DETECTING CREDIT CARD FRAUDS USING PARALLEL NEURAL NETWORK

By

Ayush Dhiman 19BCE1463

Eshan Madnani 19BCE1470

A project report submitted to

Dr. SUDHA A.

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING

in partial fulfilment of the requirements for the course of

CSE4001 – PARALLEL AND DISTRIBUTED COMPUTING

B.Tech. (Computer Science Engineering)

VIT UNIVERSITY, CHENNAI
Vandalur – Kelambakkam, Chennai - 600127

CERTIFICATE

Certified that this project report entitled "**Detecting Credit Card Frauds using Parallel Neural Networks"** is a bonafide work of Ayush Dhiman 19BCE1463 and Eshan Madnani 19BCE1470 who carried out the "J"-Project work under my supervision and guidance for CSE4001– Parallel and Distributed Computing.

Dr. Sudha A.

SCHOOL OF COMPUTING SCIENCE AND ENGINEERING

VIT University, Chennai

Chennai – 600 127

ACKNOWLEDGEMENT

We are highly indebted to **Dr. Sudha A.** for their guidance and constant supervision as well as for providing necessary information regarding the project & also for their support in completing the project.
We would like to express our gratitude towards our university VIT Chennai for their kind co-operation and encouragement which helped us in completion of this project.
Our thanks and appreciations also go to our colleagues in developing the project and people who have willingly helped me out with their abilities.

TABLE OF CONTENTS

ABSTRACT

A credit card is a payment card gave to clients (cardholders) to empower the cardholder to pay a vendor for labor and products dependent on the cardholder's guarantee to the card guarantor to pay them for the sums in addition to the next concurred charges. The card backer (generally a bank) makes
a spinning record and awards a credit extension to the cardholder, from which the cardholder can get cash for payment to a vendor or as a loan. Nowadays we observe a ton of fraud credit cards being utilized on the lookout. In this way, we have made a task which would assist with distinguishing fraud credit cards utilizing equal neural organizations.
For this undertaking we have utilized errand and information level parallelism utilizing python strings. In this the PC codes across various processors in equal figuring conditions. It centers around appropriating undertakings simultaneously performed by cycles or strings across various processors during the preparation of the neural organizations to be utilized for ordering the credit card cases. Afterward, we have utilized 2 AI models, for example, Decision tree and arbitrary timberland classifier too for investigating elective Machine Learning ways to deal with fit our concern articulation and dataset. These models helped in the learning technique for characterization, relapse and different errands that work by building a large number of choice trees at preparing time and yielding the class.
We attempted our distinctive neural organization models across various hyper boundaries to see which model will give the best exactness and which hyper boundary will be best for our task. In our dataset we discovered that a large portion of the cases were negative and the positive cases were exceptionally less so we were getting one-sided results. So to adjust it and to get legitimate outcomes we utilized a technique for over examining and attempted various models with various boundaries and afterward discovered the one which gave the best precision.
In our model we utilized strategic relapse as an arrangement calculation which assisted with anticipating the likelihood of an objective variable. For this situation the objective variable being fraud credit cards.
All things considered in this undertaking we have attempted to build the security of monetary exchanges so nobody gets tricked or cheated and individuals become more mindful of such frauds.

INTRODUCTION

A credit card is a dainty convenient plastic card that contains ID data like a mark or picture, and approves the individual named on it to charge buys or administrations to his record - charges for which he will be charged intermittently. Today, the data on the card is perused via computerized teller machines (ATMs), store perusers, bank and is likewise utilized in web-based web banking framework. They have a special card number which is of most extreme significance. Its security depends on the actual security of the plastic card just as the protection of the credit card number. There is a fast development in the quantity of credit card exchanges which has prompted a generous ascent in fraudulent exercises. Credit card fraud is a wide-running term for burglary and fraud submitted utilizing a credit card as a fraudulent wellspring of assets in a given exchange. By and large, the factual strategies and numerous information mining calculations are utilized to tackle this fraud location issue. The vast majority of the credit card fraud identification frameworks depend on man-made consciousness, Meta learning and example coordinating.
Credit card fraud, an idea remembered for the more extensive thought of monetary frauds , is a point drawing in an expanding consideration from established researchers. This is expected, from one perspective, to the increasing costs that they produce for the framework, arriving at billions of dollars in yearly misfortunes and a rate loss of incomes equivalent to the 1.4% of on-line payments. Then again, credit card frauds have significant social results and consequences, as they support coordinated wrongdoing, illegal intimidation subsidizing, and global opiates dealing.
Credit card frauds can be made in numerous ways like straightforward burglary, application fraud, fake cards, never gotten issue (NRI) and online fraud (where the card holder is absent). In web-based fraud, the exchange is made from a distance and just the card's subtleties are required. A manual signature, a PIN or a card engrave are not needed at the hour of procurement. However anticipation instruments like CHIP&PIN decline the fraudulent exercises through straightforward burglary, fake cards and NRI and online frauds (web and mail request frauds) are as yet expanding in both sum and number of exchanges. There has been a developing measure of monetary misfortunes because of credit card frauds as the use of the credit cards become increasingly normal. Many papers detailed colossal measures of misfortunes in various nations. As per Visa reports about European nations, around half of the entire credit card fraud misfortunes in 2008 are because of online frauds. Credit card fraud location is a very troublesome, yet in addition famous issue to address. Initially, there comes just a restricted measure of information with the exchange being submitted, for example, exchange sum, date and time, address, shipper classification code (MCC) and acquirer number of the vendor. There are a large number of potential spots and online business destinations to utilize a credit card which makes it very hard to coordinate with an example. Additionally, there can be past exchanges made by fraudsters which likewise fit an example of typical (genuine) conduct. Besides, the issue

has numerous requirements. Most importantly, the profile of typical and fraudulent conduct changes continually. Besides, the advancement of new fraud identification techniques is made more troublesome by the reality

that the trading of thoughts in fraud discovery, particularly in credit card fraud identification is seriously restricted because of safety and protection concerns. Thirdly, informational indexes are not made accessible and the

