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:	Dr M. Jamnik & B. Dimanov

Original Aims of the Project

The primary aim of the project was to implement and compare some deep learning techniques, alongside some baseline models, 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].

Chapter 2

Preparation

2.1 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:

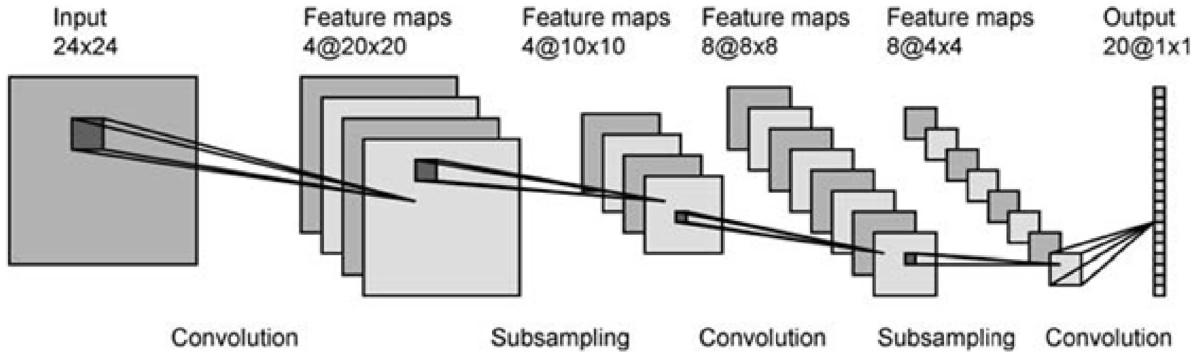

Figure 2.1: Example of a CNN network

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.

2.2 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.

