CRIME PREDICTION IN GHANA USING MACHINE LEARNING OPTIMISED FEATURES OF URBANISM

BY

KOBENA BADU ENYAM AND DESMOND HAMOND

March, 2022

ABSTRACT

With the continuous phenomenal rise in violent and non-violent crimes in Ghana, it's impacts on society are becoming more pronounced and severe. Law enforcement agencies are therefore poised to find new and modern approaches to improve crime analytics in order to provide optimal protection for their respective communities.

Thanks to the availability of large curated data this thesis investigates the collinearities between urban features and crime occurrences and how they can be used for crime prediction models. A detailed quantitative study on the conventional and artificial intelligence techniques such as such as Support Vector Regression, Random Forest Regression, Decision tree and others has been conducted to critically establish a gap for a thesis. Also, deep learning techniques and algorithms were assessed to provide a holistic investigation into the problem.

A Deep Convolutional-Dense hybrid denoising autoencoder network was developed for data imputation and subsequently, a Transformer Architecture is proposed in this thesis to develop a robust crime prediction model to assist law enforcement agencies in Ghana for optimized sharing of security resources across the country to reduce crime rate in Ghana. The model in the context of crime intelligence analysis and law enforcement has the potential to improve crime analytics and decision making.

From the experimental results, the Transformer architecture had Mean Absolute error of 1.01% as compared to the Support Vector Machine, Random Forest Regressor and the Multi Output Random Forest Regressor by effectively utilizing geospatial information provided in longitude latitude form.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF ABBREVIATIONS

CHAPTER ONE

INTRODUCTION

1.1 Background of the Study

What causes an individual to exhibit criminal behavior?", this age-old question has attracted universal attention and interest. The UN Secretary General's report of 2005, identified crime as of one the major challenges in attaining the Millennium Development Goals (MDGs) as it serves as potential threat to global peace and security (Annan, 2005). Crime without a doubt affects economies negatively, from increasing emigration rates to brain drain to reducing foreign direct investment (Aning, 2006). Early psychologists identified an individual's social environment and biological traits as being the main cause of criminal behavior (*Routine Activity Theory - Miró - - Major Reference Works - Wiley Online Library*, n.d.). Later research has indicated that socioeconomic and political developments and polices in a particular geographical area greatly contributes to the insurgence of crime (Oteng-Ababio et al., 2016). Changing demographics of geographical areas also plays a vital role in identifying the underlying cause of recent crime trends (Bogomolov et al., 2014).

A report published in 2021 by the Bureau of Public Safety brought to light a 40.8% surge in violent crimes in Ghana's as compared to crime incidences in 2020. From the report the Ashanti Region recorded the highest crime incidents and also pointed out certain urban features such as demographics, geography, socio-economic challenges and political development that immensely contributes to surge in crime in the region(*Ghana Public Safety and Crime Report – FIRST HALF 2021*, 2021). Measures have been played down to reduce the upsurge of crime including: fixing bad roads, fixing street lights, equipping the police, tightening regulation on inflows and movement of arms, increasing police visibility and encouraging electronic banking (Dogbevi, 2018). Crime analysis based on historical data is an unexplored solution yet to be implemented in Ghana.

Crime analysis is a field that is proving very useful in identifying crime areas and trends based on historical crime data, that has not been adopted yet by security agencies in Ghana. Crime analysis involving exploring population demographics data, income levels of the populace, the Gross Domestic Product growth of the country and literary rate of the population. Crime analysis is usually performed using data mining techniques (Sivanagaleela & Rajesh, 2019). Data mining techniques involves various methodologies utilized to identify trends and patterns in

large data to make meaning decisions for the future. Data mining techniques include: association- for discovering patterns with variables in compiled data, classification- for classification of data variables into classes or groups, clustering- segmentation of data into categories for identification of correlating results, and decision trees- a tree like model for decision support for a given available data(Yahya & Osman, 2019).

Several crime prediction models have been developed several research scientists and data analysts. Conventional and machine learning techniques that have been proposed by authors include: logistic regression, Gaussian processes, Support Vector Machine, Autoregressive Integrated Moving Average (ARIMA), fuzzy logic, extreme Gradient Boosting, Naïve Bayes and decision tree. These machine learning models usually have lower prediction accuracy and are only efficient for spatial and temporal data analysis and are usually computationally overburdened when urban indicators are also analyzed. Deep learning architectures such as: Convolutional Neural Network (CNN), Long Short-Term Memory (LSTM), Recurrent Neural Network (RNN), K-Nearest Neighbors (KNN) and the Transformer Architecture have been introduced to mitigate the demerits of conventional models and give higher accuracy.

In this thesis, The Transformer Architecture, which is the most efficient modern artificial intelligence usually applied for natural language processing because of its ability to analyze large sets of data and making meaning out of it collectively due its efficient large memory size has been applied to develop a model for crime prediction for effective operations of the Ghana Police Service.

1.2 Problem Statement

The current increase in crime rate in Ghana poses numerous negative impacts on the Ghanaian society and the country as a whole. The negative impacts include death and disability which could deprive families of breadwinners in effect degrading the quality of life for family that remain, fear and panic as well as victimization of certain ethnic groups which impedes national development and loss of potential investors for industrial growth. According to a United Kingdom Home Office report in 2015, increase in crime rate results in direct and indirect financial losses on a country. The cost incurred include: cost in anticipation of crime (Defensive and insurance expenditure), cost as a consequence of crime (Property damaged, physical and emotional harm to the victim, health services), cost in response to crime (Police costs). Various

strategies to assess and mitigate these costs incurred have been proposed which could be limited due to the multifaceted nature and approach to tackling the problem. A machine learning model that takes into the linearities and non-linearities between urban features and crime occurrences to predict future crime trends for effective distribution of security resources is key to ensuring security institutions in Ghana are adequately prepared to curb criminal operations based on certain urban indicators.

1.3 Aim and Objectives

The ultimate goal of this thesis is to develop a machine learning model to analyze the linear and non-linearities between urban features and crime incidences to predict future crime trends for effective allocation of security resources amongst communities in Ghana.

To achieve this goal, the following sub-objectives have been set:

1. To explore the relationship, that is linear or non-linear, between crimes and urban features in Ghana.
2. To investigate the presence of possible collinearities among features of urbanism.
3. To implement optimized machine learning prediction algorithms suitable for the relationship between crimes and urban features in Ghana.
4. To evaluate optimized machine learning prediction algorithms suitable to data.
5. To specify a methodology for predicting crime data in Ghana based on urban features.

1.4 Significance of this Study

Ghana has seen a 57.34% increase in crime rates, with violent crimes such as assault and armed robbery currently pegged at 54.31% which has been a major concern. Ghana remains one of the fast-growing economies (IMF 2019; World Bank, 2021). However, this growth could come to a halt with the rampant and higher rates of violent crimes perpetrated in the country. It is evident that higher rates of crime and violence adversely impact investment, increase unemployment and inflation as well as deteriorate a country's GDP (Haroon and Jehan, 2020). The Ministry of Interior through the Ghana Police Service has launched a series of programs to help combat crime including: equipping the Crime and Combat Unit to tackle the issue of armed robber, expanding the service and populating communities with friendly police personnel

bringing policing closer to the doorsteps of Ghanaians and retraining the Motor Transport and Traffic Directorate (MTTD) with the state-of-the-art equipment to ensure effective traffic management (2022 Annual Budget Ministry of interior). The social interventions and financial commitment by the government has not proven to be effective in dwindling the crime rate in the country. Scientifically, Various strategies and methodologies to assess and mitigate this emerging problem have been proposed which can only go far enough due to the multi-objective nature of the problem. The project is therefore justified on these two arguments. First, it is highly relevant for a country such as Ghana to have its crimes modelled and predicted for better policy implementations and secondly, there is a clear absence of machine learning modelling of crimes based on urban features in Ghana. A suitable computational model, which accurately analyses the relationship between urban features and crime incidences detailed in the demographics of the population statistics of communities in a particular country is key to ensuring the effective operation of police personnel and distribution of police resources. With the current modernization of statistical tools and methods, population census data serve as huge statistical data mines which more or less are difficult to consolidate into proper security system and policy making decisions. By leveraging on the power of Machine learning (ML) or Artificial Intelligence (AI) like the encoder-decoder framework and the transformer architecture, a data-driven approach can be realized to harness these readily available census data to dynamically model crime trends while keeping track of parameter evolution over time. Such an algorithm can be harnessed and relied on for optimal grid management decisions. Also, indirect constraints such as operational, economic and social benefits can be quantified and easily integrated into the algorithm without complicated modifications, hence providing the Ghana Police Service and other security agencies, the flexibility of taking more holistic operational decisions without directly compromising security and optimized distribution of police resources.

1.5 Limitations of this Study

Acquisition of Ghana's population census data and crime statistics was a major stumbling block in undertaking this thesis as a report on complete population census and crime data was unavailable. The population census data for undertaking this thesis was a little noisy with incomplete columns which could increase the error of prediction.

1.6 Research Questions

1. How can population demographics provide insight into crime patterns in Ghana?
2. How can modern artificial intelligence algorithms be applied for effective crime analysis?
3. Can an artificial intelligence model tailored for the Ghana Police Service accurately predict future crime occurrences for proper distribution of Police resources?

1.7 Thesis Organization

This thesis is organized as follows:

Chapter 2 provides an overview crime prediction tools, techniques and algorithms that have been explored for crime prediction. Firstly, this chapter introduces the step by step processes involved in crime prediction. After that, a detailed literature review of various crime prediction issues and identified research gaps are discussed. Then, traditional and conventional machine learning models were reviewed. Finally, novel deep learning architectures were also reviewed to introduce the methodology of the thesis.

Chapter 3 provides the theoretical background, mathematical equation and theorems used in this research. The Deep Convolution Neural Network Architecture- input representation, convolution layer, padding, linear sub sampling layers, flatten layer, fully connected output layer (FC layer), Parameters Initialization, Activation functions (Non-linear), Dropout Layer (Regularized), loss functions, Optimizers, Gradient descent, signal denoising, Overview of Encoder-Decoder Network, Undercomplete Autoencoders and the transformer Architecture.

Chapter 4 presents the methodology of the thesis including the description of input data, implementation of the encoder-decoder architecture, implementation of the transformer architecture and code platform for model development.

Chapter 5 concludes the work done for this research and highlights the main findings. The possible future direction of further research is also presented in this chapter.

CHAPTER TWO

LITERATURE REVIEW

2.1 Introduction

In this chapter, firstly, an overview of the procedures to be taken when undertaking crime prediction modeling, data mining techniques, review of conventional and artificial intelligence methodologies utilized for crime prediction and as well as deep learning architectures for crime prediction.

2.1.1 Overview of Crime Prediction Modelling

To effectively model crime several sequential steps have to be taken. The including gathering historic crime data of a particular geographical area, preprocessing of crime data, model building with the appropriate software tools and theories, testing, tuning and training of model and deploying of the model for prediction.

I. Data Gathering.

First step in crime analysis involves gathering together crime data from various sources such as annual police reports, demographics information, police arrest records, previous investigation files, criminal background information, bank accounts and credit card statements, travel and flight itineraries and photographs (Ozgul et al., 2011). (Chen et al., 2015) utilized Twitter data emanating from the Chicago Twitter Community for a particular period in time and weather data for Chicago for a particular period. (Ivan et al., 2017) obtained crime data from the UCI machine learning repository with 128 attributes and 1994 instances. (Alves et al., 2018) utilized homicide data from the informatics department of the Brazilian Public Health System and 10 urban indicators of the Brazilian National Census in 2000. (Kupilik & Witmer, 2018) used spatio-temporal data obtained from media reports of violent events for sub-Saharan Africa from 1980-2012.

II. Data Preprocessing.

Data cleansing or cleaning involves the process of detecting, rectifying and removing inaccurate and corrupted information from the crime dataset or database. In addition, it involves recognizing inaccurate or unfinished parts of data, filling the missing ones, and removing dirty data. Usually, missing information or discrepancies identified are caused by human typing mistakes, by transmission, or by storing deficiency. This process saves training time, improves the

quality and efficiency of the dataset, and facilitates the identification the important relationships among elements in the exploration process. Moreover, it increases the prediction accuracy of the machine learning model. In this thesis, the missing data was filled in with the encoder-decoder architecture was effective crime modelling.

III. Model Building

Modelling of crime data can be seen as a classification problem where the machine learning model needs to classify the input features and predict to which type the crime belongs. Assuming that our predicted variable is categorical (e.g. no fraud and other types of fraud), binary classification cannot be applied in this case and categorical classification techniques are used to solve this problem. Most literatures explored Support Vector Machine, Random Forest, naive bayes, nearest neighbor, and decision tree algorithms to detect the status of the claims for insurance applications whether fraudulent or non-fraudulent.

IV. Training Testing and Tuning of Model

Training of the most is mostly done with a part or full data set to allow the machine learning algorithm learn the relationships between data features to generation the crime prediction model. Testing of the machine model is also executed with a portion of the data set not used for training or a whole new crime data set to test the accuracy of the model. For example: Gaurav et al, trained their model with 60% of the data set and trained with the remains 40%.

V. Prediction

Prediction is the same as classification or estimation, except that the records are classified according to some predicted future behavior or estimated future value. In a prediction task, the only way to check the accuracy of the classification is to wait and see. The primary reason for treating prediction as a separate task from classification and estimation is that in predictive modeling there are additional issues regarding the temporal relationship of the input variables or predictors to the target variable.

Any of the techniques used for classification and estimation can be adapted for use in prediction by using training examples where the value of the variable to be predicted is already known, along with historical data for those examples. The historical data is used to build a model that explains the current observed behavior. When this model is applied to current inputs, the result is a prediction of future behavior.

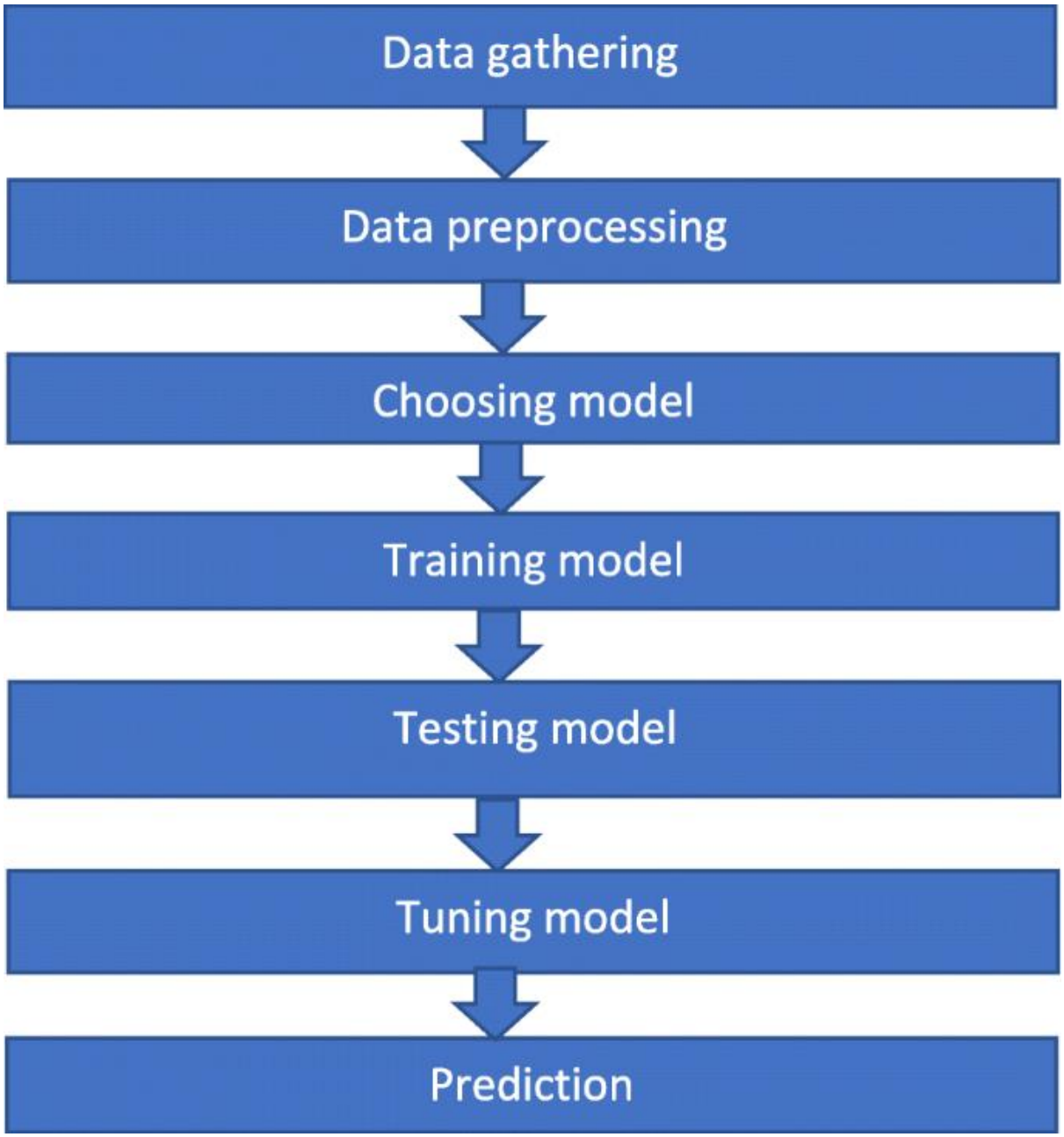


Figure 1 Step by step processes for crime prediction.

2.1.2 Review of Data Mining Techniques

Data mining, as we use the term, is the exploration and analysis of large quantities of data in order to discover meaningful patterns and rules. For the purposes of this book, we assume that the goal of data mining is to allow a corporation to improve its marketing, sales, and customer support operations through a better understanding of its customers. Keep in mind, however, that the data mining techniques and tools described here are equally applicable in fields ranging from law enforcement to radio astronomy, medicine, and industrial process control.

a. **Affinity Grouping and Association Rules**

The task of affinity grouping is to determine which things go together. The prototypical example is determining what things go together in a shopping cart at the supermarket, the task at the heart of market basket analysis. Crime prediction can use affinity grouping to plan the arrangement of crime into various types.

b. Classification

Classification, one of the most common data mining tasks, seems to be a human imperative. In order to understand and communicate about the world, we are constantly classifying, categorizing, and grading. We divide living things into phyla, species, and general; matter into elements; dogs into breeds; and people into races.

Classification consists of examining the features of a newly presented object and assigning it to one of a predefined set of classes. The objects to be classified are generally represented by records in a database table or a file, and the act of classification consists of adding a new column with a class code of some kind.

c. Estimation

Classification deals with discrete outcomes: yes or no; measles, rubella, or chicken pox. Estimation deals with continuously valued outcomes. Given some input data, estimation comes up with a value for some unknown continuous variable such as income, height, or credit card balance. In practice, estimation is often used to perform a classification task. A credit card company wishing to sell advertising space in its billing envelopes to a ski boot manufacturer might build a classification model that put all of its cardholders into one of two classes, skier or nonskier. Another approach is to build a model that assigns each cardholder a "propensity to ski score." This might be a value from 0 to 1 indicating the estimated probability that the

cardholder is a skier. The classification task now comes down to establishing a threshold score. Anyone with a score greater than or equal to the threshold is classed as a skier, and anyone with a lower score is considered not to be a skier.

d. Clustering

Clustering is the task of segmenting a heterogeneous population into a number of more homogeneous subgroups or clusters. What distinguishes clustering from classification is that clustering does not rely on predefined classes. In classification, each record is assigned a predefined class on the basis of a model developed through training on pre-classified examples. In clustering, there are no predefined classes and no examples. The records are grouped together on the basis of self-similarity. It is up to the user to determine what meaning, if any, to attach to the resulting clusters. Clusters of symptoms might indicate different diseases. Clusters of customer attributes might indicate different market segments.

2.2 Review of Conventional and Machine learning Models for Crime Prediction

Andriy et al developed an algorithm for identifying fictitious businesses based on Support Vector Machine Classification using data obtained from 1,100 companies in Ukraine. The Support Vector Machine Classification was modelled based on the linear, polynomial and radial approach. The training sample and testing sample show an accuracy of 100% and 99.7% respectively. However, the methods proposed are slow and computationally intensive. Yang et al developed that predicts the criminal tendencies of high-risk personnel based on behavioral data. The radial Vector Machine Classification algorithm. The radial function outperformed the rest. A testing accuracy polynomial and sigmoid kernel functions of the Support of 93.1% for the radial function can be further improved. K. Kianmehr et al proposed a Support Vector Machine based approach to predict crime location hot-spots based on crime and spatial data. The SVM based approach was compared with neural network-based approach and spatial auto-regression-based approach. The SVM approach outperformed the latter. However, the proposed approach performs poorly when there also a lot incomplete sections in the data available.

(Flaxman, n.d.) proposed a Gaussian Process (GP) in a hierarchical Bayesian modelling framework that seeks to predict future crime occurrences based on both spatial and time domain analysis. The mean relative forecasting error of the proposed was 4.4% for long term forecasting. Other geographic factors such as changing population demographics and social economic conditions were not considered for accurate modelling off crime prediction algorithm since

these factors affect the rate of crime occurrences. In (Kupilik & Witmer, 2018), developed 4 different data-driven Gaussian process (GP) for crime prediction. The first model only uses latitude and longitude and time as indicators for crime prediction, the second model adds socio-odemographic, geographic and climate as additional indicators, the third model only uses the second model's parameters but drops the latitude and longitude and the fourth model was a generalized crime prediction model. The second model outperformed all the models with a root meanm squared error closer to one due to the addition of other indicators. The GP model very well during training but performed poorly during testing. Two types of kernels: exponential kernel and the periodic kernel was proposed by (Perez & Wang, n.d.) for crime prediction. The models proposed performed badly with large Kullback-Leibler (KL) divergence. Finding an optimal solution for the periodic kernel becomes very challenging and computationally intensive as hyperparameters increase.

(Chen et al., 2015) developed a logistic regression model based on sentiment analysis and weather data. The model performed better than traditional logistic regression models with Kernel Density Estimation. The logistic regression model could not handle the non-linearities between variables in the data set used in this publication. (Antolos et al., 2013) developed a logistic regression model to predict the burglary activities happening in neighboring places with respect to the event epicenter. For small distances of 1Km, the prediction accuracy was 96.1% and decreased rapidly over large distances.

In (Vural & Gök, 2017) proposed a practical model based on Naïve Bayes to narrow down a crime suspect based on the crime, location, date and the acquaintances of the criminals. The proposed model had a prediction accuracy of 80% which could be further improved. (Ivan et al., 2017) proposed a decision tree model for crime analysis and crime prediction. The prediction accuracy achieved by the decision tree model was 94.25247% but this was because only 12 out of 128 features of the crime data obtained was used for prediction of the preprocessing stage. An apriori algorithm for trend and pattern identification and decision trees to predict future crime prone areas in India was proposed by (Sathyadevan et al., 2014) The prediction model could not predict the time in which future crimes would happen.

(Oh et al., 2021) proposed a random forest approach for crime prediction modelling based sociodemographic, behavioral and environmental indicators. The random forest model performed better than the logistic regression model. The average prediction accuracy for various scenarios simulated was 60% for the random forest model. The random forest algorithm was investigated

by (School of Computing, the University of Southern Mississippi, Hattiesburg, MS, U.S.A. et al., 2017) for crime prediction. An accuracy of 66.8% was obtained when the raw inconsistent demographic data was used and 65.71% when the missing data was filled in using random variable with pre-determined distribution. A decrease in accuracy with increasing data points, brings to light the overburdening of the random forest with increasing data points.

(Islam & Raza, n.d.) propose the Autoregressive Integrated Moving Average (ARIMA) model for crime prediction. A prediction of accuracy of 80% was achieved however the data features used for the modelling of the crime prediction algorithm were very few and limited to actually test the accuracy of the model. A time series model built on the ARIMA concept was introduced by (S.Vijayarani et al, 2021.) for crime prediction. The model achieved an accuracy of 91.67% however only a limited number of data features was considered.

(Hajela et al., 2020) proposed a spatio-temporal approach for crime prediction using the K-means clustering algorithm. Results showed that the K-means clustering approach greatly improves the performance of conventional approaches such as: naïve byes and decision tree networks for crime prediction. However the proposed method cannot work efficiently with incomplete data. (*Ndieb et al:*, 2019.) proposed an extreme gradient boosting model for automatic detection insurance fraud and classifying them into different fraud types.

2.3 Review of Deep Learning Models for Crime Prediction

(Stec & Klabjan, 2018) implemented a hybrid Recurrent Neural Network (RNN) and Convolutional Neural Network (CNN) architecture for crime prediction. The RNN was a good fit for temporal data analysis and the CNN was the best for the spatial aspects of data and achieved an accuracy of 75.6% and 65.3% for the Chicago and Portland. However a hybrid architecture such as the one presented in (Stec & Klabjan, 2018) requires huge compute resources. According to (Shamsuddin et al., 2017) the RNN is in principle a linear model but are usually difficult to train and usually does not coverage to one global solution due to differences in their initial weight set. For Convolutional Neural Networks the main disadvantage is long length of time it takes to train (Defferrard et al., 2016).

(Safat et al., 2021) implemented the Long-Term Short Term (LSTM) for crime prediction. The authors of this paper came out with a finding the LSTM technique is best suited for time series

classification and is a better alternative to Recurrent Neural Networks' vanishing gradient problem. However, the time series prediction of an downward trend in crime 2015 and the years beyond which was an inaccurate prediction because other factors contribute to crime that has to be factored in. (S.Wang et al, 2019.) proposed a spatio-temporal crime modelling approach using the Long Short Term Memory (LSTM) technique. The model achieved an 80% accuracy which could be improved since the data size for modelling was not huge.

(Sui et al., 2021) implemented an encoder-decoder network for crime prediction. The encoder portion was used for analyzing the short-term dynamics of event deployment and the time invariant correlation among events. The decoder portion does the prediction of future crimes events. The model was compared to other logistic regression models and other deep neural network topologies such as LSTMs and it outperformed all of them. The proposed encoder-decoder architecture could not identify the location of events without overfitting and only suitable for tasks with sparse observations. The Transformer Architecture, specifically invented for Natural Language Processing (NLP), has the advantages of overcoming the LSTM's context limitation using an Attention mechanism and greater throughput as inputs are processed in parallel with no sequential dependency, has yet been applied for crime data analysis(Singh & Mahmood, 2021). The Transformer Architecture outperforms all other deep learning architectures as it totally avoids recursion by processing data as a whole and learning relationships between words thanks to multi head attention mechanisms and positional embeddings (Han et al., 2021).

2.4 Summary

All the studies reviewed so far have demonstrated that crime data analysis is one of the key issues for security agencies and police departments across various countries. It has been discussed that attributes such as spatio-temporal features of crime data for a particular geographical area, population demographics and other social indicators are critical parameters when formulating crime prediction models. The merits and demerits of conventional artificial intelligence models such as support vector machine and others discussed above have been explored. Deep learning architecture have proven efficient in providing solutions to the disadvantages of

conventional artificial intelligence. The next chapter presents the theoretical basis for the encoder and decoder model and transformer architecture and software tools used for this research.

CHAPTER THREE

THEORETICAL BACKGROUND

3.1 What is AI and ML?

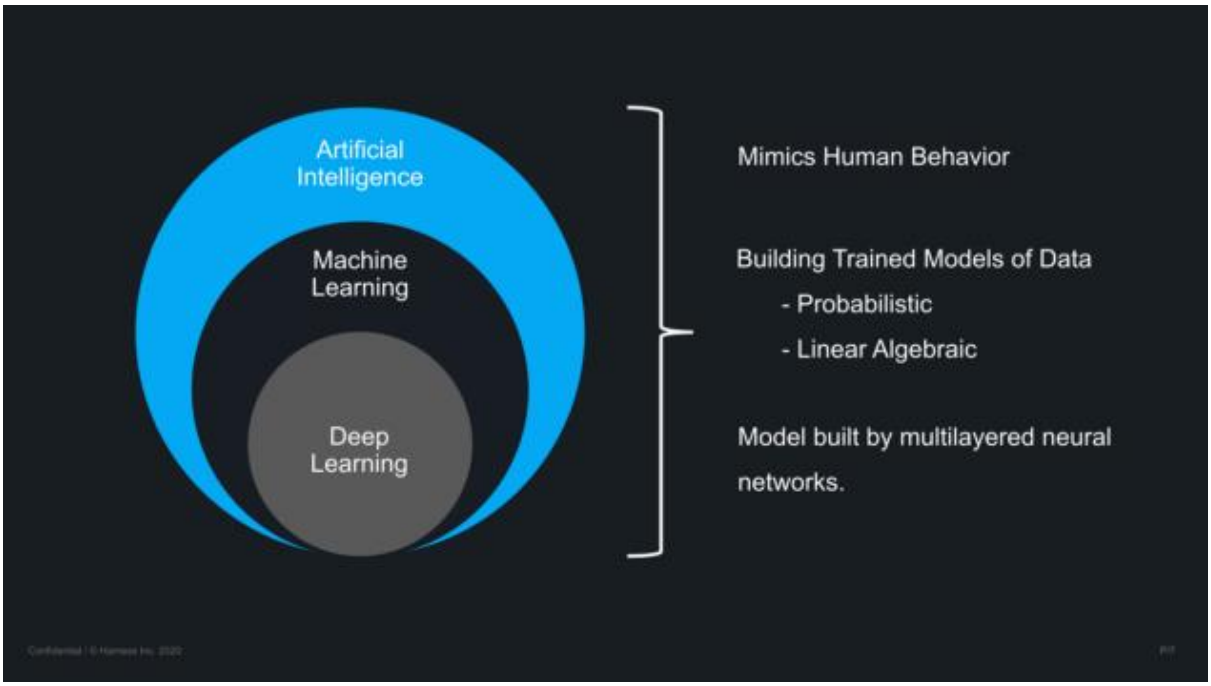


Figure 2 An overview of AI and ML

Artificial intelligence (AI) is a very large research field, where machines show cognitive capabilities such as learning behaviors, proactive interaction with the environment, inference and deduction, computer vision, speech recognition, problem solving, knowledge representation, perception, and many others (for more information, refer to this article: Artificial Intelligence: A Modern Approach, by S. Russell and P. Norvig, Prentice Hall, 2003). More colloquially, AI denotes any activity where machines mimic intelligent behaviors typically shown by humans. Artificial intelligence takes inspiration from elements of computer science, mathematics, and statistics. Artificial Intelligence is a technique for building systems that mimic human behavior or decision-making.

Machine Learning is a subset of AI that uses data to solve tasks. These solvers are trained models of data that learn based on the information provided to them. This information is derived from probability theory and linear algebra. ML algorithms use our data to learn and automatically solve predictive tasks. Machine learning (ML) is a subbranch of AI that focuses on teaching computers how to learn without the need to be programmed for specific tasks (for

more information refer to Pattern Recognition and Machine Learning, by C. M. Bishop, Springer, 2006). In fact, the key idea behind ML is that it is possible to create algorithms that learn from and make predictions on data. There are three different broad categories of ML. In supervised learning, the machine is presented with input data and desired output, and the goal is to learn from those training examples in such a way that meaningful predictions can be made for fresh unseen data. In unsupervised learning, the machine is presented with input data only and the machine has to find some meaningful structure by itself with no external supervision. In reinforcement learning, the machine acts as an agent interacting with the environment and learning what are the behaviors that generate rewards. A common technique is to utilize deep learning with reinforcement learning to derive relationships between features of a data set that may not otherwise be solved through human research.

Deep Learning is a subset of machine learning which relies on multilayered neural networks to solve these tasks. Deep learning (DL) is a particular subset of ML methodologies using artificial neural networks (ANN) slightly inspired by the structure of neurons located in the human brain. Informally, the word deep refers to the presence of many layers in the artificial neural network, but this meaning has changed over time. While 4 years ago, 10 layers were already sufficient to consider a network as deep, today it is more common to consider a network as deep when it has hundreds of layers. DL is a real tsunami for machine learning in that a relatively small number of clever methodologies have been very successfully applied to so many different domains (image, text, video, speech, and vision), significantly improving previous state-of the- art results achieved over dozens of years. The success of DL is also due to the availability of more training data (such as ImageNet for images) and the relatively low-cost availability of GPUs for very efficient numerical computation. Google, Microsoft, Amazon, Apple, Facebook, and many others use those deep learning techniques every day for analyzing massive amounts of data.

3.2 Deep Convolutional Neural Network Architecture

This field is aimed at enabling machines to visualize or perceive the world as humans do and even apply the knowledge for many tasks such as Image & Video recognition, Media Recreation, Natural Language Processing (NLP) and Image Analysis & Classification. With time,

Computer Vision advancements with Deep Learning has been perfected as they are primarily hinged on the Convolutional Neural Network algorithm.

The Convolutional Neural Network (CNN, or ConvNet) is a variant of deep neural networks, popularly applied in visual imagery analysis. Due to their shared-weights architecture and translation invariance nature, they are also called shift or space invariant artificial neural networks (SIANN), which implies that extracted features from a given input do not change despite the translation of the input by any small amount. CNNs are regularized versions of multilayer perceptron or fully connected neural networks where each neuron in one layer is connected to all neurons in the next layer. The nature of these networks makes overfitting data inherent. CNNs leverage the hierarchical pattern in data and construct more complex patterns using smaller and simpler patterns for regularization. Hence minimizing the complexity and scale of the network. The pattern of connectivity between neurons resembles the organization structure of the animal visual cortex. The response of individual cortical neurons is confined to a restricted region of the visual field called receptive field which partially overlap in order to entirely cover the visual field.

CNNs do not have a preconceived architecture, parameter selection methods or rules regarding the number of convolutional layers . They have three major design features namely, local receptive fields, weight sharing and spatial sub-sampling.

a. Local Receptive Fields

Each sample is represented with a row vector and encoding of a local structure is achieved by connecting a sub-vector of adjacent input neurons to a single hidden neuron located in the next layer, representing one local receptive field. This operation is called convolution and it gives the name to this type of network.

More information can be encoded by overlapping submatrices. For instance, let's suppose that the size of each single submatrix is 1xn and that those submatrices are used with an input representation of 1xm. Then we will be able to generate 1x (m-n) local receptive field neurons in the next hidden layer by sliding submatrices by only (m-n) positions before touching the borders of the input matrix. The size of each single submatrix is called *stride length*, and this is a hyperparameter that can be fine-tuned during the construction of CNNs.

Multiple feature maps can exist from one layer to another layer that learn independently from each hidden layer.

b. Shared Weights and Bias

Ideally, the input representation could just be fed into a Multi-Level Perceptron for classification purposes. Unfortunately, the following problems are associated with this technique.

In the case of binary images, the Multi-Level Perceptron (MLP) tend to (in most cases) show an average precision score when performing prediction of classes but poor or no accuracy when it when exposed to complex images which have pixel dependencies throughout.

Where the same feature is to be detected independently from the location where it is placed in the input image, the same set of weights and bias is used for all the neurons in the hidden layers. A such, each layer learns a set of position-independent latent features derived from the image.

Due to the reusability of weights and considerable reduction in the number of parameters involved, Convolutional Neural Networks can effectively capture the spatial and temporal dependencies in an image by the application of relevant filters. Thus, the architecture can be trained to comprehend the sophistication of the image better.

c. Spatial sub-sampling

Spatial sub-sampling is to down-sample the output of a convolution layer along both the spatial dimensions of height and width. This is to reduce the number of parameters to be learned by the network thus, reducing overfitting and thereby increasing the overall performance and accuracy of the network. This function is performed by the pooling layers.

CNNs require relatively less data pre-processing as compared to the other neural network variants. This means that the network learns the filters that would have been manually engineered in the other algorithms. CNNs leverage this independence from prior information and human effort in feature extraction over the other similar algorithms. Popular architectures of CNNs are available which provide key foundation for other algorithms such as; LeNet-5**,** AlexNet**,** VGG, and ResNet.

Multiple neural network layers with two different types of layers are typically alternated (i.e. the convolutional and pooling layers) forming a deep convolutional neural network (DCNN). Output stage (last stage) typically consists of one or more fully connected layers.

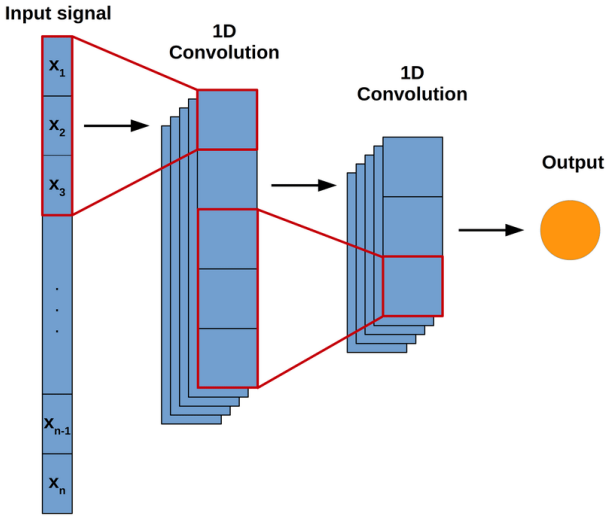


Figure 3 A typical Convolutional Neural Network (CNN).

Convolutional layers in a convolutional neural network attempt to summarize the presence of features in an input representation by systematically applying learned filters to input images in order to create feature maps. Stacking convolutional layers in deep models allows layers close to the input to recognize low-level visual features like lines, edges, gradient orientation and color, and deeper layers tend to learn more sophisticated visual features, like specific shapes or objects giving the network a complete or wholesome understanding of images in the dataset, similar to how animals would. Below is a breakdown of the convolutional neural network architecture.

3.2.1 Input Representation

The goal of CNNs is to reduce the input into forms which can be easily processed, without compromising on features which are critical for a satisfactory model output. This is especially necessary when designing an architecture capable of not only recognizing intrinsic features but is also scalable to large datasets. Increase in input dimensions result in increased computational cost. Thus, high dimensional inputs tend to demand more computational power hence crippling other computational processes.

3.2.2 Convolution Layer – The Filter or Kernel

It is responsible for carrying out the convolutional operation throughout the input representation.

1	0	1
0	1	0
1	0	1

Figure 4 A 3x3 filter.

Given an input dimension 50 (Height) x 50 (Breadth). The Kernel/Filter denoted by K (3, 3) is a matrix specifying the width and height of the 2D convolution window. The Kernel shifts 47 times with a stride length (1) every time performing a matrix multiplication operation between K and the portion of the image over which the kernel hovers. The filter traverses the entire width of the input image from right to left at the specified stride value after which it hops down with the same stride value to the beginning (left) of the image and repeats the process until the entire image is covered. The results are summed with the bias to generate a compressed convolutional feature output matrix.

3.2.3 Padding

By default, a convolutional kernel or filter begins at the left of the input with the left side of the filter sitting on the cells located to the far left of the input. The kernel is then stepped across the input one column at a time until the right-hand side of the filter is sitting on the far-right cell of the input.

Another approach is to apply a filter to the input such that each cell in the input is given an opportunity to be at the center of the filter. By default, this is not the case, as the cells on the edge of the input are only ever exposed to the edge of the filter. By starting the filter outside the frame of the input, it enables more interaction of the cells on the border with the filter thus, more features are detected by the filter, resulting in an output feature map that has the same shape as the input representation.

For example, in the case of applying a 3×3 filter to the 8×8 input matrix, we can add a border of one cell around the outside of the matrix. This has the effect of artificially creating a 10×10 input matrix. When the 3×3 filter is applied, it results in an 8×8 feature map. The added cell values could have a zero value that has no effect with the dot product operation when the filter is applied.

There are two types of results to the convolution operation;

a. Valid Padding

The convolved feature is reduced in dimensionality as compared to the input representation. Meaning that, the convolution is only computed where the input and the filter fully overlap. Therefore, the output is smaller than the input. For instance, if we perform the convolution operation on a 4x4 input with a 3x3 filter, a resultant matrix with reduced dimensions (2x2) is produced.