results are regularly edited, making them hard to survey. On account of this issue, there is no possibility of benchmarking for the models constructed. Indeed, a portion of the examinations are finished utilizing artificially produced information [9-10]. None of the past examinations with genuine information in the writing give insights regarding the information and the factors utilized in classifier models. Fourthly, credit card fraud informational indexes are exceptionally slanted sets with a proportion of around 10000 genuine exchanges to a fraudulent one.
Finally, the informational collections are additionally continually advancing making the profiles of typical and fraudulent practices continually evolving. The most usually utilized fraud discovery techniques are ANNs, choice trees, Support Vector Machines (SVM), LR, and meta-heuristics like hereditary calculations, k-implies bunching and closest neighbor calculations. These methods can be utilized alone or in joint effort utilizing gathering or metalearning procedures to construct classifiers.
In this review, a credit card fraud location framework dependent on various ANN and LR technique alongside choice tree and irregular backwoods is created. We have completed the representation part utilizing string level parallelism and envisioned the dataset utilizing relationship framework, disarray grid, boxplot, scatterplot, TSNE and histogram. We got more understanding on the dataset and how the qualities were inside related by isolating it into two subclasses while representation. We prepared the model utilizing strategic relapse, fostered the expense work and utilized it to decide the expectation of the fraud utilizing string level parallelism. We likewise thought about the time taken by model utilizing diverse number of strings and prepared the dataset utilizing the best outcomes. This dataset additionally contains consequences of aritificail neural network(ANN) by differing the hyperparamters and utilizing the RELU and TANH actuation capacities. The work done by both RELU and TANH was looked at and in this manner the end was made appropriately.

MOTIVATION

A wonder of the advanced time is the way probably the littlest things which can't be recognized can make huge harm somebody. While we as a whole use credit cards yet we can't abandon which are fraud and which are unique. That is a large portion of us use credit cards these days for every one of our payments however can't recognize whether or not they are fraud. Knowing whether it is fraud or not will have critical ramifications.

Credit card fraud cases are rising each. Often we can see articles in the paper of individuals asserting that they have lost critical measure of cash because of fraud exchanges or banks guaranteeing that a huge level of clients itself were viewed as fraud.

Physically deciding or breaking down a credit card fraud case is extremely challenging as we have a great many clients. A simple prudent answer for this is to utilize the aid Data Analytics in this area, where we can surrender it to PC calculations to study and dissect existing example in the fraud cases, gain from this information and train itself to become fit for anticipating new fraud cases. In spite of the comfort of information investigation, another issue that emerges in this field is the size of information. Ordinary credit card clients increments significantly. Not to mention people, it has become troublesome in any event, for PCs to monitor such a colossal data set of clients. Consequently acts the hero Parallel and Distributed Computing. Through this we can diminish the heap on a solitary machine or processor and split our errand between various processors through equal calculations to frame a proficient arrangement of this issue articulation.

This undertaking centers around deciding the fraud credit cards through Deep Learning, Machine Learning and Parallel Computing in a precise and time proficient way.

IMPLEMENTATION DETAILS:

This project can be divided into four parts for implementation:

- **Exploratory Data Analysis and Visualization using threads**
- **Training Single Neural Network using Data Parallelism among threads**
- **Training Multiple Neural Networks using Functional Parallelism among threads**

For all the above tasks, implementation has been achieved using Python Programming Language and threads architecture available in python under the _threads library.

The dataset that we have used in this project is a credit card fraudulent dataset available at the following link:
https://www.kaggle.com/mlg-ulb/creditcardfraud

The dataset consists of 31 columns, which represents the various features of over 2 lakh credit card users and a target variable indicating whether they are fraud or not.

A glimpse of the dataset has been shown below:

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	
1	Time	V1	V2	V3	V4	V5	V6	V7	V8	V9	V10	V11	V12	V13	V14	V15	V16	V17	V18	V1
2	0	-1.35981	-0.07278	2.536347	1.378155	-0.33832	0.462388	0.239599	0.098698	0.363787	0.090794	-0.5516	-0.6178	-0.99139	-0.31117	1.468177	-0.4704	0.207971	0.025791	0.
3	0	1.191857	0.266151	0.16648	0.448154	0.060018	-0.08236	-0.0788	0.085102	-0.25543	-0.16697	1.612727	1.065235	0.489095	-0.14377	0.635558	0.463917	-0.1148	-0.18336	-(
4	1	-1.35835	-1.34016	1.773209	0.37978	-0.5032	1.800499	0.791461	0.247676	-1.51465	0.207643	0.624501	0.066084	0.717293	-0.16595	2.345865	-2.89008	1.109969	-0.12136	-2
5	1	-0.96627	-0.18523	1.792993	-0.86329	-0.01031	1.247203	0.237609	0.377436	-1.38702	-0.05495	-0.22649	0.178228	0.507757	-0.28792	-0.63142	-1.05965	-0.68409	1.965775	-1
6	2	-1.15823	0.877737	1.548718	0.403034	-0.40719	0.095921	0.592941	-0.27053	0.817739	0.753074	-0.82284	0.538196	1.345852	-1.11967	0.175121	-0.45145	-0.23703	-0.03819	0.
7	2	-0.42597	0.960523	1.141109	-0.16825	0.420987	-0.02973	0.476201	0.260314	-0.56867	-0.37141	1.341262	0.359894	-0.35809	-0.13713	0.517617	0.401726	-0.05813	0.068653	-(
8	4	1.229658	0.141004	0.045371	1.202613	0.191881	0.272708	-0.00516	0.081213	0.46496	-0.09925	-1.41691	-0.15383	-0.75106	0.167372	0.050144	-0.44359	0.002821	-0.61199	-(
9	7	-0.64427	1.417964	1.07438	-0.4922	0.948934	0.428118	1.120631	-3.80786	0.615375	1.249376	-0.61947	0.291474	1.757964	-1.32387	0.686133	-0.07613	-1.22213	-0.35822	0.
10	7	-0.89429	0.286157	-0.11319	-0.27153	2.669599	3.721818	0.370145	0.851084	-0.39205	-0.41043	-0.70512	-0.11045	-0.28625	0.074355	-0.32878	-0.21008	-0.49977	0.118765	0.
11	9	-0.33826	1.119593	1.044367	-0.22219	0.499361	-0.24676	0.651583	0.069539	-0.73673	-0.36685	1.017614	0.83639	1.006844	-0.44352	0.150219	0.739453	-0.54098	0.476677	0.
12	10	1.449044	-1.17634	0.91386	-1.37567	-1.97138	-0.62915	-1.42324	0.048456	-1.72041	1.626659	1.199644	-0.67144	-0.51395	-0.09505	0.23093	0.031967	0.253415	0.854344	-(
13	10	0.384978	0.616109	-0.8743	-0.09402	2.924584	3.317027	0.470455	0.538247	-0.55889	0.309755	-0.25912	-0.32614	-0.09005	0.362832	0.928904	-0.12949	-0.80998	0.359985	0.
14	10	1.249999	-1.22164	0.38393	-1.2349	-1.48542	-0.75323	-0.6894	-0.22749	-2.09401	1.323729	0.227666	-0.24268	1.205417	-0.31763	0.725675	-0.81561	0.873936	-0.84779	-(
15	11	1.069374	0.287722	0.828613	2.71252	-0.1784	0.337544	-0.09672	0.115982	-0.22108	0.46023	-0.77366	0.323387	-0.01108	-0.17849	-0.65556	-0.19993	0.124005	-0.9805	-(
16	12	-2.79185	-0.32777	1.64175	1.767473	-0.13659	0.807596	-0.42291	-1.90711	0.755713	1.151087	0.844555	0.792944	0.370448	-0.73498	0.406796	-0.30306	-0.15587	0.778265	2.
17	12	-0.75242	0.345485	2.057323	-1.46864	-1.15839	-0.07785	-0.60858	0.003603	-0.43617	0.747731	-0.79398	-0.77041	1.047627	-1.0666	1.106953	1.660114	-0.27927	-0.41999	0.
18	12	1.103215	-0.0403	1.267332	1.289091	-0.736	0.288069	-0.58606	0.18938	0.782333	-0.26798	-0.45031	0.936708	0.70838	-0.46865	0.354574	-0.24663	-0.00921	-0.59591	-(
19	13	-0.43691	0.918966	0.924591	-0.72722	0.915679	-0.12787	0.707642	0.087962	-0.66527	-0.73798	0.324098	0.277192	0.252624	-0.2919	-0.18452	1.143174	-0.92871	0.68047	0.
20	14	-5.40126	-5.45015	1.186305	1.736239	3.049106	-1.76341	-1.55974	0.160842	1.23309	0.345173	0.91723	0.970117	-0.26657	-0.47913	-0.52661	0.472004	-0.72548	0.075081	-(

