somewhat intuitive given that if we think that we are removing information from the system, the model has less to learn on and hence more misclassifications of the genuine class will occur.

3. **Over-Sampled dataset results**: Comparing this set of results with the original dataset, an observation can be made that on the whole, recall scores increase however precision scores suffer. Considering that this form of resampling is duplicating fraud data points, it is understandable why recall would increase. Here it is also observed that tree based classifiers remain high performers.

4. **SMOTE dataset results**: Similarly the tree based classifiers perform better than the others. Some of the classifiers performed worse with SMOTE, which is perhaps due to how the model has learned differently from duplicated data vs different synthetic data. However, it is interesting to note that Random Forest and MLP remain consistent, which is a good sign as theoretically we would prefer to use SMOTE given that the data points are different.

Further Tuning the RandomForest

Seeing as Random Forest was clearly a classifier that performed well across the board with this type of application/data, I took some of it's most prominent parameters and performed a grid search, to give an indication of what could be achieved from further tuning. The tuning was done on the SMOTE data, this was because it gave interesting improvements on recall on the dataset, compared with that of the original dataset. This is interesting because for other classifiers, performing something like smote dramatically kills precision.

The parameters I searched across were:

1. **n_estimators={10, 100, 200}**
2. **criterion={'gini', 'entropy'}**
3. **max_features={'auto', 'log2'}**

The original F1, precision and recall respectively were: **0.805452 0.844639 0.782520**.
The best parameters, found by searching were:
criterion='entropy', max_features='auto', n_estimators=200.

Table 4.2 below shows the results of this tuning.

	F1	Precision	Recall
Before Tuning	0.805452	0.844639	0.782520
After Tuning	0.827025	0.853782	0.813008

Table 4.2: Random Forest Parameter Tuning Results.

4.2.2 CNN Models

CNN Version 1 Results

Below, I summarise the results of the version 1 models. Table 4.3 shows the results based on using the original dataset and Table 4.4 shows those using SMOTE as a dataset resampling technique inside the cross-validation.

	F1	Precision	Recall	AUC-ROC
CNN Model 1	0.809863	0.881838	0.760163	0.879979
CNN Model 1.2	0.821249	0.881579	0.774390	0.887097

Table 4.3: CNN Model 1 Results for Original Dataset.

	F1	Precision	Recall	AUC-ROC
CNN Model 1	0.645623	0.586726	0.804878	0.901706
CNN Model 1.2	0.596756	0.552336	0.815041	0.906084

Table 4.4: CNN Model 1 Results for SMOTE Dataset.

Key observations from these results are that the original dataset performs better, without any resampling involved and that the learning rate parameter of the Adam optimiser plays a key role in the model tuning. In the case of the original dataset results, a learning rate that is a factor of 10 lower than the default value ($lr = 0.00001$) leads to better scores. In the case of the SMOTE data, a higher learning rate was better. The auc scores suggest that with different probabilistic classification thresholds, the smote data could potentially achieve better f1 scores.

CNN Version 2 Results

Below, I report the best results for the version 2 model. Table 4.5 shows the results for the default case of the model. This default case uses 100 as the batch-size and 5 for the window-height. These correspond to how many transactions are taken and temporally ordered and the height of the sliding window, respectively.

	F1	Precision	Recall	AUC-ROC
CNN Model 2	0.674887	0.934534	0.530247	0.952731

Table 4.5: CNN Model 2 Results for batch-size = 100, window-height = 5.

Having a recall of ≈ 0.53 is not particularly great for the problem at hand, despite the impressive precision of the model. Therefore I do a search across some varying parameter values, to give an indication of the improvements to the results that can be made. Table 4.6 reports the results of tuning these two parameters.

(batch-size , window-height)	F1	Precision	Recall	AUC-ROC
(100 , 5)	0.674887	0.934534	0.530247	0.952731
(200 , 5)	0.683986	0.966952	0.531695	0.938287
(500 , 5)	0.607998	1.0	0.439607	0.924830
(100 , 10)	0.734242	0.933769	0.608294	0.930754
(200 , 10)	0.715631	0.934084	0.586319	0.911789
(500 , 10)	0.62118	0.962963	0.459854	0.902490

Table 4.6: CNN Model 2 Results for Varying (batch-size , window-height) Values.

These results indicate that perhaps a smaller batch size but higher window height performs the best. One trend to note, however, are the very high auc scores. This suggests that the models could achieve better f1 scores if allowing the classification to be done on a threshold different to 0.5. This is promising and means there is potential to find a better tradeoff of precision and recall, which better suits the business need.

4.2.3 GAN Models

In this section I report findings from the GAN approaches in application to the credit card fraud data. These results were all carried out on the original dataset . It is clear that this is a very experimental use-case with the low reported precision scores seen below, but the effects of network variations are still interesting nonetheless.

GAN version 1 and its variations

In these versions, only the fraud data is being used, as the aim is to see how the GAN learns to model fraudulent transactions and to see what the loss curves look like. Below I describe the results from the variations on the adversarial training.

Version 1.1

In this version, standard adversarial training is applied. Both the generator and discriminator are trained separately in each epoch.

Figure 4.3 below shows the training losses for the model. It can be observed that between 500 and 1500 epochs, the generator improves and learns to fool the discriminator, but

after this point the generator's loss slowly diverges as clearly the discriminator overpowers it, after eventually learning the data.

Figure 4.3: GANv1 Losses Graph.

As an example of what generated output looks like and what was used to visualise fraud generation, Figure 4.4 below shows the plot that was generated over 6 regular save intervals during training. In this particular case you can see the actual fraud data on the left and the generated fraud on the right, for every 1000 epochs. It can be observed that the model is slowly learning the shape and distribution of the data, albeit not perfectly.

For the sake of brevity, I will omit future generated output plots for the following versions, but they can be seen in the Appendix.