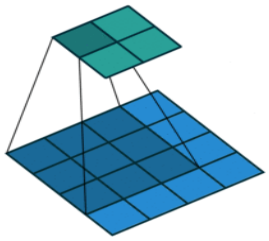


Figure 5 Valid Padding of a 4x4 input.

b. Same Padding

Here, the dimensionality of the output obtained is either the same size as the input or increased, for which the area around the input is padded with zeros. When we augment by padding the 5x5 matrix into a 6x6 matrix and then apply the 3x3 kernel over it, the resulting convolved matrix has a dimension of 5x5. Hence, the name *Same Padding*. Given a stride length of 1 with a padding of zeros, the input and output volume will always have the same spatial dimensions expressed as;

$$Zero\ Padding = \frac{(K-1)}{2},$$

where K is the filter size.

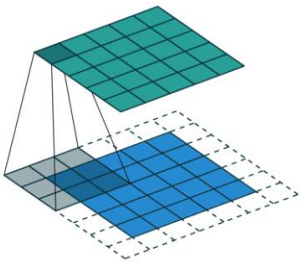


Figure 6 Same Padding of a 4x4 input.

The output size of any convolutional layer can be determined by the following expression;

$$0 = \frac{(W - K + 2P)}{S} + 1$$

where O is the output length or height, W is the input length or height, P is the padding, S is the stride and K is the filter size.

3.2.4 Linear Sub Sampling Layers

A pooling layer is a layer mostly added after the convolutional layer. Explicitly, after a nonlinearity known as activation function (e.g. ReLU) has been applied to the feature maps produced by a convolutional layer. The spatial contiguity of the output produced from one feature map and aggregated values of a submatrix squashed into a single output value synthetically describes the meaning associated with that physical region. This tends to summarize the output of a feature map. The limitation of the output feature map of convolutional layers (i.e. memorizing the precise position of features in the input is commonly addressed by downsampling (used in signal processing). Here, a lower resolution version of the input signal is generated while maintaining large or important structural elements, eliminating fine details that may not be useful to the task.

Changing the stride of convolution across the representation achieves downsampling but a more robust and common approach is to use a pooling layer. Pooling layers provide an approach to downsampling feature maps by summarizing the presence of features in sections of the feature map. Some common pooling methods include max-pooling and average pooling.

3.2.5 Flatten Layer

The flatten layer is an operator that typically unrolls the elements of a given input vector beginning at the last dimension. It reduces the dimensionality of a given input such that, the output dimension is a one-dimensional matrix or vector having a width or length equal to the total/combined population of all elements located in the initial input. For example, if flatten is applied to layer having input shape as (3, 3), then the output shape of the layer will be (1, 9).

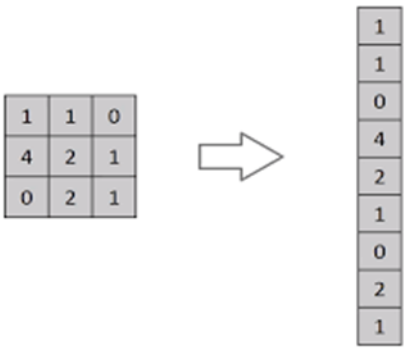


Figure 7 A 3x3 matrix flattened to a one-dimensional vector

3.2.6 Fully Connected Output Layer (FC Layer).

A common approach of learning non-linear mappings or function of the high-level features produced by the output of the convolutional layer is by adding a Fully-Connected layer. After a series of image conversions and feature extractions from our Multi-Level Perceptron, the final output feature map is then flattened into a column vector which is fed to a feed-forward neural network and backpropagation algorithm is applied to every training iteration. The model distinguishes between dominating and certain low-level features and using activation functions, as discussed above, the corresponding output behavior is produced.

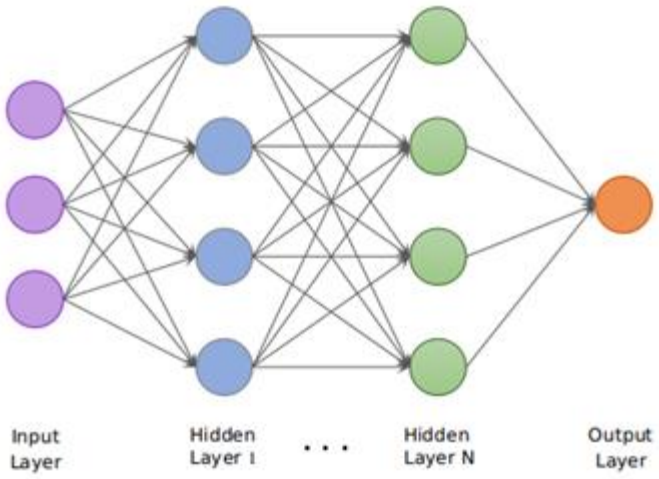


Figure 8 Fully connected deep feedforward neural networ(Defferrard et al., 2016)k

Each network neuron (except those in the input layer) is actually a sum of all its inputs; which are in fact the outputs from the previous layer multiplied by some weights. An additional term called bias is added to this sum. And a nonlinear function known as activation function is applied to the result.

These parameters (weights, bias) are exactly the numerical values that we'll try to adjust by training the network with an already labeled dataset, as in any other supervised Machine Learning problem. The final result will be a model built from that data, able to make predictions over future samples.

3.2.7 Parameter Initialization

To train a neural network from scratch (without a pre-trained model) parameter initialization is essential. It is typically proceeded as follows:

1. Weights are initialized randomly in order to breaking the network symmetry. This prevents all neurons in a layer from learning the similar attributes.
2. For weight initialization, the Xavier initializer(Defferrard et al., 2016) is used. This draws samples from a uniform distribution within [-limit, limit], where limit = sqrt(6 / (fan_in + fan_out)) (fan_in is the number of input units in the weight tensor and fan_out is the number of output units).
3. Biases are usually initialized to zero.

3.2.8 Activation Functions (Non-Linear)

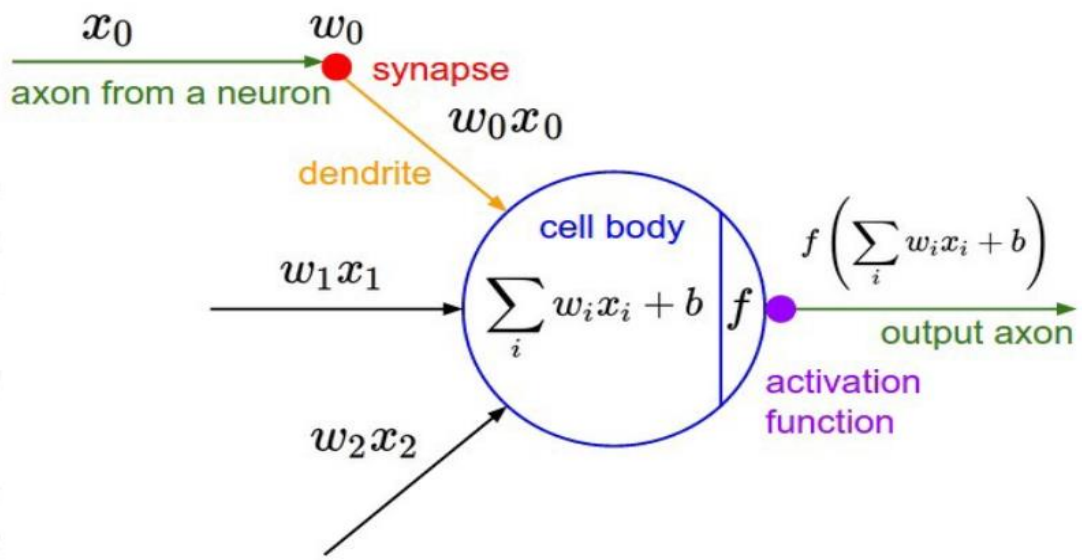


Figure 9 A Perceptron with activation function, f

In a simple neural network, function f(x) computes an output given x as inputs, w weights and bias, b. This output may be inherited as input to another layer or can be used as the final output of the network. The activation function is an inherent process without which the output signal becomes a simple linear function and neural network will act as a linear regression with poor learning capability. Activation functions are needed create models capable of processing complex real-world information such as image, sound, video and text. For the model to generalize well with a wide variety of data and to differentiate between the outputs, nonlinear activation functions are employed for adjusting weights and biases. These functions are generally monotonic (either entirely non-increasing or non-decreasing) and gradient functions.

a. Sigmoid or Logistic Activation Function

Sigmoid functions produce a return (response) value commonly monotonically increasing but could be decreasing. Sigmoid functions mostly produce a return value (y axis) in the range 0 to 1. sigmoid function is a type of activation function that squashes or limits the output to a range between 0 and 1, given a domain of all real numbers (-∞, +∞). This function is used in the prediction of probabilities.

The sigmoid function can be expressed mathematically as;

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

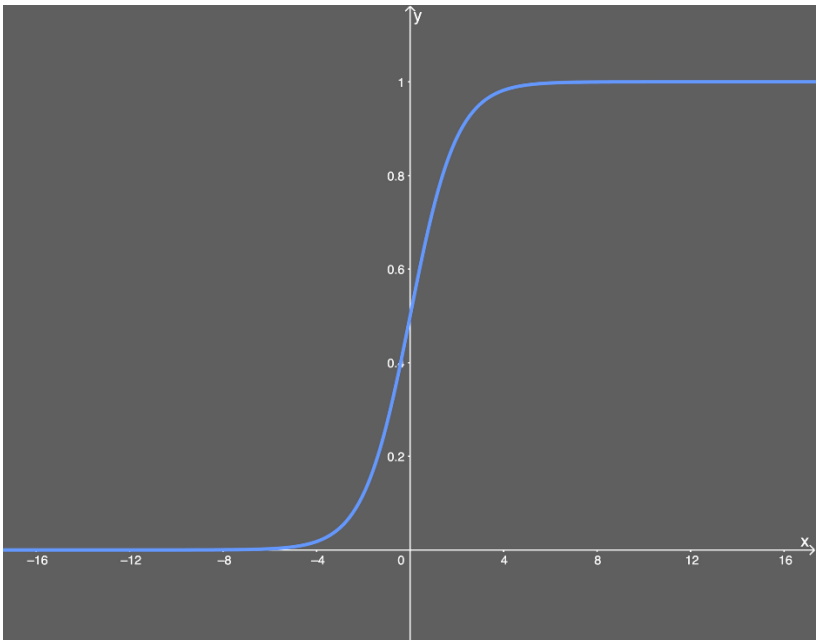


Figure 10 Sigmoid function graph.

Artificial neural networks learn by updating constituent weights and biases in the direction indicated by the gradients. A drawback associated with backpropagation is that the gradients can get too small, a problem called *vanishing gradients* [20]. When a network suffers from vanishing gradients, the weights seize to adjust and minimal or no learning takes place. On a more abstract level, deep neural networks multiply many gradients during backpropagation. From the graph above, it can be observed that towards the extreme ends of both axes, the derivative(gradient) values are very small and tend to converge to 0. Gradients closer to zero tend to drop the whole product. This initiates a chain reaction where further gradients drop. This phenomenon leads to the optimization algorithm or loss function (minimizes error) converging to a local minimum value instead of global minimum, hindering the performance of the model.

b. Softmax Activation Function

The softmax function is a more generalized form of the logistic or sigmoid activation function. It is commonly preferred in the output layer of deep learning models for multiclass classification. Expressed mathematically as;

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \ for\ i = 1, \dots, K\ and\ z = (z_1, \dots, z_k) \in \mathbb{R}^K$$

this function normalizes an input vector z of K real numbers, and it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. Thus, interpreting the inputs as probabilities. Larger input components will correspond to larger probabilities and vice versa. Softmax is often used in neural networks to perform probabilistic interpretation of classes by mapping the non-normalized output of a network to a probability distribution over predicted output classes [21]. Softmax also suffers from gradient vanishing problem. In the binary classification both sigmoid and softmax function produce the same results whereas, in the multi-class classification the Softmax function is used.

c. Rectified Linear Units (ReLU) Activation Function

Research has found out that ReLU activated layers supersede the popular Sigmoid and tanh functions because of improved speed (computational efficiency) without compromising on accuracy. It also helps to alleviate the vanishing gradient problem, where the lower layers of the network train very slowly because the gradient decreases exponentially through the layers. The ReLU is given by;

$$f(\alpha, x) = \lambda \begin{cases} \alpha(e^x - 1), & for\ \ x < 0 \\ x, & for\ \ x \geq 0 \end{cases}$$

That is, the output is positive, if the input is positive, and 0 otherwise. That characteristic prevents vanishing gradients since there is no gradient close to zero, which could diminish other gradients. The parameter α, controls the steepness of the line for x < 0(negative values of x) and is set to 0. Any value < 1.0 converts this activation into Leaky ReLU and setting it to 1.0 converts it into a Linear activation.

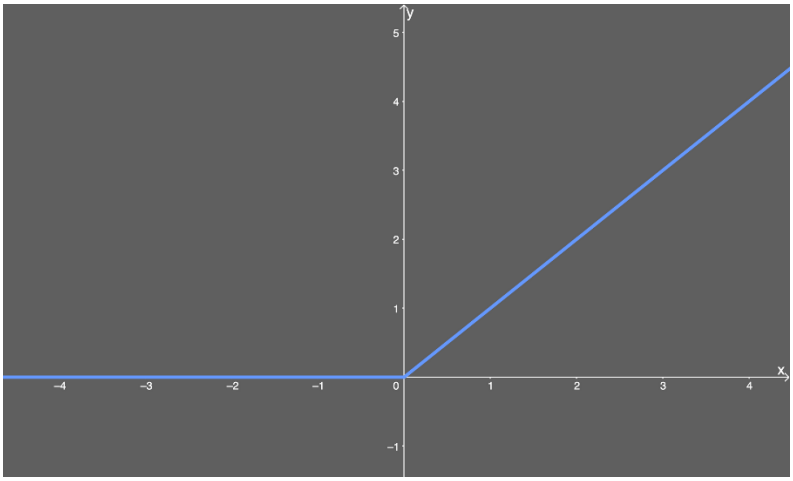


Figure 11 Rectified Linear Unit (ReLU) activation.

However, a major issue with ReLUs is that they can get stuck in a dead or inactive state [22]. That is, the change in weight is so large and the resulting z becomes so negligible in the next iteration such that the activation function is stuck at the left side of zero. Therefore, affected neuron is unable contribute to the learning of the network anymore, and its gradient stays zero. The power of the trained network is significantly handicapped especially when many neurons or cells get affected. This problem typically arises the network architecture is set with a high learning rate. Hence though of vanishing gradients has been dealt with, dying ReLU problem present itself.

Due to the added advantage of computational simplicity, a good approach is to change the left side of the function such that the ReLU never deactivates as shown below:

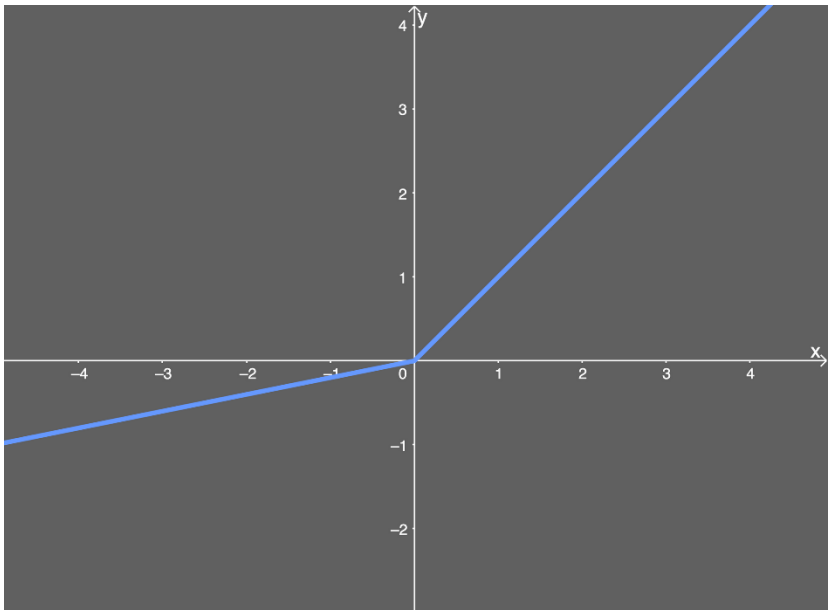


Figure 12 Leaky Rectified Linear Unit (Leaky ReLU) activation function.

From the figure above, a slope is now introduced on the left side, usually tiny (about 0.01), but it exists so learning always takes place and the neuron cannot die out by controlling the gradient. This function is known as the Leaky ReLUs.

d. Scaled Exponential Linear Units (SELU) Activation Function

SELU presents a tradeoff between the risk of dying neurons (ReLU) or a self-imposed risk vanishing gradients (leaky ReLU). SELU functions like ReLU for all z values greater than zero and the gradient approaches zero for all values less than zero, but SELU tends to introduce normalization. Neural networks may be normalized at different stages

1. **Input normalization**.

This involves transforming or scaling data values (usually between zero and one). This method, though not strictly applicable to neural networks, proves to be effective when used for certain tasks.

2. **Batch normalization.**

Here, values are transformed so that their mean their standard deviation between each layer of the network is zero and one respectively. One main advantage of this normalization is that it limits the values and prevents the occurrence of outliers.

3. **Internal normalization**

With this type of normalization, each layer maintains the mean and variance from the preceding layer. SELU leverages on this technique by adjusting the mean and variance [23].

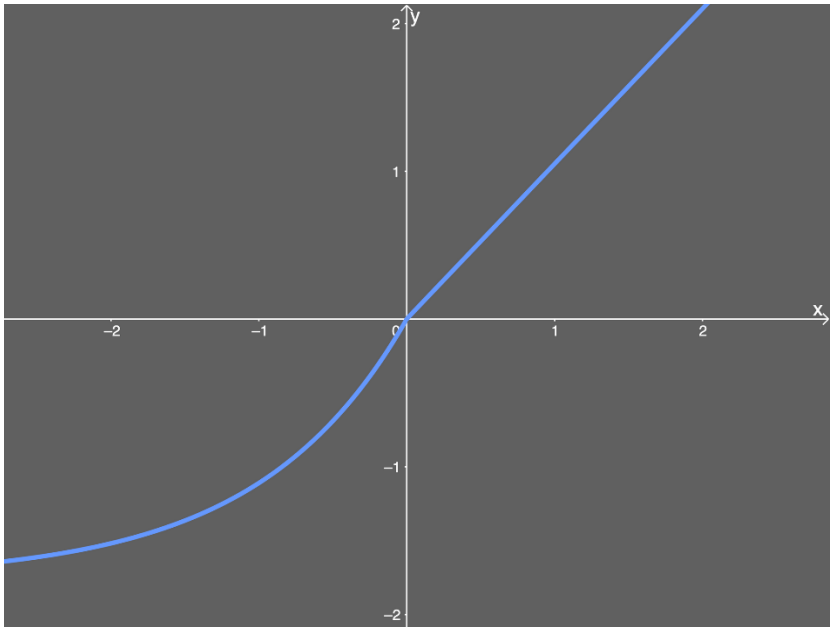


Figure 13 Scaled Exponential Linear Unit (SELU) activation function.

SELU activation function makes use of both positive and negative values for y in order to adjust the mean. The gradients are also used to adjust the variance. The activation function needs a region with a gradient larger than one to increase it.

$$selu = \lambda \begin{cases} x, & if\ x > 0 \\ \alpha e^x - \alpha, & if\ x \leq 0 \end{cases}.$$

SELU makes use of an added scaling parameter, λ, which increases or decreases the gradient subsequently as its value increases or decreases respectively. This centers the values with a fixed variance hence, eliminating vanishing gradients. For standard scaled inputs (mean 0, standard deviation 1), the values are α=1.6732~, λ=1.0507~ [24].

Advantages of SELU

1. Like ReLUs, SELUs make modeling of deep neural networks possible.
2. SELUs do not enter inactive state (cannot die).
3. SELUs, without any combination with other normalization, improve learning speed and accuracy.

e. Gaussian Error Linear Unit (GELU) Activation Function

The GELU activation function is describing a standard Gaussian cumulative distribution function. The GELU nonlinearity weights inputs by their percentile, rather than gates inputs by their sign as in ReLUs:

$$\text{GELU}(x) = 0.5x(1+\tanh(\sqrt{(2/\pi)}\ (x+0.044715x\text{^}3)))$$

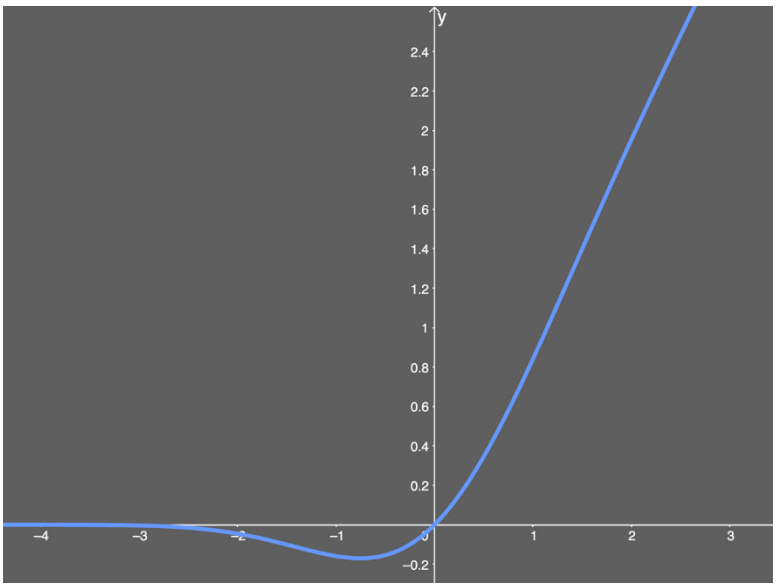


Figure 14 Gaussian Error Linear Unit (GELU) Activation Function

It consists of a negative coefficient, which shifts to a positive coefficient. The approximate derivative of this function can be expressed as:

$$\text{GELU}'(x) = 0.5\tanh(0.0356774x^3 + 0.797885x)$$
$$+(0.0535161x^3 + 0.398942x)\text{sech}^2(0.0356774x^3 + 0.797885x) + 0.5$$

This is effectively a linear combination of hyperbolic functions.

Advantages of SEGELULU

1. GELUs make modeling of deep neural networks possible.
2. GELUs do not enter inactive state (cannot die) however, unlike SELUs, it limits the extent of negative activations which may lead to poor convergence or a bad model.
3. GELUs, without any combination with other normalization, improve learning speed and accuracy.

3.2.9 Dropout Layer (Regularizer)

Small datasets are likely to drive a deep neural network into overfitting data. A situation whereby the network learns the detail and noise in the training data so well to the extent that, it performs poorly when exposed to unseen data (poor generalization). It is characterized by large weights in a neural network which are a sign of a more complex network that has overfit the training data. Various methods of circumventing this problem have been invented which tend to add more computational burden to models.

Dropout offers a remarkably effective regularization method without compromising computational power to reduce overfitting and improve generalization error in most, if not all deep neural network architectures [32].

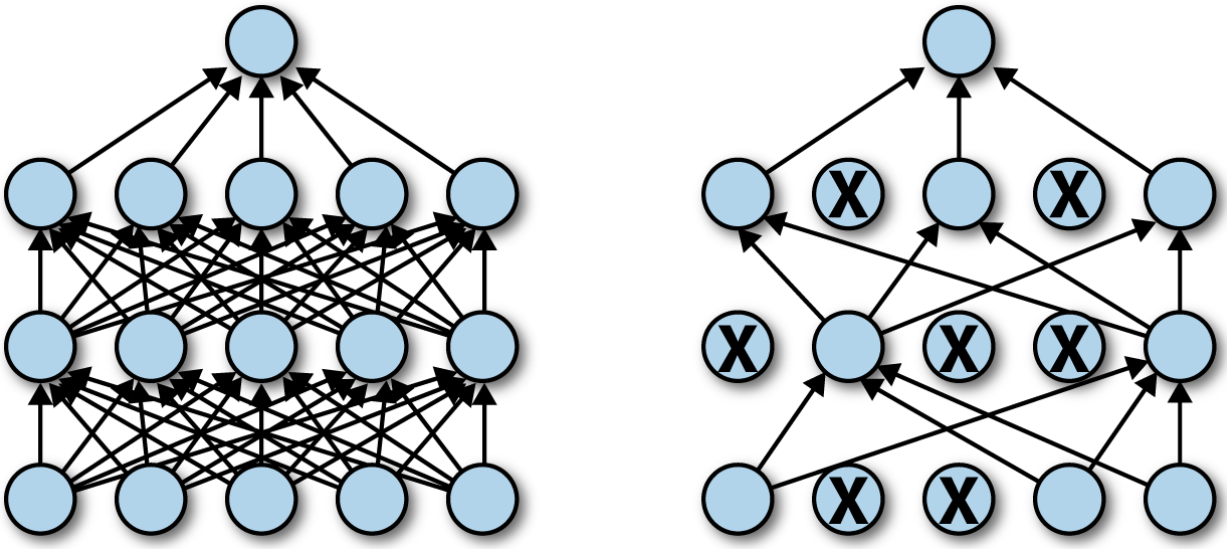


Figure 15 A Fully connected Neural network without Dropout(left) and with Dropout (Right).

Nodes are probabilistically or randomly dropped out during training thus simulating many different network architectures with just a single model. Dropout can be conceptualized as creating situations where units or layers may change and co-adapt in a way that they fix up the mistakes of the other units or layers. Each unit must learn to produce the best possible result without depending on the collective effort of other units thus making it more robust.

Dropout introduces a hyperparameter that specifies the probability at which outputs of the layer are dropped out, or inversely, the probability at which outputs of the layer are retained. Dropout is deactivated after training when predicting with the trained network. Weights are first scaled by the chosen dropout rate before finalizing the network since the weights of the network will

be larger than normal due to dropout. The probability p, of retention of a unit during training is multiplied by outgoing weights of that unit during testing.

3.2.10 Loss Functions

Loss function is one of many objective functions which is used to calculate the error of the neural network model when designing and configuring an ANN model. It is desirable to minimize the loss as low as possible in order to improve upon the accuracy of the model. Most widely used loss functions include Mean Squared Error (MSE), Categorical Cross Entropy (CC), Binary Cross Entropy etc.

Mean Squared Error (MSE)

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \tilde{y}_i)^2$$

Where y_i represents the true value and $\tilde{y}_i$ represents the predicted value. The goal is to minimize this mean, which will provide us with the regression line for fitting data. The real difference emerges at how MSE punishes bad performance. MSE punishes larger errors exponentially and reduces the influence of small ones.

3.2.11 Optimizers

An optimizer is an algorithm solely aimed at minimizing the objective function (loss function). In abstraction, it computes the rate of change of weights within a neural network, so that the error becomes lower on each iteration.

The choice of an optimizer may be determined by many factors like the architecture of a network, nature of data, nature of task or problem, etc. Quality of training is also affected by numerous hyper parameters of the of the algorithm and most often than not their default values are maintained except for the learning rate. A popular example of optimizers is Adam [25]. RMSprop [26], Stochastic Gradient Descent (SGD) [27] and AMSGrad [28].

a. Adam

Adam is an adaptive learning rate algorithm built mainly for deep neural networks. In that, it computes individual learning rates for different parameters within the network. It uses first and second moment estimations of gradient to adapt the learning rate for each weight of the neural network hence the name Adam (Adaptive moment). Expressed mathematically as;

$$m_n = E[X^n]$$

the n-th moment of a random variable, X can be defined as the expected value of that variable to the power of n. The first moment is mean, and the second moment is uncentered variance (mean is not subtracted during variance calculation). Moments estimation is done using exponentially moving averages, computed on the gradient evaluated on a current mini-batch. Moving averages of gradient and squared gradient are given by;

$$m_t = \beta_1 m_{t-1} + (1 - \beta_1) g_t$$
$$v_t = \beta_2 v_{t-1} + (1 - \beta_2) g_t^2$$

Where m and v are moving averages, g is gradient on current mini-batch, and recently introduced beta hyper-parameters, have default values of 0.9 and 0.999 respectively which have proven to be effective. From the expected values of the moving averages, the following property can be deduced:

$$E[m_t] = E[g_t]$$
$$E[v_t] = E[g_t^2]$$

Initial gradient values become less dominant due to multiplication by decreasing beta values as the value of m is expanded. The resulting moving average can be expressed as;

$$m_t = (1 - \beta_1) \sum_{i=0}^{t} \beta_1^{t-i} \, g_i$$

Expanding the gradient m_t equation there is an inherent bias correction for the first momentum estimator.

$$E[m_t] = E\left[(1 - \beta_1) \sum_{i=0}^{t} \beta_1^{t-i} \, g_i\right]$$
$$= E[g_i](1 - \beta_1) \sum_{i=1}^{t} \beta_1^{t-i} + \xi$$
$$= E[g_i](1 - \beta_1^t) + \xi$$

There are two things we should note from that equation.

1. A biased estimator: This is not just true for Adam only, the same holds for algorithms, using moving averages (SGD with momentum, RMSprop, etc.).
2. It won't have much effect unless it's the beginning of the training, because the value beta to the power of t is quickly going towards zero.

To correct the estimator to obtain the expected values (usually referred to as bias correction), the final formulas for our estimator will be as follows:

$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}$$

$$\hat{v}_t = \frac{v_t}{1-\beta_2^t}$$

Computed moving averages are then used to scale learning rate individually for each parameter (update rule for Adam) as shown below;

$$w_t = w_{t-1} - \eta \frac{\widehat{m}_t}{\sqrt{\hat{v}_t} + \epsilon}$$

Where **w** is model weights, eta, $\boldsymbol{\eta}$ is th step size.

1. The step size taken by the Adam in each iteration is approximately bounded by the step size hyper-parameter. This property adds intuitive understanding to the learning rate hyper-parameter.
2. Step size of Adam update rule is unaffected by the magnitude of the gradient, which helps a lot when going through areas with tiny gradients (e.g. saddle points or ravines).
3. Adam combines the advantages of Adagrad (which works well with sparse gradients) and RMSprop (which works well in on-line settings). This enables Adam to be well adapted for a wider range of tasks. Adam may be approached as a combination of RMSprop and SGD with momentum.

Adam also exists in two other variants**,** Adamax [29] and Nadam [30]. Since values of step size are often decreasing over time, a proposed fix is by keeping the maximum of values V and use it instead of the moving average to update parameters resulting in an algorithm called AMSGrad.

b. AMSGrad.

It has been noticed that in some scenarios like object recognition or machine translation, they fail to produce optimal global convergence and are outperformed by SGD with momentum [31]. Reddi et al. (2018), upon validation of the issue, pinpointed that the poor generalization of adaptive learning rate methods is due to the exponential moving average of past squared gradients

Though the introduction of the exponential average prevents the learning rates from becoming infinitesimally smaller after each training step, the short-term memory of the gradients poses a challenge in other cases. In instances where Adam converges to a suboptimal solution, it has been observed that, exponential averaging diminishes the influence of rarely occurring mini batches providing large and informative gradients hence, poor convergence. To correct this, AMSGrad which uses the maximum of past squared gradients (vt) rather than the exponential average to update the parameters was proposed. Instead of using vt (or its bias-corrected version, $\hat{v}_t$) directly as used in Adam above;

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

the previous vt−1 is employed if it is larger than the current one:

$$\hat{v}_t = max\ (\hat{v}_{t-1}, v_t)$$

Thus, a non-increasing step size characteristic which prevents the problems of Adam. The debiasing step as seen in Adam is removed. The complete AMSGrad update without bias-corrected estimates is expressed as:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1) g_t$$

$$v_t = \beta_2 v_{t-1} + (1-\beta_2) g_t^2$$

$$\hat{v}_t = max\ (\hat{v}_{t-1},\ v_t)$$

$$\theta_{t+1} = \theta_t - \frac{\eta}{\sqrt{\hat{v}_t} + \epsilon} m_t$$

3.2.12 Gradient descent

The most used algorithm to train neural networks is gradient descent. Gradient is a numeric computation indicating the adjustment rate of the network parameters to minimize output deviation (minimize the difference between the actual output and the one estimated). Graphically it would be the slope of the tangent line to the loss function at the current point (evaluating the current parameter values). Mathematically it's a vector that gives us the direction in

which the loss function increases faster, so we should move in the opposite direction if we try to minimize it. Gradient descent exists in several forms depending on the number of samples that we introduce to the network for each iteration.

Batch gradient descent: all available data is injected at once. This version implies a high risk of getting stuck, since the gradient will be calculated using all the samples, and the variations will be minimal sooner or later. As a general rule: for a neural network it's always positive to have an input with some randomness.

Stochastic gradient descent: a single random sample is introduced on each iteration. The gradient will be calculated for that specific sample only, implying the introduction of the desired randomness, and making more difficult the possibility of getting stuck. The main problem with this version is its slowness, since it needs many more iterations. Additionally, it doesn't take advantage of the available computing resources.

Mini-batch (Stochastic) gradient descent: instead of feeding the network with single samples, N random items are introduced on each iteration. This preserves the advantages of the second version and also getting a faster training due to the parallelization of operations. We'll use this version of the algorithm and choose a value for N that provides a good balance between randomness and training time (and not too large for the available GPU memory).

Its norm will be greater the steeper the tangent's slope is at the point at which it's calculated; therefore, the closer we are to a minimum of the loss function the smaller its norm will be (at a minimum the slope is zero). Thus the goal is to adjust the steps in the opposite direction to the gradient, to reach the global minimum:

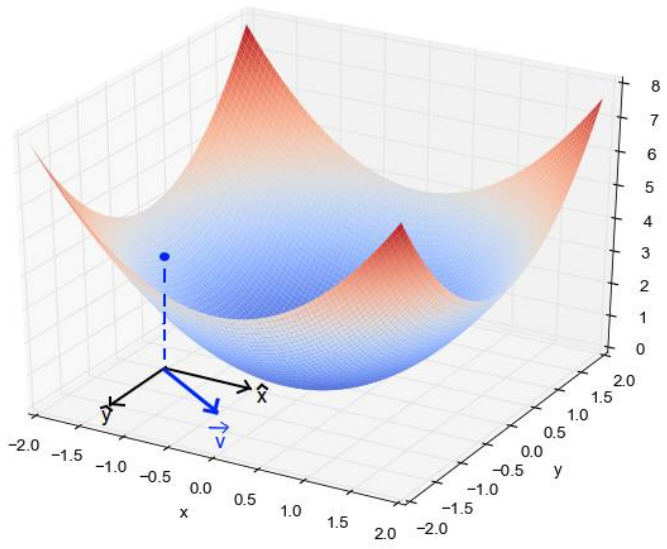


Figure 16 3D plot of convex loss plane

Loss functions are much more complex and non-convex due to the multiobjective and multi-dimensional nature of problem or input parameters. The surface of the function will present local minimums that in training could be easily confused with the global minimum.

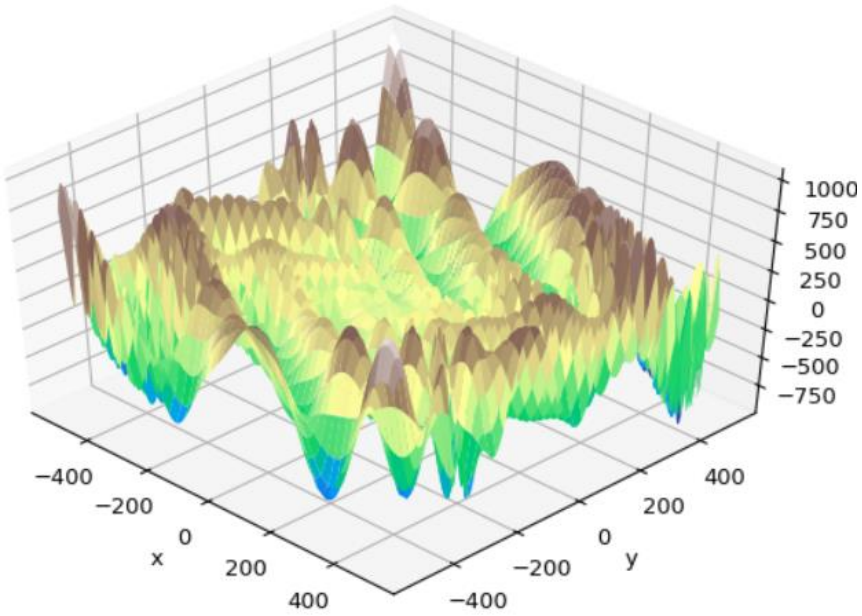


Figure 17 Multidimensional non-convex loss plane

The gradient vector calculation is complicated due to the large number of parameters and their arrangement in multiple layers. The **back-propagation** (or backward propagation) calculates the partial derivatives of the loss function with respect to the parameters of the last

layer, which don't influence over any other network parameters. Once these derivatives are obtained, the partial derivatives of the loss function with respect to the parameters of the next successive layers is calculated by means of the chain rule until the beginning of the network.

These are the iterations of the Gradient Descent algorithm using its third version:

4. A mini-batch input with N random samples from previously labeled training dataset is loaded into the network.
5. In a step known as forward propagation, computations on each layer of the network are performed and the outputs predicted.
6. The loss function for the mini-batch is computed. The value of this loss function is the number the algorithm is trying to minimize, and the following steps are oriented towards this.
7. The gradient is computed as the multi-variable derivative of the loss function with respect to all the network parameters.

8. Once the gradient vector is obtained, network parameters are updated by subtracting the product of the gradient value and learning rate to adjust the magnitude of steps. The learning rate is a factor used to adjust the rate of parameter update in the network. It shouldn't be too high (possible oscillations around optimum) or too low (slow convergence to optimum).

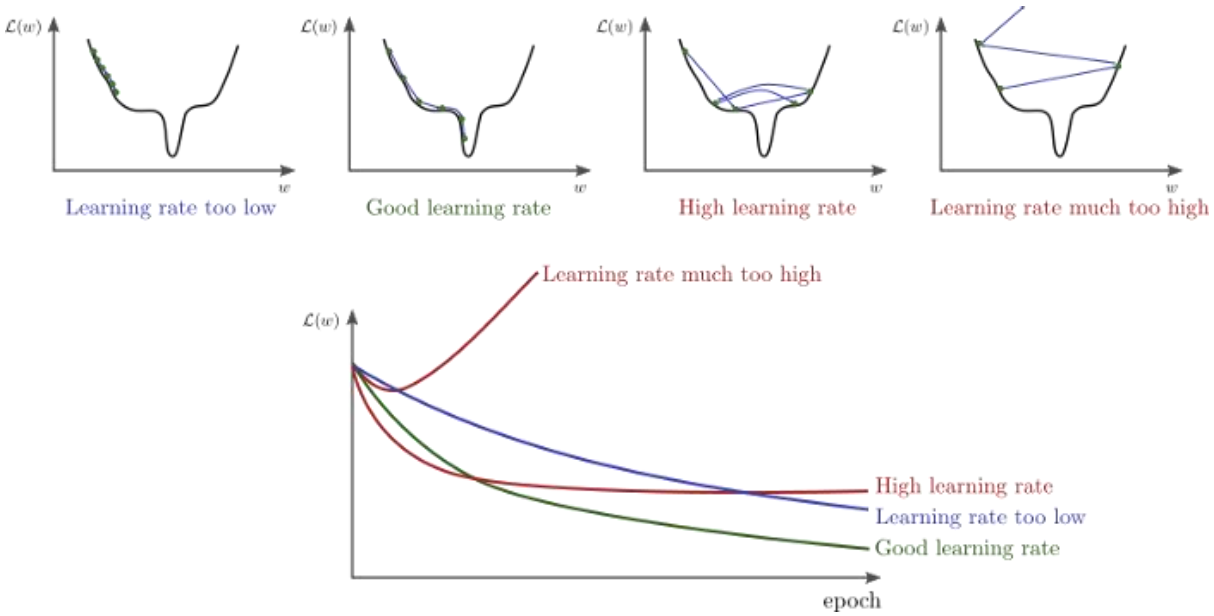


Figure 18 Comparison of the effects of different learning rate during gradient descents

A fixed or variable learning rate can be reduced over time, to avoid oscillations when we are close to the minimum of the loss function. This avoid overfitting which may be caused by a

small dataset compared to the network size or by a learning rate being too small and not allowing sufficiently quick parameter optimization.