All the variables in the dataset are associated with the real time data of the credit card users. However due to user privacy policy the names of these variables are not disclosed.

PART 1 - Exploratory Data Analysis and Visualization using threads

This is the introductory part of the project which focuses on examining the distribution and pattern of the values in the dataset and obtaining the visualizations for better understanding of the same.

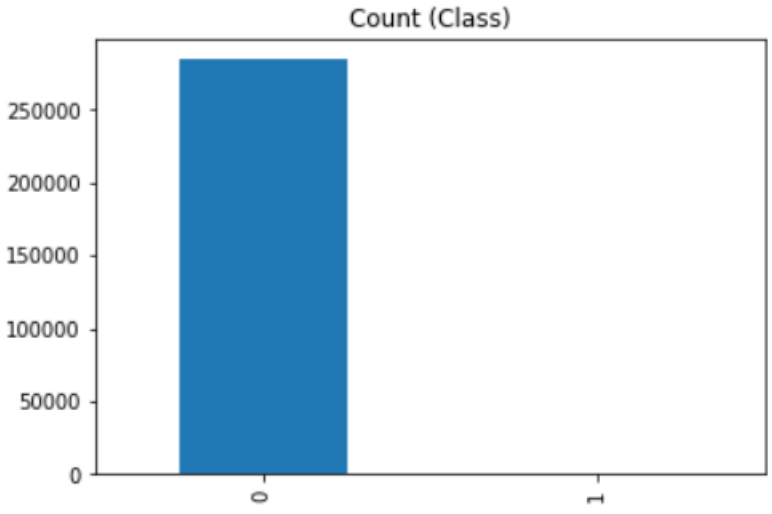
The following tasks has been performed under EDA (Exploratory Data Analysis)

- **Found the count of frauds in the dataset. Class 1 represents fraud and class 2 represents non-fraud.**

```
In [6]:  1 df.Class.value_counts()

Out[6]: 0    284315
        1       492
        Name: Class, dtype: int64
```

- **Plotted graph for both class 0 and class 1**

```
In [7]:  1 df.Class.value_counts().plot(kind='bar', title='Count (Class)')

Out[7]: <AxesSubplot:title={'center':'Count (Class)'}>
```



Here we can observe a clear case of class imbalance as the number of cases involved in fraud = 492 is significantly less than the number of non-fraud cases = 284315.
If we proceed with this data in the deep learning and machine learning models training part of the project then will always cause a biased decision towards the majority class.

To solve this problem of class imbalance we have used two methods:

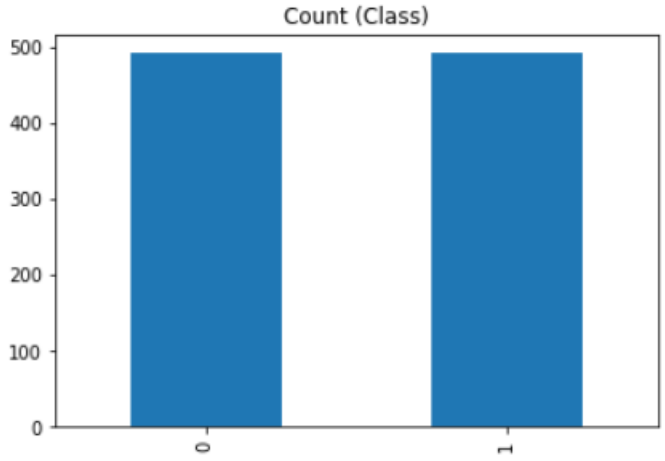
A. Random under sampling

In this method we take a certain number of random cases from the majority class, i.e., the non-fraud class equal to the number of cases present in the minority class, i.e., the fraud class and merge then together to form a new dataset as shown below.

In [8]:
```
class_count = df.Class.value_counts()
min_class = min(class_count)
print(min_class)

# Divide DataFrame by class
df_class_0 = df[df['Class'] == '0']
df_class_1 = df[df['Class'] == '1']

#Undersampling
df_class_0_under = df_class_0.sample(min_class)

df_us = pd.concat([df_class_0_under,df_class_1], axis = 0)
print(df_us.Class.value_counts())
df_us.Class.value_counts().plot(kind='bar', title='Count (Class)')
```

```
492
0    492
1    492
Name: Class, dtype: int64
```

Out[8]: <AxesSubplot:title={'center':'Count (Class)'}>



As we can see the number of cases in both classes in now = 492

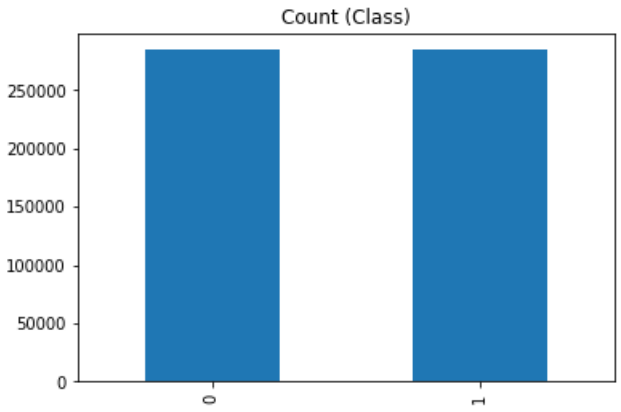
B. Oversampling:

In this method we replicate the records of the minority class, i.e., the fraud cases till it becomes equal to the number of records in the majority class, i.e., the non-fraud cases as shown below:

In [9]:
```
class_count = df.Class.value_counts()
max_class = max(class_count)
print(max_class)

# Divide DataFrame by class
df_class_0 = df[df['Class'] == '0']
df_class_1 = df[df['Class'] == '1']

# Oversampling
df_class_1_over = df_class_1.sample(max_class,replace = True)

df_os = pd.concat([df_class_0,df_class_1_over], axis = 0)
print(df_os.Class.value_counts())
df_os.Class.value_counts().plot(kind='bar', title='Count (Class)')
```

```
284315
0    284315
1    284315
Name: Class, dtype: int64
```

Out[9]: <AxesSubplot:title={'center':'Count (Class)'}>

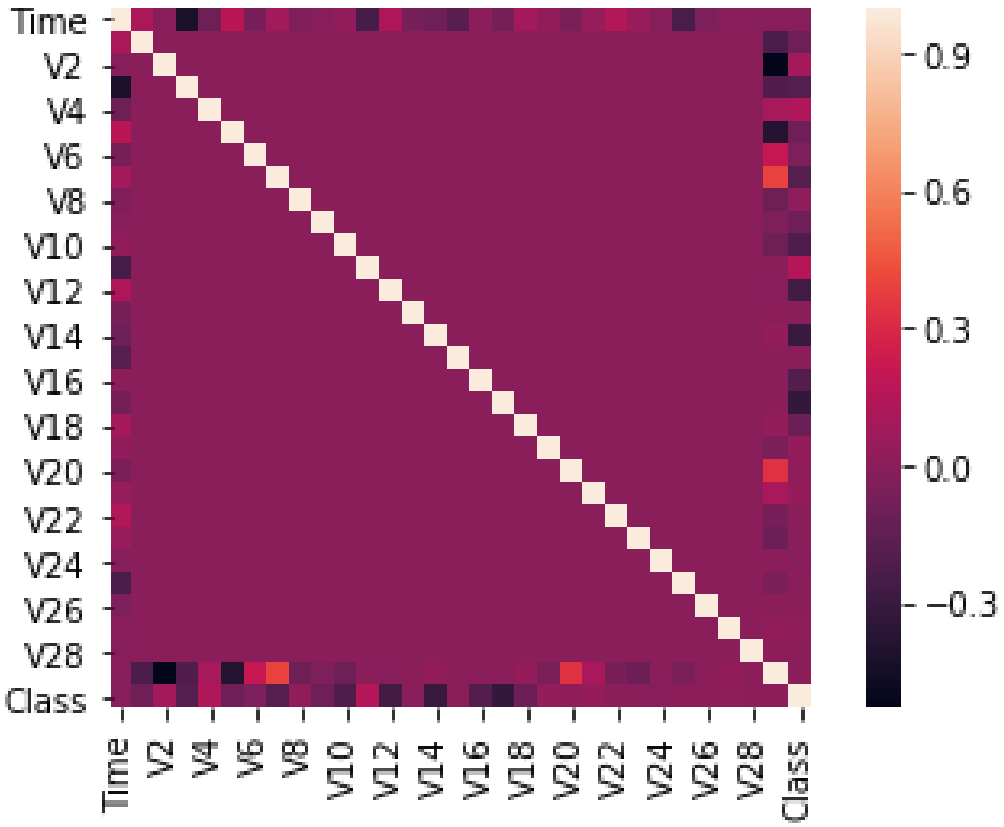


As we can see, now both the classes have 284315 cases.

- **Checking the amount of transaction involved in the top 10 frauds**

	Amount	Class
274771	25691.160156	0.0
58465	19656.529297	0.0
151296	18910.000000	0.0
46841	12910.929688	0.0
54018	11898.089844	0.0
169457	11789.839844	0.0
284249	10199.440430	0.0
227921	10000.000000	0.0
74699	8790.259766	0.0
245474	8787.000000	0.0

- **The correlation matrix between all the columns and rows of dataset**

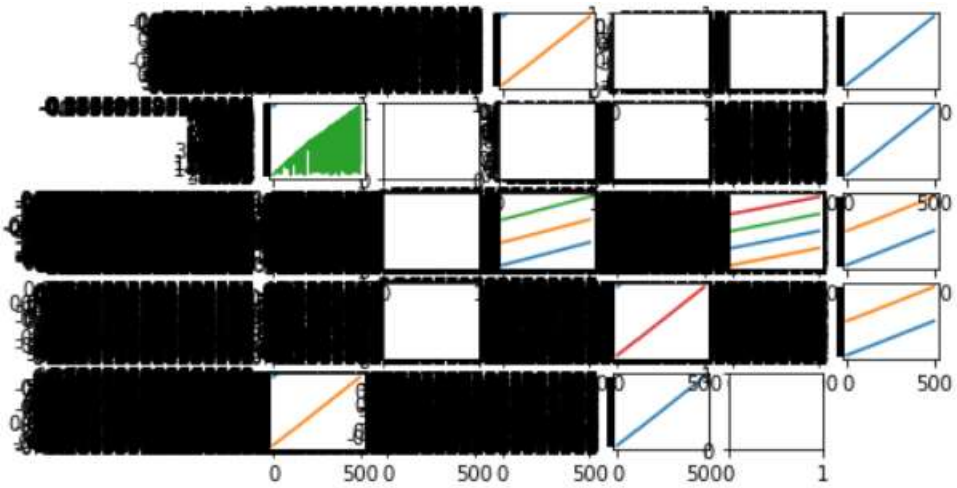


This graph shows that weak negative correlation around -0.3 is observed between variables V2 and V28, V6 and V28, V18 and Class variable, V14 and class variables.

Moderately strong correlation around 0.5 is observed between V20 and V28, V6 and V28. If we had known the name of these variables, their significance and relation would have been clearer.

- **Graphs plotted between different columns of the dataset**

These graphs show the distribution of each of the variables from V1 to V28 throughout all the cases in the dataset.



Firstly we plotted the points without using threads and the total execution time was coming 13 minutes 43 seconds.