Version 1.2

In version 1.2, I adapt the adversarial training to a setup whereby the discriminator is pre-trained alone for 500 epochs and then the generator is left to train for the rest. Figure 4.5 shows the associated losses graph for this.

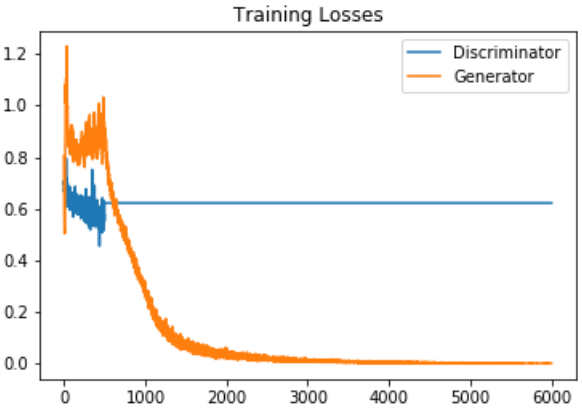


Figure 4.5: GANv1.2 Losses Graph.

The resulting graph for this variation is intuitive. After the discriminator is stopped training, the generator simply continues to learn to fool the now static discriminator. You

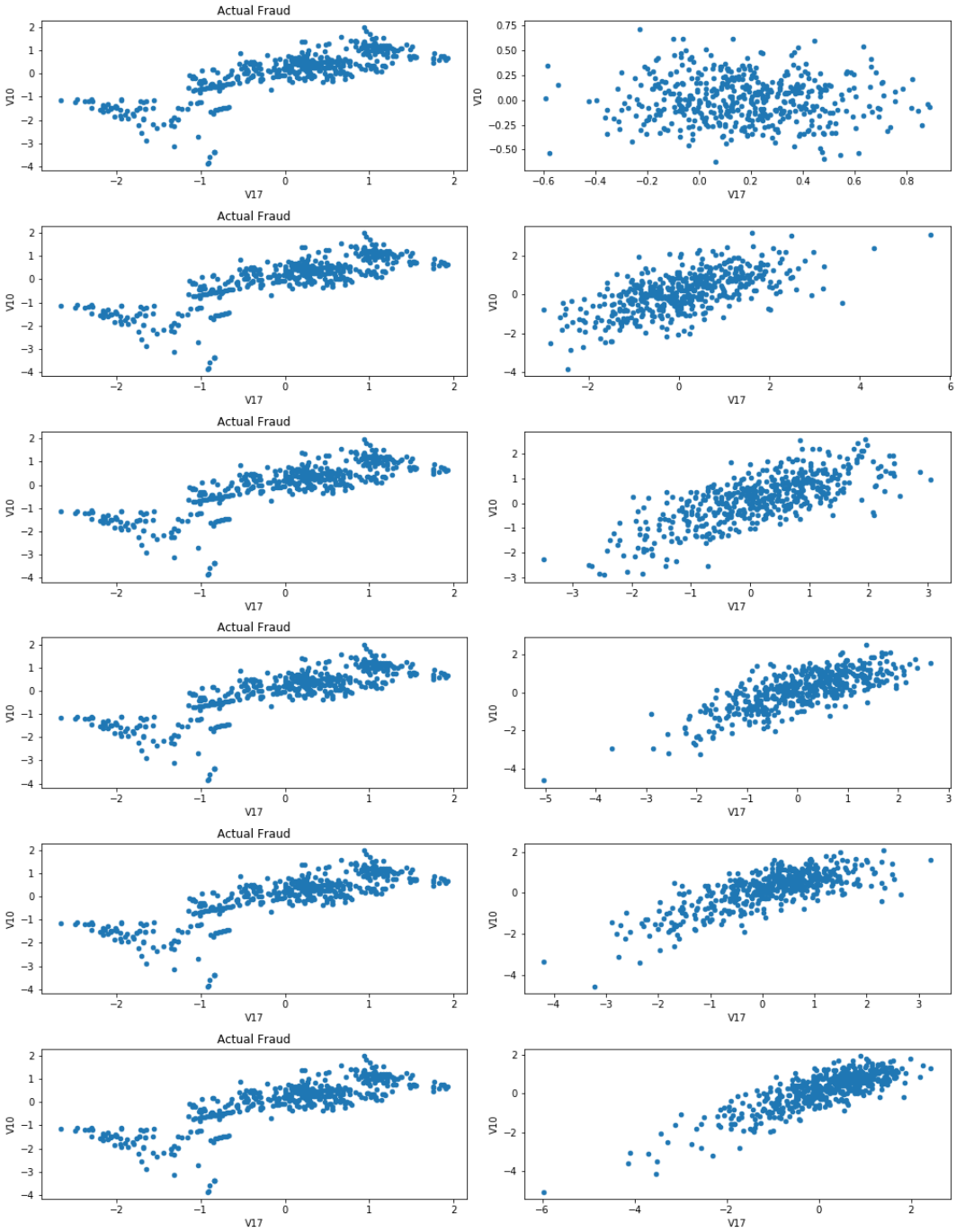


Figure 4.4: GANv1 Generated Output Graph.

can observe in the graph that the orange loss curve tends to zero very quickly afterwards. Figure A.1 in the appendix shows that the generator concentrates a small subset that are able to fool the discriminator.

Version 1.3

In version 1.3, the adversarial training was staggered between the two networks. Each network training separately for 500 epochs at a time. Figure fig:ganv1.3-losses shows how the losses fluctuated in this scenario. Figure A.2 in the appendix shows that the generator concentrates a small subset that are able to fool the discriminator.