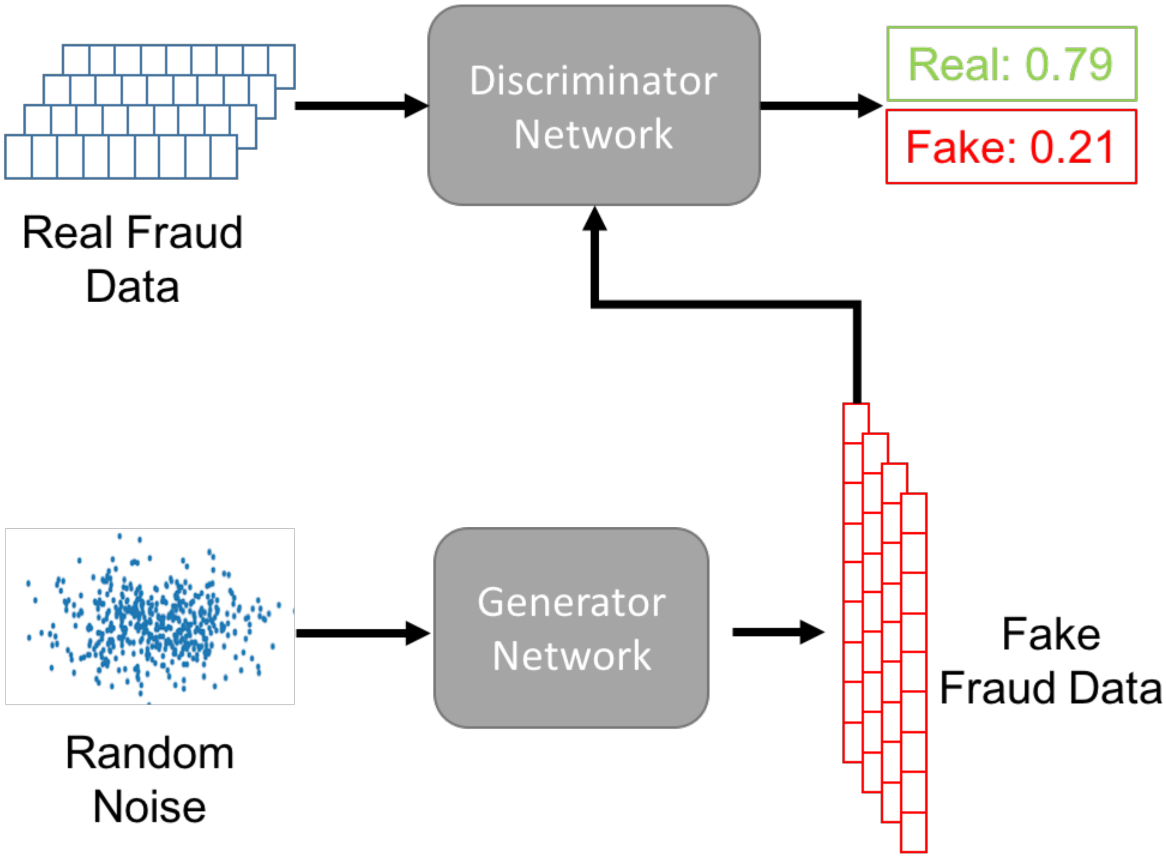

Figure 2.2: Overview of a GAN network

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.

2.3 Machine Learning Evalutation Practises

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.

2.3.1 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.

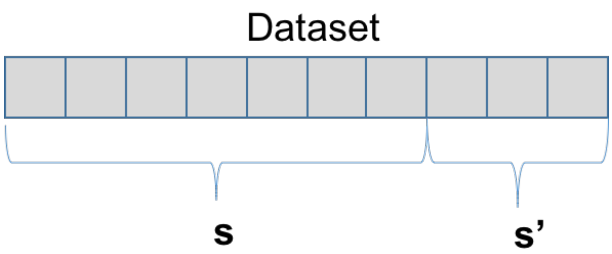

Figure 2.3: A 70:30 train test split of data

2.3.2 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.

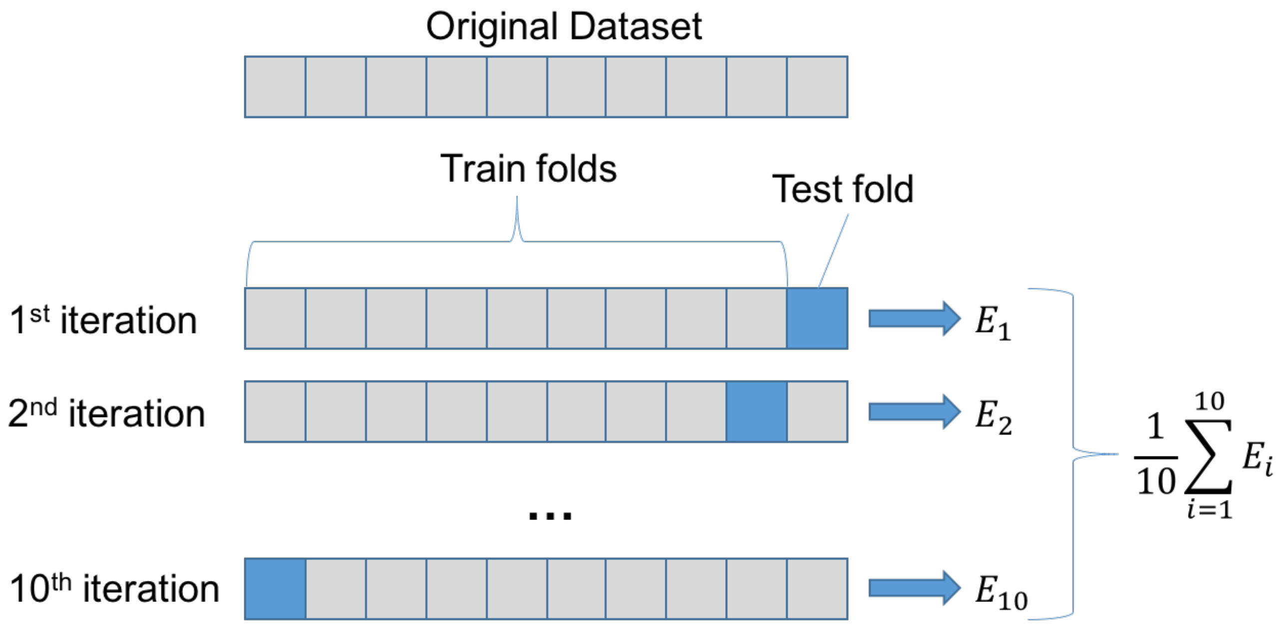

Figure 2.4: K-Fold Cross Validation

2.3.3 Resampling Methods

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. 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. Then, 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.