In [13]:
```
start_time = datetime.now()
plot1()
plot2()
plot3()
plot4()
plot5()
plot6()
plot7()
plot8()
plot9()
plot10()
plot11()
plot12()
plot13()
plot14()
plot15()
plt.show()
end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

```
Duration: 0:13:43.793801
```

Then we used threads, these graphs are plotted in parallel by assigning all these graphs and distributing them among 15 threads as shown in the code below. The total execution time was 9 minutes 35 seconds, we can see a total of 48% decrease in time.

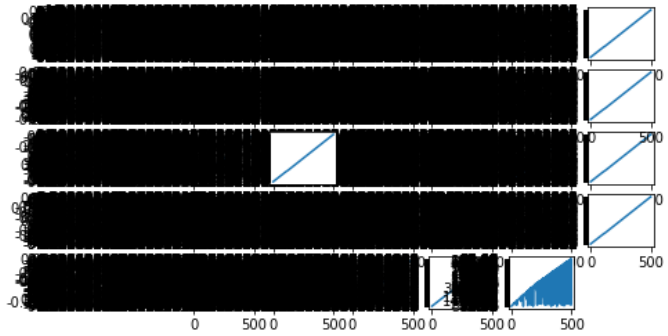
In [5]:

```
fig = plt.figure(figsize = (15, 12))
def plot1():
    plt.subplot(5, 6, 1) ; plt.plot(df.V1) ; plt.subplot(5, 6, 15) ; plt.plot(df.V15)
def plot2():
    plt.subplot(5, 6, 2) ; plt.plot(df.V2) ; plt.subplot(5, 6, 16) ; plt.plot(df.V16)
def plot3():
    plt.subplot(5, 6, 3) ; plt.plot(df.V3) ; plt.subplot(5, 6, 17) ; plt.plot(df.V17)
def plot4():
    plt.subplot(5, 6, 4) ; plt.plot(df.V4) ; plt.subplot(5, 6, 18) ; plt.plot(df.V18)
def plot5():
    plt.subplot(5, 6, 5) ; plt.plot(df.V5) ; plt.subplot(5, 6, 19) ; plt.plot(df.V19)
def plot6():
    plt.subplot(5, 6, 6) ; plt.plot(df.V6) ; plt.subplot(5, 6, 20) ; plt.plot(df.V20)
def plot7():
    plt.subplot(5, 6, 7) ; plt.plot(df.V7) ; plt.subplot(5, 6, 21) ; plt.plot(df.V21)
def plot8():
    plt.subplot(5, 6, 8) ; plt.plot(df.V8) ; plt.subplot(5, 6, 22) ; plt.plot(df.V22)
def plot9():
    plt.subplot(5, 6, 9) ; plt.plot(df.V9) ; plt.subplot(5, 6, 23) ; plt.plot(df.V23)
def plot10():
    plt.subplot(5, 6, 10) ; plt.plot(df.V10) ; plt.subplot(5, 6, 24) ; plt.plot (df.V24)
def plot11():
    plt.subplot(5, 6, 11) ; plt.plot(df.V11) ; plt.subplot(5, 6, 25) ; plt.plot(df.V25)
def plot12():
    plt.subplot(5, 6, 12) ; plt.plot(df.V12) ; plt.subplot(5, 6, 26) ; plt.plot (df.V26)
def plot13():
    plt.subplot(5, 6, 13) ; plt.plot(df.V13) ; plt.subplot(5, 6, 27) ; plt.plot(df.V27)
def plot14():
    plt.subplot(5, 6, 14) ; plt.plot(df.V14) ; plt.subplot(5, 6, 28) ; plt.plot (df.V28)
def plot15():
    plt.subplot(5, 6, 29) ; plt.plot(df.Amount)
```

```
<Figure size 1080x864 with 0 Axes>
```

In [16]:

```
start_time = datetime.now()

try:
    _thread.start_new_thread(plot1,())
    _thread.start_new_thread(plot2,())
    _thread.start_new_thread(plot3,())
    _thread.start_new_thread(plot4,())
    _thread.start_new_thread(plot5,())
    _thread.start_new_thread(plot6,())
    _thread.start_new_thread(plot7,())
    _thread.start_new_thread(plot8,())
    _thread.start_new_thread(plot9,())
    _thread.start_new_thread(plot10,())
    _thread.start_new_thread(plot11,())
    _thread.start_new_thread(plot12,())
    _thread.start_new_thread(plot13,())
    _thread.start_new_thread(plot14,())
    _thread.start_new_thread(plot15,())
except:
    print("Unable to start thread")
plt.show()
end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

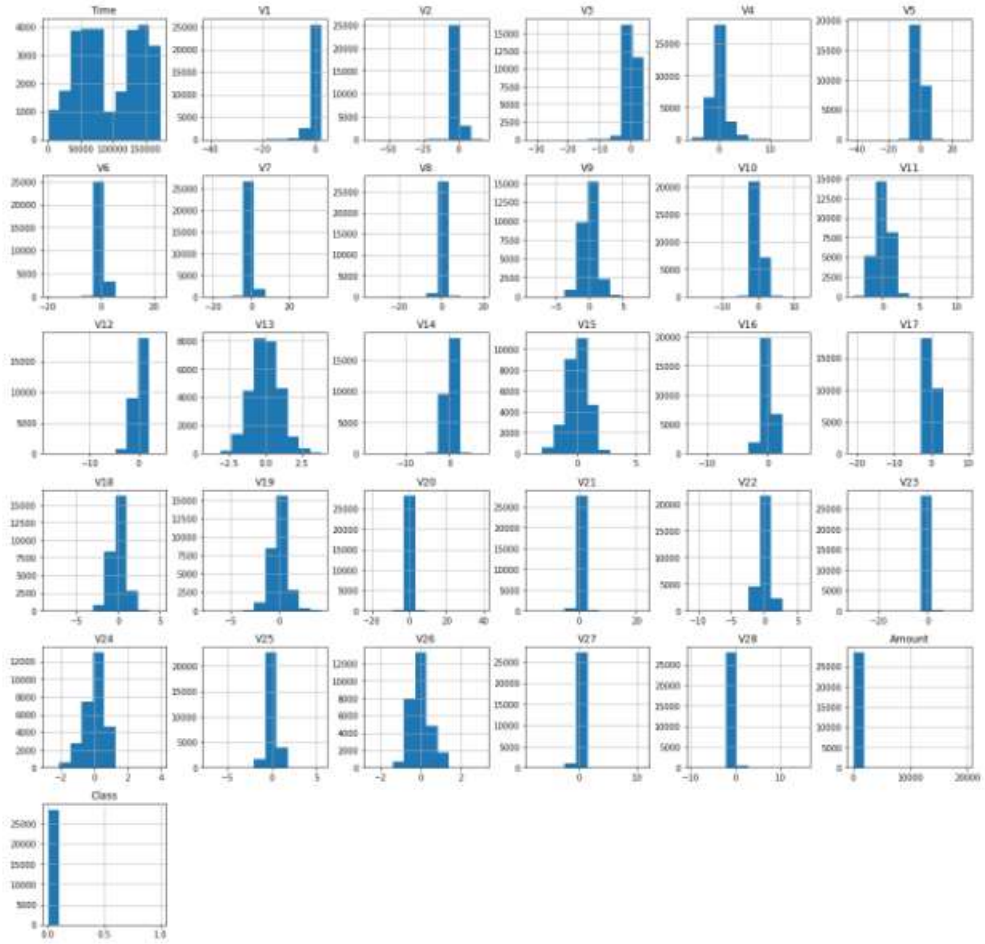


```
Duration: 0:09:35.130959
```

Plotting 10% of dataset for better visualization

```
In [60]:  1 data=data.sample(frac=0.1,random_state=1)
          2 print(data.shape)
          3
```

```
(28481, 31)
```