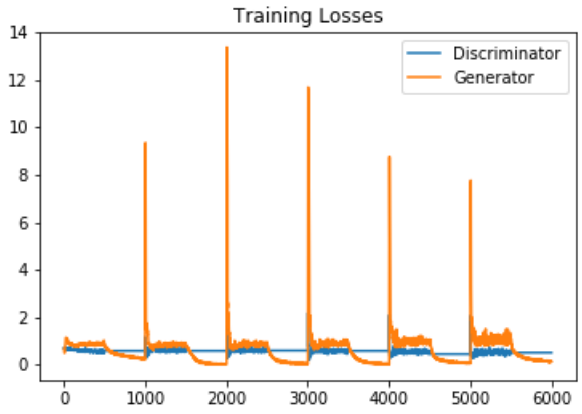


Figure 4.6: GANv1.3 Losses Graph.

Version 1.4

In version 1.4, I used SMOTE data with standard adversarial training. Figure fig:ganv1.4-losses shows how the losses fluctuated in this scenario. It is clear that Figure A.3 in the appendix shows how even with the increased size of the output data, the generator still focuses on a subset of the actual data distribution.

Figure 4.7: GANv1.4 Losses Graph.

GAN version 2

On the whole, the results for the second GAN model (using a previous CNN as a generator) showed relatively similar performance to the prior model, exhibiting similar loss and output graphs. It seems to be a common case with this data, that eventually the discriminator simply overpowers the generator in a slow diverging manner. Despite loading weights from the CNN model, the network continues quickly to expose any data distribution that it learns very quickly and after this point the loss just deteriorates as with previous models.

Semi Supervised Results

In this section I report findings from the semi-supervised approach to the GAN in application to the credit card fraud data. I report two main sets of results, one for a smaller amount of epochs and one for a larger amount, to give an indication of the differences in results and also because during experimentation these were key points. Table 4.7 below shows the reported scores for 600 epochs and 6000 epochs respectively.

Variation 1 - Using the original dataset in training (i.e the generator attempts to generate both fraud and non-fraud).

	F1	Precision	Recall	AUC-ROC
GANv1-SSL (600 epochs)	0.00426	0.00213	0.93893	0.564891
GANv1-SSL (6000 epochs)	0.01221	0.00615	0.83761	0.807437

Table 4.7: GANv1-SSL Results for Original Dataset.

It is clear from these results that the precision is very poor, despite rather good recall scores. An interesting trend to note from these is that increasing the amount of epochs that the network trains, decreases the recall but improves the precision. Although this improvement seems insignificant, this is just due to the amount of genuine examples compared with that of the true positives (the fraud). Figure 4.8 shows the confusion matrix for the 600 epochs run and it is observed that 57507 genuine transactions were misclassified as fraud. That is 81% of all genuine transactions. In contrast, Figure 4.9 shows the confusion matrix for the 6000 epochs run and the false negative count has decreased to 22%. In raw numbers, that is over 40,000 transactions that have been now correctly classified. So clearly, the increased epochs performs better and allows the network to learn better which transactions are genuine.

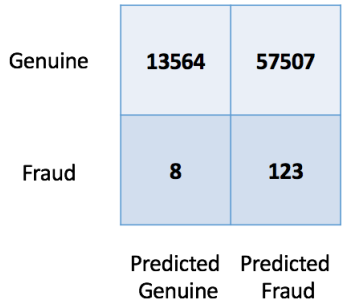


Figure 4.8: GANv1-SSL Confusion Matrix for 600 epochs.

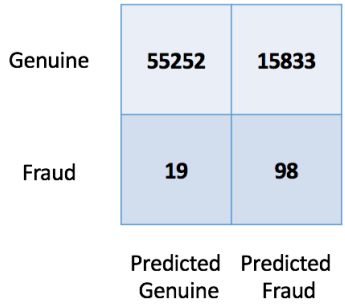


Figure 4.9: GANv1-SSL Confusion Matrix for 6000 epochs.

Based on this observation it was fitting to try running with a number of epochs a factor of 10 above, to see if this improvement continues.

	F1	Precision	Recall	AUC-ROC
GANv1-SSL (30000 epochs)	0.01365	0.00688	0.83871	0.813788
GANv1-SSL (60000 epochs)	0.01410	0.00712	0.68293	0.759097

Table 4.8: GANv1-SSL increased epochs.

Variation 2 - Similar to version 1, but ensuring the discriminator takes a 50:50 ratio of fraud and genuine transacations, from the real portion of the input.

Here, when the discriminator takes a half batch of real and generated input, I ensure that the real input has an equal amount of fraud and genuine data. The results for this are as follows:

	F1	Precision	Recall	AUC-ROC
GANv1-SSL (6000 epochs)	0.00827	0.00415	0.98496	0.771414
GANv1-SSL (30000 epochs)	0.71212	0.63514	0.81034	0.904792
GANv1-SSL (60000 epochs)	0.00993	0.00499	0.85484	0.778857

Table 4.9: GANv1-SSL Variation 2 Results.

Table 4.9 shows that there is a sweet spot at around 30000 epochs in which the classification portion of the GAN performs reasonably well in comparison to some of the results seen previously in the project. It also indicates how performance can fluctuate.

Variation 3 - Fraud data only, but classifying on both genuine and fraud. In this case, the network was not useful at all. Due to the fact that the network learns the shape of fraudulent data only, it simply misclassified all benign transactions even for high epoch numbers.

Chapter 5

Conclusions

The aims of the project are to experiment whether the types of deep learning models described, are feasible and useful, in application to credit card fraud detection. I successfully show that these models can be applied, in their variations, to this problem space. The question of how useful they are, then comes down to certain trade-offs, described henceforth.

