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:	
Project Originators:	H. Graham & B. Dimanov
Supervisors:	B. Dimanov & Dr M. Jamnik

Original Aims of the Project

The primary aim of the project was to see whether we can use certain deep learning techniques on time series data, in particular for credit card fraud detection (CCFD). More specifically, I aimed to experiment with two popular types of architecture, namely Convolutional Neural Networks (CNNs) [1] and Generative Adversarial Networks (GANs). These have been successful in the image classification space and the aim of this project was to shed light on 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. I have also gone on to do some extension work relating to further investigation on the models I have experimented with. This is in the form of parameter tuning and further analysis not originally set out in the project proposal.

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. Not to mention, 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. In this project I explore the performance of some particular models, focusing on deep learning, applied to the particular problem of credit card fraud detection (CCFD).

In particular, the aim is to shed light on the use of 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 [3] and is novel to credit card fraud data. I first explore 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 easily available, in other words without any 'deep' learning components. Here, I also establish techniques and methods for processing and evaluating the data i.e cross-validation, datapoint scaling, and data visualisation.

Then the project shifts to experimenting with Convolutional Neural Networks (CNNs) [1] and Generative Adversarial Networks (GANs) [4]. The main aim of the project is to determine whether these can be applied to time-series data and in particular, the credit card fraud data. I experiment with two different variations of each, which I will outline in this dissertation. In the context of the GAN, I focus on whether this adversarial model is useful in learning the data before trying a semi-supervised, multitask learning approach in order to draw comparisons as a classification model.

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 and 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[4], used Git for version control, and used virtual environments with pip for project package management and requirements.

I made heavy use of Jupyter 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]https://github.com/harrygraham/DeepLearning-CreditCardFraud

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 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 can take many parameters, the most prominent though are:

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 **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:

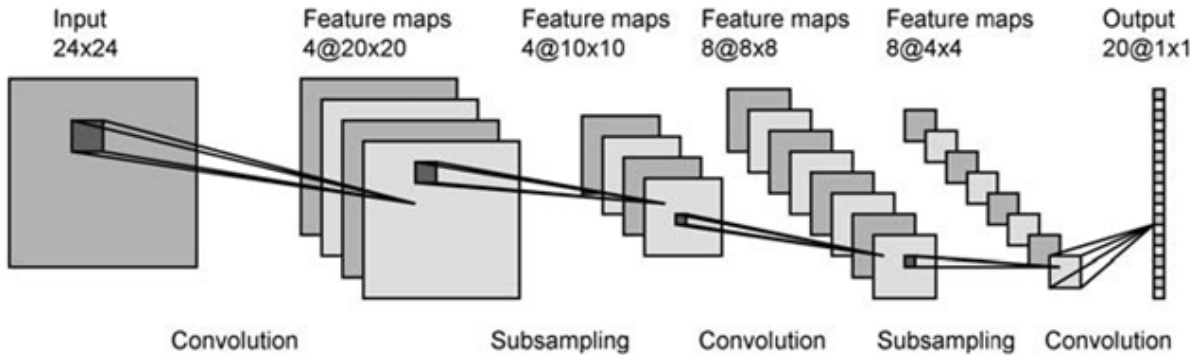

Figure 2.1: Example of a CNN network

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. 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 taking the average 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. The idea is that we have two networks: a generator and a discriminator. The generator network tries to produce, from random noise, data in the form of the training data we want. This could be an image, in the common use-case, or in this one, 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 discriminator tries to determine whether the current input is real or generated. Effectively these two networks play a game with each other: the generator tries to fool the discriminator whilst the discriminator tries to catch out the generator. This continues until each network doesn't get any better and the GAN stabilises. Figure 2.1 represents an overview of a GAN network.