3.2.13 Signal Denoising

Data noise removal is a process called denoising. There are a number of intrinsic or extrinsic conditions that can contribute to noisy data, and dealing with them is practically challenging.

Denoising is a fundamentally difficult problem in the fields of image processing, computer vision, and signal processing. As a result, it is crucial in many different fields where having access to the original data is crucial for reliable performance.

Autoencoders have been widely used for feature extraction in audio and computer vision areas. It can be described as trying to find a close representation to itself, during this procedure some hidden features could be found out.

$$x = \phi(\varphi(\tilde{x})).$$

Where $\varphi(\cdot)$ maps the input $\tilde{x} \in R^n$, an denoised data of x, to the hidden representation $\varphi(\tilde{x}) \in R^n$ of $\tilde{x}$, can be called the encoder, $\phi(\cdot): R^n \to R^n$ is the decoder which project the hidden layer back to the original x. The hidden layer can be treated as another representation of the input data.

And that is what we want to get. Then the learning objective is as the formula shows:

$$\mathrm{m}\ \ L(x, \hat{\phi}(\hat{\varphi}(\tilde{x}))) = \mathrm{m}\ \ \| x - \hat{\phi}(\hat{\varphi}(\tilde{x})) \|_2^2.$$

Where L is the loss between original input and the reconstructed value. $\hat{\phi}$ and $\hat{\varphi}$ are the estimated function of ϕ and φ. In more detail, ϕ and φ can be written as:

$$\hat{\varphi} = f(W_1 x + b_1).$$
$$\hat{\phi} = f(W_2 \tilde{\varphi}(x) + b_2).$$

Where $f(\cdot)$ can usually be set as the sigmoid or tanh activation function. Set $\theta = \{W_1, W_2, b_1, b_2\}$, then the target optimal objective can be:

$$\theta = \arg\ \mathrm{m}_{\theta}\ \ \| x - \hat{\phi}(\hat{\varphi}(\tilde{x})) \|_2^2.$$

3.2.14 Overview of Encoder-Decoder Network (Autoencoders)

Autoencoder is one of the major branches of deep learning. For it can learn the network to represent the input data itself, it is deployed in many feature extracting tasks. Deep learning methods avoid traditional designing features or detectors by hand. It learns from the data automatically, which could be much more effective and robust.

Autoencoder is an unsupervised artificial neural network that is trained to attempt duplicating its input to output. In the case of image data, the autoencoder will first encode the image into a lower-dimensional representation known as latent space representation or codings, then decodes that representation back to the image. They are trained similarly to via backpropagation. Encoder-Decoder automatically consists of the following two structures:

1. **The Encoder-** This network down sizes the sample data into lower dimensions. This part of the model takes in parameter the input data and compresses it according the equation:
    a. E(x) = c where x is the input data, c the latent representation and E our encoding function.
2. **The Decoder-** This network reconstructs the original data from the lower dimension representation. This part takes in parameter the latent representation and try to reconstruct the original input according to the equation:
    a. D(c) = x' where x' is the output of the decoder and D our decoding function.

Autoencoders have the limitation of being able to compress already trained data. They are also lossy in nature which means that the output will be degraded with respect to the original input.

3.2.15 Undercomplete Autoencoders

For the training phase of the autoencoder architecture, the goal is for the network to be able to learn to reconstruct the given input data. The Autoencoder model architecture illustrates the concept below.

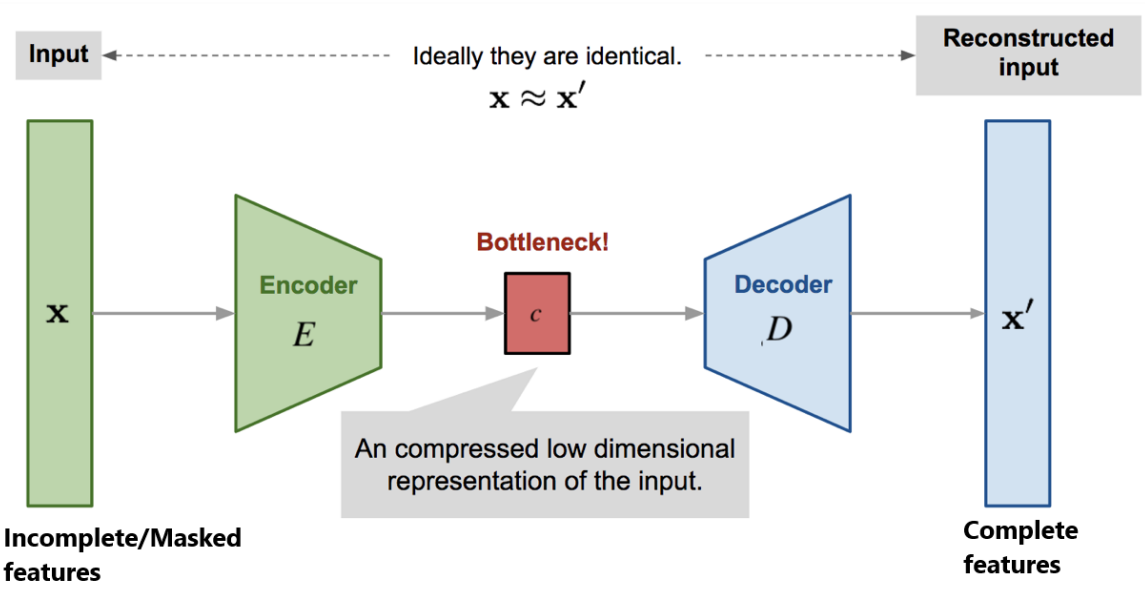


Figure 19 Encoder-Decoder Architecture

However, most of the time, it is not the output of the decoder that is of interests but rather the latent space representation. The decoder, is used to train the autoencoder end-to-end. Training the Autoencoder end-to-end will then allow the encoder to find useful features in our data.

To highlight important properties, for example, the latent space is constrained to be smaller than the dimension of the inputs. In this case, the model is an Undercomplete Autoencoders. In the majority of cases this type of autoencoders are used because of the main applications of its architecture which is its dimensionality reduction.

The learning process is regular and is aimed at minimizing a loss function.

$$\mathcal{L}(x, D(E(x)))$$

There are different metrics to quantify this loss function such as the Mean Square Error or the cross-entropy (when the activation function is a sigmoid for instance). This loss must penalize the reconstruction for being dissimilar from x Linear Autoencoders & Principal Component Analysis. One of the main applications of Autoencoders is dimensionality reduction, just like a Principal Component Analysis (PCA). In fact, if the decoder is linear and the cost function is the Mean Square Error, an Autoencoder learns to span the same subspace as the PCA.

3.2.16 Linear Autoencoders & Principal Component Analysis

One of the main applications of Autoencoders is dimensionality reduction, just like a Principal Component Analysis (PCA). if the decoder is linear and the cost function is the Mean Square Error, an Autoencoder learns to span the same subspace as the PCA.

Autoencoders: Training it with the Mean Square Error, which is aimed at minimizing.

$$\frac{1}{m}\sum_{i=1}^{m} \left\| x^{(i)} - D\left(E\left(x^{(i)}\right)\right)\right\|^2$$

where $E(x) = f(Wx) = c$ and $D(c) = g(Vc), W, V$ respectively the weights of the encoder and the decoder and f, g their activation functions

The objective function, is convex**.** Autoencoders are feedforward neural networks and are therefore trained as such with Gradient Descent algorithms. Deep-learning based autoencoders are typically prone to too much capacity with many hidden layers, thus such models learn the task of copying data in inputs without extracting important information leading overfitting. This phenomenon occurs as well with Undercomplete AE as Overcomplete AE (when the codings have higher dimensions than the inputs). Regularized Autoencoders are employed which ensures the model develops new properties and generalizes better. Commonly used forms of Regularized AE are:

Sparse Autoencoders: These are autoencoders trained with a sparsity penalty added to his original loss function. This sparsity penalty is a regularizer term, Ω(E(x)) added to a feedforward network.

$$\mathcal{L}(x, D(E(x))) + \Omega(E(x))$$

They are typically used for classification tasks. Denoising Autoencoders: Adding noise (Gaussian for example) to the inputs forces our model to learn important features from our data.

3.2.17 Denoising Autoencoders (DAEs)

This is a variant of Autoencoder concept discussed above, which is prone to a high risk of overfitting. With the Denoising Autoencoder, the data is partially corrupted by stochastic addition of noise to the input vector. The model is then trained to predict the original, uncorrupted data point as output. They are a feedforward network that can be trained with a gradient-based approximate minimization.

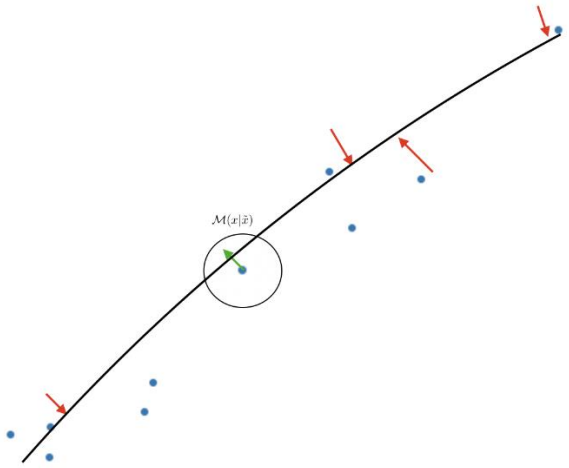


Figure 20 Finding the optimal regression line

(Source: inspired by I. Goodfellow, Y. Bengio, A. Courville Deep Learning)

In the image above, data is represented by the blue dots**.** The corrupted data will remain in the black circle of equiprobable corruption. During training, the aim is to minimize the MSE cost function. This optimization leads to minimizing the distance between the corrupted input and the black manifold which characterizes inputs.

Thus, the model learns a reconstruction vector field D(E(x))-x, some of these vectors are represented by the red arrows.

The main advantages of this architecture:

1. Learns more robust filters and prevents from learning a simple identify function.
2. Less prone to overfitting.

3. Autoencoders with pure dense layers are usually not powerful enough to extract important features of the data. Convolutional Autoencoders are therefore employed which consist of regular CNN that reduces the spatial dimension of the inputs by increasing the depth. The decoder does the reverse operation by typically using transpose convolutional layers.

3.3 **The Transformer Architecture**

The paper titled “Attention Is All You Need” published in 2017 by Google, introduced an encoder-decoder architecture based on attention layers, termed as the transformer. It is based on the application of transformer on NMT (Neural Machine Translator).

A transformer model is a neural network that learns context and thus meaning by tracking relationships in sequential data like the words in a sentence. Transformer models apply an evolving set of mathematical techniques, called attention or self-attention, to detect subtle ways in which even distant data elements in a series influence and depend on each other. Thus, providing insights and understanding into decisions taken by machine learning algorithms. The architecture aims to solve sequence-to-sequence tasks while handling long-range dependencies with ease.

Recently, transformers have been adopted in many applications, replacing the most popular types of deep learning models such as convolutional and recurrent neural networks (CNNs and RNNs). Currently, transformers are the driving algorithms behind the most popular search engines and language translation. One main difference between the transformer and other machine learning models is that it leverages high GPU parallelization to speed up training.

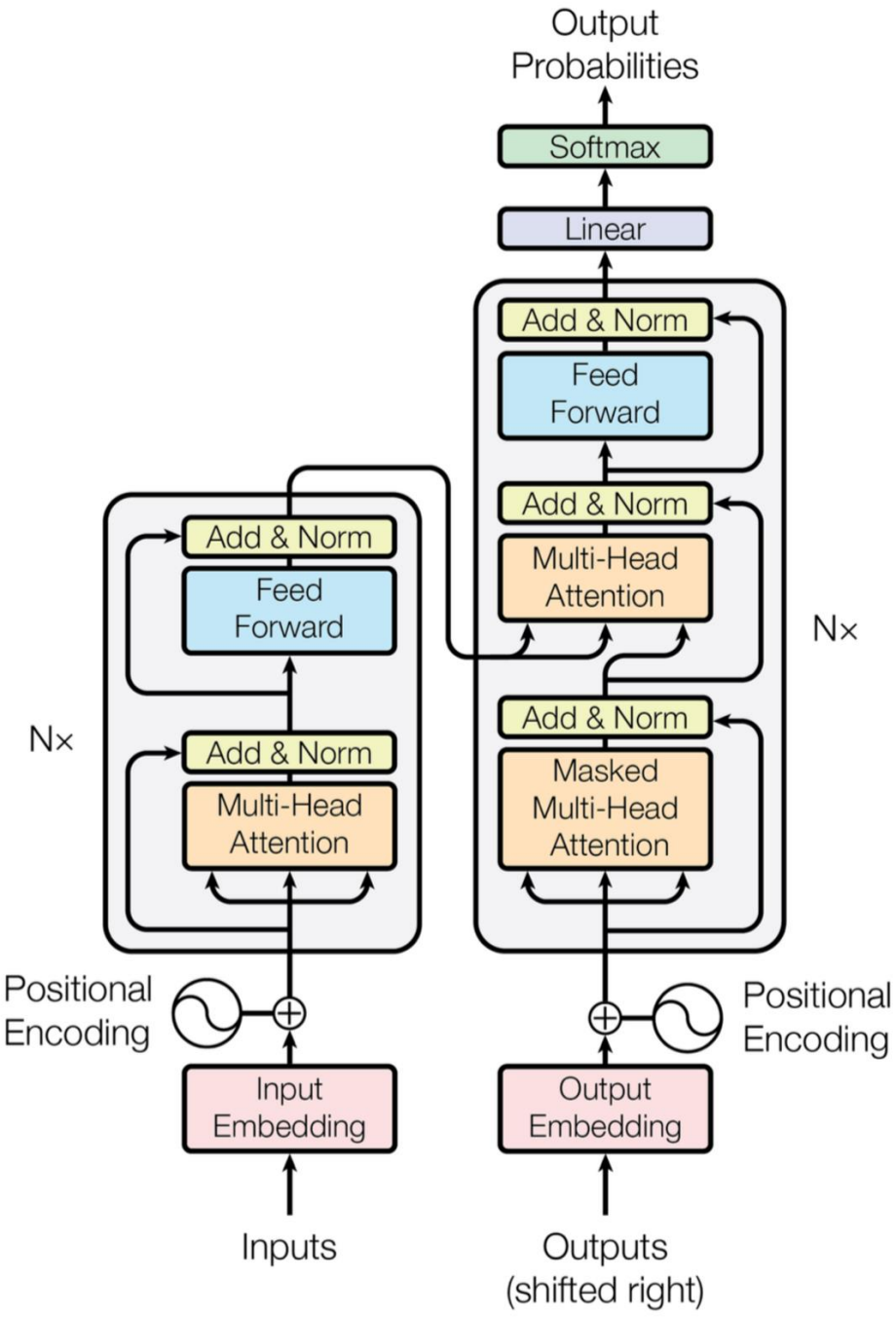


Figure 21 Transformer architecture

Similar to most neural networks, transformer models are essentially large encoder(left)/decoder(right) blocks that process data. Little but strategic additions to these blocks (shown above) make transformers uniquely powerful. Both Encoder and Decoder are/can be composed of repeatably stackable modules, described by Nx in the figure above. These modules consist mainly of Multi-Head Attention and Feed Forward layers.

The Encoder takes the input sequence and maps it into a higher dimensional space (n-dimensional vector). That abstract vector is fed into the Decoder which turns it into an output sequence. The output sequence can be in another language, symbols, a copy of the input, etc.

3.2.1 Input Sequence Repreentation Mechanism

One key difference of the model is the input representation. The embedding layer converts integer representation of words from a dictionary (tokenized words) into an Embedding Space which is a vector projection of words where words of similar meanings are grouped together or are present close to each other in that space. This allows language comprehension or manipulation by the model.

Positional encoding is also added the different inputs. Due to the absence of recurrent networks that can remember how sequences are fed into a model, each input is given a relative position since the order of its elements in a sequence is particularly. Transformers use positional encoders to tag the embedded representation (n-dimensional vector) of each element. Attention units follow these tags, calculating a pseudo algebraic map of how each of the interaction between elements. This allows the algorithm to recognize patterns as humans do. elements coming in and out of the network.

3.2.1 Attention Mechanism

The attention-mechanism looks at element in the input and computes the “importance” of other elements in order to provide context. For each element, the attention-mechanism takes into account several other elements simultaneously and decides which inputs are important by attributing different weights to those inputs. The Decoder will then take as input the encoded representation and the weights provided by the attention-mechanism.

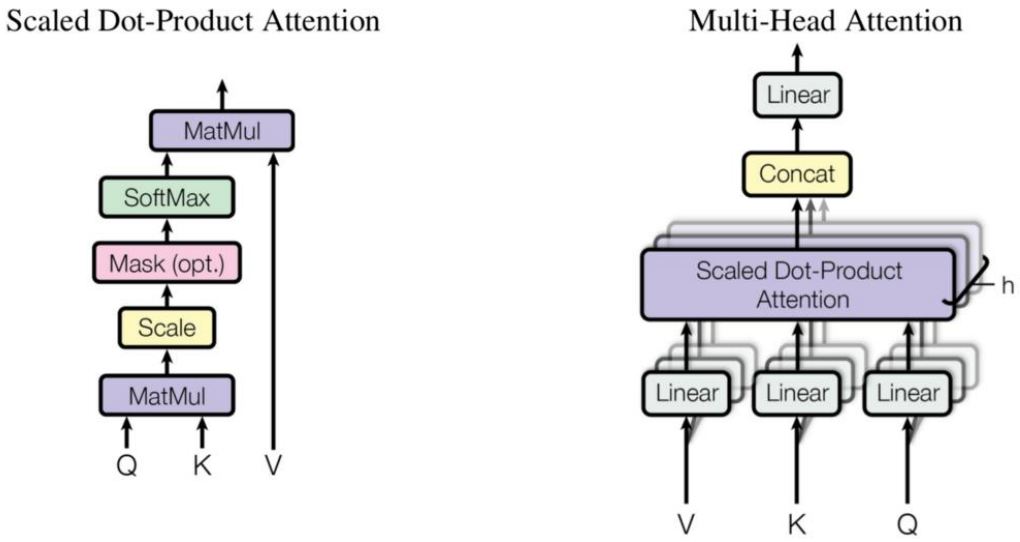


Figure 22 (left) Scaled Dot-Product Attention. (right) Multi-Head Attention with several parallel attention layers.

$$Attention(Q, K, V) = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)V$$

Where Q is a matrix that contains the query (vector representation of one word in the sequence), K are all the keys (vector representations of all the words in the sequence) and V are the values, which are again the vector representations of all the words in the sequence.

For the encoder and decoder, multi-head attention modules, V consists of the same word sequence than Q. However, for the attention module that is taking into account the encoder and the decoder sequences, V is different from the sequence represented by Q. Simply put, the values in V are multiplied and summed with some attention-weights a, where our weights are defined by:

$$a = softmax\left(\frac{QK^T}{\sqrt{d_k}}\right)$$

The weights a are defined by how each word of the sequence (represented by Q) is influenced by all the other words in the sequence (represented by K). This describes how the model focuses how relevant a particular input feature is with respect to other features in that input matrix. It is represented as an attention vector. For every input element, we can have an attention vector generated, which captures the contextual relationship called attention scores. The only problem associated with this function is that for every input element, self-product leads to higher weighs on itself than in the entire input, whereas the main focus is its interaction with other element of the input. The attention scores computed by this step are also normalized to prevent extremely large wights which affect model stability and exploding gradients. Multiple attention vectors per word is determined called the Multi-Headed Attention Block and a weighted average is taken, to compute the final attention vector of every word. Incorporation of a multi-headed attention layer also significantly solves the vanishing gradient issue.

The SoftMax function is then applied to the weights a to have a distribution between 0 and 1. Magnifying important input elements and suppressing unimportant input elements. These weights are then applied to all the inputs that are introduced in V.

To parallelize the attention-mechanism multiple mechanisms can be repeated multiple times with linear projections of Q, K and V computed by their products with weight matrices that are learned during the training. Thus, allowing the system to learn from different beneficial representations of Q, K and V. These linear representations are done using a feedforward dense network.

To attend on either the whole encoder input sequence or a part of the decoder input sequence, the matrices Q, K and V are different for each position of the attention modules in the structure depending on whether they are in the encoder, decoder or in-between encoder and decoder.

After the multi-attention heads in both the encoder and decoder, we have a pointwise feed-forward layer. This shallow feed-forward network has identical parameters for each position, which can be described as a separate, identical linear transformation of each element from the given sequence. It is applied to every attention vector, to mainly transform the attention vectors into a form that is acceptable by the next encoder or decoder layer or to suite the outputs. Given a classification or NLP task, the final Softmax Layer, transforms the input into a probability distribution, which is human interpretable.

3.4 Adaptation of Original Model

In this thesis, the embedding space would not be used as data to be utilized is already numeric and not a time series data. Additionally, the task at hand is an auto-regression task and not a classification task thus the encoder block is sufficient. The SoftMax layer from the output of the Transformer will be replaced since the output nodes are not probabilities but real values. The loss function used will be the mean squared error. The diagram below shows the encoder only architecture of the transformer model to be implemented.

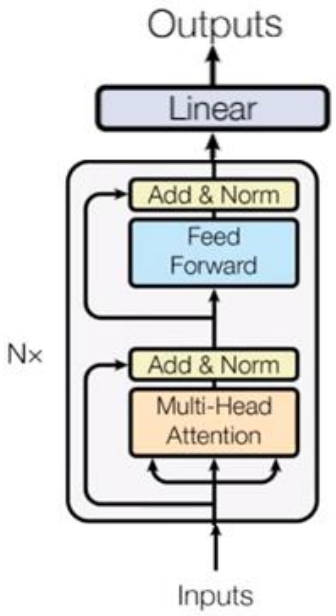


Figure 23 Encoder of the Transformer for regression task

CHAPTER FOUR

METHODOLOGY

4.1 Input Data

4.1.1 Title: Communities and Crime

Communities within the United States. The data combines socio-economic data
from the 1990 US Census, law enforcement data from the 1990 US LEMAS survey, and crime data from the 1995 FBI UCR.

Data Set Characteristics: Multivariate
Attribute Characteristics: Real
Associated Tasks: Regression
Number of Instances: 1994
Number of Attributes: 128
Missing Values? Yes
Area: Social
Date Donated: 2009-07-13

4.1.2 Source:

Creator: Michael Redmond (redmond 'at' lasalle.edu); Computer Science; La Salle University; Philadelphia, PA, 19141, USA
-- culled from 1990 US Census, 1995 US FBI Uniform Crime Report, 1990 US Law Enforcement Management and Administrative Statistics Survey, available from ICPSR at U of Michigan.
-- Donor: Michael Redmond (redmond 'at' lasalle.edu); Computer Science; La Salle University; Philadelphia, PA, 19141, USA
-- Date: July 2009

The dataset used for this study is real and authentic. The dataset was acquired from UCI machine learning repository website. The title of the dataset is ‘Communities and Crime’

This dataset contains a total number of 128 attributes and 1994 instances. All data provided in this dataset is numeric and normalized. The complete details of all 128 attributes can be acquired from the UCI machine learning repository website.

4.1.3 Attribute Information:

Attribute Information: (122 predictive, 5 non-predictive, 1 goal)
-- state: US state (by number) - not counted as predictive above, but if considered, should be consided nominal (nominal)
-- county: numeric code for county - not predictive, and many missing values (numeric)
-- community: numeric code for community - not predictive and many missing values (numeric)
-- communityname: community name - not predictive - for information only (string)
-- fold: fold number for non-random 10 fold cross validation, potentially useful for debugging, paired tests - not predictive (numeric)
-- population: population for community: (numeric - decimal)
-- householdsize: mean people per household (numeric - decimal)
-- racepctblack: percentage of population that is african american (numeric - decimal)
-- racePctWhite: percentage of population that is caucasian (numeric - decimal)
-- racePctAsian: percentage of population that is of asian heritage (numeric - decimal)
-- racePctHisp: percentage of population that is of hispanic heritage (numeric - decimal)
-- agePct12t21: percentage of population that is 12-21 in age (numeric - decimal)
-- agePct12t29: percentage of population that is 12-29 in age (numeric - decimal)
-- agePct16t24: percentage of population that is 16-24 in age (numeric - decimal)
-- agePct65up: percentage of population that is 65 and over in age (numeric - decimal)
-- numbUrban: number of people living in areas classified as urban (numeric - decimal)
-- pctUrban: percentage of people living in areas classified as urban (numeric - decimal)
-- medIncome: median household income (numeric - decimal)

-- pctWWage: percentage of households with wage or salary income in 1989 (numeric - decimal)
-- pctWFarmSelf: percentage of households with farm or self employment income in 1989 (numeric - decimal)
-- pctWInvInc: percentage of households with investment / rent income in 1989 (numeric - decimal)
-- pctWSocSec: percentage of households with social security income in 1989 (numeric - decimal)
-- pctWPubAsst: percentage of households with public assistance income in 1989 (numeric - decimal)
-- pctWRetire: percentage of households with retirement income in 1989 (numeric - decimal)
-- medFamInc: median family income (differs from household income for non-family households) (numeric - decimal)
-- perCapInc: per capita income (numeric - decimal)
-- whitePerCap: per capita income for caucasians (numeric - decimal)
-- blackPerCap: per capita income for african americans (numeric - decimal)
-- indianPerCap: per capita income for native americans (numeric - decimal)
-- AsianPerCap: per capita income for people with asian heritage (numeric - decimal)
-- OtherPerCap: per capita income for people with 'other' heritage (numeric - decimal)
-- HispPerCap: per capita income for people with hispanic heritage (numeric - decimal)
-- NumUnderPov: number of people under the poverty level (numeric - decimal)
-- PctPopUnderPov: percentage of people under the poverty level (numeric - decimal)
-- PctLess9thGrade: percentage of people 25 and over with less than a 9th grade education (numeric - decimal)
-- PctNotHSGrad: percentage of people 25 and over that are not high school graduates (numeric - decimal)
-- PctBSorMore: percentage of people 25 and over with a bachelors degree or higher education (numeric - decimal)
-- PctUnemployed: percentage of people 16 and over, in the labor force, and unemployed (numeric - decimal)
-- PctEmploy: percentage of people 16 and over who are employed (numeric - decimal)
-- PctEmplManu: percentage of people 16 and over who are employed in manufacturing (numeric - decimal)

-- PctEmplProfServ: percentage of people 16 and over who are employed in professional services (numeric - decimal)
-- PctOccupManu: percentage of people 16 and over who are employed in manufacturing (numeric - decimal) ########
-- PctOccupMgmtProf: percentage of people 16 and over who are employed in management or professional occupations (numeric - decimal)
-- MalePctDivorce: percentage of males who are divorced (numeric - decimal)
-- MalePctNevMarr: percentage of males who have never married (numeric - decimal)
-- FemalePctDiv: percentage of females who are divorced (numeric - decimal)
-- TotalPctDiv: percentage of population who are divorced (numeric - decimal)
-- PersPerFam: mean number of people per family (numeric - decimal)
-- PctFam2Par: percentage of families (with kids) that are headed by two parents (numeric - decimal)
-- PctKids2Par: percentage of kids in family housing with two parents (numeric - decimal)
-- PctYoungKids2Par: percent of kids 4 and under in two parent households (numeric - decimal)
-- PctTeen2Par: percent of kids age 12-17 in two parent households (numeric - decimal)
-- PctWorkMomYoungKids: percentage of moms of kids 6 and under in labor force (numeric - decimal)
-- PctWorkMom: percentage of moms of kids under 18 in labor force (numeric - decimal)
-- NumIlleg: number of kids born to never married (numeric - decimal)
-- PctIlleg: percentage of kids born to never married (numeric - decimal)
-- NumImmig: total number of people known to be foreign born (numeric - decimal)
-- PctImmigRecent: percentage of _immigrants_ who immigated within last 3 years (numeric - decimal)
-- PctImmigRec5: percentage of _immigrants_ who immigated within last 5 years (numeric - decimal)
-- PctImmigRec8: percentage of _immigrants_ who immigated within last 8 years (numeric - decimal)
-- PctImmigRec10: percentage of _immigrants_ who immigated within last 10 years (numeric - decimal)
-- PctRecentImmig: percent of _population_ who have immigrated within the last 3 years (numeric - decimal)

-- PctRecImmig5: percent of _population_ who have immigrated within the last 5 years (numeric - decimal)
-- PctRecImmig8: percent of _population_ who have immigrated within the last 8 years (numeric - decimal)
-- PctRecImmig10: percent of _population_ who have immigrated within the last 10 years (numeric - decimal)
-- PctSpeakEnglOnly: percent of people who speak only English (numeric - decimal)
-- PctNotSpeakEnglWell: percent of people who do not speak English well (numeric - decimal)
-- PctLargHouseFam: percent of family households that are large (6 or more) (numeric - decimal)
-- PctLargHouseOccup: percent of all occupied households that are large (6 or more people) (numeric - decimal)
-- PersPerOccupHous: mean persons per household (numeric - decimal)
-- PersPerOwnOccHous: mean persons per owner occupied household (numeric - decimal)
-- PersPerRentOccHous: mean persons per rental household (numeric - decimal)
-- PctPersOwnOccup: percent of people in owner occupied households (numeric - decimal)
-- PctPersDenseHous: percent of persons in dense housing (more than 1 person per room) (numeric - decimal)
-- PctHousLess3BR: percent of housing units with less than 3 bedrooms (numeric - decimal)
-- MedNumBR: median number of bedrooms (numeric - decimal)
-- HousVacant: number of vacant households (numeric - decimal)
-- PctHousOccup: percent of housing occupied (numeric - decimal)
-- PctHousOwnOcc: percent of households owner occupied (numeric - decimal)
-- PctVacantBoarded: percent of vacant housing that is boarded up (numeric - decimal)
-- PctVacMore6Mos: percent of vacant housing that has been vacant more than 6 months (numeric - decimal)
-- MedYrHousBuilt: median year housing units built (numeric - decimal)
-- PctHousNoPhone: percent of occupied housing units without phone (in 1990, this was rare!) (numeric - decimal)
-- PctWOFullPlumb: percent of housing without complete plumbing facilities (numeric - decimal)
-- OwnOccLowQuart: owner occupied housing - lower quartile value (numeric - decimal)
-- OwnOccMedVal: owner occupied housing - median value (numeric - decimal)
-- OwnOccHiQuart: owner occupied housing - upper quartile value (numeric - decimal)

-- RentLowQ: rental housing - lower quartile rent (numeric - decimal)
-- RentMedian: rental housing - median rent (Census variable H32B from file STF1A) (numeric - decimal)
-- RentHighQ: rental housing - upper quartile rent (numeric - decimal)
-- MedRent: median gross rent (Census variable H43A from file STF3A - includes utilities) (numeric - decimal)
-- MedRentPctHousInc: median gross rent as a percentage of household income (numeric - decimal)
-- MedOwnCostPctInc: median owners cost as a percentage of household income - for owners with a mortgage (numeric - decimal)
-- MedOwnCostPctIncNoMtg: median owners cost as a percentage of household income - for owners without a mortgage (numeric - decimal)
-- NumInShelters: number of people in homeless shelters (numeric - decimal)
-- NumStreet: number of homeless people counted in the street (numeric - decimal)
-- PctForeignBorn: percent of people foreign born (numeric - decimal)
-- PctBornSameState: percent of people born in the same state as currently living (numeric - decimal)
-- PctSameHouse85: percent of people living in the same house as in 1985 (5 years before) (numeric - decimal)
-- PctSameCity85: percent of people living in the same city as in 1985 (5 years before) (numeric - decimal)
-- PctSameState85: percent of people living in the same state as in 1985 (5 years before) (numeric - decimal)
-- LemasSwornFT: number of sworn full time police officers (numeric - decimal)
-- LemasSwFTPerPop: sworn full time police officers per 100K population (numeric - decimal)
-- LemasSwFTFieldOps: number of sworn full time police officers in field operations (on the street as opposed to administrative etc) (numeric - decimal)
-- LemasSwFTFieldPerPop: sworn full time police officers in field operations (on the street as opposed to administrative etc) per 100K population (numeric - decimal)
-- LemasTotalReq: total requests for police (numeric - decimal)
-- LemasTotReqPerPop: total requests for police per 100K popuation (numeric - decimal)
-- PolicReqPerOffic: total requests for police per police officer (numeric - decimal)
-- PolicPerPop: police officers per 100K population (numeric - decimal)

-- RacialMatchCommPol: a measure of the racial match between the community and the police force. High values indicate proportions in community and police force are similar (numeric - decimal)
-- PctPolicWhite: percent of police that are caucasian (numeric - decimal)
-- PctPolicBlack: percent of police that are african american (numeric - decimal)
-- PctPolicHisp: percent of police that are hispanic (numeric - decimal)
-- PctPolicAsian: percent of police that are asian (numeric - decimal)
-- PctPolicMinor: percent of police that are minority of any kind (numeric - decimal)
-- OfficAssgnDrugUnits: number of officers assigned to special drug units (numeric - decimal)
-- NumKindsDrugsSeiz: number of different kinds of drugs seized (numeric - decimal)
-- PolicAveOTWorked: police average overtime worked (numeric - decimal)
-- LandArea: land area in square miles (numeric - decimal)
-- PopDens: population density in persons per square mile (numeric - decimal)
-- PctUsePubTrans: percent of people using public transit for commuting (numeric - decimal)
-- PolicCars: number of police cars (numeric - decimal)
-- PolicOperBudg: police operating budget (numeric - decimal)
-- LemasPctPolicOnPatr: percent of sworn full time police officers on patrol (numeric - decimal)
-- LemasGangUnitDeploy: gang unit deployed (numeric - decimal - but really ordinal - 0 means NO, 1 means YES, 0.5 means Part Time)
-- LemasPctOfficDrugUn: percent of officers assigned to drug units (numeric - decimal)
-- PolicBudgPerPop: police operating budget per population (numeric - decimal)
-- ViolentCrimesPerPop: total number of violent crimes per 100K popuation (numeric - decimal) GOAL attribute (to be predicted)

4.1.4 Data Set Information:

Many variables are included so that algorithms that select or learn weights for attributes could be tested. However, clearly unrelated attributes were not included; attributes were picked if there was any plausible connection to crime (N=122), plus the attribute to be predicted (Per Capita Violent Crimes). The variables included in

the dataset involve the community, such as the percent of the population considered urban, and the median family income, and involving law enforcement, such as per capita number of police officers, and percent of officers assigned to drug units.

The per capita violent crimes variable was calculated using population and the sum of crime variables considered violent crimes in the United States: murder, rape, robbery, and assault. There was apparently some controversy in some states concerning the counting of rapes. These resulted in missing values for rape, which resulted in incorrect values for per capita violent crime. These cities are not included in the dataset. Many of these omitted communities were from the midwestern USA.

Data is described below based on original values. All numeric data was normalized into the decimal range 0.00-1.00 using an Unsupervised, equal-interval binning method. Attributes retain their distribution and skew (hence for example the population attribute has a mean value of 0.06 because most communities are small). E.g. An attribute described as 'mean people per household' is actually the normalized (0-1) version of that value.

The normalization preserves rough ratios of values WITHIN an attribute (e.g. double the value for double the population within the available precision - except for extreme values (all values more than 3 SD above the mean are normalized to 1.00; all values more than 3 SD below the mean are nromalized to 0.00)).

However, the normalization does not preserve relationships between values BETWEEN attributes (e.g. it would not be meaningful to compare the value for whitePerCap with the value for blackPerCap for a community)

A limitation was that the LEMAS survey was of the police departments with at least 100 officers, plus a random sample of smaller departments. For our purposes, communities not found in both census and crime datasets were omitted. Many communities are missing LEMAS data.

D. Feature Engineering

The objective of our analysis (crime prediction) requires as many variables recorded as possible for analysis hence for data preparation reduction was not employed.

There are different methods available for attribute or feature selection but manual method is usually chosen or attribute selection based on human understanding of data set. When dealing with a large number of attributes it is practical to use human knowledge to make decisions on the attributes and also taken in account that only those attributes are chosen which do not contain any missing values.

The Spearman correlation coefficient was implemented in this thesis to fully analyze the relationships whether linear or non-linear between the features of the data set and crime incidences and also the relationships between amongst the features. The Spearman correlation coefficient is defined as the Pearson correlation coefficient between the rank variables. The mathematical formula for the Spearman correlation coefficient is explained below:

For a sample of size n, the n raw scores X_i, Y_i are converted to ranks $\mathrm{R}(X_i), \mathrm{R}(Y_i)$, and r_s, the Spearman Rank Correlation is computed as

$$r_s = \rho_{\mathrm{R}(X),\mathrm{R}(Y)} = \frac{\mathrm{cov}\,(\mathrm{R}(X), \mathrm{R}(Y))}{\sigma_{\mathrm{R}(X)}\sigma_{\mathrm{R}(Y)}},$$

where

ρ denotes the usual Pearson correlation coefficient, but applied to the rank variables, $\mathrm{cov}\,(\mathrm{R}(X), \mathrm{R}(Y)$ is the covariance of the rank variables, $\sigma_{\mathrm{R}(X)}$ and $\sigma_{\mathrm{R}(Y)}$ are the standard deviations of the rank variables.

Only if all n ranks are distinct integers, it can be computed using the popular formula:

$$r_s = 1 - \frac{6\sum\ d_i^2}{n(n^2 - 1)}$$

where

$d_i = \mathrm{R}(X_i) - \mathrm{R}(Y_i)$ is the rank difference between each observation, n is the number of observations.

The mechanism of the Spearman's rank The mechanism of the Pearson Speraman's racorrelation does not enable it to capture geospatial information. Hence the pseudo zero correlation between the crimes can be observed. Also the normalization of some features limits the performance of the PScorrelation analysis.

correlation does not enable it to capture geospatial information. Hence the pseudo zero correlation between the crimes can be observed. Also the normalization of some features limits the performance of the Spearman correlation analysis.

In order to preserve the geospatial information provided by the names of each city, the input data must contain in some sense the location of the cities. However, the names of these cities provided as strings are incompatible with deep learning models since it only processes numerical data. With the use of Geolocate API by Google, each of the city names were converted into their respective coordinates (Longitude and Latitude) as additional geospatial information or feature to the model. Thus, the notion of geographical distribution of the recorded data can be learned from this new embedding.