The results of the CNN models are promising, especially those of CNN Model 2, with the sliding window variations. The ROC-AUC score for this model was very high at 0.95, which suggests that for classification thresholds other than 0.5, this model could potentially report higher precision/recall scores. This suggests that using convolutional based networks definitely has a place in the time-series and credit card fraud domain. This stems from the experimentation in this project but also from previously seen success, as mentioned with WaveNet and other papers that have recently started to use convolutional techniques on time-series data. With deep learning architectures like this, there are many variables and methods that can be undertaken to further tune and improve the performance of the models. In this project, I scratched the surface of these but with further experimentation, I suspect we could achieve even more comparable results.

In the GAN experiments, I show from visual inspection of the loss graph and generated outputs, that it is possible to train a model to learn what fraudulent transactions look like. A tradeoff is that the GAN falls victim of mode-collapse, learning only a specific subset particularly well. Also, the discriminator overpowers the generator after increased training epochs. The semi-supervised extension provides indication that there is the possibility of effective performance when using the network as a classification model. A result of interest is a ROC-AUC score of 0.90 at 30000 epochs, for the second variation. This seemed to be a sweet-spot in the training process and hence is suggestive that with more manual labour of finding the right conditions, a comparatively useful classifier could be achieved. However, other than this specific result, the overall performance was substandard in comparison to the previous models in the project. This may be an indication that GANs would be limited when actually applying this to the real-world problem of credit card fraud detection. There is a fixed amount of information in a dataset and a theoretically optimal classifier would use all of that information. The GAN trained on the dataset does not actually create new information, it just tries to learn best the distribution of the original data. This is advantageous in that the model handles noise in the data well and homes in on a certain distribution of data. To this end, GANs may

indeed be useful for data-augmentation purposes.

For fraud detection in general, we have to also consider the GAN in comparison to what we can already achieve with other models used in the project. Based on reported results and experimentation it would appear that a lot more manual effort is required in order to fine-tune the GAN to perform well as a classifier. Further to this, a lot more data would be needed in order to achieve more effective results and we would need to train for a very long time to uncover proportionally smaller incremental improvements. Therefore, it is suggestive that it is perhaps better to expend manual effort tuning parameters and optimising something like the Random Forest Classifier.

The project was a success in that the work set out to be completed, was achieved. Implementation included extra work and experimentation than first anticipated, including a novel extension.

Bibliography

[1] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[2] T. N. R. I. . . O. 2017, "Card fraud losses reach $22.80 billion." `https://www.nilsonreport.com/`, 2017.

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative Adversarial Networks," *ArXiv e-prints*, June 2014.

[4] A. van den Oord, S. Dieleman, H. Zen, K. Simonyan, O. Vinyals, A. Graves, N. Kalchbrenner, A. W. Senior, and K. Kavukcuoglu, "Wavenet: A generative model for raw audio," *CoRR*, vol. abs/1609.03499, 2016.

[5] Z. Wang, W. Yan, and T. Oates, "Time series classification from scratch with deep neural networks: A strong baseline," in *Neural Networks (IJCNN), 2017 International Joint Conference on*, pp. 1578–1585, IEEE, 2017.

[6] G. Lemaître, F. Nogueira, and C. K. Aridas, "Imbalanced-learn: A python toolbox to tackle the curse of imbalanced datasets in machine learning," *Journal of Machine Learning Research*, vol. 18, no. 17, pp. 1–5, 2017.

[7] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, pp. 1097–1105, 2012.

[8] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *CoRR*, vol. abs/1511.06434, 2015.

[9] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *CoRR*, vol. abs/1412.6980, 2014.

[10] A. Odena, "Semi-supervised learning with generative adversarial networks," *arXiv preprint arXiv:1606.01583*, 2016.