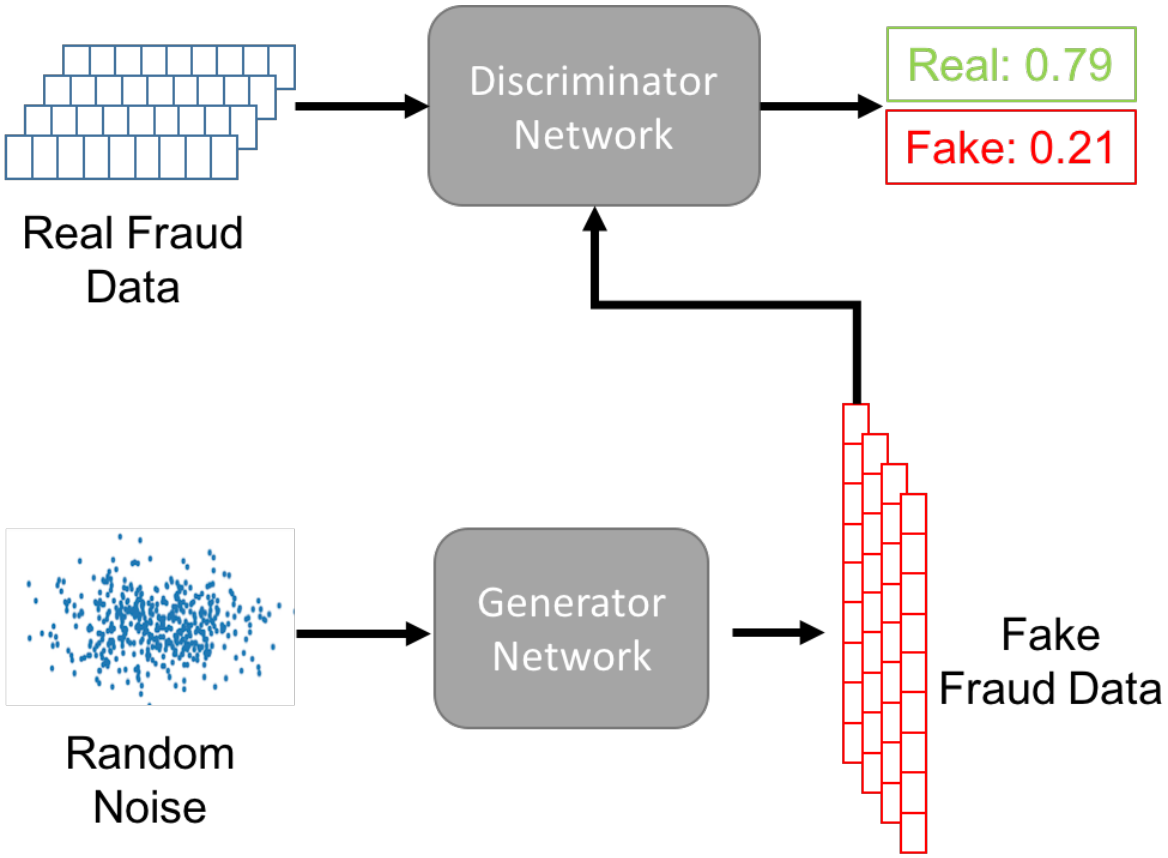

Figure 2.2: Overview of a GAN network

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

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

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 $D(x) \to 1$. The second term represents generated data passed through the discriminator. The generator tries to minimise such that $D(G(z)) \to 1$ (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 improvments

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 I had to have a clear idea of, prior to implementation of this project.

2.4.1 Train and Test Splits

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 $h : \mathbb{R}^n \to Y$ 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 $\mathbf{s} = [(\mathbf{x}_1, \mathbf{y}_1), (\mathbf{x}_2, \mathbf{y}_2), \ldots, (\mathbf{x}_m, \mathbf{y}_m)]$.

When training a model, to try and achieve our $h_w(x)$, we use data from the training set. When we train a model, it will often report some metrics or loss function to reflect how the performance increased during the training process.

The problem with this however 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.

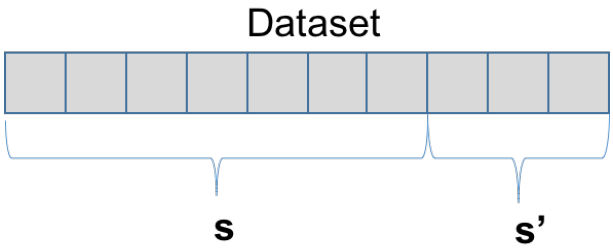

Figure 2.3: A 70:30 train test split of data.

2.4.2 Cross-validation

Cross-validation takes this 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.