The diagram below shows the features of the data set (First column from the left) and the types of crime (first row from the top)

	murders	murdPerPop	rapes	rapesPerPop	robberies	robbbPerPop	assaults	assaultPerPop	burglaries	burglPerPop	larcenies	larcPerPop	autoTheft	autoTheftPerPop	arsons	arsonsPerPop	ViolentCrimesPerPop	nonViolPerPop
LAT	0.078756	-0.149522	0.020848	-0.022099	-0.012609	-0.176514	-0.036439	-0.264123	-0.060608	-0.302183	-0.057050	-0.174565	-0.031980	-0.187663	-0.030183	-0.038577	-0.244733	-0.240106
LON	0.083481	-0.025038	-0.016574	-0.053455	-0.013279	-0.066463	-0.030033	-0.083873	-0.049552	-0.139018	-0.062495	-0.132168	-0.034406	-0.136618	-0.037688	-0.127314	-0.107790	-0.152136
fold	-0.022126	-0.012148	-0.066983	-0.014550	-0.049101	-0.036742	-0.040494	-0.016329	-0.047136	-0.023189	-0.042832	-0.026726	-0.048941	-0.026505	-0.069315	-0.041726	-0.025909	-0.026601
population	0.806243	0.175397	0.872701	0.102467	0.960922	0.310564	0.946003	0.134472	0.970511	0.118934	0.964768	0.066117	0.979800	0.244625	0.878431	0.141843	0.218363	0.122131
householdsize	0.031855	-0.013787	-0.043747	-0.129639	-0.013930	-0.002296	-0.015868	-0.015386	-0.033926	-0.102225	-0.048046	-0.237233	-0.011481	0.022872	0.005138	-0.020670	-0.018113	-0.201258
racepctblack	0.033112	0.561350	0.218193	0.408010	0.142891	0.621525	0.172483	0.544548	0.210246	0.547533	0.182164	0.361882	0.157621	0.406973	0.145603	0.310433	0.633457	0.479075
racePctWhite	-0.008717	-0.534437	-0.225233	-0.334420	-0.173168	-0.679859	-0.207508	-0.564386	-0.250618	-0.566043	-0.226155	-0.321949	-0.206583	-0.556921	-0.181032	-0.337536	-0.673175	-0.478256
racePctAsian	-0.033612	-0.101788	0.035189	-0.139841	0.057736	0.116986	0.054521	-0.035904	0.068098	-0.013081	0.070551	-0.095275	0.077859	0.182624	0.042947	-0.004461	0.018448	-0.044997
racePctHisp	-0.027407	0.121776	0.045009	-0.019537	0.072150	0.238461	0.092017	0.204887	0.109687	0.213107	0.111398	0.067454	0.105898	0.350240	0.086736	0.125504	0.241340	0.162583
agePct12t21	0.065296	0.039641	0.003817	0.083325	-0.003900	0.008743	-0.001486	0.054340	-0.003284	0.039017	-0.002425	0.048106	-0.009948	-0.058739	0.007175	0.028672	0.042466	0.033724
agePct12t29	0.057763	0.040195	0.064576	0.132881	0.028376	0.103265	0.041160	0.096074	0.054175	0.089807	0.061679	0.109645	0.036777	0.069399	0.046110	0.063744	0.120157	0.115238
agePct16t24	0.064338	0.014120	0.029362	0.091060	0.012438	0.048994	0.017149	0.051080	0.022402	0.053912	0.027835	0.083814	0.011060	-0.003256	0.017187	0.020263	0.059471	0.074023
agePct65up	-0.061660	0.083366	-0.067665	0.036538	-0.015239	0.017217	-0.026930	0.073308	-0.035821	0.124221	-0.040743	0.145548	-0.030759	0.002752	-0.053606	0.008081	0.045317	0.141149
numbUrban	0.778795	0.205828	0.836388	0.135049	0.924487	0.312985	0.913612	0.183401	0.934435	0.174761	0.928696	0.122994	0.941612	0.245414	0.845465	0.156692	0.258197	0.176010
pctUrban	-0.046328	0.109673	0.021171	0.087569	0.029224	0.108958	0.032824	0.129181	0.045735	0.170024	0.050672	0.198897	0.031118	0.120221	0.016468	0.080416	0.145719	0.217036
medIncome	0.010842	-0.366634	-0.090894	-0.425496	-0.052690	-0.273766	-0.077629	-0.401551	-0.101106	-0.419965	-0.103210	-0.455067	-0.063336	-0.180363	-0.078113	-0.295413	-0.408267	-0.479958
pctWWage	0.044758	-0.312934	-0.027764	-0.263280	-0.039487	-0.204245	-0.042325	-0.324682	-0.047693	-0.356787	-0.034435	-0.307044	-0.035806	-0.129567	-0.020499	-0.221915	-0.298713	-0.344444
pctWFarmSelf	0.017524	-0.050094	-0.084742	-0.061467	-0.046000	-0.199251	-0.061787	-0.097251	-0.076531	-0.096216	-0.065103	-0.035036	-0.067556	-0.226548	-0.043659	-0.082806	-0.156765	-0.082334
pctWInvInc	0.037729	-0.455678	-0.122512	-0.423753	-0.081923	-0.431218	-0.116671	-0.536007	-0.143598	-0.530387	-0.128122	-0.387572	-0.106773	-0.375398	-0.115237	-0.371888	-0.563234	-0.492975
pctWSocSec	-0.054364	0.163699	-0.063298	0.110434	-0.018518	0.044222	-0.028967	0.147754	-0.040742	0.170564	-0.051107	0.174842	-0.035103	0.015104	-0.051477	0.076916	0.107191	0.177152
pctWPubAsst	-0.008790	0.495169	0.173276	0.423795	0.117283	0.481439	0.140031	0.563149	0.165225	0.537849	0.139209	0.353694	0.140221	0.389821	0.133647	0.438650	0.580710	0.472169
pctWRetire	-0.041849	0.005370	-0.082444	-0.008908	-0.046460	-0.093831	-0.054366	-0.050090	-0.070360	-0.061454	-0.074500	-0.036264	-0.056890	-0.076762	-0.061184	-0.003149	-0.090990	-0.059677
medFamInc	0.010679	-0.377242	-0.094836	-0.427095	-0.056502	-0.284214	-0.083152	-0.414796	-0.105534	-0.421359	-0.105402	-0.436082	-0.069659	-0.199634	-0.087216	-0.306204	-0.421871	-0.467947
perCapInc	0.001037	-0.296393	-0.050915	-0.358178	-0.025070	-0.200350	-0.046393	-0.332046	-0.058414	-0.295713	-0.056450	-0.309859	-0.035396	-0.140737	-0.062644	-0.264879	-0.328899	-0.333442
whitePerCap	0.025705	-0.192067	0.007718	-0.299276	0.025799	-0.079612	0.015100	-0.229754	0.012902	-0.182509	0.012665	-0.230081	0.021479	-0.055699	-0.013566	-0.215286	-0.206689	-0.231902
blackPerCap	0.014153	-0.182478	-0.051188	-0.244439	-0.028655	-0.138739	-0.042246	-0.222024	-0.056279	-0.217788	-0.058984	-0.253631	-0.031181	-0.073651	-0.044112	-0.157799	-0.225544	-0.257839
indianPerCap	-0.008349	-0.072045	-0.027587	-0.115056	-0.006623	-0.024929	-0.012364	-0.074541	-0.014556	-0.065211	-0.017723	-0.096510	-0.009958	-0.015857	-0.009048	-0.040111	-0.064143	-0.091273
AsianPerCap	-0.012384	-0.116342	-0.072049	-0.197365	-0.029643	-0.112589	-0.041297	-0.132808	-0.057663	-0.155484	-0.062476	-0.190977	-0.040024	-0.098601	-0.046686	-0.149724	-0.146121	-0.190374
OtherPerCap	-0.008650	-0.136499	-0.048567	-0.133551	-0.030604	-0.102628	-0.042160	-0.130126	-0.054878	-0.153430	-0.055663	-0.150132	-0.041211	-0.069657	-0.051347	-0.094379	-0.136867	-0.165172
HispPerCap	-0.002257	-0.191423	-0.072044	-0.249340	-0.041261	-0.148420	-0.057172	-0.221671	-0.073361	-0.238661	-0.075118	-0.239166	-0.055448	-0.112586	-0.071729	-0.186676	-0.232931	-0.261679
NumUnderPov	0.826114	0.231573	0.867943	0.132459	0.967066	0.341811	0.950525	0.162808	0.970624	0.147007	0.958050	0.087444	0.983064	0.257882	0.882680	0.168432	0.248236	0.148255
PctPopUnderPov	0.005353	0.453375	0.150209	0.419832	0.095937	0.385753	0.124111	0.513966	0.156906	0.540067	0.149853	0.453543	0.112273	0.261337	0.115742	0.342490	0.520517	0.518668
PctLess9thGrade	-0.028664	0.320926	0.027950	0.168687	0.059325	0.281426	0.070904	0.365795	0.074075	0.341659	0.061875	0.223904	0.070179	0.265175	0.058583	0.211738	0.370423	0.302321
PctNotHSGrad	-0.031915	0.408063	0.064458	0.288087	0.072900	0.372559	0.088770	0.460130	0.097858	0.429844	0.078406	0.289035	0.086373	0.330159	0.079339	0.298849	0.469370	0.384540
PctBSorMore	0.043051	-0.276979	-0.026327	-0.307906	-0.017144	-0.199045	-0.029814	-0.309588	-0.031500	-0.273885	-0.019677	-0.235165	-0.026179	-0.200107	-0.035232	-0.260832	-0.307451	-0.280266
PctUnemployed	0.011395	0.469228	0.157977	0.396434	0.099508	0.416378	0.124858	0.502599	0.145600	0.476723	0.123479	0.294382	0.124288	0.369808	0.116825	0.388037	0.508584	0.401829
PctEmploy	0.021293	-0.342149	-0.038743	-0.267604	-0.048159	-0.229537	-0.054979	-0.359860	-0.060223	-0.367200	-0.044034	-0.277939	-0.047556	-0.157871	-0.036706	-0.239484	-0.327944	-0.329800
PctEmplManu	-0.041080	-0.013256	-0.038822	0.016974	-0.042618	-0.059101	-0.049422	-0.016558	-0.077137	-0.097450	-0.086317	-0.086616	-0.050060	-0.047778	-0.038737	0.031057	-0.029642	-0.098128
PctEmplProfServ	0.070735	0.000615	0.018754	-0.037455	0.015082	-0.023868	0.007984	-0.057683	0.011921	-0.032937	0.016007	-0.002821	0.003325	-0.114399	0.003504	-0.063985	-0.060692	-0.025203
PctOccupManu	-0.048283	0.286014	0.027424	0.285295	0.013519	0.204709	0.020593	0.302144	0.012705	0.244188	-0.003812	0.193636	0.012959	0.158370	0.024614	0.237944	0.303180	0.237860
PctOccupMgmtProf	0.044061	-0.280780	-0.039642	-0.337001	-0.017906	-0.226567	-0.032843	-0.328640	-0.037060	-0.291436	-0.026457	-0.250334	-0.027077	-0.210182	-0.035779	-0.268806	-0.338088	-0.296747
MalePctDivorce	-0.061615	0.403291	0.181509	0.508614	0.083026	0.434972	0.123110	0.467659	0.172479	0.536204	0.172999	0.513315	0.113785	0.370716	0.111204	0.389099	0.519896	0.578771
MalePctNevMarr	0.103886	0.162963	0.162706	0.151656	0.115464	0.356528	0.130185	0.205455	0.156134	0.217379	0.152865	0.148374	0.135687	0.292127	0.112533	0.140765	0.287341	0.214102
FemalePctDiv	-0.062427	0.410558	0.187892	0.475551	0.097049	0.461646	0.139074	0.482103	0.192554	0.562392	0.192119	0.510150	0.134007	0.415430	0.130578	0.392201	0.541567	0.591551
TotalPctDiv	-0.063311	0.418576	0.187688	0.500736	0.092364	0.458424	0.133852	0.486460	0.186174	0.562698	0.185879	0.524105	0.126515	0.400601	0.123297	0.397802	0.543047	0.599116
PersPerFam	0.034003	0.120733	0.052205	-0.033896	0.055175	0.175429	0.066791	0.126050	0.066537	0.038874	0.051845	-0.132703	0.076648	0.213945	0.083353	0.092302	0.155679	-0.053993
PctFam2Par	-0.003338	-0.565019	-0.229783	-0.538820	-0.149085	-0.629868	-0.184158	-0.623623	-0.229225	-0.663874	-0.212476	-0.548400	-0.173671	-0.465790	-0.156536	-0.437474	-0.702327	-0.657237

Figure 24 Correlation coefficients of urban features

4.2 ConvDense Encoder-Decoder Architecture for Data Imputation.

The Encoder-Decoder model was developed using Python and Tensorflow API as development and simulation tools. The encoder-decoder network was built to fill in missing data in the crime data for accurate modelling for effective prediction since more data points will fine the model. The encoder-decoder architecture being a machine learning model, was fined tuned by training and testing the model with different scenarios of the same data set.

Out of the 2215 locations recorded in the dataset, only 325 had completely filled information representing approximately 15 percent of the entire dataset which could lead to an inadequate model or biasness in predictions.

The ConvDense Encoder-Decoder model was trained and used for imputation in order to have a complete homogeneous dataset and increase crime prediction confidence.

4.2.1 Training of Encoder-Decoder Architecture.

The following steps were taken to prepare and train the model.

1. CSV data was imported as Pandas DataFrame for efficient manipulation.
2. Complete samples (locations) with missing data were dropped in order to obtain a complete dataset.
3. Each feature within the data was normalized using min-max scaling in order to have values ranging from 0 to 1. This is to speed up convergence of the model during training
4. The data is then split into training, validation and test set with 80% for training 10% for validation and 10% for testing.
5. A custom generator is used to randomly mask 30% of the features as inputs to the model for each iteration.
6. Target input to the model is the unmasked or complete features which the model must predict.
7. The mean squared error loss function was used as a metric to train the network.
8. The model was then compiled with the AMSgrad optimization function for adjusting the weights and biases.
9. 9. 51 1D convolutional layers are assembled into a ConvDense network. A fully connected layer with fewer neurons is added in steps of five between each succeeding convolutional layer to reduce the dimensionality or subsample until the bottleneck layer is reached.
10. Upsampling is similarly achieved by increasing the number of neurons in the dense layer between each successive convolutional layer in steps of 5 until the same number of inputs is reached at the output layer of the decoder network. This compresses or decompresses the features into a lower-dimensional vectors for feature extraction.
11. Each convolutional layer consists of Kernels/Filters which perform the convolution. The number of filters decreases and increases by 5 after each downsampling and upsampling fully connected layer respectively.
12. From the final convolutional layer of the main/deeper branch of the network, a two-dimensional average pooling layer is added to aggressively reduce the parameters and cut down the need for dense neural network.

13. The output from the average pooling layer is then fed into a dropout layer having a dropout rate of 0.01. This means 10% of the neurons will be randomly dropped during each training step. This is to ensure the network does not learn to memorize the training data but rather extract meaningful features of the data through the weights.The output layer has the same dimensionality as the input layer.
14. All layers are activated using the Gaussian Exponential Linear Unit function (GELU). For the magnitude prediction (a linear regression model), the loss function for updating the weights and biases chosen is the Mean-Squared Error (MSE).
15. The gradient-descent algorithm for backpropagation was chosen as the AMSGrad (Reddi, 2018), with an initial learning rate of 1e-3 to prevent overshoot during optimization and facilitate convergence to the global minimum.
16. Callback functions were implemented when a training parameter was updated after each epoch. The callback functions that were implemented are:

    i. Model checkpoint saving - It is called when the validation test set loss improves (best weights will be saved).

    ii. Reduce learning rate - Learning rate or the rate of adjusting the weights and biases of the neurons in the network is reduced when the learning plateaus, to prevent the model from running into a saddle point (where there is no improvement in accuracy or the loss function does not reduce). Minimum learning rate was set to 10e-7 with a decay factor of 0.9.

    iii. Early Stopping - Implemented when the validation loss does not improve after a certain number of epochs to prevent overfitting and poor generalization of the model. This is set to 20.

4.3 Transformer Architecture for Crime Prediction.

The Transformer model was developed using Python and Tensorflow API as development and simulation tools. The crime and urban features of the data set including the missing dates filled in by the encoder-decoder network utilized for this thesis were used as inputs to the transformer architecture. The transformer architecture being a machine learning model, was fined tuned by training and testing the model with different scenarios of the same data set.

4.3.1 Training Transformer Architecture.

The following steps were taken to prepare and train the model.

1. Imputed CSV data was imported as Pandas DataFrame for efficient manipulation.

2. Crime and urban features were selected as inputs to the model (X data). The various recorded crimes types were used as target predictions or outputs of the model (Y data).

3. Target data was normalized using min-max scaling in order to have values ranging from 0 to 1. This is to speed up convergence of the model during training.

4. The data is then split into training, validation and test set with 80% for training 10% for validation and 10% for testing.

5. The mean squared error loss function was used as a metric to train the network.

6. The model was then compiled with the AMSgrad optimization function for adjusting the weights and biases.

2. A transformer network of 4 transformer block layers is compiled.

3. The output of the transformer blocks which has the same dimension as the input features is fed into 2 successive fully connected linear layers consisting of 127 and 90 neurons each.

4. The output of the linear layers is then fed into the prediction layer of the model which has the same dimension of the number of target crime data to be predicted, thus having 18 neurons.

5. The output of the linear layers is then fed into the prediction layer of the model which has the same dimension of the number of target crime data to be predicted, thus having 18 neurons.Upsampling is similarly achieved by increasing the number of neurons in the dense layer between each successive convolutional layer in steps of 5 until the same number of inputs is reached at the output layer of the decoder network. This compresses or decompresses the features into a lower-dimensional vectors for feature extraction.

6. All layers are activated using the Gaussian Exponential Linear Unit function (GELU). For the magnitude prediction (a linear regression model), the loss function for updating the weights and biases chosen is the Mean-Squared Error (MSE).

7. The gradient-descent algorithm for backpropagation was chosen as the AMSGrad (Reddi, 2018), with an initial learning rate of 1e-3 to prevent overshoot during optimization and facilitate convergence to the global minimum.

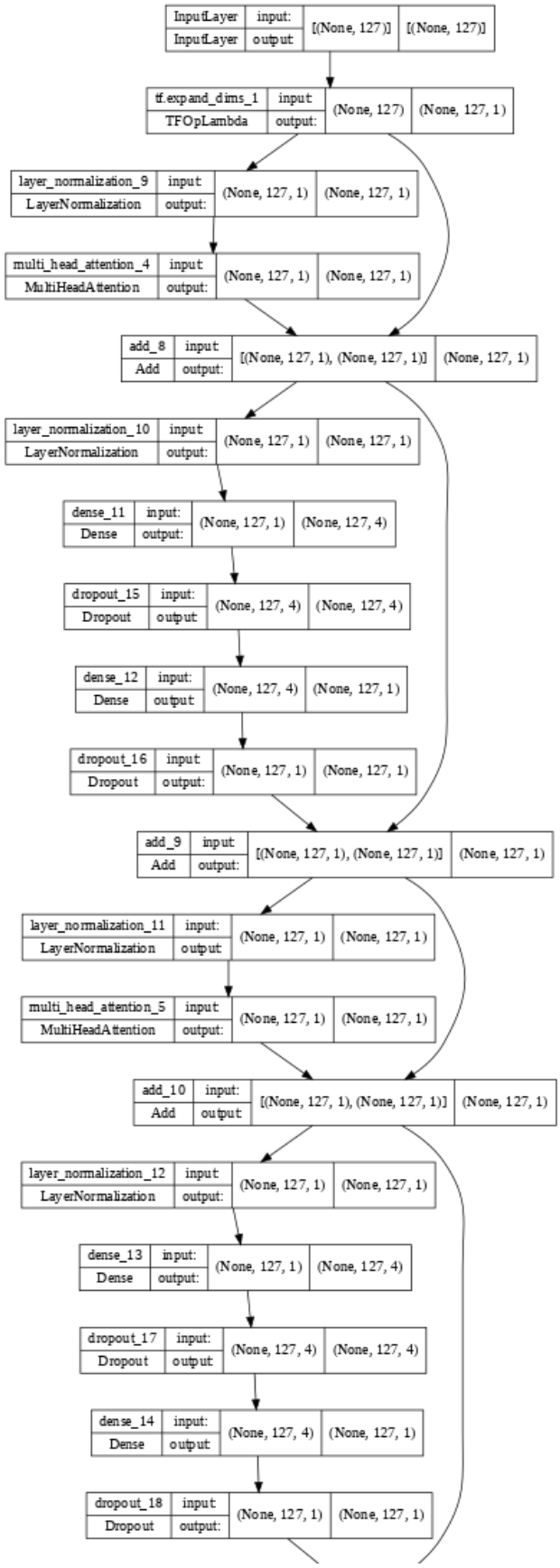
8. Callback functions were implemented when a training parameter was updated after each epoch. The callback functions that were implemented are:

i. Model checkpoint saving - It is called when the validation test set loss improves (best weights will be saved).

ii. Reduce learning rate - Learning rate or the rate of adjusting the weights and biases of the neurons in the network is reduced when the learning plateaus, to prevent the model from running into a saddle point (where there is no improvement in accuracy or the loss function does not reduce). Minimum learning rate was set to 10e-7 with a decay factor of 0.9.

iii. Early Stopping - Implemented when the validation loss does not improve after a certain number of epochs to prevent overfitting and poor generalization of the model. This is set to 20.

4.4.3 Transformer Architecture



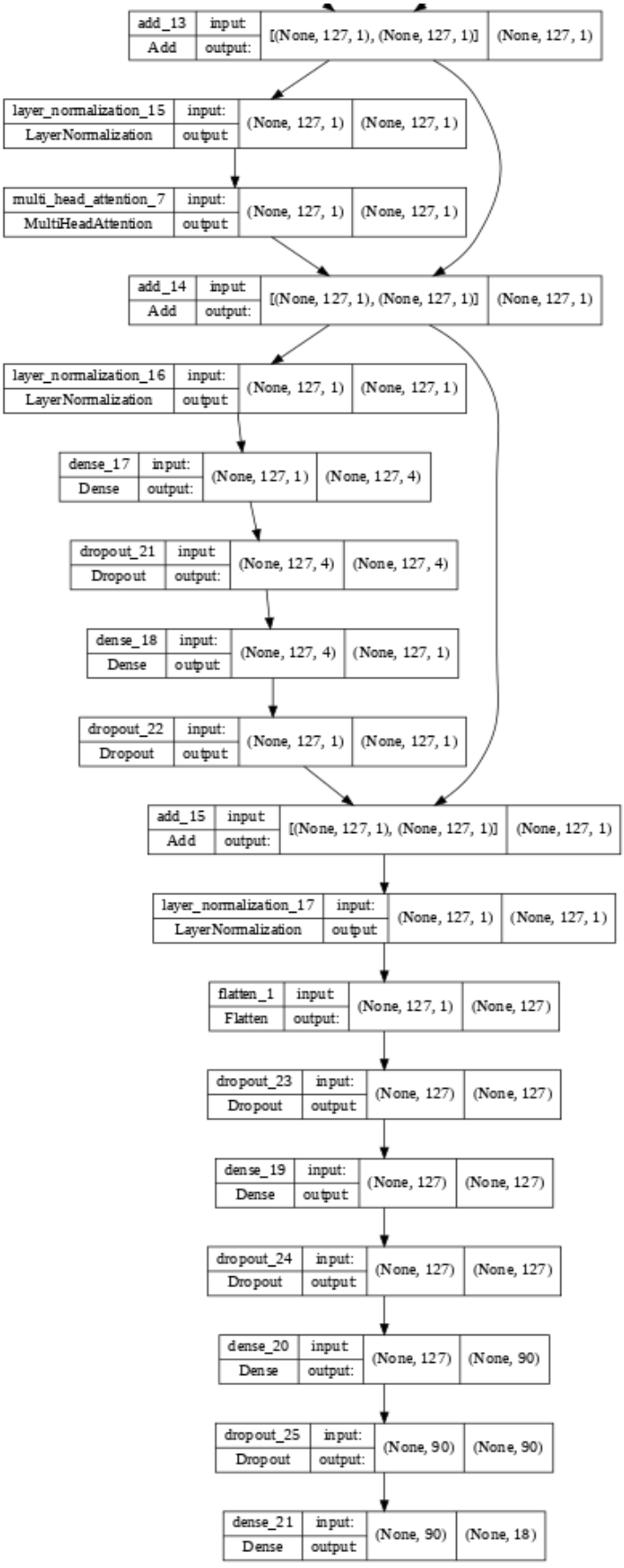


Figure 25 Conv-Dense Encoder-Decoder model

4.5 Code Platform and Computer Specifications

Graphics Processing Unit (GPU)

A CPU (central processing unit) works together with a GPU (graphics processing unit) to boost the throughput of data and scale up the number of concurrent computations being performed. Originally intended for computer graphics and video game consoles like the PlayStation and Xbox, GPUs were made to handle visuals. The use of GPUs for accelerating calculations requiring enormous amounts of data, such as those for autonomous vehicles, 3D rendering, and artificial intelligence or machine learning, has been around since 2010..

A CPU is an inherent component of every computer and can never be fully replaced by a GPU. In that, a GPU performs repetitive calculations in parallel, complementing the CPU architecture while the main program however, continues to run on the CPU. The CPU coordinates a host of general-purpose computing tasks due to the large or rich instruction sets embedded, while the GPU, having a limited instruction set, performs a more specialized (usually mathematical) tasks. The power of parallelism, enables a GPU to complete more work in the same amount of time as compared to a CPU thereby making heavy computing in real-time achievable.

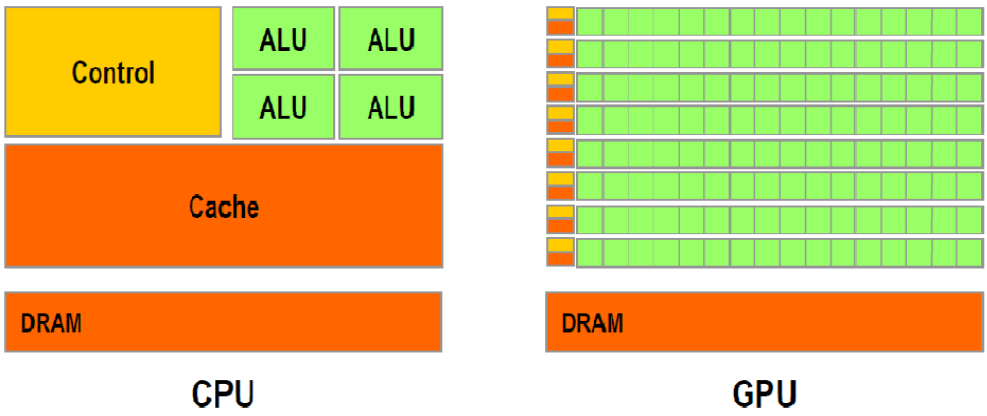


Figure 3.1 CPU architecture vs. GPU architecture.

The GPU contains thousands of processing cores as compared to a CPU making it capable of massive data parallelism. Though the CPU contains a few cores, they are much faster in terms of clock speed which makes up for their inability to process massive information concurrently.

Due to its superior math capabilities, a GPU increases the quantity of data a CPU can process in a given amount of time. The GPU can offload specialized applications like machine learning and deep learning that require sophisticated mathematical computations. By doing this, the CPU has more time and resources available to perform other activities or processes more effectively (running the sequential part of workloads).

By utilizing the capability of GPUs, CUDA®, a parallel computing platform and programming model developed by NVIDIA for general computing on graphical processing units (GPUs), dramatically increases computation speed for applications [34]. Express parallelism through extensions can be achieved when utilizing CUDA with just a few written keywords, making it simple for developers to write programs in well-known languages like C, C++, Fortran, Python, and MATLAB.The Google TensorFlow API provides a variety of toolkits that allow easy implementation of machine learning codes to any level of abstraction. Its also allows creation of architectures compatible with any computing device of choice (CPU, GPU etc.).

Python, although, not highly recommended for building Machine Learning models, it enables efficient data manipulation and development of data driven numerical solutions due to a host of libraries and support available. Analysis and development of the project will be carried out with Python environment.

CPU: Intel(R) Core™ i7-7700HQ CPU @2.80 GHz, GPU: NVIDIA GeForce GTX 1060, Installed Memory (RAM): 16GB
Computation was performed on the GPU to leverage its numerous powerful cores in performing repetitive mathematical computations in parallel. The CPU coordinates and manages the process while the GPU performs a more specific task of housing the prediction model, and all mathematical computation tasks thereby effectively reducing the computational burden.

CHAPTER FIVE

DISCUSSION OF RESULTS AND TESTING

5.1 Introduction.

In this chapter, the urban features that show perfect collinearities from correlation matrix of the urban features are discussed, conventional models: Support Vector Machine (SVM), Multi Output Random Forest Regressor (Mo-RF) and Random Forest Regressor (RFR) were developed and modelled with the same crime data and compared to our novel transformer architecture for justification of the proposed methodology of this thesis and other observed trends in the results are also discussed.

5.2 Collinearity Analysis

Collinearity in geometry is described as a collection of points that lie along the same path. A linear relationship between two features on a graph is created when colinear points are combined. If two features have a colinear relationship, A virtually or about 1 or -1 spearman rank correlation. Two features are directly proportional to one another when they are close to 1, and they are inversely proportional when they are close to -1. The figures below show the collinearities between each of the features.

.

PctNotSpeakEnglWell	PctSpeakEnglOnly	-0.927451
PctSpeakEnglOnly	racePctHisp	-0.910258
pctWWage	pctWSocSec	-0.902436
PctRecImmig5	PctRecentImmig	0.973580
TotalPctDiv	MalePctDivorce	0.975785
RentHighQ	MedRent	0.976395
perCapInc	whitePerCap	0.976993
medFamInc	medIncome	0.979392
RentHighQ	RentMedian	0.979471
NumUnderPov	NumKidsBornNeverMar	0.981922
PctPersOwnOccup	PctHousOwnOcc	0.982166
PctRecImmig10	PctRecImmig5	0.983234
FemalePctDiv	TotalPctDiv	0.983879
OwnOccMedVal	OwnOccHiQuart	0.984659
PctLargHouseFam	PctLargHouseOccup	0.985382
PctKids2Par	PctFam2Par	0.985596
NumUnderPov	population	0.986054
MedRent	RentMedian	0.987347
OwnOccMedVal	OwnOccLowQuart	0.991415
PctRecImmig5	PctRecImmig8	0.992012
PctRecImmig8	PctRecImmig10	0.993566

Figure 26 A table of the Spearman's Rank Correlation between urban features.

The figure above shows the collinearities between the urban two features indicated by computing their Spearman's Rank correlation. The PctRecImming8 (percentage of immigrants who immigrated within the last 8 years) and PctRecImming10 (percentage of immigrants in the last 10 years) have the highest collinearity of 0.993566 showing they are directly proportional to each other but not perfectly linear. The PctNotSpeakEnglWell (percentage of people who do not speak English well) and PctSpeakEnglOnly (percentage of people who speak English only) have a lowest collinearity of -0.927451 showing they are inversely proportional to each other but not perfectly linear.

assaults	NumImmig	0.903911
autoTheft	HousVacant	0.903958
assaults	numbUrban	0.913612
larcenies	NumKidsBornNeverMar	0.920386
autoTheft	NumImmig	0.922009
robberies	numbUrban	0.924487
larcenies	numbUrban	0.928696
	HousVacant	0.933013
burglaries	numbUrban	0.934435
	HousVacant	0.935570
robberies	NumImmig	0.939166
autoTheft	numbUrban	0.941612
burglaries	NumKidsBornNeverMar	0.943564
assaults	NumKidsBornNeverMar	0.946456
	population	0.948003
	NumUnderPov	0.950525
larcenies	NumUnderPov	0.958050
robberies	population	0.960922
larcenies	population	0.964768
autoTheft	NumKidsBornNeverMar	0.964769
robberies	NumUnderPov	0.967066
burglaries	population	0.970511
	NumUnderPov	0.970624
robberies	NumKidsBornNeverMar	0.977501
autoTheft	population	0.979800
	NumUnderPov	0.983064

Figure 27 A table showing the Spearman's Rank Correlation between urban features and the types of crimes.

The figure above shows the collinearities between the one urban feature and a crime feature indicated by computing their Spearman's Rank correlation. The autoTheft (car theft) and NumUnderPov (number of people under the poverty line) have the highest collinearity of 0.983064 showing they are directly proportional to each other but not perfectly linear.

larcenies	robberies	0.913590
nonViolPerPop	larcPerPop	0.924688
burglaries	robberies	0.933656
assaults	larcenies	0.937411
autoTheft	larcenies	0.948865
burglaries	assaults	0.949401
assaults	autoTheft	0.949919
robberies	assaults	0.957121
autoTheft	robberies	0.962140
	burglaries	0.970357
burglaries	larcenies	0.983269
murders	murders	1.000000

Figure 28 A table showing the Spearman's Rank Correlation between the types of crimes.

The figure above shows the collinearities between the one two crime features indicated by computing their Spearman's Rank correlation. The burglaries (car theft) and larcenies (number of people under the poverty line) have the highest collinearity of 0.983269 showing they are directly proportional to each other but not perfectly linear.

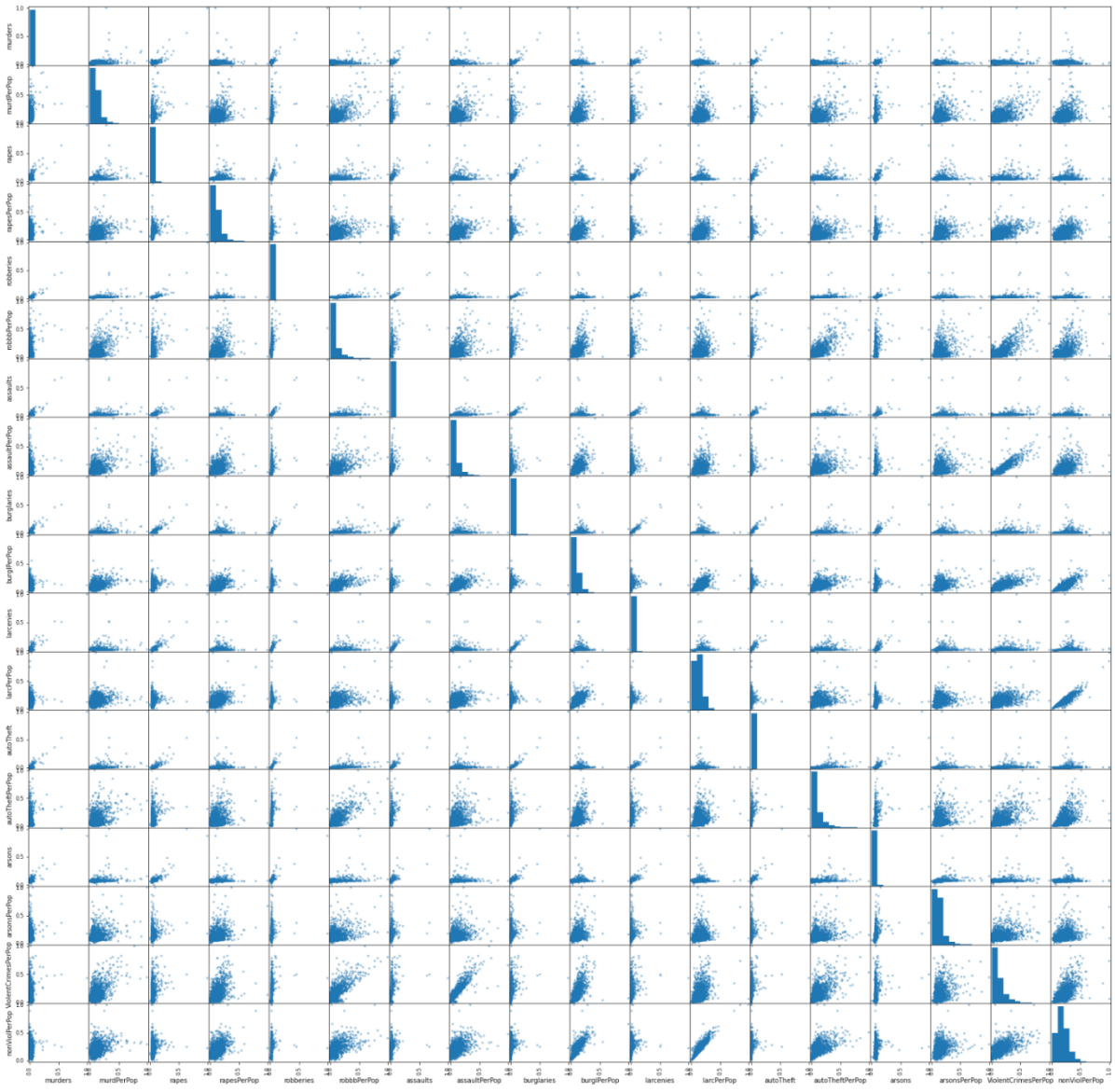


Figure 29 A graph depicting a scatter plot of the various level of collinearities between urban features and the types of crimes.

The figure above depicting the collinearities between urban features and the types of crime and goes a long way to show that the relationships are not perfectly linear.

5.3 Baseline Analysis

5.3.1 Modeling of RFR, Mo-RF and SVM with full urban features.

For the justification of our proposed Deep learning method, three separate conventional methods were developed: Support Vector Machine (SVM), Multi Output Random Forest Regressor (Mo-RF) and Random Forest Regressor (RFR) and these models were compared with each

other in terms of the Mean Absolute Percentage Error (MAPE) of the various crimes detailed in the crime data utilized in this thesis when all the urban features were all the urban were included in the modeling process.

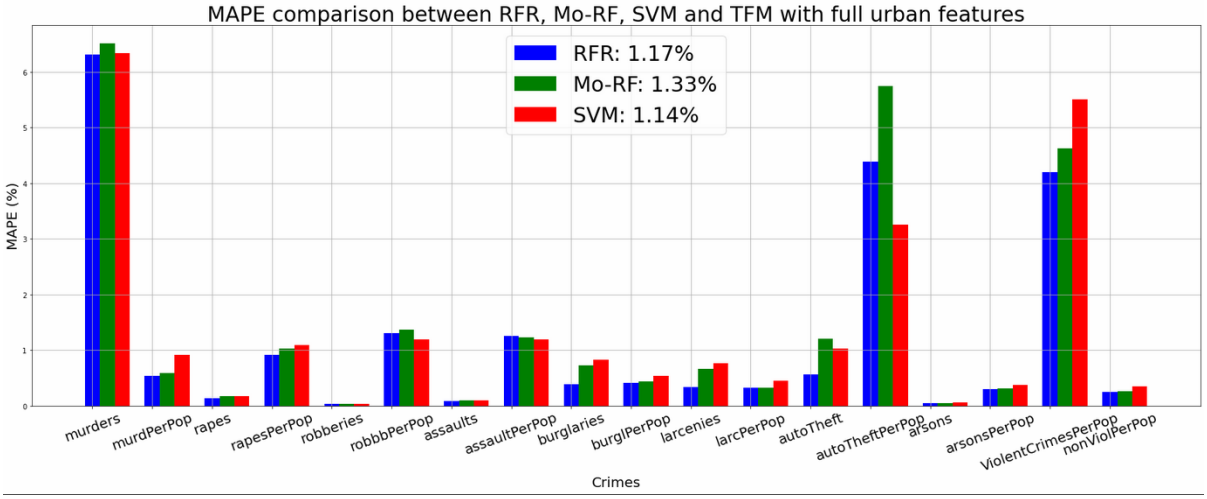


Figure 30 A comparison between RFR, Mo-RF and SVM with full urban features.

From the graph above, the prediction of murders showed the highest MAPE for all the three models. The non-violent crimes per population had the lowest MAPE for all the three models being compared. On the average, the Support Vector Machine model outperformed the RFR, and Mo-RF models with an average of 1.4%.

5.3.2 Modeling of RFR, Mo-RF and SVM without geolocation

As special addition to the thesis, the geographical information of the various states were converted into longitude and latitude for to improve the performance of the three models. The three models were developed without this feature to evaluate the performance of these models as shown below.