Appendix A

GAN Generated Outputs

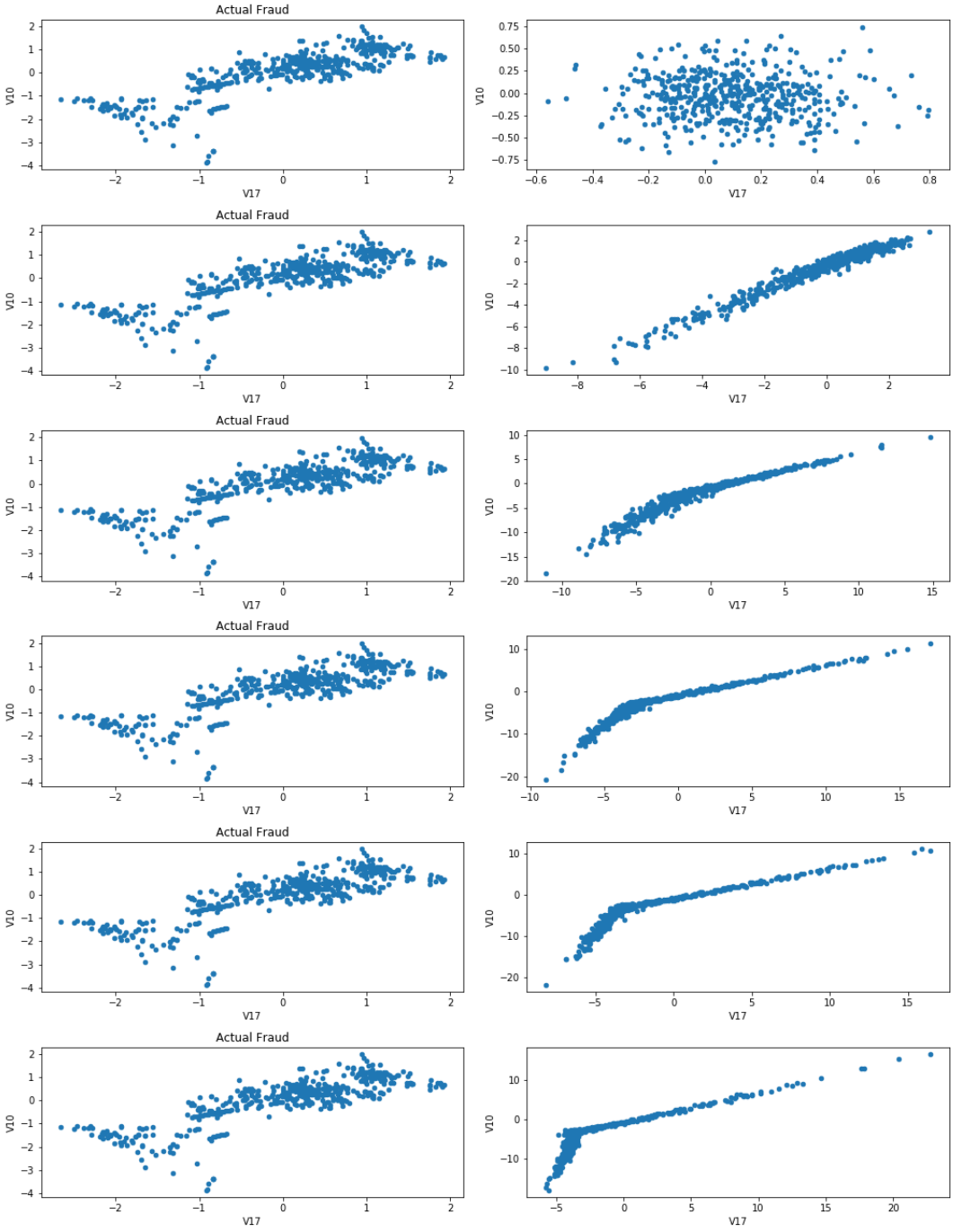


Figure A.1: GANv1.2 Generated Outputs for every 1000 epochs.

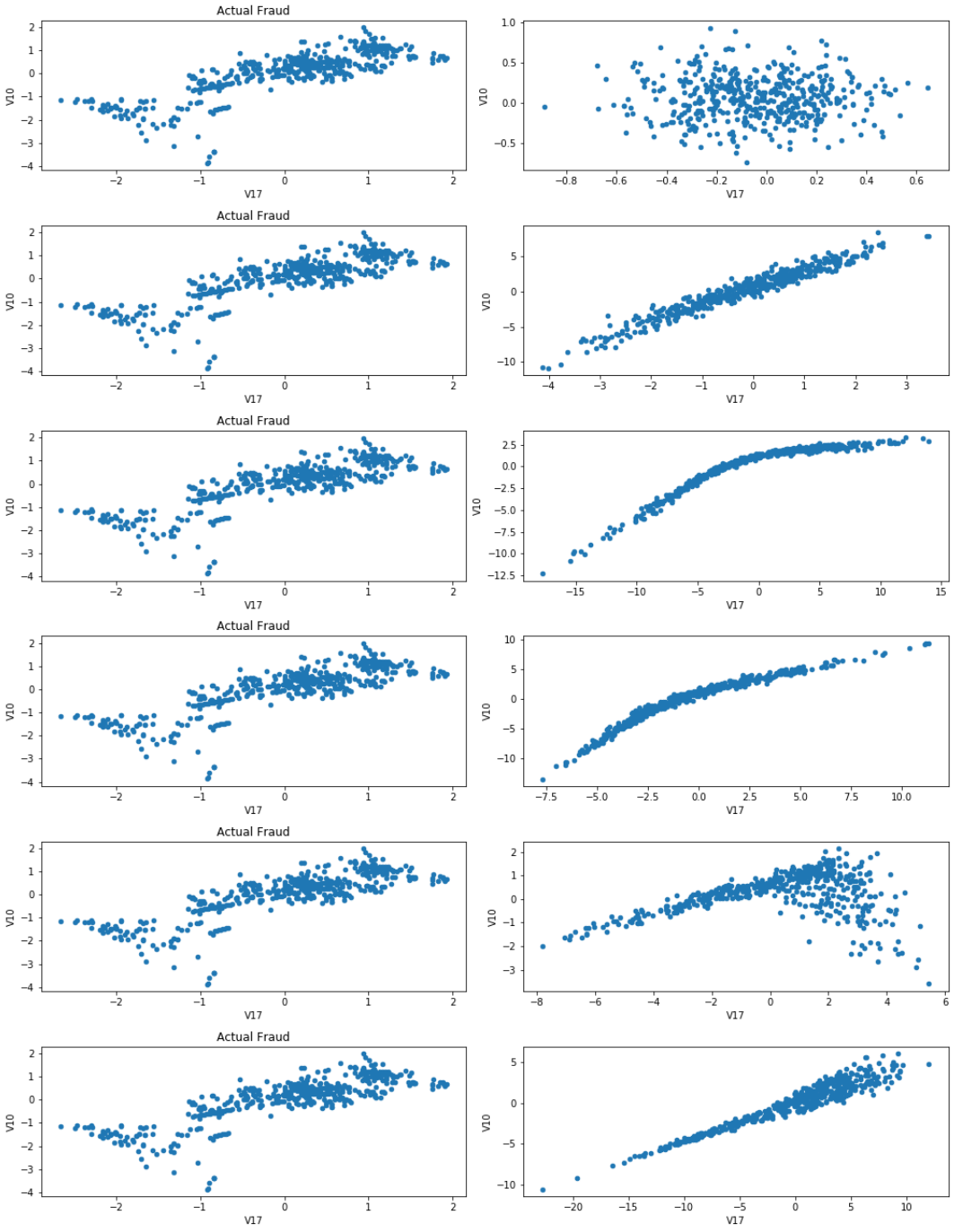


Figure A.2: GANv1.3 Generated Outputs for every 1000 epochs.