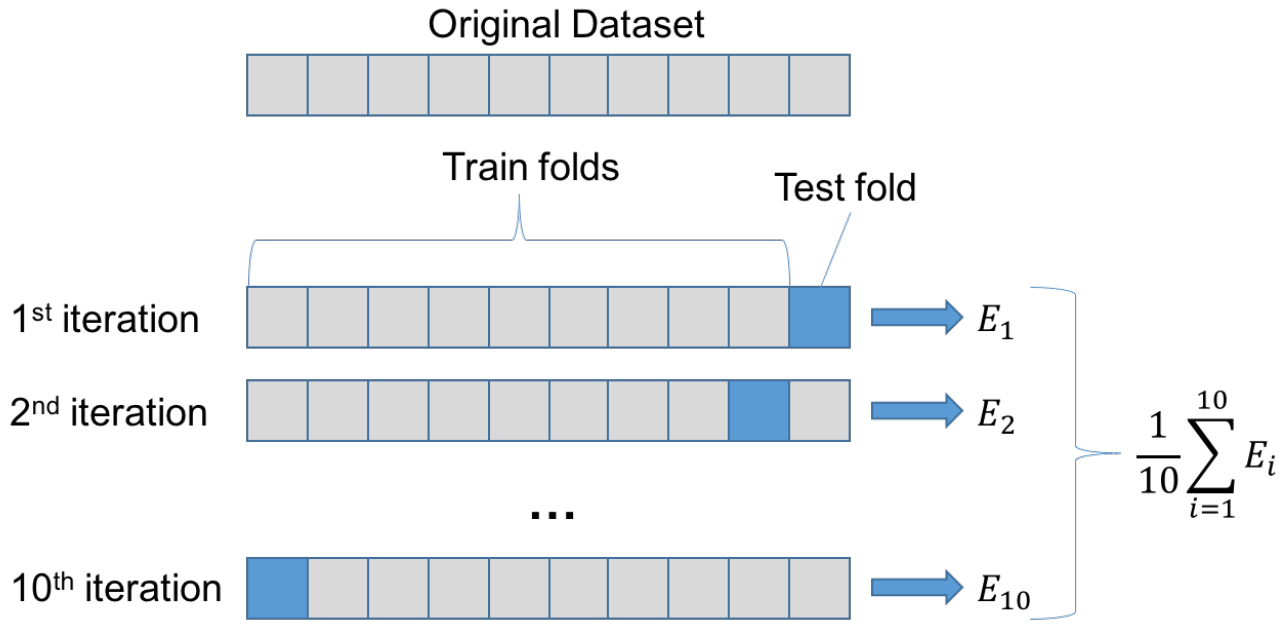

Figure 2.4: K-Fold Cross Validation.

2.4.3 Resampling Methods

During the baseline work of the project 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 explored were:

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.

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.

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

When employing oversampling techniques, this completely affects the way we handle cross-validation. It no longer is valid to simple oversample the dataset and then perform cross-validation. The reason for this is the potential of overfitting. Figure 2.5 shows what can happen in this situation.

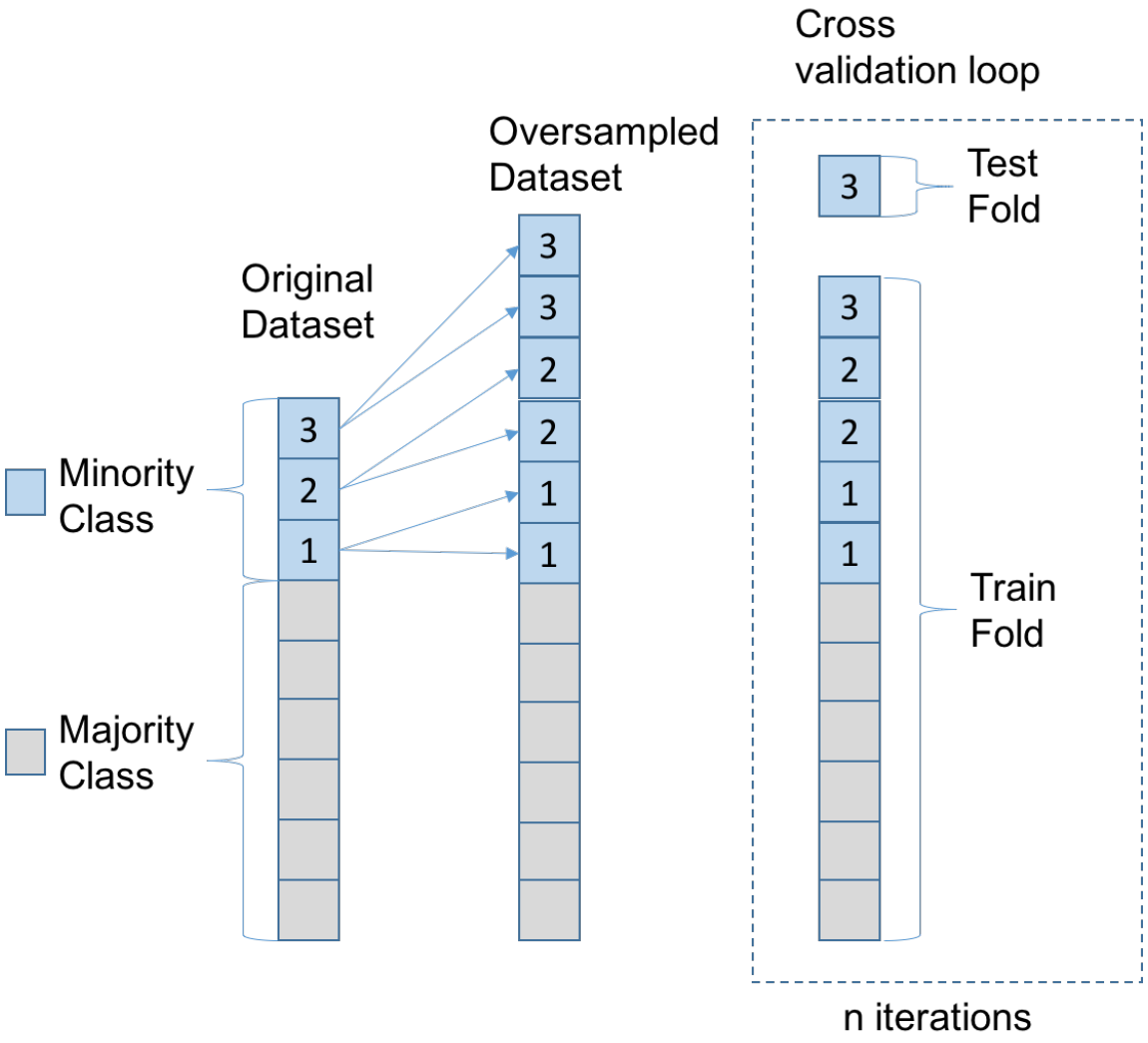

Figure 2.5: Oversampling OUTSIDE the cross-val loop.