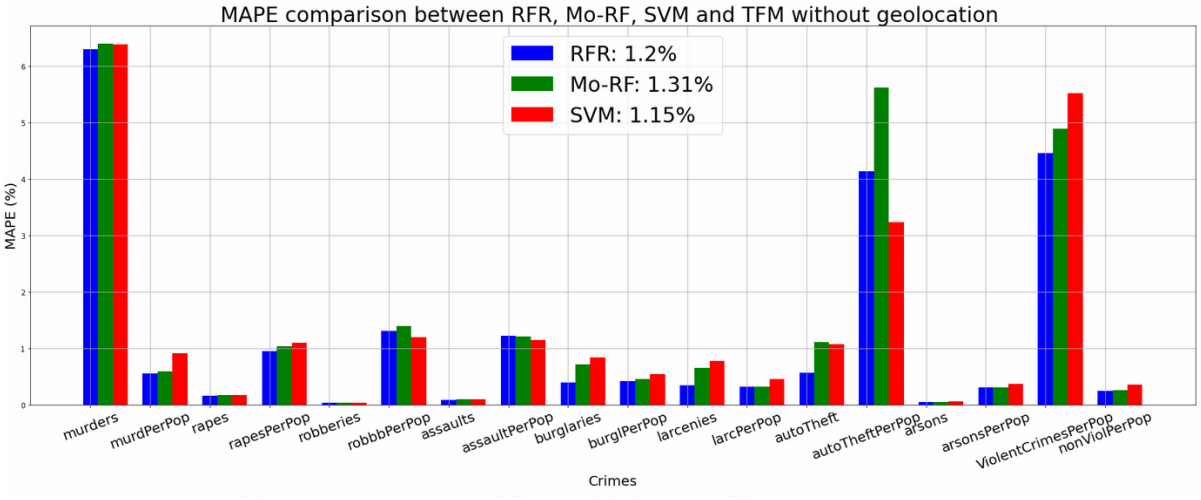


Figure 31 A graph showing a comparison between RFR, Mo-RF and SVM without geolocation

From the graph above, the MAPE for the prediction of murders still was the highest and non-violent crimes per population had the lowest MAPE. The Support Vector Machine outperformed the RFR and Mo-RF models with a MAPE of 1.15%.

5.4 Transformer Architecture Training and Evaluation Metrics

After 138 epochs of training, the model achieved a test accuracy of approximately 55% in predicting the target crime magnitudes at an average of 72ms per step in each epoch. This means that, the model is potentially faster during deployment because of data preprocessing bottle neck and lack of parallelization and data pipelining (data generation on the fly).

Due to the Transformer architecture and GPU memory limitations, the whole dataset cannot be loaded into the GPU memory for training. Instead, a custom-coded generator object is used to feed the data in batches or chunks into the network during training. This custom generator object is at the same time responsible for preprocessing the input and expected outputs into the right format. The GPU's higher processing speed coupled with the CPU and large cache memory data fetching operations, creates a bottleneck whereby, the GPU ends up in an idle state, waiting for processed input data.

An ideal solution would be to create multiple threads to handle different operations. Here, both data generation and minibatch training would occur concurrently so that the next input data is provided always available on demand (parallelization). The architecture, however, is incompatible with the thread safe data generators within the machine learning environment. During actual operation (after training), all these sub processing will be absent (no sifting through large

datasets stored in cache and no expected output to be prepared) which will significantly improve prediction time.

The loss curve and accuracy rate curve were selected to describe the performance of the trained CNN model. The abscissa (horizontal axis) of the following graphs is the training iteration or epoch and the ordinate (vertical axis) represents the parameter or metric (i.e. epoch loss, prediction accuracy) being monitored during training.

5.4.1 Epoch Loss

The figure below represents breakdown of the computed loss per epoch which is mean value of the loss per steps in each epoch.

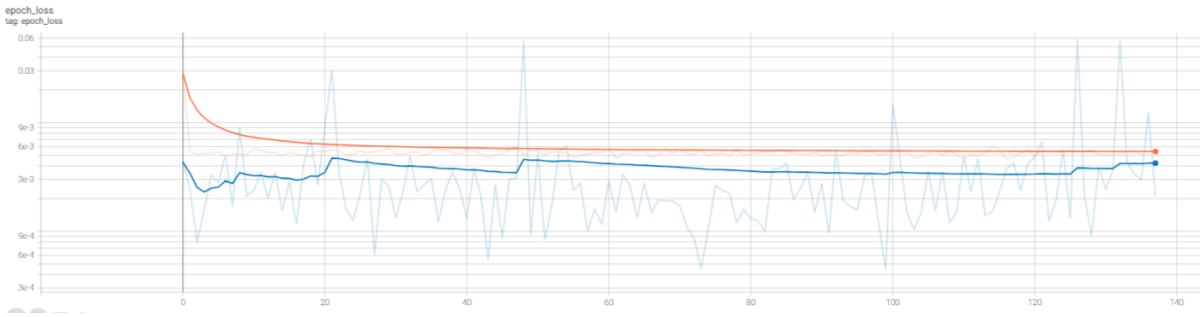


Figure 32 Graph of training and validation magnitude loss.

It is observed that both training and validation loss assume relatively high values (0.02855 and 0.0043613 respectively) at the initial stage of the training process. At this point, initialized weights of the network have assumed values which sparsely represent the input data; hence, the model produces outputs that deviate significantly from the expected values. As epochs progress, the curve begins to plateau or flatten out indicating that the loss value is approaching a local or global minimum. Validation epoch loss is monitored to check overfitting of the model. After 138 epochs, it is seen the validation loss failed to improve from 0.0045491 which means the model's best ability to generalize has been achieved with the current architecture of the model.

5.4.2 Epoch Location Mean Absolute Error (MAE)

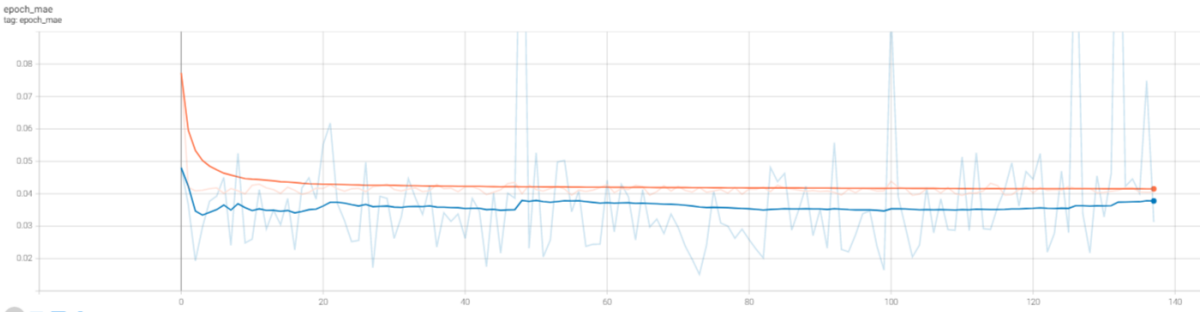


Figure 33 Graph of Epoch Location MAE

It is observed that both training and validation loss assume relatively high values (0.07727 and 0.004802 respectively) at the initial stage of the training process. At this point, initialized weights of the network have assumed values which sparsely represent the input data; hence, the model produces outputs that deviate significantly from the expected values just like the mean squared error loss discussed above the sparse initialization of network weights also affects the initial value however this value quickly settles with the help of the sufficient number of training iterations available to the AMSgrad optimizer. As epochs progress, the curve settled at a value of indicating that the cessation of learning. 0.044802 or flatten out indicating that the cessation of learning value.

5.5 Testing

5.5.1 Testing of RFR Model

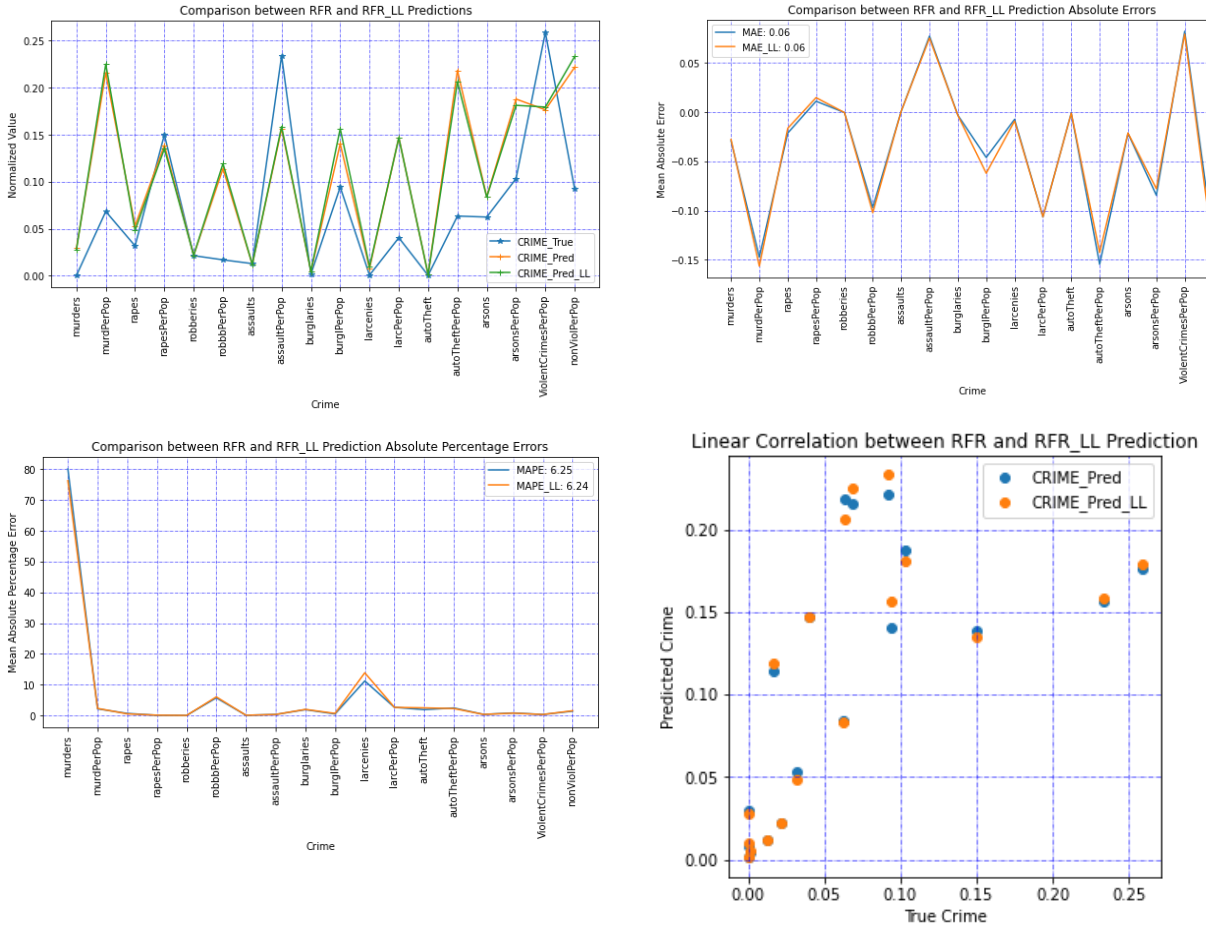


Figure 34 Graphs depicting the performance of the RFR model during testing.

The prediction capability of the RFR model was tested using data of three cities and from the data set with or without geolocation information and the results has been displayed above. A comparison between the prediction capabilities of the RFR with geolocation considered in the features of the data set used for training and RFR with geolocation excluded was provided. The model performed inherently the same in every instance therefore the geographical information had no significant impact on the prediction accuracy of the model. The mean square absolute error of the model varied from city to city depending on the type of crime depicting dynamic and distinct scenarios every city presents in the data which truly tested the performance of the model.

5.5.2 Testing of Mo-RF Model

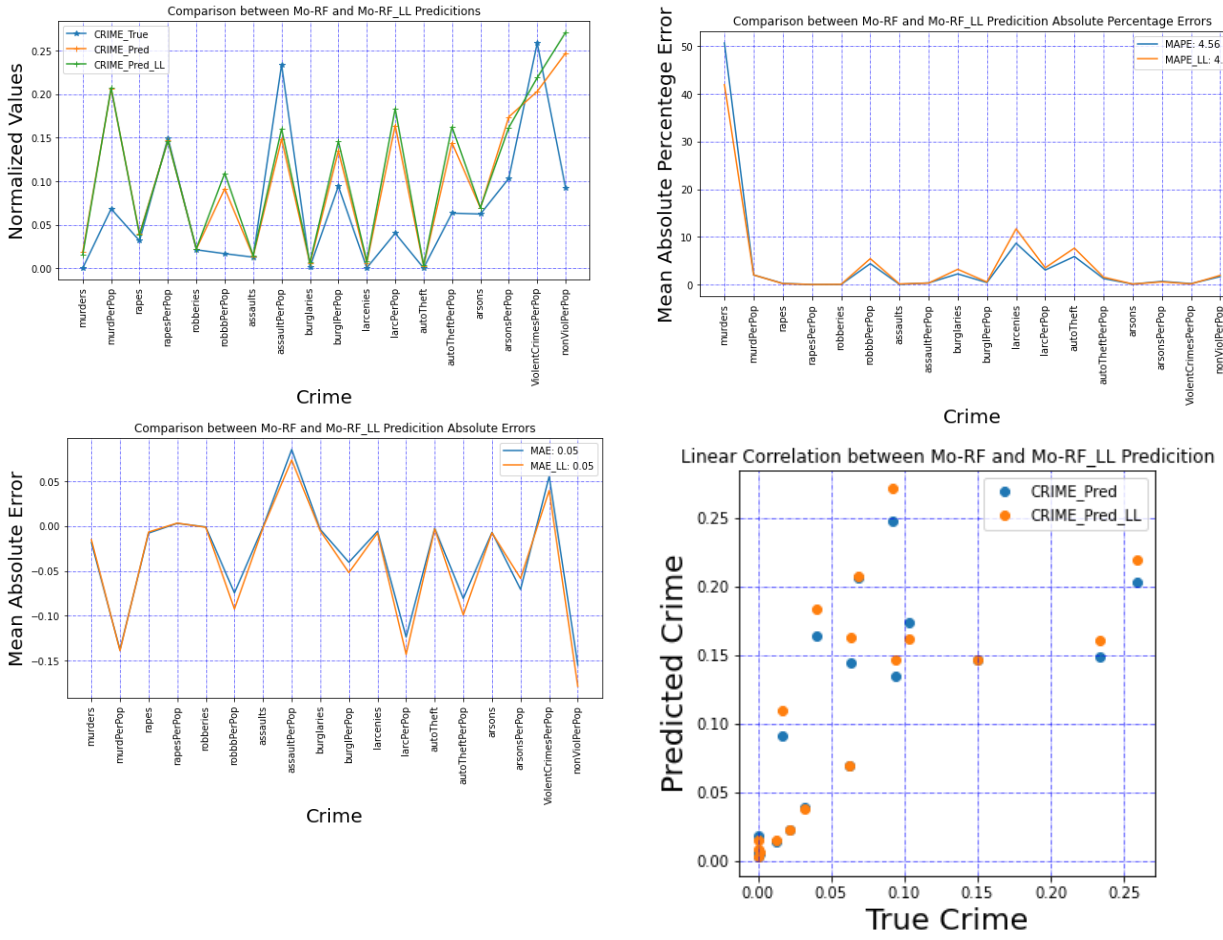


Figure 35 Graphs depicting the performance of the RFR model during testing.

The prediction capability of the Mo-RF model was tested using the data of two cities and from the data set with or without geolocation information during training and the results has been displayed in the diagrams above. A comparison between the prediction capabilities of the Mo-RF with geolocation considered in the features of the data set used for training and Mo-RF with geolocation excluded was also provided above. It was observed that the geolocation information the same had no significant impact on the prediction accuracy of the model with an average MAE of 0.05 in both scenarios. The mean square error of the model varied from city to city depending on the type of crime occurrence depicting dynamic and distinct scenarios every city presents in the data which truly tested the performance of the model. The Mo-RF model with an average MAPE of 4.56, the model outperforms the RFR model.

5.5.3 Testing of SVM Model

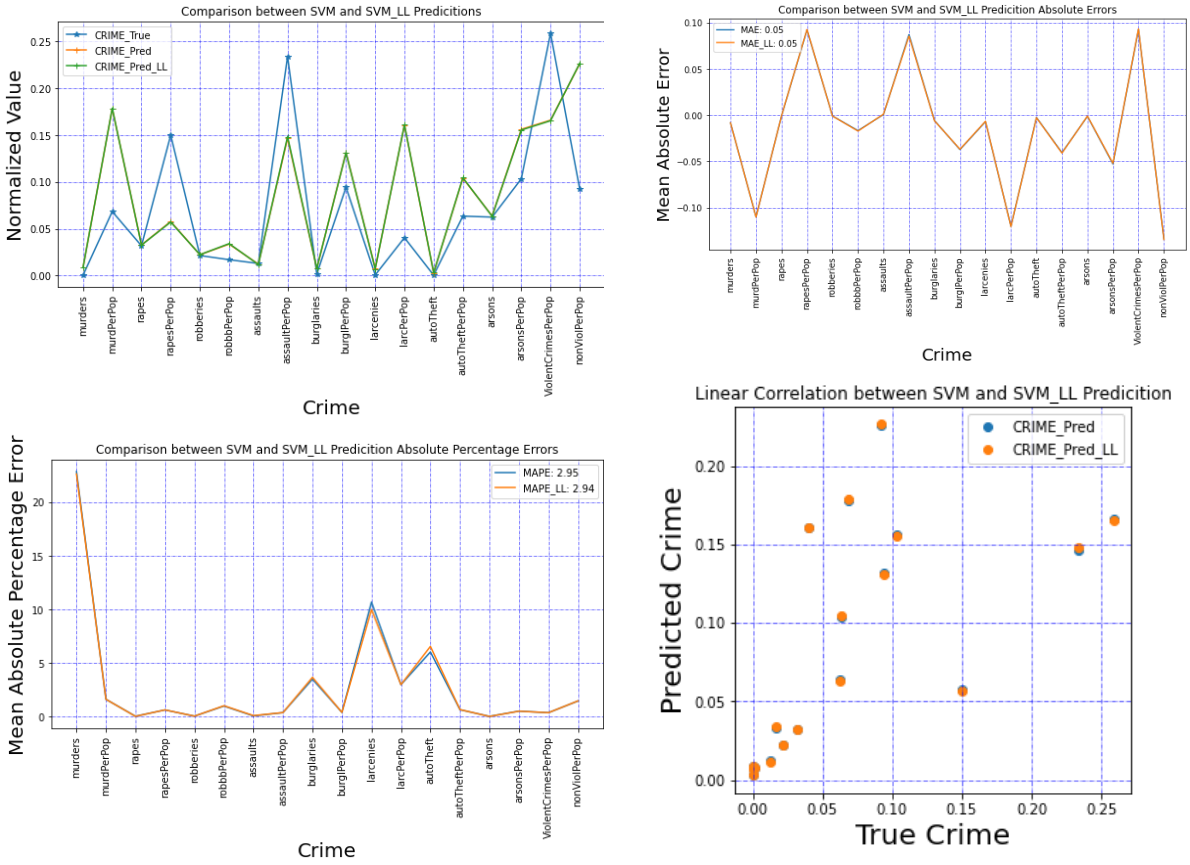


Figure 36 Graphs depicting MAE and MAP values, a scatter diagram of predicted crimes and true crimes to assess the prediction accuracy of the SVM models for when geolocation information was considered and when not considered.

The prediction capability of the SVM model was tested using the data of three cities and from the data set with or without geolocation information during training and the results and a scattered of the pre has been displayed in the diagrams above. A comparison between the prediction capabilities of the SVM with geolocation taken into consideration in the features of the data set used for training and SVM with geolocation excluded was also provided above. As depicted in the diagram that the geolocation information the same had no significant impact on the prediction accuracy of the model with an average MAE of 0.05 in both scenarios. The mean square error of the model varied from city to city depending on the type of crime occurrence depicting

dynamic and distinct scenarios every city presents in the data which truly tested the performance of the model. The SVM model with an average MAPE of 2.95, the model outperforms the Mo-RF model.

5.5.4 Testing of Transformer Model

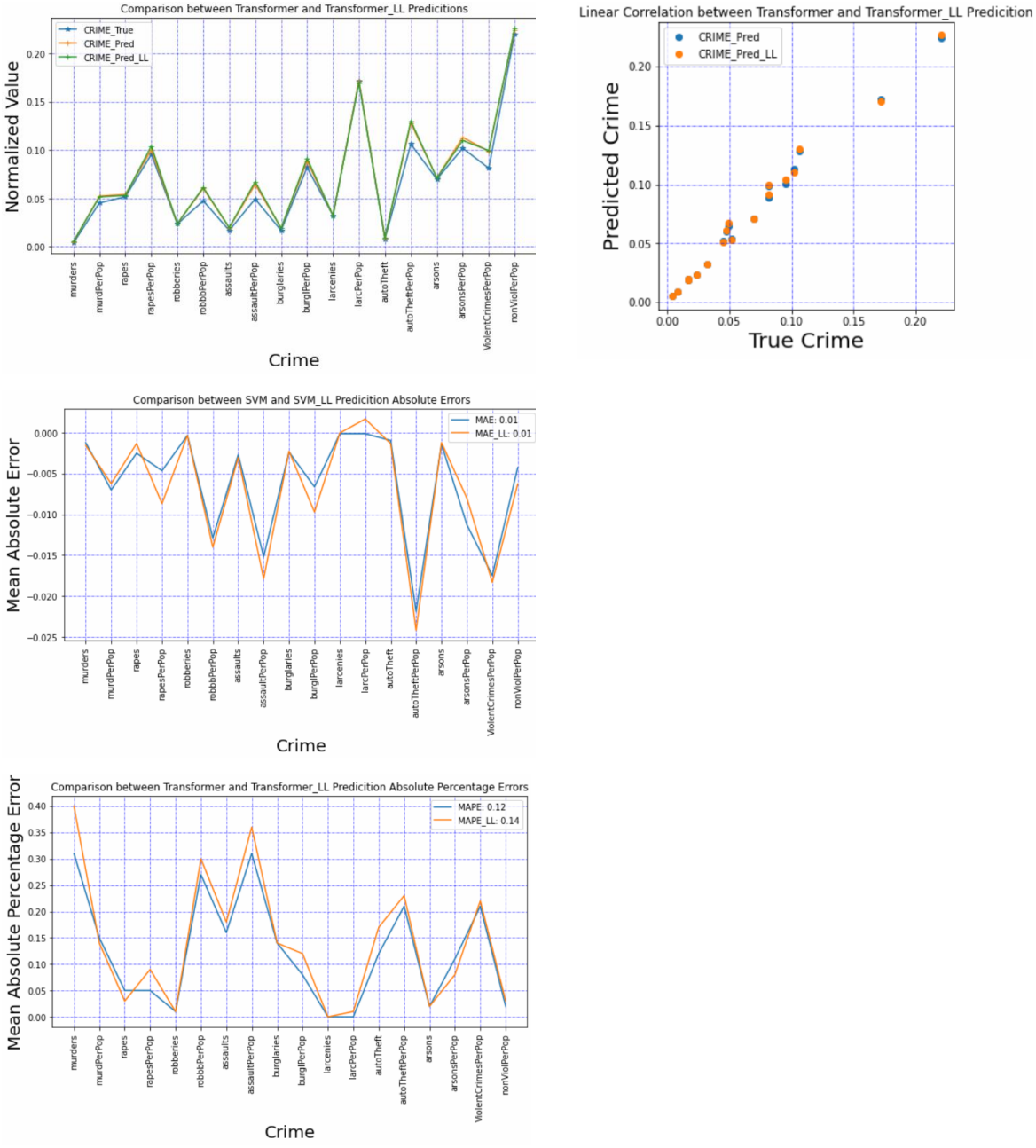


Figure 37 Graphs depicting MAE and MAP values, a scatter diagram of predicted crimes and true crimes to assess the prediction accuracy of the SVM models for when geolocation information was considered and when not considered.

The prediction capability of the Transformer model was tested using the data of three cities and from the data set with or without geolocation information during training and the results and a scattered of the predicted crime versus true crime has been displayed in the diagrams above. A comparison between the prediction capabilities of the Transformer Architecture with geolocation taken into consideration in the features of the data set used for training and Transformer Architecture with geolocation excluded was also provided above. As depicted in the diagram that the geolocation information the same had a significant or huge impact on the prediction accuracy of the model with an average MAE of 0.01 in both scenarios drastically reducing the error of prediction. The transformer model took into account the collinearities between urban features to accurate design the interdependencies between urban features and crime types due its effective encoder-decoder layer to accurately model the crime data, an advantage the SVM, Mo-RF, and RFR could not effectively model. The mean square error of the model varied from city to city depending on the type of crime occurrence depicting dynamic and distinct scenarios every city presents in the data which truly tested the performance of the model. The SVM model with an average MAPE of 0.14, the model outperforms the three conventional models: SVM, Mo-RF, RFR.

5.5.4 Testing of Transformer Model

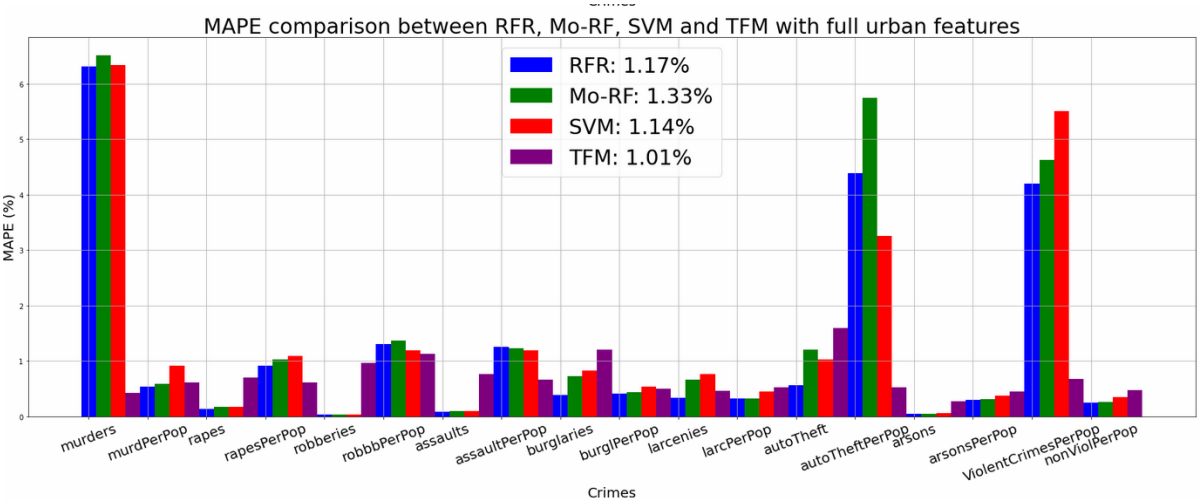


Figure 38 A bar chart showing the comparison between the SFR, Mo-RF, SVM and Transformer (TFM) models with full urban features including geolocation information.

From the graph above, the Transformer Architecture performed better than the conventional artificial intelligence techniques modelled in this thesis with a MAPE of 1.01%. The MAPE

for murders which was usually high for the SVM, Mo-RF and RFR models, was drastically reduced by the transformer architecture contributing the lower average MAPE value.

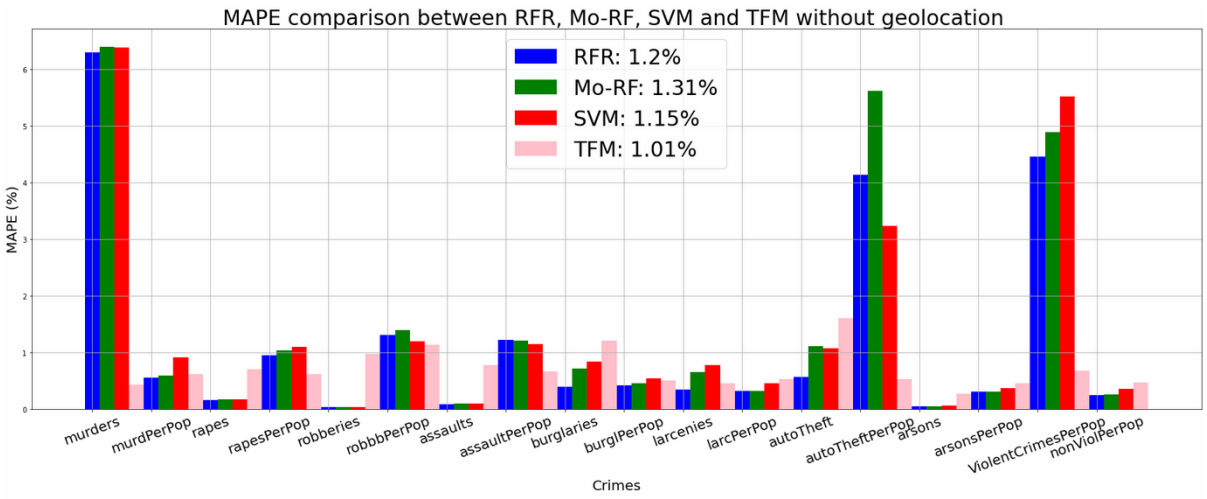


Figure 39 A Graph showing a comparison between the RFR, Mo-RF, SVM without geolocation.

From the graph above, with geolocation information excluded from the data set before training, the Transformer Architecture performed better than the SVM, RFR and Mo-RF techniques modelled in this thesis. The transformer model effectively reduced the overall MAPE since it utilized the spatial information provided the geolocation information to reduce murder MAPE since murder was one feature significantly contributing to an increased overall MAPE.

CHAPTER FIVE

CONCLUSION

A Transformer architecture has been presented for predicting various crimes given a set of urban features. Various methods have been proposed which have their shortcomings from over-dependence on large data, poor accuracy, large networks, inability to properly identify and model the collinearities between urban features and crime types and lack of geographical information. The proposed technique was designed to resolve these and other problems with these schemes.

The model is able to achieve a MAPE of 1.01% and accurately predicts future crime occurrences based on certain crime features given to the model as inputs. The architecture proposed in this thesis accurately models the collinearities between urban features and the types of crime to accurately understand the relationships between urban features to accurately future crime occurrences.

The large number of input features makes the model robust even with insufficient crime features. The significantly reduced number of parameters of this model reduces training time and enables faster development towards deployment with the help of GPU accelerated computing.

The results also show that it is possible to use the Transformer architecture for non-time-series data although significant effort is required for hyperparameter tuning as opposed to typical use cases.

With the rising crime rates in Ghana over the years, the Transformer Architecture model developed in this thesis can be utilized by Ghana Police Service and other law enforcement agencies in Ghana for effective distribution of security resources in the various regions in Ghana to curb serious impact of crime in the society.

Further Works

While satisfactory experimental results have been achieved by the proposed technique, further testing and optimization of hyper parameters is still necessary to improve training time and accuracy to achieve the best performance possible.

REFERENCES

Alves, L. G. A., Ribeiro, H. V., & Rodrigues, F. A. (2018). Crime prediction through urban metrics and statistical learning. *Physica A: Statistical Mechanics and Its Applications*, *505*, 435–443. https://doi.org/10.1016/j.physa.2018.03.084

Aning, E. K. (2006). *An Overview Of The Ghana Police Service*. 37.

Annan, K. (2005). "In Larger Freedom": Decision Time at the UN. *Foreign Affairs*, *84*(3), 63. https://doi.org/10.2307/20034350

Antolos, D., Liu, D., Ludu, A., & Vincenzi, D. (2013). Burglary Crime Analysis Using Logistic Regression. In S. Yamamoto (Ed.), *Human Interface and the Management of Information. Information and Interaction for Learning, Culture, Collaboration and Business*, (Vol. 8018, pp. 549–558). Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-39226-9_60

Bogomolov, A., Lepri, B., Staiano, J., Oliver, N., Pianesi, F., & Pentland, A. (2014). *Once Upon a Crime: Towards Crime Prediction from Demographics and Mobile Data* (arXiv:1409.2983). arXiv. http://arxiv.org/abs/1409.2983

Chen, X., Cho, Y., & Jang, S. Y. (2015). Crime prediction using Twitter sentiment and weather. *2015 Systems and Information Engineering Design Symposium*, 63–68. https://doi.org/10.1109/SIEDS.2015.7117012

Defferrard, M., Bresson, X., & Vandergheynst, P. (2016). Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in Neural Information Processing Systems*, *29*.

Dogbevi, E. (2018, March 24). 25 ways to reduce crime in Ghana. *Ghana Business News*. https://www.ghanabusinessnews.com/2018/03/24/25-ways-to-reduce-crime-in-ghana/

Flaxman, S. R. (n.d.). *A General Approach to Prediction and Forecasting Crime Rates with Gaussian Processes*. 32.

Ghana public safety and crime report – FIRST HALF 2021. (2021). 16.

Hajela, G., Chawla, M., & Rasool, A. (2020). A Clustering Based Hotspot Identification Approach For Crime Prediction. *Procedia Computer Science*, *167*, 1462–1470. https://doi.org/10.1016/j.procs.2020.03.357

Han, K., Xiao, A., Wu, E., Guo, J., XU, C., & Wang, Y. (2021). Transformer in Transformer. *Advances in Neural Information Processing Systems*, *34*, 15908–15919. https://proceedings.neurips.cc/paper/2021/hash/854d9fca60b4bd07f9bb215d59ef5561-Abstract.html

IEEE Xplore Full-Text PDF: (n.d.). Retrieved June 14, 2022, from https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=6906719&casa_token=LJjDm3UuMkkAAAAA:r3DAVOPbUyxWRV2MTi-XB0xQqnEV6wnTEFuKU-XYuwD8SFqALN3XN9I9FRrzeWOH3xybg1kA1Fo&tag=1

Islam, K., & Raza, A. (n.d.). *Forecasting Crime Using ARIMA Model*. 14.

Ivan, N., Ahishakiye, E., Omulo, E. O., & Taremwa, D. (2017). *Crime Prediction Using Decision Tree (J48) Classification Algorithm*. https://idr.kab.ac.ug/xmlui/handle/20.500.12493/113

Kupilik, M., & Witmer, F. (2018). Spatio-temporal violent event prediction using Gaussian process regression. *Journal of Computational Social Science*, *1*. https://doi.org/10.1007/s42001-018-0024-y

[No title found]. (n.d.). *International Journal of Research Publication and Reviews*.

Oh, G., Song, J., Park, H., & Na, C. (2021). Evaluation of Random Forest in Crime Prediction: Comparing Three-Layered Random Forest and Logistic Regression. *Deviant Behavior*, 1–14. https://doi.org/10.1080/01639625.2021.1953360

Oteng-Ababio, M., Owusu, G., Wrigley-Asante, C., & Owusu, A. (2016). Longitudinal analysis of trends and patterns of crime in Ghana (1980–2010): A new perspective. *African Geographical Review*, *35*(3), 193–211. https://doi.org/10.1080/19376812.2016.1208768

Ozgul, F., Atzenbeck, C., Celik, A., & Erdem, Z. (2011). Incorporating data sources and methodologies for crime data mining. *Proceedings of 2011 IEEE International Conference on Intelligence and Security Informatics*, 176–180. https://doi.org/10.1109/ISI.2011.5983995

Perez, L., & Wang, A. (n.d.). *Gaussian Processes for Crime Prediction*. 8.

Routine Activity Theory—Miró—- Major Reference Works—Wiley Online Library. (n.d.). Retrieved June 7, 2022, from https://onlinelibrary.wiley.com/doi/pdf/10.1002/9781118517390.wbetc198

Safat, W., Asghar, S., & Gillani, S. A. (2021). Empirical Analysis for Crime Prediction and Forecasting Using Machine Learning and Deep Learning Techniques. *IEEE Access*, *9*, 70080–70094.