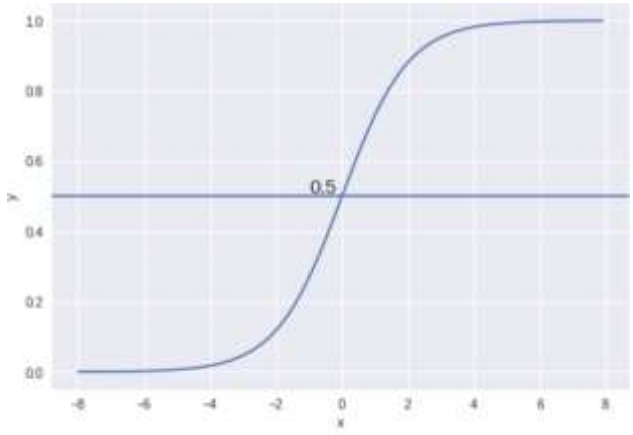
PART 2 – Training Single Neural Network using Data Parallelism among threads

Intuition:

A single unit parallel Neural Network is also known as a Neuron or A Logistic Regression Unit. Logistic Regression is a type of regression which is used when the target variable is categorical. In our case, the target variable is a categorical one having two possible values such that '1' stands for fraud cases and '0' stands for not fraud cases. Therefore a binary logistic regression model is perfectly suited in this case.

In simple terms, a logistic regression model has the same hypothesis as a linear regression model, but with a sigmoid activation function applied to it such that the output values strictly

ranges from 0 to 1, unlike the unbounded values predicted in linear regression. This can be clearly observed through the sigmoid graph which has a range of 0 to 1.



sigmoid function

Thus the logistic regression model in our case can be summarized as:

Model

$$\text{Output} = 0 \text{ or } 1$$

$$\text{Hypothesis} => Z = WX + B$$

$$h\Theta(x) = \text{sigmoid}(Z)$$

Where, Z refers to the output class predicted by the model, X is the input features matrix, W is the weights assigned to the features and B is the bias terms added.

Also, in this model, we have added the threshold as 0.5 as can be seen from the sigmoid graph. This means if the outcome of sigmoid(Z)>0.5, the predicted class is 1 otherwise it is 0.

For deriving the above weights and biases, we have defined the following cost function for our model and minimized the cost function using gradient descent approach iteratively till a stable minimized value is achieved.

Cost Function

Like Linear Regression, we will define a cost function for our model and the objective will be to minimize the cost.

The cost function for a single training example can be given by:

$$cost = \begin{cases} -log(h(x), & \text{if } y = 1 \\ -log(1 - h(x)), & \text{if } y = 0 \end{cases}$$

We can combine both of the equations using:

$$cost(h(x), y) = -ylog(h(x)) - (1 - y)log(1 - h(x))$$

The cost for all the training examples denoted by $J(\theta)$ can be computed by taking the average over the cost of all the training samples

$$J(\theta) = -\frac{1}{m}\sum_{i=1}^{m}[y^i log(h(x^i)) + (1 - y^i)log(1 - h(x^i))]$$

where ***m*** is the number of training samples.

Implementation in code using Data Parallelism:

The cost function used to compute the cost after each iteration and the gradient function used to calculate the gradient or slope of cost function using Gradient Descent Algorithm is shown below:

For minimising the cost function, we use fmin_tnc function from the scipy library. It can be used to compute the minimum for any function. It takes arguments as

```
def cost_function(theta, x, y):
    # Computes the cost function for all the training samples
    m = x.shape[0]
    total_cost = -(1 / m) * np.sum(
        y * np.log(probability(theta, x)) + (1 - y) * np.log(
            1 - probability(theta, x)))
    return total_cost

def gradient(theta, x, y):
    # Computes the gradient of the cost function at the point theta
    m = x.shape[0]
    return (1 / m) * np.dot(x.T, sigmoid(net_input(theta,   x)) - y)
```

- func: the function to minimize
- x0: initial values for the parameters that we want to find
- fprime: gradient for the function defined by ‘func’
- args: arguments that needs to be passed to the functions.

```
def fit(x, y, theta):
    opt_weights = fmin_tnc(func=cost_function, x0=theta,
                  fprime=gradient,args=(x, y.flatten()))
    return opt_weights[0]
parameters = fit(X, y, theta)
```

The train function is used to train the logistic regression model and accordingly find the weights and biases after training and store in the parameters variable. The train function calls the above defined fit function to minimise the cost function.

```
def train(dfs,idx):
    X = dfs[idx].iloc[:, :-1]
    y = dfs[idx].iloc[:, -1]
    X = np.c_[np.ones((X.shape[0], 1)), X]
    y = y[:, np.newaxis]
    theta = np.zeros((X.shape[1], 1))
    parameters = fit(X, y, theta)
    parameters_list.append(parameters)
    print(parameters)
```

We use **Data Parallelism** to Parallelize the above Algorithm.

Since we have a very large dataset, this is achieved by dividing the dataset among different threads and training the model parallelly till each of the parts converge. After individual convergence of the threads we take the average of the trained parameters from both the models. This significantly reduces the time taken by the models to execute and train as the training is done in parallel.