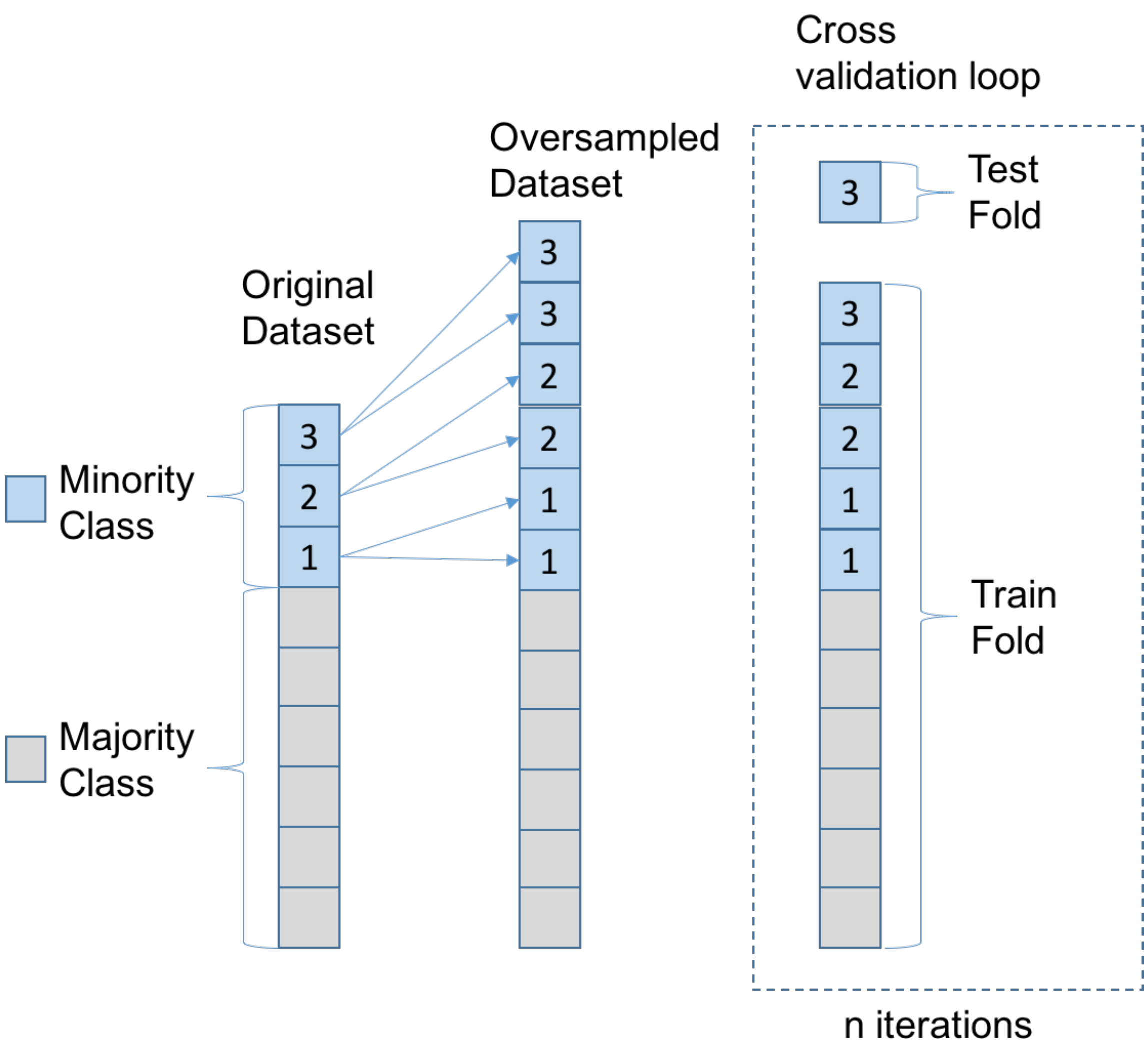

Figure 2.5: Oversampling OUTSIDE the cross-val loop