Inside the cross-validation loop, the current train-test folds split means that some of the oversampled minority class (node 3) that was portioned into the test set is also in the training set. 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 split the data into train-test first and then oversample the training data fold only. This preserves the testing fold, keeping a portion of the original dataset untouched, for evaluation. This is a lot more effective in testing the generalisability of a model.

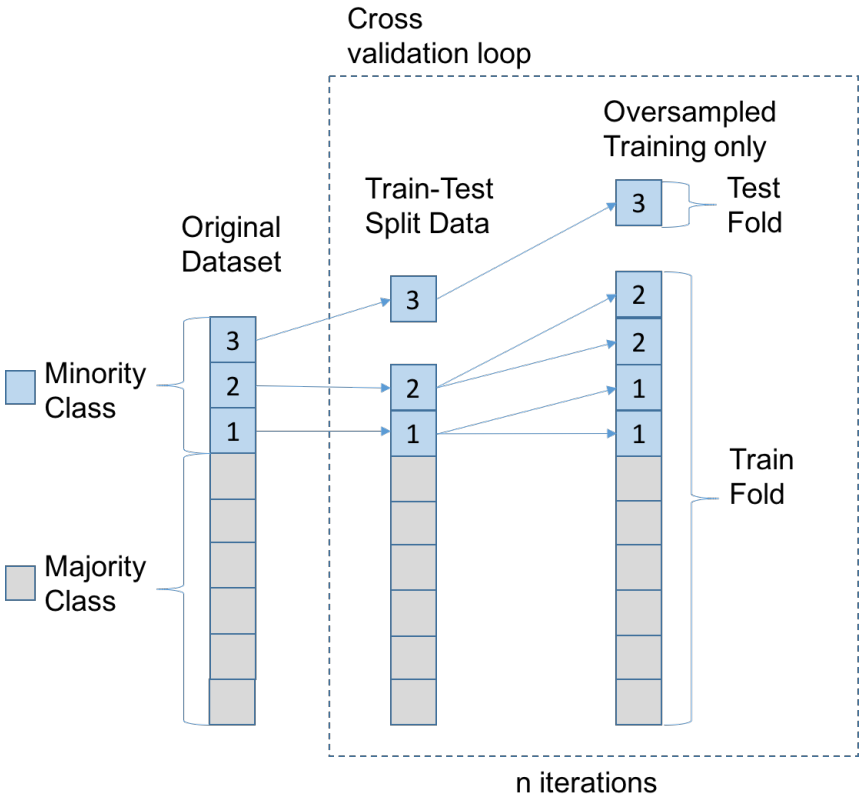

Figure 2.6: Oversampling INSIDE the cross-val loop [n=1].

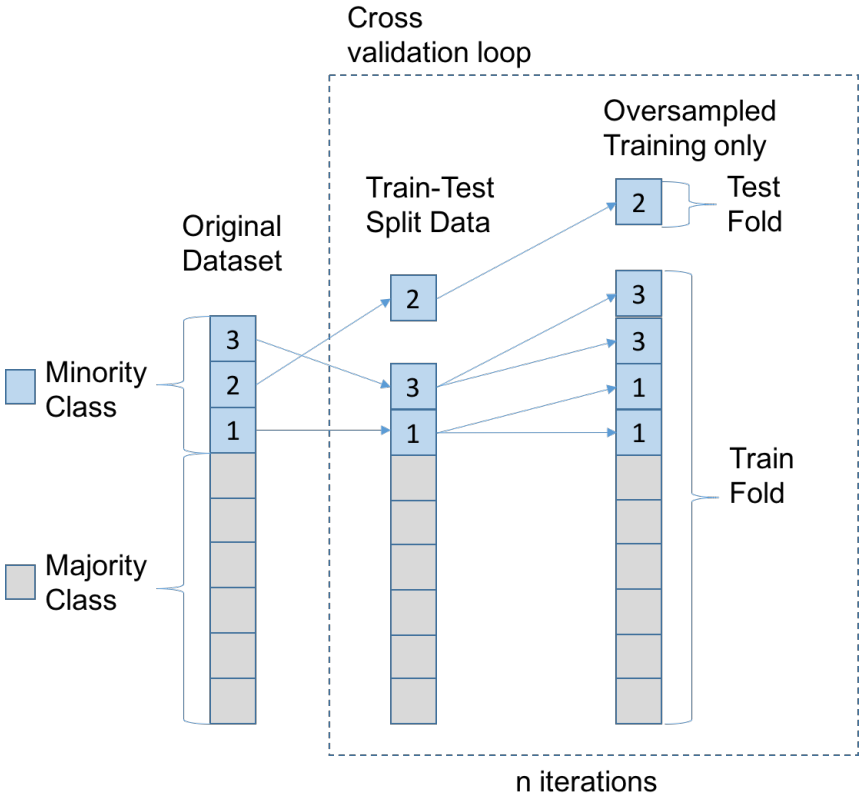

Figure 2.7: Oversampling INSIDE the cross-val loop [n=2].