Sathyadevan, S., Devan, M. S., & Gangadharan, S. S. (2014). Crime analysis and prediction using data mining. *2014 First International Conference on Networks & Soft Computing (ICNSC2014)*, 406–412. https://doi.org/10.1109/CNSC.2014.6906719

School of Computing, the University of Southern Mississippi, Hattiesburg, MS, U.S.A., Nguyen, T. T., Hatua, A., & Sung, A. H. (2017). Building a Learning Machine Classifier with Inadequate Data for Crime Prediction. *Journal of Advances in Information Technology*, 141–147. https://doi.org/10.12720/jait.8.2.141-147

Shamsuddin, N. H. M., Ali, N. A., & Alwee, R. (2017). An overview on crime prediction methods. *2017 6th ICT International Student Project Conference (ICT-ISPC)*, 1–5. https://doi.org/10.1109/ICT-ISPC.2017.8075335

Singh, S., & Mahmood, A. (2021). The NLP Cookbook: Modern Recipes for Transformer Based Deep Learning Architectures. *IEEE Access*, *9*, 68675–68702. https://doi.org/10.1109/ACCESS.2021.3077350

Sivanagaleela, B., & Rajesh, S. (2019). Crime Analysis and Prediction Using Fuzzy C-Means Algorithm. *2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI)*, 595–599. https://doi.org/10.1109/ICOEI.2019.8862691

Stec, A., & Klabjan, D. (2018). *Forecasting Crime with Deep Learning* (arXiv:1806.01486). arXiv. http://arxiv.org/abs/1806.01486

Sui, Y., Wu, G., & Sanner, S. (2021). *Multi-axis Attentive Prediction for Sparse EventData: An Application to Crime Prediction* (arXiv:2110.01794). arXiv. http://arxiv.org/abs/2110.01794

Vural, M., & Gök, M. (2017). Criminal prediction using Naive Bayes theory. *Neural Computing and Applications*, *28*. https://doi.org/10.1007/s00521-016-2205-z

Yahya, A. A., & Osman, A. (2019). Using Data Mining Techniques to Guide Academic Programs Design and Assessment. *Procedia Computer Science*, *163*, 472–481. https://doi.org/10.1016/j.procs.2019.12.130