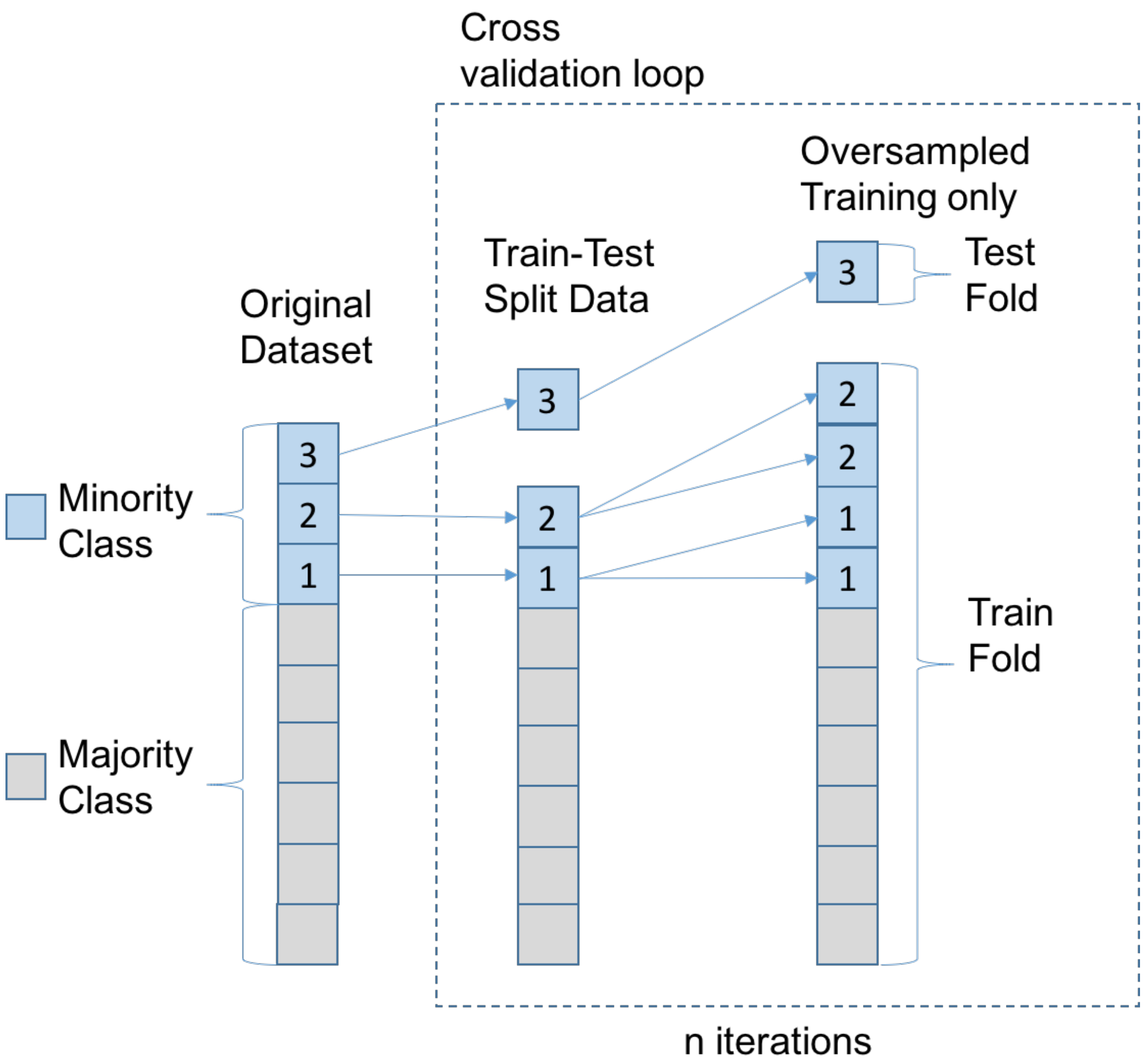

Figure 2.6: Oversampling INSIDE the cross-val loop [n=1]