Chapter 3

Implementation

3.1 Models Overview

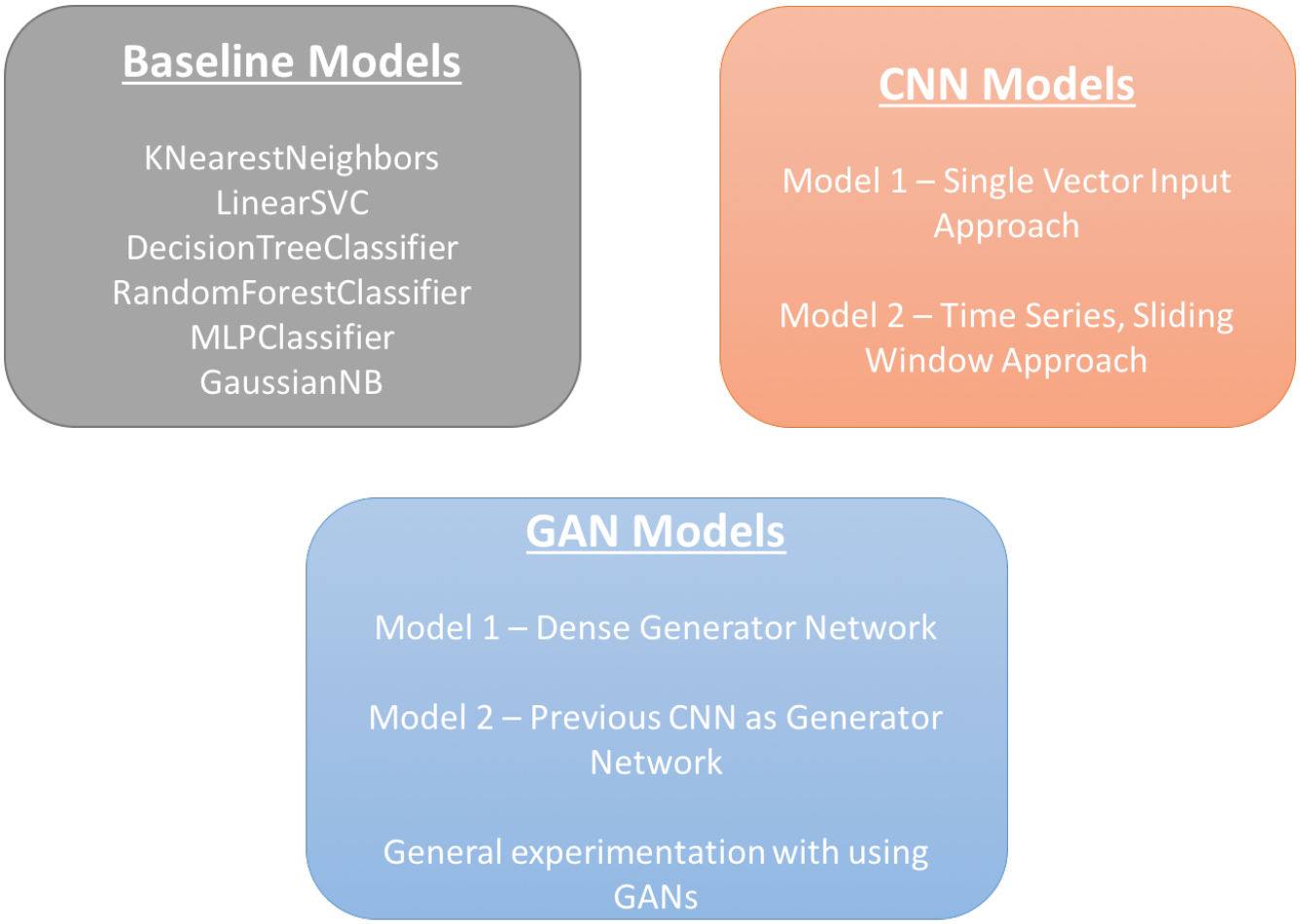

Figure 3.1: Overview of models experimented with.

3.2 Baseline Models

3.2.1 Data Preparation

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 on different ranges and the salary feature may negatively impact 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, though, as the data is already post-PCA and the time-series span of the transactions is only over 2 days. I decided not to do any feature engineering and to just employ algorithms on the original dataset features.

Sanity Checks

Also, 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.

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 and that the results are as confident as they can be. As far as implementation is concerned, It was not enough to use the built in SKLearn cross-val function as it did not provide enough granularity in it's output. I wanted very specific metrics and not to mention, 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 don't 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. You give it a set of predicted

values and the set of true values and it calculates the metrics. I use this inside my function to report the metrics we are interested in, something I could not do originally.