APPENDIX

APPENDIX I – Python Scripts for Project implementation

```
# %%
from __future__ import absolute_import, division, print_function, unicode_literals

import matplotlib
from scipy import signal
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import numpy.ma as ma
import pandas as pd
import os
from sklearn.preprocessing import MinMaxScaler, QuantileTransformer, RobustScaler,
StandardScaler
from sklearn.metrics import mean_squared_error
from numpy import genfromtxt
from sklearn.model_selection import train_test_split
# !pip install cupy
# import cupy
import pylab as pl
import seaborn as sns
from pathlib import Path
import shutil
from tqdm.notebook import tqdm_notebook
# !pip install googlemaps
# import googlemaps
import sys
import math

# %%
# %%
import tensorflow.keras.backend as K
# !pip install tensorflow-addons
import tensorflow_addons as tfa
import tensorflow_probability as tfp
from tensorflow.keras import layers
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential, load_model, Model
from tensorflow.keras.optimizers import RMSprop,Adam,SGD,Adadelta,Adagrad,Adamax
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, TensorBoard, Re-
duceLROnPlateau
from tensorflow.compat.v1.keras.backend import set_session
```

```
from tensorflow.keras.utils import plot_model, to_categorical, normalize
from tensorflow.python.ops.numpy_ops import np_config

np_config.enable_numpy_behavior()
np.set_printoptions(threshold=sys.maxsize)

# %%
# tf.debugging.experimental.enable_dump_debug_info('.', tensor_de-
bug_mode="FULL_HEALTH", circular_buffer_size=-1)
# tf.debugging.set_log_device_placement(True)
from tensorflow.python.client import device_lib
try:
  device_name = tf.test.gpu_device_name()
  if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
  print('Found GPU at: {}'.format(device_name))

  config = tf.compat.v1.ConfigProto()
  config.gpu_options.allow_growth = True
  config.gpu_options.per_process_gpu_memory_fraction = 0.1
  sess = tf.compat.v1.InteractiveSession(config=config)
  set_session(sess)
  print(device_lib.list_local_devices())
  gpus = tf.config.experimental.list_physical_devices('GPU')
  for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
  print("Num GPUs Available: ", len(gpus))

except Exception as error:
  print("error trying to configure computing device")
  print(error)

# %% [markdown]
# This was developed using Python 3.6 (Anaconda) and package versions:

# %%
from tensorflow.python.platform import build_info as tf_build_info
print("Tensorflow verison: ",tf.__version__)
print("CUDA verison: ", tf_build_info.build_info['cuda_version'])
print("CUDNN verison: ", tf_build_info.build_info['cudnn_version'])

# %%
# tf.keras.backend.floatx()
# tf.keras.backend.set_floatx('float16')
# tf.keras.mixed_precision.experimental.set_policy('mixed_float16')
tf.keras.backend.floatx()


# %% [markdown]
```

```
# ### Load  Data
#
#

# %%
file=r'crimeDataFixed.csv'

myData = pd.read_csv(file, delimiter=',', index_col=0) # usecols = ['Va', ' Vb', ' Vc', ' Ia', ' Ib
', ' Ic ', ' dIa ', ' dIb ',' dIc ', ' dOmega_elec', ' Ia_', ' Ib_ ', ' Ic_ ', ' Te', ' Omega_elec', '
cTheta_elec', ' sTheta_elec'])
# myData.round(decimals=6)
# myData=myData.astype(np.float32)
# myData=myData.astype(np.float16)
myData.describe()

# %%
# myData.convert_dtypes('float16')
myData.dtypes, myData.shape


# %% [markdown]
# List of the variables used in the data-set.

# %% [markdown]
# These are the top rows of the data-set.

# %%
data_top = myData.columns.values
data_top

# %%
myData.head(20)

# %%
myData.values.astype(np.float32)
myData.shape



# %%
input_names = myData.columns[:-18]
target_names = myData.columns[-18:]

# %%
# # tfp.stats.correlation( newFilled, y=None, sample_axis=0, event_axis=-1, keepdims=False,
name=None)
# myData_corr = myData.corr()[target_names][:-18]
# myData_corr.where(np.tril(np.ones(myData_corr.shape)).astype(np.bool_)).style.back-
ground_gradient(cmap='coolwarm').set_properties(**{'font-size': '5'})
```

```
# myData_corr.style.background_gradient(cmap='coolwarm').set_properties(**{'font-size':
'5'})


# # 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# # newFilled_corr.describe()

# %% [markdown]
# # tfp.stats.correlation( myDataClean, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
# new_corr = pd.concat([myDataClean_corr, myDataDirty_corr], axis=1).corr()
# new_corr = new_corr.loc[:,~new_corr.columns.duplicated()]
# new_corr = new_corr.iloc[:-int(len(new_corr)/2)]
# new_corr = new_corr.where(np.tril(np.ones(new_corr.shape)).astype(np.bool_))
# new_corr.style.background_gradient(cmap='coolwarm').set_properties(**{'font-size': '5'})
# # 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# # new_corr.describe()

# %% [markdown]
# ### Missing Data
#
#
#
# Because we are using resampled data, we have filled in the missing values with new values
that are linearly interpolated from the neighbouring values, which appears as long straight
lines in these plots.
#
# This may confuse the neural network. For simplicity, we will simply remove these two sig-
nals from the data.
#
# But it is only short periods of data that are missing, so you could actually generate this data
by creating a predictive model that generates the missing data from all the other input signals.
Then you could add these generated values back into the data-set to fill the gaps.

# %%
# plt.figure(figsize=(30,5*16))
# myData.plot(subplots=True, figsize=(30,5*16))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.show()

# %%
# plt.figure(figsize=(25,5))
# myData['Ia'].plot()
# plt.grid(color='r', linestyle='-.', linewidth=0.5)
# plt.show()

# plt.figure(figsize=(25,5))
# myData['Ib'].plot()
# plt.grid(color='y', linestyle='-.', linewidth=0.5)
```

```
# plt.show()

# plt.figure(figsize=(25,5))
# myData['Ic'].plot()
# plt.grid(color='g', linestyle='-.', linewidth=0.5)
# plt.show()

# %% [markdown]
# ### Target Data for Prediction
#
# We will try and predict the future Forex-data.

# %%
input_names = myData.columns[:-18]
target_names = myData.columns[-18:]
# input_names = myData.columns[:]
# target_names = myData.columns[:]

# %% [markdown]
# We will try and predict these signals.

# %%
df_targets = myData[target_names]
df_targets

# %% [markdown]
# ### NumPy Arrays
#
# We now convert the Pandas data-frames to NumPy arrays that can be input to the neural network. We also remove the last part of the numpy arrays, because the target-data has `NaN` for the shifted period, and we only want to have valid data and we need the same array-shapes for the input- and output-data.
#
# These are the input-signals:

# %%
x_data = myData[input_names].values.astype(np.float32)
x_data, x_data.dtype, np.isinf(x_data).any(), np.isnan(x_data).any()

# %%
print(type(x_data))
print("Shape:", x_data.shape)

# %% [markdown]
# These are the output-signals (or target-signals):

# %%
y_data = df_targets.values.astype(np.float32, casting='unsafe')
```

```
y_data, y_data.dtype, np.isinf(y_data).any(), np.isnan(y_data).any()

# %%
print(type(y_data))
print("Shape:", y_data.shape)

# %% [markdown]
# This is the number of observations (aka. data-points or samples) in the data-set:

# %%
num_data = len(x_data)
num_data

# %% [markdown]
# This is the fraction of the data-set that will be used for the training-set:

# %%
batch_size = 10
train_split = 0.8
num_train = int(train_split * num_data)
num_val = int(0.5*(num_data - num_train))
num_test = (num_data - num_train) - num_val
#steps_per_epoch = int((num_train/batch_size)/40)
steps_per_epoch=1
#train_validation_steps = int((num_val/batch_size))
train_validation_steps = 1
#test_validation_steps = int((num_test/batch_size))
test_validation_steps = 1
print('num_train:',num_train, 'num_val:',num_val, 'num_test:',num_test)
print('steps_per_epoch:', steps_per_epoch)
print('train_validation_steps:', train_validation_steps, 'test_validation_steps:', test_valida-
tion_steps)

# %%
x_scaler = MinMaxScaler().fit(myData[input_names].values.astype(np.float32))
y_scaler = MinMaxScaler().fit(myData[target_names].values.astype(np.float32))

x_data_scaled = x_scaler.transform(x_data) + 1e-5
y_data_scaled = y_scaler.transform(y_data) + 1e-5

x_train, x_test, y_train, y_test = train_test_split(x_data_scaled, y_data_scaled,
train_size=train_split, random_state=None, shuffle=True)
# x_train, x_test, y_train, y_test = train_test_split(x_data, y_data, train_size=train_split, ran-
dom_state=None, shuffle=True)


# %% [markdown]
# This is the number of observations in the training-set:
```

```
# %%
num_train = len(x_train)
num_train

# %% [markdown]
# This is the number of observations in the test-set:

# %%
num_test = len(x_test)
num_test

# %% [markdown]
# These are the input-signals for the training- and test-sets:

# %%
len(x_train) + len(x_test)

# %% [markdown]
# These are the output-signals for the training- and test-sets:

# %%
len(y_train) + len(y_test)

# %% [markdown]
# This is the number of input-signals:

# %%
num_x_signals = x_data.shape[1]
num_x_signals

# %% [markdown]
# This is the number of output-signals:

# %%
num_y_signals = y_data.shape[1]
num_y_signals

# %% [markdown]
# ### Scaled Data
#
# The data-set contains a wide range of values:

# %%
print('x_train min:', x_train.min())
print('x_train max:', x_train.max())

print('y_train min:', y_train.min())
print('y_train max:', y_train.max())
```

```
print('x_test min:', x_test.min())
print('x_test max:', x_test.max())

print('y_test min:', y_test.min())
print('y_test max:', y_test.max())

# %% [markdown]
# ## Data Generator
#
# The data-set has now been prepared as 2-dimensional numpy arrays. The training-data has almost 300k observations, consisting of 20 input-signals and 3 output-signals.
#
# These are the array-shapes of the input and output data:

# %%
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)
print('x_test shape:', x_test.shape)
print('y_test shape:', y_test.shape)

# %%

batch_size = 32
sequence_length = 1
mask_percentage =.05


class CustomDataGen(tf.keras.utils.Sequence):

    def __init__ (self, x_data, y_data, batch_size=None, sequence_length=None, train=True, validation=True, mask_percentage=0.01, random_batch=False, random_idx=False):

        self.x_train = x_data[0]
        self.x_test = x_data[1]
        self.y_train = y_data[0]
        self.y_test = y_data[1]
        self.batch_size = batch_size
        self.sequence_length = sequence_length
        self.train = train
        self.validation = validation
        self.random_batch = random_batch
        self.random_idx = random_idx
        self.mask_percentage = mask_percentage
        self.n = int(self.x_train.shape[0])

    def on_epoch_end(self):
        #do nothing
        return
```

```
def __getitem__(self, index):
    if self.train:
        # print('using train samples')
        x_samples = self.x_train
        y_samples = self.y_train
        self.n = x_samples.shape[0]

    elif self.validation:
        # print('using validation samples')
        x_samples = self.x_test[:num_val]
        y_samples = self.y_test[:num_val]
        self.n = x_samples.shape[0]
    else:
        # print('using test samples')
        x_samples = self.x_test[-num_test:]
        y_samples = self.y_test[-num_test:]
        self.n = x_samples.shape[0]

        # Allocate a new array for the batch of input-signals.
    if self.train:
        # sequence_length_ = np.random.randint(1,self.sequence_length)
        sequence_length_ = self.sequence_length
    else:
        sequence_length_ = self.sequence_length

    if self.random_batch:
        batch_size_ = np.random.randint(1,self.batch_size)
    else:
        batch_size_ = batch_size

    x_shape = (batch_size_, x_samples.shape[1])
    y_shape = (batch_size_, y_samples.shape[1])
    x_batch = np.zeros(shape=x_shape, dtype=np.float32)
    y_batch = np.zeros(shape=y_shape, dtype=np.float32)

    # Fill the batch with random sequences of data.
    for i in range(batch_size_):
        # Get a random start-index.

        if self.random_idx:
            sample_idx = np.random.randint(1, x_samples.shape[-2])

        # This points somewhere into the training-data.
        x_batch[i] = x_samples[sample_idx]
        y_batch[i] = y_samples[sample_idx]

    # return np.ma.expand_dims(x_batch, axis=0), np.ma.expand_dims(y_batch, axis=0)
```

```
        return x_batch, y_batch

    def __len__(self):
        return int(self.n / self.batch_size)


x_train_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=True, validation=False,
mask_percentage=mask_percentage, random_batch=False, random_idx=True)
x_train_batch, y_train_batch=x_train_generator.__getitem__(1)

print('x_train shape: ', x_train_batch.shape, 'x_train dtype:', x_train_batch.dtype)
print('y_train shape: ', y_train_batch.shape, 'y_train dtype:', y_train_batch.dtype)

x_val_generator = CustomDataGen((x_train, x_test), (y_train, y_test), batch_size=batch_size,
sequence_length=sequence_length, train=False, validation=True, mask_percent-
age=mask_percentage, random_batch=False, random_idx=True)
x_val_batch, y_val_batch=x_val_generator.__getitem__(1)

print('x_val shape: ', x_val_batch.shape, 'x_val dtype:', x_val_batch.dtype)
print('y_val shape: ', y_val_batch.shape, 'y_val dtype:', y_val_batch.dtype)

x_test_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=False, validation=False,
mask_percentage=mask_percentage, random_batch=False, random_idx=True)
x_test_batch, y_test_batch=x_test_generator.__getitem__(1)

print('x_test shape: ', x_test_batch.shape, 'x_test dtype:', x_test_batch.dtype)
print('y_test shape: ', y_test_batch.shape, 'y_test dtype:', y_test_batch.dtype)



# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_train_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_train_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_val_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
```

```
# seq = y_val_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_test_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_test_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

np.isnan(x_train_batch).any(), np.isnan(x_val_batch).any(), np.isnan(x_test_batch).any()

# %%

# %%
learning_rate = 1e-4
weight_decay = 1e-4
dropout_rate = 0.25
projection_dim = 1                  # Dimension of the patch representation or embedding to be used (non-restrictive) in this case we are assuming that our embedding has the same dimensionality as our features which means we could technically sckip the embedding layer altogether.
num_heads = 4                       # Total number of differnt copies of Q K V matrices. These will be aggregated as you move from one layer to the next.
transformer_units = [projection_dim*4, projection_dim]
transformer_layers = 4              # Total number of complete transformer blocks or layers to stack (non-restrictive). Mine computation, network size, GPU memory, speed, etc.
mlp_head_units = [127, 90]          # This is your model output head (non-restrictive). Size and activation functions depend on task to be performed.


# %% [markdown]
# ## Implement Multilayer Perceptron (MLP)

# %%
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation='gelu')(x)
        x = layers.Dropout(dropout_rate)(x)
    return x

# %% [markdown]
def create_model():
```

```
    inputs = keras.Input(name='InputLayer', shape=(num_x_signals))

    # Create Embedding.
    # embeddings = layers.Embedding(input_dim=num_x_signals, output_dim=projec-
tion_dim)(inputs)
    embeddings = tf.expand_dims(inputs, axis=-1)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = LayerNormalization(epsilon=5e-5, center=True, scale=True, axis=-1)(embed-
dings)                      # Normalize input
        # print(f'x1 shape {x1.shape}')
        # Create a multi-head attention layer.
        attention_output = layers.MultiHeadAttention(num_heads=num_heads,
                                             key_dim=projection_dim,
                                             dropout=dropout_rate)(x1, x1)

        # print(f'Attention Unit {i+1} Output shape {attention_output.shape}')                #
Don't undertand the (x1, x1) input

        # Skip connection 1.
        x2 = layers.Add()([attention_output, embeddings])                                # Elementwise
addition of attention_output matrix and initial or processed/received matrix of original en-
coded_patches
        # print(f'x2 shape {x2.shape}')

        # Layer normalization 2.
        x3 = LayerNormalization(epsilon=5e-5, center=True, scale=True, axis=-
1)(x2)                              # Normalize array
        # print(f'x3 shape {x3.shape}')

        # MLP.
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=drop-
out_rate)                    # Why not just a single layer having number of units equal to projec-
tion dimension?
        # print(f'x3 MLP output shape {x3.shape}')

        # Skip connection 2.
        embeddings = layers.Add()([x3, x2])                                        # Output of trans-
former block to be fed as an initial input to the next block or on to prediciton stage

    # Create a [batch_size, projection_dim] tensor.
    representation = LayerNormalization(epsilon=5e-5, center=True, scale=True, axis=-1)(em-
beddings)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(dropout_rate)(representation)
    # print(f'Transformer encoded representation shape {representation.shape}')
```

```
    # Add MLP.
    features = mlp(representation, hidden_units=mlp_head_units, dropout_rate=dropout_rate)

    # Add Output.
    outputs = Dense(units=num_y_signals,  activation='gelu', use_bias=False)(features)

    # Create the Keras model.
    model = keras.Model(inputs=inputs, outputs=outputs, name='Crime_Transformer_Model')

    # Create Optimizer.
    optimizer = Adam(learning_rate=1e-4, amsgrad=True)
    # moving_avg_Adam = tfa.optimizers.MovingAverage(optimizer)
    stocastic_avg_Adam = tfa.optimizers.SWA(optimizer)

    model.compile(loss='mse', optimizer=stocastic_avg_Adam, metrics=['mse', 'mae', 'mape',
'acc'], run_eagerly=True)

    return model

CRIME_model = create_model()

# CRIME_model.summary()

plot_model(CRIME_model, show_shapes=True, to_file='CRIME_model.png',
show_layer_names=True, rankdir='TB', expand_nested=True, dpi=50)


# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_test_batch[batch, :, signal_]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# seq = y_test_batch[batch, :, signal_]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

y_test_pred = CRIME_model.predict(x_test_batch, steps=1, verbose=1)
print('y_predict shape: ', y_test_pred.shape, 'y_predict dtype:', y_test_pred.dtype)
# seq = y_test_pred[signal_, :]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# CRIME_model.fit(x_test_batch,y_test_batch, epochs=10, verbose=1)
```

```
# %% [markdown]
# ### Callback Functions
#
# During training we want to save checkpoints and log the progress to TensorBoard so we create the appropriate callbacks for Keras.
# This is the callback for writing checkpoints during training.

# %%
path_checkpoint = r'CRIME_model_3.h5'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=False,
                                      restore_best_weights=True,
                                      save_best_only=True)

# %%
# %%
path_checkpoint_MA = r'CRIME_model_avg_MA.h5'
path_checkpoint_SWA = r'CRIME_model_avg_SWA.h5'

callback_MA = tfa.callbacks.AverageModelCheckpoint(filepath=path_checkpoint_MA,
                                                   update_weights=True)

callback_SWA = tfa.callbacks.AverageModelCheckpoint(filepath=path_checkpoint_SWA,
                                                   update_weights=True)


# %% [markdown]
# This is the callback for stopping the optimization when performance worsens on the validation-set.

# %%
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=400, verbose=1)

# %% [markdown]
# This is the callback for writing the TensorBoard log during training.

# %%
dirpaths = [Path('.\Tensorboard_3')]
for dirpath in dirpaths:
    if dirpath.exists() and dirpath.is_dir():
        try:
            shutil.rmtree(dirpath, ignore_errors=True)
            os.chmod(dirpath, 0o777)
            os.rmdir(dirpath)
            os.removedirs(dirpath)
            print("Directory '%s' has been removed successfully", dirpath)
```

```
        except OSError as error:
            print(error)
            print("Directory '%s' can not be removed", dirpath)


callback_tensorboard = TensorBoard(log_dir=r'TensorBoard_3',
                                   histogram_freq=1,
                                   write_graph=True)
                                  #  profile_batch = '300,320')

# %% [markdown]
# This callback reduces the learning-rate for the optimizer if the validation-loss has not improved since the last epoch (as indicated by `patience=0`). The learning-rate will be reduced by multiplying it with the given factor. We set a start learning-rate of 1e-3 above, so multiplying it by 0.1 gives a learning-rate of 1e-4. We don't want the learning-rate to go any lower than this.

# %%
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.999,
                                       min_lr=1e-7,
                                       patience=2,
                                       verbose=1)


# %%
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_SWA,
             callback_tensorboard,
             callback_reduce_lr]


# %% [markdown]
# #### Load weights from last checkpoint

# %%
filepath = r'CRIME_model_3.h5'
def train_model(resume, epochs, initial_epoch, batch_size, model):
    def fit_model():
        with tf.device('/device:GPU:0'):
            print(model.summary())
            history=model.fit(x_train_generator,
                              steps_per_epoch=steps_per_epoch,
                              epochs=EPOCHS,
                              verbose=1,
                              callbacks=callbacks,
                              validation_data=x_val_generator,
                              validation_steps=train_validation_steps,
```

```
                        #validation_freq=5,
                        #class_weight=None,
                        #max_queue_size=10,
                        #workers=8,
                        #use_multiprocessing=True,
                        #shuffle=True,
                        initial_epoch=initial_epoch)
            model.load_weights(path_checkpoint)
            model.save(filepath)
            model.evaluate(x_test_generator, steps=test_validation_steps)

            return history

    if resume:
        try:
            #del model
            model = load_model(filepath, custom_objects = {"CustomLoss": CustomLoss})
            # model.load_weights(path_checkpoint)
            # print(model.summary())
            print("Model loading....")
            model.evaluate(x_test_generator, steps=test_validation_steps)

        except Exception as error:
            print("Error trying to load checkpoint.")
            print(error)

    # Training the Model
    return fit_model()



# %%
EPOCHS = 2000

steps_per_epoch = int(num_train/batch_size)

for _ in range(1):
  try:
    CRIME_model.load_weights(r'CRIME_model_3.h5')
    CRIME_model.save(r'CRIME_model_3.h5')
    CRIME_model = load_model(r'CRIME_model_3.h5')
    print("Checkpoint Loaded.")
  except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)

# Train model
with tf.device('/device:GPU:0'):
```

```
    history = train_model(resume=False, epochs=EPOCHS, initial_epoch=0,
batch_size=batch_size, model=CRIME_model)
    CRIME_model.history

# %% [markdown]
# ## Performance on Test-Set

# %% [markdown]
# ### Load Checkpoint
#
# Because we use early-stopping when training the model, it is possible that the model's per-
formance has worsened on the test-set for several epochs before training was stopped. We
therefore reload the last saved checkpoint, which should have the best performance on the
test-set.

# %%
with tf.device('/device:GPU:0'):
    try:
        CRIME_model = load_model(r'CRIME_model_3.h5')
        # CRIME_model.load_weights(path_checkpoint)
    except Exception as error:
        print("Error trying to load checkpoint.")
        print(error)

# %% [markdown]
# We can now evaluate the model's performance on the test-set. This function expects a batch
of data, but we will just use one long time-series for the test-set, so we just expand the array-
dimensionality to create a batch with that one sequence.

# %%
with tf.device('/device:GPU:0'):
    CRIME_model.evaluate(x_train_generator, steps=5)
    CRIME_model.evaluate(x_val_generator, steps=5)
    CRIME_model.evaluate(x_test_generator, steps=5)

# %%
# %%
test_length = 30

# %%
with tf.device('/device:GPU:0'):
    try:
        CRIME_model = load_model(r'CRIME_model_3.h5')
        # CRIME_model.load_weights(r'CRIME_model.h5')
    except Exception as error:
        print("Error trying to load checkpoint.")
        print(error)
```

```
with tf.device('/device:GPU:0'):
    CRIME_model.evaluate(x_train_generator, steps=5)
    CRIME_model.evaluate(x_val_generator, steps=5)
    CRIME_model.evaluate(x_test_generator, steps=5)

with tf.device('/device:GPU:0'):
    mape = tf.keras.losses.MeanAbsolutePercentageError()

    y_masked = np.zeros(shape=(test_length,num_x_signals))
    y_pred = np.zeros(shape=(test_length,num_y_signals))
    y_true = np.zeros(shape=(test_length,num_y_signals))


    for i in range (test_length):
            x_test_batch, y_test_batch=x_train_generator.__getitem__(1)
            y_masked = x_test_batch
            y_true = y_test_batch
            y_pred = CRIME_model.predict(x_test_batch)
            break


for sample in range(test_length):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred[sample, :]

        # Get the true output-signal from the data-set.
        signal_true = y_true[sample, :]

        error = np.zeros(len(signal_true))
        p_error = np.zeros(len(signal_true))

        for i in range(len(signal_true)):
            error[i] = signal_true[i]-signal_pred[i]
            p_error[i] = mape((signal_true[i]+max(signal_true)).reshape(-1,1), (sig-
nal_pred[i]+max(signal_true)).reshape(-1,1))


        # Make the plotting-canvas bigger.
        plt.figure(figsize=(30,5))

        # Plot and compare the two signals.
        plt.plot(signal_true, '-*', label='true')
        plt.plot(signal_pred,  '-+', label='pred')

        # Plot labels etc.
        # plt.ylabel(target_names[signal_])
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
```

```
        plt.show()

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(30,5))

        # Plot and compare the two signals.
        plt.plot(error, label='Error')


        # Plot labels etc.
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(30,5))

        # Plot and compare the two signals.
        plt.plot(p_error, label='Error Percent')

        # Plot labels etc.
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(30,30))

        # Plot and compare the two signals.
        plt.scatter(signal_true, signal_pred)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()
        break

target_names.values.tolist()

# %%
with tf.device('/device:GPU:0'):
    mape = tf.keras.losses.MeanAbsolutePercentageError()

    y_masked = np.zeros(shape=(test_length,num_x_signals))
    y_pred = np.zeros(shape=(test_length,num_y_signals))
    y_true = np.zeros(shape=(test_length,num_y_signals))


    for i in range (test_length):
```

```
        # x_test_batch, y_test_batch=x_train_generator.__getitem__(1)
        # y_masked = x_test_batch
        y_true = y_test
        y_pred = regr_rf.predict(X_test[:,2:])
        y_pred_LL = regr_rf_LL.predict(X_test)
        break


for sample in range(test_length):
    # Get the output-signal predicted by the model.
    signal_pred = y_pred[sample, :]
    signal_pred_LL = y_pred_LL[sample, :]

    # Get the true output-signal from the data-set.
    signal_true = y_true[sample, :]

    error = np.zeros(len(signal_true))
    p_error = np.zeros(len(signal_true))
    error_LL = np.zeros(len(signal_true))
    p_error_LL = np.zeros(len(signal_true))

    for i in range(len(signal_true)):
        error[i] = signal_true[i]-signal_pred[i]
        p_error[i] = mape((signal_true[i]+max(signal_true)).reshape(-1,1), (sig-
nal_pred[i]+max(signal_true)).reshape(-1,1))
        error_LL[i] = signal_true[i]-signal_pred_LL[i]
        p_error_LL[i] = mape((signal_true[i]+max(signal_true)).reshape(-1,1), (sig-
nal_pred_LL[i]+max(signal_true)).reshape(-1,1))

    mape_ = mape(signal_true, signal_pred)
    mape_LL = mape(signal_true, signal_pred_LL)

    # Make the plotting-canvas bigger.
    # plt.figure(figsize=(30,5))

    # Plot and compare the two signals.
    plt.plot(signal_true, '-*', label='true')
    plt.plot(signal_pred,  '-+', label='pred')
    plt.plot(signal_pred_LL,  '-+', label='pred_LL')

    # Plot labels etc.
    # plt.ylabel(target_names[signal_])
    plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
    plt.grid(color='b', linestyle='-.', linewidth=0.5)
    plt.legend()
    plt.show()

    # Make the plotting-canvas bigger.
    # plt.figure(figsize=(30,5))
```

```
        # Plot and compare the two signals.
        plt.plot(error, label=f'Error {np.round(mape_)}')
        plt.plot(error_LL, label=f'Error_LL {np.round(mape_LL)}')

        # Plot labels etc.
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # Make the plotting-canvas bigger.
        # plt.figure(figsize=(30,5))

        # Plot and compare the two signals.
        plt.plot(p_error, label='Error Percent')
        plt.plot(p_error_LL, label='Error Percent_LL')

        # Plot labels etc.
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # Make the plotting-canvas bigger.
        # plt.figure(figsize=(30,30))

        # Plot and compare the two signals.
        plt.scatter(signal_true, signal_pred)
        plt.scatter(signal_true, signal_pred_LL)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()


# Evaluate the regressor
mae_one[1] = mean_absolute_error(y_test, y_pred)
mae_one_LL[1] = mean_absolute_error(y_test, y_pred_LL)
for i in range(num_y_signals):
  mae_two[1, i] = mean_absolute_error(y_test[:,i], y_pred[:,i])
  mae_two_LL[1, i] = mean_absolute_error(y_test[:,i], y_pred_LL[:,i])
print(f'MAE for first regressor: {mae_one[1]} - Second regressor: {mae_two[1]} Third re-
gressor: {mae_two_LL[1]}')


mape_one[1] = mean_absolute_percentage_error(y_test, y_pred)
mape_one_LL[1] = mean_absolute_percentage_error(y_test, y_pred_LL)
for i in range(num_y_signals):
  mape_two[1, i] = mean_absolute_percentage_error(y_test[:,i], y_pred[:,i])
```

```
  mape_two_LL[1, i] = mean_absolute_percentage_error(y_test[:,i], y_pred_LL[:,i])

print(f'MAPE for first regressor: {mape_one[1]} - Second regressor: {mape_two[1]} Third
regressor: {mape_two_LL[1]}')

# Make the plotting-canvas bigger.
# plt.figure(figsize=(30,5))
# Plot and compare the two signals.
plt.plot(mae_two[1], label=f'MAE: {mae_one[1]}')
plt.plot(mae_two_LL[1], label=f'MAE: {mae_one[1]}')
# Plot labels etc.
plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.legend()
plt.show()

# Make the plotting-canvas bigger.
# plt.figure(figsize=(30,5))
# Plot and compare the two signals.
plt.plot(mape_two[1], label=f'MAPE: {mape_one[1]}')
plt.plot(mape_two_LL[1], label=f'MAPE_LL: {mape_one_LL[1]}')
# Plot labels etc.
plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rotation=90)
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.legend()
plt.show()



# %%
from __future__ import absolute_import, division, print_function, unicode_lit-
erals

import matplotlib
from scipy import signal
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import numpy.ma as ma
import pandas as pd
import sys
import os
from sklearn.preprocessing import MinMaxScaler, QuantileTransformer, Robust-
Scaler, StandardScaler
from sklearn.metrics import mean_squared_error
from numpy import genfromtxt
from numba import njit, cuda,jit
from sklearn.model_selection import train_test_split
import cupy
```

```
import pylab as pl
import seaborn as sns
from pathlib import Path
import shutil
from tqdm.notebook import tqdm_notebook
import googlemaps
import math

# %%
# %%
import tensorflow.keras.backend as K
import tensorflow_addons as tfa
import tensorflow_probability as tfp
from tensorflow.keras import layers
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential, load_model, Model
from tensorflow.keras.optimizers import RMSprop,Adam,SGD,Adadelta,Adagrad,Ada-
max
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, Tensor-
Board, ReduceLROnPlateau
from tensorflow.compat.v1.keras.backend import set_session
from tensorflow.keras.utils import plot_model, to_categorical, normalize
from tensorflow.python.ops.numpy_ops import np_config
np_config.enable_numpy_behavior()
np.set_printoptions(threshold=sys.maxsize)

# %%
# tf.debugging.experimental.enable_dump_debug_info('.', tensor_de-
bug_mode="FULL_HEALTH", circular_buffer_size=-1)
# tf.debugging.set_log_device_placement(True)
from tensorflow.python.client import device_lib

device_name = tf.test.gpu_device_name()
if device_name != '/device:GPU:0':
  raise SystemError('GPU device not found')
print('Found GPU at: {}'.format(device_name))

config = tf.compat.v1.ConfigProto()
config.gpu_options.allow_growth = True
config.gpu_options.per_process_gpu_memory_fraction = 0.1
sess = tf.compat.v1.InteractiveSession(config=config)
set_session(sess)
print(device_lib.list_local_devices())
gpus = tf.config.experimental.list_physical_devices('GPU')
for gpu in gpus:
  tf.config.experimental.set_memory_growth(gpu, True)
```

```
print("Num GPUs Available: ", len(gpus))

# %% [markdown]
# This was developed using Python 3.6 (Anaconda) and package versions:

# %%
from tensorflow.python.platform import build_info as tf_build_info
print("Tensorflow verison: ",tf.__version__)
print("CUDA verison: ", tf_build_info.build_info['cuda_version'])
print("CUDNN verison: ", tf_build_info.build_info['cudnn_version'])

# %%
# tf.keras.backend.floatx()
# tf.keras.backend.set_floatx('float16')
# tf.keras.mixed_precision.experimental.set_policy('mixed_float16')
tf.keras.backend.floatx()


# %% [markdown]
# ### Load  Data
#
#

# %%
file=r'crimeDataFixed.csv'

myData = pd.read_csv(file, delimiter=',', index_col=0)# usecols = ['Va', '
Vb', ' Vc', ' Ia', ' Ib ', ' Ic ', ' dIa ', ' dIb ',' dIc ', ' dOmega_elec', '
Ia_', ' Ib_ ', ' Ic_ ', ' Te', ' Omega_elec', ' cTheta_elec', ' sTheta_elec'])
# myData.round(decimals=6)
# myData=myData.astype(np.float32)
# myData=myData.astype(np.float16)
myData.describe()

# %%
# myData.convert_dtypes('float16')
myData.dtypes, myData.shape


# %% [markdown]
# List of the variables used in the data-set.

# %%
data_top = myData.columns.values
data_top

# %% [markdown]
# These are the top rows of the data-set.
```

```
# %%
myData.head(20)

# %%
myData.values.astype(np.float32)
myData.shape

# %%
# tfp.stats.correlation( newFilled, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
# myData_corr = myData.corr()
# myData_corr.where(np.tril(np.ones(my-
Data_corr.shape)).astype(np.bool_)).style.background_gradient(cmap='cool-
warm').set_properties(**{'font-size': '5'})
# myData_corr.style.background_gradient(cmap='coolwarm').set_proper-
ties(**{'font-size': '5'})

# 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# newFilled_corr.describe()

# %% [markdown]
# # tfp.stats.correlation( myDataClean, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
# new_corr = pd.concat([myDataClean_corr, myDataDirty_corr], axis=1).corr()
# new_corr = new_corr.loc[:,~new_corr.columns.duplicated()]
# new_corr = new_corr.iloc[:-int(len(new_corr)/2)]
# new_corr = new_corr.where(np.tril(np.ones(new_corr.shape)).astype(np.bool_))
# new_corr.style.background_gradient(cmap='coolwarm').set_properties(**{'font-
size': '5'})
# # 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# # new_corr.describe()

# %% [markdown]
# ### Missing Data
#
#
#
# Because we are using resampled data, we have filled in the missing values
with new values that are linearly interpolated from the neighbouring values,
which appears as long straight lines in these plots.
#
# This may confuse the neural network. For simplicity, we will simply remove
these two signals from the data.
#
# But it is only short periods of data that are missing, so you could actually
generate this data by creating a predictive model that generates the missing
data from all the other input signals. Then you could add these generated val-
ues back into the data-set to fill the gaps.
```

```
# %%
# plt.figure(figsize=(30,5*16))
# myData.plot(subplots=True, figsize=(30,5*16))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.show()

# %%
# plt.figure(figsize=(25,5))
# myData['Ia'].plot()
# plt.grid(color='r', linestyle='-.', linewidth=0.5)
# plt.show()

# plt.figure(figsize=(25,5))
# myData['Ib'].plot()
# plt.grid(color='y', linestyle='-.', linewidth=0.5)
# plt.show()

# plt.figure(figsize=(25,5))
# myData['Ic'].plot()
# plt.grid(color='g', linestyle='-.', linewidth=0.5)
# plt.show()

# %% [markdown]
# ### Target Data for Prediction
#
# We will try and predict the future Forex-data.

# %%
# input_names = myData.columns[:-18]
# target_names = myData.columns[-18:]
input_names = myData.columns[:]
target_names = myData.columns[:]

# %% [markdown]
# We will try and predict these signals.

# %%
df_targets = myData[target_names]
df_targets

# %% [markdown]
# ### NumPy Arrays
#
# We now convert the Pandas data-frames to NumPy arrays that can be input to
the neural network. We also remove the last part of the numpy arrays, because
the target-data has `NaN` for the shifted period, and we only want to have
valid data and we need the same array-shapes for the input- and output-data.
```

```
#
# These are the input-signals:

# %%
x_data = myData[input_names].values.astype(np.float32)
x_data, x_data.dtype, np.isinf(x_data).any(), np.isnan(x_data).any()

# %%
print(type(x_data))
print("Shape:", x_data.shape)

# %% [markdown]
# These are the output-signals (or target-signals):

# %%
y_data = df_targets.values.astype(np.float32, casting='unsafe')
y_data, y_data.dtype, np.isinf(y_data).any(), np.isnan(y_data).any()

# %%
print(type(y_data))
print("Shape:", y_data.shape)

# %% [markdown]
# This is the number of observations (aka. data-points or samples) in the
data-set:

# %%
num_data = len(x_data)
num_data

# %% [markdown]
# This is the fraction of the data-set that will be used for the training-set:

# %%
batch_size = 10
train_split = 0.8
num_train = int(train_split * num_data)
num_val = int(0.5*(num_data - num_train))
num_test = (num_data - num_train) - num_val
#steps_per_epoch = int((num_train/batch_size)/40)
steps_per_epoch=1
#train_validation_steps = int((num_val/batch_size))
train_validation_steps = 1
#test_validation_steps = int((num_test/batch_size))
test_validation_steps = 1
print('num_train:',num_train, 'num_val:',num_val, 'num_test:',num_test)
print('steps_per_epoch:', steps_per_epoch)
```

```
print('train_validation_steps:', train_validation_steps, 'test_valida-
tion_steps:', test_validation_steps)

# %%
x_scaler = MinMaxScaler().fit(myData[input_names].values.astype(np.float32))
y_scaler = MinMaxScaler().fit(myData[target_names].values.astype(np.float32))

x_data_scaled = x_scaler.transform(x_data)
y_data_scaled = y_scaler.transform(y_data)

x_train, x_test, y_train, y_test = train_test_split(x_data_scaled,
y_data_scaled, train_size=train_split, random_state=None, shuffle=False)
# x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,
train_size=train_split, random_state=None, shuffle=False)


# %% [markdown]
# This is the number of observations in the training-set:

# %%
num_train = len(x_train)
num_train

# %% [markdown]
# This is the number of observations in the test-set:

# %%
num_test = len(x_test)
num_test

# %% [markdown]
# These are the input-signals for the training- and test-sets:

# %%
len(x_train) + len(x_test)

# %% [markdown]
# These are the output-signals for the training- and test-sets:

# %%
len(y_train) + len(y_test)

# %% [markdown]
# This is the number of input-signals:

# %%
num_x_signals = x_data.shape[1]
num_x_signals
```

```
# %% [markdown]
# This is the number of output-signals:

# %%
num_y_signals = y_data.shape[1]
num_y_signals

# %% [markdown]
# ### Scaled Data
#
# The data-set contains a wide range of values:

# %%
print('x_train min:', x_train.min())
print('x_train max:', x_train.max())

print('y_train min:', y_train.min())
print('y_train max:', y_train.max())

print('x_test min:', x_test.min())
print('x_test max:', x_test.max())

print('y_test min:', y_test.min())
print('y_test max:', y_test.max())

# %% [markdown]
# ## Data Generator
# The data-set has now been prepared as 2-dimensional numpy arrays. The train-
ing-data has almost 300k observations, consisting of 20 input-signals and 3
output-signals.
# These are the array-shapes of the input and output data:

# %%
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)
print('x_test shape:', x_test.shape)
print('y_test shape:', y_test.shape)

# %%

batch_size = 1
sequence_length = 1
mask_percentage =.05


class CustomDataGen(tf.keras.utils.Sequence):
```

```
    def __init__ (self, x_data, y_data, batch_size=None, sequence_length=None,
train=True, validation=True, mask_percentage=0.01, random_batch=False, ran-
dom_idx=False):

        self.x_train = x_data[0]
        self.x_test = x_data[1]
        self.y_train = y_data[0]
        self.y_test = y_data[1]
        self.batch_size = batch_size
        self.sequence_length = sequence_length
        self.train = train
        self.validation = validation
        self.random_batch = random_batch
        self.random_idx = random_idx
        self.mask_percentage = mask_percentage
        self.n = int(self.x_train.shape[0])

    def on_epoch_end(self):
        #do nothing
        return

    def __getitem__(self, index):
        if self.train:
            # print('using train samples')
            x_samples = self.x_train
            y_samples = self.y_train
            self.n = x_samples.shape[0]

        elif self.validation:
            # print('using validation samples')
            x_samples = self.x_test[:num_val]
            y_samples = self.y_test[:num_val]
            self.n = x_samples.shape[0]
        else:
            # print('using test samples')
            x_samples = self.x_test[-num_test:]
            y_samples = self.y_test[-num_test:]
            self.n = x_samples.shape[0]

            # Allocate a new array for the batch of input-signals.
        if self.train:
            # sequence_length_ = np.random.randint(1,self.sequence_length)
            sequence_length_ = self.sequence_length
        else:
            sequence_length_ = self.sequence_length

        if self.random_batch:
            batch_size_ = np.random.randint(1,self.batch_size)
```

```
        else:
            batch_size_ = batch_size

        x_shape = (batch_size_, sequence_length_, x_samples.shape[1])
        y_shape = (batch_size_, sequence_length_, y_samples.shape[1])
        # x_batch = np.zeros(shape=x_shape, dtype=np.float32)
        # y_batch = np.zeros(shape=y_shape, dtype=np.float32)

        # Fill the batch with random sequences of data.
        for i in range(batch_size_):
            # Get a random start-index.

            if self.random_idx:
                sample_idx = np.random.randint(1, x_samples.shape[-2])

            # This points somewhere into the training-data.
            x_batch = x_samples[sample_idx]
            y_batch = y_samples[sample_idx]
            x_batch[-18:] = 0
            # y_batch[:-18] = 0

        return np.ma.expand_dims(x_batch, axis=0), np.ma.expand_dims(y_batch,
axis=0)

    def __len__(self):
        return int(self.n / self.batch_size)


x_train_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=True, valida-
tion=False, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_train_batch, y_train_batch=x_train_generator.__getitem__(1)

print('x_train shape: ', x_train_batch.shape, 'x_train dtype:',
x_train_batch.dtype)
print('y_train shape: ', y_train_batch.shape, 'y_train dtype:',
y_train_batch.dtype)

x_val_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=False, valida-
tion=True, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_val_batch, y_val_batch=x_val_generator.__getitem__(1)

print('x_val shape: ', x_val_batch.shape, 'x_val dtype:', x_val_batch.dtype)
print('y_val shape: ', y_val_batch.shape, 'y_val dtype:', y_val_batch.dtype)
```

```
x_test_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=False, valida-
tion=False, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_test_batch, y_test_batch=x_test_generator.__getitem__(1)

print('x_test shape: ', x_test_batch.shape, 'x_test dtype:',
x_test_batch.dtype)
print('y_test shape: ', y_test_batch.shape, 'y_test dtype:',
y_test_batch.dtype)

# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_train_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_train_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_val_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_val_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_test_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_test_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

np.isnan(x_train_batch).any(), np.isnan(x_val_batch).any(), np.is-
nan(x_test_batch).any()


with tf.device('/device:GPU:0'):
```

```
    @tf.function(experimental_relax_shapes=True, jit_compile=True)
    def CustomLoss(y_true, y_pred):
        # Calculate the MSE MAE MAPE MSLEloss for each value in these tensors.
        # This outputs a 3-rank tensor of the same shape.
        mse = tf.keras.losses.MeanSquaredError()
        mae = tf.keras.losses.MeanAbsoluteError()
        mape = tf.keras.losses.MeanAbsolutePercentageError()
        msle = tf.keras.losses.MeanSquaredLogarithmicError()
        y_true = tf.convert_to_tensor(y_true[-18:])
        y_pred = tf.convert_to_tensor(y_pred[-18:])
        mse = mse(y_true, y_pred)
        mae = mae(y_true, y_pred)
        mape = mape(y_true, y_pred)
        msle = msle(y_true, y_pred)
        loss = tf.add_n([mae, mse, mape, msle])
        loss_mean = tf.reduce_mean(loss)
        return loss_mean


    def create_model():
        inputs = keras.Input(name='InputLayer', shape=(num_x_signals))
        dropout=0.01
        num_layers=25
        num_units = num_x_signals
        steps = 5

        x_encode = Dense(units=num_units,  activation='gelu',
use_bias=True)(inputs)
        x_encode = tf.expand_dims((x_encode), axis=-2)
        x_encode = Conv1D(filters=num_units, kernel_size=4, strides=1, activa-
tion='gelu', data_format='channels_first')(x_encode)
        x_encode = Dropout(dropout)(x_encode)

        for i in range(num_layers):
            num_units = num_units - steps
            # x_encode = LayerNormalization()(x_encode)
            x_encode = Dense(units=num_units,  activation='gelu',
use_bias=True)(x_encode)
            # x_encode = tf.expand_dims((x_encode), axis=1)
            x_encode = Conv1D(filters=num_units, kernel_size=4, strides=1, ac-
tivation='gelu', data_format='channels_first')(x_encode)
            x_encode = Dropout(dropout)(x_encode)

        x_decode=x_encode

        for i in range(num_layers):
            num_units = num_units + steps
```

```
            # x_decode = LayerNormalization()(x_decode)
            # x_decode = tf.expand_dims((x_decode), axis=1)
            x_decode = Conv1D(filters=num_units, kernel_size=4, strides=1, ac-
tivation='gelu', data_format='channels_first')(x_decode)
            x_decode = Dense(units=num_units,  activation='gelu',
use_bias=True)(x_decode)
            x_decode = Dropout(dropout)(x_decode)

        # x_decode = LayerNormalization()(x_decode)
        # x_decode = tf.expand_dims((x_decode), axis=1)
        x_decode = Conv1D(filters=num_units, kernel_size=4, strides=1, activa-
tion='gelu',  data_format='channels_first')(x_decode)
        x_decode = Flatten()(x_decode)
        x_decode = Dropout(dropout)(x_decode)
        x_decode = Dense(units=num_y_signals,  activation='gelu',
use_bias=True)(x_decode)
        outputs = Reshape(name='ReshapeLayer', target_shape=(-1, num_y_si-
gnals))(x_decode)

        optimizer = Adam(learning_rate=1e-4, amsgrad=True)
        CRIME_model = Model(inputs, outputs, name='CRIME_model')
        CRIME_model.compile(loss='mse', optimizer=optimizer, run_eagerly=True)
        # CRIME_model.build((None, 1, num_x_signals))
        # CRIME_model.summary()

        return CRIME_model

    CRIME_model = create_model()



    y_test_pred= CRIME_model.predict(x_test_batch, steps=1, verbose=1)
    print('y_predict shape: ', y_test_pred.shape, 'y_predict dtype:',
y_test_pred.dtype)
    # seq = y_test_pred[batch, :, signal_]
    # plt.figure(figsize=(15,5))
    # plt.grid(color='b', linestyle='-.', linewidth=0.5)
    # plt.plot(seq)

    # CRIME_model.fit(x_test_batch,y_test_batch, epochs=10, verbose=1)


# %% [markdown]
# ### Callback Functions
#
# During training we want to save checkpoints and log the progress to Tensor-
Board so we create the appropriate callbacks for Keras.
# This is the callback for writing checkpoints during training.
```

```
# %%
path_checkpoint = r'CRIME_model.h5'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=False,
                                      restore_best_weights=True,
                                      save_best_only=True)

# %% [markdown]
# This is the callback for stopping the optimization when performance worsens
on the validation-set.

# %%
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=400, verbose=1)

# %% [markdown]
# This is the callback for writing the TensorBoard log during training.

# %%
dirpaths = [Path('.\Tensorboard')]
for dirpath in dirpaths:
    if dirpath.exists() and dirpath.is_dir():
        try:
            shutil.rmtree(dirpath, ignore_errors=True)
            os.chmod(dirpath, 0o777)
            os.rmdir(dirpath)
            os.removedirs(dirpath)
            print("Directory '%s' has been removed successfully", dirpath)
        except OSError as error:
            print(error)
            print("Directory '%s' can not be removed", dirpath)


callback_tensorboard = TensorBoard(log_dir=r'TensorBoard',
                                   histogram_freq=1,
                                   write_graph=True)
                                   # profile_batch = '500,520')

# %% [markdown]
# This callback reduces the learning-rate for the optimizer if the validation-
loss has not improved since the last epoch (as indicated by `patience=0`). The
learning-rate will be reduced by multiplying it with the given factor. We set
a start learning-rate of 1e-3 above, so multiplying it by 0.1 gives a learn-
ing-rate of 1e-4. We don't want the learning-rate to go any lower than this.

# %%
```

```
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
                                       factor=0.95,
                                       min_lr=1e-5,
                                       patience=10,
                                       verbose=1)


# %%
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_tensorboard,
             callback_reduce_lr]


# %% [markdown]
# #### Load weights from last checkpoint

# %%
filepath = r'CRIME_model.h5'
def train_model(resume, epochs, initial_epoch, batch_size, model):
    def fit_model():
        with tf.device('/device:GPU:0'):
            print(model.summary())
            history=model.fit(x_train_generator,
                              steps_per_epoch=steps_per_epoch,
                              epochs=EPOCHS,
                              verbose=1,
                              callbacks=callbacks,
                              validation_data=x_val_generator,
                              validation_steps=train_validation_steps,
                              initial_epoch=initial_epoch)
            model.load_weights(path_checkpoint)
            model.save(filepath)
            model.evaluate(x_test_generator, steps=test_validation_steps)

            return history

    if resume:
        try:
            #del model
            model = load_model(filepath, custom_objects = {"CustomLoss": Cus-
tomLoss})
            # model.load_weights(path_checkpoint)
            print(model.summary())
            print("Model loading....")
            model.evaluate(x_test_generator, steps=test_validation_steps)

        except Exception as error:
            print("Error trying to load checkpoint.")
```

```
            print(error)

    # Training the Model
    return fit_model()

with tf.device('/device:GPU:0'):
    def plot_train_history(history, title):
        loss = history.history['loss']
        accuracy = history.history['acc']
        mape = history.history['mape']
        mae = history.history['mae']
        val_loss = cupy.asnumpy(history.history['val_loss'])
        val_accuracy = cupy.asnumpy(history.history['val_acc'])
        val_mae = cupy.asnumpy(history.history['val_mae'])
        val_mape = cupy.asnumpy(history.history['val_mape'])
        epochs = range(len(loss))
        plt.figure(figsize=(30,5))
        plt.plot(epochs, loss, label='training_loss')
        plt.plot(epochs, val_loss, label='validation_loss')
        plt.show()
        plt.figure(figsize=(30,5))
        plt.plot(epochs, accuracy, label='training_accuracy')
        plt.plot(epochs, val_accuracy, label='validation_accuracy')
        plt.show()
        plt.figure(figsize=(30,5))
        plt.plot(epochs, mae, label='training_mae')
        plt.plot(epochs, val_mae, label='validation_mae')
        plt.show()
        plt.figure(figsize=(30,5))
        plt.plot(epochs, mape, label='training_mape')
        plt.plot(epochs, val_mape, label='validation_mape')
        plt.show()
        return

# %%
EPOCHS = 2000

# steps_per_epoch = int((num_train/batch_size)/10)
steps_per_epoch = int(235)

for _ in range(1):
  try:
    CRIME_model.load_weights(r'CRIME_model2.h5')
    # CRIME_model = load_model(r'CRIME_model.h5')
    CRIME_model.trainable=True
  except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)
```

```
# Train model
with tf.device('/device:GPU:0'):
    history = train_model(resume=False, epochs=EPOCHS, initial_epoch=0,
batch_size=batch_size, model=CRIME_model)
    plot_train_history(history, 'Model Training History')
    CRIME_model.history

# %% [markdown]
# ### Load Checkpoint
#
# Because we use early-stopping when training the model, it is possible that
the model's performance has worsened on the test-set for several epochs before
training was stopped. We therefore reload the last saved checkpoint, which
should have the best performance on the test-set.

# %%
with tf.device('/device:GPU:0'):
    try:
        CRIME_model = load_model(r'CRIME_model.h5', custom_objects = {"Custom-
Loss": CustomLoss})
        # CRIME_model.load_weights(path_checkpoint)
    except Exception as error:
        print("Error trying to load checkpoint.")
        print(error)

# %% [markdown]
# ## Performance on Test-Set
#
# We can now evaluate the model's performance on the test-set. This function
expects a batch of data, but we will just use one long time-series for the
test-set, so we just expand the array-dimensionality to create a batch with
that one sequence.

# %%
with tf.device('/device:GPU:0'):
    CRIME_model.evaluate(x_train_generator, steps=train_validation_steps)
    CRIME_model.evaluate(x_val_generator, steps=train_validation_steps)
    CRIME_model.evaluate(x_test_generator, steps=test_validation_steps)
```

```
# %%
from __future__ import absolute_import, division, print_function, unicode_lit-
erals

import matplotlib
```

```
from scipy import signal
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import numpy.ma as ma
import pandas as pd
import os
from sklearn.preprocessing import MinMaxScaler, QuantileTransformer, Robust-
Scaler, StandardScaler
from sklearn.metrics import mean_squared_error
from numpy import genfromtxt
from sklearn.model_selection import train_test_split
# !pip install cupy
# import cupy
import pylab as pl
import seaborn as sns
from pathlib import Path
import shutil
from tqdm.notebook import tqdm_notebook
# !pip install googlemaps
# import googlemaps
import sys
import math

# %%
# %%
import tensorflow.keras.backend as K
# !pip install tensorflow-addons
import tensorflow_addons as tfa
import tensorflow_probability as tfp
from tensorflow.keras import layers
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential, load_model, Model
from tensorflow.keras.optimizers import RMSprop,Adam,SGD,Adadelta,Adagrad,Ada-
max
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, Tensor-
Board, ReduceLROnPlateau
from tensorflow.compat.v1.keras.backend import set_session
from tensorflow.keras.utils import plot_model, to_categorical, normalize
from tensorflow.python.ops.numpy_ops import np_config

np_config.enable_numpy_behavior()
np.set_printoptions(threshold=sys.maxsize)

# %%
```

```
# tf.debugging.experimental.enable_dump_debug_info('.', tensor_de-
bug_mode="FULL_HEALTH", circular_buffer_size=-1)
# tf.debugging.set_log_device_placement(True)
from tensorflow.python.client import device_lib
try:
  device_name = tf.test.gpu_device_name()
  if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
  print('Found GPU at: {}'.format(device_name))

  config = tf.compat.v1.ConfigProto()
  config.gpu_options.allow_growth = True
  config.gpu_options.per_process_gpu_memory_fraction = 0.1
  sess = tf.compat.v1.InteractiveSession(config=config)
  set_session(sess)
  print(device_lib.list_local_devices())
  gpus = tf.config.experimental.list_physical_devices('GPU')
  for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
  print("Num GPUs Available: ", len(gpus))

except Exception as error:
  print("error trying to configure computing device")
  print(error)

# %% [markdown]
# This was developed using Python 3.6 (Anaconda) and package versions:

# %%
from tensorflow.python.platform import build_info as tf_build_info
print("Tensorflow verison: ",tf.__version__)
print("CUDA verison: ", tf_build_info.build_info['cuda_version'])
print("CUDNN verison: ", tf_build_info.build_info['cudnn_version'])

# %%
# tf.keras.backend.floatx()
# tf.keras.backend.set_floatx('float16')
# tf.keras.mixed_precision.experimental.set_policy('mixed_float16')
tf.keras.backend.floatx()


# %% [markdown]
# ### Load  Data
#
#

# %%
file=r'crimeDataFixed.csv'
```

```
myData = pd.read_csv(file, delimiter=',', index_col=0) # usecols = ['Va', '
Vb', ' Vc', ' Ia', ' Ib ', ' Ic ', ' dIa ', ' dIb ',' dIc ', ' dOmega_elec', '
Ia_', ' Ib_ ', ' Ic_ ', ' Te', ' Omega_elec', ' cTheta_elec', ' sTheta_elec'])
# myData.round(decimals=6)
# myData=myData.astype(np.float32)
# myData=myData.astype(np.float16)
myData.describe()

# %%
# myData.convert_dtypes('float16')
myData.dtypes, myData.shape


# %% [markdown]
# List of the variables used in the data-set.

# %% [markdown]
# These are the top rows of the data-set.

# %%
data_top = myData.columns.values
data_top

# %%
myData.head(20)

# %%
myData.values.astype(np.float32)
myData.shape



# %%
input_names = myData.columns[:-18]
target_names = myData.columns[-18:]

# %%
# # tfp.stats.correlation( newFilled, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
# myData_corr = myData.corr()[target_names][:-18]
# myData_corr.where(np.tril(np.ones(my-
Data_corr.shape)).astype(np.bool_)).style.background_gradient(cmap='cool-
warm').set_properties(**{'font-size': '5'})
# myData_corr.style.background_gradient(cmap='coolwarm').set_proper-
ties(**{'font-size': '5'})


# # 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# # newFilled_corr.describe()
```

```
# %% [markdown]
# # tfp.stats.correlation( myDataClean, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
# new_corr = pd.concat([myDataClean_corr, myDataDirty_corr], axis=1).corr()
# new_corr = new_corr.loc[:,~new_corr.columns.duplicated()]
# new_corr = new_corr.iloc[:-int(len(new_corr)/2)]
# new_corr = new_corr.where(np.tril(np.ones(new_corr.shape)).astype(np.bool_))
# new_corr.style.background_gradient(cmap='coolwarm').set_properties(**{'font-
size': '5'})
# # 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# # new_corr.describe()

# %% [markdown]
# ### Missing Data
#
#
#
# Because we are using resampled data, we have filled in the missing values
with new values that are linearly interpolated from the neighbouring values,
which appears as long straight lines in these plots.
#
# This may confuse the neural network. For simplicity, we will simply remove
these two signals from the data.
#
# But it is only short periods of data that are missing, so you could actually
generate this data by creating a predictive model that generates the missing
data from all the other input signals. Then you could add these generated val-
ues back into the data-set to fill the gaps.

# %%
# plt.figure(figsize=(30,5*16))
# myData.plot(subplots=True, figsize=(30,5*16))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.show()

# %%
# plt.figure(figsize=(25,5))
# myData['Ia'].plot()
# plt.grid(color='r', linestyle='-.', linewidth=0.5)
# plt.show()

# plt.figure(figsize=(25,5))
# myData['Ib'].plot()
# plt.grid(color='y', linestyle='-.', linewidth=0.5)
# plt.show()

# plt.figure(figsize=(25,5))
```

```
# myData['Ic'].plot()
# plt.grid(color='g', linestyle='-.', linewidth=0.5)
# plt.show()

# %% [markdown]
# ### Target Data for Prediction
#
# We will try and predict the future Forex-data.

# %%
input_names = myData.columns[:-18]
target_names = myData.columns[-18:]
# input_names = myData.columns[:]
# target_names = myData.columns[:]

# %% [markdown]
# We will try and predict these signals.

# %%
df_targets = myData[target_names]
df_targets

# %% [markdown]
# ### NumPy Arrays
#
# We now convert the Pandas data-frames to NumPy arrays that can be input to
the neural network. We also remove the last part of the numpy arrays, because