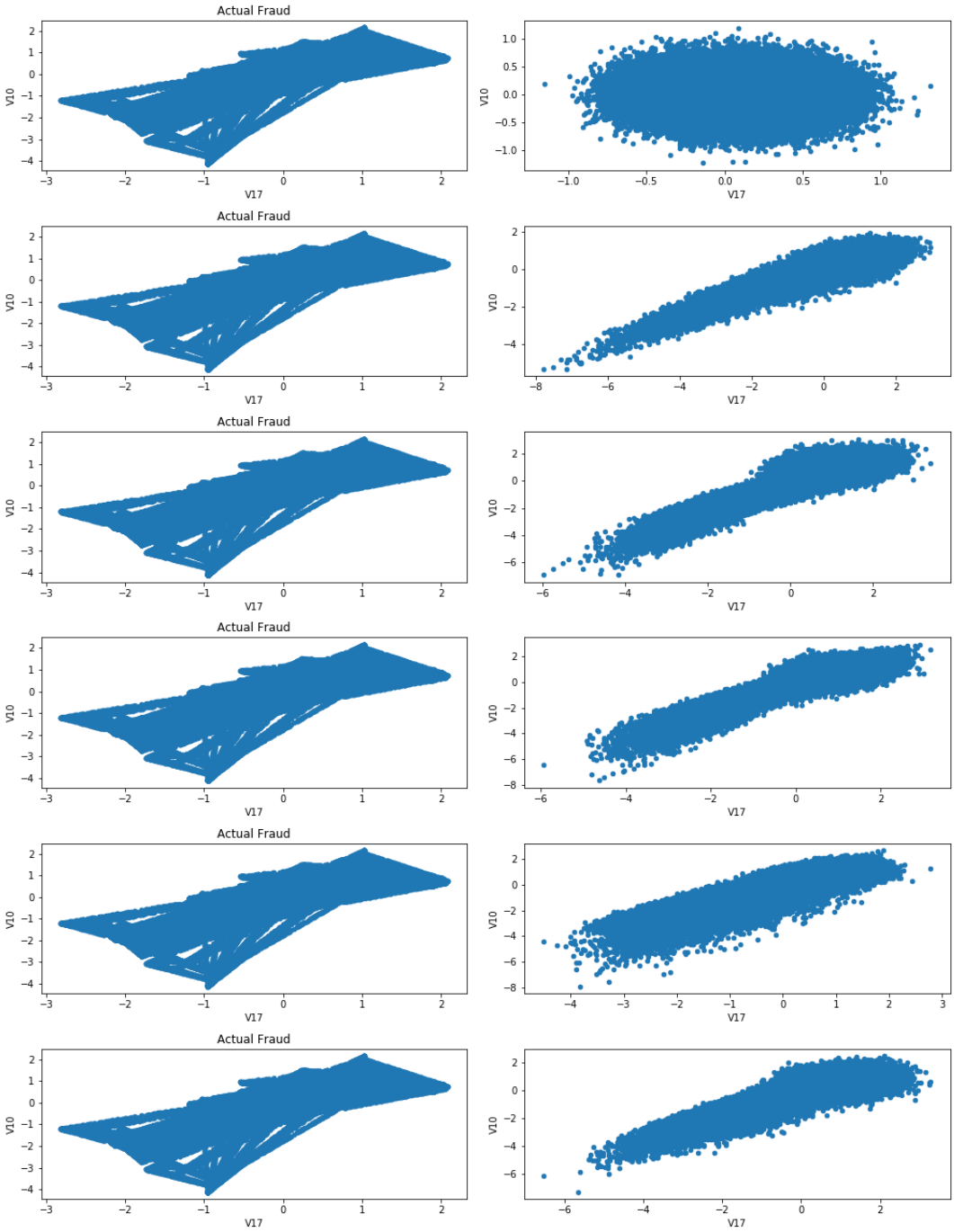


Figure A.3: GANv1.4 Generated Outputs for every 1000 epochs.

Appendix B

Project Proposal

UNIVERSITY OF CAMBRIDGE

Harry Graham
Christ's College
hg402

Computer Science Tripos - Part II Project Proposal

Deep Learning Techniques for Credit Card Fraud Detection

October 20, 2017

Project Supervisor: Dr M. Jamnik & B. Dimanov

Project Originator: H. Graham & B. Dimanov

Director of Studies: Dr R. Mortier

Project Overseers: Dr S. B. Holden & Dr N. R. Krishnaswami

Introduction

Background

A lot of machine learning concepts have been around for decades but ongoing research into deep learning architectures and their applications, makes for an interesting experimentation space. In this project I will explore the performance of some machine learning models, focusing on deep learning, applied to the particular problem of credit card fraud detection (CCFD). This is a globally significant and increasing problem, for example: Annual global fraud losses reached $22.80 billion last year alone, up 4.4% over 2015, according to Nilson Report [1].

Therefore, when techniques prove effective in other domains, or are of recent popularity, it becomes an immediate interest if these methods can be applied on the CCFD space.

The Project

The aim of this project is to explore the use of deep learning techniques in application to the CCFD space. The project comprises core components and possible extension work (mentioned in a later section).

The core project can be split up, at a high level, into the following portions:

- **Baseline Models**
  Setting the scene by exploring a set of very common, broadly used classification models to form a baseline for comparison. The idea being that these techniques are broadly used, there is lots of literature and they are becoming a 'standard' in the toolkit of many data scientists and developers. This baseline will serve as a series of metrics that represent what we can achieve *without* the more elaborate, deep learning techniques.

- **Deep Learning Models**
  This project looks towards comparing and analysing deep learning architectures that have recent examples of success in other domains and have sparked interest in the last decade and are somewhat novel to this data space. The idea is that data we harness, can be seen in many different conceptual views, we are the creator and can customise techniques and models that are already out there to our own uses.