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
```

This is something I could then use for varying **Resample** methods.

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 approach that perhaps would be a logical one, would be to simply do this multiple times and average the results. This approach, however, will give better results than expected (which I showed in my

experiments, results outlined in the evaluation chapter). The reason for this is due to the nature of test_train_split and it's randomness.

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 we are not capturing that in our results. Also, It is not making maximal use of the data we have.

I ran my classifiers using this approach, too, as a means of comparison of what can go wrong and give invalid results, if you're not careful. The correct way, the way which was detailed above, is using KFold cross validation to split the data during a loop and doing any customisations or resampling inside this loop.

3.2.4 Resampling functions

To implement resampling functions I made use of the **imbalanced-learn**[5] 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.

Duplicate Oversampling
For random, duplicate Oversampling I used their **imblearn.over_sampling.RandomOverSampler.** This does as I described in the preparation chapter. Randomly selects a datapoint and duplicates it, or in other words 'selecting random data points, with replacement'.

SMOTE
For SMOTE, I used their **imblearn.over_sampling.SMOTE**. The default parameter of k_neighbors was used, which is a value of 5, which was fine for this use.

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'm interested in and running cross-validation for each of them, for each resampling method and collating results in a Python DataFrame log.

The algorithm outline can be viewed as follows:

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
```

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. Prior to implementing these models, I had to do thorough reading of how CNNs work (detailed briefly in the preparation chapter).

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 basic CNNv1 network.

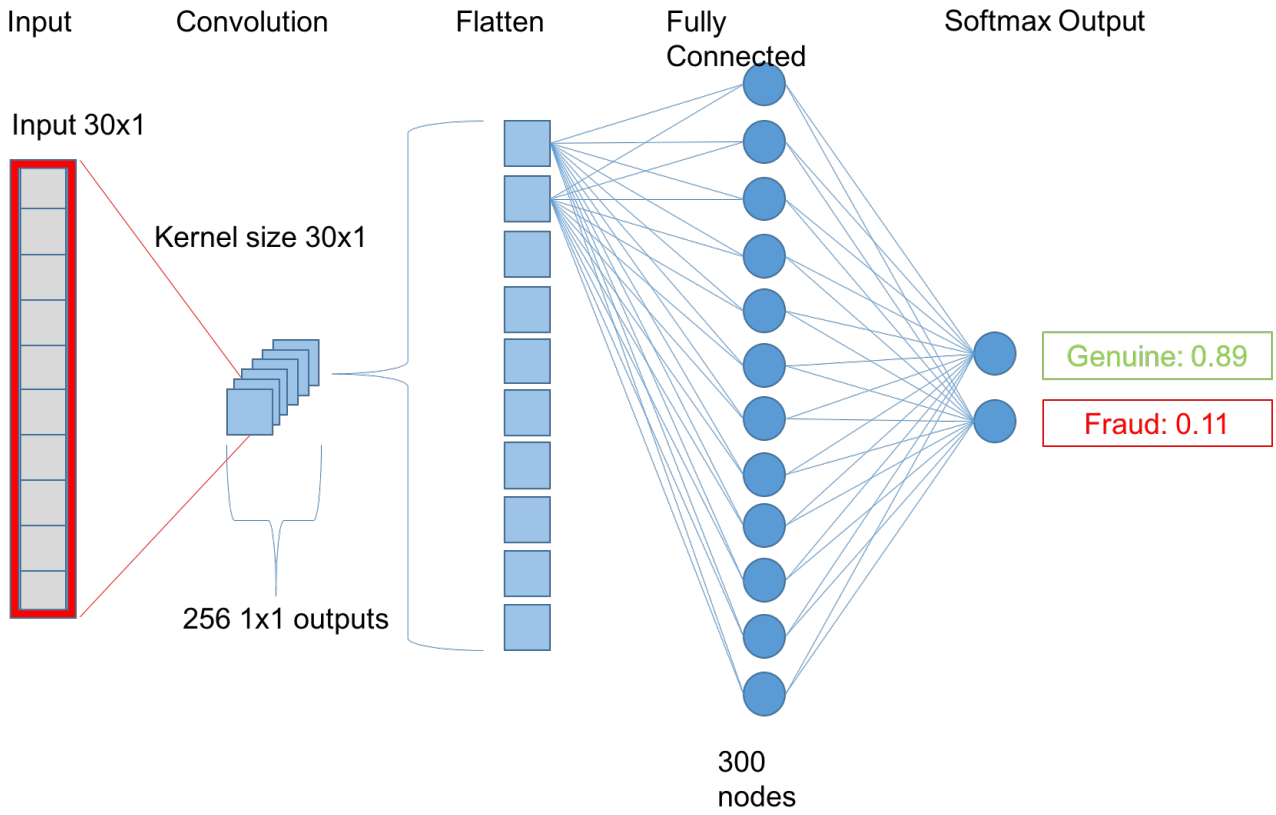

Figure 3.2: Overview of CNN Model 1.