the target-data has `NaN` for the shifted period, and we only want to have
valid data and we need the same array-shapes for the input- and output-data.
#
# These are the input-signals:

# %%
x_data = myData[input_names].values.astype(np.float32)
x_data, x_data.dtype, np.isinf(x_data).any(), np.isnan(x_data).any()

# %%
print(type(x_data))
print("Shape:", x_data.shape)

# %% [markdown]
# These are the output-signals (or target-signals):

# %%
y_data = df_targets.values.astype(np.float32, casting='unsafe')
y_data, y_data.dtype, np.isinf(y_data).any(), np.isnan(y_data).any()

# %%
```

```
print(type(y_data))
print("Shape:", y_data.shape)

# %% [markdown]
# This is the number of observations (aka. data-points or samples) in the
data-set:

# %%
num_data = len(x_data)
num_data

# %% [markdown]
# This is the fraction of the data-set that will be used for the training-set:

# %%
batch_size = 10
train_split = 0.8
num_train = int(train_split * num_data)
num_val = int(0.5*(num_data - num_train))
num_test = (num_data - num_train) - num_val
#steps_per_epoch = int((num_train/batch_size)/40)
steps_per_epoch=1
#train_validation_steps = int((num_val/batch_size))
train_validation_steps = 1
#test_validation_steps = int((num_test/batch_size))
test_validation_steps = 1
print('num_train:',num_train, 'num_val:',num_val, 'num_test:',num_test)
print('steps_per_epoch:', steps_per_epoch)
print('train_validation_steps:', train_validation_steps, 'test_valida-
tion_steps:', test_validation_steps)

# %%
x_scaler = MinMaxScaler().fit(myData[input_names].values.astype(np.float32))
y_scaler = MinMaxScaler().fit(myData[target_names].values.astype(np.float32))

x_data_scaled = x_scaler.transform(x_data) + 1e-5
y_data_scaled = y_scaler.transform(y_data) + 1e-5

x_train, x_test, y_train, y_test = train_test_split(x_data_scaled,
y_data_scaled, train_size=train_split, random_state=None, shuffle=True)
# x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,
train_size=train_split, random_state=None, shuffle=True)


# %% [markdown]
# This is the number of observations in the training-set:

# %%
num_train = len(x_train)
```

```
num_train

# %% [markdown]
# This is the number of observations in the test-set:

# %%
num_test = len(x_test)
num_test

# %% [markdown]
# These are the input-signals for the training- and test-sets:

# %%
len(x_train) + len(x_test)

# %% [markdown]
# These are the output-signals for the training- and test-sets:

# %%
len(y_train) + len(y_test)

# %% [markdown]
# This is the number of input-signals:

# %%
num_x_signals = x_data.shape[1]
num_x_signals

# %% [markdown]
# This is the number of output-signals:

# %%
num_y_signals = y_data.shape[1]
num_y_signals

# %% [markdown]
# ### Scaled Data
#
# The data-set contains a wide range of values:

# %%
print('x_train min:', x_train.min())
print('x_train max:', x_train.max())

print('y_train min:', y_train.min())
print('y_train max:', y_train.max())

print('x_test min:', x_test.min())
```

```
print('x_test max:', x_test.max())

print('y_test min:', y_test.min())
print('y_test max:', y_test.max())

# %% [markdown]
# ## Data Generator
#
# The data-set has now been prepared as 2-dimensional numpy arrays. The train-
ing-data has almost 300k observations, consisting of 20 input-signals and 3
output-signals.
#
# These are the array-shapes of the input and output data:

# %%
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)
print('x_test shape:', x_test.shape)
print('y_test shape:', y_test.shape)

# %%

batch_size = 1
sequence_length = 1
mask_percentage =.05


class CustomDataGen(tf.keras.utils.Sequence):

    def __init__ (self, x_data, y_data, batch_size=None, sequence_length=None,
train=True, validation=True, mask_percentage=0.01, random_batch=False, ran-
dom_idx=False):

        self.x_train = x_data[0]
        self.x_test = x_data[1]
        self.y_train = y_data[0]
        self.y_test = y_data[1]
        self.batch_size = batch_size
        self.sequence_length = sequence_length
        self.train = train
        self.validation = validation
        self.random_batch = random_batch
        self.random_idx = random_idx
        self.mask_percentage = mask_percentage
        self.n = int(self.x_train.shape[0])

    def on_epoch_end(self):
        #do nothing
        return
```

```
def __getitem__(self, index):
    if self.train:
        # print('using train samples')
        x_samples = self.x_train
        y_samples = self.y_train
        self.n = x_samples.shape[0]

    elif self.validation:
        # print('using validation samples')
        x_samples = self.x_test[:num_val]
        y_samples = self.y_test[:num_val]
        self.n = x_samples.shape[0]
    else:
        # print('using test samples')
        x_samples = self.x_test[-num_test:]
        y_samples = self.y_test[-num_test:]
        self.n = x_samples.shape[0]

        # Allocate a new array for the batch of input-signals.
    if self.train:
        # sequence_length_ = np.random.randint(1,self.sequence_length)
        sequence_length_ = self.sequence_length
    else:
        sequence_length_ = self.sequence_length

    if self.random_batch:
        batch_size_ = np.random.randint(1,self.batch_size)
    else:
        batch_size_ = batch_size

    x_shape = (batch_size_, x_samples.shape[1])
    y_shape = (batch_size_, y_samples.shape[1])
    x_batch = np.zeros(shape=x_shape, dtype=np.float32)
    y_batch = np.zeros(shape=y_shape, dtype=np.float32)

    # Fill the batch with random sequences of data.
    for i in range(batch_size_):
        # Get a random start-index.

        if self.random_idx:
            sample_idx = np.random.randint(1, x_samples.shape[-2])

        # This points somewhere into the training-data.
        x_batch[i] = x_samples[sample_idx]
        y_batch[i] = y_samples[sample_idx]
```

```
        # return np.ma.expand_dims(x_batch, axis=0), np.ma.ex-
pand_dims(y_batch, axis=0)
        return x_batch, y_batch

    def __len__(self):
        return int(self.n / self.batch_size)


x_train_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=True, valida-
tion=False, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_train_batch, y_train_batch=x_train_generator.__getitem__(1)

print('x_train shape: ', x_train_batch.shape, 'x_train dtype:',
x_train_batch.dtype)
print('y_train shape: ', y_train_batch.shape, 'y_train dtype:',
y_train_batch.dtype)

x_val_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=False, valida-
tion=True, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_val_batch, y_val_batch=x_val_generator.__getitem__(1)

print('x_val shape: ', x_val_batch.shape, 'x_val dtype:', x_val_batch.dtype)
print('y_val shape: ', y_val_batch.shape, 'y_val dtype:', y_val_batch.dtype)

x_test_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=False, valida-
tion=False, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_test_batch, y_test_batch=x_test_generator.__getitem__(1)

print('x_test shape: ', x_test_batch.shape, 'x_test dtype:',
x_test_batch.dtype)
print('y_test shape: ', y_test_batch.shape, 'y_test dtype:',
y_test_batch.dtype)



# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_train_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_train_batch[batch, : ]
# plt.figure(figsize=(15,5))
```

```
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_val_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_val_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_test_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)
# seq = y_test_batch[batch, : ]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

np.isnan(x_train_batch).any(), np.isnan(x_val_batch).any(), np.is-
nan(x_test_batch).any()

# %%

# %%
learning_rate = 1e-4
weight_decay = 1e-4
dropout_rate = 0.5
projection_dim = num_y_signals                # Dimension of the patch repre-
sentation or embedding to be used (non-restrictive) in this case we are assum-
ing that our embedding has the same dimensionality as our features which means
we could technically sckip the embedding layer altogether.
num_heads = num_y_signals                     # Total number of differnt cop-
ies of Q K V matrices. These will be aggregated as you move from one layer to
the next.
transformer_units = [projection_dim*2, projection_dim]
transformer_layers = 4                            # Total number of complete
transformer blocks or layers to stack (non-restrictive). Mine computation,
network size, GPU memory, speed, etc.
mlp_head_units = [num_y_signals*2, num_y_signals]         # This is your model
output head (non-restrictive). Size and activation functions depend on task to
be performed.
```

```
# %% [markdown]
# ## Implement Multilayer Perceptron (MLP)

# %%
def mlp(x, hidden_units, dropout_rate):
    for units in hidden_units:
        x = layers.Dense(units, activation='gelu')(x)
        x = layers.Dropout(dropout_rate)(x)
    return x

# %% [markdown]
def create_model():
    inputs = keras.Input(name='InputLayer', shape=(num_x_signals))

    # Create Embedding.
    inputs = tf.math.multiply_no_nan(inputs, 1000)
    # embeddings = tf.expand_dims(inputs, axis=-1)
    embeddings = Embedding(input_dim=num_x_signals, output_dim=projec-
tion_dim)(inputs)

    # Create multiple layers of the Transformer block.
    for _ in range(transformer_layers):
        # Layer normalization 1.
        x1 = LayerNormalization(epsilon=1e-3)(embed-
dings)                          # Normalize input
        x1 = (embeddings)                               # Normalize input
        # print(f'x1 shape {x1.shape}')
        # Create a multi-head attention layer.
        attention_output = layers.MultiHeadAttention(num_heads=num_heads,
                                                     key_dim=projection_dim,
                                                     dropout=dropout_rate)(x1,
x1)

        # print(f'Attention Unit {i+1} Output shape {attention_out-
put.shape}')                   # Don't undertand the (x1, x1) input

        # Skip connection 1.
        x2 = layers.Add()([attention_output, embed-
dings])                                 # Elementwise addition of atten-
tion_output matrix and initial or processed/received matrix of original en-
coded_patches
        # print(f'x2 shape {x2.shape}')

        # Layer normalization 2.
        x3 = LayerNormalization(epsilon=1e-
3)(x2)                                      # Normalize array
        # x3 = (x2)                                             # Normalize array
```

```
        # print(f'x3 shape {x3.shape}')

        # MLP.
        x3 = mlp(x3, hidden_units=transformer_units, dropout_rate=drop-
out_rate)                        # Why not just a single layer having number
of units equal to projection dimension?
        # print(f'x3 MLP output shape {x3.shape}')

        # Skip connection 2.
        embeddings = layers.Add()([x3,
x2])                                          # Output of transformer
block to be fed as an initial input to the next block or on to prediciton
stage

    # Create a [batch_size, projection_dim] tensor.
    representation = LayerNormalization(epsilon=1e-3)(embeddings)
    # representation = (embeddings)
    representation = layers.Flatten()(representation)
    representation = layers.Dropout(dropout_rate)(representation)
    # print(f'Transformer encoded representation shape {representa-
tion.shape}')

    # Add MLP.
    features = mlp(representation, hidden_units=mlp_head_units, drop-
out_rate=dropout_rate)

    # Add Output.
    outputs = Dense(units=num_y_signals,  activation='gelu',
use_bias=True)(features)

    # Create the Keras model.
    model = keras.Model(inputs=inputs, outputs=outputs, name='Crime_Trans-
former_Model')

    # Create Optimizer.
    optimizer = Adam(learning_rate=1e-3, amsgrad=True)
    # moving_avg_Adam = tfa.optimizers.MovingAverage(optimizer)
    stocastic_avg_Adam = tfa.optimizers.SWA(optimizer)

    model.compile(loss='mse', optimizer=stocastic_avg_Adam, metrics=['mse',
'mae', 'mape', 'acc'], run_eagerly=True)

    return model

CRIME_model = create_model()

# CRIME_model.summary()
```

```
plot_model(CRIME_model, show_shapes=True, to_file='CRIME_model.png',
show_layer_names=True, rankdir='TB', expand_nested=True, dpi=50)


# batch = 0   # First sequence in the batch.
# signal_ = 0  # First signal from the 20 input-signals.
# seq = x_test_batch[batch, :, signal_]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# seq = y_test_batch[batch, :, signal_]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

y_test_pred = CRIME_model.predict(x_test_batch, steps=1, verbose=1)
print('y_predict shape: ', y_test_pred.shape, 'y_predict dtype:',
y_test_pred.dtype)
# seq = y_test_pred[signal_, :]
# plt.figure(figsize=(15,5))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.plot(seq)

# CRIME_model.fit(x_test_batch,y_test_batch, epochs=10, verbose=1)


# %% [markdown]
# ### Callback Functions
#
# During training we want to save checkpoints and log the progress to Tensor-
Board so we create the appropriate callbacks for Keras.
# This is the callback for writing checkpoints during training.

# %%
path_checkpoint = r'CRIME_model_3.h5'
callback_checkpoint = ModelCheckpoint(filepath=path_checkpoint,
                                      monitor='val_loss',
                                      verbose=1,
                                      save_weights_only=False,
                                      restore_best_weights=True,
                                      save_best_only=True)

# %%
# %%
path_checkpoint_MA = r'CRIME_model_avg_MA.h5'
path_checkpoint_SWA = r'CRIME_model_avg_SWA.h5'
```

```
callback_MA = tfa.callbacks.AverageModelCheckpoint(filepath=path_check-
point_MA,
                                                   update_weights=True)

callback_SWA = tfa.callbacks.AverageModelCheckpoint(filepath=path_check-
point_SWA,
                                                   update_weights=True)


# %% [markdown]
# This is the callback for stopping the optimization when performance worsens
on the validation-set.

# %%
callback_early_stopping = EarlyStopping(monitor='val_loss',
                                        patience=400, verbose=1)

# %% [markdown]
# This is the callback for writing the TensorBoard log during training.

# %%
dirpaths = [Path('.\Tensorboard_3')]
for dirpath in dirpaths:
    if dirpath.exists() and dirpath.is_dir():
        try:
            shutil.rmtree(dirpath, ignore_errors=True)
            os.chmod(dirpath, 0o777)
            os.rmdir(dirpath)
            os.removedirs(dirpath)
            print("Directory '%s' has been removed successfully", dirpath)
        except OSError as error:
            print(error)
            print("Directory '%s' can not be removed", dirpath)


callback_tensorboard = TensorBoard(log_dir=r'TensorBoard_3',
                                   histogram_freq=1,
                                   write_graph=True)
                                  #  profile_batch = '300,320')

# %% [markdown]
# This callback reduces the learning-rate for the optimizer if the validation-
loss has not improved since the last epoch (as indicated by `patience=0`). The
learning-rate will be reduced by multiplying it with the given factor. We set
a start learning-rate of 1e-3 above, so multiplying it by 0.1 gives a learn-
ing-rate of 1e-4. We don't want the learning-rate to go any lower than this.

# %%
callback_reduce_lr = ReduceLROnPlateau(monitor='val_loss',
```

```
                                        factor=0.9999,
                                        min_lr=1e-7,
                                        patience=1,
                                        verbose=1)


# %%
callbacks = [callback_early_stopping,
             callback_checkpoint,
             callback_SWA,
             callback_tensorboard,
             callback_reduce_lr]


# %% [markdown]
# #### Load weights from last checkpoint

# %%
filepath = r'CRIME_model_3.h5'
def train_model(resume, epochs, initial_epoch, batch_size, model):
    def fit_model():
        with tf.device('/device:GPU:0'):
            print(model.summary())
            history=model.fit(x_train, y_train,
                              verbose=1,
                              callbacks=callbacks,
                              validation_split=0.2,
                              epochs=EPOCHS,
                              batch_size=32,
                              #validation_freq=5,
                              #class_weight=None,
                              #max_queue_size=10,
                              #workers=8,
                              #use_multiprocessing=True,
                              shuffle=True)
            model.load_weights(path_checkpoint)
            model.save(filepath)
            model.evaluate(x_test_generator, steps=test_validation_steps)

            return history

    if resume:
        try:
            #del model
            model = load_model(filepath, custom_objects = {"CustomLoss": Cus-
tomLoss})
            # model.load_weights(path_checkpoint)
            # print(model.summary())
            print("Model loading....")
```

```
            model.evaluate(x_test_generator, steps=test_validation_steps)

        except Exception as error:
            print("Error trying to load checkpoint.")
            print(error)

    # Training the Model
    return fit_model()



# %%
EPOCHS = 2000

steps_per_epoch = int(num_train/batch_size)

for _ in range(1):
  try:
    # CRIME_model.load_weights(r'CRIME_model_3.h5')/
    CRIME_model.save(r'CRIME_model_3.h5')
    CRIME_model = load_model(r'CRIME_model_3.h5')
    print("Checkpoint Loaded.")
  except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)

# Train model
with tf.device('/device:GPU:0'):
    history = train_model(resume=False, epochs=EPOCHS, initial_epoch=0,
batch_size=batch_size, model=CRIME_model)
    CRIME_model.history

# %% [markdown]
# ## Performance on Test-Set

# %% [markdown]
# ### Load Checkpoint
#
# Because we use early-stopping when training the model, it is possible that
the model's performance has worsened on the test-set for several epochs before
training was stopped. We therefore reload the last saved checkpoint, which
should have the best performance on the test-set.

# %%
with tf.device('/device:GPU:0'):
    try:
        CRIME_model = load_model(r'CRIME_model_3.h5')
        # CRIME_model.load_weights(path_checkpoint)
    except Exception as error:
```

```
        print("Error trying to load checkpoint.")
        print(error)

# %% [markdown]
# We can now evaluate the model's performance on the test-set. This function
expects a batch of data, but we will just use one long time-series for the
test-set, so we just expand the array-dimensionality to create a batch with
that one sequence.

# %%
with tf.device('/device:GPU:0'):
    CRIME_model.evaluate(x_train_generator, steps=5)
    CRIME_model.evaluate(x_val_generator, steps=5)
    CRIME_model.evaluate(x_test_generator, steps=5)
```

```
# %% [markdown]
# ## Imports

# %%
!pwd

# %% [markdown]
# We need to import several things from Keras.

# %%
try:
  from google.colab import drive
  IN_COLAB=True
except:
  IN_COLAB=False

if IN_COLAB:
  print("We're running Colab")

if IN_COLAB:
  # Mount the Google Drive at mount
  mount='/content/drive'
  print("Colab: mounting Google drive on ", mount)

  drive.mount(mount)

  # Switch to the directory on the Google Drive that you want to use
  import os
  drive_root = mount + "/My Drive/Colab Notebooks/Crime"

  # Create drive_root if it doesn't exist
```

```
  create_drive_root = True
  if create_drive_root:
    print("\nColab: making sure ", drive_root, " exists.")
    os.makedirs(drive_root, exist_ok=True)

  # Change to the directory
  print("\nColab: Changing directory to ", drive_root)
  %cd $drive_root
  !pwd

# %%
from __future__ import absolute_import, division, print_function, unicode_lit-
erals

import matplotlib
from scipy import signal
import matplotlib.pyplot as plt
import tensorflow as tf
import numpy as np
import numpy.ma as ma
import pandas as pd
import os
from sklearn.preprocessing import MinMaxScaler, QuantileTransformer, Robust-
Scaler, StandardScaler
from sklearn.metrics import mean_squared_error
from sklearn.ensemble import RandomForestRegressor
from sklearn.model_selection import train_test_split
from sklearn.multioutput import MultiOutputRegressor
from sklearn.datasets import make_regression
from sklearn.svm import SVR
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error, mean_absolute_error, mean_ab-
solute_percentage_error
from numpy import genfromtxt
import pylab as pl
import seaborn as sns
from pathlib import Path
import shutil
from tqdm.notebook import tqdm_notebook
# !pip install googlemaps
# !pip install tensorflow_addons
# import googlemaps
import sys
import math

# %%
# import tensorflow.keras.backend as K
!pip install tensorflow_addons
```

```
import tensorflow_addons as tfa
import tensorflow_probability as tfp
from tensorflow.keras import layers
from tensorflow import keras
from tensorflow.keras.datasets import mnist
from tensorflow.keras.layers import *
from tensorflow.keras.models import Sequential, load_model, Model
from tensorflow.keras.optimizers import RMSprop,Adam,SGD,Adadelta,Adagrad,Ada-
max
from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint, Tensor-
Board, ReduceLROnPlateau
from tensorflow.compat.v1.keras.backend import set_session
from tensorflow.keras.utils import plot_model, to_categorical, normalize
from tensorflow.python.ops.numpy_ops import np_config

np_config.enable_numpy_behavior()
np.set_printoptions(threshold=sys.maxsize)

# %%
# tf.debugging.experimental.enable_dump_debug_info('.', tensor_de-
bug_mode="FULL_HEALTH", circular_buffer_size=-1)
# tf.debugging.set_log_device_placement(True)
from tensorflow.python.client import device_lib
try:
  device_name = tf.test.gpu_device_name()
  if device_name != '/device:GPU:0':
    raise SystemError('GPU device not found')
  print('Found GPU at: {}'.format(device_name))

  config = tf.compat.v1.ConfigProto()
  config.gpu_options.allow_growth = True
  config.gpu_options.per_process_gpu_memory_fraction = 0.1
  sess = tf.compat.v1.InteractiveSession(config=config)
  set_session(sess)
  print(device_lib.list_local_devices())
  gpus = tf.config.experimental.list_physical_devices('GPU')
  for gpu in gpus:
    tf.config.experimental.set_memory_growth(gpu, True)
  print("Num GPUs Available: ", len(gpus))

except Exception as error:
  print("error trying to configure computing device")
  print(error)

# %% [markdown]
# This was developed using Python 3.6 (Anaconda) and package versions:

# %%
```

```
from tensorflow.python.platform import build_info as tf_build_info
print("Tensorflow verison: ",tf.__version__)
print("CUDA verison: ", tf_build_info.build_info['cuda_version'])
print("CUDNN verison: ", tf_build_info.build_info['cudnn_version'])

# %%
# tf.keras.backend.floatx()
# tf.keras.backend.set_floatx('float16')
# tf.keras.mixed_precision.experimental.set_policy('mixed_float16')
tf.keras.backend.floatx()


# %% [markdown]
# ### Load  Data
#
#

# %%
file=r'crimeDataFixed.csv'

myData = pd.read_csv(file, delimiter=',', index_col=0) # usecols = ['Va', '
Vb', ' Vc', ' Ia', ' Ib ', ' Ic ', ' dIa ', ' dIb ',' dIc ', ' dOmega_elec', '
Ia_', ' Ib_ ', ' Ic_ ', ' Te', ' Omega_elec', ' cTheta_elec', ' sTheta_elec'])
# myData.round(decimals=6)
# myData=myData.astype(np.float32)
# myData=myData.astype(np.float16)
myData.describe()

# %%
# myData.convert_dtypes('float16')
myData.dtypes, myData.shape


# %% [markdown]
# List of the variables used in the data-set.

# %% [markdown]
# These are the top rows of the data-set.

# %%
data_top = myData.columns.values
data_top

# %%
myData.head(20)

# %%
myData.values.astype(np.float32)
myData.shape
```

```
# %%
input_names = myData.columns[:-18]
target_names = myData.columns[-18:]

# %%
# data_scaler = QuantileTransformer()
# myData[myData.columns] = data_scaler.fit_transform(myData.values)
myData.min(), myData.max()

# %%
pd.plotting.scatter_matrix(myData[target_names], diagonal='hist', fig-
size=(30,30))
plt.savefig('matrix.png')

# %%
# tfp.stats.correlation( newFilled, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
myData_corr = myData.corr()[target_names][:-18]
# myData_corr = myData_corr.unstack().sort_values().drop_duplicates().[((my-
Data_corr >= .9) | (myData_corr <= -.9)) & (myData_corr !=1.000)]
# myData_corr = myData_corr[((myData_corr >= .9) | (myData_corr <= -.9)) &
(myData_corr !=1.000)].drop_duplicates()
myData_corr.where(np.tril(np.ones(my-
Data_corr.shape)).astype(np.bool_)).style.background_gradient(cmap='cool-
warm').set_properties(**{'font-size': '5'})
myData_corr.style.background_gradient(cmap='coolwarm').set_proper-
ties(**{'font-size': '5'})

# 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# newFilled_corr.describe()

# %% [markdown]
# # tfp.stats.correlation( myDataClean, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
# new_corr = pd.concat([myDataClean_corr, myDataDirty_corr], axis=1).corr()
# new_corr = new_corr.loc[:,~new_corr.columns.duplicated()]
# new_corr = new_corr.iloc[:-int(len(new_corr)/2)]
# new_corr = new_corr.where(np.tril(np.ones(new_corr.shape)).astype(np.bool_))
# new_corr.style.background_gradient(cmap='coolwarm').set_properties(**{'font-
size': '5'})
# # 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# # new_corr.describe()

# %%
# tfp.stats.correlation( newFilled, y=None, sample_axis=0, event_axis=-1,
keepdims=False, name=None)
myData_corr = myData[input_names].corr()
```

```
myData_corr.where(np.tril(np.ones(my-
Data_corr.shape)).astype(np.bool_)).style.background_gradient(cmap='cool-
warm').set_properties(**{'font-size': '5'})
myData_corr.style.background_gradient(cmap='coolwarm').set_proper-
ties(**{'font-size': '5'})

# 'RdBu_r', 'BrBG_r', & PuOr_r are other good diverging colormaps
# newFilled_corr.describe()

# %%
myData_corr = pd.DataFrame(myData[input_names].corr().unstack().sort_val-
ues().drop_duplicates())
myData_corr = myData_corr[~myData_corr.iloc[:, 0].between(-.9, .97, inclu-
sive=True)]
myData_corr.where(np.tril(np.ones(my-
Data_corr.shape)).astype(np.bool_)).style.background_gradient(cmap='cool-
warm').set_properties(**{'font-size': '5'})
myData_corr = myData_corr.style.background_gradient(cmap='coolwarm').set_prop-
erties(**{'font-size': '5'})
myData_corr.columns.value=['coeff']
myData_corr

# %%
myData_corr = pd.DataFrame(myData[target_names].corr().unstack().sort_val-
ues().drop_duplicates())
myData_corr = myData_corr[~myData_corr.iloc[:, 0].between(-.9, .9, inclu-
sive=True)]
myData_corr.where(np.tril(np.ones(my-
Data_corr.shape)).astype(np.bool_)).style.background_gradient(cmap='cool-
warm').set_properties(**{'font-size': '5'})
myData_corr = myData_corr.style.background_gradient(cmap='coolwarm').set_prop-
erties(**{'font-size': '5'})
myData_corr

# %%
myData_corr = myData.corr()[target_names][:-18]
myData_corr = pd.DataFrame(myData_corr.unstack().sort_values().drop_dupli-
cates())
myData_corr = myData_corr[~myData_corr.iloc[:, 0].between(-.9, .9, inclu-
sive=True)]
# myData_corr = myData_corr[((myData_corr >= .9) | (myData_corr <= -.9)) &
(myData_corr !=1.000)].drop_duplicates()
myData_corr.where(np.tril(np.ones(my-
Data_corr.shape)).astype(np.bool_)).style.background_gradient(cmap='cool-
warm').set_properties(**{'font-size': '5'})
myData_corr.style.background_gradient(cmap='coolwarm').set_proper-
ties(**{'font-size': '5'})
```

```
# %% [markdown]
# ### Missing Data
#
#
#
# Because we are using resampled data, we have filled in the missing values
with new values that are linearly interpolated from the neighbouring values,
which appears as long straight lines in these plots.
#
# This may confuse the neural network. For simplicity, we will simply remove
these two signals from the data.
#
# But it is only short periods of data that are missing, so you could actually
generate this data by creating a predictive model that generates the missing
data from all the other input signals. Then you could add these generated val-
ues back into the data-set to fill the gaps.

# %%
# plt.figure(figsize=(30,5*16))
# myData.plot(subplots=True, figsize=(30,5*16))
# plt.grid(color='b', linestyle='-.', linewidth=0.5)
# plt.show()

# %%
# plt.figure(figsize=(25,5))
# myData['Ia'].plot()
# plt.grid(color='r', linestyle='-.', linewidth=0.5)
# plt.show()

# plt.figure(figsize=(25,5))
# myData['Ib'].plot()
# plt.grid(color='y', linestyle='-.', linewidth=0.5)
# plt.show()

# plt.figure(figsize=(25,5))
# myData['Ic'].plot()
# plt.grid(color='g', linestyle='-.', linewidth=0.5)
# plt.show()

# %% [markdown]
# ### Target Data for Prediction
#
# We will try and predict the future Forex-data.

# %%
input_names = myData.columns[:-18]
target_names = myData.columns[-18:]
# input_names = myData.columns[:]
```

```
# target_names = myData.columns[:]

# %% [markdown]
# We will try and predict these signals.

# %%
df_targets = myData[target_names]
df_targets

# %% [markdown]
# ### NumPy Arrays
#
# We now convert the Pandas data-frames to NumPy arrays that can be input to
the neural network. We also remove the last part of the numpy arrays, because
the target-data has `NaN` for the shifted period, and we only want to have
valid data and we need the same array-shapes for the input- and output-data.
#
# These are the input-signals:

# %%
x_data = myData[input_names].values.astype(np.float32)
x_data, x_data.dtype, np.isinf(x_data).any(), np.isnan(x_data).any()

# %%
print(type(x_data))
print("Shape:", x_data.shape)

# %% [markdown]
# These are the output-signals (or target-signals):

# %%
y_data = df_targets.values.astype(np.float32, casting='unsafe')
y_data, y_data.dtype, np.isinf(y_data).any(), np.isnan(y_data).any()

# %%
print(type(y_data))
print("Shape:", y_data.shape)

# %% [markdown]
# This is the number of observations (aka. data-points or samples) in the
data-set:

# %%
num_data = len(x_data)
num_data

# %% [markdown]
# This is the fraction of the data-set that will be used for the training-set:
```

```
# %%
batch_size = 10
train_split = 0.8
num_train = int(train_split * num_data)
num_val = int(0.5*(num_data - num_train))
num_test = (num_data - num_train) - num_val
#steps_per_epoch = int((num_train/batch_size)/40)
steps_per_epoch=1
#train_validation_steps = int((num_val/batch_size))
train_validation_steps = 1
#test_validation_steps = int((num_test/batch_size))
test_validation_steps = 1
print('num_train:',num_train, 'num_val:',num_val, 'num_test:',num_test)
print('steps_per_epoch:', steps_per_epoch)
print('train_validation_steps:', train_validation_steps, 'test_valida-
tion_steps:', test_validation_steps)

# %%
x_scaler = MinMaxScaler()
y_scaler = MinMaxScaler()
x_data_scaled = x_scaler.fit_transform(x_data)
y_data_scaled = y_scaler.fit_transform(y_data)

# %%
x_train, x_test, y_train, y_test = train_test_split(x_data_scaled,
y_data_scaled, train_size=train_split, random_state=None, shuffle=True)
# x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,
train_size=train_split, random_state=2, shuffle=True)


# %% [markdown]
# This is the number of observations in the training-set:

# %%
num_train = len(x_train)
num_train

# %% [markdown]
# This is the number of observations in the test-set:

# %%
num_test = len(x_test)
num_test

# %% [markdown]
# These are the input-signals for the training- and test-sets:

# %%
```

```
len(x_train) + len(x_test)

# %% [markdown]
# These are the output-signals for the training- and test-sets:

# %%
len(y_train) + len(y_test)

# %% [markdown]
# This is the number of input-signals:

# %%
num_x_signals = x_data.shape[1]
num_x_signals

# %% [markdown]
# This is the number of output-signals:

# %%
num_y_signals = y_data.shape[1]
num_y_signals

# %% [markdown]
# ### Scaled Data
#
# The data-set contains a wide range of values:

# %%
print('x_train min:', x_train.min())
print('x_train max:', x_train.max())

print('y_train min:', y_train.min())
print('y_train max:', y_train.max())

print('x_test min:', x_test.min())
print('x_test max:', x_test.max())

print('y_test min:', y_test.min())
print('y_test max:', y_test.max())

# %% [markdown]
# ## Data Generator
#
# The data-set has now been prepared as 2-dimensional numpy arrays. The train-
ing-data has almost 300k observations, consisting of 20 input-signals and 3
output-signals.
#
# These are the array-shapes of the input and output data:
```

```
# %%
print('x_train shape:', x_train.shape)
print('y_train shape:', y_train.shape)
print('x_test shape:', x_test.shape)
print('y_test shape:', y_test.shape)

# %%

batch_size = 1
sequence_length = 1
mask_percentage =.05

with tf.device('/device:GPU:0'):
    # @tf.function(experimental_relax_shapes=True)
    def mask_data(input_data, mask_percentage=0.05):
        num_masks = int(mask_percentage * input_data.shape[-1])
        mask_indices = np.random.randint(0, input_data.shape[0],
size=num_masks)

        mask = np.zeros(shape = input_data.shape)
        for i in mask_indices:
            mask[i] = 1

        masked_array = ma.array(input_data, mask = mask)

        # pd.DataFrame(masked_array).plot()
        # pd.DataFrame(mask).plot()
        # pd.DataFrame(x_train[0]).plot()
        return ma.asarray(masked_array)


class CustomDataGen(tf.keras.utils.Sequence):

    def __init__ (self, x_data, y_data, batch_size=None, sequence_length=None,
train=True, validation=True, mask_percentage=0.01, random_batch=False, ran-
dom_idx=False):

        self.x_train = x_data[0]
        self.x_test = x_data[1]
        self.y_train = y_data[0]
        self.y_test = y_data[1]
        # self.x_train = MinMaxScaler().fit_transform(x_data[0])
        # self.x_test = MinMaxScaler().fit_transform(x_data[1])
        # self.y_train = MinMaxScaler().fit_transform(y_data[0])
        # self.y_test = MinMaxScaler().fit_transform(y_data[1])
        self.batch_size = batch_size
        self.sequence_length = sequence_length
        self.train = train
```

```
        self.validation = validation
        self.random_batch = random_batch
        self.random_idx = random_idx
        self.mask_percentage = mask_percentage
        self.n = int(self.x_train.shape[0])

    def on_epoch_end(self):
        #do nothing
        return

    def __getitem__(self, index):
        if self.train:
            # print('using train samples')
            x_samples = self.x_train
            y_samples = self.y_train
            self.n = x_samples.shape[0]

        elif self.validation:
            # print('using validation samples')
            x_samples = self.x_test[:num_val]
            y_samples = self.y_test[:num_val]
            self.n = x_samples.shape[0]
        else:
            # print('using test samples')
            x_samples = self.x_test[-num_test:]
            y_samples = self.y_test[-num_test:]
            self.n = x_samples.shape[0]

            # Allocate a new array for the batch of input-signals.
        if self.train:
            # sequence_length_ = np.random.randint(1,self.sequence_length)
            sequence_length_ = self.sequence_length
        else:
            sequence_length_ = self.sequence_length

        if self.random_batch:
            batch_size_ = np.random.randint(1,self.batch_size)
        else:
            batch_size_ = batch_size

        x_shape = (batch_size_, sequence_length_, x_samples.shape[1])
        y_shape = (batch_size_, sequence_length_, y_samples.shape[1])
        # x_batch = np.zeros(shape=x_shape, dtype=np.float32)
        # y_batch = np.zeros(shape=y_shape, dtype=np.float32)

        # Fill the batch with random sequences of data.
        for i in range(batch_size_):
            # Get a random start-index.
```

```
            if self.random_idx:
                sample_idx = np.random.randint(1, x_samples.shape[-2])


            # This points somewhere into the training-data.
            x_batch = mask_data(x_samples[sample_idx], mask_percent-
age=self.mask_percentage)
            y_batch = y_samples[sample_idx]

        return np.ma.expand_dims(x_batch, axis=0), np.ma.expand_dims(y_batch,
axis=0)

    def __len__(self):
        return int(self.n / self.batch_size)


x_train_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=True, valida-
tion=False, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_train_batch, y_train_batch=x_train_generator.__getitem__(1)

print('x_train shape: ', x_train_batch.shape, 'x_train dtype:',
x_train_batch.dtype)
print('y_train shape: ', y_train_batch.shape, 'y_train dtype:',
y_train_batch.dtype)

x_val_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=False, valida-
tion=True, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_val_batch, y_val_batch=x_val_generator.__getitem__(1)

print('x_val shape: ', x_val_batch.shape, 'x_val dtype:', x_val_batch.dtype)
print('y_val shape: ', y_val_batch.shape, 'y_val dtype:', y_val_batch.dtype)

x_test_generator = CustomDataGen((x_train, x_test), (y_train, y_test),
batch_size=batch_size, sequence_length=sequence_length, train=False, valida-
tion=False, mask_percentage=mask_percentage, random_batch=False, ran-
dom_idx=True)
x_test_batch, y_test_batch=x_test_generator.__getitem__(1)

print('x_test shape: ', x_test_batch.shape, 'x_test dtype:',
x_test_batch.dtype)
print('y_test shape: ', y_test_batch.shape, 'y_test dtype:',
y_test_batch.dtype)
```

```
batch = 0   # First sequence in the batch.
signal_ = 0  # First signal from the 20 input-signals.
seq = x_train_batch[batch, : ]
plt.figure(figsize=(15,5))
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.plot(seq)
seq = y_train_batch[batch, : ]
plt.figure(figsize=(15,5))
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.plot(seq)

batch = 0   # First sequence in the batch.
signal_ = 0  # First signal from the 20 input-signals.
seq = x_val_batch[batch, : ]
plt.figure(figsize=(15,5))
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.plot(seq)
seq = y_val_batch[batch, : ]
plt.figure(figsize=(15,5))
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.plot(seq)

batch = 0   # First sequence in the batch.
signal_ = 0  # First signal from the 20 input-signals.
seq = x_test_batch[batch, : ]
plt.figure(figsize=(15,5))
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.plot(seq)
seq = y_test_batch[batch, : ]
plt.figure(figsize=(15,5))
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.plot(seq)

np.isnan(x_train_batch).any(), np.isnan(x_val_batch).any(), np.is-
nan(x_test_batch).any()

# %%
# myData_train, myData_test, _, _ = (myData, myData, train_size=train_split,
random_state=None, shuffle=True)
# myData = pd.DataFrame(myData.values, columns=myData.columns.values)
# myData_train, myData_test, _, _ = train_test_split(myData, myData,
train_size=train_split, random_state=2, shuffle=True)

# train_ds = tfdf.keras.pd_dataframe_to_tf_dataset(myData_train, label=tar-
get_names)
# test_ds = tfdf.keras.pd_dataframe_to_tf_dataset(myData_test, label=tar-
get_names)
```

```
# %%
# max_depth = 50
# n_estimators = 200

# regr_multirf = MultiOutputRegressor(RandomForestRegressor(n_estimators=n_es-
timators, max_depth=max_depth, random_state=2))
# regr_multirf.fit(x_train[:,2:], y_train)

# regr_rf = RandomForestRegressor(n_estimators=n_estimators,
max_depth=max_depth, random_state=2)
# regr_rf.fit(x_train[:,2:], y_train)

# regr_multirf_LL = MultiOutputRegressor(RandomForestRegressor(n_estima-
tors=n_estimators, max_depth=max_depth, random_state=2))
# regr_multirf_LL.fit(x_train, y_train)

# regr_rf_LL = RandomForestRegressor(n_estimators=n_estimators,
max_depth=max_depth, random_state=2)
# regr_rf_LL.fit(x_train, y_train)

# mor = MultiOutputRegressor(SVR(epsilon=1e-6, gamma='auto', verbose=True,
max_iter=-1))
# mor = mor.fit(x_train[:,2:], y_train)

# mor_LL = MultiOutputRegressor(SVR(epsilon=1e-6, gamma='auto', verbose=True,
max_iter=-1))
# mor_LL = mor_LL.fit(x_train, y_train)



# %%
import joblib

filenames = ['regr_multirf.sav', 'regr_rf.sav', 'regr_multirf_LL.sav',
'regr_rf_LL.sav', 'mor.sav', 'mor_LL.sav']
# models = [regr_multirf, regr_rf, regr_multirf_LL, regr_rf_LL, mor, mor_LL]

# for model, filename in zip(models, filenames):
#   print(f'Saving {model} as {filename}')
#   joblib.dump(model, filename)

# load the model from disk
regr_multirf, regr_rf, regr_multirf_LL, regr_rf_LL, mor, mor_LL = [job-
lib.load(filename) for filename in filenames]

# %%
# Predict on new data
y_multirf = regr_multirf.predict(x_test[:,2:])
y_rf = regr_rf.predict(x_test[:,2:])
```

```
y_pred = mor.predict(x_test[:,2:])

y_multirf_LL = regr_multirf_LL.predict(x_test)
y_rf_LL = regr_rf_LL.predict(x_test)
y_pred_LL = mor_LL.predict(x_test)

# %%
# Plot the results
plt.figure()
s = 50
a = 0.4

plt.scatter(
    y_test[:, 0],
    y_test[:, 1],
    edgecolor="k",
    c="navy",
    s=s,
    marker="s",
    alpha=a,
    label="Data",
)

plt.scatter(
    y_multirf[:, 0],
    y_multirf[:, 1],
    edgecolor="k",
    c="cornflowerblue",
    s=s,
    alpha=a,
    label="Multi RF score=%.2f" % regr_multirf.score(x_test[:,2:], y_test),
)

plt.scatter(
    y_rf[:, 0],
    y_rf[:, 1],
    edgecolor="k",
    c="c",
    s=s,
    marker="^",
    alpha=a,
    label="RF score=%.2f" % regr_rf.score(x_test[:,2:], y_test),
)

plt.xlim([-6, 6])
plt.ylim([-6, 6])
plt.xlabel("target 1")
plt.ylabel("target 2")
```

```
plt.title("Comparing random forests and the multi-output meta estimator")
plt.legend()
plt.show()

# %% [markdown]
# #### Load weights from last checkpoint

# %% [markdown]
# ## Performance on Test-Set

# %%
# x_train, x_test, y_train, y_test = train_test_split(x_data, y_data,
train_size=train_split, random_state=2, shuffle=True)
X_test = x_test

# X_test[:,:] = 200


X_test[:1, :].shape

# %%
test_length = 3

mape_two =np.zeros(shape=(4, num_y_signals))
mae_two =np.zeros(shape=(4, num_y_signals))
mape_two_LL =np.zeros(shape=(4, num_y_signals))
mae_two_LL =np.zeros(shape=(4, num_y_signals))

mape_one =np.zeros(shape=(4))
mae_one =np.zeros(shape=(4))
mape_one_LL =np.zeros(shape=(4))
mae_one_LL =np.zeros(shape=(4))

# %%
model_number = 0

with tf.device('/device:GPU:0'):
    y_masked = np.zeros(shape=(test_length,num_x_signals))
    y_pred = np.zeros(shape=(test_length,num_y_signals))
    y_true = np.zeros(shape=(test_length,num_y_signals))


    for i in range (test_length):
            # x_test_batch, y_test_batch=x_train_generator.__getitem__(1)
            # y_masked = x_test_batch
            y_true = y_test
            y_pred = regr_multirf.predict(X_test[:,2:])
            y_pred_LL = regr_multirf_LL.predict(X_test)
            break
```

```
# Evaluate the regressor
mae_one[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred)).round(2)
mae_one_LL[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred_LL)).round(2)
for i in range(num_y_signals):
  mae_two[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred[:,i]).round(2)
  mae_two_LL[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred_LL[:,i]).round(2)

mape_one[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred)).round(2).clip(0,100)
mape_one_LL[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred_LL)).round(2).clip(0,100)
for i in range(num_y_signals):
  mape_two[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred[:,i]).round(2).clip(0,100)
  mape_two_LL[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred_LL[:,i]).round(2).clip(0,100)



for sample in range(test_length):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred[sample, :]
        signal_pred_LL = y_pred_LL[sample, :]

        # Get the true output-signal from the data-set.
        signal_true = y_true[sample, :]

        error = np.zeros(len(signal_true))
        p_error = np.zeros(len(signal_true))
        error_LL = np.zeros(len(signal_true))
        p_error_LL = np.zeros(len(signal_true))

        for i in range(len(signal_true)):
            error[i] = signal_true[i]-signal_pred[i]
            p_error[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred[i].reshape(-1,1)).round(2)
            error_LL[i] = signal_true[i]-signal_pred_LL[i]
            p_error_LL[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred_LL[i].reshape(-1,1)).round(2)

        mae = mean_absolute_error(signal_true, signal_pred).round(2)
```

```
        mae_LL = mean_absolute_error(signal_true, sig-
nal_pred_LL).round(2)
        mape = mean_absolute_percentage_error(signal_true, sig-
nal_pred).round(2)
        mape_LL= mean_absolute_percentage_error(signal_true, sig-
nal_pred_LL).round(2)

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(10,5))
        plt.plot(signal_true, '-*', label='CRIME_True')
        plt.plot(signal_pred,  '-+', label='CRIME_Pred')
        plt.plot(signal_pred_LL,  '-+', label='CRIME_Pred_LL')
        plt.xlabel('Crime')
        plt.ylabel('Normalized Value')
        plt.title('Comparison between RFR and RFR_LL Predictions')
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(10,5))
        # plt.plot(error, label=f'MAE: {mae}')
        # plt.plot(error_LL, label=f'MAE_LL: {mae_LL}')
        # plt.xlabel('Crime')
        # plt.ylabel('Mean Absolute Error')
        # plt.title('Comparison between RFR and RFR_LL Prediction Absolute Er-
rors')
        # plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()

        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(10,5))
        # plt.plot(p_error, label=f'MAPE: {mape}')
        # plt.plot(p_error_LL, label=f'MAPE_LL: {mape_LL}')
        # plt.xlabel('Crime')
        # plt.ylabel('Mean Absolute Percentage Error')
        # plt.title('Comparison between RFR and RFR_LL Prediction Absolute
Percentage Errors')
        # plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()
```

```
        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(5,5))
        # plt.scatter(signal_true, signal_pred, label=f'CRIME_Pred')
        # plt.scatter(signal_true, signal_pred_LL, label=f'CRIME_Pred_LL')
        # plt.title('Linear Correlation between RFR and RFR_LL Prediction')
        # plt.xlabel('True Crime')
        # plt.ylabel('Predicted Crime')
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()
        # # break

# %%
model_number=1

with tf.device('/device:GPU:0'):
    y_masked = np.zeros(shape=(test_length,num_x_signals))
    y_pred = np.zeros(shape=(test_length,num_y_signals))
    y_true = np.zeros(shape=(test_length,num_y_signals))


    for i in range (test_length):
            # x_test_batch, y_test_batch=x_train_generator.__getitem__(1)
            # y_masked = x_test_batch
            y_true = y_test
            y_pred = regr_rf.predict(X_test[:,2:])
            y_pred_LL = regr_rf_LL.predict(X_test)
            break

# Evaluate the regressor
mae_one[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred)).round(2)
mae_one_LL[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred_LL)).round(2)
for i in range(num_y_signals):
  mae_two[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred[:,i]).round(2)
  mae_two_LL[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred_LL[:,i]).round(2)

mape_one[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred)).round(2).clip(0,100)
mape_one_LL[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred_LL)).round(2).clip(0,100)
for i in range(num_y_signals):
```

```
  mape_two[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred[:,i]).round(2).clip(0,100)
  mape_two_LL[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred_LL[:,i]).round(2).clip(0,100)


for sample in range(test_length):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred[sample, :]
        signal_pred_LL = y_pred_LL[sample, :]

        # Get the true output-signal from the data-set.
        signal_true = y_true[sample, :]

        error = np.zeros(len(signal_true))
        p_error = np.zeros(len(signal_true))
        error_LL = np.zeros(len(signal_true))
        p_error_LL = np.zeros(len(signal_true))

        for i in range(len(signal_true)):
            error[i] = signal_true[i]-signal_pred[i]
            p_error[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred[i].reshape(-1,1)).round(2)
            error_LL[i] = signal_true[i]-signal_pred_LL[i]
            p_error_LL[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred_LL[i].reshape(-1,1)).round(2)

        mae = mean_absolute_error(signal_true, signal_pred).round(2)
        mae_LL = mean_absolute_error(signal_true, sig-
nal_pred_LL).round(2)
        mape = mean_absolute_percentage_error(signal_true, sig-
nal_pred).round(2)
        mape_LL= mean_absolute_percentage_error(signal_true, sig-
nal_pred_LL).round(2)

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(10,5))
        plt.plot(signal_true, '-*', label='CRIME_True')
        plt.plot(signal_pred,  '-+', label='CRIME_Pred')
        plt.plot(signal_pred_LL,  '-+', label='CRIME_Pred_LL')
        plt.xlabel('Crime', size=20)
        plt.ylabel('Normalized Values', size=20)
        plt.title('Comparison between Mo-RF and Mo-RF_LL Predicitions')
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()
```

```
        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(10,5))
        # plt.plot(error, label=f'MAE: {mae}')
        # plt.plot(error_LL, label=f'MAE_LL: {mae_LL}')
        # plt.xlabel('Crime', size=20)
        # plt.ylabel('Mean Absolute Error', size=20)
        # plt.title('Comparison between Mo-RF and Mo-RF_LL Predicition Abso-
lute Errors')
        # plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()

        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(10,5))
        # plt.plot(p_error, label=f'MAPE: {mape}')
        # plt.plot(p_error_LL, label=f'MAPE_LL: {mape_LL}')
        # plt.xlabel('Crime', size=20)
        # plt.ylabel('Mean Absolute Percentege Error', size=20)
        # plt.title('Comparison between Mo-RF and Mo-RF_LL Predicition Abso-
lute Percentage Errors')
        # plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()

        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(5,5))
        # plt.scatter(signal_true, signal_pred, label=f'CRIME_Pred')
        # plt.scatter(signal_true, signal_pred_LL, label=f'CRIME_Pred_LL')
        # plt.title('Linear Correlation between Mo-RF and Mo-RF_LL Predici-
tion')
        # plt.xlabel('True Crime', size=20)
        # plt.ylabel('Predicted Crime', size=20)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()
        # break

# %%
model_number = 2

with tf.device('/device:GPU:0'):
    y_masked = np.zeros(shape=(test_length,num_x_signals))
    y_pred = np.zeros(shape=(test_length,num_y_signals))
    y_true = np.zeros(shape=(test_length,num_y_signals))
```

```
    for i in range (test_length):
            y_true = y_test
            y_pred = mor.predict(X_test[:,2:])
            y_pred_LL = mor_LL.predict(X_test)
            break


# Evaluate the regressor
mae_one[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred)).round(2)
mae_one_LL[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred_LL)).round(2)
for i in range(num_y_signals):
  mae_two[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred[:,i]).round(2)
  mae_two_LL[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred_LL[:,i]).round(2)

mape_one[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred)).round(2).clip(0,100)
mape_one_LL[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred_LL)).round(2).clip(0,100)
for i in range(num_y_signals):
  mape_two[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred[:,i]).round(2).clip(0,100)
  mape_two_LL[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred_LL[:,i]).round(2).clip(0,100)


for sample in range(test_length):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred[sample, :]
        signal_pred_LL = y_pred_LL[sample, :]

        # Get the true output-signal from the data-set.
        signal_true = y_true[sample, :]

        error = np.zeros(len(signal_true))
        p_error = np.zeros(len(signal_true))
        error_LL = np.zeros(len(signal_true))
        p_error_LL = np.zeros(len(signal_true))

        for i in range(len(signal_true)):
            error[i] = signal_true[i]-signal_pred[i]
```

```
            p_error[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred[i].reshape(-1,1)).round(2)
            error_LL[i] = signal_true[i]-signal_pred_LL[i]
            p_error_LL[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred_LL[i].reshape(-1,1)).round(2)

        mae = mean_absolute_error(signal_true, signal_pred).round(2)
        mae_LL = mean_absolute_error(signal_true, sig-
nal_pred_LL).round(2)
        mape = mean_absolute_percentage_error(signal_true, sig-
nal_pred).round(2)
        mape_LL= mean_absolute_percentage_error(signal_true, sig-
nal_pred_LL).round(2)

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(10,5))
        plt.plot(signal_true, '-*', label='CRIME_True')
        plt.plot(signal_pred,  '-+', label='CRIME_Pred')
        plt.plot(signal_pred_LL,  '-+', label='CRIME_Pred_LL')
        plt.xlabel('Crime', size=20)
        plt.ylabel('Normalized Value', size=20)
        plt.title('Comparison between SVM and SVM_LL Predicitions')
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(10,5))
        # plt.plot(error, label=f'MAE: {mae}')
        # plt.plot(error_LL, label=f'MAE_LL: {mae_LL}')
        # plt.xlabel('Crime', size=20)
        # plt.ylabel('Mean Absolute Error', size=20)
        # plt.title('Comparison between SVM and SVM_LL Predicition Absolute
Errors')
        # plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()

        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(10,5))
        # plt.plot(p_error, label=f'MAPE: {mape}')
        # plt.plot(p_error_LL, label=f'MAPE_LL: {mape_LL}')
        # plt.xlabel('Crime', size=20)
        # plt.ylabel('Mean Absolute Percentage Error', size=20)
```

```
        # plt.title('Comparison between SVM and SVM_LL Predicition Absolute
Percentage Errors')
        # plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()

        # # Make the plotting-canvas bigger.
        # plt.figure(figsize=(5,5))
        # plt.scatter(signal_true, signal_pred, label=f'CRIME_Pred')
        # plt.scatter(signal_true, signal_pred_LL, label=f'CRIME_Pred_LL')
        # plt.title('Linear Correlation between SVM and SVM_LL Predicition')
        # plt.xlabel('True Crime', size=20)
        # plt.ylabel('Predicted Crime', size=20)
        # plt.grid(color='b', linestyle='-.', linewidth=0.5)
        # plt.legend()
        # plt.show()
        # # break

# %% [markdown]
#

# %%
model_number = 3
# %%
try:
    CRIME_model = load_model(r'CRIME_model_3.h5')
    # CRIME_model.load_weights(path_checkpoint)
    CRIME_model.summary()
except Exception as error:
    print("Error trying to load checkpoint.")
    print(error)


with tf.device('/device:GPU:0'):
    y_pred = np.zeros(shape=(test_length,num_y_signals))
    y_true = np.zeros(shape=(test_length,num_y_signals))


    for i in range (test_length):
            y_true = y_test
            y_pred = CRIME_model.predict(X_test)
            y_pred_LL = CRIME_model.predict(X_test)
            break

# Evaluate the regressor
mae_one[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred)).round(2)
```

```
mae_one_LL[model_number] = mean_absolute_error(y_scaler.inverse_trans-
form(y_true), y_scaler.inverse_transform(y_pred_LL)).round(2)
for i in range(num_y_signals):
  mae_two[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred[:,i]).round(2)
  mae_two_LL[model_number, i] = mean_absolute_error(y_true[:,i],
y_pred_LL[:,i]).round(2)

mape_one[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred)).round(2).clip(0,100)
mape_one_LL[model_number] = mean_absolute_percentage_error(y_scaler.in-
verse_transform(y_true), y_scaler.inverse_trans-
form(y_pred_LL)).round(2).clip(0,100)
for i in range(num_y_signals):
  mape_two[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred[:,i]).round(2).clip(0,100)
  mape_two_LL[model_number, i] = mean_absolute_percentage_error(y_true[:,i],
y_pred_LL[:,i]).round(2).clip(0,100)



for sample in range(test_length):
        # Get the output-signal predicted by the model.
        signal_pred = y_pred[sample+100, :]
        signal_pred_LL = y_pred_LL[sample+100, :]

        # Get the true output-signal from the data-set.
        signal_true = y_true[sample+100, :]

        error = np.zeros(len(signal_true))
        p_error = np.zeros(len(signal_true))
        error_LL = np.zeros(len(signal_true))
        p_error_LL = np.zeros(len(signal_true))

        for i in range(len(signal_true)):
            error[i] = signal_true[i]-signal_pred[i]
            p_error[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred[i].reshape(-1,1)).round(2)
            error_LL[i] = signal_true[i]-signal_pred_LL[i]
            p_error_LL[i] = mean_absolute_percentage_error(signal_true[i].re-
shape(-1,1), signal_pred_LL[i].reshape(-1,1)).round(2)

        mae = mean_absolute_error(signal_true, signal_pred).round(2)
        mae_LL = mean_absolute_error(signal_true, sig-
nal_pred_LL).round(2)
        mape = mean_absolute_percentage_error(signal_true, sig-
nal_pred).round(2)
```

```
        mape_LL= mean_absolute_percentage_error(signal_true, sig-
nal_pred_LL).round(2)

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(10,5))
        plt.plot(signal_true, '-*', label='CRIME_True')
        plt.plot(signal_pred,  '-+', label='CRIME_Pred')
        plt.plot(signal_pred_LL,  '-+', label='CRIME_Pred_LL')
        plt.xlabel('Crime', size=20)
        plt.ylabel('Normalized Value', size=20)
        plt.title('Comparison between Transformer and Transformer_LL Predic-
tions')
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(10,5))
        plt.plot(error, label=f'MAE: {mae}')
        plt.plot(error_LL, label=f'MAE_LL: {mae_LL}')
        plt.xlabel('Crime', size=20)
        plt.ylabel('Mean Absolute Error', size=20)
        plt.title('Comparison between SVM and SVM_LL Predicition Absolute Er-
rors')
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(10,5))
        plt.plot(p_error, label=f'MAPE: {mape}')
        plt.plot(p_error_LL, label=f'MAPE_LL: {mape_LL}')
        plt.xlabel('Crime', size=20)
        plt.ylabel('Mean Absolute Percentage Error', size=20)
        plt.title('Comparison between Transformer and Transformer_LL Predic-
tion Absolute Percentage Errors')
        plt.xticks(range(0,len(target_names)), target_names.values.tolist(),
rotation=90)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()

        # Make the plotting-canvas bigger.
        plt.figure(figsize=(5,5))
```

```
        plt.scatter(signal_true, signal_pred, label=f'CRIME_Pred')
        plt.scatter(signal_true, signal_pred_LL, label=f'CRIME_Pred_LL')
        plt.title('Linear Correlation between Transformer and Transformer_LL
Predictions')
        plt.xlabel('True Crime', size=20)
        plt.ylabel('Predicted Crime', size=20)
        plt.grid(color='b', linestyle='-.', linewidth=0.5)
        plt.legend()
        plt.show()
        # break

# %%
mape_one = mape_one.round(2)
mape_one_LL = mape_one_LL.round(2)
mape_two = mape_two.round(2)
mape_two_LL = mape_two_LL.round(2)
width = 0.25

X=np.arange(num_y_signals)
plt.figure(figsize=(30,10))
plt.bar(X, mape_two[0], color = 'b', width = width, label='1')
plt.bar(X+0.25, mape_two[1], color = 'g', width = width, label='2')
plt.bar(X+0.50, mape_two[2], color = 'r', width = width, label='3')
plt.bar(X+0.75, mape_two[3], color = 'pink', width = width, label='4')
# plt.bar(X+1, 0, color = 'pink', width = width, label='4')
plt.ylabel('MAPE (%)', size=20)
plt.xlabel('Crimes', size=20)
# plt.yticks(np.arange(9), size=20)
# plt.ylim(0,10)
plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rota-
tion=20, size=20)
plt.legend(labels=[f'RFR: {mape_one[0]}%', f'Mo-RF: {mape_one[1]}%', f'SVM:
{mape_one[2]}%', f'TFM: {mape_one[3]}%'], prop={'size': 30})
plt.title('MAPE comparison between RFR, Mo-RF, SVM and TFM without geolocation
', size=30)
plt.grid()
plt.show()

X=np.arange(num_y_signals)
plt.figure(figsize=(30,10))
plt.bar(X, mape_two_LL[0], color = 'b', width = width, label='1')
plt.bar(X+0.25, mape_two_LL[1], color = 'g', width = width, label='2')
plt.bar(X+0.50, mape_two_LL[2], color = 'r', width = width, label='3')
plt.bar(X+0.75, mape_two_LL[3], color = 'purple', width = width, label='4')
plt.ylabel('MAPE (%)', size=20)
plt.xlabel('Crimes', size=20)
# plt.yticks(np.arange(30), size=20)
```

```
plt.xticks(range(0,len(target_names)), target_names.values.tolist(), rota-
tion=20, size=20)
plt.legend(labels=[f'RFR: {mape_one_LL[0]}%', f'Mo-RF: {mape_one_LL[1]}%',
f'SVM: {mape_one_LL[2]}%', f'TFM: {mape_one_LL[3]}%'], prop={'size': 30})
plt.title('MAPE comparison between RFR, Mo-RF, SVM and TFM with full urban
features', size=30)
# plt.ylim(0,10)
plt.grid()
plt.show()

# %%
y_true = y_test + 1e-4

y_pred_TF = CRIME_model.predict(X_test)
y_pred_TF_LL = CRIME_model.predict(X_test)

y_pred_SVM = mor.predict(X_test[:,2:])
y_pred_SVM_LL = mor_LL.predict(X_test)

y_pred_MoRFR = regr_multirf.predict(X_test[:,2:])
y_pred_MoRFR_LL = regr_multirf_LL.predict(X_test)

y_pred_RFR = regr_rf.predict(X_test[:,2:])
y_pred_RFR_LL = regr_rf_LL.predict(X_test)


# Make the plotting-canvas bigger.
plt.figure(figsize=(30,30))
plt.scatter(y_true, y_pred_TF, label=f'CRIME_Pred_TF')
plt.scatter(y_true, y_pred_TF_LL, label=f'CRIME_Pred_TF_LL')
plt.scatter(y_true, y_pred_SVM, label=f'CRIME_Pred_SVM')
plt.scatter(y_true, y_pred_SVM_LL, label=f'CRIME_Pred_SVM_LL')
plt.scatter(y_true, y_pred_MoRFR, label=f'CRIME_Pred_MoRFR')
plt.scatter(y_true, y_pred_MoRFR_LL, label=f'CRIME_Pred_MoRFR_LL')
plt.scatter(y_true, y_pred_RFR, label=f'CRIME_Pred_RFR')
plt.scatter(y_true, y_pred_RFR_LL, label=f'CRIME_Pred_RFR_LL')
plt.scatter(y_true, y_true, label=f'Ideal')

plt.title('Linear correlation between all model predicitions')
plt.xlabel('True Crime', size=20)
plt.ylabel('Predicted Crime', size=20)
plt.grid(color='b', linestyle='-.', linewidth=0.5)
plt.legend()
plt.show()
```