  Convolutional Neural Nets (CNNs) [2]
  CNNs have been popular in the image recognition space. The aim is to experiment with two uses: 1) Treating as a classification problem and disregarding the time component of data, feeding single vector convolutions through the model and 2) Incorporating the time component and trying to utilise the temporal ordering, in a more realistic representation of real life context, using a higher dimension structure and CNNs' weight sharing, spatial locality properties to learn the data.

Generative Adversarial Networks (GANs) [3]
GANs are an interesting use of two neural networks that 'fight' each other and learn from each other's mistakes. The aim here is again, to experiment with this architecture with a few different proposed approaches. Due to the nature of GANs, we can see if the model can learn and simulate fraud data as it comes in based on whether our generative network can fool the discriminator. Further to this we can go down the route of changing the top layer of our CNN to generative and use this pre-trained model from before in our GAN implementation, or we can take one from scratch. This will give rise to a number of different metric comparisons to see which combination works and performs best.

Starting point

Code

Python SciKit-Learn [1] is a machine learning library, that will assist in the construction of the learning models. In general, the Python API is well documented and easy to use, so I plan to use this. Not to mention that Python itself is well suited to data science and data processing.

In addition to this, for the deep architectures I will utilise Keras and TensorFlow. Keras[2] is a Python deep learning library that can be run on top of TensorFlow and TensorFlow[3] is a machine intelligence library from Google. With the intention that I can do most work using Keras with a TensorFlow backend engine, but being able to drop down and fine tune directly with TensorFlow if needed. I plan to use these as, again, they are fairly well documented with intentions for this kind of work. Also, TensorFlow includes compatibility for GPU acceleration, when training models.

Computer Science Tripos

There are a number of courses in the Tripos that serve as a starting point for the reading and undergoing of this project:

- **Artificial Intelligence I** - neural networks, back propagation algorithm
- **Machine Learning and Bayesian Inference** - more machine learning tips, practical issues.
- **Algorithms, Software Engineering** - General software engineering and programming.

The courses in the undergraduate Tripos do not cover much practical detail regarding learning models and certainly do not go into the more advanced deep learning techniques that i'll be using in this project and so I plan to bridge this gap through extensive personal reading and experimentation as well as help from my project supervisors.

[1] http://scikit-learn.org/stable/
[2] https://keras.io
[3] https://www.tensorflow.org

Experience

My own personal experience is of a course a factor in the starting point of this project. I have working-proficiency experience in the Python programming language, from a recently completed summer internship, which will aid implementation. I also have some prior knowledge of machine learning techniques, both from lectures and also from personal reading, which will also act as a starting point for the preparatory reading of this project. I have no prior experience with TensorFlow, however, but this will be mitigated through self-learning in which I will make use of online resources and Google's documentation, as well as tapping into the experience of my supervisors.

Resources required

For this project I shall mainly use my own computer, iMac (27-inch, Late 2013), that runs MacOS Sierra. Backup will be to GitHub and weekly backups will be made to external drive. I will also make use of iCloud drive and a late 2013 MacBook pro too, should I need another machine. The training of models will be done on my own machines which should suffice.

Some the deep networks may require more performance in order to train the models, which could potentially take a long time on personal machines. In this case I will utilise the University's High Performance Computing Service[4].

I may also use computers in the intel lab to do any lighter, more portable work that is possible and certainly if both of my machines decide to break.

In terms of Data I will be using, I will start off with a popular CCFD from Kaggle[5]. The dataset has been collected and analysed during a research collaboration of Worldline and the Machine Learning Group (http://mlg.ulb.ac.be) of ULB (Universite Libre de Bruxelles) on big data mining and fraud detection [4]. There is also hope to obtain a larger dataset from FeatureSpace[6], a Cambridge-based world leader on CCFD. this would mean I have access to a lot more data which could be used in either in the core component of the project or certainly as an extended piece, investigating the effects of increased dataset size.

[4]https://www.cl.cam.ac.uk/local/sys/resources/hpc/
[5]https://www.kaggle.com/dalpozz/creditcardfraud/data
[6]https://www.featurespace.com

Work to be done

Baseline models

The first core component of this project is about implementing various supervised learning models in an attempt to set the scene for 'everyday' standard machine learning techniques, for classification problems. The breakdown of work in this section is as follows:

- **Resampling and Data Preparation**
  I will investigate resampling methods:
  under-sampling, over-sampling and the SMOTE method
  due to the imbalance of data, testing on a simple logistic regression classifier and optimising by tuning relevant parameters. Which ever method proves most effective, is what will be used feeding into the other classifiers to follow.

- **Implementation of various classifiers**
  I will implement and train the following classifiers on the data:
  K Nearest Neighbours, Linear SVM, Gaussian Process, Decision Tree, Random Forest, Neural Net, Naive Bayes.

Deep learning models

The second and more extensive core component of the project is focused on the deep learning techniques, that are novel to this CCFD data. I plan to investigate and ultimately compare the following techniques:

- **Convolutional Neural Networks** [2]
  The first stage here will be to plan and map out the customised CNN model for the data. After planning, the implementation will split into two sub-parts:
  1) Experimentation of using single vector convolutions, treating as a classification problem
  2) Using the temporal ordering and fixed-window, multiple vector approach.
  After implementing both of these CNN avenues, I will then experiment with some possible optimisations such as hyper-parameter tuning (e.g. on the size of the window, or how many vectors do we pass in the second approach).

- **Generative Adversarial Networks** [3]
  A similar approach of planning will be taken before implementing the sub-paths:
  1) Use our pre-trained CNN from the previous experiments, for the basis of our generator network in the GAN
  2) Use a new network from scratch, to give rise to lots of metrics on which we can compare and contrast and see what performs best.
  3) See if we can utilise the very nature of GANs and simulate fraud data.

In both cases (baseline and deep models), collating all the results gained for reference and comparison, in the evaluation chapter.