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 was also reported and compared with Version 1-1.

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

```
# 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'))
    conv.add(Dense(100, 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 3.2: 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 and there are valid reasons for this, one perhaps most prominent is mitigating 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. We see that 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 we multiply near-zero quantities every time and hence we have 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[6] paper by Krizhevsky et al, they show the effect of using ReLU with respect to speed of training.

3.3.3 CNN Version 2

Overview

Version 2 aims to explore more 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 5 x 30 block which slides down the temporally ordered block and convolves with the features, to give the output.

Doing some maths here, 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).

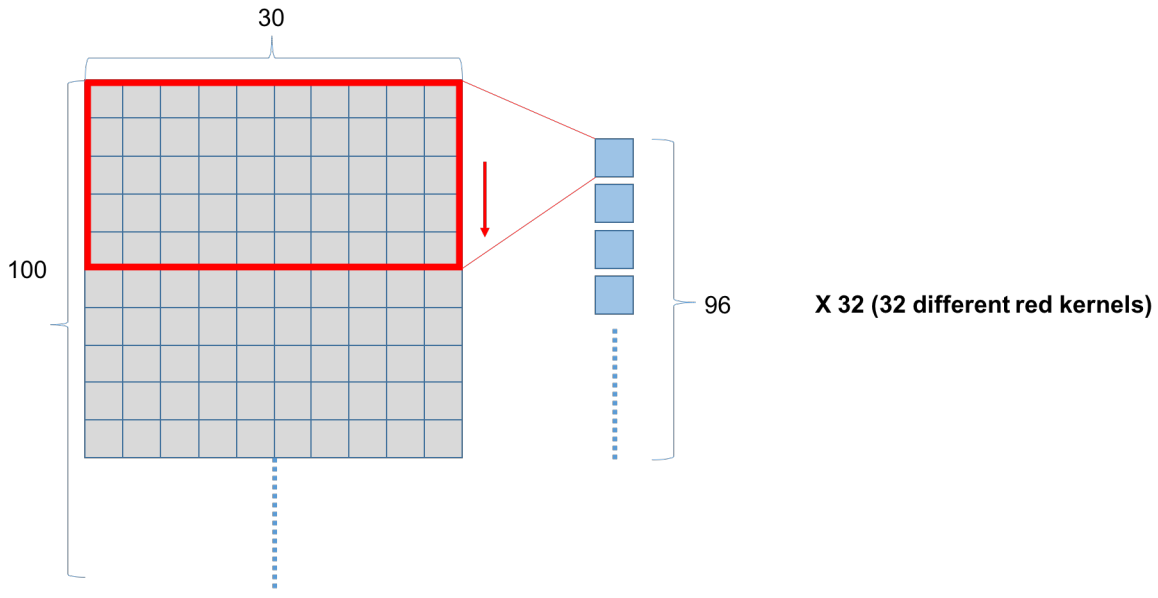

Figure 3.3: Overview of CNN Model 2 - Part 1.

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.

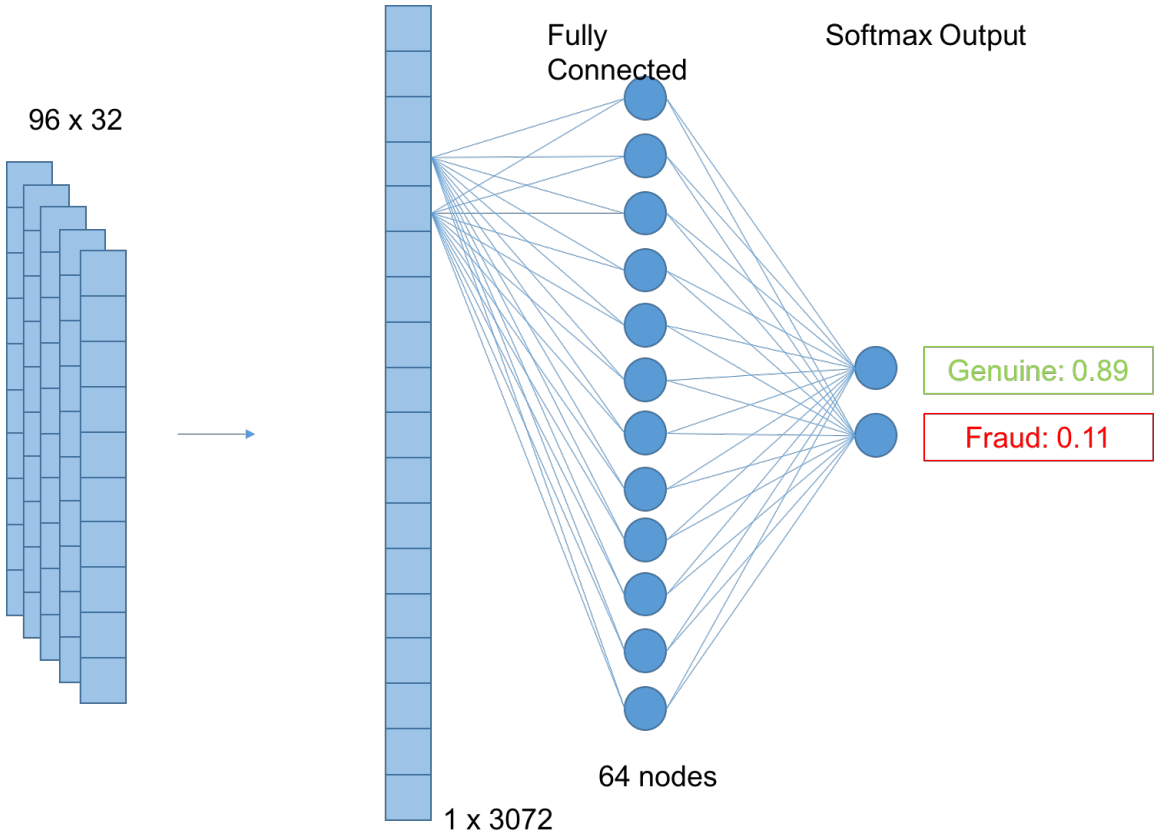

Figure 3.4: Overview of CNN Model 2 - Part 2.

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))
```

```
8     return np.array(np.split(a, batch_num))
```