The following snapshot demonstrates the code where the dataset is divided into two parts and the train function is called using two threads simultaneously using the _thread library in python. To measure the time taken in training parallelly, timer has been started before training and ended after all the threads converges.

```
dfs = np.array_split(df, 2, axis=0)

```

```
from datetime import datetime
start_time = datetime.now()
try:
   _thread.start_new_thread( train, (dfs, 0, ) )
   _thread.start_new_thread( train, (dfs, 1, ) )
except:
   print("Error: unable to start thread")
end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

```
final_parameters = (parameters_list[0]+parameters_list[1])/2
print(final_parameters)
```

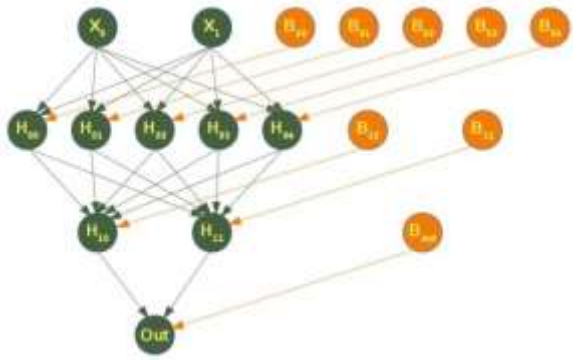
Next we define the predict and accuracy function to predict the results on the testing data after training completes and find the accuracy by comparing with the original results in the dataset.

```
def predict(x):
    theta = parameters[:, np.newaxis]
    return probability(theta, x)
def accuracy(x, actual_classes, probab_threshold=0.5):
    predicted_classes = (predict(x) >=
                         probab_threshold).astype(int)
    predicted_classes = predicted_classes.flatten()
    accuracy = np.mean(predicted_classes == actual_classes)
    return accuracy * 100
```

PART 3 - TRAINING MULTIPLE NEURAL NETWORKS USING FUNCTIONAL PARALLELISM AMONG THREADS

Intuition:

An Artificial Neural Network is also known as a Multi-Layer Perceptron, i.e., multi layers of the single logistic regression unit we saw in the previous section.



This is an example of a simple Artificial Neural Network Model which has 2 layers of intermediate logistic regression units, each with 5 and 2 neural nodes, 2 input parameters and one output variable.

In case of our dataset we have 30 input parameters representing the features of the users available from the dataset and one output variable denoting the class of user as fraud (‘1’) or not fraud (‘0’). The number of hidden layers and other hyper-parameters varies from model to model during training.

Model Hyperparameters are the various properties that govern the training process of the model. They include variables which determines the network structure (for example number of hidden layers and units) and the variables which determine how the network is trained (for example learning rate). Model Hyperparameters are set before training i.e., before optimizing the weights and the bias.

The different hyperparameters that we have used in our models are as follows:

1. Learning Rate or Alpha

If the learning rate is too smaller than optimal value, it will take a much longer time (hundreds or thousands) of epochs to reach a preferred ideal state. If the learning rate is much larger than optimal value, then it would overshoot the ideal state and the algorithm might not converge. A reasonable and universally accepted starting learning rate = 0.001. In our models we have varied the learning rates from 0.001 to 0.01.

The above graph demonstrates the relation between the variation of learning rate and training errors over the epochs.

2. Number of Neurons and hidden layers

The number of hidden units is the main measure of model’s learning capacity.

For a simple function, it might need fewer number of hidden units. The more complex the function, the more learning capacity the model will need.

Slightly more number of units then optimal number is not a problem, but a much larger number will lead to overfitting (i.e. if you provide a model with too much capacity, it might tend to overfit, trying to “memorize” the dataset, therefore affecting capacity to generalize)

For our models we have kept 2 hidden layers in each model and varied the number of neurons from 5 to 20 for these models.

3. Activation Function

In a neural network, numeric data points, called inputs, are fed into the neurons in the input layer. Each neuron has a weight, and multiplying the input number with the weight gives the output of the neuron, which is transferred to the next layer.

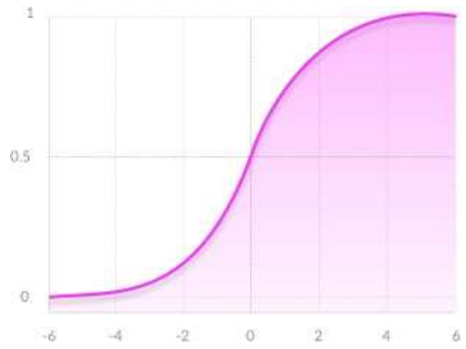
The activation function is a mathematical “gate” in between the input feeding the current neuron and its output going to the next layer. It can be as simple as a step function that turns the neuron output on and off, depending on a rule or threshold. Or it can be a transformation that maps the input signals into output signals that are needed for the neural network to function.

The above diagram explains the role of an activation function in a neural network model training process.

For our models we have used three types of non-linear activation functions:

- **Logistic or Sigmoid Activation Function**

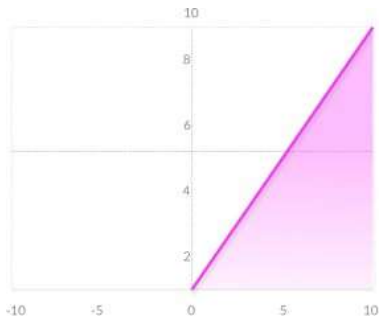
  - It is a function which is plotted as '**S**' shaped graph.
  - **Equation :**
    $A = 1/(1 + e^{-X})$
  - **Nature :** Non-linear. Notice that X values lies between -2 to 2, Y values are very steep. This means, small changes in x would also bring about large changes in the value of Y.
  - **Value Range :** 0 to 1
  - **Uses :** Usually used in output layer of a binary classification, where result is either 0 or 1, as value for sigmoid function lies between 0 and 1 only so, result can be predicted easily to be ***1*** if value is greater than **0.5** and ***0*** otherwise.



- **RELU or REctified linear Unit**

RELU stands for Rectfied Linear Unit. It is the most widely used activation function chiefly implemented in hidden layers of Neural network.

  - **Equation :-** ***A(x) = max(0,x)***. It gives an output x if x is positive and 0 otherwise.
  - **Value Range :-** [0, inf)
  - **Nature :-** non-linear, which means we can easily backpropagate the errors and have multiple layers of neurons being activated by the ReLU function.
  - **Uses :-** ReLu is less computationally expensive than tanh and sigmoid because it involves simpler mathematical operations. At a time only a few neurons are activated making the network sparse making it efficient and easy for computation.



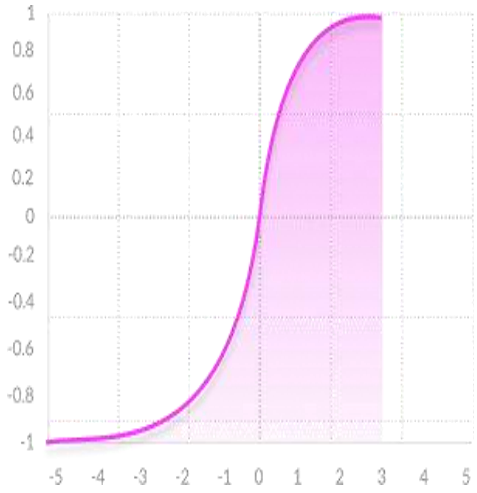
- **Hyperbolic tangent or tanh**

3). Tanh Function :- The activation that works almost always better than sigmoid function is Tanh function also knows as **Tangent Hyperbolic function**. It's actually mathematically shifted version of the sigmoid function. Both are similar and can be derived from each other.

- **Equation :-**

$$f(x) = \tanh(x) = 2/(1 + e^{-2x}) - 1$$
OR
$$\tanh(x) = 2 * \text{sigmoid}(2x) - 1$$

- **Value Range :-** -1 to +1
- **Nature :-** non-linear
- **Uses :-** Usually used in hidden layers of a neural network as it's values lies between **-1 to 1** hence the mean for the hidden layer comes out be 0 or very close to it, hence helps in *centering the data* by bringing mean close to 0. This makes learning for the next layer much easier.



In this part of the project we have used an inbuilt python library called MLPClassifier to build simple artificial neural networks.
Here we build a total of 10 neural network models by varying the hyperparameters and compared them in terms of accuracy to find the best model amongst them and hence the best working combination of hyperparameters.
This is an example of functional parallelism where we execute the same function simultaneously for all the threads.

IMPLEMENTATION CODE OF TRAINING MULTIPLE NEURAL NETWORKS IN PARALLEL

The training function using the MLPClassifier library for the neural netwrok model is shown below.