Success Criteria and Evaluation

Overview

The project will be a success in a mixture of quantitive and qualitative results. In terms of the work to be done, a summary of success would be:

- **Implemented and analysed baseline models**
- **Implemented deep learning models**
- **Conclusive analysis and insight of techniques, from gathered experimental results**

Metrics we're interested in

When evaluating the project, there are certain metrics that we are interested in that is not just the accuracy of prediction, like in many machine learning applications. Due to the nature of CCFD data, there will always be an imbalance in the data classes i.e. more non-fraud examples than fraudulent. Do to this property, we are not only interested in model accuracy, but both precision / recall too.

$$precision = \frac{T_p}{T_p + F_p} \quad , \quad recall = \frac{T_p}{T_p + F_n} \qquad T_p = TruePositive, F_n = FalseNegative$$

The idea here is that we care more about catching false negatives than we do about letting some false positives get through, so these metrics are important in evaluating the models used on CCFD data. In general, confusion matrix (TP, TN, FP, FN) results will be important for comparisons, visualised with receiver operating characteristic (ROC) curves.

The basis for evaluation will be using this data and comparing this with what is achieved by the deep models, proposed in the second part of the project. This will help give an intuition to how the performance compares with baseline models.

Of course, in some cases we will pull interesting results such as training time, In particular in the case of experimenting with GANs. Seeing how much training time is reduced by using our existing CNN and correlating this with performance, will be an interesting insight.

Evaluation Breakdown

Evaluation can be broken down into the following:

- **Baseline models**
  — Evaluating which sampling methods perform best, using the simple logistic regression classifier, using metrics described above.
  — Tuning parameters of the classifier to try and achieve better results.
  — Comparing the resulting metrics from all the implemented classifiers with one another, focusing on precision-recall curves.
  — Summarising this empirical data.

- **Deep learning models**
  When evaluating the deep models implemented, there will be a number of variations in what comparisons are drawn. Primarily between experiment, with that of the baseline models and then between the deep experiments themselves, in the following fashion:
  — Comparing results of CNN method 1 and 2, with that of baseline models.
  — Comparing results of CNN method 1, with that of method 2.
  — Comparing results of GAN method 1 and 2, with that of baseline models.
  — Comparing results of GAN method 1, with that of method 2.

  Using metrics described (precision-recall, training time, confusion matrix etc) and on the same dataset, using a training-test data split that is appropriate. For example a 70:30 split.

In addition to this, there will of course be a substantial qualitative analysis of the techniques implemented, covering areas such as intuition gained from the experiments, difficulty and overhead of training, a word on resource consumption, problems encountered / overcome and a general analysis on how the deeper learning techniques perform in this space.

Possible extensions

If the core parts of the project are successful and completed within reasonable time then possible extensions may lead to a few further high level investigations. Namely:

- **Implementing a DNC architecture**
  This would be an attempt at using the cutting edge technique proposed by a team from DeepMind [5] and exploring whether the use of external dynamic memory means that we can learn the entire history of the data and whether this is promising for the CCFD space.

- **Comparing deep auto encoders for pre-training** This extension would comprise of implementing a few deep auto encoder models and using these with baseline models and the deep models and seeing how effective these are.

Timetable

Planned starting date is 19/10/2017.

1. **Michaelmas weeks 3–5, [19/10 - 02/11]:** Preparatory reading on learning models, experiment with SciKit-Learn, Keras and TensorFlow. Experiment with importing and manipulating datasets. Reading on deep learning techniques. Look at under/over sampling techniques.

2. **Michaelmas weeks 5–6, [02/11 - 09/11]:** Implement baseline models and run training on dataset. Draw results for baseline models. Overflow on deep learning reading.

   Milestone**: Have implemented all baseline work and summarised results**

3. **Michaelmas weeks 6–8, [09/11 - 30/11]:** Investigate, plan and experiment with the first CNN implementation on the data. Build the network setup using Keras + TensorFlow. Run training on data.

   Milestone**: Trained the first implementation of the CNN, ready to setup the second**

4. **Michaelmas vacation** Write progress report. Generate visualisations of test results and how models performed, for demonstration purposes. Give insight into CNN work for CCFD. Prepare any necessities for implementing the second CNN implementation.

   Milestone**: Have a completed progress report and prepared a presentation for demonstration purposes**

5. **Lent weeks 0–2, [14/01 - 01/02]:** Continue / finish CNN work. Immediately start planning and setting down framework for implementing the GAN using Keras + TensorFlow.

   Milestone**: Completed and finalised results from CNN work.**

6. **Lent weeks 2–5, [01/02 - 22/02]:** Implement and train GAN. Start to finalise results and conclusions, allowing some overflow period for this main part of the project.

7. **Lent weeks 5–8, [22/02 - 15/03]:** Overflow period. If time permits, have a go at extensions. Clean-up, re-runs.

   Milestone**: Completed GAN work and all core project components.**

8. **Easter vacation:** Writing dissertation main chapters. Possible overflow of extensions.

9. **Easter term 0–2, [22/04 - 04/05]:** Complete dissertation. Proof reading and then an early submission so as to concentrate on examination revision.

References

[1] The Nilson Report Issue 1118 — Oct 2017. Card fraud losses reach $22.80 billion. `https://www.nilsonreport.com/`, 2017.

[2] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[3] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio. Generative Adversarial Networks. *ArXiv e-prints*, June 2014.

[4] Andrea Dal Pozzolo, Olivier Caelen, Reid A Johnson, and Gianluca Bontempi. Calibrating probability with undersampling for unbalanced classification. In *Computational Intelligence, 2015 IEEE Symposium Series on*, pages 159–166. IEEE, 2015.

[5] Alex Graves, Wayne, et al. Hybrid computing using a neural network with dynamic external memory. *Nature*, 538(7626):471–476, 2016.