Listing 3.3: CNNv2 batch reshape function.

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 import 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.

```
1  class GAN():
2
3      def init_v17_v10_plot(self):
4          '''Functions to initalise figure/plot variables for specific
       features,
5          by setting up figure and plotting real fraud data on the left hand
       side for comparisons. '''
6      def init_v17_v14_plot(self):
7
8      def load_fraud_data(self):
9          ''' Read in the dataset and do all of the necessary preprocessing (
       Feature scaling etc)
10         and store in accessible class variables.'''
11
12     def __init__(self):
13         '''Calls any initalisation functions, builds generator and
       discriminator networks,
14         compile the networks, set up combined model.'''
15
16     def build_generator(self):
17         '''Uses Keras functional API to create the generator network.'''
18
```

```
19      def build_discriminator(self):
20          '''Uses Keras functional API to create the discriminator network.
        '''
21
22      def train(self, epochs, batch_size=128, save_interval=200):
23          '''The adversarial training function for the GAN.'''
24
25      def save_loss_plot(self):
26          '''Utility to save the plot of losses for the GAN.'''
27
28      def save_imgs(self, epoch, img, gen_imgs):
29          '''Utility to save the plot of generated output for specified save
        intervals,
30          during the training process.'''
31
32      def test_as_classifier(self):
33          '''For use in GAN-SSL. Wraps up all the evaluation procedures for
        predictions and retrieving results metrics.'''
34
35  if __name__ == '__main__':
36      gan = GAN()
37      gan.train(epochs=6000, batch_size=32, save_interval=1000)
38      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 below in a high-level pseudo-code algorithm:

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 we train the discriminator on real and fake data separately, to give a stable and clear separation of the data.

3.4.3 Utility Functions

3.4.4 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 emplyed here. This is something that is quite common amongst GAN papers, in particular in DCGAN(deep convolutional GAN)[7]. 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[8] whereby it is stated "Using large models and datasets, we demonstrate Adam can efficiently solve practical deep learning problems" and showing it's 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.

3.4.5 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 in which the output 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.

3.4.6 Semi-Supervised Learning for Classification

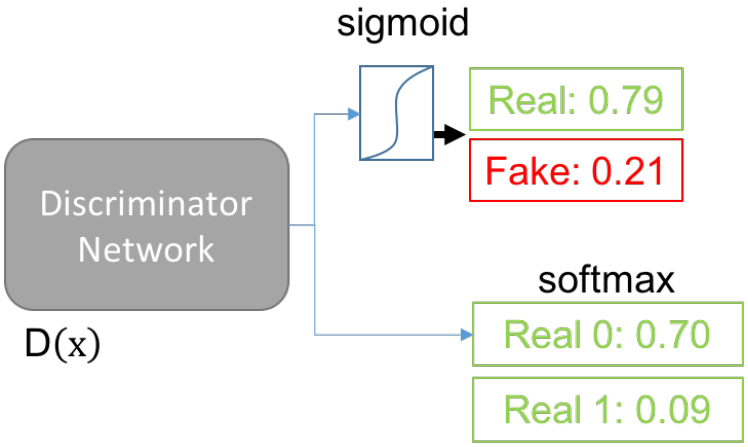

Figure 3.5: Overview of GAN-SSL discriminator output.

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*.

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

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.

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.

Taking the area under this curve is known as AUC, therefore gives an overall statistic for comparing classifiers.

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.

4.2 Results Overview

Chapter 5

Conclusions

1. Can you apply DL architectures to this problem? Yes, you can apply them, good/bad?
2. What can you compare it to, is it useful, how does it compare
3. What could the recommendation be, would you recommend to use DNN

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] 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.

[4] 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.

[5] 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.

[6] 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.

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

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