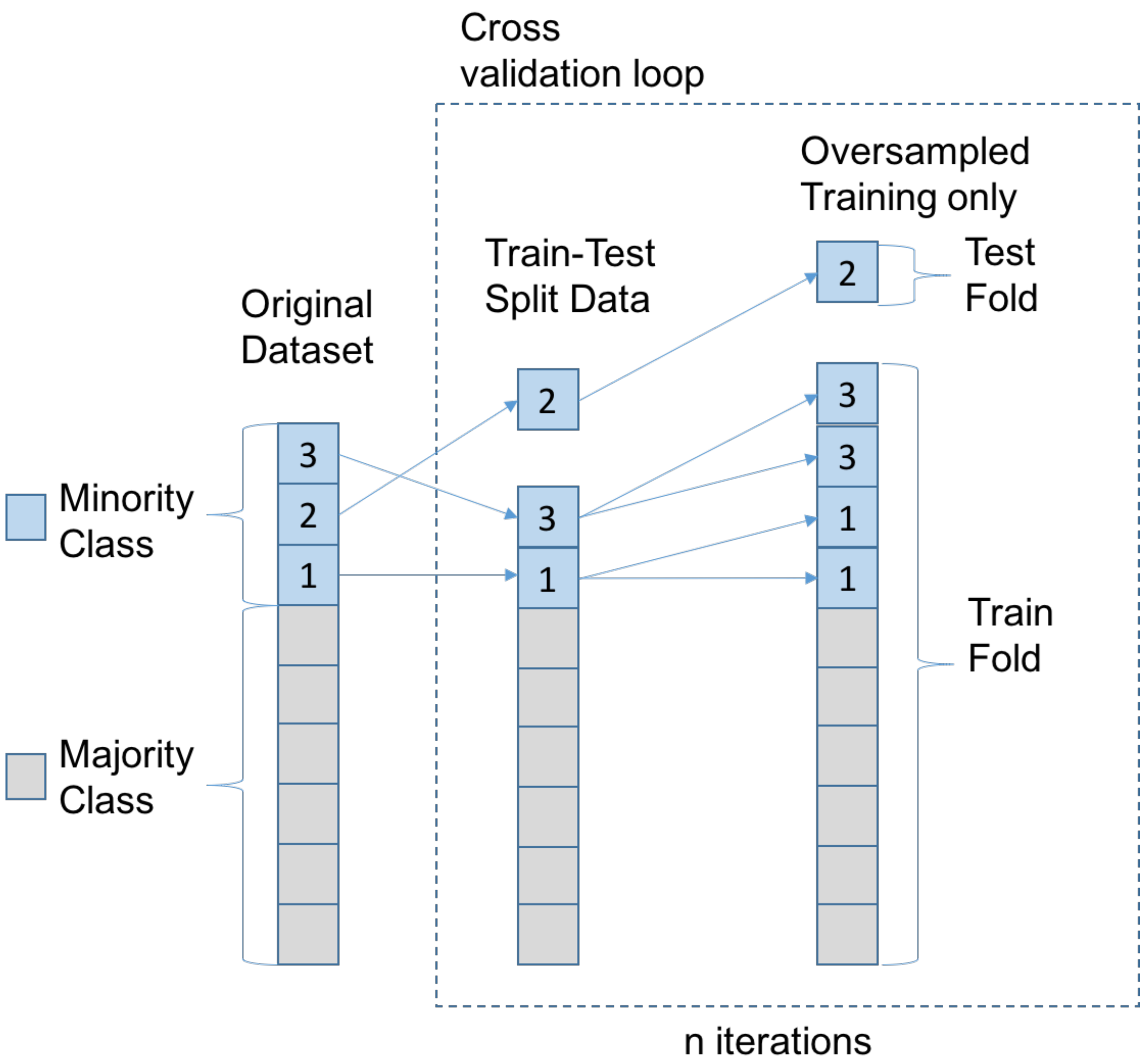

Figure 2.7: Oversampling INSIDE the cross-val loop [n=2]

2.4 Software Engineering

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

2.4.1 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.

2.4.2 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.

[1]https://keras.io
[2]https://www.tensorflow.org
[3] http://scikit-learn.org/stable/
[4]https://github.com/harrygraham/DeepLearning-CreditCardFraud

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.

2.4.3 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 had significant experience in Python, however, from a summer internship in industry as well as experience with Git.

Chapter 3

Implementation

3.1 Models Overview

3.2 Baseline Models

3.3 Convolutional Neural Network Models

3.4 Generative Adversarial Network Models

Chapter 4

Evaluation

4.1 Evaluation Methodology

4.1.1 Metrics

4.1.2 Visual Inspections

4.2 Results Overview

Chapter 5

Conclusions

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.