```
1  def train(tid, alpha_value, acti, layers):
2      clf = MLPClassifier(alpha=alpha_value, activation=acti, hidden_layer_sizes=(layers, 2),early_stopping=True, random_state
3      clf.fit(X_train, y_train)
4      result = clf.predict(X_test)
5      accuracy = np.mean(result == y_test)
6      accuracy = round ((accuracy*100),2)
7      print ("Accuracy from Thread id: ", tid, " model = ", accuracy ,"%")
```

The MLPClassifier takes as input the alpha value or the learning rate, the number of hidden layers and the activation function. It returns a classifier object based on these parameters which is used in the train function to fit the model, predict the results on the test data and return the model accuracy.

Firstly we executed without using threads and the total execution time was coming 6 minutes 14 seconds.

```
start_time = datetime.now()
train (1, 0.001,'logistic', 65)
train (2, 0.001,'relu', 50)
train (3, 0.01,'logistic', 5)
train (4, 0.01,'relu', 5)
train (5, 0.01,'logistic', 10)
train (6, 0.01,'relu', 10)
train (7, 0.001,'tanh', 5)
train (8, 0.001,'tanh', 10)
train (9, 0.01,'tanh', 15)
train (10, 0.01,'tanh', 20)
end_time = datetime.now()
print('Duration: {}'.format(end_time - start_time))
```

```
Duration: 0:06:14.005047
```

To achieve functional parallelism on this train function, we start 10 threads and call the train function using all these threads at the same time but by varying the hyperparameters taken as input by the function as shown below. Total Run time was coming to be 5 minutes 31 seconds, 13% faster.

```
from datetime import datetime
start_time = datetime.now()
t1 = threading.Thread(train (1, 0.001,'logistic', 65) )
t2 = threading.Thread(train (2, 0.001,'relu', 50)  )
t3 = threading.Thread(train (3, 0.01,'logistic', 5) )
t4 = threading.Thread(train (4, 0.01,'relu', 5)  )
t5 = threading.Thread(train (5, 0.01,'logistic', 10))
t6 = threading.Thread(train (6, 0.01,'relu', 10) )
t7 = threading.Thread(train (7, 0.001,'tanh', 5) )
t8 = threading.Thread(train (8, 0.001,'tanh', 10) )
t9 = threading.Thread(train (9, 0.01,'tanh', 15) )
t10 = threading.Thread(train (10, 0.01,'tanh', 20) )

try:
    t1.start()
    t2.start()
    t3.start()
    t4.start()
    t5.start()
    t6.start()
    t7.start()
    t8.start()
    t9.start()
    t10.start()
except:
   print("Error: unable to start thread")


end_time = datetime.now()

print('Duration: {}'.format(end_time - start_time),flush=True)
```

```
Duration: 0:05:31.751531
```

CONCLUSION

As usage of credit cards become more and more common in every field of the daily life, credit card fraud has become much more rampant. To improve security of the financial transaction systems in an automatic and effective way, building an accurate and efficient credit card fraud detection system is one of the key tasks for the financial institutions. In this study, 13 classification methods were used to build fraud detecting models. The work demonstrates the advantages of applying the data mining techniques including ANN and LR to the credit card fraud detection problem for the purpose of reducing the bank's risk. The results show that the proposed LR classifiers outperform ANN classifiers in solving the problem under investigation. However, as the distribution of the training data sets become more biased, the performance of all models decrease in catching the fraudulent transactions. The accuracy observed by Logistic Regressor Classifier was 99.85 and accuracy observed by Artificial Neural Network was 94.5. The reason for this maximum accuracy was the dataset. It contained minimal fraud values compared to the large non-fraud values. As by visualisation, we observed there were 284315 non-fraud values compared to just 492 fraud values.

We have also tested our models to some of the test values and it results in desired output. The fact that we have used parallelism in our training models helped us in training the models faster. Also, we had an insight on how the threads work, how the hyperparameters work and what is the ideal amount of parallelism that should be applied in order to achieve maximum efficiency. Since our dataset contained 31 columns, parallelism that we used in visualising the dataset helped us to reduce time of compilation and helped us to visualise more number of plots concurrently thus saving the time providing with desired outputs. All results have been obtained with a unique data set, comprising all transactions managed during two years by a major Spanish bank and including more than 180 million operations.

FUTURE ENHANCEMENT

As a future work, instead of making performance comparisons just over the prediction accuracy, these comparisons will be extended to include the comparisons over other performance metrics, especially the cost based ones. For example we could use different cost metrics like log likelihood or Adams, etc to evaluate different models in parallel.

Algorithms such as genetic algorithm can be applied and can be used to test the accuracy. Also, the dataset can be improved by making it large, taking from a different real time bank. This will help us to train our models more efficiently and with more accuracy.

Different dataset of real time different banks can be taken and they can be compared among themselves and this visualisation can be used to find out which parameters are more likely to be varied if fraud is happening. So that the banks can take care or take some measures to prevent it.

Moreover to make the above study accessible to the general users, we could develop this to a front-end user application, provided we know the name of the variables in the above dataset, which seemed to be a limitation in this project. If provided with that this project could be converted to a user interface for the users to make online predictions using personal or customized data.